Broadview
WWW.BROADVIEW.COM.CN

Microsoft

包含
高级诊断
技巧和
崩溃分析



4

Fourth Edition

深入解析 Windows 操作系统，第4版

——Microsoft Windows Server 2003/ Windows XP/Windows 2000技术内幕

Microsoft Windows Internals, Fourth Edition:Microsoft Windows Server 2003, Windows XP, and Windows 2000

Jim Allchin 作序
David N. Cutler 题写“Windows NT的历史全景”

[美] Mark E. Russinovich
David A. Solomon 著

潘爱民 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

深入解析 Windows 操作系统

第 4 版

——Microsoft Windows Server 2003/Windows XP/Windows 2000 技术内幕

Microsoft Windows Internals

Fourth Edition:

Microsoft Windows Server 2003, Windows XP, and Windows 2000

[美] Mark E. Russinovich
David A. Solomon 著

潘爱民 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

内 容 简 介

本书是著名的操作系统内核专家 Mark Russinovich 和 David Solomon 撰写的 Windows 操作系统原理的最新版著作，全面和深入地阐述了 Windows 操作系统的整体结构以及内部工作细节。本书针对 Windows Server 2003、Windows XP 和 Windows 2000 做了全面更新，通过许多练习实验让你直接感受到 Windows 的内部行为。另外，本书还介绍了一些高级诊断技术，以便使你的系统运行得更加平稳和高效。无论你是开发人员还是系统管理员，你都可以在本书中找到一些关键的、有关体系结构方面的知识，通过这些知识你可以更好地做系统设计、调试，以及性能优化。

全书内容丰富、信息全面，主要包括的 Windows 操作系统深度知识有：理解 Windows 的关键机制，包括系统服务分发和调度机制、启动和停机，以及注册表；挖掘 Windows 的安全模型，包括访问控制、特权和审计；利用内核调试器和其他的工具来检查内部系统结构；检查与进程、线程和作业相关的数据结构和算法；观察 Windows 如何管理虚拟内存和物理内存；理解 NTFS 的操作和格式，诊断文件系统访问问题；从上往下查看 Windows 的网络栈，包括映射、API、名称解析和协议驱动程序；诊断引导问题，执行崩溃分析。

本书适合广大 Windows 平台开发人员、IT 专业从业人员等参考使用。

Copyright © 2007 by Microsoft Corporation. All rights reserved.

Original English language edition ©2005 by Microsoft by David A. Solomon, Mark E. Russinovich.

All rights reserved. Simplified Chinese edition published by arrangement with the original publisher, Microsoft Corporation, Redmond, Washington, U.S.A.

本书中文简体版专有出版权由 Microsoft Corporation 授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2005-0907

图书在版编目（CIP）数据

深入解析 Windows 操作系统：第 4 版：Microsoft Windows Server 2003/Windows XP/Windows 2000 技术内幕 /（美）罗斯（Russinovich,M.E.），（美）所罗门（Solomon,D.A.）著；潘爱民译.
—北京：电子工业出版社，2007.4
书名原文:Microsoft Windows Internals，Fourth Edition: Microsoft Windows Server 2003，Windows XP，and Windows 2000
ISBN 978-7-121-03969-0

I. 深… II.①罗…②所…③潘… III.服务器—操作系统（软件），Windows IV.TP316.86

中国版本图书馆 CIP 数据核字（2007）第 031620 号

策划编辑：方　舟
责任编辑：方　舟　陈元玉
印　　刷：北京智力达印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×980　1/16　印张：61.75　　字数：1 000 千字
印　　次：2007 年 4 月第 1 次印刷
定　　价：99.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：（010）68279077；邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

www.TopSage.com

译　序

这是一本随Windows NT一起发展和成长起来的经典图书。我在1995年的时候阅读过这本书的第1版中文版，通过该书了解到了Windows NT设计的诸多考虑。它对于我理解Windows曾经起到了重要的作用。除了这本书以外，另外一套讲述Windows SDK开发指南的图书（记得有四卷）也深刻地影响了我对于Windows的理解。虽然第4版与早期的版本有了很大的变化（原作者也不相同，请参考引言部分关于本书历史的介绍），但由于Windows NT的内核结构一直沿袭下来了，无疑本书新的版本更趋成熟，而且新版作者们编写的许多工具更是使本书增色不少。

时隔10年以后，当编辑找到我，希望我来翻译这本书的第4版时，我的第一感觉是，我不能翻译这本书，所以我直截了当地拒绝了编辑。我的顾虑在于两个方面：首先，这是一本重量级的书，更适合于操作系统领域中的资深专家来把握和诠译；其次，我担心没有足够的时间来及时地完成这本书的翻译工作。两周以后，当编辑再次找到我时，我答应找一个帮手来翻译这本书，但是，合适的帮手并不好找。最后，我决定独立翻译本书，但需要一年时间。而实际上，我花了一年半时间才完成本书的翻译工作。无奈，在质量和进度之间，我选择了前者。

这是一本独特的书，它既不像教材那样宣讲Windows操作系统的原理，也不教读者如何编写内核驱动程序或者如何更好地配置Windows系统。相反地，它按照Windows操作系统的体系结构，把内核几乎翻了个遍，目的在于帮助读者理解Windows内核的每一部分是如何工作的，并且通过各种工具让你真正能够看到Windows内部的数据结构和状态，甚至一些运行过程。阅读这本书的过程，其实也是玩转Windows的过程。经过作者们的剖析，Windows已经完完全全不再神秘。如果你想知道Windows的内部工作原理，那么，这本书对你来说再合适不过了。

Microsoft提供了很多工具（并且许多工具可直接通过Microsoft的Web站点获得），可用来观察或控制系统内核的方方面面，其中最为重要的当然非内核调试器（Windbg）莫属了。除此以外，本书作者们也提供了大量实用的工具（有的还相当好玩，见本书第14章最后一节。所有这些工具都可通过*www.sysinternals.com*下载而得到），你不仅在阅读本书的时候能使用这些工具来帮助你更好地理解书中的内容，在日常的工作或生活中也可以使用这些工具来解决一些实际问题。例如，我和同事们通过Process Explorer（在本书中我将其翻译为“进程管理器”）

www.TopSage.com

发现过机器上的流氓软件的痕迹，有时候，当目录无法删除或移动设备无法卸载的时候，通过它也能查到哪个进程还在抓着不放手。对于有些工具，作者们还在书中介绍了实现原理，比如Filemon就是一个很好的例子。

这本书确实打破了Windows操作系统的神秘感，但是，更让Windows大白于天下的，当是Windows的源代码了。2006年7月，Microsoft真正开放了一份可编译的内核源代码（仅核心部分），编译得到的内核文件可以在Windows Server 2003 （SP1）上启动和运行。如果读者有资格获得这一份代码，那么，在阅读本书的时候，你甚至可以直接参考源代码。我想，在这一点上，Microsoft的确体现了诚意。有关详细的信息，请参考*http://www.microsoft.com/resources/sharedsource/Licensing/WindowsAcademic.mspx*。

说到本书的权威性，看一看David Culter（NT内核的最初设计者，奠定了Windows NT的基础）和Jim Allchin为本书写的序言就能知晓。作者之一David Solomon从事Windows NT内部机理的培训和研讨有10多年经历了，而且也在Microsoft内部对员工进行培训。我有幸参加了他2006年春天在北京的一次培训。而另一名作者Mark Russinovich呢，*www.sysinternals.com*上这么多优秀的工具皆出自他的手笔，现在已经加入Microsoft了。有兴趣的读者，可以到网站上看一看Mark的blog，一些技术文章非常有意思。

这一年多翻译本书的过程，于我也是一个学习的过程。这些知识帮助我解决了在工作中遇到的许多难题，也让我更好地理解了Windows操作系统。我经常向同事和实习生们推荐这本书，希望能帮助他们解决手头的一些技术难题，以及更好地在Windows平台上开展工作。如果你的工作也需要紧密地接触Windows，那么，不管是科研人员、开发人员，还是系统管理员，都可以从这本书中获益良多。

如何更有效地发挥本书的作用呢？以我的感觉而言，这本书的叙述并不像教材那样严谨，也不像教材那样按照学习的顺序来组织内容，它应该是一本讲述Windows系统内部机理的实用参考书。所以，如果你没有计算机科学的专业背景，我建议你配合学习一本讲述操作系统原理的书籍。有了操作系统理论的基础以后，再阅读本书无疑会有更好的效果。另外，在阅读过程中，若有条件，一定要动手做一做书中描述的实验。这些实验很容易做，你若能举一反三，则掌控Windows就不在话下了。

最后，借此机会，感谢两位作者写了这么一本有用的书以及一组实用工具。谢谢周筠编辑让我翻译这本书，并容许我这么久才完稿。今年4月份她说的一句话让我感动，她说“我不催你，我知道你很忙”，而实际上，原书的出版社正在催她，她是顶着压力说这句话的。谢谢微软亚洲研究院高校关系经理马歆小姐，她在过去几年中为推广Windows在国内高校的教学和科

www.TopSage.com

研做了大量幕后工作，包括组织国内高校的操作系统教师编写了《Windows操作系统原理》一书，正是在跟她的接触和交流过程中，让我更加意识到了这本书有多么重要。也要谢谢本书编辑方舟和陈元玉，他们配合我的翻译进度，使得我们的工作能以流水线方式进行，他们的工作态度让我感受到了从未有过的编辑对译者的尊重。

在翻译过程中，我也尽可能地改正原著中的一些错误，但我相信这本书还远没有达到完美，尤其是，因翻译而新引入的错误更是在所难免。尽管我花了四个月的时间来复查一遍译稿，但交稿之后还是能发现一些翻译不妥之处，甚至错译的句子，请读者原谅。另外，本书正文之后列出了英汉习惯用语对照表，以方便阅读。

潘爱民

2006年12月于北京

www.TopSage.com

六年前开始的等待

—出版人感言—

2000年暮秋的一个傍晚，我同时结识了蒋涛和潘老师。蒋涛那时刚过而立，白衣飘飘，一身仙气。潘老师与蒋涛同岁，略显老成，斯文和气，令我一见如故。当天，依稀记得我们一起在聊蒋涛即将创办的《程序员》杂志，聊潘老师受到读者追捧的《VC++技术内幕》（第四版），聊我要做的侯捷老师的《深入浅出MFC》（第二版）。

从那一天起，我和潘老师就成为了朋友。他这个人，微笑比较多，言语向来不多。偶然，我在网上发现他写一笔漂亮的钢笔行书，还发现他在练太极。这都是我感兴趣的，问他，他的回答依然寥寥，曰不过是一点兴趣而已，并无深入研究。我趁机向他约稿，约翻译，也约写作。他总是说：会有机会的。

每年去北京的时候，都会和潘老师约着一起喝喝茶，他偶尔兴致好，也有余暇的时候，愿意随我和一帮80后的程序员们聚会，也往往都是带着他不变的微笑听小年轻们意气风发、挥斥方遒，依然话不多。但，他看人其实是独到的，一两句话，就能点到人家的穴位，这时，一双浓眉下的眼睛会显得特别犀利明亮。我一直在想他这样的表情像谁。有一天，他和我喝茶，闲聊到他原来是徐志摩和金庸的老乡——浙江海宁人，我猛然意识到他真的有点像也是浙江人的围棋天才马晓春，呵呵，结果有朋友对我的这一发现也表示赞同。

浙江人的聪明勤奋，我已多次从合作的作译者身上见识到了；这次潘老师依然让我深刻地体会到了这一点。等待与他的合作，已经等了六年。这六年间，IT专业出版潮起潮落，而潘老师却能无视这些沉浮，安然地做着他想做的事情，几乎年年都会在出版上给读者带来惊喜。他也总是鼓励着我，鼓励我耐心做好自己的份内事，不要因追求数量而伤害品质，不要为虚名所累。

2003年春，我做了母亲，潘老师正好来武汉开会，专门来看我和还未满月的女儿。博文视点那时还在筹建中，我对未来一派迷茫，他真诚的鼓励给了我很大的信心。人的一生，太需要这样的朋友了。

2004年底，策划编辑方舟选定了这本《Windows Internals,4/e》，我问他找谁翻译为好，他说：那肯定是潘老师最合适了。我就去请潘老师，几番蹉跎后，潘老师总算首肯了。我发短信

www.TopSage.com

给方舟道喜，方舟回复短信：那就十全十美了！

我的同事——这本书的编辑方舟、陈元玉以及市场负责人余广均以自己耐心细致的工作赢得了潘老师由衷的称赞，我为伙伴们的工作而自豪。

六年前开始的友谊，六年后将通过彼此的合作而继续绵长。衷心盼望来自作译者和出版方的真诚合作，能带给读者美好的阅读体验。

周　筠

2007年2月于武汉

www.TopSage.com

Windows NT的历史全景

我又一次发现自己真的要感谢David Solomon和Mark Russinovich了，他们提供了机会让我在他们合著的关于Windows内部机理的系列图书的最新版本中写一些话。在这一系列图书中，上一版本的出版距新版本已经有三年了，在此期间Windows已经有了两个主要的发行版本：一个是对客户系统的重大更新，另一个则是对刚刚准备就绪要发行的服务器系统的重大更新。

对于像这样一本书的作者，他们面临的两个日益显著的问题是，跟踪Microsoft Windows NT系统在实现方面的进展，以及记录下在每一个版本中哪些特性的实现发生的变化。最终，本书的作者们完成了这一杰出的工作，为全书提供了例子和解释。



（从左至右）David Solomon、David Cutler和Mark Russinovich

当我第一次碰到David Solomon时，我还在DEC（Digital Equipment Corporation）为VAX的VMS操作系统工作，当时他只有16岁。从那时起，他一直涉足于操作系统的开发和对操作系统内部机理的教学。我认识Mark Russinovich的时间要短一些，但知晓他在操作系统领域的专业特长也很有一段时间了。他曾经做出了很杰出的工作，比如可在Microsoft Windows 98上运行的NTFS文件系统，以及他的“实时”Windows内核调试器（通过该工具可以在Windows系统运行的时候探查其内部状态）。

Windows NT项目最初是从1988年10月份开始的，起初的目标是实现一个可移植的系统，能够解决OS/2兼容性、安全性、POSIX、多处理能力、集成的网络功能，以及可靠性等诸多问题。随着Windows 3.0的发行和巨大成功，这些系统目标很快也随之发生变化，变成了直接处理Windows的兼容性，而把OS/2兼容性移到一个子系统中。

www.TopSage.com

我们最初认为，可以在两年多时间内完成第一个Windows NT系统。实际上，第一个发行版本花了四年半时间，到1993年夏天才完成，这一发行版本支持Intel i386、Intel i486和MIPS R4000处理器。六周以后，我们也引入了对于Digital Alpha处理器的支持。

Windows NT的第一个发行版本比预期的要大、要慢，所以，下一个重要的阶段是一个称为Daytona的项目（用Florida的一条高速公路来命名）。这一发行版本的主要目标是减小系统的尺寸，提高系统的速度，当然，也要使它更加可靠。1994年秋天我们发布了Windows NT 3.5，过了6个月，又发布了Windows NT 3.51，此更新版本包含了对于IBM PowerPC处理器的支持。

Windows NT下一个版本的目标是，更新用户界面以便与Windows 95兼容，以及将Microsoft已经开发多年的各种Cairo技术融合进来。这一系统的开发花了两年多时间，最终在1996年夏天作为Windows NT 4.0面市。

NT的下一个版本是Windows 2000，这是最后一个客户和服务器系统同时发布的系统。这一版本建立在与以前版本相同的Windows NT技术基础之上，同时也引入了一些重要的新特性，比如活动目录。花了三年半时间来开发Windows 2000，对当时的Windows NT技术作了全面的测试和调节。开发Windows 2000是过去11年来跨越4种体系结构的系统开发之巅峰。

在Windows 2000开发结束时，我们又启动了一个宏伟计划，要在新的客户和服务器系统中融入新的、增强的客户特性和改进的服务器能力。在这一计划实施过程中，有一点越来越清楚，即服务器特性的实现将会导致客户特性实现的滞后，因此，客户和服务器系统的版本被分开了。在2001年8月，Windows XP Professional和Windows XP Home Edition发布了，一年多以后，在2003年3月，Microsoft Windows Server 2003也发布了。除了对Intel x86体系结构的支持以外，这些系统还包含了对于Intel IA-64的支持，这标志着Windows NT第一次进入了64位处理的领域。

本书是有关Windows XP和Windows Server 2003的内部结构和工作机理的一部权威之作。而且，它也提供了Windows将来转移到64位计算上的大略介绍，包括AMD在2003年引入的x64体系结构（AMD64）和Intel于2004年2月份宣称的64位支持（EM64T）。完全支持x64的客户和服务器版本计划在2005年上半年发行，本书包含了有关x64系统实现细节的诸多精辟观点。

当x86体系结构开始显现出旧时代征兆之时，x64体系结构即是Windows NT的新时代的开始。这一体系结构提供了32位x86兼容性，以保护过去的软件投资，同时它也提供了64位寻址能力以便满足新应用程序更大的空间需求。这既保护了32位软件的投资，同时也为Windows NT

www.TopSage.com

进入下一个10年或走得更远而提供了必不可少的支撑。

尽管在过去几年中，Windows NT系统经历了几次名称上的变化，但是，它仍然完全建立在最初的Windows NT代码基（code base）的基础之上。随着时间的推移和各种发明创造的诞生，许多内部特性的实现发生了重大的变化。本书作者已经做了令人赞不绝口的工作来诠释Windows NT代码基的细节，及其在不同版本之间和不同平台之间的实现上的差异，同时他们也开发了一些例子和工具来帮助读者理解Windows NT内部是如何工作的。每一个认真的操作系统开发人员都应该在自己的案头有这本书。

David N. Cutler
Senior Distinguished Engineer
Microsoft Corporation

www.TopSage.com

序　言



Microsoft Windows成为我生活中的核心部分已经有14年了。在这段时间中，随着发行版本的不断推出，此操作系统已经在广度和深度上前进了很多。今天，推出Windows无疑是全球最重要和最复杂的项目之一。差不多有5 000名工程师在为Windows而工作。Windows的用户几乎跨越了所有的文化，从最关键的业务人员到年轻的小孩子，构成了完整的用户图谱。使用Windows的客户们几乎在各个方面都有不断增强的需求——从能够运行最大的服务器，到足够易用以便于学龄前儿童也能够使用。Windows有各种形态和大小，从嵌入式版本到媒体中心版本，再到数据中心版本。所有这些版本都使用同样的Windows核心，此核心会随着每一个发行版本而发展和增强。

本书是有关Windows内部机理核心的权威之作。如果你想要知道Windows内部是如何工作的，那么，这就是你想要的书。理解如此庞大产品的每一个细节是一项艰巨的任务。但是，如果你从系统的核心概念开始深入钻研，那么，把这些谜题拼接起来相对要容易得多。正如Windows本身已经有了长足的进步一样，本书的涵盖面也在不断拓宽，现在本书已经到了第4版。近些年来，我们使用这本书以前的版本来培训新加入Microsoft的员工，对于这一版本，也必然如此。

如果你跟我一样，那么你也喜欢弄清楚事情是如何真正工作的。阅读“如何使用…”或者“…经验和技巧” 之类的书籍是从来不能让我满足的。如果你理解了事情的内部工作原理，那么，你就会知道如何更好地使用它，使性能和安全性尽可能地最大化，以及如何诊断失败。诚然，也一定会有更多的乐趣。如果你跟我一样，想要“深入浅出”地看一看Windows，那么，这本书正好是你的一个起点。

David和Mark已经做了突出的工作，他们详细地讲述了Windows内部的技术故事。他们强调的这些工具是一份很好的资源，有助于直接的实验训练和诊断工作。当你阅读了这本书以后，你一定会更好地理解Windows操作系统是如何组织起来的，系统中有哪些最新的改进，以及如何更好地发挥这些新的技术进展的作用。

深入解析Windows操作系统，第4版

www.TopSage.com

这是一次技术之旅——尚在行进之中的技术旅行。开始阅读本书吧，深入地挖掘迄今最为激动人心的操作系统。

Jim Allchin
Group Vice President, Platforms
Microsoft Corporation

www.TopSage.com

致　谢

首先，要特别感谢以下人员。

- **Dave Cutler,** 高级杰出工程师（Senior Distinguished Engineer），Microsoft Windows NT的第一任架构师。Dave最初同意David Solomon访问Windows的源代码，并且支持他的工作，通过他的培训业务与“Inside Windows NT（Second Edition）”和“Inside Microsoft Windows 2000（Third Edition）”的写作来解释Windows NT的内部机理。Dave除了审阅本书中关于进程和线程这一章以外，还回答了许多关于Windows系统的内核体系结构方面的问题，以及为这一版本写了“Windows NT的历史全景”。
- **Jim Allchin**, 我们的执行赞助人，为本书写了前言，支持我们在Microsoft内部的课程。
- **Rob Short**, 副总裁，保证我们能够得到所需要的资源，以及能够访问相关的人员。

我们也要感谢Windows部门中的两位开发人员，他们编写的新内容也被合并到这一版本中，他们是：

- **Adrian Marinescu,** 在内存管理一章中，他写了可扩展堆管理器这一部分；
- **Samer Arafeh,** 他描述了Wow64。

感谢我们的老朋友Jeffrey Richter，他写了第1章中“关于.NET和WinFX”的辅助内容，并且在许多次一起吃晚饭时不断地提醒我们，按照他的看法人们会如何看待我们在书中所谈到的内容。

如果没有Microsoft Windows开发组的关键成员的审阅、建议和支持，本书的技术细节不可能有现在这样的深度，也不可能有这样的精确度，因此，我们感谢以下人员，他们为本书做了技术审阅并提出了建议。

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

www.TopSage.com

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

其他一些人也为本书作出了贡献，他们或者在走廊和自助餐厅回答我们的问题，或者提供了技术资料——如果我们漏掉了你，请原谅！

也要感谢Azius开发人员培训公司（Azius Developer Training, *www.azius.com*）的Jamie Hanrahan，他与David合著了最初的Windows Internal Architecture课程的教材，本书第2版建立在此基础之上。Jamie真正懂得如何把复杂的概念用简单和实际的方式解释出来，他编写了一些概念说明，并设计了一些结构图表。

感谢Dave Probert，他提供了共享环境，使得审阅草稿得以发布给Microsoft的内部审阅者们。

还要感谢AMD的Jonathan Sloves，他安排了AMD64测试系统，并且将测试系统发送给我们以帮助我们编写64位的内容，以及把Sysinternals的一些工具移植到x64上。

最后，我们要感谢Microsoft Press的下列人员，感谢他们为本书作出的贡献。

- Robin van Steenburgh，策划编辑（acquisitions editor），谢谢他与我们一起耐心地工作，直到完成这一项目。
- Sally Stickney，曾经一度继续作为我们的项目编辑，但后来卷入到管理事务中。这次我们失去了跟他一起工作的机会！
- Valerie Woolley，接替了Sally作为我们的项目编辑。他很棒（不像Sally在上两个版本中那样总是将就我们）！
- Roger LeBlanc，辛勤地统编了所有章节的文字，找到了不一致的地方，总而言之，使本书的书稿达到了Microsoft Press的高标准。

David Solomon和Mark Russinovich
2004年9月

www.TopSage.com

引　言

《深入解析Windows操作系统，第4版》的读者对象是那些想要理解Microsoft Windows 2000、Windows XP和Microsoft Windows Server 2003操作系统的核心组件内部工作机理的高级计算机专业人员（包括开发人员和系统管理员）。开发人员利用这些知识，可以在构建Windows平台上的应用程序时更好地理解各种设计决策背后的基本原理。这样的知识也可以帮助开发人员调试复杂的问题。系统管理员也可以从这些信息中获益，因为理解了操作系统背后的工作原理，可以有助于理解系统的性能行为，以及当事情变糟时更易于诊断各种系统问题。在阅读了这本书以后，你应该可以更好地理解Windows是如何工作的，以及它为什么有这样那样的表现。

本书的结构

前两章（“概念和工具”和“系统结构”）奠定了本书后面用到的术语和概念的基础。接下去的三章——“系统机制”、“管理机制”以及“启动和停机”——讲述了系统中关键的底层机制。接下来的八章解释了操作系统的核心组件：进程、线程和作业、内存管理、安全性、I/O系统、存储管理、缓存管理器、文件系统，以及网络，最后一章介绍了崩溃转储分析。

本书的历史

本书以前的名称是*Inside Windows NT*（Microsoft Press，1992，中文版的名称是《Windows NT技术内幕》），现在是第4版。第1版是由Helen Custer著的（在Microsoft Windows NT 3.1的最初发布以前出版）。*Inside Windows NT*是第一本关于Windows NT的书籍，它提供了有关Windows NT系统的体系结构和设计方面的关键要点。*Inside Windows NT, Second Edition*（Microsoft Press，1998）是由David Solomon著的。该书在内容上做了更新，涵盖了Windows NT 4.0，并且大大地提高了技术深度的层次。*Inside Windows 2000, Third Edition*（Microsoft Press，2000）是由David Solomon和Mark Russinovich合著的。第3版增加了许多新的话题，比如启动和停机、Windows服务的内部机理、注册表的内部机理、文件系统驱动程序、网络，以及Windows 2000中内核的变化，其中，关于Windows 2000中内核的变化，包含了Windows驱动程序模型（WDM，Windows Driver Model）、即插即用、电源管理、Windows管理规范（WMI，Windows Management Instrumentation）、加密、作业对象和终端服务。

www.TopSage.com

第4版的变化

这一最新的版本，现在称为“*Microsoft Windows Internals*（*Fourth Edition*）”，在内容上做了更新，以覆盖Windows XP和Windows Server 2003中所做的内核变化，包括对于64位系统的支持。练习用的实验也相应地做了更新，以反映出工具中的变化；新增加的实验用到了一些在第3版写作时尚未可用的新工具。

由于从Windows 2000到后续版本之间的内核变化相对较小（与“Windows NT 4.0和Windows 2000之间的变化”相比较而言），所以，本书中绝大部分内容适用于Windows 2000、Windows XP和Windows Server 2003。因此，除非特别声明，否则一切内容都适用于这三个版本。

练习实验

即使没有访问源代码，你也可以通过一些工具（比如内核调试器）来获得许多有关Windows内部机理的知识。每当可以通过一个工具来揭示或演示Windows内部行为的某一方面时，本书中的“实验”辅助章节就会列出让你自己试用该工具时遵从的步骤。这样的实验遍布全书，我们鼓励你在阅读本书时试一试这些实验——看一看Windows内部是如何工作的，这比你仅仅读一遍本书所得到的印象要深刻得多。

本书没有涵盖的话题

Windows是一个大而复杂的操作系统。本书并没有涵盖与Windows内部机理相关的一切内容，而是把焦点集中在基本的系统组件上。例如，本书没有讲述COM+（Windows分布式面向对象编程基础设施），也没有讲述.NET框架（下一代托管代码的应用程序的基础）。

因为这是一本讲述内部机理的书籍，不是一本用户指南、程序设计或系统管理类型的书籍，所以，本书没有描述如何使用、编程或配置Windows。

提醒和告诫

因为本书讲述的是Windows操作系统中未文档化的内部结构和内部操作的行为（比如内核结构和函数），所以，这些内容有可能会在不同的发行版本中有所变化（外部的接口，比如Windows API，则不会受到不兼容变化的影响）。

说到“受版本变化的影响”，我们并不是指，本书讲述的细节会在不同发行版本中一定有所变化，但是你不能认为它们不会改变。任何使用了这些未文档化接口的软件都有可能在将来

www.TopSage.com

的Windows版本上无法正常工作。更糟的是，在内核模式下运行并且用到了这些未文档化接口的软件（比如设备驱动程序）在新的Windows发行版本中运行时可能会导致系统崩溃。

支持

我们做了各种努力来确保本书的精确性。如果你遇到了任何问题，或者有任何疑问，请参考下面列出的资源。

来自作者的支持

本书远未达到完美。毫无疑问，它包含一些不精确的地方；或者也有可能，我们忽略了一些本该讲述的话题。如果你发现了任何你认为不正确的地方，或者你相信我们应该包含一些尚未涵盖的材料，欢迎发送电子邮件到*windowsinternals@sysinternals.com*。本书的更新和修正将张贴在*www.microsoft.com/technet/sysinternals/information.windowsinternals.mspx*页面上。

来自出版商Microsoft Press的支持

Microsoft也通过Web为本书提供修正内容，地址如下：

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

你可以直接连接到Microsoft学习知识库（Microsoft Learning Knowledge Base），针对你学习过程中可能碰到的问题，输入一个相关的查询请求，Web地址是：*http://www.microsoft.com/learning/support/search.asp*。

除了直接给作者们发送反馈信息以外，如果你对本书的表述或用途有任何建议、疑问或者想法，可以通过以下任何一种方法将它们发送给Microsoft。

通信地址：

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

电子邮件地址：

mspinput@microsoft.com

请注意，通过上述地址并不提供产品支持。关于Microsoft Windows的支持信息，请访问*www.microsoft.com/windows*。你也可以在工作日的太平洋时间上午6点到下午6点打电话给（425） 635-7011，呼叫标准支持程序，或者搜索Microsoft的在线支持站点：*suppport.microssft.com/support*。

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

目录一览

第 1 章　概念和工具 1

第 2 章　系统结构 35

第 3 章　系统机制 85

第 4 章　管理机制 183

第 5 章　启动和停机 251

第 6 章　进程、线程和作业 289

第 7 章　内存管理 375

第 8 章　安全性 485

第 9 章　I/O 系统 537

第 10 章　存储管理 615

第 11 章　缓存管理器 655

第 12 章　文件系统 689

第 13 章　网络 787

第 14 章　崩溃转储分析 845

www.TopSage.com

目　录

第 1 章　概念和工具 1

1.1　Windows 操作系统的版本 1

1.2　基础概念和术语 3

Windows API 3

服务、函数和例程 5

进程、线程和作业 6

虚拟内存 14

内核模式和用户模式 16

终端服务及多个会话 21

对象和句柄 22

安全性 23

注册表 24

Unicode 25

1.3　挖掘 Windows 内部机理 25

性能工具 27

Windows 支持工具箱 27

Windows 资源工具箱 27

内核调试 28

Platform SDK 33

DDK（设备驱动程序开发工具） 34

Sysinternals 工具 34

1.4　本章总结 34

第 2 章　系统结构 35

2.1　需求和设计目标 35

2.2　操作系统模型 36

2.3　总体结构 37

可移植性 40

www.TopSage.com

对称多处理 41
可伸缩性 46
客户和服务器版本之间的差异 47
检查版本 49
2.4 关键的系统组件 51
环境子系统和子系统 DLL 53
硬件抽象层（HAL） 67
设备驱动程序 69
系统进程 75
2.5 本章总结 84
第 3 章 系统机制 85
3.1 陷阱分发 85
中断分发 87
异常分发 109
系统服务分发 119
3.2 对象管理器 124
执行体对象 126
对象结构 128
3.3 同步 149
高 IRQL 的同步 151
低 IRQL 的同步 155
3.4 系统辅助线程 166
3.5 Windows 全局标志 168
3.6 本地过程调用（LPC） 171
3.7 内核事件追踪 175
3.8 Wow64 178
Wow64 进程地址空间布局结构 179
系统调用 179
异常分发 179
用户回调 179
文件系统重定向 180
注册表的重定向和反射 180
I/O 控制请求 181
16 位安装器应用程序 182
打印 182

www.TopSage.com

一些限制 182
3.9 本章总结 182
第 4 章 管理机制 183
4.1 注册表 183
查看和修改注册表 183
注册表用法 184
注册表数据类型 185
注册表逻辑结构 186
注册表问题的诊断 192
注册表的内部机理 197
4.2 服务 211
服务应用 212
服务账户 217
服务控制管理器 223
服务启动 225
启动错误 229
接受当前引导和“最后已知的好控制集” 230
服务失败 231
服务停机 232
共享的服务进程 233
服务控制程序 236
4.3 Windows 管理规范 237
WMI 体系结构 237
提供者 239
公共信息模型（CIM）和可管理对象的格式语言 240
WMI 名字空间 243
类关联 244
WMI 实现 247
WMI 安全性 248
4.4 本章总结 249
第 5 章 启动和停机 251
5.1 引导过程 251
x86 和 x64 引导准备 251
x86/x64 引导扇区和 Ntldr 255
IA64 引导过程 264

www.TopSage.com

初始化内核和执行体子系统 266
Smss、Csrss 和 Winlogon 269
自动启动的映像文件 273
5.2 引导和启动问题的故障检查 274
最后已知的好配置 274
安全模式 274
安全模式下的驱动程序加载 275
恢复控制台（Recovery Console） 279
解决常见的引导问题 281
5.3 停机 286
5.4 本章总结 288
第 6 章 进程、线程和作业 289
6.1 进程的内部机理 289
数据结构 289
内核变量 297
性能计数器 297
有关的函数 298
6.2 CreateProcess 的流程 300
阶段 1：打开将要被执行的映像 302
阶段 2：创建 Windows 执行体进程对象 304
阶段 3：创建初始线程，以及它的栈和执行环境 308
阶段 4：将新进程通知 Windows 子系统 309
阶段 5：启动初始线程的执行 310
阶段 6：在新进程环境下执行进程初始化 310
6.3 线程的内部机理 313
数据结构 313
内核变量 320
性能计数器 321
有关的函数 322
一个线程的产生 322
6.4 检查线程活动 323
6.5 线程调度 325
Windows 调度的概述 326
优先级别 327
Windows 调度 API 330

www.TopSage.com

有关的工具.....331
实时优先级.....333
线程状态.....334
分发器数据库.....338
时限.....340
调度情形.....345
环境切换.....347
空闲（Idle）线程.....348
优先级提升.....348
多处理器系统.....357
多处理器的线程调度算法.....366
6.6 作业对象.....369
6.7 本章总结.....374
第 7 章 内存管理.....375
7.1 内存管理器简介.....375
内存管理器组件.....376
内部同步.....377
配置内存管理器.....378
检查内存的使用情况.....378
7.2 内存管理器提供的服务.....382
大页面和小页面.....382
保留的和提交的页面.....384
锁住内存.....385
分配粒度.....385
共享内存和映射文件.....386
保护内存.....388
“不可执行”页面保护.....390
写时复制.....392
堆管理器.....394
地址窗口扩展.....399
7.3 系统内存池.....401
配置内存池的大小.....401
监视内存池的使用.....404
预读列表（Look-Aside List）.....408
驱动程序检验器（Driver Verifier）.....409

www.TopSage.com

7.4 虚拟地址空间的布局结构 413
x86 用户地址空间的布局结构 415
x86 系统地址空间的布局结构 417
x86 会话空间 418
系统页表项（PTE，Page Table Entry） 421
64 位地址空间布局结构 422
7.5 地址转译 425
x86 虚拟地址转译 425
地址转译快查缓冲区 434
物理地址扩展（PAE） 435
IA-64 虚拟地址转译 437
x64 虚拟地址转译 438
7.6 页面错误处理 439
无效 PTE 440
原型 PTE 441
页面换入 I/O 443
冲突的页面错误 444
页面文件 444
7.7 虚拟地址描述符 448
7.8 内存区对象 450
7.9 工作集 457
按需换页 458
7.10 逻辑预取器 458
放置策略 462
工作集管理 463
平衡集管理器和交换器 466
系统工作集 467
7.11 页面帧编号数据库 469
页面列表的动态变化 472
已修改页面写出器 475
PFN 数据结构 476
低内存通知和高内存通知 479
7.12 本章总结 483
第 8 章 安全性 485
8.1 安全系统组件 488

www.TopSage.com

8.2 保护对象 492
访问检查 493
安全描述符和访问控制 506
8.3 账户权限和特权 516
账户权限 517
特权 518
超级特权 523
8.4 安全审计 524
8.5 登录（Logon） 526
Winlogon 初始化 528
用户登录步骤 529
8.6 软件限制策略 533
8.7 本章总结 535
第 9 章 I/O 系统 537
9.1 I/O 系统组件 537
I/O 管理器 539
典型的 I/O 处理过程 540
9.2 设备驱动程序 541
设备驱动程序的类型 541
驱动程序的结构 548
驱动程序对象和设备对象 550
打开设备 555
9.3 I/O 处理 561
I/O 类型 561
映射文件 I/O 和文件缓存 564
I/O 请求包 564
针对单层驱动程序的 I/O 请求 569
针对分层的驱动程序的 I/O 请求 577
I/O 完成端口 585
驱动程序检验器（Driver Verifier） 589
9.4 即插即用（PnP）管理器 590
即插即用支持的级别 591
驱动程序对于即插即用的支持 592
驱动程序加载、初始化和安装 594
驱动程序安装 603

www.TopSage.com

9.5 电源管理器 607
电源管理器的操作 609
驱动程序的电源操作 610
驱动程序对于设备电源的控制 613
9.6 本章总结 613
第 10 章 存储管理 615
10.1 有关存储的术语 615
10.2 磁盘驱动程序 616
Ntldr 616
磁盘类、端口和小端口驱动程序 617
磁盘设备对象 620
分区管理器 622
10.3 卷的管理 622
基本磁盘 624
动态磁盘 626
多分区卷的管理 632
卷名字空间 638
卷的 I/O 操作 646
虚拟磁盘服务 648
卷影像（shadow）拷贝服务 649
10.4 本章总结 654
第 11 章 缓存管理器 655
11.1 缓存管理器的关键特性 655
单个中心化的系统缓存 656
内存管理器 656
缓存一致性 656
虚拟块缓存 658
流式缓存机制 658
对可恢复文件系统的支持 658
11.2 缓存的虚拟内存管理 660
11.3 缓存的大小 662
LargeSystemCache 662
缓存的虚拟大小 663
缓存的工作集大小 665

www.TopSage.com

缓存的物理大小 667
11.4 缓存的数据结构 668
系统范围的缓存数据结构 669
针对每个文件的缓存数据结构 670
11.5 文件系统接口 674
从缓存中来回拷贝数据 676
通过映射和锁定接口进行缓存 677
通过直接内存访问接口进行缓存 678
11.6 快速 I/O 679
11.7 预读（Read Ahead）和滞后写（Write Behind） 682
智能预读 682
回写缓存（Write-Back Caching）和延迟写（Lazy Writing） 683
写节流（Write Throttling） 686
系统线程 687
11.8 本章总结 688
第 12 章 文件系统 689
12.1 Windows 文件系统格式 690
CDFS 690
UDF 691
FAT12、FAT16 和 FAT32 691
NTFS 694
12.2 文件系统驱动程序总体结构 694
本地 FSD 695
远程 FSD 696
文件系统操作 700
文件系统过滤型驱动程序 705
12.3 诊断文件系统的问题 711
Filemon 的基本和高级模式 711
Filemon 诊断技巧 712
12.4 NTFS 设计目标和特性 717
高端（High-End）文件系统的需求 717
NTFS 的高级特性 719
12.5 NTFS 文件系统驱动程序 729
12.6 NTFS 在磁盘上的结构 732
卷（volume） 732

www.TopSage.com

簇（cluster）.....732
主文件表（MFT）.....733
文件引用号.....739
文件纪录.....740
文件名.....742
驻留的和非驻留的属性.....744
数据压缩和稀疏文件.....747
变化日志文件.....752
索引.....753
对象 ID.....754
配额跟踪.....755
统一的安全性.....756
重解析点.....758
12.7 NTFS 的恢复支持.....758
文件系统设计的演变.....759
日志记录.....761
恢复.....767
NTFS 的坏簇恢复.....771
12.8 加密文件系统（EFS）安全性.....775
第一次加密一个文件.....778
解密过程.....783
加密文件的备份.....784
12.9 本章总结.....785
第 13 章 网络.....787
13.1 Windows 的网络总体结构.....787
OSI 参考模型.....787
Windows 网络组件.....789
13.2 网络 API.....791
Windows 套接字（Windows Sockets）.....791
远过程调用.....798
Web 访问 API.....803
命名管道和邮件槽.....804
NetBIOS.....811
NetBIOS 的操作.....812
其他的网络 API.....813

www.TopSage.com

13.3 多重定向器支持 815
多提供者转发器 816
多 UNC 提供者 818
13.4 名称解析 820
域名系统 820
Windows Internet 名称服务 820
13.5 协议驱动程序 821
TCP/IP 的扩展 824
13.6 NDIS 驱动程序 828
NDIS 小端口的变化形式 832
面向连接的 NDIS 832
外接 NDIS（Remote NDIS） 835
QOS 836
13.7 绑定 838
13.8 分层的网络服务 839
远程访问（Remote Access） 839
活动目录 840
网络负载平衡 841
文件复制服务 843
分布式文件系统 843
13.9 本章总结 844
第 14 章 崩溃转储分析 845
14.1 Windows 为什么会崩溃 845
14.2 蓝屏 846
14.3 崩溃转储文件 849
崩溃转储的生成 852
14.4 Windows 错误报告 853
14.5 在线崩溃分析 854
14.6 基本的崩溃转储分析 855
Notmyfault 855
基本的崩溃转储分析 856
详细的分析 858
14.7 使用崩溃诊断工具 860
缓冲区溢出和特殊内存池 861

www.TopSage.com

代码改写和系统代码写保护 863
14.8 高级的崩溃转储分析 864
栈破坏 865
挂起的系统或无响应的系统 866
当没有崩溃转储时 869
术语表 871
术语对照表 895
索引 901

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

Concepts and Tools

第1章 概念和工具

本章中，我们将介绍Microsoft Windows操作系统的关键概念和术语，比如Windows API、进程、线程、虚拟内存、内核模式和用户模式、对象、句柄、安全性和注册表等。本书贯穿始终将会用到这些概念和术语。同时也要介绍一些可让你用来探查到Windows内部的工具，比如内核调试器、性能工具，以及来自*www.sysinternals.com*的一些关键工具。而且，我们还将解释如何利用Windows的DDK（Device Driver Kit）和SDK（Platform Software Development Kit）作为资源，来找到关于Windows内部机理的进一步信息。

请一定要理解本章中讲述的所有内容，本书其余的章节建立在此基础之上。

1.1 Windows操作系统的版本

本书涵盖了Microsoft操作系统家族中三个最新的基于Windows NT代码基础的版本：Windows 2000、Windows XP（32位和64位版本），以及Windows Server 2003（32位和64位版本）。除非特别声明，否则本书中的讲解适用于所有这三个版本。作为背景信息，表1.1列出了Windows NT各个版本的发布时间、它们的内部版本号以及外部产品名称。

表1.1 Windows操作系统各个版本

产品名称	内部版本号	发布时间
Windows NT 3.1	3.1	1993年7月
Windows NT 3.5	3.5	1994年9月
Windows NT 3.51	3.51	1995年5月
Windows NT 4.0	4.0	1996年7月
Windows 2000	5.0	1999年12月
Windows XP	5.1	2001年8月
Windows Server 2003	5.2	2003年3月

www.TopSage.com

Windows NT和Windows 95

在关于Windows NT最初的声明中，Microsoft就已经明确地表示，从长远考虑，它将会替代Windows 95（以及Windows 95的后续版本，包括Windows 98和Windows Me）。下面的列表突出了两者之间的一些结构性差异，以及Windows NT（和它的后续版本）优于Windows 95（和它的后续版本）的方面。

- Windows NT支持多处理器系统，Windows 95不支持。
- Windows NT文件系统支持安全性（比如自主访问控制），Windows 95文件系统不支持。
- Windows NT是完全32位（现在已是64位）的操作系统——除了因为要运行16位Windows应用程序而提供的支持代码以外，它不包含任何16位代码。Windows 95则从Windows 3.1和MS-DOS继承了大量老的16位代码。
- Windows NT是完全可重入的。Windows 95中有相当部分是不可重入的（主要是从Windows 3.1继承过来的16位代码）。这些不可重入的代码包括大部分图形函数和窗口管理函数（GDI和USER）。在Windows 95中，当一个32位应用程序企图调用一个系统函数时，若这个系统函数位于不可重入的16位代码中，则该应用程序必须首先获得一个系统全局范围的锁（或互斥体），以阻止其他的线程进入到不可重入的代码中。更糟的是，16位应用程序在运行的时候一直占着该锁。其结果是，虽然Windows 95的核心包含了一个抢先式的32位多线程调度器，但是应用程序通常运行在单线程下，因为Windows 95系统的相当部分代码仍然是不可重入的。
- Windows NT提供了一种选择，让16位Windows应用程序运行在它们自己的地址空间中；而Windows 95则总是在一个共享的地址空间中运行16位Windows应用程序，在这个共享的地址空间中，这些程序相互影响，甚至一个程序破坏（和挂起）别的程序。
- 在Windows NT上，进程共享的内存，只对那些映射了这一共享内存区的进程才是可见的。在Windows 95上，所有共享的内存都是可见的，而且在任何一个进程内都可以写这些内存。因此，任何一个进程都可以改写或者破坏其他协作进程所使用的共享内存。
- 在Windows 95中，有一些关键的操作系统页面在用户模式下是可写的，从而允许一个用户程序破坏系统，或者使系统崩溃。

但是有一件事情，Windows 95可以做到，而基于Windows NT的系统却无法做到，这就是运行所有老的MS-DOS和Windows 3.1应用程序（特别是那些要求直接访问硬件的应用程序）以及16位MS-DOS设备驱动程序。不过，与MS-DOS和Windows 3.1百分之百兼容是Windows 95的一个强制目标，而Windows NT的最初目标是，能够运行**大多数**原有的16位应用程序，同时又不破坏系统的完整性和可靠性。

www.TopSage.com

1.2 基础概念和术语

在本书的讲解过程中，我们将会引用一些结构和概念，对于某些读者来说，可能并不十分熟悉这些术语。在这一节中，我们将定义这些贯穿全书的术语。在阅读后续章节之前，你应该先熟悉这些术语。

Windows API

Windows应用编程接口（API）是针对Microsoft Windows操作系统家族的系统编程接口，这样的系统包括Windows 2000、Windows XP、Windows Server 2003、Windows 95、Windows 98、Windows Me（Millennium Edition）和Windows CE。每个操作系统实现了Windows API的不同子集。本书不考虑Windows 95、Windows98、Windows Me和Windows CE。



注 SDK(Platform Software Development Kit,平台软件开发工具)文档描述了Windows API。（参见本章后面的“SDK（Platform Software Development Kit）”部分）。通过*msdn.microsoft.com*，可以在线免费查阅SDK文档，这其中也包含了所有订阅级的Microsoft Developer Network（MSDN），MSDN是指Microsoft对开发人员的支持计划。有关更多的信息，请参考*msdn.microsoft.com*。关于如何在Windows API基础上编写程序，一份极好的资料是：Jeffrey Richter的书*Programming Applications for Microsoft Windows*（第4版，Microsoft Press，1999）。

在引入64位版本的Windows XP和Windows Server 2003以前，32位版本Windows操作系统的编程接口被称为Win32 API，以便与以前16位版本Windows的编程接口（16位Windows API）区别开来。在本书中，术语Windows API既指Windows 2000的32位接口，也指Windows XP和Windows Server 2003的32位和64位编程接口。

Windows API包括几千个可调用的函数，它们可以被分成以下几个大类：

- 基本服务；
- 组件服务；
- 用户界面服务；
- 图形和多媒体服务；
- 消息和协作；
- 网络；
- Web服务。

本书的焦点集中在一些关键基本服务（比如进程和线程、内存管理、I/O和安全性）的内部机理上。

www.TopSage.com

关于.NET和WinFX

.NET框架是由一个被称为框架类库（FCL，Framework Class Library）的类库和一个提供了托管代码执行环境的公共语言运行库（CLR，Common Language Runtime）组成的，后者提供的托管代码执行环境包含以下一些特性：即时编译、类型检查、垃圾回收和代码访问安全性等。由于CLR具有这些特性，因此它所提供的开发环境能够提高开发人员的生产效率，减少常见的编程错误（关于.NET框架及其核心结构的精彩描述，请参见Jeffrey Richter的*Applied Microsoft .NET Framework Programming*）。

CLR的具体实现形式是一个典型的COM服务器，它的代码位于一个用户模式的标准Windows DLL中。实际上，.NET框架中所有组件的实现形式都是用户模式的标准Windows DLL，它们建立在非托管的Windows API之上（.NET框架中没有一个组件运行在内核模式下）。图1.1显示了这些组件之间的关系。

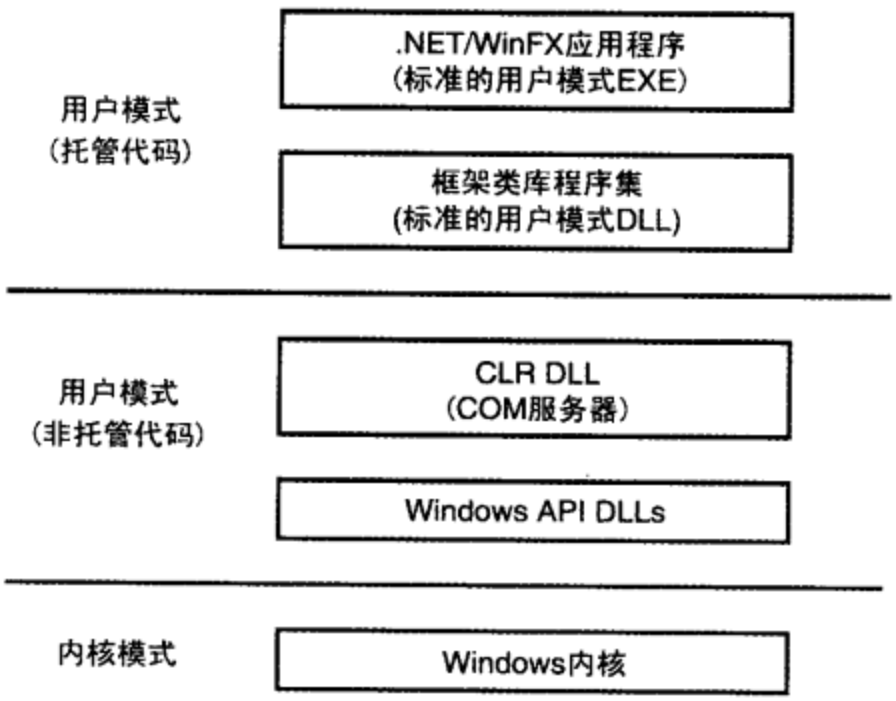


图1.1 .NET框架的组件之间的关系

WinFX是“新的Windows API”，它是随Windows下一个重要版本“Longhorn”一起发布的.NET框架演进版本。它也可以被安装在Windows XP和Windows Server 2003上。WinFX为建立在Windows操作系统之上的下一代应用程序提供了一个基础平台。

www.TopSage.com

Win32 API的历史

有意思的是，Win32并不是Microsoft Windows NT最初预定的编程接口。因为Windows NT项目在启动之初，是作为OS/2第2版的一个替代产品，所以，它的主要编程接口是32位OS/2显示管理器（Presentation Manager）API。然而，项目进行了一年以后，Microsoft Windows 3.0进入市场，并且呈现出很好的发展势头。因此，Microsoft改变了方向，使Windows NT成为未来Windows产品家族的替代品，而不是OS/2的替代品。也正是在这个时候，才真正有必要制定Windows API——在此之前，只有16位接口的Windows API。

尽管Windows API将会引入很多新的函数，甚至这些函数在Windows 3.1上也还无法使用，但是，Microsoft还是决定让新的API与16位Windows API在函数名、语义和数据类型的用法上尽可能地兼容，以便减轻将已有的16位Windows应用程序移植到Windows NT上的负担。所以，当你们有些人第一次看到Windows API的时候，会发现许多函数名和接口看起来并不一致，你们应该要知道，这种不一致的原因之一，正是因为要确保Windows API与老的16位Windows API保持兼容。

服务、函数和例程

在Windows的用户文档和程序设计文档中，有几个术语在不同的上下文环境中有不同的含义。例如，“服务”可以指操作系统中一个可被调用的例程、一个设备驱动程序或者一个服务器进程。下面的列表描述了一些特定的术语在本书中的含义。

- **Windows API函数** 指Windows API中已被文档化的可调用子例程，例如*CreateProcess*、*CreateFile*和*GetMessage*。
- **原生的系统服务**（或者**执行体系统服务**） 指操作系统中未文档化的、可在用户模式下调用的底层服务。例如，*NtCreateProcess*是一个内部系统服务，Windows的*CreateProcess*函数调用该服务来创建一个新的进程（有关原生函数的定义，请参见第3章的“系统服务分发”一节）。
- **内核支持函数**（或**例程**） 指位于Windows操作系统内部且只能在内核模式下调用的子例程（关于内核模式，本章后文将给出定义）。例如，*ExAllocatePool*是一个内核例程，设备驱动程序调用该例程可以向Windows的系统堆申请内存。

www.TopSage.com

- **Windows服务** 指由Windows服务控制管理器启动的进程（虽然注册表将Windows设备驱动程序定义为“服务”，但是，我们在本书中并不这样引用）。例如，“任务调度器（Task Scheduler）”服务运行在一个用户模式的进程中，它支持*at*命令（类似于UNIX命令*at*或*cron*）。
- **DLL（动态链接库）** 指一组可调用的子例程，合起来被链接成一个二进制文件，使用这些子例程的应用程序可以动态地加载此二进制文件。例如Msvcrt.dll（C运行库）和Kernel32.dll（一个Windows API子系统库）。Windows的用户模式组件和应用程序大量使用了DLL。DLL比静态库的优势在于，应用程序可以共享DLL，Windows保证在内存中只有一份DLL代码，供所有引用该DLL的应用程序共享。

进程、线程和作业

尽管表面上看起来程序和进程非常类似，但本质上它们却是截然不同的。**程序**是指一个静态的指令序列，而**进程**则是一个容器，其中包含了当执行一个程序的特定实例时所用到的各种资源。从最高层次的抽象来看，一个Windows进程是由以下元素构成的。

- 一个**私有的虚拟地址空间**，这是指该进程可以使用的一组虚拟内存地址。
- 一个可执行的程序，它定义了初始的代码和数据，并且被映射到该进程的虚拟地址空间中。
- 一个已打开句柄的列表，这些句柄指向各种系统资源，比如信号量、通信端口和文件，该进程内所有的线程都可以访问这些系统资源。
- 一个被称为**访问令牌**的安全环境，它标识了与该进程关联的用户、安全组和特权。
- 一个被称为**进程ID**的惟一标识符（在内部被称为**客户ID**）。
- 至少一个执行线程。

每个进程也指向它的父进程或者创建者进程。然而，如果父进程退出的话，子进程中的父进程信息并不会被更新。因此，一个进程有可能指向一个已不再存在的父进程。这并不是一个问题，因为任何一个进程都不必依赖于父进程信息的有效性。下面的实验说明了这样一种情况。

www.TopSage.com



实验：查看进程树

大多数工具在显示一个进程的属性时，通常不会显示该进程的父进程或者创建者进程的ID。你可以利用性能工具，通过查询Creating Process ID（或者编写程序）来得到父进程ID。Tlist.exe工具（在Windows调试工具箱中）的*/t*开关打开以后，可以显示进程树。下面是*tlist /t*的一个输出示例：

```
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)
```

以上列表利用缩进格式显示了进程之间的父/子关系。对于那些父进程已不再活着的进程，显示的时候被调整到最左边（在上面的例子中譬如Explorer.exe），因为即使祖父进程仍然活着，也无法再找到这种关系。Windows仅仅维护了创建者进程的ID，无法再链回到创建者进程的创建者进程，如此等等。

为了证明Windows仅仅维护了父进程ID而确实没有记录更多的信息，请按照下面的步骤操作。

1. 打开一个命令提示（Command Prompt）窗口。
2. 输入**start cmd**（这样做会启动第二个命令提示窗口）。
3. 打开任务管理器。
4. 切换到第二个命令提示窗口。
5. 输入mspaint（运行Microsoft Paint）。

www.TopSage.com

6. 点击中间的（第二个）命令提示窗口。
7. 输入**exit**（请注意，Paint程序仍然在运行）。
8. 切换到任务管理器。
9. 点击Applications（应用程序）视图。
10. 右键点击Command Prompt任务（即第一个命令提示窗口），然后选择“Go To Process（转到进程）”命令。
11. 点击加亮显示的Cmd.exe进程。
12. 右键点击该进程，选择“End Process Tree（结束进程树）”命令。
13. 在任务管理器的警告消息框上点击“Yes（是）”。

第一个命令提示窗口将会消失，但是你应该仍然看到Paintbrush窗口，因为它是你所终止的命令提示进程的孙子进程；而且，由于中间的进程（即Paintbrush的父进程）已经被终止了，所以父进程和孙子进程之间已经没有链接了。

能够查看（和修改）进程和进程信息的工具有很多可以使用。下面的实验说明了当你使用这些工具时可以看到的各种进程信息。这些工具有的是Windows自带的，有的在Windows支持工具箱中，或者在Windows调试工具箱中，或者在Windows的资源工具箱中，或者在Platform SDK中，或者来自于*www.sysinternals.com*。这些工具大多数显示了有关进程和线程的核心信息的一个子集，有时候使用不同的名称来标识同样的信息。

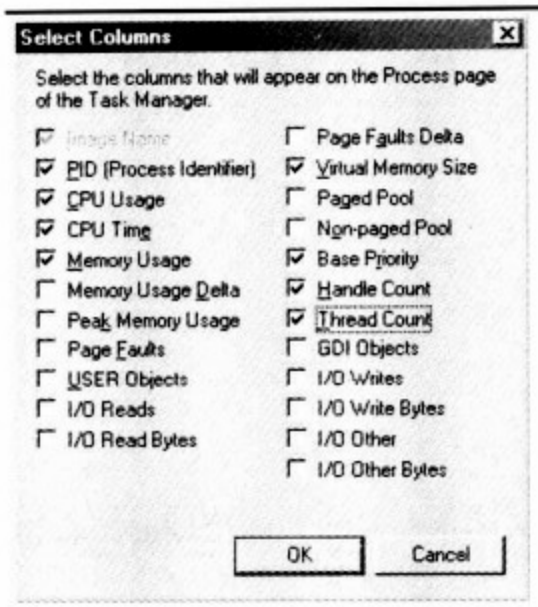
最经常用来检查进程行为状态的工具可能是任务管理器（有意思的是，在Windows的内核中，并不存在“任务”的概念，所以，任务管理器实际上是一个管理进程的工具）。下面的实验显示了任务管理器的应用程序视图和进程视图之间的区别。



实验：用任务管理器来查看进程信息

Windows内置的任务管理器提供了一个快速的进程列表，可以显示出当前系统中运行的进程。你可以使用以下三种方式之一来启动任务管理器：（1）按下Ctrl+Shift+ESC；（2）在任务栏上点击右键，然后选择任务管理器（Task Manager）；或者（3）按下Ctrl+Alt+Del，然后点击任务管理器（Task Manager）按钮。一旦启动了任务管理器以后，点击“Processes（进程）”选项标签可以看到当前正在运行的进程列表。请注意，每个进程是由它的映像文件的名称来命名的。与Windows中其他一些对象不同的是，进程不能被赋予全局的名称。为了显示进一步的细节，你可以从View菜单中选择“Select Columns”命令，然后选择要增加的列，如下图所示：

www.TopSage.com



在任务管理器的进程视图中你所看到的无疑是一个进程列表，然而，在Applications视图中所显示的内容可能就不那么显然了。Applications视图列出了在当前交互的窗口站中所有桌面上的顶级可见窗口(在默认情况下,有两个桌面对象——你可以利用Windows的*CreateDesktop*函数创建更多的桌面对象）。Status列表明了窗口所属的线程是否处于Windows消息等待状态。“Running（正在运行）”表示该线程正在等待窗口输入；“Not Responding（没有响应）”表示该线程并不在等待窗口输入（例如，线程可能正在运行，或者正在等待I/O，或者正在等待某一个Windows同步对象）。

从Applications视图中，你可以在某个任务名称上右键点击，然后选择“Go To Process”，这样就可以将这个任务与它所属的线程以及该线程所在的进程联系起来。

*www.sysinternals.com*的进程管理器（Process Explorer）所提供的关于进程和线程的细节信

www.TopSage.com

息比任何其他的工具都要多，正因为如此，你将会在本书的许多实验中看到这个工具的应用。以下是进程管理器在功能方面的一些独特之处：

- 显示被执行的映像文件的完整路径名；
- 显示进程安全令牌（安全组和特权的列表）；
- 加亮显示进程和线程列表中的变化；
- 列出了服务进程中的服务，包括显示名和说明；
- 显示作业（job）中的进程，以及作业的细节；
- 显示运行.NET/WinFX应用的进程，以及与.NET相关的细节（比如应用域的列表和CLR性能计数器）；
- 显示进程和线程的启动时间；
- 显示内存映射文件（不仅仅包括DLL）的完整列表；
- 能够挂起一个进程；
- 能够杀死单个线程；
- 很容易识别出一段时间以来哪些进程消耗了绝大多数CPU时间（性能工具可以显示出一组给定进程的CPU利用率，但是它不能自动显示那些在性能监视会话启动之后被创建的进程）。

通过进程管理器，也可以很容易地从一个中心位置处，访问到可以通过其他工具来提供的信息，比如：

- 进程树（能够杀掉进程树的一部分）；
- 无需特别设置就可以访问一个进程内的已打开句柄（Microsoft提供的可显示已打开句柄的工具要求先设置一个系统范围的标记，并重新引导系统，然后才可以使用）；
- 列出一个进程内的DLL（和内存映射文件）；
- 一个进程内的线程活动；
- 用户模式的线程栈（而且，利用调试工具的符号引擎，将地址映射为名字）；
- 针对系统线程的内核模式线程栈（而且，利用调试工具的符号引擎，将地址映射为名字）；
- 环境切换的差量（正如第6章所解释的那样，这是一种更好的表达CPU活动的方法）；
- 内核内存（换页的和非换页的内存池）限制（其他工具只显示当前大小）。

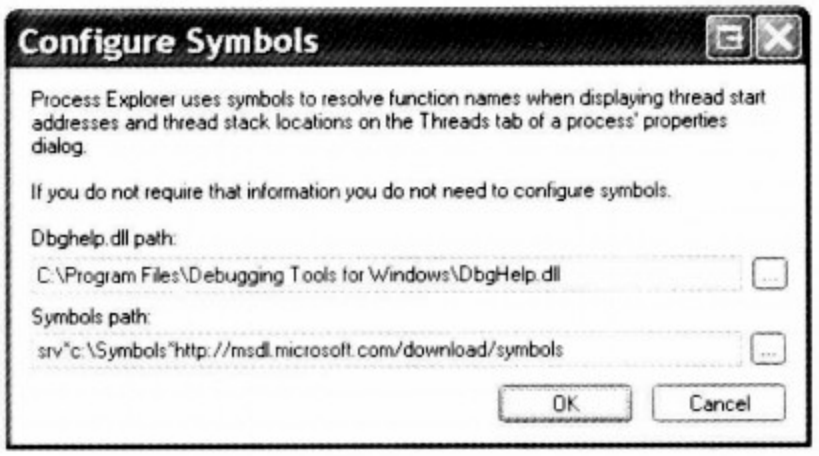
关于进程管理器的用法，一个介绍性的实验如下。

www.TopSage.com



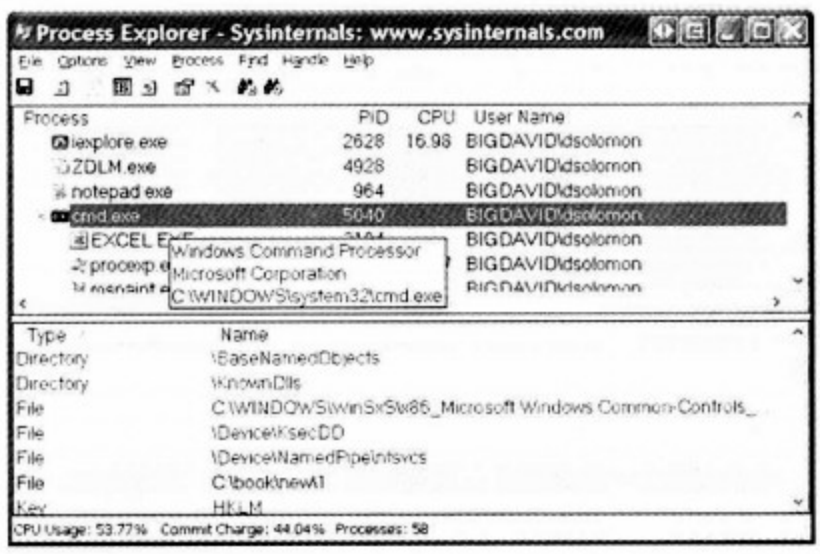
实验：利用进程管理器查看进程的细节

从*www.sysinternals.com*下载一份最新的进程管理器（Process Explorer）软件，然后将它运行起来。当你第一次运行的时候，你会看到一条消息，指出符号尚未被配置。如果符号被正确地配置好的话，则进程管理器可以访问有关的符号信息，以便显示出线程启动函数的符号名，以及线程调用栈中的函数的符号名（双击一个进程，然后点击Threads标签就可以看到线程调用栈）。如果想要知道一个进程内的线程正在做什么事情，则这个功能是非常有用的。为了访问这些符号，你必须已经安装了调试工具箱（本章后面将要介绍）。然后，点击“Options”，选择“Configure Symbols”，填入正确的符号路径（Symbols path）。如下图所示：



在上述例子中，按需符号服务器被用于访问符号，并且这些符号文件被拷贝到本地机器的c:\symbols文件夹中。有关符号服务器配置的更多信息，请参考*http://www.microsoft.com/whdc/ddk/debugging/symbols.mspx*。

当进程管理器启动时，按默认方式，上半部分显示进程列表，下半部分显示出在当前选中的进程内已打开的句柄。当你将鼠标移动到进程名称上的时候，它还会显示出映像文件的描述信息、公司名称和完整的路径。



www.TopSage.com

以下是一些操作步骤，可以让你领略到进程管理器的一些基本功能。

1. 在View菜单中，不要选中“Show Lower Pane”，就可以关掉下面的窗格（下面的窗格可以显示打开的句柄或者被映射的DLL和内存映像文件——第3章和第7章将介绍这些内容）。
2. 注意，在默认配置下，包含服务的进程被加亮显示成粉色，而你自己的进程被显示成蓝色（这些颜色可以被重新配置）。
3. 当你将鼠标移动到进程的映像名上时，可以在提示框中看到完整的路径名。
4. 点击View菜单，选择Columns命令，加入“Image Path（映像路径）”。
5. 按进程列排序，可以注意到，树状视图不见了（你要么显示树状视图，要么按所显示的任何一列进行排序）。再次点击进程列，则从Z到A排序。然后，再次点击，则回到树状视图。
6. 在View菜单中，不要选中“Show Processes From All Users”，这样只显示你自己的进程。
7. 在Options菜单中，选择“Difference Highlight Duration”命令，将编辑框内的值改成5秒。然后启动一个新的进程（任何进程都可以），可以注意到，新进程被加亮显示成绿色5秒钟。再退出这个新的进程，可以看到，该进程在消失之前，被加亮显示成红色5秒钟。这项功能非常有用，利用这项功能，你可以看到系统中新创建的和退出的进程。
8. 最后，在某一个进程上双击，你可以在进程属性对话框的各个标签视图之间来回跳转查看各种信息（在贯穿本书的各个实验中将引用这些信息，那里会解释视图上所显示的信息）。

线程是一个进程内部的实体，也是Windows执行此进程时的调度实体。如果没有线程，则进程的程序不可能运行。线程包括以下一些最基本的部件：

- 一组代表处理器状态的CPU寄存器中的内容；
- 两个栈，一个用于当线程在内核模式下执行的时候，另一个用于线程在用户模式下执行的时候；
- 一个被称为线程局部存储区（TLS，thread-local storage）的私有存储区域，各个子系统、运行库和DLL都会用到该存储区域；
- 一个被称为**线程ID**（*thread ID*，线程标识符）的惟一标识符（在内部也被称为**客户ID**——进程ID和线程ID是在同一个名字空间中生成的，所以它们永远不会重叠）；
- 有时候线程也有它们自己的安全环境，如果多线程服务器应用程序要模仿其客户的安全环境，则往往可以利用线程的安全环境。

易失的寄存器、栈和私有存储区域合起来被称为线程的**环境**（*context*）。因为这些信息随着Windows所在机器的体系结构的不同而有所不同，所以，这些数据结构必须是与底层体系结构相关的。通过Windows提供的*GetThreadContext*函数，你可以访问到这一与体系结构相关的信息（称为CONTEXT块）。

www.TopSage.com

纤程（fiber）与线程

纤程使得一个应用程序可以调度它自己的“线程”的执行过程，而不必依赖于Windows内置的基于优先级的调度机制。纤程常常也被称为“轻量级”的线程。从调度的角度来看，它们对于内核是不可见的，因为它们是在用户模式下实现的，位于Kernel32.dll中。为了使用纤程，首先要调用Windows的*ConvertThreadToFiber*函数。该函数将当前线程转变成一个正在运行的纤程。之后，在转变得到的纤程中，通过调用*CreateFiber*函数，它又可以创建新的纤程（每个纤程可以有它自己的一组纤程）。然而，与线程不同的是，纤程不会自动被执行，你必须手工调用*SwitchToFiber*函数选中一个纤程，才能使它运行。新的纤程一直运行，直到它退出，或者直到它调用*SwitchToFiber*，再次选择运行另一个纤程。有关更多的信息，请参见Platform SDK中有关纤程函数的文档。

虽然线程有自己的执行环境，但是，同一个进程内部的所有线程共享该进程的虚拟地址空间以及其余的属于该进程的资源，这意味着，一个进程内的所有线程可以读或者写另一个线程的内存。然而，一个进程中的线程不可能直接引用另一个进程的地址空间，除非两种情况：第二个进程将它的一部分私有地址空间变成**共享内存区**（在Windows API中称为**文件映射对象**）；或者，第一个进程有权打开第二个进程，从而可以使用诸如*ReadProcessMemory*和*WriteProcessMemory*等跨进程的内存函数。

除了私有的地址空间和一个或多个线程以外，每个进程还有一个安全标识和一个已打开句柄的列表，这些句柄指向诸如文件、共享内存区，或者像互斥体、事件或信号量等同步对象，等等，如图1.2所示。

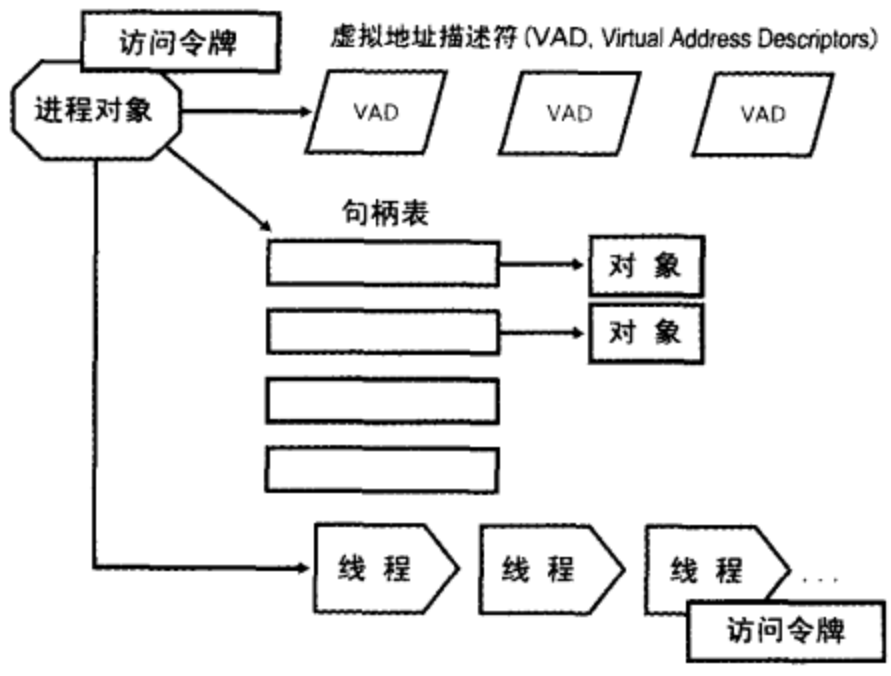


图1.2 进程和它的资源

www.TopSage.com

每个进程都有一个安全环境，被存储在一个称为**访问令牌**的对象中。进程的访问令牌包含了该进程的安全标识和凭证。在默认情况下，线程没有自己的访问令牌，但是它们也可以包含一个访问令牌，因此单独的线程可以模仿另一个进程的安全环境——包括在远程Windows系统上运行的进程——而不会影响到当前进程中的其他线程（更多的细节，请参见第8章中有关进程和线程安全的内容）。

虚拟地址描述符（VAD，*virtual address descriptor*）是指一些数据结构，内存管理器利用这些数据结构来记录一个进程所使用的虚拟地址。第7章将会深入地介绍这些数据结构。

Windows在进程模型上作了一个扩展，称为**作业**（*job*）。作业对象的主要功能是，使得一组进程被当做一个整体来管理和维护。作业对象允许对特定的属性进行控制，也允许对一个进程，或者所有与作业相关联的进程进行限制。作业对象也为所有与该作业相关联的进程记录下基本的审计信息，其中也包括曾经与该作业关联过但是已经终止了的进程的信息。在某些方面，作业对象不仅弥补了Windows平台上缺乏结构化的进程树的不足，而且，它的功能在许多方面比UNIX风格的进程树更为强大。

在第6章中你将会找到更多有关这些话题的信息：作业、进程和线程的内部结构，进程和线程的创建机制，以及线程调度算法。

虚拟内存

Windows实现了一个基于平面（线性）地址空间的虚拟内存系统，每个进程感觉自己独立拥有一个很大的私有地址空间。虚拟内存提供了一个内存逻辑视图，它可能并不对应于内存的物理布局。在运行的时候，内存管理器借助于硬件的支持，将虚拟地址翻译或者**映射**成真正存放数据的物理地址。操作系统通过控制这一层保护或者映射机制，可以保证一个进程不会闯入到另一个进程中，也不会改写操作系统的数据。图1.3显示了三个连续的虚拟页面被映射到三个在物理内存中并不连续的页面上。

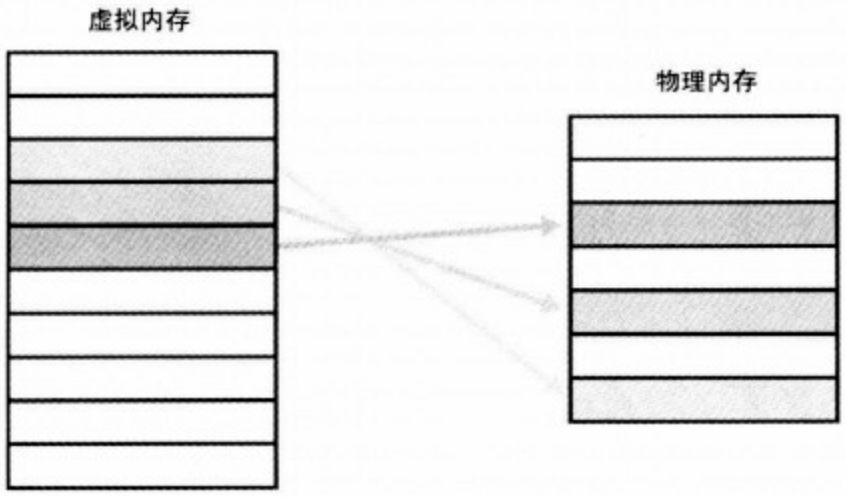


图1.3 将虚拟内存映射到物理内存

www.TopSage.com

因为大多数系统所拥有的物理内存，比当前正在运行的进程所用到的虚拟内存总量要少得多，所以，内存管理器将有些内存中的内容转移或者翻（*page*）到磁盘上。将数据翻到磁盘上以后，就可以释放出这部分物理内存，因此，这些物理内存可以被别的进程所用，或者被交给操作系统本身。当一个线程访问一个已被翻到磁盘上的虚拟地址时，虚拟内存管理器将磁盘上的信息装回到内存中。应用程序无需任何改变就可以利用这种翻页功能，因为在硬件的支持下，内存管理器无需任何关于进程或线程的知识，也无需进程或线程的协助，就可以实现翻页。

虚拟地址空间的大小随着硬件平台的不同而有所差异。在32位x86系统中，总的虚拟地址空间有一个理论上的最大值4GB。在默认情况下，Windows将这部分地址空间的一半（4GB虚拟地址空间较低的一半，从x00000000到x7FFFFFFF）分配给进程，作为它们独有的私有存储区，而另一半（地址空间较高的一半，从x80000000到xFFFFFFFF）则用做它自己的受保护的操作系统内存。低一半地址空间的映射关系反映了当前执行进程的虚拟地址空间，但是，高一半地址空间的映射关系总是由操作系统的虚拟内存构成的。Windows 2000 Advanced Server、Windows 2000 Datacenter Server、Windows XP（SP2及后续版本），以及Windows Server 2003支持一些引导时刻的选项（Boot.ini中的/3GB和/USERVA修饰符，参见第5章中的描述），这使得由那些带有特殊标记的程序（在可执行映像文件的头部设置了大地址空间感知标志）启动起来的进程可以使用多达3GB的私有地址空间（给操作系统留下1GB）。这一选择使得像数据库服务器这样的应用程序可以将更多的数据库内容映射到进程的地址空间中，从而减少了映射数据库子集视图的需求。图1.4显示了32位Windows所支持的两种虚拟地址空间布局结构。

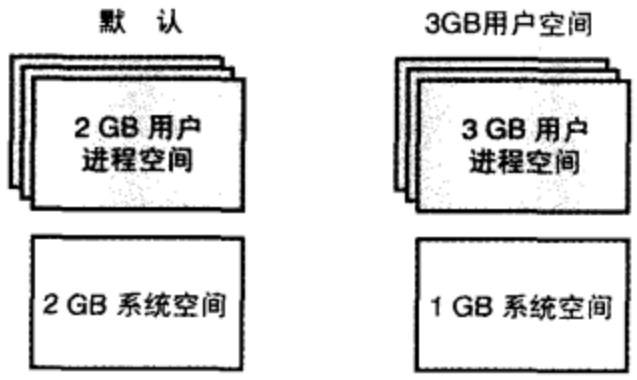


图1.4 32位Windows的地址空间布局结构

虽然3GB的系统明显好于2GB系统，但是，对于映射非常大的数据库（许多个GB）而言，虚拟地址空间仍然不足。针对在32位系统上的这种需求，Windows提供了一种被称为**地址窗口扩展**（AWE，*Address Windowing Extension*）的机制，使得32位应用程序可以申请多达64GB物理内存，然后将内存视图或者窗口映射到它的2GB虚拟地址空间中。虽然AWE将管理“虚拟内存-物理内存”映射关系的负担放到了程序员的身上，但是，它确实解决了问题，使得一

www.TopSage.com

个进程能够直接访问更多的物理内存，超过了它的32位进程地址空间一次能够容纳的数量。

64位Windows为它的进程提供了更大的地址空间：在Itanium系统上为7152GB，在x64系统上为8192GB。图1.5显示了一个简化的64位地址空间布局结构图（有关详细的描述，参见第7章）。请注意，这些大小值并不代表这些平台上的体系结构限制，而是当前版本64位Windows的实现限制。

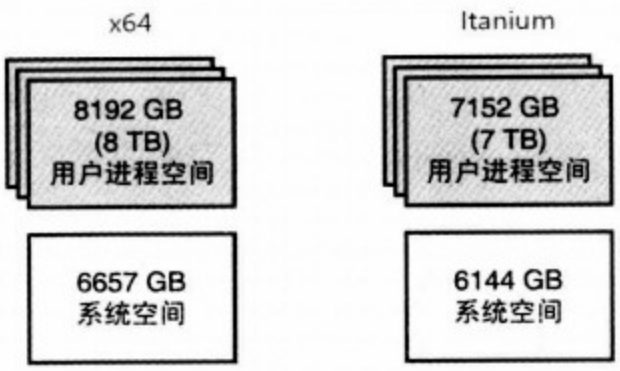


图1.5 64位Windows的地址空间布局结构

内存管理器的实现细节，包括地址翻译如何工作、Windows如何管理物理内存等，将在第7章中讨论。

内核模式和用户模式

为了避免用户应用程序访问和/或修改关键的操作系统数据，Windows使用了两种**处理器访问模式**（即使Windows所在的底层处理器支持多于两种的模式）：**用户模式**和**内核模式**。用户程序代码运行在用户模式下，而操作系统代码（比如系统服务和设备驱动程序）运行在内核模式下。内核模式是指这样一种处理器执行模式：它允许访问所有的系统内存和所有的CPU指令。处理器让操作系统软件比应用软件有更高的特权级，从而为操作系统设计者提供了一层保护的基础，确保一个行为不正常的应用程序不会破坏系统整体的稳定性。



注 Intel x86处理器的体系结构定义了四种特权级，或者称为四个环，来保护系统代码和数据不会被低级别的代码恶意地或无意地改写掉。Windows使用特权级0（或称0环）作为内核模式，特权级3（或称3环）作为用户模式。Windows之所以只用两级的原因是，过去支持的有些硬件体系结构（比如Compaq Alpha和Silicon Graphics MIPS）只实现了两个特权级。

www.TopSage.com

虽然每个Windows进程有它自己私有的内存空间，但是，内核模式的操作系统和设备驱动程序共享同一个虚拟地址空间。虚拟内存中的每一个页面都被标记了处理器必须在什么访问模式下才可以读和/或写该页面。系统空间中的页面只能在内核模式下才可以访问，而用户地址空间中的所有页面在用户模式下都是可以访问的。只读页面（比如那些包含可执行代码的页面）在任何模式下都不是可写的。

对于在内核模式下运行的组件，Windows对它们的读写系统内存并不加以保护。换句话说，一旦进入了内核模式，操作系统和设备驱动程序的代码可以完全访问系统空间的内存，也可以绕过Windows的安全机制直接访问对象。因为有大量的Windows操作系统代码运行在内核模式下，所以，对在内核模式下运行的组件必须要谨慎设计和测试，以确保它们不会破坏系统的安全性，这是非常重要的。

由于缺少保护，因此，在加载第三方设备驱动程序的时候更需要加倍当心，因为一旦进入了内核模式，这些软件就可以完全访问所有的操作系统数据。这一脆弱性正是Windows之所以要引入驱动程序签名的原因之一。引入驱动程序签名机制以后，当一个未经授权的（未签名的）驱动程序企图加入到系统中时，Windows会警告用户（关于驱动程序签名的更多信息，请参见第9章）。而且，一种被称为驱动程序检验器（Driver Verifier）的机制可以帮助设备驱动程序的编写者找到程序中的错误（比如缓冲区溢出或者内存泄漏）。第7章也会介绍驱动程序检验器。

正如你将在第2章中看到的那样，用户应用程序在发出一个系统服务调用的时候，就会从用户模式切换到内核模式下。例如，Windows的*ReadFile*函数最终要调用Windows内部的一个例程，由该例程真正完成从一个文件中读取数据的任务。由于该例程需要访问内部的系统数据结构，所以，它必须运行在内核模式下。从用户模式转换到内核模式，可以通过专门的处理器指令来完成，这条指令会将处理器切换到内核模式下。操作系统捕捉到这条指令，注意到有一个系统服务请求到来，它检查用户线程传递给系统函数的所有实参的有效性，然后执行相应的内部函数。在将控制返回给用户线程以前，处理器的模式被切换回到用户模式。通过这种方法，操作系统将自身保护起来，同时也保护了它的数据不会被用户进程看到或者修改。



注　从用户模式到内核模式的转换（或者从内核模式回到用户模式的转换）本身并不会影响线程的调度——模式转换并不是环境切换。有关系统服务分发的更进一步细节，请参见第3章的内容。

www.TopSage.com

因此，对于一个用户线程来说，它的一部分时间在用户模式下运行，另一部分时间在内核模式下运行，这是非常正常的。实际上，因为大部分图形和窗口系统也运行在内核模式下，所以，图形密集的应用程序花在内核模式下的时间比在用户模式下的时间还要多。一种很容易测试这一现象的办法是，运行一个图形密集的应用程序，比如Microsoft Paint或者Microsoft Pinball，然后利用表1.2列出的性能计数器，就可以看到该应用程序在用户模式和内核模式下的时间分配情况。

表1.2 与模式有关的性能计数器

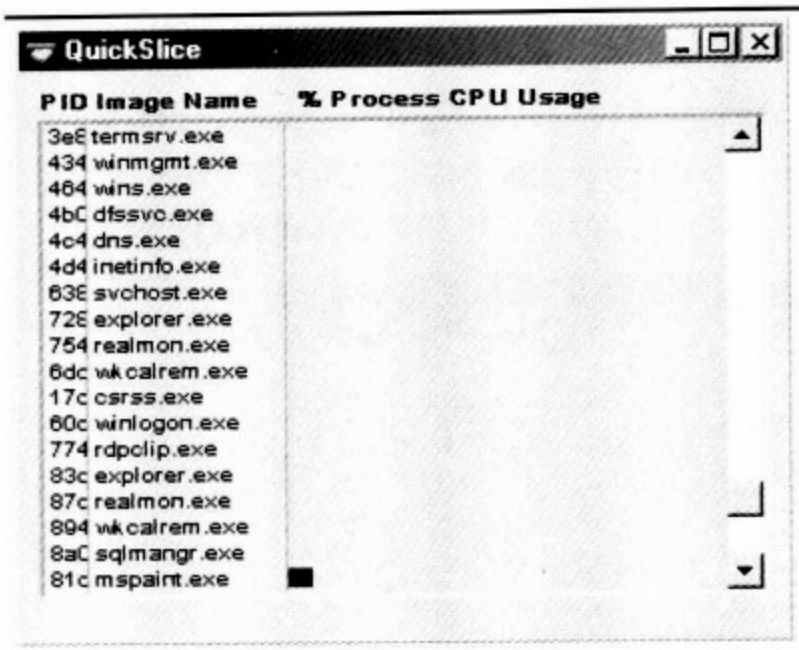
对象：计数器	功 能
Processor: % Privileged Time	在指定的间隔内，单个CPU（或者所有的CPU）运行在内核模式下的时间所占的百分比
Processor: % User Time	在指定的间隔内，单个CPU（或者所有的CPU）运行在用户模式下的时间所占的百分比
Process: % Privileged Time	在指定的间隔内，一个进程中的线程运行在内核模式下的时间所占的百分比
Process: % User Time	在指定的间隔内，一个进程中的线程运行在用户模式下的时间所占的百分比
Thread: % Privileged Time	在指定的间隔内，一个线程运行在内核模式下的时间所占的百分比
Thread: % User Time	在指定的间隔内，一个线程运行在用户模式下的时间所占的百分比



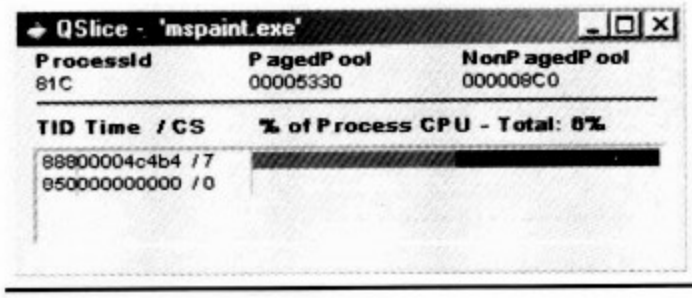
实验：利用QuickSlice来查看线程活动情况

QuickSlice动态地显示了当前系统中正在运行的每个进程所使用的系统和内核时间的比例。在每个进程的这一行上，红色部分显示了该进程在内核模式下所花CPU的时间，蓝色部分显示了用户模式下所花的时间（虽然在下面的窗口中被画成了黑色和白色，但是在实际的显示器上，这些状态条分别是红色和蓝色的）。在QuickSlice窗口中显示的所有状态条，加起来的总和应该是100%的CPU时间。要想运行QuickSlice，你可以点击Start按钮，选择Run，然后输入Qslice.exe（假定你已经将Windows 2000资源工具箱加入到了搜索路径中了）。例如，试着运行一个图形密集的应用程序，比如Paint（Mspaint.exe）。先后打开QuickSlice和Paint，然后在Paint窗口中随便画一些字样。这样做之后，你就会在QuickSlice窗口中看到正在运行的Mspaint.exe如下图所示：

www.TopSage.com



要想查看一个进程中有关线程的进一步信息，你可以在进程上双击鼠标（既可以双击进程名，也可以双击彩色栏）。这样，你可以看到该进程中的所有线程，以及每个线程所使用的相对CPU时间（不能跨系统累计时间），如下图所示。



实验：内核模式和用户模式

你可以使用性能工具来查看你的系统分别在内核模式和用户模式下执行了多少时间。请按照下面的步骤进行操作。

1. 运行性能工具。做法是，打开Start（开始）菜单，依次选择Programs（程序）、Administrative Tools（管理工具）、Performance（性能）命令。
2. 在工具栏上点击“Add”按钮（+）。
3. 在选择了性能对象“Processor”之后，点击“% Privileged Time”计数器，然后，在按住Ctrl键的同时，点击“% User Time”计数器。
4. 点击“Add”按钮，然后点击“Close”。
5. 快速地来回移动鼠标。你应该注意到，当你移动鼠标的时候，“% Privileged Time”性能线快速地升了上去，表明鼠标中断和窗口系统的图形部分花了不少时间（其原因是，正如第2章将要解释的那样，窗口和图形也作为设备驱动程序运行在内核模式下）。参见图1.6。

www.TopSage.com

6. 完成以后，点击工具栏上的“New Counter Set（新计数器集）”按钮（或者关闭该工具）。

利用任务管理器你也可以很快地看到这种行为。只需点击Performance（性能）标签，然后从View（查看）菜单中选择“Show Kernel Times（显示内核时间）”命令。“CPU Usage（CPU使用）”一栏中将会用绿色显示CPU使用总时间，用红色显示内核模式时间。

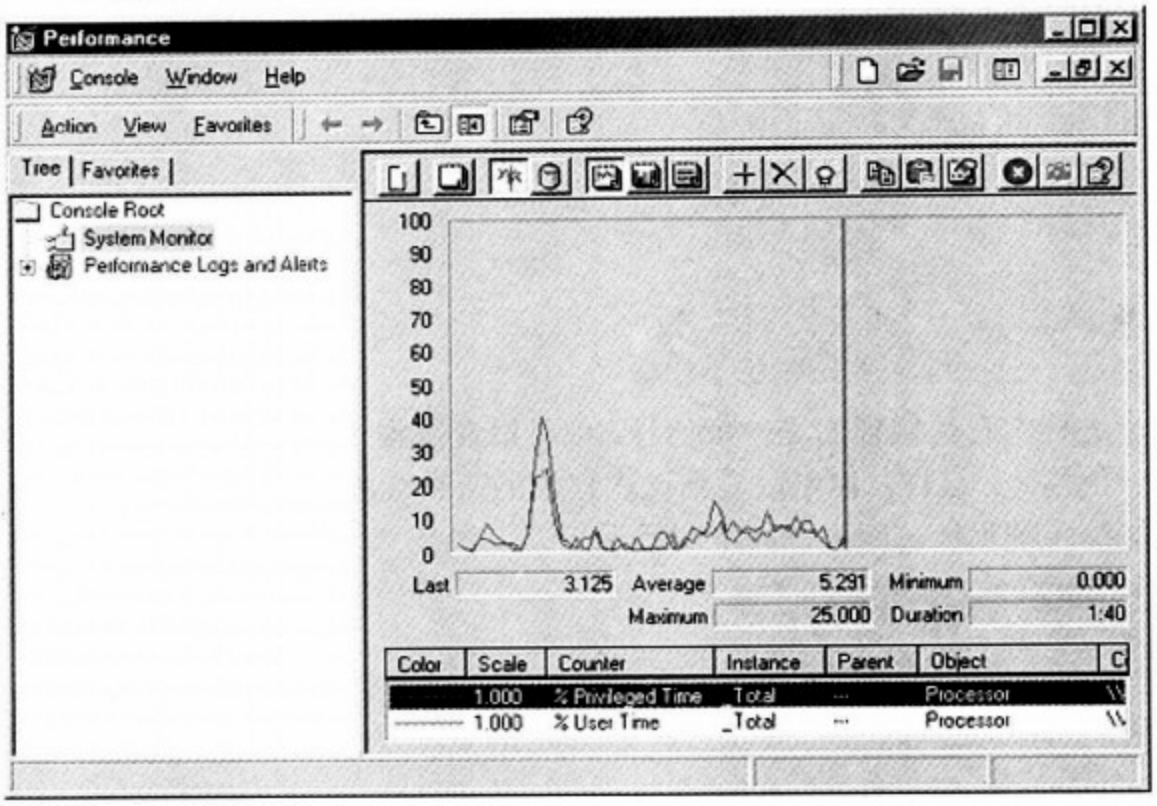


图1.6 性能工具显示了内核模式和用户模式的时间分布情况

为了看清楚性能工具自身如何使用内核时间和用户时间，请再次运行性能工具，但是要为系统中的每个进程加入单进程计数器“% User Time”和“% Privileged Time”：

1. 如果性能工具还没有启动的话，请再次运行性能工具（如果已经在运行了，请按一下工具栏上的“New Counter Set”按钮，以便从一个空白的窗口开始下面的步骤）。
2. 点击工具栏上的“Add”按钮。
3. 将Performance Object（性能对象）改变成“Process”。
4. 选择“% Privileged Time”和“% User Time”计数器。
5. 在Instance框中选择所有的进程（除了“_Total”进程）。
6. 点击“Add”按钮，然后点击“Close”。
7. 快速地回来移动鼠标。
8. 按住Ctrl+H，以打开加亮显示模式。这将会加亮显示当前选中的计数器，在Windows 2000上，加亮成白色，在Windows XP和Windows Server 2003中加亮成黑色。
9. 滚动窗口底部的计数器，以识别出哪些进程的线程在你移动鼠标时正在运行，同时也可以注意到，它们是运行在用户模式下还是运行在内核模式下。

www.TopSage.com

你应该可以看到，当你移动鼠标的时候，性能工具进程（可以在Instance栏中查找mmc进程）的内核模式和用户模式时间也分别升上去了，因为它也在用户模式下执行应用代码，同时也调用了运行于内核模式下的Windows函数。你还可以注意到，在你移动鼠标时有一个名为csrss的进程的内核模式线程活动。之所以会有这样的活动，是因为Windows子系统的内核模式原始输入线程（它负责处理键盘和鼠标输入）也被附在这个进程中（有关系统线程的更多信息，请参见第2章）。最后，你看到有一个名为Idle的进程总是把将近100%的时间花在内核模式下，它并不是一个真正的进程，而是一个假的进程，只是用来记录空闲的CPU周期而已。你可以观察Idle进程中的线程运行在哪种模式下，当Windows无事可做时，它运行在内核模式下。

终端服务及多个会话

终端服务指的是Windows为了在单个系统中支持多个可交互的用户会话而提供的能力。利用Windows的终端服务，一个远程用户可以在另一台机器上建立一个会话，并且登录进去，在该服务器上运行应用程序。服务器把图形用户界面传送到客户机，客户机把用户的输入传回到服务器上（终端服务与UNIX系统上的X窗口有所不同，X窗口允许在服务器系统上运行单独的应用程序，其显示部分远程传送到客户机，但是在终端服务中，整个用户会话都是远程的，不仅仅是单个应用程序）。

在机器的物理控制台上的第一个登录会话被认为是控制台会话，或者零号会话。其他的会话可以通过远程桌面连接程序（Mstsc.exe）来建立，或者在Windows XP系统中，通过使用快速用户切换（后面将会讲述）来建立。

Windows 2000 Server也提供了这种创建远程会话的能力，但是Windows 2000 Professional不支持。Windows XP Professional只允许一个远程用户连接到本地机器上，但如果有人已经在控制台上登录进去了，则工作站被锁住（也就是说，一个人可以在本地或远程使用一个系统，但不能同时以两种方式使用该系统）。

Windows 2000 Server和Windows Server 2003支持两个并发的远程连接。（这是为了方便远程管理，例如，有些管理工具要求登录到被管理的机器中才可以使用。）Windows 2000 Server、Advanced Server、Datacenter Server、Windows Server 2003 Enterprise Edition和Datacenter Edition可以支持两个以上的会话，只需要经过适当的授权许可，并且系统被配置成一个终端服务器即可。

尽管Windows XP Home和Professional版本并不支持多个远程桌面连接，但是，利用一种被称为快速用户切换的特性，它们支持在本地创建多个会话（在Windows XP Professional中，如果系统已经加入到一个域的话，则这项特性被禁止）。当一个用户选择断开会话，而不是注销其会话的话，例如，点击开始菜单，再点击“Log Off（注销）”，并选择“Switch User（切换用户）”，或者在按住Windows键的同时按下“L”键，则当前会话（即在该会话中运行的

www.TopSage.com

进程以及描述该会话的所有数据结构）仍然保留在系统中，而系统返回到主登录屏幕。如果一个新的用户登录进来，则新创建一个会话。

如果应用程序想要知道自己是否运行在一个终端服务器会话中，则Windows提供的一组API可以让应用程序检测到这一点，同时也允许它们控制终端服务的各个方面（细节请参见Platform SDK）。

第2章简要地介绍了这些会话是如何被创建的，同时通过一些实验展示了如何利用各种工具（包括内核调试器）来查看会话信息。第3章的“对象管理器”一节讲述了系统对象名字空间是如何以会话为基础来建立实例的，若应用程序需要知道同一系统中它们的其他实例，该如何做到这一点。最后，第7章介绍了内存管理器如何建立和管理那些会话范围的数据。

对象和句柄

在Windows操作系统中，**对象**是指某一个静态定义的对象类型的单个运行时的实例。**对象类型**包含了一个系统定义的数据类型、在该数据类型的实例上进行操作的一些函数，以及一组对象属性。如果你编写Windows应用程序，那么，你可能会遇到进程、线程、文件和事件对象，这只是对象的一些例子而已。这些对象都是以Windows创建和管理的一些低层对象为基础的。在Windows中，任何一个进程都是进程对象类型的一个实例，文件是文件对象类型的一个实例，如此等等。

对象属性是对象中的数据域，每个对象属性定义了对象的一部分状态。例如，类型为**进程**的对象，其属性包括进程ID、一个基本调度优先级和一个指向访问令牌对象的指针。**对象方法**，即操纵对象的手段，通常读取或者改变对象的属性。例如，进程的*open*方法接受一个进程标识符作为输入，返回一个指向该进程对象的指针作为输出。



> **注** 虽然在使用Windows API或者原生对象服务来创建一个对象的时候，有一个名为*ObjectAttributes*的参数需要调用者来提供，但是，请不要将该参数与本书中使用的更为一般意义的术语“对象属性”相混淆。

对象和普通数据结构之间最根本的区别是，对象的内部结构是被隐藏的。你必须调用一个对象服务才可以获得对象内部的数据，或者把数据置到对象内部。你不可以直接读取或者改变一个对象内部的数据。这一区别将对象的底层实现与那些仅仅使用该对象的代码隔离开来，此技术使得对象的实现可以随着时间而很容易地改变。

www.TopSage.com

对象技术提供了一种便捷的途径来实现下列四个重要的操作系统任务：

- 为系统资源提供了可供人读的名字；
- 在进程之间共享资源和数据；
- 保护资源，避免未授权的访问；
- 引用跟踪，这使得系统能够知道何时一个对象已经不再有用了，因而可以被自动释放。

并不是Windows操作系统中的所有数据结构都是对象。只有确实需要被共享、保护、命名或者让用户模式的程序看得到（通过系统服务）的数据才被放到对象中。仅仅被操作系统的某个组件用来实现其内部函数的数据结构并不是对象。第3章将详细地讨论对象和句柄（即指向对象实例的引用）。

安全性

Windows从一开始设计就要求是安全的，能够满足政府和工业界各种正式的安全等级的需求，比如CCITSE（Common Criteria for Information Technology Security Evaluation，信息技术安全评估的公共标准）规范。达到政府认可的安全等级，可使得一个操作系统在这个领域中更有竞争力。当然，在这些要求中，有许多是针对任何一个多用户系统都有益处的特性。

Windows的核心安全功能包括：针对所有可共享对象（比如文件、目录、进程、线程，等等）的自主保护（need-to-know，有需要知道的理由），安全审计（针对主体或者用户和他们发起的动作的纪录），登录时候的口令认证，以及在一个用户释放了某一资源（比如内存或磁盘空间）以后，另一个用户无法通过访问未初始化资源的办法来看到前一个用户留下的资源。

针对系统内部的对象，Windows有两种访问控制形式。第一种控制形式称为自主访问控制。当大多数人一想到操作系统安全性的时候，自然就会想到这种保护机制。其做法是，由对象（比如文件或者打印机）的所有者授权或者拒绝其他人访问这些对象。当用户登录到系统中时，他们会得到一组安全凭证，或者一个安全环境。当他们试图访问对象的时候，系统会将他们的安全环境与他们要访问的对象上的访问控制列表进行比较，以确定他们是否允许执行所请求的操作。

第二种控制形式称为特权访向控制。当自主访问控制不能满足需要的时候，特权访问控制也是有必要的。这种访问控制方法可以确保：即使无法联系到对象的所有者，某些人也能够访问被保护的对象。例如，如果某员工离开了一家公司，则管理员必须有办法访问到该员工的文件，即使这些文件只有该员工才可以访问。在这种情况下，在Windows中，管理员能够接管这些文件，从而你可以根据需要管理这些文件的权限。

安全性遍及Windows API接口的各个方面。Windows子系统实现了基于对象的安全性，其

www.TopSage.com

方式与操作系统的做法相同：Windows子系统保护共享的Windows对象，在这些对象上设置了Windows安全描述符，从而可避免它们被非法访问。当一个应用程序第一次试图访问一个共享对象的时候，Windows子系统验证该应用程序是否有权这样做。如果安全检查成功，则Windows子系统允许该应用程序继续执行。

Windows子系统在许多共享对象上实现了对象安全性，其中有些共享对象建立在原生的Windows对象之上。Windows对象包括桌面对象、窗口对象、菜单对象、文件、进程、线程和一些同步对象。

有关Windows安全性的全面描述，请参见第8章。

注册表

如果你曾经与Windows操作系统打过交道，那么，你可能听说过或者看到过注册表。要想在不引用注册表的情况下谈论Windows的内部机理，这是不可能做到的，因为注册表是系统数据库，它包含了引导和配置系统所必要的信息、系统范围的控制Windows操作的软件设置、安全数据库，以及针对每个用户的配置信息（比如使用哪个屏幕保护程序）。

同时，注册表是一个反映内存中易失数据的窗口，比如系统中当前硬件的状态（哪些设备驱动程序已经加载到系统中了，它们用到了哪些资源，等等）以及Windows的性能计数器。性能计数器实际上并不“位于”注册表之中，但可以通过注册表函数来访问。第4章更多地介绍了如何通过注册表来访问性能计数器信息。

虽然许多Windows用户和管理员从来都不会直接查看注册表（因为你可以通过标准的管理工具来查看或者改变大多数管理设置），但是，注册表仍然是一个非常有用的查看Windows内部信息的来源，因为它包含了许多会影响系统性能和行为的设置（如果你打算直接改变注册表设置，那你必须绝对非常小心；任何改变都有可能使系统性能变差，或者更糟的是，使得系统无法正常引导）。在本书中，你将会看到许多注册表键，当本书介绍某些组件的时候，就会提及相应的注册表键。本书中引用的绝大多数注册表键位于HKEY_LOCAL_MACHINE下面，我们将该键缩写为HKLM。

有关注册表和它的内部结构的进一步信息，请参见第4章。

www.TopSage.com

Unicode

Windows操作系统区别于大多数其他操作系统的特点之一是，它的大多数内部文本串是以16位宽度的Unicode字符来存储和处理的。Unicode（统一的字符编码标准）是一个国际字符集标准，它为世界上绝大多数已知的字符集定义了惟一的16位值（有关Unicode的更多信息，请参考*www.unicode.org*以及MSDN Library中有关的程序设计文档）。

因为许多应用程序只处理8位（单字节）ANSI字符串，所以，接受字符串参数的Windows函数都有两个入口点：一个Unicode（宽字符，16位）和一个ANSI（窄字符，8位）版本。Windows的以下几个版本，即Windows 95、Windows 98和Windows ME，并没有为所有的Windows函数实现Unicode接口，所以，如果所设计的应用程序要考虑到运行在这些操作系统之上，则往往使用窄字符版本。如果你调用一个Windows函数的窄字符版本，则输入的字符串参数在被系统处理之前，先转成Unicode，而输出的参数则在被返回给应用程序之前，从Unicode转成ANSI字符串。因此，如果你有一个老的服务程序或者一段代码需要运行在Windows上，而这份代码是按照ANSI字符串来编写的，那么Windows将把ANSI字符转成Unicode，以便于内部使用。然而，Windows永远不会转换文件内部的数据——由应用程序来决定是否要存储为Unicode或者ANSI。

在Windows的以前版本中，亚洲和中东的版本是美国和欧洲核心版本的一个超集，其中也包含了额外的Windows函数，它可以满足处理更加复杂的文本输入和排版需求（比如从右至左的文本输入）。到了Windows 2000，所有语种的版本都包含同样的Windows函数。Windows不再使用单独的语言版本，而是有一份全球统一的二进制代码，因此，同一份安装可以支持多种语言（只需加入各种语言包即可）。应用程序也可以利用这一套Windows函数，做到同一份应用程序二进制代码可以支持多种语言。

1.3 挖掘Windows内部机理

虽然本书中的许多信息是在阅读Windows源代码，以及与开发人员交谈的基础上获得的，但是，你也不要相信书中的一切。关于Windows内部机理的许多细节，也可以通过各种实用工具来检查或证实，比如Windows本身随带的工具、Windows支持工具箱、Windows资源工具箱中的工具，以及Windows的调试工具。这一节后面部分将简要地介绍这些工具包。

为了鼓励你自己挖掘Windows的内部机理，我们在本书中插入了名为“实验”的辅助章节，用于说明你可以采取什么样的步骤来检查Windows在某个特殊方面的内部行为（在本章的前面部分你已经看到过这样的章节了）。我们鼓励你试一试这些实验，这样你可以看到，本书中讲述的许多内部细节是确确实实存在的。

表1.3列出了本书中用到的工具，以及它们的来源。

www.TopSage.com

表1.3 用于查看Windows内部机理的工具

工 具	映像名	来 源
启动程序查看器(Startup Programs Viewer)	AUTORUNS	www.sysinternals.com
相依性检查器(Dependency Walker)	DEPENDS	Windows支持工具箱, Platform SDK
DLL列表(DLL list)	LISTDLLS	www.sysinternals.com
EFS信息转储器(EFS Information Dumper)	EFSDUMP	www.sysinternals.com*
文件监视器(File Monitor)	FILEMON	www.sysinternals.com
全局标记(Global Flags)	GFLAGS	Windows支持工具箱
句柄查看器(Handle Viewer)	HANDLE	www.sysinternals.com
连接工具(Junction tool)	JUNCTION	www.sysinternals.com
内核调试器(Kernel debuggers)	WINDBG、KD	调试工具箱，Platform SDK，Windows DDK
实时内核调试器(Live Kernel Debugging)	LIVEKD	www.sysinternals.com
登录会话(Logon Sessions)	LOGINSESSIONS	www.sysinternals.com
对象查看器(Object Viewer)	WINOBJ	www.sysinternals.com
打开的句柄(Open Hanldes)	OH	资源工具箱
页面错误监视器(Page Fault Monitor)	PFMON	Windows支持工具箱、资源工具箱、Platform SDK
未完成的文件移动(Pending File Moves)	PENDMOVES	www.sysinternals.com
性能工具(Performance tool)	PERFMON.MSC	Windows内置工具
管道列表工具(Pipe List tool)	PIPELIST	www.sysinternals.com
内存池监视器(Pool Monitor)	POOLMON	Windows支持工具箱、Windows DDK
进程管理器(Process Explorer)	PROCEXP	www.sysinternals.com
获取SID的工具(Get SID tool)	PSGETSID	www.sysinternals.com
进程统计工具(Process Statistics)	PSTAT	Windows支持工具箱、Windows 2000资源工具箱、Platform SDK、www.reskit.com
进程查看器(Process Viewer)	在Windows支持工具箱中为PVIEWER；在Platform SDK中为PVIEW	Platform SDK
Quick Slice	QSLICE	Windows 2000资源工具箱
注册表监视器(Registry Monitor)	REGMON	www.sysinternals.com
服务控制(Service Control)	SC	Windows XP、Platform SDK、Windows 2000资源工具箱
任务(进程)列表[Task（Process）List]	TLIST	Windows调试工具箱
任务管理器(Task Manager)	TASKMGR	Windows内置的工具
TDImon	TDIMON	www.sysinternals.com

www.TopSage.com

性能工具

在本书中，我们经常会提到性能工具，你可以通过Start菜单的Administrative Tools文件夹找到该工具（也可以通过控制面板找到它）。性能工具有三项功能：系统监视、查看性能计数器日志，以及设置报警。为简化起见，我们在提到性能工具的时候，实际是指该工具内部的系统监视功能。

性能工具所提供的有关操作系统当前运行状况的信息，比任何其他单个工具所能提供的信息要多得多。它针对各种对象提供了几百个计数器。对于本书中讲述的每个主要话题，都附有一张相关的Windows性能计数器的表格。

性能工具为每个计数器提供了一份简短的描述。要想查看这些描述信息，请在“Add Counter”窗口中选择一个计数器，然后点击“Explain”按钮；或者打开资源工具箱中的“Performance Counter Reference”帮助文件。要想准确地解释这些计数器以便检测瓶颈或者做好容量计划，你可以参考*Windows 2000 Server Operations Guide*中的“性能监视（Performance Monitoring）”部分的内容，该操作手册是Windows 2000 Server资源工具箱的一部分。对于那些非常想了解Windows性能的人来说，“性能监视”部分中的章节提供了一种极好的描述。对于Windows XP和Windows Server 2003，你可以参考Windows Server 2003资源工具箱中的性能计数器参考文档（可以通过*www.microsoft.com*在线获得）。

注意，通过编写程序可以访问所有的Windows性能计数器。第4章的“HKEY_PERFORMANCE_DATA”部分简要地介绍了一些组件，其中涉及通过Windows API来获得性能计数器的做法。

Windows支持工具箱

Windows支持工具箱中包含了大约40个工具，它们对于管理和维护Windows系统非常有用。在这些工具中，有许多原来是Windows NT资源工具箱的一部分。

你只需在任何一个Windows产品发布介质的\Support\Tools文件夹中运行Setup.exe，就可以安装Windows支持工具箱。Windows 2000有4个不同版本，即Windows 2000 Professional、Server、Advanced Server和Datacenter Server，其支持工具箱是相同的。Windows XP有它自己的支持工具箱版本，Windows Server 2003也一样。

Windows资源工具箱

Windows资源工具箱补充了Windows支持工具箱，增加了一些用于系统管理和技术支持的工具。Windows 2003资源工具箱中的工具可通过*www.microsoft.com*免费下载（搜索“resource kit tools”）。这些工具可以安装在Windows XP或者Windows Server 2003上。

www.TopSage.com

Windows 2000资源工具箱有两个版本：Windows 2000 Professional资源工具箱和Windows 2000 Server资源工具箱（Supplement 1是最新的版本）。虽然后者是前者的超集，而且后者也可以安装到Windows 2000 Professional系统中，但是，本书中没有一个实验是使用仅包含于Windows 2000 Server资源工具箱中的工具来完成的。与Windows Server 2003资源工具箱不同的是，这些工具不是免费下载的。然而，MSDN和TechNet的订户可以在他们的订阅包中获得Windows 2000 Server资源工具箱。

内核调试

内核调试意味着检查内核的内部数据结构，以及/或者跟踪内核中的函数。能够探查Windows的内部机理，这是一条很有用的途径，因为你可以显示那些通过其他工具无法得到的内部系统信息，并且对于内核内部的代码流有一个更加清晰的认识。

通过以下一些工具可以进行内核调试：Microsoft的Windows调试工具箱、*www.sysinternals.com*提供的LiveKD、Compuware NuMega的SoftIce。在介绍这些工具之前，我们先介绍一个文件，你在执行任何一种内核调试时都需要这个文件。

内核调试所需要的符号

符号文件包含了函数和变量的名称。它们是由链接器产生的，在调试过程中被调试器用来引用和显示这些名称。这些名称信息通常并不存储在二进制映像文件中，因为在执行代码的时候并不需要用到这些信息。这意味着，二进制映像文件更加小巧，也更加快速。然而这也意味着，在调试的时候，你必须确保在一个调试会话中，调试器能够访问到与正在调试的映像文件相关联的符号文件。

为了使用任何一个内核调试工具来检查Windows内核的内部数据结构（比如进程列表、线程块、已加载的驱动程序列表、内存使用信息，等等），你必须至少得到了内核映像Ntoskrnl.exe的正确符号文件（第2章的“总体结构”一节介绍了有关这个文件的更多信息）。符号表文件必须与这些符号所在的映像文件的版本完全匹配。例如，如果你安装了Windows的一个服务补丁（Service Pack）或者热补丁（Hot Fix），那么，你必须至少获得与该内核映像相匹配的、更新过的符号文件；否则，当你试图用内核调试器来加载符号文件时，你会得到一个校验和错误。

为各种版本的Windows下载并安装符号文件是有可能做到的，但是，为每一个热补丁更新符号文件则并不总是可行的。要想获得当前调试所需符号的正确版本，最容易的办法是，利用Microsoft的按需符号服务器，你只要在调试器中用一种特殊的语法来指定符号路径即可。例如，下面的符号路径将使得调试工具从Internet的符号服务器上加载所需要的符号，并且在本地的C:\symbols文件夹下面维护一份拷贝：

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


www.TopSage.com

有关如何使用符号服务器的更详细指令，请参考Windows调试工具箱的帮助文件或者Web页面*www.microsoft.com/whdc/ddk/debugging/symbols.mspx*。

Windows调试工具箱

Windows调试工具箱包含了本书中用于挖掘Windows内部机理的高级调试工具。你可以在*www.microsoft.com/whdc/ddk/debugging*上找到最新的版本。这些工具既可以用于调试用户模式的进程，也可以调试内核（请参见下面副标题中的辅助内容）。



> **注** Windows调试工具箱的版本更新非常频繁，并且独立于Windows操作系统的版本，所以请经常检查最新的版本。

用户模式的调试

调试工具也可以用来附载到一个用户模式的进程中，然后检查和改变该进程的内存。当附载到一个进程上时有以下两种选择。

- **侵入式的** 除非另行指定，否则，当你将调试器附载到一个正在运行的进程上时，将通过Windows函数*DebugActiveProcess*，在调试器和被调试的程序之间建立一个连接。这使得程序员可以检查或者修改该进程的内存，设置断点，执行其他的调试功能。在Windows 2000中，当调试器退出的时候，被调试的进程也随之被杀掉。然而，从Windows XP开始，你可以断开调试器的连接，不用杀掉目标进程。
- **非侵入式的** 在这种选择方式下，调试器只是简单地利用*OpenProcess*函数打开被调试的进程。但是，它并不作为一个调试器被附载到目标进程中。这使得你可以检查和修改目标进程中的内存，但是你不能设置断点。这种选择的好处是，在Windows 2000上你可以在不杀掉目标进程的情况下退出调试器。

你也可以利用调试工具来打开用户模式的进程转储文件。在第3章介绍异常分发的内容中将会解释用户模式的进程转储文件。

可用于内核调试的Microsoft调试器有两种主要的形式：命令行版本（Kd.exe）和图形用户界面（GUI）版本（Windbg.exe）。这两个版本提供了同样的命令集，所以，你可以根据自己的习惯和爱好选择其中之一。利用这些工具你可以执行以下三种类型的内核调试。

- 打开一个崩溃转储文件。当Windows系统崩溃的时候，会产生这样的文件（有关崩溃转储的更多信息，请参见第14章）。
- 连接到一个正在运行的系统上，并检查该系统的状态；或者，如果你正在调试设备驱动程序的代码，则可以设置断点。这种操作要求有两台计算机——目标计算机和控制主机。目标计算机是指要调试的目标系统，控制主机是指运行调试器的主机。目标系统既可以是本

www.TopSage.com

地的（通过零调制解调器线或者IEEE 1394线连接至控制主机），也可以是远程的（通过调制解调器连接至控制主机）。目标系统在引导的时候必须带有/DEBUG修饰符（可以在引导过程中按下F8，然后选择调试模式，或者在Boot.ini中加入一个引导选择项）。

- 对于Windows XP和Windows Server 2003系统，连接到本地的系统上，然后检查系统的状态。这称为**本地内核调试**。为了启动本地内核调试，选择File菜单项，再选择“Kernel Debug”，点击Local标签，再点击OK按钮。图1.7显示了输出屏幕的一个示例。在本地内核调试模式下，内核调试器的有些命令不能正常工作（比如查看内核栈、用.dump命令创建一个内存转储）。然而，当基本的本地内核调试功能不能工作的时候，你可以使用*www.sysinternals.com*的免费LiveKd工具（参见下一部分的介绍）。

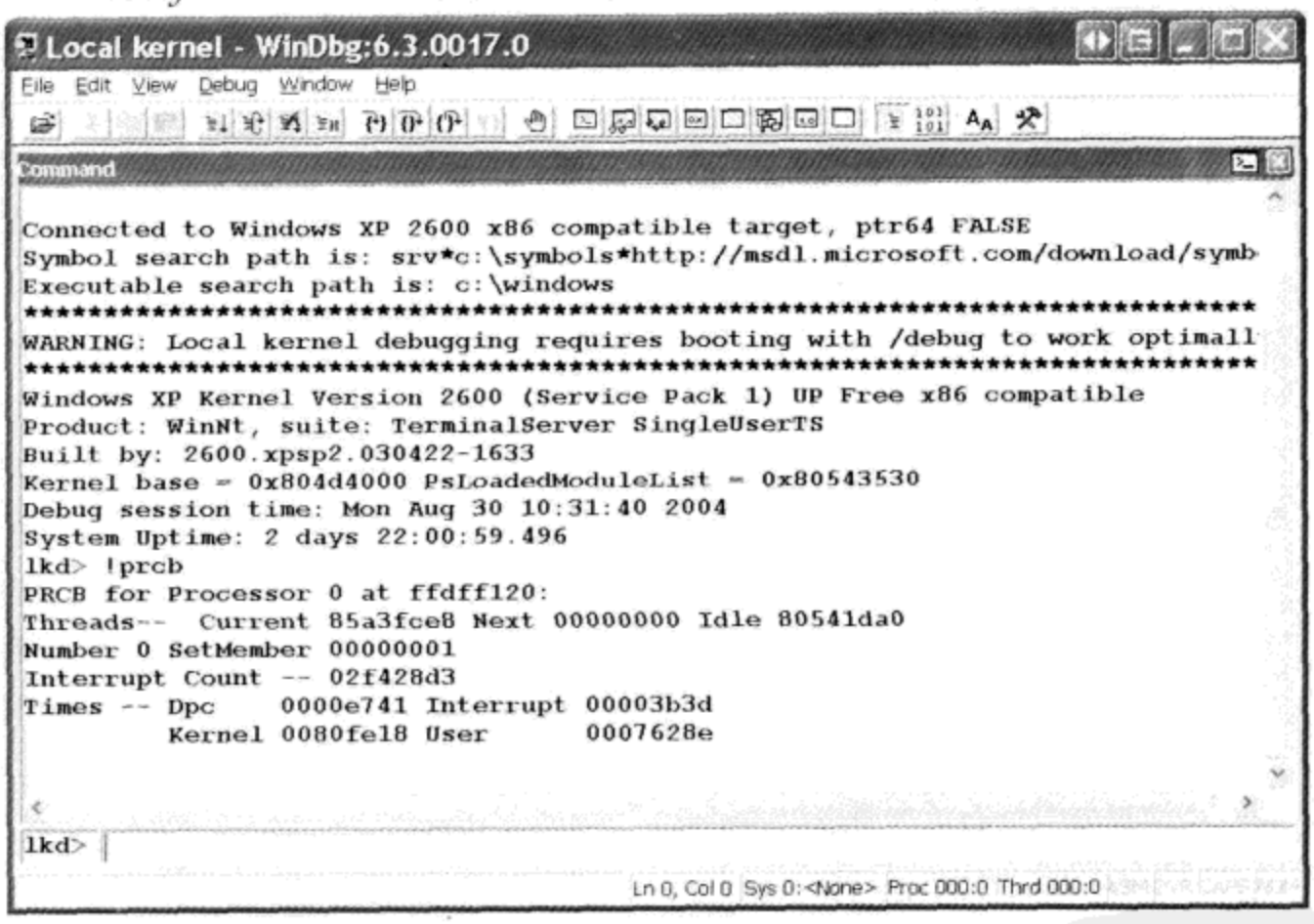


图1.7 本地内核调试

一旦连接到了内核调试模式中，你可以使用众多的**内核扩展命令**（以“!”开头）中的任何一个来显示内部数据结构的内容，比如线程、进程、I/O请求包，以及内存管理信息。在本书中，每当讨论到有关的话题时，就会提到可适用于该话题的内核调试命令，以及相应的输出。而且，*dt*（显示类型）命令可以对400多个内核数据结构进行格式化，因为调试器可以利用Windows 2000 Service Pack 3、Windows XP和Windows Server 2003的内核符号文件中所包含的类型信息进行数据结构格式化。

www.TopSage.com



实验：显示内核数据结构的类型信息

为了显示内核符号中所包含的类型信息的内核结构列表，可在内核调试器中输入**dt nt!_***。下面显示了一个示例性的部分输出：

```
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
```

你也可以利用dt命令的通配符查找能力，来查找某一些特定的数据结构。例如，如果你要查找一个中断对象的结构名，你可以输入**dt nt!_*interrupt***：

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

然后，你可以用dt来对一个特定的数据结构进行格式化，如下所示：

```
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
```


www.TopSage.com

注意，dt在默认情况下并不显示子结构（结构内部的结构）。为了递归显示子结构，请使用“-r”开关。例如，利用这个开关来显示内核中断对象，可以得到存储在InterruptListEntry域中的_LIST_ENTRY结构的格式：

```
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
```

Windows调试工具箱的帮助文件解释了如何安装和使用内核调试器。针对设备驱动程序的编写者如何使用内核调试器的更进一步细节，可以在Windows DDK文档中找到。除此以外，还有几篇非常有用的针对内核调试器的Knowledge Base文章，你在Windows Knowledge Base（在*support.microsoft.com*上，是一个在线的技术文章数据库）中通过搜索“debugref”可以找到它们。

LiveKd工具

LiveKd是*www.sysinternals.com*上的一个免费工具，它使得你可以使用上面刚刚介绍的标准Microsoft内核调试器来检查一个正在运行的系统，而无需另一台计算机作为控制主机（通过零调制解调器线）。标准内核调试器内置的本地内核调试模式只能在Windows XP和Windows Server 2003上才能工作，而LiveKd使得在Windows NT、Windows 2000、Windows XP和Windows Server 2003所有这些系统上都可以使用本地内核调试模式。

你可以像运行Windbg或者Kd那样运行LiveKd。LiveKd把你指定的所有命令行选项传递给你所选择的调试器。在默认情况下，LiveKd运行新的命令行内核调试器（Kd）。要想运行GUI调试器（Windbg），请指定-w开关。要想查看有关LiveKd开关的帮助文件，请指定-?开关。

LiveKd提供给调试器的是一个模拟的崩溃转储文件，所以，你在LiveKd中可以执行任何一个针对崩溃转储而支持的操作。因为LiveKd依赖于物理内存来支持模拟的崩溃转储，所以内核调试器有可能会运行到这样一种状态：有些数据结构正处于被系统修改的过程之中，因而是不一致的。每当调试器被启动起来时，它会获得当前系统状态的一个快照，所以，如果你想要刷新系统快照，则可以退出调试器（用“q”命令），LiveKd将问你是否要再次启动起来。如果调试器进入到一个打印输出的循环之中，你可以按下Ctrl+C中断输出，退出，并返回。如果调试器挂起了，请按下Ctrl+Break，这将会终止调试器进程，并且问你是否要再次运行调试器。

www.TopSage.com

SoftICE

另一个调试工具也不要求用两台机器来调试一个正在运行的系统，这就是被称为SoftICE的第三方内核调试器。你可以从Compuware NuMega购买该工具（详细情况请参见*www.compuware.com*）。在本质上，SoftICE具有与Windows调试工具相同的调试能力，但是它支持在用户模式和内核模式之间的跳转，也支持Microsoft内核扩展DLL，所以，我们在本书中讲述的大多数命令在SoftICE中同样可以工作。图1.8显示了SoftICE的用户界面。当你在运行SoftICE的机器上按下SoftICE的激活键（默认是Ctrl+D）时，桌面上就会出现此窗口。

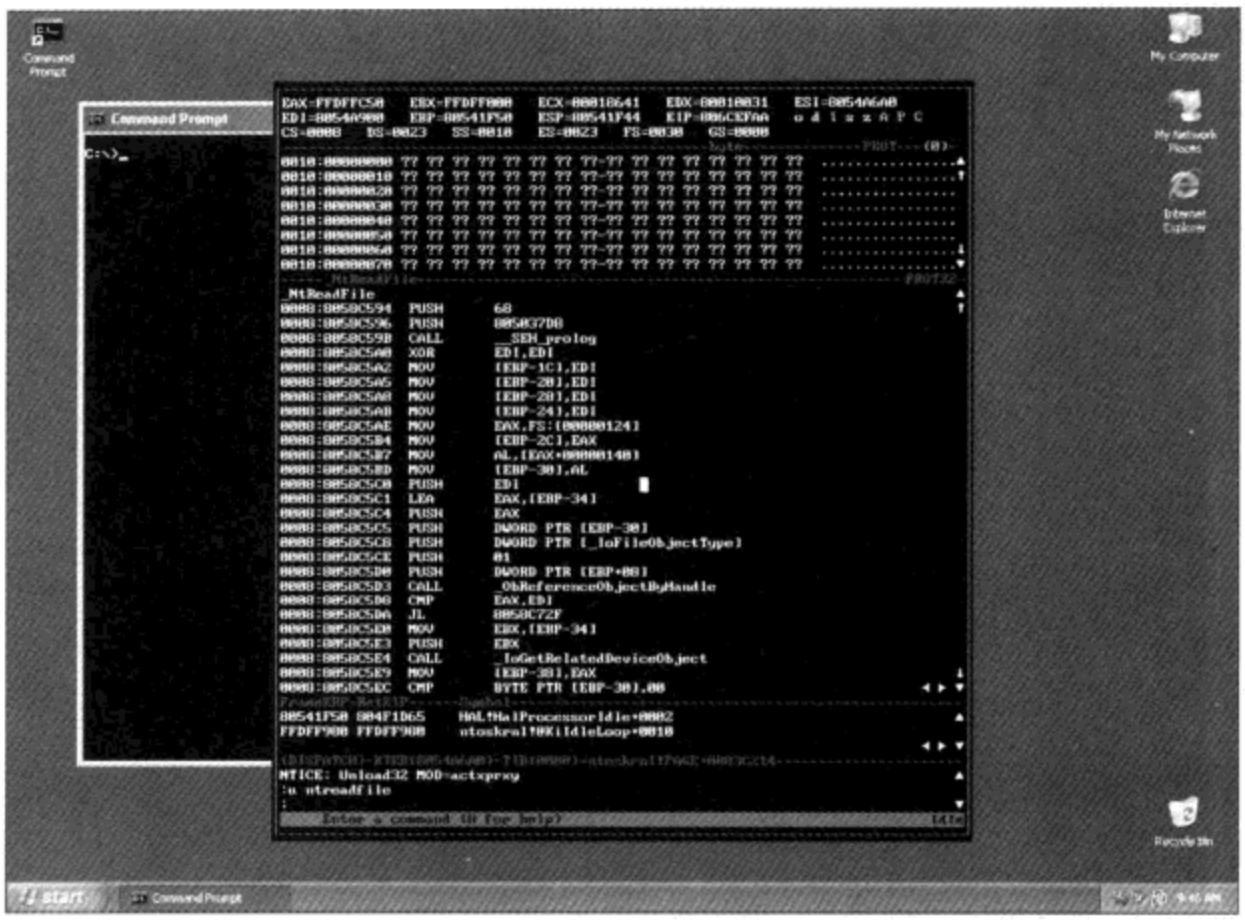


图1.8 SoftICE界面

Platform SDK

Platform SDK是MSDN Professional和更高级订阅包的一部分，或者你也可以从*msdn.microsoft.com*上免费下载。它包含了各种用于编译和链接Windows应用程序所必需的文档、C头文件和库文件（虽然Microsoft Visual C++也带了这些头文件，但是，Platform SDK中包含的头文件版本总是符合Windows操作系统的最新版本，而Visual C++随带的头文件可能是老的版本，只符合Visual C++刚刚发布时的系统版本）。从考察内部机理的角度来看，Platform SDK中令人感兴趣的项目有Windows API头文件（\Program Files\Microsoft SDK\Include）以及一些实用工具（Pfmon.exe、Pstat.exe、Pview.exe、Vadump.exe和Winobj.exe）。Platform SDK中的有些工具也包含在Windows支持工具箱和资源工具箱中。最后，其中有些工具的源代码也包含在Platform SDK和MSDN Library的例子源代码中。

www.TopSage.com

DDK（设备驱动程序开发工具）

Windows DDK包含在MSDN Professional，以及更高级的订阅包中。但是，与Platform SDK不同的是，它不可以从网络上免费下载（但你可以订购CD-ROM，只需支付少量的费用）。Windows DDK的文档也包含在MSDN Library中。

虽然DDK针对的目标用户是设备驱动程序的开发人员，但它也是考察Windows内部机理的一个丰富的信息源。例如，虽然第9章描述了I/O系统结构、驱动程序模型，以及基本的设备驱动数据结构，但是它并没有详细地描述单独的内核支持函数。DDK文档包含了所有用于设备驱动程序的Windows内核支持函数和各种机制的详细说明，而且有两种形式：教程形式和参考形式。

DDK中除了包含文档以外，还包含了相应的头文件（特别是Ntddk.h和Wdm.h），这些头文件定义了关键的内部数据结构和常数，以及许多内部系统例程的接口。当你用内核调试器来挖掘Windows内部数据结构的时候，这些文件非常有用，因为尽管本书中给出了这些数据结构的一般性布局形式和内容，但是域（field）一级的详细描述（比如大小和数据类型）并没有给出。然而，这些数据结构中的相当一部分[比如对象分发器的头、等待块、事件、突变体（mutant）、信号量，等等]在DDK中有详细的描述。

所以，如果你想在阅读本书的基础上，进一步钻研I/O系统和驱动程序模型，则可以阅读DDK的文档[特别是内核模式驱动程序结构设计指南和参考手册（Kernel-Mode Driver Architecture Design Guide and Reference manuals）]。另一份很有价值的参考资源是Walt Oney著的*Programming the Microsoft Windows Driver Model*（第2版，Microsoft Press）。

Sysinternals工具

本书中的许多实验用到了一些免费软件工具，你可以从*www.sysinternals.com*下载。这些工具中的绝大部分是由本书的作者之一Mark Russinovich编写的。最为流行的工具包括进程管理器（Process Explorer）、Filemon和Regmon。请注意，这些工具中有许多要安装和执行内核模式设备驱动程序，因此需要管理员特权。

1.4 本章总结

本章介绍了本书中将要用到的一些关键的Windows技术概念和术语。许多是可用于挖掘Windows内部机理的实用工具，你也已经看到了。现在，我们已经准备好挖掘系统的内部设计了，首先，我们从总体系统结构和它的关键部件开始我们的挖掘之旅。

www.TopSage.com

System Architecture

第2章 系统结构

现在我们已经了解了必须熟悉的术语、概念和工具，所以，我们准备开始挖掘Microsoft Windows操作系统的内部设计目标和数据结构。这一章讲述系统的总体结构——关键的部件、它们相互之间如何交互，以及它们分别运行在什么样的环境下。为了提供一个有助于理解Windows内部机理的框架，首先回顾一下最初的需求和设计目标，这些需求和目标基本上勾画出了Windows系统最初的设计和规范。

2.1 需求和设计目标

回到1989年，下面的需求导致了Windows NT的以下规范：

- 提供一个真正32位的、抢先式的（preemptive）、可重入的（reentrant）虚拟内存操作系统；
- 在多种硬件体系结构和平台上运行；
- 可在对称多处理器系统（symmetric multiprocessing systems）上运行，并且能很好地适应处理器的数量；
- 成为一个主要的分布式计算平台，无论是作为网络客户还是服务器；
- 能够运行大多数已有的16位MS-DOS和Microsoft Windows 3.1应用程序；
- 符合政府对于POSIX 1003.1兼容性的要求；
- 符合政府和工业界对于操作系统安全性方面的要求；
- 支持Unicode，以便很容易地适应全球市场。

要创建一个满足这些需求的系统，必须做出数千个决定；为了便于做出这些决定，Windows NT设计小组在项目开始之初选择了下面的设计目标：

- **扩展性**（*Extensibility*） 编写的系统代码必须能够随着市场需求的变化而自如地增长和改变；
- **可移植性**（*Portability*） 系统必须能运行在多种硬件体系结构上，必须能根据市场的需要，相对容易地移到新的体系结构上；
- **可靠性和健壮性**（*Reliability and Robustness*） 系统应该能够保护自己，不会因内部的错误和外部的篡改而不能工作。应用程序应该无法伤害操作系统或者其他的应用程序；

www.TopSage.com

- **兼容性**（*Compatibility*） 虽然Windows NT应该扩展已有的技术，但是它的用户界面和API应该与老版本的Windows和MS-DOS兼容。而且它也应该能与其他的系统，比如UNIX、OS/2和NetWare，很好地互操作；
- **性能**（*Performance*） 在其他设计目标的约束下，系统在每一种硬件平台上应尽可能运行得更快，对外部的响应尽可能地及时。

随着我们挖掘Windows内部结构和内部操作的各种细节，你将会看到，这些原始的设计目标和市场要求是如何成功地融入到系统的构造过程中的。但是，在我们开始挖掘内部细节之前，先来看一下Windows的总体设计模型，并且将它与其他的现代操作系统作一比较。

2.2 操作系统模型

在大多数多用户操作系统中，应用程序与操作系统本身是隔离的——操作系统内核代码运行在处理器的特权模式下（在本书中称为**内核模式**，*kernel mode*），可以访问系统数据和硬件；应用程序代码运行在处理器的非特权模式下（称为**用户模式**，*user mode*），只有很有限的一组接口可以使用，对系统数据的访问受到限制，并且无法直接访问硬件。当用户模式程序调用一个系统服务时，处理器捕获到该调用，然后将调用线程切换到内核模式。当该系统服务完成时，操作系统将线程环境切换回用户模式，并允许调用者继续执行。

与大多数UNIX系统类似，Windows是一个庞大而完整的操作系统；操作系统的大部分代码与设备驱动程序代码共享同样的受保护的内核模式内存空间。这意味着，操作系统的任一组件或者设备驱动程序都有可能破坏其他系统组件所使用的数据。

Windows是一个微内核系统吗

尽管有些人如此声称，但是按照微内核（microkernel）的典型定义，Windows并不是一个微内核系统。在微内核的定义中，操作系统的主要组件（比如内存管理器、进程管理器和I/O管理器）运行在各自独立的进程中，它们有自己私有的地址空间，在这一组组件之上是微内核提供的一组原语服务。例如，CMU（Carnegie Mellon University，卡耐基·梅隆大学）的Mach操作系统是一个现代的微内核体系结构的例子，它实现了一个最小的内核，包括线程调度、消息传递、虚拟内存和设备驱动程序。任何其他的组件，包括各种API、文件系统和网络等，都运行在用户模式下。然而，Mach微内核操作系统的商业版本实现，往往至少将所有的文件系统、网络和内存管理代码放在内核模式中运行。原因很简单：纯粹的微内核设计在商业上是不切实际的，因为它的效率太低。

www.TopSage.com

那么，Windows中有那么多代码运行在内核模式下，是否就意味着它比一个真正的微内核操作系统一定更加容易崩溃呢？并非如此。请考虑下面的情景。假设在操作系统的文件系统代码中有一个错误，该错误会导致文件系统时不时地崩溃。在一个传统的操作系统中，内核模式代码（比如内存管理器或者文件系统）中的错误很可能会使整个操作系统崩溃。在一个纯粹的微内核操作系统中，这样的组件运行在用户模式下，所以，在理论上说，这样的错误只是意味着该组件的进程将会退出。但是，从实际的角度来看，系统将会崩溃，因为要从这样一个关键进程的失败中恢复过来往往是不可能的。

当然，操作系统的所有组件都是受保护的，不会被错误的应用程序破坏，因为应用程序不能直接访问操作系统中特权部分的代码和数据（尽管它们可以快捷地调用其他的内核服务）。Windows无论是作为一个应用服务器，还是一个工作站平台，从操作系统核心服务（比如虚拟内存管理、文件I/O、网络，以及文件和打印共享）的角度来看，它都很快地赢得了健壮性和稳定性两方面的良好声誉，之所以如此，操作系统组件的这种保护性正是原因之一。

Windows的内核模式组件也体现了基本的面向对象设计原则。例如，它们通常并不会直接进入另一个组件的数据结构来访问该组件所维护的信息。相反，它们利用正式的接口来传递参数，以及访问或修改相应的数据结构。

虽然Windows内部普遍地使用了对象来表达共享的系统资源，但是，从严格意义上讲，Windows并不是一个面向对象的系统。为了移植性的考虑，Windows操作系统的大多数代码是用C语言编写的，因为C语言的开发工具更为广泛可用。C语言并不直接支持面向对象的概念，比如数据类型的动态绑定、多态函数或者类继承等。因此，Windows中C语言版本的对象实现只是借用了（但并不依赖于）特定面向对象语言的特性。

2.3 总体结构

在这一节关于Windows设计目标和包装方式的简短介绍中，我们先来看一看Windows总体结构中的关键系统组件。一个简化版本的总体结构如图2.1所示。请记住，这个图只是一个基本结构，它并没有显示任何特定内容（例如，网络组件和各种类型设备驱动程序的层次并没有显示）。

www.TopSage.com

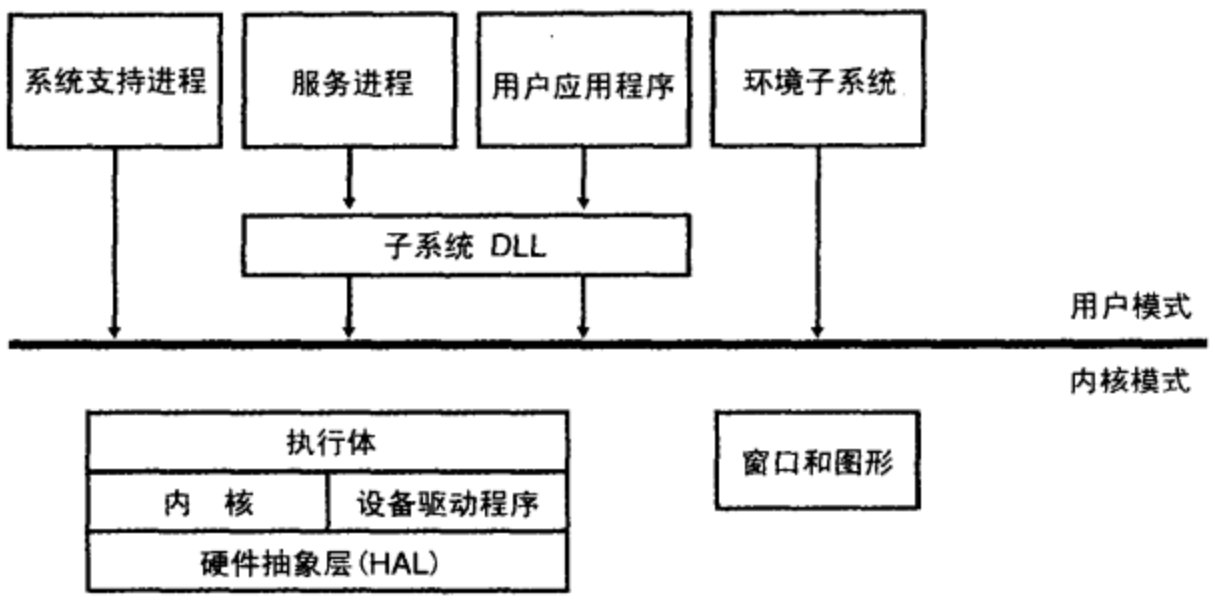


图2.1 简化的Windows结构图

在图2.1中，首先可注意到中间有一条线把Windows操作系统的用户模式（*user mode*）和内核模式（*kernel mode*）两部分划分开来。线上面的方框代表了用户模式的进程，线之下的组件是内核模式的操作系统服务。正如在第1章中曾经提到的，用户模式的线程在一个受保护的进程地址空间中执行（不过，当它们在内核模式中执行的时候，它们可以访问系统空间）。因此，系统支持进程、服务进程、用户应用程序和环境子系统都有它们各自的私有进程地址空间。

四种基本的用户模式进程分别描述如下。

- 固定的（或者硬性指定的）**系统支持进程**（*system support processes*），比如登录（logon）进程和会话管理器（session manager），它们并不是Windows的服务。也就是说，它们不是由服务控制管理器来启动的（第4章将会详细地介绍Windows服务）；
- **服务进程**（*service processes*）宿纳（host）的是Windows服务，比如任务调度器（Task Scheduler）和假脱机服务。Windows服务的运行通常要独立于用户登录。许多Windows服务器应用，比如Microsoft SQL Server和Microsoft Exchange Server，也包含了一些以Windows服务方式来运行的组件；
- **用户应用程序**（*user applications*）有六种类型：Windows 32位、Windows 64位、Windows 3.1 16位、MS-DOS 16位、POSIX 32位或者OS/2 32位；
- **环境子系统服务器进程**（*environment subsystem server processes*）实现了操作系统**环境**（*environment*）的支持部分。这里所谓的环境是指操作系统展示给用户或者程序员的个性化部分。Windows NT最初发布的时候带了三个不同的环境子系统：Windows、POSIX和OS/2。然而，Windows 2000是最后一次带OS/2发布，之后OS/2就被去掉了。到了Windows XP，在基本的产品中只有Windows子系统随产品一起发布——不过，一个增强的POSIX子系统也可以使用，它是针对Unix产品的免费服务的一部分。

在图2.1中，请注意在“服务进程”和“用户应用程序”方框下面的“子系统DLL”方框。在Windows下，用户应用程序并不直接调用原始的Windows操作系统服务，相反，它们通过一

www.TopSage.com

个或者多个**子系统动态链接库**（DLLs）来发起调用。子系统DLL的角色是，将一个已文档化的函数转化为一些恰当的内部（通常是未文档化的）Windows系统服务调用。这个转化过程可能会——也可能不会——向正在为用户应用程序提供服务的环境子系统进程发送一个消息。

Windows的内核模式组件包含以下部分。

- **Windows执行体**（*executive*）包含了基本的操作系统服务，比如内存管理、进程和线程管理、安全性、I/O、网络和跨进程通信。
- **Windows内核**（*kernel*）是由一组低层次的操作系统功能构成的，比如线程调度（thread scheduling）、中断（interrupt）和异常分发（exception dispatching），以及多处理器同步。它也提供了一组例程和基本对象。执行体的其余部分利用这些例程和对象实现更高层次的功能。
- **设备驱动程序**（*device drivers*）既包括硬件设备驱动程序，也包括文件系统和网络驱动程序。其中硬件设备驱动程序将用户的I/O函数调用转换成特定的硬件设备I/O请求。
- **硬件抽象层**（HAL，*Hardware Abstraction Layer*）是指一层特殊的代码，它把内核、设备驱动程序和Windows执行体的其余部分，跟与平台相关的硬件差异（比如不同主板的差异）隔离开来。
- **窗口和图形系统**（*windowing and graphic system*）实现了图形用户界面（GUI）函数（更为人们熟知的叫法是Windows USER和GDI函数），比如对窗口的处理、用户界面控件，以及绘制等。

表2.1列出了Windows操作系统核心组件的文件名。（你有必要知道这些文件名，因为我们将会用名字来引用某些系统文件）。本章后面部分以及后续的章节中将会详细地介绍其中的每一个组件。

表2.1 Windows的核心系统文件

文件名	组 件
Ntoskrnl.exe	执行体和内核
Ntkrnlpa.exe (仅用于32位系统)	执行体和内核，支持物理地址扩展(PAE)，使得系统可寻址多达64GB物理内存
Hal.dll	硬件抽象层
Win32k.sys	Windows子系统的内核模式部分
Ntdll.dll	内部支持函数，以及执行体函数的系统服务分发存根(stub)
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Windows的核心子系统DLL

在探讨这些系统组件的细节之前，我们先来看一看，Windows如何实现跨多种硬件体系结构的可移植性。

www.TopSage.com

可移植性

Windows的一个设计目标是，要能够运行在各种不同的硬件体系结构之上，包括基于Intel的CISC系统以及RISC系统。Windows的初始发行版本支持x86和MIPS体系结构。稍后又加入了对DEC（Digital Equipment Corporation，该公司被Compaq购买了，而Compaq后来又与HP即Hewlett Packard兼并了）Alpha AXP的支持（尽管Alpha AXP是一个64位处理器，但Windows NT运行在32位模式下。在Windows 2000的开发过程中，一个原生的64位版本可以在Alpha AXP上运行，但是这个版本没有被发布）。在Windows 3.51中又加入了对第四种处理器结构Motorola PowerPC的支持。然而，由于市场需求的变化，在Windows 2000开发前夕，对MIPS和PowerPC体系结构的支持被放弃了。后来，Compaq又撤消了对Alpha AXP体系结构的支持，所以最终的结果是，Windows 2000只支持x86体系结构。在最新的Windows版本，即Windows XP和Windows Server 2003中，新增加了对三种64位处理器族的支持，它们分别是Intel Itanium IA-64族、AMD x86-64族，以及Intel针对x86的64位扩展技术（EM64T，它与AMD x86-64体系结构相兼容，但是在所支持的指令方面有略微的差别）。后两种处理器族称为64位扩展系统（*64-bit extended systems*），在本书中引用为*x64*（关于Windows如何在64位版本中运行32位应用程序，第3章将会给出解释）。

Windows主要通过以下两种方法来实现可移植性，以支持多种硬件体系结构和平台。

- Windows有一个分层设计，系统的低层部分是与处理器体系结构相关的，或者是与平台相关的，这些部分被隔离到独立的模块中，所以，系统的高层部分可以不考虑体系结构之间的差别，也不用关心硬件平台上的差异。有两个关键的组件为操作系统提供了可移植性，它们是内核（包含在Ntoskrnl.exe）和硬件抽象层（或称HAL，包含在Hal.dll中）。本章后面将会详细地描述这两个组件。与体系结构相关的功能，比如线程环境切换（thread context switching）和陷阱分发（trap dispatching）是在内核中实现的。在同样的体系结构中，不同系统之间有所差异的功能（例如，不同的主板）则是在HAL中实现的。还有另外一个组件中也有相当数量的代码是与体系结构相关的，那就是内存管理器，但是与整个系统相比，这仍然只是一小部分而已。
- Windows的绝大部分代码是用C语言编写的，少部分是用C++编写的。只有那些需要直接与系统硬件进行通信的部分（比如中断陷阱处理器，interrupt trap handler），或者对性能极端敏感的操作系统部分（比如环境切换，context switching），才是用汇编语言编写的。不仅在内核和HAL中有汇编语言代码，而且在核心操作系统的其他一些地方也有汇编语言代码（比如实现了互锁指令的例程，以及本地过程调用设施中的一个模块），既可能在Windows子系统的内核模式部分，甚至也可能在某些用户模式库中，比如Ntdll.dll中的进

www.TopSage.com

程启动代码（本章后面将会介绍系统库*Ntdll.dll*）。

对称多处理

多任务（*multitasking*）是指在多个执行线程之间共享同一个处理器的操作系统技术。然而，当一台计算机有不止一个处理器的时候，它可以同时执行两个线程。因此，虽然一个多任务操作系统只不过看起来好像同一时刻可以执行多个线程，但是，一个多处理操作系统可以真正地做到同一时刻执行多个线程，在每个处理器上执行一个线程。

正如本章开始时所提到的，Windows的一个关键设计目标是，它必须能够很好地在多处理器计算机系统上运行。Windows是一个**对称多处理**（SMP，*symmetric multiprocessing*）操作系统。在这些处理器中没有主处理器——操作系统和用户线程可以被调度到任何一个处理器上运行。而且，所有的处理器共享惟一的内存空间。这种模型与**非对称多处理**（ASMP，*asymmetric multiprocessing*）不同，在一个典型的非对称多处理系统中，操作系统选择其中一个处理器来执行操作系统内核代码，而其他的处理器只运行用户代码。这两种多处理模型的区别如图2.2所示。

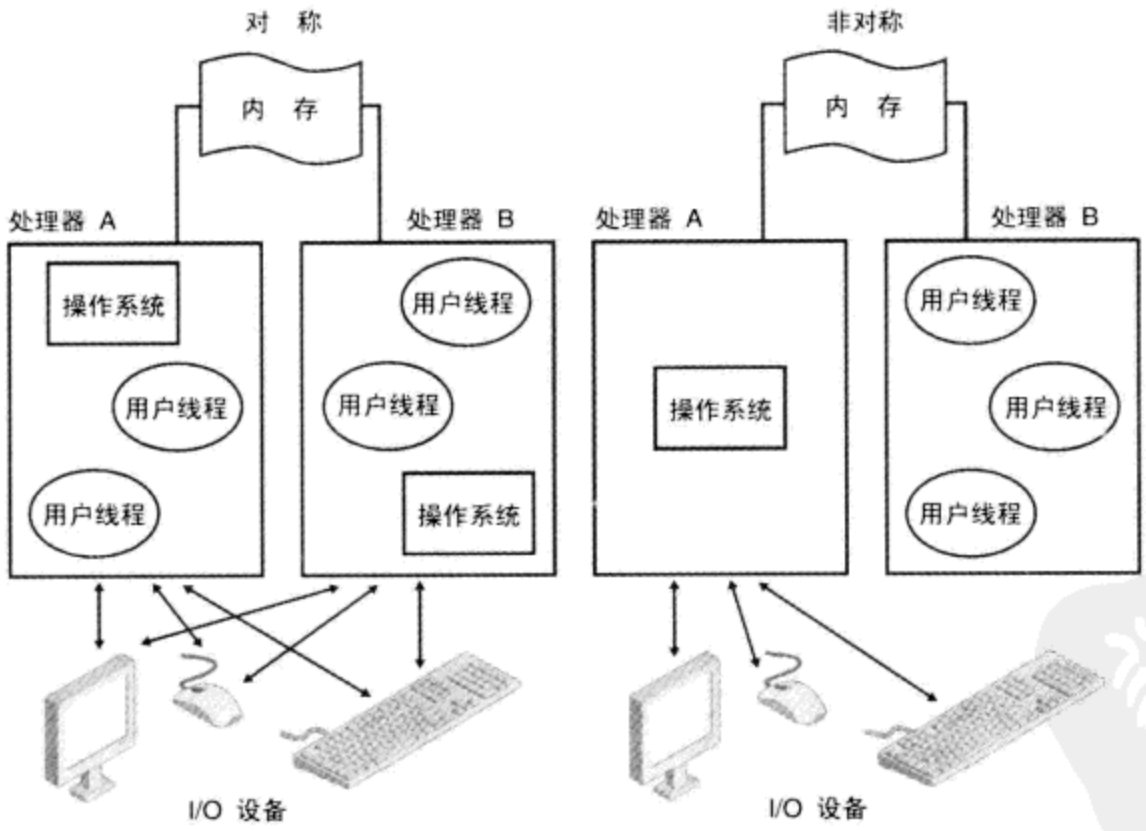


图2.2 对称和非对称多处理

Windows XP和Windows Server 2003支持两种新的多处理器系统：超线程（hyperthreading）和NUMA（非一致的内存结构，non-uniform memory architecture）。在下面的段落中将会简要地介绍这两种系统（有关Windows对这些系统的调度支持的详细描述，请参见第6章中线程调度一节）。

www.TopSage.com

超线程是Intel引入的一项技术，它可以在一个物理处理器上提供多个逻辑处理器。每个逻辑处理器有它自己的CPU状态，但是执行引擎和芯片上的高速缓存则是共享的。这使得一个逻辑CPU可以在其他逻辑CPU正忙着（比如正在执行中断处理工作，从而阻止了在这个逻辑处理器上运行线程）的时候继续执行。Windows XP的调度算法已经被改进过了，它可以最佳地利用多处理器超线程机器，例如，原来的做法是，将线程调度到一个空闲的物理处理器上，现在改进为“选择一个物理处理器上的空闲逻辑处理器（该处理器的其他逻辑处理器可能正在忙着）”。

在非一致的内存结构NUMA系统中，处理器被组织成更小的单元，称为节点（nodes）。每个节点有它自己的处理器和内存，并且通过一个缓存一致（cache-coherent）的互联总线连接到更大的系统上。NUMA系统上的Windows仍然作为一个SMP系统来运行，其中所有的处理器都可以访问所有的内存——只不过，节点本地的内存访问起来比其他节点的内存更快一些而已。系统想要提高性能的做法是，根据线程用到的内存所在的节点，将线程调度到同一节点中的处理器上。这种做法的目标是，尽可能在节点内部满足内存申请的请求，只有在必要的时候才从其他的节点分配内存。

虽然Windows最初的设计只能支持最多32个处理器，但是，在多处理器设计中，并没有本质的因素限制处理器的个数最多为32——很显然，此数字只是为了方便而引入的一个限制，因为32个处理器可以很容易地使用一个原生的32位数据类型作为位掩码来表示。实际上，Windows的64位版本可以最多支持64个处理器，因为在64位机器上，一个字的原生大小是64位。

能支持的处理器的实际数目取决于所用的Windows的版本（参见表2.3和2.4）。该数据被存放在注册表值HKLM\SYSTEM\CurrentControlSet\Control\Session\Manager\ LicensedProcessors中（记住，篡改此数据将违反软件许可规定，而且，要想通过修改注册表来使用更多的处理器，将会涉及更多的工作，而不仅仅修改这个数据）。

考虑性能的原因，内核和HAL（在Windows 2000中，还有其他几个关键的系统文件）有单独的单处理器版本和多处理器版本。在Windows 2000上，六个系统文件（在下面的注记中将会解释）在多处理器系统和单处理器系统上分别有所不同；在32位Windows XP和Windows Server 2003系统上，只有三个系统文件有所不同（见表2.2）。在64位Windows系统上，没有PAE内核，所以，只有内核和HAL在单处理器和多处理器系统上有所不同。

在安装的时候，安装程序将正确的文件选择出来，并且拷贝到本地的\Windows\System32目录中。为了确定哪些文件应该被拷贝，请参见文件\Windows\Repair\Setup.log，它列出了所有要被拷贝到本地系统磁盘的文件，以及将它们从发布介质上拷贝到哪个目录中。

www.TopSage.com

表2.2 针对多处理器的系统文件和针对单处理器的系统文件

在系统磁盘上的文件名	在发布介质上单处理器版本的名称	在发布介质上多处理器版本的名称
Ntoskrnl.exe	Ntoskrnl.exe	Ntkrnlmp.exe
Ntkrnlpa.exe（PAE内核；只有32位系统才有）	Ntkrnlpa.exe in \Windows\<arch>\Driver.cab	Ntkrpamp.exe in \Windows\<arch>\Driver.cab
Hal.dll	取决于系统类型(参见表2.6中的HAL列表)	取决于系统类型（参见表2.6中的HAL列表）
下面的文件只针对Windows 2000		
Win32k.sys	\I386\UNIPROC\Win32k.sys	\I386\Driver.cab中的Win32k.sys
Ntdll.dll	\I386\UNIPROC\Ntdll.dll	\I386\Ntdll.dll
Kernel32.dll	\I386\UNIPROC\Kernel32.dll	\I386\Kernel32.dll



注 如果你查一下Windows 2000发布介质文件树中的\I386\UNIPROC文件夹，你将会看到一个名为Winsrv.dll的文件。虽然此文件被放在名为UNIPROC的文件夹的下面，好像暗示着这是一个单处理器版本，但是，实际上，该文件只有一个版本，可同时适用于多处理器和单处理器系统。在Windows XP和Windows Server 2003中，该文件夹已经被去除了。



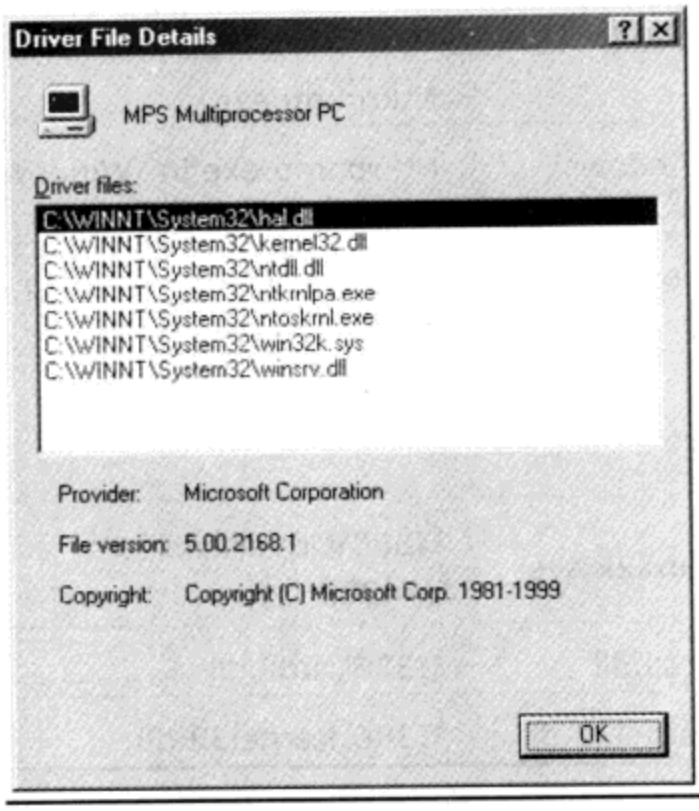
实验：在Windows 2000上查看与多处理器相关的支持文件

你只要在设备管理器（Device Manager）中检查“计算机（Computer）”对象的驱动程序细节，就可以看到，对于一个32位Windows 2000多处理器系统，哪些文件是有所不同的。

1. 打开系统的属性对话框（你可以从控制面板中选择“系统（System）”，或者在你的桌面上右键点击“我的电脑（My Computer）”，然后选择“属性（Properties）”命令)。
2. 点击“硬件（Hardware）”标签。
3. 点击“设备管理器（Device Manager）”。
4. 展开“计算机（Computer）”对象。
5. 双击“计算机（Computer）”下面的子节点。
6. 点击“驱动程序（Driver）”标签。
7. 点击“驱动程序详细信息（Driver Details）”。

www.TopSage.com

对于多处理器系统，你将会看到下面的对话框：



这些关键的系统文件之所以有单处理器版本，是因为性能的原因——相比单处理器系统而言，多处理器的同步本质上是非常复杂和耗时的，所以，让这些关键的系统文件有专门的单处理器版本，就可以做到在单处理器系统上避免这些同步开销（大多数运行Windows的系统都是单处理器的）。

有趣的是，Ntoskrnl的单处理器版本和多处理器版本是通过在源代码中插入条件编译指令来生成的，而Windows 2000中的Ntdll.dll和Kernel32.dll的单处理器版本则是通过修改x86 LOCK和UNLOCK指令来创建的，这两条指令被用于同步多个线程，在单处理器版本中，利用NOP指令（什么也不做）来做同步。

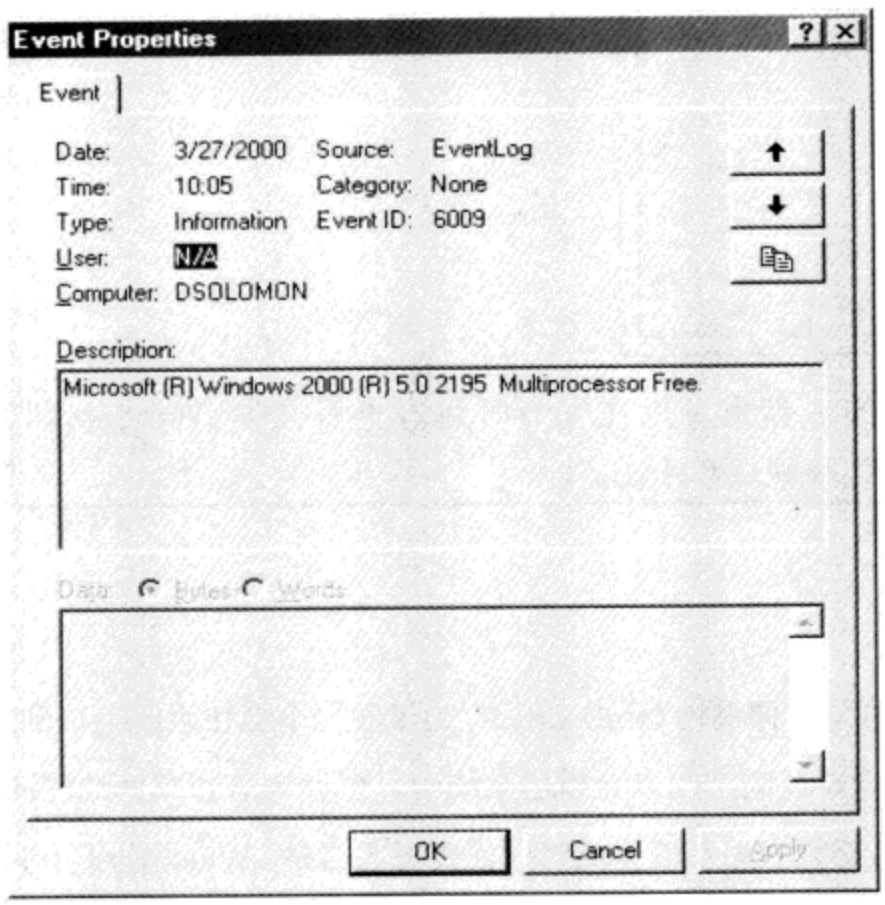
Windows的其他系统文件（包括所有的实用工具、库和设备驱动程序）都使用同样的版本，不再区分单处理器系统和多处理器系统（也就是说，它们可以正确地处理好多处理器的同步问题）。你在你自己的软件中也应该使用这种做法，你的软件无论是一个Windows应用程序还是一个设备驱动程序——当你设计软件的时候总是记住多处理器同步的问题，并且在单处理器系统和多处理器系统上都要做测试。

www.TopSage.com



实验：检查你正在运行的Ntoskrnl的版本

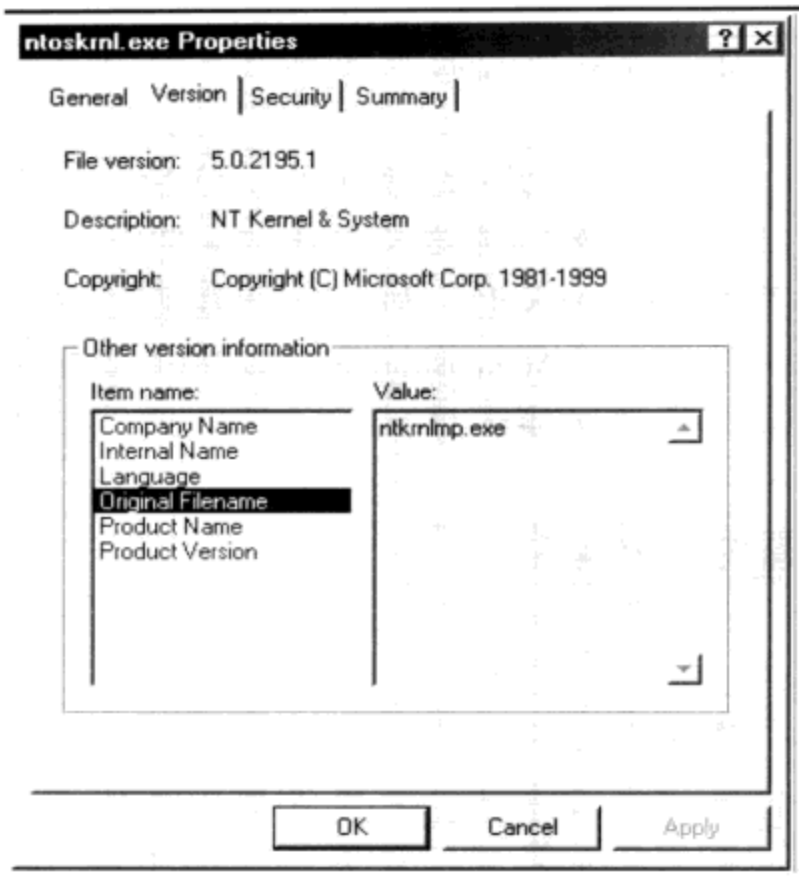
在Windows 2000及其以后的版本中，不再有工具可以显示你正在运行的Ntoskrnl的版本。然而，每次当系统启动的时候，有一条日志信息记录了加载进来的内核映像的类型——单处理器的或多处理器的，自由的（free）或者检查的（checked），如下面的屏幕截图所示。从“开始（Start）”菜单选择“程序（Programs）/管理工具（Administrative Tools）/事件查看器（Event Viewer）”，再选择“系统（System Log）”，双击事件（Event） ID为6009的日志项，此ID表明这一项是系统启动时候写入的。



此日志项并不表明你是否引导了内核映像的PAE版本（Ntkrnlpa.exe，支持大于4GB的物理内存）。然而，你可以通过查看注册表值HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions来确定是否引导了PAE内核。此外，如果你引导了PAE内核，则注册表值HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ PhysicalAddressExtension将被设置为1。

你也可以通过检查文件属性来确定自己是否安装了多处理器版本的Ntoskrnl（或Ntkrnlpa），做法如下：运行Windows资源管理器，到\Windows\System32文件夹下，右键点击Ntoskrnl.exe，然后选择“属性（Properties）”。然后，点击“版本（Version）”标签，选择“源文件名（Original Filename）”属性——如果你正在运行多处理器版本的话，将会看到下面的对话框：

www.TopSage.com



最后，正如前面所提到过的，你也可以在安装时通过查看文件\Windows\Repair\Setup.log来确定到底选择了哪个内核映像和HAL。

可伸缩性

多处理器系统的一个关键问题是**可伸缩性**（*scalability*）。为了在一个SMP系统上正确地运行，操作系统的代码必须遵守严格的指导规则。在多处理器系统中，资源竞争和其他性能问题比在单处理器系统中要复杂得多，而且必须在系统设计的时候就要考虑清楚。Windows集下面几个特性于一身，这些特性对于Windows作为一个成功的多处理器系统起到了以下至关重要的作用：

- 能够在任何一个可用的处理器上运行操作系统代码，也可以同时在多个处理器上运行系统代码；
- 在单个进程内执行多个线程，这些线程可以在不同的处理器上并行地执行；
- 内核内部（比如自旋锁、排队的自旋锁以及压栈锁，这些将在第3章详述）以及设备驱动程序和服务器进程内部的细粒度同步，这使得多个组件可以并行地在多个处理器上运行。
- 诸如I/O完成端口（I/O completion port，在第9章中介绍）之类的编程机制，使得可以实现高效的多线程服务器进程，并且这样的程序在多处理器系统上有很好的伸缩性。

www.TopSage.com

Windows内核的伸缩性一直在不停地进步。比如，Windows Server 2003使用了针对每个CPU的调度队列，这使得可以在多个CPU上并行地执行线程调度。第6章讨论了多处理器线程调度的细节。有关多处理器同步的进一步细节可以在第3章中找到。

客户和服务器版本之间的差异

Windows既发行客户版本的零售软件包，也发行服务器版本的零售软件包。在Windows 2000中，客户版本被称为Windows 2000 Professional；Windows 2000的3个服务器版本分别是：Windows 2000 Server、Advanced Server和Datacenter Server。

Windows XP有6个客户版本：Windows XP Home Edition、Windows XP Professional、Windows XP Starter Edition、Windows XP Tablet PC Edition、Windows XP Media Center Edition和Windows XP Embedded。后3个版本是Windows XP Professional的超集，本书并不详细地讨论这3个版本，因为它们建立在与Windows XP Professional相同的核心操作系统之上。

Windows Server 2003有6个变种：Windows Server 2003 Web Edition、Standard Edition、Small Business Server、Storage Server、Enterprise Edition和Datacenter Edition。

这些版本在以下方面有所不同：

- 所支持的处理器的数目不同；
- 所支持的物理内存的数量不同；
- 所支持的并行网络连接的数量（例如，在客户版本中，最多允许同时连接10个文件和打印服务）不同；
- Server版本随带的分层服务（layered services）并不包含在Professional版本中，例如目录服务（directory services）、集群服务（clustering），以及对多用户终端服务（multiuser Terminal Services）的支持。

表2.3摘要列举了Windows 2000的4个版本在内存和处理器支持方面的不同。表2.4列举了Windows XP和Windows Server 2003S之间同样的信息。关于Windows Server 2003不同版本之间的详细比较，请参考*http://www.microsoft.com/windowsserver2003/evaluation/features/compareeditions.mspx*。

表2.3 Windows 2000 Professional和Server几个版本之间的差异

版 本	支持的处理器的数目	支持的物理内存
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

www.TopSage.com

表2.4 Windows XP和Windows Server 2003之间的差异

	支持的处理器的数目（32位版本）	支持的物理内存（32位版本）	支持的处理器的数目（64位版本）	支持的物理内存（Itanium版本）	支持的物理内存（x64版本）
Windows XP Home Edition	1	4 GB	不可用	不可用	不可用
Windows XP Professional	2	4 GB	2	不可用	128 GB
Windows Server 2003 Web Edition	2	2 GB	不可用	不可用	不可用
Windows Server 2003 Small Business Server	2	2 GB	不可用	不可用	不可用
Windows Server 2003 Standard Edition	4	4 GB	不可用	不可用	不可用
Windows Server 2003 Enterprise Edition	8	32 GB	8	64 GB	64 GB
Windows Server 2003 Datacenter Edition	32	64 GB	64	1024 GB	1024GB

虽然Windows操作系统有多个客户版本和服务器版本的零售软件包，但是它们共享同一组核心系统文件，包括内核映像Ntoskrnl.exe（以及PAE版本Ntkrnlpa.exe）、HAL库、设备驱动程序，以及基本的系统辅助工具和DLL。这些文件在Windows 2000的所有版本中都是相同的。



注 Windows XP是Windows NT代码基（code base）的第一个客户版本，并且没有对应的服务器版本。不过，在Windows XP发行之后的一年多时间里，针对服务器版本的开发工作仍然在进行，这就是后来的Windows Server 2003。因此，Windows XP和Windows Server 2003中的核心系统文件是不同的。即便如此，两者的差异并不显著（实际上，有许多组件并没有改变）。

所以，如果Windows 2000 Professional和Windows 2000 Server的内核映像完全相同的话（针对Windows XP和Windows Server 2003，情况也类似），那么系统如何知道引导了哪个版本呢？只需查询注册表的HKLM\ SYSTEM\CurrentControlSet\Control\ProductOptions键下的ProductType和ProductSuite两个值。ProductType可用来区分当前的系统是一个客户系统还是一个服务器系统，它的有效取值如表2.5所示。此结果值存放在系统全局变量*MmProductType*中，在

www.TopSage.com

设备驱动程序中可以利用内核模式的支持函数*MmIsThisAnNtAsSystem*查询到此值。关于此函数，请参见Windows DDK的文档。

表2.5 ProductType注册表值

Windows版本	ProductType的值
Windows 2000 Professional、Windows XP Professional、Windows XP Home Edition	WinNT
Windows Server（域控制器）	LanmanNT
Windows Server（仅服务器）	ServerNT

另一个注册表值ProductSuite，既可以区别Windows服务器系统的不同版本（Stardard、Enterprise、Datacenter，等等），还可以区分Windows XP Home和Windows XP Professional系统。

如果用户程序需要判断当前运行的是哪个版本的Windows系统，他们可以调用Windows的*VerifyVersionInfo*函数。Platform SDK中有关于该函数的文档。设备驱动程序可以调用内核模式下的函数*RtGetVersion*。Windows DDK中有关于此函数的文档。

既然系统的核心文件在客户版本和服务器版本中本质上是相同的，那么，系统在运行过程中又是如何有所不同呢？简而言之，在默认配置下，服务器系统针对系统吞吐量做了优化，使之可以成为高性能应用服务器；而客户版本呢，虽然它也有服务器的能力，但是它针对交互式桌面用途下的响应时间做了优化。例如，在系统引导时候，根据产品类型的不同，有几项资源分配的决策是不同的，比如操作系统堆（heap）或者池（pool）的大小、内部的系统辅助线程（system worker threads）的数量，以及系统数据高速缓存的大小。而且，运行时刻的策略——比如内存管理器用何种方式来平衡系统和进程的内存需求——在服务器版本和客户版本中也是不同的。甚至线程调度中的有些细节在这两大产品族中也有不同的默认行为，比如时间片的默认长度，或者线程**时限**（*quantum*）（详见第6章）。凡是在两大产品族中有重要运行差异的地方，在本书接下来的有关章节中都会加以强调。除非另有特殊说明，否则，本书中介绍的内容可同时适用于客户版本和服务器版本。

检查版本

Windows 2000 Professional、Windows XP Professional和Windows Server 2003都有一个特殊的调试版本，被称为检查版本（*checked build*，只有MSDN Professional或更高的订阅级别才可以获得）。这是Windows源代码重新编译得到的一个版本，编译时定义了一个称为“DBG”的标志，置了该标志以后，用于调试和跟踪的编译时刻条件代码也被包含进来。而且，为了更加易于理解生成得到的机器代码，针对Windows二进制代码的后处理优化（为了执行得更快而进行的代码布局结构优化）并没有被执行（参见Debugging Tools帮助文件中的“Performance-Optimized Code”部分）。

www.TopSage.com

之所以提供这样的检查版本，主要是为了帮助设备驱动开发人员，因为它可以针对那些被设备驱动程序或其他系统代码所调用的内核模式函数执行更为严格的错误检查。例如，如果驱动程序（或者其他的内核模式代码）对一个系统函数执行了一个无效调用（比如，如果在错误的中断级别上获取一个自旋锁即spinlock），并且该系统函数会检查传递进来的参数，那么，当问题被检测到以后系统就会停止，而不是允许某些数据结构被破坏，从而导致过一段时间以后系统可能会崩溃。



实验：确认一下你是否正在运行Windows的检查版本

没有一个内置的工具可以显示出你当前正在运行的系统是检查版本还是零售版本（也称为自由版本，free build）。然而，通过WMI（Windows Management Instrumentation，Windows管理规范）的Win32_OperatingSystem类的“Debug”属性，你可以获得这一信息。下面简单的Visual Basic脚本可以显示这一属性：

```
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
```

为了测试这段代码，请输入上述代码，并将它保存到一个文件中。下面是运行这段脚本的输出：

```
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
```

该系统并没有在运行Windows的检查版本，这里显示的Debug标志为False。

在检查版本的二进制代码中，大多数的附加代码都是使用了ASSERT宏之后的结果，该宏被定义在DDK头文件Ntddk.h中，DDK文档中有关于此宏的介绍。ASSERT宏测试一个条件（比如一个数据结构或者参数的有效性），如果表达式的计算结果是FALSE，那么该宏调用内核模式函数*RtlAssert*，该函数又进一步调用*DbgPrint*，将调试消息的文本发送到一个调试消息缓冲区中。如果有一个内核调试器被附载到当前系统中，则该消息被自动显示在一个提示符的后面，请用户来决定该如何处理这一断言失败（断点、忽略、终止进程，或者终止线程）。

www.TopSage.com

如果当前系统不是与内核调试器一起引导起来的（在Boot.ini中使用/DEBUG开关），并且也没有附载内核调试器，那么，ASSERT测试的失败将会导致系统崩溃。关于一些内核支持例程所执行的各种ASSERT检查，请参见Windows DDK文档中的“Checked Build ASSERTs”部分。



注 在检查版本中，如果你比较一下Ntoskrnl.exe和Ntkrnlmp.exe，或者比较一下Ntkrnlpa.exe和Ntkrpamp.exe，你就会发现它们是完全相同的——它们都是相同文件的多处理器版本。换句话说，Windows的检查版本并没有提供内核映像的单处理器调试版本。

检查版本对于系统管理员也是非常有用的，因为他们可以针对特定的组件跟踪到进一步的细节信息（有关详细的指令，参见Microsoft Knowledge Base中编号为314743的文章，题目为“*HOWTO: Enable Verbose Debug Tracing in Various Drivers and Subsystems*”）。这些信息被通过*DbgPrint*发送到内部的调试消息缓冲区，前面已经提到过了。为了查看这些调试消息，你或者将某个内核调试器附载到目标系统上（这要求在调试模式下引导目标系统），并且在执行本地内核调试的过程中使用*dbgprint*命令；或者利用来自*www.sysinternals.com*的Dbgview.exe工具。

为了利用操作系统的调试版本，你不必安装整个检查版本。你可以只拷贝内核映像（Ntoskrnl.exe）和正确的HAL（Hal.dll）的检查版本到一个普通的零售版本系统中。这种做法的好处是，设备驱动程序和其他的内核代码受到了检查版本的严格检查，但是又不必运行其他所有组件的较慢的调试版本。如何做到这一点呢？详细的指令请参见Windows DDK文档中的“Installing Just the Checked Operating System and HAL”部分。因为Microsoft并没有提供Windows 2000 Server的检查版本，所以，你也可以利用这项技术在Windows 2000 Server系统上运行检查版本的内核。

最后，检查版本对于测试那些仅仅因为系统的定时（timing）而有所不同的用户模式代码也是非常有用的（这是因为，附加的检查只是在内核中进行的，并且内核的组件都是未经编译优化的）。通常，多线程同步错误往往跟特定的定时条件有关。如果你在检查版本的系统上（或者至少内核和HAL是检查版本的）运行测试程序，那么由于整个系统的定时是不同的，所以这有可能使那些在普通的零售系统中不会发生的潜在定时错误暴露出来。

2.4 关键的系统组件

我们已经看过了Windows的高层体系结构，现在我们来深入地研究Windows的内部结构，以及每个关键的操作系统组件所扮演的角色。图2.3是一个关于Windows系统核心结构和组件的图，它比本章前面给出的结构图（见图2.1）更加详细和完整。请注意，它仍然没有显示所有的组件，特别是没有显示网络组件（第13章将会介绍网络组件）。

www.TopSage.com

这一节将详细阐述图2.3中的每一个主要元素。第3章解释Windows使用的主要控制机制（比如对象管理器、中断，等等）。第5章讲述启动和关闭Windows的过程，第4章详细介绍各种管理机制，比如注册表、服务进程，以及WMI（Windows Management Instrumentation，Windows管理规范）。然后，剩下的章节更加细节地讨论各个关键区域的内部结构和操作，比如进程和线程、内存管理、安全性、I/O管理器、存储管理、高速缓存管理器、Windows文件系统（NTFS）和网络。

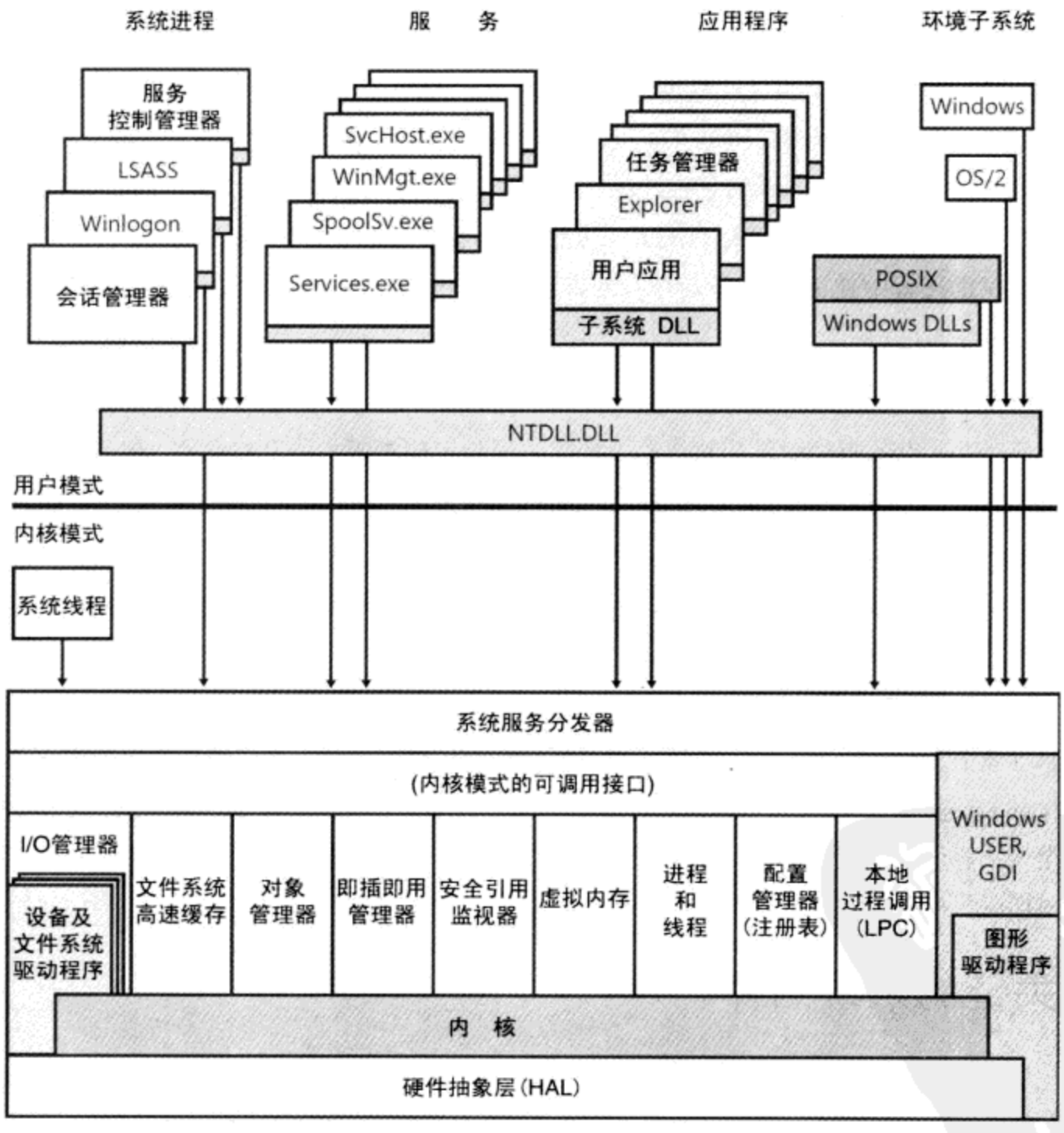


图2.3 Windows系统核心结构和组件

www.TopSage.com

环境子系统和子系统DLL

正如图2.3所示，Windows最初有三个环境子系统：OS/2、POSIX和Windows。如前所述，OS/2子系统最后一次是随Windows 2000发布。到了Windows XP以后，原来随Windows一起发行的基本POSIX子系统已经不再随系统一起发行了，尽管如此，一个增强的POSIX子系统版本作为免费提供给UNIX产品的服务的一部分，可供用户使用。

正如稍后将要解释的那样，三个子系统中的Windows子系统是非常特殊的，Windows非它不能运行（它拥有键盘、鼠标和显示器，而且即使在没有交互用户登录的服务器系统上，它也是必须存在的）。实际上，其他两个子系统被配置成按需启动，而Windows子系统则必须总是在运行。

子系统启动信息被保存在注册表键HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems的下面。图2.4显示了该键下面的值。

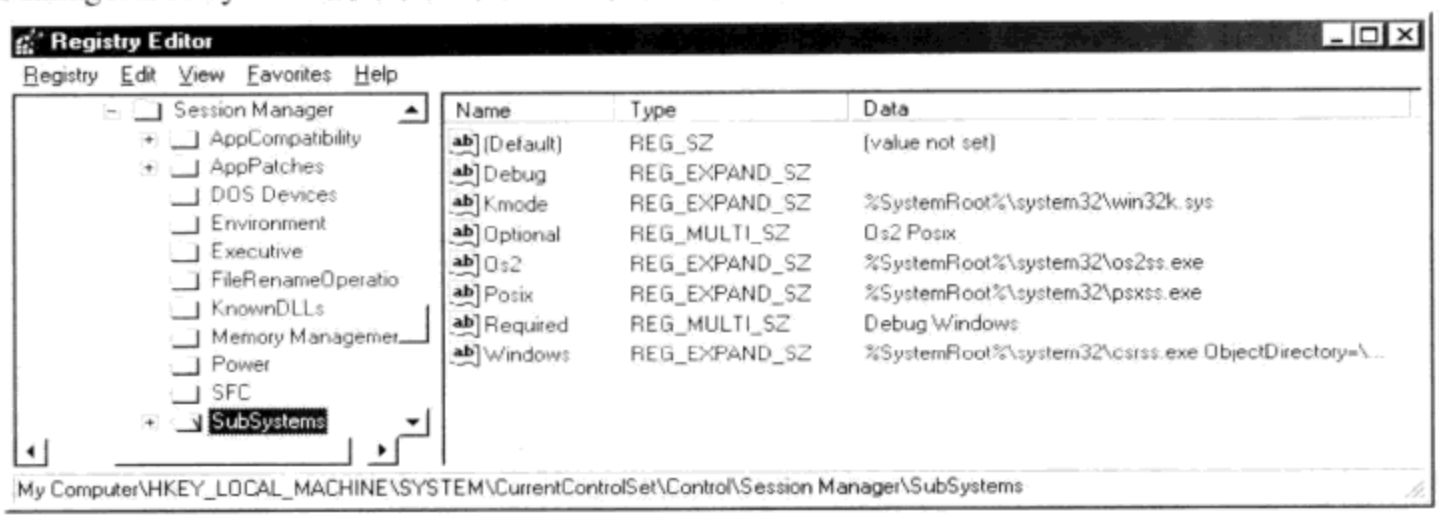


图2.4 注册表编辑器显示了Windows子系统启动信息

Required值列出了当系统引导时加载的子系统。该值有两个字符串：Windows和Debug。Windows值包含了Windows子系统的文件规范，Csrss.exe，它代表了客户/服务器运行时子系统（Client/Server Run-Time Subsystem）（参见本节后面的注）。 Debug值是空的（因为它被用于内部测试），因此什么也不做。Optional值表明了OS/2和POSIX子系统将被按需启动。注册表值Kmode包含了Windows子系统的内核模式部分的文件名Win32k.sys（本章后面将会进一步解释）。

环境子系统的角色是，将Windows基本系统服务的部分子集暴露给应用程序。每个子系统都提供了对于Windows原生服务不同部分的访问能力。这意味着，建立在某一个子系统上的应用程序可以做到的事情，是另一个建立在不同子系统上的应用所无法做到的。例如，一个Windows应用程序不能够使用POSIX的*fork*函数。

每一个可执行的映像（.exe）被绑定到一个（且是惟一的）子系统上。当一个映像文件被运行时，创建进程的代码会检查映像头部的子系统类型代码，所以它可以通知正确的子系统，有新的进程被创建了。在Visual C++中，*link*命令的/SUBSYSTEM修饰符可以指定此类型代码，利用Windows资源工具箱中的Exetype工具可以查看此类型代码。

www.TopSage.com



注 这是一个有关Windows历史的注释，Windows子系统进程之所以被称为Csrss.exe，是因为在Windows NT的原始设计中，所有的子系统都位于全局惟一的环境子系统进程中，作为该进程中的线程来执行的。在POSIX和OS/2子系统被移出来并放到它们自己的进程中以后，Windows子系统进程的文件名没有改变。

不同子系统之间，函数调用不能混合。换句话说，一个POSIX应用只能调用POSIX子系统导出的服务，Windows应用只能调用Windows子系统导出的服务。正如稍后你将要看到的那样，这一限制正是为什么“最初的POSIX子系统（它实现了一个非常有限的函数集合，仅仅POSIX 1003.1）并非是一个移植UNIX应用程序的有用环境”的原因之一。

正如前面所提及的，用户应用程序并不直接调用Windows的系统服务。相反，它们通过一个或者多个子系统DLL来进行。这些库导出的接口都有很好的文档说明，凡是链接到这些子系统的程序都可以调用这些接口。例如，Windows子系统DLL（比如Kernel32.dll、Advapi32.dll、User32.dll和Gdi32.dll）实现了Windows API函数。POSIX子系统DLL（Psxdll.dll）实现了POSIX API函数。



实验：查看可执行映像的子系统类型

你可以利用Windows资源工具箱中的Exetype工具，或者Windows支持工具箱和Platform SDK中的相依性查找工具（Depends.exe），来看到映像的子系统类型，例如两个不同的Windows映像：Notepad.exe（简单的文本编辑器，即记事本）和Cmd.exe（Windows命令提示环境），其子系统类型如下：

```
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
```

这表明Notepad是GUI程序，而Cmd则是控制台或者基于字符的程序。而且，Exetype工具的输出暗示了：虽然针对GUI和基于字符的程序有两个不同的子系统，但是，Windows子系统却只有一个。而且，在Intel 386处理器（就此而言，或者486）上是不支持Windows的——Exetype程序的文本输出尚未被更新。

www.TopSage.com

当一个应用程序调用子系统DLL中的某个函数的时候，可能会发生下述三件事情之一。

- 该函数完全是在该子系统DLL中实现的，在用户模式下运行。换句话说，该函数并没有给环境子系统进程发送消息，而且也没有调用Windows执行体的系统服务。该函数是在用户模式下完成的，运行的结果被返回给调用者。此类函数的例子有*GetCurrentProcess*（它总是返回-1，在所有与进程相关的函数中，利用该值可以引用当前进程）和*GetCurrentProcessId*（对于一个正在运行的进程，进程ID不会改变，所以进程ID可以从某个缓存的地方获取到，从而无须每次调用至内核中）。
- 该函数要求调用Windows执行体一次或者多次。例如，Windows的*ReadFile*和*WriteFile*函数分别要调用底层的（且无文档的）内部Windows I/O系统服务*NtReadFile*和*NtWriteFile*。
- 该函数要求在环境子系统进程中完成某些工作（环境子系统进程运行在用户模式下，它负责维护那些在其控制下运行的客户应用程序的状态）。在这种情况下，该函数以消息的形式给环境子系统发送一个客户/服务器请求，从而让环境子系统执行某个操作。然后子系统DLL等待应答，收到应答之后再返回给调用者。

有些函数可能是以上列出的第2和第3项的组合，比如Windows的*CreateProcess*和*CreateThread*函数。

尽管Windows在设计的时候，要求支持多个独立的环境子系统，但是从实践的角度来看，让每个子系统都实现所有的代码来处理窗口和显示I/O，显然会导致大量重复的系统函数，无论是对系统的空间大小，还是对系统的性能，最终必然带来负面影响。因为Windows是原生的子系统，所以Windows的设计者决定将这些基本的函数放在Windows子系统中，并且让其他的子系统调用Windows子系统来完成显示I/O。因此，POSIX和OS/2子系统调用Windows子系统中的服务来完成显示I/O（实际上，如果你检查这些映像的子系统类型，你将会看到，它们都是Windows可执行文件）。

下面我们来仔细地看一看每一个环境子系统。

Windows子系统

Windows子系统是由以下几个主要的组件构成的。

- 环境子系统进程（Csrss.exe）包含下列支持：
  - 控制台（文本）窗口；
  - 创建或删除进程和线程；
  - 对16位虚拟DOS机（VDM）进程的一部分支持；

www.TopSage.com

- ❑ 其他一些函数，比如*GetTempFile*、*DefineDosDevice*、*ExitWindowsEx*，以及几个自然语言支持函数。

- 内核模式设备驱动程序（Win32k.sys）包含：
  - ❑ 窗口管理器（window manager），它控制窗口显示，管理屏幕输出，采集来自键盘、鼠标和其他设备的输入，同时也负责将用户的消息传递给应用程序；
  - ❑ 图形设备接口（GDI，Graphics Device Interface），它是专门针对图形输出设备的函数库，其中包括线、文本和图形的绘制函数，以及绘图控制函数。
- 子系统DLL（比如Kernel32.dll、Advapi32.dll、User32.dll和Gdi32.dll），它们将已经文档化的Windows API函数，翻译成Ntoskrnl.exe和Win32k.sys中恰当的且绝大多数未文档化的内核模式系统服务调用。
- 图形设备驱动程序是指与硬件相关的图形显示器驱动程序、打印机驱动程序和视频微端口驱动程序。

应用程序调用标准的USER函数以便在显示器上创建用户界面控件，比如窗口和按钮。窗口管理器将这些请求发送给GDI，而GDI又将它们传递给图形设备驱动程序，在设备驱动程序内部，这些请求被经过格式化以适合特定的显示器设备。显示器驱动程序与视频微端口驱动程序是成对的，以实现对视频显示的支持。

GDI提供了一组标准的二维函数，使得应用程序无须知道任何有关图形设备的信息就可以与设备进行通信。GDI函数位于应用程序（比如显示器驱动程序和打印机驱动程序）和图形设备的中间。GDI负责解释应用程序的图形输出请求，并且将这些请求发送给图形显示器驱动程序。它也为应用程序提供了标准的接口，以便这些应用程序可以使用各种不同的图形输出设备。此接口使得应用程序代码可以独立于硬件设备和它们的驱动程序。GDI对它的消息进行剪裁，以符合设备的能力，它通常要将应用程序的请求分解成多个易于处理的部分。例如，有些设备可以理解画椭圆的命令，而其他的设备则要求GDI将该命令解释成“将一系列像素放置在特定的坐标处”。有关图形和视频驱动程序结构的更多信息，请参考Windows DDK中*Graphics Drivers*一书的“Design Guide”一节。

在Windows NT 4以前，窗口管理器和图形服务是用户模式Windows子系统进程的一部分。在Windows NT 4中，这些原本运行在Windows子系统进程环境中的窗口和图形代码被移到了一组可调用的系统服务中，直接运行在内核模式下（在文件Win32k.sys中）。

www.TopSage.com

之所以做这样的迁移，主要的理由是为了提高总体系统性能。虽然原始的设计已经被高度优化了，但是，把Windows图形子系统放在一个分开的服务器进程中，必将导致多次线程和进程环境的切换，从而消耗掉相当数量的CPU周期和内存资源。

例如，对于客户方的每个线程，在Windows子系统进程中都有一个专门的、配对的服务器线程在等待客户线程的请求。一个被称为*fast LPC*的特殊的跨进程通信设施，可用来在这些线程之间发送消息。与普通的线程环境切换不同的是，在这些配对的线程之间通过fast LPC来传递消息并不会使内核中产生一个重新调度（rescheduling）的事件，这样，服务器线程在等待内核的抢先式线程调度器将执行权交给它以前，可以利用客户线程尚未用完的时间片来运行自己的代码。利用共享的内存缓冲区，可以传递一些大的数据结构，比如位图；客户可以直接（但只读）访问关键的服务器数据结构，以便使客户和Windows子系统服务器之间的线程/进程转换尽可能地减少到最少。并且，GDI操作可以分批处理（现在也是如此）。所谓批处理（*batching*），意味着Windows应用程序发出的一系列图形调用先被放到一个GDI批处理队列中，直到一个队列填满了才被“推（pushed）”到服务器并且在输出设备上绘制出来。你可以使用Windows的*GdiSetBatchLimit*函数来设置队列的大小，而且在任何时候都可以通过*GdiFlush*函数刷空队列。相反地，GDI的只读属性和数据结构，一旦从Windows子系统进程中获得，就被缓存在客户端，以便后续的访问更加快速。

然而，尽管有了这些优化，对于图形密集的应用程序，总体的系统性能仍然是不够的。显而易见的解决方案是，通过将窗口和图形系统转移到内核模式中，就可以去除那些附加的线程，以及由此带来的环境切换。而且，一旦应用程序调用进入了窗口管理器和GDI中，这些子系统就可以直接访问其他的Windows执行体组件，而无须付出用户模式或者内核模式转换的代价。若一个进程通过GDI调用视频驱动程序，并且以很高的频率和带宽与视频硬件进行交互，则在此情况下，这样的直接访问尤为重要。

那么，有哪些还保留在Windows子系统的用户模式进程部分呢？控制台或者文本窗口的所有绘制和更新操作都是在这一部分中处理的，因为控制台应用程序并没有重绘一个窗口的概念。下面这样的行为很容易看得到：打开一个命令提示窗口，然后拽着另一个窗口在它上面拖动，这时你将会看到，当Windows子系统重绘控制台窗口的时候它正在消耗CPU时间。但是，除了对控制台窗口的支持以外，其他只有少数几个Windows函数会导致给Windows子系统进程发送消息：进程和线程的创建和终止、网络驱动器字母映射，以及临时文件的创建。一般而言，一个正在运行的Windows应用程序不会引起很多（如果有的话）至Windows子系统进程的环境切换。

www.TopSage.com

将USER和GDI放在内核模式中，Windows会更加不稳定吗

有些人可能想知道，将这些代码放到内核模式中是否会严重影响系统的稳定性。其实，对系统稳定性的影响已经减到了最低限度，理由是，在Windows NT 4之前（这在今天仍然是正确的），用户模式Windows子系统进程（Csrss.exe）中的一个错误（比如访问违例，access violation）会导致系统崩溃，因为Windows子系统进程对于系统的运行来说，是（现在仍然是）一个至关重要的进程。因为那些描述显示器上当前窗口信息的数据结构正是位于此进程之中，所以，一旦该进程死了，则整个用户界面就被毁了。然而，即使是一个作为服务器运行的Windows操作系统，它没有任何交互进程，若没有这个进程，则系统也就无法运行，因为服务器进程可能要利用窗口消息机制来驱动应用程序的内部状态。对于Windows来说，同样这部分代码现在运行在内核模式中，代码中的访问违例（access violation）只会使系统崩溃得更快一些而已，因为内核模式中的异常会导致系统崩溃。

然而，有个额外的理论危险则是在窗口和图形系统被移到内核模式以前并不存在的。由于这部分代码现在运行在内核模式下，所以，任何错误（bug，比如使用了坏的指针）都可能会破坏掉内核模式保护的数据结构。在Windows NT 4以前，这样的错误指针引用会引发一个访问违例，因为内核模式页面在用户模式下是不能写的。然后导致系统崩溃，如前所述。现在这部分代码运行在内核模式下，若一个坏指针引用导致了对某个内核模式页面的一次写操作，那么，它有可能并不立刻使系统崩溃；相反，如果它破坏了某个数据结构，则有可能在不久之后导致系统崩溃。然而，这里也存在一种小的可能性，也就是，这样的坏指针引用将破坏内存缓冲区（而不是数据结构），可能的结果是，把破坏了的数据返回给用户程序，或者把坏数据写到了磁盘上。

将图形驱动程序移到了内核模式中以后，还有另外一种可能的影响。先前，图形驱动程序的一部分代码在Csrss中运行，其他部分在内核模式下运行。现在，整个驱动程序都运行在内核模式下。尽管Windows中支持的图形设备驱动程序并不全部是Microsoft开发的，但是Microsoft直接与硬件制造商合作，以努力确保这些厂商能够制造出可靠和有效的驱动程序。随系统一起发行的所有驱动程序如同其他的执行体组件一样，都要经过同一套严格的测试过程。

最后，理解下面的一点也是很重要的：从根本上讲，这样的设计（指在内核模式下运行窗口和图形子系统）并不冒险。它等同于许多其他设备驱动程序所使用的方法（例如，网卡驱动程序和硬盘驱动程序）。从Windows有了高度可靠性的需求之初，所有这些驱动程序就一直在内核模式下运行。

有些人曾经推测，将窗口管理器和GDI移到内核模式中，必将伤害到Windows的抢先式多任务调度能力。他们的理论是，由于所有附加的Windows处理时间都是花在内核模式中的，所以，其他线程被抢先运行的机会就更少了。

www.TopSage.com

这种观点是建立在一种对Windows体系结构的误解的基础之上的。在其他许多有名无实的抢先式操作系统中，这确实是真的，在内核模式中的执行过程永远不会被操作系统调度器抢占掉——或者说，只能被少量预定义的内核重入点（points of kernel reentrancy）抢占掉。然而，在Windows中，在执行体（executive）任何地方运行的线程，都与用户模式下运行的线程一起被抢占和调度，执行体内部的所有代码都是完全可重入的（reentrant）。还有其他的理由，为了在SMP硬件上达到高度的系统扩展性，这种能力是必要的。

另一种推测是，SMP的扩展性将会因为这一改变而受到影响。其理论大致如下：原先的时候，应用程序和窗口管理器或者GDI之间的交互会涉及两个线程，一个在应用程序中，另一个在Csrss.exe中，因此，在SMP系统上，两个线程可能会并行运行，从而提高系统的吞吐量。这种分析说明，他们误解了Windows NT技术在Windows NT 4以前的版本中是如何工作的。在大多数情况下，从客户应用程序到Windows子系统进程的调用是同步运行的，也就是说，客户线程在等待服务器线程的时候是完全阻塞的，只有当服务器线程完成了该调用时客户线程才开始继续运行。因此，在SMP硬件上根本做不到并行性。这种现象很容易观察到，你只需在一个SMP系统上用性能工具来查看一个繁忙的图形应用程序即可。你将会发现，在一个双处理器的系统上，每个处理器都是将近50%的负担，而且，很容易可以发现，一个Csrss线程与繁忙的应用线程结成了一对呼应的线程。实际上，因为这两个线程相互之间关系很紧密，而且共享了状态，所以，处理器的高速缓存必须要经常被刷新，以维护两者的一致性。正是因为这种经常性的刷新动作，所以在Windows NT 3.51系统中，单线程图形应用程序在SMP机器上运行起来比在单处理器系统上往往要略慢一些。

因此，Windows NT 4中的这一变化使那些大量使用窗口管理器和GDI的应用程序在SMP机器上的吞吐量大大提高，特别是当多个应用线程在忙的时候。当两个应用线程在一台具有双处理器的Windows NT 3.51机器上忙于运行的时候，总共四个线程（应用程序中有两个，再加上Csrss中两个）在竞争这两个处理器的时间。尽管在任何时候往往只有两个线程准备运行，但是由于缺乏一致的模式（即这些线程的运行模式），从而导致丢失了内存引用的局部性，以及高速缓存的一致性。之所以会丢失这种局部性和一致性，是因为繁忙的线程更有可能从一个处理器转移到另一个处理器上。在Windows NT 4的设计中，这两个应用线程的每一个都有它自己的处理器，Windows的自动线程亲和性总是倾向于在同样的处理器上运行同一个线程，因而使内存引用的局部性尽可能地最大化，并且尽可能地同步减少每个处理器私有高速缓存的需求。

所以，简而言之，将窗口管理器和GDI从用户模式移到内核模式，不仅提高了性能，而且在系统稳定性和可靠性方面没有任何显著的退化，即使是在一个配置了终端服务的系统中创建了多个会话的情况下仍是如此。

www.TopSage.com

POSIX子系统

POSIX差不多可以看成“一个基于UNIX的可移植的操作系统接口（a Portable Operating System Interface based on UNIX）”的首字母缩写，它指的是针对UNIX风格的操作系统接口的一组国际标准。POSIX标准鼓励厂商们实现UNIX风格的接口，从而使它们保持兼容，这样程序员们就可以很容易地将他们的应用程序从一个系统转移到另一个系统上。

Windows仅仅实现了众多POSIX标准中的一个标准，即POSIX.1，正式的名称是ISO/IEC 9945—1:1990或者IEEE POSIX standard 1003.1—1990。该标准之所以被包含进来，主要是为了满足美国政府在20世纪80年代中后期制定的政府采购要求，即POSIX.1兼容性是强制性的，这是在美国标准和技术委员会（National Institute of Standards and Technology）开发的FIPS（Federal Information Processing Standard，联邦信息处理标准）151—2中规定的。Windows NT 3.5、3.51和4已经被正式测试过，并且通过了FIPS 151—2的鉴定。

因为POSIX.1兼容性是Windows的一个强制性目标，所以，在设计Windows操作系统时，必须要保证，所设计的基本系统能够实现POSIX.1子系统——比如*fork*函数是在Windows执行体中实现的，而针对文件硬链接（hard file links）的支持则是在Windows文件系统中的。然而，因为POSIX.1定义了很有限的一组服务（比如进程控制、跨进程通信、简单的字符单元I/O即simple character cell I/O，等等），所以，随Windows 2000带的POSIX子系统并不是一个完整的编程环境。而且，因为应用程序不能在不同的Windows子系统之间进行混合调用，所以默认情况下，POSIX应用程序严格受限于POSIX.1中定义的各种服务。这一限制意味着，Windows上的POSIX可执行程序无法创建线程，也无法创建窗口，或者使用远过程调用（RPC）或套接字（sockets）。

为了解决这一限制带来的不方便，Microsoft提供了一个称为“Windows Services for Unix”的产品，它（从3.5版本开始）包含了一个增强的POSIX子系统环境，该环境提供了将近2000个UNIX函数，以及差不多300个UNIX类的实用工具（有关更多关于Windows Services for Unix的信息，请参考*http://www.microsoft.com/windows/sfu/default.asp*）。

这一增强的POSIX子系统有助于将UNIX应用程序移植到Windows平台上。然而，因为这些程序仍然被链接为POSIX可执行程序，所以它们不能调用Windows函数。为了将UNIX应用移植到Windows中，并且允许使用Windows函数，你可以购买UNIX-to-Windows移植软件包，比如来自Mortice Kern System Inc公司（*www.mkssoftware.com*）的MKS工具箱产品。通过这种方法，UNIX应用程序可以被重新编译和链接成一个Windows可执行程序，然后再慢慢地加入一些对Windows原生函数的调用。

www.TopSage.com



实验：观察POSIX子系统的启动

在默认情况下，POSIX子系统被配置成：当第一次有一个POSIX可执行程序被运行的时候，该子系统被启动起来。所以你只需运行一个POSIX程序就可以观察到它被启动起来，比如说，你可以运行某一个随“Windows Services for Unix”带的实用工具（在Windows 2000资源工具箱的介质上，你可以在\Apps\POSIX文件夹中找到一小部分POSIX实用工具。注意，这些工具并不会被作为资源工具箱的一部分而安装到系统中）。按照下面的步骤可以观察到POSIX子系统被启动起来。

1. 启动一个命令提示窗口。
2. 运行 Process Explorer（进程管理器），检查一下POSIX子系统当前并没有在运行（也就是说，在系统中并没有Psxss.exe进程）。确保Process Explorer以树状视图的形式来显示进程列表（按下Ctrl+T即可）。
3. 运行一个POSIX程序，比如随“Windows Services for Unix”带的C Shell或者Korn Shell（或者Windows 2000资源工具箱中的POSIX工具，比如\Apps\POSIX\Ls.exe）。
4. 回到Process Explorer中，注意到新的Psxss.exe进程是Smss.exe的一个子进程（根据你的机器上高亮显示间隔的长短不同，该进程可能作为一个新进程而仍然是高亮显示的）。

要想在Windows中编译和链接一个POSIX应用程序，要求使用Platform SDK中的POSIX头文件和库文件。POSIX可执行文件需要链接POSIX子系统库Psxdll.dll。因为在默认情况下，Windows被配置成按需启动POSIX子系统，所以当你第一次运行一个POSIX应用程序的时候，POSIX子系统进程（即Psxss.exe）必须被启动起来。在系统重新引导以前，它会一直保持运行（如果你杀掉了POSIX子系统进程，那么你就不能再运行其他的POSIX应用程序了，除非你重新引导系统）。POSIX映像文件本身并不直接运行。相反，一个特殊的称为Posix.exe的支持映像文件被启动起来，然后它再创建一个子进程来运行POSIX应用程序。

OS/2子系统

如同内置的POSIX子系统一样，OS/2环境子系统在有用性方面也非常有限，它仅支持OS/2 1.2的16位基于字符的或者视频I/O（VIO）的应用程序。虽然Microsoft也确实出售过一个替代OS/2 1.2的Windows NT 4展示管理器子系统，但它并不支持OS/2 2.x（或更以后版本的）应用程序（而且在Windows 2000及以后，它也不再可用了）。

另外，因为Windows不允许用户应用程序直接访问硬件，所以，如果OS/2程序中包含了企图执行IN/OUT指令（以便访问某些硬件设备）和高级视频I/O（AVIO）的I/O特权代码，则这样的程序在Windows平台上无法支持。

www.TopSage.com

而那些使用了CLI/STI指令的应用程序则是支持的，但是系统中所有其他的OS/2应用程序，以及发出了CLI指令的进程中所有其他的线程，都被挂起，直到执行了一条STI指令为止。

在原生的OS/2 1.2上有16MB的内存限制，但这种限制并不适用于Windows —— OS/2子系统利用Windows的32位虚拟地址空间，可以向OS/2 1.2应用程序提供多达512MB内存，如图2.5所示。

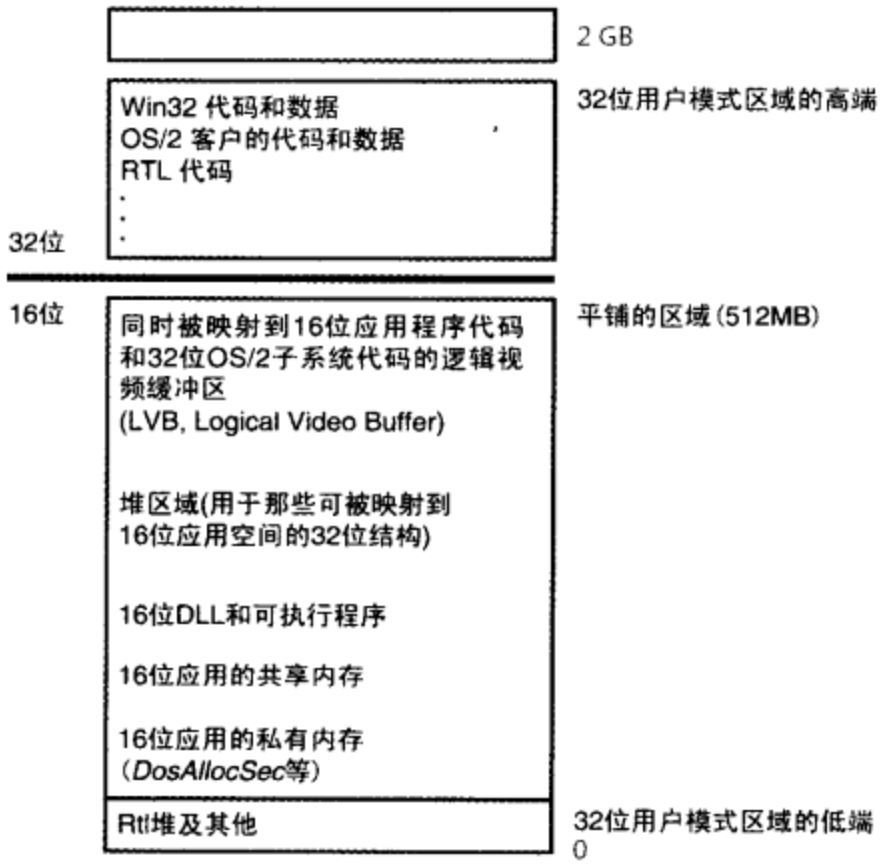


图2.5 OS/2子系统虚拟内存布局结构

平铺的区域（tiled area）是指虚拟地址空间中预先被保留，然后当16位应用程序需要段的时候被提交或解除提交的512MB内存。OS/2子系统为每个进程维护了一个局部描述符表(LDT，Local Descriptor Table），而共享的内存段则是在所有OS/2进程的同样LDT条目中。

正如我们将要在第6章中讨论的那样，线程（thread）是一个程序的执行元素，所以它们必须能被调度，以获得处理器时间。虽然Windows优先级的范围从0到31，但是64个OS/2优先级（0到63）仅仅被映射到Windows动态优先级1至15。OS/2线程永远也不会接收Windows实时优先级16至31。

如同POSIX子系统一样，OS/2子系统也是在你第一次激活一个兼容的OS/2映像文件时才自动启动起来。它会一直保持运行状态，直到系统被重新引导为止。

有关Windows如何处理正在运行的POSIX和OS/2应用程序的更多信息，请参见第6章的“*CreateProcess*的流程”一节。

www.TopSage.com

Ntdll.dll

Ntdll.dll是一个特殊的系统支持库，主要用于子系统DLL。它包含两种类型的函数：

- 系统服务分发存根（stubs），它们会调用Windows执行体系统服务；
- 内部支持函数，供子系统、子系统DLL以及其他的原生映像文件使用。

第一组函数为Windows执行体系统服务提供了接口，在用户模式下可以通过这些接口函数调用Windows执行体的系统服务。这样的函数超过了200个之多，比如*NtCreateFile*、*NtSetEvent*等等。正如前面所说明的，这些函数的大多数功能可以通过Windows API来访问得到（然而，有些函数则不然，它们被用于操作系统内部）。

对于每一个这样的函数，Ntdll包含了一个同名的入口点。函数内部的代码包含了与处理器体系结构相关的模式切换指令，通过该指令可转换到内核模式下，从而调用系统服务分发器（system service dispatcher，第3章将会详细地介绍）。分发器在检验了某些参数以后，再调用真正的内核模式系统服务，其中包括Ntoskrnl.exe内部的实际代码。

Ntdll也包含许多支持函数，比如映像加载器（以*Ldr*开头的函数）、堆管理器、Windows子系统进程通信函数（以*Csr*开头的函数），以及一般的运行库过程（以*Rtl*开头的函数）。它也包含了用户模式下的异步过程调用（APC，Asynchronous Procedure Call）分发器和异常分发器（关于APC和异常，将在第3章中解释）。

执行体

Windows执行体是Ntoskrnl.exe中的上层（内核是其下层）。执行体包含以下类型的函数。

- 可在用户模式下调用的导出函数。这些函数称为**系统服务**（*system services*），并且通过Ntdll被导出。这些服务绝大多数可通过Windows API来访问，或者通过另一个环境子系统的API来访问。然而对于有些服务，通过任何一个文档化的子系统函数都无法访问它们（这样的例子有：LPC、诸如*NtQueryInformationProcess*这样的查询函数，以及诸如*NtCreatePagingFile*这样的专用函数，等等）。
- 可通过*DeviceIoControl*函数来调用的设备驱动器函数。这为从用户模式到内核模式提供了一个通用的接口，因而在用户模式下可以调用设备驱动程序中并不与读或者写操作关联的函数。
- 只能在内核模式下调用的导出函数，并且这些函数在Windows DDK或者Windows IFS Kit（Installable File System Kit，可安装文件系统的开发工具箱）中已经文档化（有关Windows IFS Kit的更多信息，请浏览*http://www.microsoft.com/whdc/ddk/ifskit/default.mspx*）。

www.TopSage.com

- 在内核模式下调用，但未在Windows DDK或者IFS Kit中文档化的导出函数（比如以*Inbv*开头的、在引导视频驱动程序中调用的函数）。
- 定义为全局符号但是未被导出的函数。它们包括在Ntoskrnl内调用的内部支持函数，比如以*Iop*或者*Mi*开头的那些函数（分别是内部I/O管理器支持函数和内部内存管理支持函数）。
- 未定义为全局符号，而是在一个模块内部的函数。

执行体包含了以下的主要组件（在本书后续的章节中将会详细地介绍这些组件）。

- **配置管理器**（*configuration manager*，在第4章中介绍）负责系统注册表的实现和管理。
- **进程和线程管理器**（*process and thread manager*，在第6章中介绍）创建或终止进程和线程。针对进程和线程的底层支持是在Windows的内核中实现的；而执行体则在这些底层对象的基础上又加上了附加的语义和功能。
- **安全引用监视器**（*security reference manager*或称SRM，在第8章中讲述）强制在本地计算机上实行安全策略。它守护着操作系统的资源，执行对运行时对象的保护和审计。
- **I/O管理器**（*I/O manager*，在第9章中解释）实现了与设备无关的I/O操作，负责将这些操作分派到恰当的设备驱动程序以供进一步处理。
- **即插即用（PnP）管理器**（*Plug and Play manager*，在第9章中解释）的任务是，为了支持一个特定的设备，确定哪些驱动程序是必需的，同时它也负责加载这些驱动程序。它在设备列举过程中，获取到每个设备的硬件资源需求。PnP管理器根据每个设备的资源需求，分配适当的硬件资源，比如I/O端口、IRQ、DMA通道和内存位置。当系统中的设备变化（增加或者移除设备）时，它也负责发送恰当的事件通知。
- **电源管理器**（*power manager*，在第9章中解释）负责协调电源事件，并且向设备驱动程序产生电源管理I/O通知。电源管理器可以配置成：当系统空闲的时候，通过将CPU置于睡眠状态从而降低电源消耗。单独设备的电源消耗变化可以由设备驱动程序来处理，但是需要电源管理器来协调。
- **WDM Windows管理规范例程**（*WDM Windows Management Instrumentation routines*，在第4章中解释）允许设备驱动程序可以发布有关性能和配置的信息，以及接收来自用户模式WMI服务的命令。WMI信息的消费客户可以在本地机器上，也可以在跨越网络的远程机器上。

www.TopSage.com

- **高速缓存管理器**（*cache manager*，在第11章中解释）提高了以文件为基础的I/O操作的性能，其做法是，让最近引用过的磁盘数据驻留在主内存中以便快速访问（并且延迟磁盘写操作，在将更新数据发送到磁盘之前先在内存中停留一小段时间）。你将会看到，它利用内存管理器的映射文件的支持来做到这一点。
- **内存管理器**（*memory manager*，在第7章中解释）实现了虚拟内存。这是一种内存管理方案，为每个进程提供了一个巨大的私有地址空间，其数量可以大大超过当前可用的物理内存。内存管理器也为高速缓存管理器提供最基本的底层支持。
- **逻辑预取器**（*logical prefetcher*，在第7章中解释）加速系统和进程的启动过程。对于要在系统或进程启动过程中引用的数据，它优化了这些数据的加载过程。

另外，Windows执行体还包含四组主要的支持函数，以上这些执行体组件也会用到这些支持函数。在这些支持函数中，差不多有三分之一可以在DDK中找到相应的文档，因为设备驱动程序也要用到它们。以下是四大类支持函数。

- **对象管理器**（*object manager*），它创建、管理和删除Windows执行体对象和抽象数据类型，这些对象和数据类型往往代表了操作系统的资源，比如进程、线程和各种同步对象。本书第3章介绍了对象管理器。
- **LPC设施**（在第3章中解释），它在同一台机器上的客户进程和服务器进程之间传递消息。LPC是RPC（*Remote Procedure Call*，远过程调用）的一个更加灵活的优化版本。这里的RPC是指跨网络的客户进程和服务器进程之间的通信设施工业标准。
- 一组涉及范围广泛的**公共运行库**函数，比如字符串处理、算术操作、数据类型转换，以及安全结构处理等。
- **执行体支持例程**，比如系统内存分配（换页的和非换页的内存池）、互锁的内存访问，以及两种特殊类型的同步对象：资源和快速互斥体（fast mutexes）。

内核

内核是由Ntoskrnl.exe中的一组函数以及对于硬件体系结构的低层支持（比如中断和异常分发）构成的。Ntoskrnl.exe中的这组函数提供了一些最为基本的机制，比如线程调度和同步服务，供执行体组件使用；而对硬件的低层支持则随着每个处理器结构的不同而有所区别。内核代码主要是用C编写的，对于那些要用到特殊的处理器指令和寄存器，并且不容易在C代码中访问的任务，则保留使用汇编代码的形式。

www.TopSage.com

如同上一部分中提到的各种执行体支持函数一样，内核中的许多函数也在DDK中有详细的文档描述（通过搜索以*Ke*打头的函数，可以找到它们）。因为在实现设备驱动程序的时候也需要用到它们。

内核对象

内核提供了一组定义明确的、可预知的操作系统低层原语和机制，从而使得执行体中的高层组件可以做它们需要做的事情。内核实现了操作系统的基本机制，并且避免了各种策略决定，从而将自己与执行体的其余部分分离开。它几乎将所有的策略决定都留给了执行体，惟一的例外是线程调度和分发，这是由内核自己来实现的。

从内核外部来看，执行体将线程和其他可共享的资源都表示为对象。这些对象要求一些策略开销，比如用以维护它们的对象句柄（object handle），以及用以保护它们的各种安全检查，还有相应的资源配额（当它们被创建的时候资源配额就会被扣除）。这些开销在内核中是不存在的，内核实现了一组更为简单的对象，称为**内核对象**（*kernel objects*），它们帮助内核控制好中心处理过程，并且支持执行体对象的创建工作。执行体层的绝大多数对象包装了一个或者多个内核对象，把它们的内核属性合并起来。

一组称为**控制对象**（*control objects*）的内核对象建立了有关控制各种操作系统功能的语义。这包括APC对象、DPC（*Deferred Procedure Call*，**延迟过程调用**）对象，以及I/O管理器使用的一些对象，比如中断对象等。

另一组称为**分发器对象**（*dispatcher objects*）的内核对象融合了同步的能力，它们改变或者影响了线程的调度。分发器对象包括内核线程、互斥体（内部称为**突变体**即*mutant*）、事件、内核事件对（pair）、信号量（semaphore）、定时器，以及可等待的定时器（waitable timer）。执行体利用内核函数来创建内核对象的实例，并维护这些对象实例，同时构建更加复杂的、提供给用户模式的对象。第3章中将更加详细地介绍对象，第6章讲述进程和线程。

硬件支持

内核的另一个主要任务是将执行体和设备驱动程序从Windows所支持的各种硬件体系结构中抽象出来，或者隔离出这些变种之间的差异。这项任务包括要处理各种功能（比如中断处理、异常分发和多处理器同步）方面的变化情况。

即便是对这些与硬件相关的功能，在设计内核的时候也企图使公共代码尽可能地最大化。内核支持一组可移植的接口，这组接口的语义在不同的体系结构上是等同的。而且，实现这组可移植接口的大部分代码在不同的体系结构上也是等同的。

然而，在这组接口中，有些在不同的体系结构上有不同的实现；或是在有些接口的实现中，部分代码与体系结构相关。

www.TopSage.com

这些与体系结构独立的接口可以在任何一台机器上被调用，而且无论实现代码是否随体系结构的不同而不同，接口的语义总是相同的。有些内核接口（比如自旋锁例程，将在第3章中讲述）实际上是在HAL（在下一部分讲述）中实现的，因为它们的实现即使在同一体系结构族的系统中也可能有所不同。

内核中有一小部分代码涉及与x86有关的接口，之所以需要这些代码是为了支持老的MS-DOS程序。这些x86接口并不是可移植的，因为在任何其他体系结构的机器上它们是不可能被调用的；而且它们也不会出现在这样的机器上。例如，与x86相关的代码提供了相应的功能来维护全局描述符表（GDT）和LDT，这些正是x86的硬件特性。

在内核中与体系结构相关的代码的另一个例子，是提供“翻译缓冲区（translation buffer）”和“CPU高速缓存”支持的接口。为了提供这样的支持，不同的体系结构需要不同的代码，因为高速缓存的实现方式各有不同。

另一个例子是环境切换（context switching）。尽管从高层来看，线程选择和环境切换可以使用同样的算法（上一个线程的执行环境被保存起来，新线程的环境被加载进来），但是在不同处理器上的实现之间，还是存在一些体系结构方面的差异。因为执行环境是由处理器的状态（寄存器等）来描述的，所以哪些信息应该被保存或加载，则是随着体系结构的不同而有所不同的。

硬件抽象层(HAL)

正如本章开始时所提到的，Windows设计的关键要素之一是，它能被移植到各种不同的硬件平台上。硬件抽象层是使得这种移植性成为可能的一个关键部分。HAL是一个可加载的、内核模式的模块，它提供了针对Windows当前运行所在的硬件平台的低层接口。它隐藏了与硬件相关的细节，比如I/O接口、中断控制器，以及多处理器通信机制——任何与体系结构相关或者与机器相关的功能。

所以，Windows内部组件以及用户编写的设备驱动程序并不直接访问硬件；相反，当它们需要获得与平台相关的信息时，它们可以通过调用HAL例程来保持可移植性。出于这一原因，针对这些HAL例程，也有相应的文档包含在Windows DDK中。要想找到更多有关HAL及其在设备驱动程序中用法的信息，请参考DDK。

虽然Windows随带了几个HAL（见表2.6），但是在安装的时候只有一个HAL被选中，并且被拷贝到系统磁盘上，其文件名为Hal.dll（其他的操作系统，比如VMS，在系统引导的时候选择一个等价于HAL的模块）。因此，如果支持两个不同处理器的HAL有所不同的话，你就不能这样假设：一个x86已安装系统的系统磁盘也可以在另一个处理器上引导。

www.TopSage.com

表2.6 在\Windows\Driver Cache\i386\Driver.cab中的x86 HAL列表

HAL文件名	所支持的系统
Hal.dll	标准PC
Halacpi.dll	高级配置和电源接口(ACPI) PC
Halapic.dll	高级可编程中断控制器(APIC) PC
Halaacpi.dll	APIC ACPI PC
Halmps.dll	多处理器PC
Halmacpi.dll	多处理器ACPI PC
Halborg.dll	Silicon图形工作站(仅仅在Windows 2000中才有，市场上已经不再销售)
Halsp.dll	Compaq SystemPro (仅仅在Windows XP中才有)



注 从Windows Server 2003开始，在Windows基本系统中不再随带与特定厂商相关的HAL。



实验：查看Windows中包含的基本HAL

要想检查Windows中包含的HAL，请打开\Windows\Driver Cache下面某个体系结构文件夹中的Driver.cab文件。（例如，对于x86系统来说，该文件名是\Windows\Driver Cache\i386\Driver.cab。）滚动到以Hal为开头的文件处，你应该可以看到表2.6中列出的那些文件。



实验：确定你当前正在运行哪个HAL

有以下两种方法可以确定你当前正在运行哪个HAL。

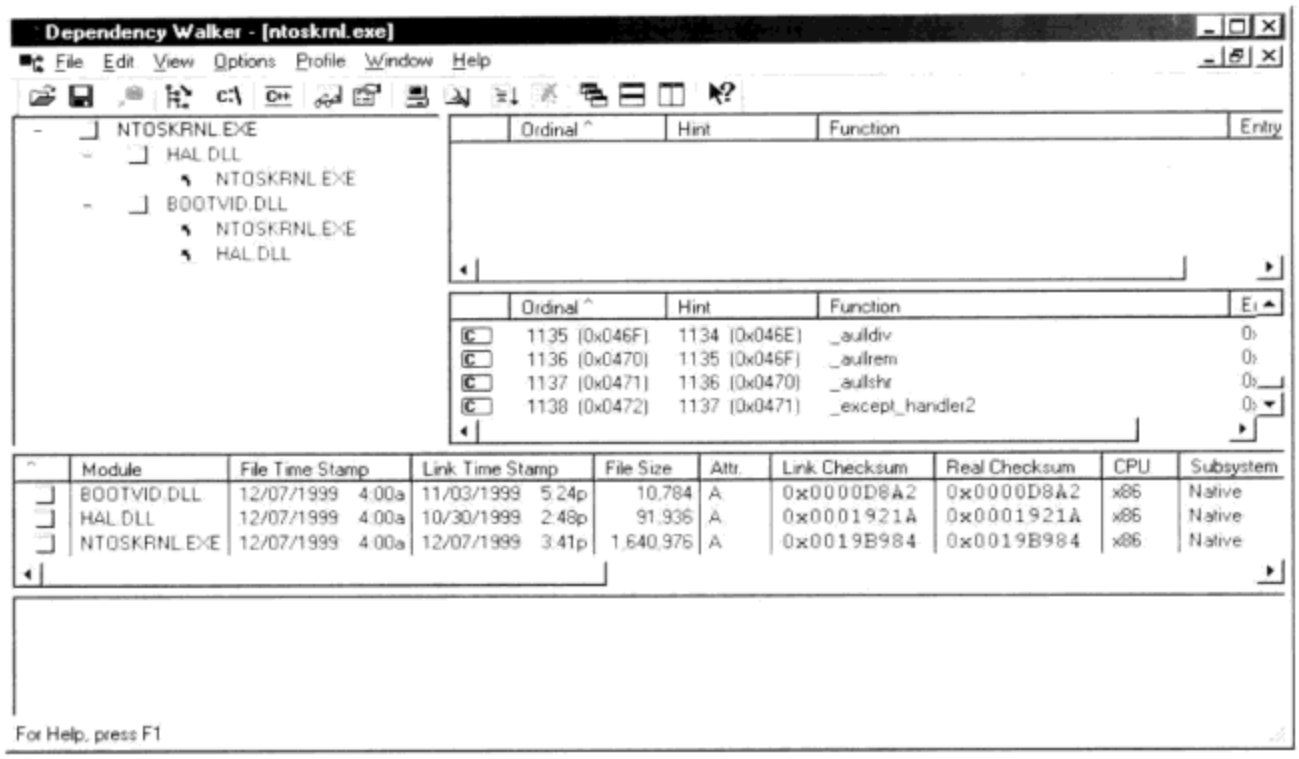
1. 打开文件\Windows\Repair\Setup.log，搜索Hal.dll，检查等号后面的文件名。该文件名是指在安装介质上从Driver.cab中提取出来的HAL的名字。
2. 在设备管理器中（在你的桌面上右键点击“我的电脑（My Computer）”），选择“属性（Properties）”，然后点击“硬件（Hardware）”标签，再点击“设备管理器（Device Manager）”，看一看在“计算机（Computer）”设备类型下面的“*driver*”的名字。例如，下面的屏幕拷贝是从某一个运行ACPI HAL的系统中截取下来的：

www.TopSage.com

实验：查看NTOSKRNL和HAL的映像相依性

你可以利用相依性检查工具（Dependency Walker tool，Depends.exe）来检查内核和HAL映像文件的导出和导入表，从而了解它们之间的关系。至于此相依性检查工具，你可以从Windows支持工具箱或者Platform SDK中找到。要想在该工具中检查一个映像文件，你可以从File菜单中选择Open命令，来打开目标映像文件。

如果你使用该工具来查看Ntoskrnl的相依性，下图是你可能看到的一个输出例子：



注意，Ntoskrnl链接了HAL，而HAL又链接了Ntoskrnl（它们相互使用了对方的函数）。Ntoskrnl也链接了Bootvid.dll，这是用来实现GUI启动屏幕的引导视频驱动程序。在Windows XP及其以后，你在列表中还会看到一个附加的DLL：Kdcom.dll。该DLL包含了内核调试器的基础设施代码，在过去，这些代码是Ntoskrnl.exe的一部分。

有关相依性检查工具的详细描述信息，请参考Dependency Walker的帮助文件（Depends.hlp）。

设备驱动程序

虽然要到第9章才介绍设备驱动程序，不过，这一部分先大致介绍一下驱动程序的类型，并且解释一下如何列出你的系统中已经安装和加载的驱动程序。

www.TopSage.com

设备驱动程序是可加载的内核模式模块（通常以.sys结尾），它们在I/O管理器和相应的硬件之间建立起链接。设备驱动程序运行在内核模式下，位于以下三种执行环境之一：

- 在发起I/O功能的用户线程的环境中；
- 在内核模式系统线程的环境中；
- 作为一个中断的结果（因此它不在任何特定的进程或者线程的执行环境中——当该中断产生时，无论当前进程或者线程是哪个）。

正如上一部分所说的，Windows中的设备驱动程序并不直接维护硬件，相反，它们调用HAL中的函数与硬件打交道。驱动程序往往是用C（有时候用C++）来编写的，因此，通过正确地使用HAL例程，驱动程序可以在Windows所支持的CPU体系结构之间进行源代码级的移植，而在同一个体系结构族内则是二进制可移植的。

设备驱动程序有以下几种类型。

- **硬件设备驱动程序**（*hardware device drivers*）通过HAL操纵硬件，从而将输出写到物理设备或网络中，或者从物理设备或网络上接收输入。硬件设备驱动程序也有许多类型，比如总线驱动程序、人机界面驱动程序、大容量存储器驱动程序，等等。
- **文件系统驱动程序**（*file system drivers*）是指这样的Windows驱动程序：可以接受面向文件的I/O请求，并且将这些请求转换成针对某一特定设备的I/O请求。
- **文件系统过滤器驱动程序**（*file system filter drivers*），比如那些执行磁盘镜像和加密的驱动程序，或那些截取了I/O请求并且执行某些增值处理之后再传递给下一层的驱动程序。
- **网络重定向器和服务器**（*network redirectors and servers*），分别是指那些将文件系统I/O请求传递给网络上某一台机器，或者从网络上接收此类请求的文件系统驱动程序。
- **协议驱动程序**（*protocol drivers*）实现了诸如TCP/IP、NetBEUI和IPX/SPX之类的网络协议。
- **内核流式过滤器驱动程序**（*kernel streaming filter drivers*），这样的驱动程序被串接起来，以便对数据流进行信号处理，比如录制或者播放视音频。

要想在系统中添加用户编写的内核模式代码，安装驱动程序是惟一的方法，所以有些程序员把编写设备驱动程序当做是一种访问操作系统内部函数和数据结构的简便方法（在用户模式下是不可访问的，但是在DDK中有文档以及相应的支持）。例如，来自*www.sysinternals.com*的许多工具是由一个Windows GUI应用程序和一个驱动程序组成的，其中驱动程序用来收集内部的系统状态，以及调用那些只可在内核模式下访问（从用户模式中无法访问）的Windows API。

www.TopSage.com

Windows驱动程序模型

Windows 2000增加了对即插即用、电源选项的支持，同时也扩展了Windows NT的驱动程序模型，新的模型称为WDM（Windows Driver Model，Windows驱动程序模型）。Windows 2000及以后的版本都可以运行老的Windows NT 4驱动程序，但是因为这些驱动程序不支持即插即用和电源选项，所以运行这些驱动程序的系统在这两方面的能力都会有所退化。

从WDM的角度来看，有以下三种驱动程序。

- **总线型驱动程序**（*bus driver*），它为总线控制器、适配器、桥或任何带有子设备的设备提供服务。总线驱动程序是必需的驱动程序，通常Microsoft会提供此类驱动程序；系统中的每一种总线类型（比如PCI、PCMCIA和USB）都有一个总线驱动程序。第三方也可以编写总线驱动程序来为新的总线提供支持，比如VMEbus、Multibus和Futurebus。
- **功能型驱动程序**（*function driver*）是主要的设备驱动程序，它们为相应的设备提供了可操作的接口。功能型驱动程序也是必需的，除非相应的设备可以直接使用（其I/O可通过总线驱动程序和总线过滤型驱动程序来完成，比如SCSI PassThru）。根据定义，功能型驱动程序最了解某一特定的设备，而且它往往是惟一能访问与该设备相关的寄存器的驱动程序。
- **过滤型驱动程序**（*filter driver*）用来为某一设备（或已有的驱动程序）增加新的功能，或者修改来自其他设备的I/O请求或应答（也常常用来修补那些未能提供正确的硬件资源需求信息的硬件设备）。过滤型驱动程序是可选的，可以有任意数目，可以放在一个功能型驱动程序之上或之下，也可以放在总线型驱动程序之上。通常，系统原始设备制造商（OEM）或者独立硬件销售商（IHV）会提供过滤型驱动程序。

在WDM驱动程序环境中，对于一个设备而言，并不是由单个驱动程序来控制它的所有方面：一个总线型驱动程序负责向PnP管理器报告其总线上的设备，而一个功能型驱动程序操纵该设备。

在大多数情况下，低层次的过滤型驱动程序改变设备硬件的行为。例如，若一个设备向其总线型驱动程序报告它需要4个I/O端口，但它实际却需要16个I/O端口，那么一个与该设备相关的低层次功能过滤型驱动程序可以截获总线型驱动程序向PnP管理器报告的硬件资源列表，并更新一下I/O端口的数目。

上层的过滤型驱动程序通常为一个设备提供一些增值特性，比如一个针对键盘的上层过滤型驱动程序可以强制加上安全检查。

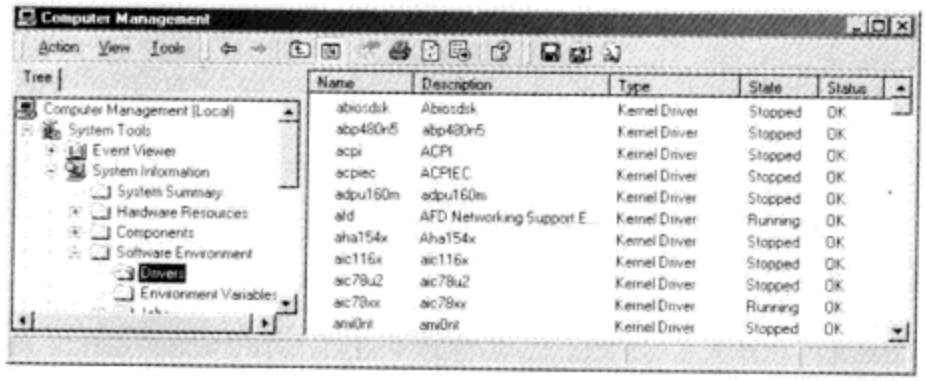
关于中断处理将在第3章中解释。有关I/O管理器、WDM、即插即用和电源选项的详细介绍，请参考第9章。

www.TopSage.com



实验：查看已安装的设备驱动程序

通过运行Msinfo32程序[1]，你就可以列出当前系统中已安装的驱动程序：在“开始（Start）”菜单中，选择“运行（Run）”，然后输入“Msinfo32”并回车，打开“系统信息”程序，从树图中展开“软件环境”，选择“系统驱动程序[2]”。下面是一个例子，其中显示了当前已安装的驱动程序的列表：



该窗口显示了注册表中定义的设备驱动程序列表、它们的类型以及它们的状态（正在运行或者已经停止）。设备驱动程序和Windows服务进程都是在同一个地方被定义的：HKLM\SYSTEM\CurrentControlSet\Services。然而，它们是通过一个类型代码来区分的——例如，类型1是指内核模式的设备驱动程序（在注册表中存储的有关设备驱动程序的完整信息列表，请参见表4.7）。

另外一种方法是，你可以利用Drivers实用工具（Drivers.exe，在Windows 2000资源工具箱中）或者Pstat实用工具（Pstat.exe，在Windows XP支持工具箱、Windows Server 2003支持工具箱、Windows 2000资源工具箱，以及Platform SDK中均包含了该工具）列出当前系统中已经加载的设备驱动程序。下面是Drivers实用工具的一部分输出：

```
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
```

[1]此处是根据原书勘误改正过来的，原书的叙述是：在“开始”菜单中，选择“程序”，然后选择“管理工具”，再选择“计算机管理”，或者从控制面板中，打开“管理工具”，再选择“计算机管理”。在计算机管理程序中，展开“系统信息”，然后展开“软件环境”，再打开“驱动程序”。此操作适用于Windows 2000但不适用于Windows XP（因为Windows XP的“计算机管理”里面没有“系统信息”栏目）。——编者

[2]在Windows XP里，“软件环境”下显示的是“系统驱动程序”，而在Windows 2000里显示的是“驱动程序”。——编者

www.TopSage.com

每一个已被加载的内核模式组件（Ntoskrnl、HAL以及设备驱动程序）都被列出来了，其中也包括每个映像中各个区域的大小。

Pstat实用工具也可以显示已被加载的驱动程序列表，但是要等它显示了进程列表和每个进程中的线程之后才显示驱动程序列表。有一个重要的信息只有Pstat才能显示，Drivers工具并没有显示出来，那就是每个模块在系统空间中的加载地址。正如我们稍后将要解释的那样，为了将正在运行的系统线程映射到它们所在的设备驱动程序中，此加载地址是必需的。

看一看尚未文档化的接口

在关键的系统映像中查看那些导出符号或全局符号的名字或许是非常有启发的——你可以对“Windows可能会做些什么样的事情”以及“今天那些已被文档化并提供了很好支持的技术”有一个大致的概念。当然，仅仅因为你知道了这些函数的名字并不意味着你可以或者应该调用这些函数——接口并未被文档化，很可能会有变化。我们的建议是，你纯粹是为了对Windows执行的内部功能有所认识才查看这些函数，而并非是为了绕过那些正式支持的接口。

例如，查看Ntdll.dll中的函数列表可以让你了解到，Windows为用户模式子系统DLL提供的所有系统服务，以及相对于每个子系统暴露出来的子集。尽管这些函数中有许多被明确地映射到文档中定义和支持的Windows函数，但还是有几个函数没有通过Windows API暴露出来（请参考*www.sysinternals.com*上的“Inside the Native API”一文）。

相反地，检查一下Windows子系统DLL（比如Kernel32.dll或Advapi32.dll）的导入表，以及它们调用了Ntdll中的哪些函数也是非常有意思的。

另一个有趣的、值得转储的映像文件是Ntoskrnl.exe——尽管内核模式设备驱动程序使用的许多导出函数在Windows DDK中已经有了文档了，但是仍然有相当多的函数尚未被文档化。你可能会发现，看一看Ntoskrnl和HAL的导入表也是十分有意思的；该导入表显示了Ntoskrnl使用了HAL中哪些函数，以及反过来HAL使用了Ntoskrnl的哪些函数。

www.TopSage.com

表2.7列出了执行体组件常用的绝大多数函数名前缀。这些主要的执行体组件，每个都会使用前缀的变形方式——前缀的第一个字母后面跟一个*i*（代表*internal*即**内部的**），或者整个前缀后面跟一个*p*（代表*private*即**私有的**）来标记内部函数。例如，*Ki*代表内部内核函数，而*Psp*指内部进程支持函数。

表2.7 常用的前缀

前缀	组件
Cc	高速缓存管理器（Cache manager）
Cm	配置管理器（Configuration manager）
Ex	执行体支持例程（Executive support routines）
FsRtl	文件系统驱动程序运行库（File system driver run-time library）
Hal	硬件抽象层（Hardware abstraction layer）
Io	I/O管理器（I/O manager）
Ke	内核（Kernel）
Lpc	本地过程调用（Local procedure call）
Lsa	本地安全认证（Local Security Authority）
Mm	内存管理器（Memory manager）
Nt	Windows系统服务（绝大多数被导出为Windows函数）（Windows system services）
Ob	对象管理器（Object manager）
Po	电源管理器（Power manager）
Pp	PnP管理器（PnP manager）
Ps	进程支持（Process support）
Rtl	运行库（Run-time library）
Se	安全性（Security）
Wmi	Windows管理规范（Windows Management Instrumentation）
Zw	以Nt开头的系统服务入口点的镜像，它把原先的访问模式设置为内核模式，从而消除了参数的有效性检查过程，因为Nt系统服务只有当原来的访问模式为用户模式时才进行参数检查

如果你理解了Windows系统例程的命名规范，就可以很容易地将这些导出函数的名称解析出来。一般的格式是：

<前缀><操作><对象>

在这一格式中，“前缀”部分是导出该例程的内部组件；“操作”部分说明了在对象或者资源上做了什么工作；“对象”部分标记了是在什么上进行操作的。

例如，*ExAllocatePoolWithTag*是一个负责从换页的或者非换页的池里进行内存分配的执行体支持例程。*KeInitializeThread*则是分配并建立内核线程对象的例程。

www.TopSage.com

系统进程

以下的系统进程会出现在每一个Windows系统中（其中两个——空闲进程和System进程——并不是完整的进程，因为它们并不是在运行一个用户模式的可执行文件）：

- 空闲（Idle）进程（每个CPU一个线程，占用空闲的CPU时间）；
- System进程（包含大多数内核模式系统线程）；
- 会话管理器（Smss.exe）；
- Windows子系统（Csrss.exe）；
- 登录进程（Winlogon.exe）；
- 服务控制管理器（Services.exe）和它创建的子服务进程（比如系统提供的通用服务宿主进程Svchost.exe）；
- 本地安全认证服务器（Lsass.exe）。

为了理解这些进程的关系，查看一下系统的进程“树”是很有帮助的。也就是说，进程之间是父/子关系。看一看哪个进程创建了其他哪个进程，将有助于理解每个进程是从哪里来的。图2.6的屏幕截图显示了部分进程树，其中前面少数几个系统进程有相应的注释（comment）。（Process Explorer允许你为每个进程加上一条注释，然后以可选的方式显示在屏幕上，作为窗口中的一列。）

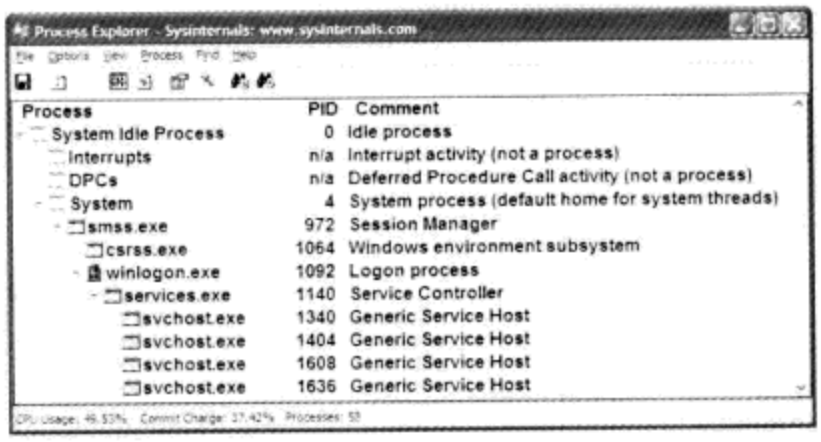


图2.6 初始的系统进程树

接下去的几部分将讲述图2.6中显示的关键系统进程。尽管这些部分简要地说明了进程启动的顺序，但是，第5章将会详细地描述在Windows引导和启动过程中涉及的步骤。

空闲进程

图2.6中列出的第一个进程是系统空闲进程。正如我们将在第6章中介绍的那样，进程是由它们的映像文件名来标记的。

www.TopSage.com

然而，这个进程（以及名为System的进程）并没有运行一个实际的用户模式映像文件（也就是说，在\Windows目录下没有名为“System Idle Process.exe”的文件）。而且，在不同的工具中，该进程的显示名称是不同的（由于实现细节的原因）。表2.8列出了空闲进程的一些名称（进程ID为0）。第6章中详细解释了空闲进程。

表2.8 进程ID为0在各种工具中的名称

实用工具	进程ID为0的名称
任务管理器	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

现在我们来看一看系统线程，以及每一个真正运行映像文件的系统进程的用途。

中断和DPC

标记为“Interrupts（中断）”和“DPCs”的这两行代表了用于中断和延迟过程调用的时间。第3章将会介绍这两种机制。请注意，虽然在Process Explorer中，它们被显示在进程列表中，但它们并不是进程。它们之所以被显示在这里，是因为它们所消耗的CPU时间并没有计算在任何一个进程中（例如，若一个应用系统有大量的中断任务，则它不会被显示成一个很消耗CPU时间的进程）。注意，任务管理器将中断和DPC时间包含在系统空闲时间中。因此，如果使用任务管理器的话，那么像这种具有大量中断任务的应用系统看起来似乎是空闲的。

System进程和系统线程

System进程（在Windows 2000，为进程ID 8；在Windows XP和Windows Server 2003中则是进程ID 4）是一种特殊线程的母体，这种特殊线程只能在内核模式下运行，称为**内核模式系统线程**（*kernel-mode system thread*）。系统线程具备普通用户模式线程的所有属性和环境（比如硬件环境、优先级，等等），但是不同的地方在于，它们只在内核模式下运行系统空间中加载的代码，无论这些代码是在Ntoskrnl.exe中，还是在任何其他加载进来的设备驱动程序中。而且，系统线程没有一个用户进程地址空间，因此，任何的动态存储空间，它们都必须从操作系统的内存堆中分配，比如从一个换页的或者非换页的池中分配。

系统线程是通过*PsCreateSystemThread*函数（DDK中有文档说明）来创建的，该函数只能从内核模式中调用。Windows以及各种设备驱动程序在系统初始化阶段创建系统线程，以便执行各种要求线程环境的操作，比如发出和等待I/O或其他对象，或者选举（poll）一个设备。例如，内存管理器使用系统线程来实现诸如“将脏页面写到页面文件或者映射文件中”、“将进程在内存中换进换出”之类的功能。

www.TopSage.com

内核会创建一个称为**平衡集管理器**（*balance set manager*）的系统线程，它每秒钟被唤醒一次，从而有可能发出各种与调度和内存管理相关的事件。高速缓存管理器也使用系统线程来实现“预读（read-ahead）”和“滞后写（write-behind）”I/O。文件服务器设备驱动程序（Srv.sys）利用系统线程来响应那些与网络共享（即磁盘分区上被共享到网络上的文件数据）有关的网络I/O请求。甚至软盘驱动程序也有一个系统线程来选举软盘设备（在这种情况下，选举更加有效，因为一个靠中断驱动的软盘驱动程序要消耗大量的系统资源）。有关特定系统线程的进一步信息，包含在讲述相应组件的章节中。

在默认情况下，系统线程是属于System进程的，但是，一个设备驱动程序可以在任何的进程中创建系统线程。例如，Windows子系统设备驱动程序（Win32k.sys）在Windows子系统进程（Csrss.exe）中创建系统线程，因而这些线程可以很容易地访问该进程用户模式地址空间中的数据。

当你在诊断或者检查一个系统的时候，若能够将单独的系统线程的执行过程映射到驱动程序中，甚至映射到包含该代码的子例程中，则一定非常有用。例如，在一个负载很重的文件服务器上，System进程很有可能会消耗掉相当可观的CPU时间。但是，仅仅知道“当System进程在运行的时候‘某一个系统线程’正在运行”，是不足以确定哪个设备驱动程序或者操作系统组件正在运行的。

所以，如果System进程中的线程正在运行的话，首先要确定哪些线程正在运行（例如，通过性能工具）。一旦你找到了当前正在运行的一个或者多个线程，就可以查看一下该系统线程是在哪个驱动程序中开始执行的（至少可以告诉你可能是哪个驱动程序创建了这个线程），或者检查一下所涉及的线程的调用栈（call stack），通过调用栈可以知道该线程当前正在哪里执行。

下面的两个实验演示了这两项技巧。



实验：标识出System进程中的系统线程

你可以看到，System进程内部的线程一定是内核模式的系统线程，因为每个线程的启动地址都大于系统空间的开始地址（默认从0x80000000开始，除非系统引导时在Boot.ini中用了/3GB开关）。而且，如果你看一看这些线程的CPU时间，你将会看到，这些线程累积起来的CPU时间都是在内核模式下运行的。

为了找出是哪个驱动程序创建了某个系统线程，请检查一下该线程的启动地址（你可以用Pviewer.exe来显示此信息）。然后查找哪个驱动程序的基地址离该线程的启动地址最接近（但是位于该地址的前面）。Pstat实用工具（其输出的末尾）和内核调试器命令*!drivers*都可以列出每个已被加载的设备驱动程序的基地址。

为了很快地找到一个线程的当前地址，请在内核调试器中使用*!stacks 0*命令。下面是从一个实际的系统中输出的样例（使用了LiveKd）：

www.TopSage.com

```
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
```

第一列是进程ID和线程ID（按照格式“进程ID.线程ID”显示）。第二列是线程的当前地址。第三列表明了该线程是处于等待（wait）状态、就绪（ready）状态，还是运行（running）状态（请参见第6章关于线程状态的讲述）。最后一列是该线程的栈中最顶上的地址。这最后一列的信息可以使得我们很容易看出，每个线程是在哪个驱动程序中被启动起来的。对于Ntoskrnl中的线程，函数的名字进一步指明了该线程正在做什么。

然而，如果正在运行的线程是一个系统辅助线程（ExpWorkerThread），那么你仍然不知道该线程在做什么，因为任何一个设备驱动程序都可以将工作提交给一个系统辅助线程。因此，追溯辅助线程行为的惟一方法是，在ExQueueWorkItem中设置一个断点。

www.TopSage.com

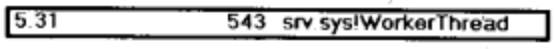
当你到达该断点的时候，输入**!dso work_queue_item esp+4**。该命令将会转储第一个实参，它指向ExQueueWorkItem（一个工作队列结构），其中包含了在辅助线程环境中被调用的辅助例程的地址。此外，你在内核调试器中利用 *k* 命令（该命令显示当前的调用栈）可以看到调用者。当前的调用栈将会显示出，是哪个驱动程序将自己的工作排入队列中，并得到了此辅助线程（相对于从辅助线程中被调用的例程而言）。



实验：将系统线程映射到一个设备驱动程序上

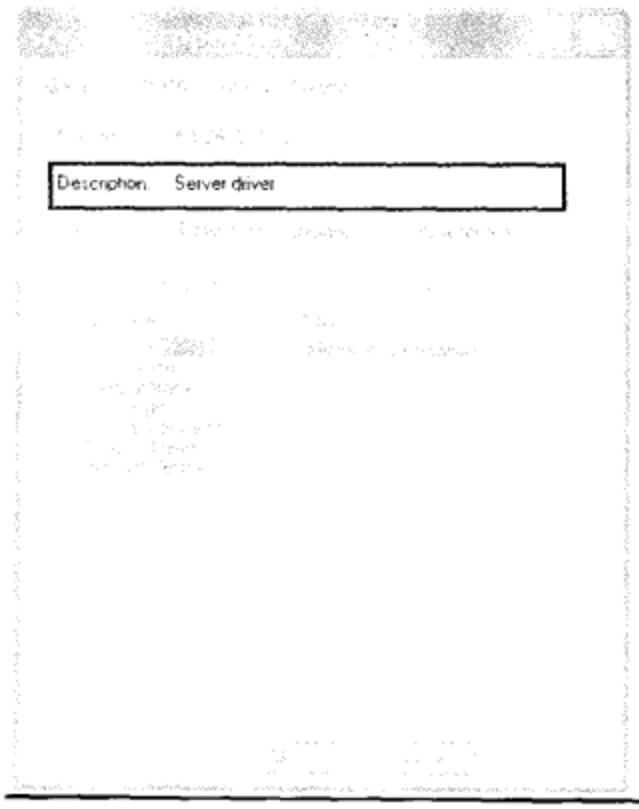
在这个实验中，我们会看到，如何将System进程中的CPU活动映射到产生该活动的相应的系统线程上（以及它所在的驱动程序）。这很重要，因为当System进程正在运行的时候，你必须在线程粒度上才能真正理解当前正在发生的事情。在这个实验中，我们将在你的机器上产生文件服务器的活动，以此来生成系统线程的活动。（文件服务器驱动程序，Srv.sys，创建一些系统线程来处理它所接到的针对文件I/O的请求。有关该组件的更多信息，请参考第13章）。

1. 打开一个命令提示窗口。
2. 通过一个网络路径来访问你的C驱动器，列出你的整个C驱动器的所有目录。例如，如果你的计算机名称是COMPUTER1，则输入“**dir \\computer1\c$ /s**”（/s开关列出所有的子目录）。
3. 运行Process Explorer，用鼠标在System进程上双击。
4. 点击Threads标签。
5. 按照“CSwitch Delta（context switch delta，环境切换差量）”列排序。你应该看到，在Srv.sys中有一个或者多个线程正在运行，如下图所示：



www.TopSage.com

如果你看到一个系统线程正在运行，但你不确定对应的驱动程序是哪个，那么，请按下Module按钮，这将会弹出文件属性对话框。如上图那样，当Srv.sys中的线程被选中的时候按下Module按钮，就会得到下面的显示结果。



会话管理器

会话管理器（\Windows\System32\Smss.exe）是系统中第一个创建的用户模式进程。负责完成执行体和内核初始化工作的内核模式系统线程在最后阶段创建了实际的Smss进程。

在启动Windows的过程中，会话管理器负责许多重要的步骤，比如打开额外的页面文件、执行延迟的文件改名和删除操作、创建系统环境变量。它也将子系统进程（通常只有Csrss.exe）和Winlogon进程启动起来，Winlogon进程依次会创建其他的系统进程。

在注册表的HKLM\SYSTEM\CurrentControlSet\Control\Session Manager下面，你可以找到许多配置信息，它们驱动了Smss的初始化步骤。其中有些信息在第5章介绍Smss的时候将会进一步解释（有关这些注册表键和值的详细描述，请参考Windows 2000资源工具箱中的注册表条目帮助文件Regentry.chm）。

Smss中的主线程在执行了这些初始化步骤以后，一直在Csrss和Winlogon的进程句柄上等待。如果这两个进程中的任何一个非正常终止了，则Smss让系统崩溃掉（崩溃的代码是STATUS_SYSTEM_PROCESS_TERMINATED或0xC000021A），因为Windows要依赖于这两个进程的存在才能运行得下去。

www.TopSage.com

同时，Smss等待加载子系统的请求、调试事件，以及创建新的终端服务器会话（terminal server sessions）的请求（关于终端服务的介绍，请参见第1章的“终端服务及多个会话”一节）。

终端服务会话（Terminal Services session）的创建是由Smss来完成的。当Smss接到一个创建会话的请求时，它首先调用NtSetSystemInformation，请求建立内核模式的会话数据结构。这又依次调用内部的内存管理器函数**MmSessionCreate**，该函数建立起会话虚拟地址空间，该地址空间中包含会话中的换页内存池，以及由Win32子系统的内核模式部分（Win32k.sys）和其他的会话空间设备驱动程序所分配的、属于每个会话的数据结构（更多的细节请参考第7章）。然后，Smss为该会话创建Winlogon和Csrss的实例。

Winlogon、LSASS和Userinit

Windows登录进程（\Windows\System32\Winlogon.exe）处理交互式用户的登录和注销。当安全注意序列（SAS，Secure Attention Sequence）组合键被按下的时候，Winlogon就会接到一个用户登录请求。在Windows上默认的SAS是组合键Ctrl+Alt+Del。使用SAS的原因是为了保护用户避免那些模拟登录过程的口令窃听程序，因为用户模式应用程序不可能截取这一键盘序列。

登录过程的身份识别和认证是在一个名为GINA（Graphical Identification and Authentication，图形识别和认证）的可替换的DLL中实现的。标准的Windows GINA，即Msgina.dll，实现了默认的Windows登录界面。然而，开发人员可以提供他们自己的GINA DLL来实现其他的身份识别和认证机制——比如基于声波纹（voice print）的方法——以替换Windows标准的用户名/口令方法。而且，Winlogon可以加载那些需要执行二级认证的附加网络提供者DLL。这种能力使得多个网络提供者可以在正常的登录过程中一次采集到所有的身份识别和认证信息。

一旦用户名和口令已经捕捉到了，就可以将它们送到本地安全认证服务器进程（\Windows\System32\Lsass.exe，第8章中讲述）进行认证。LSASS调用适当的认证包（实现为DLL的形式）以执行实际的验证操作，比如检查该口令是否符合存储在活动目录或者SAM（属于注册表的一部分，其中包含了关于用户和组的定义）中的口令信息。

在成功完成了认证以后，LSASS调用安全引用监视器中的一个函数（例如，*NtCreateToken*），以生成一个访问令牌对象（access token object），该对象包含了当前用户的安全轮廓。然后，Winlogon利用此访问令牌来创建该用户会话中的初始进程。这（或这些）初始进程被存储在注册表键HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon下的注册表值*Userinit*中（默认是Useinit.exe，但是，在该注册表值中可以列出多个映像）。

Userinit执行该用户环境中的一些初始化工作（比如运行登录脚本、应用组策略），然后在注册表中查找*Shell*值（在前面提到过的同样的Winlogon键的下面），并且创建一个进程来运行系统定义的外壳程序（默认是Explorer.exe）。

www.TopSage.com

然后，Useinit退出。这正是为什么Explorer.exe在进程树中没有父进程的原因——它的父进程已经退出，而且正如前面所解释的，tlist将所有父进程已不在运行的进程调整到最左边（看待这一现象的另一种方法是：Explorer是Winlogon的孙子进程）。

Winlogon不仅当用户登录和注销的时候是活动的，而且无论何时，当它截取到键盘的SAS时也是活动的。例如，当你登录进来后，若按下了Ctrl+Alt+Del，则Windows安全对话框就会出现，提示“注销”、“启动任务管理器”、“锁住工作站”、“关闭系统”等选项。Winlogon是负责处理这些交互工作的进程。

关于登录过程中各个步骤的完整描述，请参考第5章的“Smss、Csrss和Winlogon”一节。有关安全认证的更多细节，请参考第8章。关于在跟LSASS交互时可调用函数（以**Lsa**开头的函数）的细节，请参考Platform SDK中的文档。

服务控制管理器

回顾本章前面提到过的，Windows中的“服务（services）”既可以指一个服务器进程，也可以指一个设备驱动程序。这一节讲的服务是指用户模式的进程。服务就像UNIX的“守护进程（daemon process）”或者VMS的“分派进程（detached process）”，这些进程可以被配置成能在系统引导时候自动启动起来，而无需交互式的登录过程。它们也可以被配置成手工启动（比如，通过运行Services管理工具，或者调用Windows的*StartService*函数）。在通常情况下，这些服务并不与登录用户进行交互，尽管在有些特殊的条件下这是可能的（参考第4章）。

服务控制管理器是一个特殊的系统进程，它运行的映像文件是\Windows\System32\Services.exe，它负责启动、停止服务进程，也负责与这些服务进程进行交互。所谓服务程序，实际上只是调用了一些特殊Windows函数的Windows映像。它们通过这些特殊的Windows函数，与服务控制管理器进行交互，以便执行诸如此类的动作：注册一个服务的成功启动、响应那些对状态的请求，以及暂停或停止一个服务。在注册表的HKLM\SYSTEM\CurrentControlSet\Services下定义了一个系统的服务。资源工具箱的Registry Entries帮助文件（Regentry.chm）记载了针对系统服务的子键和值。

记住，服务有三种名称：你在系统中看到的正在运行的进程名、注册表中的内部名称，以及在Services管理工具中给出的显示名（并不是所有的服务都有显示名——如果一个服务没有显示名的话，则显示内部名称）。服务还可以有一个描述域（description field），以进一步详细解释一个服务的功能。

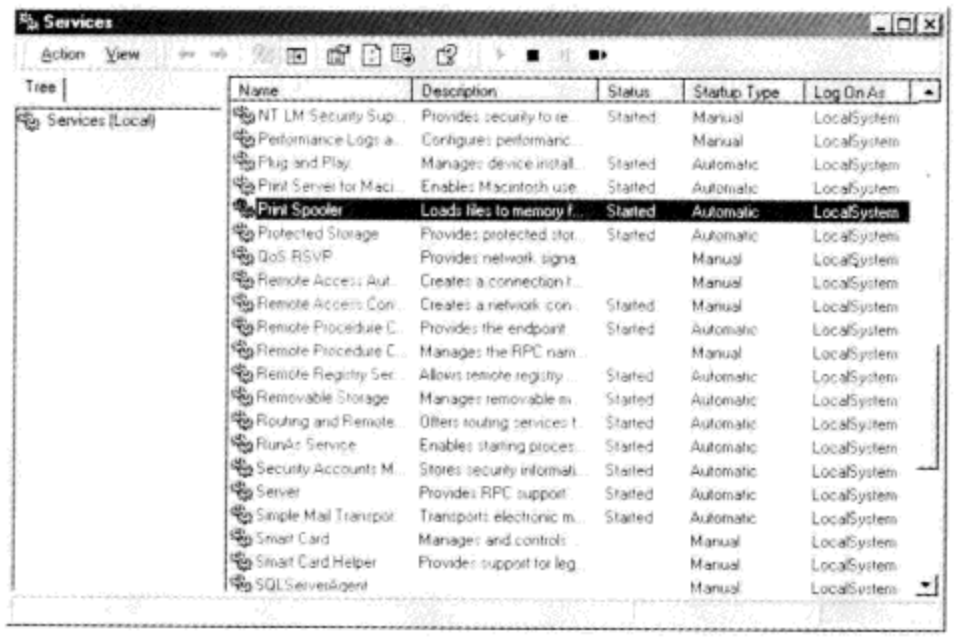
要想从一个服务进程映射到该进程所包含的服务，可以使用 *tlist /s* 命令。请注意，在服务进程和所运行的服务之间并不总是一一对应的，因为有的服务与其他的服务共享一个进程。在注册表中，一个服务的类型代码指明了该服务是运行它自己的进程，还是与同一映像中的其他服务共享一个进程。

有许多Windows组件是以服务的方式来实现的，比如Spooler（假脱机）、事件日志、任务调度器，以及各种网络组件。

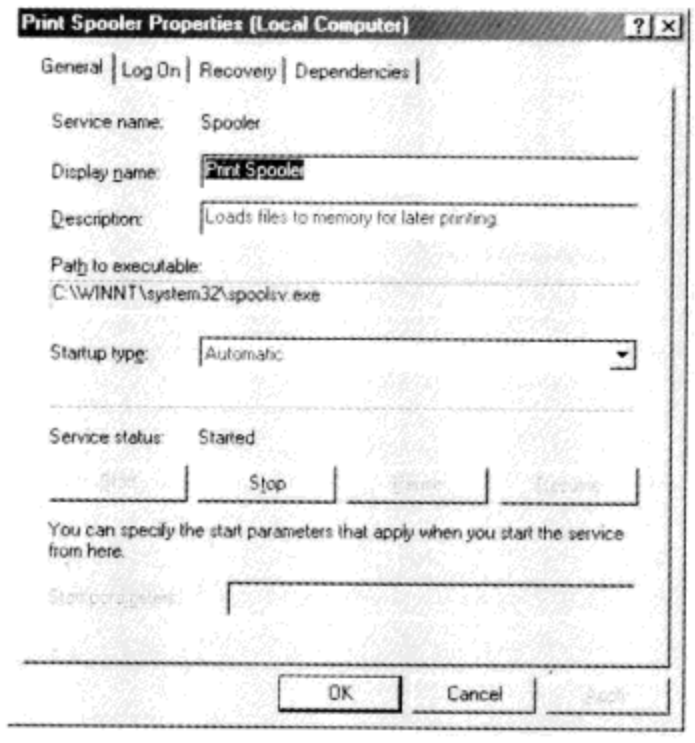
www.TopSage.com

实验：列出当前安装的服务

为了列出当前系统中所安装的服务，请从控制面板中选择“管理工具（Administrative Tools）”，然后选择“服务（Services）”。你应该会看到如下所示的输出：



要想看到有关一个服务的详细属性，请在这个服务上右键点击，然后选择“属性（Properties）”。例如，下面是打印假脱机（Print Spooler）服务（在上一个图中已被加亮显示）的属性：



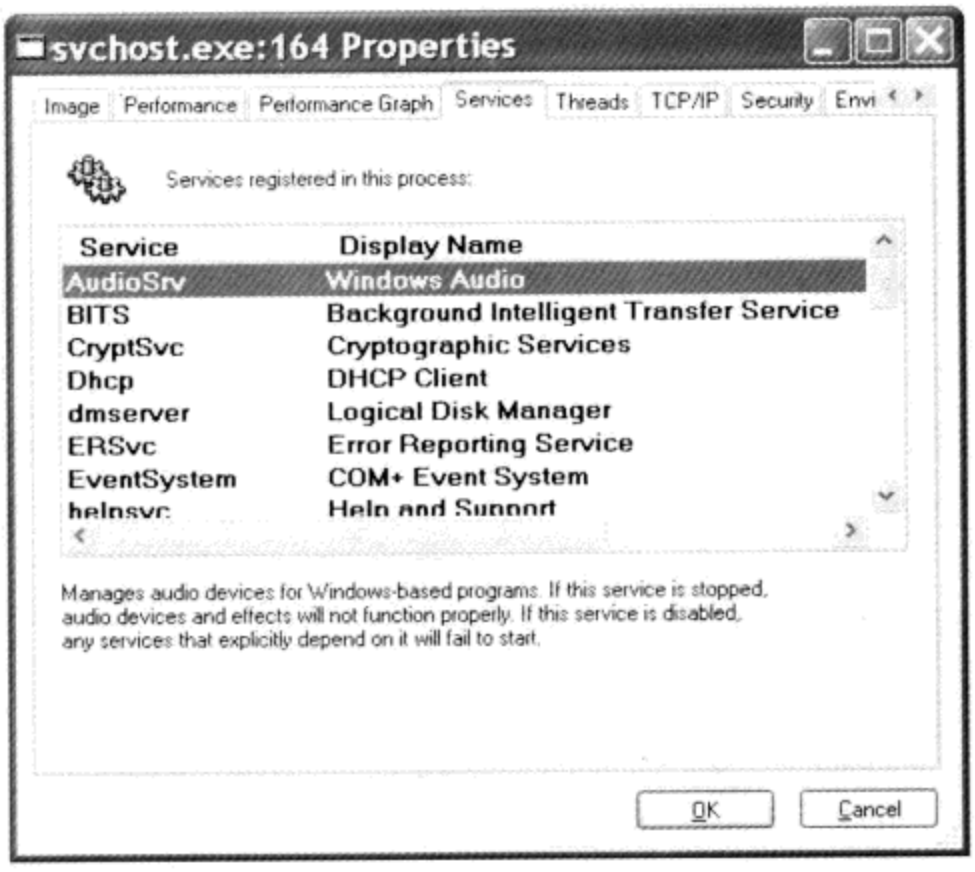
注意，“可执行文件的路径（Path To Executable）”域指明了包含这个服务的程序。记住，有的服务与其他的服务共享同一个进程——从服务到进程之间的映射关系并不总是一一对应的。

www.TopSage.com



实验：检查服务进程内部的有关服务的细节

Process Explorer可以加亮显示那些包含了一个或者多个服务的进程(你只需在Options菜单中选择“Configure Highlighting”中的项目就可以配置成这样）。如果你在一个包含了服务的进程上双击鼠标，你将会看到有一个Services视图，其中列出了该进程内部的服务：定义了该服务的注册表键的名称、管理员可以看得见的显示名，以及描述该服务的文本信息（如果有的话）。例如，在Windows XP上，一个运行在System账号下的Svchost.exe进程中所包含的服务如下：



有关服务的更多细节，请参见第4章。

2.5 本章总结

在本章中，我们概括地看了看Windows的总体系统结构。我们检查了Windows的关键组件，看到了它们是如何相互关联起来的。在下一章，我们将更加详细地看一看这些组件所赖以建立起来的核心系统机制，比如对象管理器（object manager）和同步机制（synchronization）。

www.TopSage.com

System Mechanisms

第 3 章　系统机制

Microsoft Windows提供了一些基本的机制供内核模式的组件（比如执行体、内核和设备驱动程序）来使用。本章将解释下面的系统机制，并且也介绍了如何使用这些机制：

- 陷阱分发，包括中断、延迟的过程调用（DPC）、异步过程调用（APC）、异常分发，以及系统服务分发；
- 执行体对象管理器；
- 同步，包括自旋锁、内核分发器对象，以及等待是如何实现的；
- 系统辅助线程；
- 其他的机制，比如Windows全局标志；
- 本地过程调用；
- 内核事件追踪；
- Wow64。

3.1　陷阱分发

中断和异常是导致处理器转向正常控制流之外代码的两种操作系统条件。硬件或者软件都可以检测到这两种条件。术语“**陷阱**（*trap*）”指的是，“当异常或者中断发生时，处理器捕捉到一个执行线程，并且将控制权转移到操作系统中某一固定地址处”的机制。在Windows中，处理器将控制权转给一个**陷阱处理器**（*trap handler*）。所谓陷阱处理器，是指与某个特殊的中断或异常相关联的一个函数。图3.1显示了一些能激活陷阱处理器的条件。

内核对待中断和异常是有区别的，表现在以下一些方面。**中断**是一个异步事件（可以在任何时候发生），并且与处理器当前正在执行的任务毫无关系。中断主要是由I/O设备、处理器时钟，或者定时器产生的，另外，中断可以被允许（打开），或者被禁止（关闭）。相反地，**异常**是一个同步条件，它是一个特殊指令执行的结果。在同样的条件下用同样的数据第二次运行一个程序可以重现原来的异常。异常的例子有：内存访问违例、特定的调试器指令，以及除零错误。内核也把系统服务调用看做异常（不过，从技术上而言，它们是系统陷阱）。

www.TopSage.com

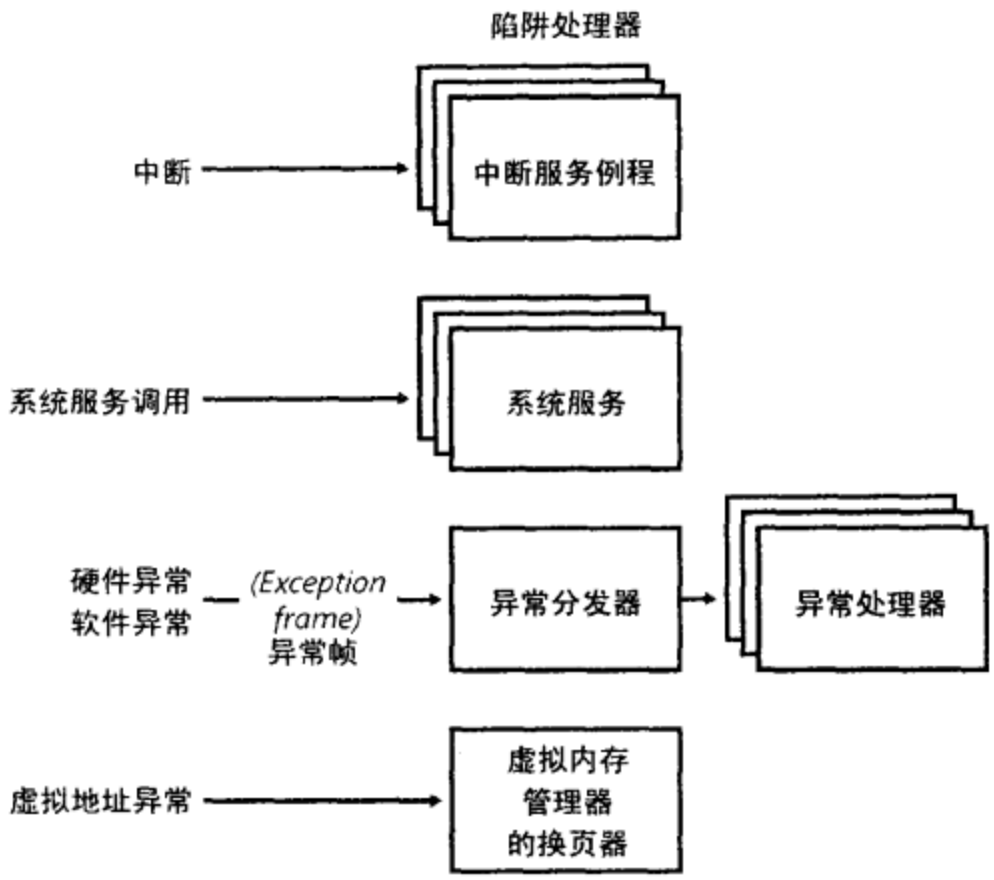


图3.1 陷阱分发

无论是硬件还是软件都能够产生异常和中断。例如，总线错误异常是由于硬件问题引起的，而除零异常则是软件错误的结果。同样，I/O设备可以产生中断，或者内核本身也可以发出软中断（比如APC或者DPC，本章后面将会讲到这些机制）。

当一个硬件异常或者中断产生的时候，处理器在被中断的线程的内核栈中记录下足够多的机器状态信息，因而它可以回到控制流中该点处继续执行，就好像什么也没有发生过一样。如果该线程在用户模式下执行，那么Windows就切换到该线程的内核模式栈。然后，Windows在被中断的线程的内核栈上创建一个**陷阱帧**（*trap frame*），并且把线程的执行状态保存到陷阱帧里。陷阱帧是一个线程的完整执行环境的一个子集，你在内核调试器中输入**dt nt!_ktrap_frame**就可以看到陷阱帧的定义（第6章讲述了线程环境）。内核在处理软中断时，或者将软中断当做硬中断处理的一部分，或者当一个线程调用与软中断相关的内核函数时以异步方式进行处理。

在大多数情况下，内核安装了前端陷阱处理函数，在内核将控制权转交给与该陷阱相关的其他函数之前或者之后，由这些前端陷阱处理函数来执行一些常规的陷阱处理任务。例如，如果陷阱条件是一个设备中断，则内核硬件中断陷阱处理器将控制权转交给一个由设备驱动程序提供给该中断设备的**中断服务例程**（ISR，*interrupt service routine*）。如果陷阱条件是因为调用一个系统服务而引发的，那么，通用的系统服务陷阱处理器将控制权转交给执行体中指定的

www.TopSage.com

系统服务函数。内核也会为它不希望看到的或者根本不处理的陷阱安装陷阱处理器。这些陷阱处理器一般的做法是执行系统函数*KeBugCheckEx*，当内核检测到可能导致数据破坏的有问题行为或者不正确行为时，该函数会停止计算机（有关错误检查的更多信息，请参考第14章）。下面的章节将会更加详细地讲述中断、异常和系统服务分发机制。

中断分发

硬件产生的中断往往是由I/O设备激发的，当这些设备需要服务的时候，它们以中断的方式通知处理器。中断驱动的设备使得操作系统可以交替地进行中心处理和I/O操作，从而最大限度地发挥处理器的能力。当一个线程启动了一个与设备之间传输数据的任务以后，它在该设备完成此传输任务的过程中，可以进行其他有用的工作。在设备完成传输任务以后，它中断处理器，请求服务。定点设备、打印机、键盘、磁盘驱动器和网卡一般来说都是中断驱动的。

系统软件也可能产生中断。例如，内核可能会激发一个软件中断来触发线程分发过程，同时也以异步方式打断了一个线程的执行。内核也可以禁止中断，因而处理器不会被中断，但是它只能在很少情况下可以这样做——只有在关键时候才可以这样做，比如当处理器正在处理一个中断，或者正在分发一个异常的时候。

内核安装了中断陷阱处理器来响应设备的中断。中断陷阱处理器或者将控制权传递给一个负责处理该中断的外部例程（ISR），或者传递给一个响应该中断的内部内核例程。设备驱动程序提供了ISR来处理设备中断，而内核则为其他类型的中断提供了中断处理例程。

在下面的小节中，你将会了解到：硬件是如何向处理器通知中断的、内核支持的中断类型，设备驱动程序是如何与内核进行交互的（作为中断处理的一部分），以及内核如何识别软件中断（以及用于实现这些软件中断的内核对象）的。

硬件中断处理

在Windows所支持的硬件平台上，外部I/O中断进入到中断控制器的某一根线上。该控制器然后在一根线上中断处理器。一旦处理器被中断了，它询问控制器以获得此**中断请求**（IRQ，*interrupt request*）。中断控制器将该IRQ转译成一个中断号，利用该号码作为在**中断分发表**（*interrupt dispatch table*）中的索引，并且将控制权传递给恰当的中断分发例程。在系统引导的时候，Windows会填充IDT，其中包含了指向内核中负责处理每个中断和异常的例程的指针。

www.TopSage.com



实验：查看IDT

利用*!idt*内核调试器命令，你可以查看IDT的内容，其中包含了有关Windows为每个中断（包括异常和IRQ）所分配的陷阱处理器的信息。*!idt*命令若不带任何标志，则会显示出每个中断被映射至除了Ntoskrnl.exe之外的其他模块中的相应地址。

下面的例子显示了*!idt*命令输出的样子：

```
kd> !idt

Dumping IDT:

30:      806b14c0 hal!HalpClockInterrupt
31:      8a39dc3c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8a39dc00)
34:      8a436dd4 serial!SerialCIsrSw (KINTERRUPT 8a436d98)
35:      8a44ed74 NDIS!ndisMIsr (KINTERRUPT 8a44ed38)
                  portcls!CInterruptSync::Release+0x10 (KINTERRUPT 899c44a0)
38:      806abe80 hal!HalpProfileInterrupt
39:      8a4a8abc ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 8a4a8a80)
3b:      8a48d8c4 pcmcia!PcmciaInterrupt (KINTERRUPT 8a48d888)
                  ohci1394!OhciIsr (KINTERRUPT 8a41da18)
                  VIDEOPRT!pVideoPortInterrupt (KINTERRUPT 8a1bc2c0)
                  USBPORT!USBPORT_InterruptService (KINTERRUPT 8a2302b8)
                  USBPORT!USBPORT_InterruptService (KINTERRUPT 8a0b8008)
                  USBPORT!USBPORT_InterruptService (KINTERRUPT 8a170008)
                  USBPORT!USBPORT_InterruptService (KINTERRUPT 8a258380)
                  NDIS!ndisMIsr (KINTERRUPT 8a0e0430)
3c:      8a39d3ec i8042prt!I8042MouseInterruptService (KINTERRUPT 8a39d3b0)
3e:      8a47264c atapi!IdePortInterrupt (KINTERRUPT 8a472610)
3f:      8a489b3c atapi!IdePortInterrupt (KINTERRUPT 8a489b00)
```

在输出以上结果的实验系统中，键盘设备驱动程序（I8042prt.sys）的键盘ISR位于中断号0x31上，另外，有几个设备（包括视频适配器、PCMCIA总线、USB和IEEE 1394端口，以及网络适配器）共用中断0x3B。

Windows将硬件IRQ映射至IDT中的中断号上，同时它也利用IDT来为异常配置陷阱处理器。例如，x86和x64针对页面错误的异常号（当一个线程企图访问虚拟内存中某一个尚未定义或者存在的页面时就会发生这种异常）是0xe。因此，在IDT中0xe这一项指向系统的页面错误处理器。虽然Windows所支持的硬件体系结构允许最多可达256个IDT项，但是，一台特定机器所能支持的IRQ的数量则是由该机器所用的中断控制器的具体设计所决定的。

www.TopSage.com

每个处理器都有单独的IDT，所以，如果合适的话，不同的处理器可以运行不同的ISR。例如，在一个多处理器系统中，每个处理器都接收时钟中断，但是，只有一个处理器在响应该中断的时候更新系统时钟。然而，所有的处理器都使用该中断来测量线程的时限（quantum）使用情况，当一个线程的时限结束时，通过该中断发起重新调度过程。类似地，在有些系统配置中，可能要求由某一个特定的处理器来处理特定的设备中断。

x86中断控制器

绝大多数x86系统或者依赖于i8259A可编程中断控制器（PIC），或者依赖于i82489高级可编程中断控制器（APIC）的一个变种；大多数新的计算机都包含一个APIC。PIC标准起源于最初的IBM PC。PIC只能在单处理器系统上工作，它有15条中断线。APIC和SAPIC（稍后讨论）可以在多处理器系统上工作，它们有256条中断线。Intel和其他公司已经定义了多处理器规范（MP Specification），这是一种针对采用了APIC作为中心控制的x86多处理器系统的设计标准。为了保持与单处理器操作系统的兼容性，以及使引导代码可在单处理器模式下启动一个多处理器系统，APIC支持一种仅包含15个中断的PIC兼容模式，并且只将这些中断递交给主处理器。图3.2显示了这种APIC结构。APIC实际上是由几个部件构成的：一个专门接收设备中断的I/O APIC，以及一些本地APIC，和一个与i8259A兼容的中断控制器。本地APIC在一条私有的APIC总线上接收来自I/O APIC的中断，而i8259A兼容的中断控制器则负责将APIC输入转换成与PIC等价的信号。I/O APIC负责实现中断转送算法（interrupt routing algorithm）——这些算法是可以由软件来选择的（在Windows上，由硬件抽象层或者HAL来作出选择）——既可以平衡各个处理器之间的设备中断负载，也可以设法利用位置特性，将设备中断递交给刚刚承担了前一个同样类型中断的同一个处理器。

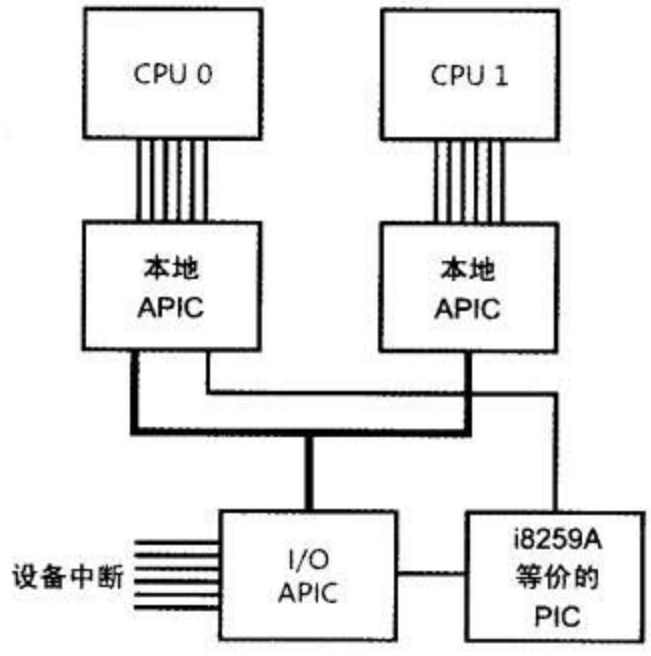


图3.2 x86 APIC体系结构

www.TopSage.com

x64中断控制器

因为x64体系结构与x86操作系统兼容，所以x64系统必须提供与x86同样的中断控制器。然而，一个重要的区别是，Windows的x64版本将无法运行于不具有APIC的系统上，而且它们使用APIC进行中断控制。

IA64中断控制器

IA64体系结构依赖于改进的高级可编程中断控制器（SAPIC），这是APIC的一个演化。APIC与SAPIC体系结构之间的一个重要区别是，APIC系统上的I/O APIC利用一条私有的APIC总线将中断递交给本地APIC，而在SAPIC系统中，这些中断通过I/O和系统总线来进行更加快速的递交。另一个区别是，在APIC系统上，中断转送和负载均衡是由APIC总线来处理的，但对于SAPIC系统，由于它没有私有的APIC总线，因而要求这两方面的支持被编入到固件（firmware）中。即使固件中有了中断转送和负载均衡的功能，Windows也没有使用它们，相反，它采用轮转方式静态地将中断分配给相应的处理器。



实验：查看PIC和APIC

你可以利用内核调试器命令*!pic*或*!apic*，来查看一个单处理器系统上有关PIC的配置情况，或者一个多处理器系统上有关APIC的配置情况（在这个实验中，你不能使用LiveKd，因为LiveKd不能访问硬件）。下面是一个单处理器系统上*!pic*命令的输出结果（请注意，如果你的系统当前正在使用的是一个APIC HAL，则*!pic*命令无法正常工作）。

```
lkd> !pic
----- IRQ Number ----- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
Physically in service:  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
Physically masked:      .  .  .  Y  .  .  Y  Y  .  .  Y  .  .  Y  .  .
Physically requested:   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
Level Triggered:        .  .  .  .  .  Y  .  .  .  Y  .  Y  .  .  .  .
```

以下是一个正在运行MPS HAL的系统上的*!apic*命令的输出。调试器提示符的前缀“0:”表示这些命令运行在0号处理器上，所以这是0号处理器的I/O APIC：

```
lkd> !apic
Apic @ fffe0000  ID:0 (40010)  LogDesc:01000000  DestFmt:ffffffff  TPR 20
TimeCnt: 0bebc200clk  SpurVec:3f  FaultVec:e3  error:0
Ipi Cmd: 0004001f  Vec:1F  FixedDel    Dest=Self      edg  high
Timer..: 000300fd  Vec:FD  FixedDel    Dest=Self      edg  high       masked
Linti0.: 0001003f  Vec:3F  FixedDel    Dest=Self      edg  high       masked
Linti1.: 000184ff  Vec:FF  NMI         Dest=Self      lvl  high       masked
TMR: 61, 82, 91-92, B1
IRR:
ISR:
```


www.TopSage.com

下面是*!ioapic*命令的输出，其中显示了I/O APIC（即连接至设备的中断控制器部件）：

```
0: kd> !ioapic
IoApic @ ffd02000  ID:8 (11)  Arb:0
Inti00.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti01.: 00000962  Vec:62  LowestDl  Lg:03000000  edg
Inti02.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti03.: 00000971  Vec:71  LowestDl  Lg:03000000  edg
Inti04.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti05.: 00000961  Vec:61  LowestDl  Lg:03000000  edg
Inti06.: 00010982  Vec:82  LowestDl  Lg:02000000  edg  masked
Inti07.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti08.: 000008d1  Vec:D1  FixedDel  Lg:01000000  edg
Inti09.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0A.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0B.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0C.: 00000972  Vec:72  LowestDl  Lg:03000000  edg
Inti0D.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0E.: 00000992  Vec:92  LowestDl  Lg:03000000  edg
Inti0F.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti10.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti11.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti12.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti13.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti14.: 0000a9a3  Vec:A3  LowestDl  Lg:03000000  lvl
Inti15.: 0000a993  Vec:93  LowestDl  Lg:03000000  lvl
Inti16.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti17.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
```

软件中断请求级别（IRQL）

虽然中断控制器已经实现了一层中断优先级，但是，Windows仍然强迫使用它自己的中断优先级方案，称为**中断请求级别**（IRQL，*interrupt request level*）。在x86系统中，内核在内部使用从0至31的数值来代表IRQL；而在x64和IA64系统中，内核采用从0至15的数值来代表IRQL。这里数值越大，代表中断的优先级越高。内核为软件中断定义了一组标准的IRQL，而HAL则将硬件中断号映射为IRQL。图3.3给出了针对x86体系结构的IRQL定义，图3.4则给出了针对x64和IA64体系结构的IRQL定义。



> 注　SYNCH_LEVEL，多处理器版本的内核利用该级别来保护针对每个处理器的处理器控制块（PRCB），图中之所以没有显示该级别，是因为它的值会随着Windows版本的不同而有所不同。参见第6章关于SYNCH_LEVEL及其可能值的介绍。

www.TopSage.com

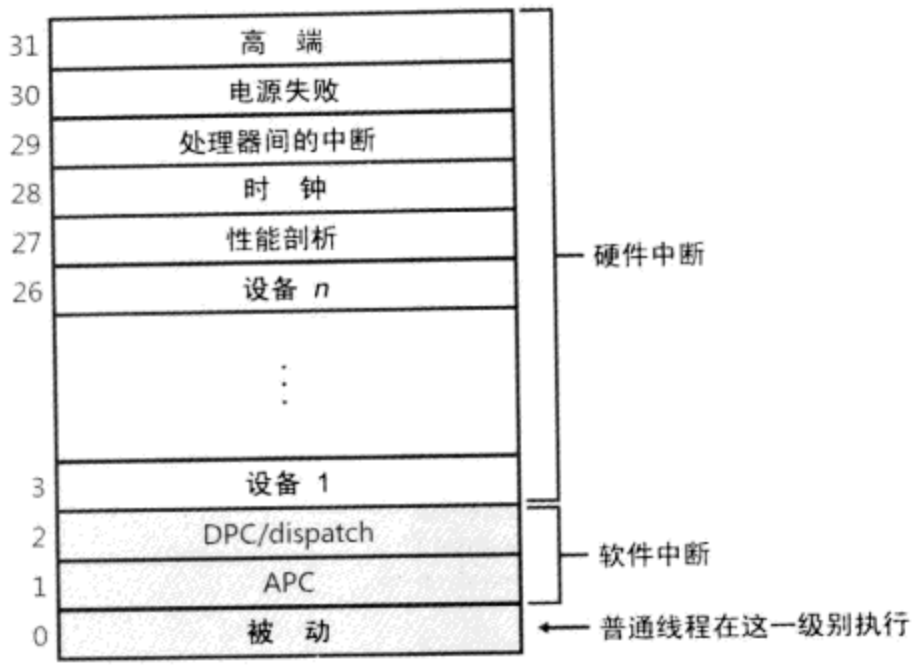


图3.3 x86中断请求级别（IRQL）



图3.4 x64和IA64中断请求级别（IRQL）

中断是按照优先级处理的，高优先级的中断会抢占低优先级中断的执行权。当一个高优先级的中断发生的时候，处理器把被中断的线程的状态保存起来，并调用与该中断关联的陷阱分发器。该陷阱分发器提升IRQL，并调用该中断的服务例程。当服务例程执行完以后，中断分发器再降低处理器的IRQL，使其回到中断发生之前的级别，然后装入被保存起来的机器状态。被中断的线程从原来停止的地方又恢复运行。当内核降低了IRQL的时候，被屏蔽的低优先级中断有可能得以出现。如果这样的话，内核会重复上述过程来处理新的中断。

与线程调度优先级（第6章中讲述）比起来，IRQL优先级别有着完全不同的含义。调度优先级是线程的一个属性，而IRQL则是中断源（比如键盘和鼠标）的一个属性。而且，每个处

www.TopSage.com

理器有一个可随着操作系统代码的执行而变化的IRQL设置。

每个处理器的IRQL设置决定了该处理器可以接收哪些中断。IRQL也被用来实现对一些内核模式数据结构的同步访问（在本章后面你将会看到更多有关同步的信息）。当一个内核模式的线程运行时，它或者通过调用*KeRaiseIrql*和*KeLowerIrql*直接地提升或降低处理器的IRQL，或者，更一般地，通过调用那些获取内核同步对象的函数间接地提升或降低处理器的IRQL。正如图3.5所展示的，如果中断源的IRQL高于处理器当前的级别，则从该源处发生的中断会打断处理器；如果中断源的IRQL等于或者低于当前的级别，则这样的中断会被**屏蔽**（*mask*），直到有一个执行线程降低IRQL级别为止。

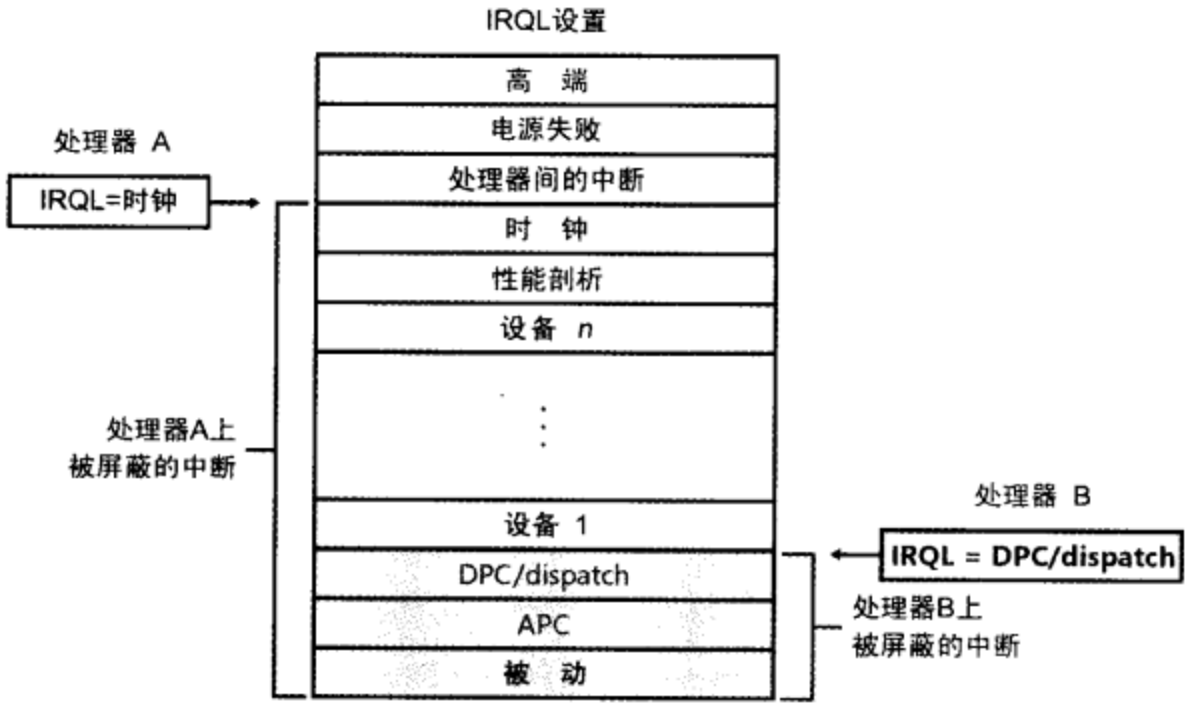


图3.5 屏蔽中断

因为访问PIC是一个相对比较慢的操作，所以，那些使用了PIC的HAL，它们实现了一种性能优化，称为**延迟IRQL**，以避免PIC访问。当IRQL被提升时，HAL在内部记录下新的IRQL，而不是改变中断屏蔽值。如果一个更低优先级的中断随后发生的话，则HAL将中断屏蔽值设置成对于第一个中断较为正确的值，并且将更低优先级的中断延迟至IRQL被降低下来为止。因此，如果当IRQL被提升时没有更低优先级的中断发生，则HAL并不需要修改PIC。

一个内核模式的线程根据它想要做什么事情，来提升或降低它运行所在的处理器的IRQL。例如，当一个中断发生的时候，陷阱处理器（或者，也可能是处理器本身）将该处理器的IRQL提升至中断源的IRQL。这一提升会屏蔽住那些属于该IRQL或者比它低的所有中断（仅仅在同一处理器上），这样可确保处理该中断的处理器不会被同一级别或者更低级别的中断劫掉。被屏蔽的中断或者由另外的处理器来处理，或者被保留着，直到IRQL降下来。

www.TopSage.com

因此，系统的所有组件，包括内核和设备驱动程序，都试图让IRQL保持在被动级别（有时候也称为低级别）上。它们之所以这样做，是因为，如果处理器的IRQL没有被长时间地保持在不必要的高级别上的话，设备驱动程序就可以更加及时地响应硬件中断。



> 注 “提升IRQL会阻塞同一级别或者更低级别中断”这一规则有一个例外跟APC_LEVEL中断有关。如果一个线程将IRQL提升至APC_LEVEL，然后又被重新调度的话，则系统有可能将一个APC_LEVEL的中断递交给新被调度的线程。因此，APC_LEVEL可以看做是线程局部的IRQL，而不是处理器范围内的IRQL。



实验：查看IRQL

如果你在Windows Server 2003上运行内核调试器，就可以利用*!irql*命令查看一个处理器的IRQL：

```
kd> !irql
Debugger saved IRQL for processor 0x0 -- 0 (LOW_LEVEL)
```

注意，在一个称为**处理器控制区**（PCR，*processor control region*）的数据结构以及它的扩展结构**PRCB**（*processor control block*，**处理器控制块**）中有一个称为IRQL的域。这两个数据结构包含了系统中每个处理器的状态信息，比如当前的IRQL、指向硬件IDT的指针、当前正在运行的线程，以及接下来将要选择运行的线程。内核和HAL利用这些信息来执行各种与体系结构相关或者与机器相关的动作。在Windows DDK（Device Driver Kit）头文件Ntddk.h中公开定义了PCR和PRCB的各部分信息，所以，如果你想要了解这些结构的完整定义，可以检查该文件。

在内核调试器中使用*!pcr*命令你可以查看PCR的内容：

```
kd> !pcr
PCR Processor 0 @ffdff000
        NtTib.ExceptionList: f8effc68
            NtTib.StackBase: f8effdf0
           NtTib.StackLimit: f8efd000
         NtTib.SubSystemTib: 00000000
              NtTib.Version: 00000000
          NtTib.UserPointer: 00000000
              NtTib.SelfTib: 7ffde000
```


www.TopSage.com

```
          SelfPcr: ffdff000
             Prcb: ffdff120
             Irql: 00000000
              IRR: 00000000
              IDR: ffff28e8
    InterruptMode: 00000000
              IDT: 80036400
              GDT: 80036000
              TSS: 802b5000

    CurrentThread: 81638020
       NextThread: 00000000
       IdleThread: 8046bdf0
```

不幸的是，在那些不使用延迟IRQL的系统上，Windows并没有维护IRQL域，所以，在大多数系统上该域总是0。

因为改变一个处理器的IRQL对于系统操作有如此重要的影响，所以，这种改变只能在内核模式中进行——用户模式线程不能改变处理器的IRQL。这意味着，当一个处理器正在执行用户模式代码的时候，它的IRQL总是在被动级别上。只有当处理器正在执行内核模式代码的时候，它的IRQL才有可能更高。

每个中断级别都有特定的目的。例如，内核发出一个**处理器间的中断**（IPI，*interprocessor interrupt*），以请求另一个处理器执行一个动作，比如调度一个特定的线程使其执行，或者更新它的地址转译快查缓冲区（translation look-aside buffer）的高速缓存。系统时钟按照规则的间隔产生中断，内核响应时钟中断的方式是，更新系统时钟并测量线程的执行时间。如果硬件平台支持两个时钟，则内核加入另一个时钟中断级别来测量性能。HAL提供了许多中断级别以用于中断驱动的设备；中断级别的确切数目随着处理器和系统配置的不同而有所不同。内核使用软件中断（本章后文将要讲述）来激发线程调度，以及以异步方式打断一个线程的执行。

将中断映射到IRQL IRQL级别与中断控制器定义的中断请求（IRQ）并不相同——Windows所在的体系结构并没有在硬件上实现IRQL的概念。那么，Windows如何决定为一个中断分配哪个IRQL呢？答案在于HAL中。在Windows中，一种称为总线型驱动程序的设备驱动程序确定它的总线（PCI、USB等等）上有哪些设备，以及哪些中断可以分配给某一个设备。总线型驱动程序将这些信息汇报给即插即用管理器，即插即用管理器在考虑了所有其他设备的可接受的中断分配方案以后，确定为每个设备分配哪个中断。然后，它调用HAL函数*HalpGetSystemInterruptVector*，该函数将中断映射到对应的IRQL。

www.TopSage.com

对于Windows所包含的各种HAL，其中断分配算法也有所不同。在单处理器x86系统上，HAL执行的是直接的翻译：对于一个给定的中断向量，用27减去中断向量值，就是它的IRQL。因此，如果一个设备使用了中断向量值5，则它的ISR在IRQL 22上执行。在一个x86多处理器系统上，映射关系没这么简单。APIC支持不止200个中断向量，所以，没有那么多的IRQL可以一一对应。所以，多处理器HAL按照循环方式，在设备IRQL（DIRQL）的范围内循环利用，将IRQL分配给中断向量。结果是，在一个x86多处理器系统上，你很难找到合适的方法来预测或者知道Windows将哪个IRQL分配给某个APIC IRQ。最后，在x64和IA64系统上，对于一个给定的IRQ，HAL将该IRQ对应的中断向量除以16，作为它的IRQL。

预定义的IRQL 接下来我们更进一步看一看预定义的IRQL的使用情况。从图3.5中显示的最高级别开始，它的使用情况如下。

- 只有当内核在*KeBugCheckEx*中停止了系统并屏蔽了所有中断的时候，内核才会使用高级别的IRQL。
- **电源失败**级别最初出现在Microsoft Windows NT的原始设计文档中，该文档指定了系统电源失败代码的行为，但是此IRQL从来没有被真正用到过。
- **处理器间的中断**级别被用于向另一个处理器请求执行一个动作，比如将一个DISPATCH_LEVEL中断排到队列中以便为它调度一个特定的线程来执行其任务、更新处理器的地址转换快查缓冲区（TLB，translation look-aside buffer）的高速缓存、系统关闭或者系统崩溃。
- **时钟**级别主要用于系统的时钟，内核利用该中断级别来跟踪具体的时刻，以及为线程测量或分配CPU时间。
- 当内核的性能剖析功能（即一种性能测量机制）被打开的时候，系统的实时时钟就会用到**性能剖析**（*profile*）级别。当内核的性能剖析功能被激活时，内核的性能剖析陷阱处理器就会记录下中断发生时被执行的代码的地址。随着时间的推移，一张地址采样表建立起来，因而有关的工具就可以提取相应的信息并进行分析。你可以从*http://www.microsoft.com/whdc/system/sysperf/krview.mspx*下载一个称为Kernrate的内核性能剖析工具，并用它来配置和查看有关性能方面的统计量。关于使用此工具的更多信息，请参见下面的Kernrate实验。
- **设备**IRQL被用来对设备中断进行优先级区分（关于如何将硬件中断映射到IRQL上，请参见上一部分）。
- *DPC/Dispatch*级别和*APC*级别的中断是由内核和设备驱动程序产生的软件中断（本章后面将会详细解释DPC和APC）。
- 最低的IRQL为**被动**级别，它实际上根本不是一个中断级别；它是普通线程运行时的设置，此时所有的中断都允许发生。

www.TopSage.com



实验：使用内核性能剖析程序来分析性能

你可以使用内核性能剖析工具来打开系统的性能剖析定时器，并且将定时器激发时刻所执行的代码采样收集起来，然后显示一个摘要，说明各个映像文件和函数的频率分布。它也可用来跟踪单个进程所消耗的CPU比率，以及内核模式下独立于进程的时间消耗（例如，中断服务例程）。当你想要得到系统时间消耗的明细表时，内核性能剖析是非常有用的。

在Kernrate最简单的形式中，它对每个内核模块（比如Ntoskrnl、驱动程序等等）所花时间的地方进行采样。例如，在安装了先前提到过的Krview软件包以后，试着执行下面的步骤。

1. 打开命令提示窗口。
2. 输入**cd c:\program files\krview\kernrates**。
3. 输入**dir**（你可以看到针对每个平台的Kernrate映像文件）。
4. 运行符合你的平台的映像文件（不需要带命令行参数或者开关）。例如，Kernrate_i386_XP.exe 是针对Windows XP的x86系统的映像文件。
5. 当Kernrate正在运行的时候，在系统上执行一些其他的行为。例如，运行Windows媒体播放器并播放一些音乐，运行一个图形密集型的游戏，或者执行网络操作，比如连接一个远程共享目录。

按下Ctrl+C停止Kernrate。这会导致Kernrate显示出它在采样过程中获得的统计信息。

在以下示例性的Kernrate输出中，Windows媒体播放器正在运行，它在播放一张CD上的一首乐曲。

```
C:\Program Files\KrView\Kernrates>Kernrate_i386_XP.exe
 /============================\
<         KERNRATE LOG         >
 \============================/
Date: 2004/05/13   Time:  9:48:28
Machine Name: BIGDAVID
Number of Processors: 1
PROCESSOR_ARCHITECTURE: x86
PROCESSOR_LEVEL: 6

Kernrate User-Specified Command Line:
Kernrate_i386_XP.exe

***> Press ctrl-c to finish collecting profile data
===> Finished Collecting Data, Starting to Process Results

------------Overall Summary:--------------
P0      K 0:00:03.234 (11.7%)  U 0:00:08.352 (30.2%)  I 0:00:16.093 (58.1%)
 DPC 0:00:01.772 ( 6.4%)  Interrupt 0:00:00.350 ( 1.3%)
```


www.TopSage.com

```
     Interrupts= 52899, Interrupt Rate= 1911/sec.Time   7315 hits, 19531 events per hit ---
-----
 Module                                    Hits   msec  %Total  Events/Sec
gv3                                        4735     27679    64 %     3341135
smwdm                                       872     27679    11 %      615305
win32k                                      764     27679    10 %      539097
ntoskrnl                                    739     27679    10 %      521457
hal                                         124     27679     1 %       87497
```

总摘要部分表明了，系统在内核模式下花了11.7%的时间，在用户模式下花了30.2%的时间，剩下的58.1%空闲时间，在DPC上花了6.4%的时间，在中断上花了1.3%的时间。碰撞率最高的模块是GV3.SYS，这是Pentium M Geyserville族的处理器驱动程序。它用于性能信息采集，这也正是为什么它出现在首位的原因。碰撞率第二高的模块是Smwdm.sys，这是测试机器上声卡的音频驱动程序。这是有道理的，因为该系统上正在进行的大多数活动是Windows媒体播放器将声音I/O发送至声音驱动程序。

如果你的系统上的符号信息可以使用的话，你可以在单独的模块上进一步放大，根据函数名可以看出每个函数的时间花销。例如，如果在屏幕上快速地拖动一个窗口，同时对系统进行性能剖析的话，可以导致以下输出（部分）：

```
C:\Program Files\KrView\Kernrates>Kernrate_i386_XP.exe -z ntoskrnl -z win32k
 /============================\
<         KERNRATE LOG         >
 \============================/
Date: 2004/05/13   Time: 10:26:55

Time   4087 hits, 19531 events per hit --------
 Module                                    Hits   msec  %Total  Events/Sec
win32k                                     1649     10424    40 %     3089660
ati2dvag                                   1269     10424    31 %     2377670
ntoskrnl                                    794     10424    19 %     1487683
gv3                                         162     10424     3 %      303532

----- Zoomed module win32k.sys (Bucket size = 16 bytes, Rounding Down) ------- Module
                          Hits   msec  %Total  Events/Sec
EngPaint                                    328     10424    19 %      614559
EngLpkInstalled                             302     10424    18 %      565844

----- Zoomed module ntoskrnl.exe (Bucket size = 16 bytes, Rounding Down) -----
Module                                   Hits   msec  %Total  Events/Sec
KiDispatchInterrupt                       243     10424    26 %      455298
ZwYieldExecution                           50     10424     5 %       93682
InterlockedDecrement                       39     10424     4 %       73072
```

碰撞率最高的模块是Win32k.sys，这是窗口系统的驱动程序。列表中碰撞率第二高的模块是视频驱动程序。这些结果是合理的，因为该系统中主要的活动是在屏幕上绘图。请注意，在Win32k.sys的放大显示部分中，碰撞率最高的函数是EndPaint，该函数是在屏幕上绘图时的主要GDI函数。

www.TopSage.com

对于运行在DPC/Dispatch级别或更高级别上的代码，一个重要的限制是它不能等待一个对象，如果它这么做的话，将迫使调度器选择另一个线程来执行，而这是一个非法的操作，因为调度器在DPC/Dispatch级别上和它的数据结构同步，因此它不能被调用来执行重新调度的操作。另一个限制是，在DPC/Dispatch级别或者更高级别的IRQL上，只能访问非换页的内存。这条规则实际上是第一个限制的副作用，这是因为，试图访问未驻留的内存将会导致一个页面错误。当页面错误发生的时候，内存管理器激发一个磁盘I/O，然后需要等待文件系统驱动程序从磁盘上读入该页面。这一等待又依次要求调度器执行一个环境切换（如果没有用户线程在等待运行的话，则可能切换到空闲线程）。因此违反了此时调度器不能被调用的规则（因为在磁盘读操作的时候，IRQL仍然在DPC/Dispatch或更高的级别上）。如果这两个限制被违反了，系统就会崩溃，并且崩溃代码为IRQL_NOT_LESS_OR_EQUAL（参见第4章关于系统崩溃的完整讨论）。在设备驱动程序中，违反这些限制是一种常见的错误。在第7章的“驱动程序检验器”一节中将要介绍的Windows驱动程序检验器有一个选项，你设置了该选项之后可以帮助找到这种特定类型的错误。

中断对象 内核提供了一种可移植的机制使得设备驱动程序可以为它们的设备注册ISR，这是一个称为中断对象的内核控制对象。中断对象中包含了所有“供内核将一个设备的ISR与一个特定级别的中断关联起来而需要”的信息，包括该ISR的地址、该设备中断时所在的IRQL级别，以及内核中与该ISR关联的IDT项。当一个中断对象被初始化的时候，少量的汇编语言代码指令（称为分发代码）被从一个中断处理模板*KiInterruptTemplate*中拷贝过来，保存在该对象中。当中断发生的时候，这些代码将被执行。

这份驻留在中断对象中的代码调用了实际的中断分发器，通常是内核的*KiInterruptDispatch*或*KiChainedDispatch*例程，并且将指向该中断对象的指针传递给它。*KiInterruptDispatch*这个例程只适用于那些仅注册了一个中断对象的中断向量，而*KiChainedDispatch*用于那些在多个中断对象之间共享的中断向量。中断对象中所包含的信息使得这第二个分发器例程能够找到并正确地调用设备驱动程序所提供的ISR。中断对象中也包含了与该中断关联的IRQL，所以*KiChainedDispatch*或者*KiChainedDispatch*在调用ISR之前将IRQL提升到正确的级别上，在ISR返回之后再降低IRQL。此两步骤过程是必要的，因为初始分发是由硬件来完成的，所以，在初始分发时是没有办法传递一个指向中断对象的指针的（或者任何此类的其他参数）。在一个多处理器系统中，内核为每个CPU分配一个中断对象，并对它进行初始化，使该CPU上的本地APIC能够接受特定的中断。图3.6显示了那些与中断对象关联在一起的典型的中断控制流。

www.TopSage.com

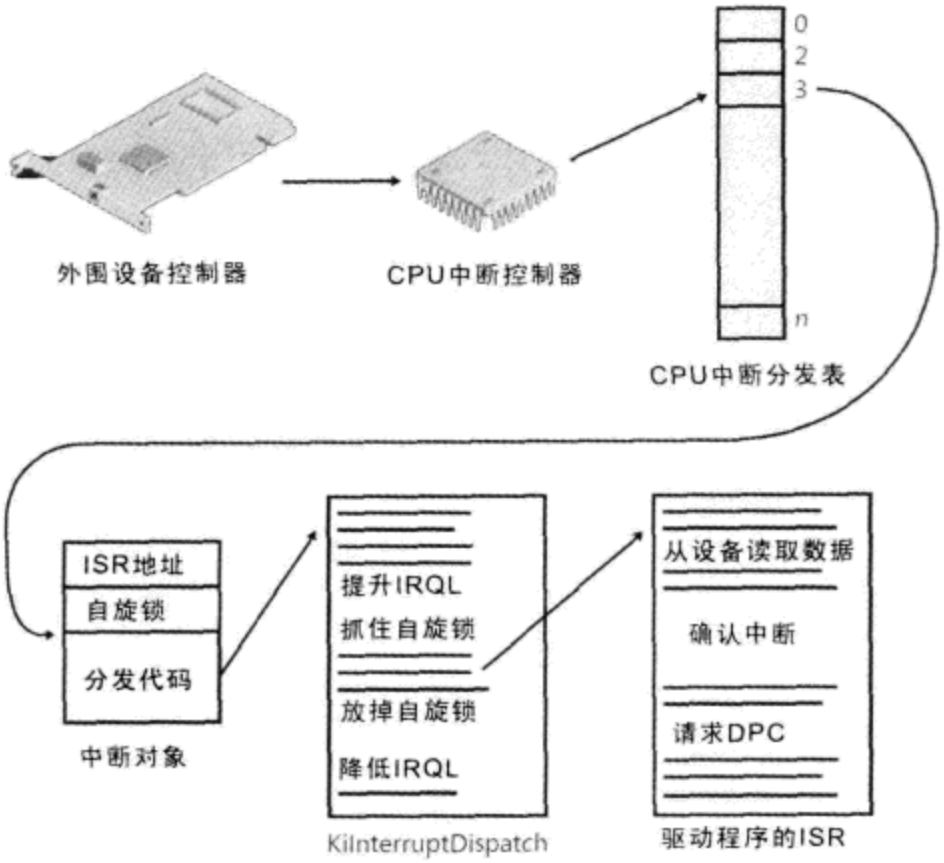


图3.6 典型的中断控制流

实验：检查中断内部

利用内核调试器，你可以查看一个中断对象的细节，包括它的IRQL、ISR地址以及自定义的中断分发代码。首先，执行*!idt*命令，并找到包含有*I8042KeyboardInterruptService*的那一条，这里*I8042KeyboardInterruptService*是针对PS2键盘设备的ISR例程：

```
31:      8a39dc3c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8a39dc00)
```

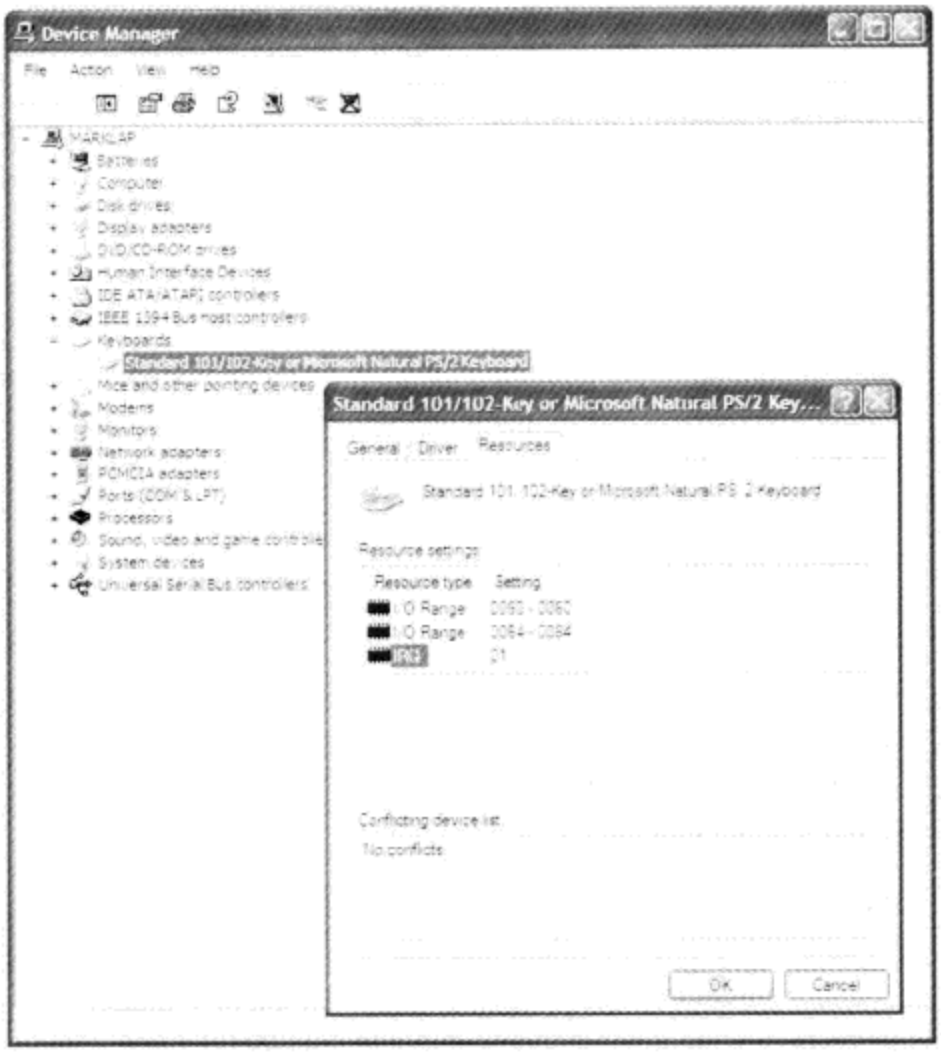
为了检查与该中断关联的中断对象的内容，请执行**dt nt!_kinterrupt**命令，并跟上KINTERRUPT后面的地址：

```
kd> dt nt!_kinterrupt 8a39dc00
nt!_KINTERRUPT
   +0x000 Type             : 22
   +0x002 Size             : 484
   +0x004 InterruptListEntry : _LIST_ENTRY [ 0x8a39dc04 - 0x8a39dc04 ]
   +0x00c ServiceRoutine   : 0xba7e74a2     i8042prt!I8042KeyboardInterruptService+0
   +0x010 ServiceContext   : 0x8a067898
   +0x014 SpinLock         : 0
   +0x018 TickCount        : 0xffffffff
   +0x01c ActualLock       : 0x8a067958  -> 0
   +0x020 DispatchAddress  : 0x80531140     nt!KiInterruptDispatch+0
   +0x024 Vector           : 0x31
   +0x028 Irql             : 0x1a ''
   +0x029 SynchronizeIrql  : 0x1a ''
   +0x02a FloatingSave     : 0 ''
```


www.TopSage.com

```
+0x02b Connected          : 0x1 ''
+0x02c Number             : 0 ''
+0x02d ShareVector        : 0 ''
+0x030 Mode               : 1 ( Latched )
+0x034 ServiceCount       : 0
+0x038 DispatchCount      : 0xffffffff
+0x03c DispatchCode       : [106] 0x56535554
```

在这个例子中，Windows分配给该中断的IRQL是0x1a（即十进制的26）。因为上面的实验是在一台单处理器的系统上进行的，所以我们计算该IRQ是1，因为在x86单处理器系统上，IRQL是从27减去IRQ而得到的。要想验证这一点，只要打开Device Manager（设备管理器，在控制面板的System小程序的Hardware视图上），找到PS/2键盘设备，检查它的资源分配情况，如下图所示。



在一台多处理器x86系统上，IRQ从本质上来说是随机分配的；而在x64或者IA64系统上，你将会看到，该IRQ是用16除中断向量号（在这个例子中是0x31，即十进制49）。

该中断对象的ISR的地址保存在ServiceRoutine域中（正如在*!idt*的输出中所显示的），当中断发生时，实际执行的中断代码被存放在该中断对象末尾的*DispatchCode*数组中。这里存储的中断代码被编写成：首先在栈中建立起陷阱帧，然后调用在DispatchAddress域中存储的函数（在这个例子中是*KiInterruptDispatch*），并且将指向该中断对象的指针传递给它。

www.TopSage.com

Windows和实时处理过程

对时间期限的需求，无论是硬件的还是软件的，都体现了实时环境的特点。硬实时系统（例如核电站控制系统）中有一些时间限制是系统必须要满足的，从而可以避免诸如财产或者生命损失之类的灾难性失败。在软实时系统（例如汽车的燃料节约优化系统）中，有一些时间限制尽管是可以错过的，但及时性仍然是一个期望的特征。在实时系统中，计算机有传感器输入设备和控制输出设备。一个实时计算机系统的设计者必须要清楚，从一个输入设备产生中断的时间点，到该设备的驱动程序能够控制输出设备来响应此中断的时间点之间，在最差情形下其延迟该是多少。在进行最差情形分析的时候，必须要考虑操作系统引入的延迟，以及应用程序和设备驱动程序所强加的延迟。

因为Windows并没有以任何一种可控的方式对设备IRQ进行优先级处理，而且用户应用程序只能在处理器的IRQL为被动级别的时候才可以执行，所以，Windows并不总是适合于作为一个实时操作系统。最终决定“最差情形延迟”的是系统的设备和设备驱动程序而并非Windows。当实时系统的设计者使用市场上的硬件产品时，这就成了一个问题。设计者难以确定市场上每一个设备的ISR或者DPC在最差情形下需要多长时间。即使经过了测试，设计者也不能保证实际系统在特殊情形下不会漏掉某个重要的时间期限。而且，系统的DPC和ISR可能引入的所有延迟的总和通常会远远超过一个时间敏感系统的容忍度。

虽然许多类型的嵌入系统（比如，打印机和汽车传动计算机）有实时性的需求，但是，Windows XP Embedded并没有实时特性。它只是Windows XP的一个版本，其中用到了Microsoft从VenturCom得到许可的系统设计技术，从而有可能产生出这样一个适合于在资源受限的设备上运行的Windows XP小版本。例如，一个没有联网功能的设备将会省略掉所有与网络有关的Windows XP组件，包括网络管理工具，以及适配器和协议栈设备驱动程序。

尽管如此，有一些第三方厂商为Windows提供了实时内核。这些厂商所使用的办法是，将它们的实时内核嵌在一个自定义的HAL中，并且将Windows当做实时操作系统中的一个任务来运行。这一运行Windows的任务正好可作为系统的用户界面，而且该任务的优先级比其他负责管理此设备的任务要低。关于Windows第三方实时内核扩展的例子，请参考VenturCom的Web站点*www.venturcom.com*。

将一个ISR与某个特定的中断级别关联起来，这一操作称为**连接一个中断对象**，而将一个ISR与IDT项断开关联，则称为**断开一个中断对象**。这些操作是通过调用内核函数*IoConnectInterrupt*

www.TopSage.com

和*IoDisconnectInterrupt*来完成的，正是这两个操作，使得设备驱动程序可以在加载到系统中的时候“打开”一个ISR，而如果驱动程序被卸载的话，则可以“关闭”该ISR。

使用中断对象来注册一个ISR，可以避免设备驱动程序直接操纵中断硬件（在不同的处理器体系结构之间中断硬件也有所不同），也可以不必知道任何有关该IDT的细节。这一内核特性有助于创建可移植的设备驱动程序，因为它使得开发人员无须使用汇编语言来编码，也无须在设备驱动程序中反映出处理器的差异来。

中断对象还提供了其他一些好处。通过使用中断对象，内核可以将该ISR的执行过程与设备驱动程序中其他可能与ISR共享数据的部分同步起来（有关设备驱动程序如何响应中断的更多信息，请参见第9章）。

而且，中断对象使得内核很容易为任何一个中断级别调用多个ISR。如果多个设备驱动程序创建了中断对象，并且将它们连接到同一个IDT项上，那么，在指定的中断线上发生中断的时候，中断分发器会调用每一个例程。这种能力使得内核很容易支持“稚菊链”配置，即几个设备共享同样的中断线。当某一个ISR通过给中断分发器返回一个状态，来声明自己拥有该中断的时候，这条链就被打破了。如果共享同一中断的多个设备同时请求服务，那么，有些设备，由于其ISR没有应答，一旦中断分发器降低了IRQL，它们将会再次中断系统。只有当所有想要使用同一中断的设备驱动程序向内核表示它们可以共享该中断时，这种链式结构才是允许的；如果它们未能向内核做出这种表态，则即插即用管理器会重新组织它们的中断分配方案，以确保尊重每一个设备的共享需求。如果中断向量被共享的话，中断对象会调用*KiChainedDispatch*，该函数依次会调用每一个注册的中断对象的ISR，直至它们中的某一个中断对象声明拥有该对象，或者所有的ISR都已经被执行为止。在前面的*!idt*示例输出中，向量0x3b被连接到几个链式的中断对象上。

软件中断

虽然大多数中断都是硬件产生的，但是，Windows内核也为各种各样的任务产生软件中断，这样的任务包括：

- 激发线程分发；
- 并非时间紧急的中断处理；
- 处理定时器到期；
- 在特定线程的环境中异步执行一个过程；
- 支持异步I/O操作。

这些任务将在下面的小节中进一步讲述。

www.TopSage.com

分发或者延迟过程调用（DPC）中断 当一个线程不能继续运行的时候，比如因为线程已经终止了或者它主动进入到等待状态，内核就会直接调用分发器，从而立即导致一个环境切换。然而，有时候，内核检测到它已经深入到许多层代码中了，这时应该进行重新调度。在这种情况下，内核请求分发操作，但将它推迟到完成了当前的行为以后再进行。使用DPC软件中断是实现这种延迟的一种便捷途径。

当内核需要对共享的内核数据结构的访问进行同步的时候，它总是将处理器的IRQL提升到DPC/Dispatch级别或更高。这禁止了另外的软件中断和线程分发动作。当内核检测到应该进行分发的时候，它请求一个DPC/Dispatch级别的中断；但是，因为IRQL是在此级别或更高级别上，所以处理器保留住该中断。当内核完成了它当前的活动时，它知道要把IRQL降低到DPC/Dispatch级别以下，并且查看一下是否有分发中断正在等待处理。如果有的话，将IRQL降低到DPC/Dispatch级别上，于是这些分发中断就可以得到处理了。利用软件中断来激活线程分发器，这是一种将分发过程推迟到条件合适时才进行的方法。然而，Windows也使用软件中断来推迟其他类型的处理过程。

除了线程分发以外，内核也在此IRQL上处理延迟的过程调用（DPC）。DPC是完成某一项系统任务的一个函数——完成这一系统任务的时间不像完成当前任务那样紧迫。这些函数之所以称为**延迟的**，是因为它们可能不会立即执行。

DPC赋予操作系统这样一种能力：产生一个中断并且在内核模式下执行一个系统函数。内核利用DPC来处理定时器到期（并解除那些正在等待定时器的线程），以及当一个线程的时限到期以后重新调度处理器。设备驱动程序利用DPC来完成I/O请求。为了给硬件中断提供及时的服务，Windows——在设备驱动程序的配合下——企图将IRQL保持在低于设备IRQL级别之下。达到这一目标的一种方法是，让设备驱动程序ISR执行最少最必要的工作来响应它们的设备、将易变的中断状态保存起来，并且将数据传输或者其他并非时间紧迫的中断处理活动推迟到一个位于DPC/Dispatch IRQL级别上的DPC中再执行（有关DPC和I/O系统的更多信息，请参见第9章）。

DPC是通过一个**DPC对象**来表示的。DPC对象是一种内核控制对象，它对于用户模式程序是不可见的，但是对于设备驱动程序和其他的系统代码则是可见的。DPC对象所包含的最重要的信息是，当内核处理该DPC中断时将要调用的那个系统函数的地址。正在等待执行的DPC例程被存储在由内核管理的队列中，每个处理器都有一个这样的队列，称为**DPC队列**。要想请求一个DPC，系统代码调用内核来初始化一个DPC对象，然后把它放在一个DPC队列中。

www.TopSage.com

在默认情况下，内核把DPC对象放在发出该DPC请求的处理器（通常ISR在该处理器上执行）的DPC队列末尾。然而，设备驱动程序只需指定一个DPC优先级（低级、中级和高级，默认是中级）并且将该DPC定在特定的处理器上，就可以改变这种默认的行为。目标定在某个特定CPU上的DPC被认为是**定向的DPC**（*targeted DPC*）。如果一个DPC的优先级是低级或者中级，则内核把该DPC放在队列的末尾；如果DPC是高优先级，则内核将该DPC对象插在队列的前端。

当处理器的IRQL将要从DPC/Dispatch或者更高级的IRQL降低到某个更低级的IRQL（APC或者被动级别）时，内核就会处理DPC。Windows确保该IRQL仍然在DPC/Dispatch级别上，并且将当前处理器的队列中的DPC对象拉出来，一直到队列为空（也就是说，内核“抽干”该队列），同时依次调用每一个DPC函数。只有当队列为空的时候，内核才让IRQL降低到DPC/Dispatch级别以下，让正常的线程执行过程继续进行。DPC的处理过程如图3.7所描述的那样。

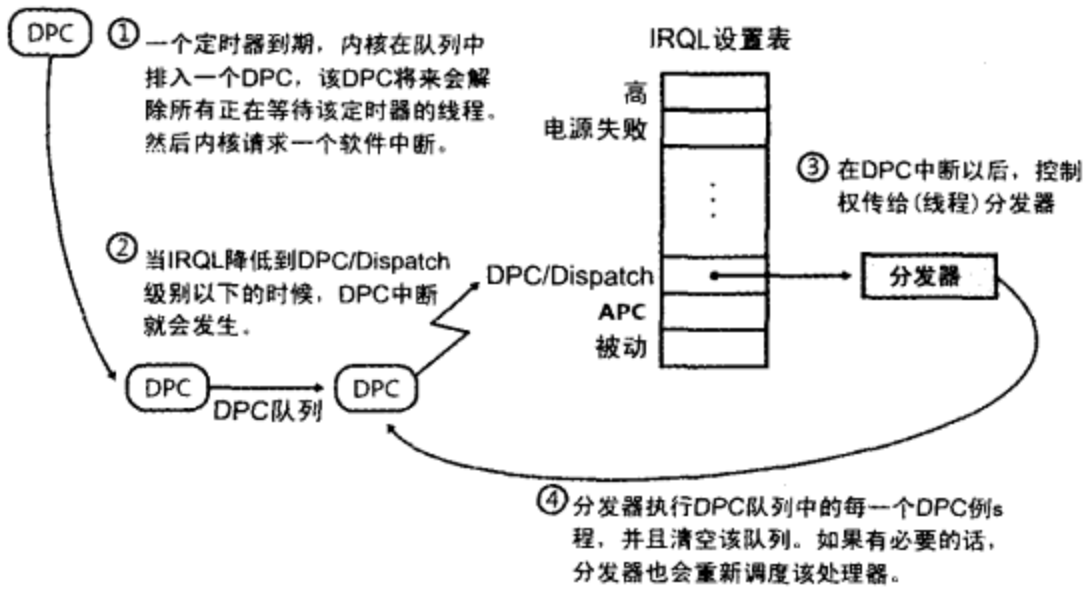


图3.7 交付一个DPC

DPC优先级可以以另一种方式影响到系统的行为。内核通常通过一个DPC/Dispatch级别的中断来激发DPC队列的“抽干”动作。只有当一个DPC被定位在ISR所在的处理器上，并且该DPC的优先级是高级或者中级时，内核才会产生这样一个中断。如果该DPC的优先级是低级，则只有当针对该处理器并且尚未处理的DPC请求的数量达到一定的阈值，或者在一个时间窗口内该处理器上的DPC请求的数量很少的时候，内核才会请求中断。如果一个DPC被定位在一个不同于其ISR运行时所在的CPU之上，并且该DPC的优先级是高级，那么，内核立即用一个信号通知目标CPU（其做法是，给它发送一个分发IPI），以便“抽干”它的DPC队列。如果优先级为中级或者低级，则目标处理器上排队的DPC的数量必须超过一定的阈值，这样内核才会激发一个DPC/Dispatch中断。系统空闲线程也会为它所在的处理器，抽干DPC队列。尽管DPC的目标定位和优先级别是灵活的，但设备驱动程序很少需要改变其DPC对象的默认行为。表3.1总结了几种激发DPC队列“抽干”动作的情形。

www.TopSage.com

表3.1 DPC中断产生规则

DPC优先级	DPC被定位在ISR的处理器上	DPC被定位在另一个处理器上
低级	DPC队列长度超过最大的DPC队列长度值，或者DPC请求率小于最小的DPC请求率	DPC队列长度超过最大的DPC队列长度值，或者系统空闲
中级	总是激发	DPC队列长度超过最大的DPC队列长度值，或者系统空闲
高级	总是激发	总是激发

因为用户模式线程在低的IRQL上执行，所以对于一个DPC来说，中断一个普通用户的线程的执行过程是很好的机会。DPC例程在执行的时候根本不考虑当前哪个线程正在运行，也即意味着，当一个DPC例程运行的时候，它不会假设自己当前被映射到哪个进程地址空间中。DPC例程可以调用内核函数，但是它们不能调用系统服务、产生页面错误，或者创建或等待分发器对象（本章后文将会进一步解释）。然而，它们可以访问非换页的系统内存地址，因为系统地址空间总是被映射的，不管当前是哪个进程。

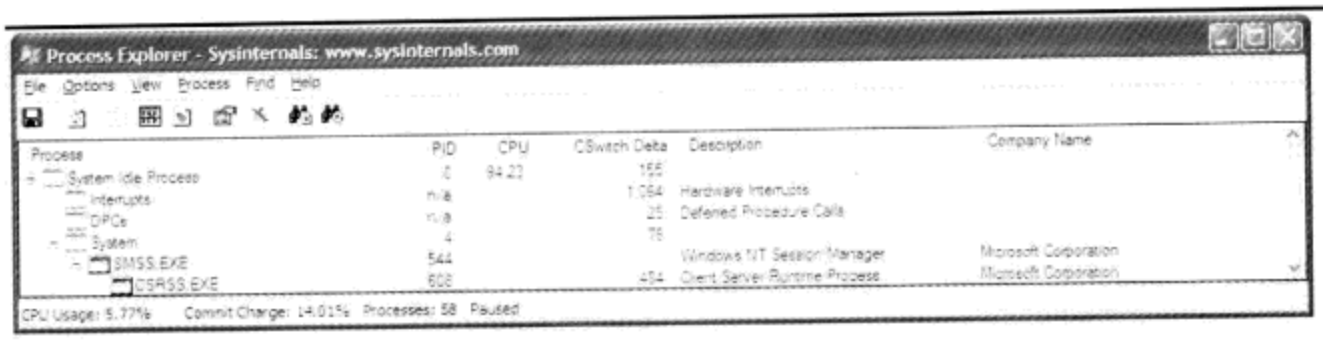
DPC主要是为设备驱动程序而提供的，但是内核也使用DPC。内核最经常使用一个DPC来处理时限到期事件。在系统时钟的每一个嘀嗒点上，就会发生一个时钟IRQL级别的中断。时钟中断处理器（运行在时钟IRQL级别上）对系统时间进行更新，然后将一个记录了当前线程已运行多长时间的计数器进行递减。当该计数器减到0的时候，该线程的时限到期，内核可能需要重新调度该处理器，这是一个更低优先级的任务，应该在DPC/Dispatch IRQL级别上完成。时钟中断处理器将一个DPC插入到队列中以便激发线程分发过程，然后结束它的工作并且降低该处理器的IRQL。因为DPC中断的优先级低于设备中断，所以，任何在时钟中断完成之前出现并且尚未处理的设备中断，都将在DPC中断发生之前被处理。



实验：监视中断和DPC活动

你可以使用进程管理器来监视中断和DPC活动，做法很简单，只需加入“Context Switch Delta”列，并且观察Interrupts和DPCs进程。它们并不是真正的进程，只是为方便起见而将它们显示成进程，因此它们并不会招致运行环境切换。进程管理器中针对这些伪进程的环境切换计数器反映了在上一个刷新周期内，中断或者DPC的发生次数。你只要在屏幕上快速地移动鼠标，就可以显著地激发中断和DPC。

www.TopSage.com



利用Windows XP SP2和Windows Server 2003 SP1及其以后版本中内置的事件追踪支持功能（本章后面将会介绍），你还可以追踪特定的中断服务例程和延迟的过程调用的执行情况。

1. 通过输入下面的命令，你就可以启动事件捕捉功能：

```
tracelog -start -f kernel.etl -b 64 -UsePerfCounter -
eflag 8 0x307 0x4084 0 0 0 0 0 0
```

2. 要停止捕捉事件，只需输入：

```
tracelog -stop to stop logging.
```

3. 要为事件捕捉过程生成报告，可以输入：

```
tracerpt kernel.etl -df -o -report
```

4. 这将会生成两个文件：workload.txt和dumpfile.csv。

5. 打开“workload.txt”，你将会看到在每个驱动程序类型的ISR和DPC上所花时间的摘要信息。 打开第3步中生成的文件“dumpfile.csv”；搜索一行内第二个值为“DPC”或“ISR”的行。例如，利用上述命令生成的dumpfile.csv文件中，下面三行显示了一个定时器DPC、一个DPC和一个ISR：

```
    PerfInfo,   TimerDPC, 0xFFFFFFFF,   127383953645422825,          0,          0, 1273
83953645421500, 0xFB03A385, 0, 0
    PerfInfo,        DPC, 0xFFFFFFFF,   127383953645424040,          0,          0, 1273
83953645421394, 0x804DC87D, 0, 0
    PerfInfo,        ISR, 0xFFFFFFFF,   127383953645470903,          0,          0, 1273
83953645468696, 0xFB48D5E0, 0, 0, 0
```

在内核调试器中，针对每个事件记录中的开始地址（上面每一行的第8个值）执行一个“ln”命令，就会显示出执行相应DPC或ISR的函数的名字：

```
lkd> ln 0xFB03A385
(fb03a385)   rdbss!RxTimerDispatch   |  (fb03a41e)   rdbss!RxpWorkerThreadDispatcher
lkd> ln 0x804DC87D
(804dc87d)   nt!KiTimerExpiration   |  (804dc93b)   nt!KeSetTimerEx
lkd> ln 0xFB48D5E0
(fb48d5e0)   atapi!IdePortInterrupt   |  (fb48d622)   atapi!IdeCheckEmptyChannel
```

第一个是文件系统重定向器客户驱动程序插入队列的一个定时器到期DPC；第二个是一个通用的定时器到期的DPC；第三个地址是ATAPI端口驱动程序的ISR的地址。有关更多的信息，请参见*http://www.microsoft.com/whdc/driver/perform/mmdrv.mspx*。

www.TopSage.com

异步过程调用（APC）中断 异步过程调用（APC）提供了一种在特定用户线程环境（从而也是在特定进程的地址空间）中执行用户程序和系统代码的途径。因为APC是经过排队的，以便在特定线程的环境中执行，并且运行在低于DPC/Dispatch级别的IRQL上，所以，它们并没有像DPC那样的限制。APC例程可以访问资源（对象）、等待对象句柄、引发页面错误，以及调用系统服务。

APC是由一个内核控制对象来描述的，称为**APC对象**。正在等待执行的APC驻留在一个由内核管理的**APC队列**中。DPC队列是系统范围的，与此不同的是，APC队列是与特定线程相关的，即每个线程有它自己的APC队列。当内核接到请求，要将一个APC排队时，它将该APC插入到将来执行此APC例程的那个线程的队列中。然后，内核请求一个APC级别的软件中断，当该线程最终开始运行的时候，它就会执行此APC。

有两种APC类型：内核模式和用户模式。内核模式的APC并不要求从目标线程获得“许可”就可以运行在该线程的环境中，而用户模式的APC必须先获得许可才可以。内核模式的APC无须目标线程的干涉或者同意，就可以中断该线程并执行一个过程。内核模式的APC也有两种类型：普通的和特殊的。一个线程只需将IRQL提升到APC_LEVEL或者调用*KeEnterGuardedRegion*（在Windows Server 2003中引入的函数），就可以禁止这两种类型的内核模式APC。*KeEnterGuardedRegion*通过设置调用线程的KTHREAD结构（在第6章中进一步讲述）中的SpecialApcDisable域，来禁止APC被交付。一个线程要想禁止普通类型的APC，惟一的办法是调用*KeEnterCriticalRegion*，它会设置该线程的KTHREAD结构中的KernelApcDisable域。

执行体使用内核模式的APC来完成那些必须要在特定线程的地址空间（执行环境）中才能完成的操作系统任务。它可以利用特殊的内核模式APC来指示某个线程停止执行一个可中断的系统服务，比如，将一个异步I/O操作的结果记录在线程的地址空间中。环境子系统利用特殊的内核模式APC，让一个线程挂起或终止自身，或者获取或设置它的用户模式执行环境。POSIX子系统利用内核模式APC来模仿POSIX信号被递交给POSIX进程的过程。

设备驱动程序也可以使用内核模式APC。例如，如果有一个I/O操作被激发了，线程进入等待状态，那么，另一个进程中的另一个线程就会被调度到运行状态。当设备完成了数据传输工作时，I/O系统必定以某种方式回到原先激发I/O操作的线程的环境中，所以它可以将I/O操作的结果拷贝到包含该线程的进程地址空间中的缓冲区中。I/O系统利用一个特殊的内核模式APC来完成这一任务（第9章将会更加详细地讨论I/O系统中APC的这种用法）。

www.TopSage.com

有几个Windows API，比如*ReadFileEx*、*WriteFileEx*和*QueueUserAPC*，也用到了用户模式APC。例如，*ReadFileEx*和*WriteFileEx*函数允许调用者指定一个完成例程，当I/O操作完成的时候该完成例程就会被调用。I/O完成机制是通过在发起I/O的线程里插入一个APC来实现的。然而，完成例程的回调动作并不一定发生在APC被插入队列的时候，因为只有当一个线程处于**可警告的等待状态**（*alertable wait state*）时，用户模式APC才可以交付给该线程。一个线程可以通过以下两种方法进入到等待状态：等待一个对象句柄并指定它的等待是可警告的（使用Windows的*WaitForMultipleObjectsEx*函数），或者直接测试一下它是否有未处理的APC（使用*SleepEx*）。在这两种情况下，如果有一个用户模式的APC正在等待处理，则内核就会中断该线程，将控制权传递给APC例程，当APC例程完成时再恢复线程的运行。内核模式的APC运行在APC级别上，与此不同的是，用户模式的APC运行在被动级别上。

APC的交付可能会导致等待队列的重新排序，这里，等待队列是指哪些线程正在为了什么而等待，以及它们等待的顺序（本章后面的“低IRQL的同步”小节中讲述了等待解析过程）。如果当一个APC被交付时，线程正处于等待状态，则在APC例程完成以后，此等待动作将被重新发起或者被重新执行。如果该等待动作尚未被解析，则线程返回到等待状态，但现在，在它所等待的对象中，它位于等待列表的末尾。例如，因为APC也被用于将一个线程从执行状态挂起，所以，如果该线程正在等待任何对象，那么，它的等待要到该线程被重新执行的时候才会被移除，之后该线程将处于所有也在等待同样对象的线程列表的末尾。

异常分发

中断可以在任何时候发生，而与此不同的是，异常则是直接由当前正在运行的程序在执行过程中产生的条件。Windows引入了一种称为**结构化异常处理**的设施，使得应用程序可以在异常发生的时候获得控制。然后应用程序可以修正此条件，并返回到异常发生之处，将栈展开（从而使引发该异常的子例程的执行过程终止）；或者，向系统报告，该异常不可识别，因而系统应该继续搜索一个有可能处理此异常的异常处理器。本小节假设你已经熟悉了Windows的结构化异常处理机制背后的一些基本概念——如果你不熟悉的话，则在继续往下阅读以前，应该读一读Platform SDK中Windows API参考文档中的总体介绍部分，或者Jeffrey Richter著的*Programming Applications for Microsoft Windows*（第四版，Microsoft Press, 2000）中的第23至25章。请记住，尽管通过语言的扩展（例如，Microsoft Visual C++中的*__try*语法结构）你可以访问异常处理机制，但异常处理是一种系统机制，因此它并不是与特定语言相关的。Windows异常处理机制的其他用户，除了C++异常以外还有Java异常。

www.TopSage.com

在x86平台上，所有的异常都有预定义的中断号，这些中断号直接对应于IDT中的表项，每个表项指向了某一个特定异常的陷阱处理器。表3.2显示了x86定义的异常，以及为它们分配的中断号。因为IDT中前面的表项是用于异常的，所以，硬件中断被分配了后面的表项，就像前面所提到的那样。

表3.2 x86异常和它们的中断号

中断号	异 常
0	除法错误
1	调试陷阱
2	NMI/NPX错误
3	断点
4	溢出
5	BOUND/打印屏幕
6	无效操作码
7	NPX不可用
8	双重异常（Double Exception）
9	NPX段溢出
A	无效任务状态段 （TSS）
B	段不存在
C	栈错误
D	一般保护
E	页面错误
F	Intel保留
10	符点
11	对齐检查

所有的异常，除了那些简单到通过陷阱处理器就可以解决的以外，都是由一个称为**异常分发器**（*exception dispatcher*）的内核模块来服务的。异常分发器的任务是找到一个能够“处置”该异常的异常处理器。内核定义的与体系结构无关的异常包括内存访问违例、整数被零除、整数溢出、浮点异常，以及调试器断点。关于与体系结构无关的异常的完整列表，请查阅Windows API参考文档。

内核以透明于用户程序的方式捕获并处理以上某些异常。例如，在以调试方式执行一个程序时碰到一个断点就会产生一个异常，内核通过调用调试器来处理该异常。内核在处理其他某些特定的异常时，只是简单地向调用者返回一个未成功的状态码。

有些异常也允许原封不动地滤回到用户模式。例如，内存访问违例或者算术溢出产生的异常，操作系统不会对它们进行处理。环境子系统可以建立起**基于帧的异常处理器**（*frame-based exception handler*）来处理这些异常。

www.TopSage.com

术语“基于帧”是指将一个异常处理器与一个特定的过程激活动作关联起来。当一个过程被调用的时候，代表该过程激活动作的一个“栈帧（stack frame）”被压到栈中。一个栈帧可以关联一个或者多个异常处理器，每个异常处理器保护源程序中一块特定的代码。当发生一个异常时，内核查找与当前栈帧关联在一起的某个异常处理器。如果没有找到，则内核继续查找与上一个栈帧关联在一起的某个异常处理器，以此类推，直至找到一个基于帧的异常处理器。如果最终还是没有找到异常处理器，则内核调用它自己默认的异常处理器。

当异常发生的时候，不管它是由软件显式发出的，还是由硬件隐式发出的，在内核中会发生一系列事件。CPU硬件将控制权传递给内核陷阱处理器，内核陷阱处理器又创建一个陷阱帧（就如同中断发生时所做的那样）。正是由于该陷阱帧，在异常被解决以后，系统可以从它停止的地方恢复运行。陷阱处理器也会创建一条异常纪录，其中包含了该异常之所以发生的原因以及其他的有关信息。

如果在内核模式中发生了异常，则异常分发器只是简单地调用一个例程来找到一个基于帧的异常处理器，将来由它处理该异常。因为未处理的内核模式异常被认为是严重的操作系统错误，所以你可以假定，分发器总是会找到一个异常处理器。

如果异常发生在用户模式下，则异常分发器会做一些更加精细的工作。在第6章中你将会看到，Windows子系统有一个调试器端口和一个异常端口，通过它们来接收Windows进程中用户模式异常的通知。内核在它默认的异常处理中用到了这些端口，如图3.8所示。

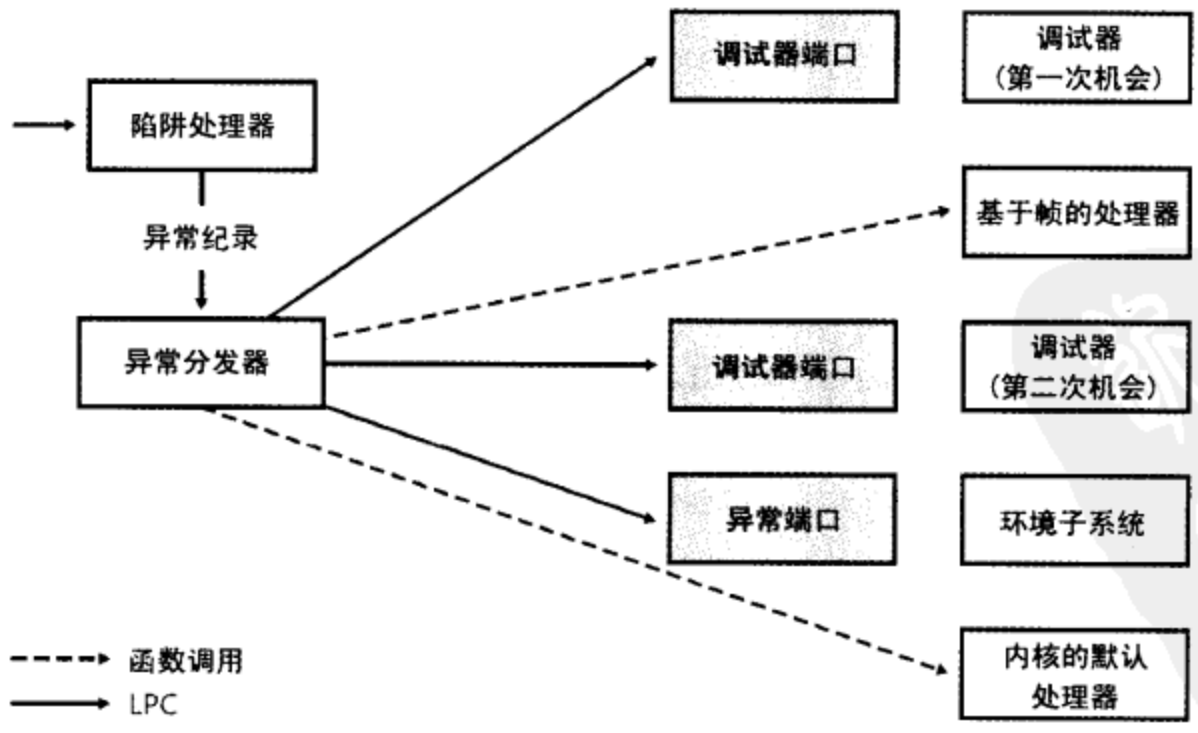


图3.8 分发一个异常

www.TopSage.com

调试器端口是最常见的异常来源。因此，异常分发器采取的第一个动作是，看一看引发该异常的进程是否有一个相关联的调试器进程。如果它确实有关联的调试器进程，而且该系统是Windows 2000，那么，异常分发器通过LPC向引发该异常的进程所关联的调试器端口发送第一次机会的调试消息。该LPC消息被发送给会话管理器进程，会话管理器进程随后又将它分发给适当的调试器进程。在Windows XP和Windows Server 2003上，异常分发器向与该进程关联的**调试对象**（在系统内部它称为一个端口）发送一个调试器对象消息。

如果该进程没有附载的调试器进程，或者调试器并没有处理该异常，那么，异常分发器切换到用户模式下，将陷阱帧按照CONTEXT数据结构（在Platform SDK中有文档介绍）的格式拷贝到用户栈中，并且调用一个例程来找到一个基于帧的异常处理器。如果没有找到，或者虽然找到了但它不处理该异常的话，则异常分发器切换回到内核模式下，并且再次调用调试器以便让用户做更多的调试工作（这被称为第二次机会的通知）。

如果调试器不在运行，并且没有找到基于帧的处理器，那么，内核向与该线程的进程关联在一起的异常端口发送一个消息。该异常端口如果存在的话，则一定是由控制该线程的环境子系统注册的。该异常端口使得环境子系统（它肯定在监听该端口）有机会将一个异常翻译成一个与环境相关的信号或者异常。CSRSS（Client/Server Real-Time Subsystem，客户/服务器运行时子系统）简单地弹出一个消息框来通知用户发生了错误，并终止进程；当POSIX从内核那里得到一个消息，指明它的一个线程产生了某个异常的时候，POSIX子系统向引发该异常的线程发送一个POSIX格式的信号。然而，如果内核在处理该异常的过程中已经向前走得很远了，而且子系统并没有处理该异常，那么，内核执行一个默认的异常处理器，它只是简单地将引发该异常的线程所在的进程终止掉。

未处理的异常

所有的Windows异常都有一个异常处理器，它位于栈顶上，负责处理所有未被处理的异常。该异常处理器是在Windows内部的**进程启动函数**或者**线程启动**函数中声明的。进程启动函数是当一个进程的第一个线程开始执行时才运行的，它调用映像中的主入口函数；线程启动函数是当一个用户创建附加线程的时候才开始运行的，它调用的是在*CreateThread*中指定的由用户提供的线程启动例程。

www.TopSage.com



实验：查看Windows线程真正的用户启动地址

每个Windows线程都是从一个由系统提供的函数开始执行的，这也正说明了为什么系统中每个Windows进程的0号线程的启动地址都是相同的（以及为什么其他线程的启动地址也是相同的）。Windows进程中0号线程的启动地址是Windows的进程启动函数；任何其他线程的启动地址则是Windows的线程启动函数。要想查看由用户提供的函数的地址，可以使用Windows支持工具箱中的Tlist实用工具。输入“**tlist <进程名>**”或者“**tlist进程id**”，就可以得到详细的进程信息输出，其中包括用户函数的地址。例如，比较一下在Pstat（在Platform SDK中）和Tlist这两个工具中Windows Explorer进程的线程启动地址：

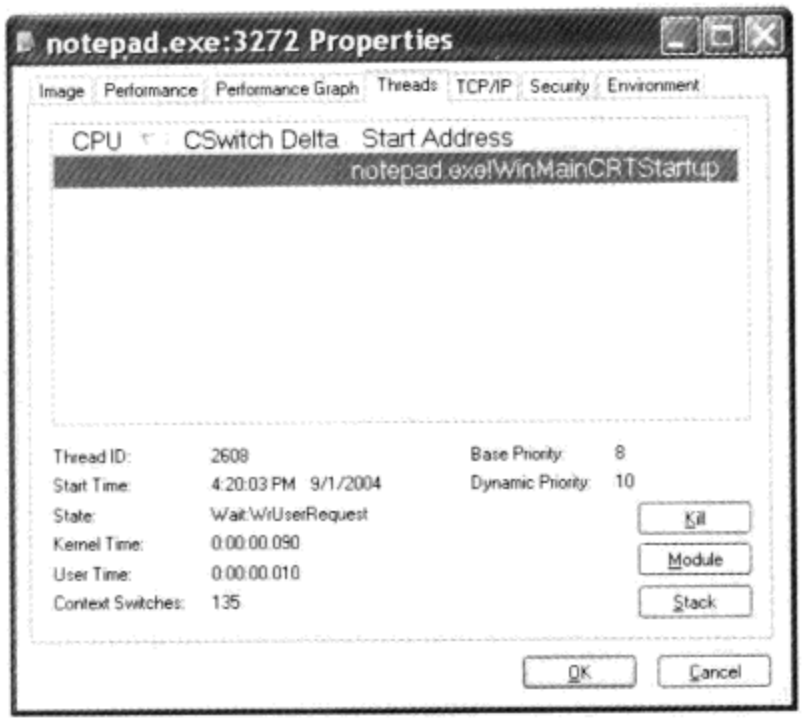
```
C:\> pstat
§
pid:3f8 pri: 8 Hnd:  329 Pf:  80043 Ws:   4620K explorer.exe
 tid pri Ctx Swtch StrtAddr   User Time Kernel Time  State
  7c   9     16442 77E878C1 0:00:01.241 0:00:01.251 Wait:UserRequest
 42c  11    157888 77E92C50 0:00:07.110 0:00:34.309 Wait:UserRequest
 44c   8      6357 77E92C50 0:00:00.070 0:00:00.140 Wait:UserRequest
 1cc   8      3318 77E92C50 0:00:00.030 0:00:00.070 Wait:DelayExecution
 §

C:\> tlist explorer
1016 explorer.exe      Program Manager
  CWD:     C:\
  CmdLine: Explorer.exe
  VirtualSize:    25348 KB   PeakVirtualSize:    31052 KB
  WorkingSetSize:  1804 KB   PeakWorkingSetSize:  3276 KB
  NumberOfThreads: 4
   149 Win32StartAddr:0x01009dbd LastErr:0x0000007e State:Waiting
    86 Win32StartAddr:0x77c5d4a5 LastErr:0x00000000 State:Waiting
    62 Win32StartAddr:0x00000977 LastErr:0x00000000 State:Waiting
   179 Win32StartAddr:0x0100d8d4 LastErr:0x00000002 State:Waiting
```

Pstat报告的0号线程的启动地址是Windows内部的进程启动函数；1至3号线程的启动地址则是Windows内部的线程启动函数。另一方面，Tlist显示的则是由用户提供的Windows启动地址（即由Windows内部的启动函数调用的用户函数）。

www.TopSage.com

因为在Windows进程中，大多数线程都是从某一个由系统提供的包装函数开始执行的，所以，进程管理器在显示一个进程中的线程的启动地址时，跳过初始的调用帧，即不显示包装函数的地址，而是显示栈中的第二个调用帧。例如，请注意Notepad.exe进程的线程启动地址：



进程管理器在显示调用栈的时候，显示了完整的调用层次。当点击Stack按钮时，可以看到下面的结果：



上图中，编号为12的行是栈中的第一个帧，即进程包装函数的启动地址。第二个帧（11行）是Notepad.exe的主入口点。

www.TopSage.com

这些内部启动函数的一般性代码如下：

```
void Win32StartOfProcess(
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm){

    __try {

        DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
        ExitThread(dwThreadExitCode);

    } __except(UnhandledExceptionFilter(
            GetExceptionInformation())) {

        ExitProcess(GetExceptionCode());
    }
}
```

请注意，如果一个线程的异常没有被处理，则Windows的未处理异常过滤器将会被调用。这个函数的目的是，当一个异常未被处理时，可以提供一种系统统一的行为和方法，其具体做法取决于HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug注册表键中的内容。该键有两个重要的值：Auto和Debugger。Auto值告诉未处理异常过滤器，是自动运行调试器，还是询问用户该怎么做。在默认情况下，该值为1，意味着它将会自动启动调试器。然而，一旦安装了像Visual Studio这样的开发工具，该值会被改成0。Debugger值是一个字符串，指向调试器可执行文件的路径，一旦有未处理的异常，该调试器可执行文件就会运行。

默认的调试器是\Windows\System32\Drwtsn32.exe（也称为Dr. Watson），它实际上不是调试器，而是一个事后分析工具，它捕获了应用程序“崩溃”时的状态，并且将状态记录在一个日志文件（Drwtsn32.log）和一个进程崩溃转储文件（User.dmp）中，在默认设置下，这两个文件放在\Documents And Settings\All Users\Documents\DrWatson文件夹中。要想查看（或者修改）Dr. Watson的配置，可以通过交互方式运行该程序——它会显示一个窗口，其中包含当前的设置，如图3.9所示。

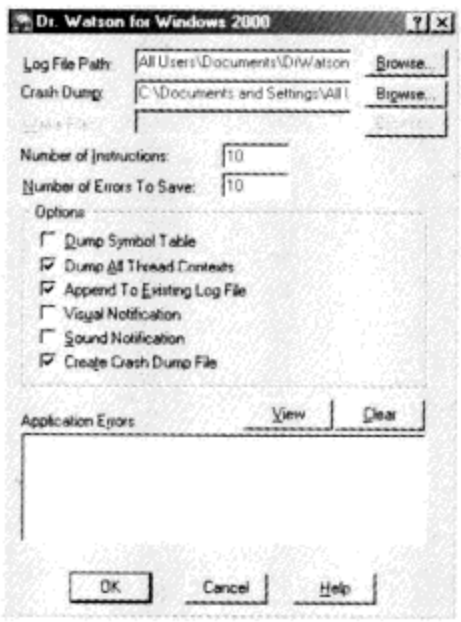


图3.9 Windows 2000中Dr. Watson默认设置

www.TopSage.com

日志文件中包含了基本的信息，比如异常代码、失败的映像文件的名称、加载的DLL的列表，以及导致该异常的线程的栈和指令痕迹。有关日志文件中内容的详细描述，请运行Dr. Watson，并点击如图3.9所示的Help按钮。

崩溃转储文件中包含了该异常发生时进程中私有的页面（该文件并不包含从EXE或者DLL文件中映射过来的代码页面）。该文件可以用WinDbg来打开，WinDbg是调试工具箱中随带的Windows调试器，或者也可以通过Visual Studio 2003及其以后的版本来打开。因为每次当一个进程崩溃的时候User.dmp文件都会被改写，所以，除非当每次进程崩溃之后你重新命名或者拷贝该文件，否则你的系统中只有最新的一份拷贝。

在Windows 2000 Professional系统中，在默认设置下可视通知是打开的。Dr. Watson在产生了崩溃转储文件并在日志文件中记录了相应的信息以后，会显示如图3.10所示的消息框。

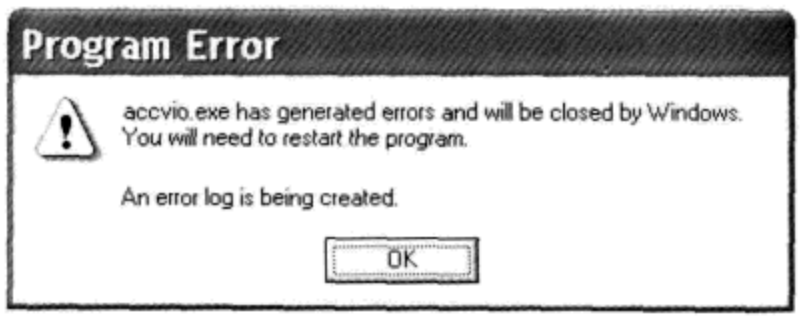


图3.10 Windows 2000中Dr. Watson错误消息

Dr. Watson进程一直保持到消息框消失为止，这也正是为什么在Windows 2000 Server系统上可视通知默认是关掉的。这种默认方式是有用的，因为如果一个服务器应用程序失败了的话，通常在控制台旁边不会有人看到它，并关掉消息框。与此相反，服务器应用程序应该将错误记录到Windows事件日志中。

在Windows 2000中，如果Auto值被设置为0的话，则如图3.11所示的消息框将会显示出来。

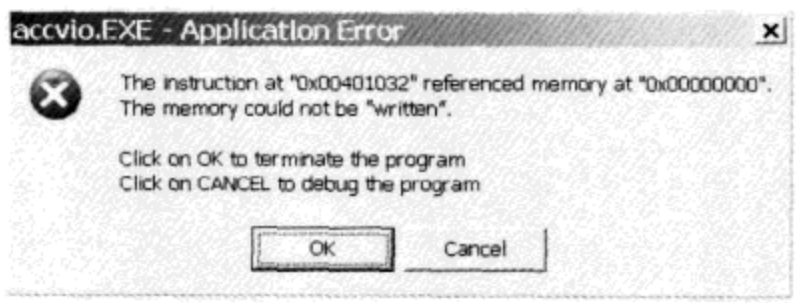


图3.11 Windows 2000中的未处理异常消息

如果用户点击了OK按钮，则进程就会退出。如果点击了Cancel按钮，则系统定义的调试器进程（前面提到的注册表中的Debugger值所指定的调试器路径）被启动起来。

www.TopSage.com



实验：未处理的异常

要想看一个Dr. Watson日志文件的例子，你可以下载并运行Accvio.exe程序。从*www.sysinternals.com/windowsinternals.shtml*你可以下载到该程序。该程序试图在地址0处写入数据，从而产生一个内存访问违例，因为在Windows进程中，地址0总是一个无效地址（参见第7章中的表7.6）。

1. 运行注册表编辑器，找到HKLM\SOFTWARE\ Microsoft\Windows NT\CurrentVersion\ AeDebug。
2. 如果Debugger值是“drwtsn32 -p %ld -e %ld –g”，则你的系统已经被配置成运行Dr. Watson。作为默认调试器，直接转到第4步。
3. 如果Debugger的值没有被配置成运行Drwtsn32.exe，那么，你仍然可以测试Dr. Watson。测试方法很简单，只要临时将它安装上，然后再恢复以前的调试器设置即可：
   a. 将当前的值保存在某个地方（例如，保存在一个Notepad文件中，或者在当前的剪贴板缓冲区中）；
   b. 从任务栏的Start菜单中选择Run，然后输入**drwtsn32 –i**（这将会初始化Debugger域，以便运行Dr. Watson）。
4. 运行测试程序Accvio.exe。
5. 你应该能够看到前面讲述过的某个消息框（取决于你运行的Windows的版本）。
6. 如果你有默认的Dr. Watson设置，那么，你现在应该可以在转储文件目录中检查日志文件和转储文件了。要想查看Dr. Watson的具体设置，可以运行drwtsn32，不需任何参数（从Start菜单中选择Run，然后输入**drwtsn32**）。
7. 或者另一种做法是，在Dr. Watson显示的Application Errors列表中，点击最后一条，然后点击View按钮——Dr. Watson日志文件中有关Accvio.exe访问违例的详细信息就会显示出来（关于日志文件格式的细节信息，请打开Dr. Watson的帮助窗口，然后选择Dr. Watson Log File Overview）。
8. 如果Debugger原来的值不是默认的Dr. Watson设置，则恢复步骤3中保存的值。

作为另一个实验，你可以试着将Debugger的值改成另一个程序，比如Notepad.exe（记事本编辑器）或者Sol.exe（纸牌游戏）。重新运行Accvio.exe，就可以看到Debugger值中指定的程序开始运行了。也就是说，这里并不会验证Debugger值中定义的程序是否真的是一个调试器。请一定要恢复原来的注册表设置（正如步骤3b中所注明的，要想将Debugger值恢复到系统默认的Dr. Watson设置，你只需在Run对话框中或者命令行提示符下输入**drwtsn32 –i**即可）。

www.TopSage.com

Windows错误报告

Windows XP和Windows Server 2003有一种全新的、更加复杂的错误报告机制，称为Windows错误报告（Windows Error Reporting），它可以自动提交用户模式的进程崩溃和内核模式的系统崩溃（要想知道这种机制如何适用于系统崩溃的情形，请参见第14章）。

为了配置Windows错误报告机制，请转到“My Computer”，选择“Properties”、“Advanced”，然后选择“Error Reporting”（这将会弹出如图3.12所示的对话框），或者通过System、Error Reporting下的本地或域的组策略设置进行配置。这些设置被保存在注册表键HKLM\Software\Microsoft\PCHealth\ErrorReporting中。

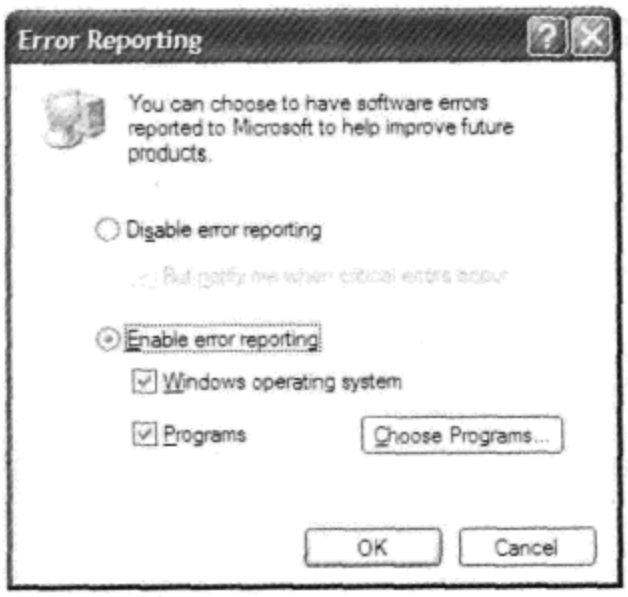


图3.12 错误报告配置对话框

当一个未处理的异常被未处理异常过滤器（前面已经介绍过）捕捉到的时候，首先执行一个基本的检查，看是否要启动Windows错误报告机制。如果HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto注册表值被设置为0，或者Debugger字符串包含了文本“Drwtsn32”，则未处理异常过滤器将\Windows\System32\Faultrep.dll加载到失败进程中，并且调用它的*ReportFault*函数。然后，*ReportFault*函数检查HKLM\Software\Microsoft\PCHealth\ErrorReporting下保存的错误报告配置，以确定该进程崩溃是否应该被报告，以及如果要报告的话，又该怎么办。在一般情况下，*ReportFault*创建一个进程来运行\Windows\System32\Dwwin.exe，它会显示一个对话框来报告进程崩溃事件，并且提供一个选项可以将错误报告提交给Microsoft，如图3.13所示。

www.TopSage.com

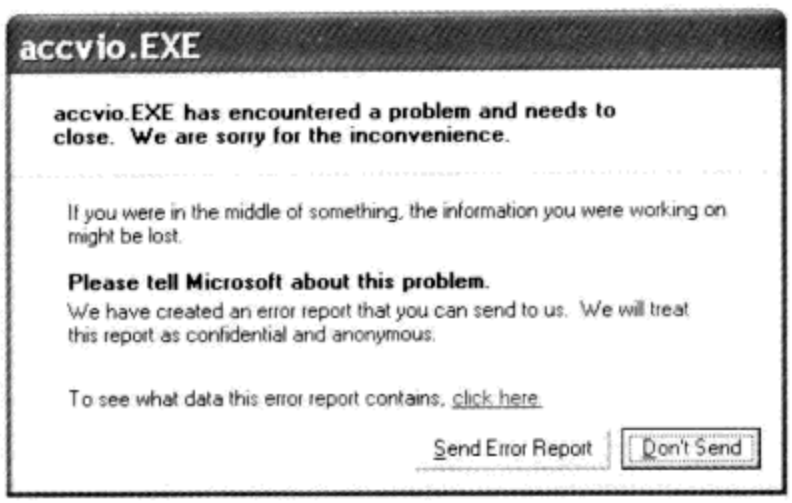


图3.13 Windows错误报告对话框

如果“Send Error Report”按钮被点击的话，则错误报告（一个微型转储文件以及一个文本文件，文本文件中包含了该进程中加载的DLL版本号信息）被发送至Microsoft的在线崩溃分析服务器：Watson.Microsoft.com（不同于内核模式系统崩溃，在那种情况下，没有办法在提交报告的时候确定一个方案是否可行）。 然后，未处理异常过滤器创建一个进程，以便运行系统定义的调试器（通常是Drwtsn32.exe），该调试器默认地又会创建它自己的转储文件和日志纪录。不同于Windows 2000的情形，该转储文件只是一份微型转储，而不是完整的转储。所以，一旦需要完整的进程内存转储才能调试一个失败的应用程序时，你可以改变Dr. Watson的配置，做法很简单，就如同上一小节介绍的那样，不带任何参数运行Dr. Watson。

如果系统没有连接至Internet，或者管理员想要控制哪些错误报告才可以提交给Microsoft，在这样的环境中，错误报告的目的地可以被配置成一台内部文件服务器。Microsoft向一些有资格的客户提供了一个称为企业错误报告（Corporate Error Reporting）的工具集，它能理解Windows错误报告机制创建的目录结构；同时也给管理员提供了一个选项，让他们可以选择部分错误报告提交给Microsoft（有关更多的信息，请参见*http://www.microsoft.com/resources/satech/cer*）。

系统服务分发

正如图3.1所示的，内核的陷阱处理器分发中断、异常和系统服务调用。在前面的小节中，你已经看到了中断和异常是如何被处理的；在这一小节中，你将会了解有关系统服务的内容。每当执行一条专门分配给系统服务分发的指令时，系统服务分发功能就会被触发。Windows用于系统服务分发的指令取决于它执行时所在的处理器。

32位系统服务分发

在Pentium II之前的x86处理器上，Windows使用int 0x2e指令（十进制是46），它会导致一

www.TopSage.com

个陷阱。Windows填充IDT的46号表项，使其指向系统服务分发器（参见表3.1）。该陷阱导致执行线程转换到内核模式中，并且进入系统服务分发器。在处理器的寄存器EAX中传递的数值参数指明了所请求的系统服务号。EBX寄存器指向调用者传递给该系统服务的参数列表。

在x86 Pentium II处理器以及更高级的处理器上，Windows使用专门的*sysenter*指令，这是Intel特别为快速系统服务分发而定义的指令。为了支持这一指令，Windows在引导时刻将内核的系统服务分发器的地址保存在与该指令相关联的一个寄存器中。该指令被执行时，就会导致变换到内核模式下，并且执行系统服务分发器。系统服务号是通过处理器的EAX寄存器来传递的，而EDX寄存器则指向调用者参数的列表。为了返回到用户模式，系统服务分发器通常执行*sysexit*指令（在有些情况下，例如当处理器上单步标志被打开的时候，系统服务分发器改而使用*iretd*指令）。

在K6和更高级的32位AMD处理器上，Windows使用专门的*syscall*指令，它的功能类似于x86的*sysenter*指令，Windows同样配置一个与*syscall*关联的处理器寄存器，其中包含了内核的系统服务分发器的地址。系统调用号也是在EAX寄存器中传递的，而调用者的参数则保存在栈中。在完成了分发之后，内核执行*sysret*指令。

在引导时，Windows检测当前运行所在的处理器的类型，从而建立起以后使用的适当的系统调用代码。用户模式下的NtReadFile系统服务代码看起来像这样：

```
ntdll!NtReadFile:
77f5bfa8 b8b7000000      mov     eax,0xb7
77f5bfad ba0003fe7f      mov     edx,0x7ffe0300
77f5bfb2 ffd2            call    edx
77f5bfb4 c22400          ret     0x24
```

系统服务号是0xb7（十进制183），*call*指令执行由内核建立起来的系统服务分发代码，在这个例子中位于地址0x7ffe0300处。因为这是从Pentium M中取出来的，它使用了*sysenter*：

```
SharedUserData!SystemCallStub:
7ffe0300 8bd4            mov     edx,esp
7ffe0302 0f34            sysenter
7ffe0304 c3              ret
```

64位系统服务分发

在x64体系结构上，Windows使用*syscall*指令进行系统服务分发，该指令的功能类似于AMD K6的*syscall*指令，系统调用号通过EAX寄存器来传递，前4个参数放在寄存器中，除此之外的任何其他参数都放在栈中：

www.TopSage.com

```
ntdll!NtReadFile:
00000000`77f9fc60 4c8bd1          mov     r10,rcx
00000000`77f9fc63 b8bf000000      mov     eax,0xbf
00000000`77f9fc68 0f05            syscall
00000000`77f9fc6a c3              ret
```

在IA64体系结构上，Windows使用epc（Enter Privileged Code，进入特权代码）指令。前8个系统调用参数通过寄存器来传递，其余参数通过栈来传递。

内核模式的系统服务分发

正如图3.14所显示的，内核利用传递进来的参数，找到**系统服务分发表**中的系统服务信息。系统服务分发表类似于本章前面讲述的中断分发表，只不过在这里每个表项都包含了一个指向某个系统服务的指针，而不是指向某个中断处理例程。



> **注** 系统服务号可能会随着不同的服务包（service pack）而有所不同——Microsoft偶尔会增加或者去除某些系统服务，而且系统服务号是作为内核编译的一部分而自动产生的。

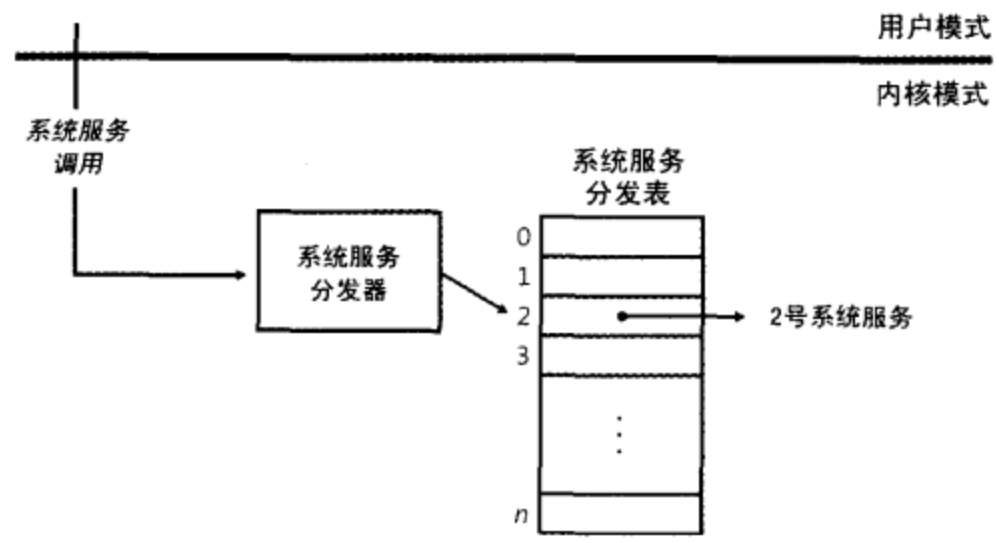


图3.14 系统服务异常

系统服务分发器*KiSystemService*将调用者的参数从线程的用户模式栈中拷贝到内核模式栈中（所以，当内核在访问参数时，用户不能改变它们），然后执行该系统服务。如果传递给一个系统服务的参数指向了用户空间中的缓冲区，那么，内核模式的代码在拷贝数据到这些缓冲区中，或者从缓冲区中拷贝数据以前，必须先要查明这些缓冲区是否是可以访问的。

正如你在第6章中将要看到的那样，每个线程都有一个指针指向它的系统服务表。Windows有两个内置的系统服务表，它最多可以支持4个系统服务表。系统服务分发器确定哪个表包含了所请求的服务，它将32位系统服务号中的其中一个2位域解释成一个表索引。系统服务号的低12位被用于在该表索引所指定的表中进行索引。这些域如图3.15所示。

www.TopSage.com

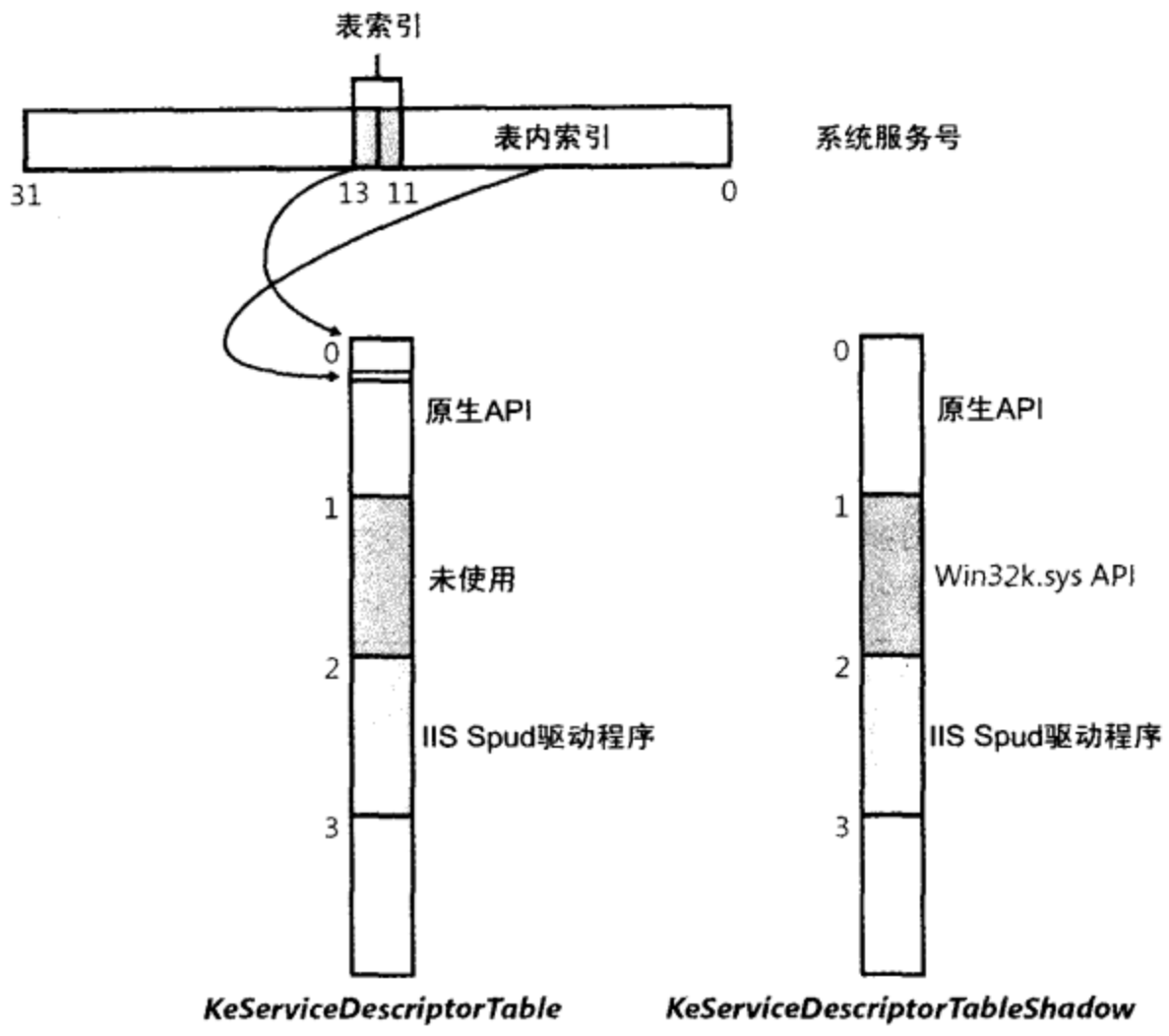


图3.15 从系统服务号至系统服务的翻译过程

服务描述符表

一个主要的默认数组表（*KeServiceDescriptorTable*）定义了Ntosrknl.exe中实现的核心执行体系统服务。另一个默认数组表（*KeServiceDescriptorTable*）包含了在Windows子系统的内核模式部分Win32k.sys中实现的Windows USER和GDI服务。当Windows线程第一次调用一个Windows USER和GDI服务时，该线程的系统服务表的地址被改变成指向一个包含Windows USER和GDI服务的表格。*KeAddSystemServiceTable*函数使得Win32k.sys和其他的设备驱动程序可以添加系统服务表。如果你在Windows 2000上安装了Internet信息服务（Internet Information Services，IIS），那么它的支持驱动程序（Spud.sys）在加载的时候定义了一个附加的服务表，从而给第三方只留下一个表的空间。除了Win32k.sys服务表以外，其他通过*KeAddSystemServiceTable*添加的服务表被同时拷贝到*KeServiceDescriptorTable*数组和*KeServiceDescriptorTableShadow*数组中。除了核心表和Win32表以外，Windows只支持两个附加的系统服务表。

www.TopSage.com



> **注** Windows Server 2003 SP1和更高的服务包不支持除了通过Win32k.sys加入的系统服务表以外的其他服务表，所以，通过添加系统服务表的方式来扩展系统功能的做法在这些系统上是行不通的。

针对Windows执行体服务的系统服务分发指令位于系统库Ntdll.dll中。子系统DLL调用Ntdll.dll中的函数来实现它们已经文档化的函数。惟一的例外是Windows USER和GDI函数，在这些函数中，系统服务分发指令是直接在User32.dll和Gdi32.dll中实现的，没有涉及Ntdll.dll。图3.16显示了这两种情形。

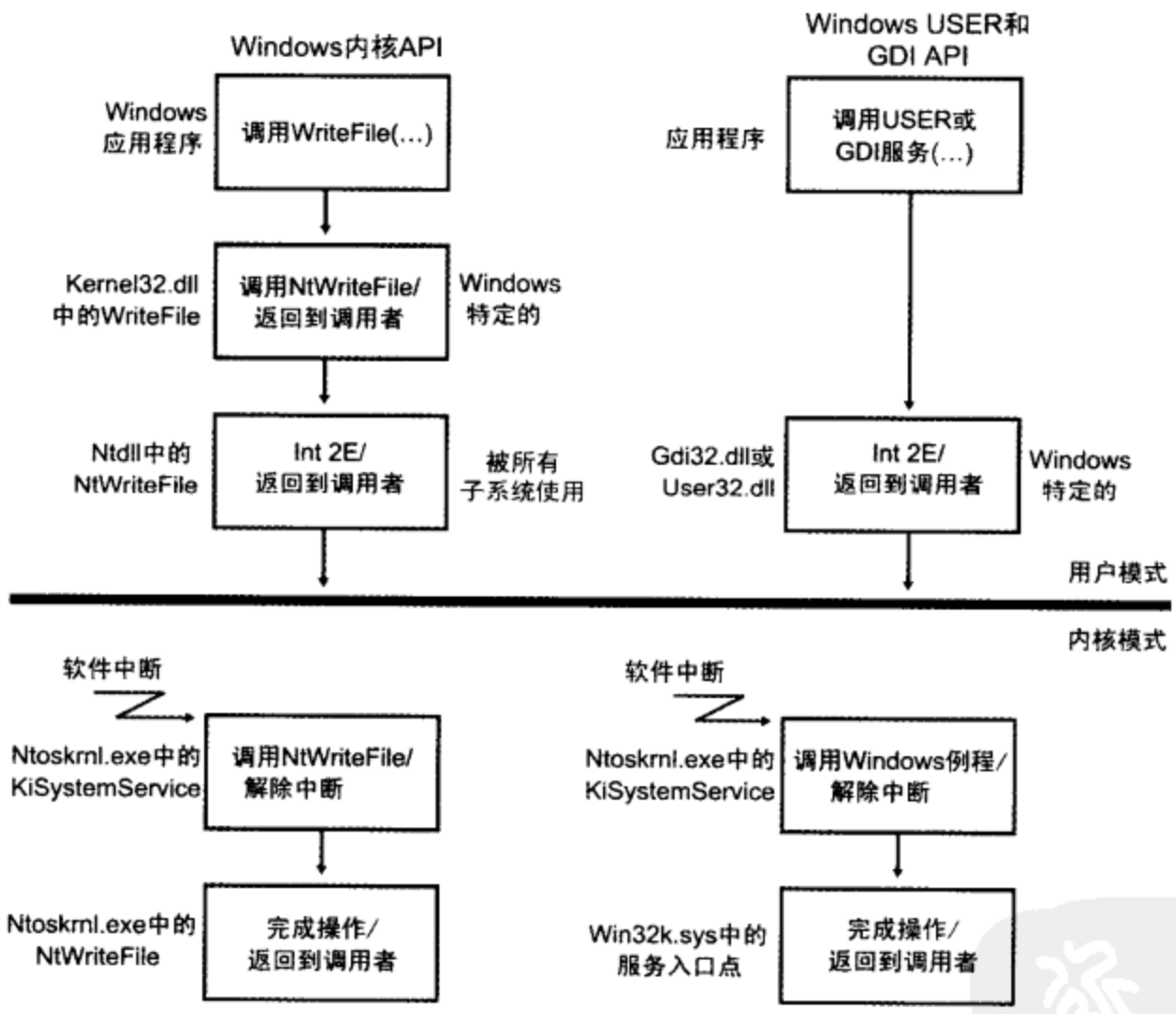


图3.16 系统服务分发过程

正如图3.16中所显示的，位于Kernel32.dll中的Windows *WriteFile*函数调用Ntdll.dll中的*NtWriteFile*函数，而后*NtWriteFile*函数又执行适当的指令以便引发一个系统服务陷阱，并且把代表*NtWriteFile*的系统服务号传递过去。系统服务分发器（Ntoskrnl.exe中的*KiSystemService*函数）然后调用真正的*NtWriteFile*来处理该I/O请求。对于Windows USER和GDI函数，系统服务分发器调用Windows子系统可加载内核模式部分（Win32k.sys）中的函数。

www.TopSage.com



实验：查看系统服务的行为

通过观察“System”对象中的“System Calls/Sec”性能计数器，你可以监视系统服务的行为。运行性能工具，然后在图表视图中，点击Add按钮可以在图表中加入一个计数器；选择“System”对象，再选择“System Calls/Sec”计数器，然后点击Add按钮就可以将该计数器加入到图表中。

3.2 对象管理器

正如在第2章中所提到的那样，Windows实现了一个对象模型，以便为执行体中实现的各种内部服务提供一致的、安全的访问途经。本节将介绍Windows的对象管理器(object manager)，即执行体内部负责创建、删除、保护和跟踪对象的组件。对象管理器将那些本来有可能散落在整个操作系统各处的资源控制操作集中在一起。对象管理器的设计意图是为了实现本章稍后列出的一系列目标。



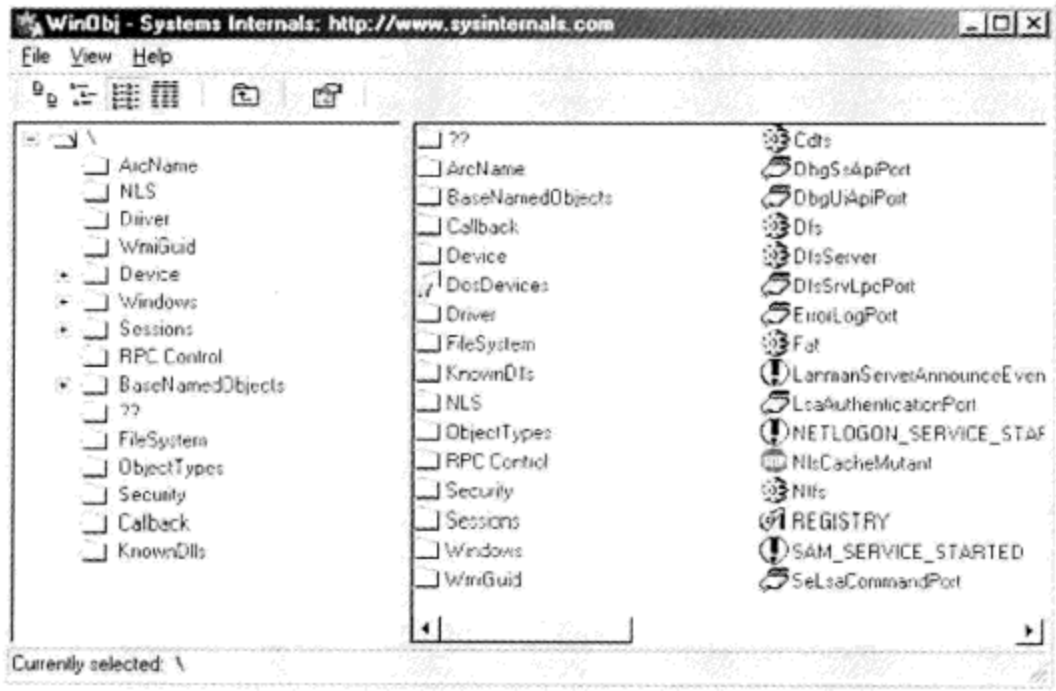
实验：考察对象管理器

在这一节中，你将会看到，有一些实验展示了如何窥探对象管理器数据库。这些实验用到了下面的工具，如果你对这些工具尚不熟悉的话，应该先花时间熟悉一下为好。

- Winobj （从*www.sysinternals.com*上下载） 显示了内部的对象管理器的名字空间。在Platform SDK中也有Winobj的一个版本（位于\Program Files\Microsoft Platform SDK\Bin\Winnt\Winobj.exe），但是，来自*www.sysinternals.com*的Winobj显示了更为精确的对象信息（比如引用计数、已打开句柄的数目、安全描述符，等等）。
- 来自*www.sysinternals.com*的进程管理器和Handle工具（在第1章中介绍过）显示了一个进程中已打开的句柄。
- Oh.exe（位于Windows资源工具箱中）显示了一个进程中已打开的句柄，但它要求设置一个全局标志才能运行。
- *Openfiles /query*命令（在Windows XP和Windows Server 2003中）显示了一个进程中已打开的句柄，但它要求设置一个全局的标志才能运行。
- 内核调试器的*!handle*命令显示了一个进程中已打开的句柄。

www.TopSage.com

对象查看器提供了一种方法来遍历对象管理器所维护的名字空间（正如稍后我们将要介绍的，并不是所有的对象都有名字）。 试着运行一下从*www.sysinternals.com*下载的WinObj对象管理器工具，并且检查一下其布局结构，如下图所示：



正如前面所说明的，OH工具和Openfiles /query命令要求设置一个称为“*maintain objects list*”的Windows全局标志（关于全局标志的更多细节信息，请参见本章后面的“Windows全局标志”一节）。 OH将会设置该标志，如果它尚未被设置的话。如果你输入**Openfiles /Local**，它会告诉你该标志是否已被设置。通过*Openfiles /Local ON*命令，你可以设置该标志。无论采用哪种方法，你的系统都必须重新引导，以便使该设置生效。来自*www.sysinternals.com*的进程管理器和Handle工具都不要求打开对象跟踪功能，因为它们使用一个设备驱动程序来获得有关的信息。

对象管理器的设计目标是满足下面的要求：

- 提供一种公共的、统一的机制来使用系统资源；
- 将对象保护隔离到操作系统的统一区域中，从而可以做到C2安全等级；
- 提供一种机制来记录进程使用对象的数量，从而可以对系统资源的使用加上限制；
- 建立一套对象命名方案，它可以很方便地融合现有的对象，比如设备、文件、文件系统中的目录，或者其他独立的对象集合；

www.TopSage.com

- 支持各种操作系统环境的需求，比如一个进程能够从它的父进程继承资源（Windows和POSIX都需要这样的能力）、创建大小写敏感的文件名的能力（POSIX需要这种能力）；
- 建立统一的规则来维护对象的保持力（也就是说，总是保持一个对象是可用的，直至所有的进程都用完了这个对象为止）。

Windows的内部有两种类型的对象：**执行体对象**和**内核对象**。所谓执行体对象，是指由执行体的各种组件 （比如进程管理器、内存管理器、I/O子系统，等等） 所实现的对象。内核对象是指由Windows内核实现的一组更为基本的对象；这些对象对于用户模式的代码而言是不可见的，它们只是在执行体内部被创建和使用。内核对象提供了最为基本的能力，比如同步等，执行体对象正是建立在它们之上的。因此，许多执行体对象包含（封装）了一个或者多个内核对象，如图3.17所示。

有关内核对象数据结构的细节信息，以及如何利用内核对象来实现同步的详细情况，本章后面将会介绍。在这一节的余下内容中，我们将集中介绍对象管理器是如何工作的，以及执行体对象、句柄和句柄表的数据结构。这里，我们也将粗略地描述一下对象是如何参与到Windows的安全访问检查机制中的，在第8章中我们将会全面地讨论这一话题。

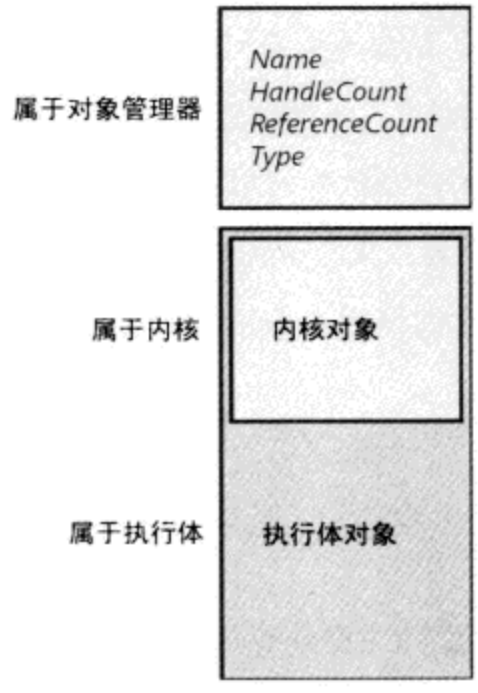


图3.17 包含了内核对象的执行体对象

执行体对象

每个Windows环境子系统总是把操作系统的不同面貌呈现给它的应用程序。执行体对象和对象服务是环境子系统用于构建其自己版本的对象和其他资源的基础。

www.TopSage.com

执行体对象往往或者是由一个环境子系统代表一个用户应用程序而创建的，或者是由操作系统的各种组件作为它们常规操作的一部分而创建的。例如，为了创建一个文件，Windows应用程序调用Windows的*CreateFile*函数，该函数是在Windows子系统DLL Kernel32.dll中实现的。在经过了一些验证和初始化工作以后，*CreateFile*依次又调用原生的Windows服务*NtCreateFile*来创建一个执行体文件对象。

环境子系统提供给其应用程序的对象集合可能比执行体提供的对象集合要大一些，或者小一些。Windows子系统使用执行体对象来导出它自己的对象集合，其中许多对象直接对应于执行体对象。例如，Windows的互斥体和信号量是直接建立在执行体对象之上的，而这些执行体对象又依次建立在对应的内核对象之上。另外，Windows子系统提供了命名管道（named pipe）和邮件槽（mailslot），它们是建立在执行体文件对象基础之上的资源。有些子系统，比如POSIX，甚至根本不提供对象方式的支持。POSIX子系统使用执行体对象和服务作为基础，来向其应用程序表达POSIX风格的进程、管道和其他资源。

表3.3列出了执行体提供的基本对象，以及简要地描述了它们所代表的含义。在本书后面专门讲述有关执行体组件的章节中，你可以找到更多有关执行体对象的细节。如果执行体对象被直接导出至Windows应用程序的话，你可以在Windows API参考文档中找到有关的细节信息。



> 注 在Windows 2000中，执行体总共实现了27种对象类型，而在Windows XP和Windows Server 2003中，总共实现了29种（这些新一点版本的Windows增加了*DebugObject* 和*KeyedEvent*对象）。 这些对象中的大多数仅仅被用于其定义所在的执行体组件，通过Windows API是不能直接访问它们的。这样的对象包括驱动程序（*Driver*）、设备（*Device*）和事件对（*EventPair*）。

表3.3 暴露给Windows API的执行体对象

对象类型	所代表的含义
符号链接（Symbolic link）	间接地引用一个对象名字的机制
进程（Process）	虚拟地址空间，以及为了执行一组线程对象而必需的控制信息
线程（Thread）	进程内部的一个可执行实体。
作业（Job）	指一组进程，通过作业机制，可以像单个实体那样来管理它们
内存区（section）	共享内存的一个区域 （在Windows中也称为文件映射对象）
文件（File）	一个已打开的文件或者I/O设备的实例
访问令牌（Access token）	一个进程或者线程的安全轮廓 （安全ID、用户权限等）
事件（Event）	一种具有持久状态（有信号的，或者无信号的）的对象，可被用于同步或者通知
信号量（Semaphore）	信号量是一个计数器，它提供了资源门控能力，对于该信号量所保护的资源只允许某个最大数目的线程访问它
互斥体（Mutex）*	用于顺序访问一个资源的一种同步机制
定时器（Timer）	这是一种当固定长时间过去时，通知一个线程的机制

www.TopSage.com

续表

对象类型	所代表的含义
IO完成（IoCompletion）	使线程能够将“I/O操作完成通知”进出队列的一种方法（在Windows API中被称为I/O完成端口）
键（Key）	这是一种引用注册表中数据的机制。虽然键出现在对象管理器的名字空间中，但它们是由配置管理器来管理的，其管理方式类似于文件系统驱动程序管理文件对象的方式。与键对象关联在一起的有零个或者多个键值；一个键的键值中包含了有关该键的数据
窗口站（WindowStation）	该对象包含了一个剪贴板、一组全局原子（atom）和一组桌面对象
桌面（Desktop）	这是一个被包含在窗口站内部的对象。桌面对象有一个逻辑显示器表面，其中包含了窗口、菜单和钩子



注 在外部Windows API中，突变体（mutant）被称为互斥体（*mutex*）。而在内部，位于互斥体之下的内核对象被称为突变体。

对象结构

正如图3.18中所示的，每个对象都有一个对象头和一个对象体。对象管理器控制了对象头，而执行体组件则控制了由它们创建的对象类型的对象体。每个对象头指向一个进程列表，该列表中的每个进程都打开了此对象；并且每个对象头也指向一个被称为类型对象（type object）的特殊对象，该对象包含的信息对于它的每个实例都是公共的。

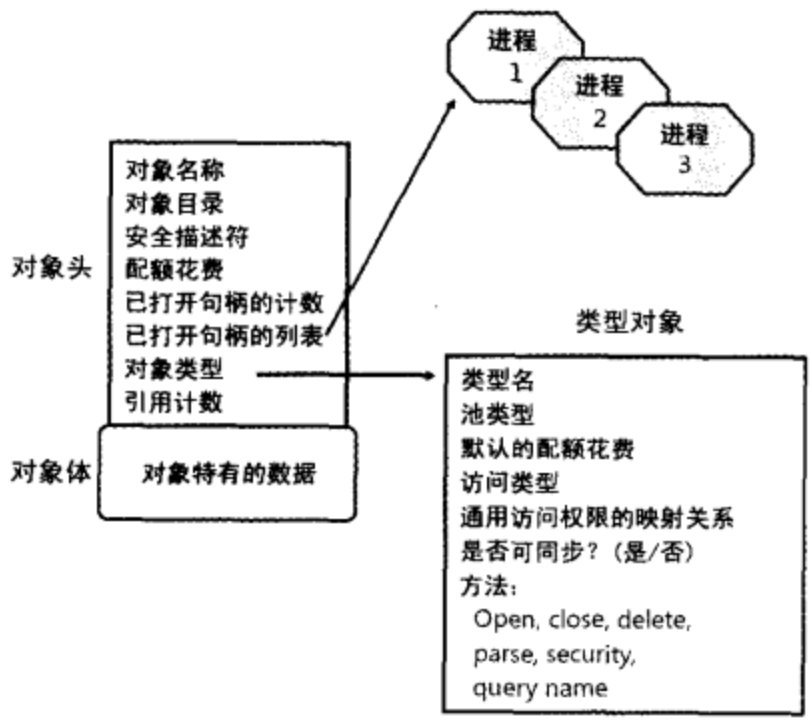


图3.18 对象的结构

www.TopSage.com

对象头和对象体

对象管理器使用对象头中保存的数据来管理这些对象，而无须关心它们的类型。表3.4简要地描述了对象头的属性。

表3.4 标准的对象头属性

属 性	用 途
对象名称	使一个对象对于其他的进程也是可见的，以便于共享
对象目录	提供了一个层次结构来存储对象名称
安全描述符	决定谁可以使用该对象，以及允许它们如何使用它（注：对于没有名称的对象来说，安全描述符可以是空[null]）
配额花费	列出了当一个进程打开一个指向该对象的句柄时，针对该进程收取的资源花费额
已打开句柄的计数	记录了“打开一个句柄来指向该对象”的次数
已打开句柄的列表	指向一个进程列表，其中的每个进程都打开了指向该对象的句柄（并不是所有的对象都出现该列表）
对象类型	指向一个类型对象，该对象包含了针对这种类型的对象都是公共的属性
引用计数	记录了一个内核模式组件引用该对象地址的次数

除了对象头以外，每个对象也有一个对象体，并且其格式和内容只有这种对象类型才有；同一类型的所有对象共享同样的对象体格式。一个执行体组件通过创建一个对象类型，并且为它提供一些服务，就可以控制和维护所有这种类型的对象体中的数据。

对象管理器提供了少量通用的服务，通过这些服务可以对一个对象头中保存的属性进行操作，并且可以用在任何类型的对象上（但是，有些通用服务对于某些特定的对象并没有意义）。表3.5中列出了这些通用的服务，Windows子系统允许其中的某一些服务可直接被Windows应用程序使用。

虽然这些通用的对象服务都支持所有的对象类型，但是，每个对象都有它自己的创建、打开和查询服务。例如，I/O系统为它的文件对象实现了一个创建文件的服务，而进程管理器则为它的进程对象实现了一个创建进程的服务。尽管实现一个通用的创建对象的服务也是有可能的，但是这样的过程将会相当复杂，因为，用于初始化一个文件对象所需要的参数集合与初始化一个进程对象所需要的参数集合会有极大的差异。而且，每次当一个线程调用一个对象服务来确定一个句柄所指对象的类型，再调用正确版本的服务时，对象管理器将会招致额外的处理开销。出于这样那样的理由，创建、打开和查询服务是针对每一种对象类型单独实现的。

www.TopSage.com

表3.5 通用的对象服务

服 务	用 途
关闭	关闭一个指向某个对象的句柄
复制	通过先复制一个句柄，再将它交给另一个进程的方法来共享一个对象
查询对象	获得关于一个对象的标准属性的信息
查询安全性	获取一个对象的安全描述符
设置安全性	改变一个对象上的保护设置
等待一个对象	用一个对象来同步一个线程的执行
等待多个对象	用多个对象来同步一个线程的执行

类型对象

对象头中包含的数据对于所有的对象都是公共的，但是每个对象实例可以取不同的值。例如，每个对象有一个惟一名称，也可以有惟一的安全描述符。然而，对象也可以包含一些对于某种特定类型的所有对象皆为常数的数据。例如，当你打开一个指向某种类型的对象的句柄时，你可以从一组特定于该对象类型的访问权限中进行选择。执行体为线程对象提供了终止和挂起（及其他）访问操作，为文件对象提供了读、写、添加和删除（及其他）访问操作。与对象类型相关的属性的另一个例子是同步，稍后将会介绍。

为了节省内存，对象管理器只在创建一个新的对象类型时，才存储这些静态的、特定于对象类型的属性。它使用其自己的一个对象（一个类型对象）来记录这些数据。如图3.19所示，如果对象跟踪调试标志（参见本章后面的“Windows全局标志”一节中所讲述的）被设置了的话，类型对象也会将同一类型（在图中是进程类型）的所有对象链接起来，从而使得对象管理器可以在必要的时候找到这些对象，并且对它们进行遍历。

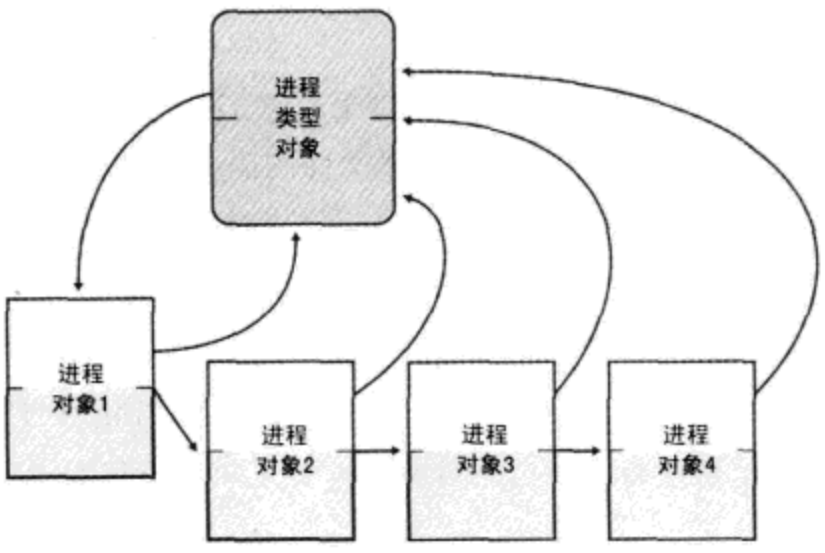


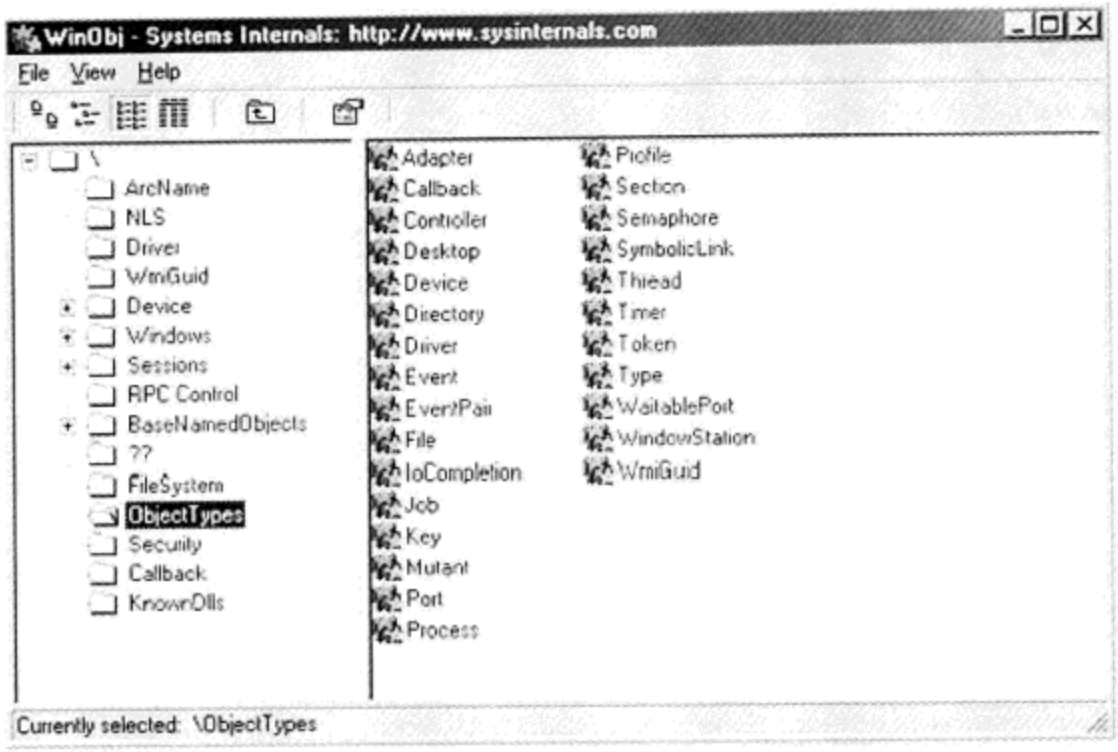
图3.19 进程对象和进程类型对象

www.TopSage.com



实验：查看对象头和类型对象

利用从*www.sysinternals.com*下载的Winobj工具，你可以看到那些在对象管理器中声明的类型对象的列表。在运行了Winobj之后，打开\ObjectTypes目录，如下图所示：



在内核调试器中，你可以看一看进程对象类型的数据结构，做法是，首先用*!process*命令找到一个进程对象：

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
PROCESS 8a4ce668  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1001c88  HandleCount: 474.
    Image: System
```

然后，执行*!object*命令，用以上进程对象的地址作为参数：

```
kd> !object 8a4ce668
Object: 8a4ce668  Type: (8a4ceca0) Process
    ObjectHeader: 8a4ce650
    HandleCount: 2  PointerCount: 89
```

注意，该对象头是位于对象体开始之前的0x18（十进制24）字节。用下面的命令你可以查看对象头：

```
kd> dt _object_header 8a4ce650
nt!_OBJECT_HEADER
   +0x000 PointerCount     : 79
   +0x004 HandleCount      : 2
   +0x004 NextToFree       : 0x00000002
   +0x008 Type             : 0x8a4ceca0
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
```


www.TopSage.com

```
   +0x00f Flags             : 0x22 '"'
   +0x010 ObjectCreateInfo  : 0x80545620
   +0x010 QuotaBlockCharged : 0x80545620
   +0x014 SecurityDescriptor : 0xe10001dc
   +0x018 Body              : _QUAD
```

现在，从对象头数据结构的Type域中，你可以得到对象类型数据结构的地址，进一步用下面的命令可以查看对象类型的数据结构：

```
kd> dt _object_type 8a4ceca0
ntdll!_OBJECT_TYPE
   +0x000 Mutex             : _ERESOURCE
   +0x038 TypeList          : _LIST_ENTRY [ 0x8a4cecd8 - 0x8a4cecd8 ]
   +0x040 Name              : _UNICODE_STRING "Process"
   +0x048 DefaultObject     : (null)
   +0x04c Index             : 5
   +0x050 TotalNumberOfObjects : 0x30
   +0x054 TotalNumberOfHandles : 0x1b4
   +0x058 HighWaterNumberOfObjects : 0x3f
   +0x05c HighWaterNumberOfHandles : 0x1b8
   +0x060 TypeInfo          : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key               : 0x636f7250
   +0x0b0 ObjectLocks       : [4] _ERESOURCE
```

上面的输出显示出：对象类型结构包含了对象类型的名称，记录了这种类型活动对象的总数，以及这种类型的句柄和对象的尖峰数目。TypeInfo域保存了一个指针，该指针所指向的数据结构保存了对于该对象类型的所有对象都公共的属性，以及一组指向该对象类型的方法的指针：

```
kd> dt _object_type_initializer 8a4ceca0+60
ntdll!_OBJECT_TYPE_INITIALIZER
   +0x000 Length            : 0x4c
   +0x002 UseDefaultObject  : 0 ''
   +0x003 CaseInsensitive   : 0 ''
   +0x004 InvalidAttributes : 0xb0
   +0x008 GenericMapping    : _GENERIC_MAPPING
   +0x018 ValidAccessMask   : 0x1f0fff
   +0x01c SecurityRequired  : 0x1 ''
   +0x01d MaintainHandleCount : 0 ''
   +0x01e MaintainTypeList  : 0 ''
   +0x020 PoolType          : 0 ( NonPagedPool )
   +0x024 DefaultPagedPoolCharge : 0x1000
   +0x028 DefaultNonPagedPoolCharge : 0x288
   +0x02c DumpProcedure     : (null)
   +0x030 OpenProcedure     : (null)
   +0x034 CloseProcedure    : (null)
   +0x038 DeleteProcedure   : 0x805abe6e      nt!PspProcessDelete+0
   +0x03c ParseProcedure    : (null)
   +0x040 SecurityProcedure : 0x805cf682      nt!SeDefaultObjectMethod+0
   +0x044 QueryNameProcedure : (null)
   +0x048 OkayToCloseProcedure :  (null)
```


www.TopSage.com

在用户模式下不能操纵类型对象，因为对象管理器并没有提供任何有关类型对象的服务。然而，它们定义的有些属性通过特定的原生服务或者通过Windows API例程也是可见的。表3.6描述了在类型对象中保存的属性。

表3.6 类型对象的属性

属 性	用 途
类型名称	此种类型的对象的名称（“process”、“event”、“port”等等）
池类型	指明了这种类型的对象是从换页的还是非换页的内存中分配
默认的配额花费	默认从进程配额中扣除的换页内存池值和非换页内存池值
访问类型	当一个线程打开某个指向该类型的对象时可以请求的访问类型（“读”、“写”、“终止”、“挂起”等等）
通用访问权限的映射关系	在四种通用的访问权限（读、写、执行和全部）和特属于该类型的访问权限之间的映射关系
同步	指明了一个线程是否可以等待这种类型的对象
方法	在一个对象的生命期的特定点上，对象管理器自动调用的一个或者多个例程

同步是一个对于Windows应用程序可见的属性，它指的是，一个线程通过等待某个对象从一种状态改变成另一种状态，从而达到同步其执行过程的能力。一个线程可以对执行体的作业、进程、线程、文件、事件、信号量、互斥体和定时器对象进行同步。其他的执行体对象不支持同步。一个对象是否有能力支持同步，取决于该对象是否包含了一个内嵌的分发器对象，这是一种内核对象，本章后面“低IRQL的同步”小节中将会介绍此种类型的内核对象。

对象方法

表3.6中最后一个属性，即方法，是由一组内部例程构成的，这些例程类似于C++的构造函数和析构函数——当一个对象被创建或者销毁时自动被调用的例程。对象管理器扩展了这种思想，它也可以是在其他的一些场合下调用一个对象的方法，比如当有人打开或关闭一个指向某个对象的句柄，或者有人企图改变一个对象上的保护属性时。有些对象类型指定了这些方法，而其他的对象类型却不会，这要取决于一个对象类型将被如何使用。

当一个执行体组件创建了一个新的对象类型时，它可以向对象管理器注册一个或者多个方法。因此，对象管理器会在此种类型的对象的生命期过程中，某些明确定义的点上调用这些方法，通常是在一个对象被创建、删除或者以某种方式被修改的时候调用这些方法。对象管理器支持的方法如表3.7所示。

www.TopSage.com

表3.7 对象方法

方 法	何时该方法被调用
Open	当一个对象句柄被打开的时候
Close	当一个对象句柄被关闭的时候
Delete	在对象管理器删除一个对象之前
Query name	当一个线程请求在一个从属名字空间中查询一个对象（比如一个文件对象）的名称时
Parse	当对象管理器在一个从属名字空间中搜索一个对象名称时
Security	当一个进程读取或者改变一个对象（比如文件对象）在其从属名字空间中的保护属性时

对象管理器创建一个指向对象的句柄时，就会调用open方法；当一个对象被创建或者打开的时候，该方法都会被调用。然而，只有一个对象类型（Windowstation（窗口站））定义了Open方法。Windowstation对象类型要求有一个Open方法，这样，Win32k.sys能够与“可当做桌面内存池”的进程共享同一块内存。

Close方法的一个用法例子是在I/O系统中。I/O管理器为文件对象类型注册一个Close方法，对象管理器每次当关闭一个对象句柄时就会调用Close方法。此Close方法首先做检查，查看一下正在关闭该文件句柄的进程是否拥有任何用于该文件并且尚未完成的锁，如果拥有未完成的锁，则去除这些锁。检查文件锁并不是对象管理器所能完成或者应该由它来做的事情。

对象管理器在从内存中删除一个临时对象以前，会调用Delete方法，如果该方法已经被注册了的话。例如，内存管理器为内存区对象类型注册一个Delete方法，它会释放该内存区所使用的物理页面。它在删除一个内存区对象以前，也会验证一下内存管理器为该内存区所分配的任何内部数据结构都已经被删除了。同样，对象管理器不可能自己做这项工作，因为它对于内存管理器的内部工作一无所知。针对其他类型的对象的Delete方法也完成类似的功能。

Parse方法（类似于Query name方法）允许对象管理器把查找一个对象的控制权交给一个从属的对象管理器，如果它发现一个对象存在于对象管理器名字空间之外的话。当对象管理器查找一个对象名称的时候，如果它在搜索路径上碰到一个关联了Parse方法的对象，就会暂时挂起该搜索过程。对象管理器调用Parse方法，将当前正在查找的对象名称的剩余部分传递给它。除了对象管理器的名字空间以外，在Windows中还有两个名字空间：注册表名字空间和文件系统名字空间。注册表名字空间是配置管理器实现的，而文件系统名字空间则是I/O管理器在文件系统驱动程序的帮助下实现的（关于配置管理器的更多信息，请参见第5章；关于I/O管理器和文件系统驱动程序，请参见第9章）。

www.TopSage.com

例如，当一个进程打开一个名为\Device\Floppy0\docs\resume.doc的文件的句柄时，对象管理器遍历它的名称树，直到到达了名为Floppy0的设备对象。它看到该对象有一个关联的Parse方法，于是它调用此方法，并且将当前正在搜索的对象名称的剩余部分（在这个例子中，是字符串\docs\resume.doc）传递给它。设备对象的Parse方法是一个I/O例程，因为I/O管理器定义了设备对象类型，并且为它注册了一个Parse方法。I/O管理器的Parse例程接受名称字符串，并且将它传递给适当的文件系统，该文件系统会找到磁盘上的文件，并且打开此文件。

Security方法也是I/O系统使用的方法，它类似于Parse方法。一旦当一个线程试图查询或者改变那些用于保护一个文件的安全信息时，该方法就会被调用。这些安全信息对于文件和其他对象是不同的，因为安全信息是存储在文件对象中的，而不是在内存中。因此，必须要调用I/O系统才能找到安全信息，并且将它们读出来或者进行修改。

对象句柄和进程句柄表

当一个进程根据名称来创建或者打开一个对象时，它接收到一个句柄，以后通过此句柄来访问该对象。通过句柄来访问一个对象，要比直接使用名称来访问对象快得多，因为对象管理器可以跳过名称查找过程，而直接找到目标对象。进程也可以在其创建时刻，通过继承句柄的方式来获得句柄，（要求创建者在*CreateProcess*调用中指定了继承句柄的标志，并且一个句柄已被标记为可继承的，既可以是在该句柄被创建的时候进行标记，也可以事后通过Windows的*SetHandleInformation*函数来设定。）或者从另一个进程接收一个复制的句柄（参见Windows的*DuplicateHandle*函数）。

所有的用户模式进程在其线程使用一个对象以前，必须先拥有一个指向该对象的句柄。使用句柄来维护系统资源并不是一个新的想法。例如，C和Pascal（以及其他的语言）运行库会把已打开文件的句柄返回给应用程序。句柄被用做指向系统资源的间接指针。这一层间接性使得应用程序不用直接与系统数据结构打交道了。



> **注** **执行体组件和设备驱动程序可以直接访问对象，因为它们运行在内核模式下，因此可以访问系统内存中的对象结构。然而，它们必须显式地声明自己要使用某一个对象了，其做法是增加该对象的引用计数，这样做的结果是，它们在使用该对象的过程中它不会被释放掉（更多的细节，请参见本章稍后的“对象保持力”一节）。**

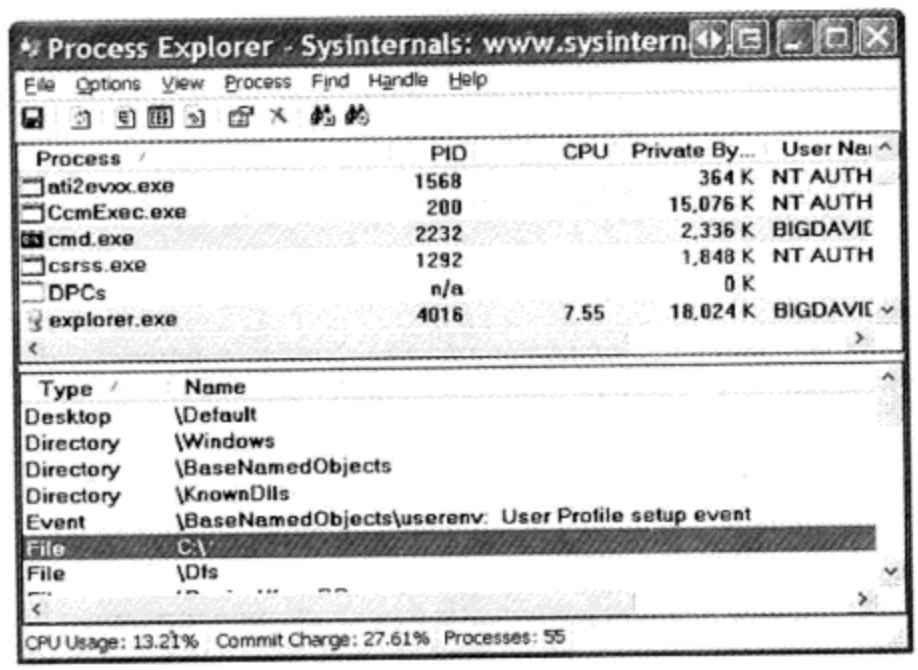
对象句柄还提供了额外的一些好处。第一，除了它们所指的内容不同以外，文件句柄、事件句柄和进程句柄没有区别。这种相似性使得可以用一个统一的接口来引用对象，而无须关心它们的类型。第二，对象管理器有独占的权限来创建句柄，以及找到一个句柄所指的对象。这意味着，对象管理器可以仔细地审查每一个可能会影响对象的用户模式动作，看一看调用者的安全轮廓是否允许在该对象上执行所请求的操作。

www.TopSage.com



实验：查看已打开的句柄

运行进程管理器，确保下面的窗格已打开，并且被配置成显示已打开的句柄（点击View，Lower Pane View，然后点击Handles）。然后打开一个命令提示窗口，查看一下新的Cmd.exe进程的句柄表。你应该可以看到一个打开的句柄指向当前的目录。例如，假设当前目录是C:\，则进程管理器应该显示如下：



如果接下来你用CD命令改变了当前目录，那么你在进程管理器中将会看到，指向原来当前目录的句柄被关闭了，而一个新的句柄被打开，它指向新的当前目录。原来的句柄被暂时地加亮显示成红色，而新的句柄被加亮显示成绿色。加亮显示的时间长短可以通过先点击Options，再点击Difference Highlight Duration来进行调整。

进程管理器使用不同的加亮显示特性，使得用户很容易看到句柄表中的变化。例如，如果一个进程正在泄漏句柄，那么，只需用进程管理器来查看句柄表就可以很快发现哪个句柄或者哪些句柄被打开了，但是没有被关闭。这些信息可以帮助程序员找到句柄泄漏的问题。

www.TopSage.com

利用*www.sysinternals.com*的命令行Handle工具，你也可以显示句柄表。例如，你可以注意到，以下Handle命令的部分输出显示了在改变当前目录之前和之后Cmd.exe进程的句柄表情况。

```
C:\>handle -p cmd.exe
Handle v2.2
Copyright (C) 1997-2004 Mark Russinovich
Sysinternals - www.sysinternals.com
------------------------------------------------------------
cmd.exe pid: 3184 BIGDAVID\dsolomon
   b0: File          C:\
C:\>cd windows
C:\WINDOWS>handle -p cmd.exe
.
.
cmd.exe pid: 3184 BIGDAVID\dsolomon
   b4: File          C:\WINDOWS
```

对象句柄是一个索引，指向与进程相关的**句柄表**（*handle table*）中的表项，执行体进程（EPROCESS）块（将在第6章中介绍）中有一个域指向进程的句柄表。第一个句柄索引是4，第二个是8，依此类推。一个进程的句柄表包含了所有已被该进程打开的那些对象的指针。句柄表的实现方式是一个三层结构，类似于x86内存管理单元实现的从虚拟地址到物理地址的转译方式；每个进程最多不超过16 000 000个句柄（关于x86系统中内存管理的更多细节，请参见第7章）。

在Windows 2000中，当一个进程被创建的时候，对象管理器分配了句柄表的最高层结构，其中包含了指向中间层表的指针；同时也分配了中间层，其中包含了第一个指向子句柄表的指针数组；还分配了最底层，其中包含了第一个子句柄表。图3.20显示了Windows 2000进程句柄表的结构。在Windows 2000中，对象管理器将一个对象句柄值的低24位看成三个8位域，分别索引到句柄表三层结构的每一层中。在Windows XP和Windows Server 2003中，只有最底层的句柄表才在创建进程的时候被分配好，而其他的层都是根据需要才被创建的。在Windows 2000中，子句柄表是由255个可用的表项构成的；而在Windows XP和Windows Server 2003中，子句柄表中包含的表项的数目等于能填满一页的表项数减去一项，此减去的一项被用于处理审计。例如，对于x86系统，一页是4096字节，除以一个句柄表项的大小（8字节），得到512，再减去1，因此，最底层句柄表中总共有511个表项。在Windows XP和Windows Server 2003中，中间层句柄表包含了一整页的指针来指向子句柄表，所以，子句柄表的数目取决于当前系统平台上页面的大小以及指针的大小。

www.TopSage.com

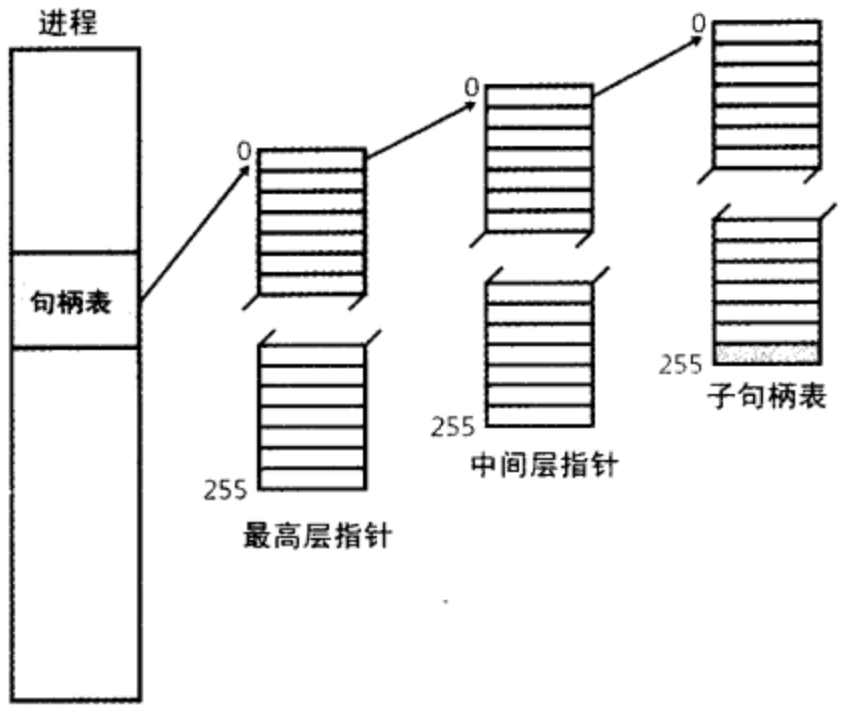


图3.20 Windows 2000进程句柄表的结构



实验：创建最多数量的句柄

*www.sysinternals.com/windowsinternals.shtml*的测试程序Testlimit有一个功能选项，可以不断地打开一个对象的句柄，直到不能再打开更多的句柄为止。你可以用该程序来检查在你的系统上，单个进程中能够创建多少个句柄。因为句柄表是从换页的内存池中分配的，所以你在到达一个进程所能创建的最多数量的句柄以前，可能会先耗光整个换页内存池。为了了解在你的系统上可以创建多少个句柄，请遵照下面的步骤。

1. 从刚才给出的链接处下载Testlimit.zip文件，并且将它解压缩到一个目录中。
2. 运行进程管理器，点击View，然后点击System Information。请注意换页内存池的当前大小和最大值（为了显示最大换页池的大小，必须要对进程管理器进行正确配置，以便可以访问内核映像文件Ntoskrnl.exe的符号）。让这一系统信息显示窗口保持运行，以便当你运行Testlimit程序时可以看到内存池的使用情况。
3. 打开一个命令提示窗口。
4. 运行Testlimit程序，打开“-h”开关（很容易做到，只需输入“**testlimit –h**”即可）。当Testlimit不能再打开一个新的句柄时，它就会显示出它所能打开的句柄的总数。如果该数目小于16兆左右，那么，你在碰到每个进程的句柄限制理论值之前，可能已经用光了换页内存池。
5. 关闭命令提示窗口，这样会杀死Testlimit进程，从而关闭所有已打开的句柄。

www.TopSage.com

如图3.21所示，在x86系统上，每个句柄项是一个包含2个32位成员的结构：一个指向对象的指针（以及一些标志），以及准许的访问掩码。在64位系统上，每个句柄表项12字节长：一个64位指针指向对象头，以及一个32位访问掩码（第8章将介绍访问掩码）。

在Windows 2000上，第一个32位成员包含了一个指向对象头的指针和四个标志。因为对象头总是8字节对齐的，所以这个域的最低3位可以被任意地用做标志位。表项的高位可以被用做一把锁。当对象管理器要把一个句柄转换成一个对象指针时，它在转换过程中会锁住该句柄项。因为所有的对象都是在系统地址空间中被分配内存的，所以对象指针的高位总是被置成1（可以保证该地址总是高于0x80000000，即使在打开了/3GB引导开关的系统上）。 因此，当一个句柄表项被解除了锁时，对象管理器可以保持该高位为0的状态，而在锁住一个表项的过程中，对象管理器可以设置该位，从而得到该对象的正确指针值。只有当一个进程创建一个新的句柄或者关闭一个已有的句柄时，对象管理器才需要利用一个与每个进程相关联的句柄表锁，将该进程的整个句柄表锁住。在Windows XP和Windows Server 2003中，对象指针的低位才是锁位。原来在Windows 2000中存放于低位的标志，现在被存放在访问掩码中一个未被使用的位中。

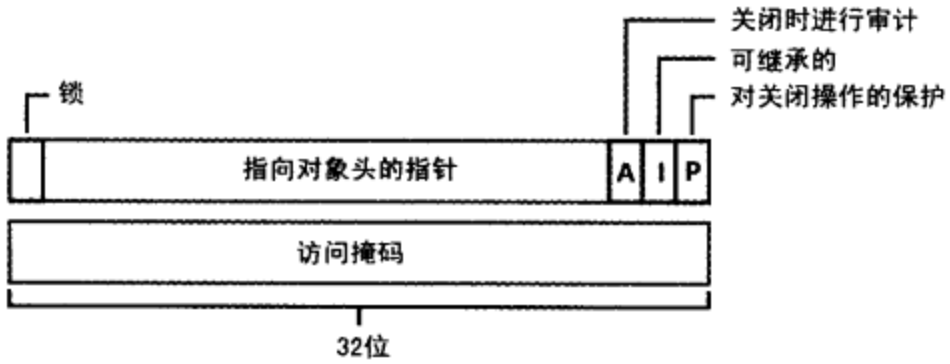


图3.21 句柄表项的结构

第一个标志说明了调用者是否允许关闭该句柄。第二个标志是继承标志，也就是说，它指明了该进程创建的子进程是否可以在它们的句柄表中有一份该句柄的拷贝。正如前面曾经提到过的，句柄继承性既可以在句柄被创建的时候指定，也可以稍后通过*SetHandleInformation*函数来指定（这一标志也可以用Windows的*SetHandleInformation*函数来指定）。第三个标志说明了关闭该对象时是否应该产生一个审计消息（这一标志并没有被暴露给Windows——对象管理器仅在内部使用该标志）。

系统组件和设备驱动程序通常需要打开一些不应该让用户模式应用程序访问的对象句柄。这可以通过在**内核句柄表**（在内部称为*ObpKernelHandleTable*）中创建句柄来实现。该表中的句柄只能在内核模式下才可以访问，但可以是在任何一个进程的环境下。这意味着，一个内核模式的函数可以在任何一个进程环境中引用这些句柄，而不会招致任何性能影响。

www.TopSage.com

对象管理器看到一个句柄的高位被设置时，就会将它识别成内核句柄表中的句柄，也就是说，指向内核句柄表中句柄的引用值大于0x80000000。在Windows 2000中，内核句柄表是一张完全独立的句柄表，但是在Windows XP和Windows Server 2003中，内核句柄表也被用做System进程的句柄表。



实验：利用内核调试器来查看句柄表

内核调试器中的*!handle*命令带三个参数：

```
!handle <句柄索引> <标志> <进程id>
```

这里，句柄索引指明了句柄表中的句柄项（零意味着显示所有的句柄）。第一个句柄是索引值4，第二个为8，依此类推。例如，输入“**!handle 4**”将显示当前进程的第一个句柄。

你可以用一个位掩码来指定标志，其中位0表明只显示该句柄项中的信息，位1表明显示空闲的句柄（而不仅仅已使用的句柄），位2表明显示与该句柄所指对象有关的信息。下面的命令显示了进程ID为0x408的进程的句柄表的全部细节信息。

```
kd> !handle 0 7 408

processor number 0
Searching for Process with Cid == 408
PROCESS 865f0790 SessionId: 0 Cid: 0408 Peb: 7ffdf000 ParentCid: 01dc
    DirBase: 04fd3000  ObjectTable: 856ca888  TableSize:  21.
    Image: i386kd.exe

Handle Table at e2125000 with 21 Entries in use
0000: free handle
0004: Object: e20da2e0  GrantedAccess: 000f001f
Object: e20da2e0  Type: (81491b80) Section
    ObjectHeader: e20da2c8
        HandleCount: 1  PointerCount: 1

0008: Object: 80b13330  GrantedAccess: 00100003 Object: 80b13330  Type: (81495100) Event
    ObjectHeader: 80b13318
        HandleCount: 1  PointerCount: 1
```

对象安全性

当你打开一个文件的时候，你必须要指定你的目的是读或是写。如果你在打开文件时指定了读访问，但又试图执行写操作，那么你就会得到一个错误。同样地，在执行体内部，当一个进程创建一个对象或者打开一个指向已有对象的句柄时，该进程必须要指定一组**期望的访问权限**（*desired access rights*），也就是说，它打算怎样操作该对象。它既可以请求一组适用于所有对象的标准访问权限（比如读、写和执行），也可以指定一组随着对象类型而有所不同的特

www.TopSage.com

殊访问权限。例如，一个进程可以请求对一个文件对象进行删除或者追加访问。类似地，它也可以要求能够挂起或者终止一个线程对象。

当一个进程打开一个对象句柄时，对象管理器调用**安全引用监视器**（*security reference monitor*），这是安全系统的内核模式部分，而且，对象管理器将该进程期望的一组访问权限传送给它。安全引用监视器检查该对象的安全描述符是否允许该进程所请求的访问类型。如果允许的话，引用监视器返回一组**准许的访问权限**（*granted access rights*），允许该进程得到这些权限，同时，对象管理器将这些权限存放在它所创建的对象句柄中。至于安全系统是如何确定谁可以访问哪些对象的，请参见第8章。

之后，无论何时当该进程的线程要使用此句柄时，对象管理器可以根据该线程调用的对象服务所隐含的用法，快速地检查这一组存储于句柄中的准许的访问权限。例如，如果调用者请求读访问一个内存区对象，但之后却调用一个服务对它进行写操作，则该服务就会失败。

对象保持力（Object Retention）

对象有两种类型：暂时的和永久的。大多数对象是暂时的，也就是说，它们只有在使用过程中才保留着，当不再需要的时候就会被释放掉。永久对象会一直保留着，直到被显式地释放掉为止。因为大多数对象是暂时的，所以本小节接下来部分介绍一下对象管理器是如何实现**对象保持力**（*object retention*）的，即只有当暂时对象还在被使用的时候，才会保留它们，而等到用完以后就会将它们删除。因为所有的用户模式进程在访问一个对象以前首先要打开一个指向该对象的句柄，所以，对象管理器可以很容易地跟踪有多少个进程正在使用一个对象，甚至于哪些进程正在访问它。跟踪这些句柄只是实现对象保持力的一部分。对象管理器通过两个阶段来实现对象保持力。第一个阶段称为**名称保持力**（*name retention*），它是受一个对象已有句柄的数目控制的。每次当一个进程打开一个对象的句柄时，对象管理器就会将该对象的头信息中的已打开句柄计数器增加1。当这些进程用完了该对象，并且关闭了指向它的句柄时，对象管理器就会相应地递减已打开句柄计数器。当该计数器减到0的时候，对象管理器从它的全局名字空间中删除该对象的名称。这样删除以后，可以避免新的进程再打开指向该对象的句柄。

实现对象保持力的第二阶段是，当对象不再有用的时候，停止保留对象本身（也就是说，删除它们）。因为操作系统代码通常通过指针而不是句柄来访问对象，所以，对象管理器必须要记录下它已经给操作系统进程分配了多少个对象指针。每次当它提供一个指向该对象的指针时，它就会递增一个专用于该对象的**引用计数**（*reference count*）；当内核模式组件用完了该指针时，它们会调用对象管理器以递减该对象的引用计数。当系统递增句柄计数的时候，它也会递增该引用计数，同样地，当句柄计数递减的时候，它也会递减引用计数，因为对于这种必须跟踪的对象来说，句柄也是它的引用（有关对象保持力的进一步细节，请参见DDK文档中的*ObReferenceObjectByPointer*和*ObDereferenceObject*函数）。

www.TopSage.com

图3.22显示了两个正在使用中的事件对象。进程A打开了第一个事件，进程B同时打开了这两个对象。而且，第一个事件同时还被某一个内核模式的结构引用了。因此，引用计数为3。所以，即使进程A和B都关闭了指向第一个事件对象的句柄，第一个事件对象仍然会存在，因为它的引用计数为1。然而，当进程B关闭了指向第二个事件对象的句柄时，该对象将被释放掉。

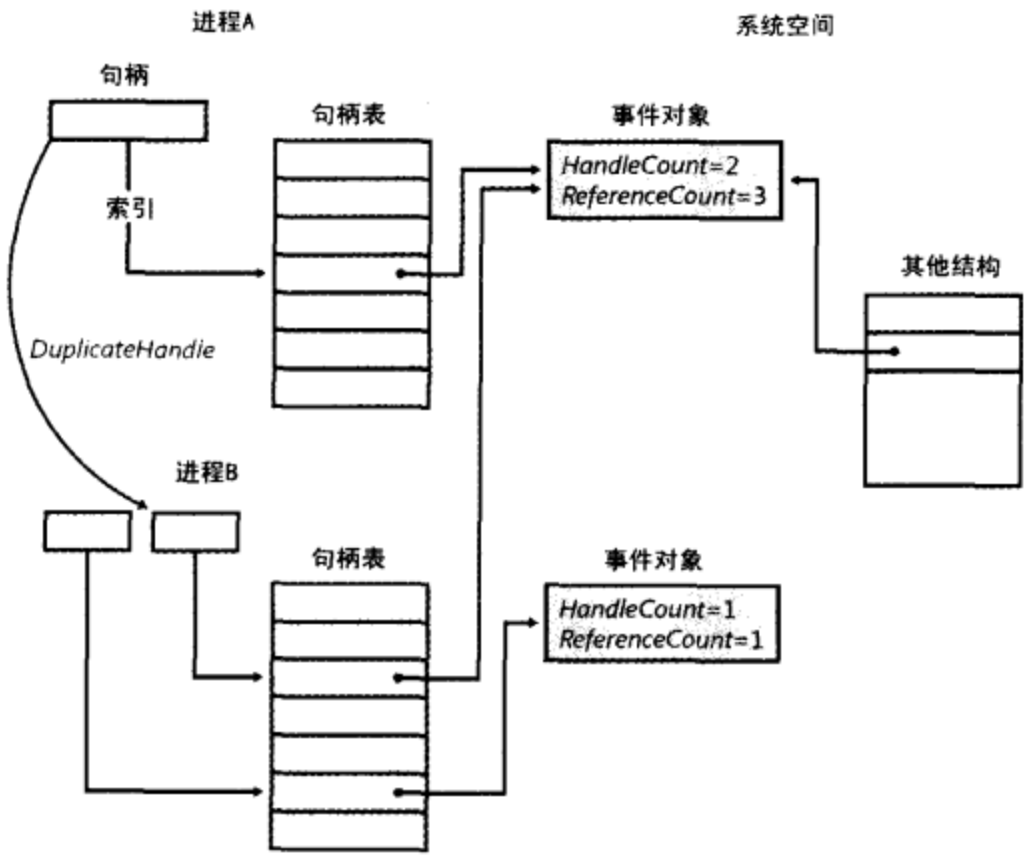


图3.22 句柄和引用计数

所以，即使当一个对象的已打开句柄计数器到达了0，该对象的引用计数可能仍然是一个正数，表明了操作系统仍然在使用该对象。最终，当引用计数减到0的时候，对象管理器就会将它从内存中删除。

由于对象保持力的工作方式，一个应用程序只需简单地保持一个已打开的句柄，并指向某个对象，就可以确保该对象和它的名称仍然在内存中。如果程序员编写的应用程序包含了两个或者多个相互协作的进程，则他们无需担心一个进程正在使用一个对象时另一个进程是否已经删除了该对象。例如，一个进程可能创建了第二个进程，让它在后台执行一个程序，然后前者立即关闭了指向第二个进程的句柄。因为操作系统需要第二个进程来运行该程序，所以它维护了一个指向该进程对象的引用。只有当后台程序完成了它的执行任务时，对象管理器才会递减第二个进程的引用计数，然后将它删除。

www.TopSage.com

资源记账

资源记账（resource accounting），如同对象保持力一样，与对象句柄用法的关系非常密切。一个正的已打开句柄计数值表明了某个进程正在使用该资源。它也表明，必定某个进程承担了该对象所占用的内存消耗。当一个对象的句柄计数和引用计数减到0的时候，原先使用该对象的那个进程应该不用再承担这些内存消耗了。

许多操作系统使用一个配额系统（quota system）来限制进程对系统资源的访问。然而，在进程上强加的配额类型有时候是多样而又复杂的，并且，跟踪配额的代码散布在整个操作系统中。例如，在有些操作系统中，I/O组件可能会记录和限制每个进程所能打开的文件数量，而内存组件可能会限制一个进程的线程所能分配的内存数量。进程组件可能会限制一个用户所能创建的新进程的数量不得超过某个最大值，或者限制一个进程内部的线程数量不得超过某个最大值。操作系统分别在不同的地方跟踪和强加这些限制值。

与此相反，Windows的对象管理器提供了一个中心设施来实现资源记账。每个对象头都包含了一个称为**配额花费**（*quota charge*）的属性，其中记录了当一个进程的线程打开一个指向该对象的句柄时，对象管理器从该进程在换页池和/或非换页池中分配得到的配额中该减去多少。

Windows中的每个进程都指向一个配额数据结构，其中记录了该进程在非换页池、换页池和页面文件中使用量的限制值和当前值（在内核调试器中输入**dt nt!_EPROCESS_QUOTA_ENTRY**就可以看到此数据结构的格式）。这些配额默认为0（表示无限制），但通过修改注册表值可以重新指定它们（参见HKLM\System\CurrentControlSet\Session Manager\Memory Management下面的*NonPagedPoolQuota*、*PagedPoolQuota*和*PagingFileQuota*）。请注意，一个交互会话中所有的进程共享同样的限额块（没有任何文档介绍了有哪一种方法可以创建一个具有自己特有限额块的进程）。

对象名称

在创建一群对象时，一个重要的考虑是，有必要设计一个成功的系统来跟踪记录这些对象。对象管理器要求以下的信息以便帮助你做到这一点：

- 一种用以区分此对象与彼对象的方法；
- 一种找到并获得一个特定对象的办法。

只要允许为对象命名，第一个需求就可以得到满足。这是大多数操作系统所提供能力的一种扩展——能够对所选择的资源（比如文件、管道或者一块共享内存）进行命名。相反地，执行体允许一个对象所代表的任何资源都有一个名称。这里的第二个需求，即找到并获得一个对象，也可以通过对象名称来满足。如果对象管理器按照名称来存储对象，那么它只需查找对象的名称就可以找到它。

www.TopSage.com

对象名称还可以满足第三个需求，即允许进程之间共享对象。执行体的对象名字空间是全局的，对于系统中的所有进程都是可见的。第一个进程创建一个对象，并且将它的名称放到全局名字空间中，第二个进程在打开对象句柄的时候只要指定该对象的名称就可以得到一个指向该对象的句柄。如果一个对象并不需要以这种方式共享，则它的创建者不必为它指定名称。

为了提高效率，对象管理器并不是在每次有人使用一个对象的时候都要查找该对象的名称。相反，它只在两种情况下才查找一个名称。第一种情况是当一个进程创建一个命名的对象时，对象管理器在把新的名称存储到全局名字空间中以前，首先查找此名称，以验证在全局名字空间中它原来并不存在。第二种情况是，当一个进程打开一个句柄，指向一个命名的对象时，对象管理器查找该名称，找到该对象，然后给调用者返回一个对象句柄。因此，调用者使用此句柄来引用该对象。当查找一个名称的时候，对象管理器允许调用者指定大小写敏感的搜索，或者指定大小写不敏感的搜索。这种特性是为了支持POSIX或者其他使用大小写敏感文件名的环境。

对象的名称存储在哪里取决于对象的类型。表3.8列出了在所有的Windows系统上都能找得到的标准对象目录，以及哪些类型的对象将它们的名称存储在这些目录下。在所列出的目录中，只有\BaseNamedObjects和\GLOBAL??（在Windows 2000上是\??）对于用户程序是可见的（参见本章后面的“会话名字空间”部分可以获得更多的信息）。

因为诸如互斥体、事件、信号量、可等待的定时器，以及内存区这些基本内核对象将它们的名称存储在同一个对象目录下，所以，这些对象中的任何两个都不能有同样的名称，即使它们是不同类型的对象。这一限制更加强调了在选择名称的时候务必要谨慎，以便不会与其他的名称发生冲突（例如，可以使用你公司和产品的名字作为前缀）。

表3.8 标准对象目录

目 录	所存储对象名字的类型
\GLOBAL??（在Windows 2000下是\??）	MS-DOS设备名（\DosDevices是一个指向此目录的符号链接）
\BaseNamedObjects	互斥体、事件、信号量、可等待的定时器和内存区对象
\Callback	回调对象
\Device	设备对象
\Driver	驱动程序对象
\FileSystem	文件系统驱动程序对象和文件系统识别器设备对象
\KnownDlls	已知DLL（在启动时候由系统映射的DLL）的内存区名称和路径
\Nls	已映射的国家语言支持表的内存区名称
\ObjectTypes	对象的类型的名称
\RPC Control	远过程调用（RPC）所使用的端口对象

www.TopSage.com

续表

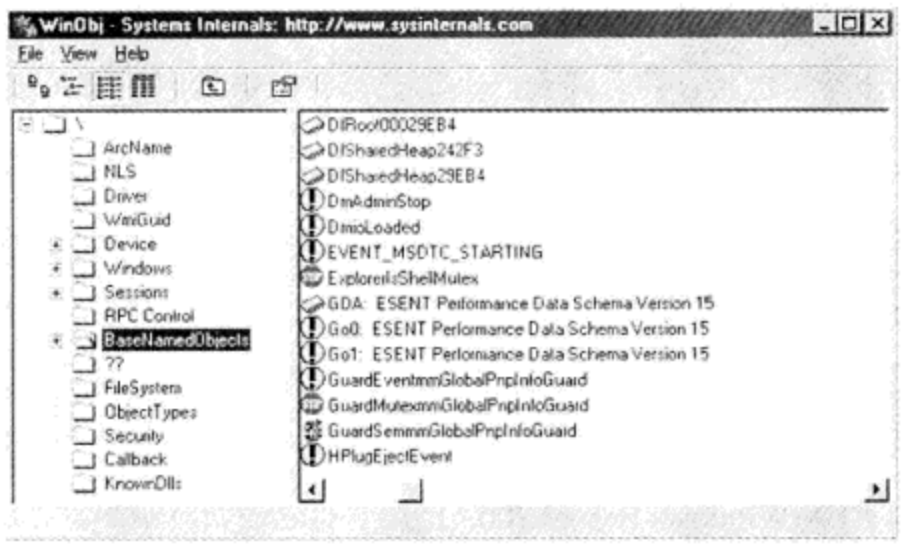
目 录	所存储对象名字的类型
\Security	与安全子系统相关的对象的名称
\Windows	Windows子系统的端口和窗口站

对象的名称相对于一台计算机而言（或者，相对于一台多处理器计算机上的所有处理器来说）是全局的，但是它们跨越网络是不可见的。然而，对象管理器解析名称的方法使得有可能访问其他计算机上的命名对象。例如，提供了文件对象服务的I/O管理器，将对象管理器的功能扩展到了远程文件上。当请求打开一个远程文件对象的时候，对象管理器调用一个解析（parse）方法，这使得I/O管理器有机会截取此请求，并且将它递交给一个网络重定向器，这里的网络重定向器是一个可跨越网络访问文件的驱动程序。远程Windows系统上的服务器代码调用它本地系统上的对象管理器和I/O管理器，以便找到所请求的文件对象，并且将信息通过网络返回去。



实验：查看基本的命名对象

利用*www.sysinternals.com*的Winobj工具，你可以看到那些有名称的基本对象的列表。运行Winobj.ext，并且点击\BaseNamedObjects，如下图所示：



命名的对象显示在右侧。其中的图标指明了对象类型。

- 互斥体被显示成一个停止符号；
- 内存区（Windows文件映射对象）被显示成内存芯片；
- 事件对象被显示成惊叹号；
- 信号量的图标像是一个交通信号灯；
- 符号链接的图标是弯曲的箭头。

www.TopSage.com

对象目录 对象目录对象是对象管理器支持这种层次型命名结构的手段。此对象与文件系统中的目录非常相似，它包含了其他对象的名称，甚至可能还有其他的对象目录。对象目录对象维护了足够多的信息以便将这些对象名称转译为指向这些对象本身的指针。对象管理器利用这些指针来构造出对应的对象句柄，然后可将这些对象句柄返回给用户模式的调用者。无论是内核模式的代码（包括执行体组件和设备驱动程序），还是用户模式的代码（比如子系统），都可以创建对象目录，并且在其中存储对象。例如，I/O管理器创建了一个名为\Device的对象目录，其中包含了各个代表I/O设备的对象的名称。

符号链接 在某些特定的文件系统（例如，在NTFS和某些UNIX系统上）中，通过符号链接，用户可以创建一个文件名或者一个目录名，当它被使用的时候，实际上被操作系统转译成另一个不同的文件或者目录名。使用符号链接，是一种让用户间接地共享一个文件或者目录中的内容的简便方法，也允许用户在普通的层次型目录结构中创建一些跨越不同目录的交叉链接。

对象管理器实现了一个称为**符号链接对象**（*symbolic link object*）的对象，它完成的功能类似于对象名称在其对象名字空间中所起的作用一样。符号链接可以出现在对象名字符串中的任何地方。当调用者引用了一个符号链接对象的名称时，对象管理器遍历它的对象名字空间，直至找到该符号链接对象。它检查该符号链接，并且找到一个取代该符号链接名的字符串。然后它重新开始查找名称的过程。

执行体使用符号链接对象的一个场所是，在将MS-DOS风格的设备名翻译成Windows内部设备名的地方。在Windows中，用户使用诸如A:、B:、C:这样的名称来引用软盘和硬盘驱动器，以及使用COM1、COM2等这样的名称来引用串行端口。Windows子系统将这些符号链接对象放到对象管理器的名字空间中，从而使它们变成受保护的全局数据；在Windows 2000上它们位于\??对象目录下，在Windows XP和Windows Server 2003上位于\Global??目录下。

会话名字空间

Windows NT最初编写的时候，有两个假设，即只有一个用户可以通过交互方式登录到系统中，以及对于任何一个交互式应用程序，系统只运行它的一个实例。在Windows 2000 Server中引入了Windows终端服务，以及在Windows XP中引入了快速用户切换以后，这些假设都发生了变化，因此对象管理器的名字空间模型也要求随之发生变化，以支持多个用户（关于终端服务和会话的初步介绍，请参见第1章）。

www.TopSage.com

一个登录到控制台会话上的用户可以访问**全局**名字空间（*global* namespace），此名字空间是第一个名字空间实例。另外的会话都可以获得该名字空间的一个私有视图，称为**局部**名字空间（*local* namespace）。在名字空间中属于每个会话的局部部分包括\DosDevices、\Windows和\BaseNamedObjects。将名字空间中同样的部分做成独立的拷贝，这称为名字空间**实例化**。实例化\DosDevices之后，每个用户都有可能拥有不同的网络驱动器字母，以及不同的Windows对象（诸如串行端口等）。在Windows 2000上，全局的\DosDevices目录被命名为\??，它是\DosDevices符号链接所指向的目录；而局部\DosDevices目录则是由该终端服务器会话的会话ID来标识的。在Windows XP及以后的版本上，全局\DosDevices目录被命名为\Global??，它也是\DosDevices符号链接所指的目录，而局部\DosDevices目录则是由登录会话ID来标识的。

\Windows目录是Win32k.sys创建交互式窗口站\WinSta0的地方。一个终端服务环境可以支持多个交互式用户，但是，每个用户需要一个单独的WinSta0版本，以便做到在感觉上他或她正在访问Windows中预设的交互式窗口站。最后，应用程序和系统创建的共享对象位于\BaseNamedObjects中，包括事件、互斥体和内存区。如果两个用户都在运行的应用程序创建了一个命名的对象，那么，每个用户会话必须拥有该对象的一个私有版本，以便该应用程序的两个不同实例在访问此同一对象时不会相互干扰。

对象管理器在\Session\X（这里X是会话标识符）下的一个与用户会话相关联的目录下，创建了上述三个目录的私有版本，从而实现了局部的名字空间。例如，当远程会话2中的一个Windows应用程序创建一个命名的事件时，对象管理器以透明的方式将对象的名称从\BaseNamedObjects重定向到\Sessions\2\BaseNamedObjects中。

所有与名字空间管理有关的对象管理器函数都知道这些实例化的目录，并且也提供了透明性支持，即非控制台会话可以与控制台会话使用同样的名字空间。Windows子系统DLL将Windows应用程序传递过来的位于\DosDevices中的对象引用加上一个\??前缀（例如，C:\Windows变成了\??\C:\Windows）。当对象管理器看到了特殊的\??前缀时，它所采取的步骤取决于Windows的版本，但是不管怎样，它总是要依赖于执行体进程对象（EPROCESS，在第6章中有进一步介绍）中一个名为*DeviceMap*的域（该域所指的数据结构是该进程与同一会话中其他进程共享的）。DeviceMap结构中的*DosDevicesDirectory*域所指的对象管理器目录代表了该进程的局部\DosDevices。根据不同的系统，此目标目录也分别有以下的不同。

- 如果当前系统为Windows 2000，并且没有安装终端服务，则该进程的DeviceMap结构中的*DosDevicesDirectory*域指向\??目录，因为系统中没有局部名字空间。

www.TopSage.com

- 如果当前系统为Windows 2000，并且安装了终端服务，那么，当一个新的会话变成活动会话时，系统将全局的\??目录中所有的对象拷贝到该会话的局部\Devices目录中，并且DeviceMap结构中的*DosDevicesDirectory*域指向此局部目录。
- 在Windows XP和Windows Server 2003上，系统并没有将全局对象拷贝到局部的DosDevices目录中。当对象管理器看到一个\??引用时，它通过DeviceMap的*DosDevicesDirectory*域找到该进程的局部\DosDevices目录。如果对象管理器在该目录中没有找到目标对象，那么，它检查该目录对象的DeviceMap域，如果它是有效的，则在DeviceMap结构的GlobalDosDevicesDirectory域所指的目录（总是\Global??）中查找目标对象。

在特定的情况下，一些能感知终端服务的应用程序，即使它们运行在远程会话中，也需要访问控制台会话中的对象。应用程序之所以要这么做，可能是想与运行在其他远程会话中的实例同步进行，或者与控制台会话本身同步进行。针对这些情形，对象管理器提供了特殊的改写符“\Global”，应用程序在任何对象名的前面加上此前缀，就可以访问全局名字空间。例如，会话2中的应用程序打开一个名为\Global\ApplicationInitialized的对象，则它被定向到\BasedNamedObjects\ApplicationInitialized，而不是\Sessions\2\BaseNamedObjects\ ApplicationInitialized。

在Windows XP和Windows Server 2003中，如果应用程序想要访问全局\DosDevices目录中的对象，那么，只要在它的局部\DosDevices目录中不存在这样的对象，它就不需要使用\Global前缀。这是因为，对象管理器如果在局部目录中没有找到该对象的话，它就会自动地在全局目录中查找该对象。然而，如果一个应用程序运行在安装了终端服务的Windows 2000上，那么它必须总是指定\Global前缀，以便访问全局\DosDevices目录中的对象。

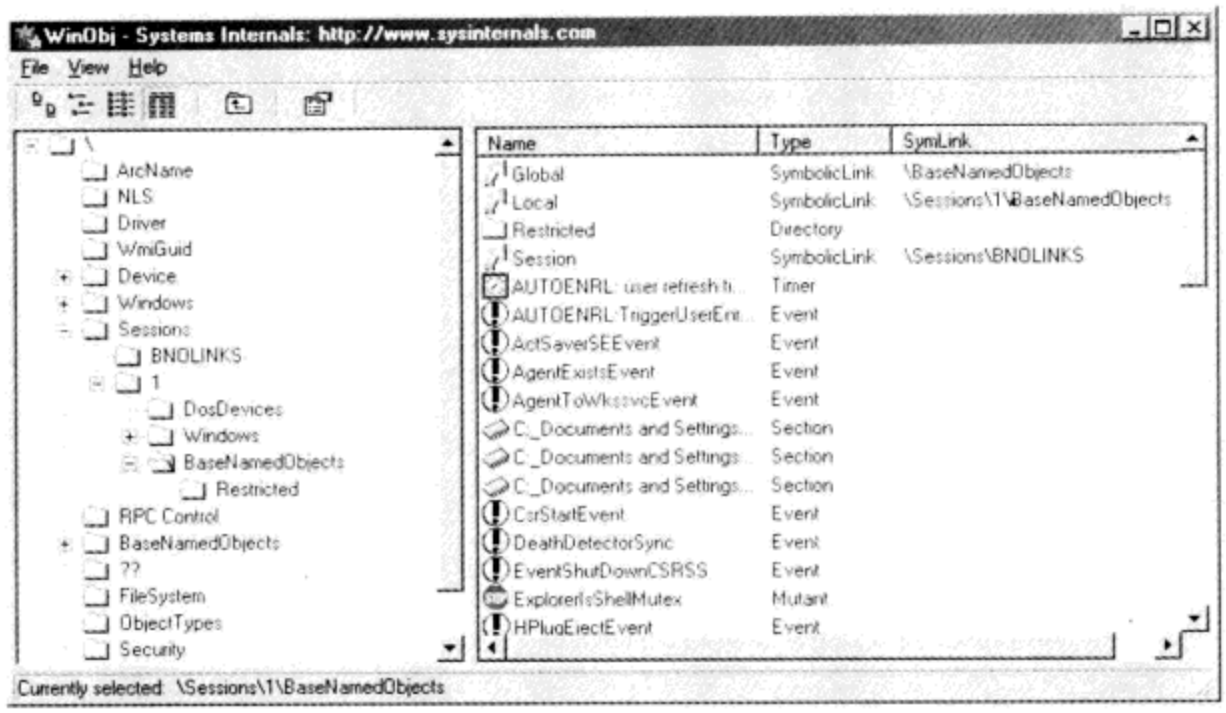


实验：查看名字空间的实例化

你只需先创建一个非控制台会话，然后检查该会话中某个进程的句柄表，就可以看到名字空间的对象管理器实例。在Windows XP的Home Edition或者在一个未加入域的Windows XP Professional系统上，注销控制台会话，（做法是，先点击Start菜单，再点击Log Off，并且选择Disconnect and Switch User，或者直接按下Windows键+L。）然后登录到一个新的账户中。如果你有Windows 2000 Server、Advanced Server或者Datacenter Server系统，则运行终端服务客户程序，并且连接到服务器上，再登录进去。

www.TopSage.com

如果你已经登录到新的会话中，则运行从*www.sysinternals.com*下载的Winobj.exe，并且点击\Sessions目录。你将会看到一个子目录，其中针对每个活动的远程会话都有一个数字名。如果你打开这些目录中的某一个，将会看到名为\DosDevices、\Windows和\BaseNamedObjects的子目录，它们是该会话的局部名字空间子目录。下面的屏幕截图显示了一个局部名字空间。



接下来运行进程管理器，并且选择一个位于新会话中的进程（比如Explorer.exe），然后查看它的句柄表（点击View、Lower Pane View，然后点击Handles）。你应该可以看到一个指向\Session*n*之下的\Windows\Windowstations\WinSta0的句柄，这里*n*是会话ID。具有全局名称的对象应该出现在\Sessions\n\BaseNamedObjects的下面。

3.3 同步

在操作系统的发展过程中，**互斥**的概念是非常重要的。它指的是，保证任何时候只有一个线程可以访问某一特定的资源。当一个资源不允许共享访问，或者共享访问将导致不可预测的后果时，互斥是必需的。例如，如果两个线程同时拷贝一个文件到打印机端口上，则它们的输出有可能被混杂在一起。类似地，如果一个线程在读某个内存位置时另一个线程正在往里写数据，则第一个线程有可能读取到不可预测的数据。一般地，可写的资源在没有限制的情况下是不能被共享的，而不会被修改的资源则可以被共享。图3.23演示了当两个运行于不同处理器上的线程同时向一个循环队列写数据时可能发生的情形。

www.TopSage.com

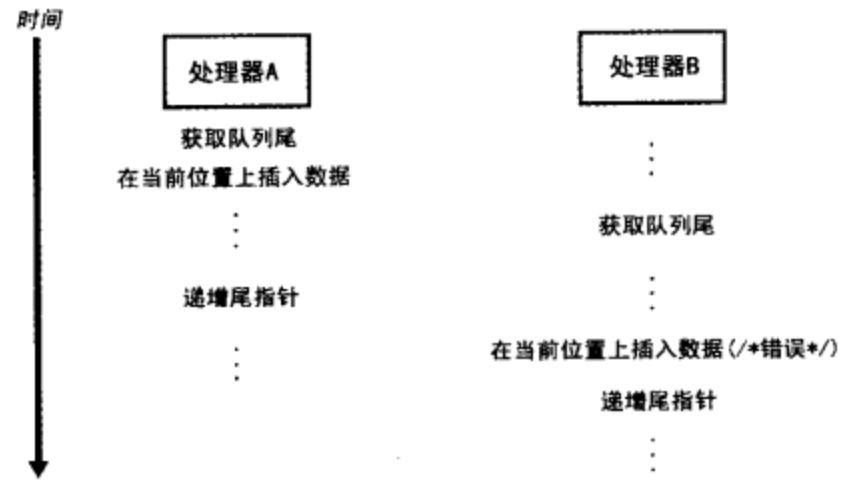


图3.23 不正确的内存共享

因为在第一个线程尚未更新队列的尾指针之前，第二个线程就获得了尾指针的值，所以第二个线程将数据插入到第一个线程刚刚使用过的位置上，从而改写了该位置上的数据，同时留下了一个空的队列元素。虽然此图演示的是在一个多处理器系统上可能发生的情形，但同样的错误也可能在单处理器上发生：如果在第一个线程更新队列尾指针之前操作系统执行了一个环境切换，将控制权交给了第二个线程，则同样的错误就会发生。

如果一段代码区访问了一个不可共享的资源，则这样的代码区称为**临界区**（*critical section*）。为了确保正确的代码，同一时刻只允许一个线程在临界区内执行。当一个线程在写一个文件、更新一个数据库，或者修改一个共享变量的时候，其他的线程不允许访问同样的资源。图3.23中显示的伪代码是一个临界区，它在毫无互斥的情况下，不正确地访问了一个共享的数据结构。

尽管互斥问题对于所有的操作系统都很重要，但是对于一个像Windows这样的**紧耦合的、对称多处理**（*SMP*）操作系统更显得尤为重要，在这样的系统中，同样的系统代码同时运行在多个处理器上，它们共享了存储在全局内存中的特定数据结构。在Windows中，内核负责提供各种机制，供系统代码用于避免两个线程同时修改同样的数据结构。内核提供的互斥原语使得它自己和执行体的其他部分可以利用这些原语来同步它们对于全局数据结构的访问。

因为在DPC/Dispatch级别的IRQL上，调度器已经对其数据结构的访问进行了同步，所以，当IRQL在DPC/Dispatch或者更高的级别（称为**提高的或者高**IRQL级别）上时，内核和执行体不能依赖于那些可能会导致页面错误或者重新调度操作的同步机制，来对各种数据结构同步访问。在下面的小节中，你将会看到，当IRQL高的时候，内核和执行体如何使用互斥机制来保护它的全局数据结构；而当IRQL低的时候（低于DPC/Dispatch级别），内核和执行体又用到了哪些互斥和同步机制。

www.TopSage.com

高IRQL的同步

在内核执行的各个阶段中，内核必须要保证，在临界区内部同一时刻只有一个处理器在执行。内核临界区是指修改某个全局数据结构的代码段，比如修改内核的分发器数据库或者它的DPC队列。除非内核能保证所有的线程都按照互斥的方式来访问这些数据结构，否则操作系统不可能正确地工作。

值得关注的最大区域是中断。例如，当中断发生的时候，内核可能正在更新一个全局数据结构，而该中断的处理例程可能也要修改此数据结构。简单的单处理器操作系统有时候采用一种简便的办法来避免发生这样的情形，即每次当它们要访问全局数据的时候就禁止所有的中断，不过，Windows的内核采用了一种更为复杂的方案。在使用一个全局资源以前，内核临时屏蔽掉那些在中断处理例程中也用到了该资源的中断。它的做法是，将处理器的IRQL提升到任何有可能访问该全局数据的中断源所用到的最高IRQL级别。例如，一个位于DPC/Dispatch级别的中断会触发分发器运行，而分发器用到了分发器数据库。因此，在内核中凡是用到了分发器数据库的代码部分都将IRQL提升到DPC/Dispatch级别上，在使用分发器数据库之前屏蔽掉DPC/Dispatch级别的中断。

这种策略对于单处理器系统是非常合适的，但是对于一个多处理器的系统还是不够的。将一个处理器上的IRQL提升起来，并不会阻止在另一个处理器上发生中断。内核也需要保证在跨越几个处理器的情况下实现互斥访问。

互锁操作

同步机制的最简单形式，莫过于直接依赖硬件上对于多处理器安全操作整数值的支持，以及针对执行比较操作的支持。这些同步机制包括诸如*InterlockedIncrement*、*InterlockedDecrement*、*InterlockedExchange*和*InterlockedCompareExchange*等函数。例如，*InterlockedDecrement*函数在减操作过程中，利用x86的*lock*指令前缀（比如*lock xadd*）来锁住多处理器总线，因此，如果另一个处理器也要修改这一被减内存单元，它就不可能在减操作读取原始值和写入结果值的过程中间修改此内存单元。内核和驱动程序用到了这种最基本的同步形式。

www.TopSage.com

自旋锁

内核用来实现多处理器互斥的机制称为**自旋锁**（*spinlock*）。自旋锁是一个与某个全局数据结构相关联的锁原语，图3.24中与自旋锁关联的数据结构是DPC队列。

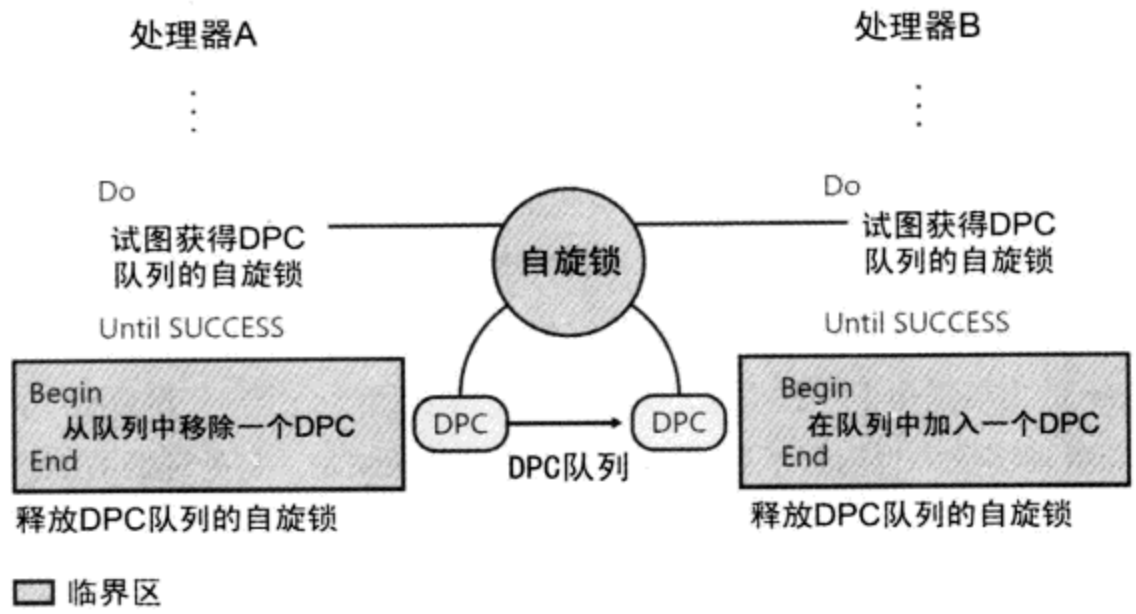


图3.24 使用自旋锁

在进入如图中所示的临界区以前，内核必须先获得与被保护的DPC队列相关联的自旋锁。如果自旋锁并非空闲的话，则内核一直尝试着获取该锁，直到成功为止。自旋锁之所以得名“（自）旋转”的由来是在于这样的事实：即使内核（因此也即处理器）在得到锁之前也被置于边界状态。

自旋锁如同它们所保护的数据结构一样，也驻留在全局内存中。获取和释放一个自旋锁的代码是用汇编语言来编写的，一方面是为了加快速度，另一方面也是要充分发掘底层处理器体系结构所提供的锁机制。在许多体系结构上，自旋锁是通过硬件支持的test-and-set操作来实现的，即在一条原子指令内测试一个锁变量的值并且获得该锁。在单条指令内测试和获取一个锁可以避免第二个线程在“第一个线程测试锁变量的时间点”与“它获得锁的时间点”之间抓取到该锁。

在Windows中，所有的内核模式自旋锁都有一个与之关联的IRQL，并且它总是在DPC/Dispatch或者更高的级别上。因此，当一个线程试图获得一个自旋锁的时候，该处理器上凡是在该自旋锁的IRQL或者更低级别上的所有其他活动都要停止下来。因为线程分发动作也工作在DPC/Dispatch级别上，所以，如果一个线程持有一个自旋锁，则它永远也不会被抢占，因为此级别的IRQL屏蔽了线程分发机制。如果一段代码正在执行一个受自旋锁保护的临界区，则这种屏蔽能力使得该段代码可以继续执行，从而它可以更快地释放该自旋锁。内核在使用自旋锁的时候非常小心，尽可能地使它在持有一个自旋锁的过程中执行最少数量的指令。

www.TopSage.com



> **注** 因为IRQL在单处理器上是一种有效的同步机制，所以，单处理器HAL的自旋锁获取和释放函数并没有实现自旋锁——它们只是简单地提升或者降低IRQL。

内核提供了一组函数，比如*KeAcquireSpinlock*和*KeReleaseSpinlock*，从而使得执行体的其他部分也可以使用自旋锁。例如，设备驱动程序为了保证设备寄存器和其他的全局数据结构在同一时刻只能被该设备驱动程序的某一部分（并且只能由一个处理器）访问，就可以请求自旋锁。自旋锁不是给用户程序使用的——用户程序应该使用下一小节讲述的同步对象。

对于使用内核自旋锁的代码，自旋锁也带来了一些限制。因为自旋锁总是有一个DPC/Dispatch或更高级别的IRQL，所以，正如上文所述的，如果一段代码正在持有一个自旋锁，若它企图让调度器执行一个分发操作，或者它引发了一个页面错误，则会导致系统崩溃。

排队的自旋锁

有一种特殊的自旋锁称为**排队的自旋锁**（*queued spinlock*），它们被用于某些非标准自旋锁的场合下。排队的自旋锁是自旋锁的一种形式，它们在多处理器上比标准的自旋锁有更好的伸缩性。一般而言，如果在特定的场合下，Windows预计对锁的竞争程度会很低，那么它只使用标准的自旋锁。

排队的自旋锁的工作方式如下：当一个处理器想要获得一个当前已被其他处理器持有的排队的自旋锁时，它把自己的标识符放在与该自旋锁关联的一个队列中。如果当前正持有该自旋锁的处理器释放了该锁，则它将该锁移交给队列中标志的第一个处理器。同时，如果一个处理器正在等待一个忙着的自旋锁，则它并不是检查该自旋锁本身的状态，而是检查一个针对每个处理器的标志；在队列中位于该处理器之前的处理器会设置这一标志，以表明该轮到这个正在等待的处理器了。

排队的自旋锁的自然结果是，它们在这些针对每个处理器的标志上旋转，而不是在全局自旋锁上旋转。这有两种效果。第一是，多处理器的总线不会因为处理器之间的同步而招致繁重的流量。第二是，对于一组正在等待获取某个自旋锁的处理器，排队的自旋锁强加了先进先出（FIFO）的获取顺序，而不是随机选择一个处理器。FIFO顺序意味着在一组访问同样锁的处理器之间有了更加一致的性能。

Windows定义了许多全局排队的自旋锁，并且在每个处理器的“**处理器控制区域**（*PCR*）”所包含的一个数组中保存了指向这些全局排队的自旋锁的指针。只需在调用*KeAcquireQueuedSpinlock*的时候将一个全局自旋锁的指针在PCR数组中的索引传递进去，就可以获得对应的全局自旋锁。全局自旋锁的数量随着操作系统的每个发行版本而不断增加，DDK头文件Ntddk.h中公开了这些全局自旋锁的索引定义表。

www.TopSage.com



实验：查看全局排队自旋锁

通过使用内核调试器命令*!qlock*，你可以看到全局排队的自旋锁（在每个处理器PCR的排队的自旋锁数组中所指向的自旋锁）的状态。该命令仅仅在多处理器系统上才有意义，因为单处理器HAL并没有实现自旋锁。下面的例子是在一个Windows 2000系统上完成的。在这个例子中，分发器数据库的排队自旋锁由处理器1持有，其他排队的自旋锁尚未有人获取（第6章讲述了分发器数据库）。

```
kd> !qlocks
Key: O = Owner, 1-n = Wait order, blank = not owned/waiting, C = Corrupt

                         Processor Number
    Lock Name      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

KE - Dispatcher       O
KE - Context Swap
MM - PFN
MM - System Space
CC - Vacb CC - Master
```

栈内排队自旋锁

除了使用全局定义的静态排队自旋锁以外，Windows XP和Windows Server 2003的内核还提供了*KeAcquireInStackQueuedSpinlock*和*KeReleaseInStackQueuedSpinlock*两个函数来支持动态分配的排队自旋锁。有几个组件——包括缓存管理器、执行体内存池管理器和NTFS——也用到了这些类型的锁，第三方驱动程序编写者需要用到的函数在DDK中有相应的文档介绍。

*KeAcquireInStackQueuedSpinlock*带一个指针参数指向一个自旋锁数据结构，以及一个自旋锁队列句柄。此自旋锁句柄实际上是一个数据结构，内核将有关该锁的状态信息保存在此数据结构中，状态信息包括该锁的所有权和一个处理器队列（该队列中的每个处理器都可能在等待该锁）。

执行体的互锁操作

内核提供许多简单的、建立在自旋锁基础之上的同步函数，来支持一些更加高级的操作，比如在单向链表和双向链表中插入和删除元素。这样的例子有，*ExInterlockedPopEntryList*和*ExInterlockedPushEntryList*支持单向链表，*ExInterlockedInsertHeadList*和*ExInterlockedRemoveHeadList*支持双向链表。所有这些函数都要求带一个标准的自旋锁作为参数，内核和设备驱动程序中到处使用了这些函数。

www.TopSage.com

低IRQL的同步

在多处理器环境中，内核之外的执行体软件也需要对全局数据结构的访问进行同步操作。例如，内存管理器只有一个页帧数据库，所以在访问该页帧数据库时将它当做一个全局数据结构，而且，设备驱动程序也需要确保它们在访问设备的时候是独占的。执行体通过调用内核函数，可以创建一个自旋锁，获取该锁，以及释放该锁。

然而，自旋锁只是部分地满足了执行体对于同步机制的需求。因为在一般情况下，等待一个自旋锁意味着要使一个处理器停止下来，所以，自旋锁只能被用于以下一些严格受限的场合：

- 对于受保护的资源，必须快速访问，并且不要与其他的代码有复杂的交互关系；
- 临界区代码的内存页不能被换出去，这些代码不能引用那些可被换页的数据，也不能调用外部过程（包括系统服务），也不能产生中断或者异常。

这些约束条件并不是在所有的情况下都能满足的。而且，除了互斥以外，执行体还需要完成其他类型的同步操作，并且它必须为用户模式提供同步机制。

以下列出一些当自旋锁不适合时可以使用的其他同步机制：

- 内核分发器对象（Kernel Dispatcher Objects）；
- 快速互斥体和受限互斥体（Fast Mutexes and Guarded Mutexes）；
- 压栈锁（Push Locks）；
- 执行体资源（Executive Resources）。

表3.9比较了这些同步机制的能力，以及它们与内核模式APC交付过程的交互。

表3.9 内核同步机制

	是否暴露给设备驱动程序使用	禁止常规的内核模式APC	禁止特殊的内核模式APC	支持递归获取操作	支持共享的和独占的获取操作
内核分发器互斥体	是	是	否	是	否
内核分发器信号量	是	否	否	否	否
快速互斥体	是	是	是	否	否
受限互斥体	否	是	是	否	否
压栈锁	否	否	否	否	是
执行体资源	是	是	否	是	是

www.TopSage.com

内核分发器对象

内核以内核对象的形式，向执行体提供了额外的同步机制，这些内核对象合起来统称为**分发器对象**（*dispatcher object*）。那些用户可见的同步对象，正是从这些内核分发器对象中获得它们的同步能力。每个支持同步的用户可见对象都封装了至少一个内核分发器对象。通过*WaitForSingleObject*和*WaitForMultipleObjects*函数，执行体的同步语义对于Windows程序员是可见的，Windows子系统通过调用类似的、由对象管理器提供的系统服务来实现这两个等待函数。Windows应用程序中的线程可以与Windows进程、线程、事件、信号量、互斥体、可等待的定时器、I/O完成端口或者文件对象进行同步操作。

值得提及的另外一种执行体同步对象称为**执行体资源**（*executive resource*）。执行体资源既提供了独占访问的能力（像一个互斥体），也提供了共享读访问的能力（多方共享了对一个数据结构的只读访问能力）。然而，它们仅仅可用于内核模式的代码，因此通过Windows API是无法访问的。执行体资源不是分发器对象，而是直接从非换页的内存池中分配的数据结构，它们有自己特有的服务来初始化、上锁、释放、查询和等待它们。Ntddk.h中定义了执行体资源的数据结构，在DDK的参考文档中介绍了有关的执行体支持例程。

本小节余下的部分介绍了等待分发器对象的实现细节。

等待分发器对象 一个线程可以与一个分发器对象进行同步操作，做法是等待该对象的句柄。这样做使得内核挂起该线程，将它的分发器状态从运行态改变为等待态，如图3.25所示。内核从分发器就绪队列中去除该线程，不再考虑它的继续执行。



> **注** 图3.25是一个进程状态变迁图，其焦点在于就绪、等待和运行状态（即与等待对象有关的状态）。关于其他状态的介绍请参见第6章。

在任何给定的时刻，一个同步对象总是处于两种状态之一：**有信号状态**（*signaled state*），或者**无信号状态**（*nonsignaled state*）。在内核将一个线程的分发器状态从等待状态改变成就绪状态以前，该线程不可能恢复执行。如果该线程正在等待一个分发器对象的句柄，而且，该分发器对象经历了一次状态改变（从无信号状态改变成有信号状态，例如，当一个线程设置了一个事件对象时），那么，这种分发器状态改变就会发生。一个线程为了与一个对象同步，调用对象管理器提供的几个等待系统服务之一，同时传递给它一个对象句柄，该句柄指向它所要同步的对象。该线程可以等待一个或者几个对象，还可以指定：如果在特定长度的时间段以内等待过程还没有结束的话，则取消等待。无论何时当内核将一个对象设置成有信号状态时，内核的*KiWaitTest*函数就会进行检查，看是否有任何线程正在等待该对象（而且并不同时还在等待其他对象变成有信号状态）。如果存在这样的线程，则内核将这些线程中的一个或者多个从它

www.TopSage.com

们的等待状态中释放出来，从而它们可以继续执行。

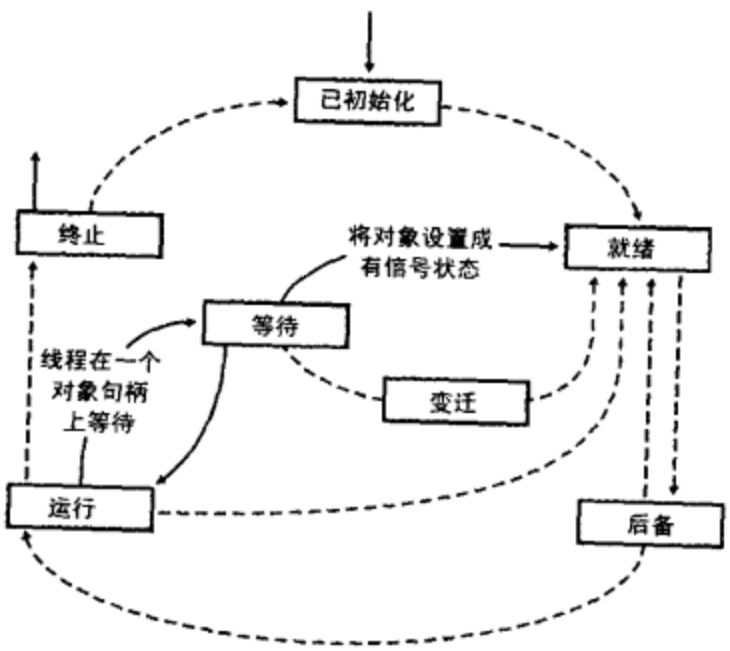


图3.25 等待一个分发器对象

以下设置事件的例子演示了同步与线程分发是如何相互影响的。

- 一个用户模式的线程等待一个事件对象的句柄；
- 内核将该线程的调度状态从就绪状态改变成等待状态，然后将该线程加入到正在等待该事件的线程列表中；
- 另一个线程设置了该事件；
- 内核沿着该事件的等待线程队列向前搜索。如果一个线程的等待条件满足的话，内核将该线程的状态从等待改变成就绪。如果它是一个可变优先级的线程，则内核可能也要提升它的执行优先级；
- 因为一个新线程已经变成就绪执行状态，所以，分发器重新进行调度。如果它找到一个正在运行的线程，其优先级低于新就绪线程的优先级，那么，它会抢占此低优先级的线程，并且发出一个软件中断，以便激发一个环境切换，切换到高优先级的线程中；
- 如果没有处理器可以抢占的话，则分发器将该就绪线程放到分发器就绪队列中，以便稍后再被调度。



注 有些线程可能在等待多个对象，所以它们会继续等待。

www.TopSage.com

如何确定对象的信号状态（*What Signals an Object*） 对于不同的对象，信号状态的定义也有所不同。一个线程对象在它的生命周期中处于无信号状态，当线程终止的时候，它被内核设置为有信号状态。类似地，当一个进程的最后一个线程终止的时候，内核将该进程对象设置为有信号状态。与此不同的是，定时器对象，就像一个闹钟一样，在特定的时候被设置为“响铃”。当它的时间到期时，内核将定时器对象设置为有信号状态。

在选择同步机制时，程序必须考虑到那些“控制各种同步对象的行为”的规则。当一个对象被设置为有信号状态时一个线程的等待是否结束，取决于该线程所等待的对象的类型而有所不同，如表3.10所示。

表3.10 有信号状态的定义

对象类型	何时设置为有信号状态	对于正在等待的线程有什么效果
进程	最后一个线程终止	全部被解除
线程	线程终止	全部被解除
文件	I/O操作完成	全部被解除
调试对象	调试消息被插入该对象	全部被解除
事件（通知类型）	线程设置了该事件	全部被解除
事件（同步类型）	线程设置了该事件	一个线程被解除，事件对象被重置
带键的事件	线程用一个键设置了该事件	正在等待该键并且与设置该事件的线程同属一个进程的线程被解除
信号量	信号量计数降低1	一个线程被解除
定时器 （通知类型）	设置的时间到了，或者时间间隔到期了	全部被解除
定时器 （同步类型）	设置的时间到了，或者时间间隔到期了	一个线程被解除
互斥体	线程释放了互斥体	一个线程被解除
文件	I/O完成	全部线程都被解除
队列	在队列中放入项目	一个线程被解除

当一个对象被设置成有信号状态时，那些正在等待该对象的线程一般会立即从它们的等待状态中解除出来。一些可导致这种状态改变的内核分发器对象和系统事件如图3.26所示。

www.TopSage.com

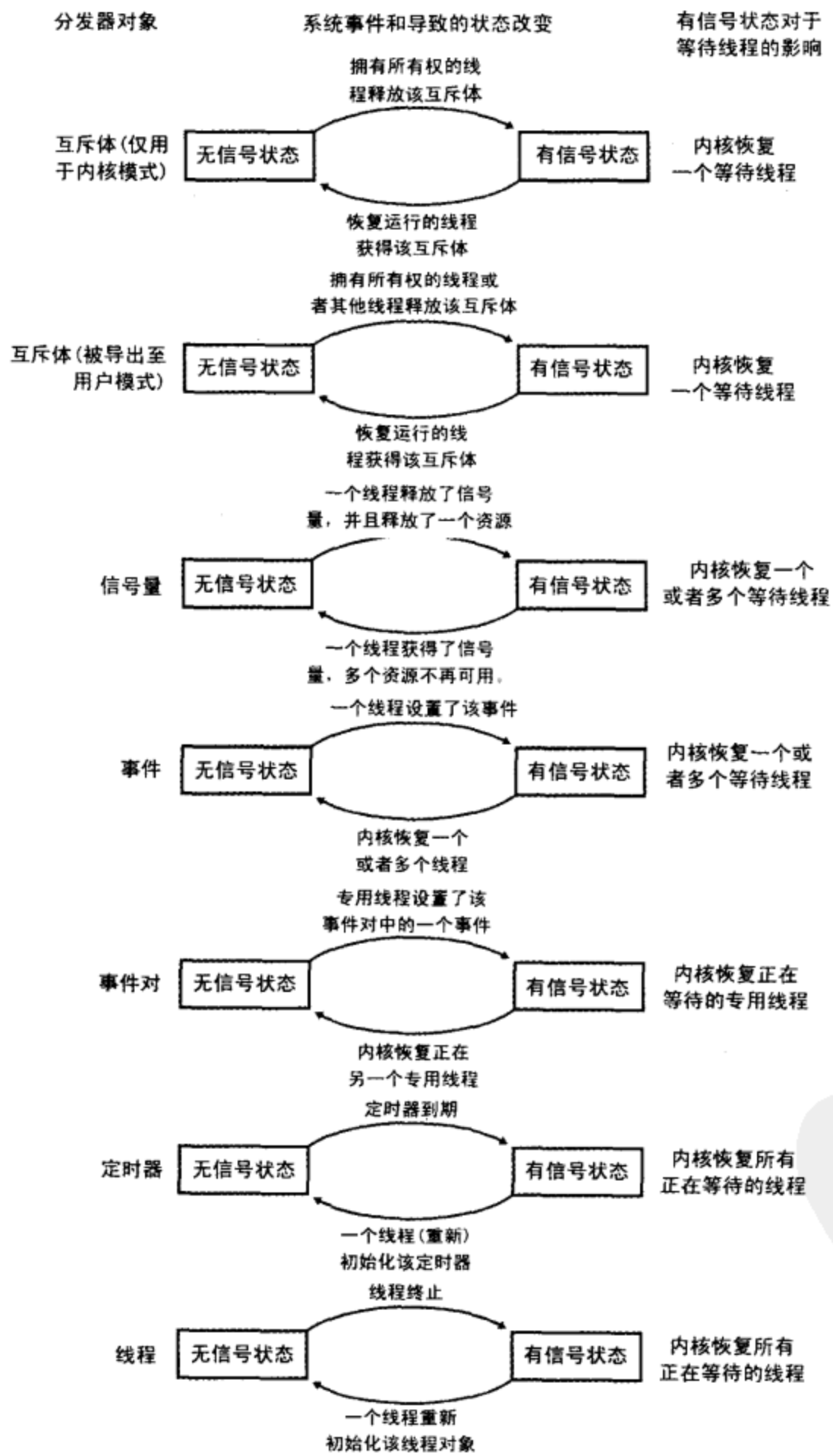


图3.26 选择出来的一部分内核分发器对象

www.TopSage.com

例如，通知类型的事件对象（在Windows API中称为手工重置的事件）被用来宣布某一件事情发生了。当该事件对象被设置为有信号状态时，所有正在等待该事件的线程都被解除。一个例外是，对于任何同时在等待多个对象的线程，这一条并不成立；这样的线程可能要继续等待，直到其他的对象也变成有信号状态。

与事件对象不同的是，互斥体对象有与之关联的所属权。所属权的用途是，获得对一个资源的互斥访问，即同一时刻只有一个线程可以持有该互斥体。当互斥体对象变成空闲的时候，内核将它设置成有信号状态，然后选择一个正在等待的线程来执行。内核选中的线程获得该互斥体对象，所有其他的线程继续等待。

带键的事件和临界区

对于Windows XP来说，一个新的同步对象，称为**带键的事件**（*keyed event*），这值得特别提及，因为在进程使用**临界区**过程中，这一同步对象在帮助进程处理低内存情形时扮演了特别的角色。带键的事件使得一个线程在等待的时候可以指定一个“键”，当同一个进程中的另一个线程用同样的键给该事件发信号时，该线程就会被唤醒。

Windows进程通常会使用临界区函数*EnterCriticalSection* 和 *LeaveCriticalSection*来同步进行该进程的线程对于私有资源的访问。使用这些函数比直接使用互斥体对象有一些好处，因为如果没有竞争的话，它们就不需要进入到内核模式中。如果有竞争的话，则*EnterCriticalSection*动态分配一个事件对象，并且，正在等待获取该临界区的那个线程要等到当前拥有该临界区的线程通过*LeaveCriticalSection*并向它发出信号。

在*EnterCriticalSection*中，如果因为系统内存太低，从而导致为该临界区分配事件对象失败的话，则*EnterCriticalSection*会使用一个名为*CritSecOutOfMemoryEvent*的、全局的、带键的事件。如果*EnterCriticalSection*不得不使用*CritSecOutOfMemoryEvent*，而非标准事件的话，则正在等待该临界区的线程使用该临界区的地址作为其键。在此情况下，即使当内存临时不够的时候，临界区函数仍然可以正确地工作。

这里简短的讨论并不是要列举出使用这些执行体对象的所有理由和应用场合，而只是列出它们的基本功能和同步行为。关于如何在Windows程序中使用这些对象的更多信息，请参见关于同步对象的Windows参考文档，或者Jeffrey Richter的*Programming Applications for Microsoft Windows*。

www.TopSage.com

数据结构 有两个数据结构对于搞清楚谁正在等待什么是至关重要的。这两个数据结构就是分发器头和等待块，它们都被公开定义在DDK头文件Ntddk.h中。为了方便起见，这里再列出它们的定义：

```
typedef struct _DISPATCHER_HEADER {
    UCHAR Type;
    UCHAR Absolute;
    UCHAR Size;
    UCHAR Inserted;
    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;

typedef struct _KWAIT_BLOCK {
    LIST_ENTRY WaitListEntry;
    struct _KTHREAD *RESTRICTED_POINTER Thread;
    PVOID Object;
    struct _KWAIT_BLOCK *RESTRICTED_POINTER NextWaitBlock;
    USHORT WaitKey;
    USHORT WaitType;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *RESTRICTED_POINTER PRKWAIT_BLOCK;
```

分发器头结构包含了对象类型、信号状态，以及正在等待该对象的线程列表。等待块结构代表了一个线程正在等待一个对象。处于等待状态的每个线程都有一个等待块列表，其中每个等待块代表了该线程正在等待的一个对象。每个分发器对象都有一个等待块列表，其中每个等待块代表了正在等待该对象的一个线程。由于分发器对象维护了这一列表，所以，当有线程向一个分发器对象发出信号时，内核可以很快地确定谁正在等待该对象。等待块结构中有一个指针指向正在被等待的对象，另一个指针指向正在等待该对象的线程，还有一个指针指向下一个等待块（如果该线程正在等待多个对象的话）。它也记录了等待的类型（某一种类型，或者所有类型），以及在*WaitForMultipleObjects*调用中，调用者线程传递的句柄数组中当前项所在的位置（如果该线程只等待一个对象，则位置为0）。

图3.27显示了分发器对象与等待块以及线程之间的关系。在这个例子中，线程1正在等待对象B，线程2正在等待对象A和B。如果对象A变成有信号状态的话，内核将会看到，因为线程2也在等待另一个对象，所以线程2不可能马上准备执行。另一方面，如果对象B变成有信号状态，则内核可以立即让线程1准备执行，因为它并没有在等待任何其他对象。

www.TopSage.com

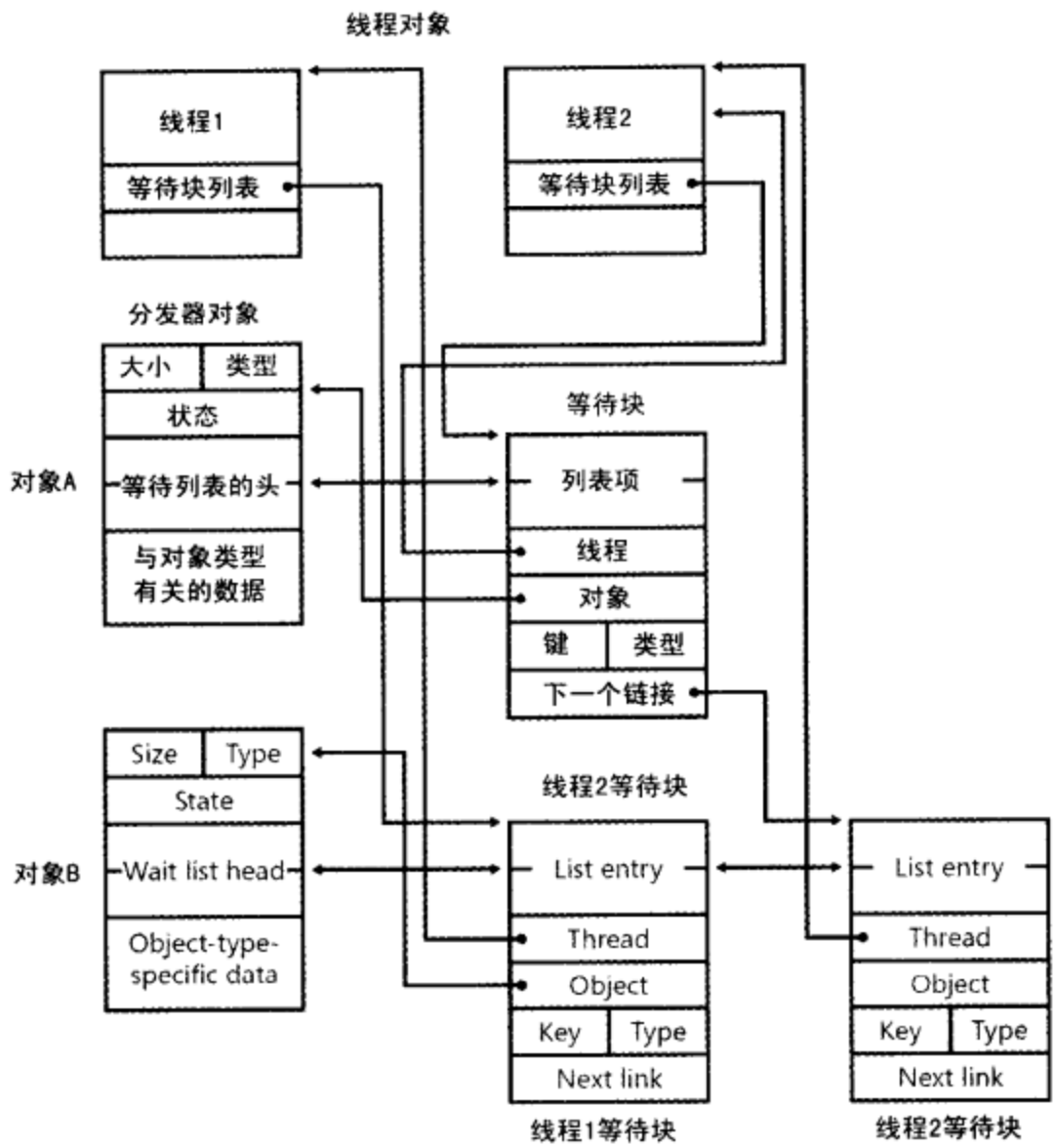


图3.27 等待数据结构



实验：查看等待队列

虽然许多进程查看器工具可以指出一个线程是否处于等待状态（如果是的话，还可以指出它的等待类型），但是，惟有通过内核调试器的*!thread*命令，你才可以看到一个线程正在等待的对象的列表。例如，下面的片断是*!process*命令的部分输出，其中显示了一个线程正在等待一个事件对象：

```
kd> !process
§
        THREAD 8a12a328  Cid 0bb8.0d50  Teb: 7ffdd000 Win32Thread: e7c9aeb0 WAIT
: (WrUserRequest) UserMode Non-Alertable
            8a21bf58  SynchronizationEvent
```


www.TopSage.com

你可以使用*dt*命令来解释该对象的分发器头，如下所示：

```
kd> dt nt!_dispatcher_header 8a21bf58
nt!_DISPATCHER_HEADER
   +0x000 Type             : 0x1 ''
   +0x001 Absolute         : 0 ''
   +0x002 Size             : 0x4 ''
   +0x003 Inserted         : 0 ''
   +0x004 SignalState      : 0
   +0x008 WaitListHead     : _LIST_ENTRY [ 0x8a12a398 - 0x8a12a398 ]
```

从这里，我们可以知道，其他没有线程也在等待该事件对象，因为等待列表头的前向和后向指针指向同一个位置（单个等待块）。将此等待块（位于地址0x8a12a398处）转储出来，得到如下的输出：

```
kd> dt nt!_kwait_block 0x8a12a398
nt!_KWAIT_BLOCK
   +0x000 WaitListEntry    : _LIST_ENTRY [ 0x8a21bf60 - 0x8a21bf60 ]
   +0x008 Thread           : 0x8a12a328
   +0x00c Object           : 0x8a21bf58
   +0x010 NextWaitBlock    : 0x8a12a398
   +0x014 WaitKey          : 0
   +0x016 WaitType         : 1
```

如果该等待列表有多个表项，那么，你可以在每个等待块的*WaitListEntry*域的第二个指针值上执行同样的命令，就可以遍历该列表，并且可看到其他哪些线程也在等待该对象（在等待块的线程指针上执行*!thread*命令）。

快速互斥体（fast mutex）和受限互斥体（guarded mutex）

快速互斥体，也称为执行体互斥体，通常比互斥体对象提供了更好的性能，原因是，尽管它们也是建立在分发器事件对象基础之上的，但如果对于快速互斥体没有竞争的话，它们无须等待该事件对象（因此，也无须等待该事件对象赖以建立起来的自旋锁）。这使得快速互斥体在一个多处理器环境中具有特别好的性能。快速互斥体广泛应用于内核和设备驱动程序中。

然而，快速互斥体仅适用于普通内核模式APC（本章前面已经介绍过）的交付能够被禁止的场合下。执行体定义了两个函数来获得快速互斥体。这两个函数就是*ExAcquireFastMutex*和*ExAcquireFastMutexUnsafe*。前一个函数将处理器的IRQL提升到APC_LEVEL级别上，从而阻止所有的APC被交付；后一个函数期望在被调用的时候普通内核模式APC交付是禁止的，这可以通过提升IRQL至APC级别或者调用*KeEnterCriticalRegion*来做到。快速互斥体的另一个局限性是，它们不能被递归获取，而互斥体对象则可以。

www.TopSage.com

受限互斥体对于Windows Server 2003来说是新的，本质上讲它与快速互斥体是相同的（不过，它在内部使用了不同的同步对象：KGATE）。通过*KeAcquireGuardedMutex*函数可以获得受限互斥体，但是，它并非通过调用*KeEnterCriticalRegion*来禁止APC（那只能禁止普通的内核模式APC），相反，它通过调用*KeEnterGuardedRegion*来禁止所有内核模式APC的交付。它们并没有被暴露到内核以外，相反，它们主要被用于内存管理器；内存管理器利用它们来保护全局的操作，比如创建页面文件、删除特定类型的共享内存区，以及执行换页内存池的扩展（有关内存管理器的更多信息，请参见第7章）。

执行体资源

执行体资源是一种支持共享和独占访问的同步机制；如同快速互斥体一样，在获取执行体资源以前，它们要求普通的内核模式APC交付已被禁止。它们也建立在分发器对象之上，不过，只有当出现竞争的时候它们才会用到分发器对象。执行体资源也被应用于整个系统之中，特别是在文件系统的驱动程序中。

如果一个线程正在等待获得对一个资源的共享访问权，则它应该等待一个与该资源相关联的信号量；如果一个线程正在等待获得对一个资源的独占访问权，则它应该等待一个事件。具有无限计数值的信号量被用于共享等待者，因为当一个独占持有者通过给信号量发信号来释放一个资源时，这些共享等待者全部被唤醒，并且被赋予对该资源的访问权。当一个线程在等待独占访问一个资源，而该资源当前正被其他线程拥有的时候，该线程等待一个同步事件对象，因为当该事件得到信号时，只有一个等待者将被唤醒。

鉴于共享和独占访问所提供的灵活性，有以下的一些函数可被用于获取资源：*ExAcquireResourceSharedLite*、*ExAcquireResourceExclusiveLite*、*ExAcquireSharedStarveExclusive*、*ExAcquireWaitForExclusive*和*ExTryToAcquireResourceExclusiveLite*。DDK中有文档介绍这些函数。



实验：列出已获得的执行体资源

内核调试器*!locks*命令在换页内存池中查找执行体资源对象，并且转储出它们的状态。在默认情况下，该命令只列出当前拥有的执行体资源，但是，利用*-d*选项，可以列出所有的执行体资源。下面是该命令的部分输出：

```
lkd> !locks
**** DUMP OF ALL RESOURCE OBJECTS ****
KD: Scanning for held locks.

Resource @ nt!MmSystemWsLock (0x805439a0)    Exclusively owned
    Contention Count = 123
     Threads: 89b36020-01<*>
KD: Scanning for held locks.....................................................
```


www.TopSage.com

```
.....
.....

Resource @ 0x89da1a68    Shared 1 owning threads
     Threads: 8a4cb533-01<*> *** Actual Thread 8a4cb530
```

注意，从资源结构中提取出来的竞争计数值（contention count）记录了“任何线程试图获得该资源但因为它已被其他线程拥有而不得不等待”的次数。

通过指定-*v*开关以及一个资源的地址，你可以检查某个特定资源对象的细节，包括拥有该资源的线程以及当前正在等待该资源的所有线程。

```
lkd> !locks -v 0x805439a0

Resource @ nt!MmSystemWsLock (0x805439a0)    Exclusively owned
    Contention Count = 123
     Threads: 89b36020-01<*>

     THREAD 89b36020  Cid 0e98.0bd4  Teb: 7ffd9000 Win32Thread: e2bcc538 RUNNING on process
or 0
     Not impersonating
     DeviceMap                 e1df7d18
     Owning Process            8999d020
     Wait Start TickCount      492582         Elapsed Ticks: 15
     Context Switch Count      532                 LargeStack
     UserTime                  00:00:01.0462
     KernelTime                00:00:00.0320
     Start Address 0x77e7d342
     Win32 Start Address 0x0101f1d0
     Stack Init a9d20000 Current a9d1fd44 Base a9d20000 Limit a9d1d000 Call 0
     Priority 11 BasePriority 8 PriorityDecrement 2 DecrementCount 16
Unable to get context for thread running on processor 0, HRESULT 0x80004001
```

压栈锁

在Windows XP中引入的压栈锁是另一种建立在事件对象基础之上的优化同步机制（在Windows Server 2003中，压栈锁建立在内部的KGATE同步对象基础之上），如同快速互斥体一样，只有当针对同一个压栈锁存在竞争的时候，它们才会等待一个事件对象。相比快速互斥体，它们提供的好处是，它们可以按照共享的或者独占的模式来获得。内核没有导出压栈锁，也没有相应的文档介绍它们，所以，它们仅被保留给操作系统使用。

有两种类型的压栈锁：普通的压栈锁和能感知缓存的压栈锁。普通的压栈锁只要求一个指针的大小（在32位系统上是4字节，在64位系统上是8字节）。当一个线程获得了一个普通的压栈锁时，如果它当前尚未被占有，则压栈锁代码将它标记为已被占有。如果该压栈锁被独占方式占有，或者该线程希望以独占方式获得该压栈锁但它却被一组线程以共享方式占有着，则该线

www.TopSage.com

程在自己的栈上分配一个等待块，并且初始化该等待块中的事件对象，然后将该等待块加入到与压栈锁相关联的等待列表中。当一个线程释放一个压栈锁的时候，如果有等待者的话，则唤醒一个等待者，其做法是，向该等待者的等待块中的事件发出信号。

能感知缓存的压栈锁建立在基本压栈锁之上，它为系统中的每个处理器分配一个压栈锁，然后将这些压栈锁与自己关联起来。当一个线程希望以共享模式获得一个能感知缓存的压栈锁时，它只是简单地以共享模式获得对应于当前处理器的那个压栈锁；如果该线程希望以独占模式获得一个能感知缓存的压栈锁，则它以独占模式获得每个处理器的压栈锁。

压栈锁的使用范围包括对象管理器和内存管理器。在对象管理器中，它们可保护全局的对象管理器数据结构和对象安全描述符；而在内存管理器中，它们可保护AWE数据结构。

利用驱动程序检验器（Driver Verifier）来检测死锁

死锁是一个同步问题，它源于两个线程或者处理器分别持有另一个想要的资源，并且谁也不会让出自己所属的资源。这种情况可能会导致系统或者进程停止。在第7章和第9章中将介绍的驱动程序检验器有一个选项可以检查与自旋锁、快速互斥体和互斥体有关的死锁问题。有关何时启动驱动程序检验器来帮助解决系统停止问题的信息，请参见第14章。

3.4 系统辅助线程

在系统初始化的过程中，Windows在System进程中创建了几个线程，这些线程称为**系统辅助线程**，它们的用途只是代表其他的线程来完成一些工作。在许多情况下，在DPC/Dispatch级别上执行的线程需要执行一些只有在更低IRQL级别上才能执行的函数。例如，一个在DPC/Dispatch级别IRQL的任意线程环境中执行的DPC例程，可能需要访问换页内存池，或者等待一个分发器对象以便与一个应用程序线程保持同步。因为DPC例程不能降低IRQL，所以，它必须要将这样的处理过程传递给一个在低于DPC/Dispatch级别的IRQL上执行的线程。

有些设备驱动程序和执行体组件创建了它们自己的线程，由这些线程专门在被动级别上处理一些工作；然而，绝大多数设备驱动程序和执行体组件使用系统辅助线程，从而可以避免在系统中因这些额外的线程而招致的不必要的调度和内存开销。设备驱动程序或者执行体组件通过调用执行体函数*ExQueueWorkItem*或 *IoQueueWorkItem*，可以请求一个系统辅助线程的服务。这两个函数把一个**工作项目**（*work item*）放在一个队列分发器对象上，系统辅助线程在这个对象上寻找工作来做（有关队列分发器对象的更多细节信息，请参见第9章中的“I/O完成端口”一节）。

www.TopSage.com

工作项目包括一个例程指针以及一个参数，当系统辅助线程处理该工作项目时它会把此参数传递给该例程。该例程是由请求被动级别执行模式的设备驱动程序或者执行体组件实现的。

例如，如果一个DPC例程必须等待一个分发器对象，那么它可以初始化一个工作项目，让它指向该驱动程序内部的一个专门等待此分发器对象的例程，可能还包括一个指向该对象的指针。在某个阶段上，系统辅助线程将该工作项目从它的队列中删除，并执行此驱动程序的例程。在驱动程序的例程完成以后，系统辅助线程检查一下，看是否还有其他的工作项目要处理。如果没有其他的工作项目了，则该系统辅助线程被阻塞，直到有新的工作项目被放到其队列中。当系统辅助线程处理一个工作项目时，其DPC例程可能已经完成执行了，也可能尚未完成（在一个单处理器系统上，DPC例程总是在它的工作项目被处理之前完成其执行，因为在DPC/Dispatch级别的IRQL上不会发生线程调度）。

系统辅助线程有以下三种类型：

- **延迟型辅助线程**，它们在优先级12上执行，处理一些并非时间紧急的工作项目，而且当它们在等待工作项目的时候允许它们的栈页面被换出到页面文件中；
- **紧急型辅助线程**，它们在优先级13上执行，处理一些时间紧急的工作项目，在Windows Server系统上，它们的栈始终位于物理内存中；
- **超紧急型辅助线程**，它们在优先级15上执行，其栈总是在内存中；进程管理器（process manager，非process explorer）使用一个超紧急的辅助线程来执行线程“回收”函数，即释放已被终止的线程。

通过执行体的*ExpWorkerInitialization*函数（在系统引导过程的早期被调用）来创建的延迟型和紧急型辅助线程的数目取决于系统中内存的数量，以及该系统是否是服务器系统。表3.11显示了在不同系统配置上创建的初始线程数目。你可以通过注册表HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Executive键下面的AdditionalDelayedWorkerThreads和AdditionalCriticalWorkerThreads，来指定*ExpInitializeWorker*创建至多16个额外的延迟型辅助线程，以及至多16个额外的紧急型辅助线程。

表3.11 系统辅助线程的初始数目

	Windows 2000	Windows 2000 Server	Windows XP和 Windows Server 2003
延迟型	3	3	7
紧急型	5	10	5
超紧急型	1	1	1

执行体试图在系统执行过程中，让紧急型辅助线程的数目符合工作负载的变化。每隔一秒

www.TopSage.com

钟，执行体函数*ExpWorkerThreadBalanceManager*确定是否应该创建一个新的紧急型辅助线程。由*ExpWorkerThreadBalanceManager*创建的紧急型辅助线程称为**动态的辅助线程**，在创建这样的辅助线程之前下面的条件必须全部被满足：

- 在紧急工作队列中存在工作项目；
- 不活动的紧急型辅助线程（指因为等待工作项目而被阻塞的线程，或者在执行一个工作例程时被阻塞在分发器对象上的线程）的数目必须少于系统中处理器的数目；
- 动态辅助线程的数目少于16个。

动态辅助线程在10分钟不活动之后就会退出。因此，当工作负载需要时，执行体可以创建至多16个动态辅助线程。



实验：列出系统辅助线程

你可以使用*!exqueue*内核调试器命令来查看按类型分类的系统辅助线程列表：

```
kd> !exqueue
Dumping ExWorkerQueue: 8046A5C0

**** Critical WorkQueue( current = 0 maximum = 1 )
THREAD 818a2d40  Cid 8.c   Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2ac0  Cid 8.10  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2840  Cid 8.14  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a25c0  Cid 8.18  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2340  Cid 8.1c  Teb: 00000000  Win32Thread: 00000000 WAIT

**** Delayed WorkQueue( current = 0 maximum = 1 )
THREAD 818a20c0  Cid 8.20  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1020  Cid 8.24  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1da0  Cid 8.28  Teb: 00000000  Win32Thread: 00000000 WAIT

**** HyperCritical WorkQueue( current = 0 maximum = 1 )
THREAD 818a1b20  Cid 8.2c  Teb: 00000000  Win32Thread: 00000000 WAIT
```

3.5 Windows全局标志

Windows有一组全局的标志，保存在一个名为*NtGlobalFlag*的系统范围的全局变量中，通过它可以打开操作系统内部的调试、跟踪和验证支持。系统变量*NtGlobalFlag*是在系统引导时候根据注册表HKLM\SYSTEM\CurrentControlSet\Control\Session Manager键中的GlobalFlag值

www.TopSage.com

来初始化的。该注册表值的默认值是0，所以很有可能在你的系统上，你没有使用任何全局标志。而且，每个映像也有一组全局标志，它们也能打开内部的跟踪和验证代码（但是，这些标志的位布局完全不同于系统范围的全局标志）。这些标志并不支持让消费者客户来使用，而且也没有文档说明，但是在检查Windows的内部操作时，它们可能是非常有用的工具。

幸运的是，Platform SDK和调试工具箱包含了一个名为Gflags.exe的工具，它使得你可以查看并改变系统全局标志（既可以是注册表中的标志值，也可以是当前正在运行的系统中的标志值）以及映像全局标志。Gflags既有命令行界面，也有GUI界面。为了看清楚命令行标志，你可以输入**gflags /?**。如果你运行该工具时不加任何开关，则显示出如图3.28所示的对话框。

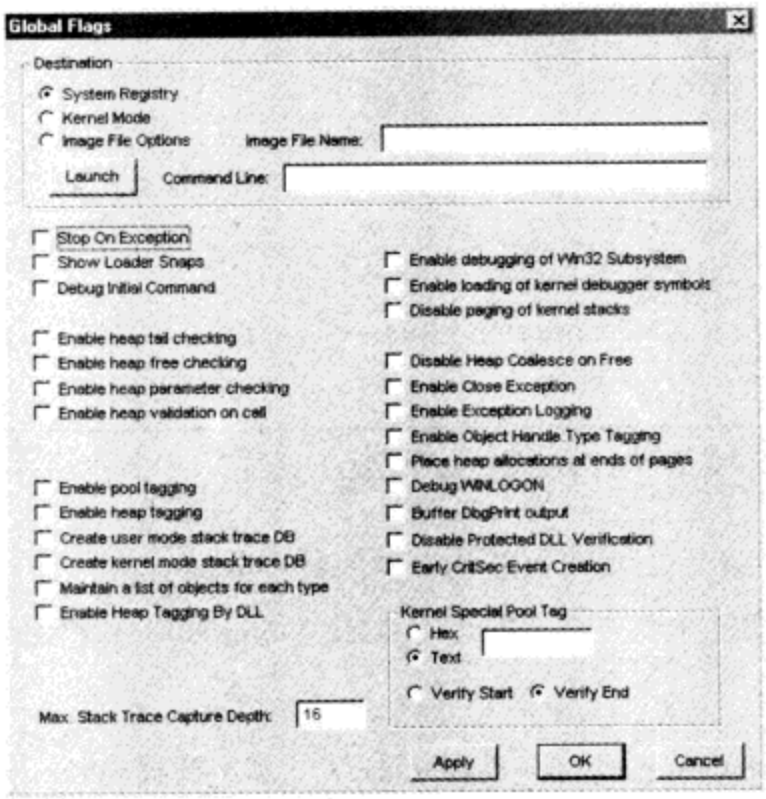


图3.28 利用Gflags工具设置系统调试选项

你可以在“注册表设置（点击System Registry）”和“系统内存中的变量当前值（点击Kernel Mode）”两种模式之间来回切换。要想使你的改变起作用，你必须点击Apply按钮（如果你点击OK按钮，那你就会退出）。 尽管你可以改变一个正在运行的系统中的标志设置，但大多数标志要求重新引导以后才能起作用，而且，关于哪些标志要求重新引导，哪些标志不要求重新引导，并没有任何文档介绍。所以，当你弄不清楚的时候，在改变了一个全局标志以后重新引导一下肯定是正确的。

“Image File Options”选项要求你填写一个有效的可执行映像的名称。该选项被用于改变一组仅适用于单个映像（而并非整个系统）的全局标志。请注意，图3.29中的标志不同于图3.28中显示的操作系统标志。

www.TopSage.com

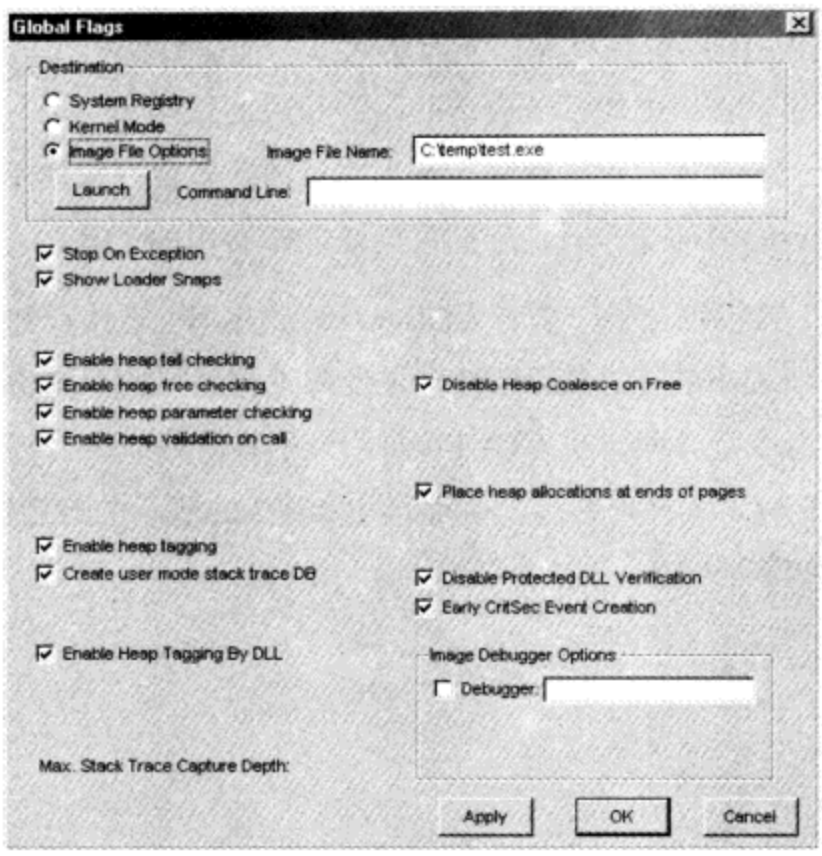


图3.29 利用Gflags工具设置映像全局标志

实验：打开映像加载器追踪功能和查看*NtGlobalFlag*

下面用一个例子来演示通过设置全局标志，你可以获得详细的痕迹追踪信息。在一个引导时支持内核调试的系统（其内核调试器被连接至一个正在运行Kd或者Windbg的主机系统）上运行Gflags。

作为一个例子，试着打开Show Loader Snaps标志。为了做到这一点，选择Kernel Mode，选中Show Loader Snaps复选框，再点击Apply按钮。然后在这台机器上运行一个映像文件，此时，你在内核调试器中将会看到类似以下的输出：

```
LDR: PID: 0xb8 started - 'notepad'
LDR: NEW PROCESS
     Image Path: C:\Windows\system32\notepad.exe (notepad.exe)
     Current Directory: C:\ddk\bin
     Search Path: C:\Windows\System32;C:\Windows\system;C:\Windows
LDR: notepad.exe bound to comdlg32.dll
LDR: ntdll.dll used by comdlg32.dll
LDR: Snapping imports for comdlg32.dll from ntdll.dll
    §
LDR: KERNEL32.dll loaded. - Calling init routine at 77f01000
LDR: RPCRT4.dll loaded. - Calling init routine at 77e1b6d5
LDR: ADVAPI32.dll loaded. - Calling init routine at 77dc1000
LDR: USER32.dll loaded. - Calling init routine at 77e78037
```


www.TopSage.com

你可以使用*!gflags*和*!gflag*内核调试器命令来查看*NtGlobalFlag*内核变量的状态。*!gflags*命令列出所有的标志，并指明哪些标志已被打开，而*!gflag*命令只报告已被打开的标志。

```
kd> !gflags

NT!NtGlobalFlag 0x4400

    STOP_ON_EXCEPTION                SHOW_LDR_SNAPS
    DEBUG_INITIAL_COMMAND            STOP_ON_HUNG_GUI
    HEAP_ENABLE_TAIL_CHECK           HEAP_ENABLE_FREE_CHECK
    HEAP_VALIDATE_PARAMETERS         HEAP_VALIDATE_ALL
   *POOL_ENABLE_TAGGING              HEAP_ENABLE_TAGGING
    USER_STACK_TRACE_DB              KERNEL_STACK_TRACE_DB
   *MAINTAIN_OBJECT_TYPELIST         HEAP_ENABLE_TAG_BY_DLL
    ENABLE_CSRDEBUG                  ENABLE_KDEBUG_SYMBOL_LOAD
    DISABLE_PAGE_KERNEL_STACKS       HEAP_DISABLE_COALESCING
    ENABLE_CLOSE_EXCEPTIONS          ENABLE_EXCEPTION_LOGGING
    ENABLE_HANDLE_TYPE_TAGGING       HEAP_PAGE_ALLOCS
    DEBUG_INITIAL_COMMAND_EX         DISABLE_DBGPRINT

kd> !gflag
NtGlobalFlag at 8046a164 Current NtGlobalFlag contents: 0x00004400
    ptg - Enable pool tagging
    otl - Maintain a list of objects for each type
```

3.6 本地过程调用（LPC）

本地过程调用是一种用于高速传递消息的跨进程通信设施。它并不是通过Windows API直接可以使用的；相反，它是一种内部机制，只能用于Windows操作系统组件。下面是一些使用LPC的例子场合：

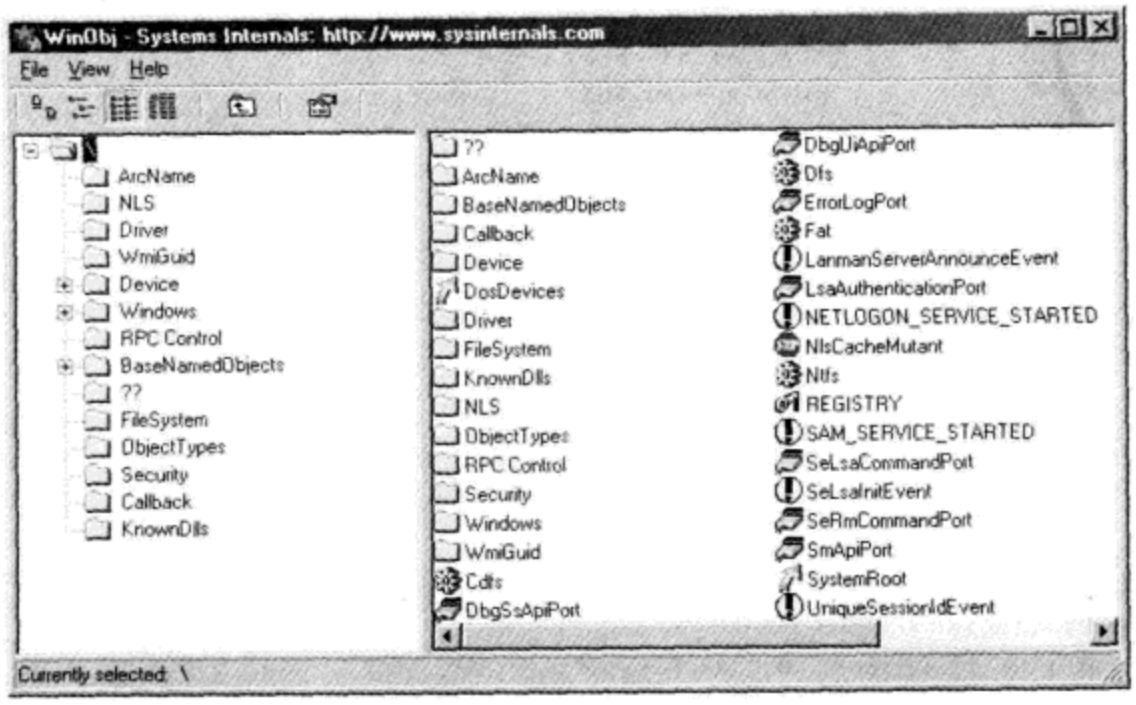
- 使用了远过程调用（RPC）的Windows应用程序，如果它们指定了本地RPC（RPC的一种形式，用于在同一个系统上的进程之间进行通信），则它们会间接地使用LPC；
- 一些Windows API要向Windows子系统进程发送消息；
- Winlogon通过LPC与本地安全认证服务器进程LSASS进行通信；
- 安全引用监视器（一个执行体组件，在第8章中介绍）使用LPC与LSASS进程进行通信。

www.TopSage.com

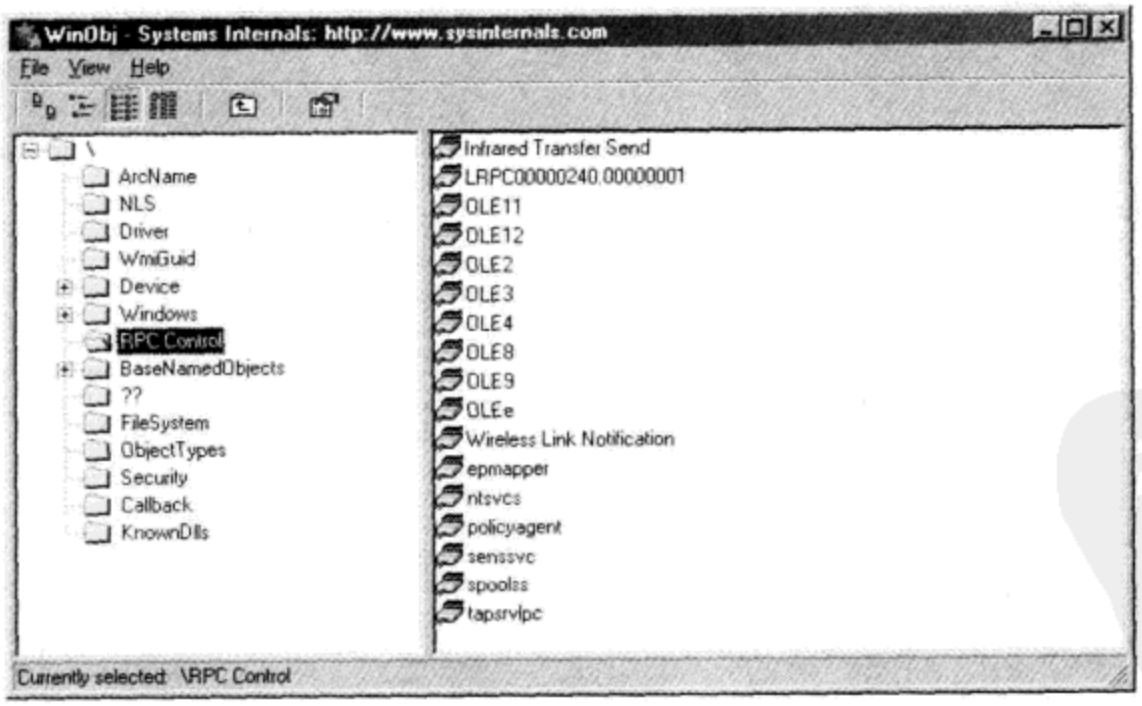


实验：查看LPC端口对象

利用来自*www.sysinternals.com*的Winobj工具，你可以看到命名的LPC端口对象。运行Winobj.exe，选择根目录。插头图标代表了这些端口对象，如下图所示：



要想找到RPC所使用的LPC端口对象，请选择\RPC Control目录，如下图所示：



你也可以利用*!lpc*内核调试器命令来查看LPC端口对象。该命令可以接受一些参数，通过这些参数来指示该命令显示LPC端口、LPC消息，或者正在等待或发送LPC消息的线程。为了显示LSASS认证端口（Winlogon向该端口发送登录请求），首先要获得系统上的端口列表：

www.TopSage.com

```
kd> !lpc
Usage:
    !lpc                        - Display this help
    !lpc message [MessageId]    - Display the message with a
                                  given ID and all related
                                  information
                                  If MessageId is not
                                  specified, dump all messages
    !lpc port [PortAddress]     - Display the port information
    !lpc scan PortAddress       - Search this port and any
                                  connected port
    !lpc thread [ThreadAddr]    - Search the thread in rundown
                                  port queues and display the
                                  port info
                                  If ThreadAddr is missing,
                                  display all threads marked
                                  as doing some lpc operations

kd> !lpc port
Scanning 206 objects
       1  Port: 0xe1360320 Connection: 0xe1360320
          Communication: 0x00000000 'SeRmCommandPort'
       1  Port: 0xe136bc20 Connection: 0xe136bc20
          Communication: 0x00000000 'SmApiPort'
       1  Port: 0xe133ba80 Connection: 0xe133ba80
          Communication: 0x00000000 'DbgSsApiPort'
       1  Port: 0xe13606e0 Connection: 0xe13606e0
          Communication: 0x00000000 'DbgUiApiPort'
       §
       1  Port: 0xe205f040 Connection: 0xe205f040
          Communication: 0x00000000 'LsaAuthenticationPort'
       §
```

在上述输出中找到名为LsaAuthenticationPort的端口，然后将它的地址传递给*!lpc*命令就可以查看该端口的信息，如下面的代码片断所示。

```
kd> !lpc port 0xe205f040
Server connection port e205f040 Name: LsaAuthenticationPort
    Handles: 1   References: 37
    Server process  : ff7d56c0 (lsass.exe)
    Queue semaphore : ff7bfcc8
    Semaphore state 0 (0x0)
    The message queue is empty
    The LpcDataInfoChainHead queue is empty
```


www.TopSage.com

LPC通常被用于在一个服务器进程和该服务器的一个或者多个客户进程之间进行通信。既可以在两个用户模式的进程之间建立起一个LPC连接，也可以在一个内核模式的组件和一个用户模式的进程之间建立起LPC连接。例如，正如在第2章中所说明的，Windows进程有时候通过LPC向Windows子系统发送消息。而且，有些系统进程也使用LPC进行通信，比如Winlogon和Lsass。关于内核模式组件使用LPC与用户进程进行通信的一个例子是，在安全引用监视器和LSASS进程之间通过LPC进行通信。

设计LPC的目的是为了实现以下三种交换消息的方法。

- 短于256字节的消息可以通过调用LPC，以及一个包含该消息的缓冲区来发送。然后，该消息从发送进程的地址空间中被拷贝到系统地址空间中，再从系统地址空间中被拷贝到接收进程的地址空间中。
- 如果一个客户和一个服务器想要交换多于256字节的数据，那么它们可以选择使用一个双方都映射了的共享内存区。发送方把消息数据放在共享内存区中，然后给接收方发送一个消息，其中包含了这些数据在共享内存区中的指针位置，以便于接收方可以找到这些数据；
- 当一个服务器想要读取或者写入大量数据，其数据量超过一个共享内存区的容量时，服务器可以直接在客户的地址空间中读取或者写入这些数据。LPC组件提供了两个函数，供服务器做到这一点。第一个函数发送的消息被用于同步消息传递过程。

LPC导出一个称为**端口对象**的执行体对象，来维护必要的通信状态。尽管LPC只使用了一种对象类型，但是它有以下几种端口。

- **服务器连接端口**，这是一个命名的端口，也是服务器连接请求点。客户通过连接到该端口上，就可以连接至服务器。
- **服务器通信端口**，这是一个未命名的端口，服务器利用该端口与一个特定的客户进行通信。针对每个活动的客户，服务器都有一个这样的端口。
- **客户通信端口**，这是一个未命名的端口，客户线程利用该端口与一个特定的服务器进行通信。
- **未命名的通信端口**，这是一个未命名的端口，创建这个端口的目的是为了在同一进程的两个线程之间进行通信。

LPC的典型用法如下：服务器创建一个命名的服务器连接端口对象。客户发起一个连接请求，要求连接至该端口。如果该请求被准许的话，则两个新的未命名端口，即一个客户通信端口和一个服务器通信端口，被创建起来。客户得到的是指向客户通信端口的句柄，服务器得到的是指向服务器通信端口的句柄。然后，客户和服务器利用这些新的端口进行两者之间的通信。

图3.30显示了一个客户和一个服务器之间的完整连接过程。

www.TopSage.com

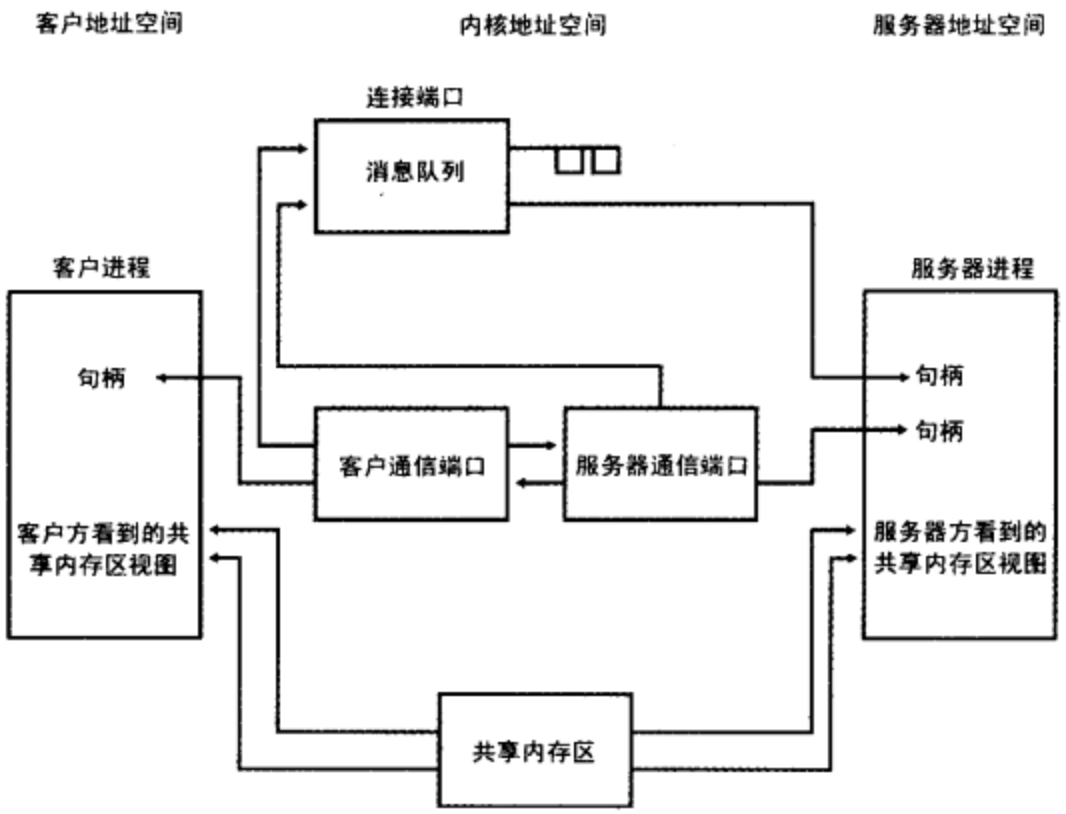


图3.30 LPC端口的用法

3.7 内核事件追踪

Windows内核和几个核心设备驱动程序的各个组件内置了一些功能来记录下其操作的痕迹数据，以便用于系统诊断。它们依赖于内核中一个公共的基础设施，由它向用户模式的ETW（Event Tracing for Windows，Windows事件追踪）设施提供痕迹数据。如果一个应用程序用到了ETW，那么，它必然是以下三类中的某一类或同属于多类。

- **控制器** 控制器启动或者停止记录会话，也管理缓冲区池。
- **提供者** 提供者为它所能产生的事件类定义GUID（全局惟一标识符），并且将它们注册到ETW中。提供者接受来自控制器的命令，以便启动或者停止它所负责的事件类的痕迹追踪。
- **消费者** 消费者针对它想要读取的痕迹数据，选择一个或者多个追踪会话。它们可以实时地接收缓冲区中的事件，也可以接收日志文件中的事件。

Windows服务器系统包含了几个内置的、用户模式下的提供者，其中有针对活动目录、Kerberos和Netlogon的提供者。ETW定义了一个名为“NT Kernel Logger”的记录会话（也称为内核记录器），专门用于内核和核心驱动程序。NT Kernel Logger的提供者是由WMI（Windows Management Instrumentation，Windows管理规范）设备驱动程序（名为Wmixwdm的驱动程序）

www.TopSage.com

实现的，该设备驱动程序是Ntoskrnl.exe的一部分（有关WMI的更多信息，请参见第5章中的WMI一节）。它除了作为内核记录器的核心以外，也管理用户模式ETW事件类型的注册工作。

WMI驱动程序导出了I/O控制接口，以供“用户模式下的ETW例程”和那些“为内核记录器提供痕迹数据的设备驱动程序”使用（关于I/O控制命令的更多信息，请参见第9章）。它还实现了一些函数供Ntoskrnl.exe中的组件在内核模式下用来产生痕迹数据。

当一个用户模式下的控制器启动内核记录器时，在\Windows\System32\Ntdll.dll中实现的ETW库向WMI驱动程序发送一个I/O控制请求，告诉它该控制器想要开始追踪哪些事件类。如果当前的配置是文件记录（相对于内存记录，即输出到一个缓冲区中），则WMI驱动程序在创建日志文件的系统进程中创建一个系统线程。当WMI驱动程序接收到来自于已启动的痕迹数据源的事件时，它将这些事件记录到一个缓冲区中。如果文件记录线程已被启动的话，则它每隔一秒钟被唤醒一次，以便将缓冲区中的内容转储到日志文件中。

针对内核记录器的痕迹纪录有一个标准的ETW痕迹事件头，其中记录了时间戳、进程、线程ID，以及关于该纪录所对应的那一类事件的信息。事件类可能提供了与它们的事件有关的额外数据。例如，磁盘事件类痕迹纪录指明了操作类型（读或者写）、该操作所在的磁盘号，以及该操作的扇区偏移和长度。

可被用于内核记录器的痕迹类，以及产生每一事件类的组件包括：

- **磁盘I/O** 磁盘类驱动程序；
- **文件I/O** 文件系统驱动程序；
- **硬件配置** 即插即用管理器（有关即插即用管理器的信息，请参见第9章）；
- **映像加载/卸载** 内核中的系统映像加载器；
- **页面错误** 内存管理器（有关页面错误的更多信息，请参见第7章）；
- **进程创建/删除** 进程管理器（Process manager，有关进程管理器的更多信息，请参见第6章）；
- **线程创建/删除** 进程管理器（Process manager）；
- **注册表活动** 配置管理器（有关配置管理器的更多信息，请参见第4章中“注册表”一节）；
- **TCP/UDP活动** TCP/IP驱动程序。

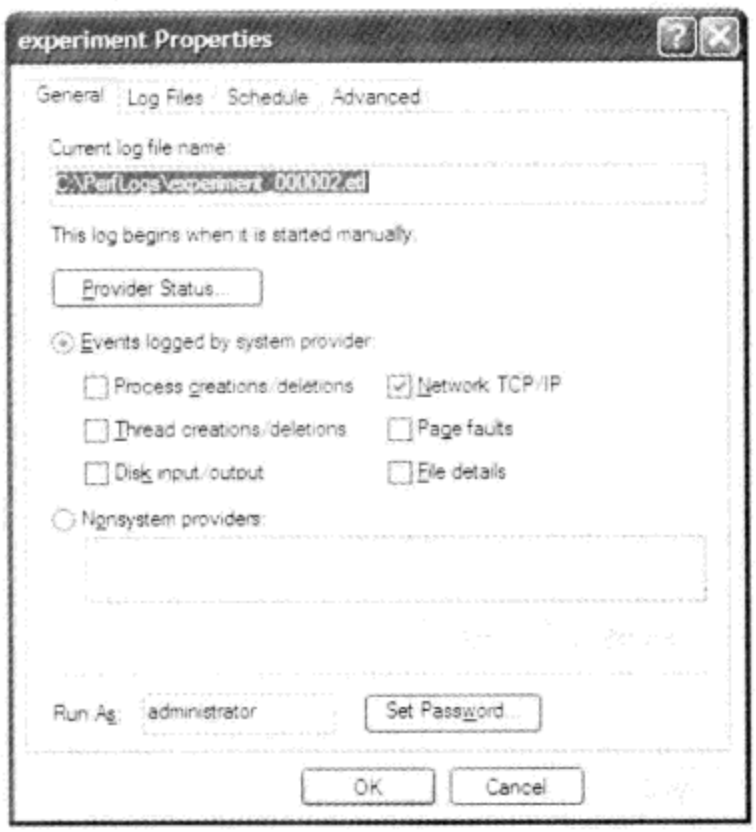
在Platform SKD中你可以找到有关ETW和内核记录器的更多信息，其中包括一些控制器和消费者的例子代码。

www.TopSage.com

实验：用内核记录器追踪TCP/IP的活动

为了启动内核记录器，并且让它产生一个针对TCP/IP活动的日志文件，请遵从以下步骤。

1. 运行性能工具，选择“Performance Logs And Alerts”节点。
2. 选择“Trace Logs”，然后从Action菜单中选择“New Log Settings”。
3. 当出现提示时，为当前的设置输入一个名字（比如**experiment**）。
4. 在打开的对话框中，选中“Events Logged By System Provider”选项，然后去除除了“Network TCP/IP”之外其他所有的选项。
5. 在“Run As”编辑框中（见下图），输入“Administrator”账户名，并且设置与之匹配的口令。



6. 离开该对话框，产生一些网络活动，比如打开一个浏览器，并访问一个Web站点。
7. 在痕迹记录节点中，选择你所创建的痕迹日志，并且从Action菜单中选择Stop。
8. 打开一个命令提示窗口，将当前目录改变到C:\Perflogs下（或者你所指定的存储痕迹日志文件的那个目录）。
9. 如果你正在运行Windows XP或者Windows Server 20003，则运行Tracerpt（位于\Windows\System32目录下），并且将痕迹日志文件的名称传递给它。如果你运行的是Windows 2000，那么，从Windows 2000的资源工具中下载并运行Tracedmp。这两个工具都会生成两个文件：dumpfile.csv和summary.txt。

www.TopSage.com

10. 在Microsoft Excel或者一个文本编辑器中打开dumpfile.csv。你应该可以看到如下所示的TCP和/或UDP痕迹纪录：

TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Recv	0xFFFFFFFF	1.27E+17	0	0	4	88	192.168.001.101	192.168.001.108	4608	0	0	0
TcpIp	Send	0xFFFFFFFF	1.27E+17	0	0	4	76	192.168.001.101	192.168.001.108	4608	0	0	0

3.8 Wow64

Wow64 （64位Windows上的Win32仿真）是指允许在64位Windows上执行32位x86应用程序的软件。它是以一组用户模式DLL的下列形式来实现的。

- Wow64.dll：管理进程和线程的创建、钩住异常分发和Ntoskrnl.exe导出的基本系统调用。它也实现了文件系统重定向，以及注册表重定向和反射。
- Wow64Cpu.dll：为每个正在Wow64内部运行的线程，管理它们的32位CPU环境；为CPU从32位到64位或者从64位到32位的模式切换，提供了与体系结构相关的支持。
- Wow64Win.dll：截取了Win32k.sys导出的GUI系统调用。

这些DLL之间的关系如图3.31所示。

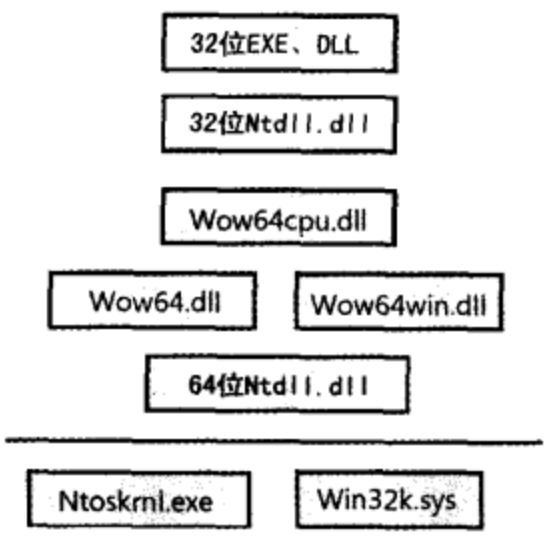


图3.31 Wow64体系结构

www.TopSage.com

Wow64进程地址空间布局结构

Wow64进程可以在2GB虚拟空间中运行，也可以在4GB虚拟空间中运行。如果映像文件的头部设置了大地址空间感知标志，则内存管理器将会保留4GB边界至用户模式边界末尾的用户模式地址空间。如果映像文件没有被标记为大地址空间感知的，则内存管理器将保留2GB之上的用户模式地址空间（有关大地址空间支持的更多信息，请参见第7章中“x86用户地址空间的布局结构”一节）。

系统调用

Wow64钩住了所有从32位代码转变至原生64位系统的代码路径，也钩住了64位原生系统需要调用至32位用户模式代码的所有代码路径。在进程创建的过程中，进程管理器（process manager）将原生的64位Ntdll.dll映射到进程地址空间中。当加载器的初始化过程被调用时，它检查映像文件的头，如果这是一个32位x86应用程序，则加载Wow64.dll。然后Wow64映射32位Ntdll.dll（存储在\Windows\ Syswow64目录下）。最后，Wow64建立起Ntdll内部的启动环境，将CPU模式切换到32位下，并开始执行32位加载器。从这个点开始，整个执行过程就如同该进程运行在原生的32位系统上一样。

Ntdll.dll、User32.dll和Gdi32.dll的特殊32位版本位于\Windows\ Syswow64文件夹下。它们调用到Wow64中，而不是发出原生的32位系统调用指令。Wow64转变到原生的64位模式下，捕获到与系统调用有关的参数（将32位指针转换为64位指针），并且发出对应的原生64位系统调用。当原生的系统调用返回时，Wow64将任何输出参数，如果有必要的话，在返回至32位模式之前从64位转换成32位格式。

异常分发

Wow64通过ntdll的KiUserExceptionDispatcher钩住了异常分发过程。无论何时当64位内核将要给一个Wow64进程分发一个异常时，Wow64会捕获住原生的异常以及用户模式下的运行环境纪录，然后准备一个32位异常和运行环境纪录，并且按照原生32位内核所做的那样将它分发出去。

用户回调

Wow64截取了所有从内核到用户模式的回调。Wow64将这样的调用也按照系统调用来对待；然而，数据转换则是按相反的顺序来完成的：输入参数从64位转换为32位，而输出参数则是在该回调函数从32位返回至64位时被转换的。

www.TopSage.com

文件系统重定向

为了维护应用程序的兼容性，以及降低从Win32到64位Windows的应用程序移植代价，系统目录名称仍然保持不变。因此，\Windows\System32包含了原生的64位映像文件。因为Wow64钩住了所有的系统调用，所以，它会解释所有与路径相关的API，将\Windows\System32文件夹的路径名替换为\Windows\ Syswow64。Wow64也将\Windows\System32\Ime重定向到\Windows\System32\IME（x86），以帮助32位应用程序在安装了远东语言的64位系统上也能保持兼容。而且，32位程序被安装在\Program Files （x86）中，而64位程序进入到普通的\Program Files文件夹中。

\Windows\System32中有一些子目录，出于兼容性的原因，这些子目录没有被重定向，所以32位应用程序在访问这些目录时实际上是在访问真正的目录。这些目录包括：

- %windir%\system32\drivers\etc
- %windir%\system32\spool
- %windir%\system32\catroot2
- %windir%\system32\logfiles

最后，Wow64提供了一种机制来禁止Wow64中内置的文件系统重定向功能，并且这种机制是以每个线程为基础的。线程使用*Wow64EnableWow64FsRedirection*函数即可禁止文件系统重定向功能，在Windows Server 2003及其以后的版本中可以使用此函数。

注册表的重定向和反射

应用程序和组件程序将它们的配置数据保存在注册表中。组件程序在安装过程中，当它们被注册的时候，通常将配置数据写到注册表中。如果同样的组件既安装和注册了一个32位二进制文件，又安装和注册了一个64位二进制文件，那么，最后被注册的那个组件将会覆盖掉以前组件的注册，因为它们写到注册表中同样的位置上。

为了以透明的方式解决这个问题，并且无须对32位组件进行任何代码修改，注册表被分成两个部分：原生的和Wow64的。在默认情况下，32位组件访问32位视图，64位组件访问64位视图。这为32位和64位组件提供了一个安全的执行环境，并且将32位应用程序的状态与64位应用程序（如果存在的话）的状态隔离开来。

为了实现这一点，Wow64截取了所有要打开注册表键的系统调用，并且重新解释这些注册表键的路径，将它们指向注册表的Wow64视图。Wow64在以下这些点上分裂注册表：

- HKLM\Software
- HKEY_CLASSES_ROOT

www.TopSage.com

- HKEY_CURRENT_USER\Software\Classes

在以上每一个键的下面，Wow64创建了一个称为Wow6432Node的键。在该键下面保存的是32位配置信息。注册表的所有其他部分对于32位应用程序和64位应用程序都是共享的（比如HKLM\System）。

如果应用程序需要显式地指定一个注册表键位于某个特定的视图中，那么，*RegOpenKeyEx*和*RegCreateKeyEx*中的下述标志可以做到这一点：

- KEY_WOW64_64KEY—— 从一个32位或者64位应用程序中显式地打开一个64位键；
- KEY_WOW64_32KEY ——从一个32位或者64位应用程序中显式地打开一个32位键。

为了允许32位和64位COM组件之间进行互操作，当注册表中某些特定的部分被更新时，Wow64也会将这些更新映射到另一个视图中。它的做法是，截取到所有针对反射键的更新操作，并且将这些变化智能地映射到注册表的另一个视图中。反射键的列表如下：

- HKEY_LOCAL_MACHINE\Software\Classes
- HKEY_LOCAL_MACHINE\Software\Ole
- HKEY_LOCAL_MACHINE\Software\Rpc
- HKEY_LOCAL_MACHINE\Software\COM3
- HKEY_LOCAL_MACHINE\Software\EventSystem

HKLM\Software\Classes\CLSID的反射是智能的。只有LocalServer32的CLSID才被反射，因为它们运行在客户进程以外，所以，它们可以被32位或64位应用程序通过COM方式来激活。然而，InProcServer32的CLSID没有被反射，因为32位COM DLL不可能被加载到一个64位进程中，与此类似，64位COM DLL不可能被加载到一个32位进程中。

在反射一个注册表键/值的时候，注册表反射器将该键做上标记，所以，它可以理解该键是由反射器创建的。这有助于处理删除的情形，即删除一个被反射的键时，反射器能够判断出它是否需要删除一个被反射的键。如果该键是由反射器写入的话，则必须要删除它。

I/O控制请求

除了普通的读和写操作以外，应用程序可以利用Windows的*DeviceIoControlFile* API，与某些设备驱动程序通过设备I/O控制函数进行通信。应用程序可能会在调用时指定一个输入和/或输出缓冲区。如果该缓冲区中包含了与指针相关的数据，并且发送该控制请求的进程是一个Wow64进程，那么，输入和/或输出结构的视图在32位应用程序和64位驱动程序之间是不相同的，因为对于32位应用程序来说，指针是4字节，而对于64位应用程序来说，指针是8字节。在这种情况下，内核驱动程序应该能够转换这些与指针相关联的结构。驱动程序可以调用

www.TopSage.com

*IoIs32bitProcess*函数来检测一个I/O请求是否是从一个Wow64进程发出的。

16位安装器应用程序

Wow64不支持运行16位应用程序。然而，由于许多应用安装器是16位程序，所以，Wow64包含一些特殊的代码，使得对于某些特别知名的16位安装器的引用能够工作。这样的安装器包括：

- Microsoft ACME Setup 版本: 2.6、3.0、3.01和3.1；
- InstallShield 版本5.x （这里x是任何一个小版本号）。

无论何时当通过CreateProcess() API创建一个16位进程时，首先Ntvdm64.dll被加载进来，然后控制权被传递给它，以检查该16位可执行文件是否是所支持的安装器中的某一个。如果是的话，则发出另一个CreateProcess调用，以便使用同样的命令行参数来激发一个32位版本的安装器。

打印

32位打印机驱动程序不能被用在64位Windows中。打印驱动程序必须要被移植为原生的64位版本。然而，由于打印机驱动程序运行在所请求进程的用户模式地址空间中，并且在64位Windows上只支持原生的64位打印机驱动程序，所以，需要一种特殊的机制来支持32位进程中的打印任务。这是这样做到的：将所有的打印函数重定向到Splwow64.exe中，这里Splwow64.exe是Wow64 RPC打印服务器。由于Splwow64是一个64位进程，所以它可以加载64位打印机驱动程序。

一些限制

Wow64不支持16位应用程序的执行（而在32位版本的Windows上它们是可以运行的），也不支持加载32位内核模式的设备驱动程序（它们必须被移植为原生的64位驱动程序）。Wow64进程只能加载32位DLL，不能加载原生的64位DLL。类似地，原生的64位进程不能加载32位DLL。

除了上述限制以外，由于页面大小的差异，在IA-64系统上的Wow并不支持*ReadFileScatte*、*WriteFileGather*、*GetWriteWatch*或者AWE（Address Window Extension，地址窗口扩展）函数。而且，通过DirectX得到的硬件加速也是不可用的（为Wow64进程提供的是软件仿真）。

3.9 本章总结

在本章中，我们看到了Windows执行体赖以建立起来的关键的基本系统机制。在下一章中，我们将会看到三个重要的、与Windows的管理基础设施有关的机制：注册表、服务，以及WMI（Windows Management Instrumentation，Windows管理规范）。

www.TopSage.com

Management Mechanisms

第 4 章 管理机制

本章讲述了Microsoft Windows中三种基本的机制，它们对于系统的管理和配置至关重要，它们是：

- 注册表
- 服务
- Windows管理设施

4.1 注册表

注册表在Windows系统的配置和控制方面扮演了一个非常关键的角色。它既是系统全局设置的存储仓库，也是每个用户的设置信息的存储仓库。虽然大多数人都把注册表想象成存储在磁盘上的静态数据，但是，在这一节中你将会看到，注册表也是Windows执行体和内核所维护的各种内存中数据结构的一个窗口。本节并不是Windows注册表中各种内容的一份完整参考手册。关于这些深入的信息，你可以在Windows 2000资源工具箱的“Technical Reference to the Windows 2000 Registry”帮助文件（Regentry.chm）中找到相关的文档；至于Windows XP和Windows Server 2003，你可以在*http://www.microsoft.com/windowsserver2003/techinfo/reskit/deploykit.mspx*上的Windows Server 2003开发工具箱中找到这些信息。

我们首先向你介绍注册表的总体结构，并讨论一下它所支持的数据类型，以及简要地浏览一下Windows在注册表中维护的关键信息。然后，我们进一步讨论**配置管理器**（*configuration manager*）的内部细节。这里的配置管理器是负责实现注册表数据库的执行体组件。我们将会涉及的话题有：注册表在磁盘上的内部结构、当一个应用程序请求配置信息时Windows如何获得相应的信息，以及Windows采用了哪些手段来保护这一关键的系统数据库。

查看和修改注册表

一般来说，你应该永远不需要直接编辑注册表：如果存储在注册表中的应用程序设置和系统设置需要手工修改，那么，它们应该有一个对应的用户界面来控制其修改。然而，正如你在

www.TopSage.com

本书中已经多次看到过的那样，有些高级设置和调试设置并没有相应的编辑用户界面。因此，Windows中包含了一些工具，它们可以让你查看和修改注册表。

Windows 2000随带了两个可编辑注册表的工具：Regedit.exe和Regedt32.exe。而Windows XP和Windows Server 2003只带了Regedit.exe。原因是，Windows 2000版本的Regedit（有灵活的搜索、导入和导出功能）是从Windows 98移植过来的，因此，它不支持编辑和查看注册表的安全属性，以及在Windows 98中未定义的注册表数据类型。Windows 2000之所以包含了Regedt32，是因为，虽然它没有强大的搜索功能，也不支持导入和导出，但它是专门为了在Windows 2000上运行而编写的，所以它支持安全属性和与Windows 2000有关的特定数据类型。Windows XP和Windows Server 2003随带的Regedit包含了安全属性编辑功能，也理解所有的注册表数据类型，因此已不再需要Regedt32了。

其他还有一些命令行的注册表工具。例如，Windows XP、Windows Server 2003，以及Windows 2000支持工具箱中包含的Reg.exe，它们能够导入、导出、备份和恢复注册表键，还可以比较、修改和删除注册表键和值。

注册表用法

配置数据主要在三个时间点上被读取：

- 在引导过程中，系统读取有关的设置信息，这些设置信息指定了应该加载哪些设备驱动程序，以及各个子系统（比如内存管理器和进程管理器[process manager]）该如何配置，它们如何调整系统的行为；
- 在登录过程中，Explorer和其他的Windows组件从注册表中读取到每个用户的参数选择，包括网络驱动器字母映射、桌面墙纸、屏幕保护程序、菜单行为和图标的摆放；
- 在应用程序启动过程中，它们读取系统全局范围的设置，比如所有可选安装的组件列表和许可数据，还读取一些针对每个用户的设置信息，可能包括菜单和工具栏的摆放，以及最近访问过的文档列表。

然而，在其他时间点上也可以读取注册表，比如在响应注册表值或键的修改请求时。有些应用程序会监视它们存放在注册表中的设置信息，当它们看到这些信息有变化时，会读取更新后的设置信息。然而，一般来说，在一个空闲的系统上，注册表不应该有任何活动。

在下面的情形下，注册表通常会被修改：

- 虽然算不上是修改，但是，注册表的初始结构和许多默认设置是由一个原型版本的注册表来定义的，此原型版本随Windows的安装介质一起发行，在安装的时候它被拷贝到一个新的系统中；
- 应用安装工具创建了默认的应用设置，以及一些可反映出安装配置选项的设置信息；

www.TopSage.com

- 在设备驱动程序的安装过程中，即插即用系统在注册表中创建了一些设置，告诉I/O管理器如何启动此驱动程序；即插即用系统还创建了其他一些用于配置该驱动程序操作的设置（有关如何安装设备驱动程序的更多信息，请参见第9章）；
- 当你通过用户界面改变了应用程序或者系统设置时，这些改变通常被保存在注册表中。



注 令人沮丧的是，有些应用程序会查询注册表来找到其中的变化，而实际上它们应该使用注册表的*RegNotifyChangeKey*函数，该函数会让一个线程进入睡眠，直至它们感兴趣的注册表区域中发生了变化。

注册表数据类型

注册表是一个数据库，其结构类似于磁盘卷的结构。注册表包含了**键**（*key*）和**值**（*value*）。键类似于磁盘的目录，而值则好比是磁盘上的文件。键是一个容器，可以包含其他的键（**子键**）和值。而值存储的是数据。最顶级的键是**根键**（*root key*）。在本节中，我们将相互交换着使用**子键**和**键**这两个词（只有根键才不是子键）。

键和值两者都借用了文件系统的命名规范。因此，你可以用名称“trade\mark”惟一标识一个存储在名为“trade”的键下面的、名为“mark”的值。此命名方案的一个例外是，每个键都可以有一个未命名的值。两个注册表编辑工具，即Regedit和Regedt32显示这些未命名值的方式不同：Regedit将未命名值显示为（Default），而Regedt32则使用<No Name>。

注册表值存储了不同类型的数据，它们可以是表4.1中所列出的15种类型之一。大多数注册表值是REG_DWORD、REG_BINARY或REG_SZ。REG_DWORD类型的值可以存储整数或者布尔值（开/关值）；REG_BINARY值可以存储超过32位的整数值，或者诸如加密后的口令之类的无结构数据；REG_SZ值存储字符串（当然是Unicode），可以表达诸如名称、文件名、路径和类型等元素。

表4.1 注册表值的类型

值类型	说 明
REG_NONE	没有值的类型
REG_SZ	固定长度的Unicode字符串
REG_EXPAND_SZ	可变长度的Unicdoe字符串，可以有内嵌的环境变量
REG_BINARY	任意长度的二进制数据
REG_DWORD	32位整数
REG_DWORD_LITTLE_ENDIAN	32位整数，低字节在前。等价于REG_DWORD
REG_DWORD_BIG_ENDIAN	32位整数，高字节在前
REG_LINK	Unicode符号链接

www.TopSage.com

续表

值类型	说 明
REG_MULTI_SZ	以NULL结尾的Unicode字符串的数组
REG_RESOURCE_LIST	硬件资源描述
REG_FULL_RESOURCE_DESCRIPTOR	硬件资源描述
REG_RESOURCE_REQUIREMENTS_LIST	资源需求
REG_QWORD	64位整数
REG_QWORD_LITTLE_ENDIAN	64位整数，低字节在前。等价于REG_QWORD
REG_QWORD_BIG_ENDIAN	64位整数，高位在前

REG_LINK类型特别值得注意，因为它让一个键可以透明地指向另一个键或者值。当你搜索注册表而碰到一个链接时，该搜索过程在此链接的目标处继续进行。例如，如果\Root1\Link是一个指向\Root2\RegKey的REG_LINK值，并且RegKey包含了值RegValue，那么，可以有两条路径标识RegValue：\Root1\Link\RegValue和\Root2\RegKey\RegValue。正如下一小节将要解释的那样，Windows自己也显式地使用了注册表链接：六个注册表根键中有三个其实是链接，它们指向另三个非链接的根键中的子键。链接是不保存的，每次引导之后它们都必须要被重新创建。

注册表逻辑结构

利用注册表中存储的数据，你可以画出注册表的组织结构。总共有6个根键（你不能加入新的根键，也不能删除已有的根键）可以存储信息，如表4.2所示。

表4.2 六个根键

根 键	说 明
HKEY_CURRENT_USER	存储一些与当前登录用户有关联的数据
HKEY_USERS	存储有关该机器上所有账户的信息
HKEY_CLASSES_ROOT	存储文件关联和组件对象模型（COM）的对象注册信息
HKEY_LOCAL_MACHINE	存储与系统有关的信息
HKEY_PERFORMANCE_DATA	存储与性能有关的信息
HKEY_CURRENT_CONFIG	存储关于当前硬件配置的一些信息

为什么根键的名称都以“H”开头？因为根键的名称代表了指向键（KEY）的Windows句柄（H）。正如第一章中所提及的那样，HKLM是代表HKEY_LOCAL_MACHINE的缩写。表4.3列出了所有的根键以及它们的缩写。下面的几小节详细地解释了这六个根键中的每一个根键的内容和用途。再次说明，有关这些键的内容细节，你可以参考Windows 2000资源工具箱中的“Technical Reference to the Windows 2000 Registry”帮助文件，或者Windows Server 2003开发工具箱中关于注册表的部分。

www.TopSage.com

表4.3 注册表根键

根 键	缩 写	说 明	链 接
HKEY_CURRENT_USER	HKCU	指向当前登录用户的用户轮廓	HKEY_USERS下对应于当前登录用户的子键
HKEY_USERS	HKU	包含了所有已加载用户轮廓的子键	不是链接
HKEY_CLASSES_ROOT	HKCR	包含了文件关联和COM注册信息	HKLM\SOFTWARE\Classes
HKEY_LOCAL_MACHINE	HKLM	占位容器——包含了其他的键	不是链接
HKEY_CURRENT_CONFIG	HKCC	当前硬件配置	HKLM\SYSTEM\CurrentControl-Set\Hardware Profiles\Current
HKEY_PERFORMANCE_DATA	HKPD	性能计数器	不是链接

HKEY_CURRENT_USER

HKCU根键包含了与当前本地登录用户的参数和软件配置有关的数据。它指向当前登录用户的用户轮廓，位于硬盘上的\Documents and Settings\<用户名>\Ntuser.dat中（要想知道根键是如何被映射到硬盘文件的，请参见本章后面的“注册表的内部机理”一节）。 无论何时当一个用户的轮廓信息被加载的时候，HKCU就会被创建出来，它是一个链接，指向HKEY_USERS下该用户的子键。表4.4列出了HKCU下面的一些子键。

表4.4 HKEY_CURRENT_USER子键

子 键	说 明
AppEvents	声音/事件的关联
Console	命令窗口设置（例如，宽度、高度和颜色）
Control Panel	屏幕保护程序、桌面方案、键盘和鼠标设置，以及可访问性和区域设置
Environment	环境变量定义
Keyboard Layout	键盘布局设置（例如U.S.或U.K.）
Network	网络驱动器的映射和设置
Printers	打印机连接设置
Software	用户特定的软件参数设置
UNICODE Program Groups	用户特定的开始菜单的组定义
Windows 3.1 Migration Status	文件状态数据（对于从Windows 3.x升级到Windows 2000和更高版本的系统）

www.TopSage.com

HKEY_USERS

HKU为系统中每个加载的用户轮廓和用户类注册数据库包含了一个子键。它也包含一个名为HKU\.DEFAULT的子键；这是一个链接，指向该系统的轮廓（对于那些运行在本地系统账户下的进程，它们会用到该轮廓信息；本章后面的“服务”一节中更加详细地讲述了有关该轮廓的信息）。譬如，这是Winlogon使用的轮廓，所以，改变了该轮廓中的桌面背景设置，就可以在登录屏幕上体现出来。当一个用户第一次登录到一个系统中，并且她的账户不依赖于一个漫游域轮廓（也就是说，用户的轮廓是在域控制器的指示下，从一个中心网络位置上获得的）的时候，系统以C:\Documents and Settings\Default User下存放的轮廓为基础，为她的账户创建一个轮廓。

系统将这些轮廓存储在哪里呢？此位置是由注册表值HKLM\Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ProfilesDirectory来定义的，该注册表值的默认设置是%SystemDrive%\Documents and Settings。ProfileList键也存放了该系统上所有出现过的轮廓的列表。针对每个轮廓的信息存放在一个对应的子键下面，其名称反映了该轮廓所对应账户的安全标识符（SID，Security Identifier）（有关SID的更多信息，请参见第8章）。在每个轮廓的键所存储的数据中，*ProfileLoadTimeLow*和*ProfileLoadTimeHigh*值中存放的是该轮廓最后一次加载的时间，*Sid*值中存放的是该账户SID的二进制表示，而在ProfileImagePath中存放的则是该轮廓所对应的储巢（hive，参见本章后面的“储巢”一节中的介绍）的硬盘路径。Windows XP和Windows Server 2003在用户轮廓管理对话框中显示了一个系统中存储的轮廓列表，如图4.1所示。你只需在控制面板的System小程序的Advanced视图中的User Profiles部分的Setting按钮上点击一下，就可以看到该对话框。

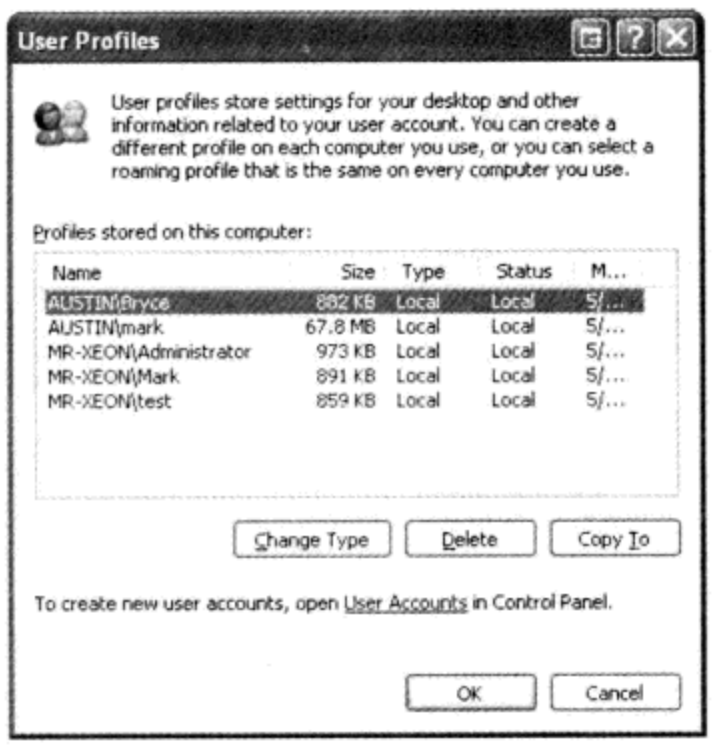


图4.1 用户轮廓管理对话框

www.TopSage.com



实验：观察轮廓加载和卸载

通过Runas命令，用一个当前并未登录到机器上的账户来激发一个进程，你就可以看到一个轮廓先被加载到注册表中，然后又被卸载出去的过程。当新进程正在运行的时候，运行Regedit，你可在HKEY_USERS下注意到加载进来的轮廓键。在终止了此进程以后，在Regedit中按一下F5键以便刷新视图，此轮廓应该已经不在了。

HKEY_CLASSES_ROOT

HKCR包含了两种类型的信息：文件扩展名关联和COM类注册信息。针对每个已注册的文件扩展名，都有一个对应的键。大多数键中包含一个REG_SZ值，指向HKCR中的另一个键，此键中包含了与该扩展名所代表的文件类相关联的信息。例如，HKCR\.xls指向了另一个键，比如HKCU\Excel.Sheet.8，其中包含了有关Microsoft Excel文件的信息。HKCU中其他的键包含了已注册到当前系统上的COM对象的详细配置信息。

HKEY_CLASSES_ROOT下的数据有以下两个来源途径。

- 针对每个用户的类注册信息：来源于HKCU\SOFTWARE\Classes （被映射至磁盘文件\Documents and Settings\<username>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat）下的数据；
- 整个系统范围的类注册信息：来源于HKLM\SOFTWARE\Classes。

针对每个用户的注册数据为什么要与系统全局范围的注册数据分开呢，其原因在于，这样做以后，可漫游的轮廓就能够包含这些特定的用户数据。它同时也关闭了一个安全缺口：非特权用户无法改变或者删除系统全局版本HKEY_CLASSES_ROOT中的键，因此也就无法影响到系统中应用程序的操作。非特权用户和应用程序可以读取系统全局范围的数据，也可以在系统全局数据中增加新的键和值（在它们的用户数据中这些键和值也被镜像过来了），但是它们只能修改其私有数据中已有的键和值。

HKEY_LOCAL_MACHINE

根键HKLM包含了所有系统全局范围的配置子键：HARDWARE、SAM、SECURITY、SOFTWARE和SYSTEM。

子键HKLM\HARDWARE维护了有关当前系统中的硬件，以及所有的从硬件设备至驱动程序的映射关系的描述信息。设备管理器工具（从控制面板中运行System小程序，点击Hardware标签，然后点击Device Manager）使你可以查看注册表硬件信息，这些信息只是简单地从HARDWARE键中读取出来的值。

www.TopSage.com



实验：利用HARDWARE键略施骗术

你只需要修改HKLM\HARDWARE\DESCRIPTION\System\CentralProcessor\0下的*ProcessorNameString*值，就可以骗你的同事或者朋友，使他们认为你的机器配置了最新的、最强大的处理器。控制面板中的System小程序在它的General页面中显示了*ProcessorNameString*值。然而，你在该键中所做的修改，比如说改成~MHZ，对于System小程序所显示的内容并没有任何影响，因为系统缓存了许多值，以便当应用程序调用某些函数来获取当前系统的处理器实际能力时，这些函数可以直接使用缓存的值。

HKLM\SAM存放了本地账户和组的信息，比如用户口令、组定义和域关联信息。作为域控制器运行的Windows Server系统将域账户和组的信息存放在活动目录（Active Directory）中。所谓活动目录，是一个用于存放域范围的各种设置和信息的数据库（本书并没有介绍活动目录）。 在默认情况下，SAM键上的安全描述符被配置成：即使管理员账户也不能访问。

HKLM\SECURITY存放了系统全局范围的安全策略和用户权限分配。HKLM\SAM被链接到HKLM\SECURITY\SAM下的SECURITY子键。在默认情况下，你不能查看HKLM\SECURITY或者HKLM\SAM\SAM键的内容，因为这些键的安全设置只有系统账户才能访问（本章后面将会极其详细地讨论系统账户）。如果你想要检查这些键的话，你可以改变此安全描述符，以便允许管理员组中的用户可以访问这些键，或者，你可以利用PsExec，在本地系统账户中运行Regedit（关于如何做到这一点，你可以参考相关的实验中所介绍的做法）。然而，仅仅看一眼这些数据是不会有多大用处的，因为这些数据是没有文档说明的，其中的口令已经通过一个单向映射关系加了密，也就是说，你是无法从加密形式得到一个口令的。

HKLM\SOFTWARE是Windows存储系统全局配置信息的地方，这些配置信息在系统引导的时候并不需要。而且，第三方应用程序也将它们的系统全局范围的设置存放在这里，比如应用程序文件和目录的路径、授权许可信息和过期日期信息。

HKLM\SYSTEM包含了引导系统所需要的全局配置信息，比如该加载哪些设备驱动程序、该启动哪些服务。因为这些信息对于启动系统是至关重要的，所以，Windows也在该键下维护了这些信息的一份拷贝，称为“*last known good control set*（**最后已知的好控制集**）”。由于维护了这样一份拷贝，因此，万一对当前控制集做了修改以后，系统无法正常引导了，管理员则可以选择以前的一份好的控制集。关于Windows何时声明当前控制集是“好”的，更详细的信息请参见“接受当前引导和最后已知的好控制集”一节。

HKEY_CURRENT_CONFIG

HKEY_CURRENT_CONFIG只是一个链接，指向HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current下的当前硬件轮廓。硬件轮廓使得管理员可以配置一些与基本的系统

www.TopSage.com

驱动程序设置不相同的设置。虽然每一次引导以后，底层最基本的轮廓可能有所不同，但通过该键，应用程序总是可以引用到当前的活动轮廓。对硬件轮廓是通过Hardware Profiles对话框来管理的，你只需在控制面板的System小程序中的Hardware页面上，点击Hardware Profiles部分的Settings就可以打开该对话框。在系统引导过程中，如果有多个轮廓的话，Ntldr将会提示你指定该使用哪个轮廓。

HKEY_PERFORMANCE_DATA

在Windows上，注册表也是访问性能计数器值的一种机制，无论这些计数器是从操作系统组件中来的，还是从服务器应用程序中来的。通过注册表来访问性能计数器，一个额外的好处是，远程性能监视工作几乎可以“免费”完成，因为通过常规的注册表API就可以很容易地访问远程的注册表信息。

你只需打开一个名为HKEY_PERFORMANCE_DATA的特殊的键，并且查询该键下面的值，就可以直接访问注册表性能计数器信息。如果你在注册表编辑器中查寻此特殊的键，你是无法找到的；这个键只能通过编程的方式，利用Windows注册表函数，例如*RegQueryValueEx*，才可以访问到。性能信息实际上并没有存储在注册表中，注册表函数利用该键来获得从性能数据提供者那里提供的信息。

你也可以利用性能数据帮助器（PDH，Performance Data Helper）API（Pdh.dll）中的性能数据帮助器函数来访问性能计数器信息。图4.2显示了在访问性能计数器信息过程中涉及的组件。

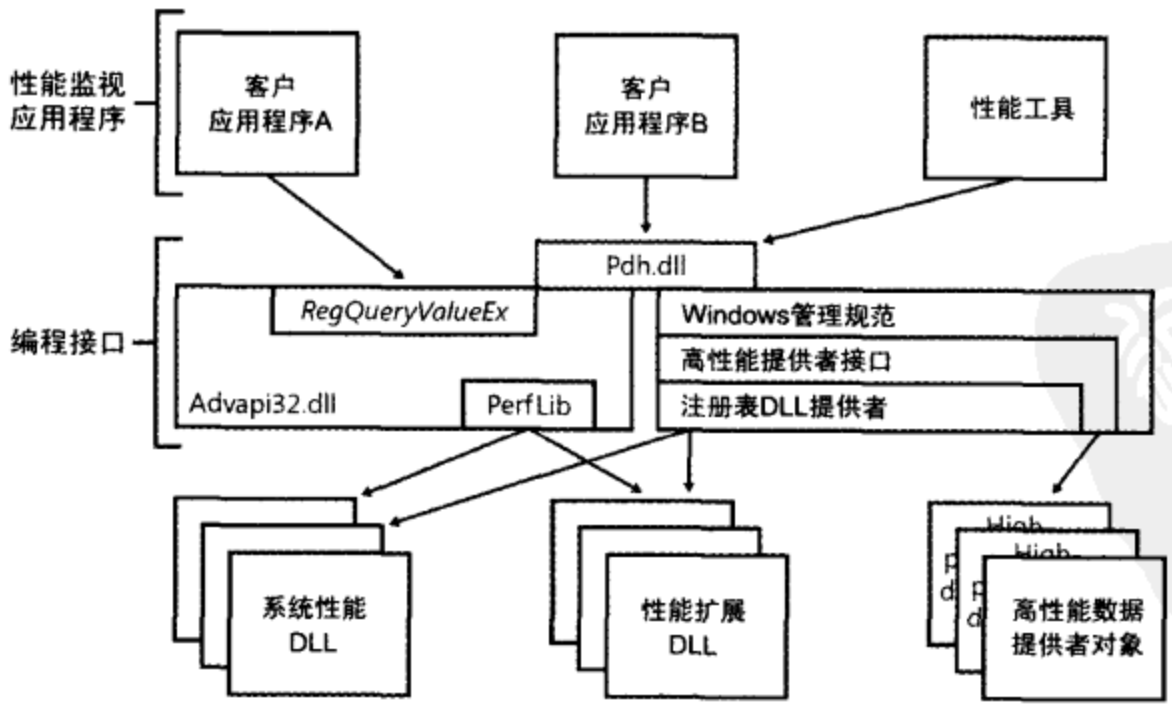


图4.2 注册表性能计数器体系结构

www.TopSage.com

注册表问题的诊断

因为系统和应用程序如此严重地依赖于配置信息来指导它们的行为，所以，系统和应用程序的失败可能是由于注册表数据和安全性的改变而导致的。当系统或者应用程序未能读取到它们认为总是能够访问到的配置数据时，它们的行为可能会全然不正常，比如崩溃、显示一些掩盖了根本原因的错误消息，或者未能以功能受限制的方式来执行。如果在系统或应用程序失败以后，不理解它们是如何访问注册表的，那么，要想知道哪些注册表键或者值被错误配置了，那是不可能的。在这样的情况下，*www.sysinternals.com*的Regmon工具也许能提供答案。

Regmon允许你监视注册表的一举一动。对于每一次注册表访问，Regmon都会向你显示这一次访问的时间、类型和结果。这些信息极其有助于看清楚应用程序和系统是如何依赖于注册表的，也有助于发现应用程序和系统将配置信息存储在哪里，以及可以诊断出那些“因漏掉了注册表键或值而发生”的应用程序问题。Regmon包含了高级的过滤和加亮显示功能，所以你可以快速地将注意力集中在与特定的键或值有关的活动上，或者与特定进程有关的活动上。

Regmon内部机理

Regmon依赖于一个设备驱动程序，该设备驱动程序是在运行时刻从Regmon的可执行映像文件中提取得到、然后再启动起来的。它第一次执行的时候，要求当前运行它的账户具有Load Driver特权，以及Debug特权；在同一次引导会话中，以后再执行的时候只需要Debug特权就可以了，因为一旦该设备驱动程序被加载以后，它就会驻留在系统中。

在Regmon可执行文件中实际上存储了三个驱动程序：一个用于Windows 95、Windows 98和Windows Millennium；一个用于Windows NT、Windows 2000和Windows XP；还有一个用于Windows Server 2003。之所以专门有一个驱动程序针对Windows Server 2003，是因为，在Windows NT、Windows 2000和Windows XP上，使一个驱动程序监视所有注册表活动的惟一途径是通过系统调用钩子（system-call hooking）机制，但是在Windows Server 2003上，驱动程序可以使用**注册表回调机制**（*registry callback mechanism*）来监视注册表的活动（Windows 95、Windows 98和Windows Millennium支持另一种不同的注册表监视机制）。

回顾一下第3章中的“系统服务分发”一节可知，系统服务函数地址被保存在内核中的系统服务分发表中。驱动程序只需将数组中某个函数的地址保存起来，并且用它自己的钩子函数来替换该表项，就可以钩住对应的系统服务。在执行了这些步骤以后，被钩住的系统服务一旦被调用，就会转向驱动程序中的钩子函数，它就可以检查或者修改该函数的参数，如果需要的话，也可以执行原来的系统服务函数。如果它调用了原来的函数，那么，驱动程序还可以检查该操作的结果，检查该函数返回的数据，比如与注册表值关联的数据等等。图4.3显示了Regmon如何在内核模式中截取注册表函数。

www.TopSage.com

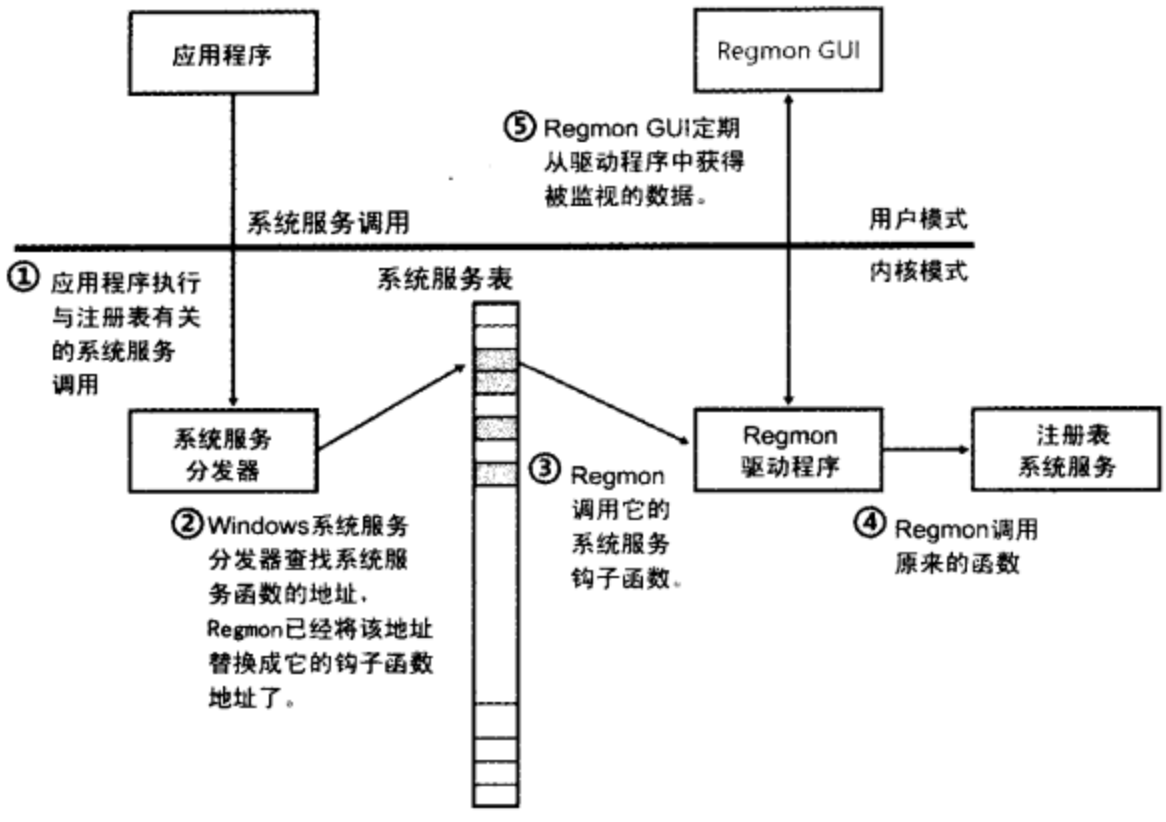


图4.3 Regmon使用了系统服务钩子功能

注册表回调机制是在Windows XP中引入的，然而，当Regmon在Windows XP上运行的时候，它仍然使用系统调用钩子机制，因为Windows XP上的回调机制并没有报告所有的注册表活动。当一个驱动程序使用回调机制的时候，它向配置管理器注册一个回调函数。在注册表系统服务执行过程中，配置管理器在特定的点上执行该驱动程序的回调函数，因此，该驱动程序对于注册表访问活动具有完全的可见性和控制能力。回调机制的另一类用户是一些反病毒产品，它们扫描注册表数据，看是否有病毒，或者防止未授权的进程修改注册表。



实验：在一个空闲的系统上查看注册表的活动

因为注册表实现了*RegNotifyChangeKey*函数，应用程序可以利用该函数来请求获得注册表的变化通知，而无须频繁地检查注册表，所以，当你在一个空闲的系统上运行Regmon时，你应该不会看到对同样的注册表键或者值进行重复访问。任何这样的行为（指重复性地访问同样的注册表键或值）都说明了一个应用程序写得极其糟糕，它在不必要地影响着系统的总体性能。

运行Regmon，经过几秒钟以后，检查输出纪录，看你是否能找到频繁的访问行为。如果能找到，则在这样的输出行上右键点击，从环境菜单中选择Process Properties，就可以查看到有关执行该活动的进程的细节情况。

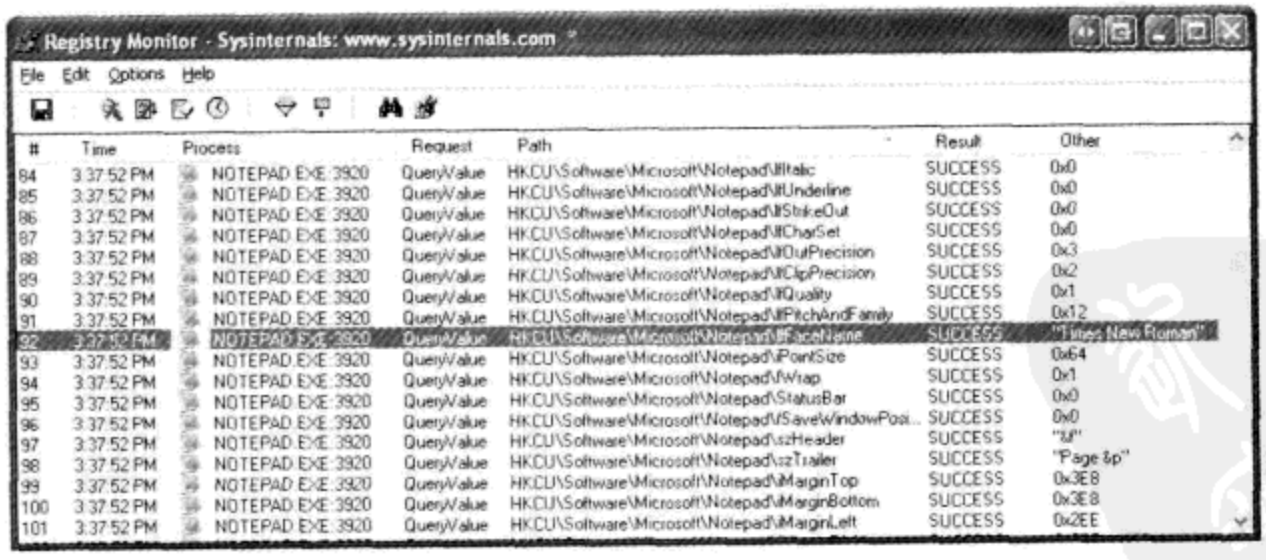
www.TopSage.com



实验：使用Regmon来找到应用程序注册表设置

在有些诊断情形中，你可能需要确定系统或应用程序将特定的设置存储在注册表的什么地方。本实验向你演示如何使用Regmon来找到Notepad的设置所在的注册表位置。如同大多数Windows应用程序一样，Notepad也保存了用户的偏好设置——比如字-卷行模式、字体和字体大小，以及窗口位置，因此每次Notepad被执行的时候可以恢复以前的设置。让Regmon来观察何时Notepad读取或者写入它的设置，你就可以找到这些设置存放在哪个（些）注册表键下。下面是具体的执行步骤。

1. 让Notepad保存一个设置，以便将来你在Regmon痕迹数据中可以很容易地搜索到该设置。做法很简单，你运行Notepad，将字体设置为Times New Roman，然后退出Notepad即可。
2. 运行Regmon。打开加亮显示过滤器对话框，在Include过滤器中输入**notepad.exe**。这将使得Regmon只记录那些在Process或者Path列中出现了notepad.exe的活动。
3. 再次运行Notepad，当它被激发起来以后，停止Regmon的事件捕获操作。做法很简单，你只要在Regmon的File菜单中关掉“Capture Events”即可。
4. 滚动到结果纪录的最上面一行，然后选中该行。
5. 按下Ctrl+F，以便打开Find对话框，然后搜索**times new**。Regmon应该会加亮显示一行，如下图中所示的那样，该行代表了Notepad从注册表中读取字体值的操作。该行邻近的其他一些操作应该与Notepad的其他设置有关。



6. 最后，在加亮显示的行上双击。Regmon将会执行Regedit（如果它不在运行的话），并且跳转到Notepad引用的注册表值上。

www.TopSage.com

Regmon诊断技巧

有两种基本的Regmon诊断技巧对于发现与注册表有关的应用程序或系统问题特别有效，它们是：

- 在Regmon的痕迹数据中，查看应用程序失败以前所做的最后一件事情；这个动作可能直指问题本身；
- 将失败了的应用程序的Regmon痕迹数据，与一个正常运行的系统的痕迹数据进行比较。

按照第一种方法，先运行Regmon，再运行应用程序。在失败发生的时候，回到Regmon中，停止记录数据（按下Ctrl+E）。然后，回到纪录的末尾，找到该应用程序失败（或者崩溃，或者挂起，或者其他的失败行为）以前最后执行的一些操作。从最后一行开始往回查找，检查被引用到的文件、注册表键，或者两者兼而有之——通常，这将有助于查明问题的原因所在。

当应用程序在一个系统上可以工作，而在另一个系统上却失败的时候，可以考虑使用第二种方法。分别在正常工作的系统上和失败的系统上捕获到该应用程序的痕迹数据，并且将这些输出数据保存到一个纪录文件中。然后用Microsoft Excel打开好的和坏的纪录文件（在Import向导中接受默认选项），删除其中的前三列（如果你不删除这前三列，比较的时候将会显示出每一行都不相同，因为前三列中包含了一些每次运行都不相同的信息，比如时间和进程ID）。最后，比较结果纪录文件（你也可以用WinDiff来比较，在Windows XP CD上的免费支持工具中包含了WinDiff工具，Windows 2000资源工具箱中也包含了该工具）。

在Regmon痕迹数据中，如果在Result一列中有“NOTFOUND”或者“ACCESS DENIED”的值，则这样的纪录项应该是你要仔细探查的。当一个应用程序试图读取一个不存在的注册表键或者值的时候，NOTFOUND就会被报告出来。在许多情况下，漏掉一个注册表键或者值是无关紧要的，因为如果进程未能从注册表中读取到它想要的设置，它就会选择使用默认的值。然而，在有些情况下，应用程序期望能找到非默认的值，如果非默认值不存在的话，则应用程序就会失败。

访问拒绝错误是一种常见的、与注册表有关的应用程序失败的根源，当应用程序没有权限访问一个它想要的键时，就会发生这种错误。如果应用程序不检查注册表操作的结果，或者不执行恰当的错误恢复过程，则应用程序就会失败。

一个可能值得怀疑的常见结果字符串是BUFROVERFLOW。它并不表明在收到此错误的应用程序中有一个缓冲区溢出漏洞。相反，配置管理器利用它来通知一个应用程序，它所指定的用于存放一个注册表值的缓冲区太小，因而容不下该值。应用程序开发人员通常利用这种行为，来确定一个缓冲区到底应该分配多大才能存放一个值。他们首先用一个0长度的缓冲区来执行一个注册表查询操作，结果返回一个缓冲区溢出错误，以及该查询操作期望的数据长度。然后，应用程序按照所指示的大小值分配一个缓冲区，重新读取该值。因此，你应该可以看到，返回BUFROVERFLOW的操作总是伴有一个成功的结果。

www.TopSage.com

在一个利用Regmon来诊断实际问题的例子中，它使得一个用户避免了完全重装他的Windows XP系统。其症状是，如果用户在启动Internet Explorer之前，不手工拨号打开Internet连接的话，Internet Explorer就会在启动的时候挂起。此Internet连接被设置成系统的默认连接，所以，一启动Internet Explorer，应该就会引发自动拨号到Internet（因为Internet Explorer被设置成：一启动就显示一个默认主页）。

在检查Internet Explorer启动活动的Regmon纪录的过程中发现，从Internet Explorer被挂起的纪录点开始往前回溯，有一个针对HKCU\Software\Microsoft\RAS Phonebook下的键的查询操作。用户报告说，在此之前他卸载了与该键关联的拨号程序，并且手工创建了拨号连接。因为拨号连接的名称与卸载的拨号程序的名称不相符，所以，看起来似乎是拨号卸载程序并没有删除该键，因而引起Internet Explorer被挂起。在删除了该键以后，Internet Explorer又正常工作了。

在非特权账户下或者登录/注销过程中的记录活动

一种常见的应用程序失败的情形是，应用程序在一个具有管理员组成员资格的账户下运行时可以正常工作，但是在一个非特权用户的账户下运行时却不能正常工作。正如以前所介绍的，为了执行Regmon，需要一些在正常情况下未分配给标准用户账户的安全特权，但是，利用Runas命令，在一个管理员账户下执行Regmon，你就可以捕获到在非特权用户的登录会话中执行的应用程序的痕迹数据。

如果一个注册表问题与账户的登录或注销有关，那么，你也可以采取特别的步骤，以便利用Regmon来捕获一个登录会话中有关这些阶段的痕迹数据。当一个用户注销的时候，在本地系统账户下运行的应用程序并不会终止，所以，你可以利用这一特性，让Regmon在一个先注销随后又登录的过程中一直保持运行。你或者利用Windows内置的At命令并指定*/interactive*标志，或者利用*www.sysinternals.com*的PsExec工具，就可以在本地系统账户下启动Regmon。PsExec工具的使用例子如下：

psexec –i –s –d c:\regmon.exe

-i开关指示PsExec，让Regmon的窗口出现在交互的控制台上；-s开关让PsExec在本地系统账户下运行Regmon；-d开关让PsExec激发Regmon，并且无须等待Regmon终止就退出。当你执行了该命令时，它执行起来的Regmon实例在你注销以后仍然还存活着，当你登录回来时，此Regmon实例仍然出现在桌面上，因此它可以捕获到注销和登录这两个动作的所有注册表活动。

在登录、注销、引导或者停机过程中监视注册表的另一种办法是利用Regmon的“记录引导”特性，你只要在Options菜单中选择“Log Boot”命令就可以打开该特性。下一次当你引导系统的时候，Regmon设备驱动程序就会从系统引导的早期开始，将注册表的活动记录到\Windows\Regmon.log中。它会一直在该文件中记录注册表的活动，直到磁盘空间耗尽、系统停机或者你运行Regmon为止。在Windows XP系统上，一个存储了从启动、登录、注销到停机的注册表痕迹的纪录文件，在通常情况下在50MB到150MB之间。

www.TopSage.com

注册表的内部机理

在这一小节中，你将会了解到，配置管理器（即实现注册表的执行体子系统）是如何组织注册表的磁盘文件的。你将会看到，当应用程序或者其他的操作系统组件读取或者改变注册表键和值的时候，配置管理器是如何管理注册表的。我们还将讨论一些注册表管理机制，配置管理器试图利用这些机制来确保注册表总是处于一种可恢复的状态，即使当注册表被修改时系统崩溃了也可以被恢复。

储巢

在磁盘上，注册表并不是简单的一个大文件，而是一组称为**储巢**（*hive*）的单独文件。每个储巢包含了一棵注册表树，有一个键作为该树的根，或者作为该树的起点。子键和它们的值存储在根的下面。你可能会认为，注册表编辑器工具所显示的根键与储巢中的根键一定是相互关联的，但实际情况并不是这样的。表4.5列出了注册表储巢，以及它们对应的磁盘文件名。除了用户轮廓的储巢以外，其他所有储巢的路径名都被编码进了配置管理器中。当配置管理器加载储巢的时候，包括系统轮廓的储巢，它都会在HKLM\SYSTEM\CurrentControlSet\Control\hivelist子键下的注册表值中记录下每个储巢的路径，如果储巢被卸载的话，则删除对应的路径（用户轮廓若未被引用，则会被卸载）。 它创建了根键，并且将这些储巢链接起来，以便建立起你所熟悉的、注册表编辑器所显示的注册表结构。

表4.5 磁盘文件对应于注册表中的路径

储巢注册表路径	储巢文件路径
HKEY_LOCAL_MACHINE\SYSTEM	\Windows\System32\Config\System
HKEY_LOCAL_MACHINE\SAM	\Windows\System32\Config\Sam
HKEY_LOCAL_MACHINE\SECURITY	\Windows\System32\Config\Security
HKEY_LOCAL_MACHINE\SOFTWARE	\Windows\System32\Config\Software
HKEY_LOCAL_MACHINE\HARDWARE	易失储巢
HKEY_LOCAL_MACHINE\SYSTEM\Clone	易失储巢（仅在Windows 2000上）
HKEY_USERS\<用户名的安全ID>	\Documents and Settings\<用户名>\Ntuser.dat
HKEY_USERS\<用户名的安全ID>_Classes	\Documents and Settings\<用户名>\Local Settings\Application Data\Microsoft\Windows\Usrclass.dat
HKEY_USERS\.DEFAULT	\Windows\System32\Config\Default

你会注意到，表4.5中列出的某些储巢是变化的，它们没有对应的关联文件。系统完全在内存中创建和管理这些储巢，因此这些储巢是临时的。系统在每次引导时创建这些易失的储巢。易失储巢的一个例子是HKLM\HARDWARE储巢，它保存了有关物理设备和这些设备分配到的资源的信息。系统每次引导时，就会进行资源分配和硬件检测，所以，不在硬盘上存储这些数据是合理的。

www.TopSage.com



实验：手工加载和卸载储巢

Windows 2000上的Regedt32，以及Windows XP和Windows Server 2003上的Regedit都具备了加载储巢的能力，通过它们的File菜单你就可以访问储巢。如果在诊断问题的时候，你想要从一个未能引导的系统或者从一个备份介质上查看或者编辑一个储巢，那么这种加载能力可能会非常有帮助。在这个实验中，你将使用Regedt32（如果你正在运行Windows 2000的话），或者Regedit（如果你正在运行Windows XP和Windows Server 2003的话）来加载HKLM\SYSTEM储巢的一个特殊版本：Windows Setup在安装过程中创建并存储在\Windows\Repair中的HKLM\SYSTEM储巢。

1. 储巢只能被加载到HKLM或者HKU的下面，所以，打开Regedit或者Regedt32，选择HKLM，然后从Regedit的File菜单或者Regedt32的Registry菜单中选择“Load Hive”。
2. 在Load Hive对话框中，跳转到\Windows\Repair目录下，选择System.bak，并且打开该文件。当出现提示的时候，输入**Test**作为该储巢对应的子键名，即它被加载进来后放在该子键下。
3. 打开新建的HKLM\Test键，查看该储巢的内容。
4. 打开HKLM\System\CurrentControlSet\Control\Hivelist，找到\Registry\Machine\Test，这演示了配置管理器如何在HiveList键下列出已被加载的储巢。
5. 选中HKLM\Test，再从Regedit的File菜单，或者Regedt32的Registry菜单中选择Unload Hive命令，以便卸载该储巢。

储巢的大小限制

在有些情况下，储巢的大小是有限制的。例如，Windows对于HKLM\SYSTEM储巢的大小是有限制的。它之所以这样做，是因为，在引导过程开始之初，当虚拟内存换页机制尚未启动时，Ntldr要将整个HKLM\SYSTEM储巢读入到物理内存中。Ntldr还要将Ntoskrnl和引导设备驱动程序加载到物理内存中，所以，它必须要对分配给HKLM\SYSTEM的物理内存数量进行限制（关于Ntldr在启动过程中所扮演的角色的更多信息，请参见第6章）。在Windows 2000上，Ntldr对它的大小有一个固定的上限，但是，在Windows XP和Windows Server 2003上，它的做法更加灵活，允许该储巢可以达到最高200MB或者该系统物理内存数量的四分之一，这取决于哪个更小一点。

在Windows 2000上，所有加载进来的注册表储巢合起来的大小也有一个限制。Windows 2000使用一种称为**换页池**（*paged pool*）的内核内存，将注册表储巢存放在该内存中，因此，加载进来的注册表数据的总数受到了当前可用的换页池内存数量的限制。内存管理器在初始化过程中创建的换页池的内存数量是建立在诸多因素基础之上的，比如系统中物理内存的数量。如果在一个系统上，内存管理器创建了最大可能内存数量的换页池，则注册表大小限制是

www.TopSage.com

376MB。因为如果没有足够的换页池内存留给其他的用途，系统将不会平稳地运行，所以，Windows 2000不会让注册表数据增长到超过换页池的80%，而且，它也遵从用户指定的注册表限额（如果该限额小于换页池80%内存数量的话）。从控制面板的System小程序的Advanced页面上，打开Performance Options对话框，点击Virtual Memory部分的Change按钮，你可以查看或者修改注册表限额设置，如在图4.4中所看到的那样。

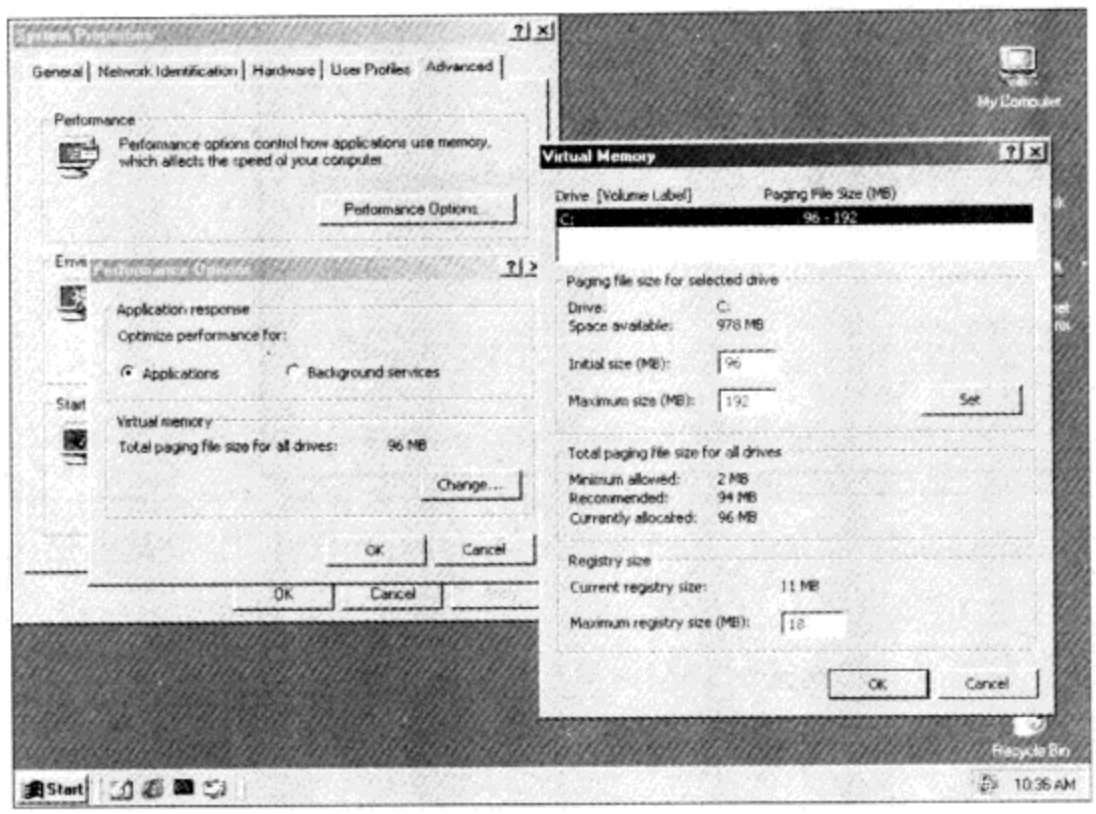


图4.4 Windows 2000注册表限额设置

加载到内存中的注册表储巢的总体大小的上限也直接导致了在一个运行终端服务的Windows 2000系统上，同时登录的用户数量也有一个限制，因为每个用户的轮廓都要占据这部分加载的储巢空间。因此，在Windows XP和Windows Server 2003上，配置管理器并不使用换页池；相反，它依赖于内存管理器的内存映射函数，只将在给定时间点上需要访问的注册表储巢部分映射到系统内存中。在Windows XP和Windows Server 2003上没有注册表限额，加载到内存中的储巢的总体大小也不会限制终端服务的扩展能力。

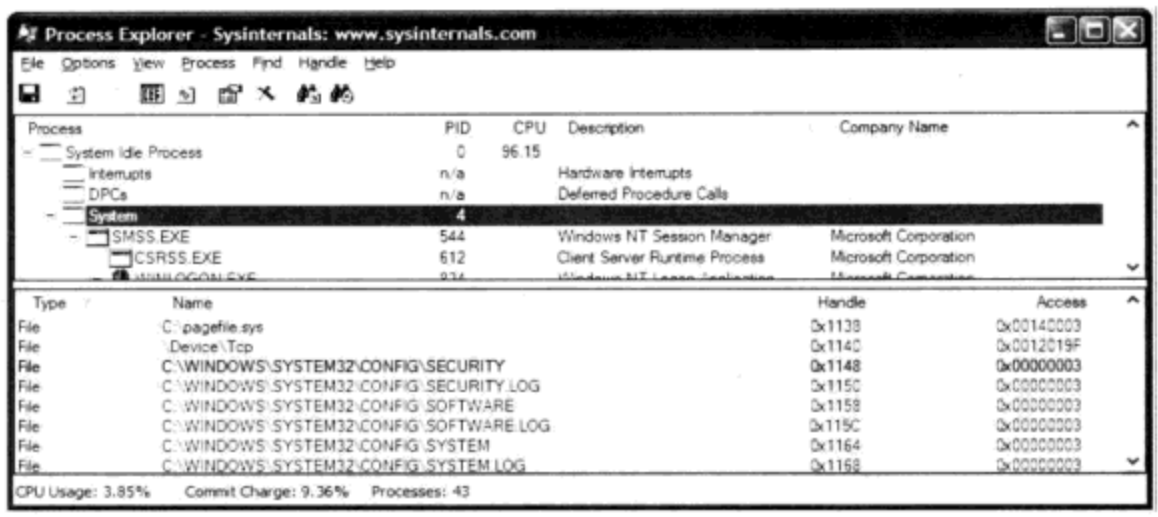


实验：查看储巢句柄

配置管理器在打开储巢的时候使用了内核句柄表（第3章中讲述），所以它可以在任何一个进程环境中访问储巢。相对于那些“通过使用驱动程序或者执行体组件，从而可在系统进程中访问那些必须与用户进程保持隔离的句柄”的方法而言，使用内核句柄表是一种很有效的方法。你可以使用*www.sysinternals.com*的进程管理器工具，来查看储巢句柄。在Windows 2000

www.TopSage.com

上，对象管理器报告说，内核句柄表中的句柄是在System Idle进程中被打开的，而在Windows XP和Windows Server 2003上，它报告说，它们是在System进程中被打开的。根据你当前正在运行的Windows版本，选择恰当的进程，然后在View菜单的Lower Pane View菜单项中选择Handles。根据句柄类型排序，滚动窗口，直至看到储巢文件，如下图所示。



一种被称为**符号链接**（*symbolic link*）的特殊键使得配置管理器有可能将这些储巢链接起来，从而构成了注册表。符号链接是一个键，它指导配置管理器到达另一个键。因此，HKLM\SAM键是一个符号链接，它连接到SAM储巢的根所在的键上。

储巢结构

配置管理器从逻辑上将一个储巢分成一些称为**块**（*block*）的分配单元，其方式类似于文件系统将一个磁盘分成簇。根据定义，注册表块的大小为4096字节（4KB）。当新的数据要扩展一个储巢时，该储巢总是按照块的粒度来增加。一个储巢的第一个块是**基本块**（*base block*）。基本块包含了有关该储巢的全局信息，包括一个特征签名*regf*（将该文件标识成储巢）、更新的序列号、一个时间戳（显示了该储巢上最后一个写操作发生的时间）、储巢格式版本号、校验和，以及该储巢文件的内部文件名（例如，\Device\HarddiskVolume1\WINDOWS\SYSTEM32\CONFIG\SAM）。当我们讲述如何将数据写到一个储巢文件中的时候，我们将会说明更新的序列号和时间戳的重要性。储巢格式版本号指明了该储巢内部的数据格式。在Windows 2000上，配置管理器使用储巢格式版本1.3；在Windows XP和Windows Server 2003上，为了与Windows 2000的漫游轮廓兼容，除了System和Software，对于所有其他的储巢，配置管理器使用格式版本1.3；而对于System和Software储巢，它使用版本1.5，因为新格式对于大的值和搜索作了特别的优化。

Windows将一个储巢所存储的注册表数据组织在一种称为**巢室**（*cell*）的容器中。一个巢室可以容纳一个键、一个值、一个安全描述符、一列子键，或者一列键值。在巢室数据开始处

www.TopSage.com

的一个域描述了该巢室数据的类型。表4.6详细列出了每个巢室数据类型。巢室的头是一个指定了该巢室大小的域。当一个巢室加入到一个储巢中，而且该储巢必须进行扩展才能包含该巢室时，系统创建一个称为**巢箱**（*bin*）的分配单元。一个巢箱是新巢室正好扩展到下一个块的边界的大小。系统将巢室的尾部和巢箱的尾部之间的任何空间都看成是空闲的空间，因而可以将它再分配给其他的巢室。巢箱也有头部，其中包含了一个特征签名*hbin*、一个记录了该巢箱在储巢文件中偏移量的域，以及该巢箱的大小。

表4.6 巢室数据类型

数据类型	说 明
键巢室	包含了一个注册表键（也称为**键节点**）的巢室。一个键巢室包含一个特征签名（对于一个键是*kn*，对于一个符号链接是*kl*）、该键最近一次更新的时间戳、该键的父键巢室的巢室索引、代表该键子键的子键列表巢室的巢室索引、该键的安全描述符巢室的巢室索引、一个代表该键类名的字符串键的巢室索引，以及该键的名称（比如CurrentControlSet）
值巢室	一个巢室，包含了关于一个键的值的信息。该巢室包含一个特征签名（*kv*）、该值的类型（例如REG_ DWORD或REG_BINARY），以及该值的名称（例如Boot-Execute）。一个值巢室也包含了另一个巢室的巢室索引，后者包含了前者对应的值的数据
子键列表巢室	由一系列键巢室的巢室索引所构成的巢室，这些键巢室是一个公共父键下的所有子键
值列表巢室	由一系列值巢室的巢室索引所构成的巢室，这些值巢室是一个公共父键的所有值
安全描述符巢室	包含了一个安全描述符的巢室。安全描述符巢室在其头部包含了一个特征签名（*ks*），以及一个引用计数。该引用计数值记录了所有共享该安全描述符的键节点的数目。多个键巢室可以共享同样的安全描述符巢室

通过使用巢箱，而不是巢室，来跟踪注册表的活动部分，Windows可以最小化其管理杂务。例如，在通常情况下系统分配和释放巢箱的频率，比分配和释放巢室的频率要低得多，这使得配置管理器可以更加有效地管理内存。当配置管理器将一个注册表储巢读进内存的时候，它可以选择只读入那些包含有巢室的巢箱（也就是说，活动的巢箱），而忽略那些空的巢箱。当系统在储巢中加入或者删除巢室的时候，储巢可能会包含空的巢箱，并且它们散布在活动的巢箱之间。这种情形类似于磁盘碎片，系统在磁盘上创建和删除文件的时候，就会产生磁盘碎片。当一个巢箱变成空的时候，配置管理器将任何相邻的空巢箱也加入到此空巢箱中，从而形成一个尽可能大的连续空巢箱。配置管理器也将相邻的已被删除的巢室连接起来，以便形成更大的空闲巢室（配置管理器只有当一个储巢尾部的储箱变成空闲的时候才会缩短该储巢。你可以通过Windows的*RegSaveKey*和*RegReplaceKey*函数，先备份注册表，再恢复注册表，从而达到压缩注册表的目的；Windows Backup工具就使用了这些函数）。

www.TopSage.com

储巢的结构是通过一些链接建立起来的，这些链接称为**巢室索引**（*cell index*）。每个巢室索引是一个巢室在储巢文件中的偏移。因此，巢室索引就像是一个指针，从一个巢室指向另一个巢室，配置管理器将巢室索引解释成相对于储巢起始处的偏移。例如，正如你在表4.6中所看到的，用于描述一个键的键巢室，它所包含的一个域指定了其父键的巢室索引；子键的巢室还指定了另一个巢室，它描述了那些隶属于该子键的子键。子键列表巢室包含了一列巢室索引，这些巢室索引指向其子键的键巢室。因此，假如你想要找到子键A的键巢室，并且A的父键是B，那么，你必须首先利用键B的巢室中的子键列表巢室索引，找到包含键B所有子键列表的那个巢室，然后利用该子键列表巢室中的巢室索引列表，找到键B的每个子键巢室。对于每个子键巢室，你检查该子键的名称（键巢室存储了键的名称）是否符合你想要找的那个子键，在这个例子中即为子键A。

巢室、巢箱和块之间的区别可能很容易让人混淆，所以，我们现在来看一个简单的注册表储巢的布局示例，以便澄清它们之间的区别。在图4.5中，示例性的注册表储巢文件包含了一个基本块和两个巢箱。第一个巢箱是空的，第二个巢箱包含了几个巢室。从逻辑上讲，该储巢只有两个键：一个是根键Root；另一个是Root的子键，即Sub Key。Root有两个值：*Val 1*和*Val 2*。通过一个子键列表巢室，可以定位到根键的子键；通过一个值列表巢室，可以定位到根键的值。第二个巢箱中的空闲空间是空的巢室。图4.5并没有显示这两个键的安全巢室，它们也应该出现在一个储巢中。

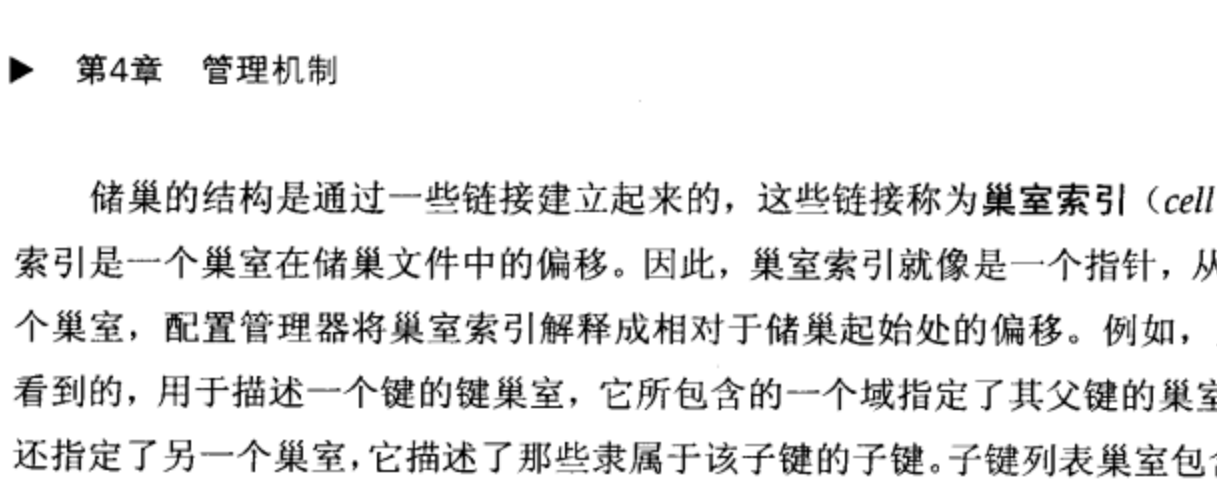


图4.5 一个注册表储巢的内部结构

图4.6显示了Disk Probe工具（Dskprobe.exe）的一个例子用法，它显示了SYSTEM储巢中的第一个巢箱。请注意该图像的右上方显示的是该巢箱的特征签名*hbin*。在巢箱特征签名的下方，你将会看到特征签名*nk*；它是键巢室的特征签名（*kn*）。该特征签名之所以反过来显示了，是因为x86计算机存储数据的方式。该巢室是SYSTEM储巢的根巢室，配置管理器在内部将它命名为$$$PROTO.HIV，正如跟在*nk*特征签名后面的名称所指示的那样。

www.TopSage.com

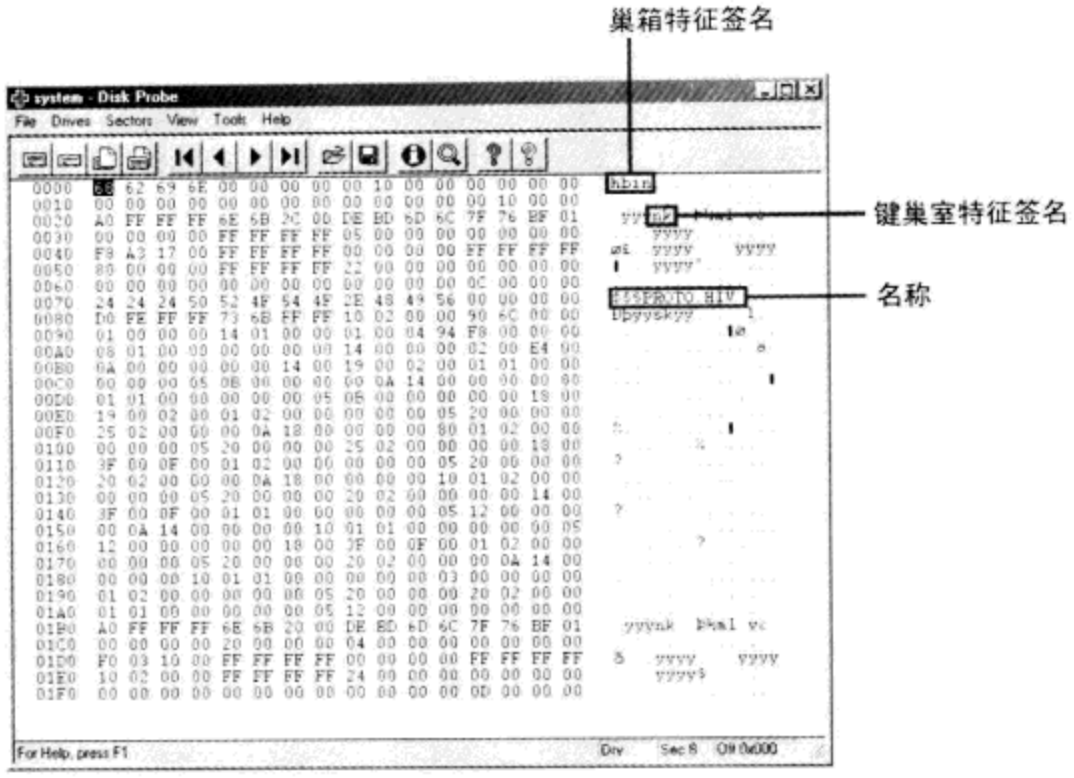


图4.6 SYSTEM储巢中第一个巢箱的二进制内容

为了优化对于值和子键的搜索过程，配置管理器按照字母表顺序来存储子键列表巢室。然后，当配置管理器要在一个子键列表中查找一个子键时，它可以执行二分搜索。配置管理器在列表的中间检查要查找的子键，如果按照字母表的顺序，配置管理器正在查找的子键的名称位于中间子键名称的前面，则配置管理器知道该子键位于子键列表的前半部分；否则该子键位于子键列表的后半部分。这样的切分过程一直进行下去，直到配置管理器找到了该子键，或者没有找到匹配的子键。然而，值列表巢室并非是排序的，所以，新的值总是被加到列表的尾部。

巢室映射表

配置管理器并不是在每次访问注册表的时候就会访问磁盘上的储巢映像文件。Windows 2000在内核的地址空间中保留了每个储巢的一个版本。当一个储巢初始化的时候，配置管理器先确定该储巢文件的大小，然后从内核的换页池中分配足够的内存来存储该储巢，之后将储巢文件读入到内存中（有关换页池的更多信息，请参见第7章）。因为所有加载的注册表储巢都被读入到换页池中，所以，这些注册表数据通常是Windows 2000中换页池的最大用户（要想检查换页池的分配情况，可使用Poolmon工具，第7章的辅助章节“监视内存池的使用”部分中讲述了此工具）。

在Windows XP和Windows Server 2003中，配置管理器当需要访问一个储巢中的部分内容时，就会将这些部分映射到内存中。它使用缓存管理器的文件映射函数，将16KB的视图映射到储巢文件中（有关缓存管理器的更多信息，请参见第10章）。为了避免储巢映射过程消耗掉

www.TopSage.com

缓存管理器的整个地址范围，配置管理器试图将每个储巢所映射的视图在任何时刻都不超过256个，其做法是，当到达了该限制时，它就会将最不经常使用（LRU）的视图解除映射。配置管理器仍然使用换页池来存储各种数据结构（包括最不经常使用的视图列表），但是，它的换页池用量也只是Windows 2000中用量的一小部分而已。



> **注** 在Windows XP和Windows Server 2003上，如果一个块的大小超过256KB的话，配置管理器将该块存储在换页池中，而不是对它进行映射。

如果储巢从来不增长，那么，配置管理器就可以在一个储巢的内存版本中执行所有的注册表管理工作，就好像该储巢是一个文件一样。给定一个巢室索引，配置管理器只需简单地将巢室索引（即在储巢文件中的偏移）加到内存中储巢映像的基地址上，就可以计算出该巢室在内存中的位置。在系统引导的早期，Ntldr对于SYSTEM储巢正是这样处理的：Ntldr将整个SYSTEM储巢作为一个只读储巢读入到内存中，并且将巢室索引加上内存中储巢映像的基地址，从而可以定位到这些巢室。不幸的是，随着储巢中加入新的键和值，储巢会随之增长，这意味着系统必须申请换页池的内存来存储新的巢箱，在这些巢箱中包含新加入的键和值。因此，在内存中存储注册表数据的换页池不必是连续的。



实验：查看储巢换页池的使用量

在Windows 2000上，没有管理层面上的工具可以向你显示注册表储巢（包括用户轮廓）所消耗的换页池内存数量。然而，*!reg dumppool*内核调试器命令不仅向你显示了每个加载的储巢消耗了换页池中多少页面，还向你显示了多少页面存储的是易失性数据，多少页面存储的是非易失性数据。该命令在输出的末尾打印出储巢内存使用总量（该命令只显示一个储巢名称的最后32个字符）。

```
kd> !reg dumppool

dumping hive at e20d66a8 (a\Microsoft\Windows\UsrClass.dat)
  Stable Length = 1000
  1/1 pages present
  Volatile Length = 0

dumping hive at e215ee88 (ettings\Administrator\ntuser.dat)
  Stable Length = f2000
  242/242 pages present
  Volatile Length = 2000
  2/2 pages present

dumping hive at e13fa188 (\SystemRoot\System32\Config\SAM)
  Stable Length = 5000
  5/5 pages present
  Volatile Length = 0

...
```


www.TopSage.com



实验：查看储巢内存使用量

在Windows XP和Windows Server 2003上，利用*!reg hivelist*命令，你可以查看储巢内存使用量的统计情况，包括它的稳定部分（在磁盘上）的大小和非易失性部分的大小、活动视图的数量，以及被锁定在内存中的视图的数量。请注意，这里的输出行被折行了：

```
-------------------------------------------------------------------------------------------
---------------------------
| HiveAddr |Stable Length|Stable Map|Volatile Length|Volatile Map|MappedViews|PinnedViews|
U(Cnt)| BaseBlock | FileName
-------------------------------------------------------------------------------------------
---------------------------
| e22f8b68 |       5000  | e22f8bc4 |       1000    |  e22f8ca0  |        2  |        0  |
    0| e2353000  | \Microsoft
\Windows\UsrClass.dat
| e28c3008 |     3fe000  | e1e84000 |       c000    |  e28c3140  |      116  |        0  |
    0| e1e48000  | ttings\Adm
inistrator\ntuser.dat
| e23ec008 |       1000  | e23ec064 |          0    |  00000000  |        1  |        0  |
    0| e23ee000  | \Microsoft
\Windows\UsrClass.dat
| e23ed760 |      37000  | e23ed7bc |       1000    |  e23ed898  |       14  |        0  |
    0| e23ef000  | ettings\Lo
calService\ntuser.dat
...
```

在上面的输出中，Administrator账户的轮廓储巢（其全路径为\Documents and Settings\Administrator\ntuser.dat，在上面的输出中被截断了）有116个映射的视图，大约占据4MB（十进制0x3f00）大小的空间。*!reg viewlist*命令将把你指定的储巢中映射的视图转储出来。当该命令作用在上述*!reg hivelist*命令输出的第一个UsrClass.dat储巢上时，其输出如下所示：

```
kd> !reg viewlist e22f8b68

   0  Pinned Views ; PinViewListHead = e22f8da0 e22f8da0

   2  Mapped Views ; LRUViewListHead = e1cf4448 e1c5d440
-------------------------------------------------------------------------------------------
------------------
| ViewAddr |FileOffset|   Size   |ViewAddress|   Bcb    |    LRUViewList     |
PinViewList| UseCount |
-------------------------------------------------------------------------------------------
------------------
| e1cf4448 |        0 |     4000 | c9a40000  | 8a4bb0e9 | e1c5d440  e22f8d98 | e1cf4450
e1cf4450 |        0 |
| e1c5d440 |     4000 |     2000 | c9a44000  | 8a4bb0e9 | e22f8d98  e1cf4448 | e1c5d448
e1c5d448 |        0 |
-------------------------------------------------------------------------------------------
------------------
```


www.TopSage.com

在上述输出中，ViewAddress列显示了*hivelist*命令报告的关于该储巢的两个视图的地址。使用调试器的*db*命令来转储出第一个视图的地址处的内存内容，表明它映射了该储巢的基本块，这通过它的*regf*特征签名就可以识别出来：

```
kd> db c9a40000
c9a40000  72 65 67 66 d5 01 00 00-d5 01 00 00 cc 20 43 c7  regf......... C.
c9a40010  3d 40 c4 01 01 00 00 00-03 00 00 00 00 00 00 00  =@..............
c9a40020  01 00 00 00 20 00 00 00-00 50 00 00 01 00 00 00  .... ....P......
c9a40030  5c 00 4d 00 69 00 63 00-72 00 6f 00 73 00 6f 00  \.M.i.c.r.o.s.o.
c9a40040  66 00 74 00 5c 00 57 00-69 00 6e 00 64 00 6f 00  f.t.\.W.i.n.d.o.
c9a40050  77 00 73 00 5c 00 55 00-73 00 72 00 43 00 6c 00  w.s.\.U.s.r.C.l.
c9a40060  61 00 73 00 73 00 2e 00-64 00 61 00 74 00 00 00  a.s.s...d.a.t...
c9a40070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

为了处理内存中存放储巢数据的非连续内存地址，配置管理器采用了一种类似于Windows内存管理器用来将虚拟内存地址映射为物理内存地址的策略。配置管理器采用一种两层方案，如图4.7所示，它接受一个巢室索引（即储巢文件中的一个偏移）作为输入，返回该巢室索引所在的块的内存地址，以及该巢室所在的块的内存地址。前面提到过，一个巢箱可以包含一个或者多个块，储巢以巢箱为粒度进行增长，所以，Windows总是用一块连续区域的内存来代表一个巢箱。因此，一个巢箱内部的所有块都出现在同一个缓存管理器视图的内部（在Windows XP和Windows Server 2003上），或者换页池的同一部分中（在Windows 2000上）。

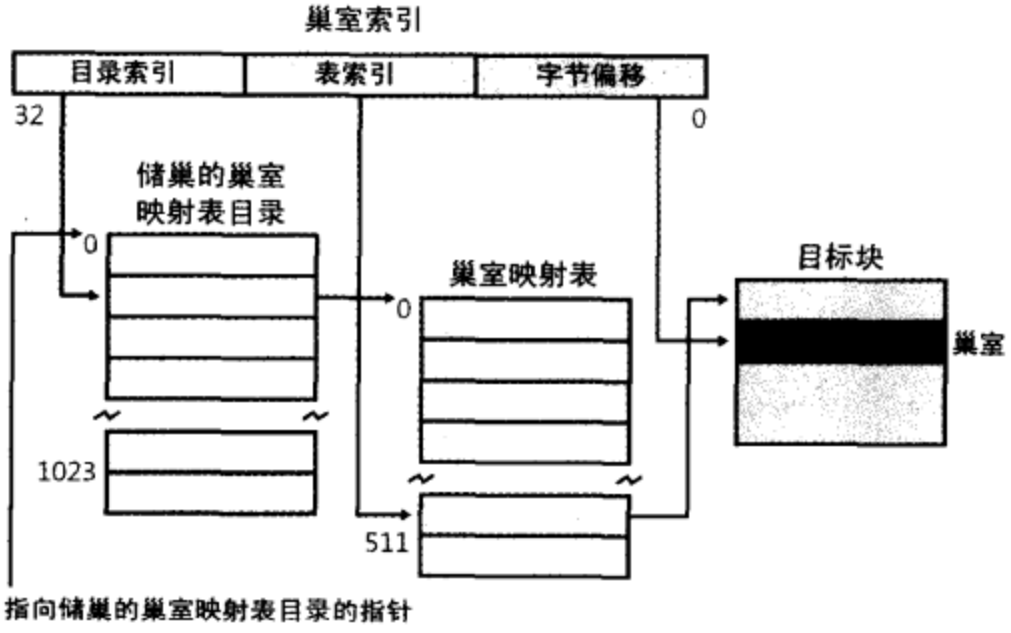


图4.7 巢室索引的结构

为了实现这种映射，配置管理器将一个巢室索引从逻辑上分成多个域，就如同内存管理器将一个虚拟地址分成多个域一样。Windows将一个巢室索引的第一个域解释成一个储巢的巢室映射表目录中的一个索引。巢室映射表目录包含1024个项目，每个项目指向一张巢室映射表，

www.TopSage.com

每张巢室映射表包含512个表项。在巢室映射表中的表项是由巢室索引中的第二个域来指定的。该表项指定了此巢室的巢箱内存地址和块内存地址。在Windows XP和Windows Server 2003中，并不是所有的巢箱都必须映射到内存中，如果查找一个巢室得到的结果地址为0，则配置管理器将该巢箱映射到内存中，如果有必要的话，将它维护的LRU表中的另一个巢箱解除映射。

在转译过程的最后一步，配置管理器将巢室索引的最后一个域解释成上述指定块中的一个偏移量，以便精确地定位到内存中的巢室。当一个储巢初始化的时候，配置管理器动态地创建这些映射表，为储巢中的每一个块指定一个表项；当储巢的大小有必要改变时，它也会从巢室目录中删除和增加映射表。

注册表名字空间和操作

配置管理器定义了一个“**键对象**（*key object*）”的对象类型，从而将注册表的名字空间与内核的总名字空间整合在一起。配置管理器在Windows名字空间的根上插入了一个名为“**注册表**（*Registry*）”的键对象，用作指向注册表的基本入口点。Regedit按照HKEY_LOCAL_MACHINE\ SYSTEM\CurrentControlSet这样的形式来显示键的名称，但是，Windows子系统将这样的名称转译成相应的对象名字空间的形式（比如\Registry\Machine\System\CurrentControlSet）。当Windows对象管理器解析此名称时，它首先碰到Registry名称的键对象，然后将名称中剩下的部分交给配置管理器。配置管理器接管了名称解析过程，它查找自己内部的储巢树，以便找到期望的键或值。在我们介绍典型的注册表操作的控制流程以前，我们首先来讨论键对象和**键控制块**（*key control block*）。无论何时，当一个应用程序打开或者创建一个注册表键的时候，对象管理器会给应用程序一个句柄，让它通过此句柄来引用该键。对应于一个键对象的句柄是配置管理器在对象管理器的帮助下分配的。通过对象管理器的对象支持，配置管理器得以充分利用对象管理器提供的安全性和引用计数功能。

对于每个打开的注册表键，配置管理器也分配一个键控制块。键控制块保存了该键的全路径名，包含了该控制块所引用的键节点的巢室索引，还包含一个标志。该标志指明，当该键最后一个句柄关闭的时候，配置管理器是否需要删除该键控制块所引用的键巢室。Windows把所有的键控制块都放在一张散列表中，从而可以快速地按照名称来搜索已有的键控制块。键对象指向它所对应的键控制块，所以，如果两个应用程序打开同一个注册表键的话，则每个应用程序都会收到一个键对象，并且这两个键对象指向一个公共的键控制块。

当一个应用程序打开一个已有的注册表键的时候，控制流程从应用程序在一个注册表API中指定了该键的名称开始（该注册表API调用了对象管理器的名称解析过程）。对象管理器碰到了在名字空间中属于配置管理器的注册表键对象以后，将路径名交给配置管理器。配置管理

www.TopSage.com

器利用内存中的储巢数据结构来搜索键和子键，以便找到指定的键。如果配置管理器找到了该键巢室，它就会搜索键控制块树，以确定该键是否是打开的（被同一个应用程序或者另一个应用程序打开）。此搜索过程已经作了优化，以便总是从最近的，并且已经有一个打开的键控制块的祖先开始找起。例如，如果一个应用程序要打开\Registry\Machine\Key1\Subkey2，并且\Registry\Machine已经打开了，那么，解析过程使用\Registry\Machine的键控制块作为起点。如果该键是打开的，则配置管理器增加已有的键控制块的引用计数。如果该键尚未被打开，则配置管理器分配一个新的键控制块，并且将它插入到树中。然后，配置管理器分配一个键对象，使该键对象指向此键控制块，再将控制返回给对象管理器，对象管理器则返回一个句柄给应用程序。

当一个应用程序创建一个新的注册表键时，配置管理器首先找到此新键的父键的键巢室。然后，配置管理器对于此新键将要存入的储巢中的空闲巢室列表进行搜索，以确定是否有足够大的巢室来容纳此新键巢室。如果没有足够大的空闲巢室，则配置管理器分配一个新的巢箱，并将它分配给新键巢室，然后将巢箱尾部的任何空间也放到空闲巢室列表中。新键巢室中填充了各种有关的信息，包括该键的名称，配置管理器将该键巢室加入到其父键的子键列表巢室的子键列表中。最后，系统也将父巢室的巢室索引保存在新子键的键巢室中。

配置管理器利用键控制块的引用计数来决定何时应该删除该键控制块。如果有多个句柄引用了一个键控制块中的键，则当所有这些句柄都关闭时，键控制块中的引用计数变为0，这表明该键控制块不再被需要了。如果一个应用程序调用一个API来删除该键，并设置了删除标志，那么，配置管理器可以从该键的储巢中删除对应的键，因为它知道没有其他的应用程序将该键保持为打开的状态。



实验：查看键控制块

你可以在内核调试器中，通过*!reg openkeys*命令来列出一个系统中所有已被分配的键控制块。或者，如果你想要查看某一个已打开的键的键控制块，则可以使用*!reg findkcb*命令：

```
kd> !reg findkcb \registry\machine\software\microsoft
Found KCB = e1034d40 :: \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
```

你可以利用*!reg kcb*命令来检查一个已报告的键控制块，如下所示：

```
kd> !reg kcb e1034d40

Key                 : \REGISTRY\MACHINE\SOFTWARE\MICROSOFT
RefCount            : 1f
Flags               : CompressedName, Stable
```


www.TopSage.com

```
ExtFlags          :
Parent            : 0xe1997368
KeyHive           : 0xe1c8a768
KeyCell           : 0x64e598 [cell index]
TotalLevels       : 4
DelayedCloseIndex: 2048
MaxNameLen        : 0x3c
MaxValueNameLen   : 0x0
MaxValueDataLen   : 0x0
LastWriteTime     : 0x 1c42501:0x7eb6d470
KeyBodyListHead   : 0xe1034d70 0xe1034d70
SubKeyCount       : 137
ValueCache.Count  : 0
KCBLock           : 0xe1034d40
KeyLock           : 0xe1034d40
```

Flags域显示了该名称是按照压缩格式存储的，SubKeyCount域显示了该键有137个子键。

稳定可靠的存储

为了确保非易失性的注册表储巢（每个都是一个磁盘文件）总是处于一种可恢复的状态，配置管理器使用了**日志储巢**（*log hive*）。每个非易失性储巢都有一个关联的日志储巢，它是一个隐藏文件，与对应的储巢有同样的基本文件名，扩展名为“*.log*”。例如，如果你检查你的系统中的\Windows\System32\Config目录（将“Show Hidden Files And Folders”文件夹选项选中），你将会看到System.log、Sam.log以及其他的.log文件。当一个储巢初始化的时候，配置管理器分配一个位阵列，其中每一个位代表了该储巢中一个512字节大小的部分，或者称为**扇区**（*sector*）。这个阵列称为**脏扇区阵列**（*dirty sector array*），因为阵列中的“on”位表示系统已经修改了内存里该储巢中的对应扇区，因此系统必须将该扇区写到储巢文件中（“off”位表示对应的扇区是最新的，与内存中储巢的内容一致）。

当创建一个新的键或值，或者修改一个已有的键或值的时候，配置管理器在该储巢的脏扇区阵列中对于所改变的扇区做上标记。然后，配置管理器调度一个延迟的写操作，或者称为**储巢同步**（*hive sync*）。执行该储巢延迟的写操作的系统线程在储巢同步请求之后5秒钟被唤醒，它把所有储巢的脏储巢扇区从内存中写到磁盘上的储巢文件中。因此，系统同时也会刷新在发出储巢同步请求的时间点和真正执行储巢同步的时间点之间的这段时间内的所有注册表修改操作。当一个储巢同步发生时，下一个储巢同步至少要等到5秒钟以后。



注 在Windows Server 2003上，你可以通过设置注册表值HKLM\System\CurrentControlSet\Session Manager\Configuration Manager\RegistryLazyFlushInterval，来改变默认的、储巢延迟写线程所使用的5秒钟延迟。

www.TopSage.com

如果延迟写入器（lazy writer）只是简单地把一个储巢的所有脏扇区写到储巢文件中，并且在写操作过程中系统崩溃了，那么，储巢文件将处于一种不一致（破坏的）和不可恢复的状态。为了避免这样的情形发生，延迟写入器首先将储巢的脏扇区阵列和所有的脏扇区写到储巢的日志文件中，如果有必要的话可以增加日志文件的大小。然后，延迟写入器更新该储巢基本块中的一个序列号，再将脏扇区写到储巢中。当延迟写入器结束时，它更新该基本块中的另一个序列号。因此，如果在写储巢操作的过程中系统崩溃的话，则下一次重新引导时，配置管理器将会注意到，储巢的基本块中的两个序列号不匹配。于是，配置管理器可以用储巢的日志文件中的脏扇区来更新该储巢，从而使储巢向前滚过去。然后，该储巢被更新，并且其状态是一致的。

在Windows 2000中，为了进一步保护关键的SYSTEM储巢的完整性，配置管理器在磁盘上维护了SYSTEM储巢的一份镜像。如果你看一看Windows 2000中\Windows\System32\Config目录下的非隐藏文件，你就会看到System.alt文件。System.alt是**替补储巢**（*alternate hive*）。无论何时，当储巢同步操作将脏扇区刷新到SYSTEM储巢中时，该同步操作也会更新System.alt文件。如果系统引导的时候，配置管理器检测到SYSTEM储巢已被破坏，它就会试图加载该储巢的替补文件。如果替补储巢可以使用的话，那么它就会使用该替补储巢来更新原来的SYSTEM储巢。

Windows XP和Windows Server 2003并没有维护这样的System.alt储巢，因为这些Windows版本上的NTLDR知道该如何对System.log文件进行处理，以便将一个在关机或者系统崩溃过程中造成了状态不一致的SYSTEM储巢正确地进行加载和更新。Windows Server 2003还有其他的容忍注册表破坏的增进措施。在Windows Server 2003以前，如果配置管理器读入的基本块、巢箱或者巢室中包含了未能通过基本一致性检查的数据，则配置管理器将会导致系统崩溃。Windows Server 2003中的配置管理器更能够容忍这样的问题，如果读入的数据被破坏得不是很严重的话，则它可以重新对破坏的数据结构进行初始化，在此过程中可能会删除一些子键，然后继续执行。如果它不得不求助于自治愈操作的话，则它会弹出一个系统错误对话框来通知用户。



> **注** 当你看一看\Windows\System32\Config中的隐藏文件，你将会看到一个名为System.sav的文件。System.Sav是SYSTEM储巢的一个版本，它是System储巢的初始拷贝，是Windows Setup从安装介质上拷贝过来的。

注册表优化

配置管理器作了一些非常显著的性能优化。首先，实际上每一个注册表键都有一个安全描述符，它可以起到保护该键的作用。然而，为储巢中的每一个键都保存一个惟一的安全描述符拷贝将是十分低效的，因为同样的安全设置往往应用在注册表的整棵子树上。在Windows 2000中，当系统将安全性作用到一个键上时，配置管理器首先检查与该键的父键所关联的安全描述符，然后检查该父键的所有子键。如果这些安全描述符中的任何一个与系统要作用在该键上的

www.TopSage.com

安全描述符相匹配，则配置管理器只是简单地让该键共用这个已有的描述符，它采用一个引用计数来跟踪有多少个键共用了同一个描述符。在Windows XP和Windows Server 2003中，每当有新的安全性被应用到一个键上时，配置管理器检查一个储巢中的安全描述符池（该池中存放的是该储巢中独一无二的安全描述符），然后它为该键共用任何已有的描述符，从而确保在一个储巢中任何一个独一无二的安全描述符至多只有一份拷贝。

配置管理器也对一个储巢中键或者值的名称的存储方式做了优化。尽管注册表具备完全的Unicode能力，它使用Unicode编码方式来表述所有的名称，但是，如果一个名称中只包含ASCII字符，那么，配置管理器在该储巢中，按照ASCII形式来存储该名称。当配置管理器读入此名称时（比如当执行名称查询时），它在内存中将该名称转换成Unicode形式。按照ASCII形式来存储名称，可以显著地降低一个储巢的大小。

为了使内存使用量尽可能地减到最小，键控制块中并没有存储完整的键路径名。相反，它们只是引用了一个键的名称。例如，一个引用了\Registry\System\Control的键控制块只是引用了名称*Control*，而不是全路径名。进一步的内存优化是，配置管理器使用键名称控制块来存储键的名称，对于所有具有同样名称的键，它们的键控制块共用同样的键名称控制块。为了优化性能，配置管理器将键控制块名称存储在一个散列表中，以便于快速查询。

为了能够快速地访问键控制块，配置管理器将频繁被访问的键控制块存储在一个缓存表中，该表也被配置成一张散列表。当配置管理器需要查找一个键控制块时，它首先检查此缓存表。最后，配置管理器还有另一个缓存：延迟的关闭表。它存储的键控制块是应用程序关闭的，所以，一个应用程序可以很快地重新打开一个刚刚被关闭的键。当配置管理器将最近被关闭的键控制块加入到延迟的关闭表中时，它也除掉该表中那些最老的键控制块。

4.2 服务

几乎每一个操作系统都有一种在系统启动时刻启动进程的机制，这些进程提供了一些不依赖于任何交互式用户的服务。在Windows中，这样的进程称为**服务**（*service*）或者**Windows服务**（*Windows Service*），因为它们依赖于Windows API与系统进行交互。Windows服务类似于UNIX的守护进程，它们通常实现了客户/服务器应用的服务器一方。Windows服务的一个例子可能是Web服务器，因为不管是否有人登录到机器上，它必须保持运行；当系统启动的时候，它必须开始运行，这样，管理员就不用总是记着，也不用待在机器跟前，并将服务启动起来。

Windows服务是由三个组件构成的：服务应用、服务控制程序（SCP，service control program），以及服务控制管理器（SCM，service control manager）。首先，我们讲述服务应用、服务账号，以及SCM的操作。然后我们将解释，那些自动启动的服务是如何在系统引导的过程中被启动起来的。我们还将介绍当一个服务在启动过程中失败时SCM所采取的步骤，以及SCM停掉服务的方法。

www.TopSage.com

服务应用

像Web服务器这样的服务应用至少包含一个作为Windows服务而运行的可执行程序。若用户想要启动、停止或者配置一个服务，他可以使用SCP。虽然Windows内置的SCP提供了一般性的启动、停止、暂停和继续功能，但是，有些服务应用包含了它们自己的SCP，管理员通过这些SCP，可以指定一些特定于他们所管理的服务的特殊设置。

服务应用也只是简单的Windows可执行程序（GUI风格或者控制台风格），加上一些代码来接收SCM的命令，以及将应用的状态反馈给SCM。因为大多数服务没有用户界面，所以它们都是按照控制台程序来创建的。

当你安装一个包含有服务的应用时，该应用的安装程序必须向系统注册它的服务。为了注册该服务，安装程序调用Windows的*CreateService*函数，这是一个在Advapi32.dll（\Windows\System32\Advapi32.dll）中实现的、与服务有关的函数。Advapi32，即“高级API（Advanced API）”DLL，实现了所有的客户端SCM API。

当一个安装程序通过调用*CreateService*来注册服务时，就会发送一个消息给该服务将要驻留的机器上的SCM。然后，SCM为该服务在HKLM\SYSTEM\CurrentControlSet\Services下创建一个注册表键。Service键是SCM数据库的非易失部分。针对每个服务的键定义了该服务所在的可执行映像文件的路径，以及一些参数和配置选项。

在创建了一个服务以后，一个安装程序或者管理应用程序可以通过*StartService*函数来启动该服务。因为有些基于服务的应用也必须在引导过程中进行初始化，所以，像下面这样的情形也就不足为奇了：安装程序将一个服务注册成一个自动启动的服务，并且请用户重新引导整个系统，以便完成安装过程，并且让SCM在系统引导过程中启动此服务。

当一个程序调用*CreateService*时，它必须指定许多参数，由这些参数来描述该服务的特征。这样的特征包括该服务的类型（它是运行在自己独立的进程中，还是与其他的服务共享一个进程）、该服务可执行映像文件的位置、一个可选的显示名、一个可选的账户名和口令（用于在特定账户的安全环境中启动该服务）、一个启动类型（指定该服务是在系统引导时自动启动，还是在SCP的指示下手工启动）、一个错误代码（指示了如果该服务在启动时检测到错误的话该如何反应），以及一些可选的信息（如果该服务自动启动的话），这些信息指定了该服务相对于其他的服务该何时启动。

SCM将每一个特征存储为该服务的注册表键下的一个值。图4.8显示了一个服务的注册表键的例子。

www.TopSage.com

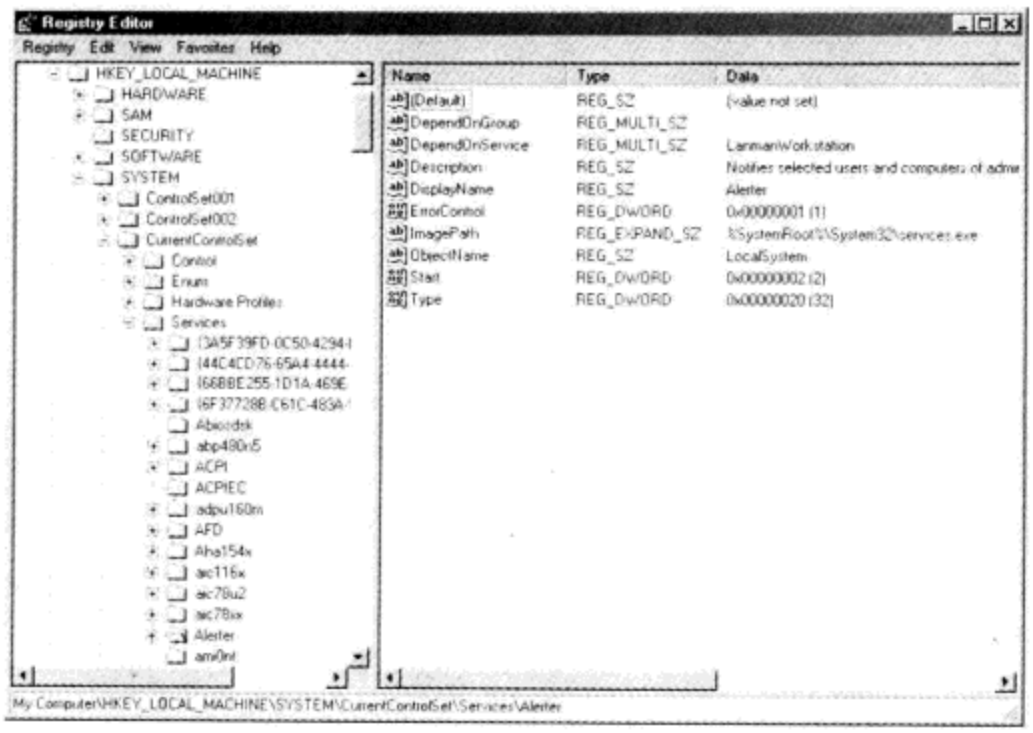


图4.8 一个服务注册表键的例子

表4.7列出了所有的服务特征，其中许多可以被应用在设备驱动程序上（并不是每一个特征都被应用在每一种类型的服务或者设备驱动程序上）。如果一个服务需要存储一些它私有的配置信息，那么，习惯的做法是，在它的服务键下面创建一个名为Parameters的子键，并将这些私有信息保存在该子键下的值中。然后，该服务可以利用标准的注册表函数来获得这些值。



> 注 在一个服务被删除以前，SCM是不会访问该服务的Parameter子键的；当该服务被删除时，SCM就会删除该服务的整个键，其中也包括像Parameters这样的子键。

表4.7 服务和驱动程序的注册表参数

值设置	值的名称	值设置的说明
Start	SERVICE_BOOT_START (0)	Ntldr或者Osloader预先加载该驱动程序，所以在引导过程中该驱动程序一直待在内存中。这些驱动程序恰好在SERVICE_SYSTEM_START驱动程序之前被初始化
	SERVICE_SYSTEM_START (1)	在内核初始化过程中，在SERVICE_BOOT_START驱动程序已初始化之后，该驱动程序被加载到内存中，并进行初始化
	SERVICE_AUTO_START (2)	在SCM进程（Services.exe）启动以后SCM启动该驱动程序或者服务
	SERVICE_DEMAND_START (3)	SCM根据需要启动该驱动程序或者服务

www.TopSage.com

续表

值设置	值的名称	值设置的说明
	SERVICE_DISABLED (4)	驱动程序或者服务不加载到内存中，也不初始化
ErrorControl	SERVICE_ERROR_IGNORE (0)	驱动程序或者服务返回的任何错误都被忽略，没有警告被记录下来或者被显示出来
	SERVICE_ERROR_NORMAL (1)	如果驱动程序或者服务报告了一个错误，则显示一个警告
	SERVICE_ERROR_SEVERE (2)	如果驱动程序或者服务返回一个错误，并且最后已知的好控制集（last known good）尚未被使用，则重新引导进入最后已知的好控制集；否则，继续引导过程
	SERVICE_ERROR_CRITICAL (3)	如果驱动程序或者服务返回一个错误，并且最后已知的好控制集尚未被使用，则重新引导进入最后已知的好控制集；否则，停止引导过程，显示蓝屏崩溃
Type	SERVICE_KERNEL_DRIVER (1)	设备驱动程序
	SERVICE_FILE_SYSTEM_DRIVER (2)	内核模式的文件系统驱动程序
	SERVICE_ADAPTER(4)	已废弃
	SERVICE_RECOGNIZER_DRIVER (8)	文件系统识别器驱动程序
	SERVICE_WIN32_OWN_PROCESS (16)	该服务运行在一个只能容纳一个服务的进程中
	SERVICE_WIN32_SHARE_PROCESS (32)	该服务运行在一个可容纳多个服务的进程中
	SERVICE_INTERACTIVE_PROCESS (256)	允许该服务在控制台上显示窗口，并且接收用户的输入
Group	组的名称	当驱动程序或者服务的组被初始化时，它也初始化
Tag	标签号	指定了在一个组的初始化顺序中的位置。该参数不适用于服务
ImagePath	服务或驱动程序可执行文件的路径	如果没有指定ImagePath的话，I/O管理器在\Windows\System32\Drivers中寻找驱动程序，SCM使用Windows的函数，通过PATH环境变量搜索映像文件
DependOnGroup	组的名称	除非指定组中的一个驱动程序或者服务已加载，否则该驱动程序或者服务不会加载

www.TopSage.com

续表

值设置	值的名称	值设置的说明
DependOnService	服务名称	直到此指定的服务加载以后，该服务才会加载。该参数不适用于除了启动类型为SERVICE_AUTO_START之外的其他设备驱动程序
ObjectName	通常是LocalSystem，但也可以是一个账户名称，比如.\Administrator	指定了该服务将在哪个账户下运行。如果没有指定ObjectName的话，则使用LocalSystem账户。该参数不适用于设备驱动程序
DisplayName	服务的名称	此服务应用在显示服务时使用该名称。如果没有指定名称的话，该服务的注册表键的名称变成该服务的名称
Description	服务的说明	该服务的说明，最多可达32767个字节
FailureActions	当服务进程非正常退出时SCM应采取的动作的说明	失败动作包括:重新启动该服务进程、重新引导系统，以及运行一个指定的程序。此值不适用于驱动程序
FailureCommand	程序命令行	只有当上面的FailureActions（失败动作）指定了“当服务失败时应该执行一个程序”时，SCM才会读取这个值。此值不适用于驱动程序
Security	安全描述符	该值包含了一个安全描述符，此安全描述符定义了谁有什么样的权限来访问SCM内部创建的服务对象

请注意，Type值包含了三个可适用于设备驱动程序的值：设备驱动程序、文件系统驱动程序，以及文件系统识别器。这些值是Windows设备驱动程序使用的，这些设备驱动程序也将它们的参数作为注册表数据存储在Services键中。SCM负责启动那些Start值为SERVICE_AUTO_START或 SERVICE_DEMAND_START的驱动程序，所以，SCM数据库中包含了驱动程序的信息，这也就不足为奇了。Windows服务使用其他的类型：SERVICE_WIN32_OWN_PROCESS和SERVICE_WIN32_SHARE_PROCESS，这两者是互斥的。可容纳多个服务的可执行程序指定的是SERVICE_WIN32_ SHARE_PROCESS类型。让一个进程运行多个服务的好处是，可以节省下在单独进程中运行每一个服务所需要的系统资源。一个潜在的缺点是，如果在同一个进程中运行的一组服务中的任何一个发生了错误，并因此而终止了该进程，那么，该进程的所有服务也随之而终止。而且，另一个限制是，所有的服务必须运行在同一个账户下。

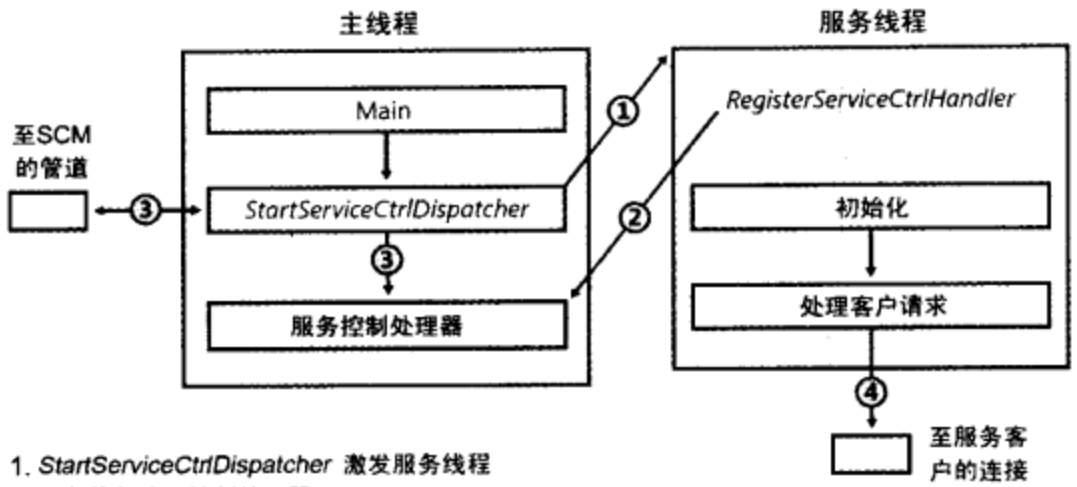
www.TopSage.com

当SCM启动一个服务进程时，该进程立即调用*StartServiceCtrlDispatcher*函数。*StartServiceCtrlDispatcher*接受一个入口点列表，每个入口点对应于该进程中的一个服务。每个入口点是由它所对应的服务的名称来标识的。*StartServiceCtrlDispatcher*创建了一个命名管道来跟SCM进行通信，在建立了该通信管道以后，它进入一个循环，等待SCM通过该管道发送过来的命令。每次SCM启动一个属于该进程的服务时，它发送一个“服务启动”命令。*StartServiceCtrlDispatcher*函数对于所接收到的每个启动命令，创建一个线程（称为服务线程），由该线程来调用所启动服务的入口点函数，并实现该服务的命令循环。*StartServiceCtrlDispatcher*一直在等待来自SCM的命令，只有当该进程的所有服务都停止时它才会将控制返回至该进程的main函数，以便服务进程在退出以前做一些资源清理工作。

每个服务的入口点的第一个动作是调用*RegisterServiceCtrlHandler*函数。该函数接收一个指向某个函数的指针，并且将该指针保存起来，此函数指针称为**控制处理器**（*control handler*）。该服务实现此函数以便处理各种来自SCM的命令。*RegisterServiceCtrlHandler*并不与SCM通信，但它为*StartServiceCtrlDispatcher*函数将此函数存储在本地进程内存中。服务入口点继续初始化该服务，包括分配内存、创建通信端点，以及从注册表中读入私有的配置数据。大多数服务都遵从的一个惯例是，将服务的参数保存在其服务注册表键下的名为Parameters的子键下面。在入口点初始化该服务的过程中，它可能会利用*SetServiceStatus*函数来定期地给SCM发送状态消息，以便告诉SCM该服务的启动过程正在如何进行。当入口点完成了初始化以后，服务线程通常进入一个循环，等待来自客户应用的请求。例如，Web服务器会初始化一个TCP监听套接字，等待进来的HTTP连接请求。

一个服务进程的主线程，即在*StartServiceCtrlDispatcher*函数中执行的线程，接收下针对该进程中各个服务的SCM命令，再调用目标服务的控制处理器函数（由*RegisterServiceCtrlHandler*保存在该进程中）。SCM命令包括停止、暂停、恢复、询问和停机，或者应用程序定义的命令。图4.9显示了一个服务进程的内部组织结构。图中的进程宿纳了一个服务，它包含两个线程：主线程和服务线程。

www.TopSage.com



1. *StartServiceCtrlDispatcher* 激发服务线程
2. 服务线程注册控制处理器
3. *StartServiceCtrlDispatcher* 在处理SCM命令时调用服务处理器
4. 服务线程处理客户请求。

图4.9 一个服务进程的内部

SrvAny工具

如果你想要让一个程序作为一个服务来运行，你需要修改它的启动代码，以便符合本节中介绍的针对服务的要求。如果你没有源代码，你可以使用Windows资源工具箱中的SrvAny工具。SrvAny使得你可以将任何一个应用程序当做一个服务来运行。它从该服务的注册表键的Parameters子键中，读入必须加载的服务文件的路径。当SrvAny启动时，它通知SCM，它正在宿纳一个特殊的服务；当它接收到一个启动命令时，它激发起该服务的可执行文件，并使之作为一个子进程。此子进程接收到SrvAny进程的访问令牌的一份拷贝，以及指向同一窗口站的一个引用，所以，此可执行文件运行在同样的安全账户下，并且有同样的交互性设置，就像你在配置SrvAny进程时指定的那样。SrvAny服务没有共享进程的Type值，所以，每一个你安装成通过SrvAny来当做服务运行的应用程序都运行在单独的进程中，它们分别是SrvAny宿主程序的不同实例。

服务账户

对于服务开发人员和管理员来说，一个服务的安全环境是一个非常重要的考虑，因为它规定了该服务进程可以访问哪些资源。除非一个服务安装程序或者管理员指定了特殊的设置，否则，绝大多数服务运行在本地系统账户（*local system account*）的安全环境下（该账户有时候被显示成SYSTEM，而其他时候则被显示成LocalSystem）。Windows XP引入了本地系统账户的两个变种：**网络服务**（*network service*）和**本地服务**（*local service*）账户。从安全角度而言，这两个新的账户比本地系统账户具有的能力要少一些；那些不需要本地系统账户能力的Windows

www.TopSage.com

内置服务都依据它们的特点，运行在适当的服务账户之下。下面的小节介绍了这些账户的一些特殊之处。

本地系统账户

本地系统账户也正是核心的Windows用户模式操作系统组件运行时所在的账户，这样的组件包括会话管理器（\Windows\System32\Smss.exe）、Windows子系统进程（Csrss.exe）、本地安全权威子系统（\Windows\System32\Lsass.exe）和WinLogon进程（\Windows\System32\Winlogon.exe）。

从安全的角度来看，本地系统账户有非常强大的能力——当涉及本地系统上的安全能力时，它比任何本地的或者域的账户要强大得多。该账户有以下一些特征。

- 它是本地管理员组中的一个成员。表4.8显示了本地系统账户所属的那些组（有关在对象访问检查时如何利用组成员关系的更多信息，请参见第8章）。
- 它有权利赋予几乎每一种特权（甚至包括那些通常并不赋予本地管理员账户的特权，比如创建安全令牌的特权）。表4.9列出了分配给本地系统账户的特权（第8章讲述了每一种特权的用途）。
- 绝大多数文件和注册表键都赋予本地系统账户完全的访问权限（即使它们没有赋予全部的访问权限，在本地系统账户下运行的进程也可以利用“接管所有权（take-ownership）”的特权来获得访问权限）。
- 在本地系统账户下运行的进程按照默认的用户轮廓（HKU\.DEFAULT）来运行。因此，它们不可能访问存储在其他账户的用户轮廓中的配置信息。
- 当一个系统是Windows域中的一个成员时，本地系统账户包含了一个服务进程运行时所在计算机的机器安全标识符（SID）。因此，一个运行在本地系统账户中的服务通过利用它的计算机账户，就可以自动地在同一个域林中的其他机器上得到身份认证（一个**林**[*forest*]是指一组域）。
- 除非该机器账户被特别赋予了对某些资源（比如网络共享体、命名管道，等等）的访问权限，否则，一个进程只能访问那些允许空会话（也即，不需要安全凭证的连接）访问的网络资源。你可以在特定的允许空会话的计算机上，通过HKLM\SYSTEM\CurrentControlSet\Services\lanmanserver\parameters下的NullSessionPipes和NullSessionShares注册表值来指定这些共享体和管道。

www.TopSage.com

表4.8 服务账户的组成员关系

本地系统（Local System）	网络服务（Network Service）	本地服务（Local Service）
Everyone	Everyone	Everyone
Authenticated Users	Authenticated Users	Authenticated Users
Administrators	Users	Users
	Local	Local
	Network Service	Local Service
	Service	Service

表4.9 服务账户的特权

本地系统（Local System）	网络服务（Network Service）	本地服务（Local Service）
SeAssignPrimaryToken Privilege	SeAssignPrimaryToken Privilege	SeAssignPrimaryTokenPrivilege
SeAuditPrivilege	SeAuditPrivilege	SeAuditPrivilege
SeBackupPrivilege	SeChangeNotifyPrivilege	SeChangeNotifyPrivilege
SeChangeNotifyPrivilege	SeIncreaseQuotaPrivilege	SeIncreaseQuotaPrivilege
SeCreateGlobalPrivilege	赋予Everyone、Authenticated Users和Users组的特权	赋予Everyone、Authenticated Users和Users组的特权
SeCreatePagefilePrivilege		
SeCreatePermanentPrivilege		
SeCreateTokenPrivilege*		
SeDebugPrivilege		
SeImpersonatePrivilege		
SeIncreaseBasePriorityPrivilege		
SeIncreaseQuotaPrivilege		
SeLoadDriverPrivilege		
SeLockMemoryPrivilege		
SeManageVolumePrivilege		
SeProfileSingleProcessPrivilege		
SeRestorePrivilege		
SeSecurityPrivilege		
SeShutdownPrivilege		
SeSystemEnvironmentPrivilege		
SeSystemTimePrivilege		
SeTakeOwnershipPrivilege		
SeTcbPrivilege		
SeUndockPrivilege		

* 在 Windows Server 2003 上，本地系统账户不包含此特权。

www.TopSage.com

网络服务账户

网络服务账户的用途是，供那些“既希望利用计算机账户来向网络上其他的机器认证身份（就好像本地系统账户所具有的认证能力那样），但是又不需要管理员组的成员所属权，或者也不需要用到那些分配给本地系统账户的诸多特权”的服务来使用的。因为网络服务账户并不属于管理员组，所以，运行在网络服务账户中的服务在默认情况下，比起运行在本地系统账户下的服务，只能访问很少量的注册表键，以及文件系统中的文件夹和文件。而且，只被赋予少量的特权，也限制了一个被攻破的网络服务进程的能力范围。例如，一个运行在网络服务账户下的进程不可能加载一个设备驱动程序或者打开任意的进程。

网络服务账户和本地系统账户之间的另一个差别是，运行在网络服务账户中的进程使用了网络服务账户的轮廓。网络服务轮廓的注册表部分被加载在HKU\S-1-5-20下面，而构成此注册表部分的文件和目录则位于\Documents and Settings\NetworkService中。

在Windows XP和Windows Server 2003中，一个运行在网络服务账户下的服务是DNS客户，它负责解析DNS名称，也负责找到域控制器。

本地服务账户

本地服务账户几乎等同于网络服务账户，两者重要的区别在于，前者只能访问那些允许匿名访问的网络资源。表4.9显示了网络服务账户与本地服务账户具有同样的特权；表4.8显示了两者属于同样的组，惟一的例外是网络服务账户属于Network Service组，而不是Local Service组。运行在本地服务账户中的进程所使用的轮廓被加载到HKU\S-1-5-19中，以及存储在\Documents and Settings\LocalService中。

在Windows XP和Windows Server 2003中，运行在本地服务账户中的服务例子包括：允许远程访问本地系统注册表的远程注册表服务（Remote Registry Service），负责接收网络广播的管理报警消息的Alerter服务，以及执行NetBIOS名称解析任务的LmHosts服务。

在哪个账户中运行服务

由于刚才提到的这些限制，有些服务需要使用一个用户账户的安全凭证来运行。你可以在创建一个服务的时候，对该服务进行配置，让它在选定的账户下运行；你也可以利用Windows Services MMC加载件（snap-in）来指定一个账户和口令，让该服务运行在此账户下。在Services加载件中，在一个服务上右键点击，再选择Properties，并点击Log On标签，然后选择This Account选项，如图4.10所示。

www.TopSage.com

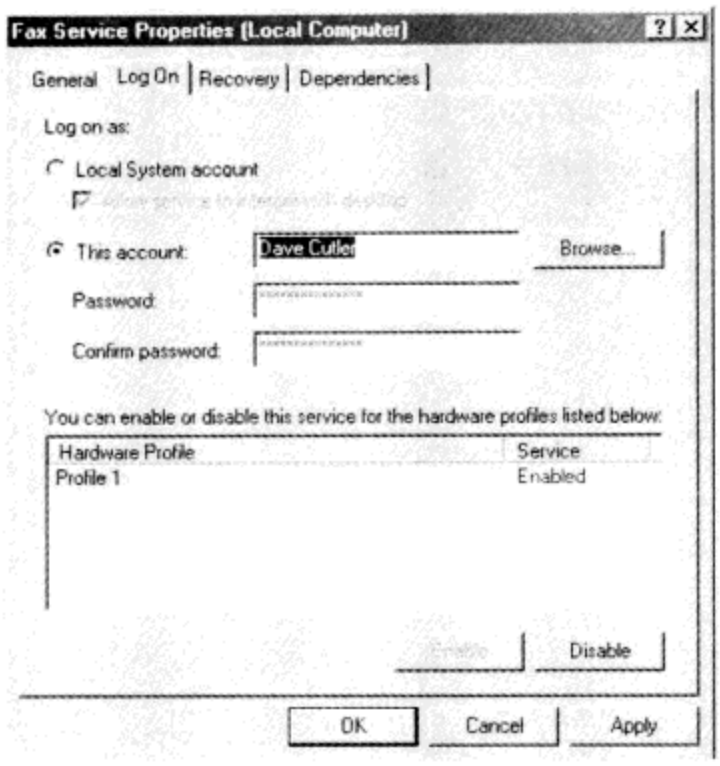


图4.10 服务账户的设置

交互式服务

对于在本地系统账户、本地服务账户和网络服务账户下运行的服务，另一个限制是，它们不能在交互式用户的桌面上显示对话框或者窗口。这一限制并非是由于它们运行在这些账户下所导致的直接结果，而是由于Windows子系统将服务进程分配给窗口站的做法所导致的结果。

Windows子系统将每个Windows进程与一个**窗口站**（*window station*）关联起来。窗口站包含了桌面，桌面包含了窗口。在一个控制台上只有一个窗口站是可见的，它可以接收用户的鼠标和键盘的输入。在终端服务环境中，每个会话有一个窗口站是可见的，但是所有的服务都作为控制台会话的一部分来运行。Windows将可见的窗口站命名为WinSta0，所有的交互式进程都使用WinSta0。

除非另行指定，否则，Windows子系统都将运行在本地系统账户下的服务跟一个名为Service-0x0-3e7$的非可见窗口站关联起来，所有的非交互式服务都共享此窗口站。此名称中的数字*3e7*，代表了Lsass分配给登录会话的登录会话标识符；对于运行在本地系统账户下的非交互式服务，SCM都使用此登录会话。

凡是配置了要运行在某个用户账户（也就是说，不是本地系统账户）下的服务，都运行在一个不同的非可见窗口站中，这些窗口站是用LSASS分配给该服务的登录会话的标识符来命名的。图4.11显示了Winobj工具（可以从*www.sysinternals.com*下载）的一个输出示例，它显示的是Windows专门放置窗口站对象的对象管理器目录。从其中可以看到交互式窗口站

www.TopSage.com

（WinSta0）、非交互式系统服务窗口站（Service-0x0-3e7$），以及分配给一个以某个用户身份而登录的服务进程的非交互式窗口站（Service-0x0-6368f$）。

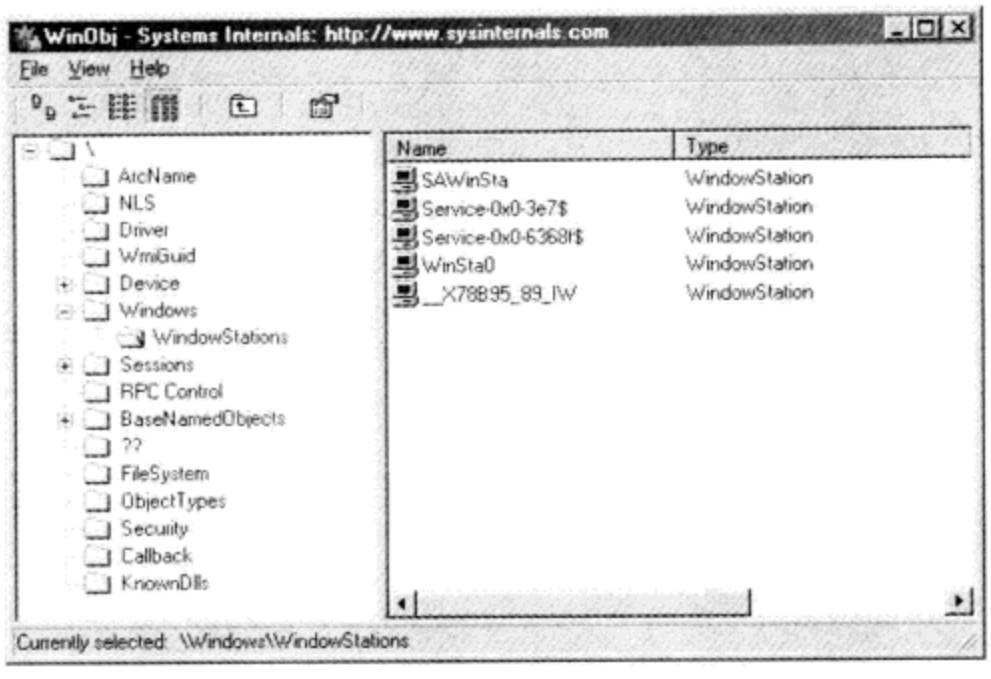


图4.11 窗口站列表

不管一个服务是运行在用户账户下，或是运行在本地系统账户下，或是本地服务账户下，还是运行在网络服务账户下，只要它没有运行在可见窗口站中，它就不能接收来自用户的输入，也不能在控制台上显示窗口。实际上，如果一个服务要在该窗口站上弹出一个普通的对话框，那么，该服务就好像挂起了一样，因为没有用户能看到该对话框，当然，用户也就无法提供键盘或者鼠标的输入来解除对话框，从而无法让服务继续执行下去（一个例外是，若*MessageBox*调用中设置了特殊的MB_SERVICE_NOTIFICATION或MB_DEFAULT_DESKTOP_ ONLY标志。如果指定了MB_SERVICE_NOTIFICATION标志的话，那么，即使该服务没有被配置成允许与用户进行交互，消息框也将总是显示在交互式窗口站中；如果指定了MB_DEFAULT_DESKTOP_ONLY标志，则消息框将显示在交互式窗口站的默认桌面上）。

在很少的情况下，一个服务可以有充分的理由通过对话框或窗口来与用户进行交互。为了将一个服务配置成有权利与用户进行交互，SERVICE_ INTERACTIVE_PROCESS修饰符必须出现在该服务的注册表键的Type参数中（注意，凡是被配置成在一个用户账户下运行的服务，不能被标记为交互式的）。当SCM启动一个被标记为交互式的服务时，它在本地系统账户的安全环境中激发起该服务的进程，但是将该服务与WinSta0连接起来，而不是与非交互式的服务窗口站连接在一起。这样连接到WinSta0，就使得该服务可以在控制台上显示对话框和窗口，从而允许这些窗口可以响应用户的输入。

www.TopSage.com



> 注 Microsoft不鼓励运行交互式的服务，特别是在本地系统账户下，因为它会导致一些内在的安全缺陷。 交互式服务所显示的窗口，很容易接收到来自恶意进程的窗口消息。这些恶意进程可能运行在非特权用户的桌面上，它们利用这些消息来引发服务进程中的缓冲区溢出，破坏服务进程，进而提升自己的安全特权。

服务控制管理器

SCM的可执行文件是\Windows\System32\Services.exe，如同大多数服务进程一样，它也是作为一个Windows控制台程序来运行的。Winlogon进程在系统引导的早期将SCM启动起来（有关引导过程的细节，请参见第5章）。 SCM的启动函数，即*SvcCtrlMain*，会有条不紊地将那些被配置成自动启动的服务激发起来。*SvcCtrlMain*在屏幕被切换成空白桌面之后不久即被执行，但通常在Winlogon加载图形化的识别和认证界面（GINA，出示一个登录对话框）之前执行。

*SvcCtrlMain*首先创建一个名为SvcCtrlEvent_ A3752DX的同步事件，该事件被初始化成无信号状态。只有当SCM完成了必要的步骤来准备接收SCP的命令以后，它才将该事件设置成有信号状态。SCP用来与SCM建立对话的函数是*OpenSCManager*。*OpenSCManager*等待SvcCtrlEvent_A3752DX事件变成有信号状态，因而，SCP在SCM完成初始化以前，不必总是试图与SCM联络。

接下来，*SvcCtrlMain*开始做它自己的事情，并调用*ScCreateServiceDB*，该函数负责建立起SCM的内部服务数据库。*ScCreateServiceDB*读取并存储HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List的内容，这是一个REG_MULTI_SZ值，它列出了定义好的服务组的名称和顺序。如果一个服务或者设备驱动程序需要控制其相对于其他组中的服务的启动顺序，那么，该服务的注册表键包含一个可选的Group值。例如，Windows网络栈是由底向上建立起来的，所以，涉及网络的Windows服务必须指定Group值，以便在启动序列中将它们放在网络设备驱动程序的后面。SCM内部创建了一个组列表，此列表保留了SCM从注册表中读入的组顺序。这些组包括（但不限于）NDIS、TDI、Primary Disk、Keyboard Port和Keyboard Class。加载件和第三方应用程序甚至可以定义它们自己的组，并且将这些组加入到列表中。例如，Microsoft Transaction Server添加了一个名为MS Transactions的组。

然后，*ScCreateServiceDB*扫描HKLM\SYSTEM\CurrentControlSet\Services中的内容，并且在服务数据库中为它所碰到的每一个键创建一个条目。数据库条目包含了所有为一个服务而定义的、与服务有关的参数，以及一些记录了该服务状态的域。SCM既为设备驱动程序添加条目，也为服务添加条目，因为SCM将所有标记为自动启动的服务和驱动程序都启动起来，并且对于那些标记为引导启动和系统启动的驱动程序，检测其启动失败的情况。它还提供了一种方法，让应用程序可以查询驱动程序的状态。在任何用户模式进程执行起来以前，I/O管理器

www.TopSage.com

将那些标记为引导启动和系统启动的驱动程序加载到系统中，因此，凡是这些启动类型的驱动程序，都在SCM启动之前被加载到系统中。

*ScCreateServiceDB*读取一个服务的Group值，以确定它是否是一个组的成员，并且将这个值与先前创建的组列表中该组的条目关联起来。此函数也通过查询该服务的DependOnGroup和DependOnService注册表值，读取到该服务的组相依性和服务相依性，并将它们记录到数据库中。图4.12显示了SCM如何组织服务条目和组顺序列表。注意，服务列表是按字母顺序来排列的。此列表之所以按字母序来排列，是因为，SCM根据Services注册表键来创建此列表，而Windows按字母顺序来存储注册表键。

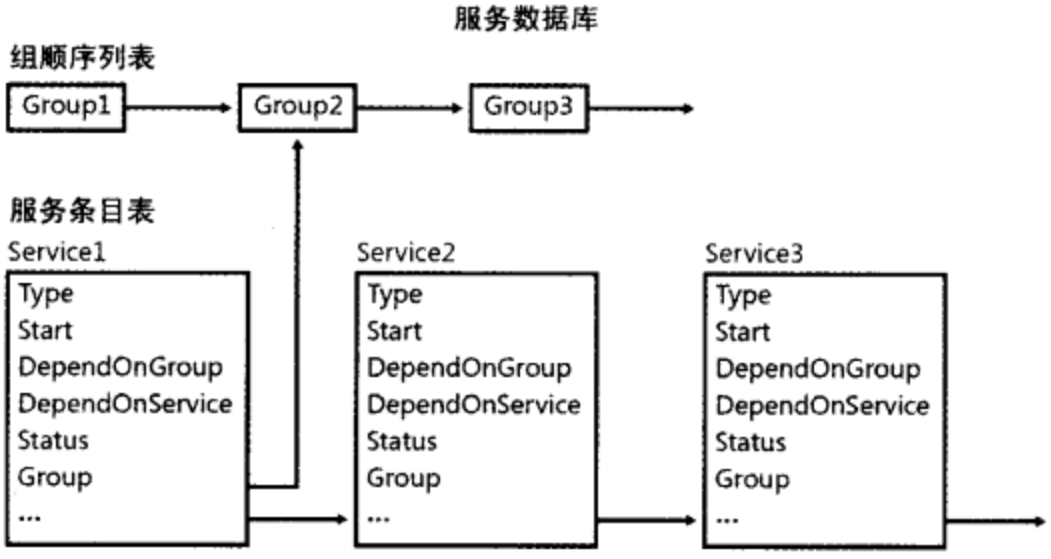


图4.12 服务数据库的组织结构

在服务启动过程中，SCM可能需要调用LSASS（比如，在一个用户账户中登录一个服务），所以SCM等待LSASS发出LSA_RPC_SERVER_ACTIVE同步事件，当LSASS完成了初始化时它就会发出此信号。Winlogon也启动LSASS进程，所以，LSASS的初始化与SCM的初始化是同时进行的，LSASS和SCM谁先完成初始化的顺序是不确定的。然后，*SvcCtrlMain*调用*ScGetBootAndSystemDriverState*来扫描服务数据库，以寻找那些自动启动的和系统启动的设备驱动程序条目。*ScGetBootAndSystemDriverState*确定一个驱动程序是否已成功启动，它的做法是，在名为\Driver的对象管理器名字空间目录中查找该驱动程序的名称。当一个设备驱动程序被成功地加载时，I/O管理器将该驱动程序的对象插入到名字空间中该目录下面，所以，如果它的名称没有出现的话，则它还没有被加载进来。在图4.13中，Winobj显示了Driver目录的内容。如果一个驱动程序没有被加载，则SCM在*PnP_DeviceList*函数返回的驱动程序列表中寻找它的名称。*PnP_DeviceList*提供了所有包含在系统当前硬件轮廓中的驱动程序。*SvcCtrlMain*将所有尚未启动的、包含在当前轮廓中的驱动程序的名称记录在一个名为ScFailedDrivers的列表中。

在启动那些“自动启动”的服务以前，SCM还要执行其他的一些步骤。它创建一个名为\Pipe\Ntsvcs的远过程调用（RPC）命名管道，然后，RPC激发一个线程，在该管道上监听来自

www.TopSage.com

SCP的消息。然后，SCM将它的初始化完成事件SvcCtrlEvent_A3752DX设置成有信号状态。注册一个控制台应用程序停机事件处理器，以及通过*RegisterServiceProcess*向Windows子系统进程进行注册，这些都为SCM做好了系统停机的准备。

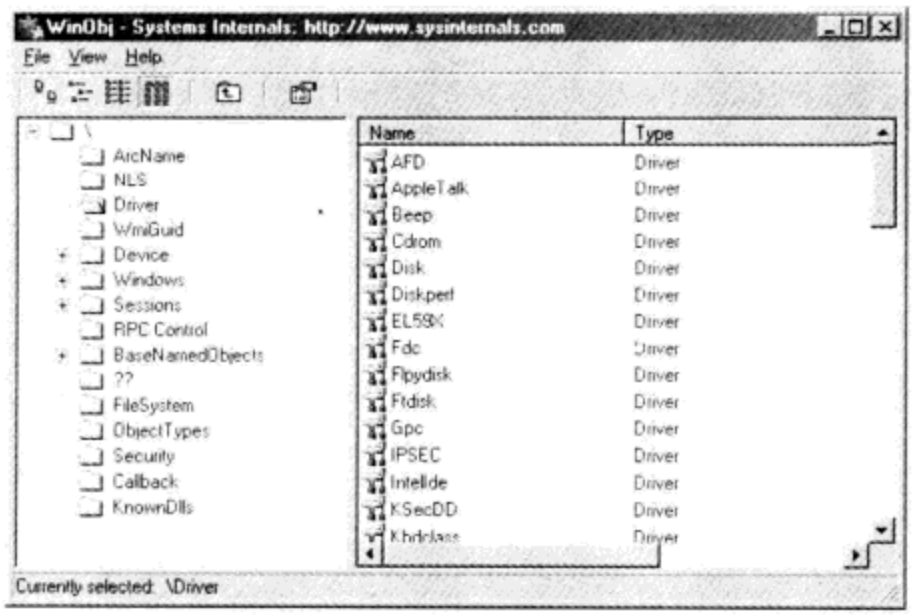


图4.13 驱动程序对象列表

网络驱动器字母

SCM除了作为服务接口的角色以外，它还有另一项完全不相关的责任：无论何时当系统创建或者删除一个网络驱动器字母连接时，它都要通知系统中的GUI应用程序。SCM等待MPR（Multiple Provider Router）用信号通知一个命名的事件\BaseNamedObjects\ScNetDrvMsg，当应用程序将一个驱动器字母分配给一个远程网络共享文件夹，或者删除了一个远程共享文件夹的驱动器字母分配时，MPR就会用信号通知此事件（有关MPR的更多信息，请参见第13章）。当MPR用信号通知该事件时，SCM调用Windows函数*GetDriveType*来获得已连接的网络驱动器字母的列表。如果在事件信号之间该列表改变了的话，则SCM发送一个类型为WM_DEVICECHANGE的Windows广播消息。SCM使用DBT_DEVICEREMOVECOMPLETE或DBT_DEVICEARRIVAL作为该消息的子类型。此消息主要是给Windows Explorer使用的，所以，它可以更新任何已打开的My Computer窗口，显示出网络驱动器字母的存在与否。

服务启动

SvcCtrlMain 调用SCM函数*ScAutoStartServices*来启动所有已被指定为“自动启动”（在Start值中）的服务。*ScAutoStartServices*也会将“自动启动”的设备驱动程序启动起来。为了避免混淆，你应该将术语“服务”看成是“服务和驱动程序”，除非另行指定。*ScAutoStartServices*中

www.TopSage.com

按照正确顺序来启动服务的算法分阶段进行处理，每个阶段对应于一个组，这些阶段按照HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List注册表值中存储的组顺序所定义的序列进行处理。如图4.14所示，List值包含了组的名称，其顺序正是SCM启动这些组的顺序。因此，将一个服务分配给一个组，这对于属于其他组的其他服务，在启动顺序上没有任何影响。

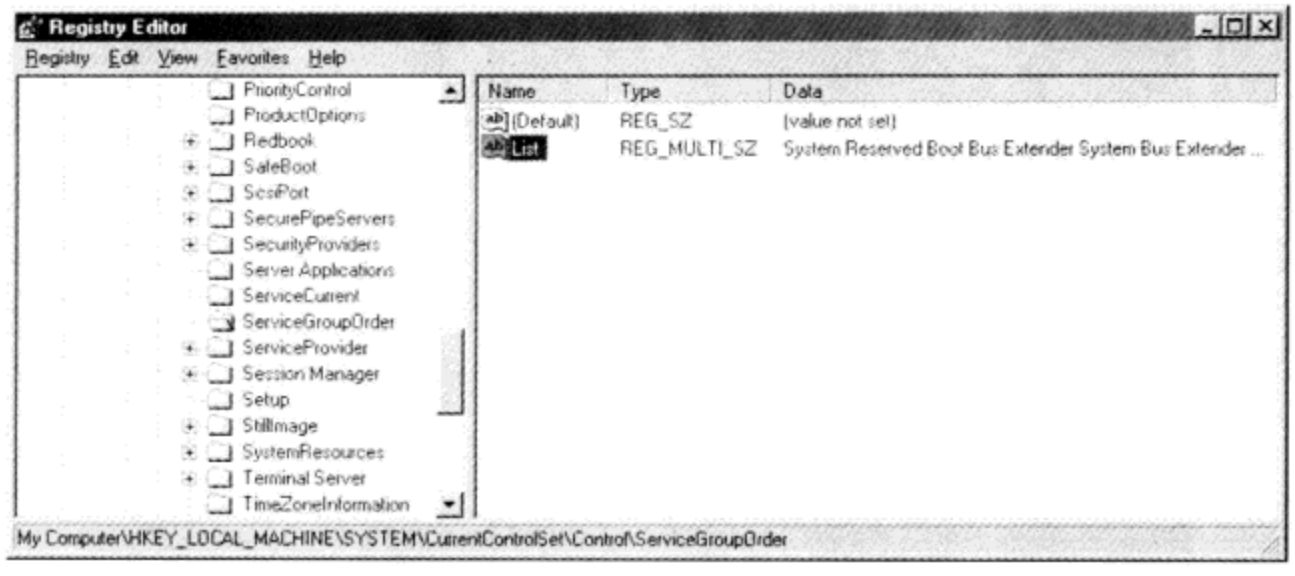


图4.14 ServiceGroupOrder注册表键

当一个阶段开始时，*ScAutoStartServices*将所有属于此阶段的组的那些服务条目标记出来，以便启动它们。然后，*ScAutoStartServices*循环检查这些标记出来的服务，看它是否能将每一个服务启动起来。作为检查过程的一部分，它根据这个服务的注册表键中的DependOnGroup值是否存在，来确定这个服务是否依赖于其他的组。如果存在这种相依性，则它所依赖的那个组必须已经初始化，该组中至少有一个服务已经成功启动起来。如果在组启动序列中，该服务所依赖的那个组，比该服务所属的组还要迟，那么，SCM记录下该服务有一个“循环相依性”错误。如果*ScAutoStartServices*正在检查的是一个Windows服务，或者一个“自动启动”的设备驱动程序，那么，它接下来检查该服务是否依赖于其他一个或者多个服务，如果是的话，它查看这些服务是否已经启动了。服务相依性是由一个服务的注册表键中的DependOnService值来指定的。如果一个服务依赖于其他一些服务，并且这些服务所属的组位于ServiceGroupOrder\List的后面，那么，SCM也会产生一个“循环相依性”错误，并且不会启动该服务。如果该服务依赖于同一个组中的任何其他尚未启动的服务，则该服务将被跳过去。

当一个服务的相依性已经被验证通过以后，*ScAutoStartServices*在启动该服务之前还要做最后的检查，看该服务是否是当前引导配置的一部分。当系统以安全模式来引导时，SCM要确保：在适当的安全引导注册表键中，该服务或者是按名称来标识的，或者是按组来标识的。注册表中有两个安全引导键：Minimal和Network，它们位于HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot的下面；SCM到底检查哪一个，取决于用户引导的是哪一种安全模式。如果用户在专门的引导菜单（在引导过程中，当出现提示时，你按下F8就可以看到该菜单）中选择

www.TopSage.com

的是Safe Mode或者Safe Mode With Command Prompt，则SCM使用的是Minimal键；如果用户选择的是Safe Mode With Networking，则SCM使用的是Network键。若SafeBoot键下存在一个名为Option的字符串值，则不仅表明了当前系统是按安全模式引导起来的，而且也指出了用户选择的安全模式类型。有关安全引导的更多信息，请参见第5章中的“安全模式”一节。

一旦SCM决定要启动一个服务，它就调用*ScStartService*，该函数对于服务和设备驱动程序分别采取不同的步骤。当*ScStartService*启动一个Windows服务时，它首先读取该服务的注册表键中的ImagePath值，以确定该服务进程的映像文件名。然后，它检查该服务的Type值，如果此值是SERVICE_WINDOWS_SHARE_PROCESS （0x20），那么，SCM保证：该服务运行所在的进程如果已经启动了的话，其登录的账户一定与该服务的指定启动账户相同。一个服务的ObjectName注册表值存储了该服务应该在哪个账户下运行。如果一个服务没有ObjectName值，或者它的ObjectName值为LocalSystem，则该服务运行在本地系统账户中。

SCM在一个称为**映像数据库**（*image database*）的内部数据库中，检查是否有针对该服务的ImagePath值的条目，以便验证该服务的进程尚未在其他的账户下被启动起来。如果在映像数据库中没有找到此ImagePath值的条目，则SCM创建一个这样的条目。当SCM创建一个新的条目时，它还将该服务的登录账户名，以及该服务的ImagePath值中的数据也存储起来。SCM要求Windows服务有一个ImagePath值。如果一个服务没有ImagePath值，则SCM报告一个错误，表示它不能找到该服务的路径，从而无法启动该服务。如果SCM在映像数据库中找到了一个与该服务的ImagePath数据相匹配的条目，那么，它要保证：它当前正在启动的服务的用户账户信息与数据库条目中存储的信息是相同的——一个进程只能以一个账户的身份来登录，所以，当一个服务指定的账户名与同一个进程中已经启动起来的其他服务的账户名不相同时，SCM会报告一个错误。

SCM调用*ScLogonAndStartImage*来登录一个服务（如果该服务的配置中指定了的话），并启动该服务的进程。SCM通过调用LSASS函数*LsaLogonUser*来登录那些并非运行在系统账户下的服务。*LsaLogonUser*在通常情况下要求一个口令，但是SCM告诉LSASS，该口令保存在注册表中HKLM\SECURITY\Policy\Secrets的下面，是该服务的LSASS“秘密”（记住，涉及安全的内容通常是不可见的，因为它的默认安全设置只允许系统账户访问）。当SCM调用*LsaLogonUser*时，它指定了登录类型参数为服务登录，所以，LSASS在Secrets键的名为*_SC_*<服务名>的子键下查找口令。

当SCP配置一个服务的登录信息时，SCM利用*LsaStorePrivateData*来指示LSASS将一个登录口令保存到Secrets键下。在登录成功以后，*LsaLogonUser*给调用者返回一个句柄，指向一个访问令牌。Windows使用访问令牌来代表一个用户的安全环境，以后，SCM将该访问令牌与实现此服务的进程关联起来。

www.TopSage.com

在成功登录以后，如果此账户的轮廓信息还没有被加载的话，SCM通过调用UserEnv DLL（\Windows\System32\Userenv.dll）的*LoadUserProfile*函数将此账户的轮廓信息加载进来。HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<用户轮廓键>\ProfileImagePath值包含了*LoadUserProfile*需要加载的注册表储巢在磁盘上的位置，从而使得该储巢中的信息成为该服务的HKEY_CURRENT_USER键。

交互式的服务必须打开WinSta0窗口站，但是，*ScLogonAndStartImage*在允许一个交互式服务访问WinSta0之前，它要检查HKLM\SYSTEM\CurrentControlSet\Control\Windows\NoInteractiveServices值是否已被设置。管理员设置此值以后，可以防止那些被标记为交互式的服务在控制台上显示窗口。在无人值守的服务器环境中，此选项是非常合适的，因为在这种环境中，交互式服务弹出的窗口是没有人来响应的。

下一个步骤是，如果该服务的进程尚未被启动（例如，为了另一个服务），则*ScLogonAndStartImage*为该服务激发一个进程。SCM通过Windows函数*CreateProcessAsUser*来启动此进程，并且将该进程的状态设置为挂起状态。接下来，SCM创建一个命名管道，以后它通过该管道与服务进程进行通信，它分配给管道的名称为\Pipe\Net\NtControlPipeX，这里X是一个数字，每次SCM创建一个管道，该数字就会递增。然后，SCM通过*ResumeThread*函数来恢复服务进程的执行，并且等待该服务连接到它的SCM管道上。如果注册表值HKLM\SYSTEM\CurrentControlSet\Control\ServicesPipeTimeout存在的话，则它决定了SCM等待一个服务调用*StartServiceCtrlDispatcher*并连接过来的时间长度，如果在这么长时间里没有等到，则SCM就会放弃，终止该进程，并得出结论：该服务未能启动。如果ServicesPipeTimeout不存在，则SCM使用默认的30秒作为超时间隔值。SCM对于它所有的服务通信都使用同样的超时间隔值。

当一个服务通过它的管道连接到SCM时，SCM向该服务发送一个启动命令。如果该服务未能在超时间隔内肯定地响应此启动命令，SCM就会放弃，并且继续启动下一个服务。当一个服务没有对启动请求作出响应时，SCM并不会像一个服务在超时间隔内没有调用*StartServiceCtrlDispatcher*的情形那样终止该进程，相反，它会在系统的事件日志（Event Log）中记录一个错误，指明该服务未能及时地启动起来。

如果SCM调用*ScStartService*启动的服务有一个Type注册表值为SERVICE_KERNEL_DRIVER或SERVICE_FILE_SYSTEM_ DRIVER，那么，该服务确实是一个设备驱动程序，所以，*ScStartService*调用*ScLoadDeviceDriver*来加载该驱动程序。*ScLoadDeviceDriver*首先使SCM进程具有加载驱动程序的安全特权，然后调用内核服务*NtLoadDriver*，将该驱动程序的注册表键中的ImagePath值的数据传递过去。与Windows服务不同的是，驱动程序不需要指定ImagePath值；如果该值不存在的话，SCM通过将驱动程序的名称附加在字符串\Windows\System32\Drivers\的后面就可以构造一个映像文件路径。

www.TopSage.com

*ScAutoStartServices*继续循环处理同属于一个组的服务，直到所有这些服务要么被启动起来，要么产生相依性错误。这种循环处理方式是SCM根据一个组中的服务的DependOnService相依性来自动对它们进行顺序处理的。SCM在较早的循环中启动那些被其他服务依赖的服务，跳过那些依赖于其他服务的服务，而在后续的循环中再启动这些服务。注意：SCM忽略了Windows服务的Tag值，你可能会在HKLM\SYSTEM\CurrentControlSet\Services键的子键下面看到这些值；I/O管理器利用Tag值来排列一个组内的引导启动（boot-start）和系统启动（system-start）的设备驱动程序的启动顺序。

一旦SCM完成了ServiceGroupOrder\List值中列出的所有组的启动阶段，它再为那些属于其他组（未列在该值中）的服务执行一个单独的阶段，最后，再为那些不属于任何一个组的服务执行一个阶段。当SCM完成了启动所有这些自动启动的服务和驱动程序时，它发信号通知\BaseNamedObjects\SC_AutoStartComplete事件。

启动错误

如果一个驱动程序或者服务报告一个错误，作为对SCM启动命令的响应，那么该服务的注册表键的ErrorControl值决定了SCM该如何作出反应。如果ErrorControl值是SERVICE_ERROR_IGNORE (0)，或者ErrorControl值没有被指定，那么，SCM只需简单地忽略该错误，继续处理它的服务启动工作即可。如果ErrorControl值是SERVICE_ERROR_NORMAL (1)，那么，SCM向系统事件日志中写入一个事件，它这样说：“由于以下错误，故<*服务名*>服务未能启动（The <service name> service failed to start due to the following error）：”。该服务向SCM返回一个Windows错误代码作为启动失败的原因；SCM在事件日志纪录中包含了此Windows错误代码的文本表示。图4.15显示了这样的事件日志纪录，它报告了一个服务启动错误。

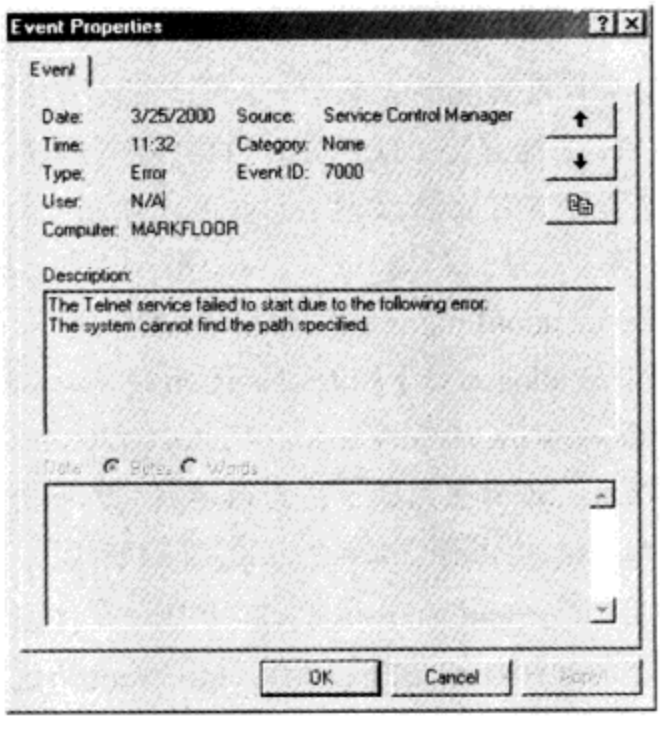


图4.15 服务启动失败的事件日志纪录

www.TopSage.com

如果一个服务报告的ErrorControl值是SERVICE_ERROR_SEVERE (2)或SERVICE_ERROR_CRITICAL (3)，那么，SCM会在事件日志中加入一条纪录，然后调用内部函数*ScRevertToLastKnownGood*。此函数将系统的注册表配置切换到一个名为“最后已知的好控制集（last known good）”的版本，此版本是该系统最后一次成功引导的注册表配置。然后，它利用执行体中实现的*NtShutdownSystem*系统服务来重新启动该系统。如果该系统当前已经是在利用“最后已知的好控制集”来引导了，那么，该系统仅仅重新启动而已。

接受当前引导和“最后已知的好控制集”

除了启动服务以外，系统也要让SCM来决定：应该何时将系统的注册表配置HKLM\SYSTEM\CurrentControlSet保存为“最后已知的好控制集”。CurrentControlSet包含了Services键，作为它的一个子键，所以，CurrentControlSet包含了SCM数据库的注册表表示形式。它也包含了Control键，其中保存了许多内核模式和用户模式的子系统配置信息。在默认情况下，一次成功的引导是由“所有自动启动服务的成功启动”和“一次成功的用户登录”构成的。如果由于一个设备驱动程序在引导过程中使系统崩溃了，或者由于一个自动启动的服务报告了一个启动错误（其ErrorControl值为SERVICE_ERROR_SEVERE或SERVICE_ERROR_CRITICAL），因而导致系统停止，则引导就失败了。

很显然，SCM知道什么时候它已经完成了所有自动启动服务的成功启动，但是Winlogon（\Windows\System32\Winlogon.exe）必须通知它什么时候有了一次成功的登录。当一个用户登录的时候，Winlogon调用*NotifyBootConfigStatus*函数，而*NotifyBootConfigStatus*则会向SCM发送一个消息。在所有自动启动的服务都成功启动以后，或者SCM接收到了*NotifyBootConfigStatus*发送过来的消息以后（看哪个来得更晚一些），SCM调用系统函数*NtInitializeRegistry*，将当前注册表的启动配置保存下来。

第三方软件开发人员可以用他们自己的定义来代替Winlogon对于一次成功登录的定义。例如，运行Microsoft SQL Server的系统要等到SQL Server能够接受或者处理数据库事务之后才能认为引导成功了。开发人员只要编写一个引导确认程序并且安装该程序，就可以强加上他们自己的对于一次成功引导的定义。安装方法很简单，只要让注册表键HKLM\SYSTEM\CurrentControlSet\Control\BootVerificationProgram中存储的值指向它在磁盘上的位置即可。而且，安装的引导确认程序必须禁止Winlogon对于*NotifyBootConfigStatus*的调用，其做法是，将HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk设置为0。在一个引导确认程序被安装到系统中以后，SCM在完成了所有自动启动的服务以后会激发该程序，然后等待它调用*NotifyBootConfigStatus*，再保存“最后已知的好控制集”。

Windows维护了CurrentControlSet的几份拷贝，CurrentControlSet实际上是一个注册表符号链接，它指向其中的一份拷贝。这些控制集有着形如HKLM\SYSTEM\ControlSet*nnn*这样的名称，其中*nnn*是诸如001或者002这样的数值。HKLM\SYSTEM\Select键中包含的值指出了每一

www.TopSage.com

份控制集的用途。例如，若CurrentControlSet 指向ControlSet001，那么Select下的Current值就是1。Select下的LastKnownGood值包含了“最后已知的好控制集”的数值，这是最后一次用来成功引导的控制集。在你的系统上另一个可能出现在Select键下的值是Failed，它指向最后一次引导不成功、并且即使在“最后已知的好控制集”的帮助下也未能成功引导而不得不退出的控制集。图4.16显示了一个系统的控制集和Select值。

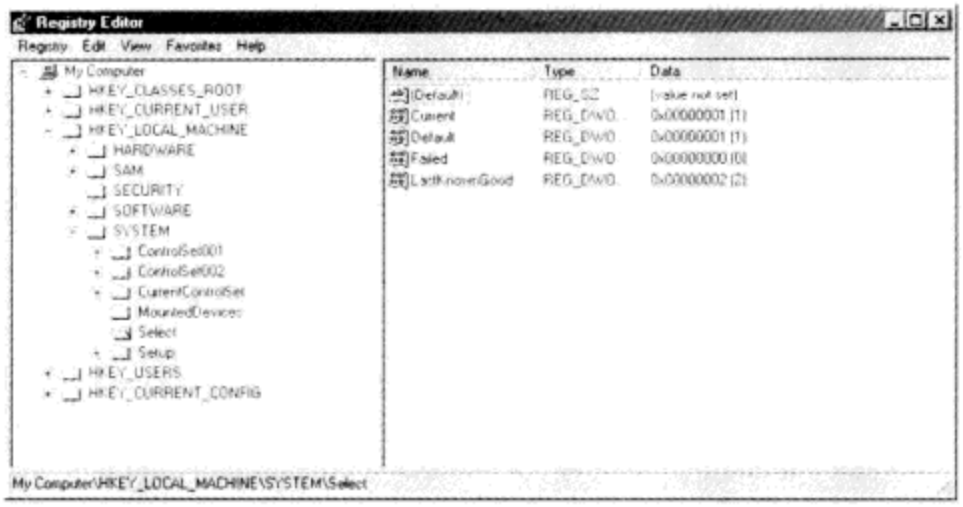


图4.16 控制集选择键

*NtInitializeRegistry*取出“最后已知的好控制集”中的内容，并且与CurrentControlSet键树中的控制集进行同步。如果这是系统第一次成功引导，则“最后已知的好控制集”并不存在，于是系统将创建一个新的控制集。如果“最后已知的好控制集”已经存在了，则系统只是简单地用它与CurrentControlSet之间的差值对它进行更新。

“最后已知的好控制集”对于因CurrentControlSet的一个变化而引起随后引导失败的情形也是非常有帮助的，比如在HKLM\SYSTEM\Control下面有一个调节系统性能的值被修改了，或者新增加了一个服务或设备驱动程序。用户可以在引导过程的早期按下F8，得到一个菜单，然后在菜单中直接选择使用“最后已知的好控制集”，将系统的注册表配置回滚到最后一次系统成功引导时的配置。第5章将会更加详细地讨论如何使用“最后已知的好控制集”和其他的恢复机制来诊断系统启动时出现的问题。

服务失败

在一个服务的注册表键中，可以有可选的FailureActions和FailureCommand值；SCM在服务启动过程中向系统作了登记，因而当服务进程退出时系统可以发信号通知SCM。当一个服务进程意外终止时，SCM确定该进程中运行的是哪些服务，并且根据其与失败有关的注册表值中指定的恢复步骤来采取行动。

一个服务可以配置的失败动作包括重新启动该服务、运行一个程序，以及重新引导计算机。而且，一个服务可以指定当它第一次失败、第二次失败，以及后续失败时可以采取的动作，它

www.TopSage.com

也可以指定，如果该服务要求重新启动的话，那么，SCM在重启该服务之前要等待多长时间。IIS Admin Service的服务失败动作是指示SCM运行IISReset应用程序，该程序会执行一些清理工作，然后重新启动该服务。你利用Services MMC加载件中一个服务的Properties对话框中的Recovery页面，就能很容易地管理该服务的恢复动作，如图4.17所示。

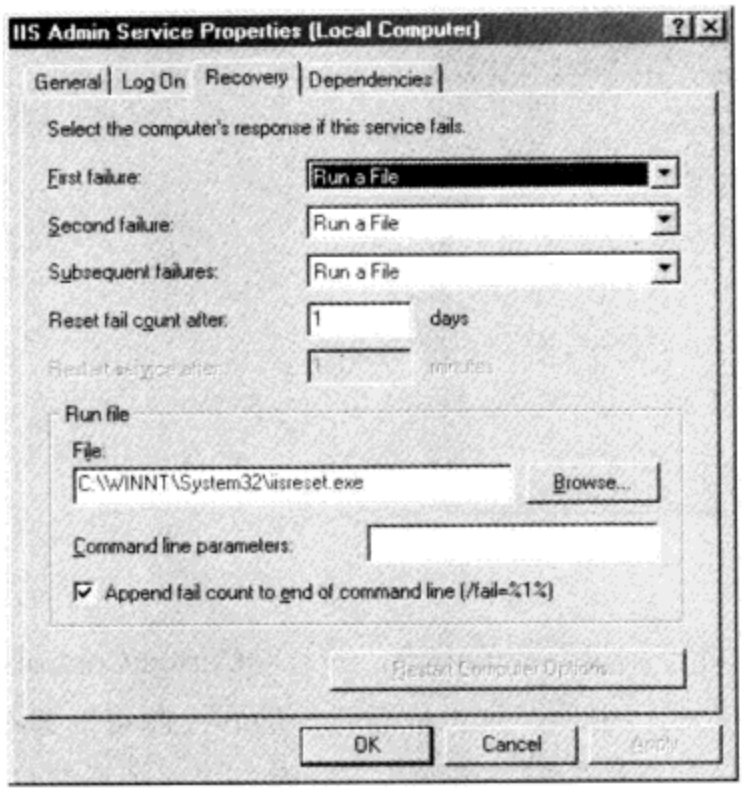


图4.17 服务恢复选项

服务停机

当Winlogon调用Windows的*ExitWindowsEx*函数时，*ExitWindowsEx*向Windows子系统进程Csrss发送一个消息，以便调用Csrss的停机例程。Csrss对所有活动的进程进行循环，通知它们系统正在停机。对于除了SCM以外的每一个系统进程，Csrss都要等待到由HKU\.DEFAULT\Control Panel\Desktop\WaitToKillAppTimeout指定的秒数（默认值为20秒），以便让该进程退出，然后再转移到下一个进程。当Csrss碰到SCM进程时，它也会通知SCM进程系统正在停机，但是使用一个专门针对SCM的超时间隔值。Csrss使用进程ID来识别SCM，当SCM在系统初始化过程中利用*RegisterServicesProcess*函数向Csrss登记时，Csrss将SCM的进程ID保存下来了。SCM的超时间隔值之所以不同于其他的进程，是因为，Csrss知道，SCM与那些在停机时需要执行清理工作的服务进行通信，所以，管理员可能需要单独调节SCM的超时间隔值。SCM的超时间隔值位于HKLM\SYSTEM\CurrentControlSet\Control\WaitToKillServiceTimeout注册表值中，其默认值也是20秒。

www.TopSage.com

凡是在初始化时请求SCM发送停机通知的Windows服务，SCM的停机处理器（shutdown handler）负责给它们发送通知。SCM函数*ScShutdownAllServices*在SCM服务数据库中进行循环，找到那些希望得到停机通知的服务，并且给每一个这样的服务发送一个停机命令。对于发送了停机命令的每一个服务，SCM记录下该服务的“等待提示（wait hint）”值，这是该服务在向SCM登记时指定的一个值。SCM跟踪记录了它所接收到的最大的等待提示值。在发送了停机消息以后，SCM开始等待，要么等到它所通知的服务有一个已经退出，或者等到指定的最大等待提示值已经过。

如果等待提示值已经到期，仍然没有哪个服务退出，那么，SCM确定是否它所等待退出的一个或者多个服务已经向SCM发送了一个消息，告诉SCM该服务正在进行停机处理。如果至少有一个服务正在进行之中，那么，SCM再等待一个等待提示值周期。SCM继续执行此等待循环，直至要么所有的服务都退出，要么在等待提示值周期内，它所等待的服务全都没有通知它停机的进度。

当SCM忙于通知服务停机并等待它们退出的过程中，Csrss也在等待SCM退出。如果Csrss的等待结束了（即WaitToKillServiceTimeout时间到期），但SCM还没有退出，那么，Csrss只是简单地跳过去，继续它的停机过程。因此，在系统停机时，那些未能及时停机的服务，连同SCM一起，仍然保持运行。不幸的是，管理员无法知道他们是否应该加大WaitToKillServiceTimeout的值，即便在他们的系统上，有的服务没有机会在系统停机前完全关闭。关于停机过程的更多信息，请参见第5章中的“停机”一节。

共享的服务进程

在单独的进程中运行每一个服务，而不是在可能的情况下让多个服务共享同一个进程，必然会浪费系统资源。然而，共享进程也意味着，如果该进程中的任何一个服务出现了错误因而导致进程退出，则该进程中所有的服务都终止了。

在Windows的内置服务中，有些运行在它们自己的进程中，而有些则与其他的服务共享一个进程。例如，SCM进程宿纳了事件日志（Event Log）服务和用户模式的即插即用（Plug and Play）服务，而LSASS进程包含了与安全相关的一些服务，比如安全账户管理器（SamSs）服务、网络登录（Netlogon）服务，以及IPSec策略引擎（PolicyAgent）服务。

有一个名为“服务宿主（Service Host）”的“通用”进程（SvcHost \Windows\System32\Svchost.exe）可以包含多个服务。SvcHost的多个实例可以运行在不同的进程中。在SvcHost进程中运行的服务有电话（TapiSrv）服务、远过程调用（RpcSs）服务和远程访问连接管理器

www.TopSage.com

（RasMan）服务。运行在SvcHost中的服务被实现为DLL，Windows在该服务的注册表键中包含了一个形如“%SystemRoot%\System32\svchost.exe -k netsvcs”这样的ImagePath定义。该服务的注册表键还必须有一个名为ServiceDll的注册表值，它位于Parameters子键下面，其中包含了该服务的DLL文件。

共享同一个SvcHost进程的所有服务必须指定同样的参数（比如，在上一段例子中的“-k netsvcs”），所以，它们在SCM的映像数据库中只有一个纪录。当SCM在服务启动过程中碰到第一个ImagePath为SvcHost且带有特定参数的服务时，它创建一个新的映像数据库纪录，并且用此参数激发一个SvcHost进程。新的SvcHost进程带有此参数，它在HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost下查找一个以此参数作为名称的注册表值。SvcHost读取该值的内容，并且将它解释成一个服务名称列表；当SvcHost向SCM登记时，它通知SCM，自己正在宿纳这些服务。图4.18给出了一个Svchost注册表键示例，它显示了一个以“-k netsvcs”为参数的SvcHost进程正准备宿纳一些与网络有关的不同服务。

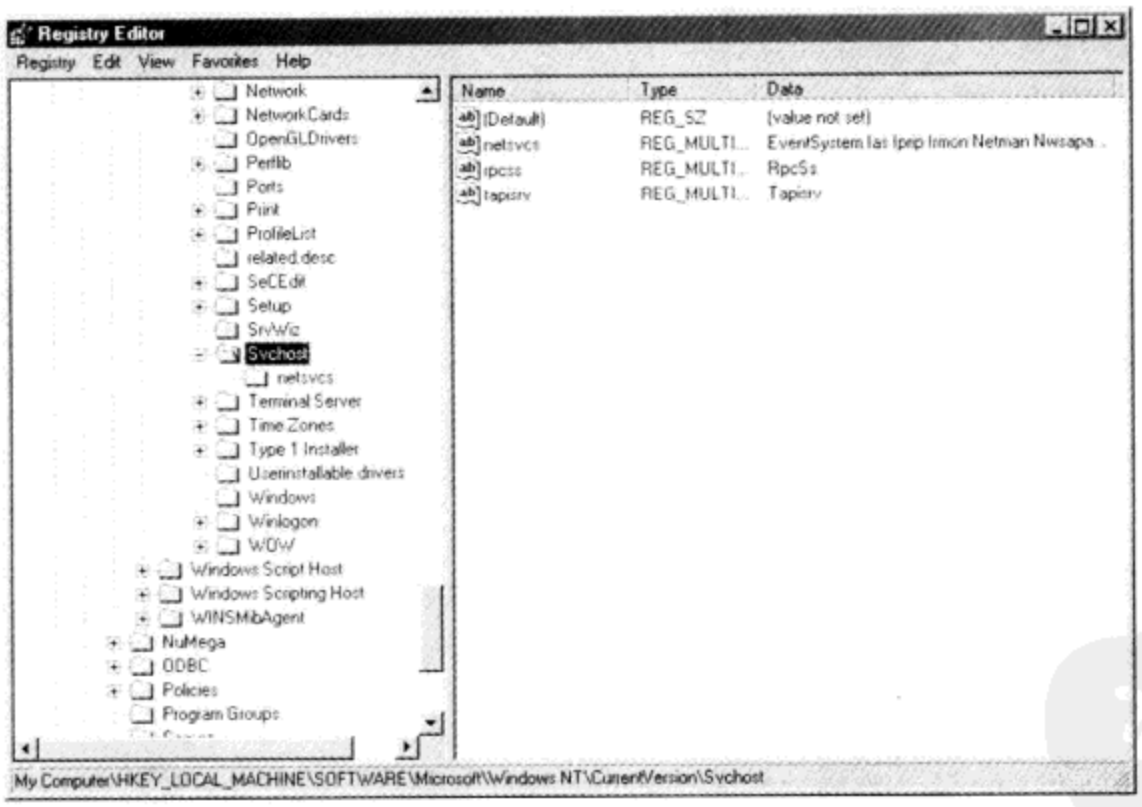


图4.18 SvcHost注册表键

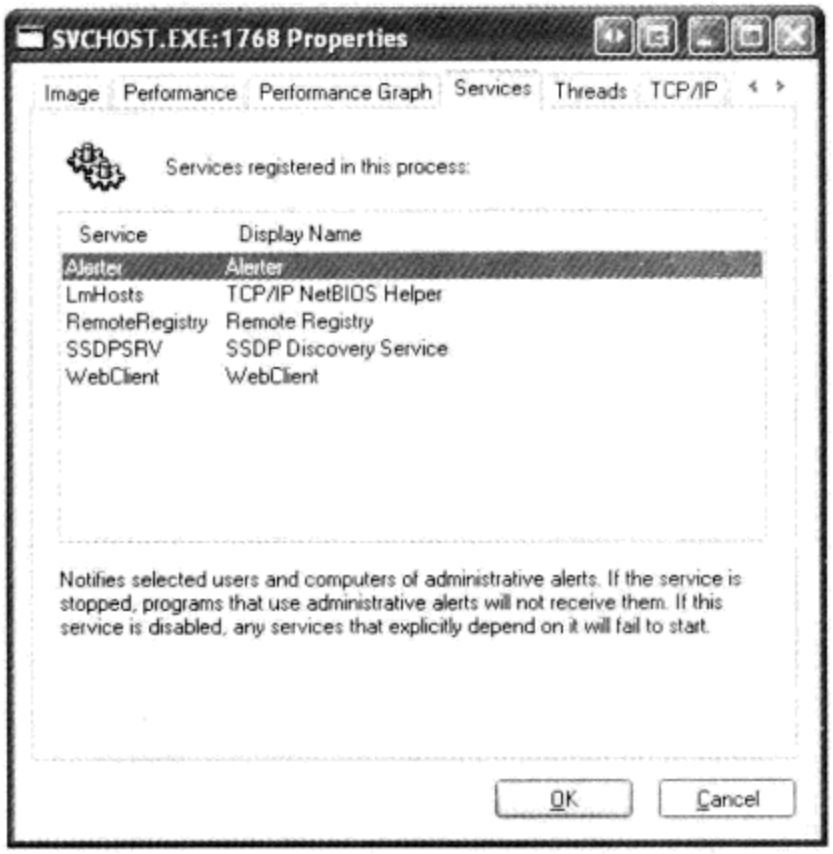
当SCM在服务启动过程中碰到一个SvcHost服务，并且其ImagePath与映像数据库中已有的纪录相匹配时，它不会再激发第二个进程，而是向早先为了该ImagePath值而启动起来的SvcHost发送一个启动命令，让它启动该服务。原先的SvcHost进程读入该服务注册表键中的ServiceDll参数，将该DLL加载到它的进程中，以便启动此服务。

www.TopSage.com



实验：查看进程中运行的服务

从*www.sysinternals.com*下载的进程管理器（Process Explorer）工具显示了关于进程内部运行的服务的细节信息。运行进程管理器，在下列进程的属性对话框上查看Services页面：Services.exe、Lsass.exe和Svchost.exe。在你的系统上有几个SvcHost实例在运行，你只要在进程管理器窗口上加入Username栏，就可以看到每个实例运行在哪个账户下，或者你在一个进程的属性对话框的Image页面上检查Username域也可以看到这一点。下面的图显示了在本地服务账户中执行的SvcHost进程内部所运行的服务列表：



这里显示的信息包括服务的名称、服务的显示名，以及服务说明（如果有的话）。当你选中一个服务时，进程管理器将该服务的说明显示在服务列表的下方。

你也可以使用Windows支持工具箱中的tlist.exe工具，或者随Windows XP和Windows Server 2003一起发行的Tasklist，从命令行上查看进程内部运行的服务列表。用Tlist来查看服务的语法是：

```
tlist /s
```

使用tasklist的语法是：

```
tasklist /svc
```

注意，这些工具并没有给出服务的显示名和说明，而只有服务名称。

www.TopSage.com

服务控制程序

服务控制程序是标准的Windows应用程序，它用到了XSCM服务管理函数，包括*CreateService*、*OpenService*、*StartService*、*ControlService*、*QueryServiceStatus*和*DeleteService*。为了使用SCM的函数，SCP必须首先调用*OpenSCManager*函数，以打开一个通向SCM的通信通道。在调用该函数的时候，SCP必须指定它想要执行的动作的类型。例如，如果一个SCP只是简单地想要枚举并显示SCM数据库中出现的服务，那么，它在调用*OpenSCManager*时请求“枚举服务”访问许可。SCM在初始化过程中，创建了一个代表SCM数据库的内部对象，并且利用Windows的安全功能，通过一个安全描述符来保护该对象。此安全描述符指定了哪些账户可以用什么样的访问许可来打开该对象。例如，此安全描述符指示Authenticated Users组可以以“枚举服务”的访问许可来打开该SCM对象。然而，只有管理员组才可以请求以创建或删除服务的访问方式来打开该对象。

如同SCM数据库对象一样，SCM也为这些服务本身实现了安全保护功能。当一个SCP利用*CreateService*函数来创建一个服务时，它指定一个安全描述符，SCM在内部将此安全描述符与该服务在服务数据库中的纪录关联起来。SCM将此安全描述符保存在该服务注册表键的Security值中， SCM在初始化过程中扫描注册表的Services键时，读入此Security值，所以，即使机器重新引导以后，安全设置也会一直有效。如同SCP必须要在*OpenSCManager*调用中指定它将要以什么类型的方式来访问SCM数据库一样，SCP也必须在*OpenService*调用中告诉SCM，它想要怎样访问一个服务。一个SCP可以请求的访问方式有：能够查询一个服务的状态，可以配置、停止和启动一个服务。

你可能最熟悉的SCP是Windows自带的Services MMC加载件，位于\Windows\System32\Filemgmt.dll中。Windows XP和Windows Server 2003包含了一个命令行服务控制程序Sc.exe（Service Controller tool，服务控制器工具），Windows 2000资源工具箱中也包含了针对Windows 2000的版本。

有时候，SCP在SCM实现的服务策略之上又叠加了一层服务策略。一个很好的例子是，Services MMC加载件在手工启动一个服务时实现了不同的超时策略。该加载件显示一个进度条，表示一个服务正在启动过程中。尽管SCM会无限地等待一个服务来响应启动命令，但Services加载件在进度条到达100%以前只等待两分钟，如果两分钟没等到，则宣称该服务未能及时启动。Windows服务在响应SCM命令（比如启动命令）时，通过设置它们的配置状态来反映它们的进度情况，从而可以间接地与SCP进行交互。SCP通过*QueryServiceStatus*函数来查询此状态。它们可以辨别何时一个服务主动更新了此状态，何时一个服务像是被挂起了。因此，SCM可以采取适当的动作来通知一个用户该服务当前的状况。

www.TopSage.com

4.3 Windows管理规范

Windows NT一直以来总是集成了性能监视工具和系统事件监视工具。应用程序和系统通常使用事件管理器（Event Manager）来报告错误消息和用于诊断的消息。事件查看器（Event Viewer）工具让管理员们既可以检查本地计算机上的事件输出，也可以检查网络上另一台计算机的事件输出。类似地，性能计数器机制让应用程序和操作系统组件可以向性能监视应用程序，比如性能监视器（Performance Monitor），报告一些与性能有关的统计信息。

尽管Windows NT 4的事件监视特性和性能监视特性满足了它们的设计目标，但是，它们仍然有局限性。例如，编程接口各不相同，这种差异性使得那些利用事件和性能监视功能来采集数据的应用程序变得非常复杂。Windows NT 4中这些监视设施的最大缺陷可能是，它们缺乏（甚至根本没有）扩展性，而且，无论是事件日志记录功能，还是性能数据的采集功能，都没有提供对于管理API来说十分必要的双向交互能力。应用程序必须按照预定义的格式来提供数据。性能API没有为应用程序提供“接收与性能有关的事件通知”的能力，并且，那些向事件管理器请求事件通知的应用程序，无法限制只接收某些特定类型的事件通知，或者只接收某些来源的事件通知。最后，无论哪种采集设施，都不可能通过事件管理器或者性能API，与事件数据或者性能数据的提供者进行通信。

为了解决这些局限性，以及为其他类型的数据源也提供管理能力，Windows还有一种新的管理机制，称为Windows管理规范（WMI，Windows Management Instrumentation）。WMI是WBEM（Web-Based Enterprise Management）的一个实现，而WBEM则是DMTF（Distributed Management Task Force，一个工业界联盟）定义的一个标准。WBEM标准包含了一套可扩展的、针对企业的数据采集和数据管理设施的设计方案，此设计方案具有很强的灵活性和扩展性；为了管理具有任意多个组件的本地和远程系统，这样的灵活性和扩展性是非常必要的。对WMI的支持是在Windows NT 4的Service Pack 4中加入进来的。在Windows 95 OSR2、Windows 98和Windows Millennium中也支持WMI。尽管本节的绝大多数内容适用于所有支持WMI的Windows平台，但是，其实现细节仅仅针对于Windows 2000、Windows XP和Windows Server 2003。

WMI体系结构

WMI是由四个主要的部件构成的，如图4.19所示。这四个部件是管理应用程序、WMI基础设施、提供者和被管理的对象。管理应用程序是Windows应用程序，它们访问有关被管理对象的数据，并且显示这些数据，或者对它们进行处理。管理应用程序的一个简单例子是，利用WMI而不是性能API来获得有关性能信息的性能工具。另一个稍微复杂一点的例子是一个企业管理工具，它使得管理员可以自动完成计算机设备的清查工作，即将他们企业中每台计算机的软硬件配置信息收集到指定的地点。

www.TopSage.com

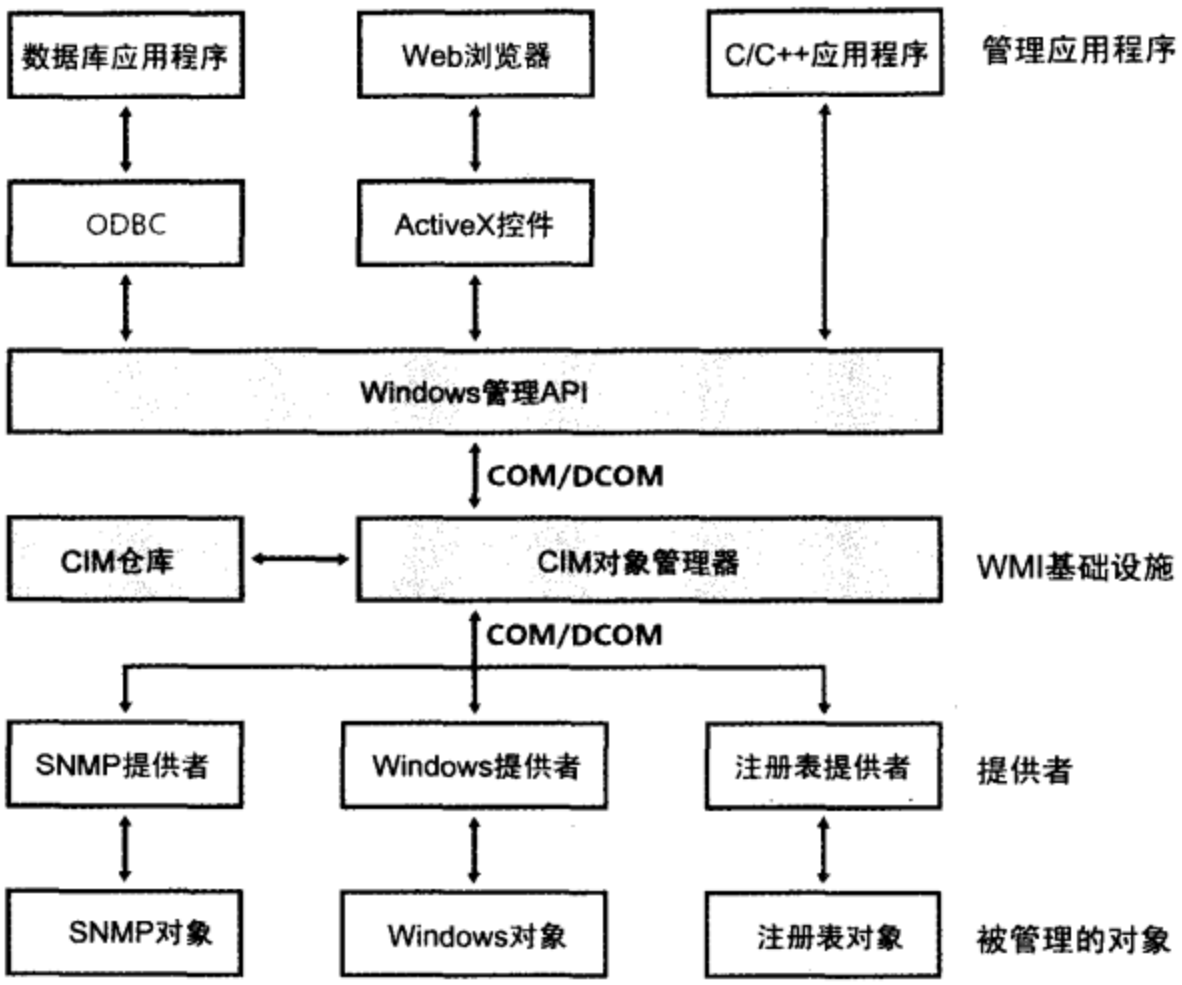


图4.19 WMI体系结构

开发人员通常将管理应用程序的目标定义为采集有关特定对象的数据，以及管理这些对象。一个对象可能代表了一个组件，比如网络适配器设备，也可能代表了一组组件，比如一台计算机（计算机对象可能包含了网络适配器对象）。提供者则需要为管理应用程序可能感兴趣的对象，定义和导出它们的表示形式。例如，网络适配器的厂商可能想要在Windows包含的网络适配器的WMI支持中，再加入一些与他们的适配器有关的属性，从而允许管理应用程序可以直接查询和设置适配器的状态和行为。在有些情况下（比如，针对设备驱动程序），Microsoft提供了一个具有专属API的提供者，以帮助开发人员用最小的编程努力为他们自己的管理对象实现一个提供者。

WMI基础设施是把管理应用程序和提供者绑定在一起的粘合剂，其核心是CIM对象管理器（CIMOM，Common Information Model Object Manager）（本章后面将会讲述CIM）。此基础设施也被用作对象-类的存储体，在许多情况下，还被用作永久对象属性的存储管理器。WMI将此存储体，或者说仓库，实现为磁盘上的数据库，称为CIMOM对象仓库。作为此基础设施的一部分，WMI支持几个API；管理应用程序通过这些API可以访问对象的数据，而提供者则通过这些API来提供数据和类的定义。

www.TopSage.com

Windows程序使用WMI COM API来直接与WMI打交道，这是最为基本的管理API。其他的API建立在此COM API基础之上，其中包括针对Microsoft Access数据库应用程序的ODBC（Open Database Connectivity）配接接口。数据库开发人员使用WMI ODBC配接接口，来嵌入对于他们自己数据库中的对象数据的引用。然后，开发人员可以通过数据库查询，来得到基于WMI的数据，从而很容易生成相应的数据报告。WMI ActiveX控件支持另一层API。Web开发人员使用ActiveX控件，来为WMI数据构造出基于Web的界面。另一个管理API是WMI脚本API，主要为了在基于脚本的应用程序和Microsoft Visual Basic程序中使用。WMI的脚本支持也是为了所有的Microsoft程序设计语言技术而存在的。

如同针对管理应用程序的情形一样，WMI COM接口也是提供者最为基本的API。然而，不像管理应用程序是COM客户，提供者是COM或者分布式COM（DCOM）服务器（也就是说，提供者实现了COM对象，而WMI与这些COM对象打交道）。WMI提供者的可能实现方式有被加载到WMI管理器进程中的DLL、单独的Windows应用程序，或者Windows服务。Microsoft包含了许多内置的提供者，它们供应的数据来自于大家熟悉的各种数据源，比如性能API、注册表、事件管理器、活动目录、SNMP和WDM（Windows Driver Model，Windows驱动程序模型）设备驱动程序。WMI SDK让开发人员可以开发第三方WMI提供者。

提供者

WBEM的核心是DMTF设计的CIM规范。CIM规定了管理系统如何从系统管理的角度出发，将一台计算机的方方面面展示给一台计算机上的程序或者设备。提供者的开发人员使用CIM来表达那些“属于他们想要管理的应用中的各种部件”。开发人员使用“可管理对象的格式”（MOF，Managed Object Format）语言来实现CIM表示。

提供者除了定义一些类来表达对象以外，还必须作为接口将WMI与这些对象连接起来。WMI根据这些提供者所支持的接口特性对它们进行分类。表4.10列出了WMI提供者的分类。注意，每个提供者可以实现一个或者多个特性，因此，举例来说，一个提供者既可以是一个类，也可以是一个事件提供者。为了清楚地阐述表4.10中给出的特性定义，我们来看一个实现了其中多个特性的提供者。事件日志（Event Log）提供者支持几个对象，包括事件日志计算机（Event Log Computer）、事件日志纪录（Event Log Record）和事件日志文件（Event Log File）。事件日志（Event Log）是一个实例（Instance）提供者，因为它可以为它的类中的某几个类定义多个实例。其中一个可以定义多个实例的类是事件日志文件类（Win32_NTEventlogFile）。事件日志提供者为系统中的每一个事件日志（即系统事件日志[System Event Log]、应用事件日志[Application Event Log]或安全事件日志[Security Event Log]）定义该类的一个实例。

www.TopSage.com

表4.10 提供者类别

分 类	说 明
类（Class）	可以提供、修改、删除和枚举一个特定于提供者的类。也可以支持查询处理。活动目录作为一个类提供者服务，是一个很罕见的例子
实例（Instance）	可以提供、修改、删除和枚举系统类的实例，或者特定于提供者的类的实例。一个实例代表一个被管理的对象。也可以支持查询处理
属性（Property）	可以提供和修改单独的对象属性值
方法（Method）	为一个特定于提供者的类提供一些方法
事件（Event）	产生事件通知
事件消费者（Event consumer）	将一个物理消费者映射到一个逻辑消费者，以便支持事件通知

事件日志提供者定义了实例数据，让管理应用程序可以枚举它的纪录。为了使管理应用程序可以使用WMI来备份和恢复事件日志文件，事件日志提供者为事件日志文件对象实现了备份和恢复方法。这样做使得事件日志提供者成了一个方法（Method）提供者。最后，管理应用程序可以注册成：无论何时当一条新的纪录被写入到任一个事件日志中时它可以接收到通知。因此，当事件日志提供者使用WMI事件通知机制来告诉WMI事件日志纪录已经到达时，它就成了一个事件（Event）提供者。

公共信息模型（CIM）和可管理对象的格式语言

CIM也遵从像C++和Java这样的面向对象语言的步骤，在这些语言中，它们用类来表示问题中的数据。由于采用了类的概念，因而开发人员可以使用强大的建模技术，包括继承和复合。子类可以继承父类的属性，也可以增加自己的特征，还可以覆盖从父类继承过来的特征。如果一个类从另外一个类继承了属性或特征，则称前一个类**派生**自后一个类。类也能够**复合**起来：开发人员可以构造一个包含了多个其他类的类。

DMTF提供了多个类，这些类作为WBEM标准的一部分。这些类是CIM的基本语言，它们表达了可适用于所有管理领域的对象。这些类是CIM**核心模型**的一部分。核心类的一个例子是CIM_ManagedSystemElement。该类包含少数几个基本属性，这些属性标识了像硬件设备这样的物理组件，以及像进程和文件这样的逻辑组件。这样的属性有标题、说明、安装日期和状态。所以，CIM_LogicalElement 和 CIM_PhysicalElement 这两个类都继承了CIM_ManagedSystemElement类的属性。这两个类也是CIM核心模型的一部分。WBEM标准将这些类称为**抽象**类，因为它们存在的惟一理由是为了让其他的类继承它们（也就是说，抽象类不存在对象实例）。因此，你也可以把抽象类想象成专门用于为其他类定义属性的模板。

www.TopSage.com

第二种类别的类表达了那些特定于管理领域但是与具体实现无关的对象。这些类构成了**公共模型**（*common model*），它们被认为是核心模型的一个扩展。公共模型类的一个例子是CIM_FileSystem类，它继承了CIM_LogicalElement的属性。因为几乎每一个操作系统，包括Windows、Linux和其他版本的UNIX，都依赖于基于文件系统的结构化存储，所以，CIM_FileSystem类是公共模型中一个很恰当的组成部分。

最后一种类别的类是**扩展模型**（*extended model*），它是由一些附加于公共模型之上的、与特定技术相关的类组成的。Windows定义了一大堆这种类来表达那些特定于Windows环境的对象。因为所有的操作系统都将数据存储在文件中，所以，CIM公共模型包含了CIM_LogicalFile类。CIM_DataFile类继承了CIM_LogicalFile类，另外，Windows针对其页面文件和快捷文件这两种文件类型分别加入了Win32_PageFile和Win32_ShortcutFile文件类。

事件日志（Event Log）提供者大量地使用了继承技术。图4.20显示了WMI CIM Studio的一个视图，这是随WMI Administrative Tools一起带的类浏览器（你可以从Microsoft Web站点的Microsoft下载中心获得WMI Administrative Tools）。你可以看到，事件日志提供者依赖于继承技术，该提供者的Win32_NTEventlogFile类派生自CIM_DataFile。事件日志文件是具有额外特性的数据文件，这些额外的特性与事件日志有关，比如日志文件名（LogfileName）、该文件所包含的纪录的数量（NumberOfRecords）。类浏览器中显示的树反映了Win32_NTEventlogFile具有多层继承性：CIM_DataFile派生自CIM_LogicalFile，而CIM_LogicalFile又派生自CIM_LogicalElement，而CIM_LogicalElement又派生自CIM_ManagedSystemElement。

正如前面所述，WMI提供者的开发人员使用MOF语言来编写他们的类。下面的输出显示了事件日志提供者的Win32_NTEventlogFile类的定义，图4.20中选中的正是该类。请注意观察，图4.20中右边窗格中列出的属性，和下面的MOF文件中这些属性的定义，两者是怎样关联起来的。CIM Studio使用黄色的箭头来标记那些因继承而得来的属性。因此，你在Win32_NTEventlogFile的定义中看不到这些特定的属性。

```
dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"), Locale(1033), UUID("{8502C57B-
5FBB-11D2-AAC1-006008C78BC7}")]
class Win32_NTEventlogFile : CIM_DataFile
{
[read] string LogfileName;
[read, write] uint32 MaxFileSize;
[read] uint32 NumberOfRecords;
[read, volatile, ValueMap{"0", "1..365", "4294967295"}] string OverWritePolicy;
[read, write, Units("Days"), Range("0-365 | 4294967295")] uint32 OverwriteOutDated;
[read] string Sources[];
```


www.TopSage.com

```
[implemented, Privileges{"SeSecurityPrivilege",
"SeBackupPrivilege"}] uint32 ClearEventlog([in] string ArchiveFileName);
[implemented, Privileges{"SeSecurityPrivilege",
"SeBackupPrivilege"}] uint32 BackupEventlog([in] string ArchiveFileName);
};
```

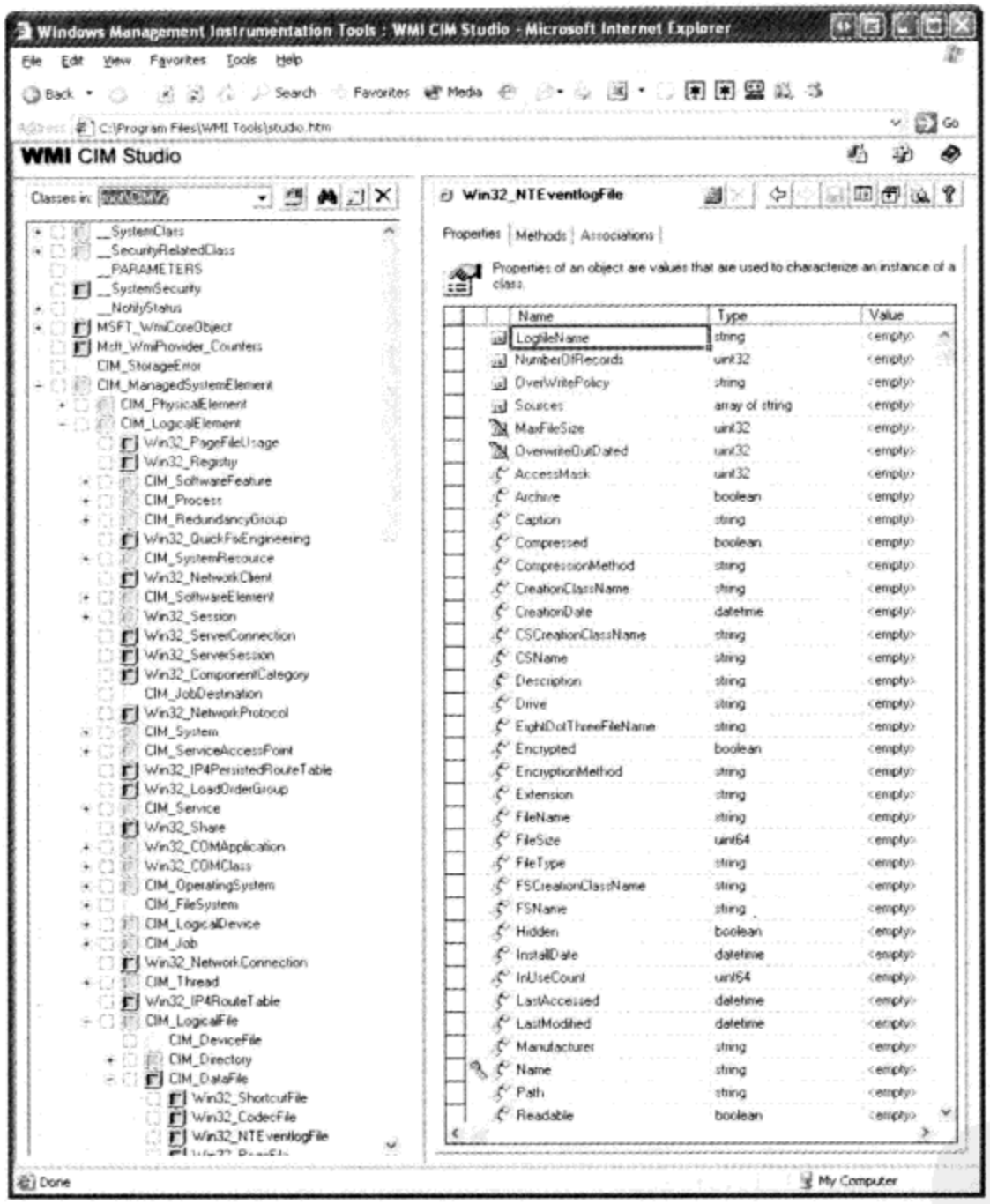


图4.20 WMI CIM Studio示例图

一个值得讨论的术语是*dynamic*（**动态的**），这是出现在上面MOF文件中的、用于说明Win32_NTEventlogFile类的指示符。*Dynamic*意味着，当一个管理应用程序查询该类的对象的属性时，WMI基础设施向WMI提供者询问与该对象相关联的属性的值。*Static*（**静态的**）类是指WMI仓库中的类；WMI基础设施仅仅依靠WMI仓库就可以得到这些值，因而无需向提供者询问这些值。因为更新WMI仓库是一个相对昂贵的操作，所以，对于那些属性频繁变化的对象，动态提供者更加高效。

www.TopSage.com



实验：查看WMI类的MOF定义

你利用Windows随带的WbemTest工具，就可以查看任何WMI类的MOF定义。在这个实验中，我们来看一看Win32_NTEventLogFile类的MOF定义。

1. 通过Start菜单的Run对话框运行Wbemtest。
2. 点击“Connect（连接）”按钮，将名字空间改变成root\cimv2，再连接。
3. 选择“Enum Classes（枚举类别）”，再选择“Recursive（递归）”选项按钮，然后点击“OK”。
4. 在列出来的类中找到Win32_NTEventLogFile，然后双击该类，以便看到它的属性。
5. 点击“Show MOF（显示MOF）”按钮，会打开一个窗口，其中显示了该类的MOF文本。

在WMI开发人员使用MOF编写了他们的类以后，他们可以通过几种不同的方式将这些类定义提供给WMI。WDM提供者的开发人员可将一个MOF文件编译成一个二进制MOF（BMF）文件（这是一种比MOF文件更加紧凑的二进制表示形式），并且将这些BMF文件交给WDM基础设施。另一种用法是，提供者编译该MOF文件，并且通过WMI COM API，将这些定义交给WMI基础设施。最后一种做法是，提供者可以使用MOF编译器（Mofcomp.exe）工具，将一个类编译之后的表示形式直接交给WMI基础设施。

WMI名字空间

类定义了对象的属性，而对象则是一个系统上的类实例。WMI使用一个名字空间将对象组织起来，该名字空间包含了几个子名字空间，WMI按照层次结构来安排这些子名字空间。管理应用程序在访问一个名字空间中的对象以前，必须先与该名字空间建立连接。

WMI将名字空间的根目录命名为root。所有安装的WMI都有四个预定义的名字空间，它们位于root的下面，分别是CIMV2、Default、Security和WMI。在这四个名字空间中，有的还有其他的子名字空间。例如，CIMV2包含的Applications和ms_209这两个名字空间作为自己的子名字空间。有时候提供者还定义了它们自己的名字空间。你可以在Windows中看到root下面的WMI名字空间（这是Windows设备驱动程序WMI提供者定义的名字空间）。

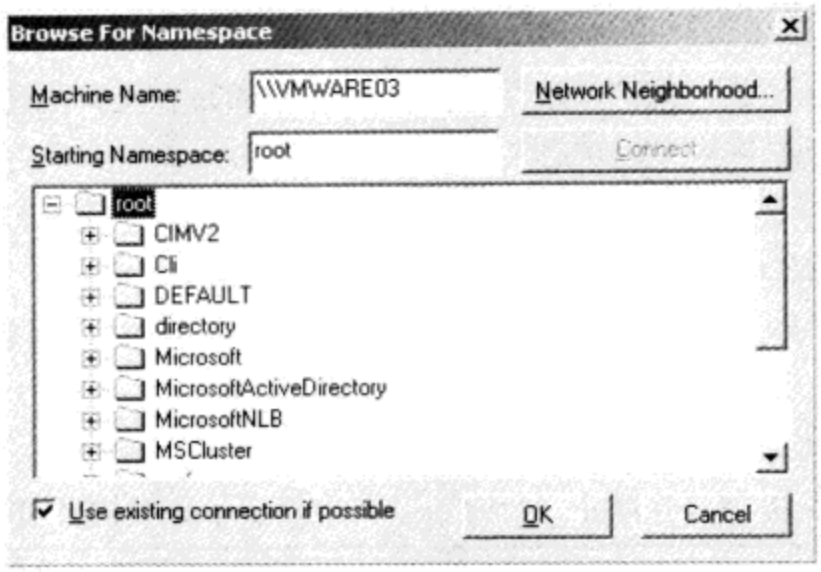


实验：查看WMI名字空间

利用WMI CIM Studio，你可以看到一个系统中定义了哪些名字空间。当你运行WMI CIM Studio时，它会出现一个连接对话框，在名字空间编辑框的右侧包括了一个名字空间浏览按钮。打开名字空间浏览器，并选中一个名字空间，这样可以让WMI CIM Studio连接到该名字空间上。Windows Server 2003定义了不止10个名字空间，它们位于root下面，其中有些名字空间在

www.TopSage.com

下面的界面中可以看到。



不像文件系统的名字空间是由目录和文件的层次结构构成的，WMI名字空间只有一层的深度。不像文件系统那样使用名称来标识文件或目录，WMI使用对象的属性来标识对象，它把这些属性定义成**键**（*key*）。管理应用程序指定类名和键名来找到一个名字空间内部的特定对象。因此，一个类的每个实例必须由它的键值惟一标识。例如，事件日志提供者使用Win32_NTLogEvent类来表达事件日志中的一个纪录。该类有两个键：一个是字符串Logfile，另一个是无符号整数RecordNumber。如果一个管理应用程序要向WMI查询事件日志纪录的实例，则它可以利用能标识出这些纪录的键对来获得它们。应用程序利用特定的语法来引用一条纪录，在下面的例子对象路径名中你可以看到这种语法：

```
\\DARYL\root\CIMV2:Win32_NTLogEvent.Logfile="Application",
                                    RecordNumber="1"
```

该名称中的第一个部件（\\DARYL）标识了该对象所在的计算机，第二个部件（\root\CIMV2）是该对象所处的名字空间。类名紧跟在冒号的后面，键名和对应的值跟在句点的后面。逗号把两个键值分隔开。

WMI提供了一些接口让应用程序可以枚举一个特定类中所有的对象，也可以发出一些查询请求以获得某个类中符合查询条件的那些实例。

类关联

许多对象类型彼此之间以某种方式联系在一起。例如，一个计算机对象可以有一个处理器、软件、一个操作系统、活动的进程，等等。WMI允许提供者构造一个**关联类**（*association class*）来表达两个不同的类之间的逻辑连接。关联类将一个类与另一个类关联起来，所以，这样的类仅有两个属性：一个类名和*Ref*修饰符。下面的输出显示了一种关联：事件日志提供者的MOF文件将Win32_NTLogEvent类与Win32_ComputerSystem类关联起来。给定一个对象，管理应用

www.TopSage.com

程序可以查询与其关联的对象。按照这种方式，提供者可以定义出一个对象层次结构。

```
[dynamic: ToInstance, provider("MS_NT_EVENTLOG_PROVIDER"): ToInstance, EnumPrivileges{
"SeSecurityPrivilege"}: ToSubClass, Locale(1033): ToInstance, UUID("{8502C57F-5FBB-11D2-
AAC1-006008C78BC7}"): ToInstance, Association: DisableOverride ToInstance ToSubClass]
class Win32_NTLogEventComputer
{
    [key, read: ToSubClass] Win32_ComputerSystem ref Computer;
    [key, read: ToSubClass] Win32_NTLogEvent ref Record;
};
```

图4.21显示了WMI对象浏览器（WMI Administrative Tools包含的另一个工具），它显示的是CIMV2名字空间中的内容。Windows的系统组件往往把它们的对象放在CIMV2名字空间中。对象浏览器首先找到Win32_ComputerSystem对象的实例MR-XEON，这是代表整个计算机的对象。然后，对象浏览器获得那些与Win32_ComputerSystem相关联的对象，并且将它们显示在MR-XEON的下面。对象浏览器的用户界面利用双箭头文件夹图标来表示关联对象。关联类类型的对象显示在该文件夹的下面。

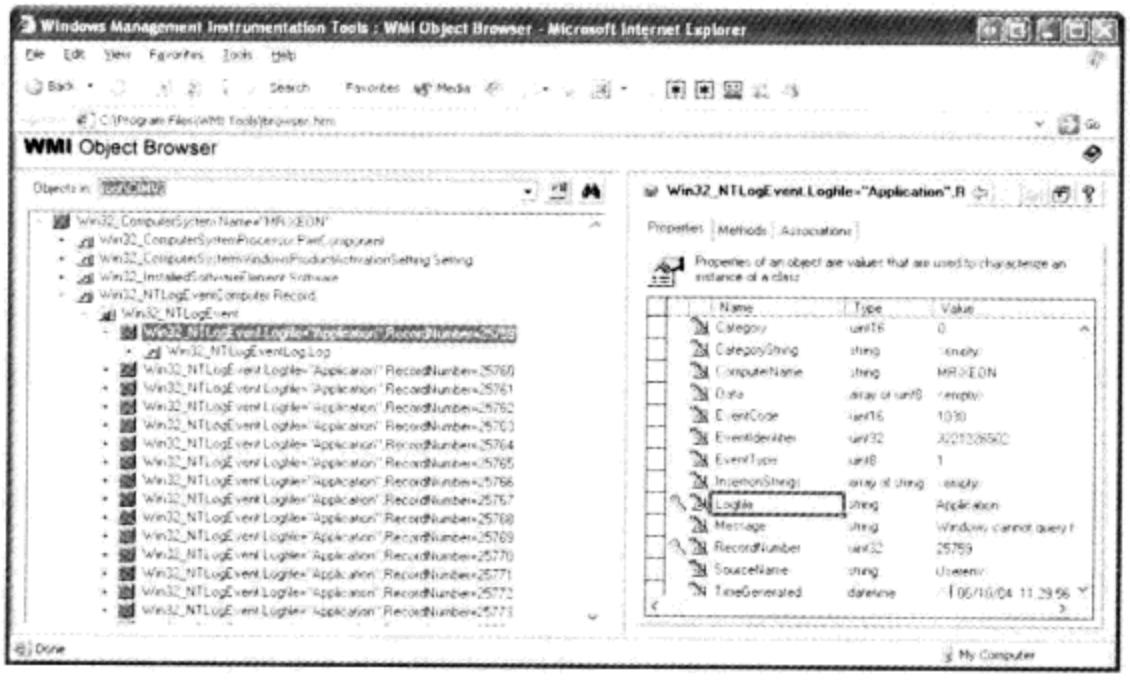


图4.21 WMI对象浏览器

在对象浏览器中，你可以看到，事件日志提供者的关联类Win32_NTLogEventComputer位于MR-XEON的下面，同时，Win32_NTLogEvent类有大量的实例。参照前面的代码输出，可以验证MOF文件定义了Win32_NTLogEventComputer类，将Win32_ComputerSystem与Win32_NTLogEvent关联起来。在对象浏览器中选择Win32_NTLogEvent的一个实例，你可以在右侧窗格的Properties表单中看到该类的属性。虽然Microsoft提供对象浏览器的用意是帮助WMI开发人员检查他们的对象，但是管理应用程序也可以执行同样的操作，并且以一种更加易于理解的方式来显示对象的属性或者收集到的信息。

www.TopSage.com



实验：使用WMI脚本来管理系统

WMI强大之处的一个方面是，它支持脚本语言。Microsoft已经生成了大量的脚本程序来执行一些公共的管理任务，譬如，管理用户账户、文件、注册表、进程和硬件设备。虽然有些脚本程序随Windows资源工具箱一起发布，但是，Web站点“Microsoft TechNet Scripting Center”则是Microsoft脚本程序的集中场所。通过该脚本程序中心来使用一个脚本非常容易，只要从你的Internet浏览器中将该脚本的文本拷贝下来，再将它保存到一个扩展名为.vbs的文件中，然后用命令“cscript **script**.vbs”来运行该脚本。这里的“script”是你为该脚本所起的名字。Cscript是Windows脚本宿主程序（WSH，Windows Script Host）的命令行界面。

下面是一个TechNet脚本示例，它首先请求，当Win32_Process对象实例被创建（当一个进程启动时就会创建这样的实例）时，它会接收到事件通知，然后它打印出该对象所代表的进程的名称。

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:"
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
Set colMonitoredProcesses = objWMIService. _
    ExecNotificationQuery("select * from __instancecreationevent " _
        & " within 1 where TargetInstance isa 'Win32_Process'")
i = 0
Do While i = 0
    Set objLatestProcess = colMonitoredProcesses.NextEvent
    Wscript.Echo objLatestProcess.TargetInstance.Name
Loop
```

调用*ExecNotificationQuery*的那一行代码，在参数中包含了一条“select”语句。此“select”语句正体现了WMI对于ANSI标准的结构化查询语言（SQL）的一个只读子集（也称为WQL）的支持，从而提供了一种灵活的方式让WMI消费者可以指定它们想要从WMI提供者那里提取什么样的信息。用Cscript来运行例子脚本，然后启动Notepad，如下所示：

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

NOTEPAD.EXE
```


www.TopSage.com

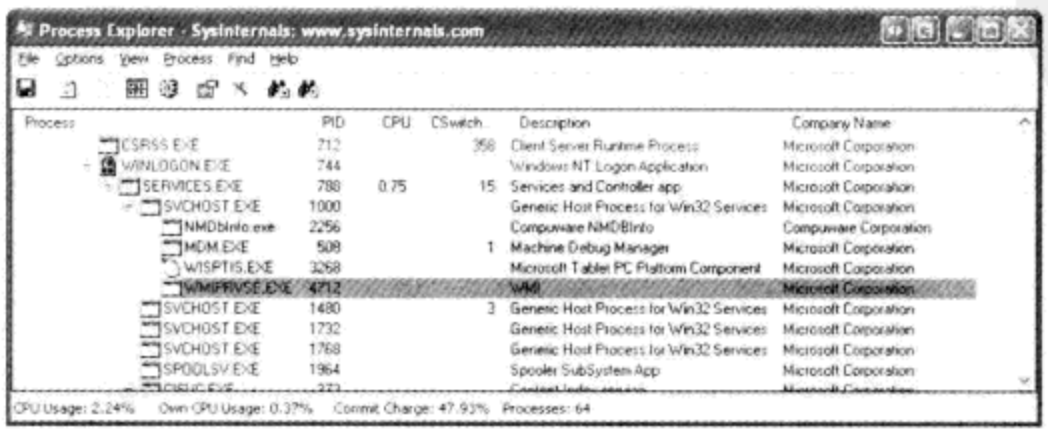
WMI实现

在Windows 2000中，WMI服务是在\Windows\System32\Winmgmt.exe中实现的，当第一次有管理应用程序或者WMI提供者试图访问WMI API时，Windows SCM就会将它启动起来。在Windows XP和Windows Server 2003中，WMI服务运行在一个共享的Svchost进程中，它在本地系统账户下执行。

在Windows 2000中，WMI将提供者加载为进程内的DCOM服务器，运行在Winmgmt服务进程内。如果一个提供者的程序错误导致WMI进程崩溃，那么，WMI服务就会退出，然后它又会重新启动，以响应下一个WMI请求。在Windows XP和Windows Server 2003中，因为WMI服务与其他的几个服务一起共享同一个Svchost进程，因此，如果某个WMI提供者有程序错误因而导致该进程退出的话，这些服务也会跟着退出，从而，WMI将提供者加载到一个专门宿纳提供者的进程Wmiprvse.exe中。Wmiprvse.exe作为RPC服务进程的一个子进程被启动起来。WMI在本地系统账户、本地服务账户或者网络服务账户下执行Wmiprvse，具体在哪个账户下取决于代表此提供者的WMI Win32Provider对象实例的*HostingModel*属性值。在该提供者被从缓存中去除以后，也就是说，它接收到最后一个提供者请求以后一分钟，Wmiprvse进程即退出。

实验：查看Wmiprvse的创建

通过运行*www.sysinternals.com*所提供的进程管理器，再执行Wmic，你就可以看到Wmiprvse被创建的情形。Wmiprvse进程将出现在宿纳RPC服务的Svchost进程的下面。如果进程管理器的作业加亮显示（job highlighting）被打开的话，则它也将以作业加亮颜色来显示。这是因为，为了防止一个失控的提供者消耗掉系统中所有的虚拟内存资源，Wmiprvse运行在一个具有限定条件的作业对象中，而该作业对象限定了它可以创建的子进程的数量、每个进程可以申请的虚拟内存数量，以及它的所有进程可以申请的虚拟内存数量（有关作业对象的更多信息，请参见第6章）。



www.TopSage.com

大多数WMI组件在默认情况下驻留在\Windows\System32和\Windows\System32\Wbem下，包括Windows MOF文件、内置的提供者DLL和管理应用程序WMI DLL。打开\Windows\System32\Wbem目录，你可以找到Ntevt.mof，这是事件日志提供者的MOF文件。你还可以找到Ntevt.dll，这是事件日志提供者的DLL文件，WMI服务要用到它。

\Windows\System32\Wbem下面的目录存放了仓库文件、日志文件和第三方的MOF文件。WMI使用一个私有版本的Microsoft JET数据库引擎来实现它的仓库，即CIMOM仓库。在Windows 2000中，此数据库文件存放在\Windows\System32\Wbem\Repository\Cim.rep中；而在Windows XP和Windows Server 2003中，此数据库驻留在\Windows\System32\Wbem\Repository\Fs中。

WMI用到了大量的注册表设置（包括各种内部性能参数，比如Windows 2000中的CIMOM备份的位置和一些时间间隔值），存放在该服务的HKLM\SOFTWARE\Microsoft\WBEM\CIMOM注册表键中。

设备驱动程序使用特殊的接口来向WMI提供数据，以及接受来自WMI的命令（称为WMI系统控制命令）。这些接口是WDM的一部分，第9章将会讲述。因为这些接口是跨平台的，所以它们位于\root\WMI名字空间的下面。

WMIC

Windows XP和Windows Server 2003包含了一个工具Wmic.exe，它使得你可以从一个能理解WMI的命令行外壳中与WMI进行交互。所有的WMI对象和它们的属性，也包括它们的方法，都可以通过该外壳进行访问，从而使得WMIC成为一个高级的系统管理控制台。

WMI安全性

WMI在名字空间层次上实现了安全性。如果一个管理应用程序成功地连接到一个名字空间，那么，此应用程序可以查看和访问该名字空间中所有对象的属性。管理员可以使用WMI Control应用程序来控制哪些用户可以访问哪个名字空间。为了启动WMI Control应用程序，你可以从Start菜单中，选择Programs，又选择Administrative Tools，再选择Computer Management。接下来，打开Services And Applications分支，右键点击WMI Control，选择Properties，就可以打开WMI Control Properties对话框，如图4.22所示。为了配置名字空间的安全性，点击Security页面标签，选中一个名字空间，再点击Security。通过WMI Control Properties对话框中的其他页面，你可以修改注册表中存储的性能和备份设置。

www.TopSage.com

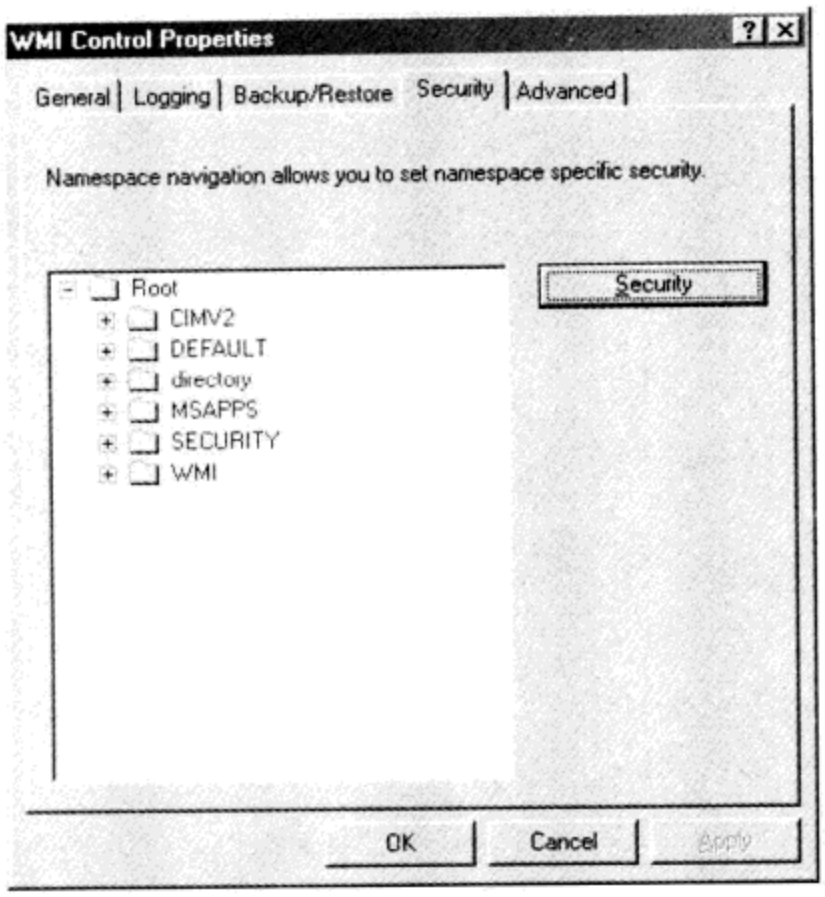


图4.22 WMI安全属性

4.4 本章总结

到现在为止，我们已经讨论了Windows的总体结构，以及该结构赖以建立起来的各种核心系统机制，还讨论了各种核心管理机制。有了这些基础作为准备，我们现在可以更加详细地讨论Windows的引导过程，以及各个单独的执行体组件。

www.TopSage.com

www.TopSage.com

Startup and Shutdown

第 5 章　启动和停机

在本章中，我们来讨论Microsoft Windows引导时必须执行的步骤，以及那些可能会影响到系统启动的选项。理解引导过程的细节，可以帮助你诊断在引导过程中出现的各种问题。然后，我们还将介绍在引导过程中可能出错的各种情况，以及如何消除这些错误。最后，我们将解释一下系统正常停机时所发生的事情。

5.1　引导过程

在讲述Windows引导过程时，我们将从Windows的安装开始，然后再介绍引导支持文件的执行情况。设备驱动程序是引导过程中的关键部分，因此，我们将描述驱动程序如何控制其在引导过程中的加载时间点和初始化时间点。然后，我们将介绍执行体子系统如何初始化，以及内核如何启动会话管理器进程（Smss.exe）、Windows子系统和登录进程（Winlogon），从而将Windows的用户模式部分激发起来。在此讲述过程中，我们将会重点点出那些在屏幕上出现各种文本提示的时间点，以帮助你将内部的引导过程与你所看到的现象联系起来。

引导过程的早期阶段对于x86和x64系统与IA64系统有重要的区别。接下来的部分首先讲述针对x86和x64引导过程的这些阶段，然后再用一小节讨论针对IA64引导过程的这些阶段。

x86和x64引导准备

Windows的引导过程并不是从你打开计算机的电源或者按下Reset按钮开始的，而是当你在自己的计算机上安装Windows的时候开始的。在Windows Setup程序执行过程中的某个点上，系统的主硬盘上已经准备好了一些代码，这些代码将会参与引导过程。在我们讨论这些代码的功能以前，先来看一看，Windows把这些代码放在硬盘上什么地方，以及如何放上去的。自从MS-DOS的早期以来，在x86系统上就存在一个将物理硬盘划分成卷的标准。Microsoft操作系统将硬盘分割成离散的区域，称为**分区**（*partition*），并且使用文件系统（比如FAT和NTFS）将每个分区格式化成一个卷。

www.TopSage.com

每个硬盘可以包含多达4个主分区。因为这种分配方案限制了一个硬盘至多只有四个卷，所以，一种称为**扩展分区**（*extended partition*）的特殊分区又可以在每个主分区内分配最多四个额外的分区。扩展分区可以包含扩展分区，后者还可以再包含扩展分区，以此类推，这使得操作系统实际上可以在一个硬盘上放置无限多个卷。图5.1显示了一个硬盘布局的示例，表5.1列出了在x86和x64引导过程中涉及的文件（后面第10章讲述了存储管理，在那里你可以学到更多有关Windows分区的知识）。

表5.1 x86和x64引导过程中涉及的组件

组 件	处理器的执行模式	职 责
主引导纪录（MBR，Master Boot Record）代码	16位实模式	读入和加载分区的引导扇区
引导扇区	16位实模式	读入根目录，以加载Ntldr
Ntldr	16位实模式，以及32位或64位保护模式；打开分页机制	读入Boot.ini，提示引导菜单，加载Ntoskrnl.exe、Bootvid.dll、Hal.dll和引导启动的设备驱动程序。如果引导的是一个32位系统，则切换到32位保护模式；如果引导的是一个64位系统，则切换到64位长模式
Ntdetect.com	16位实模式	为Ntldr执行硬件检测
Ntbootdd.sys	保护模式	对于不使用BIOS的SCSI和ATA系统（Advanced Technology Attachment），使用该组件作为磁盘I/O的设备驱动程序
Ntoskrnl.exe	支持分页的保护模式	初始化执行体子系统，并且初始化引导-启动的和系统-启动的设备驱动程序，为系统运行原生应用程序作好准备，并且运行Smss.exe
Hal.dll	支持分页的保护模式	内核模式DLL，作为Ntosknrl和驱动程序与硬件之间的接口
Smss	原生应用程序	加载Windows子系统，包括Win32k.sys和Csrss.exe，并且启动Winlogon进程
Winlogon	原生应用程序	启动服务控制管理器（SCM），启动本地安全子系统（LSASS），并且显示交互式登录对话框
服务控制管理器（SCM）	原生应用程序	加载并初始化那些自动启动的设备驱动程序和Windows服务

www.TopSage.com

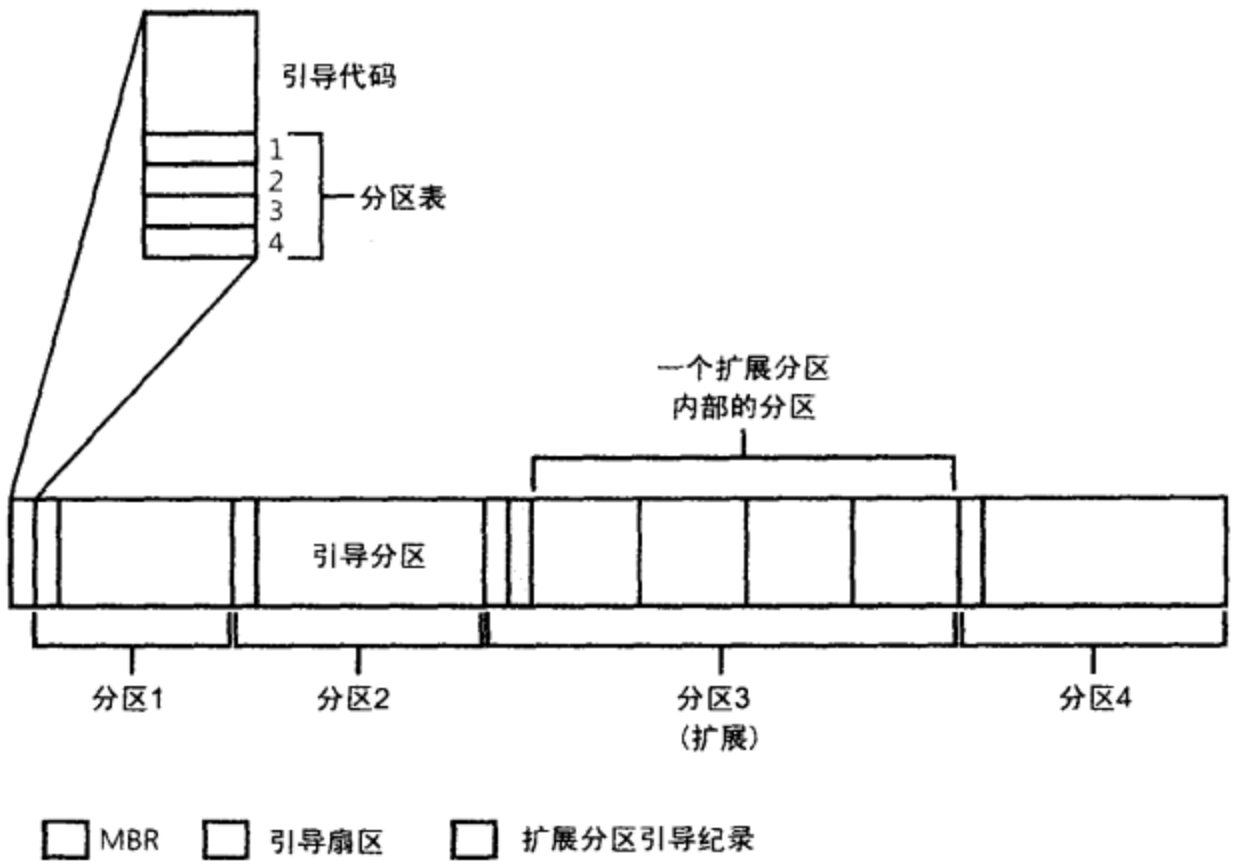


图5.1 硬盘布局示例

物理磁盘是按照一种称为**扇区**（*sector*）的单元来寻址的。在IBM兼容的PC上，一个硬盘扇区通常是512字节。为硬盘作好卷划分准备的实用工具，包括MS-DOS的Fdisk工具，或者Windows Setup程序，它们在硬盘的第一个扇区中写入一扇区数据，这些数据称为主引导纪录（MBR）（第10章将介绍MBR分区的知识）。MBR包含了固定数量的空间，其中有可执行的指令（称为**引导代码**）和一张表（称为**分区表**），该表中有四个表项，分别定义了该磁盘上主分区所在的位置。当一台IBM兼容的计算机引导时，它执行的第一段代码称为BIOS，这些代码被嵌在计算机的ROM中。BIOS选择一个引导设备，将该设备的MBR读入到内存中，然后将控制权交给MBR中的代码。

Microsoft的分区工具，比如Windows Setup中集成的分区工具，或者磁盘管理MMC加载件，它们写到磁盘上的MBR也经过一个类似于“先读取数据，再传递控制”的过程。首先，MBR的代码扫描主分区表，直至找到这样一个分区：它包含一个标记，指示该分区是可引导的。当MBR找到至少一个这样的标记时，它从对应的分区中读入第一个扇区到内存中，并且将控制权交给该分区的代码。这种分区称为**引导分区**（*boot partition*），它的第一个扇区称为**引导扇区**（*boot sector*）。针对引导分区而定义的卷称为**系统卷**（*system volume*）。

www.TopSage.com

操作系统通常在没有用户介入的情况下将引导扇区写到磁盘上。例如，当Windows Setup将MBR写到硬盘上的时候，它也会在该磁盘的第一个可引导分区中写入一个引导扇区。你可能在安装MS-DOS、Windows Me、Windows 98或者Windows 95的过程中，已经创建了一个MS-DOS引导扇区。Windows Setup在用一个Windows引导扇区来覆盖原来的引导扇区之前，会检查原来的引导扇区是否是一个有效的MS-DOS引导扇区。如果是的话，Windows Setup把该引导扇区的内容拷贝到一个名为Bootsect.dos的文件中，放在该分区的根目录下。

Windows Setup在写一个分区的引导扇区以前，用一种你所指定的文件系统类型来格式化此引导分区（以及任何其他的分区），以确保该分区已被格式化成一个Windows支持的文件系统（FAT、FAT32或者NTFS）。如果这些分区已经被格式化了，那么，你可以指示Setup程序跳过这一步骤。Setup程序将引导分区格式化以后，它把Windows用到的一些文件拷贝到该引导分区（系统卷）中，这些文件包括两个参与引导序列的文件：Ntldr和Ntdetect.com。

Setup程序的另一项任务是创建一个引导菜单文件Boot.ini，此文件也放在系统卷的根目录下。该文件包含了用于启动Windows各个版本的选项，包括当前Setup程序所安装的Windows版本，以及原先已经存在的Windows系统。如果Bootsect.dos包含一个有效的MS-DOS引导扇区，则Boot.ini中有一个项目是引导到MS-DOS下。下面的输出显示了一个例子Boot.ini文件，它来自于一台双引导的计算机，在Windows XP之前已经安装了一个MS-DOS：

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1) \WINDOWS="Microsoft Windows XP Professional" /fastdetect
C:\="Microsoft Windows"
```

你可以注意到，在这个例子文件中，指向Windows目录的路径是按照一种符合ARC（Advanced RISC Computing）命名规范的特殊语言来指定的。Windows使用的语法有三种形式。第一种是multi()语法，如上面的代码例子所示；multi()语法指示Windows使用BIOS INT 13功能来加载系统文件，因此，当引导卷所在的硬盘有一个支持INT 13的控制器时，multi()语法就会出现。multi语法遵从下面的格式：

multi(W)disk(X)rdisk(Y)partition(Z)

这里，W是磁盘控制器编号（也称为序号），通常是0。X在multi()语法中总是0。Y指定了附在控制器W上的具体物理硬盘。对于ATA控制器，该号码通常在0到3之间。对于SCSI控制器，该号码通常在0到15之间。Z指示了此引导卷在该物理磁盘上的分区号。第一个分区的号码为1。

www.TopSage.com

ARC语法scsi()告诉Windows，它应该依赖于Ntbootdd.sys（稍后介绍）所提供的磁盘I/O服务来访问引导卷上的文件。scsi()语法的格式如下：

scsi(W)disk(X)rdisk(Y)partition(Z)

在这种语法中，*W*是控制器编号。*X*是附在该控制器上的物理磁盘，通常在0到15之间。*Y*指定了引导卷所在磁盘的SCSI逻辑单元号（LUN），一般是0。最后，*Z*是对应于引导卷的分区号，编号从1开始。

Windows用到的最后一个语法是signature()语法。该语法指示Windows，无须考虑磁盘的控制器编号，直接找到“磁盘签名与括号中第一个值相匹配”的磁盘；并且也指示Windows使用Ntbootdd.sys来访问此引导卷。磁盘签名是一个全局惟一标识符（GUID），Windows Setup根据MBR中的信息生成该签名，并且将它写到磁盘上。Signature()语法如下所示：

signature(V)disk(X)rdisk(Y)partition(Z)

这里*V*是一个32位十六进制磁盘签名，它标识了该磁盘。*X*是具有特定签名的物理硬盘，它可以被附到该系统上的任何控制器上。*Y*总是0。*Z*是引导卷所在的分区号。

Windows在下列情况下使用signature()语法：

- 引导卷的大小超过了7.8GB，BIOS扩展的INT-13功能（用于访问一个磁盘超过7.8GB的部分）不能访问整个卷；
- BIOS不支持扩展的INT-13功能。

x86/x64引导扇区和Ntldr

Setup程序在写一个引导扇区以前，必须知道分区的格式，因为引导扇区的内容会随着分区格式的不同而不同。例如，如果引导分区是一个FAT分区，那么，Windows把能理解FAT文件系统的代码写到引导扇区中。但是，如果该分区是NTFS格式，那么，Windows写入能理解NTFS的代码。引导扇区代码的职责是，给Windows提供有关卷的结构和格式方面的信息，并且从该卷的根目录中读入Ntldr文件。因此，引导扇区代码包含了恰好足够的只读的文件系统代码，以完成这一任务。在引导扇区的代码将Ntldr加载到内存中以后，它把控制权交给Ntldr的入口点。如果引导扇区代码未能在该卷的根目录中找到Ntldr文件，那么，若当前正在引导的文件系统是FAT，则它显示错误消息“BOOT: Couldn't find NTLDRP”；若文件系统是NTFS，则显示“NTLDR is missing”。

www.TopSage.com

当一个系统还处于一种称为**实模式**（*real mode*）的x86操作模式下执行的时候，Ntldr就开始工作了。在实模式下，不需要从虚拟地址到物理地址的内存地址转译，这意味着，程序在使用内存地址时直接将它们解释成物理地址了；同时也意味着，它只能访问计算机物理内存的前1MB。简单的MS-DOS程序在实模式环境下运行。然而，Ntldr采取的第一个动作是，将系统切换到**保护模式**（*protected mode*）。在引导过程的这个点上，仍然不存在从虚拟地址到物理地址的地址转译，但是，这时候可以访问完整的32位内存。在系统进入了保护模式以后，Ntldr可以访问所有的物理内存。在创建了足够的页表使得16MB以下的内存通过分页机制都能访问以后，Ntldr打开分页机制。打开了分页机制的保护模式，正是Windows在正常情况下执行程序的模式。

Ntldr启动了分页机制以后，它便完全起作用了。然而，它仍然依赖于引导代码所提供的函数来访问基于IDE的系统，以及引导磁盘和显示器。引导代码函数暂时关闭分页机制，将处理器切换回到原来的模式，以便BIOS所提供的服务能够执行。如果包含引导卷的磁盘是SCSI硬盘，无法通过BIOS固件的支持来访问，那么，Ntldr加载一个名为Ntbootdd.sys的文件，用它代替引导代码函数来访问磁盘。Ntbootdd.sys是SCSI小端口驱动程序的一份拷贝，Windows在完全操作模式下用该驱动程序来访问引导盘（有关磁盘驱动程序的更多信息请参见第10章）。Ntldr接下来利用内置的文件系统代码，从根目录下读入Boot.ini文件。如同引导扇区的代码那样，Ntldr包含了只读的NTFS和FAT代码；然而，与引导扇区代码不同的是，Ntldr的文件系统代码可以读取子目录。

接下来Ntldr清除屏幕。如果在系统卷的根目录下存在一个有效的Hiberfil.sys文件，那么，它将该文件的内容读入到内存中，并且将控制权转交给一段能恢复一个休眠系统的内核代码，从而简化了引导过程。这一段代码负责重新启动那些在系统停机时仍然活动的驱动程序。只有当上次计算机停机时它是被休眠的，Hiberfil.sys才是有效的（有关休眠的信息，请参见第11章中的“电源管理器”一节）。

如果在Boot.ini中有多个引导选择项，那么，它向用户展示一个引导选择菜单（如果在Boot.ini中只有一个选择项，那么，Ntldr绕过此菜单，继续显示启动进度条）。Boot.ini中的选择项告诉Ntldr，被选中的Windows系统目录驻留在哪个分区上。这一分区可能与引导分区是相同的，也可能是另一个主分区。

如果Boot.ini中的项目指向一个MS-DOS系统（也就是说，引用C:\作为系统分区），那么，Ntldr将Bootsect.dos文件的内容读入到内存中，并且切换回到16位实模式，再调用Bootsect.dos中的MBR代码。这一动作使得Bootsect.dos得以执行起来，就好像MBR从磁盘中读入此代码一样。Bootsect.dos中的代码继续进行MS-DOS的引导过程，就好像在一台安装了Microsoft Windows Me、Windows 98或者Windows 95系统的计算机上引导这些操作系统所做的一样。

www.TopSage.com

Boot.ini中的项目可能包含一些可选的参数，Ntldr和其他一些在引导过程中涉及的组件会解释这些参数。表5.2包含了这些选项的一个完整列表，同时也列出了它们的作用。在Windows XP中引入的Bootcfg.exe工具，提供了一个便利的界面来设置各种开关。在Boot.ini中包含的所有选项都保存在注册表值HKLM\System\CurrentControlSet\Control\SystemStartOptions中。

表5.2 引导选项

引导修饰符	含 义
/3GB	增加用户进程地址空间的大小，从2GB增加到3GB（因此，也将系统空间的大小从2GB降低到1GB）。对于虚拟内存紧张的应用程序，比如数据库服务器，加大地址空间可以提高它们的性能。然而，为了让一个应用程序能够利用这一特性，必须满足两个额外的条件：该系统必须运行Windows XP、Windows Server 2003、Windows 2000 Advanced Server或者Datacenter Server；应用程序的.exe必须被标记成一个能理解3GB的应用程序（更多的信息，请参见第7章的“地址空间布局”一节）。（仅适用于32位系统）。
/BASEVIDEO	让Windows使用标准的VGA显示驱动程序来执行GUI模式的操作
/BAUDRATE=	打开内核模式调试开关，并指定一个波特率；远程内核调试器主机可以按照此波特率连接上来，默认的波特率为19200。此选项的一个例子是：/BAUDRATE=115200
/BOOTLOG	使Windows记录一份引导日志，位于文件%SystemRoot%\Ntbtlog.txt中
/BOOTLOGO	使用此开关可以让Windows XP或Windows Server 2003显示一个可安装的启动屏幕（splash screen），而不是标准的启动屏幕。首先，创建一个16色（任何16种颜色）的640x480位图，并且将它保存到Windows目录下名为Boot.bmp的文件中。然后将“/bootlogo /noguiboot”加到boot.ini的选择项上
/BREAK	使得硬件抽象层（HAL）在HAL初始化的时候停在一个断点处。当Windows内核初始化时它所做的第一件事情是初始化HAL，所以，这个断点是最早可能的断点。HAL将在该断点处无限等待下去，直到建立起一个内核调试器连接。如果此开关没有与/DEUG开关一起使用，那么，系统将会显示一个蓝屏，STOP代码为0x00000078（PHASE0_ EXCEPTION）
/BURNMEMORY=	指定Windows不能使用的内存数量（类似于/MAXMEM开关）。该值是按照兆字节为单位来指定的。例如，/BURNMEMORY=128指示Windows不能使用该机器上总物理内存中的128MB
/CHANNEL=	与/DEBUGPORT=1394联合使用，以指定IEEE 1394通道，内核调试通信数据将通过该通道来传递。这可以是0到62之间的任何数值，如果没有设置的话，默认为0

www.TopSage.com

续表

引导修饰符	含 义
/CLKLVL	使标准的x86多处理器HAL（Halmps.dll）针对一个电压控制式（level-sensitive）的系统时钟来配置它自己，而不是针对一个边界触发式（edge-triggered）的时钟来配置它自己。这里，电压控制式和边界触发式是两个用于描述硬件中断类型的术语
/CMDCONS	当引导进入到恢复控制台时跳过去（本章后面将会介绍恢复控制台）
/CRASHDEBUG	当系统引导时能将内核调试器加载进来，但除非系统崩溃，否则内核调试器不会被激活。这使得直到发生了系统崩溃，内核调试器才可以使用串行口（对比/DEBUG开关，它使得内核调试器在系统会话的生命期期间总可以使用串行口）
/DEBUG	打开内核模式调试开关
/DEBUGPORT=	打开内核模式调试功能，并且指定一个端口来覆盖默认的端口。远程内核调试器主机可以连接到此端口上(在至少两个串行端口的系统上通常默认使用COM2）。Windows XP和Windows Server 2003也支持通过IEEE 1394端口来调试。这样的例子有：/DEBUGPORT=COM2，/DEBUGPORT=1394
/EXECUTE	关闭“不可执行（no-execute）”保护机制。更多的信息，请参见/NOEXECUTE开关
/FASTDETECT	Windows的默认引导选项。替代了Windows NT 4的开关/NOSERIALMICE。该修饰符之所以存在的理由（相比较于默认仅仅让NTDETECT来执行检测操作）是，这样的话，NTDETECT可以支持引导Windows NT 4。Windows即插即用设备驱动程序负责并行和串行设备的检测工作，但是Windows NT 4期望NTDETECT来执行此检测工作。因此，指定了/FASTDETECT以后，NTDETECT就会跳过并行和串行设备的列举（在引导Windows的时候此操作并不是必需的）；而省略了此开关以后，NTDETECT就会执行这一列举操作（在引导Windows NT 4的时候，这一列举操作是必需的）
/INTAFFINITY	指示标准的x86多处理器HAL（Halmps.dll）设置中断亲和关系，这样只有最高编号的处理器才会接收中断。如果没有此开关，则HAL默认使用它的正常行为，即让所有的处理器都接收中断

www.TopSage.com

续表

引导修饰符	含 义
/KERNEL= /HAL=	让你可以覆盖掉Ntldr中使用的内核映像（Ntoskrnl.exe）和/或HAL（Hal.dll）的默认文件名。如果要在一个检查版本的内核环境和一个自由版本的(零售的)内核环境中进行切换，或者手工选择一个不同的HAL，那么，这些选项非常有用。如果你想要引导一个检查版本的环境，它只是由检查版本的内核和HAL构成的（为了测试驱动程序，这样做通常是很有必要的），那么，在一个安装了自由版本的系统上，你可以采取以下的步骤。 1. 从检查版本的CD上，将检查版本的内核映像拷贝到你的\Windows\System32目录下，给它起一个不同于默认名称的映像名。例如，如果你在一个单处理器系统上工作，那么，你可以将Ntoskrnl.exe拷贝成Ntoschk.exe，将Ntkrnlpa.exe拷贝成Ntoschkpa.exe。如果你在一个多处理器系统上工作，那么，你可以将Ntkrnlmp.exe拷贝成Ntoschk.exe，将Ntkrpamp.exe拷贝成Ntoschkpa.exe。内核文件名必须是一个8.3风格的短名称 2. 将你系统上所需要的检查版本HAL，从检查版本CD上的\I386\Driver.cab中，拷贝到你的\Windows\System32目录下，将它命名为Halchk.dll。为了确定该拷贝哪个HAL，打开\Windows\Repair\Setup.log，搜索Hal.dll，你将会找到像“\WINDOWS\system32\ hal.dll="halacpi.dll","1d8a1"”这样的一行。等号右侧的文件名正是你应该拷贝的HAL的名称。HAL文件名必须是一个8.3风格的短名称 3. 对系统的Boot.ini文件中的默认行做一份拷贝 4. 在引导选择项的字符串描述中，加入一些表明此新选择项是针对检查环境的信息(例如，“Windows XP Professional Checked”)。 5. 在新选择项行的末尾加入以下选项： /KERNEL=NTOSCHK.EXE /HAL= HALCHK.DLL 现在，当引导过程中出现选择菜单的时候，你可以选择新的项目以便引导检查版本的环境，或者选择原来的项目以引导自由版本的系统
/LASTKNOWNGOOD	使得系统引导时就好像LastKnownGood选项被选中了一样
/MAXMEM=	限制Windows忽略（不使用）超出限额的那部分物理内存。此限制值是以兆字节为单位的。例如，/MAXMEM=32将限制系统只使用物理内存的前32MB，即使实际的内存还有更多

www.TopSage.com

续表

引导修饰符	含 义
/MAXPROCSPERCLUSTER=	对于标准的x86多处理器HAL（Halmps.dll），强制使用集群模式的APIC（Advanced Programmable Interrupt Controller）编址方式（在带有82489DX外部APIC中断控制器的系统上不支持这种编址方式）
/MININT	该选项是Windows PE（Preinstallation Environment）使用的，它使得配置管理器将注册表的SYSTEM储巢加载成一个易失性的储巢，这样，在内存中对它所做的修改不会被保存回到储巢映像文件中
/NODEBUG	使得内核模式的调试功能不会被初始化。它会覆盖以下三个与调试有关的开关中的任何一个：/DEBUG、/DEBUGPORT和/BAUDRATE
/NOEXECUTE	只有当32位版本的Windows运行在AMD64处理器上，并且只有当PAE（将在/PAE开关的条目中进一步解释）也被打开的时候，这一选项才可以使用。它启动了“不可执行”保护模式，这导致内存管理器把包含数据的页面标记成不可执行的，从而它们不可能被当作代码来执行。这对于防止恶意代码发掘出缓冲区溢出错误，从而利用意外的程序输入来执行任意代码，将会非常有帮助。在AMD64处理器上运行64位版本的Windows时，“不可执行”保护方式总是打开的。有四个修饰符专门用于/NOEXECUTE开关：=OPTIN, =OPTOUT, =ALWAYSON, =ALWAYSOFF。关于这些修饰符的行为的介绍，请参见第7章
/NOGUIBOOT	指示Windows，专门负责在引导过程中显示位图图像的VGA视频驱动程序不必初始化了。该驱动程序被用于显示有关引导过程的信息，所以，禁止了该驱动程序以后，也等于禁止了Windows显示这些信息的能力
/NOLOWMEM	要求/PAE开关也打开，而且系统的物理内存超过4GB。如果这些条件满足的话，那么，支持PAE的Windows内核版本，Ntkrnlpa.exe，将不会使用前4GB物理内存。相反，它将会从4GB以上，加载所有的应用程序和驱动程序，以及分配所有的内存池。此开关仅仅对于测试设备驱动程序与大内存系统的兼容性才有用
/NOPAE	强迫Ntldr加载Windows内核的非PAE（Physical Address Extension，物理地址扩展）版本，即使该系统已被检测到它支持x86 PAE并且其物理内存超过4GB
/NOSERIALMICE=[COMx | COMx,y,z...]	已废弃的Windows NT 4修饰符，/FASTDETECT开关出现以后可以取代它。禁止对于指定COM端口的串行鼠标检测操作。如果在启动过程中，你没有附在串行口上的鼠标设备，那么此开关是很有用的。使用/NOSERIALMICE而不指定COM端口，就可以禁止在所有COM端口上的串行鼠标检测。有关更多的信息，请参见Microsoft Knowledge Base中的文章Q131976

www.TopSage.com

续表

引导修饰符	含 义
/NUMPROC=	在一个多处理器系统上，此选项指定了该系统可以使用的CPU的数目。例如，在一个四处理器系统上，“/NUMPROC=2”将阻止Windows使用这四个处理器中的两个
/ONECPU	在一个多处理器系统上，此开关使Windows只使用一个CPU
/PAE	让Ntldr加载Ntkrnlpa.exe，这是一个可充分利用x86 PAE能力的x86内核版本。PAE版本的内核提供给设备驱动程序的是64位物理地址，所以，要想测试设备驱动程序对于大内存系统的支持，该开关是非常有帮助的
/PCILOCK	阻止Windows给PCI设备动态分配IO/IRQ资源，将这些设备留给BIOS来配置。有关更多的信息，请参见Microsoft Knowledge Base中的文章Q148501
/RDPATH=	指定一个系统磁盘映像（SDI）文件的路径，它可以位于网络上，系统将使用该SDI文件来引导。此选项通常与“/RDIMAGEOFFSET=”标志一起使用，它们联合起来告诉NTLDR，系统映像位于一个文件中的什么地方
/REDIRECT	Windows Server 2003中引入的修饰符。它指示Windows打开紧急管理服务（EMS，Emergency Management Services），该服务可以报告引导信息，并且通过一个串行端口接受系统管理命令。通过Boot.ini文件的[boot loader]部分的“redirect=”和“redirectbaudrate=”文本行，可以指定与EMS一起使用的串行端口和波特率
/SAFEBOOT:	指定安全模式引导的选项。你应该永远也不需要手工指定此选项，因为当你用F8菜单来执行安全模式引导时，Ntldr会为你指定该选项（所谓安全模式引导，是指Windows只加载那些在HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot 下的Minimal或者Network注册表键中通过名称或者组来指定的驱动程序或者服务）。 在该选项中冒号的后面，你必须指定以下三个附加开关中的一个：MINIMAL、NETWORK或DSREPAIR。MINIMAL和NETWORK标志分别对应于无网络支持的安全模式引导，以及有网络支持的安全模式引导。DSREPAIR （Directory Services Repair）开关使得Windows引导到一种特殊的模式下。在该模式下，活动目录服务是离线的，而且，活动目录数据库也没有打开。这使得管理员可以对活动目录数据库执行诊断、修复或者恢复功能。你可以附加的一个选项是（ALTERNATESHELL），它告诉Windows，使用HKLM\SYSTEM\CurrentControlSet\SafeBoot\AlternateShell值中指定的程序，而不是默认的程序（即Windows Explorer）作为图形外壳程序

www.TopSage.com

续表

引导修饰符	含 义
/SCSIORDINAL:	将控制器的SCSI ID告诉Windows（在主板上带有SCSI控制器的系统中加入一个新的SCSI设备，可能会导致控制器的SCSI ID发生变化）。有关的更多信息，请参见Microsoft Knowledge Base中的文章Q103625
/SDIBOOT=	在Windows XP Embedded系统中，通过该选项，可以让Windows从一个存储在指定的系统磁盘映像（SDI）文件中的RAM磁盘映像中引导
/SOS	让Windows列出那些已被标记为要在引导时刻加载的设备驱动程序，然后显示系统版本号（包括编译版本号）、物理内存的数量，以及处理器的数目
/TIMERES=	在标准的x86多处理器HAL（Halmps.dll）上，该选项设置系统定时器的分辨率。此参数被解释成一个以百纳秒为单位的数值，但是，真正的分辨率被设置成最接近于HAL所支持的、且不超过所请求分辨率的值。HAL支持下列分辨率： 百纳秒 毫秒（ms） 97660.98 195322.00 390633.90 781257.80 默认的分辨率是7.8ms。系统定时器分辨率会影响到可等待定时器的分辨率。例如，/TIMERES=21000将该定时器设置成分辨率2.0ms
/USERVA=	只有在Windows XP和Windows Server 2003上才支持此开关。如同/3GB开关一样，此开关使得应用程序有更大的地址空间。指定2048和3072之间MB（兆字节）内存的数量。此开关与/3GB开关对应用程序有同样的要求，而且，它要求/3GB开关也要打开（仅适用于32位系统）
/WIN95	指示Ntldr：将存储在Bootsect.w40中的Consumer Windows引导扇区引导起来。此开关只有在一个三种引导的系统上才有意义，这三种引导是：MS-DOS、Consumer Windows和安装的Windows。有关的更多信息，请参见Microsoft Knowledge Base中的文章Q157992
/WIN95DOS	指示Ntldr：将存储在Bootsect.w40中的Consumer Windows引导扇区引导起来。此开关只有在一个三种引导的系统上才有意义，这三种引导是：MS-DOS、Consumer Windows和安装的Windows。有关的更多信息，请参见Microsoft Knowledge Base中的文章Q157992
/YEAR=	指示Windows的核心时间函数，忽略掉该计算机的实际时钟所报告的年份，改而使用此选项所标明的年份。因此，此选项中所使用的年份会影响到此系统上的每一个软件，包括Windows内核在内。用法举例：/YEAR=2001（建立此开关也是为了帮助Y2K测试）

www.TopSage.com

如果在Boot.ini文件中指定的超时时间到期以前，用户没有从引导项菜单中选择任何一个项目，那么，Ntldr选择默认的选择项，即boot.ini文件中项目路径与“default=”行中指定的路径最早匹配上的一个选择项。一旦引导选择项已经确定下来，Ntldr加载并执行Ntdetect.com，这是一个16位实模式程序，它使用系统的BIOS来查询该系统的基本设备和配置信息。这些信息包括：

- 存储在系统CMOS（非易失性内存）中的时间和日期信息；
- 系统中总线的类型（例如，ISA、PCI、EISA、MCA[Micro Channel Architecture，微通道结构]），以及附在这些总线上的设备的标识符；
- 系统中磁盘驱动器的个数、大小和类型；
- 连接到该系统上的鼠标输入设备的类型；
- 系统中配置的并行端口的数目和类型；
- 系统中出现的视频适配器的类型。

这些信息被收集到内部数据结构中，到引导过程的后期，它们被存放在HKLM\HARDWARE\DESCRIPTION注册表键的下面。

在Windows 2000上，Ntldr清除屏幕，并显示“Starting Windows”进度条。在Ntldr开始加载引导驱动程序以前，此进度条仍然是空的（参见下面列表中的第5步）。 在进度条的下面是一条信息“For troubleshooting and advanced startup options for Windows, press F8.”。如果用户按下F8，则出现高级引导菜单，这使得用户可以选择某个选项，以便从最后已知的好配置来引导系统，或者从安全模式来引导，或者从调试模式来引导，等等。在Windows XP和Windows Server 2003上，Ntldr显示一个标志性的启动屏幕（logo splash screen），而不是进度条。

如果Ntldr运行在一个x64系统上，并且在引导菜单中选中的项目所指定的内核是针对x64的，那么，Ntldr将处理器切换到**长模式**（*long mode*），在这种模式下，原生的字长是64位。接下来，Ntldr从**引导卷**（*boot volume*）中加载必要的文件，以便开始内核初始化。引导卷是指被引导的系统的系统目录（通常是\Windows）所在分区的那个卷。这里，Ntldr遵从的步骤如下。

1. 加载正确的内核和HAL映像（默认是Ntoskrnl.exe和Hal.dll）。如果Ntldr未能加载这些文件中的任何一个，则它打印出信息“Windows could not start because the following file was missing or corrupt”，后面跟上该文件的名称。
2. 读入SYSTEM注册表储巢，\Windows\System32\Config\System，所以，它可以确定哪些设备驱动程序必需被加载进来，才能完成这次引导（所谓储巢，是指包含了一棵注册表子树的文件。在第4章中你可以找到更多有关注册表的细节）。
3. 扫描内存中的SYSTEM注册表储巢，找到所有的**引导设备驱动程序**（*boot device driver*）。引导设备驱动程序是指为了引导该系统而必需的驱动程序。这些驱动程序在注册表中是通过用SERVICE_BOOT_START (0)作为启动（Start）值来标明的。每一个设备驱动程序在HKLM\SYSTEM\CurrentControlSet\Services下都有一个注册表子键。例如，在Services

www.TopSage.com

下有一个针对逻辑磁盘管理器驱动程序的子键，其名称是Dmio，正如你在图5.2中所看到的那样（有关Services注册表项目的详细说明，请参见第4章中“服务”一节）。

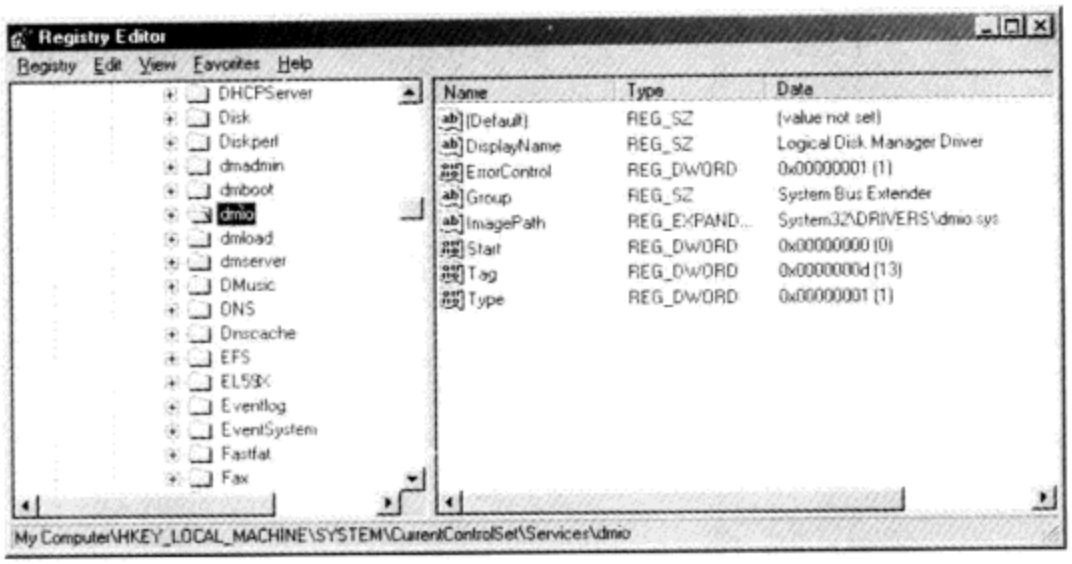


图5.2 逻辑磁盘管理器驱动程序的服务设置

4. 在将要加载的引导驱动程序的列表中，添加上为了访问系统目录而需要的文件系统驱动程序，该驱动程序实现了系统目录所在分区类型（FAT、FAT32或NTFS）的操作代码。Ntldr在这个时刻必须加载此驱动程序；如果它不这样做的话，内核将要求这些驱动程序加载它们自己，这一需求会引入循环相依性。
5. 加载引导驱动程序，应该只加载那些“像引导卷的文件系统驱动程序那样，若要求内核来加载它们就会引入循环相依性”的驱动程序。为了指示加载的进度，Ntldr对于文本“Starting Windows”下面的进度条进行更新。对于每个被加载的驱动程序，进度条都会向前移动（它假定总共有80个引导设备驱动程序——每一个成功加载以后进度条都会向前移动1.25%）。 如果在Boot.ini选择项中指定了/SOS开关，那么，Ntldr不显示进度条，而是显示每个引导驱动程序的文件名。要记住，在这个时候这些驱动程序只是被加载进来，但没有被初始化——在引导过程的后期它们再初始化。
6. 准备CPU寄存器，以便执行Ntoskrnl.exe。这一动作是Ntldr在引导过程中最后的任务。这时候，Ntldr调用Ntoskrnl.exe中的主函数，以便执行其余的系统初始化工作。

IA64引导过程

表5.3列出了在IA64引导过程中涉及的文件。IA64系统符合Intel定义的EFI（Extensible Firmware Interface，可扩展的固件接口）规范。一个EFI兼容的系统包含有可运行“引导加载器代码（boot loader code）”的固件，Windows Setup将这些引导加载器代码编进了系统的非易失性RAM（NVRAM）中。此引导代码读入对应于x86和x64 Boot.ini的IA64等价内容，它们也被存储在NVRAM中。无论是可在EFI控制台上运行的Microsoft EFI工具，还是Windows中包含的工具Bootcfg.exe，都可以修改NVRAM引导选择项和开关。

www.TopSage.com

表5.3 IA64引导过程组件

组 件	位 置	职 责
Fpswa.efi	在EFI系统分区上的EFI\Microsoft\Winnt50.x下	该文件包含了对于EFI浮点操作的支持
Ia64ldr.efi	在EFI系统分区上的EFI\Microsoft\Winnt50.x下	加载Ntoskrnl.exe、Hal.dll和引导驱动程序
Ntoskrnl.exe	\Windows\System32	初始化执行体子系统，以及引导-启动的和系统-启动的设备驱动程序；为系统运行原生的应用程序做好准备，以及运行Smss.exe
Hal.dll	\Windows\System32	内核模式的DLL，它将Ntoskrnl和驱动程序与硬件连接起来
服务控制管理器（SCM）	\Windows\System32	加载并初始化那些自动-启动的设备驱动程序和Windows服务
Smss	\Windows\System32	加载Windows子系统，包括Win32k.sys和Csrss.exe，并且启动Winlogon进程
Winlogon	\Windows\System32	启动服务控制管理器（SCM）、本地安全权威子系统（LSASS），并显示交互式的登录对话框

接下来是硬件检测，引导加载器使用EFI接口来确定以下设备的数目和类型：

- 网络适配器
- 视频适配器
- 键盘
- 磁盘控制器
- 存储设备

正如Ntldr在x86和x64系统上所做的那样，引导加载器然后显示一个引导选择菜单，它有一个可选的超时间隔。一旦确定了一个引导选择项，加载器转移到该选择项所对应的EFI系统分区上的子目录下，并且将其他一些为了继续引导而必需的文件加载进来，这些文件是：Fpswa.efi和Ia64ldr.efi。EFI规范要求：该系统有一个分区被指定成EFI系统分区，它被格式化成FAT文件系统，大小在100MB和1GB之间，或者可以达到磁盘总大小的百分之一，而且，每一个安装的Windows系统在EFI系统分区上的EFI\Microsoft下都有一个子目录。安装的第一个系统被分配到文件夹Winnt50，第二个是Winnt50.1，后续安装的每一个系统都有一个惟一的索引号跟在文件夹名的句点后面。Ia64ldr.efi负责加载Ntoskrnl.exe、Hal.dll，以及引导-启动的驱动程序，之后的引导过程完全遵从跟x86和x64系统一样的步骤。

www.TopSage.com

初始化内核和执行体子系统

Ntldr调用Ntoskrnl时，它传递的一个数据结构包含了Boot.ini中代表此次引导的菜单选择项所在的字符行，还包含三个指针，第一个指向那些由Ntldr生成的、用来描述系统中物理内存的内存表，第二个指向HARDWARE和SYSTEM注册表储巢在内存中的拷贝，第三个指向Ntldr加载的引导驱动程序列表。

然后，Ntoskrnl开始它的两阶段初始化过程中的第一个阶段。这两个阶段分别称为**阶段0**（*phase 0*）和**阶段1**（*phase 1*）。大多数执行体子系统都有一个初始化函数，该函数带有一个参数，标识了当前正在执行哪个阶段。

在阶段0过程中，中断被禁止了。此阶段的目的是，将阶段1中要用到的服务所需要的基本数据结构建立起来。Ntoskrnl的主函数调用*KiSystemStartup*，而*KiSystemStartup*又顺次为每一个CPU调用*HalInitializeProcessor*和*KiInitializeKernel*。如果*KiInitializeKernel*运行在引导CPU上，则它会执行系统范围全局的内核初始化，比如把所有CPU共享的内部链表头和其他数据结构都进行初始化。然后，*KiInitializeKernel*调用*ExpInitializeExecutive*函数，它负责阶段0的所有其他初始化工作。

*ExpInitializeExecutive*首先调用HAL函数*HalInitSystem*，这使得HAL在Windows执行进一步重要的初始化之前，有机会获得系统控制。*HalInitSystem*的一个任务是为每个CPU准备系统中断控制器，并配置时间间隔定时器中断；该定时器中断被用于CPU时间计量（关于CPU时间计量的更多信息，请参见第6章中“时限计算”一节）。

*ExpInitializeExecutive*只有在引导处理器上才执行除了调用*HalInitSystem*之外的其他初始化工作。*HalInitSystem*返回控制的时候，引导CPU上的*ExpInitializeExecutive*进一步处理Boot.ini的/BURNMEMORY开关（如果在用户选择了引导哪个系统以后，所选中的菜单选择项在Boot.ini中对应的行内包含了此开关的话），以及丢弃该开关指定的内存数量。

接下来，*ExpInitializeExecutive*为内存管理器、对象管理器、安全引用监视器、进程管理器和即插即用管理器，分别调用它们的阶段0初始化过程。这些组件执行以下的初始化步骤。

1. 内存管理器构建页表和内部数据结构，它们对于提供基本的内存服务是必需的。内存管理器也要为系统文件缓存建立并保留一个区域，同时也创建起换页的和非换页的内存池区域。其他的执行体子系统、内核和设备驱动程序使用这两个内存池来分配它们的数据结构。
2. 在对象管理器初始化期间，那些对于构建对象管理器名字空间必需的对象要先定义好，这样其他的子系统才可以在对象管理器名字空间中插入对象。在此期间，也要创建一个句柄表，从而可以开始资源跟踪。

www.TopSage.com

3. 安全引用监视器首先初始化令牌类型对象，然后使用该对象来创建并准备第一个本地系统账户令牌，并分配给初始进程（有关本地系统账户的介绍，请参见第8章）。
4. 进程管理器在阶段0执行绝大多数的初始化工作，它定义了进程和线程对象类型，并建立起相应的链表来跟踪活动的进程和线程。进程管理器也为初始进程创建一个进程对象，并且将它命名为“Idle”。作为其最后一个步骤，进程管理器创建System进程和一个系统线程，来执行Phase1Initialization例程。该线程并不立即启动运行，因为此时中断仍然是禁止的。
5. 然后是即插即用管理器的阶段0初始化过程，它只是简单地初始化一个执行体资源，该资源将被用于同步总线资源。

当每个处理器上的控制权回到*KiInitializeKernel*函数时，控制权向前进行到*Idle*循环中，然后，这使得在上面介绍的第4步中创建的系统线程开始执行阶段1（辅助处理器要等到阶段1的第5步才开始它们的初始化工作，如下面的列表所述）。 阶段1由下面的步骤组成。Windows 2000系统的引导启动屏幕包括一个进度条，而屏幕上进度条的更新步骤则包含在以下列表中。

1. 调用*HalInitSystem*，让系统做好准备接受来自设备的中断，并允许中断。
2. 调用引导视频驱动程序（\Windows\System32\Bootvid.dll），它进而显示Windows启动屏幕（在Windows XP和Windows Server 2003系统上，该驱动程序呈现的图像，与Ntldr在引导早期放在屏幕上的图像一样）。
3. 调用电源管理器的初始化过程。
4. 初始化系统时间（调用HalQueryRealTimeClock），然后存储起来作为系统引导的时间。
5. 在一个多处理器系统上，其余的处理器被初始化，并开始执行。
6. 设置进度条到5%。
7. 对象管理器创建名字空间根目录（\）、\ObjectTypes目录，以及DOS设备名映射目录（在Windows 2000上是\??，在Windows XP和Windows Server 2003上是\Global??），然后它创建\DosDevices符号链接，使其指向DOS设备名映射目录。
8. 调用执行体，以创建执行体对象类型，包括信号量、互斥体、事件和定时器。
9. 内核初始化调度器（分发器）数据结构和系统服务分发表。

www.TopSage.com

10. 安全引用监视器在对象管理器名字空间中创建\Security目录，如果审计功能被打开的话，也要初始化审计数据结构。
11. 设置进度条到10%。
12. 调用内存管理器，以创建内存区对象和内存管理器的系统辅助线程（第7章将会讲述）。
13. 将国家语言支持（NLS）表映射到系统空间中。
14. 将Ntdll.dll映射到系统地址空间中。
15. 缓存管理器初始化文件系统缓存数据结构，并创建其辅助线程。
16. 配置管理器在对象管理器名字空间中创建\Registry键对象，并且将Ntldr传递的初始注册表数据拷贝到HARDWARE和SYSTEM储巢中。
17. 初始化全局的文件系统驱动程序数据结构。
18. 即插即用管理器调用即插即用BIOS。
19. 设置进度条到20%。
20. 本地过程调用（LPC）子系统初始化LPC端口类型对象。
21. 如果此次系统引导带有引导日志开关（/BOOTLOG），则初始化引导日志文件。
22. 设置进度条到25%。
23. I/O管理器的初始化现在进行。这是系统启动过程中最为复杂的阶段，在进度条上占总体“进度”的50%。I/O管理器将每一个成功加载的驱动程序换算成引导进度的2%（如果要加载的驱动程序超过25个，则进度条在75%处停止）。I/O管理器首先初始化各种内部数据结构，并创建驱动程序对象类型和设备对象类型。接着它调用即插即用管理器、电源管理器和HAL，以便开始执行动态设备列举和初始化的各个步骤（因为这个过程非常复杂，并且与I/O系统尤其密切相关，所以，我们将细节留到第9章再介绍）。然后，Windows管理规范（WMI）子系统被初始化，该子系统为设备驱动程序提供了WMI支持（更多的信息，请参见第4章中的“Windows管理规范”一节）。 接下来，调用所有引导-启动的驱动程序，以便完成它们自己特有的初始化过程；然后加载系统-启动的驱动程序，并对它们进行初始化（关于注册表中驱动程序加载控制信息的处理细节，在第9章中介绍）。 最后，MS-DOS设备名称被创建起来，它们都是对象管理器名字空间中的符号链接。
24. 设置进度条到75%。

www.TopSage.com

25. 如果该计算机是在安全模式下被引导起来的，那么，在注册表中记录下这一事实。
26. 除非在注册表中显式地禁止了，否则内核模式代码（在Ntoskrnl和驱动程序中）的换页特性被打开。
27. 设置进度条到80%。
28. 调用电源管理器，以初始化各种与电源管理有关的数据结构。
29. 设置进度条到85%。
30. 调用安全引用监视器，以创建命令服务器线程（Command Server Thread），将来它与Lsass通信（关于Windows中安全性是如何强制实施的更多信息，请参见第8章中“安全系统组件”一节）。
31. 设置进度条到90%。
32. 最后一步是创建会话管理器子系统（Smss）进程（第2章中介绍过）。Smss负责创建用户模式环境，由用户模式环境向Windows提供可视的界面（Smss的初始化步骤将在下一节中介绍）。
33. 设置进度条到100%。

作为在执行体和内核初始化完成以前的最后一个步骤，阶段1初始化线程在会话管理器进程的句柄上等待，超时值为5秒。如果会话管理器进程在这5秒钟到期之前退出了，系统自身崩溃，错误检查代码为SESSION5_ INITIALIZATION_FAILED。

如果这5秒钟等待到时间了（也就是说，如果5秒钟到期了），则认为会话管理器已经成功启动了，于是，阶段1的初始化函数调用内存管理器的零页面线程函数（将在第7章中解释）。这样，在系统接下来的生命期内，此系统线程变成了零页面线程。

Smss、Csrss和Winlogon

Smss就好像是另外一个用户模式进程，只是有两个不同之处：第一，Windows将Smss看成是操作系统中可信的部分；第二，Smss是一个**原生**（*native*）应用程序。因为它是一个可信的操作系统组件，所以，Smss可以执行一些其他很少有进程能够执行的操作，比如创建安全令牌。因为它是一个原生应用程序，所以，Smss不使用Windows API——它只使用核心执行体API，这些API合起来称为Windows原生API。Smss不使用Windows API，是因为当Smss激发起来的时候，Windows子系统尚未执行。实际上，Smss首要的任务之一是启动Windows子系统。

然后，Smss调用配置管理器执行体子系统，来完成对注册表的初始化，充实注册表使其包含它所有的键。配置管理器的实现并不难：它知道那些核心的注册表储巢保存在磁盘上的什么地方（除了对应于用户轮廓的储巢以外），并且它在HKLM\SYSTEM\CurrentControlSet\Control\hivelist键中记录了它要加载的储巢的路径。

www.TopSage.com

Smss的主线程执行以下的初始化步骤。

1. 创建一个LPC端口对象（*\SmApiPort*），以及两个线程来等待客户的请求（比如请求加载一个新的子系统或者创建一个会话）。
2. 为MS-DOS设备名称（比如COM1和LPT1）定义符号链接。
3. 如果终端服务已被安装的话，则在对象管理器的名字空间中创建\Sessions目录（为了多会话支持）。
4. 运行在HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute中定义的任何程序。在通常情况下，这个值包含了一条命令，它运行Autochk（Chkdsk的引导时刻版本）。
5. 根据HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations和HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations2的指示，执行延迟的文件重命名和删除操作。
6. 打开已知的DLL，为它们在对象管理器名字空间中的\Knowndlls目录下创建内存区对象。这些被认为已知的DLL的列表，位于HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs中，这些DLL所在目录的路径被保存在此键的Dlldirectory值中。关于“这些已知DLL的内存区是如何在DLL加载过程中被使用的”更多信息，请参见第6章。
7. 创建附加的页面文件。页面文件的配置信息保存在HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\Paging Files下。
8. 初始化注册表。配置管理器加载HKLM\SAM、HKLM\SECURITY和HKLM\SOFTWARE键的注册表储巢，以填充注册表的内容。尽管HKLM\SYSTEM\CurrentControlSet\Control\hivelist给出了这些储巢文件在磁盘上的位置，但是，配置管理器被实现为：在\Windows\System32\Config下查找这些文件。
9. 将HKLM\System\CurrentControlSet\Session Manager\Environment中定义的系统环境变量建立起来。
10. 加载Windows子系统的内核模式部分（Win32k.sys）。Smss在HKLM\SYSTEM\CurrentControlSet\Control\Session Manager下可以找到Win32k.sys和它要加载的其他组件的路径。Win32k.sys中的初始化代码利用已加载的视频驱动程序，将屏幕切换到默认轮廓所定义的分辨率，所以，正是在这个时候，屏幕从引导视频驱动程序所使用的VGA模式，改变成该系统所选择的默认分辨率。
11. 启动子系统进程，包括Csrss（正如在第2章中所说明的那样，在Windows 2000上，POSIX和OS/2子系统被定义成按需启动）。

www.TopSage.com

12. 启动登录进程（Winlogon）。关于Winlogon的启动步骤稍后会讲述。
13. 创建LPC端口（Dbg*SsApiPort*和*DbgUiApiPort*），这些端口用于传递调试事件消息。创建相应线程在这些端口上监听。

未完成的文件重命名操作

可执行映像和DLL文件在被使用的时候都是内存映射的，这一事实使得在Windows结束了引导过程以后，系统不可能再更新核心的系统文件。Windows API函数MoveFileEx有一个选项，它允许指定将一个文件移动操作延迟至下一次引导时再执行。如果系统的服务补丁包（Service Pack）和热补丁（hotfix）必须更新正在使用中的内存映射文件，则它就会将替换文件安装到系统中的临时位置上，然后让MoveFileEx API替换这些当前正在使用中的文件。当MoveFileEx与该选项一起使用的时候，它只是简单地将这些命令记录在HKLM\SYSTEM\CurrentControlSet\Control\Session Manager下的PendingFileRenameOperations和PendingFileRenameOperations2值中。这些注册表值的类型都是MULTI_SZ，其中每个操作都按照一对文件名的方式来指定：第一个文件名是源位置，第二个是目标位置。对于删除操作，其目标路径是空字符串。你可以利用从*www.sysinternals.com*下载的Pendmoves工具，查看到已经注册的延迟的重命名和删除命令。

在执行了这些初始化步骤以后，Smss中的主线程进入无限等待，它等待Csrss和Winlogon的进程句柄。如果这两个进程中的任何一个非正常终止了，Smss就会让系统崩溃，因为Windows依赖于这两个进程的存在（在Windows XP及以后的版本中，如果Csrss出于任何一个理由退出了，那么，是内核使系统崩溃，而不是Smss）。

然后，Winlogon执行它的启动步骤，比如创建初始的窗口站和桌面对象。如果在HKLM\Software\Microsoft\Windows NT\Current Version\WinLogon\GinaDLL中指定了一个DLL，那么，Winlogon使用该DLL作为GINA；否则，它使用Microsoft的默认GINA，即Msgina（\Windows\System32\Msgina.dll），此默认GINA会显示标准的Windows登录对话框。然后，Winlogon创建服务控制管理器（SCM）进程（\Windows\System32\Services.exe），而SCM进程又依次加载所有被标记为自动-启动的服务和设备驱动程序，以及本地安全认证子系统（Lsass）进程（\Windows\System32\Lsass.exe）（有关Winlogon和Lsass的启动序列的更多细节，请参见第8章中的“Winlogon初始化”一节）。

在SCM初始化了自动-启动的服务和驱动程序，以及用户成功地登录到控制台上以后，SCM认为这次引导成功了。注册表中的最后已知的好控制集（如HKLM\SYSTEM\Select\LastKnownGood所指示）被更新成与\CurrentControlSet一致。

www.TopSage.com



> 注 因为非交互式的服务器可能永远也没有交互式的登录，所以，它们可能无法使LastKnownGood 得到更新，以反映出一次成功引导所用到的控制集。
> 你可以改变关于“成功引导”的定义，做法是，将HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\ReportBootOk设置为0；并且编写一个自定义的引导验证程序，当引导成功时该程序调用Windows API函数NotifyBootConfigStatus；然后在HKLM\System\CurrentControlSet\Control\BootVerificationProgram中输入该验证程序的路径。

Winlogon激发了SCM以后，它便等待来自GINA的交互式登录通知。它接收到一个登录，并且验证该登录确实有效（有关此验证过程的更多信息，请参见第8章中的“用户登录步骤”一节）以后，Winlogon从登录用户的轮廓中加载注册表储巢，并且将它映射到HKCU。然后，它设置该用户的环境变量（它们存储在HKCU\Environment中），并且告诉那些在HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify中登记过的Winlogon通知包：当前发生了一次登录过程。

接下来Winlogon告诉GINA启动外壳程序。作为对该请求的响应，Msgina将HKLM\Software\Microsoft\Windows NT\CurrentVersion\WinLogon\Userinit中指定的一个或多个可执行程序激发起来（用逗号来分割多个可执行程序）。在默认情况下该注册表值指向\Windows\System32\Userinit.exe。Userinit.exe执行以下步骤。

1. 处理HKCU\Software\Policies\Microsoft\Windows\System\Scripts中指定的用户脚本，以及HKLM\Software\Policies\Microsoft\Windows\System\Scripts中的机器登录脚本（因为机器脚本是在用户脚本之后运行的，所以它们可以覆盖用户的设置）。
2. 如果组策略指定了一个用户轮廓配额，则启动\Windows\System32\Proquota.exe，以强制应用当前用户的配额。
3. 将HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell中指定的用逗号分割的一个或多个外壳程序激发起来。如果此注册表值不存在，则Userinit.exe将HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell中指定的一个或多个外壳程序激发起来，该注册表值的默认值是Explorer.exe。

然后，Winlogon通知已登记的网络提供者，有一个用户登录进来了。Microsoft的网络提供者，多提供者转发器（Multiple Provider Router, \Windows\System32\Mpr.dll）将HKCU\Network和HKCU\Printers中存储的永久驱动器字母和打印机映射关系分别恢复起来。图5.3显示了在登录过程中，Useinit退出以前在进程管理器中看到的进程树。

www.TopSage.com

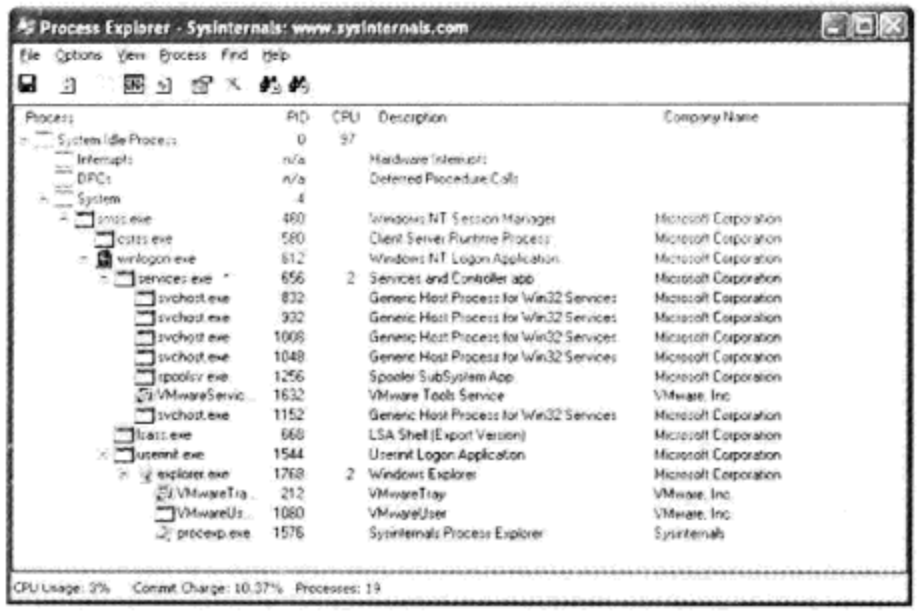


图5.3 登录过程中的进程树

自动启动的映像文件

除了Winlogon键中的Userinit和Shell注册表值以外，在引导和登录过程中，默认的系统组件还会检查许多其他的注册表位置和目录，以处理自动的进程启动事项。Msconfig工具（包含在Windows XP和Windows Server 2003中的\Windows\System32\Msconfig.exe中）会显示出在多个位置上配置的映像文件。你可以从*www.sysinternals.com*下载到Sysinternals的Autoruns工具，如图5.4所示。它检查的位置比Msconfig更多一些，并且它将已被配置成自动运行的映像文件的更多信息显示出来。在默认情况下，Autoruns只显示那些已被配置成自动执行至少一个映像文件的位置，但是，如果选中了View菜单中的Include Empty Locations菜单项，则Autoruns会显示出所有它检查过的位置。View菜单还有一些选项，可让Autoruns显示有关其他类型的自动启动映像文件的信息，比如Windows服务和Explorer加载件。

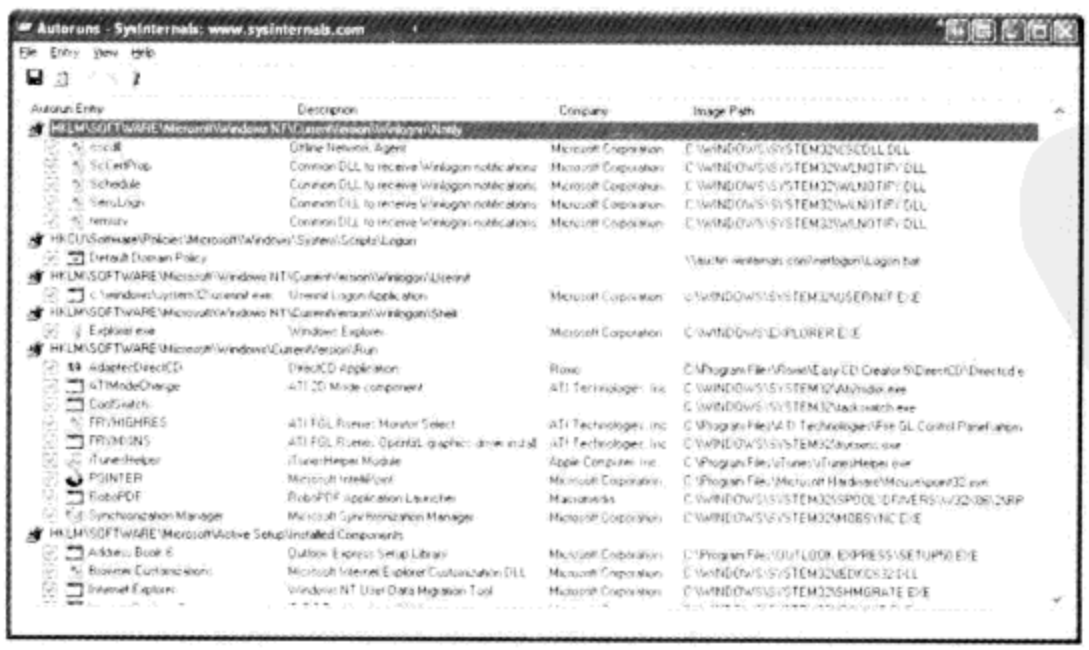


图5.4 可从www.sysinternals.com处下载的Autoruns工具

www.TopSage.com



实验：Autoruns

许多用户并不知道在他们登录过程中有多少程序执行起来了。原始设备制造商（OEM，original equipment manufacturers）通常用一些加载件工具来配置他们的系统，他们利用在自动执行处理中涉及的注册表值或者文件系统目录，在后台执行这些加载件工具，所以这些加载件通常是不可见的。你运行一下*www.sysinternals.com*的Autoruns工具，就可以看到在你的机器上哪些程序已被配置成自动启动了。比较一下Autoruns中显示的列表与Msconfig（在Windows XP和Windows Server 2003中可以使用）中显示的列表，将两者的区别标识出来。然后，确保你理解了每一个程序的用途。

5.2 引导和启动问题的故障检查

这一节展示了一些方法，这些方法能解决在Windows启动过程中可能发生的一些问题，这些问题可能是由于硬盘破坏、文件破坏、缺失文件以及第三方驱动程序错误造成的。首先，我们介绍针对Windows引导问题的三种恢复模式：最后已知的好配置、安全模式和恢复控制台。然后，我们展示一些常见的引导问题及其产生的原因，以及解决这些问题的办法。这些解决方案会涉及最后已知的好配置、安全模式、恢复控制台，以及其他一些随Windows带的工具。

最后已知的好配置

最后已知的好配置（LPK）是一种很有用的机制，通过这种机制，你可以将一个在引导过程中崩溃的系统恢复到原来可引导的状态。因为系统的配置设置信息被保存在HKLM\System\CurrentControlSet\Control中，而驱动程序和服务的配置信息被保存在HKLM\System\CurrentControlSet\Services中，所以，在注册表中对这些部分的改变可以导致一个系统无法引导。例如，如果你安装了一个有错误的设备驱动程序，其错误可以导致系统在引导过程中崩溃，那么，你可以在引导过程中按下F8键，从所得到的菜单中选择最后已知的好配置。系统通过设置HKLM\System\Select的Failed值，将用于引导该系统的控制集标记为已失败，然后，将HKLM\System\Select\Current改变成HKLM\System\Select\LastKnownGood中保存的值。它同时也会更新符号链接HKLM\System\CurrentControlSet，让它指向LastKnownGood控制集。因为新驱动程序的键没有出现在LastKnownGood控制集的Services子键中，所以，系统将可以成功引导。

安全模式

Windows系统之所以变成不可引导的最常见的理由可能是，在引导过程中有一个设备驱动程序导致机器崩溃了。因为软件或硬件的配置可以随着时间而改变，驱动程序中隐藏的错误可能会在任何时候暴露出来。Windows为管理员提供了一种方法来处理这种问题：在**安全模式**下引导系统。安全模式是Windows从Consumer Windows中借用来的一个概念，它指的是，由最小

www.TopSage.com

集合的设备驱动程序和服务构成的一种引导配置。通过仅仅依赖于那些对于引导过程真正必要的驱动程序和服务，Windows避免了加载那些可能会崩溃的第三方和其他非关键的驱动程序。

当Windows引导的时候，你按下F8键，进入到一个特殊的引导菜单中，该菜单中包含了安全模式的引导选项。通常你可以从安全模式的三种变化模式中进行选择：Safe Mode（安全模式）、Safe Mode With Networking（支持网络的安全模式）和Safe Mode With Command Prompt（支持命令提示的安全模式）。标准的安全模式是由对于引导成功真正有必要的最小数量的设备驱动程序和服务组成的。支持网络的安全模式是在标准安全模式所包含的驱动程序和服务的基础上，加上了网络驱动程序和服务。最后，支持命令提示的安全模式几乎等同于标准的安全模式，只不过当Windows系统启动GUI模式时，它运行命令提示应用程序（Cmd.exe）作为外壳程序，而不是Windows Explorer。

Windows也包含了第四种安全模式——目录服务恢复模式，它不同于标准的安全模式和支持网络的安全模式。你可以使用目录服务恢复模式来引导系统进入一种特殊的运行模式，在这种模式中，域控制器的活动目录服务是离线的，目录服务没有被打开。这使得你可以针对目录服务数据库执行一些修复操作，或者从备份介质中恢复该数据库。在目录服务恢复模式的引导过程中，除了活动目录服务以外的所有其他驱动程序和服务都被加载到系统中。当你的系统由于活动目录数据库的破坏而无法登录的时候，这种模式可以让你修复损坏的数据库。

安全模式下的驱动程序加载

Windows如何知道哪些设备驱动程序和服务属于标准安全模式或者支持网络的安全模式呢？答案就在HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot注册表键中。该键包含了Minimal和Network子键。每个子键又包含了更多的子键，由这些子键来指定设备驱动程序或服务或驱动程序组的名称。例如，vga.sys子键标识了在启动配置中包含的VGA显示设备驱动程序。VGA显示驱动程序为任何一块PC兼容的显示适配器提供了最基本的图形服务。系统使用该驱动程序作为安全模式的显示驱动程序，来代替系统中实际安装的、虽然可能利用了适配器的高级硬件特性但是也可能会导致系统无法引导的那个驱动程序。SafeBoot键下的每个子键都有一个默认值描述了该子键代表的是什么，例如vga.sys子键的默认值是“Driver”。

Boot File System子键的默认值是“Driver Group”。当开发人员设计一个设备驱动程序的安装脚本时，他们可以指定该设备驱动程序属于一个驱动程序组。系统定义的驱动程序组位于HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder键的List值中。开发人员指定一个驱动程序作为某个组的成员，这样做是告诉Windows，在引导过程的某个点上该驱动程序应该被启动起来。ServiceGroupOrder键的主要用途是定义驱动程序组的加载顺序。有些驱动程序类型必须在其他驱动程序类型之前或者之后加载。一个驱动程序的配置注册表键下面的Group值将该驱动程序与一个组关联起来。

www.TopSage.com

驱动程序和服务配置键位于HKLM\SYSTEM\CurrentControlSet\Services的下面。如果查看该键，你将会看到针对VGA显示设备驱动程序的VgaSave键。同时你也可以看到，它是Video Save组的一个成员。Windows访问系统分区所需要的任何一个文件系统驱动程序都在Boot File System组中。如果系统分区是NTFS，则NTFS驱动程序是该组的成员（在Ntfs键下的Group值是Boot File System）。否则，Fastfat文件系统驱动程序（在Windows中，它支持FAT12、FAT16和FAT32分区）属于该组。其他的文件系统驱动程序都属于File System组，标准安全模式和支持网络的安全模式的配置中也包含了File System组。

当你引导到某一种安全模式配置中时，引导加载器（Ntldr）通过命令行参数的形式，将一个相应的模式开关传递给内核（Ntoskrnl.exe），同时也把Boot.ini文件中针对当前正在引导的选择项所指定的任何开关一起传递给它。如果你引导到任何一种安全模式中，则Ntldr传递/SAFEBOOT:开关。Ntldr在/SAFEBOOT:后面附加上一个或者多个额外的字符串，到底是什么字符串要取决于你所选中的安全模式的类型。对于标准安全模式，Ntldr附加上MINIMAL，而对于支持网络的安全模式，它加上NETWORK。对于支持命令提示的安全模式，Ntldr加上MINIMAL（ALTERNATESHELL）；对于目录服务恢复模式，则加上DSREPAIR。

Windows内核在引导过程的早期，扫描引导参数，以寻找安全模式开关；并且将内部变量*InitSafeBootMode*设置成一个可反映出它所找到的开关的值。内核将*InitSafeBootMode*的值写到注册表值HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue中，所以，像SCM这样的用户模式组件可以知道系统当前处于什么引导模式下。此外，如果系统是按照命令提示安全模式来引导的，则内核将HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell值设置为1。内核把Ntldr传递给它的参数记录到HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions中。

当I/O管理器内核子系统加载HKLM\SYSTEM\CurrentControlSet\Services中指定的设备驱动程序时，I/O管理器执行函数*IopLoadDriver*。当即插即用管理器检测到一个新的设备，并且想要为检测到的设备动态加载设备驱动程序时，即插即用管理器执行函数*IopCallDriverAddDevice*。这两个函数在加载相应的驱动程序以前，都会调用函数*IopSafeBootDriverLoad*。*IopSafeBootDriverLoad*检查*InitSafeBootMode*的值，确定是否应该加载该驱动程序。例如，如果系统是按照标准安全模式来引导的，那么，若该驱动程序有组的话，*IopSafeBootDriverLoad*在Minimal子键下面查找它的组。如果*IopSafeBootDriverLoad*发现该驱动程序的组被列在Minimal子键的下面，那么，*IopSafeBootDriverLoad*告诉它的调用者，该驱动程序可以加载。否则，*IopSafeBootDriverLoad*在Minimal子键下面查找该驱动程序的名称。如果该驱动程序的名称被当作一个子键列在那里，则该驱动程序可以加载。如果*IopSafeBootDriverLoad*不能找到驱动程序组或者驱动程序名称的子键，则该驱动程序不能加载。如果系统是按照网络安全模式来引导的，则*IopSafeBootDriverLoad*在Network子键上执行这些搜索操作。如果系统不是在安全模式下引导的，则*IopSafeBootDriverLoad*让所有的驱动程序都加载进来。

www.TopSage.com

关于在安全模式引导过程中要排除掉的驱动程序，存在一个例外，即那些由Ntldr而不是内核来加载的、在注册表中Start值为0的驱动程序（该Start值指定了在引导时加载这些驱动程序）。Ntldr并不检查SafeBoot注册表键，因为它假定，所有这些Start值为0的驱动程序对于系统的成功引导总是必需的。因为Ntldr并不检查SafeBoot注册表键以识别哪些驱动程序应该加载，所以，Ntldr加载所有引导-启动的驱动程序（以后Ntoskrnl再启动它们）。

能感知安全模式的用户程序

当服务控制管理器（SCM）用户模式组件（由Services.exe实现）在引导过程中进行初始化的时候，SCM根据HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\OptionValue的值，来确定系统当前是否正在执行安全模式的引导。如果是的话，SCM也执行一遍*IopSafeBootDriverLoad*的动作。虽然SCM处理的是HKLM\SYSTEM\CurrentControlSet\Services下面列出的服务，但是，它只加载那些在对应的安全模式子键下面用名称来指定的服务。关于SCM初始化过程，你可以在第4章的“服务”一节中找到更多的信息。

Userinit（\Windows\System32\Userinit.exe）是另一个需要知道当前系统是否在安全模式下引导的用户模式组件。当一个用户登录的时候，Userinit初始化该用户的环境；它检查HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Option\UseAlternateShell。如果该值被设置的话，则Userinit运行HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\AlternateShell值中指定的程序（作为该用户的外壳程序），而不是执行Explorer.exe。Windows在安装过程中，在AlternateShell值中写入程序名Cmd.exe，这使得Windows命令提示程序成为支持命令提示的安全模式下的默认外壳。即使命令提示程序作为系统的外壳，你仍然可以在命令提示符下输入Explorer.exe，将Windows Explorer启动起来，你也可以从命令提示符下运行任何其他的GUI程序。

那么，一个应用程序如何判断当前系统是在安全模式下引导的呢？通过调用Windows的*GetSystemMetrics（SM_CLEANBOOT）*函数即可。如果当系统是在安全模式下引导时，有些脚本程序需要执行某些特定的操作，那么，它可以查找SAFEBOOT_OPTION环境变量，因为只有当系统是在安全模式下引导的时候它才会定义此环境变量。

安全模式下的引导日志

当你指示系统在安全模式下引导时，Ntldr将/BOOTLOG选项中指定的字符串以参数的形式，连同另一个请求安全模式的参数，一起传递给Windows内核。当内核初始化的时候，它检查是否出现了引导日志参数，以及是否出现了任何的安全模式参数。如果内核检测到了一个引导日志字符串，那么，内核记录下它在每一个考虑要加载的设备驱动程序上所执行的动作。

www.TopSage.com

例如，如果*IopSafeBootDriverLoad*告诉I/O管理器不要加载某个驱动程序，那么，I/O管理器调用*IopBootLog*，记录下该驱动程序未被加载。同样地，在*IopLoadDriver*成功地加载了某一个属于安全模式配置一部分的驱动程序以后，*IopLoadDriver*调用*IopBootLog*，记录下该驱动程序已被加载。你可以检查引导日志，以查看哪些设备驱动程序是某一特定引导配置的一部分。

因为内核想要避免在执行Chkdsk以前修改磁盘，所以，在引导过程的后期，*IopBootLog*不能简单地将消息转储到一个日志文件中。相反，*IopBootLog*将消息记录在HKLM\SYSTEM\CurrentControlSet\BootLog注册表值中。作为引导过程中第一个被加载的用户模式组件，会话管理器（\Windows\System32\Smss.exe）执行Chkdsk，以确保系统分区的一致性，然后执行*NtInitializeRegistry*系统调用，以完成注册表的初始化。内核将这一动作看作一个提示，表明它可以安全地打开磁盘上的一个日志文件，所以它调用函数*IopCopyBootLogRegistryToFile*。该函数在Windows的系统目录（默认为\Windows）中创建Ntbtlog.txt文件，并且将BootLog注册表值中的内容拷贝到该文件中。*IopCopyBootLogRegistryToFile*也为*IopBootLog*设置了一个标志，通过该标志让*IopBootLog*知道，现在可以直接写到日志文件中，而不用将消息记录在注册表中。下面的输出显示了一个简单的引导日志中的部分内容。

```
Service Pack 1 3 30 2004 14:05:21.500
Loaded driver \WINDOWS\system32\ntoskrnl.exe
Loaded driver \WINDOWS\system32\hal.dll
Loaded driver \WINDOWS\system32\KDCOM.DLL
Loaded driver \WINDOWS\system32\BOOTVID.dll
Loaded driver ACPI.sys
Loaded driver \WINDOWS\System32\DRIVERS\WMILIB.SYS
Loaded driver pci.sys
Loaded driver isapnp.sys
Loaded driver intelide.sys
Loaded driver \WINDOWS\System32\DRIVERS\PCIIDEX.SYS
Loaded driver MountMgr.sys
Loaded driver ftdisk.sys
Loaded driver dmload.sys
Loaded driver dmio.sys Microsoft (R) Windows 2000 (R) Version 5.0 (Build 2195)
 2 11 2000 10:53:27.500
Loaded driver \WINNT\System32\ntoskrnl.exe
Loaded driver \WINNT\System32\hal.dll
Loaded driver \WINNT\System32\BOOTVID.DLL
Loaded driver ACPI.sys
Loaded driver \WINNT\System32\DRIVERS\WMILIB.SYS
Loaded driver pci.sys
Loaded driver isapnp.sys
Loaded driver compbatt.sys
Loaded driver \WINNT\System32\DRIVERS\BATTC.SYS
Loaded driver intelide.sys
Loaded driver \WINNT\System32\DRIVERS\PCIIDEX.SYS
Loaded driver pcmcia.sys
Loaded driver ftdisk.sys
Loaded driver Diskperf.sys
Loaded driver dmload.sys
Loaded driver dmio.sys
```


www.TopSage.com

```
§
Did not load driver \SystemRoot\System32\Drivers\lbrtfdc.SYS
Did not load driver \SystemRoot\System32\Drivers\sfloppy.SYS
Did not load driver \SystemRoot\System32\Drivers\i2omgmt.SYSDid not load driver Media Contr
ol Devices
Did not load driver Communications Port
Did not load driver Audio Codecs
§
```

恢复控制台（Recovery Console）

对于那些由于某个设备驱动程序在引导过程中崩溃而变得无法引导的系统来说，安全模式是一种非常令人满意的回退措施，但是在某些情况下，安全模式的引导方式并不能帮助一个系统引导起来。例如，如果妨碍系统引导的驱动程序是Safe组的某个成员，那么，安全模式的引导也会失败。安全模式无法帮助系统引导的另一个例子是，一个第三方的驱动程序，比如病毒扫描器驱动程序，一旦在引导时加载就会导致系统无法引导（不管系统是否在安全模式下，引导-启动的驱动程序都会加载）。 安全模式引导失败的其他一些情形有：当某个属于安全模式配置一部分的系统模块或者关键的设备驱动程序文件被破坏时，或者当系统驱动器的主引导纪录（MBR）被损坏时。利用Windows的恢复控制台，你可以解决这些问题。恢复控制台允许你利用Windows CD或者引导磁盘，将系统引导到一个受限的命令行外壳中，以便在未启动已安装的系统的情况下修复该系统。

当你从Windows CD或者引导磁盘来引导一个系统的时候，你最终会看到一个屏幕，让你选择安装Windows或者修复一个已安装的系统。如果你选择修复一个已安装的系统，那么，系统提示你插入Windows CD（如果它还没有被装入到系统的CD驱动器中），然后在两个修复选项中进行选择：启动恢复控制台，或者开始紧急修复过程。如果你在安装欢迎屏幕处按下F10，那么，你可以绕过菜单选项，直接进入恢复控制台阶段。

当你启动了恢复控制台时，它给出一个Windows NT和Windows已安装版本的列表，这是它在扫描该计算机的硬盘时搜集得到的版本，并且它让你从中进行选择。在你做出了选择以后，系统提示你输入Administrator账户的口令，以便让你以管理员的身份登录到你所选择的系统中。如果你成功登录了，则系统让你进入到一个类似于MS-DOS的环境的命令外壳中。该外壳的命令集是相当灵活的，它可以让你执行一些简单的文件操作（比如拷贝、重命名和删除），允许或者禁止Windows服务和驱动程序，甚至可以修复MBR和引导纪录。然而，恢复控制台不允许你访问除了以下这些目录之外的其他目录：根目录、你登录进去的系统目录，或者诸如CD和3.5英寸软盘之类的可去除驱动器上的目录。除非你登录进去的系统的注册表的SECURITY储巢中存储的本地安全策略设置允许你访问其他的目录，否则你只能访问上述的这些目录。这种限制对于那些在通常情况下管理员可能无法访问的数据来说，提供了一层特殊的安全性。你可以在系统正常引导起来的时候，利用本地安全策略编辑器（secpol.msc），在本地策略（Local Policies）的安全选项（Security Options）夹中配置恢复控制台的设置，从而改变上述限制。

www.TopSage.com

恢复控制台利用Windows原生的系统调用接口来执行文件I/O，以支持诸如*cd*、*rename*和*move*这样的命令。*Enable*和*Disable*命令让你改变设备驱动程序和服务的启动模式，它们的工作方式有所不同。例如，当你告诉恢复控制台，你想要禁止一个设备驱动程序时，它进入到当前系统的Services键中，直接操作该驱动程序键的Start值，将此值改变成SERVICE_DISABLED。当同样的系统下一次引导时，此设备驱动程序就不再被加载进来（恢复控制台也为你登录的系统加载SYSTEM储巢[\Windows\System32\Config\System]。这一储巢包含了HKLM\SYSTEM\CurrentControlSet\Services注册表键中存储的信息）。

当你从Windows CD或者引导磁盘上引导一个系统时，在系统提示你选择安装Windows或者修复Windows以前，该CD已经引导了Windows内核的一份拷贝，包括所有必须要支持的设备驱动程序（例如，NTFS或者FAT驱动程序、SCSI驱动程序、视频驱动程序）。在x86系统上，Windows CD的I386目录中的Txtsetup.sif文件指示了如何从该CD来引导；该文件中包含的指示符标识了哪些文件需要加载，以及这些要加载的文件位于CD上的什么地方。正如你从一个硬盘来引导Windows一样，内核执行的第一个用户模式程序是会话管理器（Smss.exe），位于I386\System32文件夹中。Windows Setup使用的会话管理器不同于标准安装的Windows中的会话管理器。前者向你展示两个菜单，第一个菜单让你选择安装或者修复Windows，第二个菜单问你想要执行哪种类型的修复。如果你正在安装Windows，那么，会话管理器指导你选择哪个分区来安装系统，以及将文件拷贝到硬盘上。

当你运行恢复控制台时，会话管理器加载并启动两个设备驱动程序，它们实现了恢复控制台。这两个驱动程序分别是Spcmdcon.sys和Setupdd.sys。Spcmdcon.sys显示一个交互式命令提示窗口，并执行高层次的命令处理工作。Setupdd.sys是一个支持驱动程序，它为Spcmdcon.sys提供了一组函数，这些函数使得Spcmdcon.sys可以管理磁盘分区、加载注册表储巢，以及显示和管理视频输出。Setupdd.sys也与磁盘驱动程序进行通信，以便管理磁盘分区；它也利用Windows内核中内置的基本视频支持，以便在屏幕上显示消息。

当你选择一个已安装的系统来登录，并且恢复控制台接受了你的口令时，恢复控制台必须要验证你的登录企图，尽管该系统的Windows安全子系统尚未起来运行。因此，恢复控制台必须自己来确定你的口令是否与系统的Administrator账户相匹配。在这个过程中，恢复控制台的第一个步骤是使用Setupdd.sys，从磁盘上加载当前所选系统的安全账户管理器（SAM，Security Accounts Manager）注册表储巢，该储巢中保存了口令信息。SAM储巢驻留在\Windows\System32\Config\Sam中。在加载了该储巢以后，恢复控制台在该系统的注册表中找到系统密钥，再用系统密钥来解密内存中的SAM拷贝。SAM储巢是加密的，这是Windows NT 4的ServicePack 3中引入的一项特性；对于那些试图通过MS-DOS直接从储巢文件中读取口令的口令窃贼来说，这项特性可以增加一层保护。

www.TopSage.com

接下来，恢复控制台（Spcmdcon.sys）在SAM中找到Administrator账户的口令。在最后的认证步骤中，恢复控制台首先使用MD5散列算法（Windows的登录过程也使用了该算法），计算出用户输入的口令的散列值，然后将该值与SAM保存的口令散列值作比较。如果恢复控制台发现两者匹配，则系统认为你已经登录了；如果恢复控制台发现两者不匹配，则系统拒绝你访问恢复控制台。

解决常见的引导问题

这一节将讲述在引导过程中可能发生的问题，这里描述了它们的症状、原因和解决方案。为了帮助你找到一个你可能会遇到的问题，本节根据这些问题在引导时所处的位置来组织它们。

MBR损坏

- **症状** 一个主引导纪录（MBR）已损坏的系统，它将执行BIOS的开机自检（POST），显示BIOS的版本信息或者OEM品牌，切换到黑屏，然后挂起。根据MBR的损坏类型，你可能看到以下这些消息中的某一个：“Invalid Partition Table”、“Error Loading Operating System”或“Missing Operating System”。
- **原因** MBR之所以会损坏，可能是因为在Windows运行过程中，由于某个驱动程序的错误而导致硬盘错误、磁盘损坏，也可能由于病毒的刻意不规则访问而导致磁盘损坏。
- **解决方案** 引导到恢复控制台中，执行*fixmbr*命令。该命令会替换MBR中的可执行代码。不幸的是，它不修复分区表。要想恢复一个破损的分区表，惟一的办法是，根据它的备份拷贝来恢复它，或者使用第三方的磁盘损坏修复工具。

引导扇区损坏

- **症状** 引导扇区损坏可能看起来也像MBR损坏一样：系统在BIOS POST以后的黑屏上挂起了，或者你可能在黑屏上看到显示的消息“A disk read error occurred”、“ NTLDR is missing”或“NTLDR is compressed”。
- **原因** 引导扇区之所以会损坏，可能是因为在Windows运行过程中，由于某个驱动程序的错误而导致硬盘错误、磁盘损坏，也可能由于病毒的刻意不规则访问而导致磁盘损坏。
- **解决方案** 引导到恢复控制台中，执行*fixmbr*命令。该命令会重写你所指定的卷的引导扇区。如果系统卷和引导卷不同的话，你就应该同时在这两个卷上执行该命令。

www.TopSage.com

Boot.ini的错误配置

- **症状** 在BIOS POST以后，你会看到以下的开头消息：“Windows could not start because of a computer disk hardware configuration problem”、“ Could not read from selected boot disk”或者“Check boot path and disk hardware”。
- **原因** Boot.ini文件已被删除、已被破坏，或者由于新加入的一个分区改变了该卷的ARC（Advanced RISC Computing）名，从而Boot.ini不再指向引导卷。
- **解决方案** 引导到恢复控制台中，执行“bootcfg /rebuild”命令。该命令让恢复控制台扫描每个卷，以查找所有已安装的Windows系统。当它发现一个系统时，就会询问你，是否将它加到Boot.ini中作为一个引导选项，以及在引导菜单中应该为该系统显示什么名称。

系统文件损坏

- **症状** 系统文件（包括可执行文件、驱动程序或者DLL）的损坏可以有多种表现形式。一种形式是，在BIOS POST之后的黑屏上显示一条消息，内容是“Windows could not start because the following file is missing or corrupt”，后面跟着一个文件的名称，以及一个指示重新安装该文件的请求。另一种形式是，在引导过程中出现蓝屏崩溃现象，显示的文字如“STOP: 0xC0000135 {Unable to Locate Component}”。
- **原因** 系统文件所在的卷已被损坏，或者一个或多个系统文件已被删除，或已被破坏。
- **解决方案** 引导到恢复控制台中，执行*chkdsk*命令。Chkdsk将会试图修复被损坏的卷。如果Chkdsk没有报告任何问题，则获取当前受损系统文件的一份备份拷贝。一个检查的地方是\Windows\System32\DllCache目录，Windows把许多系统文件放在该目录中，以便于Windows的文件保护机制能够访问（参见稍后的“Windows文件保护”辅助内容）。 如果你在那里不能找到受损文件的拷贝，则你可以看一看是否能从网络上的另一个系统中找到一份拷贝。注意，此备份文件必须是从同样的服务补丁包（Service Pack）或者热补丁的系统上拷贝而来的。

在有些情况下，多个系统文件被删除，或已被破坏，所以，整个修复过程可能会牵涉到多次重新引导，在你逐个修复这些文件的过程中，可能会有多次引导失败。如果你相信系统文件的损坏涉及面很广的话，你应该考虑从一个备份映像来恢复该系统，比如从ASR（Automated System Recovery，自动的系统恢复）生成的系统映像来恢复当前系统。当你运行Windows Backup（你在Start菜单中Accessories下的System夹中可以找到）时，你可以生成一个ASR备份映像，其中包含了当前系统和引导卷上所有的文件，同时在一张软盘上保存了有关该系统的磁盘和卷的信息。为了从ASR恢复一个系统，请从Windows的安装介质中以备份方式来引导系统，当出现提示时按下F2键。

如果你没有一个可用于恢复的备份，那么，最后的办法是，从Windows的安装介质中执行

www.TopSage.com

一次Windows修复安装的引导，并且遵从向导的提示，就好像你在执行一次新的安装一样。该向导将询问你，你是想执行修复，还是执行全新的安装。当你告诉它，你想要修复的时候，Setup重新安装所有的系统文件，而保留你的应用数据和注册表设置原封不动。

Windows文件保护

Winlogon除了作为交互式的登录界面和会话管理器以外，它也实现了Windows文件保护机制（WFP，Windows File Protection）。WFP是在两个DLL中实现的，它们是\Windows\System32\Sfc.dll和\Windows\System32\Sfc_os.dll。WFP使用*ReadDirectoryChangesW*的原生API版本来监视几个目录，其中包括\Windows下的绝大多数子目录，它监视关键的驱动程序、可执行文件和DLL是否有变化。当WFP看到\Windows\System32\Sfcfiles.Dll中列出的系统文件（你使用*www.sysinternals.com*的Strings工具，就可以看到Sfcfiles.dll中列出的文件）有变化发生时，它会检查该文件，看是否经过Microsoft的数字签名（关于此签名过程的更多信息，请参见第9章的“驱动程序安装”一节）。如果该文件是由Microsoft数字签名的，则WFP允许此变化的发生，并且将该文件拷贝到WFP备份目录中。在默认情况下，该备份目录是\Windows\System32\DllCache，不过，你可以通过定义HKEY_LOCAL_MACHINE\ Software\Microsoft\Windows NT\ CurrentVersion\ Winlogon\SFCDllCacheDir注册表值，来改变此备份目录。热补丁和服务补丁包总是安装由Microsoft签名的系统文件。

如果对文件的修改并没有导致产生一个未被Microsoft签名的文件，那么，WFP从DLLCache子目录中用该文件的备份版本来替换这次修改。如果Winlogon不能在该目录中找到一个备份版本，而且，若当前系统是使用网络安装版本来安装的，则它会检查网络安装路径；若系统是从本地介质来安装的，则它会检查本地安装介质（提示插入）。

System储巢损坏

- **症状** 如果System注册表储巢（在第4章的“注册表”一节中讨论过储巢文件）被遗失或者被破坏，则NTLDR将在BIOS POST之后的黑屏上显示消息“Windows could not start because the following file is missing or corrupt: \WINDOWS\SYSTEM32\CONFIG\ SYSTEM”；
- **原因** System注册表储巢已经被破坏，或者已被删除。该储巢中包含了用于系统引导所必需的配置信息；
- **解决方案** 引导到恢复控制台中，在引导卷上执行*chkdsk*命令，以修复任何卷错误。如果该文件没有被改正，则设法获得System注册表储巢的一份备份。如果你已经对系统做了ASR备份，或者已经使用Windows的备份工具对系统状态做了备份（这是备份UI中的一个选项），则最近一次备份中的注册表储巢的备份被保存在\Windows\Repair下，所以，你可以将名为System的文件拷贝到\Windows\System32\Config中。

www.TopSage.com

如果你正在运行Windows XP，并且已经启动了系统恢复（System Restore）机制（在第12章中讨论系统恢复机制），那么，你通常可以从最近的恢复点处，得到更近期的一些注册表储巢备份，其中包括System储巢。然而，你可能无法在恢复控制台中访问这些恢复点所在的目录：\System Volume Information。Windows XP Service Pack 1版本的恢复控制台允许访问此目录，但是早一些的版本不允许访问此目录，除非系统的本地安全策略允许这么做。如果有必要的话，你可以使用本地安全策略编辑器来改变恢复控制台的设置（见前面的讲述），从而改变这一限制。你也可以使用第三方的工具来获得对其他目录的访问权。如果你可以访问恢复点所在的目录，那么，你可以通过下面的步骤来获得它们的注册表储巢。

1. 在引导卷的\System Volume Information目录下，转换到以“_restore”为名称开头的目录下。
2. 找到后缀编号最大的RP子目录，比如“RP173”。
3. 将文件_REGISTRY_MACHINE_SYSTEM从该快照子目录拷贝为\Windows\System32\Config\System。
4. 重新引导。

另一种选择是，试着使用Microsoft的ChkReg工具来修复损坏的注册表。该工具试图自动修复损坏的注册表，它是通过利用Windows XP安装软盘来工作的。在以下页面上你可以找到该工具，以及它的用法指示：*http://www.microsoft.com/downloads/details.aspx?displaylang=en&familyid=56d3c201-2c68-4de8-9229-ca494362419c*。

如果你没有做过备份，也访问不了恢复点，而且ChkReg工具也不能修复损坏的注册表（或者储巢已遗失），那么，你可以使用\Windows\Repair中保存的System储巢备份进行最后的尝试。Windows Setup在完成了安装以后，对System储巢做了这份拷贝，所以，从那以后的系统配置变化以及设备驱动程序的安装信息，都会丢失。

启动屏幕之后的崩溃或者挂起

- **症状** 在Windows的启动屏幕显示出来以后，或者桌面显示出来以后，或者你登录以后发生的问题都属于这一类问题，它们可能表现为蓝屏崩溃，也可能表现为系统挂起，即整个系统被冻住了，或者鼠标光标虽然还跟着鼠标在跑但系统却不再反应。
- **原因** 这一类问题几乎总是由于某个设备驱动程序中的软件错误造成的，但有时候，它们也可能是由于某个注册表储巢（除了System储巢以外）的损坏而引起的。
- **解决方案** 你可以通过几个步骤来试着修正该问题。你应该尝试的第一件事情是最后已知的好配置。本章的前面，以及第4章的“服务”一节，已经介绍过最后已知的好配置（LKG），它是由最后一次成功引导系统时所用到的注册表控制集构成的。因为一个控制集包含了核心系统配置，以及设备驱动程序和服务的注册数据库，所以，使用一个不反映最新变化或者最新安装的驱动程序或服务的控制集版本，可能会避免问题的根源。你在引导过程的早

www.TopSage.com

期按下F8键，通过显示出来的菜单（跟你在引导安全模式时看到的是同一个菜单）就可以访问最后已知的好配置。

正如本章前面所说明过的，当你引导进入LKG时，系统把你要避免使用的控制集保存起来，并且将它标记为**失败的**（*failed*）控制集。若LKG能让系统引导起来，那么，在这样的情况下，你可以利用这一失败的控制集来确定导致系统无法引导的原因，做法是，把当前成功引导的控制集的内容和失败控制集的内容都导出到.reg文件中。利用Regedit的导出功能你就可以做到这一点。通过Regedit的File菜单（如果你运行Windows 2000，则位于Registry菜单中），你可以使用它的导出功能：

1. 运行Regedit，选择HKLM\System\CurrentControlSet；
2. 从File菜单中选择Export，保存到文件good.reg中；
3. 打开HKLM\System\Select，读入Failed的值，然后选择名为HKLM\System\Control*XXX*的子键，这里*XXX*是Failed的值；
4. 将该控制集的内容导出到bad.reg中；
5. 使用Wordpad（可在Start菜单中的Accessories下面找到该程序），将good.reg中所有的“CurrentControlSet”实例全部替换为“ControlSet”；
6. 使用Wordpad，将bad.reg中所有的“Control*XXX*”（用失败控制集的值来替换这里的*XXX*）实例全部改变为“ControlSet”；
7. 运行Windows支持工具箱（Support Tools）中的Windiff，并比较这两个文件。

失败控制集和好控制集之间的差异可能会非常多，所以，你应该将注意力集中在Control子键，以及Services子键中已注册的驱动程序和服务的Parameters子键下面。而控制集的Services分支中的驱动程序注册表键的Enum子键中的任何变化，都可以忽略不管。

如果你所经历的问题是由于某一个在最后一次成功引导之前就已经安装在系统中的驱动程序或者服务引起的，那么，LKG将无法使系统引导起来。类似地，如果有问题的系统配置中的修改发生在控制集之外，或者是在最后一次成功引导之前发生的，那么，LKG将不会有帮助。在这些情况下，下一个尝试的方案是安全模式（本节前面讲述过）。如果该系统在安全模式下能成功地引导起来，而且你知道哪个驱动程序使得正常引导无法成功，那么，你可以通过设备管理器（Device Manager，利用控制面板的System小程序中的Hardware属性页可以访问到）来禁止此驱动程序。要做到这一点，你先选中有问题的驱动程序，然后从Action菜单中选择Disable。如果你正在运行Windows XP和Windows Server 2003，你最近更新过该驱动程序，而且你相信这次更新引入了一个错误，那么，你可以选择将该驱动程序回滚到它的前一个版本，这样，你仍然可以在设备管理器中完成。为了将一个驱动程序恢复到它的前一个版本，你可以在驱动程序上双击，打开它的属性对话框，然后在Drivers属性页上按下Roll Back Driver。

在支持系统恢复（System Restore）的Windows XP系统上，当LKG失败时的一种可选办法

www.TopSage.com

是，将所有的系统状态（由系统恢复机制定义）恢复到上一个时间点。安全模式可以检测到恢复点的存在与否，当恢复点存在的时候，它会问你，是要登录到该系统中执行手工的诊断和修复，还是要激发系统恢复向导（System Restore Wizard）。当你知道了问题的根源并且想让修复过程自动化，或者并不知道问题的根源但不想花时间找出问题原因的时候，利用系统恢复机制来让一个系统重新引导起来，这种做法是非常有吸引力的。

如果系统恢复机制不是一种可选择的方案，或者你想要知道一个系统为什么在正常引导时会崩溃，而在安全模式下却可以成功引导，那么，你可以按下F8以便访问专门的引导菜单，并选择引导日志记录选项，这样可以获得一份不成功引导的日志纪录。正如本章前面所讲述的，会话管理器（\Windows\System32\Smss.exe）在\Windows\ntbtlog.txt中保存了当前引导的一份日志，其中记录了系统加载的设备驱动程序，以及选择不加载的设备驱动程序，所以，如果系统崩溃或者挂起是在会话管理器初始化之后发生的，那么你可以得到一份引导日志。当你重新引导进入到安全模式中时，系统将新的项目追加到已有的引导日志中。将日志文件中分成两部分，一部分包含了失败引导的纪录，另一部分包含了安全模式引导的内容，并且将它们分别保存到不同的文件中。然后将文件中包含有“Did not load driver”的文本行除掉，再用一个文本比较工具，比如Windiff，对两个文件进行比较。之后，逐个禁止那些在正常引导过程中被加载的，而在安全模式下未被加载的驱动程序，直到系统再次成功引导起来为止（然后，再重新启动那些不用对本问题负责的驱动程序）。

如果你无法获得正常引导过程的日志（比如说，因为系统是在会话管理器初始化之前崩溃的），或者，如果系统在安全模式下也会崩溃，或者，如果正常引导和安全模式下的引导日志的比较并没有表现出严重的差异（例如，导致正常模式引导崩溃的驱动程序是在会话管理器初始化之后启动的），那么，接下来尝试的工具是，驱动程序检验器（Driver Verifier）结合对崩溃转储信息的分析（关于这两个话题的更多信息，请参见第14章）。

5.3 停机

如果已经有人登录到系统中，并且某个进程通过调用Windows的*ExitWindowsEx*函数发出了停机命令，那么，就会有一个消息被发送给Csrss，指示它执行停机处理。Csrss接着模仿调用者的身份，给Winlogon的一个隐藏窗口发送一个Windows消息，告诉它执行系统停机命令。然后Winlogon模仿当前登录用户的身份（该用户与发出系统停机命令的用户可能有同样的安全环境，也可能有不同的安全环境），并且用一些特殊的内部标志来调用*ExitWindowsEx*。再一次，此调用导致有一个消息被发送给Csrss，请求执行系统停机命令。

这一次，Csrss看到该请求来自Winlogon，它按照进程的**停机级别**（*shutdown level*）的反序，来经历当前交互式用户的登录会话中的所有进程。每个进程通过调用*SetProcessShutdownParameters*，可以指定一个停机级别，从而向系统表明：它相对于其他的进程而言希望什么时候退出。有效的停机级别位于0至1023之间，默认的级别是640。例如，Explorer（即默认的系统外壳程序）设置它的停机级别为2，而任务管理器指定的停机级别是1。对于每个拥有顶级窗口的进程，Csrss给该进程中每个包含Windows消息循环的线程发送一个

www.TopSage.com

WM_QUERYENDSESSION消息。如果线程返回TRUE，则系统停机过程继续进行。然后，Csrss给该线程发送Windows消息WM_ENDSESSION，请求它退出。Csrss等待一定的毫秒数值，让该线程退出。此毫秒数值被定义在HKCU\Control Panel\Desktop\HungAppTimeout中，默认为5000毫秒。

如果该线程在超时到期之前还没有退出，则Csrss显示一个终止程序对话框，如图5.5所示（你也可以禁止该对话框，做法是，将注册表值HKCU\Control Panel\Desktop\AutoEndTasks改成1）。此对话框表明了一个程序未能及时地停止，并且提供给用户一个选择：是杀死该进程，还是取消停机（该对话框上没有超时限制，这说明一个停机请求可能会在这个点上永久地等待下去）。

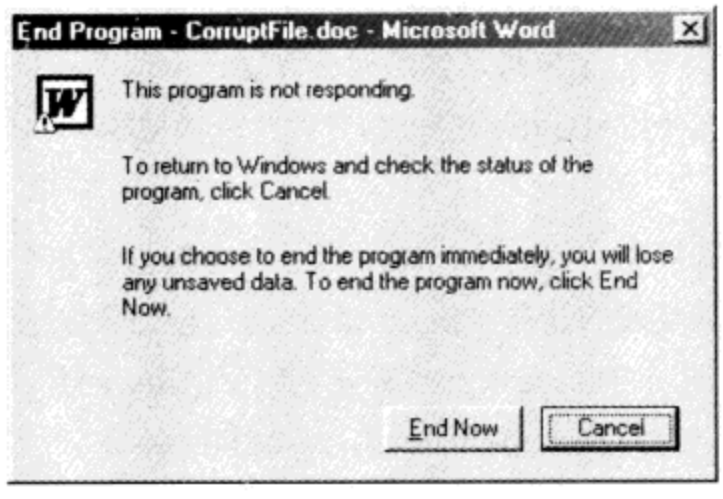


图5.5 终止程序对话框

如果一个线程在超时到期之前退出了，Csrss继续给该进程中其他拥有窗口的线程发送WM_QUERYENDSESSION/WM_ENDSESSION消息。一旦该进程中所有拥有窗口的线程都已经退出了，则Csrss终止该进程，并转向当前交互式会话中的下一个进程。



实验：验证HungAppTimeout值

你只需先运行Notepad，在它的编辑器中输入一点文本，然后注销登录，就可以看到HungAppTimeout注册表值的用法。在HungAppTimeout注册表值中指定的时间长度过去以后，Csrss就会出示一个对话框，询问你是否想要终止Notepad进程。该进程之所以未能及时退出的原因是，它在等你告诉它是否将输入的文本保存到一个文件中。如果你在对话框中按下Cancel按钮，则Csrss.exe取消停机。

如果Csrss看到一个控制台应用程序，那么，它通过发送CTRL_LOGOFF_EVENT事件，来调用控制台的控制处理器（control handler）（只有服务进程才会在停机时接收CTRL_SHUTDOWN_EVENT事件）。如果控制处理器返回FALSE，则Csrss杀死该进程。如果控制处理器返回TRUE，或者在HKCU\Control Panel\Desktop\WaitToKillAppTimeout注册表值指定的毫秒数值（默认为20 000毫秒）内没有响应，则Csrss也会显示如图5.5所示的终止程序对话框。

www.TopSage.com

接下来，Winlogon调用*ExitWindowsEx*，让Csrss终止任何一个属于交互式用户会话一部分的COM进程。

到这时候，交互式用户的会话中所有的进程都已经被终止了。Winlogon再次调用*ExitWindowsEx*，但这一次调用是在系统进程的环境中进行的，*ExitWindowsEx*再次给Csrss发送一个消息，Csrss检查所有属于系统环境的进程，并且给GUI线程发送WM_QUERYENDSESSION/WM_ENDSESSION消息（跟前面一样）。然而，它给已经登记了控制处理器的控制台应用程序发送的不是CTRL_LOGOFF_EVENT消息，而是CTRL_SHUTDOWN_EVENT。请注意，SCM是一个控制台应用程序，而且它确实登记了一个控制处理器。当它接收到停机请求时，它给所有已经登记了停机通知的服务发送一个服务停机控制消息。有关服务停机的更多细节（比如Csrss针对SCM使用的停机超时），请参见第4章中的“服务”一节。

虽然Csrss执行了同样的超时处理，就如同它在终止用户进程时一样，但是它并不显示任何对话框，也不杀死任何进程（针对系统进程超时的注册表值是从默认用户轮廓中取出来的）。这些超时值只不过是让系统进程在系统停机以前有机会做一些清理工作再退出。因此，实际上当系统停机时，许多系统进程仍然在运行，比如Smss、Winlogon、SCM和Lsass。

一旦Csrss已经完成了向系统进程传达系统停机的通知，Winlogon最后调用执行体子系统函数*NtShutdownSystem*，从而结束停机过程。*NtShutdownSystem*函数又调用*NtSetSystemPowerState*函数，来协调设备驱动程序和执行体子系统其余部分（即插即用管理器、电源管理器、执行体、I/O管理器、配置管理器和内存管理器）的停机处理。

例如，*NtSetSystemPowerState*调用I/O管理器，以便给那些已经请求过停机通知的所有设备驱动程序发送停机I/O包。这一动作使得设备驱动程序有机会在Windows退出以前执行任何必要的特殊处理。配置管理器将任何修改过的注册表数据刷新到磁盘上，内存管理器将所有已修改过的且包含了文件数据的页面写回到它们各自的文件中。如果“在停机时清除页面文件”的选项已被打开，那么，内存管理器在这时候清除页面文件。I/O管理器还会被再次调用，以便告诉文件系统驱动程序：系统正在进行停机。系统停机过程最终在电源管理器中结束。电源管理器所执行的动作取决于用户指定的是停机、重新引导，还是关闭电源。

5.4 本章总结

在本章中，我们讨论了在Windows启动和停机过程中（正常情况下，或是出错的情况下）涉及的各种细节步骤。到现在为止，我们已经看过了Windows的总体结构，以及让系统运行起来和停止下来的各种核心系统机制。有了这些基础作为准备，我们现在可以更加详细地讨论各个执行体组件，首先我们从进程和线程开始。

www.TopSage.com

Processes, Threads, and Jobs

第6章 进程、线程和作业

在本章中，我们将介绍Microsoft Windows中用于处理进程（process）、线程（thread）和作业（job）的数据结构和算法。第一节集中介绍了构成一个进程的内部结构。第二节概要地介绍了创建一个进程（和它的初始线程）时所涉及的步骤。然后介绍了线程的内部机理和线程调度机制。本章最后还讲述了作业对象。

凡是涉及性能计数器或者内核变量的地方，本章都有所提及。虽然本书并不是一本介绍Windows编程的书籍，但是，有关进程、线程和作业对象的Windows函数也都列出来了，所以，你可以进一步搜寻有关其用法的信息。

因为在Windows中进程和线程牵涉到了如此众多的组件，所以，本章中引用到了大量的术语和数据结构（比如工作集、对象和句柄、系统内存堆，等等），但是它们的细节解释却在本书别的地方。为了全面理解本章的内容，你需要熟悉第1和2章中解释的术语和概念，比如进程和线程之间的差别、Windows虚拟地址空间的布局结构，以及用户模式和内核模式之间的差别。

6.1 进程的内部机理

本节讲述了Windows进程的关键数据结构，同时也列出了关键的内核变量、性能计数器，以及与进程有关的函数和工具。

数据结构

每个Windows进程都是由一个执行体进程（EPROCESS）块来表示的。EPROCESS块中除了包含许多与进程有关的属性以外，还包含和指向了许多其他的相关数据结构。例如，每个进程都有一个或者多个线程，这些线程由执行体线程（ETHREAD）块来表示（本章后面的“线程的内部机理”一节解释了有关线程的数据结构）。EPROCESS块和相关的数据结构位于系统空间中，不过，进程环境块（PEB）是个例外，它位于进程地址空间中（因为它包含一些需要由用户模式代码来修改的信息）。

www.TopSage.com

除了EPROCESS块以外，Windows子系统进程（Csrss）为每个Windows进程（它执行了一个Windows程序）维护了一个类似的结构。而且，Windows子系统的内核模式部分（Win32k.sys）有一个针对每个进程的数据结构，当一个线程第一次调用Windows的USER或GDI函数（它们是在内核模式中实现的）时，此数据结构就会被创建。

图6.1是一个简化的关于进程和线程数据结构的框图。图中显示的每个数据结构在本章中都有详细的描述。

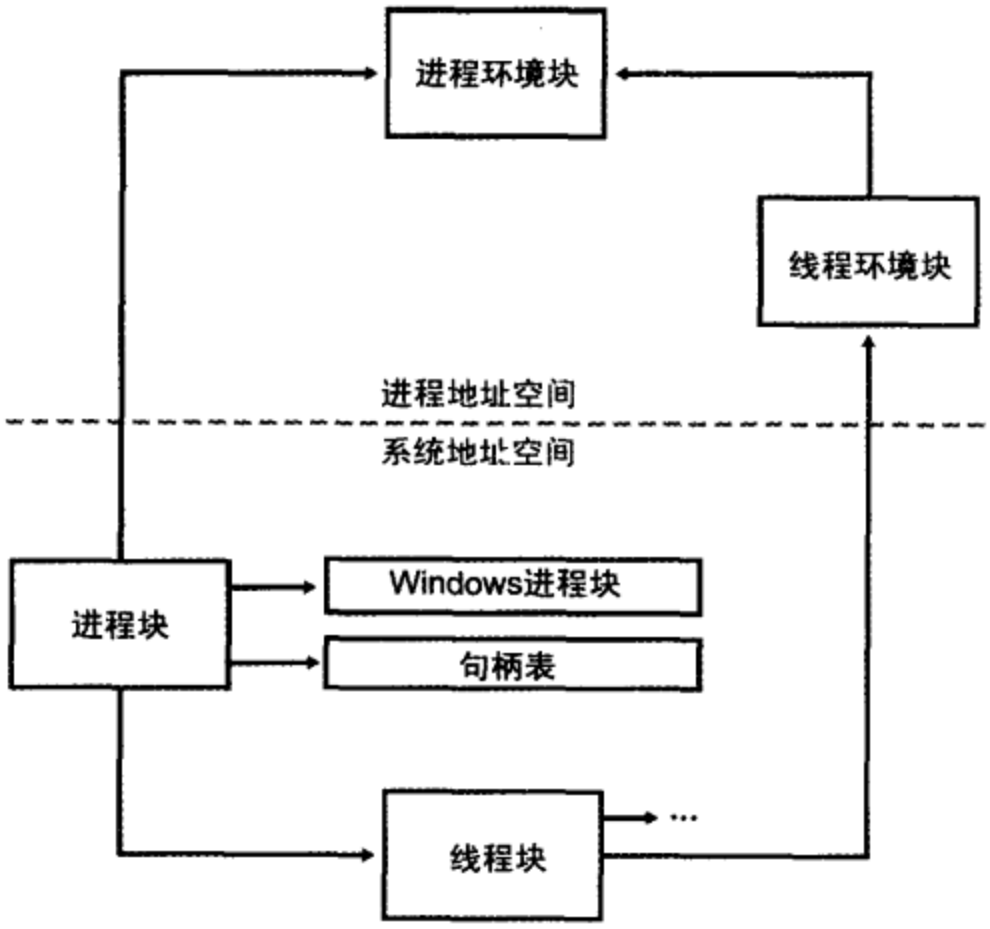


图6.1 与进程和线程相关联的数据结构

首先我们来关注进程块（在本章后面的“线程的内部机理”一节中我们将会介绍线程块）。图6.2显示了EPROCESS块中的关键域。

www.TopSage.com

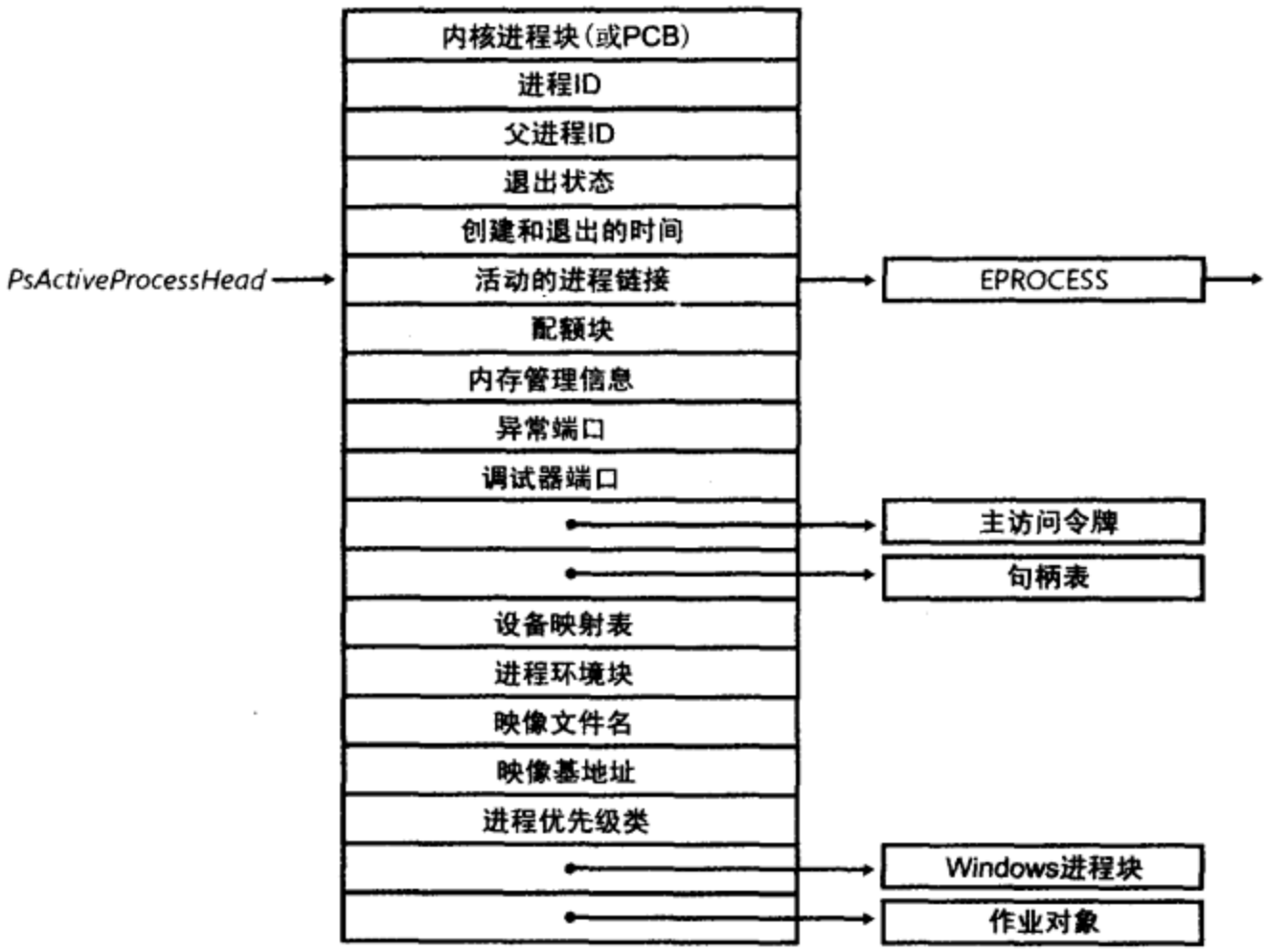


图6.2 执行体进程块的结构



实验：显示一个EPROCESS块的格式

要想看到一个EPROCESS块中所有域的列表，以及这些域在结构中的偏移量（按十六进制），你可以在内核调试器中输入**dt _eprocess**（关于内核调试器以及如何在本地系统上执行内核调试的更多信息，请参见第1章）。 其输出看起来像这样（由于空间的原因而有所删节）：

```
lkd> dt _eprocess
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
   +0x0c0 ExceptionPort    : Ptr32 Void
   +0x0c4 ObjectTable      : Ptr32 _HANDLE_TABLE
```


www.TopSage.com

```
   +0x0c8 Token              : _EX_FAST_REF
   +0x0cc WorkingSetLock     : _FAST_MUTEX
   +0x0ec WorkingSetPage     : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock     : Uint4B
   +0x114 ForkInProgress     : Ptr32 _ETHREAD
   +0x118 HardwareTrigger    : Uint4B
```

请注意，第一个域（Pcb）实际上是一个子结构：内核进程块（KPROCESS）。与调度有关的信息存储在此结构中。为了显示内核进程块的格式，请输入**dt_kprocess**：

```
lkd> dt _kprocess
nt!_KPROCESS
   +0x000 Header             : _DISPATCHER_HEADER
   +0x010 ProfileListHead    : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor      : _KGDTENTRY
   +0x028 Int21Descriptor    : _KIDTENTRY
   +0x030 IopmOffset         : Uint2B
   +0x032 Iopl               : UChar
   +0x033 Unused             : UChar
   +0x034 ActiveProcessors   : Uint4B
   +0x038 KernelTime         : Uint4B
   +0x03c UserTime           : Uint4B
   +0x040 ReadyListHead      : _LIST_ENTRY
   +0x048 SwapListEntry      : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler    : Ptr32 Void
   +0x050 ThreadListHead     : _LIST_ENTRY
   +0x058 ProcessLock        : Uint4B
   +0x05c Affinity           : Uint4B
   +0x060 StackCount         : Uint2B
   +0x062 BasePriority       : Char
   +0x063 ThreadQuantum      : Char
   +0x064 AutoAlignment      : UChar
   +0x065 State              : UChar
   +0x066 ThreadSeed         : UChar
   +0x067 DisableBoost       : UChar
   +0x068 PowerState         : UChar
   +0x069 DisableQuantum     : UChar
   +0x06a IdealNode          : UChar
   +0x06b Spare              : UChar
```

查看KPROCESS（以及EPROCESS中其他的子结构）的另一种方法是，使用dt命令的递归（*-r*）开关。例如，输入**dt _eprocess –r1**将会递归地显示出所有的子结构，递归的深度为1。

*dt*命令显示了一个进程块的格式，而不是它的内容。为了显示一个实际进程的实例，你可以指定一个EPROCESS结构的地址作为*dt*命令的一个参数。你利用*!process 0 0*命令，可以得到当前系统中所有EPROCESS块的地址。本章后面给出了该命令的一个例子输出。

www.TopSage.com

表6.1更加详细地解释了上述实验中的一些域，表中也列出了指向本书中其他地方的一些引用，在这些地方你可以找到更多有关这些域的信息。我们以前曾经说过，今后无疑还会这样说，进程和线程是Windows的有机组成部分，所以，要想在不涉及Windows许多其他部分的情况下来谈论进程和线程是不可能的。然而，为了保持本章的篇幅能得到有效的控制，我们在其他的地方讨论这些相关的主题（比如内存管理器、安全性、对象和句柄）。

表6.1 EPROCESS块的内容

元 素	用 途	其他的参考信息
内核进程（KPROCESS）块	共用的分发器对象头、指向进程页目录的指针、属于该进程的内核线程块（KTHREAD）的列表、默认的基本优先级、时限、亲和性掩码，以及该进程中的线程的内核总时间和用户总时间	线程调度（第6章）
进程标识	惟一的进程ID、创建进程的ID、被运行的映像文件名、该进程所在的窗口站	
配额块	在非换页池、换页池和页面文件中的使用量限制，以及当前和尖峰时的进程非换页池和换页池使用量（注：几个进程可以共享这个结构：所有的系统进程指向同一个全系统范围默认的配额块；交互式会话中所有的进程共享单个配额块，它是由Winlogon构建的）	
虚拟地址描述符（VAD）	一系列数据结构，描述了该进程中地址空间各个部分的状态	虚拟地址描述符（第7章）
工作集信息	一个指向工作集（working set）列表（MMWSL结构）的指针；当前的、尖峰的、最小的和最大的工作集大小；最后的修剪时间；页面错误计数值；内存优先级；页面换出标志；页面错误历史	工作集（第7章）
虚拟内存信息	当前的和尖峰的虚拟大小、页面文件使用量、进程页目录的硬件页表项	第7章
异常LPC端口	跨进程的通信通道，当该进程有一个线程引发了一个异常时进程管理器（process manager）向该通道发送一个消息	异常分发（第3章）

www.TopSage.com

续表

元 素	用 途	其他的参考信息
调试LPC端口	跨进程的通信通道，当该进程有一个线程引发一个调试事件时进程管理器向该通道发送一个消息	本地过程调用（LPC）（第3章）
访问令牌（ACCESS_TOKEN）	执行体对象，描述了该进程的安全轮廓	第8章
句柄表	针对每个进程的句柄表的地址	对象句柄和进程句柄表（第3章）
设备映射表	用于解析设备名称引用的对象目录的地址（支持多个用户）	对象名称（第3章）
进程环境块（PEB）	映像信息（基地址、版本号、模块列表）、进程堆信息，以及线程局部存储区的使用（注：指向进程堆的指针是从PEB之后的第一个字节开始的）	第6章
Windows子系统进程块（W32PROCESS）	Windows子系统的内核模式组件所需要的进程细节	

内核进程（KPROCESS）块是EPROCESS块的一个组成部分，而进程环境块（PEB）则是EPROCESS块中指向的一个结构，它们包含了有关进程对象的额外细节信息。KPROCESS块（有时候也称为PCB，即进程控制块）如图6.3所示。它包含了Windows内核在调度线程时所需要的基本信息（关于页目录的内容在第7章中介绍，而关于内核线程块的更多细节信息，则在本章后面讲述）。

PEB位于用户进程地址空间中，它包含了映像加载器、堆管理器和其他的Windows系统DLL所需要的信息，因为它们需要在用户模式下修改PEB中的信息（EPROCESS和KPROCESS块只能在内核模式下才可以访问）。PEB的基本结构如图6.4所示，在本章的最后部分将对其进行更详细的解释。

www.TopSage.com

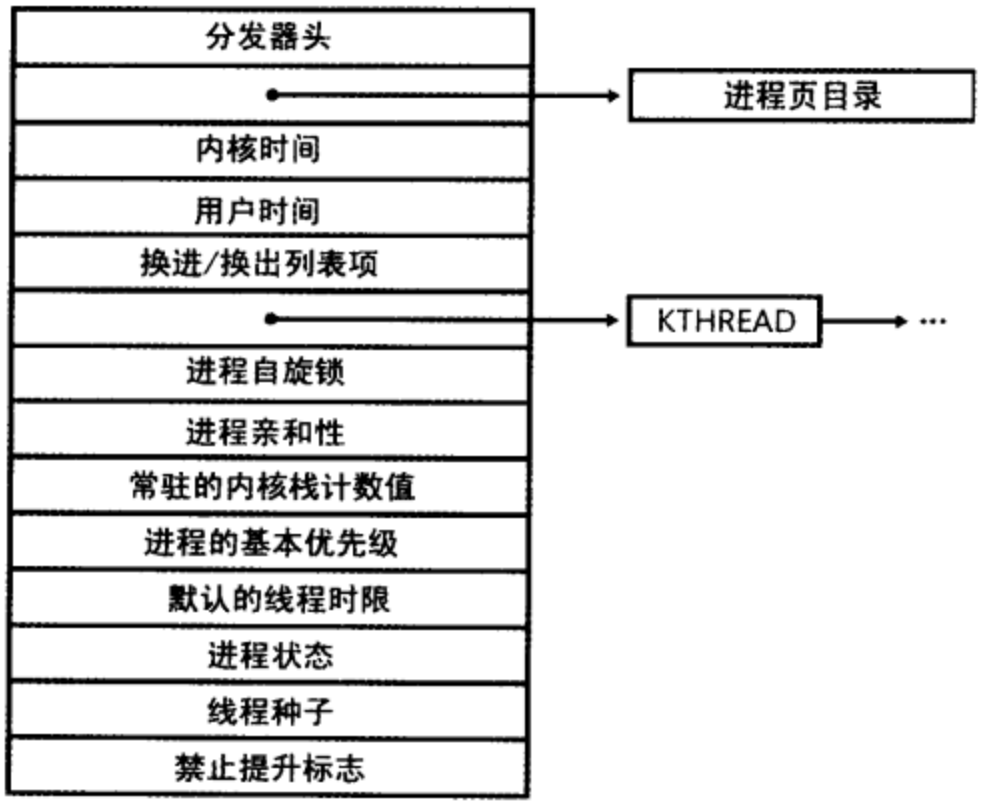


图6.4 执行体进程块的结构

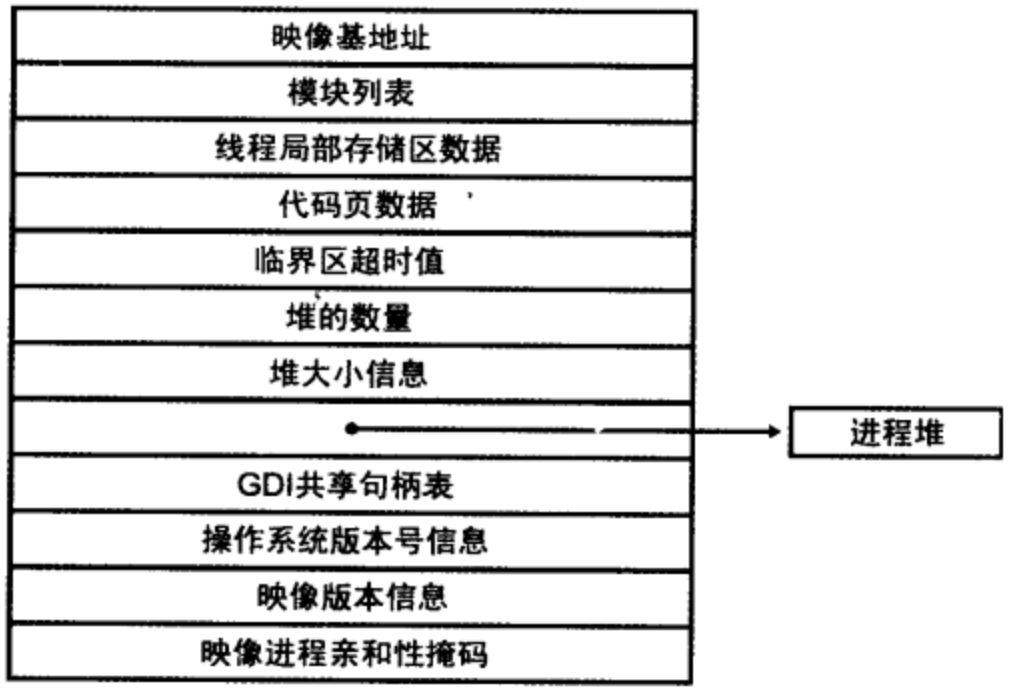


图6.4 进程环境块的域

www.TopSage.com



实验：检查PEB

在内核调试器中利用*!peb*命令，你可以将PEB结构转储出来。为了得到PEB的地址，你可以使用*!process*命令，如下所示：

```
lkd> !process
PROCESS 8575f030  SessionId: 0  Cid: 08d0    Peb: 7ffdf000  ParentCid: 0360
    DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  66.
    Image: windbg.exe
```

然后，在*!peb*命令中指定此地址，如下所示：

```
lkd> !peb 7ffdf000
PEB at 7ffdf000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         01000000
    Ldr                       00181e90
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00181f28 . 00183188
    Ldr.InLoadOrderModuleList:           00181ec0 . 00183178
    Ldr.InMemoryOrderModuleList:         00181ec8 . 00183180
            Base TimeStamp                     Module
         1000000 40478dbd Mar 04 15:12:45 2004 C:\Program Files\Debugging Tools for
             Windows\windbg.exe
        77f50000 3eb1b41a May 01 19:56:10 2003 C:\WINDOWS\System32\ntdll.dll
        77e60000 3d6dfa28 Aug 29 06:40:40 2002 C:\WINDOWS\system32\kernel32.dll
         2000000 40476db2 Mar 04 12:56:02 2004 C:\Program Files\Debugging Tools for
             Windows\dbgeng.dll
        .
    SubSystemData:     00000000
    ProcessHeap:       00080000
    ProcessParameters: 00020000
    WindowTitle:  'C:\Documents and Settings\All Users\Start Menu\Programs\Debugging
        Tools for Windows\WinDbg.lnk'
    ImageFile:    'C:\Program Files\Debugging Tools for Windows\windbg.exe'
    CommandLine:  '"C:\Program Files\Debugging Tools for Windows\windbg.exe" '
    DllPath:      'C:\Program Files\Debugging Tools for Windows;C:\WINDOWS\System32;C:\
WINDOWS\system;C:\WINDOWS;.;C:\Program Files\Windows Resource Kits\Tools\;C:\WINDOWS\
system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\Program Files\Support Tools\;c:\sysint;
C:\Program Files\ATI Technologies\ATI Control Panel;C:\Program Files\Resource Kit\;
C:\PROGRA~1\CA\Common\SCANEN~1;C:\PROGRA~1\CA\eTrust\ANTIVI~1;C:\Program Files\Common
    Files\Roxio Shared\DLLShared;C:\SFU\common\'
    Environment:  00010000
        =::=::\
        ALLUSERSPROFILE=C:\Documents and Settings\All Users          APPDATA=C:\Documents
and Settings\dsolomon\Application Data
    .
    .
    .
```


www.TopSage.com

内核变量

表6.2中列出了一些关键的、与进程相关的内核全局变量。在本章最后部分介绍创建进程的步骤时将会引用这些变量。

表6.2 与进程有关的内核变量

变 量	类 型	说 明
PsActiveProcessHead	队列头	进程块的列表头
PsIdleProcess	EPROCESS	空闲（Idle）进程块
PsInitialSystemProcess	指向EPROCESS的指针	指向初始系统进程的进程块的指针，该进程包含了系统线程
PspCreateProcessNotifyRoutine	指针数组	包含一些例程指针的数组，当进程创建和删除时这些例程将会被调用（最多8个）
PspCreateProcessNotifyRoutineCount	DWORD	已登记的进程通知例程的计数值
PspLoadImageNotifyRoutine	指针数组	包含一些例程指针的数组，当映像加载时调用这些例程
PspLoadImageNotifyRoutineCount	DWORD	已登记的映像-加载通知例程的计数值
PspCidTable	指向HANDLE_TABLE的指针	进程和线程客户ID的句柄表

性能计数器

Windows维护了许多计数器，你可以利用这些计数器来跟踪系统上正在运行的进程。你既可以通过编程的方法来获得这些计数器，也可以利用性能工具来查看它们。表6.3列出了与进程有关的性能计数器（不包括与内存管理和I/O相关的计数器，这两类计数器将分别在第7章和第9章中讲述）。

表6.3 与进程相关的性能计数器

对象：计数器	功 能
Process: % Privileged Time	描述了在指定的时间间隔内，该进程中的线程在内核模式下运行的时间百分比
Process: % Processor Time	描述了在指定的时间间隔内，该进程中的线程所使用的CPU时间的百分比。它是% Privileged Time和% User Time的和
Process: % User Time	描述了在指定的时间间隔内，该进程中的线程在用户模式下运行的时间百分比

www.TopSage.com

续表

对象：计数器	功 能
Process: Elapsed Time	描述了自从该进程创建以来，已经流逝的总时间
Process: ID Process	返回该进程的ID。只有当进程存在的时候该ID才有效，因为进程ID是要被重用的
Process: Creating Process ID	返回创建进程的进程ID。如果创建进程退出了，该值不会被更新。
Process: Thread Count	返回该进程中线程的数量
Process: Handle Count	返回该进程中已打开句柄的数量

有关的函数

出于参考的目的，表6.4中描述了一些适用于进程的Windows函数。有关更多的信息，请参考MSDN Library中的Windows API文档。

表6.4 与进程相关的函数

函 数	说 明
CreateProcess	使用调用者的安全标识来创建一个新的进程和线程
CreateProcessAsUser	用指定的另一个安全令牌来创建一个新的进程和线程
CreateProcessWithLogonW	创建一个新的进程和线程，让它在指定的用户名和口令的凭证下运行
CreateProcessWithTokenW	用指定的另一个安全令牌来创建一个新的进程和线程，包括一些附加的选项，比如允许用户轮廓被加载进来
OpenProcess	返回一个句柄，指向指定的进程对象
ExitProcess	终止一个进程，并通知所有附载的DLL
TerminateProcess	终止一个进程，但不通知其DLL
FlushInstructionCache	清空指定进程的指令缓存
GetProcessTimes	获得一个进程的时间信息，其中描述了该进程在用户模式和内核模式下花了多少时间
GetExitCodeProcess	返回一个进程的退出代码，表明了该进程如何停止，以及为什么停止
GetCommandLine	返回一个指针，指向传递给当前进程的命令行字符串
GetCurrentProcess	返回当前进程的一个伪句柄
GetCurrentProcessId	返回当前进程的ID

www.TopSage.com

续表

函 数	说 明
GetProcessVersion	返回指定进程所期望运行的Windows版本的主版本号和次版本号
GetStartupInfo	返回在CreateProcess中指定的STARTUPINFO结构的内容
GetEnvironmentStrings	返回环境块的地址
GetEnvironmentVariable	返回一个特定的环境变量
Get/SetProcessShutdownParameters	定义了当前进程的停机优先级，以及重试的次数
GetGuiResources	返回User和HDI句柄的计数值



实验：使用内核调试器的*!process*命令

内核调试器的*!process*命令显示了一个EPROCESS块中的一部分信息。在它的输出中，每个进程的信息被分成两部分。首先你看到有关一个进程的信息，如下所示（当你没有指定一个进程地址或者进程ID时，!process命令列出当前CPU上活动进程的信息）：

```
lkd> !process
PROCESS 8575f030  SessionId: 0  Cid: 08d0    Peb: 7ffdf000  ParentCid: 0360
    DirBase: 1a81b000  ObjectTable: e12bd418  HandleCount:  65.
    Image: windbg.exe
VadRoot 857f05e0 Vads 71 Clone 0 Private 1152. Modified 98. Locked 1.
DeviceMap e1e96c88
Token                             e1f5b8a8
ElapsedTime                       1:23:06.0219
UserTime                          0:00:11.0897
KernelTime                        0:00:07.0450
QuotaPoolUsage[PagedPool]         38068
QuotaPoolUsage[NonPagedPool]      2840
Working Set Sizes (now,min,max)  (2552, 50, 345) (10208KB, 200KB, 1380KB)
PeakWorkingSetSize                2715
VirtualSize                       41 Mb
PeakVirtualSize                   41 Mb
PageFaultCount                    3658
MemoryPriority                    BACKGROUND
BasePriority                      8
CommitCharge                      1566
```

在基本的进程信息输出之后，是该进程中的一系列线程。关于线程的输出信息，将在本章后面的“实验：使用内核调试器的*!thread*命令”部分中再介绍。其他的显示进程信息的命令包括*!handle*，它将进程句柄表中的信息转储出来（第3章中的“对象句柄和进程句柄表”一节详细地讲述了进程句柄表）。第8章中讲述了进程和线程的安全数据结构。

www.TopSage.com

6.2 CreateProcess的流程

本章到现在为止，你已经看到了有关进程的部分数据结构，以及你（和操作系统）可以用来操作进程的API函数。你也知道了如何使用工具来查看“进程是如何与系统打交道的”。但是，这些进程是如何产生的，一旦它们完成了任务之后又是如何退出的呢？在本节中，你将会明白Windows进程是如何活动起来的。

当一个应用程序调用某个进程创建函数，比如*CreateProcess*、*CreateProcessAsUser*、*CreateProcessWithTokenW*或*CreateProcessWithLogonW*时，一个Windows进程就被创建起来了。创建一个Windows进程的过程，是由操作系统的三个部分执行一系列步骤来完成的，这三个部分是客户方的Windows库Kernel32.dll、Windows执行体和Windows子系统进程（Csrss）。由于Windows是多环境子系统的体系结构，因此，创建一个Windows执行体进程对象（其他的子系统也可以使用），与创建一个Windows进程的工作是分离的。所以，尽管接下来关于Windows的*CreateProcess*函数的流程的介绍非常复杂，但是，要记住，这其中的一部分工作是与Windows子系统加入的语义相关的（相对于创建一个Windows执行体进程对象所需要的核心工作而言，这是额外加入的工作）。

下面的列表概括了在利用Windows的*CreateProcess*函数来创建一个进程时所涉及的主要阶段。在后续的小节中将详细地讲述在每一个阶段中执行的操作。



> **注** *CreateProcess*的许多步骤与进程虚拟地址空间的建立有关系，因此，在下面的介绍中引用了第7章中定义的许多与内存管理有关的术语和结构。

1. 打开将要在该进程中执行的映像文件（.exe）。
2. 创建Windows执行体进程对象。
3. 创建初始线程（栈、执行环境，以及Windows执行体线程对象）。
4. 通知Windows子系统新进程创建了，所以它可以为新进程和线程做好准备。
5. 开始执行初始线程（除非指定了CREATE_ SUSPENDED标志）。
6. 在新进程和线程的环境中，完成地址空间的初始化（比如加载必要的DLL），并开始执行程序。

图6.5显示了Windows在创建一个进程时所遵从的步骤的总体情况。

www.TopSage.com

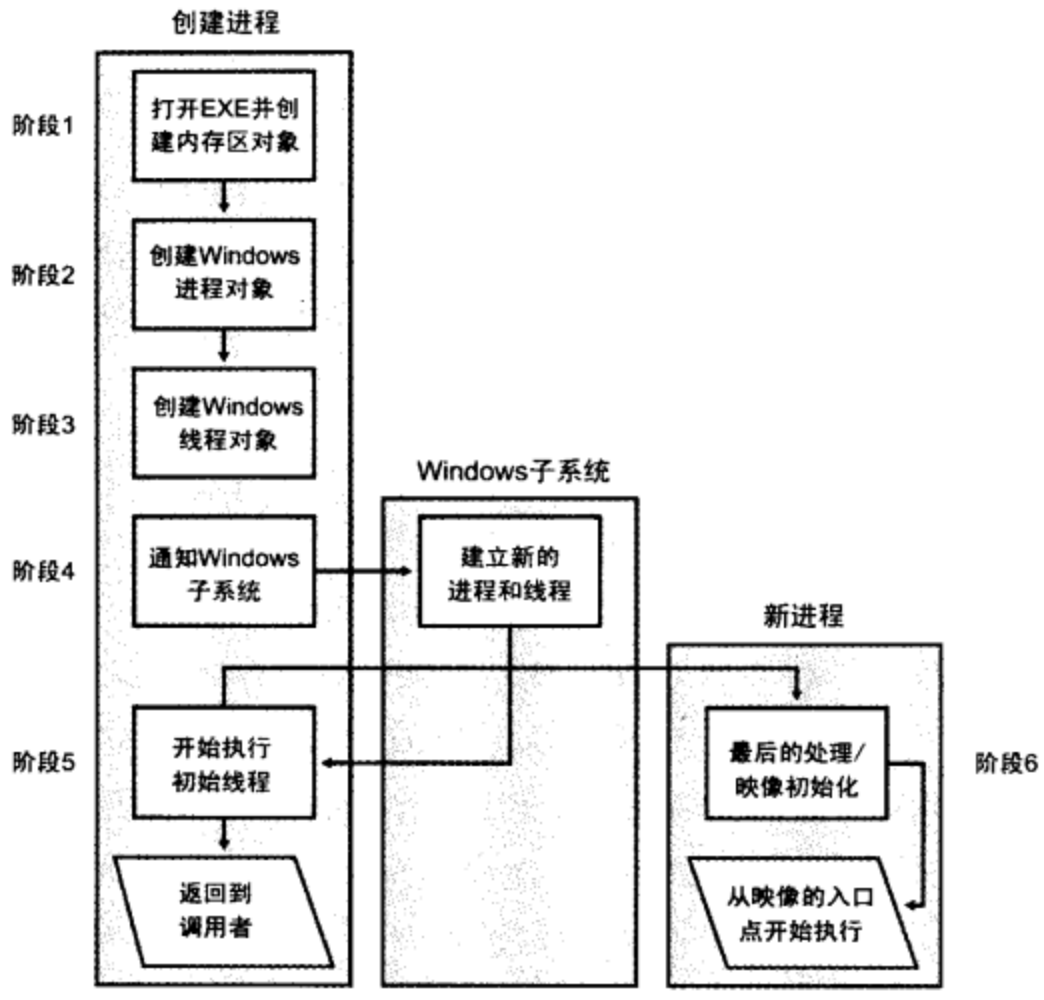


图6.5 进程创建过程的主要阶段

在打开可执行映像并使之运行以前，*CreateProcess*执行下面的步骤。

- 在*CreateProcess*中，新进程的优先级类别是在*CreationFlags*参数中由单独的位来指定的。因此，你可以为单个*CreateProcess*调用指定多个优先级类别。Windows通过选取最低优先级类别集合，来解决为进程分配优先级类别的问题；
- 如果创建新进程时没有指定优先级类别，那么，它的优先级类别默认被设置为Normal，除非创建线程的优先级类别是Idle或者Below Normal，在这种情况下，新进程的优先级类别与创建进程的优先级相同；
- 如果为新进程指定了Real-time优先级类别，但是该进程的调用者没有“Increase Scheduling Priority（增加调度优先级）”特权，则使用High优先级类别。换句话说，*CreateProcess*不会仅仅因为调用者没有足够的特权来创建Real-time优先级类别的进程而失败；新进程只是没有Real-time那么高的优先级而已；
- 所有的窗口都与桌面有关联，这里，桌面是一个工作区的图形表示。如果在*CreateProcess*中没有指定桌面，那么，该进程就与调用者的当前桌面关联在一起。

www.TopSage.com

阶段1：打开将要被执行的映像

正如图6.6所示，在*CreateProcess*中的第一个阶段是：先找到适当的Windows映像，它将运行由调用者指定的可执行文件；然后创建一个内存区对象，以便稍后将它映射到新进程的地址空间中。如果没有指定映像名称，则该命令行的第一个符号（被定义成命令行字符串中第一个空格或制表符之前的、合乎文件名规范的那一部分）被用作映像文件名。

在Windows XP和Windows Server 2003上，*CreateProcess*检查该机器上的软件限制策略是否允许该映像被运行起来（关于软件限制策略的完整描述，请参见第8章）。

如果指定的可执行文件是一个Windows .exe文件，那么，它可被直接使用。如果它不是一个Windows .exe（比如，它是一个MS-DOS、Win16或者POSIX应用程序），那么*CreateProcess*通过一系列的步骤来找到一个Windows**支持映像**（*support image*），以便运行它。这个过程是必需的，因为非Windows应用程序不能被直接运行，相反，Windows使用少数几个特定的支持映像中的某一个，由它负责实际运行这一非Windows程序。例如，如果你试图运行一个POSIX应用程序，那么，*CreateProcess*把它识别出来，并改变该映像，改变的可以在Windows可执行文件Posix.exe上运行。如果你试图运行一个MS-DOS或者Win16可执行文件，那么，被运行的映像变成了Windows可执行文件Ntvdm.exe。简而言之，你不能直接创建一个**非**Windows进程的进程。如果Windows不能找到一种办法把要激活的映像解析成一个Windows进程（如表6.5所示），那么*CreateProcess*就会失败。

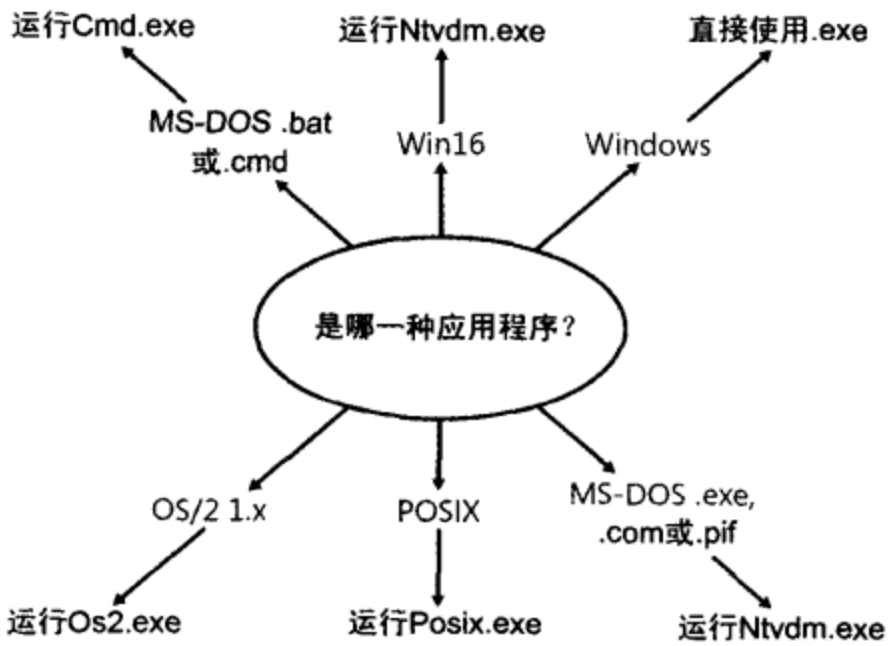


图6.6 选择一个Windows映像来激活新进程

www.TopSage.com

表6.5 CreateProcess的阶段1的决策层次

如果该映像是一个	那么，下面的程序将运行	该映像将运行
POSIX可执行文件	Posix.exe	*CreateProcess*重新开始阶段1
MS-DOS 应用程序，后缀为.exe、.com或.pif扩展名	Ntvdm.exe	*CreateProcess*重新开始阶段1
Win16应用程序	Ntvdm.exe	*CreateProcess*重新开始阶段1
命令程序（后缀为.bat或.cmd扩展名的应用程序）	Cmd.exe	*CreateProcess*重新开始阶段1

特别地，为了运行一个映像，*CreateProcess*经过的决策层次如下。

- 如果该映像是一个后缀为.exe、.com或.pif扩展名的MS-DOS应用程序，则向Windows子系统发送一个消息，以检查是否已经为该会话创建了一个MS-DOS支持进程（Ntvdm.exe，在注册表值HKLM\SYSTEM\CurrentControlSet\Control\WOW\ cmdline中指定）。如果已经有一个支持进程被创建了，则用它来运行此MS-DOS应用程序（Windows子系统给VDM[虚拟DOS机]进程发送消息，以运行此新映像）。 然后，*CreateProcess*返回。如果尚未有支持进程被创建，那么，被运行的映像改变成Ntvdm.exe，并且*CreateProcess*重新开始阶段1；
- 如果要运行的文件有一个.bat或.cmd扩展名，那么，被运行的映像是Cmd.exe，这是Windows的命令提示程序。然后*CreateProcess*重新开始阶段1（批文件的名称被作为第一个参数传递给Cmd.exe）；
- 如果该映像是一个Win16（Windows 3.1）可执行文件，则*CreateProcess*必须确定是否创建一个新的VDM进程来运行它，或者它是否应该使用整个会话共享的默认VDM进程（该进程可能还没有被创建起来）。*CreateProcess*的标志CREATE_SEPARATE_WOW_VDM和CREATE_SHARED_WOW_VDM控制了这一决定。如果这两个标志都没有被指定，那么，注册表值HKLM\SYSTEM\CurrentControlSet\Control\WOW\ DefaultSeparateVDM指定了默认的行为。如果该应用程序将被运行在一个独立的VDM中，那么，被运行的映像改变成HKLM\SYSTEM\CurrentControlSet\Control\WOW\wowcmdline中的值，并且*CreateProcess*重新开始阶段1。否则，Windows子系统发送一个消息，以确定共享的VDM进程是否存在，以及是否可以使用（如果该VDM进程运行在一个不同的桌面上，或者没有跟调用者运行在同样的安全环境下，那么，它就不能被使用，从而必须创建一个新的VDM进程）。如果有一个共享的VDM进程可以使用，那么，Windows子系统给它发送一个消息，以运行新的映像，然后*CreateProcess*返回。如果VDM进程尚未被创建（或者它虽然存在，但不能被使用），那么，被运行的映像改变成VDM支持映像，并且*CreateProcess*重新开始阶段1。

www.TopSage.com

在这个时候，*CreateProcess*已经成功地打开了一个有效的Windows可执行文件，并且为它创建了一个内存区对象。该对象尚未被映射到内存中，但它是打开的。然而，仅仅一个内存区对象被成功地创建了，并不意味着该文件是一个有效的Windows映像；它可以是一个DLL或者一个POSIX可执行文件。如果该文件是一个POSIX可执行文件，那么，该映像被改变成运行Posix.exe，并且*CreateProcess*从阶段1的起始处重新开始。如果该文件是一个DLL，则*CreateProcess*失败。

现在，*CreateProcess*已经找到了一个有效的Windows可执行映像，它在注册表中检查HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options键，看母键是否有一个子键与可执行映像的文件名和扩展名相同（但没有目录和路径信息，比如Image.exe）。如果确实有，则*CreateProcess*查找该键是否存在一个名为Debugger的值。如果存在，则该映像被改变成该值中的字符串，并且*CreateProcess*重新开始阶段1。



提示 你可以充分利用*CreateProcess*的这一行为，来调试Windows服务进程在启动其服务之前的启动代码（startup code），而不是在启动服务之后再将服务进程附载到一个调试器中，因为后一种方法并不能让你调试该进程的启动代码。

阶段2：创建Windows执行体进程对象

在这个时候，*CreateProcess*已经打开了一个有效的Windows可执行文件，并且创建了一个内存区对象，以便稍后将它映射到新的进程地址空间中。接下来，它通过调用Windows的内部系统函数*NtCreateProcess*，来创建一个Windows执行体进程对象，以运行该映像。创建执行体进程对象（是由创建线程来完成的）涉及以下子阶段。

- 建立起EPROCESS块；
- 创建初始的进程地址空间；
- 初始化内核进程块 （KPROCESS）；
- 结束进程地址空间的创建过程（包括初始化工作集列表和虚拟地址空间描述符，以及将映像映射到地址空间中）；
- 建立PEB；
- 完成执行体进程对象的创建过程。



注 不存在父进程的惟一情形是在系统初始化的时候。在此之后，父进程总是必需的，以便为新的进程提供一个安全环境。

www.TopSage.com

阶段2A：建立起EPROCESS块

该子阶段包括以下9个步骤。

1. 分配并初始化Windows EPROCESS块。
2. 从父进程处继承得到进程的亲和性掩码。
3. 进程的最小和最大的工作集尺寸被分别设置为*PsMinimumWorkingSet*和*PsMaximumWorkingSet*。
4. 将新进程的配额块设置为其父进程配额块的地址，并且递增父进程配额块的引用计数。
5. 继承Windows的设备名字空间（包括驱动器字母的定义、COM端口，等等）。
6. 将父进程的进程ID保存在新进程对象的*InheritedFromUniqueProcessId*域中。
7. 创建该进程的主访问令牌（父进程主令牌的副本）。新进程继承了其父进程的安全轮廓。如果通过*CreateProcessAsUser*来为新进程指定一个不同的访问令牌，则该令牌将在后面被正确地改正过来。
8. 初始化进程句柄表。如果父进程已经被设置了继承句柄标志，那么，从父进程的句柄表中将任何可继承的句柄拷贝到新进程中（有关对象句柄表的更多信息，请参见第3章）。
9. 将新进程的退出状态设置为STATUS_PENDING。

阶段2B：创建初始的进程地址空间

初始的进程地址空间由以下的页面组成：

- 页目录（对于超过两级页表的系统，比如PAE模式下的x86系统或者64位系统，可能会有多个页目录）；
- 超空间页面；
- 工作集列表。

为了创建这三个页面，需采取以下步骤。

1. 在适当的页表中创建页表项，以映射这些初始的页面。页面的数量可以根据内核变量*MmTotalCommittedPages*推断出来，然后加到*MmProcessCommit*上。
2. 从*MmResidentAvailablePages*推断出系统范围默认的进程最小工作集的大小（*PsMinimumWorkingSet*）。
3. 系统空间的非换页部分和系统缓存的页表页面被映射到进程中。

www.TopSage.com

阶段2C：创建内核进程块

*CreateProcess*的下一个阶段是初始化KPROCESS块，它包含一个指向内核线程列表的指针（内核没有关于句柄的知识，所以它绕过对象表）。内核进程块也指向进程的页表目录（此页表目录被用于跟踪进程的虚拟地址空间）、该进程中线程的总执行时间、该进程默认的基本调度优先级（除非父进程被设置为Idle或者Below Normal，在这种情况下子进程的基本优先级继承自父进程的设置，否则基本优先级自Normal或8开始）、该进程中线程默认的处理器亲和性，以及进程默认时限的初始值（更多的细节在本章后面的“线程调度”一节中讲述）。该初始值取自于系统全局范围的时限数组中的第一项，即*PspForegroundQuantum[0]*的值。



> **注** 对于Windows客户和服务器系统，默认的初始时限有所不同。有关线程时限的更多信息，在“线程调度”一节中的讨论。

阶段2D：结束进程地址空间的创建过程

为一个新进程建立起地址空间，这稍微有些复杂，所以，我们逐一来看一看其中涉及的步骤。要想了解这部分内容，你应该对于Windows内存管理的机理有所熟悉，第7章介绍了Windows内存管理。

- 虚拟内存管理器将该进程的最后修剪时间的值设置为当前时间。工作集管理器（它运行在平衡集管理器系统线程的环境中）利用该值来确定什么时候该触发工作集修剪过程；
- 内存管理器初始化该进程的工作集列表——现在就可以处理页面错误了；
- 现在可以将打开映像文件时创建的内存区映射到新进程的地址空间中，进程内存区的基地址被设置为该映像的基地址；
- 将Ntdll.dll映射到进程中；
- 系统范围的国家语言支持（NLS）表也被映射到该进程的地址空间中。



> **注** POSIX进程复制其父进程的地址空间，所以，它们不必经过以上这些步骤来创建一个新的地址空间。在POSIX应用程序的情况下，新进程的内存区基地址被设置为其父进程的内存区基地址，而父进程的PEB也被复制给新的进程。

www.TopSage.com

阶段2E：建立PEB

*CreateProcess*为PEB分配一个页面，并初始化其中的诸多域，如表6.6所示。

表6.6 PEB中域的初始值

域	初始值
ImageBaseAddress	内存区的基地址
NumberOfProcessors	KeNumberProcessors内核变量
NtGlobalFlag	NtGlobalFlag内核变量
CriticalSectionTimeout	MmCriticalSectionTimeout内核变量
HeapSegmentReserve	MmHeapSegmentReserve内核变量
HeapSegmentCommit	MmHeapSegmentCommit内核变量
HeapDeCommitTotalFreeThreshold	MmHeapDeCommitTotalFreeThreshold内核变量
HeapDeCommitFreeBlockThreshold	MmHeapDeCommitFreeBlockThreshold内核变量
NumberOfHeaps	0
MaximumNumberOfHeaps	（页面的大小 - PEB的大小） / 4
ProcessHeaps	PEB之后的第一个字节
OSMajorVersion	NtMajorVersion内核变量
OSMinorVersion	NtMinorVersion内核变量
OSBuildNumber	NtBuildNumber内核变量& 0x3FFF
OSPlatformId	2

如果映像文件指定了明确的Windows版本值，则这些版本信息将替代上述表6.6中显示的初始值。从映像版本信息域到PEB域的映射关系如表6.7所示。

表6.7 从映像头中取到的替代初始PEB值的版本信息

域 名	从映像头中取到的值
OSMajorVersion	OptionalHeader.Win32VersionValue & 0xFF
OSMinorVersion	（OptionalHeader.Win32VersionValue >> 8） & 0xFF
OSBuildNumber	（OptionalHeader.Win32VersionValue >> 16） & 0x3FFF
OSPlatformId	（OptionalHeader.Win32VersionValue >> 30） ^ 0x2

阶段2F：完成执行体进程对象的创建过程

在返回指向新进程的句柄以前，必须完成以下最后的一些创建步骤。

1. 如果系统范围的进程审计功能被打开（或者是本地策略设置的结果，或者是域控制器的组策略设置的结果）的话，则该进程的创建被写入到安全事件日志中。

www.TopSage.com

2. 如果父进程被包含在一个作业（job）中，则新进程也被加入到该作业中（本章最后介绍了作业）。
3. 如果映像头部特征的IMAGE_FILE_UP_SYSTEM_ ONLY标志被设置了（表明该映像只能在单处理器系统上运行），那么，选择一个CPU，以便新进程中所有的线程都运行在该CPU上。这一选择过程很简单，只是在所有可使用的处理器之间轮循而已——每次这种类型的映像被运行时，就会使用下一个处理器。通过这种方式，这些映像类型被均匀地分散在系统的处理器上。
4. 如果该映像指定了一个明确的处理器亲和性掩码（例如，在配置头中的一个域），则该值被拷贝到PEB中，稍后它被设置为默认的进程亲和性掩码。
5. *CreateProcess*将新进程块插入到Windows活动进程列表的末尾（*PsActiveProcessHead*）。
6. 设置该进程的创建时间，然后将指向新进程的句柄返回给调用者（Kernel32.dll中的*CreateProcess*）。

阶段3：创建初始线程，以及它的栈和执行环境

到这个时候，Windows执行体进程对象已经被完全建立起来了。然而，它仍然没有线程，所以，它还不能做任何事情。在创建初始线程以前，它需要一个栈以及一个可供运行的环境，所以，现在是该建立它们的时候了。初始线程的栈大小是从映像中取到的——你没有办法指定另一个大小值。

现在可以创建初始线程了，这是通过调用*NtCreateThread*来完成的。线程的参数（它不可能在*CreateProcess*中指定，但可以在*CreateThread*中指定）是PEB的地址。运行于此新线程环境中的初始化代码将会使用该参数（在阶段6中介绍）。然而，该线程仍然不会做任何事情，因为它被创建起来后直接处于挂起状态，直到该进程被完全初始化以后它才开始运行（在阶段5中介绍）。*NtCreateThread*调用*PspCreateThread*（也是一个被用来创建系统线程的函数），它执行下面的步骤。

1. 递增进程对象中的线程计数值。
2. 创建并初始化一个执行体线程块 （ETHREAD）。
3. 为新线程生成一个线程ID。
4. 在进程的用户模式地址空间中建立TEB。
5. 用户模式线程的起始地址被保存在ETHREAD中。对于Windows线程，这是系统提供的线程启动函数，位于Kernel32.dll中（对于进程中的第一个线程是*BaseProcessStart*；对于其他的线程则是*BaseThreadStart*）。用户指定的Windows启动地址被保存在ETHREAD块中的另一位置上，因而，系统提供的线程启动函数可以调用用户指定的启动函数。

www.TopSage.com

6. 调用KeInitThread来建立起KTHREAD块。该线程初始的基本优先级和当前优先级均被设置为进程的基本优先级，它的亲和性和时限被设置为进程的亲和性和时限。该函数也会设置初始线程的理想处理器（关于如何选择理想处理器，请参见“理想的和最后的处理器”）。 KeInitThread接下来为该线程分配一个内核栈，并且为它初始化与机器相关的硬件环境，包括执行环境、陷阱和异常帧。该线程的执行环境也被建立起来，因而在内核模式下此线程可在KiThreadStartup中启动起来。最后，KeInitThread将该线程的状态设置为已初始化，并返回到PspCreateThread。
7. 调用任何已登记的系统全局范围的线程来创建通知例程。
8. 该线程的访问令牌被设置为指向进程的访问令牌，然后做一次访问检查，以确定调用者是否有权创建该线程。如果你是在本进程中创建一个线程，那么这一检查将总是成功的，但是，如果你是在另一个进程中创建一个线程，并且创建线程的进程没有调试特权，那么这一访问检查可能会失败。
9. 最后，该线程做好执行准备。

阶段4：将新进程通知Windows子系统

如果软件限制策略有指示的话，则为新进程创建一个受限制的令牌。到这个点上，所有必要的执行体进程对象和执行体线程对象都已经被创建出来了。接下来Kernel32.dll给Windows子系统发送一个消息，从而它可以进一步建立起新进程和线程。该消息包含以下信息：

- 进程和线程的句柄；
- 创建标志中的各个项目；
- 该进程的创建者的ID；
- 指示该进程是否属于一个Windows应用程序的标志（所以，Csrss可以确定是否要显示启动光标）。

当Windows子系统接收到此消息时，它执行下面的12个步骤。

1. *CreateProcess*复制一份该进程和线程的句柄。在这一步，进程和线程的使用计数从1（在创建时设置的）增加到2。
2. 如果进程的优先级类别没有被指定的话，则*CreateProcess*根据本节前面描述的算法设置进程的优先级类别。
3. 分配Csrss进程块。
4. 新进程的异常端口被设置为Windows子系统的通用功能端口，这样，当该进程中发生异常时，Windows子系统将会收到一个消息（有关异常处理的更进一步信息，请参见第3章）。

www.TopSage.com

5. 如果该进程正在被调试（也就是说，如果它被附载到一个调试器进程）的话，则该进程的调试端口被设置为Windows子系统的通用功能端口。这一设置可以保证，Windows将把在新进程中发生的调试事件（比如线程创建和删除、异常，等等）作为消息发送给Windows子系统，从而它可以把这些事件分发给新进程的调试器进程。
6. 分配并初始化Csrss线程块。
7. *CreateProcess*把该线程插入到进程的线程列表中。
8. 递增该会话中的进程计数值。
9. 进程的停机级别被设置为0x280，这是默认的进程停机级别（更多的信息，请参见MSDN Library文档中的*SetProcessShutdownParameters*）。
10. 将新进程块插入到Windows子系统范围的进程的列表中。
11. Windows子系统的内核模式部分所用到的、针对每个进程的数据结构（W32PROCESS结构）被分配并初始化。
12. 显示应用程序启动光标。该光标是你所熟悉的沙漏箭头。这是Windows提醒用户的方式："我正在启动某些东西，但同时你可以使用此光标。"如果该进程在2秒钟后仍没有执行GUI调用，那么，该光标恢复成标准的指示符号。如果该进程在规定的时间内执行了一个GUI调用，则*CreateProcess*等待5秒钟，让应用程序显示一个窗口。5秒钟以后，*CreateProcess*将再次重置光标。

阶段5：启动初始线程的执行

到这时候，进程环境已经确定了，其线程将要使用的资源也已经分配好了，进程有了一个线程，Windows子系统也知道了此新进程。除非调用者指定了CREATE_ SUSPENDED标志，否则，初始线程现在被恢复起来，并开始运行，执行余下部分的进程初始化工作 （必须在新进程的环境中执行，见阶段6）。

阶段6：在新进程环境下执行进程初始化

新线程的生命是从运行内核模式的线程启动例程*KiThreadStartup*开始的。*KiThreadStartup*将线程的IRQL级别从DPC/Dispatch级别降低到APC级别，然后调用系统初始的线程例程*PspUserThreadStartup*。用户提供的线程启动地址被作为一个参数传递给此例程。

在Windows 2000上，*PspUserThreadStartup*首先打开工作集扩充特性。如果被创建的进程是一个将要被调试的进程，那么，该进程中所有的线程都被挂起（这些线程可能在阶段3中已经被创建了）。 然后向该进程的调试端口（这是Windows子系统功能端口，因为它是一个Windows进程）发送一个创建进程消息，因而子系统可以向适当的调试器进程递交一个进程启动调试事件（CREATE_PROCESS_DEBUG_INFO）。然后，*PspUserThreadStartup*等待Windows

www.TopSage.com

子系统，以得到来自于调试器的回复（通过*ContinueDebugEvent*函数）。当Windows子系统回复时，所有的线程都被恢复运行。

在Windows XP和Windows Server 2003上，*PspUserThreadStartup*检查该系统上是否已经打开了应用程序预取的特性，如果是的话，则调用逻辑预取器来处理预取指令文件（如果该文件存在的话），并且将该进程上一次启动的前10秒钟内引用的页面预取进来（有关预取器的细节，请参见第3章）。最后，*PspUserThreadStartup*把一个用户模式的APC排入到队列中，以便运行映像加载器初始化例程（Ntdll.dll中的*LdrInitializeThunk*）。当该线程试图返回到用户模式时，该APC将会被交付。

当*PspUserThreadStartup*返回到*KiThreadStartup*中时，它从内核模式中返回，该APC被交付，并且*LdrInitializeThunk*被调用。*LdrInitializeThunk*例程初始化加载器、堆管理器、NLS表、线程局部存储区（TLS）数组，以及临界区结构。然后它加载任何必要的DLL，并且用DLL_PROCESS_ ATTACH功能代码来调用DLL的入口点（关于在Windows XP中引入的、用来解决DLL版本问题的机制的介绍，请参见下面的辅助内容 “并行的程序集”）。

最后，当加载器的初始化例程返回到用户模式APC分发器时，进程映像开始在用户模式下执行，然后它调用线程的启动函数，该启动函数的地址已经在用户APC交付时被压入到用户栈中。

并行的程序集

一个长期困扰Windows用户的问题是“DLL黑洞”。当你安装一个应用程序，并且它替换了一个或者多个核心系统DLL（比如通用控件的DLL、Microsoft Visual Basic的运行库DLL，或者MFC的DLL）时，你便进入了DLL黑洞。应用安装程序之所以要做这些替换，是为了保证新安装的应用程序能够正确地运行，但同时，更新之后的DLL可能与其他已经安装的应用程序不兼容。

Windows 2000部分地解决了DLL黑洞问题，它的做法是，利用Windows文件保护（WFP）特性来避免对核心系统DLL进行修改，同时又允许应用程序使用这些核心DLL的私有拷贝。要想使用一个DLL的私有拷贝而不是系统目录中的拷贝，安装后的应用程序必须包括一个名为*Application*.exe.local的文件，这里*Application*是该应用可执行文件的名称。此local文件指示加载器首先在该目录中寻找DLL。这种类型的DLL重定向避免了应用/DLL的不兼容问题，但是它这样做的代价是牺牲了共享DLL，而共享DLL正是最初设计DLL的一个要点。而且，任何从KnownDLLs列表中加载的DLL（被永久映射到内存中的DLL），或者由这些DLL加载的DLL都无法通过这种机制来重新定向。

www.TopSage.com

为了进一步解决应用程序和DLL的兼容性同时又允许DLL共享的问题，Windows XP引入了共享的程序集。一个程序集是由一组资源构成的，包括一些DLL和一个描述了该程序集及其内容的XML清单（manifest）文件。一个应用程序只要存在它自己的XML清单，就引用了一个程序集。此清单可以是该应用程序安装目录下的一个文件，与应用程序有同样的名称，但后面又加上了“.manifest”（例如，application.exe.manifest），或者此清单也可以被作为一个资源链接到应用程序中。清单描述了该应用程序和它对程序集的依赖性。

程序集有两种类型：私有的和共享的。两者之间的区别在于，共享的程序集是经过数字签名的，所以，其内容被破坏或修改以后可以被检测出来。而且，共享的程序集被存放在\Windows\Winsxs目录下，而私有的程序集被存放在应用程序的安装目录下。因此，共享的程序集也有一个关联的目录（catalog）文件（.cat），其中包含了它的数字签名信息。共享的程序集可以是“并行的”程序集，因为一个DLL的多个版本可以同时驻留在一个系统上，凡是依赖于某个特定版本DLL的应用程序总是使用此特殊的版本。

一个程序集的清单文件通常有这样一个名称：它包含了程序集的名称、版本信息，以及一些文本表达了一个惟一的签名，最后是扩展名“.manifest”。此清单被存放在\Windows\Winsxs\Manifests中，程序集的其余资源被存放在\Windows\Winsxs的子目录中，该子目录的名称与对应的清单文件同名，当然刨除尾部的.manifest扩展名。

共享程序集的一个例子是Windows通用控件DLL，即comctl32.dll的版本6。它对于Windows XP是新的，它的清单文件被命名为\Windows\Winsxs\Manifest\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.0.0_x-ww_1382d70a.manifest。它有一个关联的目录（catalog）文件（文件名相同，只不过扩展名为.cat），以及在Winsxs下有一个子目录，其中包含了comctl32.dll。

Comctl32.dll的版本6包含了与Windows XP主题特性（Windows XP Themes）的集成，而且，如果有些应用程序在编写的时候没有考虑到对于主题特性的支持，那么，它跟新的DLL一起工作时可能会表现得不正确，所以，只有当应用程序显式地引用了包含版本6的共享程序集时才可以正常工作。安装在\Windows\System32中的Comctl32.dll版本是版本5.x的一个实例，它不能理解Windows XP的主题特性。当一个应用程序加载的时候，加载器查找该应用程序的清单，如果找到一个清单的话，则从指定的程序集中加载它的DLL。对于那些在清单引用的程序集中没有包含的DLL，则仍然按照传统的方式来加载。因此，以前的应用程序链接的是\Windows\System32中的版本；而那些能理解主题特性的应用程序则可以在它们的清单中指定新的版本。

www.TopSage.com

你只需用两种不同的方式（用清单文件来指示系统使用新的通用控件库，或者不用清单文件）运行用户状态迁移向导（User State Migration Wizard, \Windows\System32\Usmt\Migwiz.exe），就可以看到一个清单的效果：

1. 运行向导，注意观察向导中的按钮上的Windows XP主题。
2. 在Notepad中打开Migwiz.exe.manifest文件，定位到包含版本6通用控件库的那一行。
3. 将Migwiz.exe.manifest改名为**Migwiz.exe.manifest.bak**。
4. 返回向导，注意观察非主题化的按钮。
5. 将清单文件恢复为原来的名称。

共享程序集的最后一个优势是，发布者可以发行一个发布者配置，它将所有使用某个特定程序集的应用程序重新定向为使用一个更新的版本。如果发布者在消除代码错误的时候又希望保留后向兼容性，则发布者可以这样做。然而，最终而言，由于程序集模型中内在的灵活性，一个应用程序可以决定覆盖新的设置，或者继续使用老的版本。

6.3 线程的内部机理

我们已经讨论了进程，现在我们将注意力转到线程的结构上来。除非显式地声明，否则，你总是可以假设本节中所有讨论的内容既适用于用户模式的线程，也适用于内核模式的系统线程（第2章中讲述了系统线程）。

数据结构

在操作系统层次上，Windows线程是由一个执行体线程（ETHREAD）块来表示的，如图6.7所示。ETHREAD块和它所指向的结构都位于系统地址空间中，惟一的例外是线程环境块（TEB），它位于进程地址空间中。而且，Windows子系统进程（Csrss）为Windows进程中创建的每个线程维护了一个平行的结构。另外，对于那些已经调用了任一个Windows子系统USER或GDI函数的线程，Windows子系统的内核模式部分（Win32k.sys）为每个这样的线程维护了一个数据结构，称为W32THREAD结构，线程的ETHREAD块指向此结构。

www.TopSage.com

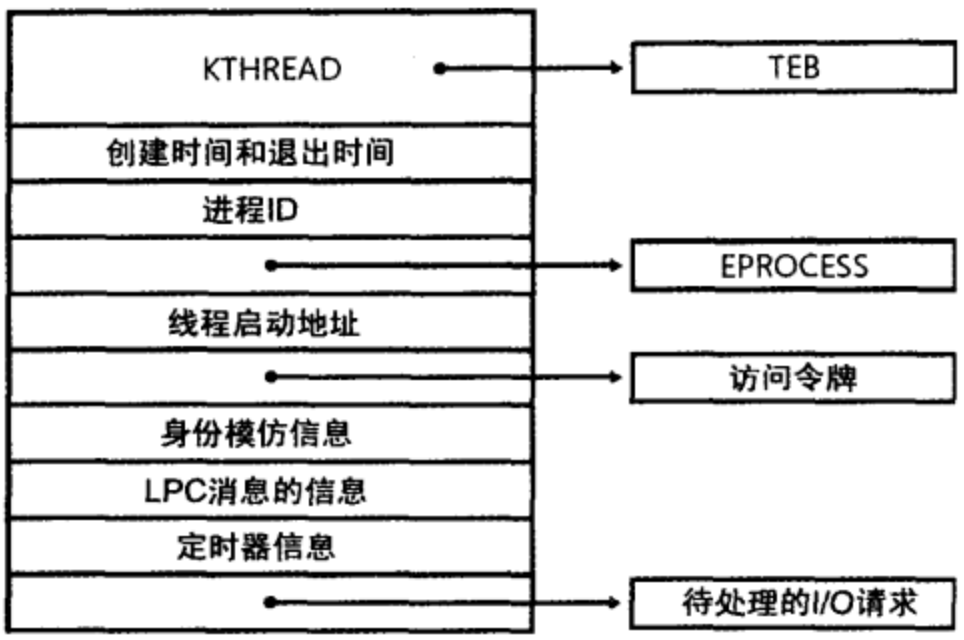


图6.7 执行体线程块的结构

图6.7中显示的绝大多数域已经不言自明了。第一个域是内核线程（KTHREAD）块。紧随其后的是线程标识信息、进程标识信息（包括一个指向所有者进程的指针，因而可以访问它的环境信息）、安全信息（其形式包括一个指向访问令牌的指针，以及身份模仿信息），最后是与LPC消息和待处理I/O请求有关的域。正如你在表6.8中看到的那样，有些关键域将在本书其他地方详细讲述。关于ETHREAD块的内部结构的更多细节，你可以使用内核调试器*dt*命令来显示该结构的格式。

表6.8 执行体线程块的关键内容

元 素	说 明	其他的参考信息
KTHREAD	见表6.9	
线程时间	关于线程创建时间和退出时间的信息	
进程标识	进程ID，以及指向该线程所属进程的EPROCESS块的指针	
启动地址	线程启动例程的地址	
身份模仿信息	访问令牌和模仿级别（如果该线程正在模仿一个客户的话）	第8章
LPC信息	该线程正在等待的消息ID，以及消息的地址	本地过程调用（第3章）
I/O信息	待处理的I/O请求包（IRP）的列表	I/O系统（第9章）

现在我们来更仔细地看一看在上面段落中引用到的两个关键的线程数据结构：KTHREAD块和TEB。KTHREAD块包含了Windows内核为这些正在运行的线程执行线程调度和同步而需要访问的信息。它的布局结构如图6.8所示。

www.TopSage.com

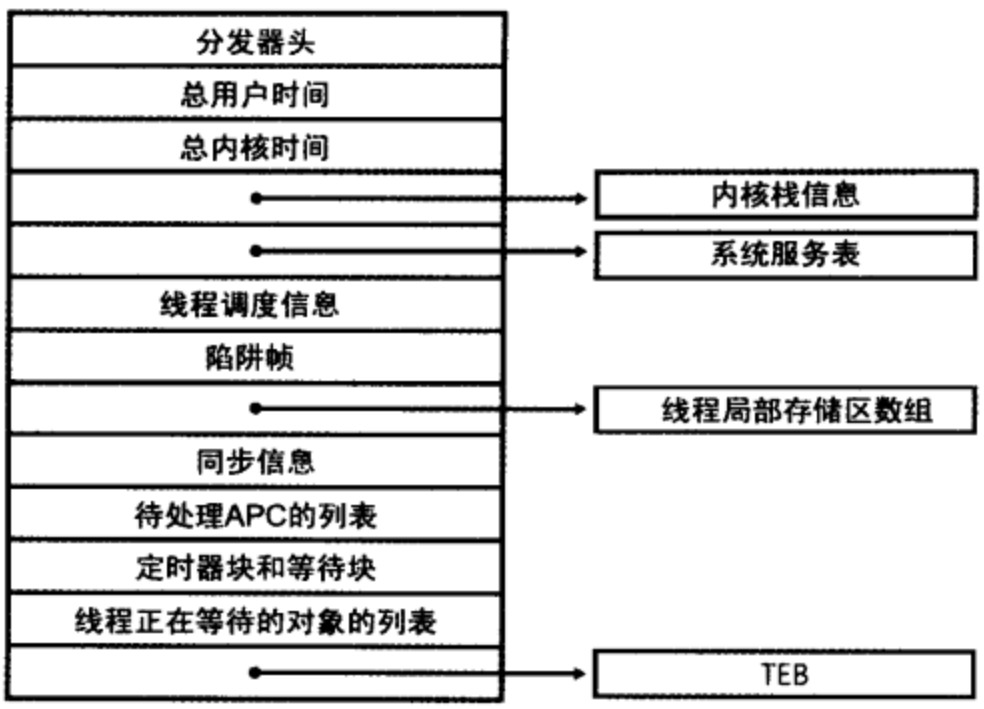


图6.8 内核线程块的结构

表6.9概要地描述了KTHREAD块中的关键域。

表6.9 KTHREAD块的关键内容

元 素	说 明	其他的参考信息
分发器头	因为线程是一个可被等待的对象，所以它的开头是一个标准的内核分发器对象头	内核分发器对象（第3章）
执行时间	总用户CPU时间和总内核CPU时间	
指向内核栈信息的指针	内核栈的基地址和上界（upper）地址	内存管理 （第7章）
指向系统服务表的指针	每个线程在开始时此域指向主系统服务表（*KeServiceDescriptorTable*）。当一个线程第一次调用Windows GUI服务时，它的系统服务表被改变为Win32k.sys中另一张包含了GDI和USER服务的服务表	系统服务分发 （第3章）
调度信息	基本优先级和当前优先级、时限、亲和性掩码、理想处理器、调度状态、冻结计数（freeze count）和挂起计数（suspend count）	线程调度
等待块	线程块包含4个内置的等待块，因而每次当线程等待什么事情的时候，不必分配和初始化等待块（一个等待块专门用于定时器）	同步（第3章）

www.TopSage.com

续表

元 素	说 明	其他的参考信息
等待信息	线程正在等待的对象的列表、等待的原因，以及该线程进入等待状态的时间	同步（第3章）
突变体（mutant）列表	此线程拥有的突变体对象的列表	同步（第3章）
APC队列	未处理的用户模式和内核模式APC，以及可提醒的标志	异步过程调用（APC）中断（第3章）
定时器块	内置的定时器块（以及一个对应的等待块）	
队列列表	指向与该线程关联的队列对象的指针	同步（第3章）
指向TEB的指针	线程ID、TLS信息、PEB指针，以及GDI和OpenGL信息	



实验：显示ETHREAD和KTHREAD结构

在内核调试器中，利用*dt*命令可以显示ETHREAD和KTHREAD结构。下面的输出显示了ETHREAD的格式：

```
lkd> dt nt!_ethread
nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
   +0x1c0 ApcNeeded        : Pos 2, 1 Bit
   +0x1c8 ExitTime         : _LARGE_INTEGER
   +0x1c8 LpcReplyChain    : _LIST_ENTRY
   +0x1c8 KeyedWaitChain   : _LIST_ENTRY
   +0x1d0 ExitStatus       : Int4B
   +0x1d0 OfsChain         : Ptr32 Void
   +0x1d4 PostBlockList    : _LIST_ENTRY
   +0x1dc TerminationPort  : Ptr32 _TERMINATION_PORT
   +0x1dc ReaperLink       : Ptr32 _ETHREAD
   +0x1dc KeyedWaitValue   : Ptr32 Void
   +0x1e0 ActiveTimerListLock : Uint4B
   +0x1e4 ActiveTimerListHead : _LIST_ENTRY
   +0x1ec Cid              : _CLIENT_ID
   +0x1f4 LpcReplySemaphore : _KSEMAPHORE
   +0x1f4 KeyedWaitSemaphore : _KSEMAPHORE
   +0x208 LpcReplyMessage  : Ptr32 Void
   +0x208 LpcWaitingOnPort : Ptr32 Void
   +0x20c ImpersonationInfo : Ptr32 _PS_IMPERSONATION_INFORMATION
   +0x210 IrpList          : _LIST_ENTRY
   +0x218 TopLevelIrp      : Uint4B
   +0x21c DeviceToVerify   : Ptr32 _DEVICE_OBJECT
   +0x220 ThreadsProcess   : Ptr32 _EPROCESS
```


www.TopSage.com

```
+0x224 StartAddress     : Ptr32 Void
+0x228 Win32StartAddress : Ptr32 Void
+0x228 LpcReceivedMessageId : Uint4B
+0x22c ThreadListEntry  : _LIST_ENTRY
+0x234 RundownProtect   : _EX_RUNDOWN_REF
+0x238 ThreadLock       : _EX_PUSH_LOCK
+0x23c LpcReplyMessageId : Uint4B
+0x240 ReadClusterSize  : Uint4B
+0x244 GrantedAccess    : Uint4B
+0x248 CrossThreadFlags : Uint4B
+0x248 Terminated       : Pos 0, 1 Bit
+0x248 DeadThread       : Pos 1, 1 Bit
+0x248 HideFromDebugger : Pos 2, 1 Bit
+0x248 ActiveImpersonationInfo : Pos 3, 1 Bit
+0x248 SystemThread     : Pos 4, 1 Bit
+0x248 HardErrorsAreDisabled : Pos 5, 1 Bit
+0x248 BreakOnTermination : Pos 6, 1 Bit
+0x248 SkipCreationMsg  : Pos 7, 1 Bit
+0x248 SkipTerminationMsg : Pos 8, 1 Bit
+0x24c SameThreadPassiveFlags : Uint4B
+0x24c ActiveExWorker   : Pos 0, 1 Bit
+0x24c ExWorkerCanWaitUser : Pos 1, 1 Bit
+0x24c MemoryMaker      : Pos 2, 1 Bit
+0x250 SameThreadApcFlags : Uint4B
+0x250 LpcReceivedMsgIdValid : Pos 0, 1 Bit
+0x250 LpcExitThreadCalled : Pos 1, 1 Bit
+0x250 AddressSpaceOwner : Pos 2, 1 Bit
+0x254 ForwardClusterOnly : UChar
+0x255 DisablePageFaultClustering : UChar
```

通过一个类似的命令，可以显示KTHREAD结构的格式：

```
lkd> dt nt!_kthread
nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY
   +0x018 InitialStack     : Ptr32 Void
   +0x01c StackLimit       : Ptr32 Void
   +0x020 Teb              : Ptr32 Void
   +0x024 TlsArray         : Ptr32 Void
   +0x028 KernelStack      : Ptr32 Void
   +0x02c DebugActive      : UChar
   +0x02d State            : UChar
   +0x02e Alerted          : [2] UChar
   +0x030 Iopl             : UChar
   +0x031 NpxState         : UChar
   +0x032 Saturation       : Char
   +0x033 Priority         : Char
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : Uint4B
   +0x050 IdleSwapBlock    : UChar
   +0x051 Spare0           : [3] UChar
   +0x054 WaitStatus       : Int4B
```


www.TopSage.com



实验：使用内核调试器的*!thread*命令

内核调试器的*!thread*命令可以将线程数据结构中的一部分信息转储出来。在内核调试器显示的信息中，有些关键的元素是不可能通过其他的工具显示出来的，它们是内部结构的地址、优先级细节信息、栈信息、待处理的I/O请求列表，以及那些处于等待状态的线程正在等待的对象的列表。

为了显示线程信息，或者使用*!process*命令（它在显示了进程块以后，再显示所有的线程块），或者使用*!thread*命令将一个指定的线程转储出来。下面显示了线程信息的输出，对一些关键的域加上了一些标记。

ETHREAD的地址　　线程ID　　线程环境块的地址

```
THREAD 83160f0  Cid:  9f.3dTeb:  7ffdc000  Win32Thread:  e153d2c8
WAIT:  (WrUserRequest)  UserMode Non-Alertable ———————— 线程状态
        808e9d60 SynchronizationEvent
                ———————————————————————— 被等待的对象
    Not imersonating
    Owning Process 81b44880 ———— 所有者进程的EPROCESS的地址
    Wait Time (seconds)              953945
    Context Switch Count   2697                        LargeStack
    UserTime                         0:00:00.0289        实际的线程
    KernelTime                       0:00:04.0644         启动地址
    Start Address kernal32!BaseProcessStart (0x77e8f268) ———┘
    Win32 Start Address 0x020d9d98 ———————— 用户线程函数的地址
    Stack Init f7818000 Current f7817bb0 Base f7818000 Limit f7812000 Call 0
    Priority 14 BasePriority 9 PriorityDecrement 6 DecrementCount 13 ———┐
Kernal stack not resident.                                          优先级信息
  ChildEBP RetAddr          Args to Child
  F7817bb0 8008f430 00000001 00000000 00000000 ntoskrnl!KiSwapThreadExit
  F7817c50 de0119ec 00000001 00000000 00000000 ntoskrnl!KeWaitForSingleObject+0x2a0
  F7817cc0 de0123f4 00000001 00000000 00000000 win32k!xxxSleepThread+0x23c
  F7817d10 de01f2f0 00000001 00000000 00000000 win32k!xxxInternalGetMessage+0x504
  F7817d80 800bab58 00000001 00000000 00000000 win32k!NtUserGetMessage+0x58
  F7817df0 77d887d0 00000001 00000000 00000000 ntoskrnl!KiSystemServiceEndAddress+0x4
  0012fef0 00000000 00000001 00000000 00000000 user32!GetMessageW+0x30
```

栈的转储输出



实验：查看线程信息

下面的输出是Windows调试工具箱（Windows Debugging Tools）中的Tlist工具产生的进程详细信息。注意，线程列表显示了“Win32StartAddress”。这是应用程序传递给*CreateThread*

www.TopSage.com

函数的地址。除了进程管理器（Process Explorer）会显示此线程启动地址以外，所有其他工具显示的都是实际启动地址（Kernel32.dll中的一个函数），而不是应用程序指定的启动地址。

```
C:\> tlist winword
 155 WINWORD.EXE       Document1 - Microsoft Word
   CWD:     C:\book\
   CmdLine: "C:\Program Files\Microsoft Office\Office\WINWORD.EXE"
   VirtualSize:    64448 KB   PeakVirtualSize:   106748 KB
   WorkingSetSize:  1104 KB   PeakWorkingSetSize:  6776 KB
   NumberOfThreads: 2
    156 Win32StartAddr:0x5032cfdb LastErr:0x00000000 State:Waiting
    167 Win32StartAddr:0x00022982 LastErr:0x00000000 State:Waiting
                      0x50000000  WINWORD.EXE
     5.0.2163.1 shp  0x77f60000  ntdll.dll
     5.0.2191.1 shp  0x77f00000  KERNEL32.dll
         §          list of DLLs loaded in process
```

图6.9显示的TEB是本节中介绍的惟一一个位于进程地址空间（相对于系统空间）中的数据结构。

TEB存储了有关映像加载器和各种Windows DLL的环境信息。因为这些组件运行在用户模式下，所以，它们需要一个在用户模式下可写的数据结构。为什么该结构位于进程地址空间中，而不是位于系统空间中呢？因为系统空间只能在内核模式下才可写。通过内核调试器的*!thread*命令，你可以找到TEB的地址。

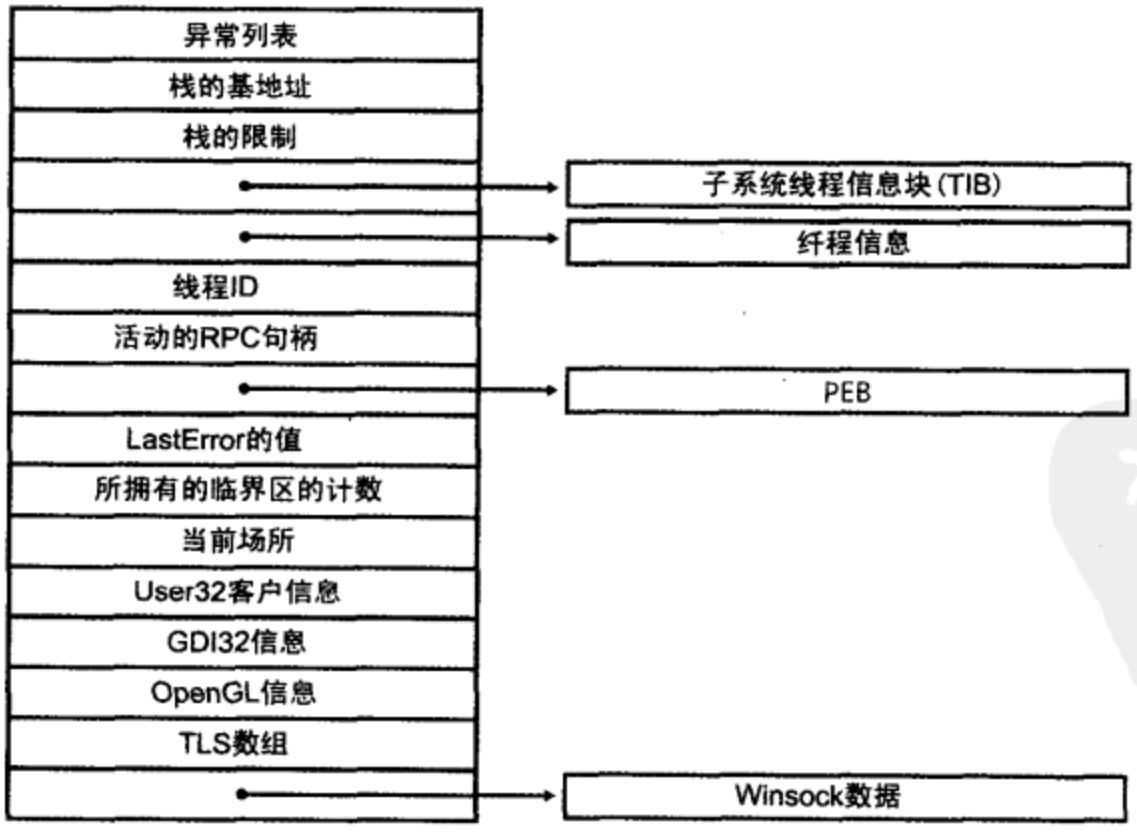


图6.9 线程环境块的域

www.TopSage.com



实验：检查TEB

在内核调试器中，你可以利用*!teb*命令转储出TEB结构。其输出类似于如下所示：

```
kd> !teb
TEB at 7ffde000
    ExceptionList:          0006b540
    StackBase:              00070000
    StackLimit:             00065000
    SubSystemTib:           00000000
    FiberData:              00001e00
    ArbitraryUserPointer:   00000000
    Self:                   7ffde000
    EnvironmentPointer:     00000000
    ClientId:               00000254 . 000007ac
    RpcHandle:              00000000
    Tls Storage:            00000000
    PEB Address:            7ffdf000
    LastErrorValue:         2
    LastStatusValue:        c0000034
    Count Owned Locks:      0
    HardErrorMode:          0
```

内核变量

如同进程的情形一样，有许多Windows内核变量控制着线程的运行。表6.10显示了与线程有关的内核模式变量。

表6.10 与线程有关的内核变量

变 量	类 型	说 明
PspCreateThreadNotifyRoutine	指针数组	指向一些例程指针的数组，在线程创建和删除时这些例程将会被调用（最多8个）
PspCreateThreadNotifyRoutineCount	DWORD	已登记的线程通知例程的计数值
PspCreateProcessNotifyRoutine	指针数组	指向一些例程指针的数组，当进程创建和删除时这些例程将会被调用（最多8个）

www.TopSage.com

性能计数器

在线程数据结构中，绝大多数关键信息被导出成性能计数器，表6.11中列出了这些性能计数器。你只需使用Windows中的性能工具，就可以提取到有关一个线程内部机理的许多信息。

表6.11 与线程有关的性能计数器

对象：计数器	功 能
Process: Priority Base	返回进程的当前基本优先级。对于该进程内部创建的线程来说，这是起始的优先级
Thread: % Privileged Time	描述了在指定的时间间隔内，该线程在内核模式下运行的时间百分比
Thread: % Processor Time	描述了在指定的时间间隔内，该线程所使用的CPU时间的百分比。它是% Privileged Time和% User Time的和
Thread: % User Time	描述了在指定的时间间隔内，该线程在用户模式下运行的时间百分比
Thread: Context Switches/Sec	返回在系统执行过程中每秒钟环境切换的次数
Thread: Elapsed Time	描述了该线程已经消耗掉的CPU时间（以秒为单位）总量
Thread: ID Process	返回该线程所属进程的ID。此ID只在该进程的生命期内才有效，因为进程ID是要被重用的
Thread: ID Thread	返回该线程的线程ID。此ID只在该线程的生命期内才有效，因为线程ID是要被重用的
Thread: Priority Base	返回该线程的当前基本优先级，此数值可能不同于线程的起始基本优先级
Thread: Priority Current	返回该线程的当前动态优先级
Thread: Start Address	返回该线程的启动虚拟地址（注：对于绝大多数线程，此地址是相同的）
Thread: Thread State	返回一个0至7之间的值，它与该线程的当前状态有关
Thread: Thread Wait Reason	返回一个0至9之间的值，它与该线程处于等待状态的原因有关

www.TopSage.com

有关的函数

表6.12显示了与创建和操作线程有关的Windows函数。与线程调度和线程优先级有关的函数并没有包含在此表中，它们包含在本章后面的“线程调度”一节中。

表6.12 Windows线程函数

函 数	说 明
CreateThread	创建一个新的线程
CreateRemoteThread	在另一个进程中创建一个线程
OpenThread	打开一个已有的线程
ExitThread	正常地终止一个线程的执行
TerminateThread	终止一个线程
GetExitCodeThread	获得另一个线程的退出代码
GetThreadTimes	返回一个线程的时间信息
GetCurrentProcess	返回当前线程的一个伪句柄
GetCurrentProcessId	返回当前线程的线程ID
GetThreadId	返回指定线程的线程ID
Get/SetThreadContext	返回或者改变一个线程的CPU寄存器
GetThreadSelectorEntry	返回另一个线程的描述符表项（仅适用于x86系统）

一个线程的产生

当一个程序创建一个新的线程时，该线程的生命周期就开始了。此创建线程的请求被传递到Windows执行体中，在这里，进程管理器（process manager）为线程对象分配空间，并调用内核来初始化内核线程块。在Windows Kernel32.dll中的*CreateThread*函数内部，采取以下的步骤来创建一个Windows线程。

1. *CreateThread* 在进程的地址空间中为新线程创建一个用户模式栈。
2. *CreateThread* 初始化该线程的硬件环境（与CPU体系结构相关）（有关线程环境块的更多信息，请参见Windows API参考文档中的CONTEXT结构）。
3. 调用*NtCreateThread* 来创建一个处于挂起状态的执行体线程对象（有关该函数执行步骤的描述，请参见“CreateProcess的流程”一节中阶段3和阶段6的介绍）。
4. *CreateThread*通知Windows子系统有关新线程的信息，子系统为新线程做一些准备工作。
5. 将线程句柄和线程ID（在第3步中生成的）返回给调用者。

www.TopSage.com

6. 除非调用者在创建线程的调用中指定了CREATE_SUSPENDED标志，否则，该线程现在恢复运行，因而它可以被调度执行。当该线程开始运行时，它在调用实际用户指定的启动地址以前，首先执行上一节的“阶段6：在新进程环境下执行进程初始化”中描述的步骤。

6.4 检查线程活动

除了性能工具，还有其他几个工具也暴露了Windows线程状态中的各种元素（在“线程调度”一节中列出了一些能显示有关线程调度信息的工具）。 这些工具如图6.10中逐条所列。



注 为了用Tlist来显示线程的细节，你必须输入tlist xxx，这里xxx是一个进程的映像名称，或者窗口标题（支持统配符）。

对象	Perfmon	Pviewer	Pstat	Qslice	Tlist	KD *!thread*	Process Explorer	Pslist
线程ID	✓	✓	✓		✓	✓	✓	✓
实际的启动地址	✓	✓	✓			✓	✓	✓
Win32启动地址					✓	✓	✓	
当前地址	✓	✓				✓	✓	
环境切换的次数	✓	✓	✓				✓	✓
总用户时间		✓	✓			✓	✓	✓
总的特权执行时间		✓	✓			✓	✓	✓
逝去的时间	✓	✓				✓	✓	✓
线程状态	✓		✓		✓	✓	✓	✓
等待状态的理由	✓		✓		✓	✓	✓	✓
最后的错误					✓		✓	
CPU时间的百分比	✓			✓			✓	
用户时间的百分比	✓			✓			✓	
特权执行时间的百分比	✓			✓			✓	
TEB的地址						✓		
ETHREAD的地址						✓		
正在等待的对象						✓		

图6.10 与线程有关的工具以及它们的功能

进程管理器（Process Explorer）提供了很容易的途径来访问一个进程内的线程活动情况。确定为什么一个正在运行的进程 （比如Svchost.exe、Dllhost.exe、Inetinfo.exe或者System进程）宿纳了多个服务，或者为什么一个进程已经挂起了，这是非常重要的。

www.TopSage.com

为了查看一个进程内的线程，你可以选择一个进程，并打开进程的属性（在你选择的进程上双击，或者点击Process菜单，并选择Properties菜单项）。然后在Threads标签上点击。此页面显示了该进程内的线程列表。对于每个线程，它显示了所消耗CPU时间的百分比（基于配置好的刷新间隔）、轮到该线程的环境切换次数，以及线程的启动地址。你可以按照任何一列来排序。

新创建的线程被加亮显示成绿色，而退出的线程则被加亮显示成红色（通过Options、Configure Hightlighting菜单项可以配置加亮显示的时间长度）。 这对于发现一个进程中不必要的线程创建操作，可能会非常有帮助（一般而言，线程的创建应该发生在进程启动的时候，而不是每次当该进程处理一个请求的时候）。

当你从线程列表中选中一个线程时，进程管理器显示它的线程ID、启动时间、状态、CPU时间计数器、环境切换的次数，以及基本优先级和当前优先级。窗口中有一个Kill按钮，它将会终止一个线程，但这一功能用起来应该要绝对小心。

环境切换差值代表了在进程管理器配置的两次刷新之间该线程开始运行的次数。它提供了另一种不同于所消耗CPU时间百分比的方法来确定线程的活动情况。在某些方面，这种方法更好，因为许多线程运行的时间如此之短，以至于当时间间隔定时器中断发生时，这些线程很少（如果有的话）正在运行，因此，它们不会被计算CPU时间。例如，如果你将环境切换差值栏添加到进程显示窗口中，并且按此栏排序的话，那么，你将会看到哪些进程虽然有线程在运行，但是其CPU时间百分比却为0（或者非常小）。

线程启动地址被显示成“*module*!*function*”的形式，这里*module*是.exe或者.dll的名称。函数名依赖于能否访问该模块的符号文件（参见第1章中的“实验：利用进程管理器查看进程的细节”）。如果你不能确定该模块是什么，你可以按下Module按钮。这将打开Windows管理器的文件属性窗口，其中显示了包含该线程启动地址的模块（例如，.exe或.dll文件）的属性。



> **注** 对于那些通过Windows的CreateThread函数创建的线程，进程管理器显示的是传递给CreateThread的函数，而不是实际的线程启动函数。这是因为，所有的Windows线程都是从一个公共的进程或线程启动封装函数（Kernel32.dll中的*BaseProcessStart*或*BaseThreadStart*）开始执行的。如果进程管理器显示的是实际启动地址，那么，进程内的绝大多数线程看起来都是从同一个地址开始执行的，这对于理解一个线程正在执行什么样的代码，将毫无帮助。

然而，这里显示的线程启动地址可能还不足以查明该线程正在做什么事情，以及该进程内的哪些组件负责该线程所消耗的CPU时间。如果线程的启动地址是一个通用的启动函数（譬如，此函数名根本没有指明该线程实际在做什么事情），则情况尤为如此。在这种情况下，检查线程的栈可能会解答这样的问题。为了查看一个线程的栈，你可以在感兴趣的线程上双击（或者选中该线程，并点击Stack按钮）。进程管理器显示了该线程的栈（如果该线程在内核模式中，则同时包括用户模式栈和内核模式栈）。

www.TopSage.com



> 注 虽然用户模式调试器（Windbg、Ntsd和Cdb）允许你附载到一个进程中，并显示一个线程的用户栈，但是，在进程管理器中，你只需轻轻点一个按钮，就可以同时显示用户栈和内核栈。你也可以使用*www.sysinternals.com*的Livekd工具，来检查用户线程栈和内核线程栈。然而，它比较难以使用。注意，在本地内核调试模式下运行Windbg（只在Windows XP或Windows Server 2003上才支持此模式）并不显示线程栈。

若能够检查线程的栈，则有助于你判断为什么一个进程被挂起了。例如，在一个系统上，Microsoft PowerPoint在启动时有一分钟时间停止不动。为了判断为什么它被挂起了，你在启动了PowerPoint之后，可以使用进程管理器来检查该进程中一个线程的栈。图6.11显示了此结果。

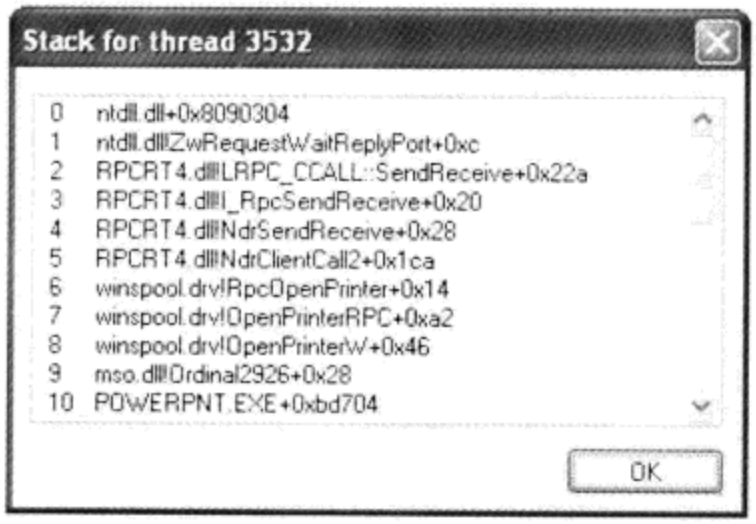


图6.11 PowerPoint中被挂起的线程的栈

此线程栈表明，PowerPoint（第10行）调用了Mso.dll（主要的Microsoft Office DLL）中的一个函数，该函数又调用了Winspool.drv（一个用于连接至打印机的DLL）中的*OpenPrinterW*函数。然后，Winspool.drv分发至函数*OpenPrinterRPC*，它又调用RPC运行库DLL中的一个函数，这说明它正在给一台远程打印机发送请求。所以，无须理解PowerPoint的内部机理，仅仅根据线程栈中显示的模块和函数名，就知道该线程正在等待连接到一台网络打印机。在这个特定的系统上，有一台网络打印机不响应了，这就解释了PowerPoint在启动时的延迟（Microsoft Office应用程序在进程启动时会连接到所有已配置的打印机上）。 在用户的系统上删除了至该打印机的连接以后，问题就迎刃而解了。

6.5 线程调度

本节描述了Windows调度策略和算法。第一小节集中描述了在Windows上线程调度是如何工作的，同时也介绍一些关键术语的定义。然后，分别从Windows API和Windows内核的角度，讲述了Windows优先级别。在回顾了有关的Windows函数，以及与调度有关的_Windows实用工

www.TopSage.com

具以后，接着又详细地介绍了构成Windows调度系统的数据结构和算法，在这部分中，先讨论单处理器系统，然后讨论多处理器系统。

Windows调度的概述

Windows实现了一个优先级驱动的、抢先式调度系统——具有最高优先级的可运行（就绪）线程总是运行，而该线程可能仅限于在允许它运行的处理器上运行，这种现象称为**处理器亲和性**（*processor affinity*）。在默认情况下，线程可以在任何一个空闲的处理器上运行，但是，你可以使用表6.13（本章后面显示）中列出的Windows调度函数，或者在映像头部设置一个亲和性掩码，来改变处理器亲和性。



实验：查看就绪的线程

你可以通过内核调试器的*!ready*命令来查看就绪线程的列表。该命令显示了在每个优先级别上，已经准备好运行的线程的列表。在下面的例子中，在优先级10上有2个线程准备运行，在优先级8上有6个线程准备运行。因为这一输出是在一个单处理器系统上利用LiveKd生成的，所以，当前线程将总是内核调试器（Kd或WinDbg）。

```
kd> !ready 1
Ready Threads at priority 10
 THREAD 810de030  Cid 490.4a8 Teb: 7ffd9000  Win32Thread: e297e008 READY
 THREAD 81110030  Cid 490.48c Teb: 7ffde000  Win32Thread: e29425a8 READY
Ready Threads at priority 8
 THREAD 811fe790  Cid 23c.274 Teb: 7ffdb000  Win32Thread: e258cda8 READY
 THREAD 810bec70  Cid 23c.50c Teb: 7ffd9000  Win32Thread: e2ccf748 READY
 THREAD 8003a950  Cid 23c.550 Teb: 7ffda000  Win32Thread: e29a7ae8 READY
 THREAD 85ac2db0  Cid 23c.5e4 Teb: 7ffd8000  Win32Thread: e297a9e8 READY
 THREAD 827318d0  Cid 514.560 Teb: 7ffd9000  Win32Thread: 00000000 READY
 THREAD 8117adb0  Cid 2d4.338 Teb: 7ffaf000  Win32Thread: 00000000 READY
```

一个线程被选中运行的时候，它运行一定长度的时间，这段时间称为**时限**（*quantum*）。时限是指，一个线程在“系统将运行权交给同一优先级别（或者更高优先级别，在多处理器系统上可能会发生这种情况）上的另一个线程”之前允许运行的时间长度。时限值可以随着系统的不同而不同，也可以随着进程的不同而不同，其理由有三：系统配置中的设置（长时限值，或者短时限值），进程的前台/后台状态，或者使用作业对象来改变时限值（在本章后面的“时限”一节中更加详细地讲述了时限）。 然而，一个线程可能未能完成它的时限。因为Windows实现了一个抢先式调度器，所以，如果另一个具有更高优先级的线程变成就绪状态时，当前正在运行的线程可能在完成它的时间片之前被抢占掉了。实际上还可能发生这样的情况：一个线程被选中作为下一个将要运行的线程，但是在它的时限开始以前就可能被抢占掉。

www.TopSage.com

Windows调度代码是在内核中实现的。然而，内核中不存在单独的“调度器”模块或者例程——调度代码散布在内核中凡是会发生与调度相关的事件的各处。执行这些任务的例程合起来称为内核的“**分发器**（*dispatcher*）”。下面的事件可能要求线程分发：

- 一个线程变成就绪执行状态时——例如，一个线程已经被新建起来了，或者刚刚从等待状态中释放出来；
- 一个线程离开了运行状态，这可能有以下的原因——一个线程的时间限额结束了，或者线程终止了，或者线程放弃执行，或者进入了等待状态；
- 一个线程的优先级改变了，可能是由于一个系统服务调用的原因，也可能是因为Windows本身改变了优先级值；
- 一个线程的处理器亲和性改变了，因而它不再在当前运行的处理器上运行了。

在以上每一个交汇处，Windows必须确定接下来该哪个线程运行。当Windows选择一个新的线程来运行时，它执行一次环境切换，切换到该线程中。环境切换是指这样一个过程：将当前正在运行的线程的易失性机器状态保存起来，并装入另一个线程的易失性状态，然后开始此新线程的执行。

正如前面已经说明过的，Windows在线程粒度上进行调度。你如果能这样想那就对了：进程并不运行，它只是提供资源和一个环境，让它的线程可以在此环境中运行。因为调度决定是严格在线程的基础上做出的，所以，无需考虑一个线程属于哪个进程。例如，如果进程A有10个可运行的线程，进程B有2个可运行的线程，并且这12个线程在相同的优先级上，那么，在理论上讲，每个线程将接收到1/12的CPU时间——Windows不会将50%的CPU时间给进程A，其余的50%给进程B。

优先级别

为了理解线程调度算法，你首先必须理解Windows所使用的优先级别。如图6.12中所示的，Windows在内部使用32个优先级别，范围从0至31。这些值划分如下：

- 16个实时级别（16至31）；
- 15个可变级别（1至15）；
- 1个系统级别（0），保留给零页面线程（zero page thread）。

www.TopSage.com

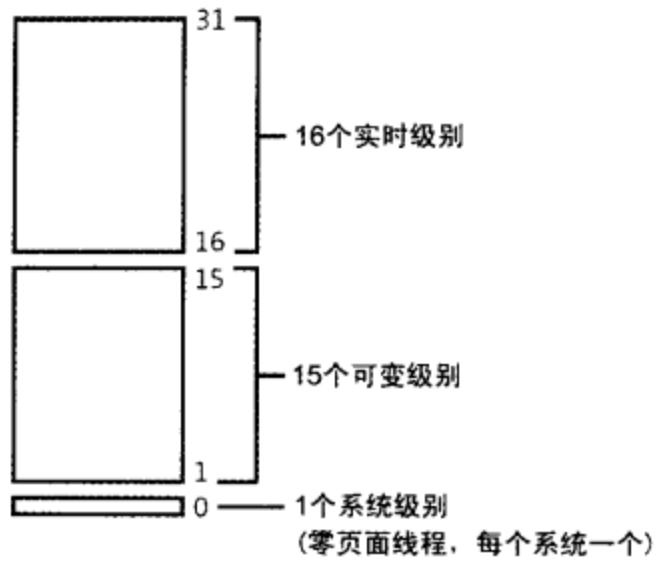


图6.12 线程优先级别

线程的优先级别是从两个不同的角度来分配的：Windows API的优先级和Windows内核的优先级。Windows API首先根据进程创建时所分配的优先级类别（实时[Real-Time]、高[High]、普通之上[Above Normal]、普通[Normal]、普通之下[Below Normal]、空闲[Idle]）来组织进程，然后根据这些进程内部单个线程的相对优先级（时间关键[Time-critical]、最高[Highest]、普通之上[Above-normal]、普通[Normal]、普通之下[Below-normal]、最低[Lowest]，以及空闲[Idle]）来组织线程。

在Windows API中，每个线程都有一个基本优先级，它是其进程优先级类别和相对线程优先级的一个函数。从Windows优先级到Windows内部的数值优先级的映射关系如图6.13所示。

虽然一个进程只有一个基本优先级值，但是，每个线程有两个优先级值：当前的和基本的。调度决定是根据当前优先级来做出的。正如在下一节关于优先级提升（priority boost）的介绍中所说的那样，在特定的情况下，系统在很短的周期内，会在动态范围（1至15）之内增加线程的优先级。Windows从来不会在实时优先级范围内（16至31）调整线程的优先级，所以，它们的基本优先级和当前优先级总是相同的。

一个线程的初始基本优先级是从进程的基本优先级继承得来的。在默认情况下，一个进程从创建它的进程那里继承得到基本优先级。这种行为可以在*CreateProcess*函数中被改变，或者使用命令行START命令也可以改变。在进程被创建以后，也可以通过使用*SetPriorityClass*函数或者暴露了同样功能的各种工具，比如任务管理器和进程管理器（在进程上右键点击，然后选择一个新的优先级类别），来改变它的优先级。例如，你可以降低一个CPU密集型进程的优先级，而它不会妨碍正常的系统行为。改变一个进程的优先级，也改变了线程优先级向上或向下，但它们的相对设置仍然不变。然而，改变一个进程内单独线程的优先级通常并没有意义，因为除非你编写了该程序，或者有源代码，否则你并不知道每个单独的线程在做什么事情，而且，改变线程的相对重要性可能会导致程序不像预期的那样来工作。

www.TopSage.com

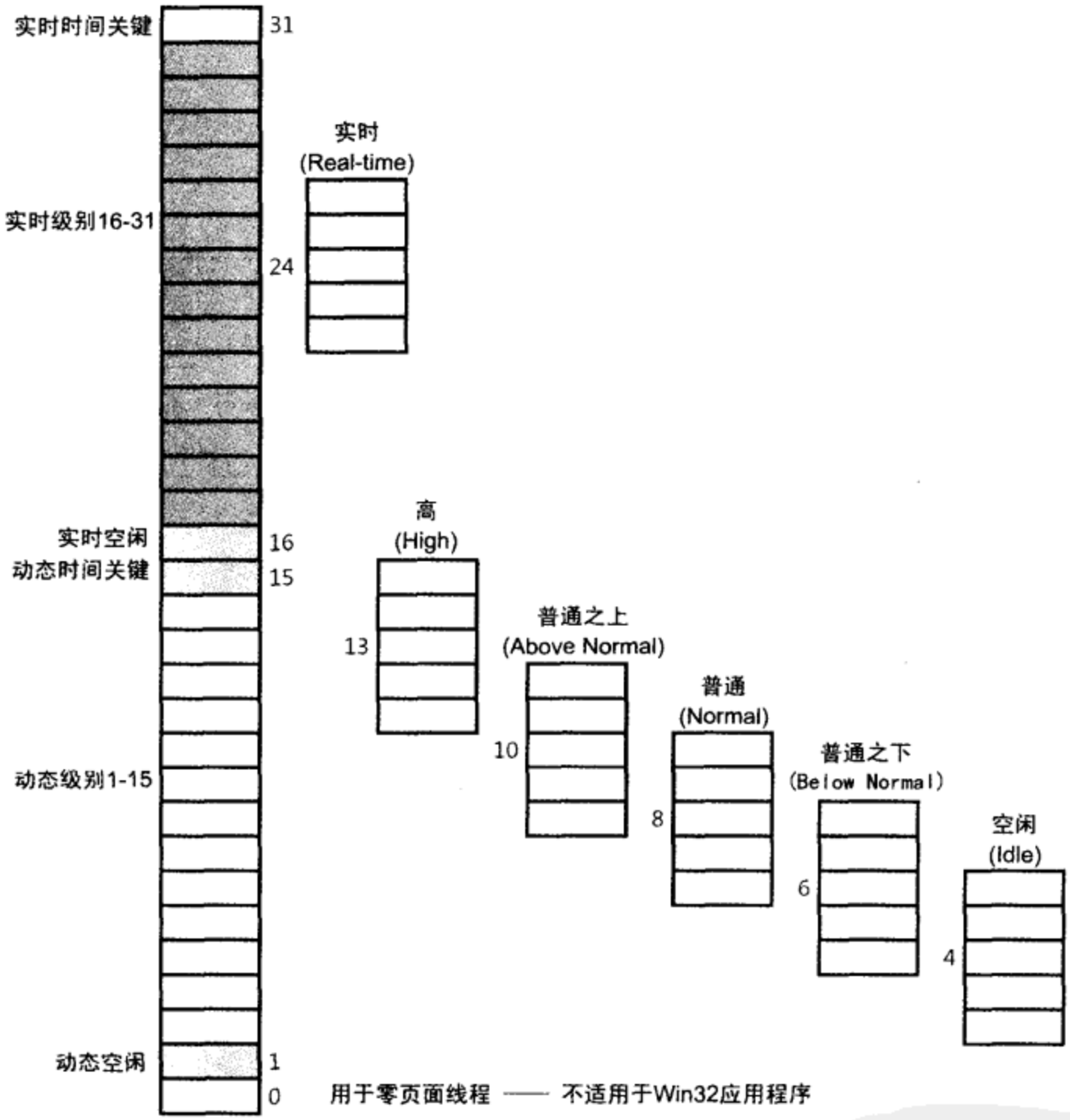


图6.13 从Windows内核优先级到Windows API之间的映射关系

在正常情况下，进程基本优先级（同样也是起始的线程基本优先级）被默认设置为每个进程优先级范围的中间值（24、13、10、8、6或4）。然而，有些Windows系统进程（比如会话管理器、服务控制器，以及本地安全认证服务器）的基本进程优先级略高于普通类别的默认值（8）。正是这一略高一点的默认值，保证了这些进程内的线程都从一个高于默认值8的优先级开始。这些系统进程使用一个内部系统调用（*NtSetInformationProcess*），来将它的进程基本优先级设置为一个不同于普通的默认初始基本优先级的数值。

www.TopSage.com

Windows调度API

表6.13列出了与线程调度有关的Windows API函数（有关更多的信息，请参见Windows API参考文档）。

表6.13 与调度有关的API，以及它们的功能

API	功 能
Suspend/ResumeThread	挂起一个线程，或者恢复一个暂停的线程
Get/SetPriorityClass	返回或者设置一个进程的优先级类别（基本优先级）
Get/SetThreadPriority	返回或者设置一个线程的优先级(相对于它的进程基本优先级)
Get/SetProcessAffinityMask	返回或者设置一个进程的亲和性掩码
SetThreadAffinityMask	设置一个线程的亲和性掩码（一定是该进程的亲和性掩码的一个子集）为一组特定的处理器，限制该线程只能在这些处理器上运行
SetInformationJobObject	设置一个作业的属性；有些属性会影响调度，比如亲和性和优先级（关于作业对象的介绍，请参见本章后面的“作业对象”一节）
GetLogicalProcessorInformation	返回有关处理器硬件配置的细节信息（针对超线程和NUMA系统）
Get/SetThreadPriorityBoost	返回或者设置“Windows临时提升一个线程的优先级”的能力（此能力仅适用于动态优先级范围内的线程）
SetThreadIdealProcessor	针对一个特定的线程建立一个首选的处理器，但并不限制该线程只能在此处理器上运行
Get/SetProcessPriorityBoost	返回或者设置当前进程默认的优先级提升控制状态（当一个线程被创建时，可通过此函数设置线程的优先级提升控制状态）
SwitchToThread	将执行权让给另一个已经准备好在当前处理器上运行的线程（在优先级1或者更高级别上）
Sleep	将当前线程置入等待状态，让它等待一段指定的时间间隔（以毫秒为单位[msec]来表示）。零值表示放弃该线程时限的剩余部分
SleepEx	使得当前线程进入一种等待状态，直至要么完成一个I/O完成回调、要么一个APC被排队到该线程中，或者指定的时间间隔结束

www.TopSage.com

有关的工具

下表列出了与线程调度相关的工具。你可以通过许多不同的工具，比如任务管理器、进程管理器（Process Explorer）、Pview或者Pviewer，来改变（或者查看）基本的进程优先级。注意，通过进程管理器，你可以杀掉一个进程中单独的线程。当然，这种做法应该绝对小心。

利用性能工具、进程管理器、Pslist、Pview、Pviewer和Pstat，你可以查看单独的线程优先级。增加或者降低一个进程的优先级可能会非常有用，但是，调整一个进程内部单独线程的优先级通常并没有意义，因为只有一个人完全理解了该程序，才能够理解该进程内部各个线程之间的相对重要性。

对象	Taskman	Perfmon	Pviewer	Pview	Pstat	KD *!thread*	进程管理器
进程优先级类别	✓		✓	✓			✓
进程基本优先级		✓			✓		✓
线程基本优先级		✓					✓
线程当前优先级		✓	✓	✓	✓	✓	✓

为一个进程指定起始优先级类别的惟一途径是，在Windows命令提示符下使用*start*命令。如果你想要让一个程序每次启动的时候都有一个指定的优先级，那么，你可以定义一个快捷文件来使用start命令，并且以**cmd /c**来开始这一命令。这样会运行命令提示窗口，执行命令行中的命令，然后终止命令提示窗口。例如，为了以低的进程优先级来运行Notepad，快捷方式命令应该是**cmd /c start /low notepad.exe**。

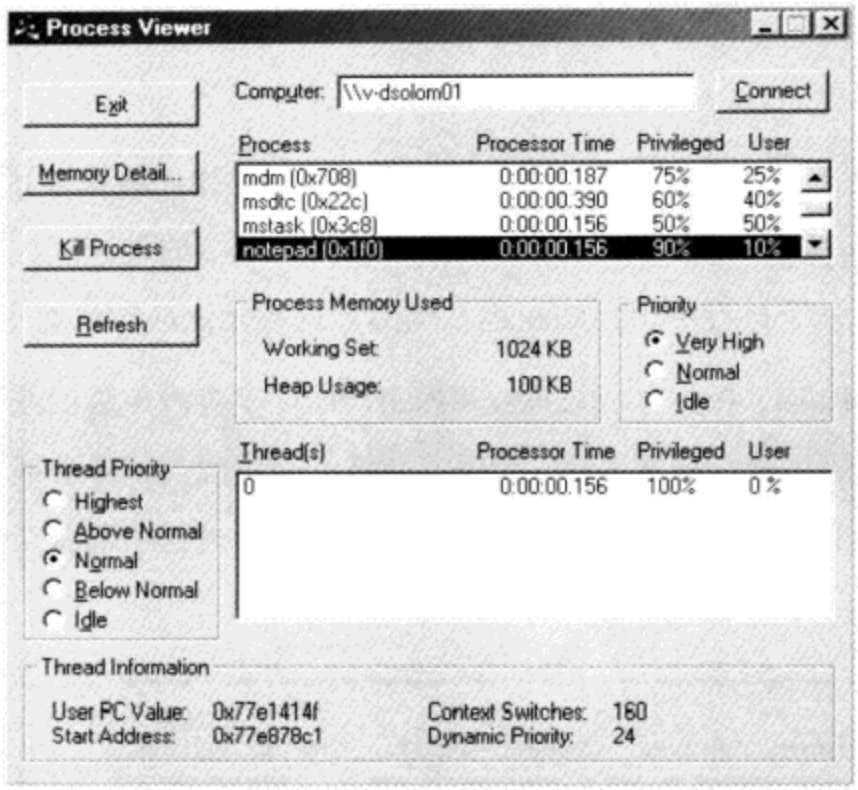


实验：检查并指定进程和线程的优先级

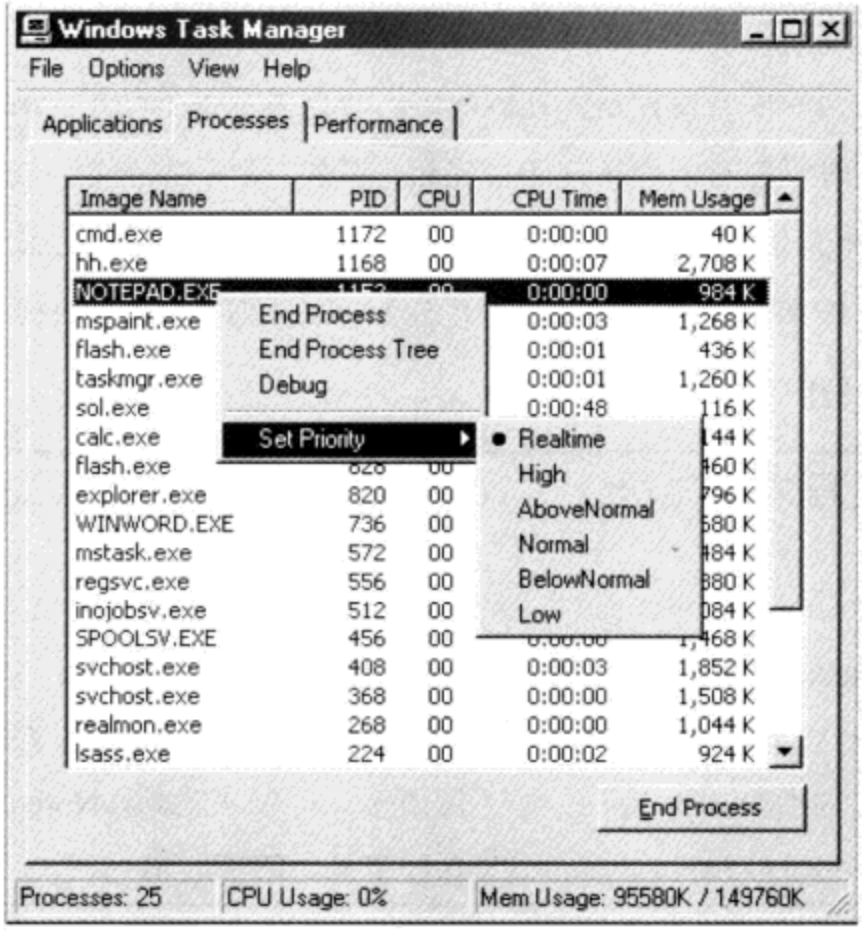
请试一试以下的实验。

1. 在命令行提示符下，输入start /realtime notepad。Notepad应该被打开。
2. 或者运行进程管理器，或者运行Windows支持工具箱（Support Tools）中的进程查看工具（Pviewer.exe），从进程列表中选择Notepad.exe，如下图所示。可以注意到，Notepad中的线程的动态优先级（Dynamic Priority）是24。这符合图6.13中显示的实时值。

www.TopSage.com



3. 任务管理器可以显示类似的信息。按下Ctrl+Shift+ESC启动任务管理器，然后转到Processes页面。右键点击Notepad.exe进程，再选择Set Priority选项。你可以看到Notepad的进程优先级类别是Realtime（实时），如下图中的对话框所示。



www.TopSage.com

Windows系统资源管理器

Windows Server 2003企业版（Enterprise Edition）和Windows Server 2003的数据中心版本（Datacenter Edition）包含一个称为Windows系统资源管理器（WSRM, Windows System Resource Manager ）的可选安装组件。它使得管理员可以配置一些策略，在这些策略中指定进程的CPU利用率、亲和性设置和内存限制（物理的和虚拟的）。而且，WSRM可以生成资源利用报告，用于客户服务协议中的记账和核查。

这些策略可以应用于特定的应用程序（通过映像文件名匹配的方式，可以包含特定的命令行参数，也可以不包含命令行参数）、用户或者组。管理员还可以安排这些策略的作用时间，可以让它们在特定的时间段内生效，也可以一直有效。

当你设置了一个资源分配策略来管理某些特定的进程以后，WSRM服务监视这些被管理进程的CPU消耗情况，当这些进程不符合其预设的CPU分配方案时，调整它们的基本优先级。

物理内存的限制是这样来实现的：通过*SetProcessWorkingSetSizeEx*函数来设置一个硬工作集最大值。而虚拟内存限制则是通过该服务检查这些进程所消耗的私有虚拟内存来实现的（有关这些内存限制的详细解释，请参见第7章）。 WSRM可以被配置成：当此限制被超过时，或者杀掉进程，或者在事件日志中写一条纪录。通过这种行为，我们可以在一个进程消耗掉系统中所有可用的已提交（committed）虚拟内存以前检测它的内存泄漏。注意，WSRM内存限制并不适用于AWE（Address Windowing Extensions，地址窗口扩展）内存、大页面内存，或者内核内存（非换页的或者换页的内存池）。

实时优先级

你可以在动态范围内提升或者降低任何一个应用程序中的线程优先级。然而，你必须有“increase scheduling priority（增加调度优先级）”的特权才能进入实时优先级范围。务必清楚，许多重要的Windows内核模式系统线程都在实时优先级范围内运行，所以，如果这些线程花了许多额外的时间在这个范围内运行，那么，它们可能会阻塞关键的系统函数（比如在内存管理器、缓存管理器或者其他的设备驱动程序中的函数）。

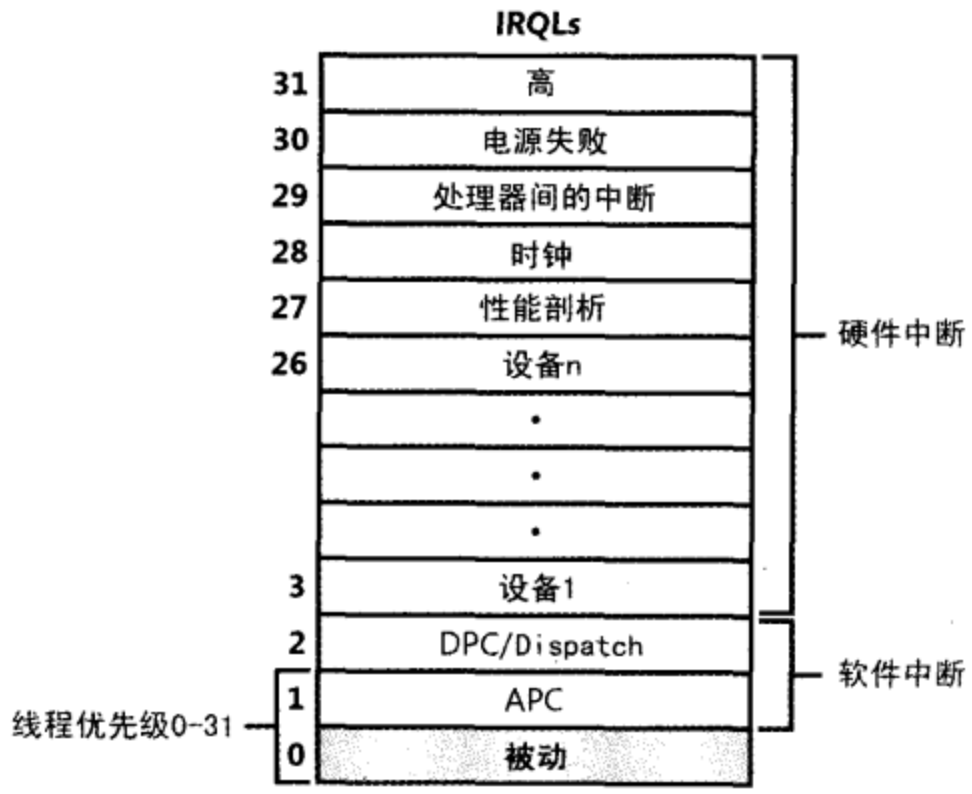


> 注 正如下页图中显示的x86中断请求级别（IRQL），尽管Windows有一组被称为*实时*（*real-time*）的优先级，但是，它们并不是通常意义上的实时定义。这是因为，Windows并没有提供真正的实时操作系统设施，比如可保证的中断延迟，或者某一种“使线程能获得一段可保证的执行时间”的办法。有关更多的信息，请参见第3章中的辅助章节 “Windows和实时处理过程”，或者MSDN Library中的文章“Real-Time Systems and Microsoft Windows NT”。

www.TopSage.com

中断级别和优先级别

正如下图中所示，在一般情况下线程运行在IRQL 0或者1级别上（关于Windows如何使用中断级别，请参见第3章中的介绍）。用户模式的线程总是运行在IRQL 0上。由于这个原因，用户模式的线程，不管其优先级如何，总是不会阻塞硬件中断（不过，高优先级的实时线程可以阻塞重要系统线程的执行）。只有内核模式的APC才在IRQL 1上执行，因为它们中断了一个线程的执行（有关APC的更多信息，请参见第3章）。 然而，运行在内核模式中的线程可以将IRQL提升到更高的级别上，比如当它执行一个涉及线程分发的系统调用时。



线程状态

在你完全弄清楚线程调度算法以前，你需要理解一个线程可能进入的各种执行状态。图6.14显示了在Windows 2000和Windows XP上线程的状态转移（这里显示的数值代表了线程状态性能计数器的值）。本节后面介绍了每一个状态转移所涉及的细节情况。

www.TopSage.com

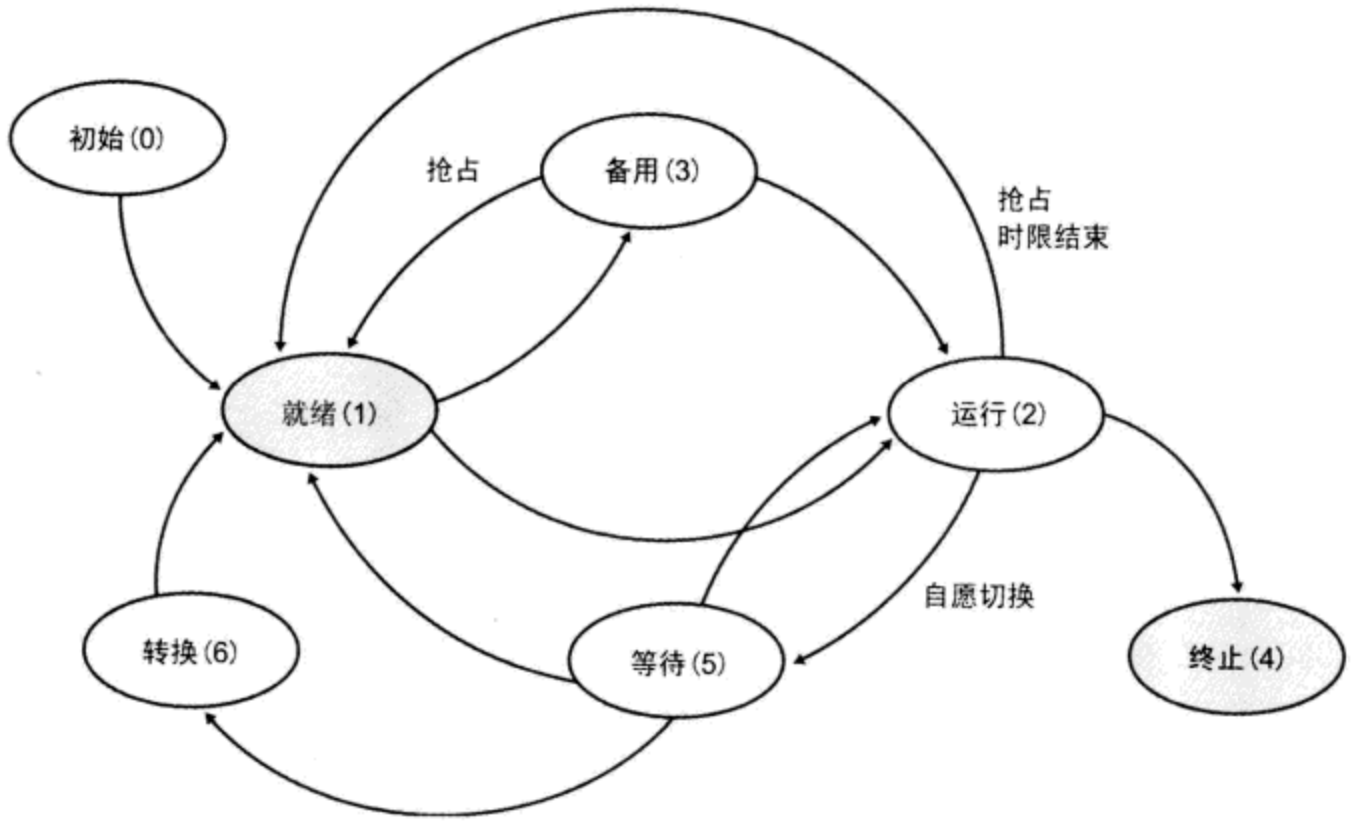


图6.14 Windows 2000和Windows XP上的线程状态

这些线程状态如下。

- **就绪**（Ready） 处于就绪状态中的线程正在等待执行。当分发器需要找一个线程来执行时，它只考虑就绪状态中的线程。
- **备用**（Standby） 处于备用状态中的线程已经被选中，接下来将在某个特定的处理器上运行。当正确的条件满足时，分发器执行一次环境切换，切换到该线程上。对于系统中的每个处理器，只能有一个线程可以处于备用状态。请注意，一个线程真正被执行以前，它可能会从备用状态中被抢占掉（比如，在备用线程开始执行以前，一个更高优先级的线程变得可以运行了）。
- **运行**（running） 一旦分发器将环境切换到一个线程，于是，该线程即进入运行状态，并开始执行。该线程一直执行下去，直到它的时限结束（同一优先级上的另一个线程准备运行）、它被一个更高优先级的线程抢占掉、它终止、它放弃执行，或者它自愿进入等待状态。
- **等待**（Waiting） 一个线程可以通过几种方法进入到等待状态中：它可以自愿等待一个对象以便同步其执行过程、操作系统可以代替该线程进入等待（比如为了解决一个换页I/O），或者环境子系统可以指示该线程自行挂起。当线程的等待结束时，根据优先级的不同，该线程或者立即开始运行，或者被回转到就绪状态。

www.TopSage.com

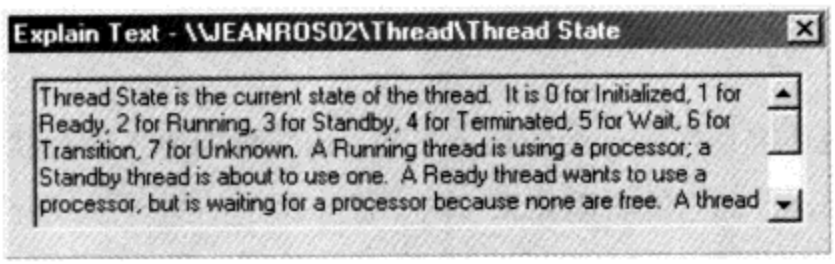
- **转换**（Transition） 如果一个线程已经准备好执行，而它的内核栈被换出了内存，那么该线程进入转换状态。一旦它的内核栈被换回到内存中，该线程将进入就绪状态。
- **终止**（Terminated） 当一个线程完成执行时，它进入终止状态。一旦该线程被终止，执行体线程块（位于非换页内存池中的用来描述该线程的数据结构）可能已被释放，可能还没有被释放（关于何时删除该对象，对象管理器会设置相应的策略）。
- **初始**（Initialized） 当一个线程被创建时，内部使用此状态。



实验：线程调度状态的改变

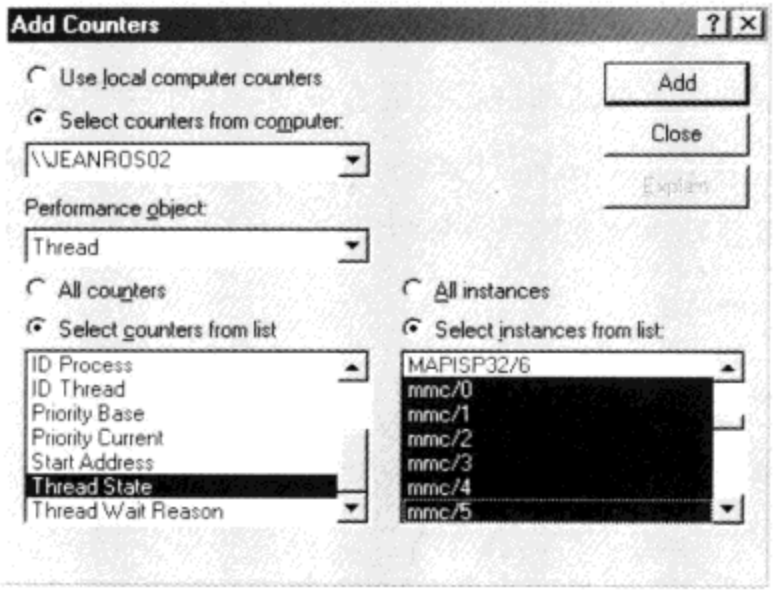
利用Windows中的性能工具，你可以观察线程调度状态的改变。当你正在调试一个多线程应用程序时，如果你对于该进程中正在运行的线程的状态不能确定的话，这一工具可能非常有用。要想利用性能工具来观察线程调度状态的改变，请按照以下步骤进行。

1. 运行Microsoft的Notepad工具（Notepad.exe）。
2. 启动性能工具（Performance tool），做法是，你从Start菜单中选择Programs，然后从Administrative Tools菜单中选择Performance。
3. 如果你不在图表视图（chart view）中，则选择图表视图。
4. 在图上右键点击，选择Properties。
5. 点击Graph标签，将图表垂直比例尺（Veritical Scale）的最大值（Maximum）设置为7（正如你将从性能计数器的说明文字中所看到的那样，线程状态编号为从0到7）。点击OK。
6. 在工具条上点击Add按钮，以打开Add Counters对话框。
7. 选择Thread性能对象，然后选择Thread State计数器。点击Explain按钮，可以看到这些状态值的定义。



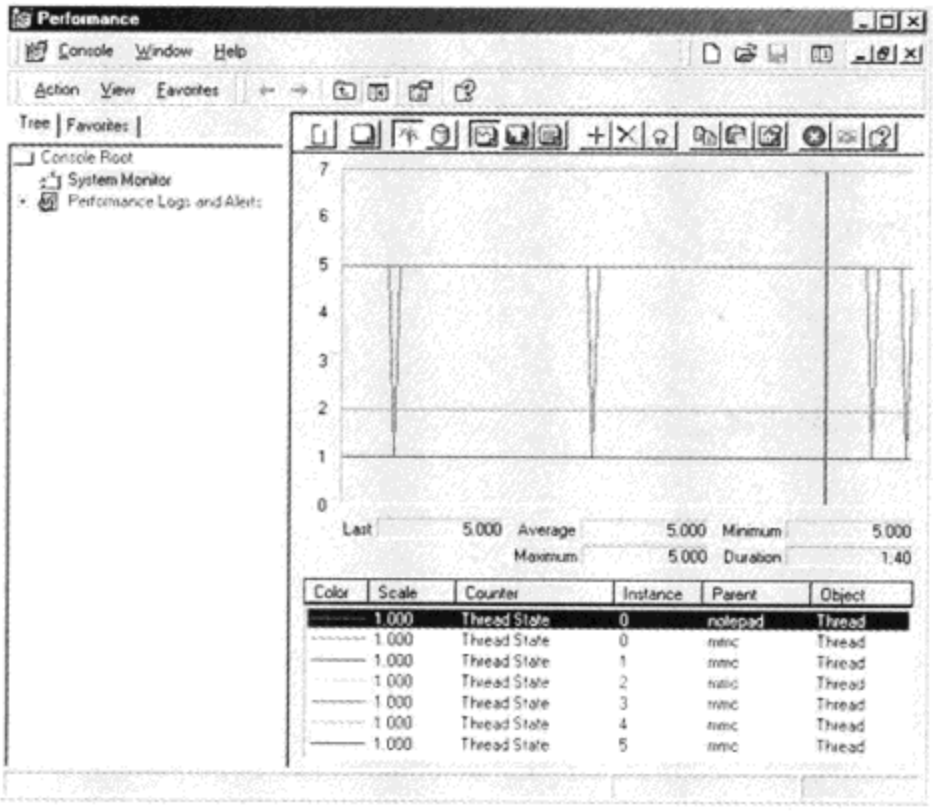
8. 在Instances框中，向下滚动，直至你看到了Notepad进程（notepad/0）；将它选中，并点击Add按钮。
9. 在Instances框中滚回到Mmc进程（运行系统监视器[System Monitor]的Microsoft Management Console进程），选择所有的线程（mmc/0、mmc/1，等等），然后点击Add按钮，将它们加入到图表中。在点击Add以前，你应该看到类似于以下的对话框：

www.TopSage.com



10. 现在点击Close按钮，关闭Add Counters对话框。

11. 你应该可以看到Notepad线程的状态是5（在下图中，很靠近上面的一条线）。正如在第5步的说明文字中所解释的那样，5代表了等待状态（因为该线程正在等待GUI输入）。



12. 你可以注意到，在Mmc进程中的一个线程（正在运行性能工具插件）正处于运行状态（数值2）。这是当前正在查询线程状态的那个线程，所以它总是被显示为处于运行状态。

13. 你永远也不会看到Notepad处于运行状态（除非你是在一个多处理器系统上），因为当Mmc在收集你所监视的这些线程的状态时，Mmc总是处于运行状态。

www.TopSage.com

图6.15显示了Windows Server 2003上的线程状态图。注意，这里加入了一种新的称为**延迟的就绪**（*deferred ready*）的状态。这种状态用来描述那些已经被选中要在某个特定的处理器上运行，而尚未被调度的线程。引入了这一新的状态以后，内核可以使调度数据库被系统全局范围内锁住的时间变得最短（在“多处理器分发器数据库”一节中对这一过程有进一步的解释）。

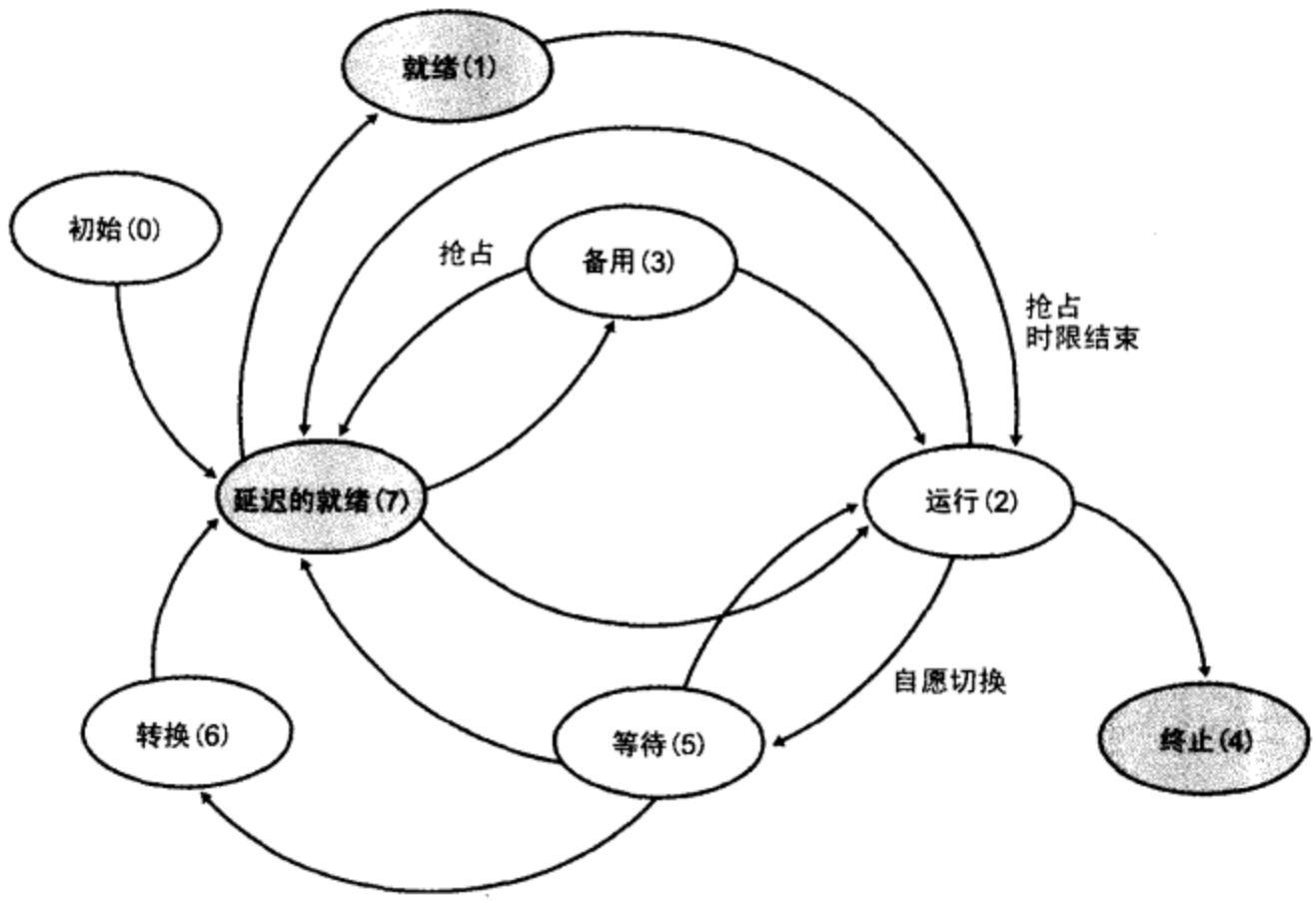


图6.15 Windows Server 2003中的线程状态

分发器数据库

为了做出有关线程调度的各种决定，内核维护了一组数据结构，它们合起来称为**分发器数据库**（*dispatcher database*），如图6.16所示。分发器数据库跟踪记录了哪些线程正在等待执行，哪些处理器正在执行哪些线程。



> **注** 单处理器系统上的分发器数据库结构与多处理器版本的Windows 2000和Windows XP系统上的分发器数据库结构是相同的，但是与Windows Server 2003系统上的不相同。在后文的“多处理器系统”一节中解释了这些差异，以及当Windows在多处理器系统上选择待运行的线程时，选择方法上的不同之处。

www.TopSage.com

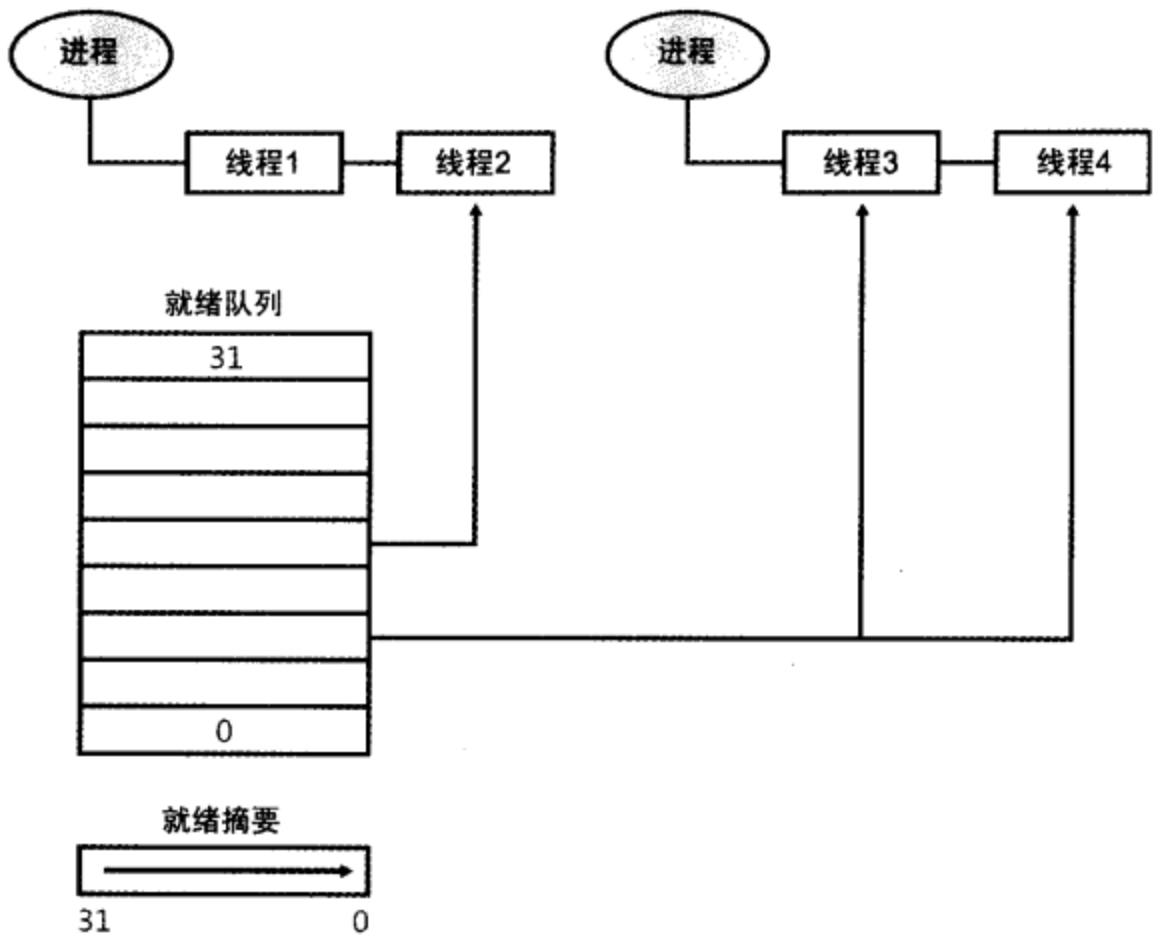


图6.16 分发器数据库（单处理器和Windows 2000/XP多处理器）

分发器的**就绪队列**（*ready queue*）（*KiDispatcherReadyListHead*）包含了那些处于就绪状态、正在等待被调度执行的线程。32个优先级别中的每一个都有一个队列。为了更加快速地选择一个线程来运行或抢占，Windows维护了一个32位的位掩码，称为**就绪摘要**（*ready summary*）（*KiReadySummary*）。每一位一旦被设置，就表明对应优先级别的就绪队列中有一个或者多个线程（位0代表优先级0，依此类推）。

表6.14列出了在单处理器系统上与线程调度相关的内核变量。

表6.14 与线程调度相关的内核变量

变 量	类 型	说 明
KiReadySummary	位掩码（32位）	位掩码，针对包含一个或者多个就绪线程的优先级别而设置
KiDispatcherReadyListHead	包含32个列表项的数组	32个就绪队列的列表头

在单处理器系统上，分发器数据库是通过将IRQL提升到DPC/Dispatch级别和SYNCH_LEVEL（两者均被定义为级别2）来同步的（关于中断优先级别的说明，请参见第3章中的“陷阱分发”一节）。按照这种方法来提升IRQL，可以防止其他的线程打断线程分发过程，因为在正常情况下线程运行在IRQL 0或者1上。在一个多处理器系统上，除了提升IRQL

www.TopSage.com

以外，还需要做更多的事情，因为在同一时刻，每个处理器都可以提升到此IRQL上并试图在分发器数据库上进行操作。关于在多处理器系统上Windows如何对分发器数据库进行同步访问，请参考本章后面的“多处理器系统”一节中的介绍。

时限

正如本章前面多次提到的，时限是指一个线程在“Windows检查是否有另一个同样优先级别的线程正在等待运行”之前所获得运行的时间长度。如果一个线程完成了它的时限，并且在它的优先级上没有其他的线程，那么，Windows允许该线程再运行另一个时限。

在Windows 2000 Professional和Windows XP上，在默认情况下，线程运行2个时钟间隔；在Windows Server系统上，在默认情况下，线程运行12个时钟间隔（后面我们将会介绍如何改变这些值）。之所以在服务器系统上默认值更长一些，是因为要使环境切换的次数尽可能地减少。有了一个长的时限以后，如果服务器应用程序一旦接收到客户请求而被唤醒，则它有更好的机会在时限结束以前完成此请求，并回到等待状态中。

时钟间隔的长度随着硬件平台的不同而有所不同。时钟中断的频率取决于HAL，而不是内核。例如，大多数x86单处理器系统的时钟间隔是10毫秒左右，而大多数x86多处理器系统则是15毫秒左右（实际的时钟速率并不恰好等于毫秒的整数倍——参看下面的实验可以检查实际的时钟间隔）。



实验：确定时钟间隔频率

Windows的GetSystemTimeAdjustment函数返回时钟间隔。为了确定时钟频率，从*www.sysinternals.com*下载Clockres程序，然后运行该程序。下面是在一台单处理器x86系统上的输出：

```
C:\>clockres

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

The system clock interval is 10.014400 ms
```

时限计算

每个进程的内核进程块中都有一个时限值。当给一个线程分配一个新的时限时，就会用到该值。一个线程运行时，在每一个时钟间隔，它的时限都会被减少。如果没有剩余的线程时限了，则会触发时限结束处理过程。

www.TopSage.com

如果在同一优先级上另外一个线程正在等待运行，则将环境切换到就绪队列中的下一个线程中。注意，当时钟中断打断了一个DPC或者另一个正在进行中的中断时，处于运行状态的线程，尽管没有运行完整个时钟间隔，但它的时限仍然被扣除。如果不这样做，而且设备中断或者DPC恰好发生在时钟间隔定时器中断之前的话，则线程的时限可能总是减不下来。

在内部，此时限值被保存为3个时钟嘀嗒的一个倍数。这意味着，在Windows 2000和Windows XP系统上，在默认情况下，线程的时限值为6（2*3），而在Windows Server系统上，时限值为36（12*3）。每次时钟中断发生时，时钟中断例程从线程时限中减去一个固定值（3）。

时限的内部存储表示方式之所以是3个时限单元（/时钟嘀嗒）的倍数，而不是一个单元的倍数，是因为要允许在等待完成时可以部分地减少时限。假设一个线程的当前优先级小于16，基本优先级小于14，当它在执行一个等待函数（比如*WaitForSingleObject*和*WaitForMultipleObjects*），并且此等待函数可立即被满足（譬如，根本不必等待）的时候，它的时限被减少1个时限单元。通过这种方式，等待线程最终用完它们的时限。

如果等待函数并不立即被满足，则低于优先级16的线程也让它们的时限减去1个单元（除非该线程被唤醒起来执行一个内核APC，因为在这种情况下，在该等待被实际满足以后，等待代码将扣除此时限）。然而，在真正减去这个时限单元以前，如果该线程位于优先级14或者之上，那么，它的时限被重置为一个完整的轮次。对于那些运行在低于14优先级的线程，如果它们不是以特定的优先级提升（boost）来运行（比如，针对前台进程的做法，或者CPU饿死的情形下），并且，如果（作为一个取消等待操作的结果） 它们正在接收一个优先级提升，那么，这些线程的时限也要被重置（下一小节解释优先级提升）。

这种部分减少时限的特性，针对的是当一个线程在时钟间隔定时器激发之前进入等待状态的情形。如果不做这种调整的话，这些线程有可能永远也不会让它们的时限减少了。例如，如果一个线程先运行，再进入等待状态，再运行，再进入另一个等待状态，而当时钟间隔定时器激发的时候它总也不是当前运行的线程，则系统永远也不会从它的时限中扣除它所运行的时间。注意，这不适用于零超时等待的情形，而只适用于那些“因为在等待的时刻所有的条件都是满足的，所以并不要求等待”的等待操作。

控制时限

你可以为所有的进程改变线程时限，但是你只能选择两种设置之一：短设置（2个时钟嘀嗒，客户机的默认设置），或者长设置（12个时钟嘀嗒，服务器系统的默认设置）。

www.TopSage.com



> 注 在运行长时限的系统上，通过使用作业对象，你可以为作业中的进程选择其他的时限值。有关作业对象的更多信息，请参见本章后面的“作业对象”一节。

要想在Windows 2000上改变时限设置，请右键点击“My Computer”，选择“Properties”（或者到控制面板中，打开System设置小程序），点击“Advanced”标签，然后点击“Performance Options”按钮。你将会看到如图6.17所示的对话框。

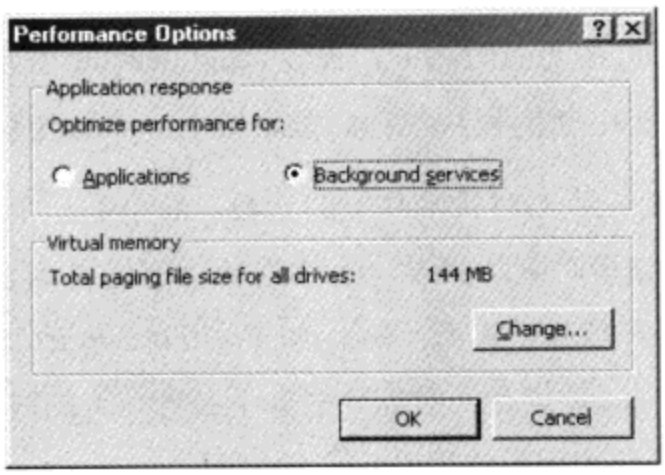


图6.17 Windows 2000上的时限配置

要想在Windows XP和Windows Server 2003上改变时限设置，请右键点击“My Computer”，选择“Properties”，点击“Advanced”标签，然后在“Performance”区域点击“Setting”按钮，最后点击“Advanced”标签。Windows XP和Windows Server 2003所显示的对话框略有所不同，如图6.18所示。

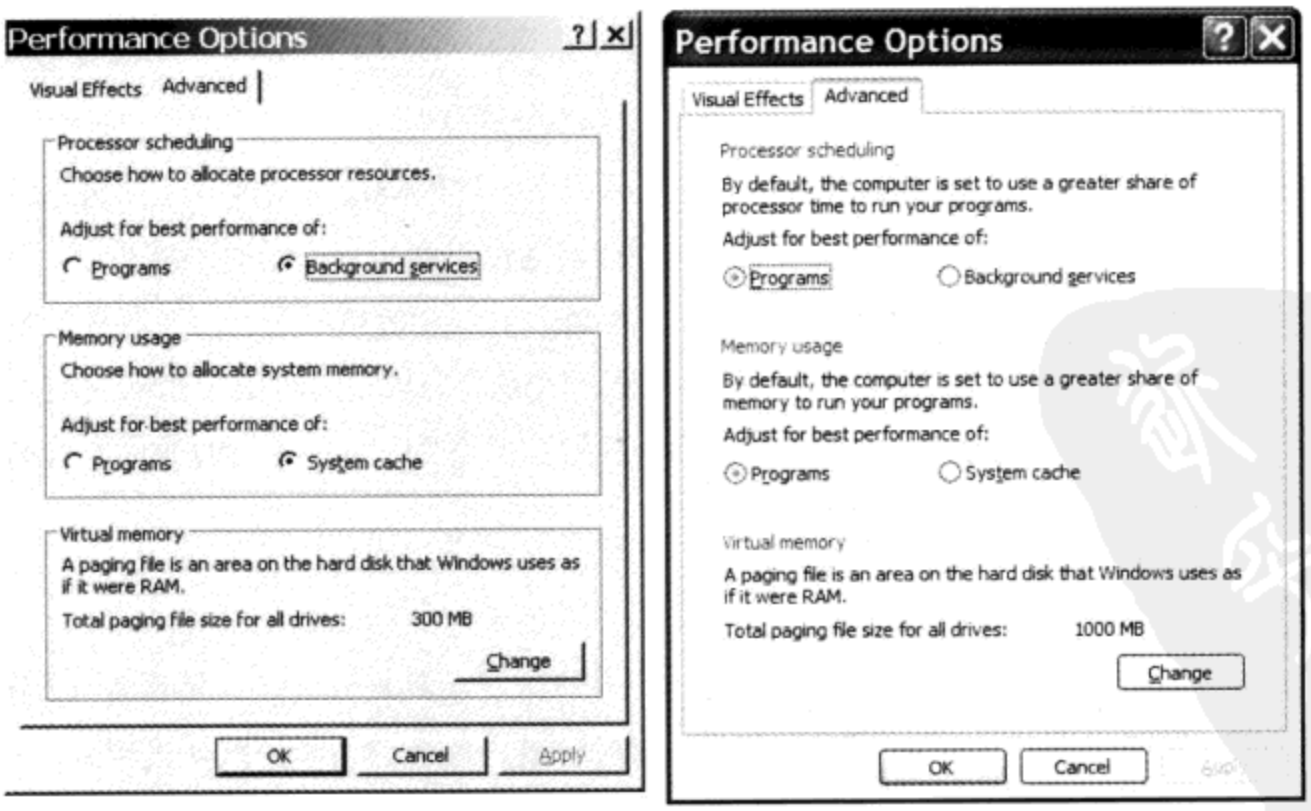


图6.18 Windows XP和Windows Server 2003上的时限配置

www.TopSage.com

“Programs”设置（在Windows 2000上称为“Applications”）指定了使用短的、可变的时限——这是Windows 2000 Professional和Windows XP的默认设置。如果你在Windows Server系统上安装了终端服务，并且将该服务器配置成应用服务器，那么，这一设置将会被选中，因而终端服务器上的用户将可以与他们在桌面系统或客户系统上惯有的时限设置相同。如果你正在运行Windows Server，并把它作为你的桌面操作系统，那么你也可以手工选择这一设置。

“Background Services”选项指定了使用长的、固定的时限——这是Windows Server系统的默认设置。在一台工作站系统上你若要选择这一设置，则惟一可能的理由是，你将这台工作站用做一个服务器系统。

在“Programs”和“Background Services”设置之间的另一个差别是，它们对于前台进程中的线程的时限有不同的影响。下一小节将解释这一点。

时限增大（quantum boosting）

在Windows NT 4.0以前，在工作站或者客户系统上，当一个窗口被带到前台时，此前台进程（即此焦点窗口所属线程所在的进程）中的所有线程都接收到优先级提升值2。如果该进程中的任何一个线程拥有此前台窗口，则此优先级提升值仍然有效。这种做法的问题是，如果你启动一个长时间运行且CPU密集型的进程（比如电子数据表重新计算），然后切换到另一个CPU密集型的进程（比如一个计算机辅助设计工具、图形编辑器或者游戏），则现在后台运行的进程将会得到很少的CPU时间，甚至得不到CPU时间，因为原来的前台进程只要还在前台，它的线程优先级就被增加了2（假设这两个进程中的线程的基本优先级是相同的）。

从Windows NT 4.0工作站开始，这种默认的行为有了不同，不再提升优先级，而是前台进程中线程的时限变成了3倍。因此，前台进程中的线程在运行时，其时限值为6个时钟嘀嗒，而其他进程中的线程只使用默认的2个时钟嘀嗒的时限值。通过这种方式，当你从一个CPU密集型的进程切换离开时，新的前台进程将按比例获得更多的CPU时间，因为当它的线程运行时，它们将获得比后台线程更多的CPU时间（同样地，假设在前台进程和后台进程中所有线程的优先级是相同的）。

注意，对于那些在Performance Options（如上一小节所讲述）中配置为Programs（在Windows 2000中为Applications）的系统，这种关于时限的调整只适用于那些优先级高于空闲级别（Idle）的进程。对于配置为Background Services（在Windows Server系统上这是默认设置）的系统，前台进程的线程时限没有变化。

www.TopSage.com

时限设置注册表值

前面介绍的用来控制时限设置的用户界面实际上修改了注册表值HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation。除了指定线程时限的相对长度（短或长）以外，此注册表值也定义了前台进程中的线程是否应该让它们的时限增大（如果增大的话，又该增大多少）。此值包含了6位，分成3个2位域，如图6.19所示。

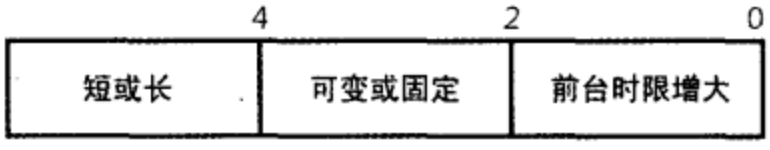


图6.19 Win32PrioritySeparation注册表值中的域

图6.19中显示的域定义如下。

- **短或长** 设置为1指定长时限，2指定短时限。0或者3表明将使用默认设置（对于Windows 2000 Professional和Windows XP为短，对于Windows Server系统为长）。
- **可变或固定** 设置为1意味着改变前台进程的时限值，2意味着不改变前台进程的时限值。设置为0或者3意味着将使用默认设置（对于Windows 2000 Professional和Windows XP是可变的，对于Windows Server系统是固定的）。
- **前台时限增大** 该域（存储在内核变量*PsPrioritySeparation*中）的值必须为0、1或者2。它是一个索引，指向一个名为*PspForegroundQuantum*的3元素字节数组，利用它可以得到前台进程中线程的时限。后台进程中线程的时限可以从该时限表中的第一项取到。表6.15显示了*PspForegroundQuantum*的可能设置。

表6.15 时限值

	短			长		
可变	6	12	18	12	24	36
固定	18	18	18	s	36	36

注意，当你在使用前面所讲述的Performance Options对话框时，你只能从两种组合中选择其一：短时限且前台时限乘3，或者长时限但前台线程的时限不变。然而，通过直接修改Win32PrioritySeparation注册表值，你可以选择其他的组合。

www.TopSage.com

调度情形

Windows根据线程优先级来回答“谁该得到CPU？”但实际上这又是如何做到的呢？下面的小节说明了优先级驱动的抢先式多任务调度是如何在线程级别上完成的。

自愿切换

首先，一个线程可能会通过调用某个Windows等待函数（比如*WaitForSingleObject*或*WaitForMultipleObjects*）来等待某个对象（比如一个事件、互斥体、信号量、I/O完成端口、进程、线程、窗口消息，等等），从而进入等待状态，主动放弃对处理器的使用。第3章详细地讲述了有关等待对象的细节。

图6.20显示了一个线程进入等待状态，Windows选择一个新的线程来运行。

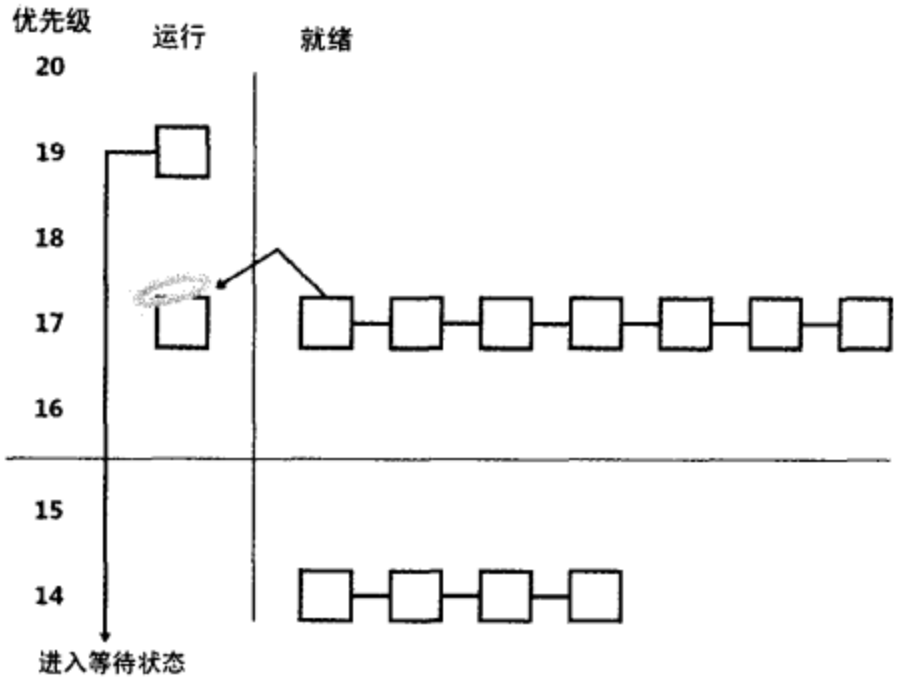


图6.20 自愿切换

在图6.20中，最上边的块（线程）自愿放弃处理器，所以就绪队列中的下一个线程可以运行（在“运行”这一列中用一圈光环来表示）。虽然从图中来看，放弃处理器的线程的优先级似乎被降低了，但其实不是这样的，它只是被移到了该线程正在等待的对象的等待队列中。该线程剩余的时限怎么办呢？当一个线程进入等待状态时，它的时限值不会被重置——实际上，正如前面所解释的，当该等待被满足时，该线程的时限值被减少1个时限单元，等于减少三分之一个时钟间隔（那些运行在优先级14或更高级别上的线程除外，对于这些线程，在等待结束以后它们的时限被重置为一个满轮次）。

www.TopSage.com

抢先模式

在这种调度情形中，当一个高优先级的线程变成就绪时，低优先级的线程被抢占。之所以发生这样的情形，有以下两个理由。

- 一个高优先级线程的等待完成了（另一个线程正在等待的事件发生了）。
- 一个线程的优先级被增大或减小。

以上无论哪一种情形，Windows必须确定：当前正在运行的线程是应该继续运行，还是应该被抢占掉，以便让更高优先级的线程运行。



> **注** 用户模式中的线程可以抢占掉在内核模式中运行的线程——线程当前所在的模式并没有关系。线程优先级是决定因素。

当一个线程被抢占时，它被放到它刚才运行时所在优先级的就绪队列的头部。图6.21显示了这种情形。

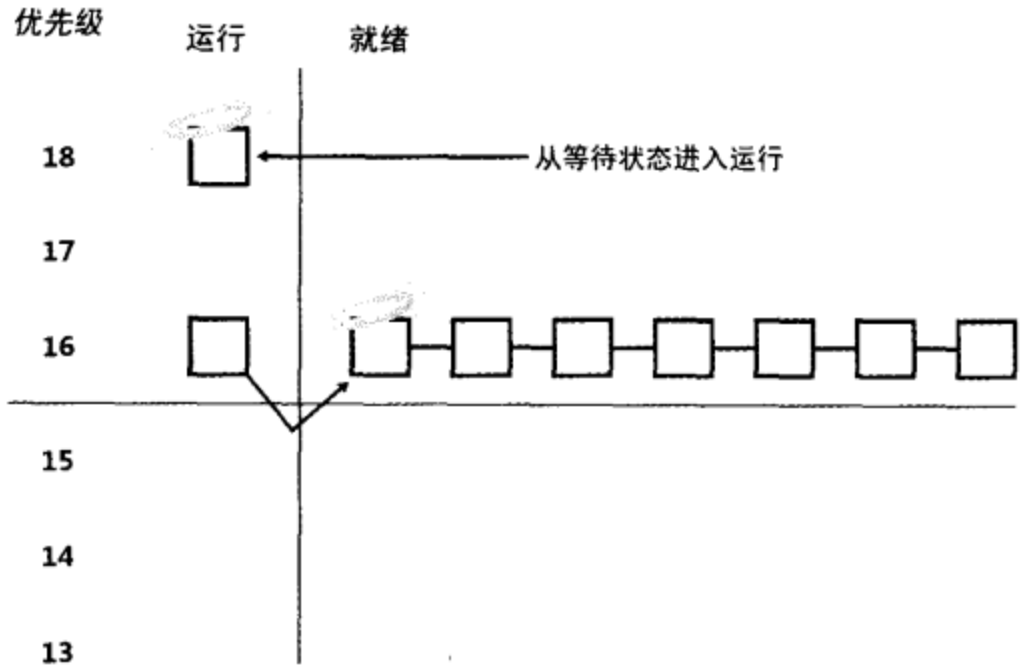


图6.21 抢先式线程调度

在图6.21中，一个优先级为18的线程从等待状态中醒来，重新拥有CPU，因而使得当前已经在运行的线程（在优先级16）被转移到就绪队列的头部。可以注意到，这个被转移的线程并不是插入到队列的末尾，而是开始处；当抢先的线程完成了运行以后，被转移的线程可以继续完成它的时限。

www.TopSage.com

时限结束

当正在运行的线程用完了它的CPU时限时，Windows必须决定该线程的优先级是否应该被降低，然后决定是否应该调度另一个线程到当前处理器上来。

如果线程优先级被降低，则Windows寻找一个更加合适的线程来调度（例如，这个更加合适的线程是，比当前运行线程的新优先级还要更高的优先级的就绪队列中的一个线程）。 如果线程的优先级没有被降低，并且在同一优先级别的就绪队列中还有其他的线程，那么，Windows从同一优先级的就绪队列中选择下一个线程，并且将原来运行的线程移到该队列的尾部（赋予它新的时限值，并且将它的状态从运行态改变为就绪态）。这一情形如图6.22所示。如果同一优先级的就绪队列中没有其他的线程在等着运行，那么该线程再获得一个时限的运行权。

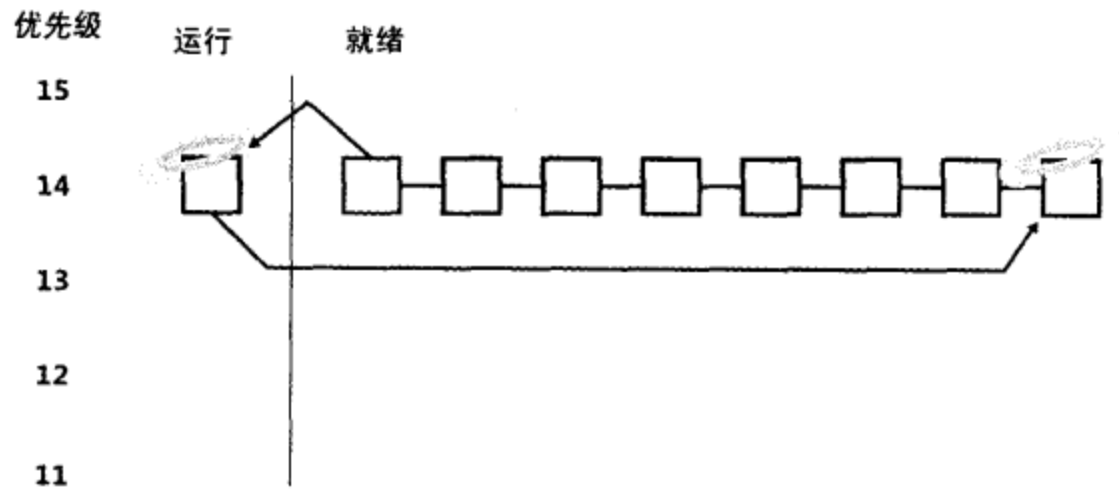


图6.22 时限结束时的线程调度

终止

当一个线程结束运行（或者因为它从主例程中返回了，或者因为调用了*ExitThread*，或者被通过*TerminateThread*杀掉了）时，它从运行状态被转移到终止状态。如果该线程对象上没有已打开的句柄，则从进程的线程列表中删除该线程，相关联的数据结构也被释放。

环境切换

一个线程的环境和用于环境切换的过程随着处理器体系结构的不同而不尽相同。一个典型的环境切换需要保存和加载以下的数据：

- 指令指针；
- 用户栈和内核栈的指针；
- 指向该线程运行所在的地址空间（该进程的页表目录）的指针。

www.TopSage.com

内核把老线程的这些信息保存起来，其做法是：将这些信息压入到当前的（即老线程的）内核模式栈中，更新栈指针，然后将此栈指针保存在老线程的KTHREAD块中。然后，内核栈指针被设置为新线程的内核栈，新线程的环境被加载进来。如果新线程位于不同的进程中，那么，它将新进程的页表目录加载到一个专门的处理器寄存器中，从而使用新进程的地址空间（参见第7章中关于地址转译的介绍）。 如果一个需要被交付的内核APC正在等待处理，则一个IRQL为1的中断就会产生。否则，处理器的控制被传递给新线程中已被恢复的指令指针，于是新线程恢复执行。

空闲（Idle）线程

当一个CPU上不存在可运行的线程时，Windows把执行权分发给该CPU的空闲线程。每个CPU都被分配了一个空闲线程，因为在一个多处理器系统上，当一个CPU正在执行某个线程的时候，另一个CPU可能没有线程要执行。

各种不同的Windows进程查看工具会使用不同的名称来报告空闲进程的情况。任务管理器和进程管理器（Process Explorer）称空闲进程为“System Idle Process”，而进程查看器（Process Viewer）将它报告为“Idle”，Pstat称其为“Idle Process”，Process Explode和Tlist称其为“System Process”，而Oslice称其为“SystemProcess”。Windows声称空闲线程的优先级为0。然而，实际上，空闲线程根本就没有优先级别，因为只有在没有实际线程要运行的时候它们才会运行（记住，每个Windows系统只有一个线程真正运行在优先级0上——零页面线程，参见第7章的解释）。

空闲循环运行在DPC/Dispatch级别上，它随意选择一些工作来做，比如交付一些延迟的过程调用（DPC），或者寻找线程来分发。尽管在不同的体系结构之间，细节流程可能有所不同，但是，空闲线程的基本控制流程如下。

1. 允许和禁止中断（允许任何正在等待处理的中断可被交付）。
2. 检查该处理器上是否有DPC（参见第3章的介绍）正在等待。如果有DPC在等待，则清除正在等待的软件中断，并交付这些DPC。
3. 检查是否有一个线程被选出来作为在该处理器上运行的下一个线程，如果有的话，分发该线程。
4. 调用HAL的处理器空闲例程（有可能电源管理功能需要被执行）。

在Windows Server 2003中，空闲线程也会扫描其他处理器上正在等待运行的线程（在接下来的多处理器调度章节中将会解释这一话题）。

优先级提升

在以下的5种情形中，Windows可以提升（增大）线程的当前优先级值：

- 当I/O操作完成时；
- 在等待执行体事件或信号量以后；

www.TopSage.com

- 在前台进程中的线程完成了一个等待操作以后；
- 当GUI线程由于窗口活动而被唤醒时；
- 当一个准备运行的线程在一段时间内还没有被运行起来（CPU饥饿）时。

这些调整的意图是，提高系统的总体运行能力和响应能力，以及解决可能的不公平调度情形。然而，如同任何一个调度算法一样，这些调整不是完美的，它们不可能让所有的应用程序都获益。



> **注** Windows从来不会提升实时范围（16至31）内的线程的优先级。因此，相对于其他一些实时范围内的线程而言，调度过程总是可预测的。Windows假定：如果你正在使用实时的线程优先级，那么你知道你在做什么。

I/O完成以后的优先级提升

Windows每当特定的I/O操作完成时就会赋予临时的优先级提升，这样，那些正在等待I/O的线程将有更多的机会被立即运行，从而尽快完成正在等待处理的任务。前面提到过，当一个线程被唤醒时，从该线程的剩余时限中减去一个时限单元，所以，那些以I/O为限制（I/O bound）的线程也不是完全不公平受优惠的。尽管你可以从DDK的头文件中找到推荐的提升值（在Wdm.h或Ntddk.h中搜索“#define IO”），但实际的提升值要取决于设备驱动程序（表6.16中列出了这些值）。正是设备驱动程序在完成了一个I/O请求时，在对内核函数*IoCompleteRequest*的调用中指定了提升值。在表6.16中，你可以注意到，越是针对那些要求更高响应性的设备的I/O请求，越是有更高的提升值。

表6.16 推荐的提升值

设 备	提升值
磁盘、CD-ROM、并行口、视频	1
网络、邮槽、命名管道、串行口	2
键盘、鼠标	6
声卡	8

以上提升值总是被应用在一个线程的基本优先级，而不是当前优先级上。正如图6.23所示，在应用了提升值以后，线程在提升后的优先级上开始运行一个时限。当线程完成了这个时限以后，它减退一个优先级别，然后运行另一个时限。此循环过程一直继续，直至该线程的优先级别减退到它的基本优先级。更高优先级的线程仍然可以抢占已被提升了优先级的线程，但是，被中断的线程在减退到下一个低一些的优先级以前，首先要完成在提升起来的优先级别上的时间片。

www.TopSage.com

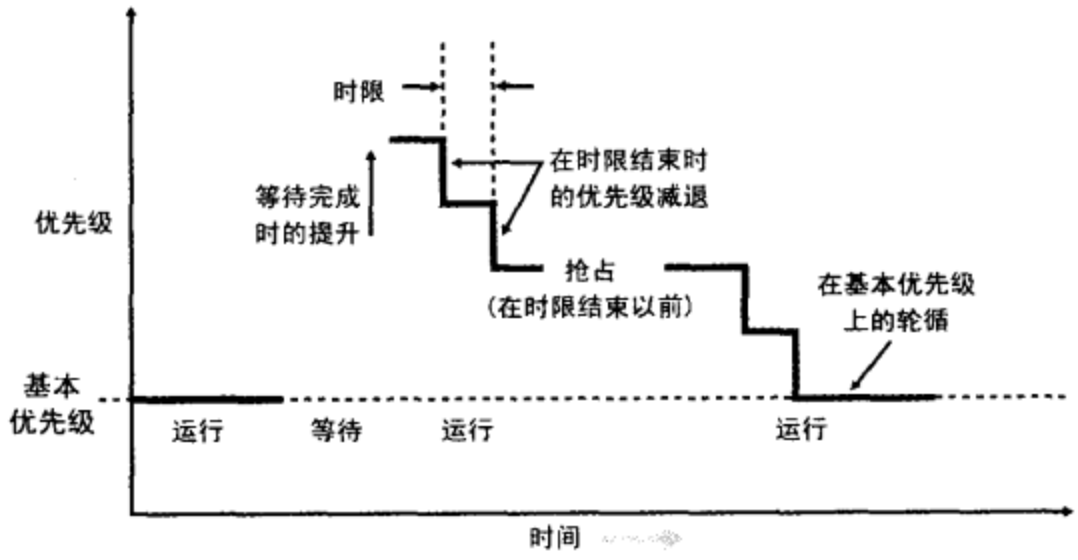


图6.23 优先级提升和减退

正如前面所说明的，这些优先级提升仅适用于动态优先级范围（0到15）内的线程。不管提升值多大，线程被提升后的优先级永远不会超过15而到达实时优先级范围内。换句话说，优先级为14的线程在接收到提升值5的时候，只能到达优先级15。优先级为15的线程在接收到提升值时，其优先级仍然是15。

在等待了事件和信号量以后的优先级提升

如果一个线程正在等待一个执行体事件或者信号量对象，那么，当它的等待被满足（由于调用了函数*SetEvent*、*PulseEvent*或*ReleaseSemaphore*）时，它接收到优先级提升值1（参见DDK头文件中的EVENT_ INCREMENT和SEMAPHORE_INCREMENT的值）。对于这些等待事件和信号量的线程，准许它们一个优先级提升，其理由与准许那些等待I/O操作的线程提升一个优先级相同：正在事件上阻塞的线程比那些以CPU为限制（CPU-bound）的线程，更少地请求CPU周期。这一调整有助于做一下平衡。

这一优先级提升与I/O完成以后发生的优先级提升有相同的执行过程，如上一小节所描述：

- 提升值总是被应用在基本优先级（而不是当前优先级）上；
- 优先级永远不会被提升到15以上；
- 线程每次减退一个优先级别以前，先在提升的优先级上运行完剩余的时限（如前面所描述，当线程退出一个等待时，时限被减少1），直至减退到原来的基本优先级。

对于那些通过两个特殊的函数，即*NtSetEventBoostPriority*（在Ntdll.dll中，用于临界区）和*KeSetEventBoostPriority*（用于执行体资源和压栈锁）来设置一个事件从而被唤醒的线程，它们将应用一个特殊的提升值。如果一个线程正在等待一个事件，它由于这些特殊的事件提升函数而被唤醒，并且其优先级是13或更低，那么，它的优先级被提升至设置此事件的线程的优先级再加1。如果它的时限小于4个时限单元，那么，它被设置为4个时限单元。在时限结束时，这一提升也被去除。

www.TopSage.com

等待以后，前台线程的优先级提升

无论何时，当前台进程中的线程完成了一个等待内核对象的操作时，内核函数*KiUnwaitThread*提升它的当前（而不是基本）优先级，所提升的值是*PsPrioritySeparation*的当前值（由窗口系统负责确定哪个进程被认为是在前台）。正如在时限控制一节中所讲述的那样，*PsPrioritySeparation*反映了时限表的索引，用于为前台应用程序中的线程选择时限。然而，在这里，它被用做一个优先级提升值。

这里提升优先级的理由是，为了增强交互式应用程序的响应性——当前台应用程序完成一个等待时，给它一个小小的提升，它就有更好的机会被立即运行，尤其是当同一优先级上的其他进程可能正处于后台运行的时候。

与其他类型的优先级提升不同的是，这种优先级提升适用于所有的Windows系统，而且你不能禁止这种提升，即使你已经使用Windows的*SetThreadPriorityBoost*函数禁止了优先级提升。

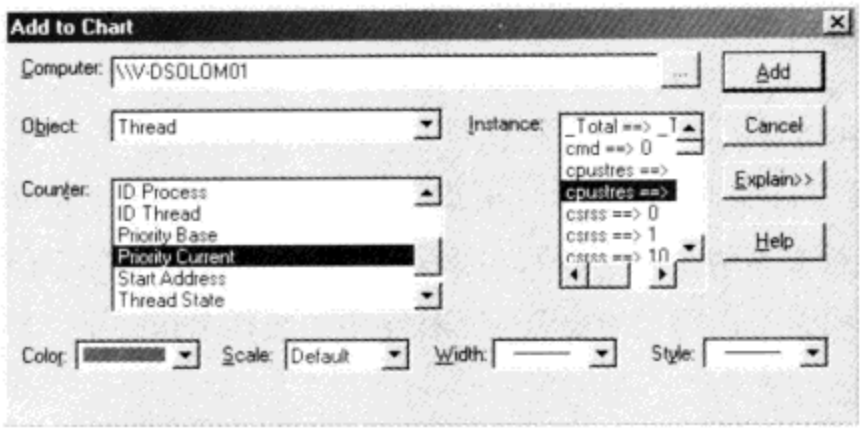


实验：观察前台优先级提升和减退

利用CPU Stress工具（在资源工具箱和Platform SDK中），你可以观察到优先级提升在起作用。可以使用下面的步骤。

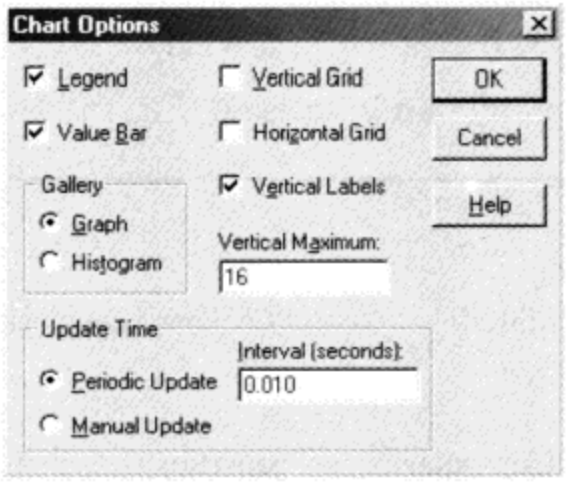
1. 打开控制面板中的System工具（或者右键点击My Computer，选择Properties），点击Advanced标签，点击Performance Options按钮。选择Applications选项，这使*PsPrioritySeparation*得到一个值2。
2. 运行Cpustres.exe。
3. 运行Windows NT 4的性能监视器工具 （Windows 2000资源工具箱中的Perfmon4.exe）。这一老版本的性能工具在这个实验中是必需的，因为它可以比Windows的性能工具以更快的频率查询性能计数器值，Windows的性能工具有一个每秒钟一次的最大间隔限制。
4. 在工具栏上点击Add Counter按钮（或按下Ctrl+I），以打开Add To Chart对话框。
5. 选择Thread对象，然后选择Priority Current计数器。
6. 在Instance框中，向下滚动列表，直至你看到cpustres进程。选择第二个线程（线程1）（第一个线程是GUI线程）。你应该看到如下类似的对话框界面：

www.TopSage.com

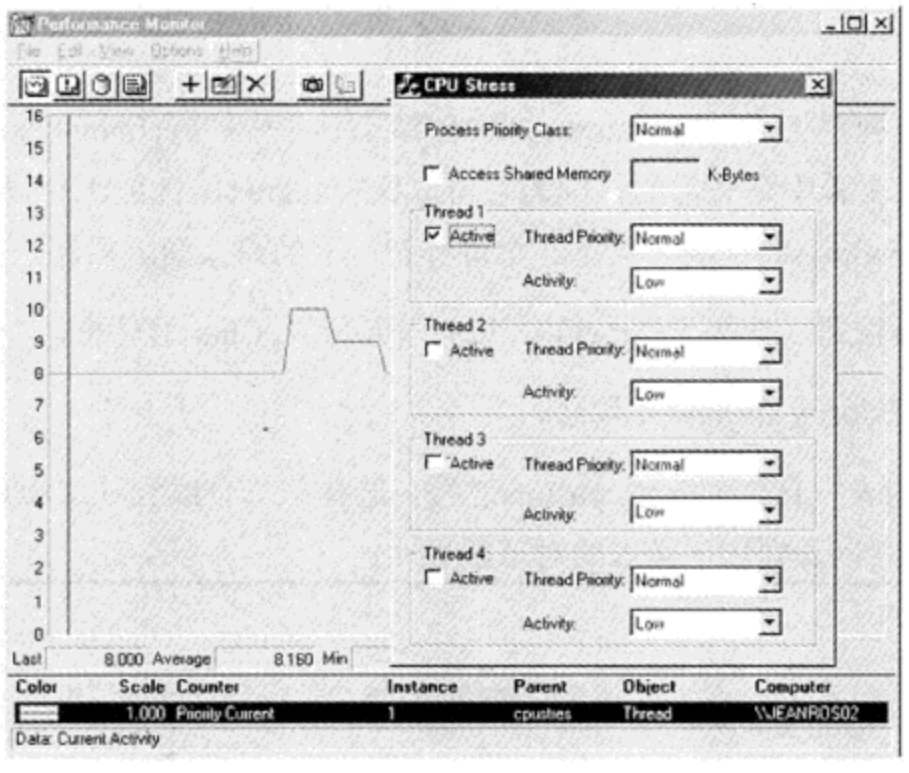


7. 点击Add按钮，然后点击Done按钮。

8. 从Options菜单中选择Chart。将Vertical Maximum改变成16， Interval改变成0.010，如下图所示，然后点击OK。



9. 现在将Cpustres进程带到前台。你应该可以看到Cpustres线程的优先级被提升了2，然后又减退到基本优先级，如下图所示。



www.TopSage.com

10. Cpustres之所以周期性地接收到提升值2，是因为，你正在监视的线程先睡眠75%的时间，然后醒过来——当该线程醒来的时候它的优先级就会被提升。要想看到该线程被提升得更加频繁，你可以增加Activity级别，从低（Low）到中（Medium），再到忙（Busy）。如果你将Activity级别设置为Maximum（最大），那么你将不会再看到任何优先级提升，因为Cpustres中的Maximum级别使得该线程直接进入一个无限循环。因此，该线程不再调用任何等待函数，从而不再接收任何优先级提升。
11. 你完成实验以后，退出性能监视器程序和CPU Stress程序。

GUI线程醒来以后的优先级提升

对于包含窗口的线程，它们由于一个窗口活动（比如窗口消息到来）而醒来的时候，会接收到一个额外的提升值2。当窗口系统（Win32k.sys）调用*KeSetEvent*来设置一个事件，因而唤醒一个GUI线程的时候，窗口系统就会应用这种优先级提升。这种优先级提升的理由与上一种类似——为了更加有利于交互式应用。



实验：观察GUI线程上的优先级提升

你只要监视一个GUI应用程序的当前优先级，并且在其窗口内外移动鼠标，就可以看到，窗口系统对于那些醒过来处理窗口消息的GUI线程应用了提升值2。只需遵从以下这些步骤。

1. 打开控制面板中的System工具（或者右键点击My Computer，并选择Properties），点击Advanced标签，点击Performance Options按钮。如果你正在运行Windows XP或者Windows Server 2003，则选择Advanced标签，并确保Programs选项被选中；如果你正在运行Windows 2000，则确保Applications选项被选中。这使得*PsPrioritySeparation*的值为2。
2. 通过Start菜单运行Notepad，你只需选择Programs/Accessories/Notepad。
3. 运行Windows NT 4的性能监视器工具 （Windows 2000资源工具箱中的Perfmon4.exe）。这一老版本的性能工具在这个实验中是必需的，因为它可以以更快的频率来查询性能计数器值，Windows的性能工具有一个每秒钟一次的最大间隔限制。
4. 在工具栏上点击Add Counter按钮（或按下Ctrl+I），以打开Add To Chart对话框。
5. 选择Thread对象，然后选择Priority Current计数器。
6. 在Instance框中，向下滚动列表，直至你看到Notepad的线程0。点击此实例，再点击Add按钮，然后点击Done按钮。
7. 如同上一个实验一样，从Options菜单中选择Chart。将Vertical Maximum改变成16，Interval改变成0.010，然后点击OK。

www.TopSage.com

8. 你应该看到，Notepad的线程0的优先级在8、9或10。因为Notepad在接收到提升值2（前台进程中的线程应该接收到的）以后立刻进入了等待状态，所以，它可能还来不及从10减退到9，然后再减到8。
9. 让性能监视器保持在前台，并横跨Notepad窗口移动鼠标（让这两个窗口在桌面上都是可见的）。你将会看到，Notepad的优先级有时候保持在10，有时候保持在9，理由如前所述（你之所以看不到Notepad的优先级在8上，是因为它在接收到GUI线程的提升值2以后只运行了很短的时间，以至于在下一次由于额外窗口活动而被唤醒，以及再次接收提升值2以前，根本来不及经历多次优先级减退）。
10. 现在将Notepad带到前台。你应该看到优先级上升到12，并且保持该优先级（或者下降到11，因为它可能会经历正常的优先级减退，这是在时限结束时发生在被提升线程上的正常过程）。因为该线程接收了两种优先级提升：应用在GUI线程上的提升值2（当它们醒过来处理窗口输入时），以及额外的提升值2（因为Notepad在前台）。
11. 如果你在Notepad上移动鼠标（它仍然在前台运行），你可能会看到它的优先级被降到11（甚至可能10），因为它经历了在时限完成时发生在被提升线程上的正常优先级减退。然而，如果Notepad仍然在前台，则因为前台进程而作用在Notepad上的提升值2一直保持不变。
12. 你完成实验以后，退出性能监视器程序和Notepad。

因为CPU饥饿而引起的优先级提升

想象下面的情形：你有一个优先级为7的线程正在运行，因而妨碍了一个优先级为4的线程获得CPU时间，然而，一个优先级为11的线程正在等待某一资源，而该资源又被这一优先级为4的线程锁住了；但是，因为位于中间的优先级为7的线程用光了所有的CPU时间，所以，优先级为4的线程永远也不会有足够的时间来做完它所要做的事情，并释放这一阻塞住优先级11线程的资源。Windows该如何来解决这种情形呢？**平衡集管理器**（*balance set manager*，这是一个系统线程，它的存在主要是为了执行内存管理功能，第7章中有更加详细的描述）每秒钟扫描就绪队列一次，查找已在就绪状态（也就是说，还没有被运行）将近4秒钟的线程。如果平衡集管理器找到了这样一个线程，则它将该线程的优先级提升到15。在Windows 2000和Windows XP上，这一线程的时限被设置为进程时限的两倍。在Windows Server 2003上，该时限被设置为4个时限单元。一旦此时限到期，线程的优先级立即被减退到它原来的基本优先级。如果该线程尚未完成时一个更高优先级的线程准备运行了，那么，减退的线程将返回就绪队列中；如果它在该队列中又待上4秒钟的话，则它再次获得另一优先级提升。

www.TopSage.com

平衡集管理器每次运行的时候并不真正扫描所有就绪的线程。为了尽最大可能减少它所用的CPU时间，它只扫描16个就绪线程；如果在一个优先级上还有更多的线程，那么，它记录下这次扫描的位置，因而下一次扫描可以从这里开始。而且，它每次扫描只提升10个线程——如果它找到了10个线程，而且这些线程都符合这种特殊的优先级提升要求（这也表明了这是一个异常繁忙的系统），那么它在这个点上停止扫描，下次扫描还从这个地方开始。

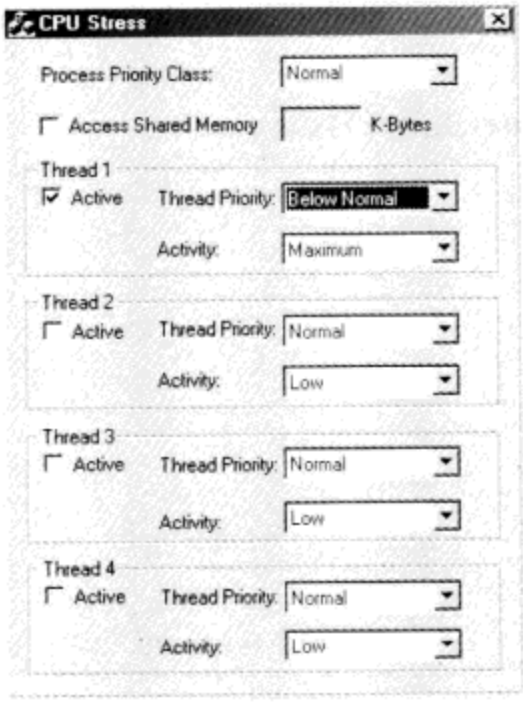
这个算法总能解决这种优先级倒置（priority inversion）的问题吗？不，无论如何它都不是完美的。但是，随着时间的推移，CPU匮乏的线程应该能得到足够的CPU时间来完成它们正在处理的事务，并重新进入等待状态。



实验：观察因CPU饥饿而引起的优先级提升

使用CPU Stress工具（在资源工具箱和Platform SDK中），你可以观察到优先级提升在起作用。在这个实验中，我们来看一看，当一个线程的优先级被提升时CPU使用率的变化情况。可以使用以下的步骤。

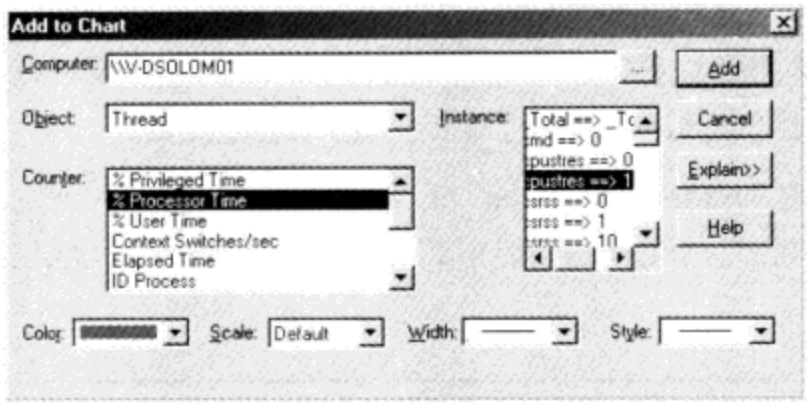
1. 运行Cpustres.exe。将活动线程（默认为Thread 1）的活动级别从低（Low）改变为最大（Maxiumu）。将线程的优先级从普通（Normal）改变成普通之下（Below Normal）。看起来，屏幕应该如下面所示的这样。



2. 运行Windows NT 4的性能监视器工具 （Windows 2000资源工具箱中的Perfmon4.exe）。同样地，在这个实验中你需要这一老版本的性能工具，因为它可以以超过1秒钟一次的频率来查询性能计数器值。
3. 在工具栏上点击Add Counter按钮（或按下Ctrl+I），以打开Add To Chart对话框。
4. 选择Thread对象，然后选择% Processor Time计数器。

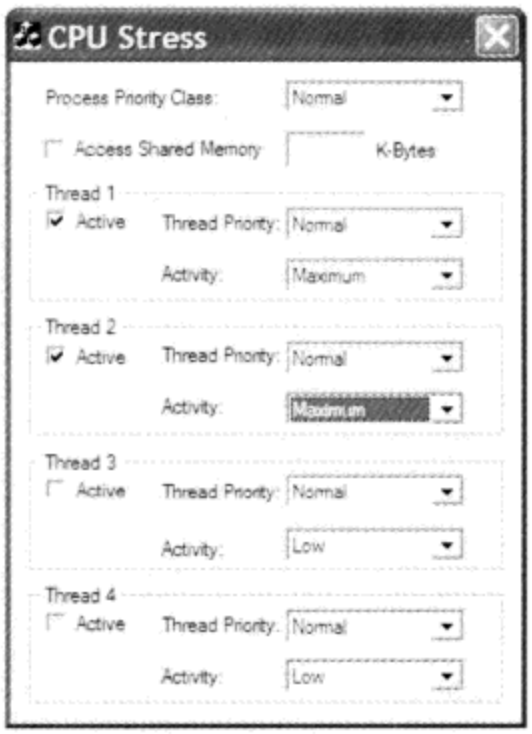
www.TopSage.com

5. 在Instance框中，向下滚动列表，直至你看到cpustres进程。选择第二个线程（线程1）（第一个线程是GUI线程）。你应该看到如下类似的对话框界面：



6. 点击Add按钮，然后点击Done按钮。
7. 运行任务管理器（Task Manager），点击Processes页面，并选择Perfmon4.exe进程，将性能监视器的优先级提升到实时优先级。在该进程上右键点击，选择Set Priority，然后选择Realtime（如果你接收到一个任务管理器警告消息框，提示你系统可能不稳定，则请点击Yes按钮）。
8. 运行CPU Stress的另一份拷贝。在这份拷贝中，将Thread 1的活动优先级从低（Low）改变成最大（Maximum）。
9. 现在切换回到性能监视器。你应该看到每隔4秒钟左右CPU的活动，因为该线程被提升到优先级15。

当你完成实验以后，退出性能监视器程序和CPU Stress的两份拷贝。



www.TopSage.com



实验："监听"优先级提升

为了"听到"因CPU饥饿而引起的优先级提升的效果，请在一个安装了声卡的系统上执行下面的步骤。

1. 运行Windows Media Player（或者其他某个音频播放程序），开始播放某一段音频内容。
2. 运行Windows 2000资源工具箱中的Cpustres程序，将线程1的活动级别设置为最大。
3. 将线程1的优先级从普通（Normal）提升至时间关键（Time Critical）级别。
4. 由于这一计算型的线程开始消耗掉所有可用的CPU时间，所以你可以听到音乐播放停止了。
5. 时不时地，你应该可以听到一点声音，因为音频播放进程中被饿着的线程其优先级被提升到15，并且运行一段足够的时间，从而向声卡发送更多数据。
6. 停止Cpustres和Windows Media Player。

多处理器系统

在一个单处理器系统上，调度过程相对比较简单：在所有想要运行的线程中，最高优先级的线程总是在运行。在一个多处理器系统上，调度过程要复杂得多，因为Windows试图将线程安排到对于它们来说最优的处理器上，它要考虑到线程的首选处理器、以前运行的处理器，以及多处理器系统的配置情况。因此，尽管Windows试图将那些最高优先级的可运行线程安排到所有可用的CPU上，但是，它只保证一个最高优先级的线程在某个处理器上运行。

我们在介绍"如何选择哪些线程在哪里、何时运行"的专门算法以前，先来看一看，Windows维护了哪些额外的信息来记录多处理器系统中线程和处理器的状态，以及Windows支持的两种新的多处理器系统类型（超线程和NUMA）。

多处理器分发器数据库

正如在本章前面的"分发器数据库"一节中所介绍的那样，分发器数据库引用了内核所维护的、用于实现线程调度的信息。如图6.16所示，在多处理器的Windows 2000和Windows XP系统上，就绪队列和就绪摘要的结构与单处理器系统上的结构相同。除了就绪队列和就绪摘要以外，Windows还维护了两个位掩码来跟踪记录当前系统中处理器的状态（在后面的"多处理器线程调度算法"一节中解释了如何使用这些位掩码）。下面是Windows维护的两个位掩码：

www.TopSage.com

- **活动处理器掩码**（*KeActiveProcessors*），系统中每个可以使用的处理器都有相应的一位（如果当前运行的Windows版本的许可限制只支持少于实际可用的物理处理器的数目，则该掩码的活动处理器数可能少于实际处理器的数目）。
- **空闲摘要**（KiIdleSummary），每一位反映了一个空闲的处理器

虽然在单处理器系统上，通过提升IRQL（在Windows 2000和Windows XP上，提升至DPC/Dispatch级别；在Windows Server 2003上，提升至DPC/Dispatch级别和同步[Synch]级别），可以锁住分发器数据库，但是，在多处理器系统上，要想锁住分发器数据库还需要更多的要求，因为在同一时刻，每个处理器都可以提升IRQL，以试图操作分发器数据库（对于任何在高IRQL上才允许访问的系统范围的数据结构，这都是一样的。参见第3章关于内核同步和自旋锁的一般性描述）。在Windows 2000和Windows XP上，利用两个内核自旋锁来同步对线程分发信息的访问，它们是：**分发器自旋锁**（*dispatcher spinlock*，*KiDispatcherLock*）和**环境交换自旋锁**（*context swap*，*KiContextSwapLock*）。如果要改变那些“可能会影响哪个线程应该运行”的数据结构，则必须持有前一个锁；在做出了调度决定，而在线程环境交换操作过程中，则需要持有后一个锁。

为了提高可伸缩性（包括线程分发的并发性在内），Windows Server 2003多处理器系统为每个处理器提供了一个分发器就绪队列，如图6.24所示。通过这种方式，在Windows Server 2003上，每个CPU可以检查它自己的就绪队列，以确定下一个该运行的线程，而无须锁住全系统范围的就绪队列。

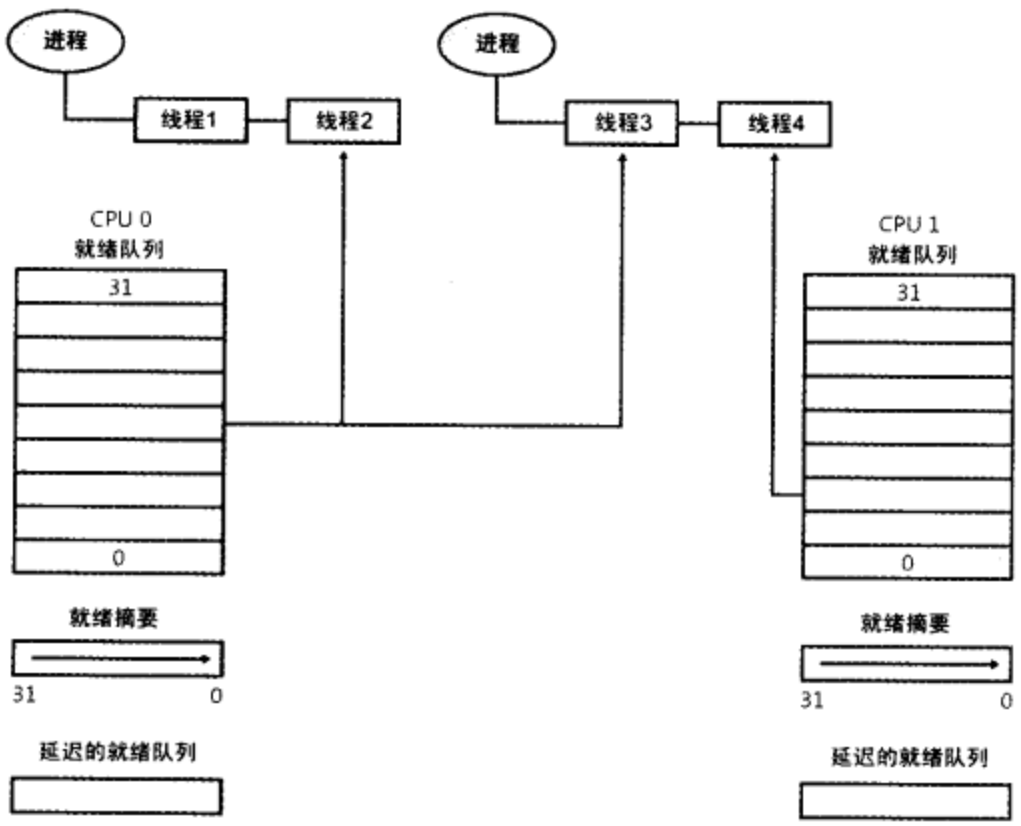


图6.24 Windows Server 2003多处理器分发器数据库

www.TopSage.com

这一针对每个处理器的就绪队列，以及针对每个处理器的就绪摘要，都是处理器控制块（PRCB）结构的一部分（要想看到PRCB中的域，可以在内核调试器中输入**dt nt!_prcb**）。因为在一个多处理器系统上，一个处理器可能要修改另一个处理器的调度数据结构（比如插入一个要求在特定处理器上才可运行的线程），所以，这些结构通过使用一个新的、针对每个PRCB的排队自旋锁来进行同步，该锁在IRQL SYNCH_LEVEL上可被持有（有关SYNCH_LEVEL的各种值，请参见表6.17）。因此，只有锁住了一个处理器的PRCB，才可以进行线程选择，而与此相比，在Windows 2000和Windows XP上，要进行线程选择，必须持有系统全局范围的分发器自旋锁。

表6.17 多处理器系统上的IRQL SYNCH_LEVEL

Windows版本	多处理器
Windows 2000	2
Windows XP在x86上	2
Windows Server 2003在x86上	27
Windows XP在x64上	12
Windows XP在IA-64上	2
Windows Server 2003在x64和IA-64上	12

对于每个CPU，还有一个线程列表记录了那些处于延迟的就绪状态中的线程。它们代表了那些虽已经就绪，但尚未准备执行的线程；实际的就绪操作已经被延迟到一个更为恰当的时候。因为每个处理器只操作和控制它自己的延迟就绪列表，所以，该列表并不需要通过PRCB自旋锁来同步。在退出线程分发器以前、在执行环境切换以前，以及在处理了一个DPC以后，此延迟的就绪线程列表就会被处理。此延迟的就绪列表中的线程或者被立即分发，或者被转移到它们所在优先级别上的、针对特定处理器的就绪队列中。

注意，在Windows Server 2003上，系统全局范围的分发器自旋锁仍然存在，并且也确实有用，但是，它只是在需要修改“可能会影响到接下来该哪个线程运行”的系统范围状态时才需要被持有。例如，对同步对象（互斥体、事件和信号量）及其等待队列的改变要求持有分发器锁，以避免多个处理器同时改变这些对象的状态（以及可能因此而改变了执行哪个就绪线程的决定）。其他的例子还包括改变一个线程的优先级、定时器到期，以及交换线程的内核栈。

最后，在Windows Server 2003上，线程环境切换的同步也有所改进，因为它现在使用一个针对每个线程的自旋锁来进行同步，而在Windows 2000和Windows XP中，环境切换是通过持有一个系统全局范围的环境交换自旋锁来进行同步的。

www.TopSage.com

超线程系统

正如在第2章的“对称多处理”一节中所描述的那样，Windows XP和Windows Server 2003通过以下两种基本的方式来支持超线程多处理器系统。

1. 逻辑处理器并不影响物理处理器的许可限制，例如，对于Windows XP Home Edition，它许可的处理器限制是1，但它在一个单处理器的超线程系统上可以使用两个逻辑处理器。
2. 在为一个线程选择处理器的时候，如果有一个物理处理器的所有逻辑处理器都是空闲的，那么，分发器将会选择该物理处理器中的一个逻辑处理器，而不会再从另外一个物理处理器（它的一个逻辑处理器已经在运行其他的线程了）中选择一个空闲的逻辑处理器。



实验：查看超线程信息

在内核调试器中，你通过*!smt*命令可以查看Windows为超线程处理器所维护的信息。下面的输出是从一个双处理器的超线程Xeon系统（四个逻辑处理器）上得到的。

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

  Number of licensed physical processors: 2
```

逻辑处理器0和1位于不同的物理处理器上（“Master”标志说明了这一点）。

NUMA系统

Windows XP和Windows Server 2003支持的另一种类型的多处理器系统，是针对那些具有非一致内存访问（nonuniform memory access，NUMA）体系结构的多处理器系统。在一个NUMA系统中，处理器被组织成稍小一些的单元（称为节点）。每个节点有它自己的处理器和内存，它们通过一条具有统一高速缓存的互联总线连接到大系统中。这些系统之所以称为“非一致”的，是因为每个节点有它自己的局部高速内存。虽然任何节点上的任何一个处理器都可以访问所有的内存，但是，访问节点的局部内存要快得多。

www.TopSage.com

在一个NUMA系统中，内核把有关每个节点的信息保存在一个称为KNODE的数据结构中。内核变量*KeNodeBlock*是一个指针数组，其中的指针元素指向了每个节点的KNODE结构。在内核调试器中使用*dt*命令，可以显示KNODE结构的格式，如下所示：

```
lkd> dt nt!_knode
nt!_KNODE
   +0x000 ProcessorMask    : Uint4B
   +0x004 Color            : Uint4B
   +0x008 MmShiftedColor   : Uint4B
   +0x00c FreeCount        : [2] Uint4B
   +0x018 DeadStackList    : _SLIST_HEADER
   +0x020 PfnDereferenceSListHead : _SLIST_HEADER
   +0x028 PfnDeferredList  : Ptr32 _SINGLE_LIST_ENTRY
   +0x02c Seed             : UChar
   +0x02d Flags            : _flags
```



实验：查看NUMA信息

在一个NUMA系统中，在内核调试器中使用*!numa*命令，你可以查看Windows为每个节点维护的信息。下面的部分输出是从一个包含32个处理器的NUMA系统（NEC，每个节点4个处理器）中得到的。

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

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

   NODE 1 (E00001597A9A2200):
       ProcessorMask    : ----****------------------------------------------------------------
       Color            : 0x00000001
       MmShiftedColor   : 0x00000040
       Seed             : 0x00000006
       Zeroed Page Count: 0x00000000001F77A0
       Free Page Count  : 0x0000000000000004
```


www.TopSage.com

下面的部分输出是从一个包含64个处理器的NUMA系统（Hewlett Packard，每个节点4个处理器）中得到的。

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

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

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

如果应用程序想要在NUMA系统上获得最好的性能，则可以设置亲和性掩码，以限制一个进程只在一个特定节点中的处理器上运行。利用表6.18中列出的函数可以获得这些信息。能够改变线程亲和性的函数如表6.13所列。

表6.18 与NUMA相关的函数

函 数	说 明
GetNumaHighestNodeNumber	获取当前最高编号的节点
GetNumaNodeProcessorMask	获取指定节点的处理器掩码
GetNumaProcessorNode	获取指定处理器的节点编号

调度算法如何考虑NUMA系统呢？在接下来的“多处理器的线程调度算法”一节中将会介绍（第7章还将介绍内存管理器中的一些优化手段以充分利用节点的局部内存）。

亲和性

每个线程都有一个亲和性掩码，它指定了该线程只允许在哪些处理器上才可以运行。线程亲和性掩码是从进程的亲和性掩码继承过来的。在默认情况下，所有的进程（因而也是所有的线

www.TopSage.com

程）的起始亲和性掩码都等于当前系统中活动处理器的集合。换句话说，系统可以自由地在所有可用的处理器上调度所有的线程。

然而，为了优化总体运行效率，以及/或者将工作负载分布到特定的处理器集合上，应用程序可以选择能改变一个线程的亲和性掩码。这可以在以下几个不同的层次上完成。

- 调用SetThreadAffinityMask，为单独的线程设置亲和性。
- 调用SetProcessAffinityMask函数，为一个进程中的所有线程设置亲和性。任务管理器和进程管理器为这个函数提供了GUI界面：你只要右键点击一个进程，并选择Set Affinity。Psexec工具（来自*www.sysinternals.com*）为这一函数提供了命令行界面 （参见-a开关）。
- 让一个进程成为一个作业的成员。对于作业对象，通过SetInformationJobObject函数，它可以有一个作业范围内的亲和性掩码（在本章后面的“作业对象”一节中介绍了作业）。
- 利用诸如Imagecfg （位于Windows 2000 Server资源工具箱补充1 [Resource Kit Supplement 1]）这样的工具，可以在映像头部指定一个亲和性掩码（有关Windows映像的细节格式的更多信息，请参见MSDN Library中的文章“Portable Executable and Common Object File Format Specification”）。

你也可以为一个映像设置“uniprocessor（单处理器）”标志（使用Imagecfg *-u*开关）。如果该标志被设置的话，则系统在创建进程的时候选择一个处理器，并以此作为该进程的亲和性掩码；系统从第一个处理器开始分配，然后通过轮循的方法遍历所有的处理器。例如，在一个双处理器系统上，当你第一次运行一个被标记为单处理器的映像时，它被分配给CPU 0；第二次，分配CPU 1；第三次，分配CPU 0；第四次，分配CPU 1；依此类推。对于那些有多线程同步错误（因为竞争条件而引起的错误，在多处理器系统上会出现，但是在单处理器系统上不会发生）的程序，这一标志可以被用做一种临时的解决方案（事实上，这种解决方案曾经分别在两种不同的情形下，解救了本书的作者）。

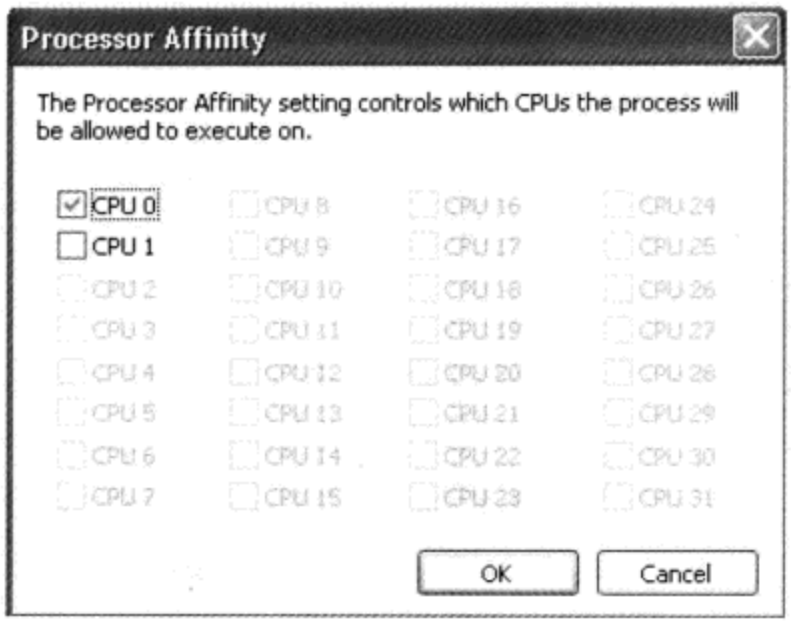


实验：查看和改变进程的亲和性

在这个实验中，你将修改一个进程的亲和性设置，并且可以看到，此进程的亲和性被新的进程所继承。

1. 运行命令提示程序（cmd.exe）。
2. 运行任务管理器或进程管理器，并在进程列表中找到cmd.exe进程。
3. 右键点击该进程，并选择Affinity。应该会显示一个处理器列表。例如，在一个双处理器系统上，你将会看到如下的界面：

www.TopSage.com



4. 选择该系统上所有可用处理器的一个子集，并按下OK。现在，该进程的线程被限制只能在你刚刚选择的处理器上运行。
5. 现在从命令提示窗口中运行Notepad.exe（只需输入Notepad.exe）。
6. 回到任务管理器或者进程管理器，并找到新的Notepad进程。右键点击该进程，并选择Affinity。你所看到的处理器列表应该与你为命令提示进程所选择的处理器列表相同。这是因为，子进程从父进程那里继承了亲和性设置。



实验：改变映像的亲和性

在这个实验（此实验要求访问一个多处理器系统）中，你将改变一个程序的亲和性掩码，以强迫它在第一个处理器上运行。

1. 从Windows 2000的资源工具箱中拷贝一份Cpustres.exe。不妨假设在你的系统上已经有了c:\temp文件夹，从命令提示窗口中，输入下面的命令：

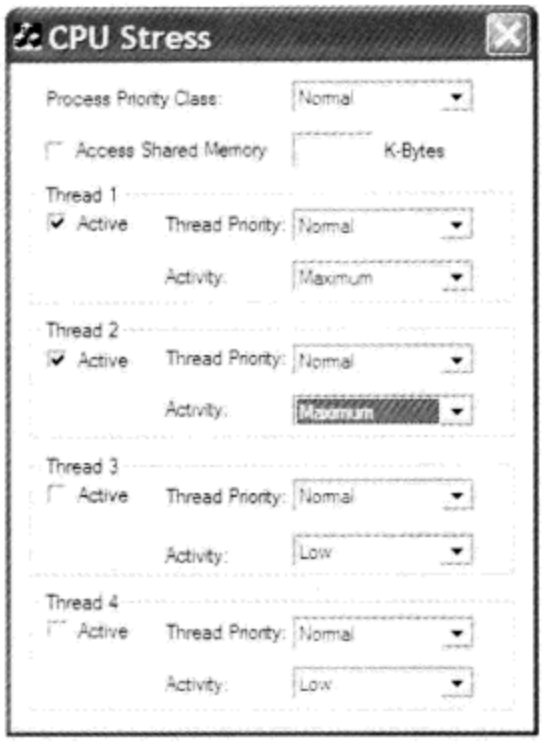
```
copy c:\program files\resource kit\cpustres.exe c:\temp\cpustres.exe
```

2. 设置该映像的亲和性掩码，以强迫该进程的线程只能在CPU 0上运行。你可以在命令提示窗口中输入下面的命令（假设资源工具箱的路径已经在命令搜索路径中）：

```
imagecfg -a 1 Cordially,:\temp\cpustres.exe
```

3. 现在从c:\temp文件夹下运行已经修改过的Cpustres。
4. 启动两个辅助线程，设置这两个线程的活动级别为最大（Maximum，注意，不是Busy）。看起来，Cpustres的屏幕应该如下面所示的这样。

www.TopSage.com



5. 在进程管理器或任务管理器中找到Cpustres进程，右键点击该进程，并选择Set Affinity。亲和性设置应该表明了该进程已绑定到CPU 0上。
6. 通过点击Show、System Information（如果在运行进程管理器的话），或者点击Performance页面（如果在运行任务管理器的话），可以检查系统范围的CPU利用率。假设没有其他的计算类型的进程，那么你应该可以看到，所消耗的CPU时间的总百分比近似等于1/CPU数目（例如，在一个双CPU的系统上为50%，在一个四CPU的系统上为25%），因为Cpustres中的两个线程被迫在一个处理器上运行，而其他的处理器总是空闲着。
7. 最后，改变Cpustres进程的亲和性掩码，允许它在所有的CPU上运行。再回去检查系统范围的CPU利用率。在一个双CPU系统上你应该可以看到100%，在四CPU系统上为50%，依此类推。

假设一个线程正在一个CPU上运行，它也可以到别的处理器上运行，而另一个线程的亲和性设置限定了它只能在第一个CPU上运行，那么，Windows不会将第一个线程移到第二个处理器上，以便让第二个线程到第一个CPU上运行。例如，考虑这样的情形：CPU 0正在运行一个优先级为8的线程，该线程可以在任何一个处理器上运行；而CPU 1正在运行一个优先级为4的线程，它也可以在任何一个处理器上运行。另有一个优先级为6的线程只能在CPU 0上运行，它现在进入就绪状态。那会怎么样呢？Windows不会将优先级为8的线程从CPU 0移到CPU 1上（抢占优先级为4的线程），以便让优先级为6的线程可以运行；相反，优先级为6的线程必须等待。

因此，改变一个进程或者一个线程的亲和性掩码可能会导致这些线程得到比在正常情况下更少的CPU时间，因为Windows被限制了，不允许其在某些特定的处理器上运行此线程。因此，设置亲和性必须非常小心——在大多数情况下，让Windows来决定在哪些CPU上运行哪些线程是最佳的做法。

www.TopSage.com

理想处理器和上一个处理器

每个线程在内核线程块中保存了两个CPU编号：

- **理想处理器**，或者，该线程应该运行于其上的首选处理器；
- **上一个处理器**，或者，该线程最近一次运行时所在的处理器。

一个线程的理想处理器是在该线程被创建时，利用进程块中的一个种子来选定的。每次当一个线程被创建时，该种子就会被递增，所以，一个进程中的每个新线程的理想处理器会循环经历当前系统上所有可用的处理器。例如，系统中第一个进程的第一个线程被分配的理想处理器是0，该进程中的第二个线程被分配到1号处理器。然而，系统中下一个进程的第一个线程的理想处理器被设置为1，第二个被设置为2，以此类推。通过这种方式，每个进程内的线程被均匀地散布在所有的处理器上。

注意，这种分配方法假定了，一个进程内的线程完成的是等量的工作。在一个多线程进程中，这往往是不成立的，通常的模式是，由一个或者多个线程做事务管理工作，其他另有一组辅助线程。因此，一个多线程应用程序想要充分利用底层平台的优势，可能会发现，利用*SetThreadIdealProcessor*函数来为它的线程指定理想处理器编号是很有用的。

在超线程系统上，下一个理想处理器是下一个物理处理器上的第一个逻辑处理器。例如，在一个双处理器的超线程系统上，它共有四个逻辑处理器，如果第一个线程的理想处理器被分配为逻辑处理器0，则第二个线程将被分配为逻辑处理器2，第三个线程为逻辑处理器1，第四个线程为逻辑处理器3，以此类推。按照这种方式，这些线程的理想处理器被均匀地散布在物理处理器上。

在NUMA系统上，当一个进程被创建的时候，该进程的理想处理器即被选出来。第一个进程被分配到节点0，第二个进程被分配到节点1，以此类推。然后，这些进程中的线程的理想处理器又是从相应进程的理想节点中选择出来的。一个进程中的第一个线程的理想处理器被分配为节点中的第一个处理器。在具有相同理想节点的进程中，随着越来越多的线程被创建，节点中的下一个处理器被用做下一个线程的理想处理器，如此类推下去。

多处理器的线程调度算法

我们已经讲述了Windows所支持的多处理器系统的类型，以及线程亲和性和理想处理器的设置。现在我们来看一看如何利用这些信息来决定哪些线程在哪些处理器上运行。这里有两个基本的决定要介绍：

- 为想要运行的线程选择一个处理器；
- 在一个想要做些事情的处理器上选择一个线程。

www.TopSage.com

当存在空闲处理器时，为一个线程选择一个处理器

当一个线程准备好运行时，Windows首先试着将该线程安排到一个空闲处理器上来运行。如果存在空闲处理器，则第一优先选择的是该线程的理想处理器，然后是该线程的上一个处理器，再然后是当前正在执行的处理器（也就是，正在运行此调度代码的CPU）。

在Windows 2000上，如果这些CPU都不是空闲的，那么，扫描空闲处理器掩码，从最高编号到最低编号，选择第一个可以运行该线程的空闲处理器。

在Windows XP和Windows Server 2003上，对空闲处理器的选择过程要复杂得多。首先，空闲处理器的集合被设置为：该线程的亲和性掩码允许它运行于其上的那些空闲处理器。如果该系统是NUMA，并且线程的理想处理器所在的节点中存在空闲处理器，那么，空闲处理器的列表被缩减为这一集合。如果这种做法排除了所有的空闲处理器，则不做缩减。接下来，如果系统正在运行超线程的处理器，并且有一个物理处理器的所有逻辑处理器都是空闲的，那么，空闲处理器的列表被缩减为这一集合。如果这种做法导致得到了一个空的处理器集合，则不做缩减。

如果当前处理器（即试图确定如何安排这一待运行线程的那个处理器）也在剩余的空闲处理器集合中，那么，该线程被调度到此处理器上运行。如果当前处理器不在剩余的空闲处理器集合中，并且这是一个超线程系统，该线程的理想处理器所在的物理处理器上有一个空闲处理器，那么，空闲处理器被缩减到这一集合。如果不是这样的话，系统检查该线程的上一个处理器所在的物理处理器是否有空闲的逻辑处理器。如果该集合非空的话，则空闲处理器被缩减至这一列表。

在剩余下来的空闲处理器集合中，任何处于睡眠状态中的CPU都被排除在外，不予考虑（同样地，如果这样做会除掉所有可能的处理器，则不做这种缩减）。最后，剩余集合中编号最低的CPU被选择出来，将其作为将要运行该线程的处理器。

不管Windows的版本是什么，一旦已经为一个线程选择了一个处理器，让它来运行，则该线程被置入备用（Standby）状态，并且空闲处理器的PRCB随之被更新，以指向该线程。当该处理器上的空闲循环运行时，它将会看到有一个线程已经被选择出来并在它上面运行，因而分发给该线程。

当没有空闲处理器时，为一个线程选择一个处理器

如果当一个线程想要运行时，没有空闲的处理器，那么，Windows比较该线程的理想处理器上正在运行的线程（或者处于备用状态中的线程）的优先级，以决定是否应该抢占此线程。在Windows 2000上，一个线程的亲和性掩码可能会排斥理想处理器（在Windows XP上，这一条件是不允许的）。如果是这样的话，Windows 2000选择该线程的前一个处理器。如果该处理器不在线程的亲和性掩码中，则选择可以运行此线程的最高编号的处理器。

www.TopSage.com

如果该线程的理想处理器已经有一个线程被选出来，并将在其上运行（在备用[standby]状态等着被调度），而且这个线程的优先级低于新的准备要执行的线程的优先级，那么，新线程把前一个线程从备用状态中抢占过来，变成该CPU的下一个线程。如果该CPU上已经有一个线程在运行了，则Windows检查当前正在运行的线程的优先级是否低于新的准备要执行的线程。如果是的话，则当前运行的线程被标记为已被抢占，Windows将一个跨处理器的中断插入到目标处理器的队列中，以便抢占当前正在运行的线程，支持这个新线程。



> **注** Windows并不查看所有CPU上的当前和后续线程的优先级，而只是查看如前所述选出来的CPU上的线程的优先级。如果在一个CPU上没有线程被抢占的话，则新的线程被放到它的优先级别的就绪队列中，它在那里等着被按序调度。因此，Windows并不保证，当前运行的都是最高优先级的线程，但是最高优先级的那一个线程总是在运行。

如果已就绪的线程不能被立即运行的话，那么，它被移到就绪状态中，在那里等着轮到它运行。注意，在Windows Server 2003中，线程总是被放在其理想处理器的就绪队列中（每个处理器都有自己独立的就绪队列）。

选择一个线程让它在一个特定的CPU上运行（Windows 2000和Windows XP）

在几种情形下（比如当一个线程进入等待状态、降低其优先级、改变其亲和性，或者延迟或放弃执行时），Windows必须找到一个线程，以便在当前执行线程所在的CPU上运行。如前所述，在一个单处理器系统上，Windows只是简单地从最高优先级的非空就绪队列中挑出第一个线程即可。然而，在一个多处理器系统上，Windows 2000和Windows XP并不是简单地从这一就绪队列中挑出第一个线程。相反，它们在最高优先级的非空就绪队列中寻找满足以下条件之一的线程：

- 上一次在此特定的处理器上运行；
- 它的理想处理器被设置为此特定的处理器；
- 处于就绪运行状态的时间超过3个时钟嘀嗒；
- 优先级大于等于24。

很显然，如果在线程的硬亲和性掩码中不包含这一特定的处理器，则它被忽略。如果没有线程满足以上条件之一，则Windows在它开始上述搜索的就绪队列的头部挑出一个线程。

为什么“一个线程上一次在哪个处理器上运行”显得如此要紧呢？如往常一样，答案是速度——优先选择该线程上一次执行所在的处理器，可以使“该线程的数据仍然在这个处理器的二级缓存中”的几率尽可能地最大化。

www.TopSage.com

选择一个线程让它在一个特定的CPU上运行（Windows Server 2003）

因为Windows Server 2003中的每个处理器都有它自己的线程列表（这些线程都等着在该处理器上运行），所以，当一个线程结束运行的时候，该处理器可以简单地检查它自己的就绪队列，以选择下一个运行的线程。如果处理器的就绪队列是空的，则运行权被调度给该处理器的空闲线程。然后，空闲线程开始扫描其他处理器的就绪队列，寻找它可以运行的线程。注意，在NUMA系统上，空闲线程首先查看它所在节点上的处理器，然后再查看其他节点的处理器。

6.6 作业对象

作业对象是一个可命名的、可保护的、可共享的内核对象，它可以将一个或者多个进程作为一个组来控制和管理。作业对象的基本功能是，允许一组进程被当作一个单元来加以管理和控制。一个进程只能是一个作业对象的成员。在默认情况下，一个进程与其作业对象的关联关系不能被打破，而且，该进程及其后代所创建的所有进程都与同样的作业对象关联起来。作业对象也为所有与它关联的进程记录了基本的审计信息，也为曾经与它关联过的，但后来已终止的进程记录审计信息。表6.19列出了用于创建和操作作业对象的Windows函数。

表6.19 与作业有关的一些Windows API函数

函 数	说 明
CreateJobObject	创建一个作业对象（用一个可选的名称）
OpenJobObject	通过名称打开一个已有的作业对象
AssignProcessToJobObject	在一个作业中加入一个进程
TerminateJobObject	终止一个作业中的所有进程
SetInformationJobObject	设置限制
QueryInformationJobObject	获得有关作业的信息，比如CPU时间、页面错误次数、进程的数量、进程ID的列表、配额或限制，以及安全限制

下面列出一些你可以为一个作业对象指定的与CPU相关的或者与内存相关的限制。

- 活动进程的最大数量　限制了作业对象中可同时存在的进程数量；
- 作业范围内的用户模式CPU时间限制　限制了该作业中的进程（包括已经在运行的和已经退出的进程）可以消耗的用户模式CPU时间的最大数量。一旦到达这一限制，在默认情况下该作业中的所有进程都将被终止（返回一个错误代码），并且该作业中不能再创建新的进程（除非此限制被重置）。该作业对象变成有信号状态，所以，任何等待该作业的线程都将从等待中释放出来。通过调用*EndOfJobTimeAction*函数，你可以改变这种默认行为；

www.TopSage.com

- 针对每个进程的用户模式CPU时间限制 允许该作业中的每个进程可以累积用户模式CPU时间到一个固定的最大值。当到达此最大值时，进程被终止（没有机会做清理工作）；
- 作业调度类别 为作业中进程的线程设置时间片（或时限）。这一设置仅适用于运行长的、固定时限的系统（Windows Server系统的默认设置）。作业调度类别的值决定了时限单元的长度，如下表所示：

调度类别	时限单元
0	6
1	12
2	18
3	24
4	30
5	36
6	42
7	48
8	54
9	如果实时，则为无限长，否则为60

- 作业的处理器亲和性 为作业中的每个进程设置处理器亲和性掩码（单个线程可以将它的亲和性改变成作业亲和性的任何子集，但是进程不能改变它们的进程亲和性设置）；
- 作业的进程优先级类别 为作业中的每个进程设置优先级类别。线程不能增加它们相对于此类别的优先级（而它们通常是可以的）。任何企图增加线程优先级的请求都会被忽略（在调用*SetThreadPriority*时不会返回错误，但是线程优先级不会增加）；
- 默认的工作集最大值和最小值 为作业中每个进程定义工作集最小值和最大值（这一设置不是针对整个作业范围的——每个进程有它自己的工作集，有同样的最小值和最大值）；
- 进程和作业中已提交的虚拟内存的限制 定义了单个进程或者整个作业可提交的虚拟地址空间的最大数量。

利用Windows的GetQueuedCompletionStatus函数，作业也可以被设置成在一个I/O完成端口对象上排队，而其他的线程可能正在等待此I/O完成端口对象。

你也可以为作业中的进程加上安全性限制。你可以这样设置一个作业：让每个进程都在同一个作业范围的访问令牌下面运行。然后，你可以创建一个作业来限制其中的进程，使其不能模仿或者创建具有本地管理员组访问令牌的进程。而且，你可以应用安全过滤器，当一个作业中的进程的线程模仿客户线程时，可从模仿令牌中去除特定的特权和安全ID（SID）。

www.TopSage.com

最后，你也可以在一个作业中的进程上加上一些有关用户界面的限制。这样的限制包括：限制作业中的进程不能打开指向该作业以外的线程所拥有的窗口的句柄，读和/或写剪贴板，以及通过Windows的*SystemParametersInfo*函数来改变许多用户界面系统参数。

Windows 2000 Datacenter Server有一个称为进程控制管理器（Process Control Manager）的工具，它允许管理员定义作业对象、为作业对象指定各种配额和限制，以及哪些进程（如果运行的话）应该被加入到一个作业中。一个服务组件监视着进程的活动，并且将指定的进程加入到作业中。注意，这个工具不再随Windows Server 2003 Datacenter Edition一起发行，但是，如果将一个Windows 2000 Datacenter Server升级到Windows Server 2003 Datacenter Edition的话，则该工具仍然在系统中。

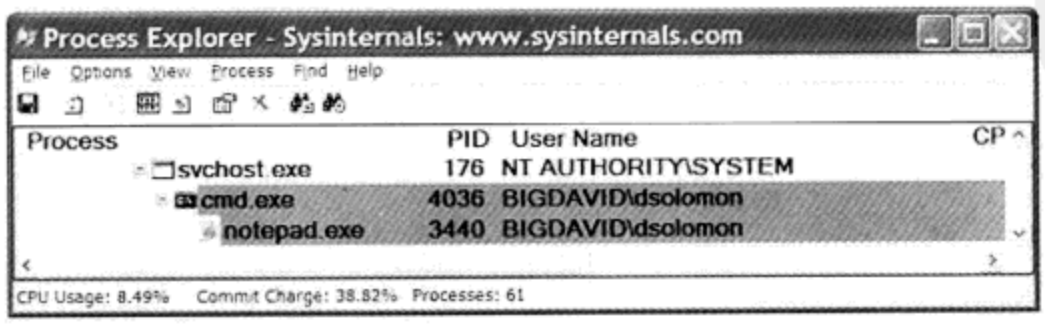


实验：查看作业对象

通过性能工具，你可以查看命名的作业对象（参见Job Object和Job Object Details性能对象）。利用内核调试器的*!job*或*dt nt!_ejob*命令，你可以查看未命名的作业。

为了看一个进程是否与一个作业关联在一起，你可以使用内核调试器的*!process*命令；或者在Windows XP和Windows Server 2003上，你可以使用进程管理器。按照下面的步骤可以创建和查看一个未命名的作业对象。

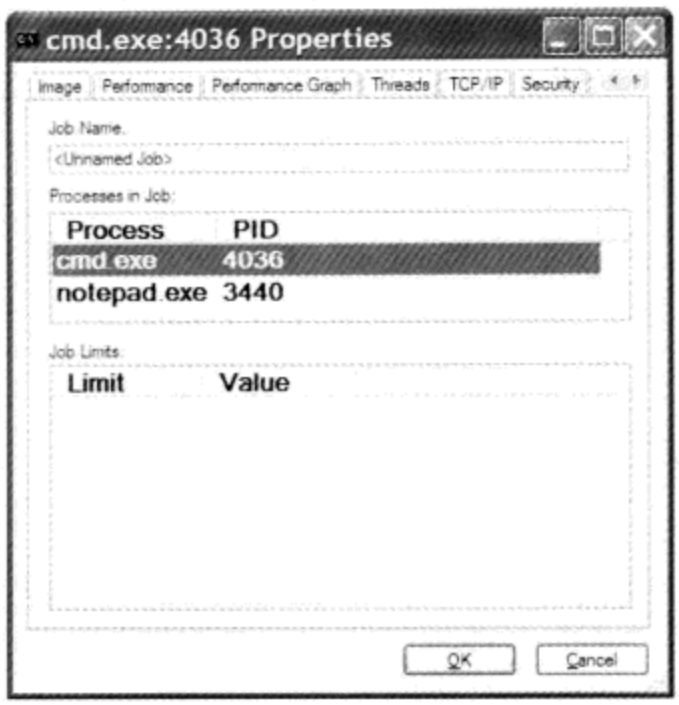
1. 从命令提示符下，使用*runas*命令来创建一个运行命令提示程序的进程（Cmd.exe）。例如，输入**runas /user:<domain>\< username> cmd**。该命令将会提示你输入口令。输入你的口令，将会出现一个命令提示窗口。执行runas命令的Windows服务将会创建一个未命名的作业来包含所有的进程（因而，在注销登录的时候它可以终止这些进程）。
2. 在新的命令提示窗口中，运行Notepad.exe。
3. 然后运行进程管理器，可注意到Cmd.exe和Notepad.exe被加亮显示，作为一个作业的一部分（你只要点击Options、Configure Highlighting，就可以配置用于表示作业成员的加亮显示颜色）。下面的屏幕截图显示了这两个进程。



4. 在Cmd.exe或者Notepad.exe进程上双击鼠标，可以打开进程属性对话框。在进程属性对话框中你将会看到一个Job标签。

www.TopSage.com

5. 点击Job标签，以便查看关于该作业的细节。在这个例子中，该作业没有设置配额，但是它有两个成员进程：



6. 现在在该系统上运行内核调试器（或者在本地内核调试模式下运行WinDbg，或者如果你是在Windows 2000上，也可以运行LiveKd），用!process显示进程列表，找到最新创建的、运行Cmd.exe的那个进程。然后，利用!process <process ID>命令显示进程块，并找到作业对象的地址，最后通过!job命令来显示该作业对象。下面是在一个实际的系统上这些命令的一部分调试器输出：

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

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

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

lkd> !job 85557988
Job at 85557988
  TotalPageFaultCount      0
  TotalProcesses           2
  ActiveProcesses          2
  TotalTerminatedProcesses 0
```


www.TopSage.com

```
  LimitFlags                0
  MinimumWorkingSetSize     0
  MaximumWorkingSetSize     0
  ActiveProcessLimit        0
  PriorityClass             0
  UIRestrictionsClass       0
  SecurityLimitFlags        0
  Token                     00000000
```

7. 最后，使用dt命令来显示该作业对象，可以注意到有关该作业的一些其他域：

```
lkd> dt nt!_ejob 85557988
nt!_EJOB
   +0x000 Event             : _KEVENT
   +0x010 JobLinks          : _LIST_ENTRY [ 0x805455c8 - 0x85797888 ]
   +0x018 ProcessListHead   : _LIST_ENTRY [ 0x8567b8dc - 0x85656324 ]
   +0x020 JobLock           : _ERESOURCE
   +0x058 TotalUserTime     : _LARGE_INTEGER 0x0
   +0x060 TotalKernelTime   : _LARGE_INTEGER 0x0
   +0x068 ThisPeriodTotalUserTime : _LARGE_INTEGER 0x0
   +0x070 ThisPeriodTotalKernelTime : _LARGE_INTEGER 0x0
   +0x078 TotalPageFaultCount : 0
   +0x07c TotalProcesses    : 2
   +0x080 ActiveProcesses   : 2
   +0x084 TotalTerminatedProcesses : 0
   +0x088 PerProcessUserTimeLimit : _LARGE_INTEGER 0x0
   +0x090 PerJobUserTimeLimit : _LARGE_INTEGER 0x0
   +0x098 LimitFlags        : 0
   +0x09c MinimumWorkingSetSize : 0
   +0x0a0 MaximumWorkingSetSize : 0
   +0x0a4 ActiveProcessLimit : 0
   +0x0a8 Affinity          : 0
   +0x0ac PriorityClass     : 0 ''
   +0x0b0 UIRestrictionsClass : 0
   +0x0b4 SecurityLimitFlags : 0
   +0x0b8 Token             : (null)
   +0x0bc Filter            : (null)
   +0x0c0 EndOfJobTimeAction : 0
   +0x0c4 CompletionPort    : 0x8619d8c0
   +0x0c8 CompletionKey     : (null)
   +0x0cc SessionId         : 0
   +0x0d0 SchedulingClass   : 5
   +0x0d8 ReadOperationCount : 0
   +0x0e0 WriteOperationCount : 0
   +0x0e8 OtherOperationCount : 0
   +0x0f0 ReadTransferCount : 0
   +0x0f8 WriteTransferCount : 0
   +0x100 OtherTransferCount : 0
   +0x108 IoInfo            : _IO_COUNTERS
   +0x138 ProcessMemoryLimit : 0
   +0x13c JobMemoryLimit    : 0
   +0x140 PeakProcessMemoryUsed : 0x256
   +0x144 PeakJobMemoryUsed : 0x1f6
   +0x148 CurrentJobMemoryUsed : 0x1f6
   +0x14c MemoryLimitsLock : _FAST_MUTEX
   +0x16c JobSetLinks       : _LIST_ENTRY [ 0x85557af4 - 0x85557af4 ]
   +0x174 MemberLevel       : 0    +0x178 JobFlags          : 0
```


www.TopSage.com

6.7 本章总结

在本章中，我们已经讨论了进程和线程的结构以及作业，看到了它们是如何被创建的，也介绍了Windows如何决定哪些线程应该运行，以及运行多长时间。

本章中许多地方引用到了与内存管理相关的话题。因为线程是在进程内部运行的，而进程在很大程度上定义了一个地址空间，所以，很自然地，下一个话题是，Windows如何进行虚拟内存和物理内存的管理，这正是第7章的主题。

www.TopSage.com

Memory Management

第7章 内存管理

在本章中，你将会学到Microsoft Windows如何实现虚拟内存，以及如何管理虚拟内存的子集，使其保持在物理内存中的知识。我们也将介绍内存管理器的内部结构和组成部件，包括关键的数据结构和算法。在讨论这些机制以前，我们先来看一看内存管理器所提供的一些基本服务，以及关键的概念，比如保留的内存（reserved memory）、提交的内存（committed memory）和共享的内存（shared memory）。

7.1 内存管理器简介

在默认情况下，在32位Windows上，一个进程的虚拟大小是2GB。如果一个映像被特别标记为能感知大地址空间，并且当前系统在引导时打开了一个特殊的开关（本章后面将会讲述），那么，在32位Windows上，一个32位进程可以增长到3GB大小，而在64位Windows上，它可以增长到4GB大小。在64位Windows上，进程虚拟地址空间的大小是7152GB（在IA-64系统上），或8192GB（在x64系统上）（在将来的发行版本中，这个值可能会增加）。

正如你在第2章中（特别是在表2.4中）已经看到的那样，Windows支持的物理内存的最大数量的范围从2GB到1024GB，这取决于你当前运行的是Windows的哪个版本。因为虚拟地址空间可能大于或者小于机器上的物理内存，所以，内存管理器有以下两个主要的任务。

- 将一个进程的虚拟地址空间转译或者映射到物理内存中，这样，当一个在该进程的环境中运行的线程读写虚拟地址空间时，它可以引用到正确的物理地址（在一个进程的虚拟地址空间中，物理上驻留在内存中的那一部分子集被称为**工作集**[*working set*]。本章后面将会更加详细地讲述工作集）。
- 当内存被过度提交（也就是说，当运行线程或者系统代码试图使用比当前可用内存更多的物理内存）时，将内存中的有些内容转移到磁盘上；并且，在以后需要这些内容时，再将它们读回到物理内存中。

除了提供虚拟内存管理以外，内存管理器还提供了一组核心服务，Windows的各个环境子系统也是在这组核心服务的基础上建立起来的。这些服务包括内存映射文件（在内部称为**内存区对象**[*section object*]）、写时复制（copy-on-write）内存，以及为应用程序使用大的、稀疏的地

www.TopSage.com

址空间提供支持。而且，内存管理器提供了一种方法，使进程能够分配和使用的物理内存，可以超过总共可被映射到进程虚拟地址空间中的大小（例如，在超过4GB物理内存的32位系统上）。在本章后面的“地址窗口扩展（Address Windowing Extensions）”一节中将会介绍这一方法。

内存管理器组件

内存管理器是Windows执行体的一部分，所以它位于文件Ntoskrnl.exe中。在HAL中不存在内存管理器的任何组成部分。内存管理器是由以下的组件构成的。

- 一组执行体系统服务，负责分配、释放和管理虚拟内存。绝大多数服务通过Windows API或者内核模式的设备驱动程序接口暴露出来。
- 为解决硬件检测到的内存管理异常的问题，以及代表一个进程使得虚拟页面驻留在内存中，而提供的一个转译无效陷阱处理器，以及一个访问错误陷阱处理器。
- 以下几个关键的组件，分别运行在以下六个不同的内核模式系统线程的环境中。
  - **工作集管理器**（优先级16），平衡集管理器（内核创建的一个系统线程）每秒钟调用它一次，以及当空闲内存的数量降低到某个特定阈值以下时平衡集管理器也会调用它。工作集管理器驱动了总体内存管理策略，比如工作集修剪、页面变老（aging），以及已修改页面的写出等；
  - **进程/栈交换器**（优先级23）执行进程栈和内核线程栈的换入和换出操作。当需要执行换入（inswap）或换出（outswap）操作时，内核中的平衡集管理器和线程调度代码唤醒此线程；
  - **已修改页面写出器**（优先级17）将修改列表（译注：指“已修改页面的列表”）上的脏页面写回到适当的页面文件中。当修改列表的大小需要减小时，此线程被唤醒（参见“已修改页面写出器”一节，可以知道如何改变这一默认值）；
  - **映射页面写出器**（优先级17）将映射文件中的脏页面写到磁盘上。当修改列表的大小需要减小，或者映射文件的页面在修改列表上超过5分钟时，它被唤醒。这第二个已修改页面写出器是必要的，因为它可能会产生页面错误，这些页面错误又会导致请求空闲页面。如果没有空闲页面，而且只有一个已修改页面写出器线程，那么，系统可能会死锁，一直等待空闲页面；

www.TopSage.com

- **解引用段线程**（*dereference segment thread*）（优先级18）负责页面缓存的减少，以及页面文件的增长和缩减。（例如，如果没有虚拟地址空间可用于换页池的增长，那么，该线程修剪页面缓存，以便这些页面所在的换页池能够释放一部分供重新使用）；
- **零页面线程**（优先级0）将空闲列表上的页面清成零，以便有一个零页面缓存可以用来满足将来的零页面错误之需（在有些情况下，内存零化是通过一个更快捷的函数*MiZeroInParallel*来完成的。参见“页面列表的动态变化”一节中的注释）。

本章后面将会详细地介绍这些组件中的每一个。

内部同步

如同Windows执行体中所有其他的组件一样，内存管理器在多处理器系统上是完全重入的，且支持并发执行——也就是说，它允许两个线程以一种相互之间不会破坏对方数据的方式来获得资源。为了达到完全重入的目标，内存管理器使用了几种不同的内部同步机制来控制对其内部数据结构的访问，比如自旋锁和执行体资源（第3章讨论了同步对象）。

内存管理器必须同步访问的系统全局资源包括页面帧编号（PFN, page frame number）数据库（通过一个自旋锁来控制）、内存区对象和系统工作集（通过压栈锁来控制），以及页面文件的创建（通过一个互斥体来控制）。在Windows XP和Windows Server 2003中，许多这样的锁要么已经被彻底去除了，要么已经被优化了，从而导致更少的竞争。例如，在Windows 2000中，通过自旋锁来对系统地址空间和内存提交的变化进行同步；然而，为了增强系统的扩展能力，到了Windows XP以后，这些自旋锁已经被去除了。针对每个进程，需要同步的内存管理数据结构的锁包括工作集锁（当对工作集列表做改变时要持有该锁）和地址空间锁（无论何时，当改变地址空间时要持有该锁）。在Windows 2000中，工作集同步是由一个互斥体来实现的；而到了Windows XP及其以后，则使用压栈锁了，这样提高了并行性和扩展性，因为压栈锁既支持共享访问，也支持互斥访问。

其他一些不再涉及取锁过程的操作包括扣除非换页池和换页池的配额、扣除页面的提交量，以及通过分配和映射那些“通过地址窗口扩展（AWE）函数来分配的物理内存”。而且，用于同步访问那些描述物理内存的结构（PFN数据库）的锁现在很少需要获取了，当需要获取该锁时，持有的时间也更短了。这些变化在多处理器系统上都导致了更强的并行性和扩展性，这是因为，内存管理器由于另一个CPU正在改变一个全局数据结构而可能不得不被阻塞的次数已经被减少了，甚至被消除了。

www.TopSage.com

配置内存管理器

如同Windows中的绝大多数组件一样，内存管理器企图能够自动地为各种规模和类型的系统上的各种变化的负载，提供最优的系统性能。虽然在注册表键HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management下面你只能加入或者修改有限数量的注册表值，以改写某些默认的性能计算，但是，一般而言，内存管理器的默认计算对于大多数工作负载已经足够了。

控制内存管理器策略决策的许多阈值和限制，是在系统引导时根据内存大小和产品类型来计算的。（Windows 2000 Professional，以及Windows XP Professional和Home版本针对桌面交互式用途作了优化，而Windows Server系统则针对运行服务器应用程序作了优化。）这些值被保存在各种内核变量中，以便将来内存管理器使用它们。要想找到这些变量中的某一些，你可以搜索Ntoskrnl.exe中以*Mm*开头的全局变量（包含了单词“maximum”或“minimum”）。（参见第2章的实验“看一看尚未文档化的接口”。）



警告 虽然你可以找到对某些变量的引用，但是你不应该改变这些引用。Windows已经测试过，用这些可计算值的当前可能组合能够正确地运行。在一个正在运行的系统上改变这些内核变量的值可能会导致不可预测的系统行为，包括系统停止甚至崩溃。

检查内存的使用情况

Memory和Process性能计数器对象提供了绝大多数关于系统和进程的内存使用情况的访问途径。贯穿本章全篇，我们在讲述内存管理组件时，也会介绍到包含了与相应组件有关的信息的性能计数器。

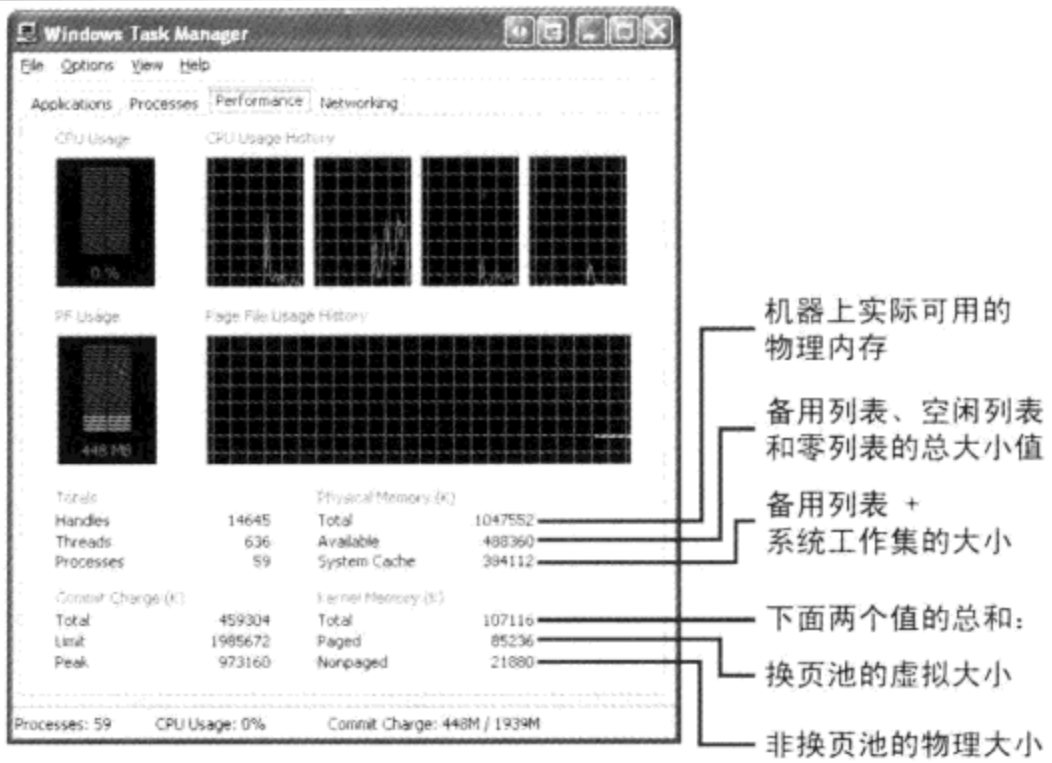
除了性能工具以外，在Windows支持工具箱和Windows资源工具箱中还有许多工具可以显示不尽相同的内存使用信息。在本章中我们将会介绍一些有关的例子和实验。然而，这里提醒一下：不同的工具在显示内存信息的时候，会使用不同的名称，有时候甚至是不一致或者混淆的名称。下面的实验说明了这一点（在后面的章节中我们将会解释这个例子中用到的术语）。



实验：查看系统内存信息

下图是从Windows XP上截取的任务管理器中的Performance页面，它显示了基本的系统内存信息。这些信息只是通过性能计数器可以获取到的详细内存信息中的一部分而已。

www.TopSage.com



Pmon.exe（位于Windows支持工具箱中）和Pstat.exe（位于Platform SDK中）都显示了系统和进程的内存信息。以下Pstat输出中的标记说明了所报告的信息（有关提交总量和限制的解释，请参见表7.15）。

```
                                                    所有进程工作集
                                                    的总量(不是总的
                                                    进程内存使用量，
                                                    该项会重复计算
                                                    共享的页面)
                                                                        驻留的
                                                                        换页池
总物理内存          备用列表、空闲列表                                  驻留的
                    和零列表的总量                                      系统代码
Memory:  64692K Avail: 3676K TotalWs:  62836K  InRam Kernel: 3788K     P: 9604K
Commit: 111848K/85808K  Limit: 182400K Peak: 113732K  Pool N: 2344K    P:14448K
                                                                        换页池的
                                                    非换页池的          虚拟大小
                                                    物理大小
                                系统工作集的大小
                                (不只是文件缓存的大小)
User Time    Kernel Time     Ws  Faults Commit Pri  Hnd  Thd Pid  Name
                          20032 194378                            File Cache
0:00:00.000  8:03:55.734    16       1      0   0    0    2   0  Idle Process
0:00:00.000  0:01:24.421    32    1601     36   8  140   27   2  System
0:00:00.140  0:00:00.468   200    2008    164  11   35    6  20  smss.exe
0:00:00.640  0:00:02.843   716    8625   1588  13  145    9  30  csrss.exe
```

为了看清楚换页池和非换页池的特殊使用量，可使用“监视内存池使用量”一节中讲述的Poolmon工具。

www.TopSage.com

最后，内核调试器中的*!vm*命令利用一组与内存相关的性能计数器，显示了基本的内存管理信息。如果你正在检查一个崩溃转储或者一个停止的系统，则该命令可能会很有用。下面是一个例子输出：

```
kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:    32620   (  130480 Kb)
        Page File: \??\C:\pagefile.sys
           Current:    204800Kb Free Space:    101052Kb
           Minimum:    204800Kb Maximum:       204800Kb
        Available Pages:     3604   (   14416 Kb)
        ResAvail Pages:     24004   (   96016 Kb)
        Modified Pages:       768   (    3072 Kb)
        NonPagedPool Usage:  1436   (    5744 Kb)
        NonPagedPool Max:   12940   (   51760 Kb)
        PagedPool 0 Usage:   6817   (   27268 Kb)
        PagedPool 1 Usage:    982   (    3928 Kb)
        PagedPool 2 Usage:    984   (    3936 Kb)
        PagedPool Usage:     8783   (   35132 Kb)
        PagedPool Maximum:  26624   (  106496 Kb)
        Shared Commit:       1361   (    5444 Kb)
        Special Pool:           0   (       0 Kb)
        Free System PTEs:  189291   (  757164 Kb)
        Shared Process:      3165   (   12660 Kb)
        PagedPool Commit:    8783   (   35132 Kb)
        Driver Commit:       1098   (    4392 Kb)
        Committed pages:    45113   (  180452 Kb)
        Commit limit:       79556   (  318224 Kb)

        Total Private:      30536   (  122144 Kb)
            IEXPLORE.EXE     3028   (   12112 Kb)
            svchost.exe      2128   (    8512 Kb)
            WINWORD.EXE      1971   (    7884 Kb)
            POWERPNT.EXE     1905   (    7620 Kb)
            Acrobat.exe      1761   (    7044 Kb)
            winlogon.exe     1361   (    5444 Kb)
            explorer.exe     1300   (    5200 Kb)
            livekd.exe       1015   (    4060 Kb)
            hh.exe            960   (    3840 Kb)
```



实验：计算物理内存的使用

通过将性能计数器中的信息与内核调试器命令的输出结合起来，你可以更加精确地知道一台运行Windows的机器上的物理内存使用情况。要想查看那些由性能计数器提供的内存使用信息，你可以运行性能工具，然后加入计数器来查看以下的信息（如果你将性能工具的图上的垂直比例尺最大值改成1000，则可以更容易看到结果）。

www.TopSage.com

- **总的进程工具集大小** 要查看此信息，请选择Process性能对象，和_Total进程实例的Working Set计数器。此数值将会大于实际的总体进程内存使用量，因为共享页面在每个进程工作集中都被计算了。要想得到一个更为精确的进程内存使用量值，可以从该机器的总物理内存中减去空闲内存（可用字节）、操作系统使用的内存（非换页池、驻留的换页池，以及驻留的操作系统和驱动程序代码），以及修改列表的大小。减去这些值以后剩下来的就是所有进程使用的内存。将这个值与性能工具报告的总进程工作集大小作一下比较，你可以对进程之间共享的内存数量有一个大致的了解。虽然检查进程的物理内存使用量很有意思，但是，一个更加值得关注的量是，进程私有的、已提交虚拟内存的使用量，因为内存泄漏时表现出来的现象是私有虚拟内存的递增，而不是工作集大小的递增。在某个点上，内存管理器将阻止一个进程继续增加物理内存大小，但它可以继续增大它的虚拟内存大小，直至到达系统全局范围的提交限制——当前系统上可以使用的私有的已提交虚拟内存的最大值（如果该进程是一个作业的成员，并且该作业有一个全作业范围的或进程的虚拟内存限制，则该进程的虚拟内存最大限制值可能还要更小一些）。有关更多的信息，请参见“页面文件”一节。

- **总的系统工作集大小** 要查看此信息，请选择Memory对象，以及Cache Bytes计数器。正如“系统工作集”一节中所说明的那样，总的系统工作集大小值包括的不仅仅是缓存的大小，它还包含了换页池的子集、可换页的操作系统代码，以及驻留在系统工作集中的可换页驱动程序代码。

- **非换页池的大小** 只需加入Memory: Pool Nonpaged Bytes计数器就可以查看此信息。

- **空闲列表、零列表和备用列表的大小** 只需加入Memory: Available Bytes计数器就可以查看这些列表的大小（使用*!memusage*内核调试器命令可以得到这些列表各自的大小）。

现在你的图表或者报告中包含了各种物理内存的数值表达，除了以下两部分你无法通过性能计数器来获得：

- 非换页的操作系统和驱动程序代码；
- 修改列表和已修改-不写出（modified no-write）的页面列表。

尽管通过内核调试器*!memusage*命令你可以很容易地得到修改列表和已修改-不写出的列表，但是，你没有很容易的办法能够得到非换页的操作系统和驱动程序代码的大小。

www.TopSage.com

7.2 内存管理器提供的服务

内存管理器提供了一组系统服务来完成以下各种任务：分配和释放虚拟内存，在进程之间共享内存，将文件映射到内存中，将虚拟页面刷新到磁盘中，获得有关一定范围内虚拟页面的信息，改变虚拟页面的保护属性，以及将虚拟页面锁在内存中。

如同其他的Windows执行体服务一样，内存管理服务允许其调用者提供一个进程句柄，来指明哪个特定进程的虚拟内存需要进行操作。因此，调用者既可以操作或维护它自己的内存，也可以操作另一个进程的内存（当然要有正确的许可）。例如，如果一个进程创建了一个子进程，则在默认情况下它有权操作该子进程的虚拟内存。因此，父进程通过调用虚拟内存服务，并且把子进程的句柄当作参数传递过去，就可以代表子进程来分配、释放、读和写内存。Windows子系统利用这一特性来管理其客户进程的内存，而且，此特性对于调试器的实现也是非常关键的，因为调试器必须能够读写被调试进程的内存。

这些服务中的大多数是通过Windows API暴露给客户的。Windows API有三组函数可用来管理应用程序中的内存：页面粒度的虚拟内存函数（*Virtualxxx*）、内存映射文件函数（*CreateFileMapping*、*MapViewOfFile*），以及堆函数（*Heapxxx*和老的接口*Localxxx*与*Globalxxx*）。（这一节后面我们将介绍堆管理器。）

内存管理器也为执行体内部的其他内核模式组件和设备驱动程序提供了许多服务，比如分配和释放物理内存，将页面锁在物理内存中以便进行直接内存访问（DMA）的传输。这些函数均以前缀*Mm*开头。而且，以*Ex*开头的执行体支持例程虽然严格来说不是内存管理器的一部分，但是它们被用于从系统堆（换页池和非换页池）中分配和释放内存，以及用来维护预读列表（look-aside list）。在本章后面的“系统内存池”一节中我们将会介绍这些话题。

虽然我们将会提到那些为了设备驱动程序而提供的Windows函数，以及内核模式的内存管理和内存分配例程，但是，我们不会涉及接口和编程的细节，而仅仅介绍这些函数的内部操作而已。关于这些函数及其接口的完整描述信息，请参考MSDN中的Platform SDK（Software Development Kit）和DDK（Device Driver Kit）文档。

大页面和小页面

虚拟地址空间被分成以页面为单位。这是因为硬件内存管理单元在页面的粒度上，将虚拟地址转译成物理地址。因此，页面是硬件级别上的最小保护单位 （在“保护内存”一节中讲述了各种页面保护机制）。页面尺寸有两种：小的和大的。根据硬件体系结构的不同，实际的尺寸值有所不同，如表7.1所示。

www.TopSage.com

表7.1 页面尺寸

体系结构	小页面尺寸	大页面尺寸
x86	4 KB	4 MB （在PAE系统上为2 MB）
x64	4 KB	2 MB
IA64	8 KB	16 MB

大页面的优势是，当引用同一大页面内其他的数据时，地址转译的速度更快。之所以有这样的优势是因为，第一次引用一个大页面内的某个字节，会触发硬件的地址转译快查缓冲区（或TLB，在“地址转译快查缓冲区”一节中讲述）把必要的信息存放在其高速缓存中，以便可以转译任何引用到该页面内其他字节的地址访问。如果使用小页面的话，则对于同样的虚拟地址范围，需要更多的TLB项目，这样，随着不断有新的虚拟地址需要转译，TLB项目的反复利用也随之增加。更进一步，这意味着，当引用到所缓存页面以外的其他页面中的虚拟地址时，不可避免地要回转到页表结构中。TLB是一个非常小的高速缓存，因此大页面可以更好地利用这一有限的资源。

为了充分地利用大页面的优势，在一台被认为有足够内存的系统（参见表7.2中给出的最小要求）上，Windows用大页面来映射核心的操作系统映像（Ntoskrnl.exe和Hal.dll）以及核心的操作系统数据（比如非换页池的初始部分，以及一些描述了每个物理内存页面状态的数据结构）。对于I/O空间请求（由设备驱动程序调用*MmMapIoSpace*），如果此请求满足大页面的长度和对齐条件，则Windows也会自动利用大页面来映射。最后，Windows也允许应用程序用大页面来映射它们的映像、私有内存，以及由页面文件支撑的内存区（pagefile-backed section）。（参考*VirtualAlloc*函数的MEM_LARGE_PAGE标志。）你也可以指定其他的设备驱动程序被映射成大页面，做法是，加入一个多字符串的注册表值HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageDrivers，并且以单独的null结尾字符串的形式来指定这些驱动程序的名称。

表7.2 大页面的最低内存要求

操作系统	为了使用大页面而要求的最低内存数量
Windows 2000	>127 MB
Windows XP、Windows Server 2003	>255 MB

大页面的一个副作用是，因为每个大页面必须用一个单独的保护项来映射（因为硬件内存保护是以每个页面为基础的），所以，如果一个大页面同时包含了只读的代码和可读写的数据，则该页面必须被标记为读/写，这意味着其中的代码也是可写的。这意味着设备驱动程序或者其他的内核模式代码可能（由于一个程序错误）修改一段本该是只读的操作系统和驱动程序代码，却不会引发任何一个内存访问违例。然而，如果使用小页面来映射内核，则NTOSKRNL.EXE和HAL.DLL的只读部分将被映射成只读的页面。

www.TopSage.com

虽然这样做降低了地址转译的效率，但是，如果一个设备驱动程序（或者其他的内核模式代码）企图修改操作系统的只读部分，则一旦手指触及到被破坏的指令，系统立即崩溃；相比之下，“先允许代码指令被破坏，而当以后其他某个组件在这些破坏的代码上绊倒时系统再崩溃”，显然这种情形更难以诊断错误。如果你怀疑自己遇到了内核代码被破坏的情形，则可以使用驱动程序检验器（Driver Verifier，本章后面讲述）。——这将会禁止使用大页面。

保留的和提交的页面

一个进程地址空间中的页面可以是空闲的、保留的，或者提交的。应用程序可以首先保留地址空间，然后提交该地址空间中的页面。或者，它们可以在同一个函数调用中保留和提交页面。这些服务是通过Windows的*VirtualAlloc*和*VirtualAllocEx*函数暴露出来的。

保留地址空间的做法，类似于一个线程保留一段虚拟地址范围以便将来使用。试图访问已保留的内存，将导致访问违例，因为该页面尚未被映射到任何可解析此引用的存储介质中。

提交的页面是指这样的页面：当它们被访问的时候，最终被转译至物理内存中的有效页面。提交的页面或者是私有的、不共享的，或者被映射到一个内存区的视图上（该内存区可能还被其他的进程映射了，也可能没有）。在接下来的两节“共享内存和映射文件”和“内存区对象”中将会介绍内存区。

如果页面是进程私有的，并且以前还没有被访问过，那么，当它们第一次被访问的时候才真正被创建起来，并且被初始化为零（*demand zero*）。私有的提交页面以后可以被操作系统自动写到页面文件中，如果对内存的需求使得有必要这样做的话。私有的提交页面对于其他的进程是不可访问的，除非通过一些跨进程的内存函数，比如*ReadProcessMemory*或*WriteProcessMemory*，来访问它们。如果提交的页面被映射为某个映射文件的一部分，那么，当它们被访问的时候，它们可能需要从磁盘中读回来，除非在此之前，该进程已经访问过此页面了，或者另一个映射了同样文件的进程已经访问过此页面了（参见本章稍后的“共享内存和映射文件”一节）。

随着页面被从进程的工作集转移到修改列表中，并最终被转移到磁盘上，在此过程中，页面被通过正常的“修改页面写操作”写到磁盘中（在本章后面将会介绍工作集和修改列表）。通过显式调用*FlushViewOfFile*，映射文件页面也可以被写回到磁盘上。

通过*VirtualFree*或*VirtualFreeEx*函数，你可以将页面解除提交，或者释放地址空间。解除提交和释放地址空间之间的差异，类似于保留和提交页面之间的差异——解除提交后的内存仍然是被保留的，但是释放的内存既不是提交的，也不是保留的（其地址是空闲的或自由的[free]）。

利用“先保留内存再提交内存”这种两步骤过程，可以降低内存使用量，它将提交页面的操作推迟到真正需要这些页面的时候，同时又可以保持虚拟相邻的便利性。在Windows中，保

www.TopSage.com

留内存操作相对比较快速，其开销也很小，因为它并不消耗任何提交的页面（这是一种宝贵的系统资源），也不消耗进程的页面文件配额（它是指一个进程可以消耗的提交页面的数量限制——不必是页面文件的空间）。所需要更新或者建立的仅仅是一些相对较小的内部数据结构，它们代表了进程地址空间的状态（在本章后面我们将解释这些数据结构，称为**虚拟地址描述符**[*virtual address descriptor*]，或者VAD）。

对于那些可能需要大块连续的虚拟内存缓冲区的应用程序来说，先保留再提交内存的做法是非常有用的：不需要为整个区域提交页面，而可以先保留地址空间，当以后需要的时候再提交页面。这一技术在操作系统中的一个使用例子是，每个线程的用户模式栈。当创建一个线程的时候，系统会保留一个栈（默认是1MB；你可以通过*CreateThread*函数调用来改变栈的大小，或者利用/STACK链接器标志在映像基础上改写栈的大小）。在默认情况下，栈中的初始页面被提交，下一个页面被标记为一个守护页面，但它并没有被提交，它的作用是，可以捕捉住对栈的已提交部分之外的引用，并扩展这一部分。

锁住内存

一般而言，最好是让内存管理器来决定哪些页面该留在物理内存中。然而，可能在一些特殊的情况下，必须将页面留在物理内存中。通过以下两种方式，页面可以被锁在物理内存中。

- Windows应用程序可以调用*VirtualLock*函数来锁住其进程工作集中的页面。一个进程可以锁住的页面数量不能超过它的最小工作集大小减去8个页面。因此，如果一个进程需要锁住更多的页面，它可以利用*SetProcessWorkingSetSize*函数来增大它的工作集最小值（关于该函数，参见“工作集管理”一节）；
- 设备驱动程序可以调用内核模式函数*MmProbeAndLockPages*、*MmLockPagableCodeSection*、*MmLockPagableDataSection*或*MmLockPagableSectionByHandle*。利用这种机制来锁住的页面一直留在内存中，除非它们被显式地解锁。虽然驱动程序可以锁在内存中的页面数量没有强制性的配额限制，但是它不可能锁住比当前可用内存能容纳的页面数量更多的页面。

分配粒度

Windows将保留的进程地址空间中的每个区域对齐成从一个完整的边界开始，此边界由系统的**分配粒度**（*allocation granularity*）值所定义。通过Windows的*GetSystemInfo*函数，你可以获得此分配粒度值。当前，该值是64KB。之所以选择这个大小，是因为，如果将来的处理器加入了对大页面大小（比如，直至64KB）的支持，或者加入了被索引的缓存，它要求全系统范围内从物理到虚拟的页面对齐，那么，那些对内存分配对齐作了假设的应用程序，要求改变这一假设条件的风险也随之降低 （Windows内核模式代码并不受制于同样的约束；它可以在单个页面的粒度上保留内存）。

www.TopSage.com

最后，当地址空间的一个区域被保留时，Windows确保该区域的大小和基地址总是系统页面大小的整数倍，而不管它们可能会怎么样。例如，因为x86系统使用4KB的页面，所以，如果你试图保留一个18KB大小的内存区域，那么，在一个x86系统上实际被保留的数量将是20KB。如果你指定了这个18KB区域的基地址为3KB，则实际被保留的数量将是24KB。

共享内存和映射文件

如同大多数现代操作系统一样，Windows提供了一种在多个进程和操作系统之间共享内存的机制。**共享内存**（*shared memory*）可以被定义为：对于多个进程可见的内存，或者出现在多个进程虚拟地址空间中的内存。例如，如果两个进程使用了同样的DLL，那么，只需将引用该DLL的代码页面加载到物理内存中一次，然后在所有映射了此DLL的进程之间共享这些页面，如图7.1所示。这种做法显然是很有意义的。

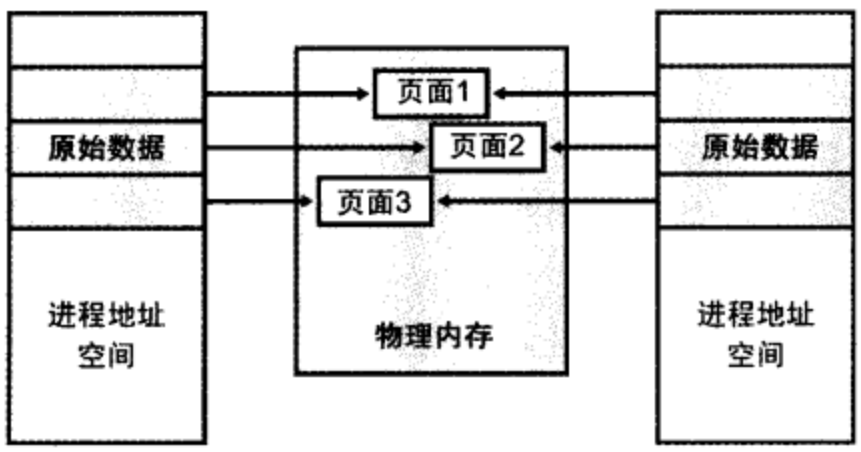


图7.1 进程之间共享内存

每个进程仍然维护它的私有内存区域，在其中存储私有数据，但是，程序指令和非修改的数据页面可以被共享，而不会有任何危害。正如我们在后面将要解释的那样，这种共享是自动进行的，因为可执行映像中的代码页面是以“仅可执行”的方式被映射的，而可写的页面被映射为“写时复制”（关于“写时复制”的更多信息，请参见“写时复制”一节）。

内存管理器中用来实现共享内存的底层原语是**内存区对象**（*section object*），在Windows API中它也被称为**文件映射对象**（*file mapping object*）。本章后面的“内存区对象”一节讲述了内存区对象的内部结构和实现细节。

内存管理器中的这一基础原语可被用来映射虚拟地址，不管它们是在主内存中还是在页面文件中，或者在其他文件中，应用程序访问这些文件中的内容就好像它们在内存中一样。内存区对象可以被一个进程打开，也可以被多个进程打开，换句话说，内存区对象并不一定等同于共享内存。

www.TopSage.com

内存区对象可以被连接到磁盘上一个打开的文件（称为映射文件），或者被连接到提交的内存上（以便提供共享内存）。映射至提交内存的内存区被称为**由页面文件支撑的内存区**（*page file backed section*），因为如果对内存的需求表明了有必要的话，这些页面被写到页面文件中（因为Windows可以在没有页面文件的情况下运行，所以，由页面文件支撑的内存区实际上可能是由物理内存来“支撑”的）。如同任何其他的对于用户模式可见的页面（比如私有的已提交页面）一样，共享的已提交页面第一次被访问的时候总是填充了零。

为了创建一个内存区对象，可以调用Windows的*CreateFileMapping*函数，同时指定要映射的文件句柄（对于一个由页面文件支撑的内存区，则指定INVALID_HANDLE_VALUE），以及一个可选的名称和安全描述符。如果内存区有名称的话，其他的进程就可以通过*OpenFileMapping*来打开该内存区。或者，你也可以通过句柄的继承性（当打开或者创建句柄的时候指定该句柄是可继承的）或者句柄复制（使用*DuplicateHandle*）来授权对于内存区对象的访问。设备驱动程序还可以利用*ZwOpenSection*、*ZwMapViewOfSection*和*ZwUnmapViewOfSection*来操作内存区对象。

内存区对象可以引用比进程的地址空间大得多的文件 （如果内存区对象是由页面文件来支撑的话，则在页面文件中必须有足够的空间来包含该对象）。为了访问一个非常大的内存区对象，一个进程可以只映射该内存区对象中它所请求的那一部分（称为该内存区的一个*视图*[*view*]），其做法是，调用*MapViewOfFile*函数，然后指定要映射的范围。映射视图的功能使得进程可以节约地址空间，因为内存区对象的视图只有在真正需要的时候才必须被映射到内存中。

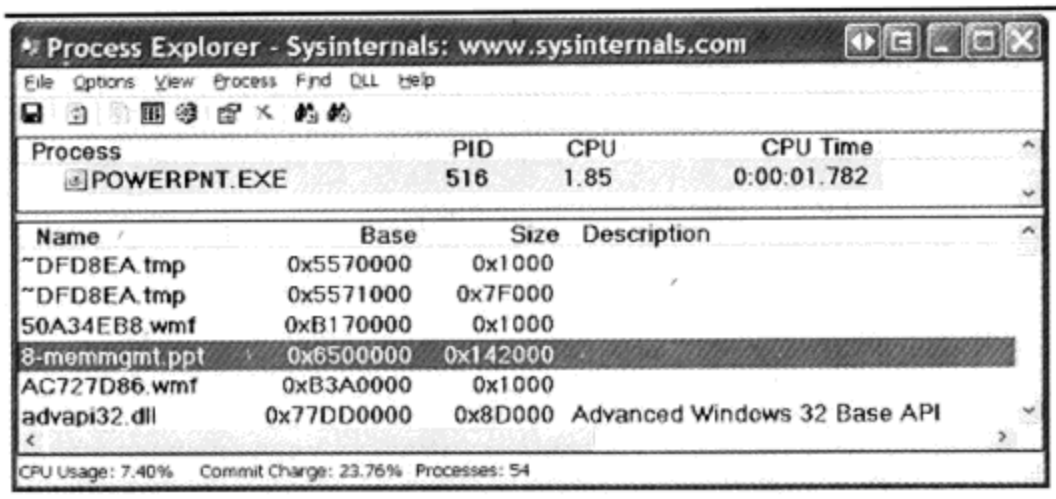
Windows应用程序可以使用映射文件来方便地执行文件I/O，它们只需使映射文件出现在其地址空间中即可。用户应用程序并不是内存区对象的惟一客户：映像加载器利用内存区对象，将可执行映像文件、DLL和设备驱动程序映射到内存中；而缓存管理器则使用内存区对象来访问缓存文件中的数据（有关缓存管理器如何与内存管理器集成在一起，请参见第11章）。共享的内存区是如何实现的？本章后面分别从地址转译和内部数据结构两个方面对此做了解释。



实验：查看内存映射文件

利用Sysinternals的进程管理器程序，你可以列出一个进程中的内存映射文件。为了通过进程管理器来查看内存映射文件，请配置下面的窗格，让它显示DLL视图（选择View，Lower Pane View，DLLs）。 注意，这不仅仅是一个DLL列表——它代表了该进程地址空间中的所有内存映射文件。在这些映射文件中，有的是DLL，一个是正在运行的映像文件（EXE），其他的项目可能代表了内存映射数据文件。例如，下图中的进程管理器显示了一个Microsoft PowerPoint进程，它将正在编辑的PowerPoint文件也映射到其地址空间中了。

www.TopSage.com



通过点击Find、DLL，你也可以搜索内存映射文件。当你想要确定哪个（些）进程正在使用一个DLL，并且你想要替换此DLL的时候，这个功能非常有用。

最后，将一个进程中加载的DLL列表与另一个系统中正在运行的同一程序的另一个实例相比较，可能有助于找出DLL配置的问题，比如一个不正确的DLL版本被加载到某个进程中。此问题也被亲切地称为“DLL黑洞”。

保护内存

正如第1章中解释的那样，Windows提供了内存保护，所以，用户进程不可能有意或无意地破坏另一个进程或者操作系统本身的地址空间。Windows通过以下4种基本方法来提供这种保护。

第一，所有被内核模式系统组件用到的系统全局范围的数据结构和内存池只有在内核模式下才可以访问，用户模式线程不能够访问这些页面。如果它们试图要这样做的话，硬件会产生一个错误，然后内存管理器向用户模式线程报告一个访问违例。



> **注** 相反地，Microsoft Windows 95、Microsoft Windows 98和Microsoft Windows Millennium Edition有一些页面虽然在系统地址空间中，但是在用户模式是可写的，因此，一个错误的应用程序可以破坏关键的系统数据结构，导致系统崩溃。

第二，每个进程有一个独立的、受保护的、私有的地址空间，任何属于其他进程的线程都不能访问此地址空间。惟一例外的是，如果该进程决定与其他的进程共享页面，或者如果另一个进程对于该进程对象有虚拟内存读写访问权限，则可以使用*ReadProcessMemory*和*WriteProcessMemory*函数。每次当一个线程引用一个地址的时候，虚拟内存硬件介入其中，与内存管理器合作将虚拟地址转译成物理地址。Windows控制了虚拟地址的转译方式，从而确保在一个进程中运行的线程不会错误地访问属于另一个进程的页面。

www.TopSage.com

第三，对从虚拟地址到物理地址的转译过程提供了隐式的保护，除此以外，Windows支持的所有处理器还提供了某种形式的由硬件控制的内存保护（比如读/写、只读，等等）。此种保护的具体细节随处理器的不同而有所不同。例如，在进程地址空间中的代码页被标记为只读，因此用户线程不能修改这些代码页。

表7.3列出了Windows API中定义的内存保护选项（参见*VirtualProtect*、*VirtualProtectEx*、*VirtualQuery*和*VirtualQueryEx*函数）。

表7.3 Windows API中定义的内存保护选项

属 性	说 明
PAGE_NOACCESS	对该区域中的任何读、写或者执行代码的操作，都会引发一个访问违例
PAGE_READONLY	试图执行写内存的操作（在支持“不可执行”的处理器上，执行其中的代码）会引发一个访问违例，但读操作是允许的
PAGE_READWRITE	页面是可读写的，但不可执行
PAGE_EXECUTE[1]	试图在该区域中写入代码将引发一个访问违例，但执行其中的代码（以及在所有现有的处理器上读取代码）是允许的
PAGE_EXECUTE_READ*	试图在该区域中写入代码将引发一个访问违例，但执行和读操作是允许的
PAGE_EXECUTE_READWRITE*	页面是可读写的、可执行的——任何动作都不会引发访问违例
PAGE_WRITECOPY	若试图在该区域中写一个内存页面，则系统为当前进程提供该页面的一份私有拷贝。在支持“不可执行”的处理器上，试图执行该区域中的代码将引发一个访问违例
PAGE_EXECUTE_WRITECOPY	若试图在该区域中写一个内存页面，则系统为当前进程提供该页面的一份私有拷贝。读和执行该区域中的代码是允许的（在这种情况下没有生成拷贝）
PAGE_GUARD	试图读或者写一个守护页面将产生一个EXCEPTION_GUARD_PAGE异常，并且关闭守护页面状态。因此，守护页面可以被当作一次性报警器。注意，这一标志可以与本表中除了PAGE_NOACCESS之外的任何其他页面保护标志一起指定
PAGE_NOCACHE	使用没有被缓存的物理内存。这不适合作为一般用法来推荐。它对于设备驱动程序是很有用的——例如，映射一个视频帧缓冲区时要求不使用缓存
PAGE_WRITECOMBINE	允许写组合的内存访问。当该标志被允许时，处理器可能把写内存的请求缓存起来，以优化性能。例如，如果多个写请求指向同样的地址，则只有最新的写请求才可能被执行

[1] 在具有必要的硬件支持的处理器（比如x64、IA-64以及将来的x86处理器）上，Windows XP Service Pack 2和Windows Server 2003 Service Pack 1及其以后的版本支持“不可执行（No Execution）”的保护。在Windows的早期版本，以及不支持“不可执行”保护的处理器上，所有的页面许可权都允许执行。下一节更加详细地介绍了“不可执行”的保护。

www.TopSage.com

第四，共享内存区对象具有标准的Windows访问控制列表（ACL），当进程试图打开这些区对象时，该ACL将会被检查，从而限制了只有那些具有适当权限的进程才可以访问共享内存。当一个线程创建一个内存区来容纳一个映射文件时，安全性也开始起作用。为了创建该内存区，此线程必须至少具有读访问底层文件对象的权限，否则该操作将失败。

一旦一个线程已经成功地打开了一个内存区句柄，它以后的动作仍然受到内存管理器和前述基于硬件的页面保护机制的限制。一个线程可以改变在一个内存区的虚拟页面上的页面层次的保护，条件是，这种改变不会违反该内存区对象的ACL中的许可限制。例如，内存管理器允许一个线程将一个只读内存区的页面改变成具有写时复制访问权，但不能改变成具有读/写访问权。写时复制是允许的，因为它对于共享此数据的其他进程没有任何影响。

Windows之所以是一个健壮的、可靠的操作系统，并且不受应用程序错误的影响，以上四种基本内存保护机制正是原因之一。

“不可执行”页面保护

虽然Windows内存管理API总是在编程接口中定义了页面保护位，以允许指定这些页面是否包含可执行的代码，但是，只有到了Windows XP Service Pack 2和Windows Server 2003 Service Pack 1以后，在那些有“不可执行”硬件保护的处理器上这种能力才得以支持，这样的处理器包括所有的AMD64处理器（包括AMD Athlon64和AMD Opteron）、特定专用的32位AMD处理器（精选的AMD Sempron处理器——在AMD的产品文档中有细节说明）、Intel IA-64，以及Intel Pentium 4和Xeon处理器（支持Intel Extended Memory 64 Technology, EM64T）。

不可执行页面保护（也称为数据执行保护，或者DEP[Data Execution Prevention]）意味着，试图将控制权传递给一个被标记为“不可执行”的页面中的指令，将会产生一个访问错误。这可以避免某些类型的病毒利用系统中的错误来执行数据页面中存放的代码。如果试图在内核模式下执行一个被标记为不可执行的页面中的代码，则系统将崩溃，其错误检查码为ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY。如果类似的操作发生在用户模式中，则试图引用非法指令的线程接收到一个STATUS_ACCESS_VIOLATION （0xc0000005）异常。如果一个进程想要分配一块需要被执行的内存，则它必须显式地标记这些页面，它可以在页面粒度的内存分配函数中指定PAGE_EXECUTE、PAGE_EXECUTE_READ、PAGE_EXECUTE_READWRITE或PAGE_EXECUTE_WRITECOPY。

在64位版本的Windows上，这种执行保护总是被应用到所有的64位程序和设备驱动程序上，而且不能被关掉。对于32位程序的执行保护取决于系统配置中的设置，稍后将进一步介绍。在64位Windows上，执行保护被应用到线程栈（用户模式和内核模式都适用）、未被标记为“可执行”的用户模式页面、内核换页池，以及内核会话池（有关内核内存池的描述信息，请参见“系统内存池”一节）。然而，在32位Windows上，执行保护仅仅被应用于线程栈和用户模式页面，而没有被应用到换页池和会话池上。而且，在32位Windows上，当执行保护被打开的时

www.TopSage.com

候，系统自动引导到PAE模式中（自动选择PAE内核，\Windows\System32\Ntkrnlpa.exe）。关于PAE的描述信息，请参见“物理地址扩展（PAE）”一节。

针对32位程序的执行保护是否起作用，取决于Boot.ini中的/NOEXECUTE开关。通过点击My Computer、Properties、Advanced、Performance Settings，再到Data Execution Prevention页面（见图7.2）中，你可以改变这一设置。当你通过DEP设置对话框选择了“不可执行”保护时，Boot.ini文件会被修改，其中加入了适当的/NOEXECUTE开关。关于此开关的各种选择，以及它如何对应于DEP设置选项，请参考表7.4。注册表键HKLM\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers下的值列出了那些排除在执行保护之外的32位应用程序，值的名称是可执行文件的全路径，数据被设置为“DisableNXShowUI”。

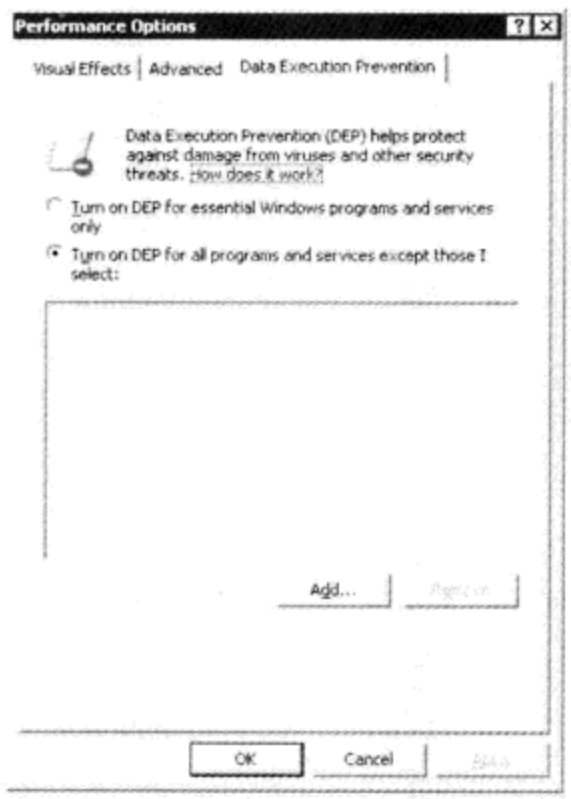


图7.2 数据执行保护（DEP）设置

在Windows XP（64位和32位版本）上，针对32位应用程序的执行保护被默认配置成：仅应用于核心的Windows操作系统可执行文件（/NOEXECUTE=OPTIN）上，这样，对于那些必须执行数据页面（未被专门标记为可执行的）中代码的32位应用程序来说，它们不会被打破。在Windows Server 2003系统上，针对32位应用程序的执行保护被默认配置成：应用于所有的32位程序（/NOEXECUTE=OPTOUT）。



注 为了获得那些已被保护的程序的完整列表，可安装Windows Application Compatibility Toolkit（Windows应用兼容性工具箱，可从*www.microsoft.com*下载），并运行Compatibility Administrator Tool（兼容性管理员工具）。点击System Database、Applications和Windows Components，在右侧的窗格上，将会显示被保护的可执行文件的列表。

www.TopSage.com

表7.4 Boot.ini/NOEXECUTE开关

Boot.ini开关	DEP设置对话框中的选项	意 义
/NOEXECUTE=OPTIN	仅对必要的Windows程序和服务才打开DEP	对于核心的Windows系统映像打开DEP
/NOEXECUTE=OPTOUT	除了所选中的程序以外，对其他所有的程序和服务都打开DEP	除了这些指定的文件以外，针对其他所有的可执行文件都打开DEP
/NOEXECUTE=ALWAYSON	（没有对应的GUI界面来选择这一选项）	对于所有组件都打开，无法排除特定的应用程序。
/NOEXECUTE=ALWAYSOFF	（没有对应的GUI界面来选择这一选项）	禁止DEP（不推荐）

软件的数据执行保护

因为在目前情况下，大多数运行Windows的处理器并不支持硬件的“不可执行”保护，所以，Windows XP Service Pack 2和Windows Server 2003 Service Pack 1及以后的版本只支持有限的、软件的（非硬件的——译注）数据执行保护（DEP）。软件DEP的一个方面是，减少了对Windows中异常处理机制的发掘机会（关于结构化异常处理的描述，请参见第3章）。如果程序的映像文件是利用安全的结构化异常处理机制（在Microsoft Visual C++ 2003编译器中的一个新特性）联编得到的，那么，在分发一个异常以前，系统首先要验证，该异常的处理器（handler）已经在映像文件的函数表中被注册了。如果该程序的映像不是利用安全的结构化异常处理机制联编得到的，则软件DEP确保，在分发一个异常以前，异常处理器所在的内存区域已经被标记为可执行的。

写时复制

写时复制页面保护机制是一种优化，内存管理器利用它可以节约内存。当进程为一个包含读/写页面的内存区对象映射了一份写时复制视图，而并非在映射该视图时创建一份进程私有的拷贝（Hewlett Packard OpenVMS操作系统就是这样做的）时，内存管理器将页面拷贝的动作推迟到页面被写入数据的时候。所有现代的UNIX系统也都使用了这项技术。例如，如图7.3所示，两个进程正在共享三个页面，每个页面都被标记为写时复制，但是，这两个进程都不打算修改页面上的任何数据。

www.TopSage.com

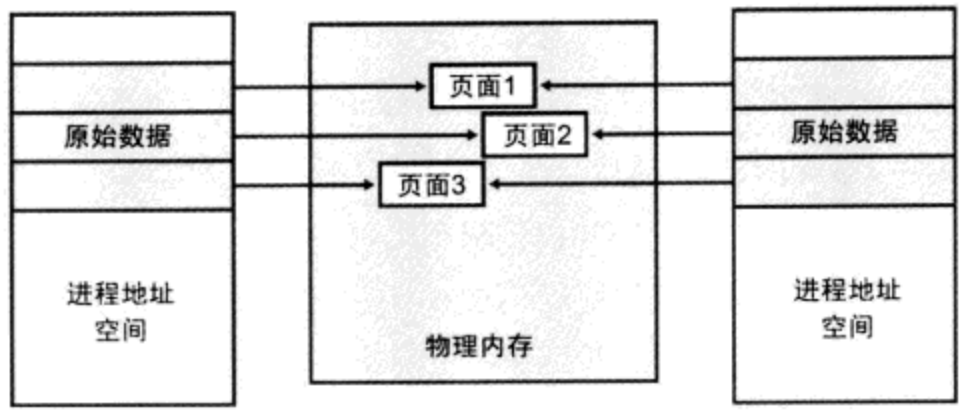


图7.3 写时复制“之前”的状态

如果这两个进程中任何一个线程对一个页面执行了写操作，则产生一个内存管理错误。内存管理器看到，此写操作作用在一个写时复制的页面上，所以，它不是将此错误报告为访问违例，而是在物理内存中分配一个新的读/写页面，并且将原始页面中的内容拷贝到新的页面中，同时也更新一下该进程中对应的页面映射信息（本章后面将会解释），使它指向新的页面位置，然后解除异常，从而使得刚才产生错误的那条指令得以重新执行。这一次，写操作成功了，但是如图7.4所示，新拷贝的页面现在对于执行写操作的那个进程来说是私有的，对于其他仍然在共享这一写时复制页面的进程来说，它是不可见的。每个往共享页面中写入数据的进程都将获得它自己的私有拷贝。

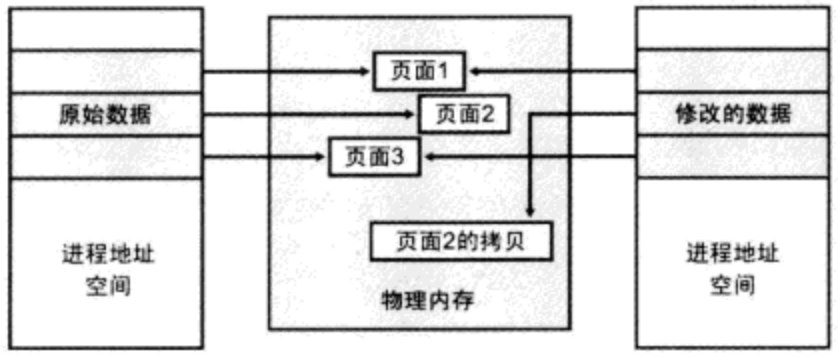


图7.4 写时复制“之后”的状态

写时复制的一个应用是，在调试器中实现断点支持。例如，在默认情况下，代码页面在起始时都是只能执行的。然而，如果一个程序员在调试一个程序时设置了一个断点，则调试器必须在代码中加入一条断点指令。它是这样做的：首先将该页面的保护模式改变成PAGE_EXECUTE_READWRITE，然后改变指令流。因为代码页面是所映射的内存区的一部分，所以，内存管理器为设置了断点的那个进程创建一份私有拷贝，同时其他进程仍然使用原先未经修改的代码页面。

写时复制是一项被称为“**延迟计算**（*lazy evaluation*）”的计算技术（evaluation technique）的一个例子，内存管理器广泛地使用了延迟计算的技术。延迟计算使得只有当绝对需要的时候才执行一个昂贵的操作——如果该操作从来也不需要的话，则它不会浪费任何一点时间。

www.TopSage.com

POSIX子系统利用写时复制来实现*fork*函数。在通常情况下，当一个UNIX应用程序调用*fork*函数来创建另一个进程时，新进程所做的第一件事情是调用*exec*函数，用一个可执行程序来重新初始化它的地址空间。在*fork*中，新进程不是拷贝整个地址空间，而是通过将页面标记为写时复制的方式，与父进程共享这些页面。如果子进程在这些页面中写入数据了，则生成一份进程私有的拷贝。如果没有写操作，则两个进程继续共享页面，不会执行拷贝动作。不管怎么样，内存管理器只拷贝一个进程试图要写入数据的那些页面，而不是整个地址空间。

要想知道写时复制错误的比率，可以检查性能计数器“Memory: Write Copies/Sec”。

堆管理器

许多应用程序可以利用页面粒度的函数，比如*VirtualAlloc*，来分配比最小分配粒度64KB小得多的内存块。无论是从内存使用效率的角度，还是从性能的角度来看，为相对较小的内存请求分配如此大的一个区域显然不是最优的。为了满足这种需求，Windows提供了一个被称为**堆管理器**（*heap manager*）的组件，它负责管理大内存区域中的内存分配，这些大内存区域是通过一些页面粒度的内存分配函数来保留的。堆管理器中的分配粒度相对比较小：在32位系统上是8字节，在64位系统上是16字节。堆管理器已经被精心设计成：在这些很小的内存分配的情况下进行了内存使用率和性能两个方面的优化。

堆管理器存在于两个地方：Ntdll.dll和Ntoskrnl.exe。子系统API（比如Windows堆API）调用Ntdll中的函数，而各种执行体组件和设备驱动程序调用Ntoskrnl中的函数。它的原生接口（以*Rtl*为前缀）只能用于内部Windows组件或者内核模式设备驱动程序。文档化的Windows API堆接口（以*Heap*为前缀）是一些瘦函数，它们调用了Ntdll.dll中的原生函数。而且，遗留的API（以*Local*或*Global*为前缀）仍然是可用的，以便支持老的Windows应用程序。最常用的Windows堆函数如下。

- *HeapCreate*或*HeapDestroy*——创建或删除一个堆。在创建的时候可以指定初始的保留大小和提交大小；
- *HeapAlloc*——分配一个堆内存块；
- *HeapFree*——释放一个原先由HeapAlloc分配的内存块；
- *HeapReAlloc*——改变一个已分配的内存块的大小（增长或缩减一个已有的内存块）；
- *HeapLock*或*HeapUnlock*——控制堆操作的互斥访问；
- *HeapWalk*——列举一个堆内部的内存项和区域。

www.TopSage.com

堆的类型

每个进程至少有一个堆：默认的进程堆。默认的进程堆是在进程启动的时候被创建的，而且在进程的生命期过程中永远不会被删除。它的默认大小为1MB，但是，利用/HEAP链接器标志，可以在一个映像文件中指定更大的起始大小。然而，此大小值只是初始的保留内存大小——根据需要它可以被自动扩充（你也可以在映像文件中指定初始的提交内存大小）。

默认的堆可以由程序显式地使用，也可以通过某些Windows内部函数隐式地使用。应用程序可以调用Windows函数*GetProcessHeap*来获得默认的进程堆。进程也可以利用*HeapCreate*函数来创建额外的私有堆。当一个进程不再需要一个私有堆的时候，它可以调用*HeapDestroy*来恢复虚拟地址空间。每个进程中维护了一个数组，其中包含了所有堆的信息，线程利用Windows函数*GetProcessHeaps*可以获得此数组的信息。

一个堆可以管理（通过*VirtualAlloc*由内存管理器保留的）大内存区中的分配情况，也可以管理该进程地址空间中内存映射文件对象中的内存分配情况。在实践中，后一种方法很少使用，但是，当内存块的内容需要在两个进程之间共享，或者在一个内核模式组件和一个用户模式组件之间共享时，它是非常合适的。如果一个堆建立在内存映射文件区域的基础之上，则该组件可以调用的堆函数有一些特定的约束。首先，由于内部堆结构使用了指针，所以它们不允许被重新定位到不同的地址上。其次，堆函数不支持跨多进程的同步，也不支持内核组件和用户进程之间的同步。而且，在用户模式和内核模式之间共享堆的情况下，用户模式的映射应该是只读的，以防止用户模式代码破坏堆的内部结构，注意，这种破坏将会导致系统崩溃。

堆管理器结构

如图7.5所示，堆管理器是一个两层结构：可选的前端堆层和核心堆层。核心堆层处理基本的功能，是最常见的跨用户模式和内核模式的堆实现。核心功能包括段内部的内存块的管理、段的管理、堆的扩展策略、提交内存和解除提交，以及大块的管理。

可选的前端堆这一层仅仅针对用户模式堆，它可以存在于已有的核心功能之上。有两种类型的前端层：预读列表（look-aside list）和低碎片堆（LFH，Low Fragmentation Heap，在Windows XP及以后的版本中才可以使用），本节后面讲述了这两种类型的前端层。对于一个堆，同一时刻只能使用一个前端层。

www.TopSage.com

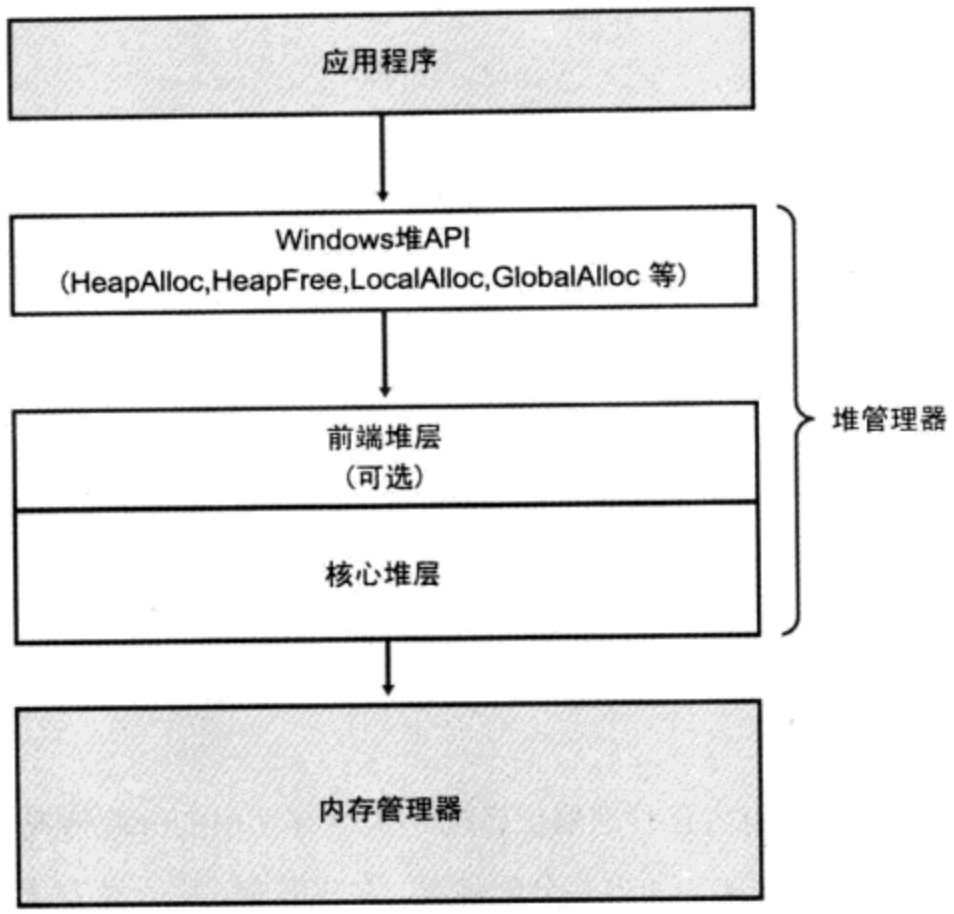


图7.5 堆管理器的层次

堆同步

在默认情况下，堆管理器支持多个线程的并发访问。然而，如果一个进程是单线程的，或者使用了外部的同步机制，那么，它可以在创建堆的时候或者在每次分配内存的时候，通过指定HEAP_NO_SERIALIZE来告知堆管理器，以避免同步的开销。

对于那些要求在多个堆调用之间保持一致性状态的操作，用户进程也可以锁住整个堆，从而阻止其他的线程执行堆操作。例如，如果多个线程可以并发地执行堆操作的话，那么，“利用Windows函数*HeapWalk*来列举一个堆内部的内存块”这一操作要求锁住这个堆。

如果堆同步被打开，则每个堆有一个锁，它用来保护所有的内部堆结构。在繁重的多线程应用程序中（尤其当运行在多处理器系统上的时候），堆锁可能会成为一个重要的竞争点。在这种情况下，允许前端堆可能会极大地提高性能，接下来讲述前端堆。

预读列表（Look-Aside List）

预读列表是单向链接的列表，它允许诸如“压入列表”或者“从列表中弹出”这样的元素操作，并且这些操作按照后进先出（LIFO）的顺序和非阻塞的方式来实现。对于Windows应用程序，通过Windows函数*InterlockedPopEntrySList*或*InterlockedPushEntrySList*，可以使用这些数据结构的一个简化版本。每个堆有128个预读列表，在32位平台上，这些预读列表可处理多

www.TopSage.com

达1KB的分配请求，在64位平台上，可处理多达2KB的分配请求。

相比普通的堆分配来说，预读列表提供了重要的性能改进：多个线程可以并发地执行分配和释放操作，而无需获得该堆的全局锁；而且，由于使用了LIFO顺序模型，以及每个堆操作只访问很少的内部数据文件，因而缓存的局部性得到了优化。

堆管理器维护了每一个预读列表中的内存块的数目，以及一些有助于独立地调节每个列表使用量的计数器。如果一个线程申请的内存块的大小在对应的预读列表中不存在，那么，堆管理器将把该调用传递给核心堆管理器，以完成这一操作。在这种情况下，堆管理器也将更新一个内部计数器，它记录了分配内存时漏掉的次数。后面介绍调节决策时将要用到此计数器。

当一个堆被创建的时候，如果没有打开调试选项，并且该堆是可扩展的，则堆管理器自动创建这些预读列表。在堆管理器使用了预读列表以后，有些应用程序可能存在兼容性问题。在这种情况下，如果在应用程序的映像文件执行选项中指定DisableHeapLookaside标志，则这些遗留的应用程序就可以正确地运行（利用Windows 2000 Server资源工具箱补充1中的Imagecfg.exe工具，可以指定映像文件的执行选项）。

低碎片堆

在Windows中运行的许多应用程序有相对较小的堆内存使用量（通常小于1兆）。对于这一类应用程序，堆管理器的最佳匹配策略（best-fit policy）有助于为每个进程维持一个小的内存印迹。然而，这种策略对于大的进程和多处理器机器来说并不具有很好的扩展性。在这些情况下，由于堆碎片的原因，堆可以使用的内存可能会减少。在有些情况下，被安排在不同处理器上运行的多个线程常常需要并发地使用特定大小的数据块，此时就要遭受性能上的退化。这是因为，几个处理器需要同时修改同一个内存位置（比如，特定大小的预读列表的头），因此使其他处理器的对应缓存线变得无效。

低碎片堆（LFH，Low Fragmentation Heap）使用核心堆管理器和预读列表来处理这些问题。对于预读列表，只要其他的堆设置不冲突，它就被默认用作前端堆；而与此不同的是，只有在应用程序调用了*HeapSetInformation*以后LFH才被打开。对于大的堆，相当百分比的内存分配往往被组织到相对少量的各种特定大小的桶中。LFH使用的分配策略是，通过有效地处理同样大小的内存块来优化这些模式的使用量。

为了解决扩展性的问题，LFH对于频繁访问的内部结构进行了扩充，使其变成许多槽，每个槽大于当前机器上进程数量的两倍。从线程到这些槽的分配安排，是由一个被称为**亲和性管理器**（*affinity manager*）的LFH组件来完成的。最初时候，LFH为堆内存的分配请求使用第一个槽；然而，如果检测到在访问某一个内部数据时有一个竞争，则LFH将当前线程切换为使用另一个不同的槽。随着竞争的进一步发生，越来越多的线程被散布到更多的槽上。对这些槽的控制，是

www.TopSage.com

针对每一种大小的桶来进行的，这样也可以提高局部性，并且使总体的内存消耗变得最小化。

堆的调试特性

堆管理器包含了一些特性，以帮助检测代码错误，以下是一些可以使用的功能。

- **允许尾部检查**（tail checking） 每个内存块的末尾包含一个签名，当内存块被释放的时候，可以检查此签名。如果缓冲区溢出破坏了整个签名或者部分签名，则相应的堆将会报告此错误；
- **允许空闲检查**（free checking） 空闲的块被填充成特定的模式，因而当堆管理器需要访问此空闲块的时候（比如从空闲列表上去除该内存块以便分配出去），它可以在各个点上对内存块进行检查。如果在内存块被释放以后，进程继续写这个内存块，则堆管理器将会检测到模式中的变化，从而报告一个错误；
- **参数检查** 这一功能是由对堆函数的参数的各种检查构成的；
- **堆的有效性检查** 在每个堆调用中，都要对整个堆进行检查；
- **堆标记和栈痕迹支持** 此项功能可以为每个内存分配指定标记，以及/或者捕捉住堆调用的用户模式栈痕迹，以帮助诊断出堆错误的可能原因。

如果一个进程是在调试器的控制下启动起来的，则前三项是默认打开的（调试器可以改变这种行为，关闭这些特性）。通过gflags工具来设置一个可执行映像头部的各种调试标记，就可以为该映像指定堆调试特性（参见第3章中的“Windows全局标记”一节）。或者，也可以在标准的Windows调试器中使用*!heap*命令来打开堆调试选项。（更多的信息请参见调试器帮助文档。）

打开堆调试器选项，会影响到进程中的所有堆。而且，如果任何一个堆调试器选项被打开，则前端堆被自动禁止，改而使用核心堆（所请求的调试选项也相应打开）。 前端堆不会被用于那些不可扩充的堆（因为要在已有的堆结构上加入额外的负载），或者那些不允许序列化的堆。

Pageheap

上一部分中讲述的尾部检查和空闲检查选项能够发现一些内存破坏，但真正检测到问题却有可能是在内存破坏很久以后了，所以，另外一个称为*pageheap*的堆调试工具可以被用来将所有或者部分堆调用转到另一个不同的堆管理器上。Pageheap是Windows Application Compatibility Toolket（Windows应用兼容性工具箱）的一部分，你可以从*www.microsoft.com*下载。pageheap在页面的的末尾加上了一些内存，因而若发生缓冲区溢出的话，它可以引发一个访问违例，从而使得更加容易地检测到错误代码。

www.TopSage.com

作为一种可选的功能，Pageheap也允许在页面的开始处加上额外的内存块，以便检测到缓冲区绕回头的问题（不过这种情形很少发生）。pageheap也可以保护空闲的页面，如果在堆内存块被释放以后，任何企图引用其中的页面都可以被检测到。

注意，使用pageheap可能会导致用完地址空间，因为它在小内存分配单元上引入了相当大的内存开销。而且，由于对零页面的需求相应地增加、局部性丢失了，以及频繁地调用函数来验证堆数据结构从而带来的额外开销，所以，性能上会遭受影响。只有进程可以指定仅仅针对特定大小的内存块，或特定地址范围内的内存块，以及/或某些DLL的内存请求，才使用pageheap，从而减小这种性能影响。



> **注** 有关pageheap的更多信息，请参考Microsoft知识库（*http://support.microsoft.com*）中的286470文章。

地址窗口扩展

虽然Windows的32位版本可以支持多达128GB物理内存，如表2.4中所示，但是，每个32位用户进程在默认情况下只有2GB虚拟地址空间（当在Boot.ini中使用/3GB和/USERVA开关时，此限制值可以被配置到多达3GB，后面的“x86用户地址空间的布局结构”一节中有进一步的描述）。为了允许一个32位进程分配和访问更多的物理内存，突破这一受限地址空间所能表达的内存范围，Windows提供了一组函数，它们被称为地址窗口扩展（AWE，*Address Windowing Extensions*）。例如，在一个具有8GB物理内存的Windows 2000 Advanced Server系统上，一个数据库服务器应用程序有可能使用AWE来分配和使用6GB内存作为数据库缓存。

通过AWE函数来分配和使用内存，可使用以下三个步骤。

1. 分配所要使用的物理内存。
2. 创建一个虚拟地址空间区域，把它当作一个窗口来映射物理内存的多个视图。
3. 将物理内存的视图映射到窗口中。

一个应用程序想要分配物理内存，可以调用Windows函数*AllocateUserPhysicalPages*（这一函数要求“Lock Pages In Memory[将页面锁在内存中]”用户权限）。然后，应用程序使用Windows的*VirtualAlloc*函数，以及MEM_PHYSICAL标志，在进程地址空间的私有部分创建一个窗口，该窗口被映射至原先分配的物理内存中的全部或者一部分。之后，AWE分配的内存几乎可以与所有的Windows API一起使用（例如，Microsoft DirectX函数不能使用AWE内存）。

如果一个应用程序在它的地址空间中创建一个256MB的窗口，并且分配4GB的物理内存（该系统的物理内存超过4GB），那么，此应用程序可以使用Windows的*MapUserPhysicalPages*或*MapUserPhysicalPagesScatter*函数，将4GB物理内存映射至256MB窗口中，从而访问此4GB物理内存中的任何部分。

www.TopSage.com

应用程序的虚拟地址空间的大小决定了该应用程序通过一次映射可以访问的物理内存的数量。图7.6显示了在一个服务器应用程序地址空间中的一个AWE窗口，它被映射至原先通过*AllocateUserPhysicalPages*分配的物理内存的一部分。

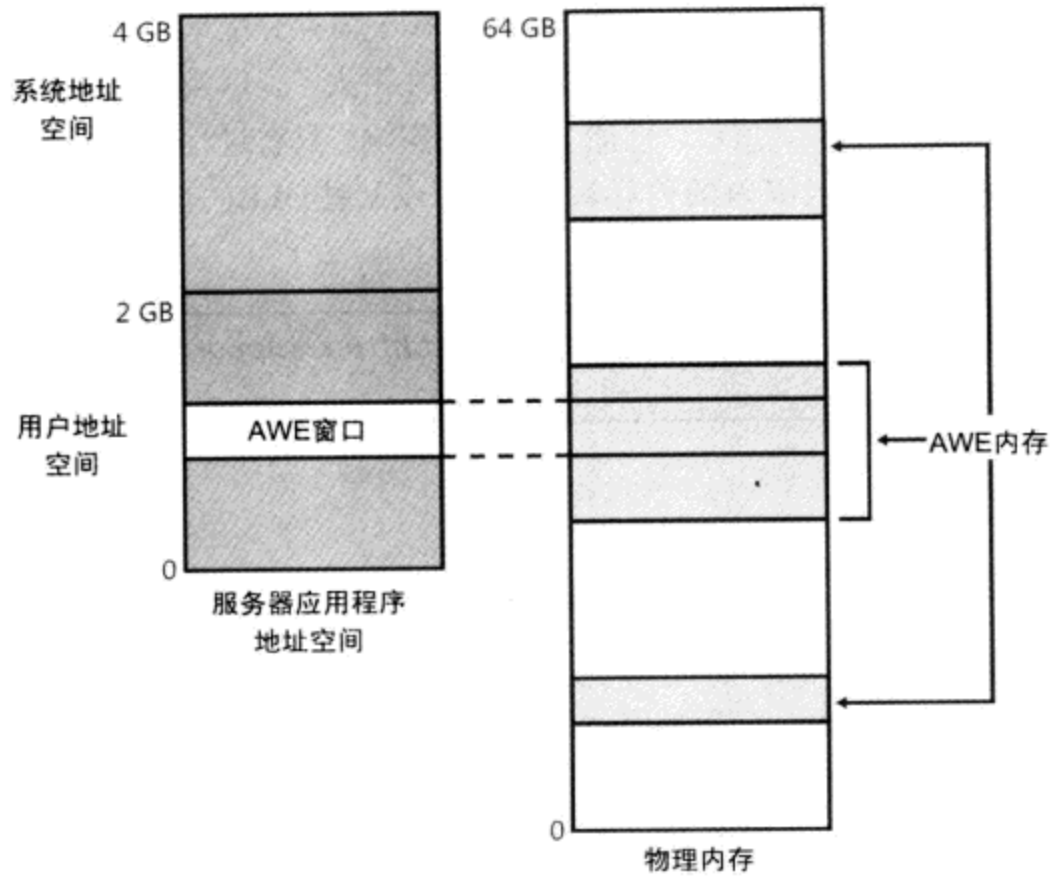


图7.6 使用AWE来映射物理内存

Windows的所有版本都有AWE功能，而且，不管一个系统有多少物理内存，AWE的功能都是可以使用的。然而，在超过2GB物理内存的系统上，AWE才是最有用的，因为对于一个32位进程来说，这是直接使用超过2GB内存的惟一途径。另一种用途是出于安全性的目的：因为AWE内存永远不会被换出，所以AWE内存中的数据永远不会在页面文件中有拷贝，从而可以避免有人将系统重新引导到另一个操作系统中，通过检查页面文件来获取数据。

最后，对于通过AWE函数来分配和映射的内存，有以下一些限制：

- 进程之间不能共享页面；
- 同样的物理页面不能被映射至同一进程中的多个虚拟地址；
- 在老版本的Windows中，对页面的保护仅限于读/写；在Windows Server 2003 Service Pack 1及以后的版本中，也支持不可访问和只读模式。

关于在超过4GB物理内存的系统上，用于映射内存的页表数据结构的描述信息，请参见“物理地址扩展（PAE）”一节。

www.TopSage.com

7.3 系统内存池

在系统初始化时，内存管理器创建了两种类型的动态大小的内存池，内核模式的组件从这两种内存池中分配系统内存。

- **非换页池** 它是由一些“可保证总是驻留在物理内存中”的虚拟地址范围构成的，由于这些地址范围总是驻留在内存中，因此任何时候（从任何IRQL级别，以及从任何进程环境中）都可以访问它们，而不会招致页面错误。之所以需要非换页池，其中一个原因是第2章中讲述的规则：在DPC/Dispatch级别或更高级别上，页面错误不可能被满足；
- **换页池** 系统空间中的一段虚拟内存区域，它可以被换入和换出系统。凡是不需要从DPC/Dispatch级别或更高级别上访问内存的设备驱动程序都可以使用换页池。在任何进程环境中它都是可以访问的。

这两种内存池都位于系统地址空间部分，并且被映射到每个进程的虚拟地址空间中（在表7.8中，你将会看到，在系统内存中，它们是从哪里开始的）。执行体提供了相应的例程以便从这两种池中分配和释放内存。有关这些例程的更多信息，请参见Windows DDK文档中以*ExAllocatePool*开头的函数。

单处理器系统有3个换页池；多处理器系统有5个。拥有多个换页池，可降低系统代码在并发调用池例程时的阻塞频率。非换页池和换页池都根据系统中物理内存的数量，从一个初始的大小开始，然后在有必要的时候再增大，一直达到一个在系统引导时已计算好的最大值。



> **注** Windows的将来版本可能会支持动态的池大小，这意味着将不再需要最大值。因此，应用程序和设备驱动程序不应该假设：对于每个系统而言，池大小的最大值是一个固定的值。

配置内存池的大小

通过将注册表键HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management中的值NonPagedPoolSize和PagedPoolSize从0（由系统来计算大小）改成期望的初始大小值（按字节为单位），你就可以改变这些内存池的初始大小。然而，你给出的值不能超过表7.5中列出的最大的池大小值。PagedPoolSize值为0xFFFFFFFF意味着，选择最大可能的大小值，这也意味着，以系统页表项（PTE）为代价，得到一个更大的换页池。

表7.5 最大的池大小值

池的类型	在32位系统上的最大值	在64位系统上的最大值
非换页池	256 MB （若以/3GB引导则为128 MB）	128 GB
换页池	491.875 MB （Windows 2000和Windows XP）；650 MB （Windows Server 2003）	128 GB

www.TopSage.com

计算得到的大小值被存放在4个内核变量中，其中三个以性能计数器的方式被暴露出来。表7.6列出了这些变量和计数器，以及能够改变大小值的两个注册表键。

表7.6 与系统池大小有关的变量和性能计数器

内核变量	性能计数器	可改写的注册表键	说 明
MmSizeOfNonPaged-PoolInBytes	Memory: Pool Nonpaged Bytes	不适用	非换页池的当前大小
MmMaximumNonPa-gedPoolInBytes	不可用	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\NonPagedPoolSize	非换页池的最大大小
不可用	Memory: Pool Paged Bytes	不适用	换页池的当前虚拟大小
MmPagedPoolPage（页面数量）	Memory: Pool Paged Resident Bytes	不适用	换页池的当前物理（驻留）大小
MmSizeOfPagedPoo-lInBytes	不可用	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagedPoolSize	换页池的最大（虚拟）大小

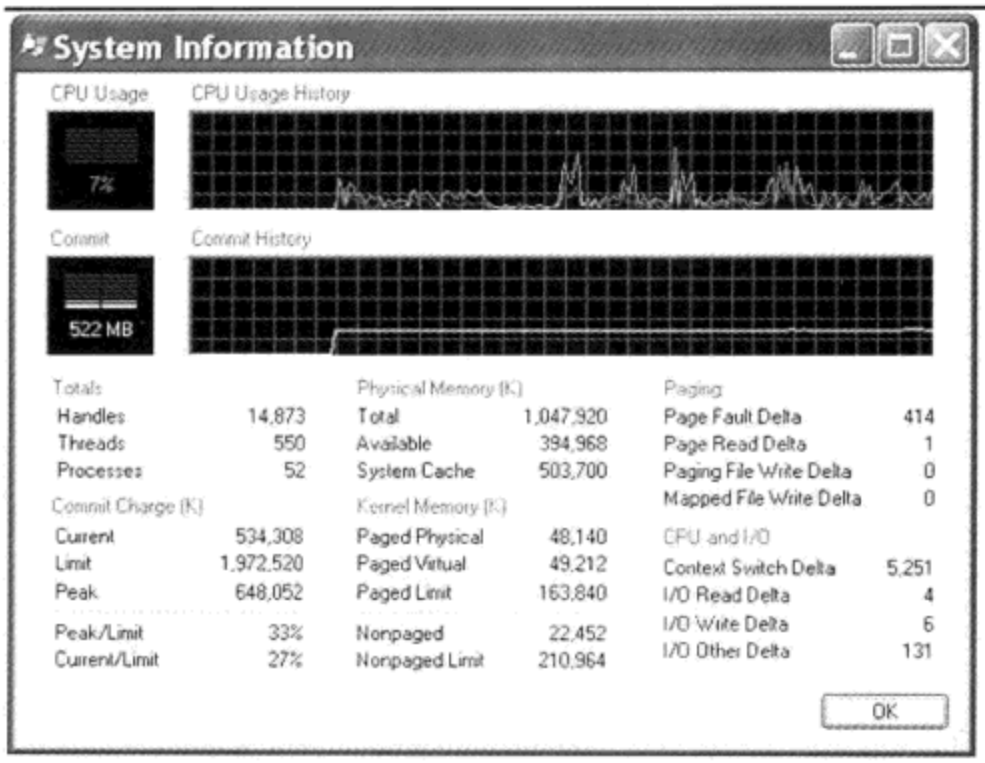


实验：确定最大的池大小值

因为换页池和非换页池代表了一种关键的系统资源，所以，知道你何时已经接近了当前系统最大的池大小值是非常重要的，这样你可以决定是否需要通过适当的注册表值来改变默认的最大值。然而，与池大小有关的性能计数器只报告了当前的大小情况，而不是最大的大小值。所以，你在真正用光池内存以前，并不知道你什么时候正在接近最大极限值（正如前面所说明的，将来的Windows版本可能支持动态大小的池，所以，检查内存池最大值的需求在将来可能是不必要的）。

利用进程管理器或者实时内核调试器（见第1章的说明），你可以获得内存池的最大值。为了通过进程管理器来查看内存池的最大值，请点击View、System Information。内存池的最大值被显示在Kernel Memory部分，如下图所示：

www.TopSage.com



注意，进程管理器为了获得这些信息，它必须访问当前系统上正在运行的内核的符号信息（关于如何配置进程管理器使它可以使用符号信息的详细描述，请参见第1章中的实验“利用进程管理器查看进程的细节”）。

要想使用内核调试器来查看同样的信息，你可以使用*!vm*命令，如下所示：

```
lkd> !vm

*** Virtual Memory Usage ***
Physical Memory:    261980   ( 1047920 Kb)
Page File: \??\C:\pagefile.sys
   Current:   1024000Kb Free Space:    862236Kb
   Minimum:   1024000Kb Maximum:      1024000Kb
Available Pages:     96077   (  384308 Kb)
ResAvail Pages:     194722   (  778888 Kb)
Locked IO Pages:        89   (     356 Kb)
Free System PTEs:   175933   (  703732 Kb)
Free NP PTEs:        31832   (  127328 Kb)
Free Special NP:         0   (       0 Kb)
Modified Pages:        529   (    2116 Kb)
Modified PF Pages:     529   (    2116 Kb)
NonPagedPool Usage:   5617   (   22468 Kb)
NonPagedPool Max:    52741   (  210964 Kb)
PagedPool 0 Usage:    6582   (   26328 Kb)
PagedPool 1 Usage:    3009   (   12036 Kb)
PagedPool 2 Usage:    3044   (   12176 Kb)
PagedPool Usage:     12635   (   50540 Kb)
PagedPool Maximum:   40960   (  163840 Kb)
```


www.TopSage.com

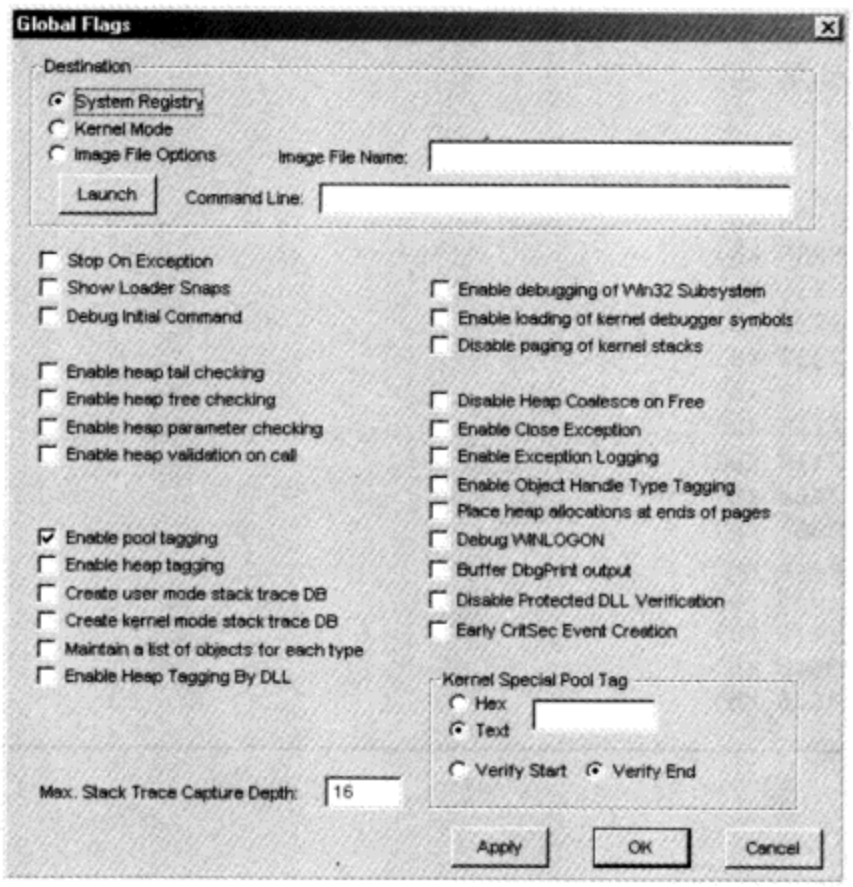
在这个系统上，非换页池和换页池离它们的最大值还差得远呢。你也可以检查表7.6中列出的内核变量的值：

```
kd> dd mmmaximumnonpagedpoolinbytes l1
8047f620  0328c000
kd> ? 328c000
Evaluate expression: 53002240 = 0328c000
kd> dd mmsizeofpagedpoolinbytes l1
80470a98  06800000
kd> ? 6800000
Evaluate expression: 109051904 = 06800000
```

从这个例子中，你可以看到，非换页池的最大值是53 002 240字节（差不多50MB），换页池的最大值是109 051 904字节（104MB）。在这个例子所用的测试系统上，非换页池的当前内存使用量是5.5MB，换页池的当前内存使用量是34MB，所以，这两个内存池都还远远未满。

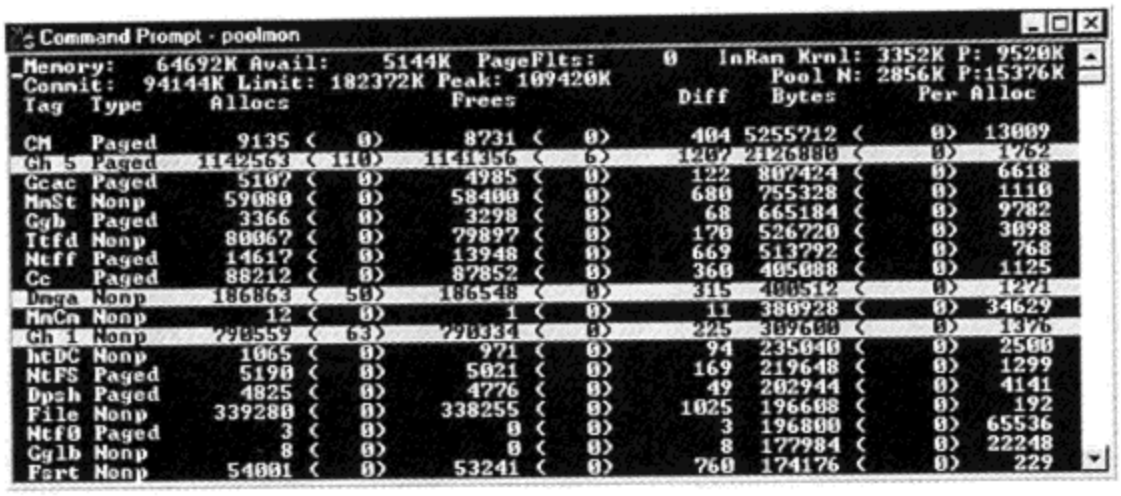
监视内存池的使用

针对非换页池和换页池的大小值（虚拟的和物理的），Memory性能计数器对象分别使用了不同的计数器。而且，Poolman工具（位于Windows支持工具箱中）允许你监视非换页池和换页池的详细使用情况。为了做到这一点，你必须打开内部的“Enable Pool Tagging（允许池标签）”选项。从Windows Server 2003以后，以及在所有的Windows检查版本（Checked Build）中，池标签是默认打开的（实际上，在Windows Server 2003以后的版本中，池标签是无法关闭的）。为了打开池标签，可以运行Windows支持工具箱、Platform SDK或DDK中的Gflags工具，选择“Enable Pool Tagging”，如下图所示：



www.TopSage.com

然后，点击Apply按钮，并重新引导系统。在系统重新引导起来以后，运行Poolmon；你应该可以看到类似于以下的界面：



这里加亮显示的行代表了显示过程中的变化（你可以禁止加亮显示的特性，只要在Poolmon运行时键入*l*即可。再键入*l*可重新打开此特性）。在Poolmon运行时键入?可打开它的帮助屏幕。你也可以配置你想要监视哪些内存池（换页的、非换页的，或者两者均监视），以及什么排列顺序。而且，命令行选项也显示出来了，这使得你可以监视特定的数据结构（或者除了一种结构类型以外的所有其他数据）。例如，命令poolmon –iCM将只监视CM类型的结构（CM类型对应于配置管理器，它管理注册表）。这里显示的列的含义如表7.7所示。

表7.7 Poolmon中的列

列	解 释
Tag	针对池内存分配的4字节标签
Type	内存池类型（换页池或非换页池）
Allocs	分配的总计数（括号中的数显示了自从上一次更新以来Alloc列的差值）
Frees	释放的总计数（括号中的数显示了自从上一次更新以来Frees列的差值）
Diff	Allocs减去Frees的值
Bytes	此数据结构类型所消耗的总字节数（括号中的数显示了自从上一次更新以来Bytes列的差值）
Per Alloc	此数据结构类型单个实例的大小（按字节）

在这个例子中，CM结构（配置管理器，或注册表）占据了换页池中的最大部分空间，MmSt结构（一个用于映射文件的、与内存管理相关的数据结构）占据了非换页池中的最大部分空间。

关于Windows所使用的池标签的含义说明，请参考文件\Program Files\Debugging Tools for Windows\Triage\Pooltag.txt（该文件作为Windows调试工具箱[Debugging Tools]的一部分被安装到系统中，第1章介绍了Windows调试工具箱）。因为第三方设备驱动程序的池标签未列在该文件中，所以，你可以在Windows Server 2003 DDK随带的Poolmon版本中，利用“-c”开关生成一个本地池标签文件（Localtag.txt）。

www.TopSage.com

该文件将包含了在你本地系统上找到的驱动程序所使用的池标签，包括第三方的驱动程序（注意，如果一个设备驱动程序被加载以后其二进制文件被删除了，则它的池标签将无法被识别出来）。

或者，另外一种办法是，你可以利用*www.sysinternals.com*提供的String.exe工具，在你系统上的设备驱动程序中搜索池标签。例如，命令

```
strings \windows\system32\drivers\*.sys > findstr /i "abcd"
```

将显示包含字符串“abcd”的驱动程序。注意，设备驱动程序不必一定要位于\Windows\System32\Drivers目录下，它们可以在任何文件夹中。为了列出所有已加载的驱动程序的全路径，你可以点击Start，再点击Run，并输入Msinfo32。然后，点击Software Environment和System Drivers。正如前面已指出的那样，如果一个设备驱动程序已经被加载，然后从系统中删除了，则这里不会把它也列出来。

另一种查看设备驱动程序的池内存使用量的方法是，打开驱动程序检验器（Driver Verifier）的池跟踪特性。本章后面将介绍驱动程序检验器。虽然这使得从池标签到设备驱动程序的映射变得不再有必要，但它要求重新引导（以便允许驱动程序检验器能够作用在期望的驱动程序上）。在打开了池跟踪特性并重新引导以后，你或者可以运行驱动程序检验器的图形化管理器（\Windows\System32\Verifier.exe），或者使用检验器的/Log命令，将内存池的使用信息发送到一个文件中。

最后，如果你正在查看一个崩溃转储文件，那么你可以利用内核调试器的*!poolused*命令来查看内存池的使用量。命令*!poolused 2*显示了非换页池的使用情况，它按照使用池内存最多的结构标签来排序。命令*!poolused 4*列出换页池的使用情况，也按照使用池内存最多的结构标签来排序。下面的例子显示了这两个命令的部分输出结果：

```
lkd> !poolused 2
  Sorting by  NonPaged Pool Consumed
 Pool Used:
           NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 MmCm      386  2237440         0        0    Calls made to MmAllocateContiguousMemory ,
Binary: nt!mm
 File    12544  2121232         0        0    File objects
 Devi      385  1487184         0        0    Device objects
 Ntfr    19163  1226888         0        0    ERESOURCE , Binary: ntfs.sys
kd> !poolused 4
   Sorting by Paged Pool Consumed

  Pool Used:
           NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 CM         11      704      5146 22311904
 Gh 5        0        0       727  2523648
 MmSt        0        0       968  1975872
 Ntff       10     2240       790   682560
 Glal        1      128       383   600544
 Ttfd        0        0       265   545440
```


www.TopSage.com

实验：诊断内存池泄漏

在这个实验中，你将要修正你的系统上一个实际的换页池泄漏问题，这样，你可以利用前面介绍的技术来跟踪此泄漏问题。此泄漏是由NotMyFault工具产生的，你从*www.sysinternals.com/windowsinternals.shtml*可以下载此工具（注意，在Sysinternals主工具列表中并没有列出此工具）。当你运行NotMyFault.exe的时候，它加载一个设备驱动程序Myfault.sys，并显示如下的对话框：



1. 点击Leak Pool按钮。这使得NotMyFault开始向Myfault设备驱动程序发送请求，以分配换页池内存（不要点击Do Bug按钮，否则你将会经历一次系统崩溃；该按钮是在第14章中用来证明各种不同类型的系统崩溃的）。NotMyFault将继续发送请求，直到你点击Stop Leaking按钮为止。注意，即使在你关闭了该程序以后，换页池的内存也不会被释放；换页池被永久地泄漏了，直到你重新引导系统为止。然而，因为你只是在一小段时间泄漏了换页池，所以，这不应该使你的系统发生任何问题。
2. 在换页池被泄漏的同时，首先打开任务管理器，并点击Performance视图。你应该注意到Paged Pool爬上去了。你也可以利用进程管理器的System Information窗口检查这一点（先点击Show，然后点击System Information）。
3. 为了确定正在泄漏的池标签，运行Poolmon，按下“b”键，以便按字节数来排序。按下“p”两次，以便Poolmon只显示换页池。你应该可以注意到，池标签“Leak”爬上了列表的顶部（通过加亮显示那些发生变化的行，Poolmon可以显示池内存分配的变化）。

www.TopSage.com

4. 现在按下“Stop Leaking”按钮，从而不再消耗系统上的换页池内存。
5. 使用前面介绍的技术，运行Strings（从*www.sysinternals.com*下载），以便找到包含了池标签“Leak”的驱动程序二进制文件：

```
Strings \windows\system32\drivers\*.sys  |  findstr Leak
```

这应该会显示，文件Myfault.sys是一个匹配的驱动程序，因而可确认它是使用“Leak”池标签的驱动程序。

预读列表（Look-Aside List）

Windows也提供了一种快速的内存分配机制，称为**预读列表**（*look-aside list*）。内存池和预读列表之间的基本差别是，一般的内存池分配操作可以是不同大小的，而预读列表只包含了固定大小的内存块。虽然从所提供功能的角度来看，一般的内存池更为灵活，但预读列表的速度更快，因为它们并没有使用任何自旋锁，并且系统不必搜寻空闲内存块来满足可变大小的内存请求。

执行体组件和设备驱动程序可以利用*ExInitializeNPagedLookasideList*和*ExInitializePagedLookasideList*函数（在DDK中有文档），针对经常要分配的数据结构的大小，创建适当大小的预读列表。为了使多处理器同步的开销降低到最小，几个执行体子系统（比如I/O管理器、缓存管理器，以及对象管理器）针对它们经常访问的数据结构，为每个处理器都创建了预读列表。执行体也针对小数据的内存分配（256字节或更少），为每个处理器创建了一个通用的换页的预读列表和非换页的预读列表。

如果一个预读列表是空的（当它刚刚被创建的时候就是这样的），则系统必须从换页的或者非换页的内存池中分配内存。但是，如果它包含一个已释放的结构，则内存申请可以很快满足（随着数据结构被归还回来，此列表也会增长）。负责内存池分配的例程根据一个设备驱动程序或执行体组件从预读列表中申请内存的频繁程度，自动调整该列表中存储的已释放的缓冲区的数量：申请得越是频繁，列表中存储的缓冲区越多。如果预读列表没有收到内存申请，则它们自动缩小尺寸（当平衡集管理器系统线程醒过来并调用*KiAdjustLookasideDepth*函数的时候，这一检查过程就会每秒钟执行一次）。



实验：查看系统的预读列表

你可以通过内核调试器*!lookaside*命令，来显示各种系统预读列表的内容和大小。下面摘录了此命令的一部分输出：

www.TopSage.com

```
kd> !lookaside

Lookaside "nt!IopSmallIrpLookasideList" @ 804758a0 "Irps"
    Type      =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      148   Max Alloc  =      592
    AllocateMisses =       32   FreeMisses =        9
    TotalAllocates =       52   TotalFrees =       32
    Hit Rate       =       38%  Hit Rate   =       71%
Lookaside "nt!IopLargeIrpLookasideList" @ 804756a0 "Irpl"
    Type      =     0000 NonPagedPool
    Current Depth  =        4   Max Depth  =        4
    Size           =      436   Max Alloc  =     1744
    AllocateMisses =     2623   FreeMisses =     2443
    TotalAllocates =     7039   TotalFrees =     6863
    Hit Rate       =       62%  Hit Rate   =       64%
"
Lookaside "nt!IopMdlLookasideList" @ 80475740 "Mdl "
    Type      =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      120   Max Alloc  =      480
    AllocateMisses =     7017   FreeMisses =     1824
    TotalAllocates =    10901   TotalFrees =     5711
    Hit Rate       =       35%  Hit Rate   =       68%
§
Total NonPaged currently allocated for above lists = 4200
Total NonPaged potential for above lists           = 6144
Total Paged currently allocated for above lists    = 5136
Total Paged potential for above lists              = 12032
```

驱动程序检验器（Driver Verifier）

驱动程序检验器是一种专门用来查找和隔离驱动程序或其他内核模式系统代码中常见错误的机制。Microsoft使用驱动程序检验器来检查它自己的设备驱动程序，以及厂商们提交的进行硬件兼容性列表（HCL，Hardware Compatibility List）测试的所有设备驱动程序。这样做可以确保HCL上的驱动程序都与Windows兼容，并且不会再有常见的驱动程序错误（还有一个对应的应用程序检验器[Application Verifier]工具，不过本书中没有介绍，它可以帮助提高Windows上用户模式代码的质量）。

驱动程序检验器包含了对于以下几个系统组件的支持：内存管理器、I/O管理器，以及HAL，它们都提供了可打开的驱动程序检验选项。这一小节将介绍驱动程序检验器提供的与内存管理相关的检验选项。与设备驱动程序相关的选项在第9章中再介绍。

驱动程序检验器配置和初始化

为了配置驱动程序检验器以及查看有关其操作的统计信息，你可以运行驱动程序检验器管理器（\Windows\System32\Verifier.exe）（图7.7显示了Windows 2000中的版本）。它显示了几个视图页面。利用Setting页面，你可以指定自己想要检验哪些设备驱动程序（有一个选项可检验所有的驱动程序），以及想要执行哪些类型的检验。

www.TopSage.com

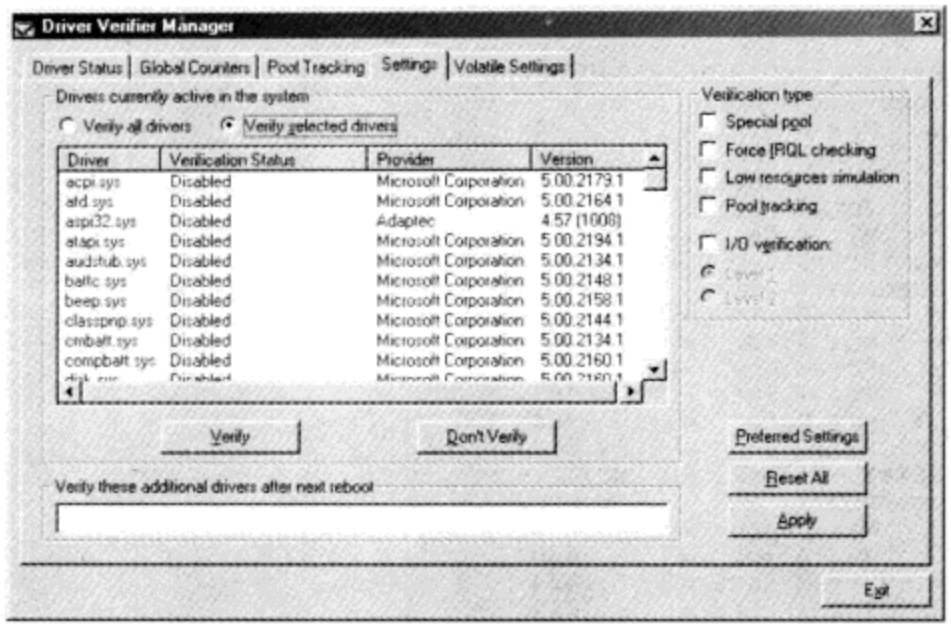


图7.7 Windows 2000中的驱动程序检验器管理器

在Windows XP和Windows Server 2003上，Verifier.exe有一个向导风格的界面，如图7.8所示。

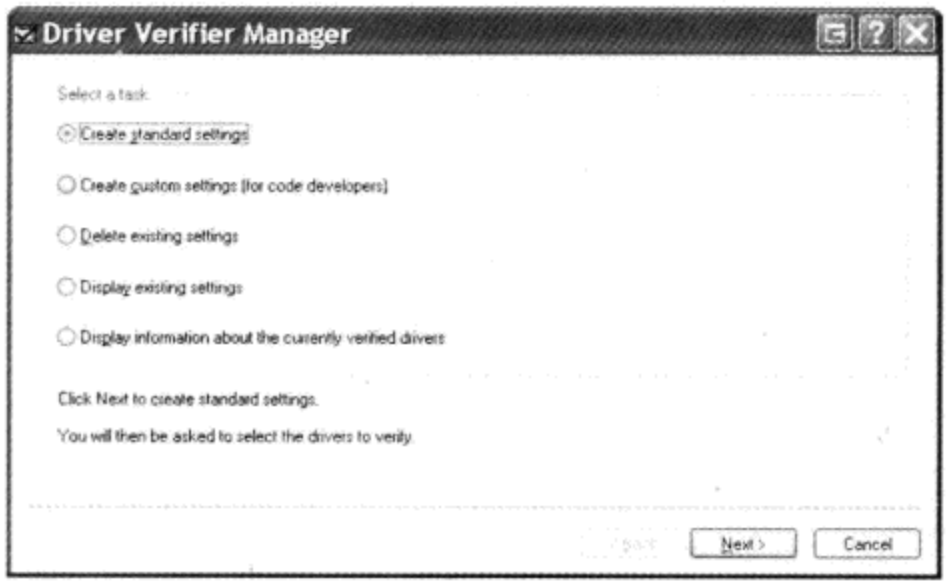


图7.8 Windows XP和Windows Server 2003中的驱动程序检验器管理器

利用驱动程序检验器的命令行界面，你也可以打开或者禁止它的选项。输入“**verifier /?**”可以看到这些开关。

驱动程序检验器的这些设置信息被保存在HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management下面。值VerifyDriverLevel包含的是一个位掩码，代表已允许的检验类型。VerifyDrivers值包含了要检验的驱动程序的名称（你在驱动程序检验器管理器中选中要检验的驱动程序以前，这些值都不会存在）。如果你选择要检验所有的驱动程序，VerifyDrivers被设置成一个星号字符（*）。根据你所做的设置，你可能需要重新引导系统，以便使你的检验选择可以起作用。

www.TopSage.com

在引导过程的早期，内存管理器读入驱动程序检验器的这些注册表值，以确定该检验哪些驱动程序，以及驱动程序检验器的哪些选项已被打开（注意，如果你在安全模式下引导，则所有的驱动程序检验器设置都被忽略）。然后，如果你选择了至少一个驱动程序要做检验，则内核检查它已加载到内存中的每一个设备驱动程序的名称，以及你选中来做检验的驱动程序列表。对于每一个同时出现在两处的设备驱动程序，内核调用*MiApplyDriverVerifier*函数，该函数所做的工作是，将该驱动程序中对于一组内核函数的引用，替换成另一组等价的驱动程序检验器版本。例如，*ExAllocatePool*被替换成对于*VerifierAllocatePool*的调用。窗口系统驱动程序也做类似的改变，以便它也使用驱动程序检验器等价版本的函数。

我们在看过了如何安装驱动程序检验器以后，现在来看4个可被应用于设备驱动程序的、与内存相关的检验选项：特殊内存池（Special Pool）、内存池跟踪、强制IRQL检查，以及低资源模拟。

特殊内存池（special pool） 特殊内存池选项使得负责内存池分配的例程用一个无效的页面将所分配的内存括起来，因而一旦引用到该内存块之前或者之后，都将导致一个内核模式访问违例，从而使系统崩溃，并且出错点指在有错误的驱动程序上。同时，特殊内存池选项也使得可以在驱动程序分配或者释放内存的时候执行一些额外的有效性检查。

当打开了特殊内存池选项时，负责内存池分配的例程申请一块内核内存区域，供驱动程序检验器使用。驱动程序检验器接到处于检验模式下的驱动程序所发出的内存分配请求时，将这些请求重定向到特殊内存池区域，而不是重定向到标准的内核模式内存池。当一个设备驱动程序从特殊内存池中分配内存时，驱动程序检验器将所分配的内存块向上舍入到整数页面边界。因为驱动程序检验器利用无效页面将所分配的页面包围起来，所以，如果一个设备驱动程序企图读或者写缓冲区末尾之后的数据，则驱动程序将访问一个无效页面，内存管理器将引发一个内核模式访问违例。

图7.9显示了一个特殊内存池缓冲区的例子，驱动程序检验器将该特殊内存池缓冲区分配给一个设备驱动程序。在这个例子中，驱动程序检验器可以检查缓冲区溢出错误。

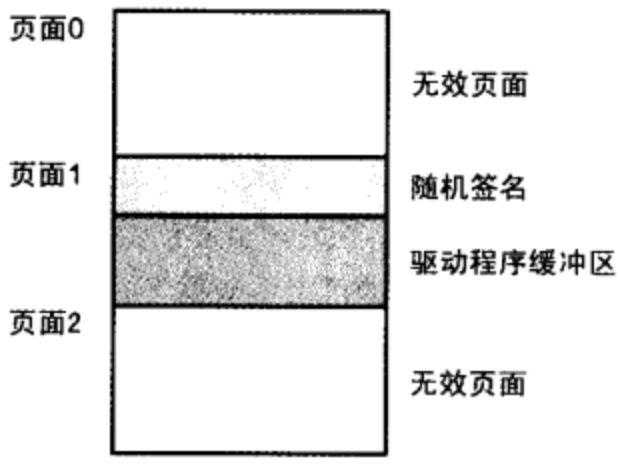


图7.9 特殊内存池的内存块的布局结构

www.TopSage.com

在默认情况下，驱动程序检验器执行缓冲区溢出（*overrun*）检测。它的做法是，把设备驱动程序使用的缓冲区放在所分配页面的末尾，并且在页面的开头填充一种随机模式。虽然驱动程序检验器管理器没有让你指定向下溢出（underrun）的检测选项，但是，你可以手工设置这种类型的检测，只需加入一个DWORD注册表值HKLM\SYSTEM\ CurrentControlSet\Control\Session Manager\MemoryManagement\PoolTagOverruns，并且将其设置为0（或者，运行Gflags工具，选择Verify Start选项，而不是默认的Verify End选项）。当Windows强制使用向下溢出检测时，驱动程序检验器在页面的开始处而不是在页面的尾部为驱动程序分配缓冲区。

有关溢出检测的配置还包含另一种检测向下溢出的方法。当驱动程序释放它的缓冲区以便将内存归还给驱动程序检验器时，驱动程序检验器确保该缓冲区前面的模式没有被改变。如果该模式已被改变了，则设备驱动程序已经向下溢出了该缓冲区，写数据的时候超出了该缓冲区。

特殊内存池的分配例程也要确保：在分配和释放内存块时处理器的IRQL是合法的。这一检查可以捕获到有些设备驱动程序可能犯的一种错误：在高于DPC/Dispatch级别的IRQL上分配可换页的内存。

你也可以手工配置特殊内存池，做法是，加入DWORD注册表值HKLM\ SYSTEM-CurrentControlSet\Control\Session Manager\Memory Management\PoolTag。该值代表了系统用于特殊内存池的分配标记。因此，即使驱动程序检验器没有被配置成检验某个特定的设备驱动程序，但只要该驱动程序在它申请的内存上关联了一个与PoolTag注册表值中指定的标记相符的标记，内存池的分配例程也将从特殊内存池中分配内存。如果你把PoolTag的值设置为0x0000002a或者通配符（*），则驱动程序分配的所有内存都来自于特殊内存池，前提是有足够的虚拟内存和物理内存（如果没有足够的空闲页面[存在一个界限]，则驱动程序将恢复到从常规内存池中分配内存的模式，但每次分配使用两个页面）。

内存池跟踪（pool tracking） 如果内存池跟踪选项被打开的话，则内存管理器在驱动程序卸载时检查它是否释放了它申请的所有内存。如果还没有全部释放，则让系统崩溃，并指明该驱动程序有错误。在驱动程序检验器管理器的Pool Tracking视图上，驱动程序检验器显示了一般的内存池统计信息。你也可以使用内核调试器命令*!verifier*。该命令显示了比驱动程序检验器更多的信息，对于驱动程序编写者这是很有用的。

强制IRQL检测 最常见的一种驱动程序错误是，当驱动程序当前执行所在的处理器处于一个提升的IRQL级别上时，该驱动程序访问可换页的数据或代码时就会产生错误。正如在第3章中所解释的那样，当IRQL处于DPC/Dispatch或更高级别时，内存管理器不能处理页面错误（page fault）。当处理器在高的IRQL级别上执行时，如果一个设备驱动程序访问的可换页数据当时碰巧驻留在内存中，则系统通常不会检测到它正在访问可换页数据的情形。然而，在其他时候，数据可能被换出去了，因而导致系统崩溃，停止码为IRQL_NOT_LESS_OR_EQUAL（即IRQL不小于或等于当前失败操作所要求的级别——在这个例子中，失败的操作是访问可换页的内存）。

www.TopSage.com

虽然测试设备驱动程序的这种错误通常非常困难，但是，驱动程序检验器使得这一任务非常容易。如果你选择了强制IRQL检查选项，则无论何时当处于检验模式下的设备驱动程序提升IRQL时，驱动程序检验器强制所有的内核模式可换页代码和数据都在系统工作集以外。实现这一功能的内部函数是*MmTrimAllSystemPagableMemory*。打开了这一设置以后，当IRQL被提升时，无论何时当处于检验模式的设备驱动程序访问可换页内存时，系统立刻检测到访问违例，结果导致系统崩溃，并标识出有错误的驱动程序。

低资源模拟 打开了低资源模拟选项以后，驱动程序检验器会随机地让处于检验模式下的设备驱动程序发出的内存分配请求失败。在过去，开发人员在编写许多设备驱动程序时，假设内核内存总是可用的，如果内核内存用完了的话，设备驱动程序无须操心这种情况，因为系统无论如何总是要崩溃的。然而，因为低内存条件有可能临时发生，所以，设备驱动程序正确地处理内存请求失败（表明内核内存已用完）是很重要的。

在系统引导起来之后7分钟开始（这段时间足够通过关键的初始化过程，在此过程中，低内存条件可能会阻止一个设备驱动程序被加载进来），驱动程序检验器开始随机地让它正在检验的设备驱动程序所发出的内存申请调用失败。如果一个驱动程序不能正确地处理内存分配失败，则这将可能导致一个系统崩溃。

对于设备驱动程序编写者来说，驱动程序检验器是他们的检验和调试工具库中的一个有益补充。许多设备驱动程序在刚开始与驱动程序检验器一起运行时，驱动程序检验器就能够暴露出它们的程序错误。因此，驱动程序检验器已经让Windows中所有正在运行的内核模式代码有了全面的质量提高。

7.4 虚拟地址空间的布局结构

这一节讲述用户和系统地址空间中的组成部分，然后再介绍32位和64位系统上的特定布局结构。这些信息有助于你理解在这两种平台上进程虚拟内存和系统虚拟内存中的一些限制。

在Windows中，有三种类型的数据被映射到虚拟地址空间中：每个进程特有的私有代码和数据、会话范围内的代码和数据，以及系统范围内的代码和数据。

正如在第1章中解释的那样，每个进程都有一个私有的地址空间，其他的进程不能访问此地址空间（除非它们得到了许可，允许打开该进程进行读或写访问）。进程内的线程永远不能访问此私有地址空间以外的虚拟地址，除非它们映射到共享内存区，以及/或者通过跨进程的内存函数以允许访问另一个进程的地址空间。在后面关于地址转译的一节中讲述了有关进程虚拟地址空间的信息，这些信息被称为**页表**（*page table*）。页表本身的页面被标记为只可以在内核模式下访问，所以，一个进程的用户线程不能修改它们自己的地址空间布局。

www.TopSage.com

在多会话的系统上（安装了终端服务的Windows 2000 Server，Windows XP和Windows Server 2003），**会话空间**（*session space*）包含了一些针对每个会话的全局信息。（关于会话的介绍，请参见第2章。）**会话**（*session*）是由进程和其他系统对象（比如窗口站、桌面和窗口）构成的，它们代表了一个用户的工作站登录会话。每个会话有一个专门的换页池区域，Windows子系统的内核模式部分（Win32k.sys）利用这部分内存区域来分配一些属于该会话私有的GUI数据结构。而且，每个会话都有它自己私有的Windows子系统进程（Csrss.exe）和登录进程（Winlogon.exe）拷贝。会话管理器进程（Smss.exe）负责创建新的会话（包括加载新会话私有的Win32k.sys拷贝）、创建该会话私有的对象管理器名字空间，以及创建该会话特有的Csrss和Winlogon进程实例。

为了虚拟化会话，所有会话范围的数据结构都被映射到系统空间中一个被称为**会话空间**（*session space*）的区域中。当一个进程被创建的时候，这一地址范围被映射到该进程所属会话的页面上。映射到会话空间中的视图的大小可以通过修改HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management下面的注册表值来配置（对于32位系统，如果系统是以/3GB方式来引导的，则这些注册表值被忽略）。

最后，**系统空间**（*system space*）包含了全局的操作系统代码和数据结构，它们对于所有的进程都是可见的。系统空间是由下面的部分组成的。

- **系统代码** 包含了操作系统映像、HAL，以及用于引导系统的设备驱动程序。
- **系统映射的视图** 用于映射Win32k.sys（即Windows子系统可加载的内核模式部分），以及系统使用的内核模式图形驱动程序（有关Win32k.sys的更多信息，请参见第2章）。
- **超空间** 一个用于映射进程工作集列表的特殊区域，在执行一些操作时它也被临时用于映射其他的物理页面，这样的操作有零化空闲列表上的页面（当零列表为空而又需要一个零页面时）、使其他页表中的页表项无效（比如当一个页面从备用列表中移除时），以及创建进程（以建立起新进程的地址空间）。
- **系统工作集列表** 此工作集列表数据结构描述了系统的工作集。*
- **系统缓存** 用于映射在系统缓存中已打开文件的虚拟地址空间（关于缓存管理器的更多信息，请参见第11章）。
- **换页内存池** 可换页的系统内存堆。

www.TopSage.com

- **系统页表项（PTE）** 系统PTE池，其中的PTE被用于映射系统页面，比如I/O空间、内核栈，以及内存描述符列表。通过检查性能工具中的“Memory: Free System Page Table Entries”计数器的值，你可以知道有多少系统PTE是可以使用的。
- **非换页内存池** 不可换页的系统内存堆，通常有两部分——在系统空间的低端和高端各有一部分。
- **崩溃转储信息** 保留的空间，用来记录有关一次系统崩溃的状态信息。
- **HAL使用** 保留给予HAL相关的数据结构的系统内存。



> 注 *在内部，系统工作集被称为*系统缓存工作集*（*system cache working set*）。然而，这一术语很容易让人误解，因为它不仅包含了系统缓存，而且也包含换页内存池、可换页的系统代码和数据，以及可换页的驱动程序代码和数据。

在讲述了Windows中虚拟地址空间的基本组成部分以后，现在我们来看一看x86、IA-64和x64平台上的特定布局情况。

x86用户地址空间的布局结构

在Windows的32位版本上，每个用户进程默认可以拥有多达2GB私有地址空间；操作系统使用另外的2GB空间。Windows 2000 Advanced Server、Windows 2000 Datacenter Server、Windows XP Service Pack 2及以后，以及Windows Server 2003（所有版本）都支持一个可允许用户地址空间多达3GB的引导时刻选项（在Boot.ini中的/3GB开关）。Windows XP和Windows Server 2003还支持另一个开关（/USERVA），它允许指定用户地址空间的大小在2GB和3GB之间（该值以兆字节为单位）。这两种可能的地址空间布局结构如图7.10所示。

能够让一个32位进程增长到2GB以上，这种能力只是作为一个短期解决方案被引入进来，目的是满足诸如数据库服务器之类的应用程序的需求，它们希望在内存中保留比原来2GB地址空间的进程还要更多的数据。要想访问比受限的2GB（或3GB）进程地址空间能容纳的数据还要更多，本章前面讲述的AWE函数提供了一种更好的方案来满足这种需求。

www.TopSage.com

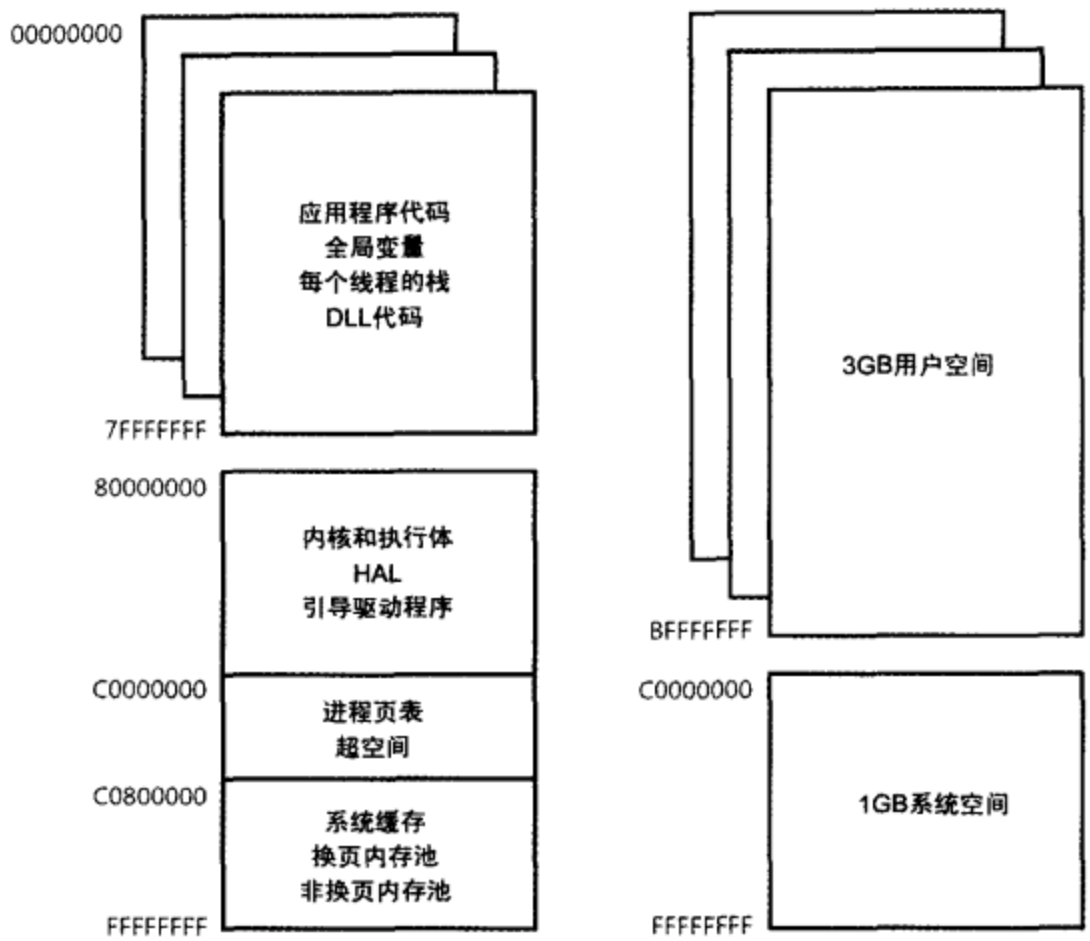


图7.10 x86虚拟地址空间布局结构

一个进程要想超越2GB地址空间的限制，其映像文件的头部必须设置了IMAGE_FILE_LARGE_ADDRESS_AWARE标志。否则，Windows为该进程保留2GB以外的地址空间，所以，应用程序不会看到大于0x7FFFFFFF的那部分虚拟地址（这是为了避免打破那些可能会打破这些地址的应用程序）。你在联编可执行文件时只要指定链接器标志/LARGEADDRESSAWARE，就可以设置映像头部的标志。如果在一个只有2GB用户地址空间的系统上运行这样的应用程序，这一标志没有任何影响（如果在一个不支持/3GB开关的Windows Server版本上用/3GB来引导，则系统空间被减少到1GB，但用户进程仍然被限制在2GB以内，即便所运行的映像文件已经设置了大地址空间感知标志）。

有几个系统映像已经被设置了大地址空间感知标志，所以，它们可以充分利用大进程地址空间的系统优势。这些系统映像是：

- Lsass.exe—本地安全权威子系统（Local Security Authority Subsystem）；
- Inetinfo.exe——Internet信息服务器（Internet Information Server）；
- Chkdsk.exe——检查磁盘的工具（Check Disk）；
- Dllhst3g.exe——Dllhost.exe的一个特殊版本（针对COM+应用）。

www.TopSage.com

最后，因为在默认情况下，通过VirtualAlloc来执行的内存分配是从低虚拟地址开始的，然后往高地址增长，所以，除非一个进程分配大量的虚拟内存，或者它有一个非常碎的虚拟地址空间，否则它永远也不会到达很高的虚拟地址上。因此，为了达到测试目的，你可以加入一个DWORD注册表值HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\AllocationPreference，并且将它设置为0x100000，从而强制从高地址开始分配内存。

x86系统地址空间的布局结构

这一部分介绍32位Windows中系统空间的详细布局结构以及内容。图7.11显示了在一个有2GB系统空间的x86系统上的总体结构。

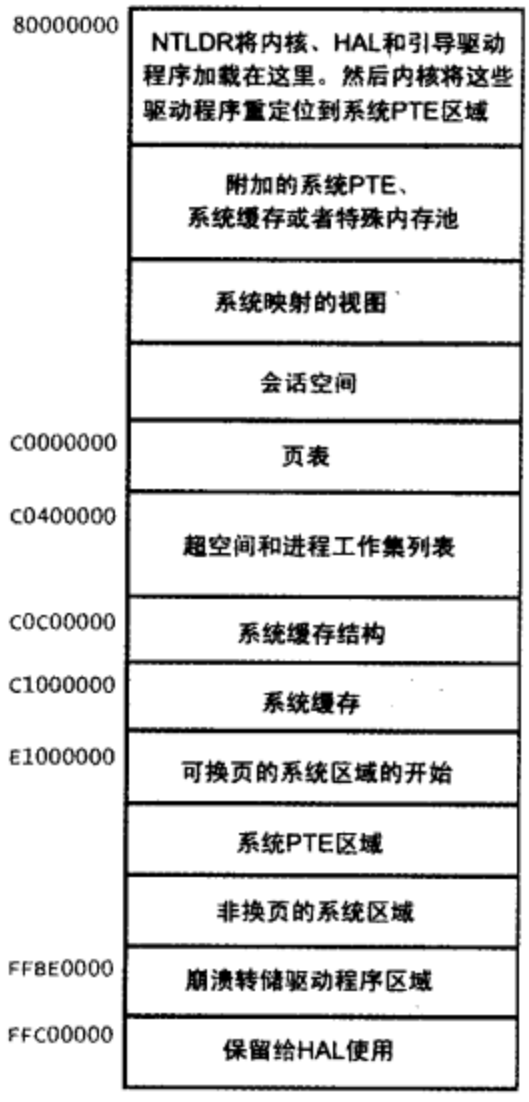


图7.11 x86系统空间布局结构（并非按比例绘制）

表7.8列出的内核变量包含了系统空间各个区域的开始和结束地址。其中有些区域是固定的，有些则是在系统引导时候根据内存大小以及所运行的Windows版本（客户版本或者服务器版本）而计算得到的。

www.TopSage.com

表7.8 描述了系统空间各个区域的系统变量

系统变量	说 明	x86 2-GB系统空间（非PAE）	x86 1-GB系统空间（非PAE）
MmSystemRangeStart	系统空间的开始地址	0x80000000	0xC0000000
MmSystemCacheWorkingSetList	系统工作集列表	0xC0C00000	计算得到
MmSystemCacheStart	系统缓存的开始	计算得到	计算得到
MmSystemCacheEnd	系统缓存的结束	计算得到	计算得到
MiSystemCacheStartExtra	系统缓存或系统PTE扩展的开始	计算得到	0
MiSystemCacheEndExtra	系统缓存或系统PTE扩展的结束	0xC0000000	0
MmPagedPoolStart	换页内存池的开始	计算得到	计算得到
MmPagedPoolEnd	换页内存池的结束	计算得到（最大尺寸是650MB）	计算得到（最小尺寸是160 MB）
MmNonPagedSystemStart	系统PTE的开始	计算得到（最低值为0xEB000000）	计算得到
MmNonPagedPoolStart	非换页内存池的开始	计算得到	计算得到
MmNonPagedPoolExpansionStart	非换页内存池扩充的开始	计算得到	计算得到
MmNonPagedPoolEnd	非换页内存池的结束	0xFFBE0000	0xFFBE0000

x86会话空间

对于有多个会话的系统，凡是每个会话特有的代码和数据都被映射到系统地址空间中，但是该会话中的进程可以共享这些代码和数据。图7.12显示了会话空间的一般布局结构。

www.TopSage.com



图7.12 x86会话空间布局结构（并非按比例绘制）

会话空间中的各个组成部分的大小是可以通过在HKLM\System\CurrentControlSet\Session Manager\Memory Management键下加入一些注册表值来配置的。表7.9列出了这些注册表值，以及一些包含了实际大小值的内核变量。

表7.9 会话空间的配置值

注册表值	内核变量
SessionImageSize	MmSessionImageSize
SessionViewSize	MmSessionViewSize
SessionPoolSize	MmSessionPoolSize



实验：查看会话

通过检查会话ID进程性能计数器，你可以显示哪些进程是哪些会话的成员。利用任务管理器、进程管理器或者性能工具，你都可以做到这一点。利用内核调试器的*!session*命令，你可以列出活动的会话，如下：

```
lkd> !session
Sessions on machine: 3
Valid Sessions: 0 1 2
Current Session 0
```

然后，你可以利用*!session –s*命令来设置活动的会话，而且，利用*!sprocess*命令，你还可以显示该会话数据结构的地址和该会话中的进程：

```
lkd> !session -s 2
Sessions on machine: 3
Implicit process is now 8144f3a0
Using session 2
lkd> !sprocess
Dumping Session 2
```


www.TopSage.com

```
_MM_SESSION_SPACE f9d5a000
_MMSESSION        f9d5ad80
PROCESS 8144f3a0  SessionId: 2  Cid: 0d18    Peb: 7ffdf000  ParentCid: 0134
    DirBase: 07ef7000  ObjectTable: e1855bb0  HandleCount:  71.
    Image: csrss.exe

PROCESS 813f6500  SessionId: 2  Cid: 0d04    Peb: 7ffdf000  ParentCid: 0134
    DirBase: 0f3fd000  ObjectTable: e18f3008  HandleCount: 131.
    Image: winlogon.exe
```

为了查看会话的细节，可以使用*dt*命令将MM_SESSION_SPACE结构转储出来，如下所示：

```
lkd> dt nt!_mm_session_space f9d4a000
nt!_MM_SESSION_SPACE
   +0x000 GlobalVirtualAddress : 0xf9d4a000
   +0x004 ReferenceCount    : 0x19
   +0x008 u                 : __unnamed
   +0x00c SessionId         : 0
   +0x010 ProcessList       : _LIST_ENTRY [ 0x817d92cc - 0x812de88c ]
   +0x018 LastProcessSwappedOutTime : _LARGE_INTEGER 0x0
   +0x020 SessionPageDirectoryIndex : 0x6a29
   +0x024 NonPagablePages   : 0xf
   +0x028 CommittedPages    : 0x3f
   +0x02c PagedPoolStart    : 0xbc000000
   +0x030 PagedPoolEnd      : 0xbc3fffff
   +0x034 PagedPoolBasePde  : 0xc0300bc0
   +0x038 Color             : 0xd6
   +0x03c ProcessOutSwapCount : 5
   +0x040 SessionPoolAllocationFailures : [4] 0
   +0x050 ImageList         : _LIST_ENTRY [ 0x81801fa8 - 0x816933a0 ]
   +0x058 LocaleId          : 0x409
   +0x05c AttachCount       : 0
   +0x060 AttachEvent       : _KEVENT
   +0x070 LastProcess       : (null)
   +0x074 ProcessReferenceToSession : 25
   +0x078 WsListEntry       : _LIST_ENTRY [ 0x8056d058 - 0x8056d058 ]
   +0x080 Lookaside         : [26] _GENERAL_LOOKASIDE
   +0xd80 Session           : _MMSESSION
   +0xdc0 PagedPoolMutex    : _KGUARDED_MUTEX
   +0xde0 PagedPoolInfo     : _MM_PAGED_POOL_INFO
   +0xe00 Vm                : _MMSUPPORT
   +0xe60 Wsle              : 0xbf400038
   +0xe64 Win32KDriverUnload : 0xbf9169ab     +ffffffffbf9169ab
   +0xe68 PagedPool         : _POOL_DESCRIPTOR
   +0x1e98 PageTables        : 0x816e7008
   +0x1e9c ImageLoadingCount : 0
```


www.TopSage.com



实验：查看会话空间的使用情况

在内核调试器中，利用*!vm 4*命令你可以查看会话空间的内存使用情况。例如，下面的输出是从一个32位Windows Server 2003（企业版）系统中得到的，它有两个活动会话：

```
lkd> !vm 4
.
.
Terminal Server Memory Usage By Session:

Session Paged Pool Maximum is 4096K
Session View Space Maximum is 49152K

Session ID 0 @ f9d4a000:
Paged Pool Usage:           0K
Commit Usage:             252K

Session ID 1 @ f9d4c000:
Paged Pool Usage:           0K
Commit Usage:             172K
```

在一个同样具有两个活动会话的x64 Windows Server 2003（企业版）系统上执行同样的命令，你可以看到更大的会话空间区域：

```
Terminal Server Memory Usage By Session:
Session Paged Pool Maximum is 65536K
Session View Space Maximum is 106496K

Session ID 0 @ fffffadfe4bb9000:
Paged Pool Usage:           0K
Commit Usage:            4608K
Session ID 1 @ fffffadfe4c5f000:
Paged Pool Usage:           0K
Commit Usage:             584K
```

系统页表项（PTE，Page Table Entry）

利用系统页表项，可以动态地映射系统页面，比如I/O空间、内核栈和内存描述符列表。系统PTE并不是无限可用的资源。例如，Windows 2000只能描述660MB可变的系统虚拟地址空间（其中440MB可以是连续的）。在32位版本的Windows XP和Windows Server 2003上，可以使用的系统页表项（PTE）数量已经增大了，因此，这样的系统可以描述1.3GB的系统虚拟地址空间，其中960MB可以是连续的。在64位Windows上，系统PTE可以描述多达128GB的连续虚拟地址空间。

www.TopSage.com

你只需在性能工具中检查“Memory: Free System Page Table Entries”计数器的值，就可以看到有多少系统PTE是可以使用的。在默认情况下，Windows在引导时根据内存的大小，计算出有多少系统PTE可以分配。你只要将注册表值HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\SystemPages设置成你期望的PTE数量，就可以覆盖系统默认的计算值（这样做可能是为了支持一个设备，它要求大量的系统PTE，比如，一块带有512MB显示内存的视频卡必须同时映射所有的显示内存）。如果该值是0xFFFFFFFF，则系统保留最大数量的系统PTE。

64位地址空间布局结构

理论上的64位虚拟地址空间是16艾字节（18446744073709551616字节，或者近似17.2十亿千兆字节）。不像在x86系统上默认的地址空间被分成两部分（一半给进程，一半给系统），64位地址被分成许多大小不同的区域，这些区域的组成部分在概念上也符合用户空间、系统空间和会话空间中的各个部分。表7.10列出了这些区域的各种大小值，它们代表了当前实现上的一些限制，在将来的版本中它们很容易被扩展（在32位平台上的大小值也包含进来了，以便于做比较）。很显然，从地址空间大小的角度来看，64位系统代表了一个巨大的跳跃。

表7.10 64位地址空间的大小

区域	IA-64	x64	x86
进程地址空间	7152 GB	8192 GB	2至3 GB
系统PTE空间	128 GB	128 GB	1.2 GB
系统缓存	128 GB	128 GB	960 MB
换页内存池	128 GB	128 GB	470至650 MB
非换页内存池	128GB	128 GB	256 MB

IA-64和x64地址空间布局结构的细节有些微不同。IA-64地址空间布局结构显示在图7.13中，而x64地址空间布局结构显示在图7.14中。

www.TopSage.com

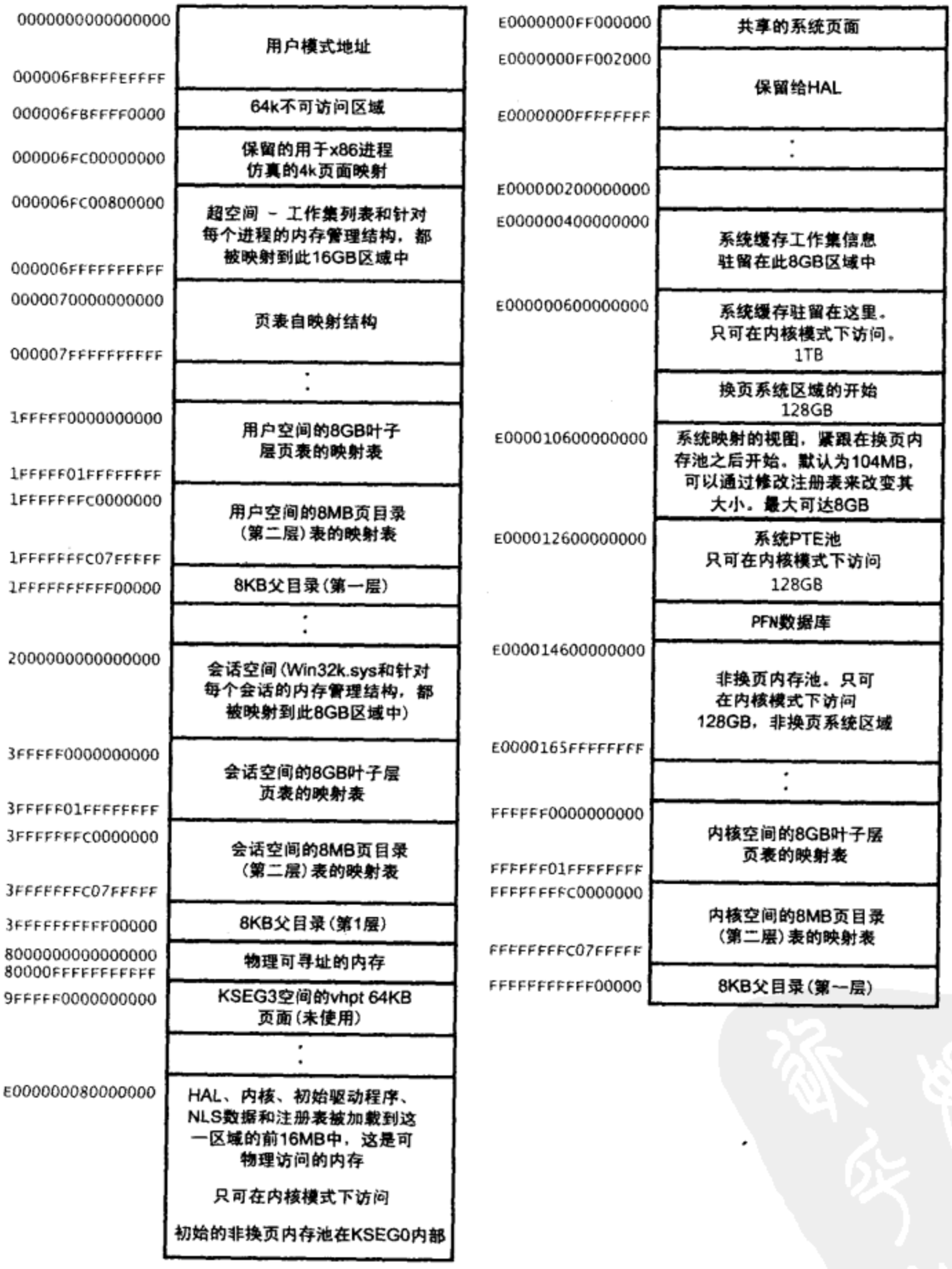


图7.13 IA-64地址空间布局结构

www.TopSage.com

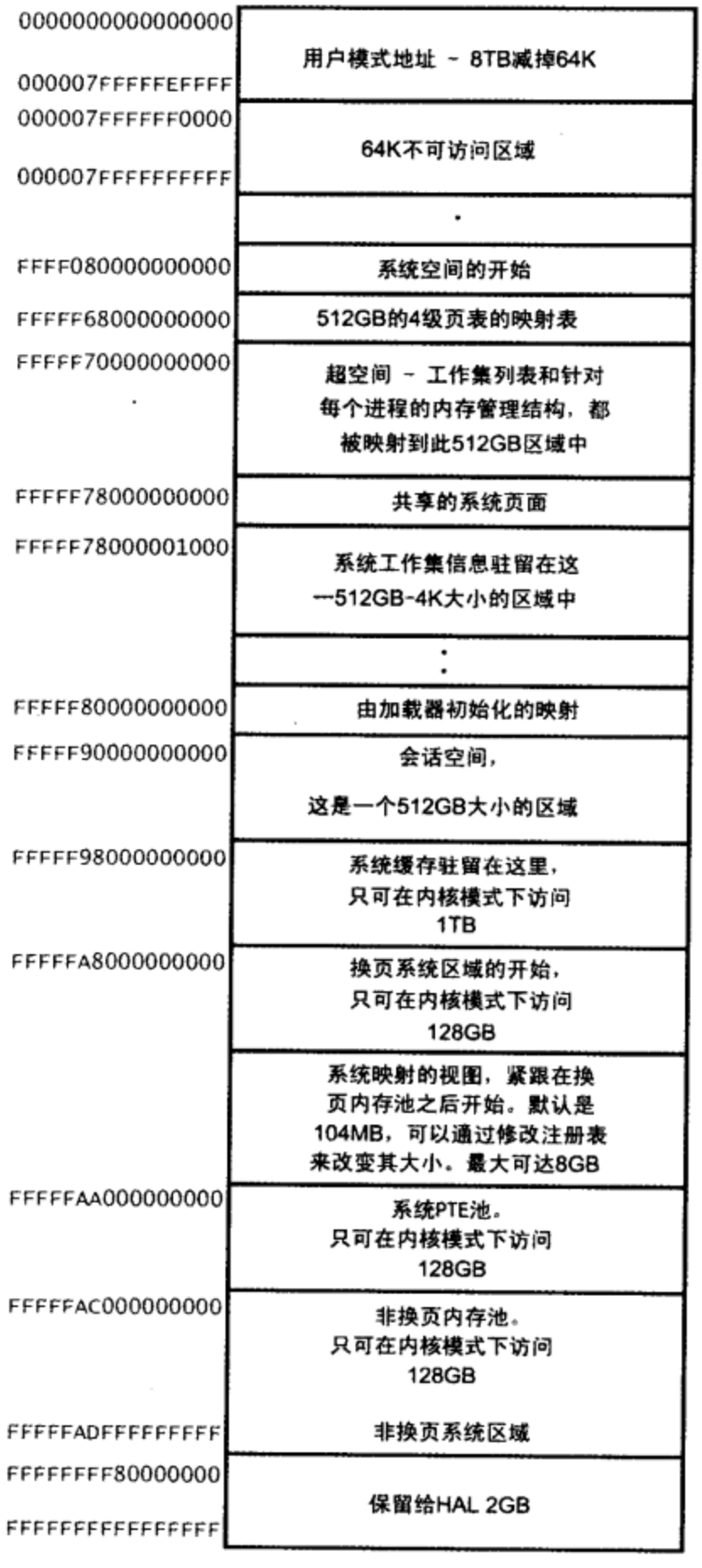


图7.14 x64地址空间布局结构

www.TopSage.com

7.5 地址转译

我们已经看过了Windows是如何构建其虚拟地址空间的，现在我们来看一看它如何将这些地址空间映射到实际的物理页面中。用户应用程序和系统代码引用的是虚拟地址。这一节首先详细地介绍32位x86地址转译，然后简要地说明在64位IA-64和x64平台上的不同之处。在下一节中，我们将要介绍，每当一个虚拟地址不能被解析到一个物理内存地址时将发生什么事情（换页），以及Windows如何通过工作集和页面帧数据库来管理物理内存。

x86虚拟地址转译

内存管理器创建和维护了一种被称为**页表**（*page table*）的数据结构，利用这种数据结构，CPU将虚拟地址转译成物理地址。每个虚拟地址都跟一个系统空间的结构联系在一起，该结构被称为**页表项**（PTE，*page table entry*），其中包含了该虚拟地址所对应的物理地址。例如，图7.15显示了在x86系统上，三个连续的虚拟页面如何被映射到三个物理上不连续的页面中。

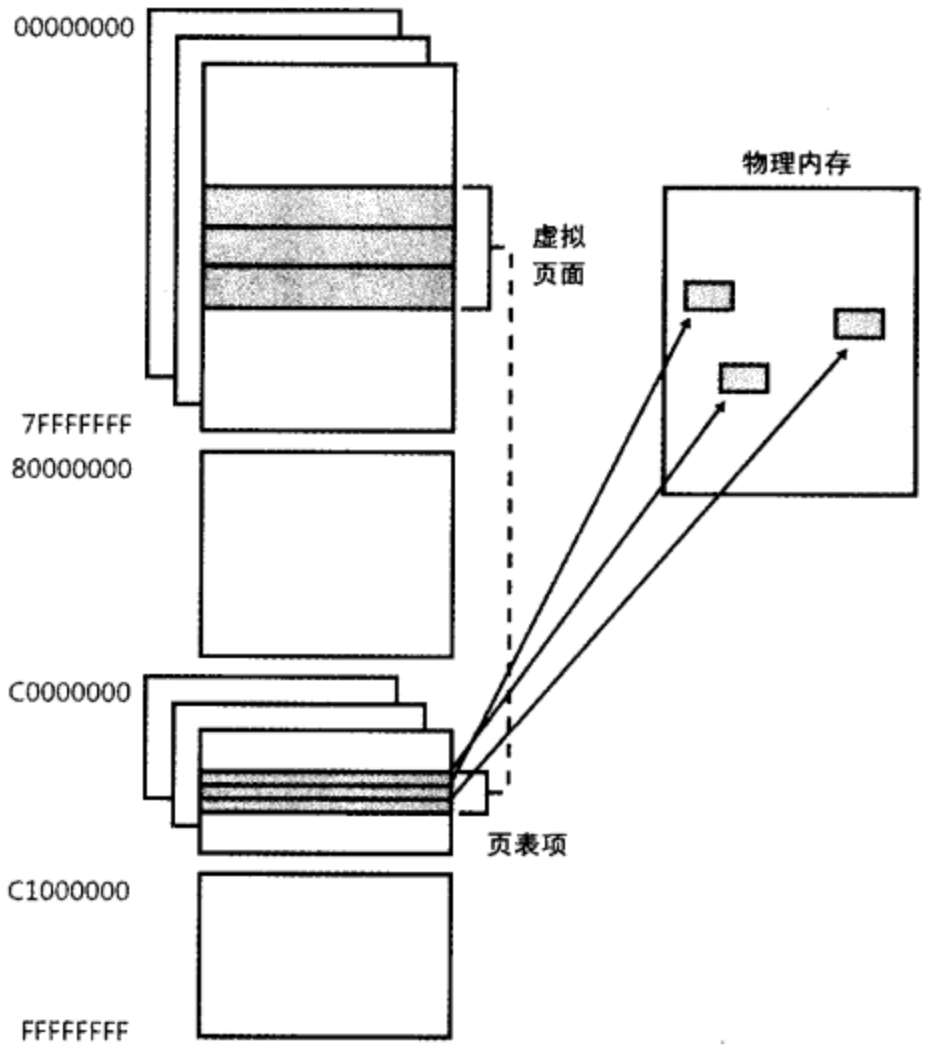


图7.15 从虚拟地址到物理内存的映射（x86）

www.TopSage.com

在图7.15中，连接虚拟页面与PTE的虚线代表了虚拟页面和物理内存之间的间接关系。



> **注** 内核模式代码（比如设备驱动程序）可以引用物理内存地址，其做法是将它们映射到虚拟地址上。有关更多的信息，请参见DDK文档中讲述的内存描述符列表（MDL）支持例程。

在默认情况下，在x86系统上，Windows使用二级页表结构来把虚拟地址转译成物理地址（运行PAE内核的x86系统用到了三级页表——这一小节假设非PAE系统）。一个32位虚拟地址被解释成三个单独的部分：页目录索引、页表索引和字节索引。这三部分都是索引值，分别指向了描述页映射的结构，如图7.16所示。页面的大小和PTE的宽度决定了页目录和页表索引域的宽度。例如，在x86系统上，字节索引是12位，因为页面是4096字节（2^{12}=4096）。

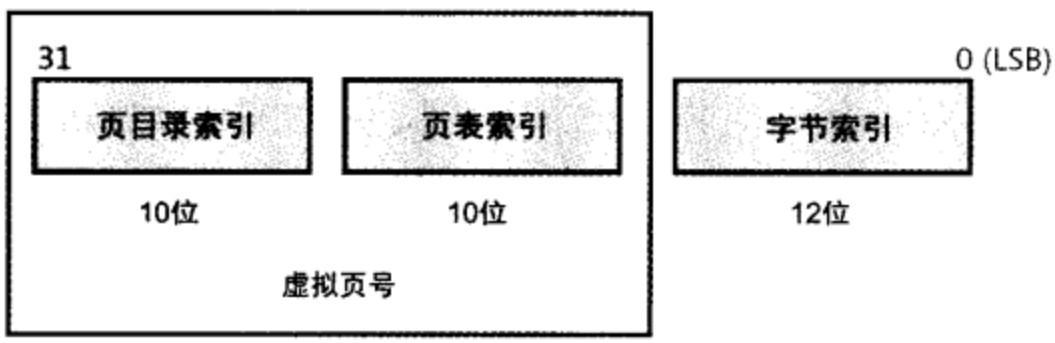


图7.16 在x86系统上，一个32位虚拟地址的组成部分

页目录索引被用来定位到该虚拟地址的PTE所在的页表。**页表索引**被用来定位到正确的PTE，如前所提及，PTE包含了一个虚拟页面所映射的物理地址。**字节索引**找到该物理页面中的正确地址。图7.17显示了这三个值的关系，以及如何通过它们将一个虚拟地址映射至一个物理地址。

www.TopSage.com

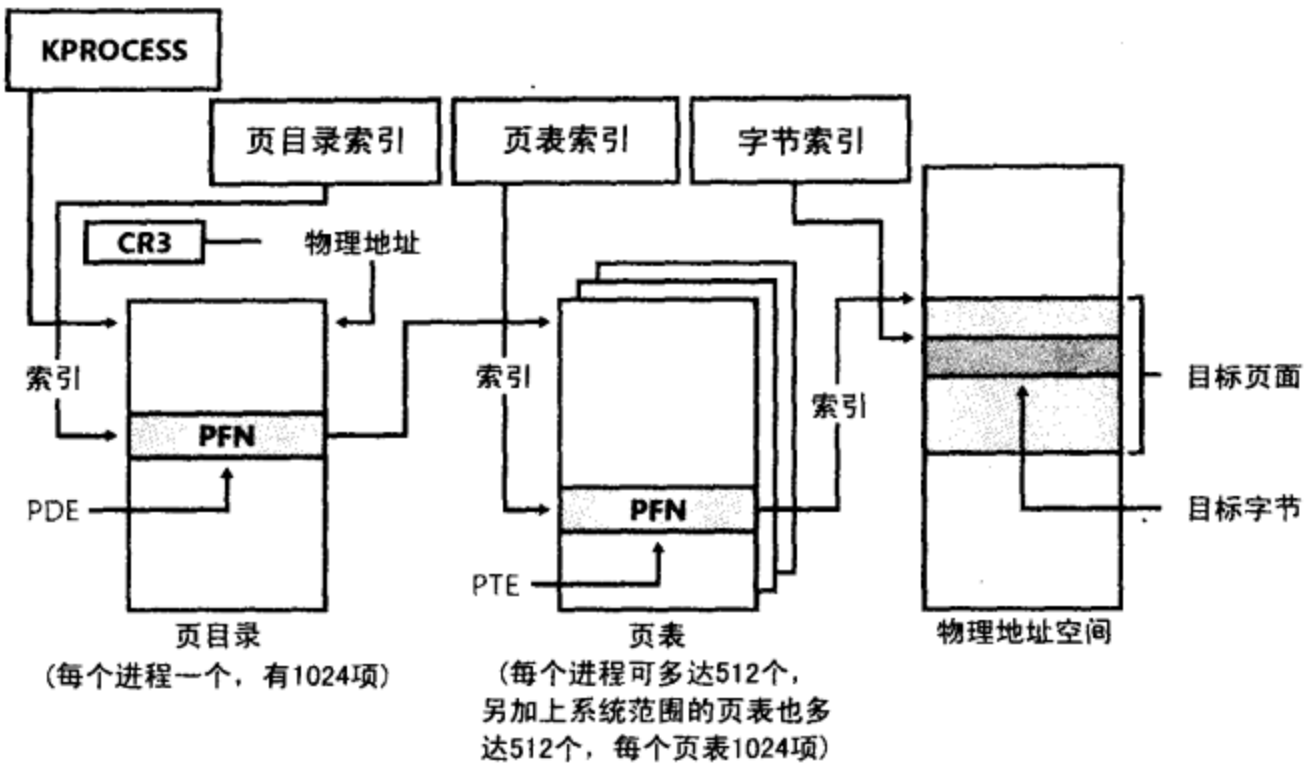


图7.17 转译一个有效的虚拟地址（针对x86）

在转译一个虚拟地址时，将涉及下面的基本步骤。

1. 内存管理硬件找到当前进程的页目录。在每次进程环境切换的时候，硬件也会得到一个新的进程页目录的地址，通常由操作系统设置一个特殊的CPU寄存器来实现。
2. 利用页目录索引作为该进程页目录中的一个索引，从而找到相应的页目录项（PDE），该页目录项描述了映射此虚拟地址所需要的页表的位置。该PDE包含了此页表的页面帧编号（PFN, page frame number）。（如果它是驻留在内存中的话——注意页表是可以被换出的。）
3. 利用页表索引作为该页表中的一个索引，从而找到那个描述了目标虚拟页面所在物理位置的PTE。
4. 利用PTE来定位到页面。如果该页面是有效的，则它包含了在物理内存中对应于此虚拟页面的物理页面的PFN。如果此PTE指示该页面是无效的，则内存管理错误处理器（fault handler）找到该页面，并试图使它变成有效的页面（参见有关页面错误处理的章节）。如果该页面不能变成有效的（例如，因为一个保护错误），则错误处理器产生一个访问违例或者一个错误检验（bug check）。
5. 当该PTE指向一个有效页面时，利用字节索引可定位到物理页面中目标数据的地址上。

你已经看过总体的结构了，现在来看一看页目录、页表和PTE的详细结构。

www.TopSage.com

页目录

每个进程都有一个**页目录**（*page directory*），这是内存管理器为了映射该进程中所有页表位置而创建的一个页面。进程页目录的物理地址被保存在内核进程（KPROCES）块中，在x86系统上，它也被虚拟映射至地址0xC0300000（在运行PAE内核映像的系统上，被映射至地址0xC0600000）。在内核模式中运行的所有代码引用的都是虚拟地址，而不是物理地址（有关KPROCESS和其他的进程数据结构的更多详细信息，请参考第6章）。

CPU知道页目录页面的位置，因为CPU内部的一个特殊寄存器（在x86系统上是CR3）中包含了页目录的物理地址（由操作系统加载此寄存器）。每次当环境切换发生时，如果将要运行的线程不在当前执行线程所属的进程中，那么，内核中负责环境切换的例程，把将要切换过去的目标进程的KPROCESS块中的页目录物理地址加载到该寄存器中。如果是在同一进程内的两个线程之间进行环境切换，则不需要重新加载页目录的物理地址，因为同一进程内的所有线程共享同样的进程地址空间。

页目录是由**页目录项**（PDE, *page directory entries*）构成的，每个PDE是4字节长（在运行PAE内核映像的系统上为8字节长），页目录描述了该进程所有可能的页表的状态和位置（正如本章稍后将要介绍的那样，页表是按需创建的，所以大多数进程的页目录只指向一小部分页表）。这里不再重复PDE的格式，因为它与硬件PTE基本上是相同的。

在非PAE模式下运行的x86系统上，若要描述完整的4GB虚拟地址空间，则需要1024个页表。映射这些页表的进程页目录包含了1024个PDE。因此，页目录索引必须是10位宽（2^{10}=1024）。在PAE模式下运行的x86系统上，一个页表内有512项（因为页目录索引是9位宽）。因为有4个页目录，所以结果是，最多可有2048个页表。

www.TopSage.com



实验：检查页目录和PDE

你只要在内核调试器的*!process*输出中检查DirBase域，就可以看到当前运行进程的页目录的物理地址：

```
                PDE的物理地址
                    |
kd> !process        |
PROCESS 80145880 | Cid: 0000    Peb: 00000000    ParentCid: 0000
    DirBase: 00030000  ObjectTable: 80695668  TableSize: 126.
    Image: Idle
    VadRoot 0 Clone 0 Private 0. Modified 0. Locked 0.
    80145A3C MutantState Locked OwningThread 0
    Process Lock Owned by Thread 0
        .
        .
        .
```

通过在内核调试器中检查一个特定虚拟地址的PTE，你可以看到页目录的虚拟地址，如下所示：

```
                          PDE的虚拟地址
                              |
                              |          PDE的内容
                              |            |
kd> !pte 50001                |            |
00050001  -  PDE at C0300000  |            |      PTE at C0000140
             contains 00700067 ------------------ contains 00E63047
             pfn 00700   --DA--UW          pfn 00E63   --D---UWV
                |           |
                |           |
          页表页面的PFN   页表页面的状态和保护
```

关于内核调试器输出中PTE部分的定义，请参见“页表项”一节。

因为Windows为每个进程提供了一个私有的地址空间，所以，每个进程都有它自己的一组进程页表，以便映射该进程的私有地址空间。然而，描述系统空间的页表则是在所有的进程之间共享的（而会话空间则是在一个会话内的所有进程之间共享的）。为了避免发生多个页表描述同样虚拟内存的情形，当一个进程被创建的时候，描述系统空间的页目录项首先被初始化，使它们指向已有的系统页表。如果该进程属于一个会话，则会话空间的页表也是共享的，即会话空间的页目录项指向已有的会话页表。

www.TopSage.com

但是，如图7.18所示，并不是所有的进程都有同样的系统空间视图。例如，如果换页内存池在扩展时要求分配一个新的系统页表，则内存管理器并不回头更新所有的进程页目录，以便让它们也指向新的系统页表；相反地，只有当这些进程引用新的虚拟地址时，它才更新进程页目录。

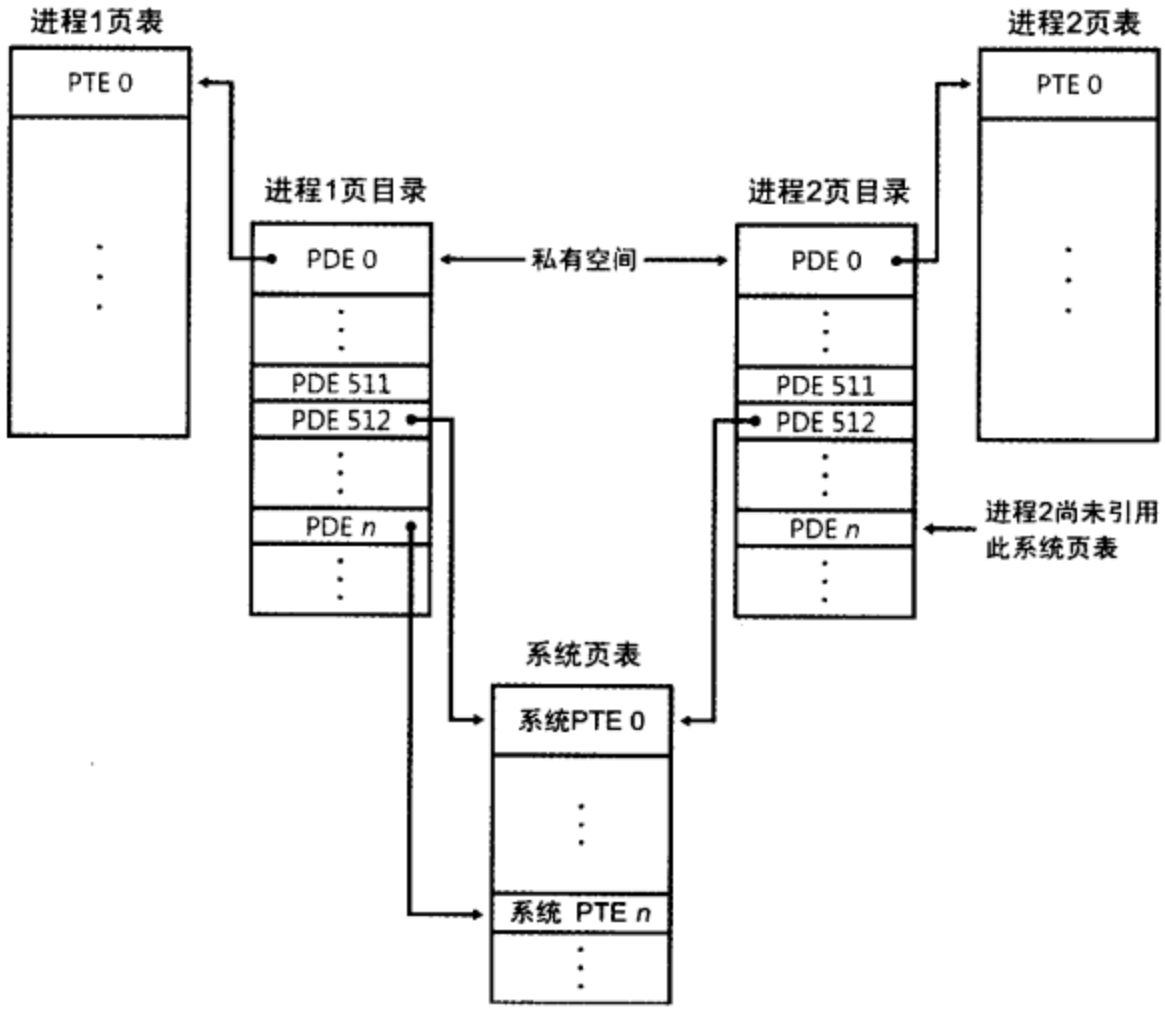


图7.18 系统页表和进程私有的页表

因此，当一个进程引用此换页内存池时，虽然该内存池中被引用的部分实际上驻留在物理内存中，但是该进程仍可能会导致一个页面错误，因为它的进程页目录尚未指向新的、描述了该内存池新区域的系统页表。当访问非换页内存池时，尽管此内存池也可以被扩展，但页面错误不会发生，因为Windows在系统初始化过程中已经建立了足够的系统页表来描述最大数量的内存。

页表和页表项

进程页目录项指向单独的页表。页表是由PTE阵列组成的。虚拟地址的页表索引域（见图7.17）指明了页表内的哪个PTE映射了所考虑的数据页面。在x86系统上，页表索引是10位宽（在PAE模式下是9位），从而允许你引用多达1024个4字节PTE（在PAE系统上为512个8字节PTE）。

www.TopSage.com

然而，因为32位Windows提供了4GB的私有虚拟地址空间，所以，为了映射整个地址空间，需要不止一个页表。那么，为了映射整个4GB进程虚拟地址空间，到底需要多少个页面呢？很简单，将4GB除以单个页表所映射的虚拟内存大小即可。曾经提到过，在x86系统上每个页表可映射4MB（在PAE模式下是2MB）数据页面。因此，1024个页表（4GB/4MB），或者针对PAE，2048个页表（4GB/2MB）可以映射整个4GB地址空间。

在内核调试器中你可以使用*!pte*命令来检查PTE。（参见实验“转译地址”）在这里我们将讨论有效的PTE，在后面的部分我们将讨论无效PTE。有效的PTE包含两个主域：数据所在的物理页面的页面帧编号（PFN）或者内存中一个页面的物理地址的PFN，以及一些标志描述了该页面的状态和保护属性，如图7.19所示。

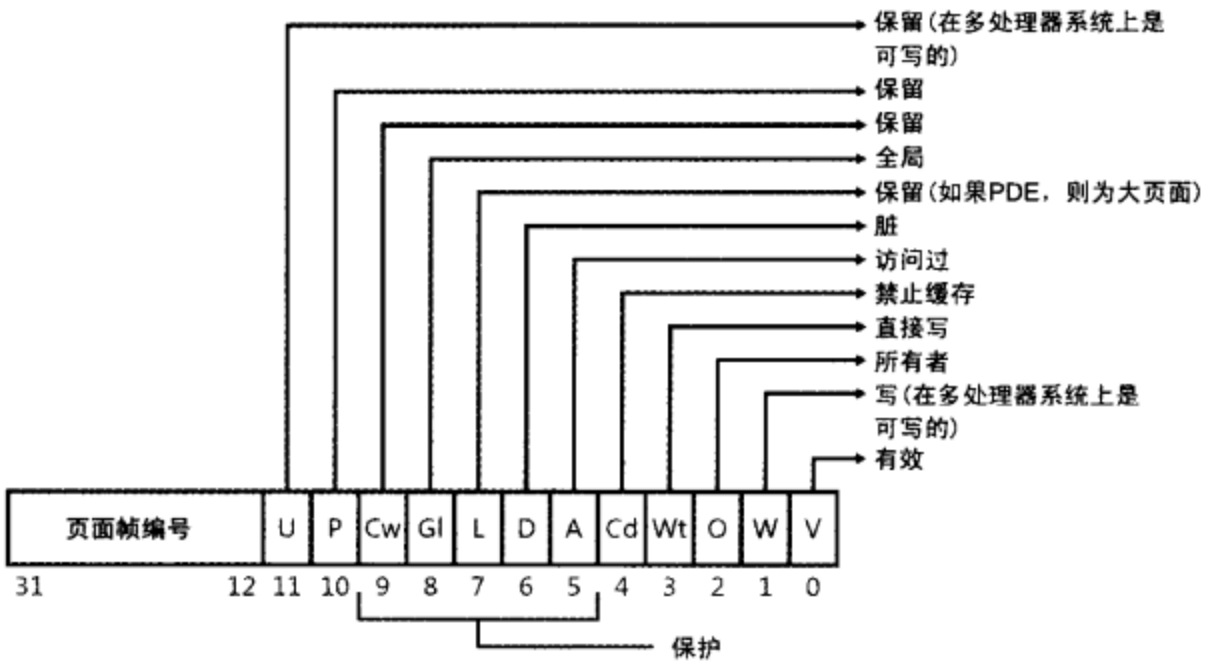


图7.19 有效的x86硬件PTE

你在稍后将会看到，图7.19中被标记为“保留”的位只有当PTE是有效的时候才会被使用（这些位由软件来解释）。表7.11简要地描述了在一个有效的PTE中由硬件定义的位。

表7.11 PTE状态和保护位

位的名称	含 义
访问过（Accessed）	页面已经被读过了
禁止缓存（Cache disabled）	针对该页面，禁止缓存
脏（Dirty）	页面已经被写过了
全局（Global）	此页面的转译适用于所有的进程（例如，转译缓冲区刷新操作不影响此PTE）

www.TopSage.com

续表

位的名称	含 义
大页面（Large page）	指明该PDE映射一个4MB页面（或者在PAE系统上为2MB）。参见本章前面的“大页面和小页面”一节
所有者（Owner）	指明用户模式代码是否可以访问该页面，或者该页面是否仅限于内核模式访问
有效（Valid）	指明此地址转译是否映射到物理内存中的一个页面
直接写（Write through）	在写这个页面时禁止缓存，所以数据的变化被直接刷新到磁盘上
写（Write）	在单处理器系统上，指明了该页面是可读写的，还是只读的；在多处理器系统上，指明了该页面是否是可写的 （此Write位被保存在PTE的一个保留位中）

在x86系统上，硬件PTE包含了一个“脏（Dirty）”位和一个“访问过（Accessed）”位。如果该PTE所代表的物理页面尚未被读过或写过，则“访问过”位被清除；当该页面被第一次读或者写的时候，处理器设置该位。只有当一个页面被第一次写的时候，处理器才会设置“脏”位。除了这两位以外，x86系统结构有一个“写（Write）”位提供了页面保护。当该位被清除的时候，页面是只读的；当它被设置的时候，页面是可读写的。如果一个线程试图往一个“写”位已被清除的页面中写入数据，则会引发一个内存管理异常，内存管理器的访问错误处理器（access fault handler，在下一部分中介绍）必须做出决定，该线程是否可以写该页面（比如，如果该页面实际上被标记为“写时复制”），或是否应产生一个访问违例。

在多处理器x86系统上，硬件PTE有一个额外的“写”位，它是由软件实现的，目的是为了避免在跨处理器刷新PTE缓存（称为**地址转译快查缓冲区**[*translation look-aside buffer*]，在下一节中介绍）时产生停滞（stall）。该位指明了一个页面已经被另一个处理器修改过了。

页面内的字节

一旦内存管理器已经找到了所考虑的物理页面，它还必须找到该页面内所要求的数据。这正是字节索引域的功能。字节索引域告诉CPU，你想要引用的是该页面内哪个字节。在x86系统上，字节索引是12位宽，从而允许你可以引用多达4096字节的数据（这是一个页面的大小）。所以，只需将字节偏移加上从PTE中获得的物理页面号，就完成了从虚拟地址到物理地址的转译过程。

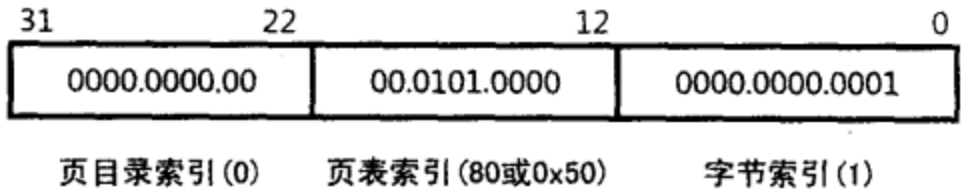
www.TopSage.com



实验：转译地址

为了进一步阐述清楚地址转译过程是如何工作的，我们来看一个实际的例子：在非PAE的x86系统上转译一个虚拟地址，我们将使用内核调试器中的工具来检查页目录、页表和PTE。在这个例子中，我们所使用的进程已经将虚拟地址0x50001映射到一个有效的物理地址上。在后面的例子中，你将会通过内核调试器看到，无效地址的地址转译是如何完成的。

首先，我们将0x50001转换成二进制表示，并将它分成三个域以便用于地址转译。0x50001的二进制形式是100.0000.0000.0000.0001。分成三个域之后如下所示：



为了开始地址转译过程，CPU需要进程页目录的物理地址，当一个进程中的线程正在运行时，该进程的页目录的物理地址存储在CR3寄存器中。要想显示此地址，你只需检查CR3寄存器的内容，或者通过*!process*命令将一个进程的KPROCESS块转储出来，如下所示：

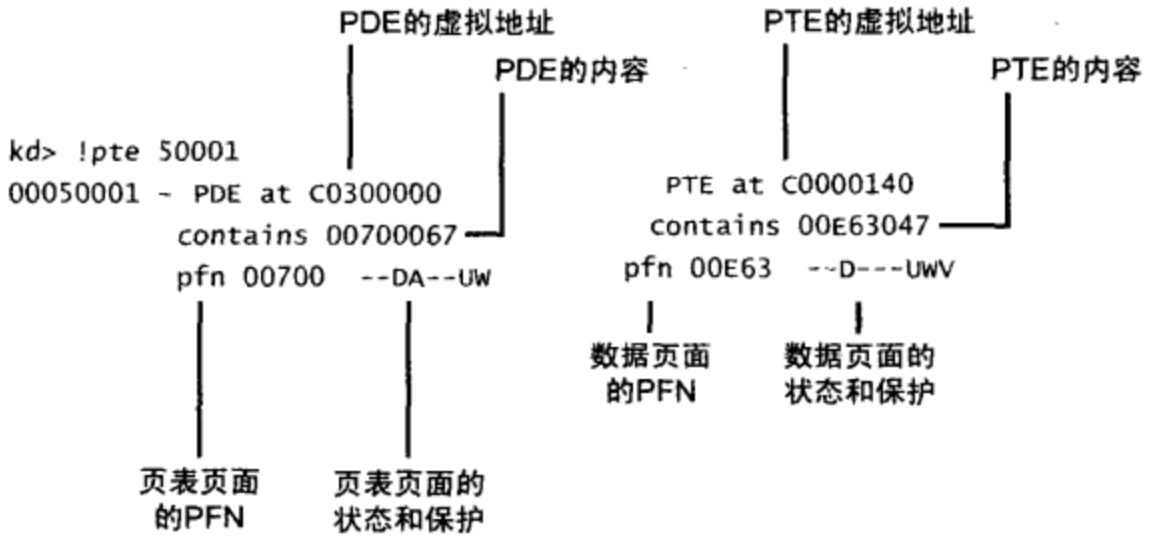
```
kd> !process
PROCESS 81555020 Cid: 0099 Peb: 7ffdf000  ParentCid: 0094
  DirBase: 012f0000  ObjectTable: 80695ba8  TableSize: 46.
  Image: IEXPLORE.EXE
        .
        .
        .
kd> r cr3
Cr3: 012F0000
```

页目录的物理地址

在这个例子中，页目录被存储在物理地址0x12F0000处。如前所述，这个例子中的页目录索引域是0，因此，对应的PDE位于物理地址0x12F0000处。

内核调试器的*!pte*命令显示了一个虚拟地址的PDE和PTE，如下所示：

www.TopSage.com



在上面的第一列，内核调试器显示了PDE，在第二列，显示了PTE。请注意，PDE地址被显示为一个虚拟地址，而不是物理地址——正如前面所说明的，在x86系统上，进程的页目录从虚拟地址0xC0300000开始。因为我们在检查页目录的第一个PDE，所以，该PDE的地址与页目录地址相同。

PTE位于虚拟地址0xC0000140处。你只需将页表索引（本例子中为0x50）乘以PTE的大小值，就可以计算出PTE的虚拟地址，即0x50乘以4等于0x140。因为内存管理器从0xC0000000开始映射页表，所以，加上140之后，可得到在内核调试器中给出的虚拟地址：0xC00000140。页表页面位于PFN 0x700处，数据页面位于0xe63处。

PTE标志被显示在PFN编号的右侧。例如，上述问题中被引用页面的PTE标志为D---UWV。这里D代表了脏（该页面已经被修改了），U代表了用户模式页面（相对于内核模式页面），W代表可写页面（而不是只读的），V代表有效的（此PTE表达了物理内存中的一个有效页面）。

地址转译快查缓冲区

前面我们已经学过了，每次地址转译都要求两次查询操作：一次是在页目录中找到正确的页表，另一次是，在页表中找到正确的表项。因为对于每个虚拟地址引用都需要两次额外的内存查询，这样会导致不可接受的系统性能下降，所以，大多数CPU都缓存了地址转译信息，因而在重复访问同一个地址时无需再进行转译。处理器提供这种缓存的形式是一种被称为**地址转译快查缓冲区**（*translation look-aside buffer*）或TLB的关联内存阵列。像TLB这样的关联内存是一个向量，其中的单元可以被并发读取，并且与目标值做比较。在TLB的情况下，该向量包含了关于最近使用过的页面的、从虚拟地址到物理地址的页面映射，如图7.20所示。而且，页面

www.TopSage.com

保护类型适用于每个页面。TLB中的每个表项就像一个缓存项，它的标记部分是虚拟地址的各个部分，而数据部分则包含了物理页面号、保护域、有效位，通常还包含一个“脏”位（它指明了此缓存PTE所对应的页面的条件）。如果一个PTE的“全局（Global）”位（用于系统空间的页面，它们对于所有的进程都是可见的）被设置了，则该TLB项不会在进程环境切换的时候被置为无效。

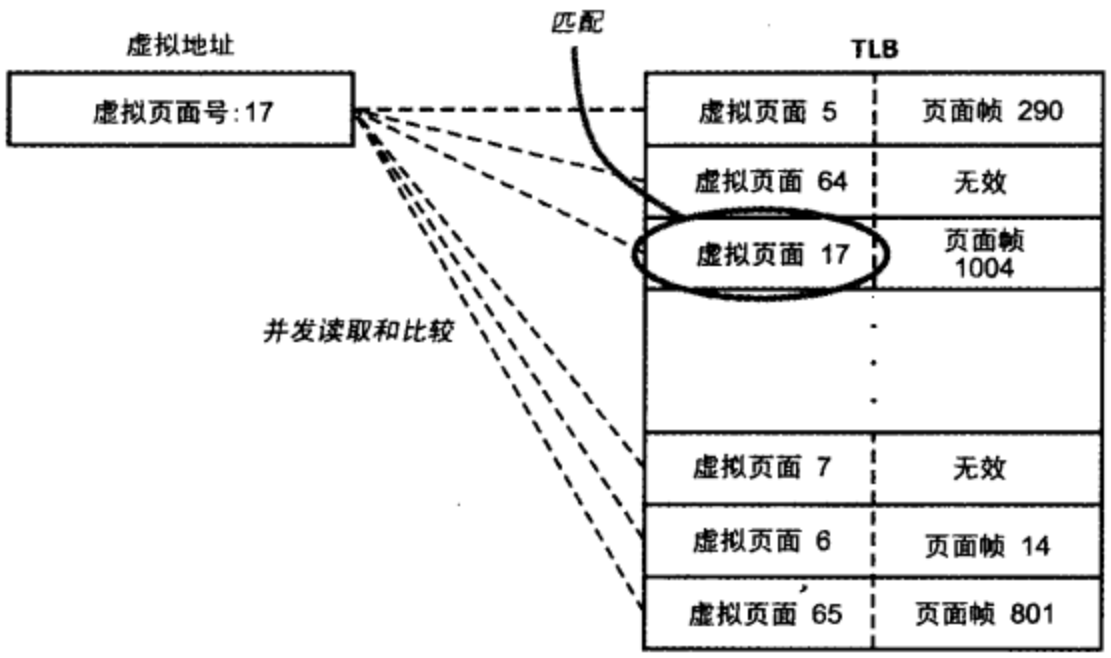


图7.20 访问地址转译快查缓冲区

那些频繁使用的虚拟地址很有可能在TLB中有对应的表项，因而实现了绝对快速的从虚拟地址到物理地址的转译过程，从而提供了快速的内存访问。如果一个虚拟地址不在TLB中，那么它可能仍然在内存中，但是需要多次内存访问才能找到它，这使得对它的访问要稍微慢一些。如果一个虚拟页面已经被换出内存了，或者，如果内存管理器已经改变了此PTE，那么，内存管理器会显式地使该TLB项变为无效。如果一个进程再次访问该页面，则产生一个页面错误，并且内存管理器将该页面换回到内存中，在TLB中重新为它建立一个表项。

为了尽可能地使用公共的代码，内存管理器只要有可能就会统一对待所有的PTE，无论它们是由硬件维护的，还是由软件维护的。例如，一个PTE从无效变为有效的时候，就会调用一个内核例程。该内核例程的任务是，按照硬件体系结构所要求的方式，将新的PTE加载到TLB中。在x86系统上，此代码是一个NOP，因为处理器无需任何软件的介入就会加载TLB。

物理地址扩展（PAE）

Intel x86 Pentium Pro处理器引入了一种称为**物理地址扩展**（PAE, Physical Address Extension）的内存映射模式。通过适当的芯片集，PAE模式使得在当前的Intel x86处理器上可以访问多达64GB的物理内存，在x64处理器上可以访问多达1024GB的物理内存（不过，由于在映射内存时所需要的PFN数据库大小的原因，Windows当前的限制是128GB）。当处理器在

www.TopSage.com

PAE模式下执行时，内存管理单元（MMU）将虚拟地址分成四个域，如图7.21所示。

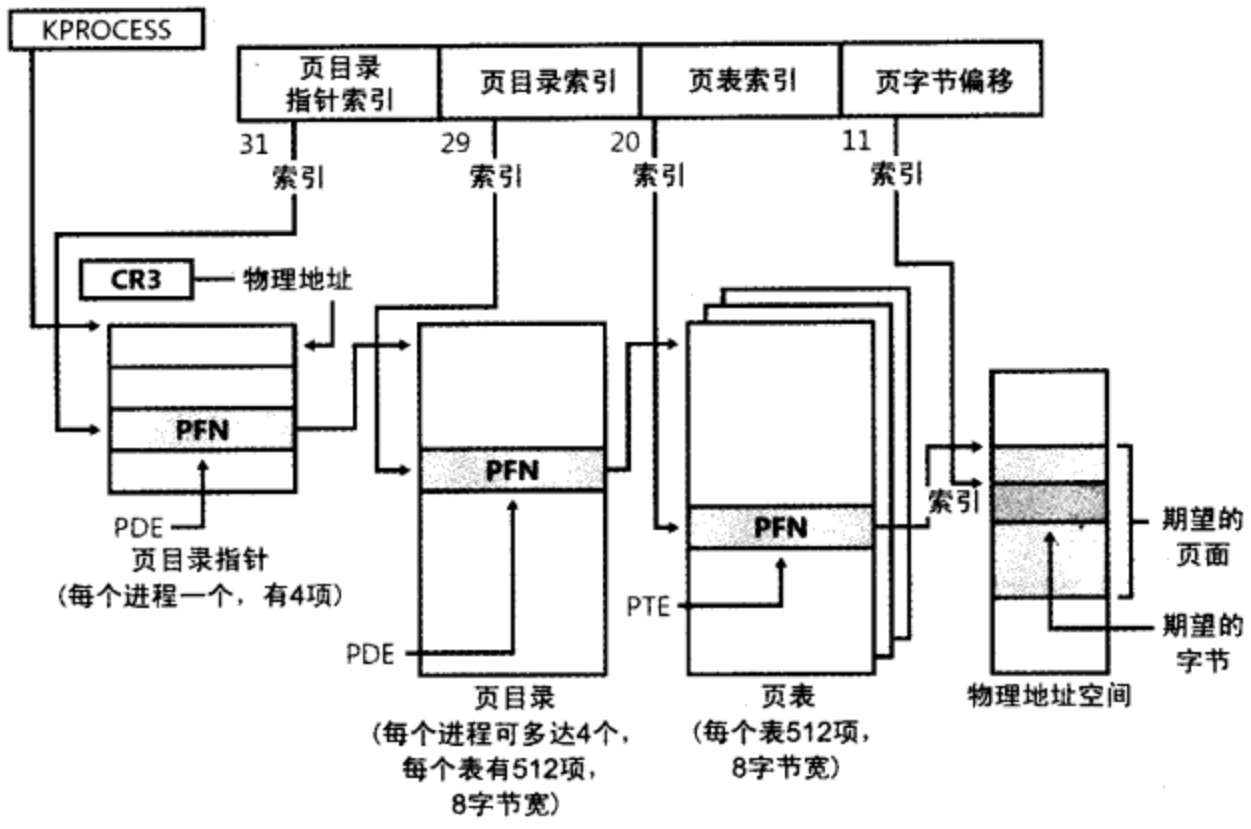


图7.21 PAE模式下的页面映射

MMU仍然实现了页目录和页表，但是在两者之上又实现了第三级：页目录指针表。在PAE模式下能访问的内存比标准转译模式下访问的内存多，这并不是因为增加了额外的一级，而是因为PDE和PTE是64位，而并非原来的32位。系统在内部用25位来表示物理地址，这使得系统有能力支持多达2^{25+12}字节，或128GB的内存。前面讲述的“地址窗口扩展”一节介绍了一种允许32位应用程序利用这么多内存的办法。然而，即使应用程序不使用这些函数，内存管理器也会通过系统缓存、备用列表和修改列表的使用，将所有这些可用的物理内存用于文件缓存数据（在本章后面“页面帧编号数据库”一节中介绍）。

正如在第2章中介绍的那样，有一个特殊版本的32位Windows内核支持PAE，称为Ntkrnlpa.exe。要想选择这一支持PAE的内核，你必须在Boot.ini中通过/PAE开关来引导系统。注意，所有的32位Windows系统上都安装了这一特殊版本的内核映像，即使小内存的Windows 2000 Professional或者Windows XP系统上也是如此。这么做是为设备驱动程序的测试提供方便。因为PAE内核呈现给设备驱动程序和其他系统代码的是64位地址，所以，即使是在小内存系统上，用/PAE来引导系统可让设备驱动程序的开发人员，测试他们开发的驱动程序中用到大地址的部分。Boot.ini中另一个相关的开关是/NOLOWMEM，它将4GB以下的内存弃置不用（假设你有至少5GB物理内存），并且将设备驱动程序重定位到此范围以上。这保证了这些驱动程

www.TopSage.com

序看到的将是大于32位的物理地址，从而使任何可能的驱动程序符号扩展错误更加易于被发现。

IA-64虚拟地址转译

IA-64的虚拟地址空间在硬件上被分成了8个区域。每个区域都可以有它自己的页表集合。Windows用到了5个区域，其中3个有页表。表7.12列出了这些区域，以及它们的用途。

表7.12 IA-64区域

区 域	用 途
0	用户代码和数据
1	会话空间代码和数据
2	未使用
3	未使用
4	Kseg3，这是一个被缓存的一对一物理内存映射。这个区域不需要页表，因为必要的TLB插入操作是直接由内存管理器来完成的
5	Kseg4，这是一个非缓存的一对一物理内存映射。只有在少数几个访问I/O资源（比如I/O端口范围）的地方，才会用到这个区域。这个区域不需要页表
6	未使用
7	内核代码和数据

64位Windows在IA-64平台上的地址转译过程用到了一个三级页表方案。每个进程有一个页目录指针结构，它包含了1024个指向页目录的指针。每个页目录包含了1024个页表指针，这些页表又指向了物理页面。图7.22显示了一个IA-64硬件PTE的格式。

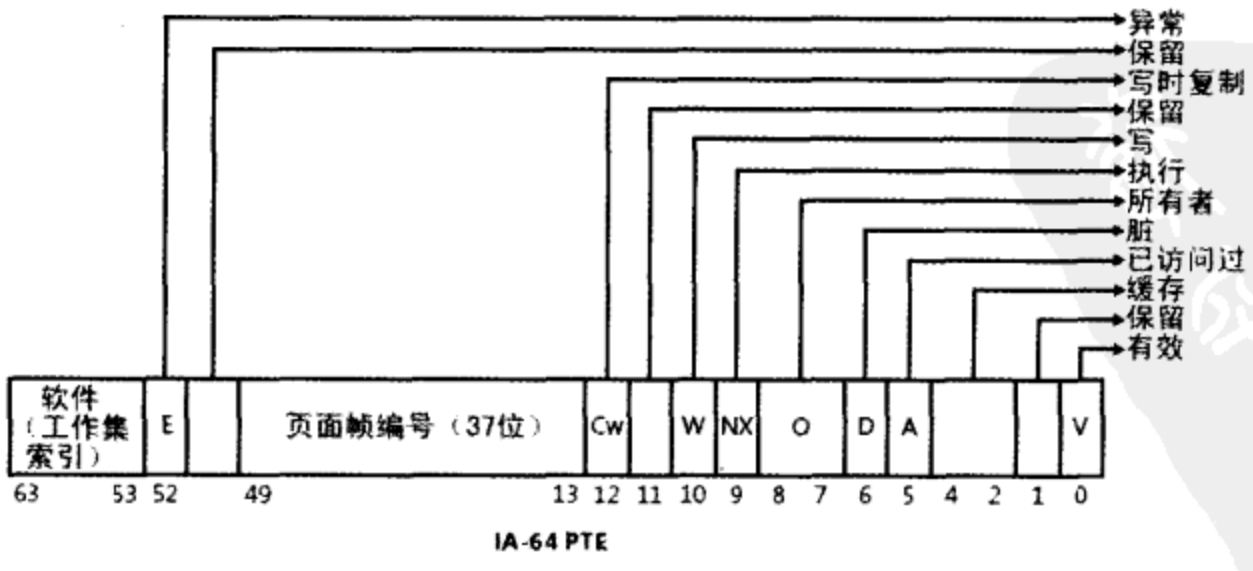


图7.22 IA-64页表项

www.TopSage.com

x64虚拟地址转译

在x64体系结构上，64位Windows使用了四级页表方案。每个进程有一个顶级的、扩展的页目录（称为**4级页映射**），它包含了512个指针，每个指针指向一个第三级结构，称为页面父目录。每个页面父目录包含512个指针，指向第二级页目录；每个页目录包含512个指针，指向单独的页表。最后，页表（每个页表包含512个页表项）指向内存中的页面。当前实现的x64体系结构限制了虚拟地址的宽度为48位。构成48位虚拟地址的各个部分如图7.23所示。这些结构之间的连接关系如图7.24所示。最后，图7.25显示了x64硬件页表项的格式。

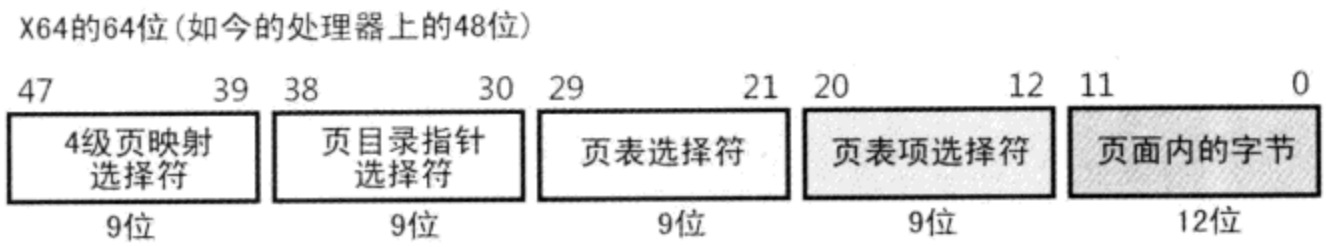


图7.23 x64虚拟地址

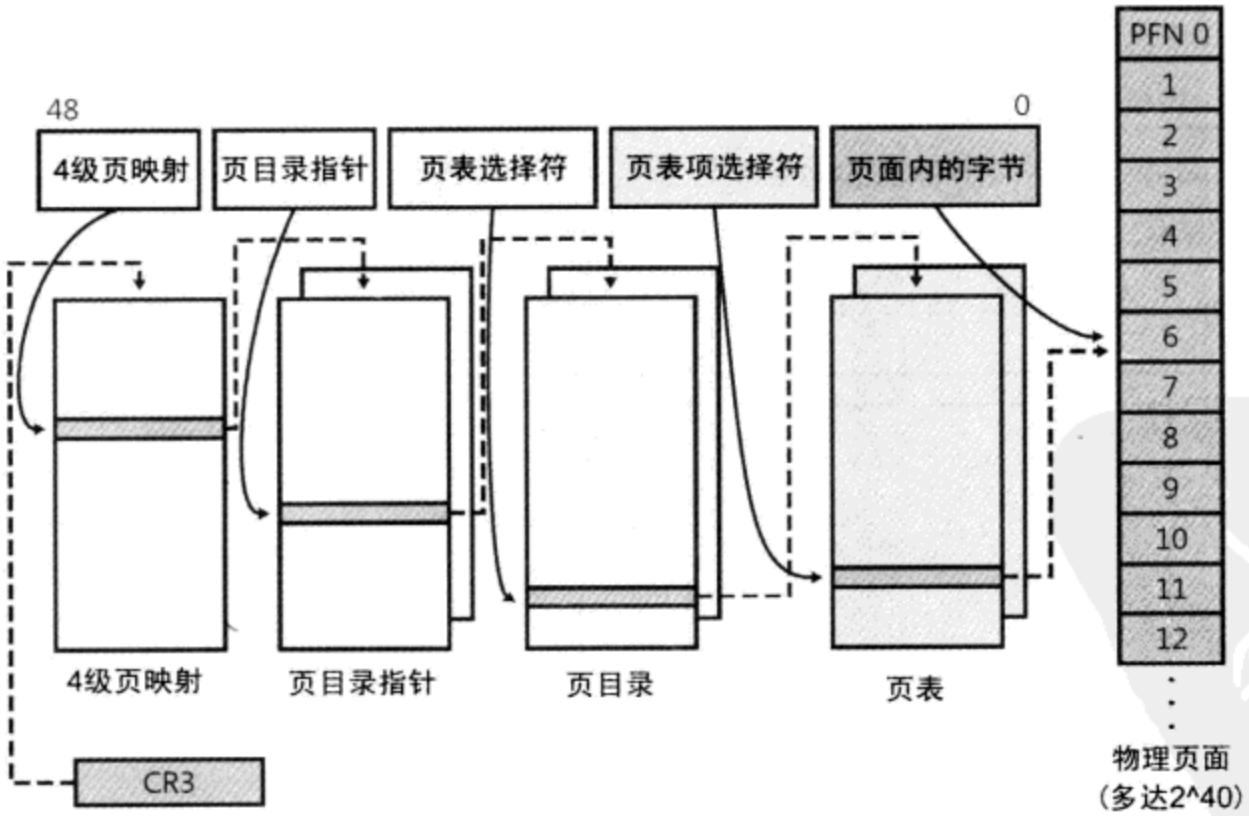


图7.24 x64地址转译结构

www.TopSage.com

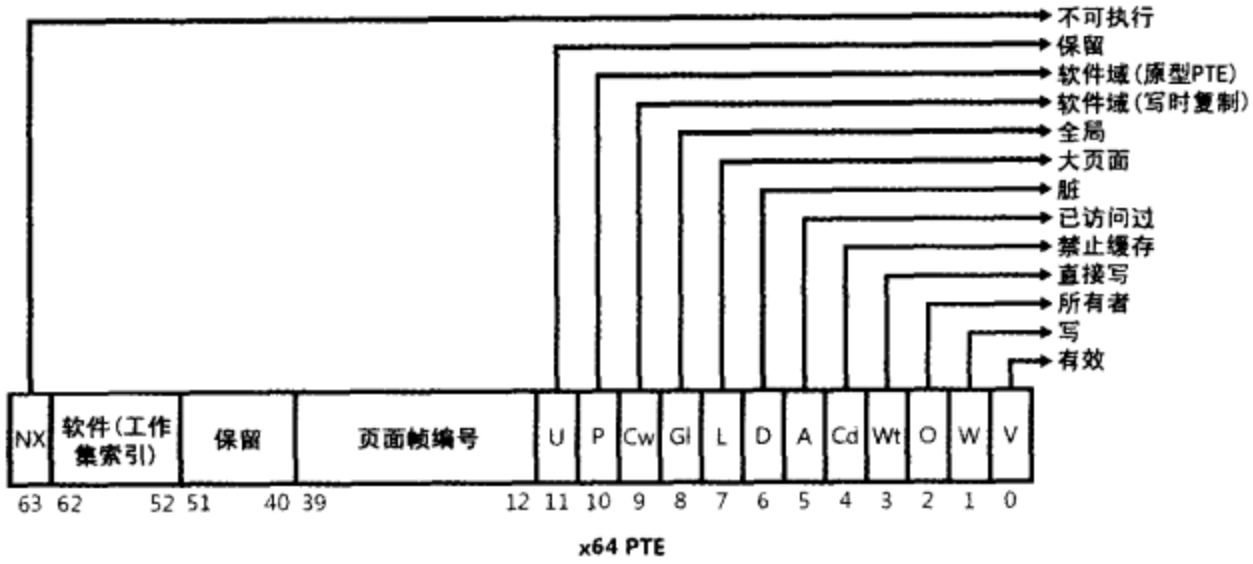


图7.25 x64硬件页表项

7.6 页面错误处理

在前面，你已经看到了当PTE有效时的地址转译解析过程。当PTE有效位被清除时，这表明，期望的目标页面出于某种原因，（当前）无法被进程所访问。这一节将讲述无效PTE的类型，以及对这些PTE的引用是如何被解析的。



> 注 本书仅仅详细介绍了32位x86 PTE的格式。64位系统的PTE包含了类似的信息，但关于它们的详细布局结构本书中没有介绍。

对一个无效页面的引用被称为**页面错误**（*page fault*）。内核陷阱处理器（在第3章的“陷阱分发”一节中介绍过）把这种错误分发给内存管理器的错误处理器（fault handler）（*MmAccessFault*）来解决。这一例程运行在引发此错误的线程的环境下，它的责任是，试图消除此错误（如果可能的话），或者引发一个适当的异常。这些错误可以是各种条件引起的，如表7.13所示。

表7.13 访问错误的种种原因

错误的原因	结 果
访问一个并非驻留在内存中，而是位于磁盘上的页面文件或者映射文件中的页面	分配一个物理页面，从磁盘上读入期望的页面，并使其进入到工作集中
访问一个位于备用列表或者修改列表中的页面	将该页面转变成进程工作集或者系统工作集
访问一个未被提交的页面（例如，在保留地址空间或者尚未分配的地址空间中）	访问违例
从用户模式下访问一个页面，但是该页面只能在内核模式下才可以被访问	访问违例

www.TopSage.com

续表

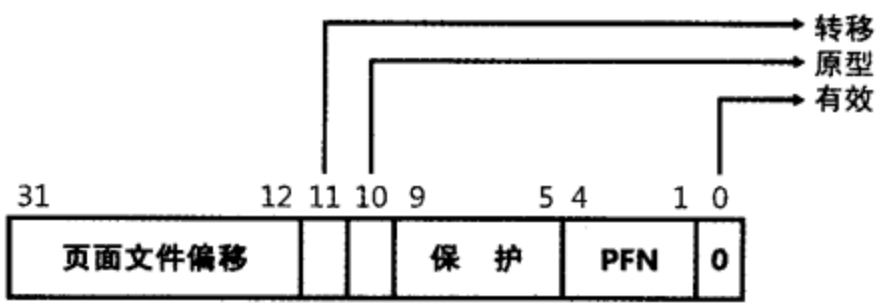
错误的原因	结 果
写一个只读页面	访问违例
访问一个要求零的页面	在进程工作集中加入一个填满零的页面
写一个守护页面	守护页面违例（如果是一个指向用户模式栈的引用，则执行自动的栈扩展）
写一个“写时复制”页面	复制一份进程私有（或者会话私有）的页面拷贝，并且在进程、会话或者系统工作集中替换掉原来的页面
引用系统空间中的一个页面，该页面是有效的，但是在进程页目录中是无效的（例如，如果在创建了进程页目录之后换页内存池被扩展了）	从主系统页目录结构中拷贝整个页目录，并解除异常
在一个多处理器系统上写一个页面，该页面是有效的，但是尚未被写过	设置PTE中的“脏”位
执行一段位于某个被标记为“不可执行”的页面中的代码	访问违例（只有在支持“不可执行”保护特性的硬件平台上才支持，而且所运行的系统必须是Windows XP Service Pack 2或者Windows Server 2003 Service Pack 1及其以后的版本）

下一小节讲述了4种基本的无效PTE，访问错误处理器（access fault handler）将会处理这些无效PTE。在此之后又介绍了一种特殊的无效PTE：原型PTE，它可被用来实现共享的页面。

无效PTE

下面的列表详细介绍了这四种无效PTE以及它们的结构。这里有些标志与表7.11中描述的硬件PTE的标志相同。

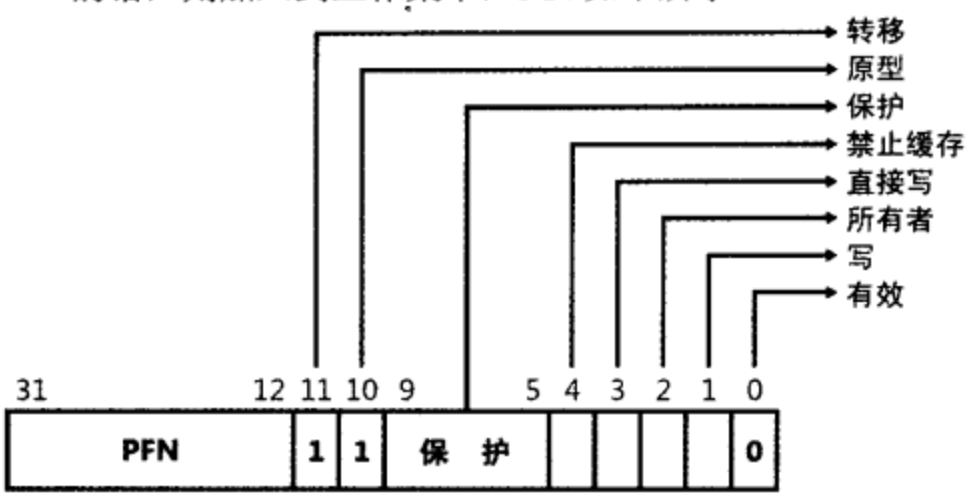
- **页面文件** 期望的页面驻留在一个页面文件中。于是激发一个页面换入（in-page）操作，此PTE如下所示：



- **要求零**（Demand zero） 期望的页面必须是一个填满零的页面。换页器（pager）检查零页面列表。如果该列表是空的，则换页器从空闲列表中取出一个页面，并且填上零。如果空闲列表也是空的，则它从备用列表中取出一个页面，并填上零。其PTE格式与上一条中显示的页面文件PTE相同，但是页面文件号和偏移都是0。

www.TopSage.com

- **转移**（Transition） 期望的页面在内存中，或者在备用列表中，或者在修改列表中，或者在已修改但不写出（modified-no-write）列表中。该页面将从此列表中删除，如果被引用的话，则加入到工作集中，PTE如下所示：



- **未知** 该PTE是零，或者页表尚未存在。在这两种情况下，此标志意味着，你应该检查虚拟地址描述符（VAD）以确定该虚拟地址是否已经被提交了。如果是的话，则建立页表来表达新提交的地址空间（参见本章后面关于VAD的讨论）。

原型PTE

如果一个页面可以在两个进程之间共享，则内存管理器依靠一种称为**原型页表项**（*prototype page table entry*, prototype PTE）的软件结构来映射这些潜在要被共享的页面。对于由页面文件支撑的内存区，当内存区对象被第一次创建的时候，同时也会创建一批原型PTE；对于映射文件，则会根据需要，在每个视图被映射的时候才创建这些原型PTE。（参见下面的注释。）这些原型PTE是本章末尾介绍的**段**（*segment*）结构的一部分。



注 在Windows 2000和Windows 2000 Service Pack 1中，内存管理器一下分配了映射整个文件所需要的所有原型页表项，即使应用程序可能在任何时刻都只是映射该文件的一小部分。因为这些结构是从一个有限的资源（换页池）中分配的，所以，若试图映射大的文件，可能会消耗光这一资源。其结果是，这限制了在任何时刻，可以使用的映射文件总量大约为200GB。

在Windows 2000 Service Pack 2及其以后，这一限制被消除了，其做法是，只有当需要创建映射文件的视图时内存管理器才分配这些原型PTE。这一变化的结果是，在任何大小的系统上，任何大小文件的备份现在都有可能做到了。而在此之前，你在一个32MB的系统上是无法备份一个500GB的文件的，因为没有足够的换页池来创建整个内存区的原型页表项。现在，由于只针对活动的视图才分配这些结构，所以，即使在小内存配置的系统上，也可以备份大的文件了。

www.TopSage.com

当进程第一次引用一个被映射到内存区对象视图中的页面时（以前提到过，只有当创建视图时才生成相应的VAD），内存管理器利用原型PTE中的信息来填充该进程页表中实际的PTE，在地址转译时使用的是这些实际的PTE。当一个共享页面被变成有效时，进程PTE和原型PTE都指向包含此数据的物理页面。为了跟踪有多少个进程PTE指向一个有效的共享页面，在PFN数据库项中有一个计数器也会随之而递增。因此，内存管理器可以确定何时一个共享页面已经不再被任何一个页表引用了，因而可以变成无效的，并且放到转移列表（transition list）中，或者写到磁盘上。

当一个共享页面被变成无效时，进程页表中的PTE被填充成一个特殊的PTE，此特殊PTE指向了描述该页面的原型PTE，如图7.26所示。

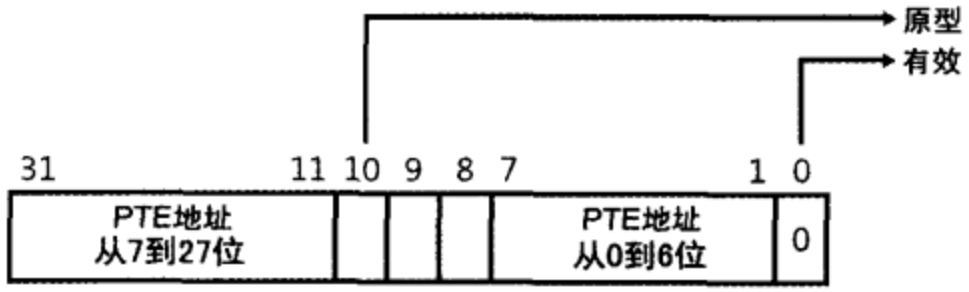


图7.26 指向原型PTE的无效PTE的结构

因此，当该页面以后被访问时，内存管理器可以使用该PTE中包含的信息来找到原型PTE，而原型PTE又描述了被引用的页面。根据原型PTE中的描述，一个共享页面可以处于以下6种不同的状态之一：

- **活动/有效** 该页面位于物理内存中，这是其他进程已经访问过它的结果；
- **转移** 期望的页面位于内存中，在备用列表或者修改列表上；
- **已修改但不写出**（modified-no-write） 期望的页面位于内存中，在已修改但不写出列表上（参见表7.20）；
- **要求零** 期望的页面应该用一个零页面来满足；
- **页面文件** 期望的页面驻留在一个页面文件中；
- **映射文件** 期望的页面驻留在一个映射文件中。

虽然这些原型PTE项的格式与前面描述的实际PTE项的格式相同，但是，这些原型PTE并不用于地址转译——它们属于页表和页面帧编号数据库之间的一层，从不直接出现在页表中。

让所有可能需要访问共享页面的地址引用指向一个原型PTE来解决页面错误，内存管理器通过这种方式就可以在无需更新每个共享此页面的进程的页表的情况下，有效地管理好共享页面。

www.TopSage.com

例如，一段共享代码或者数据页面可能要在某个时刻被换出到磁盘上。当内存管理器从磁盘上获取该页面时，它只需更新原型PTE，使其指向该页面的新物理位置，而共享此页面的每个进程的PTE仍然是相同的（“有效”位已清除，仍然指向此原型PTE）。之后，当进程引用该页面时，实际的PTE将随之得到更新。

图7.27演示了一个映射视图中的两个虚拟页面。一个是有效的，另一个是无效的。正如图中所示，第一个页面是有效的，进程PTE和原型PTE指向该页面。第二个页面在页面文件中，原型PTE包含了它的确切位置。进程PTE（和任何其他也映射了此页面的进程）指向此原型PTE。

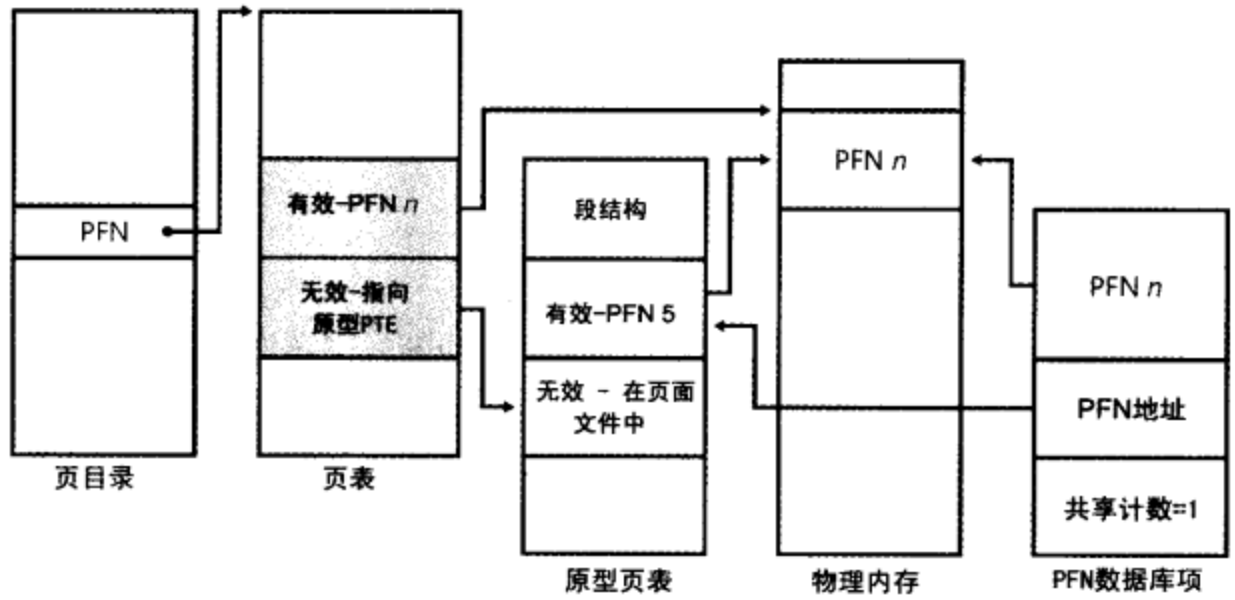


图7.27 原型页表项

页面换入I/O

当为了满足一个页面错误而必须发出一个读文件（页面文件或映射文件）操作时，页面换入I/O就会发生。而且，因为页表本身也是可换页的，所以，在处理一个页面错误时，若系统还需要加载页表的页面（该页面包含了原始被引用页面所对应的PTE或者原型PTE），则可能会招致额外的页面错误。

页面换入I/O操作是同步的，也就是说，线程等待一个事件，直至I/O完成。而且，异步过程调用（APC）的交付是不会中断这种操作的。换页器在I/O请求函数中使用一个特殊的修饰符来指明这是一个换页I/O。当换页I/O完成时，I/O系统触发一个事件，该事件唤醒换页器，以便让它继续处理页面换入过程。

当换页I/O操作正在进行时，导致页面错误的那个线程并不拥有任何关键的内存管理同步对象。在换页I/O发生的时候，该进程内的其他线程仍然可以使用虚拟内存的功能，以及处理页面错误。

www.TopSage.com

但是，以下一些有趣的条件则是换页器在I/O完成时必须认识到的：

- 同一个进程中或者不同进程中的另一个线程可以在同一个页面上产生页面错误（称为**冲突的页面错误**，下一小节讲述这种情形）；
- 页面已经被从虚拟地址空间中删掉（和重新映射）了；
- 页面上的保护域已经改变了；
- 页面错误可能是针对原型PTE的，并且映射此原型PTE的那个页面可能已经不在工作集里了。

换页器在换页I/O请求以前，在线程的内核栈中保存了足够的状态，以便来处理这些条件。因此，当换页I/O请求完成时，换页器可以检测到这些条件，如果有必要的话，它可以不将页面变成有效的，而直接解除此页面错误。若导致页面错误的指令被再次激发，则换页器再次被调用，然后根据新的状态再来评估此PTE。

冲突的页面错误

当一个页面当前正在被换入时，另一个线程或者进程也激发了该页面的页面错误，这种情形称为**冲突的页面错误**（*collided page fault*）。换页器必须以最佳的方式来检测和处理这些冲突的页面错误，因为在多线程系统中这是经常发生的。如果另一个线程或者进程在同一个页面上出错，则换页器检测到冲突的页面错误。注意到该页面正在转移过程中（in transition），以及一个读操作正在进行之中。（这些信息在PFN数据库项中。）在这种情况下，换页器发出一个等待操作，在该PFN数据库项中指定的事件上等待。此事件是由第一个为了消除该页面错误而激发I/O请求的那个线程来初始化的。

当I/O操作完成时，所有在该事件上等待的线程都满足了等待的条件。第一个获得PFN数据库锁的线程负责执行页面换入完成操作。这些操作包括：检查I/O状态以确保I/O操作已经成功完成、清除PFN数据库中的“正在读取过程中”位，以及更新PTE。

当随后的线程获取到PFN数据库锁来消除冲突的页面错误时，换页器看到“正在读取过程中”位已经清除了，它知道初始的更新已经执行过了；它检查PFN数据库项中的“页面换入错误（in-page error）”标志，以确保页面换入I/O已经成功完成。如果“页面换入错误”标志已经被置上，则该PTE不被更新，并且在导致页面错误的线程内引发一个页面换入错误异常。

页面文件

页面文件被用于存储那些已经被修改过的，虽然仍在被使用但又不得不写到磁盘上的页面（通过“已修改页面写出到磁盘”的操作）。当页面最初被提交时，页面文件的空间被保留了，但是，一直到这些页面被写出到磁盘上的时候，它们在页面文件中的位置才真正被确定下来。很重要的一点是，当私有页面被创建时，系统提交内存的限制中也扣除了这些页面。因此，性能计数器“Process: Page File Bytes”实际上是总的进程私有提交内存，这些私有提交内存可能

www.TopSage.com

都不在页面文件中，也可能有一部分在页面文件中，或者可能全部在页面文件中（实际上，它与性能计数器“Process: Private Bytes”是相同的）。

内存管理器在全局基础上，记录了私有提交内存的使用量，其术语称为**总提交量**（*commitment*），而在每个进程的基础上，则称为“**页面文件配额**（*page file quota*）”。（再说一次，此内存使用量并不代表页面文件使用量——它代表的是私有提交内存的使用量。）无论何时，若当前正在提交的虚拟地址要求新的私有物理页面，则总提交量和页面文件配额都要将其扣除。一旦全局的提交量达到限制值（物理内存和页面文件已经满了）时，再申请虚拟内存将会失败，直到有进程释放已提交的内存为止（例如，当一个进程退出时）。

当系统引导时，会话管理器进程（在第4章中已经讲述）通过检查注册表值HKLM\SYSTEM\CurrentControlSet\ Control\Session Manager\Memory Management\PagingFiles，读入要打开的页面文件的列表。这是一个多字符串注册表值，包含了每个页面文件的名称、最小尺寸值和最大尺寸值。Windows支持最多16个页面文件。在运行普通内核的x86系统上，每个页面文件最多可达4095MB。在x64系统和运行PAE内核的x86系统上，每个页面文件可以多达16TB（兆兆字节）。在IA-64系统上，每个页面文件可以是32TB。一旦页面文件被打开了，在系统运行过程中它就不能被删除了，因为System进程（在第2章中讲述）针对每个页面文件维护了一个打开的句柄。页面文件总是打开的，这一事实也说明了为什么内置的碎片整理工具无法在系统运行的时候整理页面文件的碎片。要想整理页面文件的碎片，你可以使用*www.sysinternals.com*的免费的Pagedefrag工具。它使用了与其他的第三方碎片整理工具相同的办法——它在引导过程的早期，在会话管理器打开页面文件之前运行碎片整理过程。

因为页面文件包含了进程虚拟内存和内核虚拟内存的一部分内容，所以，为了安全起见，系统可以被配置成：当系统停机的时候清除掉页面文件的内容。要想打开这一特性，你可以将注册表值HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ClearPageFileAtShutdown设置成1。否则的话，在停机以后，页面文件中包含了当系统在运行时被换出去的那些页面的内容。之后，如果有人能够从物理上访问该机器的话，他就有可能访问到这些数据。

如果系统注册表中没有指定页面文件的话，则Windows 2000在引导分区上创建一个默认的20-MB页面文件。Windows XP和Windows Server 2003并不创建这一临时页面文件，这意味着系统的虚拟内存提交限制完全基于可用内存的数量。在Windows XP和Windows Server 2003中，如果最小和最大的页面文件大小值都是0，则表明这是一个由系统管理的页面文件，系统会选择页面文件的大小，如表7.14所示。

表7.14 默认的页面文件大小

系统内存大小	最小的页面文件大小值	最大的页面文件大小值
< 1 GB	1.5 * RAM	3 * RAM
>= 1 GB	1 * RAM	3 * RAM

www.TopSage.com



实验：查看系统页面文件

要想查看页面文件的列表，请检查注册表中的HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles。这里包含了页面文件配置信息，通过控制面板中的System工具可以修改此设置。在Windows 2000中，在Advanced视图上点击Performance Options按钮，然后点击Change按钮。在Windows XP和Windows Server 2003中，点击Advanced标签，点击Performance区的Settings按钮，最后，点击Virtual Memory区中的Change按钮。

为了加入一个新的页面文件，控制面板使用Ntdll.dll中定义的*NtCreatePagingFile*系统服务（仅内部使用）。页面文件总是被创建成非压缩的文件，即使它们所在的目录是压缩的。为了避免新的页面文件被删除掉，该文件的一个句柄被复制到System进程中，因而，当创建此新页面文件的进程关闭了指向该文件的句柄时，另一个进程仍然可以打开此页面文件。

表7.15中列出的性能计数器使得你可以查看系统范围内的或者每个页面文件基础上的私有提交内存的使用量。这里没有办法确定一个进程的私有提交内存中有多少驻留在物理内存中，有多少被换出到页面文件中。

表7.15 提交内存和页面文件的性能计数器

性能计数器	说 明
Memory: Committed Bytes	已经被提交的虚拟内存（不是保留的）的字节数。此数并不一定代表页面文件的使用量，因为它包含了物理内存中从未被换出过的私有提交页面。当然，如果一个进程完全是非驻留的，则它代表所使用的页面文件空间的数量
Memory: Commit Limit	无需扩展页面文件就可以提交的虚拟内存的字节数；如果页面文件可以被扩展，则这一限制是软性的
Paging File: % Usage	页面文件中已提交部分的百分比
Paging File: % Usage Peak	页面文件中已提交部分的最高百分比

注意，这些计数器可以帮助你选择页面文件的大小。尽管通常的做法是按照内存的数量来确定页面文件的大小，但这种做法并没有意义，因为你拥有的内存越多，你需要换出的数据可能越少。如果你要根据你的系统自从引导以来已经运行的应用程序的情况，来确定你的系统到底需要多大的页面文件空间，那么你可以检查尖峰提交量（显示在任务管理器的性能页面上的Commit Charge区中；在进程管理器的System Information窗口中也有显示）。此数值就是系统自引导以来曾经在某个时刻，所需要的页面文件空间的尖峰数量（如果系统要把所有的私有提交虚拟内存都换出去的话，当然这种情况很少发生）。

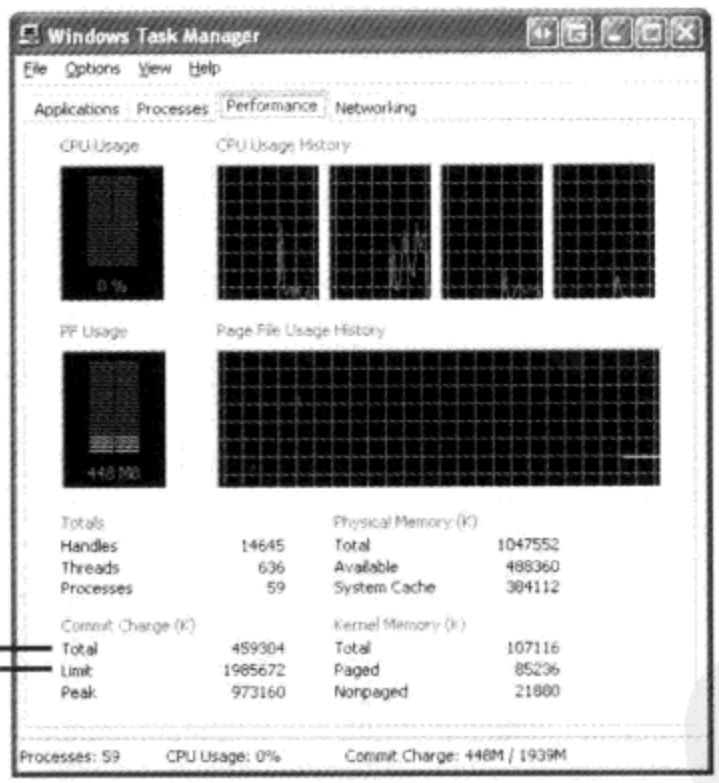
www.TopSage.com

如果你的系统上页面文件太大的话，则系统或多或少不会用到它。换句话说，增大页面文件并不会改变系统的性能，这仅仅意味着系统可以有更多的非共享的提交虚拟内存。如果页面文件相对于你当前正在运行的应用程序组合来说太小的话，你可能会得到“system running low on virtual memory（系统的虚拟内存太低）”错误消息。在这种情况下，首先要检查是否有进程存在内存泄漏。做法是，检查进程的私有字节计数（在任务管理器的进程页面中查找“VM Size[虚拟内存大小]”一栏）。如果看起来没有进程在泄漏，则检查系统换页内存池的大小——如果某个设备驱动程序正在泄漏换页内存池，那么这也可能说明了错误的原因（关于如何诊断一个内存池泄漏，请参见“系统内存池”一节中的“诊断内存池泄漏”实验）。



实验：利用任务管理器来查看页面文件使用量

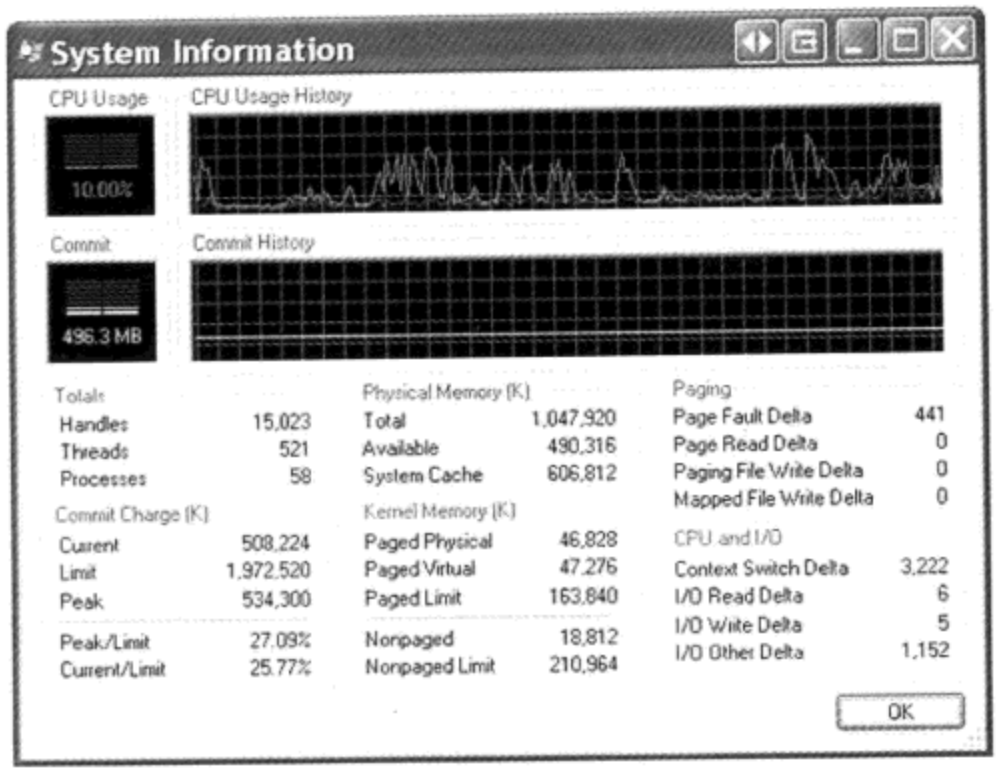
你可以利用任务管理器来查看已提交内存的使用量。做法是，点击Performance页面，你将会看到如下的有关页面文件的计数器：



注意，“Mem Usage”一栏（在Windows XP和Windows Server 2003下称为“PF Usage”）实际上是系统提交总量。该数值代表了**潜在的**页面文件使用量，而不是实际的页面文件使用量。它是指，如果系统中所有的私有提交虚拟内存不得不一次性全部被换出去的话，将需要用到多少页面文件空间。

进程管理器的System Information窗口显示了一些有关系统提交内存量的额外信息，包括尖峰使用量相比限制值的百分比，以及当前使用量相比限制值的百分比（见屏幕左下角）。

www.TopSage.com



7.7 虚拟地址描述符

内存管理器使用一个按需换页（demand-paging）的算法来计算何时将页面加载到内存中，它要等到有一个线程引用一个地址并且招致一个页面错误时，才从磁盘中获取该页面的数据。如同写时复制一样，按需换页也是一种“延迟计算（*lazy evaluation*）”——等待到真正需要的时候才执行一项任务。

内存管理器不仅在加载页面到内存中时使用延迟计算技术，而且也使用这种技术来构建那些用于描述新页面的页表。例如，当一个线程通过*VirtualAlloc*来提交一大块虚拟内存区域的时候，内存管理器可以立即构建起为访问这整块内存范围而需要的页表。但如果这范围中有一部分从来不会被访问到，则又会怎么样呢？为整个范围创建页表是一种浪费。与此相反，内存管理器一直等到有一个线程引发了一个页面错误时才创建一个页表，然后再为该页面创建一个页表。对于那些保留和/或提交了大量的内存但只是零星地访问这些内存的进程来说，这种方法可以大大地提高性能。

采用了延迟计算的算法以后，即使分配大块内存也是一个快速的操作。当一个线程申请内存时，内存管理器必须用一段地址范围作为响应，以供线程使用。为了做到这一点，内存管理器维护了另一组数据结构，以跟踪和记录在进程的地址空间中哪些虚拟地址已经被保留了，哪些虚拟地址尚未被保留。这些数据结构称为**虚拟地址描述符**（VAD，*virtual address descriptor*）。对于每个进程，内存管理器维护了一组VAD，由它们来描述该进程地址空间的状态。这些VAD被组织成一棵自平衡的二叉树，以便使查找过程非常高效。Windows Server 2003实现了一个AVL树算法（根据其发明者的名字Adelson-Velskii和Landis来命名），该算法更好地平衡了VAD树，结果是，搜索一个虚拟地址对应的VAD的平均次数更少了。图7.28显示了一棵VAD树的

www.TopSage.com

结构图。

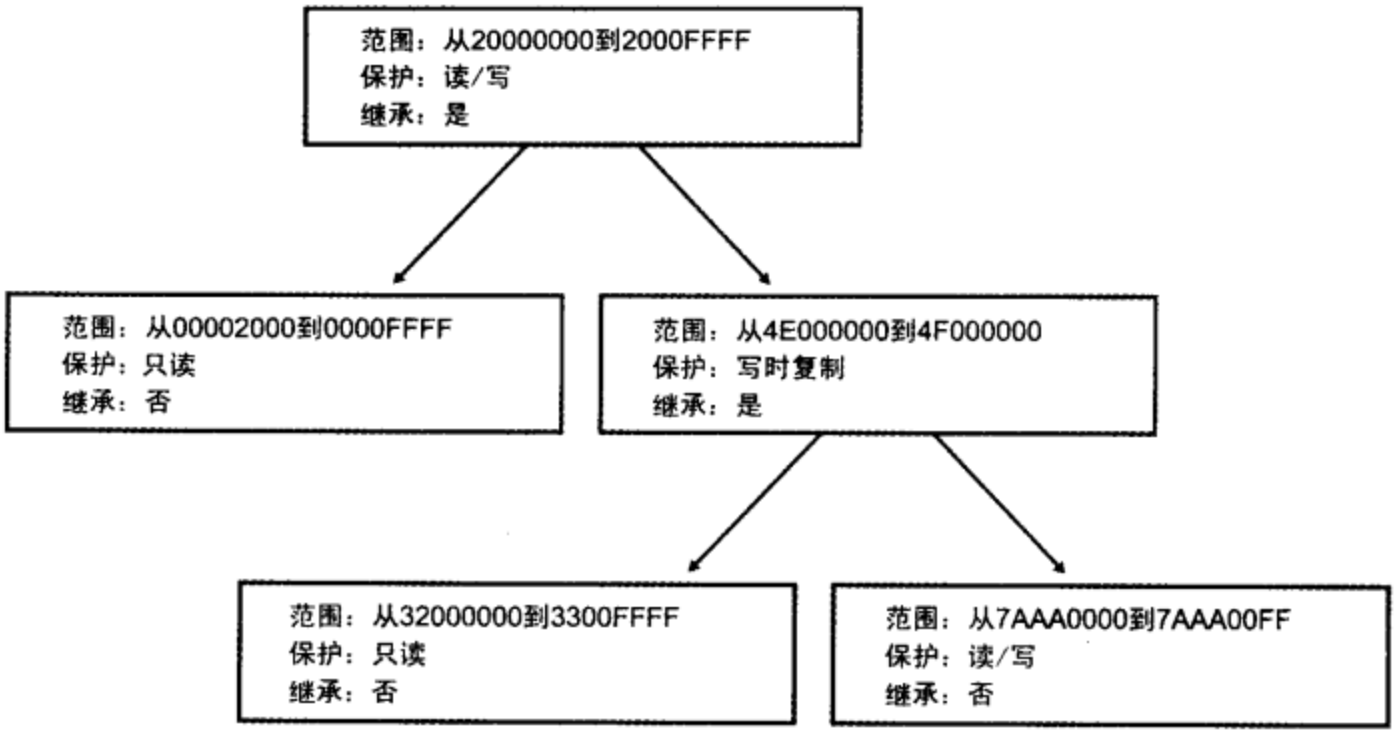


图7.28 虚拟地址描述符

当一个进程保留地址空间或者映射一个内存区的视图时，内存管理器创建一个VAD来存储此次内存请求所提供的任何信息，比如被保留的地址的范围、该范围是共享的还是私有的、子进程是否可以继承此段范围的内容，以及应用在此段范围内的页面上的保护属性。

当一个线程第一次访问一个地址时，内存管理器必须为包含该地址的页面创建一个PTE。为了做到这一点，它找到负责此地址的那个VAD（即该VAD的地址范围包含了此地址），并且利用找到的信息来填充该PTE。如果该地址落在VAD所覆盖的地址范围以外，或者在一个已保留但尚未提交的地址范围之中，那么，内存管理器知道该线程在使用此内存以前还没有申请该内存，因此产生一个访问违例。



实验：查看虚拟地址描述符

你可以使用内核调试器的*!vad*命令来查看一个进程的VAD。首先通过*!process*命令找到VAD树的根地址。然后在*!vad*命令中指定此地址，例如，在下面的例子中，查看Notepad.exe进程的VAD树：

```
kd> !process 2a0 1
Searching for Process with Cid == 2a0
PROCESS 8614d030 SessionId: 0 Cid: 02a0 Peb: 7ffdf000 ParentCid: 0554
    DirBase: 00d93000  ObjectTable: 81bc47c8  TableSize:  41.
    Image: notepad.exe
    VadRoot 8118d868 Clone 0 Private 252. Modified 0. Locked 0.
```


www.TopSage.com

```
kd> !vad 8118d868
VAD     level      start      end    commit
84df4148 ( 2)         10       10         1 Private      READWRITE
850cdbe8 ( 3)         20       20         1 Private      READWRITE
810b0ee8 ( 1)         30       6f         7 Private      READWRITE
8109d308 ( 3)         70      16f        32 Private      READWRITE
810e9a28 ( 2)        170      17f         0 Mapped       READWRITE
84aedfc8 ( 3)        180      195         0 Mapped       READONLY
8118d868 ( 0)        1a0      1ce         0 Mapped       READONLY
81190a08 ( 4)        1d0      210         0 Mapped       READONLY
85c7b928 ( 3)        220      223         0 Mapped       READONLY
86253a08 ( 4)        230      2f7         0 Mapped       EXECUTE_READ
810aab48 ( 2)        300      342         0 Mapped       READONLY
80db5448 ( 5)        350      64f         0 Mapped       EXECUTE_READ

Total VADs:    49  average level:    6  maximum depth: 13
```

7.8 内存区对象

正如在本章前面的关于共享内存一节中所介绍的那样，**内存区对象**（*section object*）代表了两个或者多个进程可以共享的一块内存，Windows子系统将其称为**文件映射对象**（*file mapping object*）。内存区对象可以被映射到页面文件中，或者磁盘上的另一个文件中。

Windows执行体利用内存区对象把可执行映像加载到内存中，缓存管理器使用内存区对象来访问缓存文件中的数据（关于缓存管理器如何使用内存区对象的更多信息，请参见第11章）。你也可以利用内存区对象把一个文件映射到进程地址空间中。然后，通过“先映射内存区对象的不同视图，再读/写内存”（一种被称为**映射文件I/O**的行为）的方式，而不是读/写文件的方式，你就可以将该文件当作大数组来访问。当该程序访问一个无效页面（一个不在物理内存中的页面）时，页面错误就会发生，内存管理器自动把该页面从映射文件读入到内存中。如果此应用程序修改了该页面，那么，内存管理器在它的正常换页操作过程中，把所做的修改写回到文件中，或者应用程序利用Windows的*FlushViewOfFile*函数刷新一个视图。

内存区对象，如同其他的对象一样，也是由对象管理器来分配和释放的。对象管理器创建并初始化一个对象头（它使用对象头来管理这些对象）；而内存管理器定义内存区对象的对象体。内存管理器也实现了一些服务，供用户模式的线程来调用，用户模式线程通过调用这些服务来获取和改变内存区对象体中存储的属性。图7.29显示了一个内存区对象的结构。

www.TopSage.com

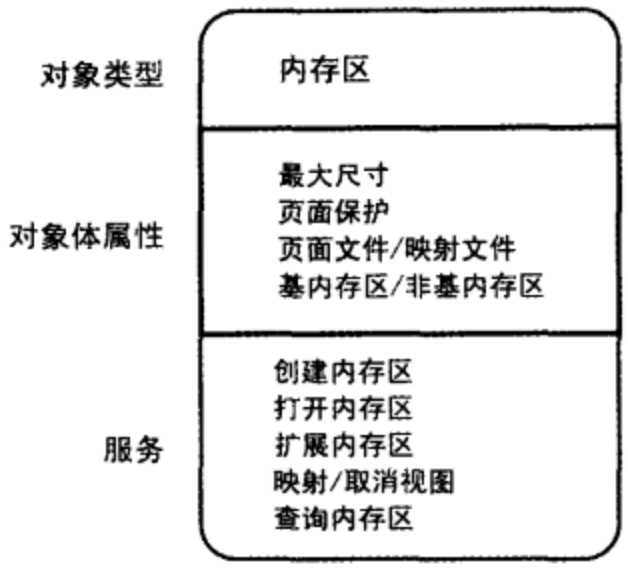


图7.29 内存区对象

表7.16概述了内存区对象中存储的特有属性。

表7.16 内存区对象体属性

属 性	用 途
最大尺寸	内存区对象可以增长到达的最大尺寸（以字节为单位）；如果映射一个文件，则最大尺寸为该文件的大小
页面保护	以页面为基础的内存保护属性，当创建一个内存区时，页面保护属性被分配给该内存区中的所有页面
页面文件/映射文件	指明了该内存区是否被创建为空（由页面文件支撑的——如早先所说明的，只有当页面需要被写出到磁盘上时，由页面文件支撑的内存区才使用页面文件资源）或者加载一个文件进来（由映射文件支撑的）
基内存区/非基内存区	指明了一个内存区是一个基内存区还是一个非基内存区。所谓基内存区，是指它在所有共享该内存区的进程中都在同样的虚拟地址上；而非基内存区，则是指在不同的进程中有不同的虚拟地址



实验：查看内存区对象

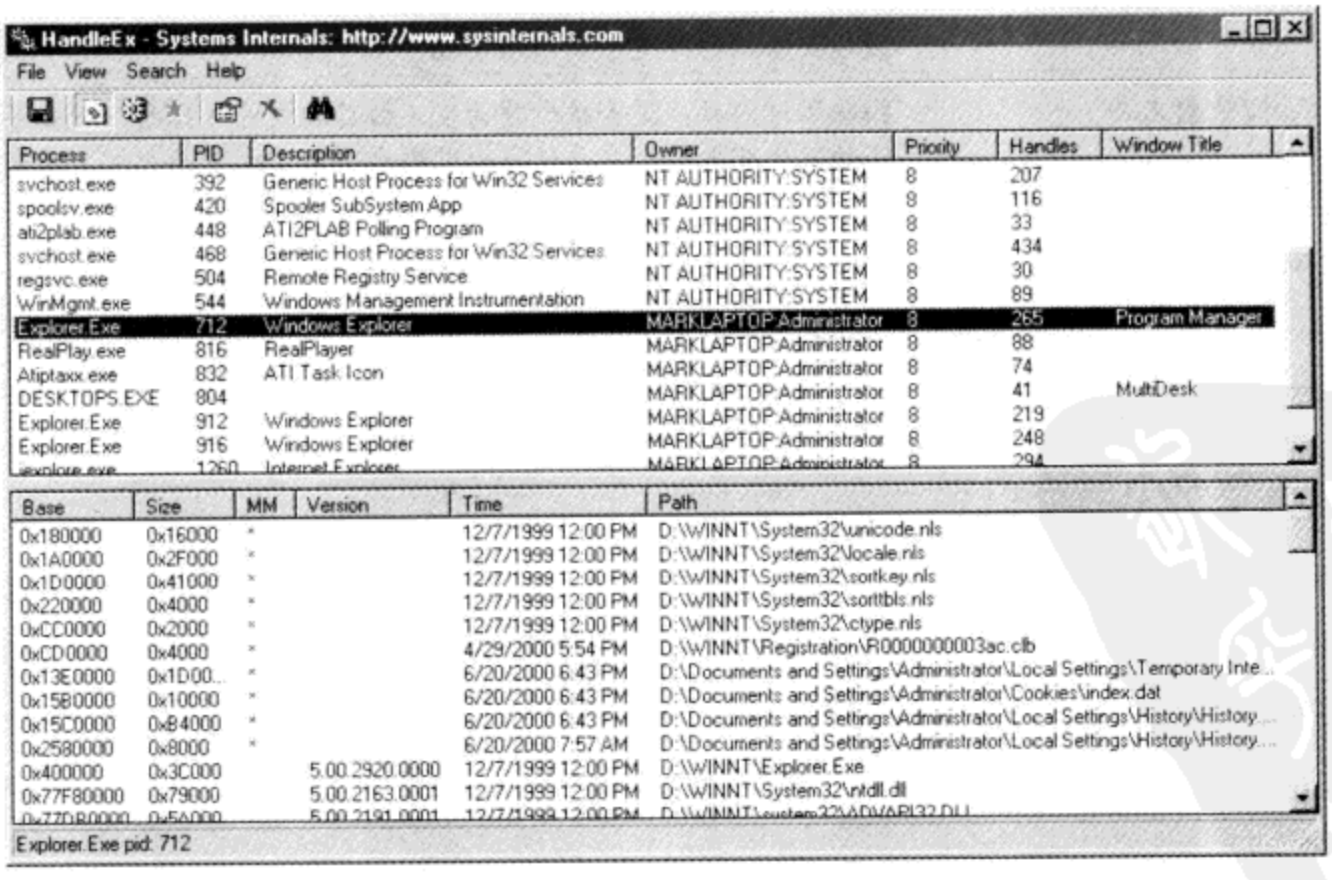
利用对象查看器（从*www.sysinternals.com*下载的Winobj.exe，或者Platform SDK中的Winobj.exe），你可以看到那些有全局名称的内存区的列表。你也可以利用在第3章的“对象管理器”一节中介绍的、可以列出已打开句柄表的任一工具，来列出指向这些内存区对象的已打开句柄（正如在第3章中解释的那样，这些名称存储在对象管理器目录\BaseNamedObjects中）。

www.TopSage.com

利用进程管理器或者Handles.exe（均来自*www.sysinternals.com*），或者Windows资源工具箱中的Oh.exe（Open Handles）工具，你可以列出指向内存区对象的已打开句柄。例如，下面的命令显示了所有指向section类型对象的已打开句柄，无论这些section类型的对象是否有名称。（只有当其他的进程需要通过名称来打开一个内存区对象时，该内存区才必须有一个名称。）

```
c:\> oh -t section -a
00000008 System          Section  0070
0000008C smss.exe        Section  0004
000000A4 csrss.exe       Section  0004
000000A4 csrss.exe       Section  0024
000000A4 csrss.exe       Section  0038
000000A4 csrss.exe       Section  0040 \NLS\NlsSectionUnicode
000000A4 csrss.exe       Section  0044 \NLS\NlsSectionLocale
000000A4 csrss.exe       Section  0048 \NLS\NlsSectionCType
000000A4 csrss.exe       Section  004c \NLS\NlsSectionSortkey
000000A4 csrss.exe       Section  0050 \NLS\NlsSectionSortTbls
000000A0 winlogon.exe    Section  0004
000000A0 winlogon.exe    Section  0034
000000A0 winlogon.exe    Section  0168 \BaseNamedObjects\mmGlobalPnpInfo
§
```

你也可以使用*www.sysinternals.com*的进程管理器来查看映射文件。从View菜单的Lower Pane View项中选择DLLs。在MM栏中被标记为一个星号的文件都是映射文件（而不是被映像加载器当作模块加载进来的DLL和其他文件）。下面是一个例子：



www.TopSage.com

内存管理器维护的用于描述内存区的数据结构如图7.30所示。这些数据结构保证了从映射文件中读取的数据总是一致的，而不管什么样的访问类型（打开文件、映射文件，等等）。

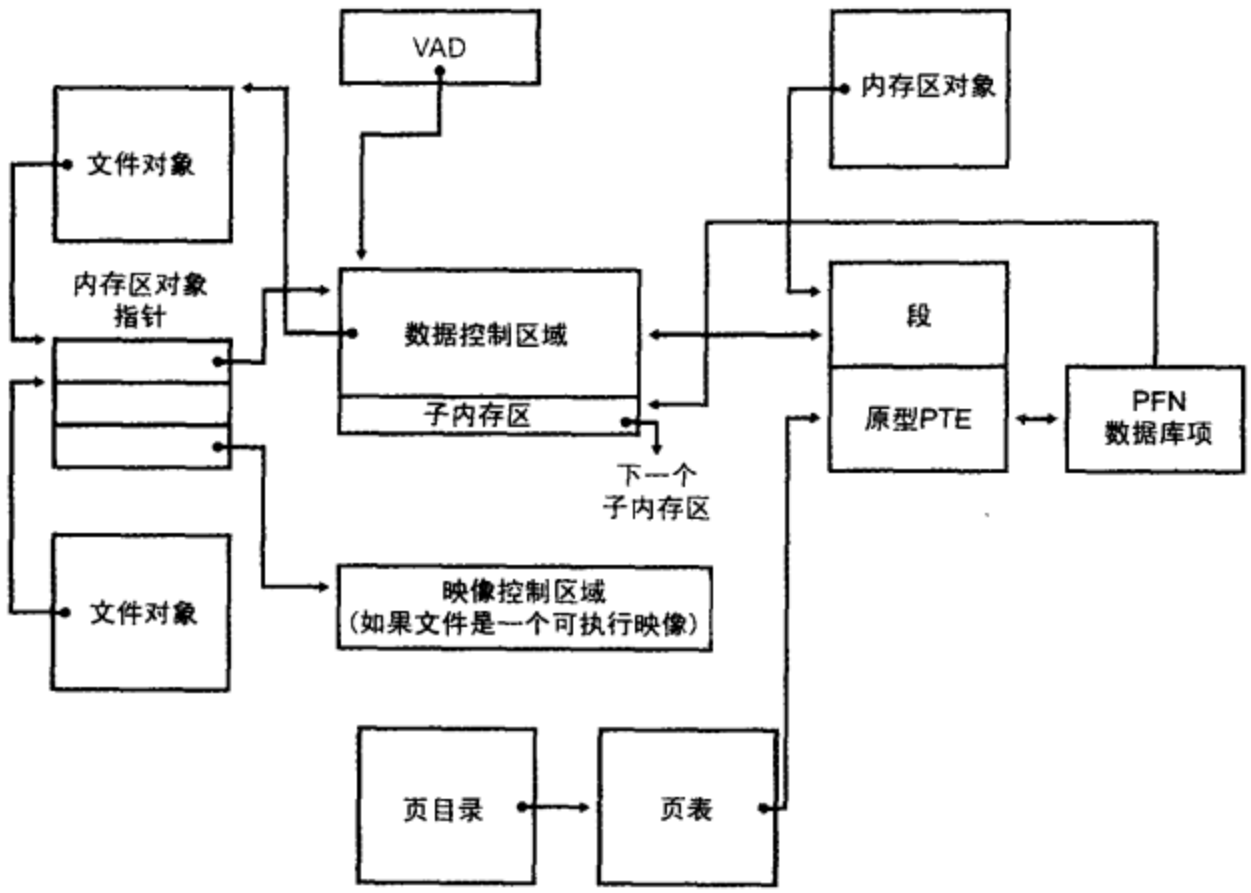


图7.30 内存区的内部结构

对于每个打开的文件（通过一个文件对象来表示），都有一个**内存区对象指针**（*section object pointer*）的结构。此结构对于维护各种类型的文件访问的数据一致性，以及为文件提供缓存能力是非常关键的。内存区对象指针结构指向一个或者两个**控制区域**（*control area*）。第一个控制区域的用途是，当该文件被当作一个数据文件来访问时，该控制区域被用于映射此文件；另一个控制区域的用途是，当该文件被当作可执行映像来运行时，它被用于映射此文件。

控制区域又依次指向**子内存区**结构，这些子内存区结构描述了该文件的每个内存区的映射信息（只读的，读写的，写时复制的，等等）。控制区域也指向一个在换页内存池中分配的**段**（*segment*）结构，而此段结构又指向了一些原型PTE，通过这些原型PTE，可映射到该内存区对象所映射的实际页面。

尽管Windows保证了任何一个访问（读或者写）文件的进程将总是看到同样的、一致的数据，但是，在有一种情况下，一个文件的页面可以有两份拷贝驻留在物理内存中（但即使在这种情况下，所有访问该文件的线程都会得到最新的拷贝，所以保持了数据一致性）。当一个映像文件先被当作一个数据文件来访问（被读取或者写入），然后又被当作一个可执行映像来运

www.TopSage.com

行（例如，当链接一个映像，然后又运行该映像时——链接器打开该文件进行数据访问，然后当映像被运行时，映像加载器将它映射为一个可执行映像）时，这种页面重复就会发生。在内部，则发生了以下的动作。

1. 如果该可执行文件是利用文件映射API（或者缓存管理器）来创建的，则创建一个数据控制区域，来表示该映像文件中被读取或者写入的数据页面。
2. 当该映像被运行起来，并且相应的内存区对象被创建出来以便映射此可执行映像时，内存管理器找到此映像文件的内存区对象指针（指向一个数据控制区域），并刷新该内存区。这一步是必要的，因为这样做可以保证，在通过映像控制区域来访问该映像以前，任何已修改的页面都已经被写到磁盘上了。
3. 然后，内存管理器为映像文件创建一个控制区域。
4. 随着该映像开始执行，它的（只读）页面通过页面错误处理，从映像文件读入到内存中。

因为通过数据控制区域映射的页面可能仍然驻留在内存中（在备用列表上），所以，这是一种“同一份数据的两份拷贝分别位于内存中两个不同的页面中”的情形。然而，这种页面重复并不会导致数据一致性问题，因为正如前面所提到的，数据控制区域已经被刷新到磁盘上了，所以，从映像中读入的页面已经被更新了（这些页面永远不会被写回到磁盘上）。



实验：查看控制区域

为了找到一个文件的控制区域结构的地址，你必须首先要找到相应的文件对象的地址。通过内核调试器你可以得到该地址，做法是，利用*!handle*命令将进程句柄表转储出来，并观察文件对象的对象地址。虽然内核调试器的*!file*命令显示了一个文件对象的基本信息，但是，它并不显示指向内存区对象指针结构的指针。然后，使用*dt*命令，格式化此文件对象，以便得到内存区对象指针结构的地址。内存区对象指针结构是由三个指针构成的：一个指针指向数据控制区域，另一个指针指向共享缓存表（在第11章中解释），还有一个指针指向映像控制区域。从内存区对象指针结构中，你可以得到该文件的控制区域的地址（如果存在的话），并且将该地址传给*!ca*命令。

www.TopSage.com

例如，如果你打开一个PowerPoint文件，并且利用*!handle*来显示该进程的句柄表，那么，你将会找到一个指向此PowerPoint文件的已打开句柄，如下所示。（有关如何使用*!handle*命令的信息，请参见第3章中的“对象管理器”一节。）

```
lkd> !handle 1 f 5c4 file
.
.
0284: Object: 8645c038  GrantedAccess: 00120089
Object: 8645c038  Type: (867ddca0) File
    ObjectHeader: 8645c020
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \slides\ntint\new\
3-systemarchitecture.ppt {HarddiskVolume1}
```

取出该文件对象的地址（8645c038），利用*dt*命令对其进行格式化，结果如下：

```
lkd> dt nt!_file_object 8645c038
nt!_FILE_OBJECT
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x86742e30
   +0x008 Vpb              : 0x867a3090
   +0x00c FsContext        : 0xe2786530
   +0x010 FsContext2       : 0xe309a378
   +0x014 SectionObjectPointer : 0x85512fec
```

然后，取出内存区对象指针结构的地址（0x85512fec），同样利用*dt*命令对其进行格式化，结果如下：

```
lkd> dt nt!_section_object_pointers 0x85512fec
nt!_SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : 0x8564f8a0
   +0x004 SharedCacheMap    : (null)
   +0x008 ImageSectionObject : (null)
```

最后，使用*!ca*以及上述地址信息显示控制区域：

```
lkd> !ca 0x8564f8a0
ControlArea @8564f8a0
  Segment:    e3817528    Flink              0   Blink              0
  Section Ref        1    Pfn Ref          25c   Mapped Views       1
  User Ref           2    WaitForDel         0   Flush Count        0
  File Object 85774580    ModWriteCount      0   System Views       0
  Flags (9008080) File WasPurged HadUserReference Accessed
  File: \slides\ntint\new\3-systemarchitecture.ppt
```


www.TopSage.com

另一项技术是，通过*!memusage*命令来显示所有控制区域的列表。以下是从该命令的输出中节选出来的：

```
kd> !memusage
 loading PFN database
loading (99% complete)
               Zeroed:      9 (    36 kb)
                 Free:      0 (     0 kb)
              Standby:   2103 (  8412 kb)
             Modified:    300 (  1200 kb)
        ModifiedNoWrite:    1 (     4 kb)
          Active/Valid:  30318 (121272 kb)
           Transition:      3 (    12 kb)
              Unknown:      0 (     0 kb)
                TOTAL:  32734 (130936 kb)
  Building kernel map
  Finished building kernel map
  Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables  name
8119b608  3000  528     0     0     0     0  mapped_file( WINWORD.EXE )
849e7c68    96    0     0     0     0     0    No Name for File
8109c388    24    0     0     0     0     0  mapped_file( DEVDTG.PKG )
81402488   236    0     0     0     0     0    No Name for File
80fba0a8   268    0     0     0     0     0  mapped_file( kernel32.pdb )
810ab168  1504  380     0     0     0     0  mapped_file( OUTLLIB.DLL )
81126a08     0  276     0     0     0     0  mapped_file( H )
81112d28   656    0     0     0     0     0    No Name for File
```

Control一栏指向了描述此映射文件的控制区域结构。通过内核调试器的*!ca*命令，你可以显示控制区域、段和子内存区。例如，在同样这个例子中，为了转储出映射文件Winword.exe的控制区域，你可以输入*!ca*命令，后面跟着该映射文件的Control数值，如下所示：

```
kd> !ca 8119b608

ControlArea @8119b608
  Segment:    e2c28000  Flink              0  Blink:             0
  Section Ref        1  Pfn Ref          372  Mapped Views:      1
  User Ref           2  Subsections        6  Flush Count:       0
  File Object 8213fb98  ModWriteCount      0  System Views:      0
  WaitForDel         0  Paged Usage     3000  NonPaged Usage   100
  Flags (90000a0) Image File HadUserReference Accessed

   File: \Program Files\Microsoft Office\Office\WINWORD.EXE

Segment @ e2c28000:
   Base address        0  Total Ptes       86a  NonExtendPtes:       86a
   Image commit       d7  ControlArea 8119b608  SizeOfSegment: 86a000
   Image Base          0  Committed          0  PTE Template:  919b6c38
   Based Addr   30000000  ProtoPtes   e2c28038  Image Info:    e2c2a1e4

Subsection 1. @ 8119b640
   ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 8
```


www.TopSage.com

```
  Base Pte     e2c28038  Ptes In subsect        1 Unused Ptes          0
   Flags             15  Sector Offset          0 Protection           1
    ReadOnly CopyOnWrite

Subsection 2. @ 8119b660
   ControlArea: 8119b608  Starting Sector 10 Number Of Sectors 3c00
   Base Pte     e2c2803c  Ptes In subsect      780 Unused Ptes          0
   Flags             35  Sector Offset          0 Protection           3
    ReadOnly CopyOnWrite

Subsection 3. @ 8119b680
   ControlArea: 8119b608  Starting Sector 3C10 Number Of Sectors 5d8
   Base Pte     e2c29e3c  Ptes In subsect       c1 Unused Ptes          0
   Flags             55  Sector Offset          0 Protection           5
    ReadOnly CopyOnWrite

Subsection 4. @ 8119b6a0
   ControlArea: 8119b608  Starting Sector 41E8 Number Of Sectors a8
   Base Pte     e2c2a140  Ptes In subsect       15 Unused Ptes          0
   Flags             55  Sector Offset          0 Protection           5
    ReadOnly CopyOnWrite

Subsection 5. @ 8119b6c0
   ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 0
   Base Pte     e2c2a194  Ptes In subsect        1 Unused Ptes          0
   Flags             55  Sector Offset          0 Protection           5
    ReadOnly CopyOnWrite
 Subsection 6. @ 8119b6e0
   ControlArea: 8119b608  Starting Sector 4290 Number Of Sectors 90
   Base Pte     e2c2a198  Ptes In subsect       12 Unused Ptes          0
   Flags             15  Sector Offset          0 Protection           1
    ReadOnly CopyOnWrite
```

7.9 工作集

在以上几节中，我们的注意力集中在Windows进程的虚拟视图上——页表、PTE和VAD。在本章的剩余部分中，我们将解释Windows如何在物理内存中维护了其虚拟地址空间的一个子集。

正如你已经看到的，用于描述物理内存中虚拟页面子集的术语称为一个**工作集**（*working set*）。有以下三种工作集。

- 进程工作集包含了一个进程内的各个线程引用过的页面；
- 系统工作集包含了以下代码或数据驻留在内存中的那一部分：可换页的系统代码（例如，Ntoskrnl.exe和驱动程序）、换页池和系统缓存；
- 在支持终端服务的系统上，每个会话都有一个工作集，其中包含了以下代码或数据驻留在内存中的那一部分：Windows子系统的内核模式部分（Win32k.sys）申请的与会话有关的内核模式数据结构、会话换页池、会话的映射视图，以及其他的会话空间的设备驱动程序。

www.TopSage.com

在讨论每一种工作集的细节以前，我们来看一看总体策略上如何确定该把哪些页面带入到物理内存中，以及这些页面在内存中驻留多久。然后，我们再来讨论各种类型的工作集。

按需换页

Windows内存管理器使用一个按需换页的算法，把页面聚集起来，并将其加载到内存中。当一个线程接收到一个页面错误时，内存管理器把出错的页面，以及该页面之前和/或之后的少量页面，一起加载到内存中。这种策略的用意是，尽可能地使一个线程招致的换页I/O的次数减到最少。因为绝大多数程序，特别是大的程序，在任何给定的时刻，总是倾向于在其地址空间的小部分区域中执行，所以，加载一簇虚拟页面，可以减少磁盘读操作的次数。对于因为引用了映像中的数据页面而招致的页面错误，这一簇的大小是3个页面；对于其他的页面错误，簇的大小是7个页面。

然而，当一个进程的线程第一次开始执行，或者在稍后某个时刻当这些线程恢复执行时，按需换页策略有可能会使该进程招致许多页面错误。为了优化一个进程（和系统）的启动过程，Windows XP和Windows Server 2003有一个智能预取引擎，称为**逻辑预取器**（*logical prefetcher*）。下一节介绍此预取引擎。

7.10 逻辑预取器

在一次典型的系统引导或者应用程序启动过程中，页面错误的顺序是这样的：从一个文件的某个部分中读入一些页面，然后可能从同一文件相距较远的另一部分中读入一些页面，然后从另一个不同的文件中读入一些页面，可能又要从一个目录中读入页面，然后再从第一个文件中读入一些页面。这种跳跃大大减慢了每一次访问的速度，因此，分析表明，磁盘寻道时间是导致引导时间和应用程序启动时间变长的主导因素。通过一次预取批量页面，可以得到一个更加合理的访问顺序（其中没有过多的来回跳动），从而改进了系统和应用程序启动的总体性能。由于在多次引导或者应用程序启动之间所访问的数据有高度的相关性，因此，哪些页面是需要的，这是可以提前知道的。

Windows XP中引入的预取器，试图加速引导过程和应用程序启动过程。其做法是，监视引导过程和应用程序启动过程中所访问的数据和代码，然后在下一次引导或应用程序启动的开始时刻利用这些信息来读入代码和数据。当预取器起作用时，内存管理器将页面错误（无论是硬错误，还是软错误）通知给内核中的预取器。所谓硬页面错误是指，要求从磁盘上读入数据的页面错误；软页面错误是指，所要求的数据已经在内存中，这些页面只需被简单地加入到一个进程的工作集中即可。预取器监视应用程序启动过程的前10秒钟。对于引导过程，预取器默认从系统启动开始跟踪，直至用户外壳（通常是Explorer）启动之后30秒钟，或者，如果失败的话，跟踪到Windows服务初始化之后60秒，或者跟踪到120秒钟，这取决于哪个先到达。

www.TopSage.com

内核中收集到的痕迹信息记录了发生在以下数据体上的页面错误：NTFS主文件表（MFT，Master File Table）元数据文件 （如果应用程序访问NTFS卷上的文件或者目录）、被引用的文件，以及被引用的目录。收集到这些痕迹信息以后，内核预取器代码将一个名为*PrefetchTracesReady*的事件对象置为有信号状态，从而通知任务调度器服务（\Windows\System32\Schedsvc.dll）的预取器组件（运行在Svchost进程中）。



> 注 通过编辑注册表DWORD值HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters\EnablePrefetcher，你可以允许或者禁止引导过程或应用程序启动过程的预取特性。将该注册表值设置为0，可禁止所有的预取特性；设置为1，仅允许应用程序启动过程的预取特性；设置为2，仅允许引导过程的预取特性；设置为3，则允许这两者的预取特性。

当*PrefetchTracesReady*事件有信号时，任务调度器执行一个内部的*NtQuerySystemInformation*系统调用，请求痕迹数据。任务调度器处理这些痕迹数据，将它们与原先收集的数据合起来，写到\Windows\Prefetch文件夹下的一个文件中。图7.31显示了此文件夹。该文件的名称是此痕迹数据所对应的应用程序的名称跟上一个减号，再跟上该文件路径的一个散列值的十六进制表示。文件的扩展名为“.pf”，所以，给出一个例子为：NOTEPAD.EXE-AF43252301.PF。

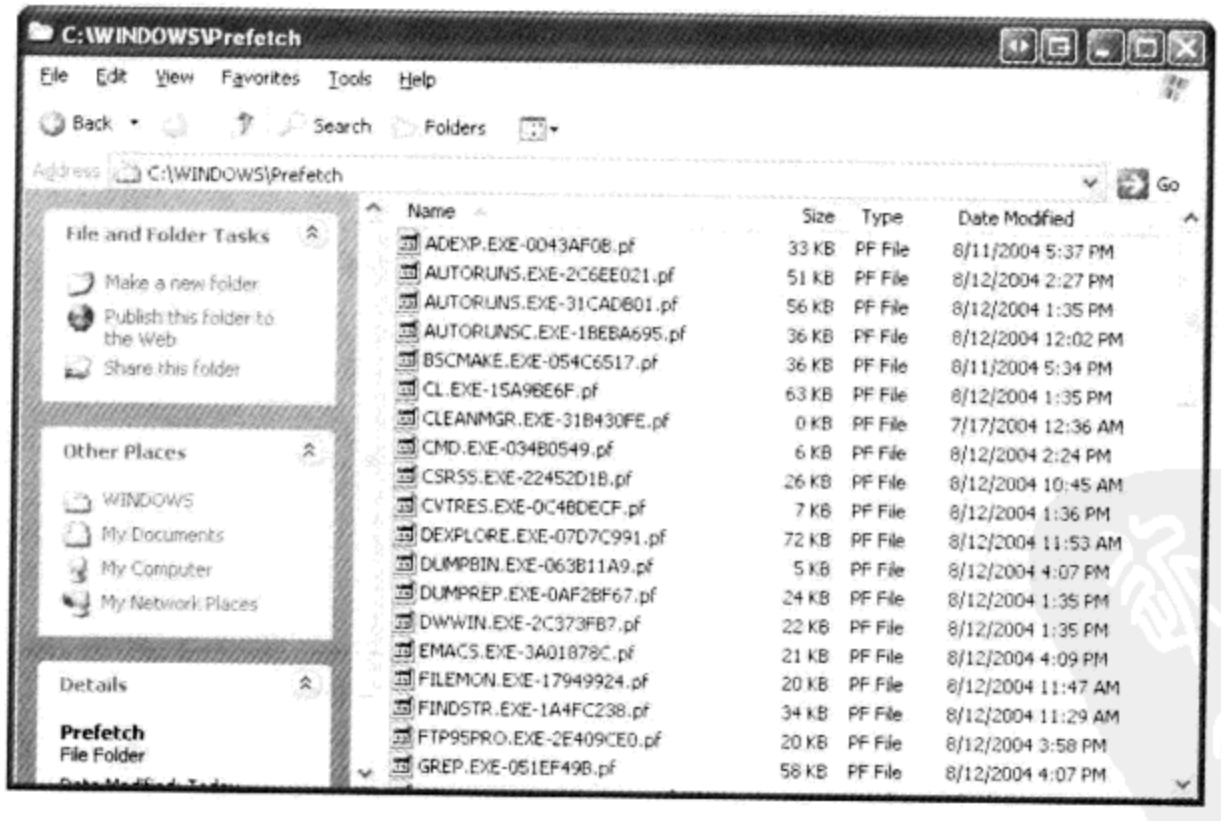


图7.31 Prefetch文件夹

此文件名规则有两个例外。第一是对于那些宿纳了其他组件的映像，包括Microsoft Management Console（\Windows\System32\Mmc.exe）和Dllhost（\Windows\System32\Dllhost.exe）。因为加载项组件是在这些应用程序的命令行上指定的，所以，预取器在生成的散列值中也包含

www.TopSage.com

了相应的命令行。因此，在命令行上用不同的组件来调用这些应用程序将会导致不同的痕迹数据。预取器在它的参数注册表键HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters中，从*HostingAppList*值中读取到它应该用这种方式来对待的可执行程序的列表。

文件名规则的另一个例外是保存引导过程痕迹的文件，它总是被命名为NTOSBOOT-B00DFAAD.PF。（如果将它读作一个单词的话，则“boodfaad”听起来有点像英语词组“*boot fast*”。）只有在预取器已经完成了引导痕迹（具体的时间前面已经定义过了。）以后，它才会为特定的应用程序收集有关页面错误的信息。



实验：仔细查看预取文件

预取文件的内容是在一次引导过程或者应用程序启动过程中所访问到的文件和目录的一个纪录，你可以使用*www.sysinternals.com*的Strings工具来查看此纪录。下面的命令列出了在最近一次引导过程中引用过的所有文件和目录：

```
C:\Windows\Prefetch>Strings ntosboot-boodfaad.pf

Strings v2.1
Copyright (C) 1999-2003 Mark Russinovich
Systems Internals - www.sysinternals.com

NTOSBOOT
SCCA
\DEVICE\HARDDISKVOLUME2\$MFT
\DEVICE\HARDDISKVOLUME2\WINDOWS\PREFETCH\NTOSBOOT-B00DFAAD.PF
\DEVICE\HARDDISKVOLUME2\SYSTEM VOLUME INFORMATION\_RESTORE{987E0331-0F01-427C-A5
8A-7A2E4AABF84D}\RP24\CHANGE.LOG
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\PROCESSR.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\FGLRYM.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\VIDEOPRT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\E1000325.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBUHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBPORT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBEHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\NIC1394.SYS
...
```

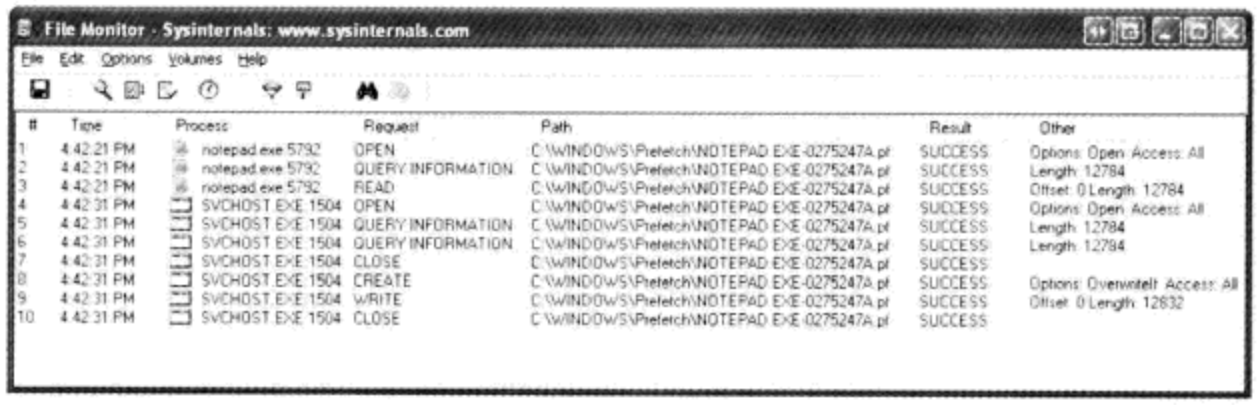
当系统引导或者应用程序启动的时候，预取器将会被调用，从而它有机会来执行预取工作。预取器检查prefetch目录，以查看是否存在与当前的预取情形相对应的痕迹文件。如果存在的话，则预取器调用NTFS来预取任何MFT元数据文件引用，读入每个被引用的目录的内容，最后打开每个被引用的文件。然后，它调用内存管理器函数*MmPrefetchPages*来读入该痕迹文件中指定的、尚未在内存中的任何数据和代码。内存管理器以异步方式发出所有的读数据请求，然后等待这些读操作完成，之后再让应用程序的启动过程继续进行。

www.TopSage.com



实验：观察预取文件的读和写

如果你在Windows XP上，用*www.sysinternals.com*的Filemon工具捕捉到了一段关于应用程序启动的痕迹数据，你就可以看到预取器检查并读入该应用程序的预取文件（如果存在的话），而且，差不多在应用程序启动之后10秒钟左右，看到预取器又写了该文件的一个新拷贝。下面是Filemon捕获到的Notepad启动的一段痕迹，其中Include过滤器被设置为“prefetch”，所以，它只显示访问\Windows\Prefetch目录的操作。



第1至3行显示了在Notepad进程启动过程中，Notepad预取文件在Notepad进程环境中被读访问了。第4至10行，其时间戳差不多比前3行晚了10秒钟，表明任务调度器更新了该预取文件。其中任务调度器运行在一个Svchost进程的环境下。

为了使磁盘定位时间尽可能地减到最少，每隔三天左右，在系统空闲过程中，任务调度器按照在一次引导过程或者应用程序启动过程中所引用的先后顺序，将引用过的文件和目录组织成一个列表，然后将该列表保存在一个名为Windows\Prefech\Layout.ini的文件中，图7.32显示了该文件的一个示例。



图7.32 为了预取而执行碎片整理的信息文件

www.TopSage.com

然后它激活系统的碎片整理器，通过命令行选项告诉碎片整理器，根据该文件的内容来整理碎片，而不是执行全局碎片整理的任务。碎片整理器在每个卷上找到一块足够大的连续区域，以容纳所有列出的位于该卷上的文件和目录，然后将它们整体移到此区域中，确保它们按照正确的先后顺序来存放。因此，将来的预取操作会更加高效，因为所有要读入的数据现在物理上也是按照它们被读取的顺序来存放的。为了预取而需要执行碎片整理任务的文件通常只是几百个文件而已，所以，此碎片整理过程比整卷的碎片整理要快得多（关于碎片整理的更多信息，请参见第12章）。

放置策略

当一个线程接收到一个页面错误的时候，内存管理器必须确定应该将此虚拟页面放在物理内存中什么地方。它用于确定最佳位置的规则集称为**放置策略**（*placement policy*）。Windows在选择页面帧的时候，考虑到了CPU内存高速缓存的大小，以便尽可能使不必要的高速缓存轮换次数减到最少。

如果当一个页面错误发生的时候，物理内存是满的，则使用一个**置换策略**（*replacement policy*）来确定必须将哪个虚拟页面从内存中移除出去，以便为新的页面腾出空间。常见的置换策略包括**最近最少使用**（LRU, *least recently used*）策略和**先进先出**（FIFO, *first in, first out*）策略。LRU算法（也称为**时钟算法**[*clock algorithm*]，在大多数UNIX版本中都有实现）要求虚拟内存系统记录下内存中的一个页面何时被使用过。当要求一个新的页面帧时，最长时间未被使用的那个页面被从工作集中去除。FIFO算法要简单得多：它将驻留在物理内存中最长时间的那个页面去除，而不考虑该页面被访问的频繁程度。

置换策略可进一步被归结为全局的或者局部的。全局置换策略使得一个页面错误可被任何一个页面帧来满足，无论该页面帧是否被另一个进程所拥有。例如，一个使用FIFO的全局置换策略将会找到位于内存中最长时间的那个页面，并释放该页面以满足一个页面错误；一个局部置换策略只能在引发页面错误的进程所拥有的页面集合中搜索最老的页面。全局置换策略使得系统中的进程易于受到其他进程行为的影响——一个行为不正常的应用程序可能会导致所有进程中发生大量额外的换页动作，从而破坏整个操作系统。

Windows实现了局部和全局置换策略的一种组合模式。当一个工作集到达它的限制，以及/或者由于物理内存的需求而有必要修剪（trim）时，内存管理器从工作集中去除一些页面，直到它认为已经有了足够的空闲页面为止。

www.TopSage.com

工作集管理

每个进程开始时都有一个默认的工作集最小值（50个页面）和一个工作集最大值（345个页面）。尽管效果并不显著，但你还是可以通过Windows的*SetProcessWorkingSetSize*函数来改变进程的工作集限制，不过，你必须有“增加调度优先级（increase scheduling priority）”的用户权限才能这么做。然而，除非你已经配置该进程使用硬工作集限制（Windows Server 2003中新引入），否则，这些限制都被忽略，因为内存管理器允许一个进程增长到超过它的最大值，假如它真的非常需要页面，而且确实有充足的页面（反之，内存管理器将会缩减一个进程到它的工作集最小值以下，假如它并不频繁需要页面，而且系统中物理内存的需求非常紧缺）。虽然在Windows 2000中，一个进程可以超越其工作集最大值的程度是其优先级的一个平衡因子，但是，到了Windows XP以后，这一决定只是建立在有多少页面已经被访问过的基础之上。

在Windows Server 2003中，通过*SetProcessWorkingSetSizeEx*函数，连同QUOTA_LIMITS_HARDWS_ENABLE标志，你可以设置硬工作集限制。此函数的一个例子客户程序是Windows系统资源管理器（WSRM，Windows System Resource Manager），如第6章中所描述。

工作集最大值不能超过系统范围内全局的最大值，这是在系统初始化时计算得到的，并存储在内核变量*MmMaximumWorkingSetSize*中的一个值。此值被设置为：在计算该值时可用页面的数量（零页面列表、空闲列表和备用列表的大小）减去512个页面。然而，对于工作集的大小，有一些硬的上限值——如表7.17所示。

表7.17 工作集最大值的上限值

Windows版本	工作集最大值
x86版本的Windows 2000、Windows XP、Windows XP SP1、Windows Server 2003	1984 MB
x86版本的Windows XP SP2、Windows Server 2003 SP1	2047.9 MB
以/3GB选项引导的x86版本Windows	3008 MB
IA-64	7152 GB
x64	8192 GB

当一个页面错误发生的时候，该进程的工作集限制值和系统中空闲内存的数量都会被检查到。如果条件允许的话，内存管理器使一个进程增长到它的工作集最大值（如果该进程没有一个硬工作集限制并且当前有足够的空闲页面可供使用的话，则可以超越该进程的工作集最大值）。然而，如果内存紧张的话，当页面错误发生的时候，Windows会替换工作集中的页面，而不是加入新的页面。

www.TopSage.com

虽然Windows通过将已修改的页面写到磁盘上，来保持有内存可以使用，但是，当以很高的速率产生这些修改的页面时，为了满足内存需求则需要更多的内存。因此，当物理内存运行到很低时，**工作集管理器**（*working set manager*），一个运行在平衡集管理器系统线程（下一小节介绍）环境下的例程，开始运行自动的工作集修剪过程，来增大当前系统中可用空闲内存的数量（通过前面提到的Windows函数*SetProcessWorkingSetSize*，你也可以为你自己的进程启动工作集修剪过程——例如，在进程初始化以后）。

工作集管理器检查可用内存的数量，并决定哪些工作集（如果有的话）需要修剪。如果内存充足的话，工作集管理器计算有多少页面可被从工作集中去除（如果需要的话）。如果需要修剪，则它查看那些超过最小值的工作集。它也会动态地调整工作集的检查频率，以及以最优的顺序来安排那些可能需要被修剪的进程的列表。例如：首先检查那些最近有很多页面未被访问过的进程；空闲时间越长的大进程，应该排在那些频繁运行的小进程的前面；运行前台应用程序的进程应该被排在最后；如此等等。

当工作集管理器利用“当前的工作集大小超过最小值”为条件找到了一些进程时，它寻找一些页面，将它们从这些进程的工作集中去除，从而使这些页面可被用于其他的用途。如果空闲内存的数量仍然太少，则工作集管理器继续从进程的工作集中去除页面，直至系统中空闲页面的数量达到一个最小值为止。

在单处理器的Windows 2000系统，和所有运行Windows XP和Windows Server 2003的系统上，工作集管理器试图去除那些最近未被访问过的页面。它的做法是，检查硬件PTE中的“访问过”位，看该页面是否被访问过。如果该位已被清除，则说明该页面是**老的**，于是，一个相应的计数器被递增，表明该页面自从上一次工作集修剪扫描以来尚未被引用过。以后，通过页面的年龄，可以确定此工作集中可被去除的候选页面。



注 在一个Windows 2000多处理器系统上，工作集管理器有一个错误，它没有检查“访问过”位，导致的结果是，无论这一位的状态是什么，工作集中的页面都会被去除。

如果硬件PTE的“访问过”位被设置的话，则工作集管理器将其清除，并继续检查工作集中的下一个页面。通过这种方法，如果工作集管理器下一次检查该页面的时候，它的“访问过”位是清除的，则它知道，自从上一次检查以来，该页面尚未被访问过。为了去除页面而执行的扫描操作继续进行，它经历整个工作集列表，直到期望数量的页面已经被去除，或者此次扫描已经回到了开始点（当下一次工作集被修剪的时候，其扫描操作从上一次停下来的地方继续进行）。

www.TopSage.com



实验：查看进程工作集大小

你可以利用性能工具来检查进程的工作集大小，只需观察下面的性能计数器即可：

计数器	说 明
Process: Working Set	所选中进程的工作集的当前大小（按字节）
Process: Working Set Peak	所选中进程的工作集的尖峰大小（按字节）
Process: Page Faults/Sec	一个进程每秒钟发生的页面错误的次数

几个其他的进程查看工具（比如任务管理器、Pview和Pviewer）也会显示进程的工作集大小。

如果你在性能工具的实例（instance）框中选择“_Total”进程，则也可以得到所有的进程工作集的总和。这（指_Total）并不是一个真实的进程——它仅仅是指，针对当前系统上运行的所有进程，这一与进程相关的计数器作用在它们身上的总和。然而，你看到的总和是很容易误导人的，因为每个进程工作集的大小值包含了那些被其他进程共享的页面。因此，如果两个或者多个进程共享了一个页面，那么，在每个进程的工作集中都计算了该页面。



实验：查看工作集列表

通过内核调试器的*!wsle*命令，你可以查看工作集中单独的项目。下面的例子显示了LiveKd的工作集列表的部分输出。（针对LiveKd进程运行该命令）

```
kd> !wsle 7

Working Set @ c0502000
    Quota:       9f  FirstFree:   40  FirstDynamic:       3
    LastEntry   1fe  NextSlot:     3  LastInitialized   257
    NonDirect    5c  HashTable:    0  HashTableSize:      0

Virtual Address  Age  Locked  ReferenceCount
c0300203           0       1               1
c0301203           0       1               1
c0502203           0       1               1
c01df201           0       0               1
c01ff201           0       0               1
c0005201           0       0               1
c0001201           0       0               1
c0002201           0       0               1
c0000201           0       0               1
c0006201           0       0               1
77e87119           0       0               1
```


www.TopSage.com

00402319	0	0	1
77e01201	0	0	1
7ffdf201	0	0	1
00130201	0	0	1
77e9e119	0	0	1
78033201	0	0	1
00230221	0	0	1
00131201	0	0	1
77d50119	0	0	1
00132201	0	0	1
c01e0201	0	0	1
00411309	0	0	1
0040d201	0	0	1
77edf201	0	0	1
77ee0201	0	0	1
77fcd201	0	0	1
0040e201	0	0	1
7ffc1009	0	0	1
00401319	0	0	1

注意，在此工作集列表中，有些项目是页表页面（地址大于0xC0000000的页面），有些来自于系统DLL（在0x7nnnnnnn范围中的页面），有些则来自于LiveKd.exe本身的代码（在0x004nnnnn范围中的页面）。

平衡集管理器和交换器

工作集的扩展和修剪是在一个被称为**平衡集管理器**（*balance set manager*）（例程*KeBalanceSetManager*）的系统线程的环境中进行的。平衡集管理器是在系统初始化的过程中创建的。尽管从技术上来说，平衡集管理器是内核的一部分，但是，它调用内存管理器的工作集管理器来完成工作集的分析和调整。

平衡集管理器等待两个不同的事件对象：一个事件被周期性地设置成有信号状态，它通过一个定时器每秒钟被触发一次；另一个是内部的工作集管理器事件，在各个点上当内存管理器确定需要调整工作集时，它就会将该事件置成有信号状态。例如，如果系统正在经历很高频率的页面错误，或者空闲列表太小，内存管理器就会唤醒平衡集管理器，以便它调用工作集管理器，来开始修剪工作集。当内存足够多的时候，工作集管理器把出错的页面换回到内存中，以允许这些发生页面错误的进程逐渐增加其工作集的大小，但这些工作集只是根据需要来增长。

当平衡集管理器通过1秒钟的定时器到期而被唤醒时，它采取以下4个步骤：

1. 每到第四次平衡集管理器被1秒钟定时器到期而唤醒时，它给另一个事件发信号，该事件唤醒另一个被称为交换器（swapper）（例程*KeSwapProcessOrStack*）的系统线程。

www.TopSage.com

2. 然后，平衡集管理器检查预读列表，如果有必要的话则调整它们的深度（以缩短访问时间，减少内存池使用量和碎片情况）。
3. 它寻找那些可能获准“因为CPU饥饿而被提升优先级”的线程（参见第6章中“因为CPU饥饿而引起的优先级提升”一节）。
4. 它调用内存管理器的工作集管理器（工作集管理器有它自己的一组内部计数器，通过这些计数器来控制何时执行工作集修剪，如何更加主动地进行修剪）。

如果一个需要运行的线程，其内核栈被换出了，或者进程被换出了，则内核中的调度代码也会唤醒交换器。交换器寻找那些已经处于等待状态7秒钟（在Windows 2000中）或者15秒钟（在Windows XP和Windows Server 2003中）的线程。如果它找到了一个，则把该线程的内核栈置入转移（transition）状态（将页面移到修改列表或者备用列表中），以便回收它的物理内存。这种做法基于的原则是，如果一个线程已经等了那么长时间，则它将会等待更长的时间。当一个进程中最后一个线程的内核栈被从内存中去除时，该进程被标记为已完全被换出。这也正是为什么在进程空闲了很长一段时间以后（比如在你登录以后Winlogon的情形），它们的工作集大小可能为零。

系统工作集

就像进程有工作集一样，操作系统中的可换页代码和数据也是通过一个**系统工作集**（*system working set*）来管理的。系统工作集中可以驻留以下5种不同类型的页面：

- 系统缓存页面；
- 换页内存池；
- Ntoskrnl.exe中的可换页代码和数据；
- 设备驱动程序中的可换页代码和数据；
- 系统映射的视图。

你可以通过性能计数器或者系统变量，如表7.18所示，来检查系统工作集的大小，或者这5个组成部分的大小。记住，性能计数器值是以字节为单位的，而系统变量则是以页面数来计量的。

你也可以通过检查性能计数器“Memory: Cache Faults/Sec”来检查系统工作集中的换页活动，该计数器描述了发生在系统工作集中的页面错误（包括硬的和软的）。

www.TopSage.com

表7.18 系统工作集性能计数器

性能计数器（以字节为单位）	系统变量（以页面为单位）	说 明
Memory: Cache Bytes[2]	*MmSystemCacheWs.WorkingSetSize*	系统工作集的总大小(包括系统缓存、换页内存池、可换页的Ntoskrnl和驱动程序代码，以及系统映射的视图)；这不是系统缓存单个部分的大小，尽管此性能计数器或系统变量的名称看起来好像是这样的
Memory: Cache Bytes Peak	*MmSystemCacheWs.Peak*	系统工作集的尖峰大小
Memory: System Cache Resident Bytes	*MmSystemCachePage*	系统缓存所使用的物理内存
Memory: System Code Resident Bytes	*MmSystemCodePage*	Ntoskrnl.exe中可换页代码所使用的物理内存
Memory: System Driver Resident Bytes	*MmSystemDriverPage*	可换页的设备驱动程序代码所使用的物理内存
Memory: Pool Paged Resident Bytes	*MmPagedPoolPage*	换页内存池所使用的物理内存

包含此计数器值的系统变量是MmSystemCacheWs.PageFaultCount。

系统工作集大小的最小值和最大值是在系统初始化时刻根据机器上物理内存的数量，以及系统运行的是Windows客户版本还是服务器版本来计算的。计算得到的工作集最小值和最大值被保存在如表7.19所示的内核变量中。这些变量并不能通过任何性能计数器而获得，但是你可以利用内核调试器来查看它们。

表7.19 保存工作集最小值或最大值的系统变量

变量	类 型	说 明
MmSystemCacheWs.MinimumWorkingSetSize	ULONG	最小工作集尺寸
MmSystemCacheWs.MaximumWorkingSetSize	ULONG	最大工作集尺寸

通过修改HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargeSystemCache的值，你可以对系统进行配置，以便给予系统工作集优先权（相对于进程工作集）。在Windows 2000服务器系统中，你可以通过设置文件服务器服务的属性来间接地调整该值；在Windows XP和Windows Server 2003中，你可以通过下述方法来调整该值：转到My Computer，选择Properties，点击Advanced标签，在Performance部分中按下Setting按钮，最后再点击Advanced标签（细节请参见第11章）。

[2] 在内部，这一工作集被称为系统高速缓存(*cache*)工作集，尽管系统高速缓存只是系统工作集的5个组成部分之一。因此，有几个实用工具在显示系统工作集的总大小时，它们认为自己显示的是文件高速缓存的大小。

www.TopSage.com

7.11 页面帧编号数据库

工作集描述了一个进程或者系统所拥有的驻留在内存中的页面，而**页面帧编号**（*PFN*，*page frame number*）**数据库**则描述了物理内存中每个页面的状态。页面状态如表7.20所示。

表7.20 页面状态

状 态	说 明
活动的（也称为有效的）	该页面属于一个工作集（可以是进程工作集，或者是系统工作集）或者它不在任何一个工作集中（例如，非换页的内核页面），有一个有效的PTE指向该页面
转移	这是一种临时状态，针对一个未被任何工作集拥有的，也不在任何一个换页列表上的页面。当一个针对该页面的I/O正在进行时，该页面处于此状态。一个PTE被如此编码以后，冲突的页面错误就可以被识别出来，并正确地进行处理 （注意，术语“转移”的这种用法不同于在“无效PTE”小节中同一词语的用法；无效的转移PTE指的是备用列表或修改列表上的页面）
备用	这样的页面原先属于某个工作集，但现在被去除了。该页面自从最后一次被写到磁盘上以来尚未被修改过。PTE仍然指向该物理页面，但已被标记为无效的和正在转移之中
已修改	这样的页面原先属于某个工作集，但现在被去除了。然而，该页面在其使用过程中已经被修改了，它的当前内容尚未被写到磁盘上。PTE仍然指向该物理页面，但已被标记为无效的和正在转移之中。在它的物理页面被重用以前，其内容必须被写到磁盘上
已修改但不写出（Modified no-write）	与已修改的页面相同，但是它这样标记以后，内存管理器的已修改页面写出器将不会把它写到磁盘上。缓存管理器在接收到文件系统驱动程序的请求时，把页面标记为已修改但不写出的标志。例如，NTFS对于那些包含文件系统元数据的页面使用此状态，因此它可以确保，在事务日志项所保护的页面被写到磁盘上以前，这些日志项首先被刷新到磁盘上（关于NTFS事务日志，请参见第12章的解释）
空闲	页面是空闲的，但它包含了未特别指明的脏数据 （出于安全性的原因，这样的页面在未用零来初始化的情况下是不能被当作一个用户页面交给用户进程的）
零化的	页面是空闲的，并且已经被零页面线程初始化为零了
Rom	该页面来自于只读内存（Windows XP之后，新的页面类型）
坏的	该页面产生了奇偶错，或其他的硬件错误，不能使用了

PFN数据库是由一个结构阵列构成的，这些结构代表了系统中内存的每个物理页面。PFN数据库和它与页面之间的关系如图7.33所示。

www.TopSage.com

如该图中所示，有效的PTE指向PFN数据库中的表项，而PFN数据库的表项（对于非原型PFN）又指回到使用这些表项的页表上。对于原型PFN，它们指回到原型PTE上。

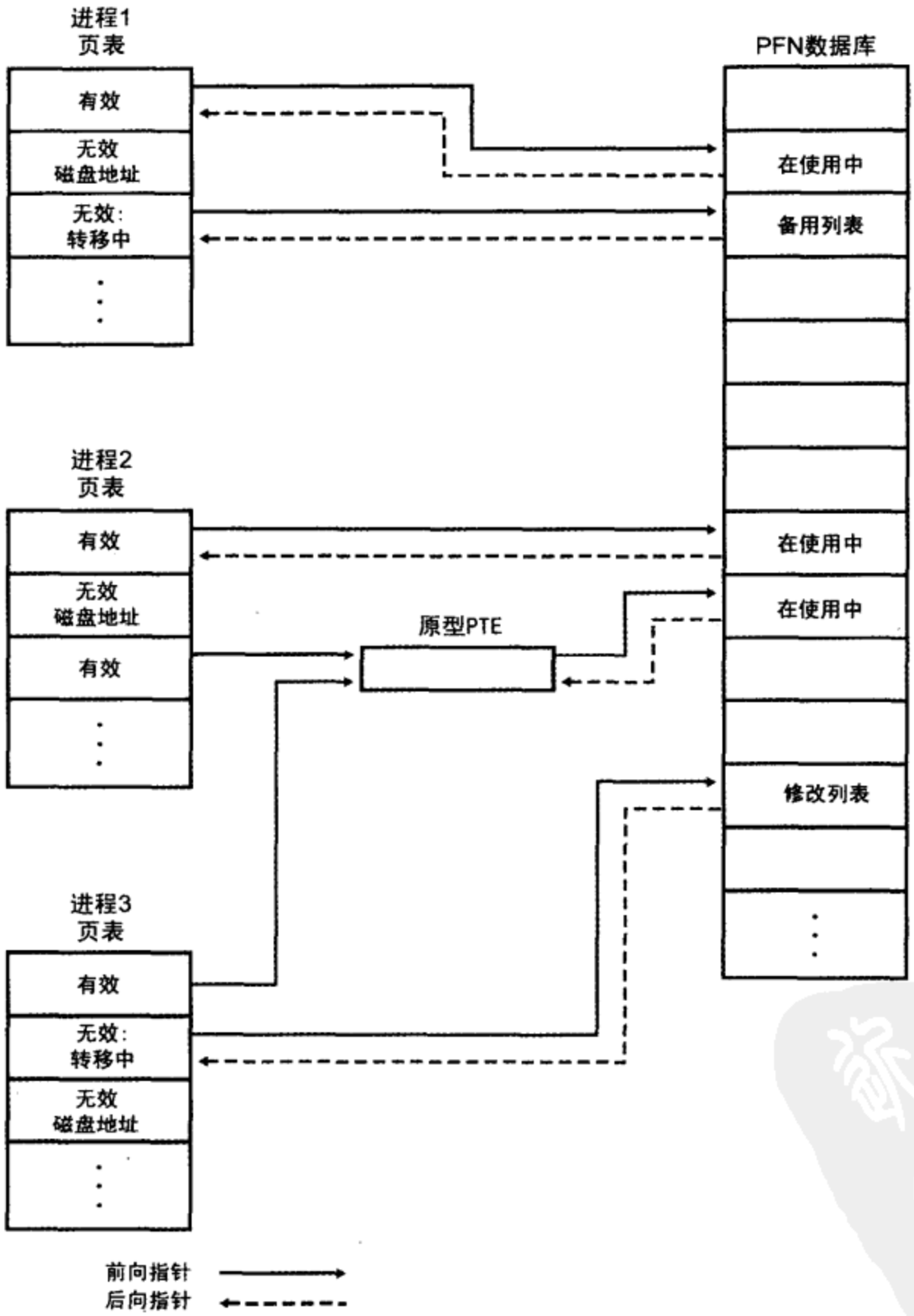


图7.33 页表和页面帧编号数据库

www.TopSage.com

在表7.20所示的页面状态中，其中6个按链表来组织，因此，内存管理器可以很快地找到指定类型的页面（活动的/有效的页面和转移类型的页面不在任何一个系统全局范围的页面列表中）。图7.34显示了一个例子，用以说明这些表项是如何被链接在一起的。

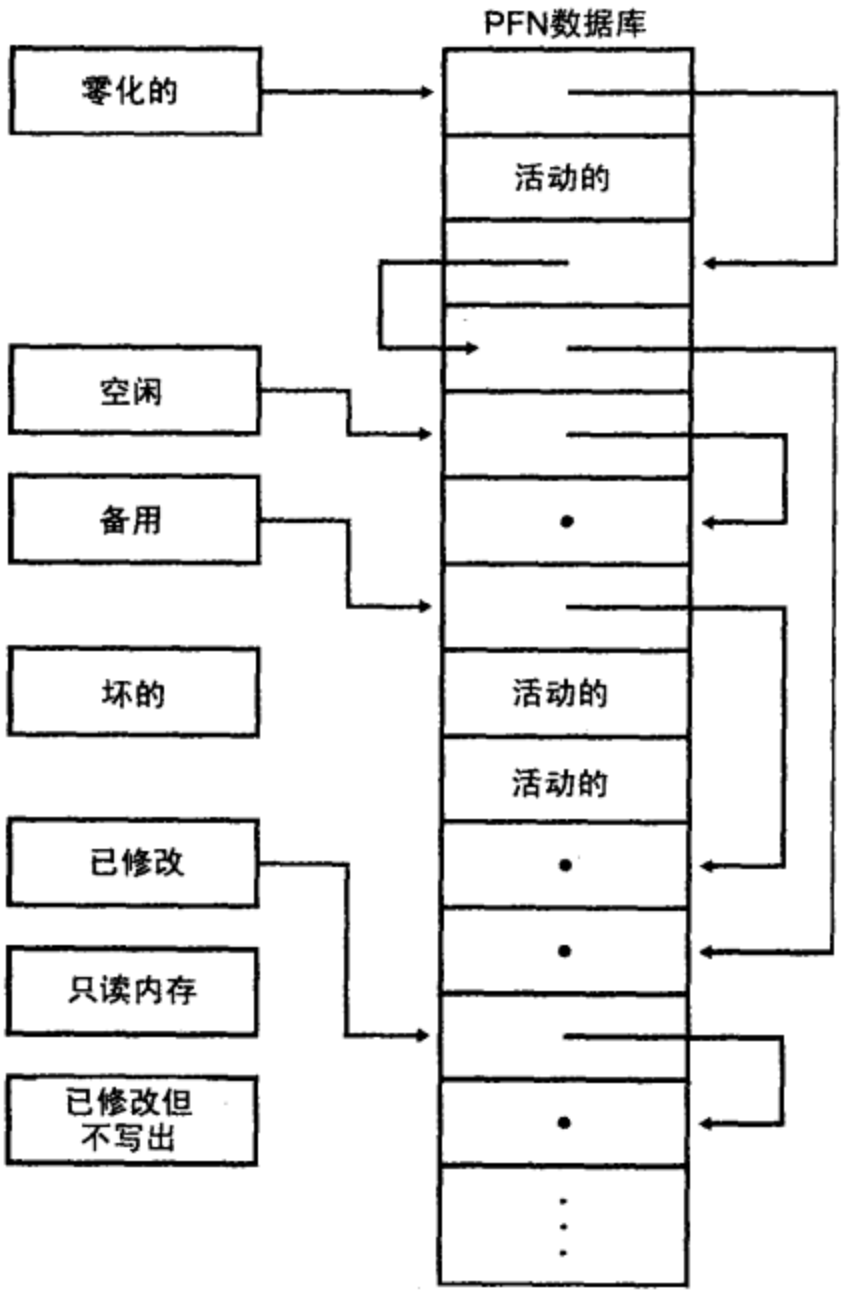


图7.34 PFN数据库中的页面列表

在下一小节中，你将会知道，这些链表是如何被用来满足页面错误的，以及页面是如何在各种列表之间转移的。

www.TopSage.com



实验：查看PFN数据库

使用内核调试器*!memusage*命令，你可以转储出各种页面列表的大小。下面是该命令的输出：

```
lkd> !memusage
 loading PFN database
loading (100% complete)
Compiling memory usage data (99% Complete).
              Zeroed:   8474 ( 33896 kb)
                Free:    256 (  1024 kb)
             Standby:  50790 (203160 kb)
            Modified:    496 (  1984 kb)
    ModifiedNoWrite:      0 (     0 kb)
       Active/Valid: 201980 (807920 kb)
         Transition:      1 (     4 kb)
            Unknown:      0 (     0 kb)
              TOTAL: 261997 (1047988 kb)
```

页面列表的动态变化

图7.35显示了页面帧的状态转移图。为了简化起见，这里没有显示“已修改但不写出”的页面列表。

页面帧按照以下方式在页面列表之间移动。

- 当内存管理器需要一个初始化为零的页面，以满足一个要求零的页面错误（引用一个被定义为全零的页面，或者一个从未被访问过的用户模式私有提交页面）时，它首先试图从零页面列表中获取一个页面。如果该列表是空的，则它从空闲页面列表中获取一个，并且零化该页面（即用零填充）。如果空闲列表也是空的，则它转到备用列表，并零化此页面。之所以要求零初始化的页面，其中一个原因是为了满足C2安全要求。C2规定了用户模式进程得到的页面帧必须是已初始化的，以避免它们读取到前一个进程的内存内容。因此，内存管理器给予用户模式进程的是零化的页面帧，除非该页面是从一个映射文件中读入进来的。如果是这种情况的话，则内存管理器首选使用未零化的页面帧，并且用磁盘上的数据对页面进行初始化。

www.TopSage.com

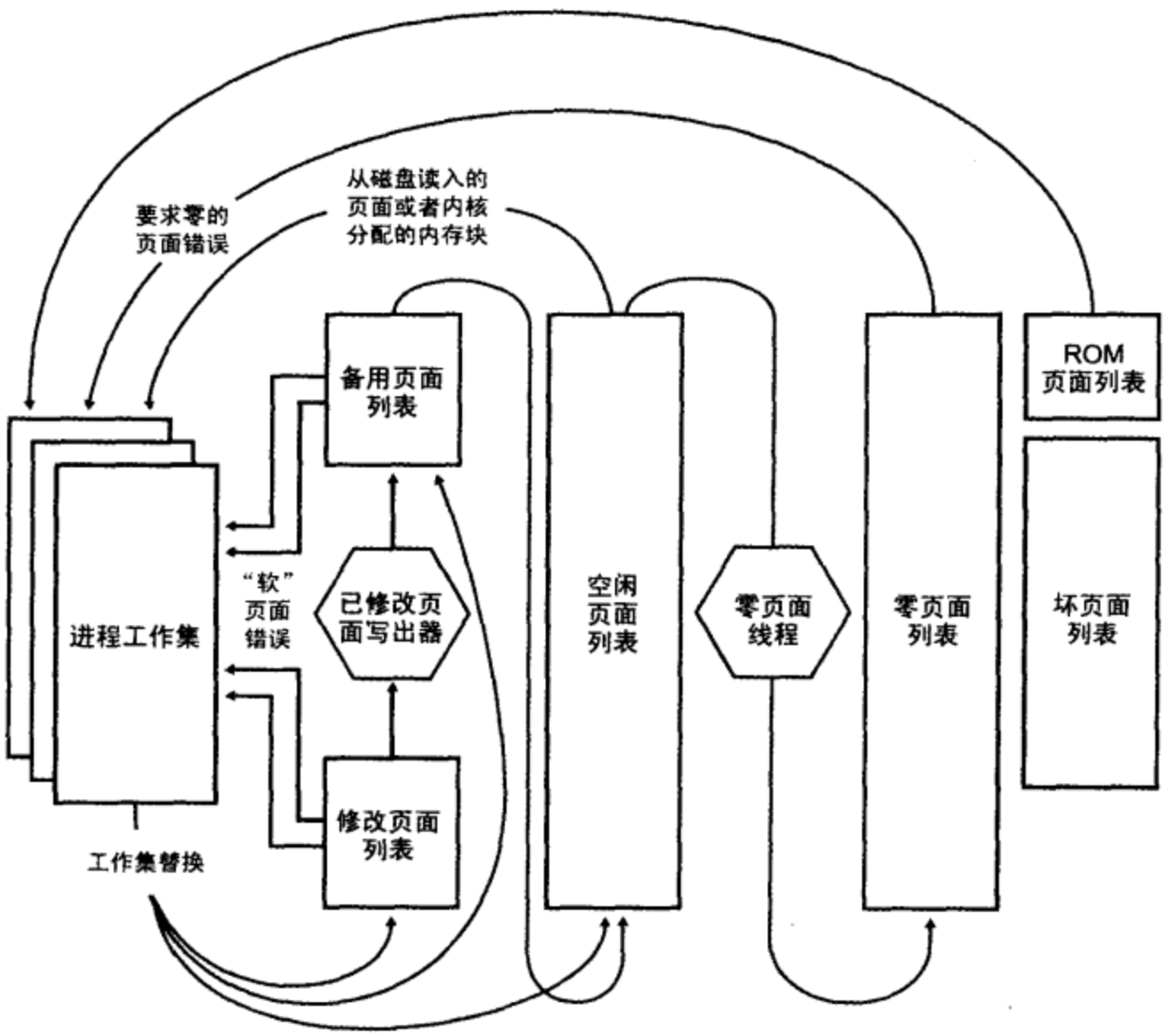


图7.35 页面帧的状态图

零页面列表是由一个被称为**零页面线程**（*zero page thread*，即System进程中的线程0）的系统线程，从空闲列表中生成的。零页面线程在一个事件对象上等待，当该对象有信号时它便开始工作。当空闲列表有8个或者更多页面时，该事件会得到信号。然而，只有当没有其他线程在运行的时候零页面线程才会运行，因为它在优先级0上运行，而用户线程可以被设置的最低优先级是1。



> 注 在Windows Server 2003及其以后的版本中，当由于驱动程序调用*MmAllocatePagesForMdl*，或者Windows应用程序调用*AllocateUserPhysicalPages*，因而一块物理页面内存需要被零化时，或者当一个应用程序申请大的页面时，内存管理器使用一个更高性能的函数*MiZeroInParallel*来零化此内存。该函数映射大的内存区域，而不像零页面线程每次只零化一个页面。而且，在多处理器系统上，它创建多个系统线程来并行地执行零化工作（在NUMA平台上以一种专门针对NUMA优化的方式来执行零化工作）。

www.TopSage.com

- 当内存管理器不要求一个零初始化的页面时，它首先到空闲列表上寻找页面。如果该列表是空的，则转到零化的列表上。如果零化的列表也是空的，则转到备用列表上。内存管理器在使用一个来自备用列表的页面帧以前，它必须首先向后回退一下，从仍然指向该页面帧的无效PTE（或者原型PTE）中删除此引用。因为PFN数据库中的表项包含了指回到原先用户的页表（或者对于共享页面而言，指向原型PTE）的指针，所以，内存管理器可以很快地找到该PTE，并且做出正确的修改。
- 当一个进程必须从它的工作集中放弃一个页面（因为它引用了一个新的页面而它的工作集是满的，或者因为内存管理器修剪了它的工作集）时，如果该页面是干净的（未被修改过），则它被转到备用列表中；如果该页面已被修改过，则转到修改列表中，同时它仍然驻留在内存中。当一个进程退出的时候，所有的私有页面都被转到空闲列表中。而且，当一个由页面文件支撑的内存区的最后一个引用被关闭时，这些页面也被转到空闲列表中。



实验：查看页面错误的行为

利用Pfmon工具（在Windows 2000和2003的资源工具箱中，也在Windows XP支持工具箱中），你可以观察页面错误发生时的行为。**软错误**（*soft fault*）是指可以从某个转移列表来满足的页面错误。**硬错误**（*hard fault*）指的是一次磁盘读操作。如果你用Pfmon来启动Notepad，然后退出，你将会看到以下的输出片断。一定要留意最后给出的有关页面错误活动的摘要信息。

```
C:\> pfmon notepad
SOFT: KiUserApcDispatcher : KiUserApcDispatcher
SOFT: LdrInitializeThunk : LdrInitializeThunk
SOFT: 0x77f61016 :  : 0x77f61016
SOFT: 0x77f6105b :  : fltused+0xe00
HARD: 0x77f6105b :  : fltused+0xe00
SOFT: LdrQueryImageFileExecutionOptions :
                        LdrQueryImageFileExecutionOptions
SOFT: RtlAppendUnicodeToString : RtlAppendUnicodeToString
SOFT: RtlInitUnicodeString : RtlInitUnicodeString

  notepad Caused   8 faults had   9 Soft  5 Hard faulted VA's
    ntdll Caused  94 faults had  42 Soft  8 Hard faulted VA's
 comdlg32 Caused   3 faults had   0 Soft  3 Hard faulted VA's
  shlwapi Caused   2 faults had   2 Soft  2 Hard faulted VA's
    gdi32 Caused  18 faults had  10 Soft  2 Hard faulted VA's
 kernel32 Caused  48 faults had  36 Soft  3 Hard faulted VA's
   user32 Caused  38 faults had  26 Soft  6 Hard faulted VA's
 advapi32 Caused   7 faults had   6 Soft  3 Hard faulted VA's
   rpcrt4 Caused   6 faults had   4 Soft  2 Hard faulted VA's
 comctl32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
  shell32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
          Caused  10 faults had   9 Soft  5 Hard faulted VA's
 winspool Caused   4 faults had   2 Soft  2 Hard faulted VA's

PFMON: Total Faults 250
       (KM 74 UM 250 Soft 204, Hard 46, Code 121, Data 129)
```


www.TopSage.com

已修改页面写出器

如果当修改列表变得太大，或者如果零化列表和备用列表的大小小于一个最小阈值（如内核变量*MmMinimumFreePages*所指示，这是在系统引导时刻计算得到的值）时，那么，两个系统线程中的一个会被唤醒，它将修改列表中的页面写回到磁盘上，并且把它们转移到备用列表中。第一个系统线程将已修改的页面写到页面文件中（*MiModifiedPageWriter*），第二个系统线程将已修改的页面写到映射文件中（*MiMappedPageWriter*）。为了避免建立死锁，使用两个线程是必要的。例如，如果映射文件页面的写操作引起页面错误，该页面错误又要求一个空闲页面，而这时恰好没有空闲页面可用（因而要求已修改页面写出器创建更多的空闲页面），则死锁就会发生。让已修改页面写出器在第二个系统线程中执行映射文件换页I/O，这样该线程可以等待，因而不会阻塞正常的页面文件I/O。

这两个线程都运行在优先级17上，在初始化以后，它们分别等待不同的事件对象来触发其操作。已修改页面写出器的事件被触发的原因有以下两个：

- 已修改页面的数量超过了在系统初始化时候计算的一个最大值（*MmModifiedPageMaximum*，当前对于所有的系统都是800个页面）；
- 可用页面的数量（MmAvailablePages）减小到MmMinimumFreePages以下。

已修改页面写出器还等待另外一个事件（*MiMappedPagesTooOldEvent*），该事件是在一个预定的秒数（*MmModifiedPageLifeInSeconds*）之后被设置，以指示那些映射的页面（不是已修改的页面）应该被写到磁盘上。在默认情况下，该值为300秒（5分钟）（你只要加入一个DWORD注册表值HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ModifiedPageLife，就可以改变该值）。之所以要引入此额外事件是因为万一系统崩溃或者电源失败了，这样做可以减少数据丢失。因为即使已修改列表尚未到达它的阈值800个页面，它最终也会把已修改的映射页面写出去。

映射页面写出器（mapped page writer）被调用的时候，它试图用单个I/O请求把尽可能多的页面写到磁盘上。其做法是，检查修改列表上的页面在PFN数据库中相应元素的原始PTE域，以便在磁盘上找到连续位置的页面。一旦建立起这样一个列表，就从修改列表中去除这些页面，并发出一个I/O请求，等到此I/O请求成功完成以后，这些页面被放到备用列表的尾部。

正在写过程中的页面也可能被另外一个线程引用。当这种情形发生的时候，代表该物理页面的PFN项中的引用计数和共享计数都会递增，以指明另一个进程正在使用该页面。当I/O操作完成的时候，已修改页面写出器注意到共享计数不再为0，因此就不会把该页面放到备用列表中。

www.TopSage.com

PFN数据结构

尽管PFN数据库项是固定长度的，但是，根据页面状态的不同，它们可以处于几种不同的状态之中。因此，根据不同的页面状态，有些域有不同的含义。PFN项的状态如图7.36所示。

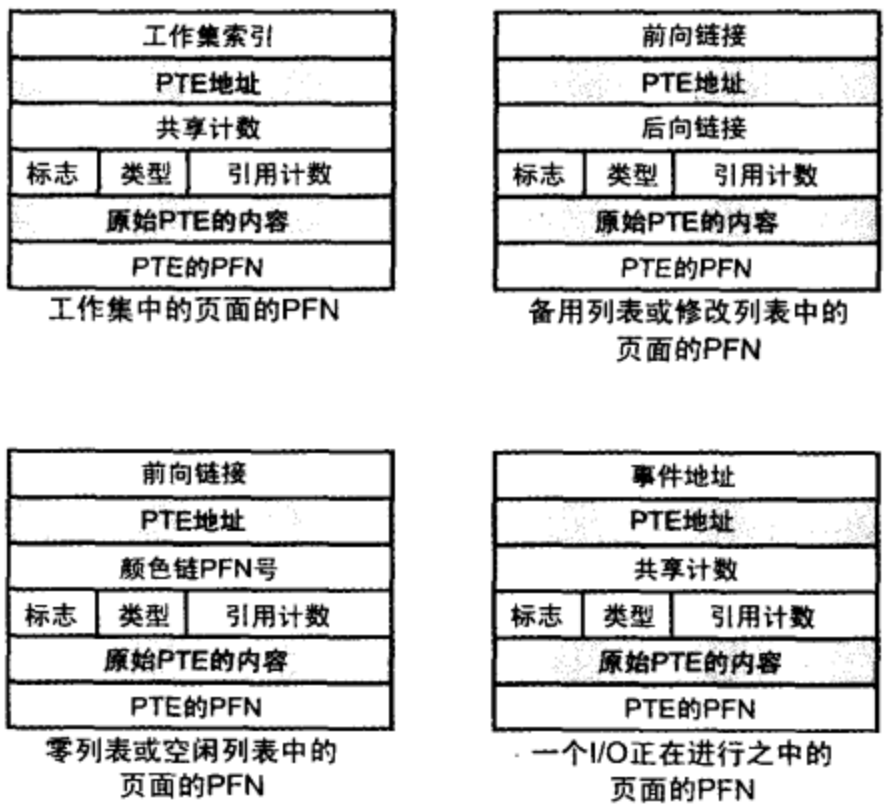


图7.36 PFN数据库项的状态

对于几种PFN类型，有些域是相同的，但其他的域则特定于指定类型的PFN。下面的域出现在多个PFN类型中。

- **PTE地址** 指向此页面的PTF的虚拟地址。
- **引用计数** 引用此页面的次数。当一个页面被第一次加入到一个工作集时，以及/或者当该页面由于I/O而被锁在内存中（比如说，设备驱动程序可能会这样做）时，引用计数就会递增。当共享计数变成0时，或者当页面被从内存中解锁时，引用计数会被递减。当共享计数变成0时，该页面不再被任何一个工作集所拥有。然后，如果引用计数也是零的话，则描述该页面的PFN数据库项也随之被更新，以便将该页面加入到空闲列表、备用列表或者修改列表中。
- **类型** 该PFN所代表的页面的类型（类型包括活动的/有效的、备用、已修改、已修改但不写出、空闲、零化的、坏的，以及正在转移）。
- **标志** 标志域中包含的信息如表7.21所示。

www.TopSage.com

- **原始PTE的内容** 所有的PFN数据库项都包含了指向该页面的PTE（可能是一个原型PTE）的原始内容。由于保存了此PTE的内容，因而使得当该物理页面不再驻留在内存中时，此PTE可以被恢复。
- **PTE的PFN** 指向该页面的PTE所在的页表页面的物理页面编号。

表7.21 PFN数据库项中的标志

标 志	含 义
修改的状态	指明该页面是否被修改了（如果该页面已被修改，那么，在从内存中去除该页面以前，必须将它的内容保存到磁盘上）
原型PTE	指明了该PFN项所引用的PTE是一个原型PTE（例如，该页面是可共享的）
奇偶错	指明了该物理页面包含了奇偶错，或者纠错控制错误
读操作正在进行中	指明了换入该页面的操作正在进行之中。第一个DWORD包含了一个事件对象的地址，当该I/O完成时，此事件对象将会被通知到；对于非换页池中的内存，该标志也被用于指明是第一个PFN
写错误正在进行中	指明了一个页面写操作正在进行之中。第一个DWORD包含了一个事件对象的地址，当该I/O完成时，此事件对象将会被通知到；对于非换页池中的内存，该标志也被用于指明是最后一个PFN
非换页池的开始	对于非换页池的页面，指明了这是某一特定的非换页池内存块的第一个PFN
非换页池的结束	对于非换页池的页面，指明了这是某一特定的非换页池内存块的最后一个PFN
页面换入错误	指明了在换入该页面的操作过程中发生了I/O错误（在这种情况下，PFN中的第一个域包含了错误代码）

剩下的域都是与PFN的类型相关的。例如，图7.36中的第一个PFN代表了一个活动的页面，该页面属于一个工作集。共享计数域代表了指向该页面的PTE的数量（被标记为只读、写时复制，或者共享读/写的页面可被多个进程所共享）。对于页表页面，该域是页表中有效PTE的数量。只要共享计数大于0，该页面就不会满足从内存中去除的条件。

工作集索引域是一个索引值，指向该物理页面对应的虚拟地址所在的进程工作集列表（或者系统工作集列表或会话工作集列表，或者，如果不在任何一个工作集中，则为零）。如果该页面是一个私有页面，则工作集索引域直接指向该工作集列表中的表项，因为该页面只被映射到单个虚拟地址上。对于共享页面的情形，工作集索引是一个线索，它只保证第一个使该页面变成有效的进程是正确的（其他的进程将在可能的情况下试着使用同样的索引）。最初设置此域的进程保证了该域指向正确的索引，并且不需要在它的工作集散列树中加入一个由该虚拟地

www.TopSage.com

址引用的工作集列表散列项。这一保证的好处是，减小了工作集散列树的大小，并且能更快地搜索这些特殊的直接表项。

图7.36中的第二个PFN是针对那些位于备用列表或者修改列表上的页面的。在这种情况下，前向和后向链接把列表中的元素链接起来。这种链接使得很容易操作这些页面来满足页面错误。当一个页面位于这些列表中的某一个时，根据定义，共享计数为0（因为没有工作集在使用该页面），因此可以用后向链接来覆盖该域。如果一个页面位于这些列表中的某一个，则引用计数也是0。如果引用计数非零（因为针对该页面的一个I/O可能正在进行之中——例如，当该页面正在被写到磁盘上时），则它首先被从该列表中除掉。

图7.36中的第三个PFN是针对空闲列表或者零化列表上的页面的。除了在这两个列表内部页面被链接起来以外，这些PFN数据库项使用一个额外的域，通过“颜色”，即它们在处理器CPU内存高速缓存中的位置，来链接物理页面。Windows试图在CPU高速缓存中使用不同的物理页面，以尽可能使不必要的CPU高速缓存轮换减到最小。它实现这种优化的做法是，只要有可能，就避免为两个不同的页面使用相同的缓存项。对于直接映射高速缓存的系统，以最优的方式来使用硬件的能力可以导致重要的性能提升。

图7.36中的第四个PFN是针对那些有一个I/O（比如，一个读页面的操作）正在进行之中的页面的。虽然此I/O正在进行之中，但是，第一个域指向一个事件对象，当此I/O完成时，该事件对象将会被通知到。如果发生页面换入错误的话，则该域包含了代表此I/O错误的Windows错误状态代码。此PFN类型可被用来消除冲突的页面错误。



实验：查看PFN项

利用内核调试器*!pfn*命令，你可以检查单独的PFN项。你首先需要以参数方式提供此PFN（例如，*!pfn 0*显示第一项，*!pfn 1*显示第二项，依此类推）。下面的例子显示了虚拟地址0x50000的PTE，紧跟着显示了包含此页目录的PFN，然后是实际的页面：

```
kd> !pte 50000
00050000  - PDE at C0300000          PTE at C0000140
          contains 00700067        contains 00DAA047
        pfn 00700 --DA--UWV      pfn 00DAA --D---UWV
kd> !pfn 700
  PFN 00000700 at address 827CD800
  flink  00000004 blink / share count 00000010 pteaddress C0300000
  reference count 0001                                 color 0
  restore pte 00000080  containing page        00030  Active     M
  Modified

kd> !pfn daa
  PFN 00000DAA at address 827D77F0
```


www.TopSage.com

```
flink  00000077 blink / share count 00000001 pteaddress C0000140
reference count 0001                                  color 0
restore pte 00000080  containing page        00700  Active     M
Modified
```

除了PFN数据库以外，表7.22中的一些系统变量也描述了总体的物理内存状态。

表7.22 描述了物理内存的系统变量

变 量	说 明
MmNumberOfPhysicalPages	系统中可用物理页面的总数
MmAvailablePages	系统中可用页面的总数——零化列表、空闲列表和备用列表中页面的总和
MmResidentAvailablePages	若每个进程都处于最小工作集的大小，则可用物理页面的总数为该变量的值

低内存通知和高内存通知

Windows XP和Windows Server 2003为用户模式进程提供了一种方法，使得当物理内存很小和/或很富裕的时候，这些进程能被通知到。这些信息可被用来确定内存使用量是否合适。例如，如果可用内存的数量很小，则应用程序可以减少内存的消耗。如果可用内存数量很大，则应用程序可以申请更多的内存。

为了能够在小内存和大内存的条件下被通知到，可调用*CreateMemoryResourceNotification*函数，并指定期望接收到小内存或大内存通知。这样得到的句柄，可用在任何一个等待函数中。当内存变小（或大）的时候，等待结束，因此通知此线程该条件已满足。此外，利用*QueryMemoryResourceNotification*函数可以在任何时候查询系统的内存条件状况。

这种通知机制是由内存管理器实现的，它将一个全局的命名事件对象*LowMemoryCondition*或*HighMemoryCondition*置成有信号状况。这些命名对象不在通常的\BaseNamedObjects对象管理器目录中，而是在一个特殊的称为\KernelObjects的目录中。当小（或大）内存条件被检测到时，适当的事件被置成有信号状态，因而就会唤醒那些正在等待该事件的线程。

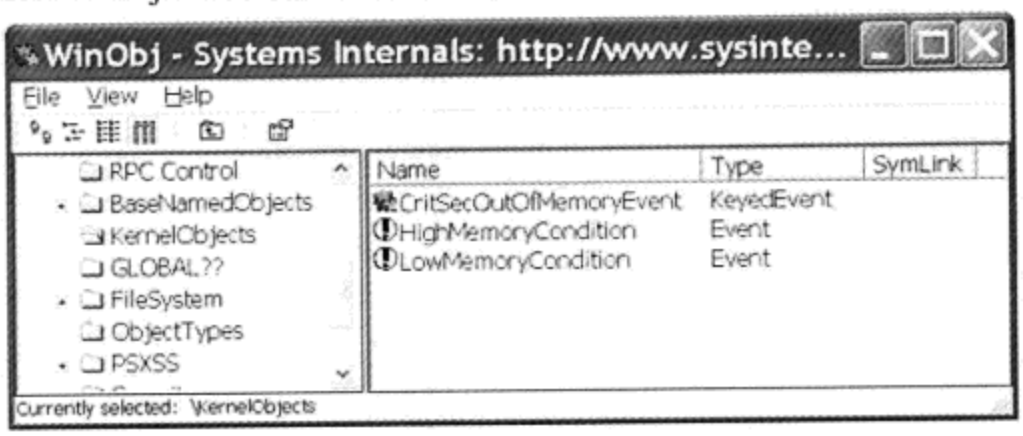
可激发小内存资源通知事件的默认可用内存数量，近似等于每4GB为32MB，但最多不超过64MB。激发大内存资源通知事件的默认可用内存数量是默认小内存值的3倍。你可以改变这些值，只要在HKEY_LOCAL_MACHINE\System\CurrentControlSet\Session Manager\Memory Management下加入DWORD注册表值*LowMemoryThreshold*和*HighMemoryThreshold*，并指定用于小内存阈值和大内存阈值的兆字节数。

www.TopSage.com



实验：查看内存资源通知事件

要想看到内存资源通知事件，你可以运行*www.sysinternals.com*的Winobj，并点击KernelObjects目录夹。你将会看到，在右边的窗格上显示了小内存和大内存条件事件。



如果你双击任何一个事件，你可以看到有多少个句柄和/或引用指向这些对象。

如果想要知道当前系统中哪些进程已经请求了内存资源通知，则可以搜索句柄表，查找指向“LowMemoryCondition”或“HighMemoryCondition”的引用。你也可以利用进程管理器的Find菜单并选择查找句柄来做到这一点，或者使用资源工具箱中的Oh.exe工具（有关句柄表的描述，请参见第3章中关于“对象管理器”的章节）。

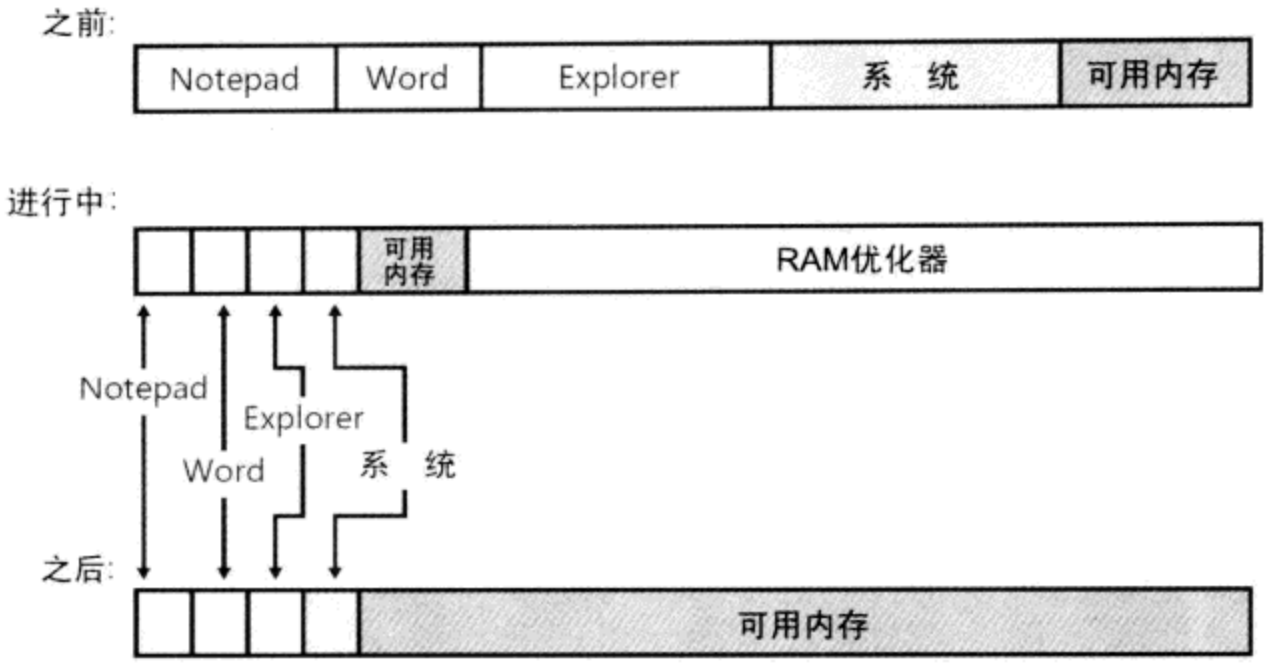
RAM优化器：是事实，还是谎言

当你在浏览Web页面的时候，你可能已经看到过浏览器弹出诸如“Defragment your memory and improve performance（整理内存碎片和改进性能）”和“Minimize application and system failures and free unused memory（使应用程序和系统失败减到最小，释放不用的内存）”这样的窗口。其给出的链接引导你到达那些承诺可以帮你做这些事情甚至更多功能的工具上。这些工具真的管用吗？

内存优化器给出的UI中，往往显示了一个标记为“可用内存”的图，以及一条线代表了一个阈值，当小于该阈值时，此产品就会有所行动。通常另一条线显示了当优化器运行的时候，它试图要释放的内存的数量。你往往可以配置一个或者两个级别，也可以手工触发内存优化过程，或者安排何时进行优化。有些工具还显示了当前系统上正在运行的进程。当一个安排好的优化任务运行时，这些工具的可用内存计数器通常开始增长，有时候增长得非常显著，这暗示着该工具正在释放内存，以供你的应用程序来使用。但是，它实际在做的事情是，让有用的内存被零化，因而人为地增加可用内存。

RAM优化器的工作原理是，先申请大量的虚拟内存，然后释放这些内存。下图显示了RAM优化器在一个系统上的效果：

www.TopSage.com



“之前”这一横条说明了在优化之前的工作集和可用内存。“进行中”这一横条显示了RAM优化器创建了一个很高的内存需求，它招致在短时间内有大量的页面错误。作为响应，内存管理器增加了RAM优化器的工作集。这一工作集的扩充是以牺牲可用内存为代价的，而且，当可用内存变小的时候，它也需要牺牲其他进程工作集为代价。“之后”这一横条显示了，当RAM优化器释放了其内存以后，内存管理器如何将所有分配给RAM优化器的页面全部转移到空闲页面列表中（这些页面最终会被零页面线程零化，并转移到零页面列表中），因而贡献给了可用内存值。许多优化器隐藏了在第一步中发生的可用内存急剧减少的事实，但是，如果你在优化过程中运行任务管理器的话，你通常可以在内存减少的时候看到这一情况。

获得更多的可用内存可能看起来像是一件好事，但其实不然。当RAM优化器强迫可用内存计数器变大的时候，它们同时也迫使其他进程的数据和代码换出内存。譬如，如果你正在运行Word，那么，所打开的文档和程序代码作为Word的工作集的一部分，在优化之前驻留在物理内存中，而优化之后当你继续编辑文档时，这些文档和程序代码又要从磁盘上读回来。在服务器上性能退化可能会非常严重，因为缓存在备用列表和系统工作集中的文件数据（以及任何正在运行的服务器应用程序所用到的代码和数据）将会被丢弃。

有些厂商为他们的RAM优化器产品作了额外的声明。你可能会看到的一种声明是，他们的产品释放的内存是那些无用进程所消耗掉的不必要的内存，比如在任务条右侧格子中运行的那些进程。只有在“这些进程在优化时刻有可变大小的工作集”的情况下，这一声明才有可能是真的。然而，因为Windows自动修剪空闲进程的工作集，所以，所有这样的声明都是不真实的。内存管理器已经处理了所有必要的内存优化。

www.TopSage.com

RAM优化器的开发人员还声明他们的产品能够整理内存碎片。先申请大量虚拟内存，然后再释放这些内存，这样做以后，很容易想见，作为一种副作用，可能会导致大块连续的可用内存。然而，因为虚拟内存对进程掩盖了物理内存的布局结构，所以，这些进程不能直接受益于“由连续的物理内存来支撑虚拟内存”的事实。尽管系统中的进程得到了连续可用的内存，但是，随着这些进程经历了工作集的修剪和增长，它们的“从虚拟内存到物理内存的映射”将会变得非常碎。有了连续可用的内存，在一种情形下可以导致性能的提高：为了尽可能地使CPU内存高速缓存的行为最为理想，内存管理器使用一种称为**页面着色**（*page coloring*）的机制，以确定将空闲列表或零化列表中的哪个页面分配给一个进程。然而，因为可用物理内存是连续的而带来的一点点好处，远不及因为从内存中丢弃有用代码和数据而造成的负面影响。

最后，厂商们通常声称RAM优化器可以重新收回那些由于泄漏而丢失的内存。这可能是最显然的虚假声明了。内存管理器始终知道哪些物理内存和虚拟内存属于一个进程。然而，如果一个进程申请了内存，然后又由于一个错误而不释放该内存（即发生了内存泄漏），则内存管理器不可能了解到所分配的内存将不会再被访问到，它必须等到该进程退出才能回收此内存了。即使对于那些尚未退出的泄漏进程，作为工作集修剪的一个自然结果，内存管理器最终可以从这些进程的工作集中，偷取到那些分配给被泄漏虚拟内存的物理页面。这些进程将泄漏的页面发送到页面文件中，从而让它们的物理页面用于其他方面。因此，内存泄漏对于可用物理内存只有很有限的影响。真正的影响在于虚拟内存消耗上，在任务管理器中即“PF Usage”和“Commit Charge”两个量。除了杀死那些正在消耗虚拟内存的进程以外，没有其他的工具可以解决虚拟内存消耗问题。

总之，常识告诉我们，如果RAM优化器是可能的（并且能够由这么多如此小巧的新贵程序来实现），那么Microsoft的开发人员早就将这些技术集成到内核中了。

www.TopSage.com

7.12 本章总结

在本章中，我们已经看过了Windows内存管理器是如何实现虚拟内存管理的。如同大多数现代操作系统一样，每个进程都有一个私有的地址空间；进程的内存被保护起来，与另一个进程的内存隔离开；但进程之间又允许有效地、安全地共享内存。一些高级的功能，比如映射文件的引入、按稀疏方式分配内存的能力，也是可以使用的。Windows环境子系统使得内存管理器的大多数功能，都可以让应用程序通过Windows API来使用。

只要有可能，内存管理器的实现依赖于延迟计算来避免执行耗时的、不必要的操作，除非有需要的时候才执行这些操作。它也具备自我调节、适应的能力，既可以在大的多处理器服务器上工作，也可以在单处理器桌面工作站上工作。

我们在本章中没有讲到的关于内存管理器的一个方面，是它与缓存管理器的紧密集成。我们将在第11章介绍缓存管理器。在此之前，我们先来看一看Windows的安全机制。

www.TopSage.com

www.TopSage.com

Security

第8章 安全性

在有多个用户可以访问同样物理资源或网络资源的环境中，防止对敏感数据的未授权访问是非常必要的。操作系统以及单独的用户，必须保护文件、内存和配置信息，避免它们被意外地查看和修改。操作系统安全性包括各种机制，诸如账户、口令和文件的保护。它也包含了一些不那么显然的机制，比如保护操作系统避免遭受破坏、阻止非特权用户执行一些动作（例如重新引导计算机），以及不允许用户程序恶意地影响其他用户的程序或者操作系统自身。

在本章中我们将解释，Microsoft Windows的设计和实现的每一个方面，是如何在强安全性的严格需求下受到影响的。

安全等级

能够用明确定义的标准来核定软件（包括操作系统），这有助于政府、企业和家庭用户保护其计算机系统中存储的私有资产和个人数据。美国和许多其他国家目前使用的安全等级标准是CC（Common Criteria，公共标准）。然而，为了理解Windows内部设计的安全能力，了解一个曾经影响了Windows设计的安全等级系统是非常有用的，这就是TCSEC（Trusted Computer System Evaluation Criteria，可信计算机系统评估标准）。

可信计算机系统评估标准（TCSEC）

美国国家计算机安全中心（NCSC, *www.radium.ncsc.mil/tpep/*）是1981年建立的，它是美国国防部（DoD）的国家安全局（NSA，National Security Agency）的一部分。NCSC的一个目标是建立一个安全等级范围，如表8.1所列。通过该列表来指明商业操作系统、网络组件和可信应用程序所提供的保护程度。这些安全等级是1983年定义的，你可以在*www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html*找到，它们通常称为“橘皮书（the Orange Book）”。

www.TopSage.com

表8.1 TCSEC安全等级

等 级	说 明
A1	可检验的设计
B3	安全域
B2	结构化的保护
B1	分类的安全保护
C2	受控制的访问保护
C1	自主访问保护（已不再使用）
D	最少的保护

TCSEC标准是由“信任度”等级构成的，高信任度是在低信任度的基础上，加入了更为严格的保护和验证要求而形成的。没有一个操作系统满足A1或“可检验的设计”等级。虽然有少数操作系统已经获得了B类等级，但就实际而言，对于一个通用操作系统来说，C2等级已经足够了，也是最高的等级了。

1995年7月，Windows NT 3.5（工作站和服务器） Service Pack3成为第一个获得C2等级的Windows NT版本。1999年3月，Windows NT 4 Service Pack 3获得了英国政府ITSEC（Information Technology Security）组织的E3等级，这相当于美国的C2等级。1999年11月，Windows NT 4 Service Pack 6a在单独配置和网络配置的环境下均获得了C2等级。

以下是C2安全等级的关键要求，现在它们仍然被认为是任何一个安全操作系统的核心要求。

- **安全的登录设施** 这要求用户可以被惟一标识，而且只有当他们被通过某种方式认证身份以后，才可以被授予对该计算机的访问权；
- **自主的访问控制** 这使得资源的所有者能够决定谁可以访问该资源，以及他们可以对它做些什么。资源的所有者为单个用户或者一组用户授予各种访问权限；
- **安全审计** 它提供了相应的能力来检测和记录与安全相关的事件，或者任何想要创建、访问或删除系统资源的意图。登录标识符记录了所有用户的身份，从而使得很容易跟踪谁在执行一个未授权的动作；
- **对象重用保护** 它防止用户看到其他用户已经删除的数据，或者访问到其他用户原先使用过后来又释放了的内存。例如，在某些操作系统中，可能发生这样的情形：先创建一个特定长度的新文件，然后检查该文件的内容，这样就可以看到磁盘上该文件所分配到的位置处原来的数据。这些数据可能是以前存储在另一用户的文件中但现在已被删除的敏感信息。对象重用保护设施可以防止这种潜在的安全漏洞，其做法是，在将对象（包括文件和内存）分配给一个用户以前，先对它们进行初始化。

www.TopSage.com

Windows也满足B等级安全性的以下两个要求：

- **可信路径功能** 它防止特洛伊木马程序在用户登录的时候能够截取到用户的名称和口令。在Windows中，可信路径功能是以Ctrl+Alt+Del登录注意序列（logon-attention sequence）的形式来做到的，非特权应用程序不可能截取到此序列。该键击序列也称为安全注意序列（SAS，secure attention sequence），它总是弹出一个登录对话框，所以，企图成为特洛伊木马的程序很容易被识别出来。当输入SAS时，显示假登录对话框的特洛伊木马程序将会被绕过去；
- **可信设施管理** 它要求针对各种管理功能有单独的账户角色作为支持。例如，针对管理工作（管理员）、负责计算机备份的用户和标准用户分别提供单独的账户。

Windows通过它的安全子系统和相关的组件来满足以上所有这些要求。

CC（公共标准，Common Criteria）

1996年1月，美国、英国、德国、法国、加拿大和荷兰发布了他们联合开发的CCITSE（Common Criteria for Information Technology Security Evaluation，信息技术安全评估公共标准）安全评估规范。CCITSE通常称为CC（Common Criteria，公共标准），是公认的针对产品安全评估的多国标准。CC的主页位于*csrc.nist.gov/cc*。

CC比TCSEC信任等级更加灵活，但在结构上更加接近于ITSEC，而并非TCSEC。CC包含了**保护轮廓**（PP，*Protection Profile*）的概念，从而可将安全需求聚集到一些更加容易规定和比较的集合中；它还包含另一个概念：安全目标（ST，Security Target），安全目标包含一组安全需求，PP可以引用这些安全需求。

Windows 2000在2002年10月，被认定为满足受控访问PP（Controlled Access PP）的安全需求，这是TCSEC C2的等价级别，同时它也被认定为满足公共标准CC的功能和保障需求。在*niap.nist.gov/cc-scheme.html*上可以找到相应的细节。关于Windows 2000满足的额外安全需求，可在*niap.nist.gov/cc-scheme/CCEVS-VID402-ST.pdf*的Windows 2000公共标准安全目标（Windows 2000 Common Criteria Security Target）中找到。以下是一些较为显著的，虽未包含在受控访问PP中，但包含在Windows安全目标中的安全需求。

- **基于密码学的自主访问控制功能** 这是由Windows 2000加密文件系统和Windows 2000数据保护API来实现的；
- **额外用户数据对象的自主访问控制策略** 这样的用户数据对象包括桌面、窗口站（由Windows 2000窗口系统实现）和活动目录对象（由Windows 2000目录服务实现）；

www.TopSage.com

- **Windows 2000作为分布式操作系统，实现了内部复制能力以确保与安全相关的数据项在各个物理上分离的部分之间保持同步** 这是Windows 2000目录服务使用了一种多份主数据松散一致性的技术，连同聚集目录复制模型来实现的；
- **物理磁盘空间的资源有效利用** 这是由Windows 2000 NTFS实现的；
- **交互式会话锁定和初始用户登录过程的可信路径** 这是由Windows 2000 Winlogon实现的；
- **Windows 2000作为分布式操作系统，实现了内部数据传输保护，以保护这些数据在各个物理上分离的部分之间传输时不会被泄漏和修改** 这是由Windows 2000 IPSEC服务实现的；
- **系统化的缺陷修补措施**这是通过Microsoft安全响应中心（Microsoft Security Response Center）和Windows支持工程组（Windows Sustained Engineering Team）所采取的操作程序来实现的。

在写作本书的时候，Windows XP Embedded、Windows XP Professional和Windows Server 2003仍然在进行CC公共标准的评估。这次评估的范围远远超出了Windows 2000的评估范围。CC权威机构目前已认可Windows XP和Windows Server 2003（Standard、Enterprise和Datacenter Edition）作为下列技术类型进行评估，如*niap.nist.gov/cc-scheme/in_evaluation.html*所示：

- 分布式操作系统；
- 敏感数据保护；
- 网络管理；
- 目录服务；
- 防火墙；
- VPN （虚拟私有网络）；
- 桌面管理；
- 公钥基础设施；
- 公钥证书发放和管理；
- 嵌入式系统操作系统。

8.1 安全系统组件

以下是实现Windows安全性的核心组件和数据库。

- **安全引用监视器（SRM）** 这是Windows执行体（\Windows\System32\Ntoskrnl.exe）中的一个组件，它负责：定义访问令牌数据结构来表示一个安全环境、执行对象的安全访问检查、管理特权（用户权限），以及生成所有的结果安全审计消息；

www.TopSage.com

- **本地安全权威子系统（Lsass）** 这是一个运行\Windows\System32\Lsass.exe映像文件的用户模式进程，它负责本地系统安全策略（比如允许哪些用户登录到本地机器上、口令策略、授予用户和用户组的特权，以及系统安全审计设置）、用户认证，以及发送安全审计消息到事件日志（Event Log）中。本地安全权威服务（Lsasrv—\Windows\System32\Lsasrv.dll）是Lsass加载的一个库，它实现了这些功能中的绝大部分；
- **Lsass策略数据库** 这是包含本地系统安全策略设置的数据库。该数据库被存储在注册表中，位于HKLM\SECURITY的下面。它包含了诸如此类的信息：哪些域是可信任的，从而可以认证用户的登录请求；谁允许访问系统，以及如何访问（交互式登录、网络登录，或者服务登录）；分配给谁哪些特权；执行哪一种安全审计。Lsass策略数据库也保存一些“秘密”，包括域登录（domain logon）在本地缓存的信息，以及Windows服务的用户-账户登录信息（关于Windows服务的更多信息，请参见第5章）；
- **安全账户管理器（SAM）服务** 这是一组子例程，它们负责管理一个数据库，该数据库包含了本地机器上已定义的用户名和组。SAM服务是在\Windows\System32\Samsrv.dll中实现的，它运行在Lsass进程中；
- **SAM数据库** 这是一个数据库，在非域控制器的系统上，它包含了已定义的本地用户和用户组，连同它们的口令和其他属性。在域控制器上，此SAM保存了该系统的管理员恢复账户的定义及其口令。该数据库被存储在注册表的HKLM\SAM的下面；
- **活动目录** 这是一个目录服务，它包含了一个数据库，其中存放了关于域中对象的信息。这里，**域**（*domain*）是由一组计算机和与它们相关联的安全组构成的，每个安全组被当作单个实体来管理。活动目录存储了有关该域中的对象的信息，这样的对象包括用户、组和计算机。域用户和组的口令信息和特权也被存储在活动目录中，而活动目录则是在一组被指定为该域的**域控制器**（*domain controller*）的机器之间进行复制的。活动目录运行在Lsass进程中，其实现模块为\Windows\System32\Ntdsa.dll。有关活动目录的更多信息，请参见第13章；
- **认证包** 这包括运行在Lsass进程和客户进程环境中的动态链接库（DLL），它们实现了Windows的认证策略。认证DLL负责检查一个给定的用户名和口令是否匹配，如果匹配的话，则向Lsass返回有关用户安全标识的细节信息，以供LSASS利用这些信息来生成令牌；
- **登录进程（Winlogon）** 这是一个用户模式进程，它运行的是\Windows\System32\Winlogon.exe，负责响应SAS和管理交互式登录会话。例如，当用户登录的时候，Winlogon创建一个用户的外壳（用户界面）进程；

www.TopSage.com

- **图形化标识和认证（GINA）** 这是一个用户模式DLL，运行在Winlogon进程中；Winlogon用它来获得一个用户的名称和口令，或者智能卡的PIN码。标准GINA是\Windows\System32\Msgina.dll；
- **网络登录服务（Netlogon）** 这是一个Windows服务（\Windows\System32\Netlogon.dll），它建立起与域控制器之间的安全通道。诸如“交互式登录（如果域控制器在运行Windows NT 4），或者LAN管理器（LAN Manager）和NTLM（NT LAN Manager）v1和v2的认证验证”之类的安全请求，均通过此安全通道来发送；
- **内核安全设备驱动程序（KSecDD）** 这是一个内核模式函数库，它实现了本地过程调用（LPC)接口；其他的内核模式安全组件，包括加密文件系统（EFS，Encrypting File System），利用该接口与用户模式下的Lsass进行通信。KSecDD位于\Windows\System32\Drivers\Ksecdd.sys中。

图8.1显示了其中一些组件和它们管理的数据库之间的关系。

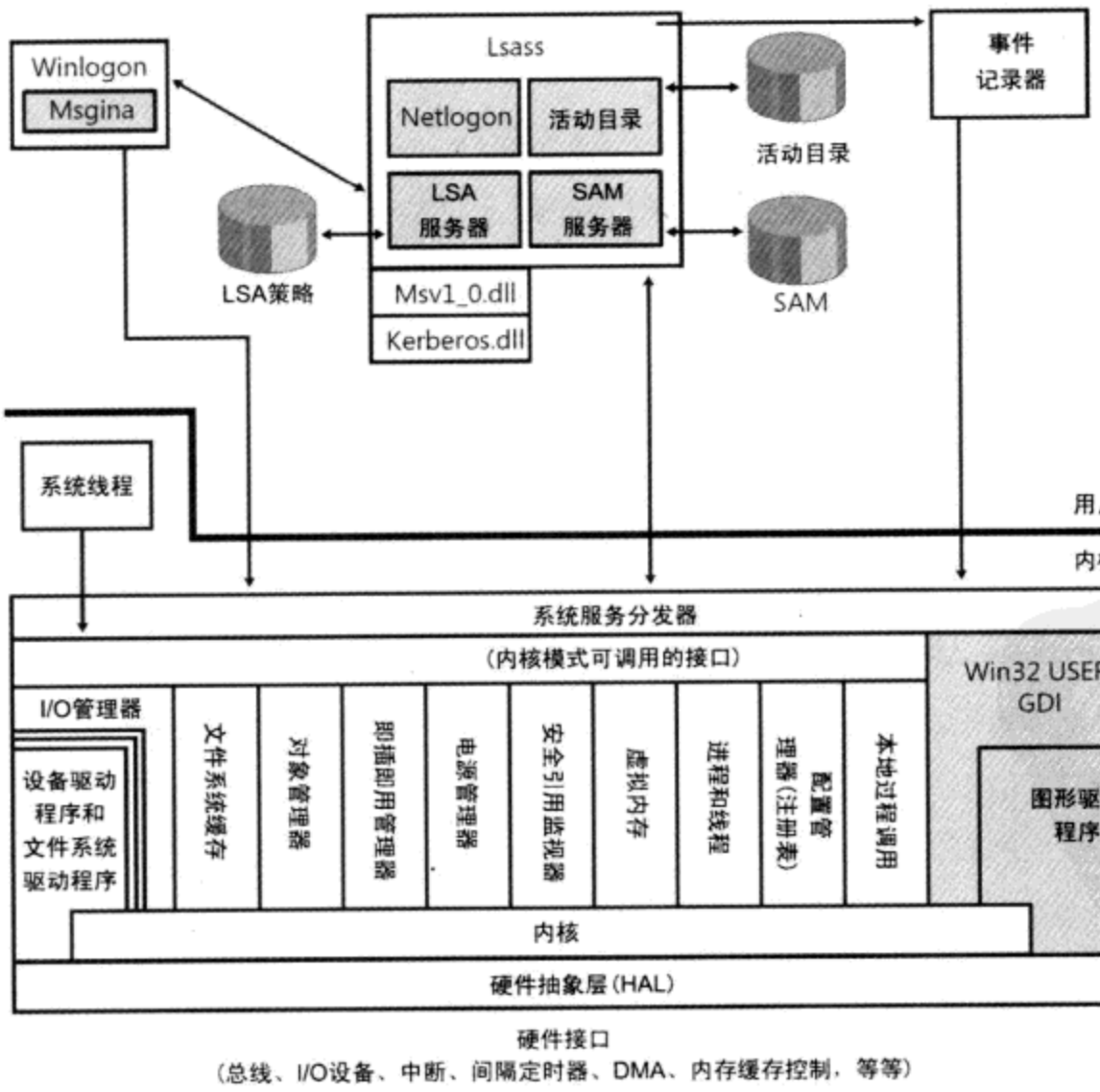


图8.1 Windows安全组件

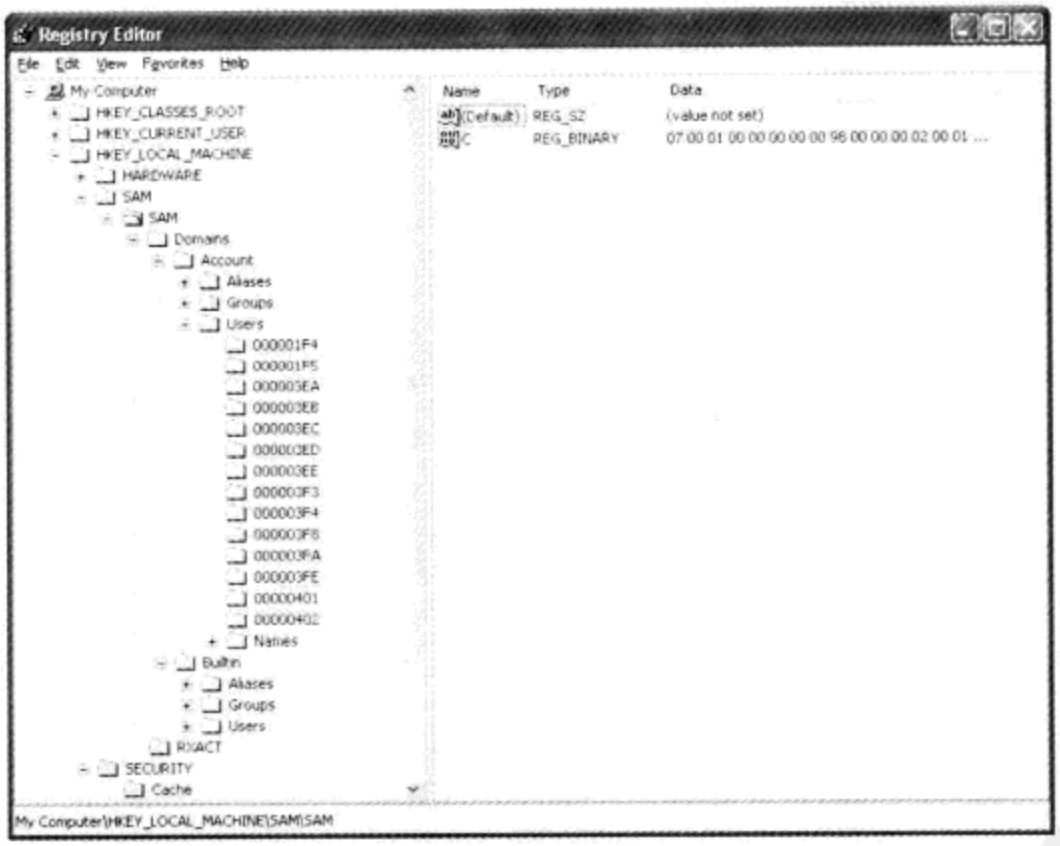
www.TopSage.com



实验：观察HKLM\SAM和HKLM\Security

注册表中的SAM键和Security键都有与之相关联的安全描述符，从而防止Local System之外的其他账户访问这些键。为了探查这些键，一种访问途径是重置它们的安全性设置，但是，这样会削弱系统的安全性。另一种办法是，在Local System账户下执行Regedit.exe：*www.sysinternals.com*提供的PsExec工具支持一个选项，可让你在Local System账户下激发进程。如下所示，使用PsExec命令来运行Regedit，从而无需改变SAM和Security键的安全性设置就可以访问SAM和Security数据库：

```
C:>psexec -s -i -d c:\windows\regedit.exe
```



SRM运行在内核模式下，而Lsass运行在用户模式下，它们利用第3章中讲述的LPC设施来进行通信。在系统初始化过程中，SRM创建了一个名为SeRmCommandPort的端口，Lsass连接到此端口上。当Lsass进程启动的时候，它创建一个名为SeLsaCommandPort的LPC端口。SRM连接到此端口上，从而建立起两者之间的私有通信端口。SRM创建一个共享内存区，用于传递那些超过256字节长度的消息，它在连接调用中把该共享内存区的句柄传递过去。在系统初始化过程中，一旦SRM和Lsass相互连接起来，它们就不再监听各自的连接端口。因此，后来的用户进程无法连接至这些端口，从而避免了它们可能会造成恶意的破坏——它们的连接请求永远也不会得到满足。

www.TopSage.com

图8.2显示了当系统初始化完成以后通信路径的情况。

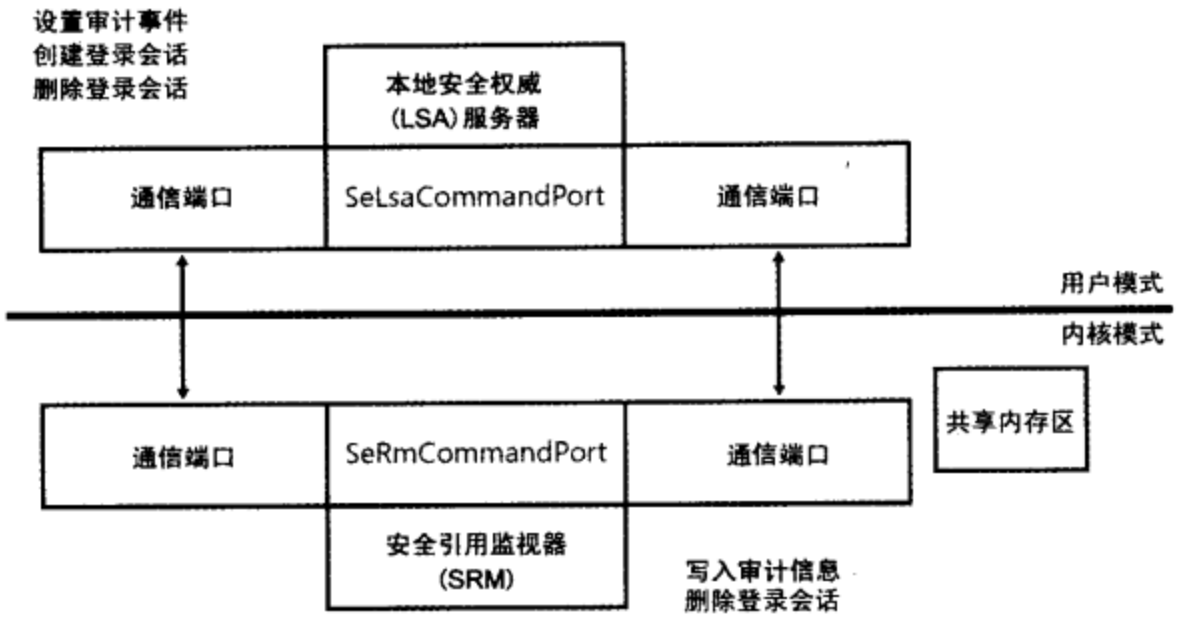


图8.2 SRM和Lsass之间的通信

8.2 保护对象

对象保护和访问日志记录是自主访问控制和审计功能的本质所在。在Windows平台上能被保护的对象包括文件、设备、邮件槽、管道（命名的和匿名的）、作业、进程、线程、事件、带键的事件、事件对、互斥体、信号量、共享内存区、I/O完成端口、LPC端口、可等待的定时器、访问令牌、卷（volume）、窗口站、桌面、网络共享体、服务、注册表键、打印机和活动目录对象。

那些被导出到用户模式中的系统资源，它们在内核中都是以对象的形式来实现的，所以，Windows对象管理器在实施对象安全性方面扮演了关键的角色（有关对象管理器的更多信息，请参见第3章）。 为了控制谁可以操纵一个对象，安全系统首先必须确定每个用户的身份。之所以需要保证用户的身份，是因为，Windows要求在访问任何系统资源以前，必须存在已认证的登录会话。当一个进程请求一个对象句柄时，对象管理器和安全系统利用调用者的安全标识，来确定是否将一个句柄赋予此调用者，从而允许该进程访问此对象。

正如本章后面将要讨论的那样，一个线程可以假设它所使用的安全环境不同于其进程的安全环境。这一机制称为**模仿**（*impersonation*）。当一个线程处于模仿模式的时候，系统的安全验证机制将使用该线程的安全环境，而不是其进程的安全环境。当一个线程不在模仿模式的时候，安全验证机制退回到使用该线程所属进程的安全环境。在脑子里记住以下这一点是非常重要的：一个进程中的所有线程共享了同一个句柄表，所以，当一个线程打开某个对象的时候，即使该线程当前处于模仿模式，该进程的所有线程也都可以访问此对象。

www.TopSage.com

访问检查

Windows安全模型要求当一个线程打开一个对象的时候，它必须指定它期望在该对象上执行哪些类型的动作。系统根据一个线程期望的访问要求来执行访问检查，如果它允许此访问，则向该线程的进程赋予一个句柄，因而该线程（或者该进程中的其他线程）可以通过此句柄在对象上执行进一步的操作。正如在第3章中所介绍的那样，对象管理器在进程的句柄表中记录下每个句柄被授予的访问许可。

一种可触发对象管理器执行安全访问验证过程的事件是：当一个进程使用一个名称来打开一个已有对象的时候。当一个对象被通过名称来打开时，对象管理器在对象管理器名字空间中执行一个查找此指定对象的操作。如果该对象不在某个从属名字空间（比如配置管理器的注册表名字空间，或者一个文件系统驱动程序的文件系统名字空间）中，那么，一旦对象管理器找到了该对象，即调用内部函数*ObpCreateHandle*。正如此函数名所暗示的，*ObpCreateHandle*在进程的句柄表中创建一个项目，并且将它与此对象关联起来。然而，只有当另一个对象管理器函数*ObpIncrementHandleCount* 指示该线程有访问该对象的许可权时，*ObpCreateHandle*才会调用执行体函数*ExCreateHandle*来创建此句柄。真正执行安全访问检查的是另一个对象管理器函数*ObCheckObjectAccess*，它把结果返回给*ObpIncrementHandleCount*。

*ObpIncrementHandleCount*把当前正执行打开操作的线程的安全凭证、该线程所请求的访问类型（读、写、删除，等等），以及一个指向该对象的指针，传递给*ObCheckObjectAccess*。*ObCheckObjectAccess*首先锁住该对象的安全性和该线程的安全环境。对象安全性锁可防止在访问检查进行过程中，系统中的另一个线程改变该对象的安全性。作用在线程安全环境上的锁则可以防止该进程的另一个线程，或者另一个进程，在安全验证进行过程中改变该线程的安全身份。然后，*ObCheckObjectAccess*调用该对象的安全（Security）方法，以获得它的安全设置信息（关于对象方法的介绍，请参见第3章）。对安全方法的调用，可能会导致调用另一个执行体组件中的函数。然而，许多执行体对象依赖于系统的默认安全性管理支持。

如果一个执行体组件定义了一个对象，它并不想改写SRM的默认安全策略，那么，它将该对象类型标记为“具有默认安全性”。无论何时，当SRM调用一个对象的安全方法时，它首先检查一下，看该对象是否具有默认安全性。具有默认安全性的对象，其安全信息存储在头部，它的安全方法是*SeDefaultObjectMethod*。一个不依赖于默认安全性的对象，它必须管理自己的安全信息，并且提供一个专门的安全方法。依赖于默认安全性的对象有：互斥体、事件、信号量；而文件对象则是一个不使用默认安全性的对象的例子。I/O管理器定义了文件对象类型，它让文件系统驱动程序来管理其所属文件的安全性（当然文件系统驱动程序也可以选择不实现安全管理能力）。因此，当系统查询一个文件对象的安全性时，它假设该文件对象代表了NTFS卷上的一个文件，那么，I/O管理器的文件对象安全方法利用NTFS文件系统驱动程序来

www.TopSage.com

获得该文件的安全性。然而，请注意，当文件被打开的时候，*ObCheckObjectAccess*并不被执行，因为它们处于从属名字空间中；只有当一个线程显式地查询或者设置一个文件的安全性（例如，通过Windows函数*SetFileSecurity*或*GetFileSecurity*）时，系统才会调用一个文件对象的安全方法。

在获得了一个对象的安全信息以后，*ObCheckObjectAccess*调用SRM函数*SeAccessCheck*。*SeAccessCheck*是一个处于Windows安全模型核心的函数。*SeAccessCheck*接受的输入参数中包括该对象的安全信息、通过*ObCheckObjectAccess*捕获的线程安全身份，以及当前线程所请求的访问模式。*SeAccessCheck*返回True或False，取决于该线程是否已被授予以此模式来访问该对象的权限。

另一种引发对象管理器执行安全访问验证过程的事件是：一个进程通过一个已有的句柄来引用一个对象。这样的引用往往是间接发生的，比如，一个进程通过Windows API来操纵一个对象，它把对象句柄传递进去。例如，一个线程在打开一个文件时，可以向允许它读取该文件的对象请求访问权。如果按照该线程的安全环境和该文件的安全设置的指示，该线程有访问该对象的许可，那么，对象管理器在该线程所属进程的句柄表中，创建一个代表该文件的句柄。对象管理器将此进程所得到的访问权与该句柄存储在一起。

随后，该线程可能企图通过Windows函数*WriteFile*，将该文件的句柄当作一个参数传递给它，从而对该文件执行写操作。*WriteFile*通过Ntdll.dll调用系统服务*NtWriteFile*，而*NtWriteFile*使用对象管理器函数*ObReferenceObjectByHandle*，以便根据该句柄获得一个指向此文件对象的指针。*ObReferenceObjectByHandle*接受的一个参数代表了调用者想要获得的针对目标对象的访问权。*ObReferenceObjectByHandle*在进程的句柄表中找到了该句柄项以后，它将调用者请求的访问权与该文件被打开时刻所授予的访问权进行比较。在本例中，*ObReferenceObjectByHandle*将指示此写操作应该失败，因为当文件被打开时，调用者没有得到写访问权。

Windows安全功能也允许Windows应用程序定义它们自己的私有对象，并且请求SRM的服务，以便在这些对象上也强制使用Windows的安全模型。对象管理器和其他执行体组件通常使用许多内核函数来保护它们自己的对象，而这些内核函数也被导出成Windows用户模式API。例如，与*SeAccessCheck*等价的用户模式函数是*AccessCheck*。因此，Windows应用程序可以充分利用此安全模型的灵活性，从而与Windows中已有的认证和管理界面透明地集成在一起。

SRM安全模型的本质是一个包含三项输入的等式，这三项输入是：一个线程的安全身份、该线程想要获得的对一个对象的访问权，以及该对象的安全设置。输出是“是”或“否”，表

www.TopSage.com

明了此安全模型是否授予该线程它所期望的访问权。下面的章节更加详细地描述了这些输入项，然后再介绍该安全模型的访问验证算法。

安全标识符（SID）

Windows并非使用名称（可能是惟一的，也可能不是）来标识系统中执行各种动作的实体，相反，Windows使用了**安全标识符**（SID，*security identifier*）。用户有SID，本地用户组、域中的用户组、本地计算机、域和域成员也都有SID。SID是一个可变长度的数值，包含三部分：一个SID结构版本号、一个48位标识符机构值，以及可变数量的32位子机构值或者**相对标识符**（RID，*relative identifier*）值。机构值标识了发放此SID的代理机构，此代理机构通常是一个Windows本地系统或者一个域。子机构值标识了此发放机构的托管者，RID只不过是Windows在一个公共SID基础上创建惟一SID的一种方法而已。因为SID有足够的长度，而且Windows负责为每个SID生成真正随机的值，所以，在世界上任何地方的机器上或者域中，Windows几乎不可能两次发放同样的SID。

当以文本方式来显示每个SID时，SID以S为前缀，每个部分用连字号来分隔，例如：

S-1-5-21-1463437245-1224812800-863842198-1128

在这个SID中，版本号是1，标识符机构值是5（Windows安全权威机构），余下部分是4个子机构值加上一个RID（1128）。此SID是一个域SID，但是，该域中的计算机的SID也具有相同的版本号、标识符机构值，以及同样数量的子机构值。

当你安装Windows的时候，Windows Setup程序给计算机发放一个SID。Windows为该计算机上的本地账户分配SID。每个本地账户SID是以此源计算机的SID为基础的，只是尾部有一个RID。用户帐户和组的RID是从1000开始的，对于每个新的用户或组，RID向上递增1。类似地，用于创建Windows新域的工具，即Dcpromo.exe，为它所创建的域分配SID。Windows为域中新账户发放SID的方法是，基于域SID，再追加一个RID（也是从1000开始，对于每个新用户或组，RID递增1）。例如，RID为1029，则表明了这是该域中发放的第29个SID。

Windows在计算机或域SID的基础上，加上一个预定义的RID，从而构成一些知名的SID，并且将这些SID发放给许多预定义的账户和组。例如，administrator账户的RID是500，guest账户的RID是501。例如，一台计算机的本地管理员账户以该计算机的SID为基础，再追加上500作为RID：

S-1-5-21-13124455-12541255-61235125-500

www.TopSage.com

Windows也定义了许多内置的本地SID和域SID来代表组。例如，标识每一个账户或者所有账户的组Everyone，或者World，其SID是S-1-1-0。另一个代表一个组的SID例子是网络组，它代表了所有从网络登录到一台机器上的用户。网络组的SID是S-1-5-2。表8.2是从Platform SDK文档中复制过来的，它显示了一些基本的知名SID、它们的数值以及用途。

最后，Winlogon为每个交互式的登录会话创建一个惟一的登录SID。登录SID的一个典型用途是，用于只在一个客户的登录会话过程中才允许访问的访问控制项（ACE）中。例如，一个Windows服务可以使用*LogonUser*函数来开始一个新的登录会话。而且，它可以从*LogonUser*函数返回的访问令牌中提取出对应的SID。然后，该服务可以将此SID用在一个“允许该客户的登录会话访问交互式窗口站和桌面”的ACE中。登录会话的SID是S-1-5-5-0，其RID是随机生成的。

表8.2 一些知名的SID

SID	组	用 途
S-1-1-0	Everyone	一个包含了所有用户的组
S-1-2-0	Local	登录到系统本地（物理）终端上的用户
S-1-3-0	Creator Owner ID	一个安全标识符，将由创建新对象的用户的标识符来替代。该SID被用在一个可继承的ACE中
S-1-3-1	Creator Group ID	标识了一个可被替代的安全描述符，将由创建新对象的用户所属的主组来替代。也被用在一个可继承的ACE中



实验：使用PsGetSid来查看账户的SID

通过运行PsGetSid实用工具（来自*www.sysinternals.com*），你可以很容易地看到当前正在使用的账户的SID。它的界面形式如下：

```
C:\>psgetsid -?

PsGetSid displays the machine SID for the local or remote Windows system.

Usage: psgetsid [\\RemoteComputer [-u Username [-p Password]]] [account | SID]
     -u        Specifies optional user name for login to
               remote computer.
     -p        Specifies optional password for user name. If you omit this
               you will be prompted to enter a hidden password.
     account   PsGetSid will report the SID for the specified user account
               rather than the computer.
     SID       PsGetSid will report the account for the specified SID.
```


www.TopSage.com

通过PsGetSid的命令行选项，可以将机器账户和用户账户的名称转译成对应的SID，或者反过来将SID转译成账户名。

如果你在运行PsGetSid时不提供任何选项，那么，它打印出被分配给本地计算机的SID。由于Administrator账户的RID总是500，基于这样的事实，你只需简单地把机器SID追加上500，再作为命令行参数传递给PsGetSid，就可以确定该管理员账户的名称（万一系统管理员出于安全的原因，而更改了该账户的名称）。

为了得到一个域账户的SID，请输入用户名，并且将域名作为其前缀：

```
c:\>psgetsid redmond\daryl
```

你只要将域名作为参数传递给PsGetSid，就可以确定一个域的SID：

```
c:\>psgetsid Redmond
```

最后，通过检查你自己账户的RID，你可以知道，在你的域中或者在你的机器上（取决于你是在使用一个域账户或者本地机器账户），已创建的安全账户的数量至少等于从你的RID中减去999。你只要用任何一个你想要查询的RID，将SID传递给PsGetSid，就可以确定该RID被分配给哪个账户了。如果PsGetSid报告说，该SID没有对应的账户名称，而且此RID低于你账户的RID，那么，你可以知道，该RID对应的账户已经被删除了。

例如，为了找到第28个RID被分配给哪个账户了，请把域SID追加上“-1027”，再传递给PsGetSid：

```
c:\>psgetsid S-1-5-21-1787744166-3910675280-2727264193-1027
Account for S-1-5-21-1787744166-3910675280-2727264193-1027:
User: redmond\daryl
```

令牌

SRM使用一个称为**令牌**（或**访问令牌**）的对象来标识一个进程或线程的安全环境。安全环境中包含的信息描述了与该进程或线程相关联的特权、账户和组。在登录过程（本章最后讲述）中，Winlogon创建了一个初始的令牌来代表该用户的这一次登录，并且将该令牌附载到它启动的初始进程（默认是Userinit.exe，也可能有多个进程）上。因为子进程在默认情况下继承其创建者令牌的一份拷贝，所以，在该用户的会话中的所有进程都运行在同样的令牌下面。你也可以利用Windows的*LogonUser*函数来生成一个令牌。

www.TopSage.com

然后，你可以把该令牌传递给Windows的*CreateProcessAsUser*函数，从而用它来创建一个进程，并且新的进程运行在通过*LogonUser*函数来登录的用户的安全环境下。*CreateProcessWithLogon*函数也通过创建一个新的登录会话和一个初始进程的方法来创建一个令牌。这正是Runas命令在其他令牌下启动进程的做法。

令牌的大小是不固定的，因为不同的用户账户有不同的特权集合，它们关联的组账户集合也不相同。然而，所有的令牌包含了同样的信息，如图8.3所示。

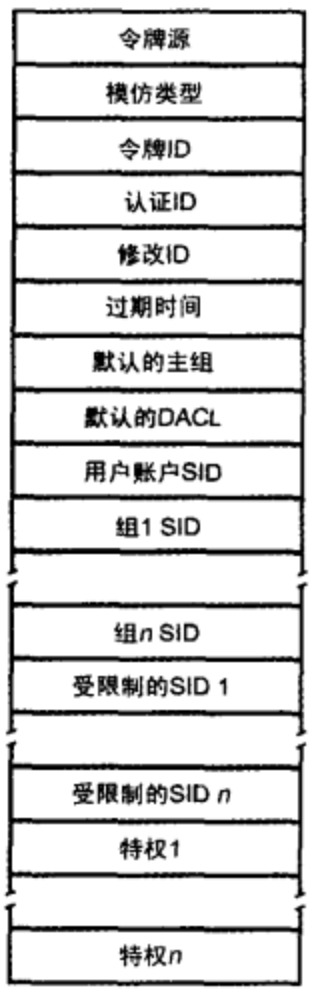


图8.3 访问令牌

Windows中的安全机制用到了令牌中的两部分信息来决定哪些对象可以被访问，以及哪些安全操作可以被执行。第一部分由令牌的用户账户SID和组SID域构成。SRM使用这些SID来决定一个进程或线程是否可以获得指定的、对于一个被保护对象（比如一个NTFS文件）的访问许可。

令牌中的组SID说明了一个用户的账户是哪些组的成员。当服务器应用程序在执行客户请求的一些动作时，它可以禁止某些特定的组，以限制一个令牌的凭证。像这样禁止一个组，其效果几乎等同于这个组没有出现在令牌中（禁止SID也被当作安全访问检查的一部分，如本章后面所描述）。

www.TopSage.com

在一个令牌中，决定该令牌的线程或进程可以做哪些事情的第二部分信息是特权集。一个令牌的特权集是一组与该令牌关联的权限的列表。关于特权的一个例子是，与该令牌关联的进程或线程具有关闭该计算机的权限。本章后面更加详细地讲述了特权。一个令牌默认的主组域和默认的自主访问控制列表（DACL）域是指这样一些安全属性：当一个进程或线程使用该令牌时，Windows将这些安全属性应用在它所创建的对象上。Windows通过将这些安全信息包含在令牌中，从而使得进程或者线程可以很方便地创建一些具有标准安全属性的对象，因为进程和线程不需要为它所创建的每个对象请求单独的安全信息。

每个令牌的类型将主令牌（一个标识了进程安全环境的令牌）和模仿令牌（线程为了临时接受另一个安全环境[通常是另一个用户的]而使用的一种令牌）区别开来。模仿令牌包含了一个模仿级别，它指明了该令牌中哪一种模仿类型是有效的（本章后面讲述了模仿）。

令牌中其他的域主要用于提供各种信息。令牌源域包含了一段短的文本信息，描述了创建此令牌的实体。如果一个程序想要知道一个令牌是从哪儿来的，它可以使用令牌源中的信息，以便在各种来源（比如Windows会话管理器、网络文件服务器，或者远过程调用（RPC）服务器）之间进行区分。令牌标识符是一个本地惟一标识符（LUID），也是SRM创建该令牌时分配给它的。Windows执行体维护了执行体LUID，这是一个计数器，执行体通过它为每个令牌分配一个惟一数值标识符。

令牌的认证ID是另一种LUID。令牌的创建者调用*LsaLogonUser*函数时，它分配该令牌的认证ID。如果创建者没有指定一个LUID，则Lsass从执行体LUID中获得此LUID。Lsass为所有从同一个初始登录令牌继承下来的令牌，拷贝其认证ID。一个程序可以获得一个令牌的认证ID，从而确定该令牌是否与它检查过的其他令牌属于同一个登录会话。

每次当一个令牌的特征被修改时，执行体LUID会刷新修改ID。应用程序通过检查一个令牌的修改ID，可以发现自上次使用该安全环境以来，其中是否发生了变化。

令牌包含了一个过期时间域，虽然该域自从Windows NT 3.1以来就一直存在，但是在Windows NT技术中一直未被使用。Windows的将来版本可能允许令牌只在过期以前的一段时间内才有效。请考虑这样一个用户：系统管理员为他设置了账户过期时间。目前，如果该用户登录到系统中，而且在账户过期时仍然登录在系统中，那么，系统将允许该用户继续访问资源。要阻止该用户访问资源的惟一办法是，强迫用户从该机器上注销登录。如果Windows支持令牌过期，那么，在令牌过期时间到达以后，系统就可以阻止用户打开资源了。

www.TopSage.com



实验：查看访问令牌

内核调试器的*dt _TOKEN*命令显示了内部令牌对象的格式。虽然该结构与Windows API安全函数返回的用户模式令牌结构有所不同，但是，其中的域是类似的。有关令牌的更进一步信息，请参考Platform SDK文档中的描述。

下面的输出来自于内核调试器的*dt _TOKEN*命令：

```
kd> dt _TOKEN
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId : _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER
   +0x030 TokenLock        : Ptr32 _ERESOURCE
   +0x034 ModifiedId       : _LUID
   +0x03c SessionId        : Uint4B
   +0x040 UserAndGroupCount : Uint4B
   +0x044 RestrictedSidCount : Uint4B
   +0x048 PrivilegeCount   : Uint4B
   +0x04c VariableLength   : Uint4B
   +0x050 DynamicCharged   : Uint4B
   +0x054 DynamicAvailable : Uint4B
   +0x058 DefaultOwnerIndex : Uint4B
   +0x05c UserAndGroups    : Ptr32 _SID_AND_ATTRIBUTES
   +0x060 RestrictedSids   : Ptr32 _SID_AND_ATTRIBUTES
   +0x064 PrimaryGroup     : Ptr32 Void
   +0x068 Privileges       : Ptr32 _LUID_AND_ATTRIBUTES
   +0x06c DynamicPart      : Ptr32 Uint4B
   +0x070 DefaultDacl      : Ptr32 _ACL
   +0x074 TokenType        : _TOKEN_TYPE
   +0x078 ImpersonationLevel : _SECURITY_IMPERSONATION_LEVEL
   +0x07c TokenFlags       : UChar
   +0x07d TokenInUse       : UChar
   +0x080 ProxyData        : Ptr32 _SECURITY_TOKEN_PROXY_DATA
   +0x084 AuditData        : Ptr32 _SECURITY_TOKEN_AUDIT_DATA
   +0x088 VariablePart     : Uint4B
```

通过*!token*命令，你可以检查一个进程的令牌。在*!process*命令的输出中，你可以找到一个进程的令牌，如下所示：

```
kd> !process 380 1
!process 380 1
Searching for Process with Cid == 380
PROCESS ff8027a0 SessionId: 0  Cid: 0380  Peb: 7ffdf000  ParentCid: 0124
  DirBase: 06433000  ObjectTable: ff7e0b68  TableSize:  23.
  Image: cmd.exe
  VadRoot 84c30568 Clone 0 Private 77. Modified 0. Locked 0.
  DeviceMap 818a3368
  Token                             e22bc730
  ElapsedTime                       14:22:56.0536
  UserTime                          0:00:00.0040
```


www.TopSage.com

```
  KernelTime                          0:00:00.0100
  QuotaPoolUsage[PagedPool]           13628
  QuotaPoolUsage[NonPagedPool]        1616
  Working Set Sizes (now,min,max)  (261, 50, 345)(1044KB, 200KB, 1380KB)
  PeakWorkingSetSize                  262
  VirtualSize                         11 Mb
  PeakVirtualSize                     11 Mb
  PageFaultCount                      313
  MemoryPriority                      FOREGROUND
  BasePriority                        8
  CommitCharge                        86

kd> !token e22bc730
_TOKEN e22bc730
TS Session ID: 0
User: S-1-5-21-1787744166-3910675280-2727264193-500
Groups:
 00 S-1-5-21-1787744166-3910675280-2727264193-513
    Attributes - Mandatory Default Enabled
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-32-544
    Attributes - Mandatory Default Enabled Owner
 03 S-1-5-32-545
    Attributes - Mandatory Default Enabled
 04 S-1-5-5-0-92587
    Attributes - Mandatory Default Enabled LogonId
 05 S-1-2-0
    Attributes - Mandatory Default Enabled
 06 S-1-5-14
    Attributes - Mandatory Default Enabled
 07 S-1-5-4
    Attributes - Mandatory Default Enabled
 08 S-1-5-11
    Attributes - Mandatory Default Enabled
Primary Group: S-1-5-21-1787744166-3910675280-2727264193-513
Privs:
 00 0x000000017 SeChangeNotifyPrivilege           Attributes - Enabled Default
 01 0x000000008 SeSecurityPrivilege               Attributes -
 02 0x000000011 SeBackupPrivilege                 Attributes -
 03 0x000000012 SeRestorePrivilege                Attributes -
 04 0x00000000c SeSystemtimePrivilege             Attributes -
 05 0x000000013 SeShutdownPrivilege               Attributes -
 06 0x000000018 SeRemoteShutdownPrivilege         Attributes -
 07 0x000000009 SeTakeOwnershipPrivilege          Attributes -
 08 0x000000014 SeDebugPrivilege                  Attributes -
 09 0x000000016 SeSystemEnvironmentPrivilege      Attributes -
 10 0x00000000b SeSystemProfilePrivilege          Attributes -
 11 0x00000000d SeProfileSingleProcessPrivilege   Attributes -
 12 0x00000000e SeIncreaseBasePriorityPrivilege   Attributes -
 13 0x00000000a SeLoadDriverPrivilege             Attributes - Enabled
 14 0x00000000f SeCreatePagefilePrivilege         Attributes -
 15 0x000000005 SeIncreaseQuotaPrivilege          Attributes -
 16 0x000000019 SeUndockPrivilege                 Attributes - Enabled
 17 0x00000001c SeManageVolumePrivilege           Attributes -
```


www.TopSage.com

```
Authentication ID:          (0,169e4)
Impersonation Level:        Anonymous
TokenType:                  Primary
Source: User32              TokenFlags: 0x9 ( Token in use )
Token ID: 3a86d0d           ParentToken ID: 0
Modified ID:                (0, 3a86d0f)
RestrictedSidCount: 0       RestrictedSids: 00000000
```

在进程管理器（Process Explorer，可从*www.sysinternals.com*下载）中，通过进程属性对话框的安全属性页面，你可以间接地查看令牌的内容。该对话框显示了当前进程的令牌中包含的组和特权。

模仿（impersonation）

在Windows的安全模型中，模仿是一项强大的、广为使用的安全特性。Windows也在它的客户/服务器程序设计模型中使用模仿特性。例如，服务器应用程序可以导出诸如文件、打印机或数据库这样的资源。客户若想要访问一个资源，则可以向服务器发送一个请求。当服务器接收到此请求时，它必须确保该客户有权在目标资源上执行它所期望的操作。例如，如果一台远程机器上的用户试图删除一个NTFS共享体中的文件，那么，导出此共享体的服务器必须决定，是否允许该用户删除此文件。要想确定一个用户是否有此许可，很显然的过程是，服务器查询该用户的账户和组SID，并且扫描该文件上的安全属性。这一过程对于程序来说是非常琐碎的，也很容易出错，而且也不允许再透明地支持新的安全特性。因此，Windows提供了模仿服务来简化服务器的工作。

模仿机制使得一个服务器可以告诉SRM，它现在临时接受了一个发出资源请求的客户的安全轮廓。然后，服务器可以代表该客户访问这些资源，SRM仍然执行访问验证过程。通常，服务器可以访问的资源比客户要多得多，所以，服务器在模仿过程中要丢失某些安全凭证（security credential）。然而，反过来也是有可能的：服务器在模仿过程中可能会获得一些新的安全凭证。

服务器只是在处理此模仿请求的线程的内部才模仿一个客户。线程的控制数据结构中包含了一个针对模仿令牌的可选项。然而，代表一个线程实际安全凭证的主令牌，在进程的控制结构中总是可以访问的。

Windows提供了几种机制来实现模仿。如果一个服务器通过命名管道与一个客户进行通信，则服务器可以使用Windows API函数*ImpersonateNamedPipeClient*来告诉SRM，它想要模仿该管道另一端的用户。如果服务器通过DDE（Dynamic Data Exchange）或者RPC与客户进行通信，那么，它可以利用*DdeImpersonateClient*和*RpcImpersonateClient*，发出类似的模仿请求。一个线程利用*ImpersonateSelf*函数，可以创建一个模仿令牌，不过，该模仿令牌只不过是其进程令牌的一个拷贝。然后，为了禁止某些SID或特权，该线程可以改变它的模仿令牌。最后，一

www.TopSage.com

个SSPI（Security Support Provider Interface，安全支持提供者接口）包可以通过*ImpersonateSecurityContext*来模仿其客户。SSPI实现了一个网络安全模型，比如LAN Manager v2或Kerberos。

服务器完成其任务以后，它恢复到原来的主安全轮廓上。这些模仿形式，对于在接收到客户请求的情形下执行一些特定的动作，或者确保对象访问操作能被正确地审计下来，都是非常方便的（例如，所生成的审计纪录给出的是被模仿客户的身份，而不是服务器进程的身份）。这些模仿形式的缺点是，它们不能在客户的环境中执行一个完整的程序。而且，模仿令牌不能访问网络共享体上的文件或打印机，除非这是一个委托级别的模仿（稍后讲述）并且有足够的凭证来向远程机器证明其身份，或者该文件或打印机共享体支持空会话（空会话是指由匿名登录产生的会话）。

如果必须在客户的安全环境中执行整个应用程序，或者必须访问网络资源，那么，该客户必须登录到系统中。Windows API函数*LogonUser*支持这一动作。*LogonUser*的输入参数有一个账户名、口令、域或计算机名称、登录类型（比如交互方式、批方式或服务方式），以及一个登录提供者。它返回一个主令牌。服务器线程可以接受此令牌作为一个模仿令牌，或者，服务器可以启动一个程序，该程序用客户的凭证作为其主令牌。从安全的角度来看，利用交互式*LogonUser*返回的令牌来创建的进程，比如通过*CreateProcessAsUser* API来创建的进程，看起来就好像用户交互式地登录到该机器上，再创建此进程一样。这种方法的缺点是，服务器必须获得该用户的账户名和口令。如果服务器要在网络上传输这些信息，则它必须非常安全地加密这些信息，以便恶意用户即使偷听网络流量也不会得到它们。

为了防止滥用模仿机制，Windows不允许服务器在没有得到客户同意的情况下执行模仿。客户进程在连接到服务器的时候可以指定一个**安全服务质量**（SQOS，*security quality of service*），以此来限制服务器进程可以执行的模仿级别。一个进程可以在Windows的*CreateFile*函数中指定SECURITY_ANONYMOUS、SECURITY_IDENTIFICATION、SECURITY_IMPERSONATION和SECURITY_DELEGATION作为标记。每个级别都使得服务器可以在客户的安全环境中执行以下不同类型的操作：

- *SecurityAnonymous*是最为限制的模仿级别——服务器不能模仿或者识别出客户；
- *SecurityIdentification*级别允许服务器得到客户的身份（SID）和特权，但是服务器不能模仿该客户；
- *SecurityImpersonation*级别允许服务器在本地系统上识别和模仿该客户；
- *SecurityDelegation*是最为随意的模仿级别，它允许服务器在本地系统或远程系统上模仿该客户；Windows NT和更早的版本不完全支持SecurityDelegation级别的模仿。

www.TopSage.com

如果客户没有设置模仿级别，则Windows默认选择SecurityImpersonation级别。*CreateFile*函数也接受SECURITY_EFFECTIVE_ONLY和SECURITY_CONTEXT_TRACKING作为与模仿设置相关的修饰符。

- SECURITY_EFFECTIVE_ONLY防止一个服务器在模仿的时候，允许或禁止一个客户的特权或组成员特性；
- SECURITY_CONTEXT_TRACKING指定了，客户对其安全环境所作的任何改变都被反映到当前正在模仿该客户的服务器中。如果该选项未被指定的话，则服务器只接受在模仿时刻的客户安全环境，以后不再接收任何改变。只有当客户和服务器进程在同一个系统上的时候，这一选项才能起作用。

受限制的令牌

受限制的令牌（*restricted token*）是通过*CreateRestrictedToken*函数，在主令牌或模仿令牌的基础上创建的。受限制的令牌是其来源令牌的一份拷贝，但可能有下面的修改：

- 从该令牌的特权集中可以删除一些特权；
- 该令牌中的SID可以被标记为**仅仅拒绝**（*deny-only*）；
- 该令牌中的SID可以被标记为**受限制的**（*restricted*）。

仅仅拒绝的SID和受限制的SID的行为待稍后再讲述。当一个应用程序想要在一个减弱的安全级别上模仿一个客户（主要是在运行不可信的代码时出于保险起见）时，受限制的令牌是非常有用的。例如，受限制的令牌可以将“系统停机”特权从令牌中去掉，从而防止在此受限制令牌的安全环境中执行的代码重新引导系统。



实验：看一看受限制的令牌

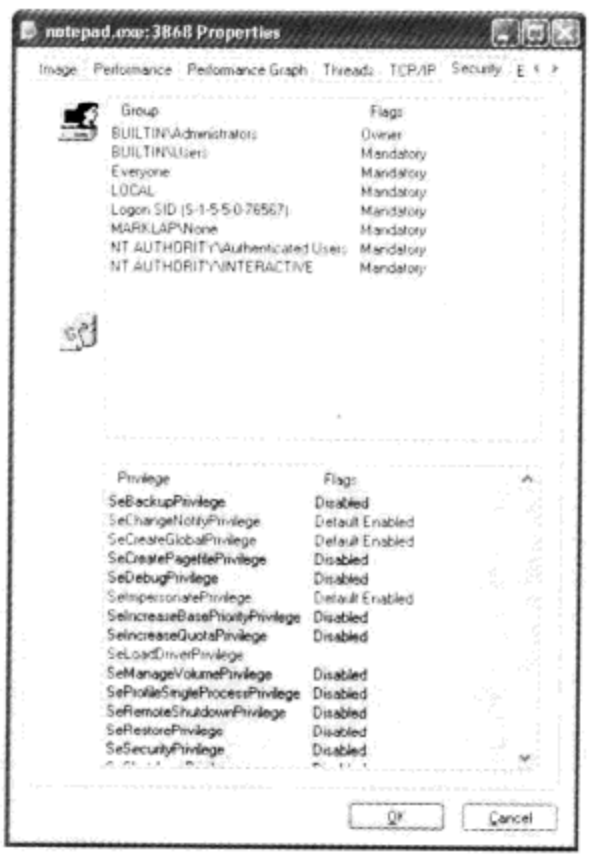
在Windows XP或Windows Server 2003上，按照下面的步骤你可以让Explorer创建一个在受限制的令牌环境中运行的进程。

1. 在你的桌面上创建一个指向\Windows\Notepad.exe的快捷文件。
2. 编辑该快捷文件的属性，选中“Run With Different Credentials”检查框。注意，该检查框的描述文本告诉你，你可以以你自己的身份来运行该程序，同时保护你的计算机避免受到未授权的程序行为的危害。
3. 关闭快捷文件的属性对话框，启动此快捷方式。
4. 在Run As对话框显示时，接受默认的选择：在当前的账户下运行该程序，并且保护你的计算机避免受到未授权程序行为的危害。

www.TopSage.com

5. 运行进程管理器，查看你刚刚启动的Notepad进程的属性对话框中的Security页面。注意，该令牌中包含了一些受限制的SID和一个“仅仅拒绝”SID，而且它只有一个特权。在下面屏幕快照图中，左侧的属性针对的是一个在非限制的令牌下运行的Notepad进程，而右侧的属性针对的是一个采用以上步骤启动起来的Notepad进程。





你看到的受限制的令牌有以下一些副作用。

- 除了SeChangeNotifyPrivilege，其他的特权都被去掉了；
- Administrators或Power Users的SID被标记为Deny-Only（仅仅拒绝）。这一SID消除了以下情形下的资源访问：由于存在一个匹配的“拒绝访问”ACE而导致管理员对资源的访问被拒绝，但可能由于在安全描述符中先有一个ACE授予管理员组的访问权限，从而又允许管理员访问该资源；
- RESTRICTED SID被加入到受限制的SID的列表中。除了该用户的SID和任何Administrators或Power Users SID以外的所有其他令牌SID都加上了相应的受限制项；
- 你运行该进程所使用的账户的SID在列表中没有对应的受限制SID。这意味着，若你的用户账户允许访问一个对象，但是你所属的组中任何某一个组不允许访问该对象，那么，此进程就不能访问该对象了。例如，在\Documents and Settings下面，你的轮廓目录有一个默认的安全描述符，它允许你的账户、管理员组、System账户有访问权。当你试图从刚刚启动的Notepad中打开该目录的时候，你将会被拒绝访问，因为内部的第二遍访问检查（用受限制的SID来执行的访问检查）将失败，原因是，该用户SID没有出现在受限制的SID的列表中。

www.TopSage.com

安全描述符和访问控制

令牌标识了一个用户的凭证，它只是对象安全等式的一部分。对象安全等式的另一部分是与一个对象关联在一起的安全信息，这些信息规定了谁可以在这个对象上执行哪些操作。这些信息所在的数据结构被称为**安全描述符**（*security descriptor*）。一个安全描述符是由以下的属性构成的。

- **版本号** 创建此描述符的SRM安全模型的版本；
- **标志** 一些可选的修饰符，定义了该描述符的行为或特征。例如，SE_DACL_PROTECTED标志可防止该描述符从另一个对象继承安全设置；
- **所有者SID** 所有者的安全ID；
- **组SID** 该对象的主组的安全ID（仅用于POSIX）；
- **自主访问控制列表（DACL）** 规定了谁可以用什么方式访问该对象；
- **系统访问控制列表（SACL）** 规定了哪些用户的哪些操作应该被记录到安全审计日志中。

访问控制列表（ACL，*access-control list*）是由一个头和零个或多个**访问控制项**（ACE，*access-control entry*）结构组成的。共有两种类型的ACL：DACL和SACL。在DACL中，每个ACE包含一个SID和一个访问掩码（和一组标志，稍后再解释）。在DACL中可以出现四种类型的ACE：访问允许、访问拒绝、允许的-对象，以及拒绝的-对象。正如你所想象的，访问允许类型的ACE将访问权限授予一个用户，而访问拒绝类型的ACE则拒绝那些在访问掩码中指定的访问权限。

允许的-对象和访问允许这两种类型之间的差别，以及拒绝的-对象和访问拒绝这两种类型之间的差别是，针对对象的类型仅用于活动目录中。这些类型的ACE有一个GUID（全局惟一标识符）域，它指明了该ACE仅应用于特定的一些对象或子对象（那些具有此GUID标识符的对象）。而且，另一个可选的GUID指明了：在该ACE所应用的活动目录容器对象中，当创建一个子对象时，哪些类型的子对象将会继承此ACE（这里，GUID是一个可保证全球惟一的128位标识符）。

这些单独的ACE所授予的访问权限累积起来，就构成了一个ACL所授予的访问权限的集合。如果在一个安全描述符中没有出现DACL（即*null* DACL），则任何人对于该对象都有完全的访问权限。如果DACL为空（即0个ACE），则没有用户对该对象有访问权限。

在DACL中使用的ACE也有一组标志来控制和指定一些与继承相关的特征。有些对象名字空间有**容器对象**（*container object*）和**叶子对象**（*leaf object*，或仅仅称为**对象**）。一个容器对象可以容纳其他的容器对象和叶子对象，它们都是它的**子对象**（*child object*）。容器对象的例子有：文件系统名字空间中的目录，注册表名字空间中的键。一个ACE中的某些特定的标志控制了该ACE如何被传递到它所在容器的子对象上。表8.3是从Platform SDK中部分摘录的，它列出了ACE标志的继承规则。

www.TopSage.com

表8.3 ACE标志的继承规则

标 志	继承规则
CONTAINER_INHERIT_ACE	容器子对象（比如目录）继承此ACE作为一个有效的ACE。继承得到的ACE也是可继承的，除非NO_PROPAGATE_INHERIT_ACE标志位也被置上了
INHERIT_ONLY_ACE	此标志指明了这是一个仅仅用于继承的ACE，它并不控制其附着对象的访问权限
INHERITED_ACE	此标志指明了该ACE是继承得到的。当系统把一个可继承的ACE传递到一个子对象上时，它设置此标志位
NO_PROPAGATE_INHERIT_ACE	如果该ACE是被一个子对象继承的，则系统清除此继承的ACE中的OBJECT_INHERIT_ACE和CONTAINER_INHERIT_ACE标志。这一动作可防止该ACE再被进一步的后代对象继承
OBJECT_INHERIT_ACE	非容器子对象继承此ACE作为一个有效的ACE。对于容器子对象，该ACE被继承为一个“仅仅被继承（INHERIT-ONLY）”的ACE，除非NO_PROPAGATE_INHERIT_ACE标志位也被置上了

SACL包含两种类型的ACE：**系统审计**ACE和**系统审计-对象**ACE。这些ACE指定了：在该对象上由指定用户或组执行的哪些操作应该被审计下来。审计信息被存储在系统审计日志（Audit Log）中。成功的操作和不成功的企图都要被审计下来。如同DACL中与对象有关的ACE一样，“系统审计-对象”ACE指定了一个GUID，指明了该ACE应用于哪些类型的对象或子对象上，它也可以指定一个可选的GUID，来控制该ACE可以被传递到特定的子对象类型上。如果一个SACL是空（null），则不在该对象上执行任何审计工作（本章后面将要介绍安全审计）。适用于DACL ACE的继承标志同样也适用于系统审计ACE和系统审计-对象ACE。

图8.4给出了一个文件对象及其DACL的一个简化图。

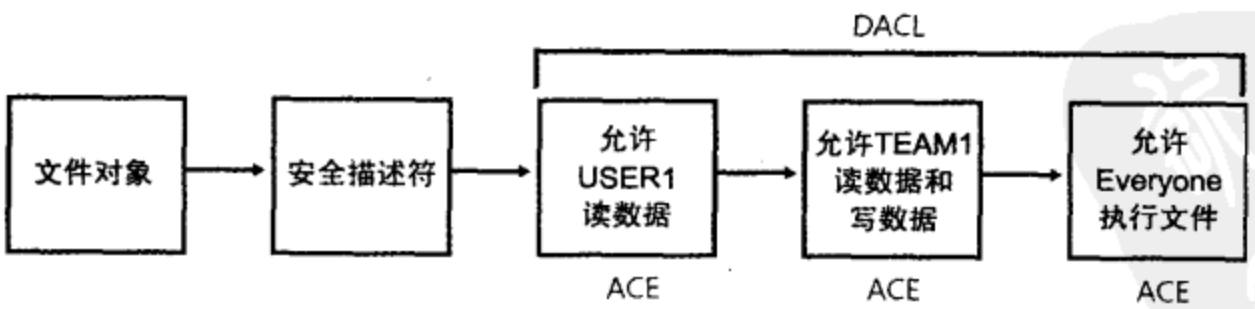


图8.4 自主访问控制列表（DACL）

正如图8.4中所示的，第一个ACE允许USER1读取该文件。第二个ACE允许TEAM1组的成员读/写该文件，第三个ACE授予所有其他的用户（Everyone）执行访问权限。

www.TopSage.com



实验：查看一个安全描述符

许多执行体子系统依赖于对象管理器的默认安全功能来管理其对象的安全描述符。对象管理器的默认安全功能使用安全描述符指针（指对象头中的一个域——译注）来存储这些对象的安全描述符。例如，进程管理器（Process Manager）使用了默认安全性，所以对象管理器分别将进程和线程的安全描述符存储在进程对象和线程对象的对象头部。事件、互斥体和信号量的安全描述符指针也保存了它们的安全描述符。利用“实时内核调试（live kernel debugging）”功能，一旦你能够定位到这些对象的头，就可以查看它们的安全描述符，步骤如下：

1. 启动内核调试器。
2. 输入**!process 0 0**来显示Winlogon的地址（如果系统中有多个活动的终端服务器会话，则将有多个Winlogon在运行）。然后用其中一个Winlogon进程的地址作为参数再输入**!process**：

```
kd> !process 88d0c020
PROCESS 88d0c020  SessionId: 0  Cid: 0bdc    Peb: 7ffdf000  ParentCid: 00a8
    DirBase: 59a58000  ObjectTable: e21df920  HandleCount:  62.
    Image: winlogon.exe
```

3. 利用上述命令的输出中的PROCESS之后的地址作为参数输入**!object**，以显示此对象数据结构：

```
kd> !object 88d0c020
Object: 88d0c020  Type: (8a0ed388) Process
    ObjectHeader: 88d0c008
    HandleCount: 1  PointerCount: 32
```

4. 输入**dt _OBJECT_HEADER**以及上述命令的输出中的对象头域的地址，以显示此对象头数据结构，其中包括安全描述符指针值：

```
lkd> dt _OBJECT_HEADER 88d0c008
   +0x000 PointerCount     : 36
   +0x004 HandleCount      : 1
   +0x004 NextToFree       : 0x00000001
   +0x008 Type             : 0x8a0ed388
   +0x00c NameInfoOffset   : 0 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x20 ' '
   +0x010 ObjectCreateInfo : 0x89e596a0
   +0x010 QuotaBlockCharged : 0x89e596a0
   +0x014 SecurityDescriptor : 0xe242b864
   +0x018 Body             : _QUAD
```

5. 对象头中的安全描述符指针使用低三位作为标志，所以，在下面的**!sd**命令中，输入对象头结构中显示的地址，但是去掉低三位，就可以转储出该安全描述符：

www.TopSage.com

```
lkd> !sd 0xe242b864 & -8
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-21-1787744166-3910675280-2727264193-500
->Group   : S-1-5-21-1787744166-3910675280-2727264193-513
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x40
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x24
->Dacl    : ->Ace[0]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[0]: ->SID: S-1-5-21-1787744166-3910675280-2727264193-500

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x14
->Dacl    : ->Ace[1]: ->Mask : 0x001f0fff
->Dacl    : ->Ace[1]: ->SID: S-1-5-18

->Sacl    :  is NULL
```

此安全描述符包含了两个访问允许的ACE，其中一个指定了管理员的账户（其RID为500，所以可识别出来），另一个指定了System账户（总是S-1-5-18）。如果不解析这些ACE的访问掩码中设置的位，也不确定这些位映射到哪些进程访问类型，则不可能很容易地说出每个账户对于Winlogon的进程对象有哪些访问权限。然而，一旦做了解析并且确定了访问类型，并借助于SDK头文件的帮助，就可以知道这两个账户都有完全访问权限。

ACL分配

为了确定给一个新的对象分配什么样的DACL，安全系统使用以下四条分配规则中第一条可以适用的规则。

1. 如果调用者在创建一个对象的时候显式地提供了一个安全描述符，则安全系统将此安全描述符应用于该对象。如果该对象有一个名称，并且驻留在一个容器对象中（例如，在对象管理器名字空间目录\BaseNamedObjects中一个命名的事件对象），系统将任何可继承的ACE（可能是从该对象的容器对象中传递过来的ACE）合并到该DACL中，除非该安全描述符设置了SE_DACL_PROTECTED标志（该标志阻止继承性）。

www.TopSage.com

2. 如果调用者没有提供一个安全描述符，并且该对象有一个名称，那么，安全系统检查此新对象名称所在的容器对象中的安全描述符。该对象目录的有些ACE可能被标记为可继承的，这意味着这些ACE应该被应用到该对象目录中新创建的对象上。如果出现了这样可继承的ACE，则安全系统将它们编成一个ACL，并附在此新对象上（分别有单独的标志指明了专门让容器对象来继承的ACE，或者由非容器对象来继承的ACE）。
3. 如果调用者没有指定安全描述符，而且该对象也不继承任何ACE，那么，安全系统从调用者的访问令牌中获得一个默认的DACL，并且将它应用到新对象上。在Windows中，有几个子系统具有硬编码的DACL，它们在创建对象的时候将这些DACL分配到对象上（例如，Windows服务、LSA和SAM对象）。
4. 如果没有指定的安全描述符，没有继承的ACE，也没有默认的DACL，那么，系统创建的对象没有DACL，这使得任何人（所有用户和组）都可以完全访问该对象。这一规则的效果，就如同第三条规则在访问令牌中包含一个空（null）的默认DACL的情形是完全一样的。

系统为一个新对象分配SACL时使用的规则，与分配DACL所使用的规则类似，但有一些例外。第一个例外是，继承得到的系统审计ACE并不传递到那些在安全描述符中被设置了SE_SACL_PROTECTED标志的对象上（该标志类似于SE_DACL_PROTECTED，只不过后者保护的是DACL）。第二个例外是，如果没有指定的安全审计SACL，也没有继承的ACE，则该对象上不应用任何SACL。这不同于应用默认DACL的做法，因为访问令牌中没有默认的SACL。

如果一个新的安全描述符中包含了可继承的ACE，则当它被应用到一个容器上时，系统自动地将这些可继承的安全描述符传递到其子对象的安全描述符中（注意：如果一个安全描述符的SE_DACL_PROTECTED标志被允许的话，则它的DACL不再接受继承的DACL ACE；如果安全描述符的SE_SACL_PROTECTED标志被置上了，则它的SACL不再继承SACL ACE）。这些可继承的ACE与已有子对象的安全描述符的合并顺序是，任何被显式应用于该ACL的ACE都位于该对象继承得到的ACE的前面。在传递这些可继承的ACE时，系统使用下面的规则：

- 如果一个不包含DACL的子对象继承了一个ACE，则结果是，该子对象有一个DACL，并且该DACL只包含这一继承的ACE；
- 如果子对象有一个空的DACL，那么，当它继承一个ACE时，其结果是，该子对象有一个DACL，并且该DACL只包含这一继承的ACE；
- 这一条仅针对活动目录中的对象，如果从一个父对象中删除了一个可继承的ACE，则自动继承性使得子对象继承的这一ACE的任何拷贝也被删除；
- 这一条仅针对活动目录中的对象，如果自动继承性导致一个子对象的DACL中的所有ACE都被删除，则该子对象有一个空的DACL，而不是没有DACL。

www.TopSage.com

正如你很快将要看到的，对于Windows的安全模型来说，一个ACL中的ACE的顺序是非常重要的。



> **注** 继承性通常并不是由对象存储体（比如文件系统、注册表，或者活动目录）直接支持的。支持继承性的Windows API（包括*SetSecurityInfo*和*SetNamedSecurityInfo*）是通过调用“安全继承性支持DLL（\Windows\System32\Ntmarta.Dll）”中适当的函数来做到继承性支持的，这些函数知道如何来回访问或者遍历这些对象存储体。

确定访问权

有以下两个算法用于确定对一个对象的访问权：

- 一个算法确定对于该对象的最大允许访问权限，这是通过Windows的*GetEffectiveRightsFromAcl*函数被导出到用户模式中的；
- 另一个算法确定期望的访问模式是否允许，这是通过Windows *AccessCheck*或*AccessCheckByType*函数来做到的。

第一个算法按以下5种方式检查DACL中的内容。

1. 如果该对象没有DACL（即一个null DACL），则该对象没有设置保护，所以安全系统授予所有的访问权限。
2. 如果调用者具有“接管-所有权（take-ownership）”的特权，则安全系统在检查该DACL之前，授予“写-所有者（write-owner）”访问权（稍后解释“接管-所有权”特权和“写-所有者”访问权）。
3. 如果调用者是该对象的所有者，则授予“读-控制（read-control）”和写-DACL访问权限。
4. 对于每个访问-拒绝的ACE，若其中包含的SID与调用者的访问令牌中的SID相符，那么，从已经授予的访问掩码中去掉该ACE的访问掩码。
5. 对于每个访问-允许的ACE，若其中包含的SID与调用者的访问令牌中的SID相符，那么，除非此ACE指定的访问掩码已经被拒绝了，否则将此访问掩码加入到要授予的访问掩码中。

在该DACL中的所有项目都被检查过以后，计算得到的要授予的访问掩码被返回给调用者，作为对该对象的最大允许访问权限。这一掩码代表了此调用者在打开该对象时能够成功地请求到的访问类型的总集合。

上述描述仅适用于该算法的内核模式形式。*GetEffectiveRightsFromAcl*实现的Windows版本算法的不同之处在于，它不执行步骤2，而且它考虑的是单个用户SID或组SID，而不是一个访问令牌。

www.TopSage.com

第二个算法用于在给定调用者访问令牌的情况下，确定它的一个访问请求是否可被准许。在Windows API中凡是涉及安全对象的每个打开（open）函数都有一个参数指定了期望的访问掩码，这是安全等式中的最后一个部分。为了确定调用者是否有访问权，系统执行下面的步骤。

1. 如果该对象没有DACL（即null DACL），则该对象没有设置保护，所以，安全系统授予调用者期望的访问权。
2. 如果调用者具有“接管-所有权（take-ownership）”的特权，则安全系统授予“写-所有者（write-owner）”访问权，然后检查DACL。然而，如果“写-所有者”访问权是一个具有“接管-所有权”特权的调用者所惟一请求的访问权，则安全系统授予此访问权，而不再检查DACL。
3. 如果调用者是该对象的所有者，则授予“读-控制（read-control）”和写-DACL访问权限。如果调用者仅仅请求这些访问权限，则安全系统授予这些访问权限，而不再检查DACL。
4. 从前往后检查DACL中的每一个ACE。对于每个ACE，如果下面的条件之一被满足，则它被处理：
   a. 该ACE是一个“访问-拒绝”ACE，而且该ACE中的SID是一个**可允许的**SID（*enabled* SID，即可被允许或禁止的SID），或者是调用者访问令牌中的一个“仅仅拒绝”SID；
   b. 该ACE是一个“访问-允许”的ACE，而且该ACE中的SID是调用者访问令牌中一个非“仅仅拒绝”类型的可允许SID；
   c. 在第二遍检查受限制的SID过程中，该ACE中的SID与调用者访问令牌中一个受限制的SID相符。
5. 如果它是一个“访问-允许”的ACE，则只要调用者请求了，就将该ACE的访问掩码中的权限授予给它；如果调用者请求的所有访问权限都已经被授予了，则访问检查成功。如果它是一个“访问-拒绝”的ACE，只要调用者请求的任何一个访问权限属于拒绝访问的权限，则调用者对该对象的访问请求被拒绝。
6. 如果已经到达DACL的末尾，而且调用者请求的有些访问权限仍然未被授予，则访问被拒绝。
7. 如果所有的访问权限都被授予了，但是调用者的访问令牌至少有一个受限制的SID，则系统重新扫描该DACL的ACE，以寻找这样的ACE：其访问掩码与用户所请求的访问权限相匹配，并且该ACE的SID与调用者的任何一个受限制的SID相匹配。只有对DACL的这两遍扫描都授予该用户所请求的权限，才允许该用户访问此对象。

以上两个访问-验证算法的行为依赖于允许ACE和拒绝ACE的相对顺序。考虑这样一个对象：它只有两个ACE，第一个ACE指定了一个特定的用户允许完全访问该对象，而第二个ACE拒绝该用户的访问权。如果允许ACE位于拒绝ACE的前面，则用户可以得到该对象的完全访问权，但如果顺序倒过来，则用户无法得到该对象的任何访问权。

www.TopSage.com

早期的诸如*AddAccessAllowedAce*之类的Windows函数，总是把ACE加在DACL的末尾，而这常常不是真正想要的行为，所以，在Windows 2000之前的绝大多数Windows应用程序没有别的办法，只好自己手工构造DACL，将拒绝ACE放在列表的前面。有几个Windows函数，比如*SetSecurityInfo*和*SetNamedSecurityInfo*，按照“拒绝ACE总是在允许ACE之前”的首选顺序来放置ACE。例如，你在编辑NTFS文件和注册表键时所使用的安全编辑器对话框使用了这些函数。*SetSecurityInfo*和*SetNamedSecurityInfo*也将ACE的继承规则应用在相应的安全描述符上。

图8.5显示了一个访问验证的例子，它演示了ACE顺序的重要性。在这个例子中，当一个用户想要打开一个文件时，虽然该对象的DACL中有一个ACE授予它访问权，但是该用户仍然被拒绝访问了，因为在授予用户访问权的ACE之前，有一个ACE拒绝该用户访问（由于该用户属于Writers组）。

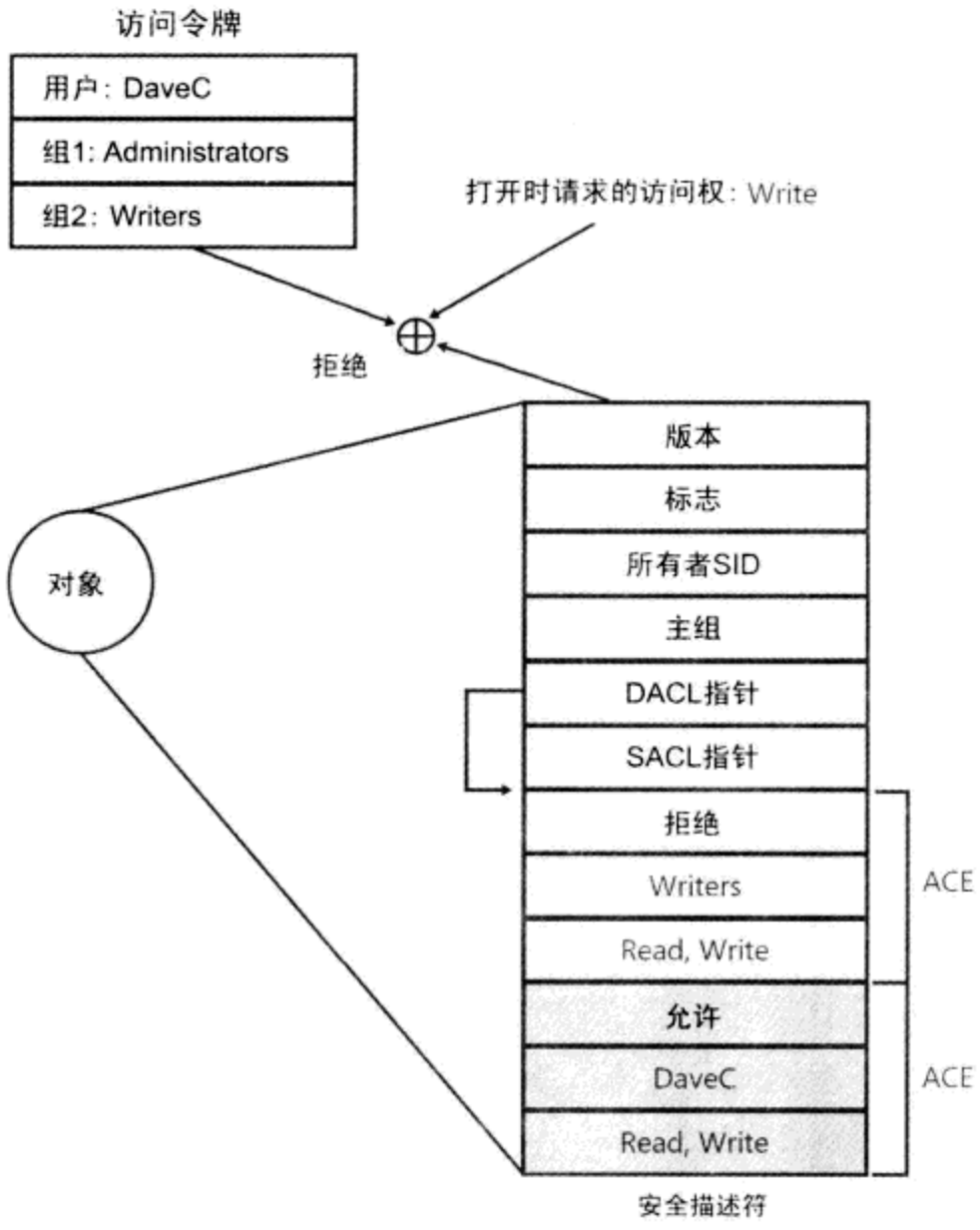


图8.5 访问验证的例子

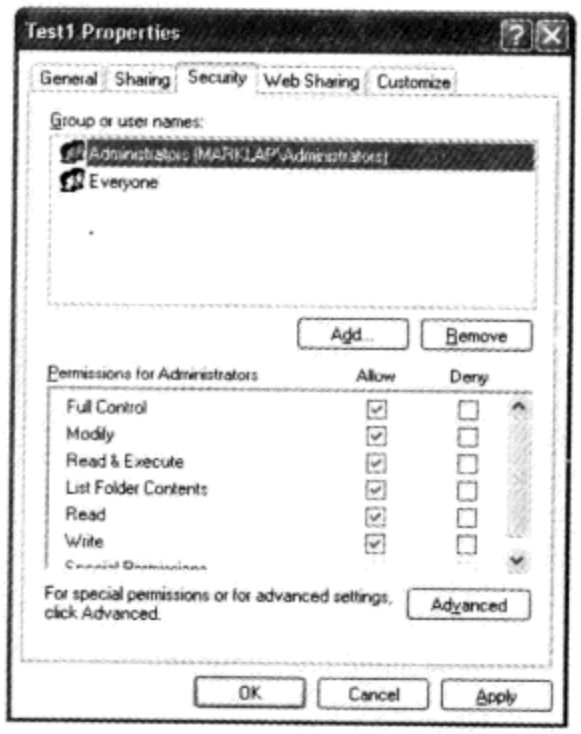
www.TopSage.com

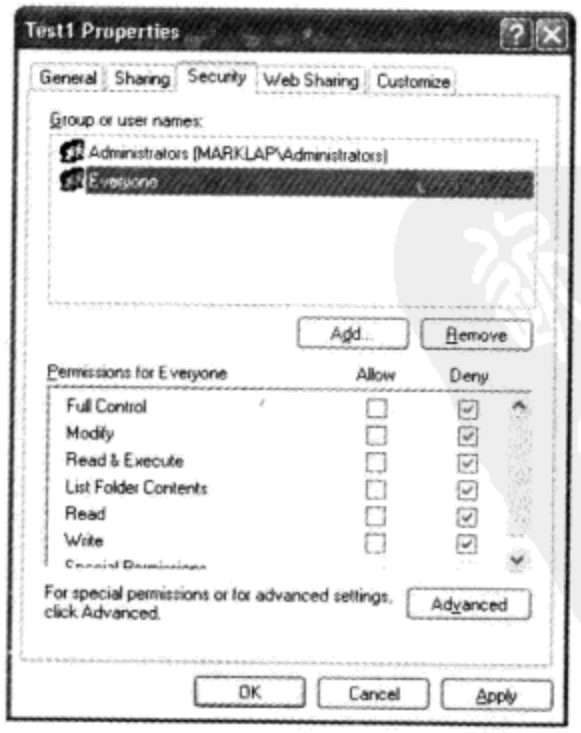
正如前面所说明的，如果每次当一个进程使用一个句柄时，安全系统都要处理DACL，则安全系统将会十分低效，所以，SRM只是在打开一个句柄的时候，而不是每次使用该句柄的时候，才执行访问检查。因此，一旦进程已经成功地打开了一个句柄，安全系统就不能撤销它已经授予的访问权限，即使该对象的DACL发生了变化。同时也要记住，因为内核模式代码使用指针而不是使用句柄来访问对象，所以，当操作系统使用对象的时候并不执行访问检查。换句话说，从安全意义上讲，Windows执行体总是“信任”它自己。

一个对象的所有者总是被授予“写-DACL”访问权，这一事实意味着，要阻止用户访问它们所拥有的对象是不可能的。如果出于某一种理由，一个对象有一个空的DACL（不可访问），则所有者仍然能够使用“写-DACL”访问权限来打开该对象，然后为该对象应用一个新的DACL，其中包含它所期望的访问许可。

一个与GUI安全编辑器有关的警告

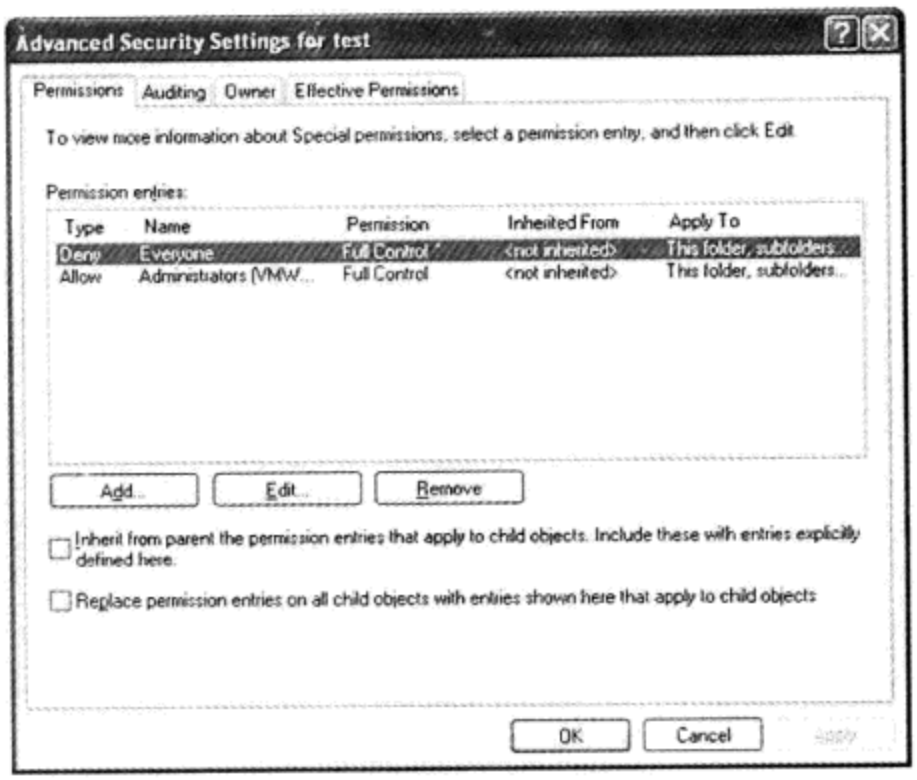
当你使用GUI许可编辑器来修改一个文件对象、注册表对象、活动目录对象或者其他受保护的对象上的安全设置时，主安全对话框向你展示的视图可能会让你对于应用在该对象上的安全性有一种误解。对话框的最上面部分按照字母顺序显示了一些组和用户，这些组或者用户在该对象的访问控制列表中都有相应的ACE项。如果你将完全控制权（Full Control）授予Administrators组，但是对Everyone组又拒绝完全控制权，那么，该列表可能会让你相信，Administrators组的访问-允许ACE在Everyone的拒绝ACE的前面，因为这正是它们在列表框中的顺序。然而，正如我们刚刚所说的，这些编辑器在对象上应用ACL时，总是将拒绝ACE项放在允许ACE项的前面。



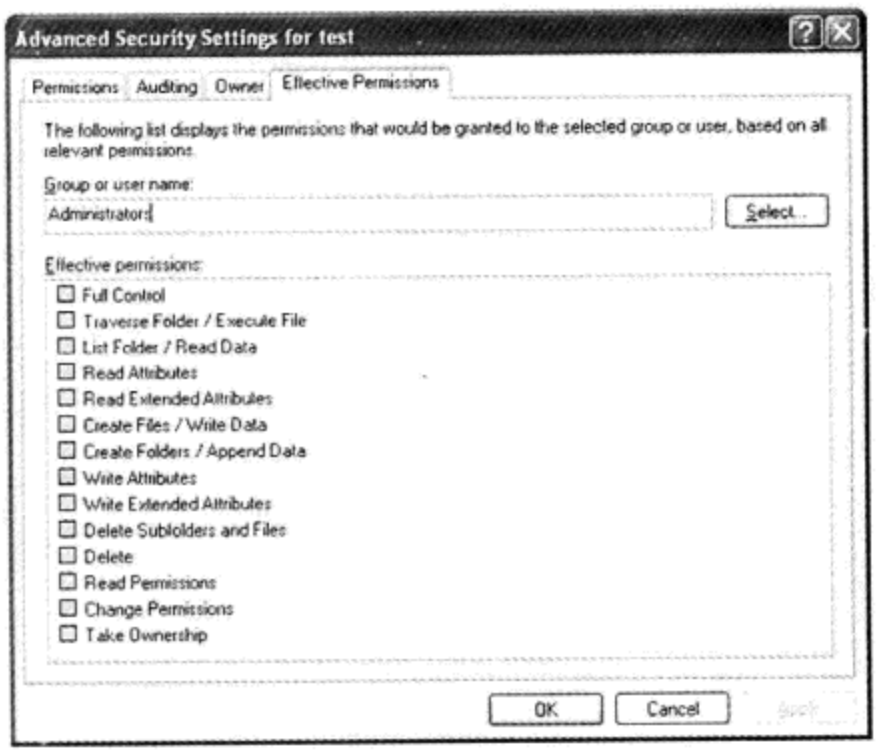


www.TopSage.com

高级安全设置（Advanced Security Settings）对话框的Permissions页面显示了此DACL中的ACE的顺序。然而，即使此对话框，也仍然十分使人迷惑，因为一个复杂的DACL可能有各种访问权限的拒绝ACE，其后面跟着其他访问类型的允许ACE。



要想知道一个特定的用户或组对于一个对象有什么样的访问权，具有确定性的惟一办法（当然不是让该用户或该组的一个成员试着访问该对象）是：使用该对话框的Effective Permissions页面，当你在对话框上按下Advanced按钮时，该页面就会出现；输入你想要检查的用户或组，对话框就会告诉你，该用户或组在此对象上有哪些许可。



www.TopSage.com

AuthZ API

Windows的AuthZ API是在Windows XP中引进的，它实现了与安全引用监视器（Security Reference Monitor）相同的安全模型，但它是完全在用户模式下实现的（位于\Windows\System32\Authz.Dll库中）。这使得那些想要保护其私有对象（比如数据库表）的应用能够充分利用Windows的安全模型，但又不必付出从用户模式到内核模式转换的代价。如果它们直接依赖于Windows的安全引用监视器，则这种模式转换的代价是必须要付出的。

AuthZ API使用了标准的安全描述符数据结构、SID和特权。但是，AuthZ并不使用令牌来表示客户，相反，它使用AUTHZ_CLIENT_CONTEXT。对于所有的访问检查和Windows安全函数，AuthZ都包含了对应的用户模式等价版本。例如，*AuthzAccessCheck*是Windows API *AccessCheck*的AuthZ版本，其中*AccessCheck*使用了安全引用监视器函数*SeAccessCheck*。

对于那些使用AuthZ的应用，另一个好处是，它们可以指示AuthZ将安全检查的结果缓存起来，以提高后续的、使用同样客户环境和安全描述符的安全检查的效率。

Platform SDK中有关于AuthZ的完整文档。

8.3 账户权限和特权

进程在运行过程中执行的许多操作是无法通过对象访问保护来授权控制的，因为这些操作并没有与一个特定的对象打交道。例如，当为了做备份而打开文件时，“能够绕过安全检查”这一能力是一个账户的属性，而不是某个对象的属性。Windows使用特权（privilege）和账户权限（account right），以使得系统管理员可以控制哪些账户能够执行与安全相关的操作。

特权是指一个账户执行某个与系统相关的操作的权限，比如，关闭计算机或者改变系统的时间。而账户权限则把“执行某一特定登录类型（比如本地登录或交互式登录到一台计算机上）的能力”授予它所针对的账户，或拒绝分配给该账户。

系统管理员利用一些工具来将特权分配给组和账户，这样的工具有针对域用户的活动目录用户和组MMC加载件（Active Directory Users and Groups MMC snap-in），或者本地安全策略编辑器（Local Security Policy Editor）。你通过控制面板的Administrative Tools文件夹，或者Start菜单（如果你已经配置了Start菜单，使它包含一个Administrative Tools链接的话）就可以启动本地安全策略编辑器。图8.6显示了本地安全策略编辑器中的用户权限分配（User Rights Assignment）配置情况，它显示了在Windows Server 2003上所有可用的特权和账户权限的完整列表。注意，该工具并不区分特权和账户权限。然而，你可以对两者进行区分，因为任何包含单词“*logon*”的用户权限都是一个账户特权。

www.TopSage.com

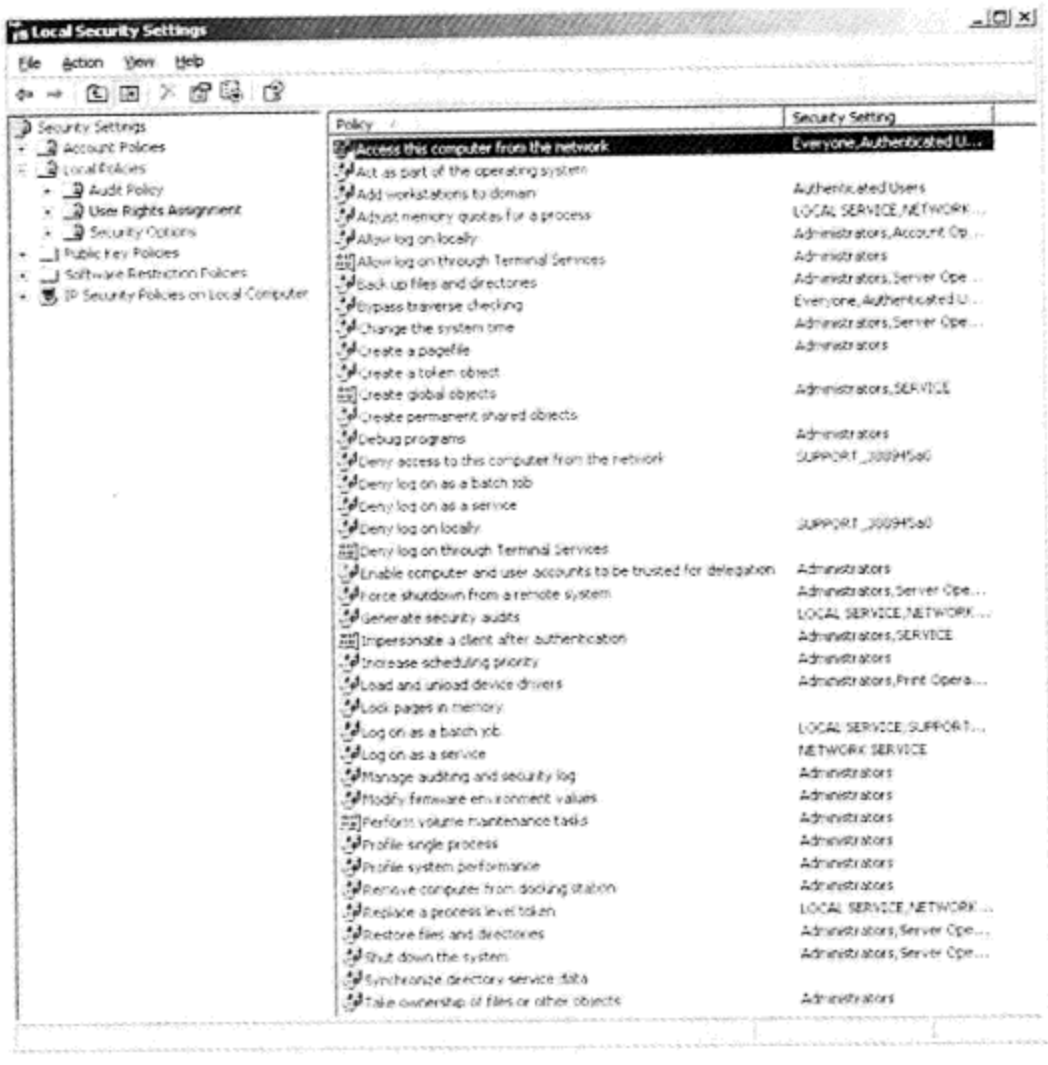


图8.6 本地安全策略编辑器的用户权限分配视图

账户权限

账户权限并不是由安全引用监视器强制实施的，也不存储在令牌中。负责登录的函数是*LsaLogonUser*。例如，当一个用户以交互方式登录到一台机器上时，Winlogon调用*LogonUser* API，而*LogonUser*又调用了*LsaLogonUser*。*LsaLogonUser*函数有一个参数指明了要执行的登录的类型，包括交互式登录、网络登录、批方式登录、服务方式登录、终端服务器客户和解除锁定。

当一个用户企图登录到系统中时，作为对登录请求的响应，本地安全权威中心（LSA）从LSA数据库中获取到已赋予该用户的账户权限。LSA根据已分配给当前登录用户的账户权限，对登录类型进行检查，如果该用户的账户没有“允许此种登录类型”的权限，或者具有“拒绝此种登录类型”的权限，则LSA拒绝该用户的登录请求。表8.4列出了Windows定义的用户权限。

通过*LsaAddAccountRights*和*LsaRemoveAccountRights*函数，Windows应用程序可以为一个账户增加或者删除用户权限；另外，通过*LsaEnumerateAccountRights*函数，它们也可以确定一个账户已经被分配了哪些权限。

www.TopSage.com

表8.4 账户权限

用户权限	角 色
拒绝交互式登录（Deny logon interactively）， 允许交互式登录（Allow logon interactively）	用于在本地机器上发起的交互式登录
拒绝通过网络登录（Deny logon over the network）， 允许通过网络登录（Allow logon over the network）	用于从远程机器发起的登录
拒绝通过终端服务来登录（Deny logon through Terminal Services）[1]， 允许通过终端服务来登录（Allow logon through Terminal Services）[1]	用于通过一个终端服务器客户来登录的情形
拒绝作为一个服务而登录（Deny logon as a service）， 允许作为一个服务而登录（Allow logon as a service）	当在一个特定的用户账户下启动一个服务时，服务控制管理器使用这些权限
拒绝作为一个批作业而登录（Deny logon as a batch job）， 允许作为一个批作业而登录（Allow logon as a batch job）	当执行一个*批*（batch）类型的登录时使用这些权限

特权

由操作系统定义的特权的数量随着时间的推移而不断增加。用户权限是由LSA在一个地方强制使用的。与用户权限不同的是，不同的特权是由不同的组件来定义的，并且也由这些组件来强制使用。例如，调试特权是由进程管理器（Process Manager）来检查的，它使得一个进程在利用Windows API函数*OpenProcess*来打开另一个进程的句柄时可以绕过安全检查。表8.5给出了一份完整的特权列表，并且也描述了系统组件何时以及如何检查这些特权。

当一个组件想要检查一个令牌是否有特权时，如果该组件运行在用户模式下，则它使用*PrivilegeCheck*或*LsaEnumerateAccountRights* API函数；如果运行在内核模式下，则使用*SeSinglePrivilegeCheck*或*SePrivilegeCheck*。与特权有关的API并不能理解账户权限，但账户权限API却能理解特权。

与账户权限不同的是，特权是可以被允许和禁止的。要想让一个特权检查能够成功地通过，该特权必须出现在当前特定的令牌中，而且它必须是允许的。这种方案背后的思想是，只有当确实有必要使用特权时，这些特权才应该是允许的，所以，一个进程不能毫无意识地执行一个有特权的安全操作。

[1] 在Windows XP中引入。

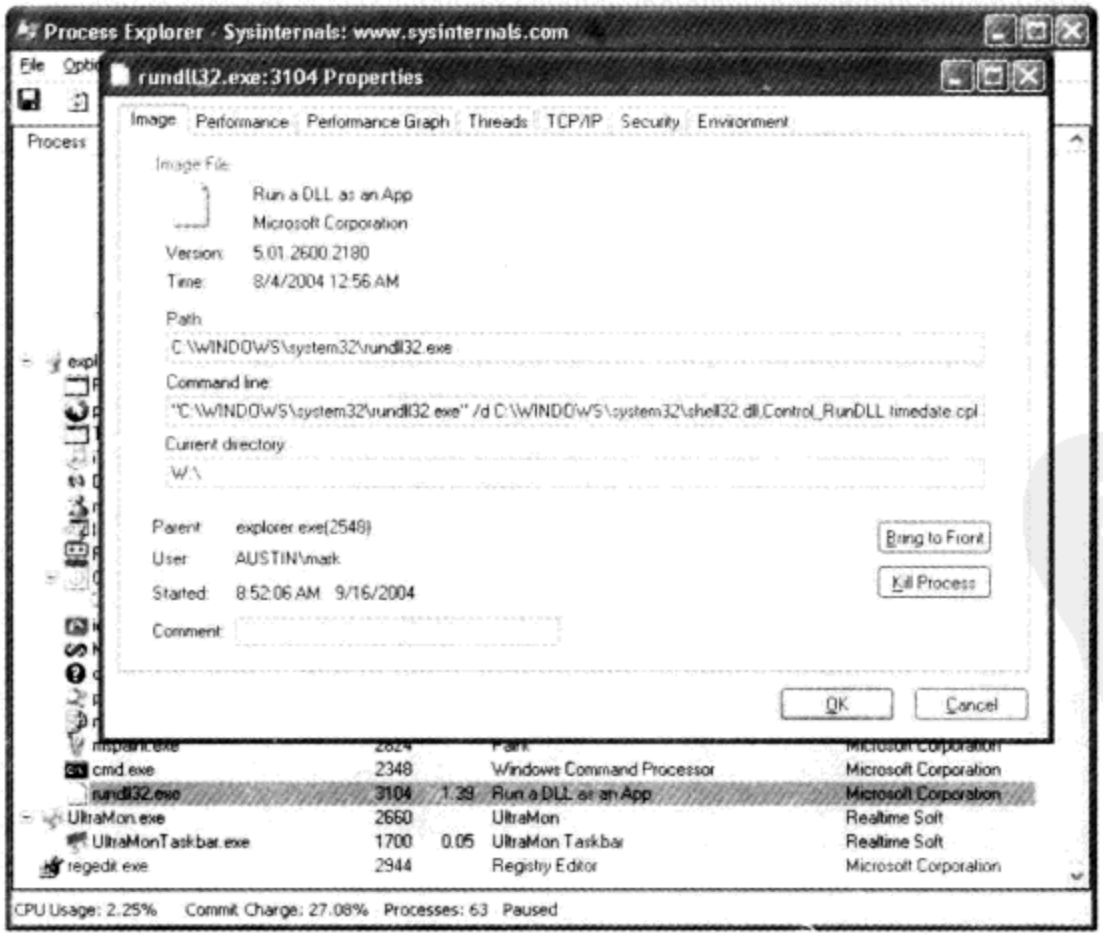
www.TopSage.com



实验：看着一个特权变成允许的

按照下面的步骤，你可以看到，当你通过控制面板中的Date and Time小程序来改变计算机的时钟或日期时，它就会允许SeSystemTimePrivilege特权。

1. 用一个具有“Change the system time（改变系统时间）”用户权限的账户登录到一个系统中（比如，你可以使用Power Users组或Administrators组中的成员账户来登录）。
2. 运行进程管理器（Process Explorer），将刷新率设置为Paused。
3. 查看Explorer外壳进程的属性对话框中的Security页面，你应该看到SeChangeSystemTimePrivilege特权是禁止的，这是通过它的标志（Flags列）以及它被显示成灰色这一事实可以看得出来。
4. 打开控制面板中的Date and Time小程序，并且重新刷新进程管理器。于是，一个新的Rundll进程出现，它被加亮显示成绿色。
5. 打开Rundll进程的属性对话框（在该进程上双击），并且验证其命令行包含了文本“Timedate.Cpl”。Rundll是一个控制面板DLL宿主进程，在命令行中出现了此参数，则意味着让它加载一个实现了可修改时间和日期的用户界面的DLL。



www.TopSage.com

6. 切换到Rundll进程属性对话框的Security页面，你应该可以看到，SeSystemTimePrivilege特权是允许的。

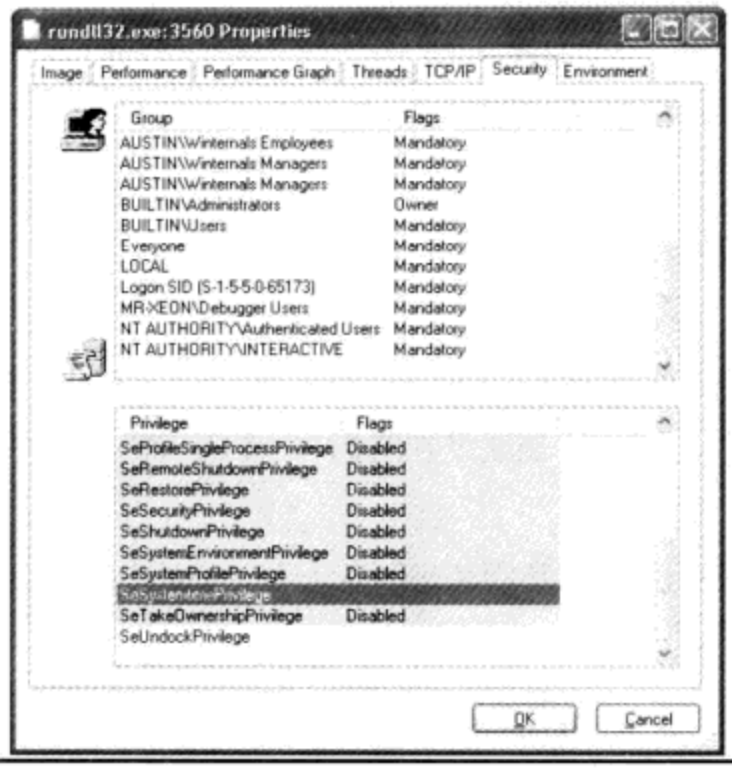


表8.5 特权

特 权	用户权限	特权用途
SeAssignPrimaryTokenPrivilege	替换一个进程层次的令牌	各种组件会检查此特权，比如*NtSetInformationJob*，它设置一个进程的令牌
SeAuditPrivilege	产生安全审计	要想利用*ReportEvent* API来产生事件，并记录到安全事件日志中，这是必需的特权
SeBackup	备份文件和目录	使得NTFS不管实际的安全描述符是什么，都授予对任何文件或目录的下列访问权： READ_CONTROL ACCESS_SYSTEM_SECURITY FILE_GENERIC_READ FILE_TRAVERSE 注意，当调用者为了备份而打开一个文件时，必须指定FILE_FLAG_BACKUP_SEMANTICS标志
SeChangeNotifyPrivilege	绕过穿越检查（Bypass traverse checking）	NTFS利用此特权，以避免在一个多层目录查找过程中的中间目录上必须要获得许可。当应用程序为了接收“文件系统结构的变化通知”而进行登记时，文件系统也使用此特权
SeCreateGlobalPrivilege[2]	创建全局对象	当一个进程在对象管理器名字空间的目录中创建内存区和符号链接对象，但这些目录却被分配给不同于调用者的会话时，需要此特权

[2] 在Windows Server 2003中引入。

www.TopSage.com

续表

特 权	用户权限	特权用途
SeCreatePagefilePrivilege	创建一个页面文件	*NtCreatePagingFile*函数检查此特权。该函数被用于创建一个新的页面文件
SeCreatePermanentPrivilege	创建永久共享对象	当对象管理器创建一个永久对象（当没有引用指向该对象时，该对象也不会被释放）时需要检查此特权
SeCreateTokenPrivilege	创建一个令牌对象	*NtCreateToken*函数检查此特权。该函数创建一个令牌对象
SeDebug	调试程序	如果调用者有这个特权的话，则进程管理器（Process Manager）允许通过*NtOpenProcess*访问任何一个进程，而不考虑该进程的安全描述符是否允许
SeEnableDelegationPrivilege[3]	允许计算机账户和用户账户被用于委托	活动目录服务利用该特权来委托已经认证的安全凭证
SeImpersonatePrivilege[1]	在认证以后模仿一个客户	当一个线程想要将一个令牌用于模仿，而且该令牌代表了一个不同于该线程的进程令牌的用户时，进程管理器检查此特权
SeIncreaseBasePriorityPrivilege	增加调度优先级	进程管理器检查此特权。要提升一个进程的优先级，这一特权是必需的
SeIncreaseQuotaPrivilege	Windows 2000：增加配额 Windows XP/Windows Server 2003：调整一个进程的内存配额	当改变注册表的配额（仅针对Windows 2000）、一个进程的工作集阈值，以及一个进程的换页内存池和非换页内存池配额时，这一特权是强制要求的
SeLoadDriverPrivilege	加载和卸载设备驱动程序	驱动程序函数*NtLoadDriver*和*NtUnloadDriver*要检查此特权
SeLockMemoryPrivilege	将页面锁在内存中	*NtLockVirtualMemory*检查此特权。该函数是VirtualLock的内核实现
SeMachineAccountPrivilege	将工作站加入到域中	当域控制器上的安全账户管理器（Security Accounts Manager）在域中创建一个机器账户时，需要检查此特权
SeManageVolumePrivilege[2]	执行卷维护的任务	在打开卷的操作过程中，文件系统驱动程序强制检查此特权。为了执行磁盘检查和碎片整理任务，这一特权是必需的
SeProfileSingleProcessPrivilege	对单个进程建立性能轮廓	预取器函数为了返回单个进程的预取信息，需要检查此特权
SeRemoteShutdownPrivilege	从一个远程系统强迫停机	Winlogon检查*InitiateSystemShutdown*函数的远程调用者具有此特权

[3] 在Windows XP中引入。

www.TopSage.com

续表

特 权	用户权限	特权用途
SeRestorePrivilege	恢复文件和目录	这一特权使得NTFS不管实际的安全描述符是什么，都授予对任何文件或目录的下列访问权： WRITE_DAC WRITE_OWNER ACCESS_SYSTEM_SECURITY FILE_GENERIC_WRITE FILE_ADD_FILE FILE_ADD_SUBDIRECTORY DELETE 注意，当调用者为了恢复而打开一个文件时，必须指定FILE_FLAG_BACKUP_SEMANTICS标志
SeSecurityPrivilege	管理审计和安全日志	为了访问一个安全描述符的SACL，读取和清除安全事件日志，这一特权是必需的
SeShutdownPrivilege	系统停机	*NtShutdownSystem* 和 *NtRaiseHardError* 检查此特权。*NtRaiseHardError*函数在交互式控制台上显示一个系统错误对话框
SeSyncAgentPrivilege	同步目录服务数据	要想使用LDAP目录同步服务，此特权是必需的。而且，它使得特权持有者可以读取目录中的所有对象和属性，不管这些对象和属性上有哪些保护特性
SeSystemEnvironmentPrivilege	修改固件环境变量	*NtSetSystemEnvironmentValue* 和 *NtQuerySystemEnvironmentValue*要求该特权以便利用HAL来修改和读取固件环境变量
SeSystemProfilePrivilege	针对系统性能建立轮廓	*NtCreateProfile*函数检查此特权。该函数被用于执行系统的性能轮廓分析。例如，Kernprof工具用到了此特权
SeSystemtimePrivilege	改变系统时间	要改变时间或日期，此特权是必需的
SeTakeOwnership	接管文件和其他对象的所有权	在未被授予自主访问的情况下，要接管一个对象的所有权，此特权是必需的
SeTcbPrivilege	作为操作系统的一部分来执行	当在一个令牌中设置会话ID时，安全引用监视器要检查此特权；当即插即用管理器创建和管理即插即用事件时也要检查此特权；LogonUser的Windows 2000版本，BroadcastSystemMessageEx当被BSM_ALLDESKTOPS参数来调用时，以及LsaRegisterLogonProcess也需要此特权
SeUndockPrivilege	从插接站(docking station）移除计算机	当一个计算机拔除操作被激发，或者一个设备弹出（eject）请求发生时，用户模式的即插即用管理器需要检查此特权

www.TopSage.com



实验："绕过穿越检查（Bypass Traverse Checking）"特权

如果你是一个系统管理员，那么，你必须知道"绕过穿越检查（Bypass Traverse Checking）"特权（在内部被称为SeNotifyPrivilege）以及它的确切含义。本实验演示了如果不理解它的行为将可能导致不正确地应用安全性。

1. 创建一个文件夹，并且在该文件夹的内部创建一个新的文本文件，然后在文本文件中输入一些示例性的文本。
2. 在Explorer中跳转到新文件上，进入其Properties对话框的Security页面。点击Advanced按钮，不要选中控制继承性的检查框。当出现提示，问你要移除或拷贝那些已被继承的许可时，请选择拷贝。
3. 接下来，修改新文件夹的安全性，以便你的账户对该文件夹没有任何访问权。要做到这一点，你可以先选中你的账户，然后在许可列表中选中所有的拒绝项。
4. 运行Notepad，并且用FilelOpen对话框浏览到新的目录中。你应该被拒绝访问该目录。
5. 在打开文件对话框的File Name域中，输入此新文件的完整路径名。该文件应该被打开。

如果你的账户没有"绕过穿越检查"特权，那么，当你试图打开一个文件时，NTFS对于该文件的路径上的每个目录都要执行一次访问检查，在这个例子中，这将导致你被拒绝访问该文件。

超级特权

有几个特权的功能如此强大，以至于一旦用户被授予这样的特权，该用户就成了一个对计算机有完全控制权的"超级用户"。用户通过各种无穷无尽的方式来使用这些特权，就有可能获得对于本该禁入的资源的未授权访问，以及执行未授权的操作。然而，我们将焦点集中在"利用特权来执行一些代码，由这些代码将本来没有分配给用户的特权授予该用户"，以及"充分利用这种能力，从而在本地机器上执行任何期望的操作"。这一小节列出这些特权，并且讨论这些特权可能被发掘和使用的途径。其他的特权，比如"Lock Pages In Physical Memory（将页面锁在物理内存中）"，也可能被发掘出来，并在一个系统上用于实施拒绝服务攻击。这里没有讨论这些特权。

- **调试程序** 具有此特权的用户可以打开系统中的任何一个进程，而不必考虑该进程上的安全描述符。例如，用户可以实现这样一个程序，它打开LSASS进程，将可执行代码拷贝到它的地址空间中，然后通过*CreateRemoteThread* Windows API注入一个线程，让它在一个更加有特权的安全环境下执行这些注入的代码。这部分代码可以授予该用户额外的特权和组成员关系；

www.TopSage.com

- **接管所有权** 这一特权使得特权持有者能够接管任何一个被保护对象的所有权。做法是，将他自己的SID写到该对象的安全描述符的所有者域中。前面曾经提到过，所有者总是被授予“读取和修改该安全描述符的DACL”的许可，所以，具有此特权的进程可以修改此DACL，以授予它自己对于该对象的完全访问权，然后关闭该对象，再用完全访问权重新打开该对象。这将使得该所有者能够看到敏感的数据，甚至用他自己的程序来替代那些作为正常系统操作的一部分来执行的系统文件（比如LSASS），而在他自己的程序中，他可以将提升的特权授予给一个用户；
- **恢复文件和目录** 被分配了此特权的用户能够用他自己的文件来替代该系统中的任何文件。他可以像上一段落中描述的那样，来替换系统的文件，从而发掘出这一强大的能力；
- **加载和卸载设备驱动程序** 一个恶意用户可以使用这种特权将一个设备驱动程序加载到系统中。设备驱动程序被认为是操作系统的可信任部分，操作系统可能会在System账户凭证下运行设备驱动程序中的代码，所以，一个驱动程序可以启动有特权的程序，这些特权程序再授予用户其他的权限；
- **创建一个令牌对象** 此特权的一种很显然的用法是，生成一些代表任意用户账户的令牌，而且其中的用户账户可以有任意的组成员关系和特权；
- **作为操作系统的一部分来执行** LsaRegisterLogonProcess函数要检查此特权。一个进程调用此函数来建立起它与Lsass之间的可信连接。具有此特权的恶意用户可以建立一个可信的Lsass连接，然后执行LsaLogonUser（一个用于创建新登录会话的函数）。LsaLogonUser要求一个有效的用户名和口令，并且接受一组可选的SID列表，它将这些SID加入到它为新登录会话创建的初始令牌中。因此，用户可以使用他的用户名和口令来创建一个新的登录会话，而新的登录会话在其令牌中包含多个特权组或用户的SID。

注意，提升起来的特权的使用范围并不会通过机器的边界，扩展到网络上，因为与另一台计算机的任何交互操作都要求通过一个域控制器进行认证，并且要验证域口令。而域口令并不以明文方式或加密形式被存储在一台计算机上，所以恶意代码是无法访问到域口令的。

8.4 安全审计

对象管理器可能会生成一些审计事件，并以此作为一次访问检查的结果；而且，有些可供用户应用程序使用的Windows函数也可以直接生成审计事件。内核模式的代码总是允许生成审计事件。有两个特权与审计有关，它们是SeSecurityPrivilege和SeAuditPrivilege。一个进程必须有SeSecurityPrivilege特权才能管理安全事件日志（Event Log），以及查看或设置一个对象的SACL。然而，如果一个进程要调用审计系统服务，那么，它必须拥有SeAuditPrivilege特权，只有这样才能成功地生成一条审计纪录。

www.TopSage.com

本地系统的审计策略控制了“审计某种特定类型的安全事件”的决定。审计策略也称为本地安全策略，它是本地系统上Lsass维护的安全策略的一部分，它可通过本地安全策略编辑器来进行配置，如图8.7所示。

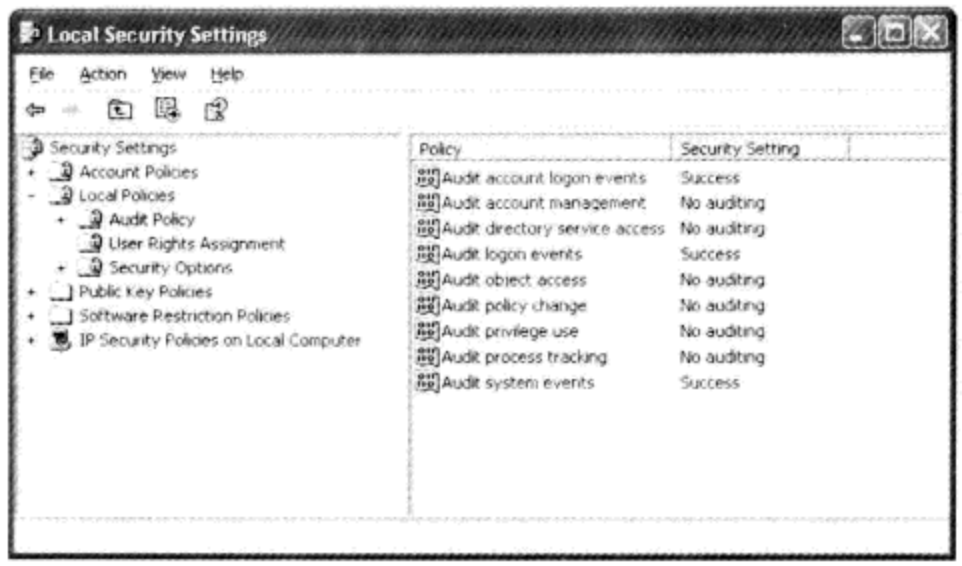


图8.7 安全策略编辑器中的审计策略配置

在系统初始化时刻，以及当审计策略改变的时候，Lsass给SRM发送消息，告诉它关于审计策略的情况。Lsass负责接收那些根据SRM的审计事件而生成的审计纪录，它也负责编辑这些纪录，并且将它们发送给事件记录器（Event Logger）。之所以由Lsass（而不是SRM）发送这些纪录，是因为它加入了有关的细节，比如为了更加完整地标识出被审计进程而需要的信息。

SRM通过它与Lsass之间的LPC连接来发送审计纪录。然后，事件记录器将审计纪录写到安全事件日志中。除了SRM传递过来的审计纪录以外，Lsass和SAM也会生成一些审计纪录，Lsass直接将它们发送给事件记录器，AuthZ API也允许应用程序生成由它们自己定义的审计纪录。图8.8描绘了这一总体流程图。

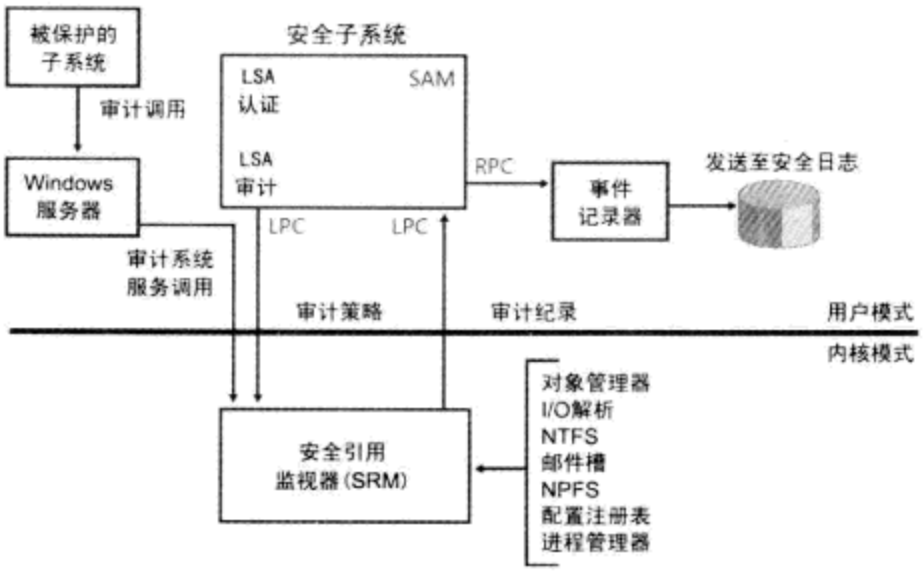


图8.8 安全审计纪录的流程图

www.TopSage.com

当SRM接收到审计纪录时，这些审计纪录被放到一个队列中，以便进一步发送至LSA——它们不是被成批提交的。审计纪录通过两种方式之一，被从SRM中转移到安全子系统中。如果一个审计纪录比较小（小于最大的LPC消息长度），那么它被当作一个LPC消息来发送。这样的审计纪录被从SRM的地址空间中拷贝到Lsass进程的地址空间中。如果审计纪录比较大，那么，SRM使用共享内存，直接让Lsass访问该消息，它在LPC消息中只是简单地传递一个指针。

图8.9将本章到目前为止所有讲述过的概念串连在一起，它说明了基本的进程和线程安全结构。要注意到图中的进程对象和线程对象都有ACL，访问令牌对象本身也有ACL。而且，在该图中，线程2和线程3各有一个模仿令牌，而线程1默认使用进程的访问令牌。

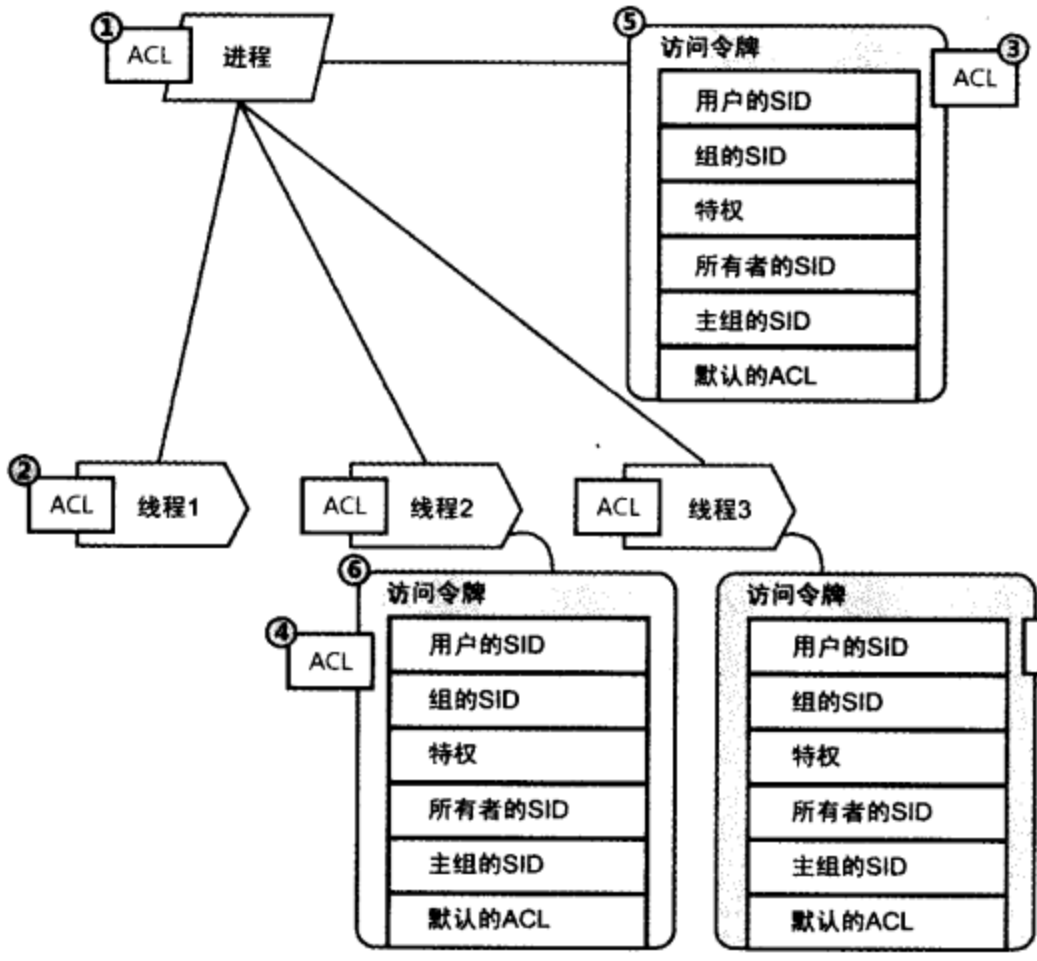


图8.9 进程和线程的安全结构

8.5 登录（Logon）

交互式登录（相对于网络登录）是通过登录进程（Winlogon）、Lsass、一个或多个认证包，以及SAM或活动目录之间的相互作用而发生的。**认证包**（*authentication package*）是指执行认证检查的DLL。

www.TopSage.com

当用户以交互方式登录到一个域中时，Windows使用Kerberos作为认证包；当用户以交互方式登录到本地计算机上时，或者登录到Windows 2000以前的可信域中时，或者当没有域控制器可以访问时，Windows使用MSV 1_0作为认证包。

Winlogon是一个可信任的进程，它负责管理与安全有关的用户交互。它协调登录过程、在登录时启动用户的第一个进程、处理注销过程，以及管理其他各种与安全有关的操作，包括在登录时输入口令、改变口令、锁住工作站和解除工作站的锁定。Winlogon进程必须确保这些与安全有关的操作对于任何其他的活动进程都是不可见的。例如，Winlogon保证，在以上任何一个操作过程中，一个不可信的进程无法通过获得对桌面的控制来得到对口令的访问。

Winlogon依赖于一个图形化标识和认证（GINA）DLL来获得一个用户的账户名和口令。默认的GINA是Msgina（\Windows\System32\Msgina.dll）。Msgina显示一个标准的Windows登录对话框。Windows允许用其他的GINA来代替Msgina，这使得Windows也可以使用其他的用户标识机制。例如，第三方可能会提供这样一个GINA：利用指纹识别设备来标识用户，并且从一个加密的数据库中提取出他们的口令。

Winlogon是惟一一个截取了来自键盘的登录请求的进程。Winlogon从GINA处得到了一个用户名和口令以后，它调用Lsass来认证这一试图要登录的用户的身份。如果该用户通过了认证，则登录进程代表该用户激活一个登录外壳。登录过程中涉及的组件之间的交互关系如图8.10所示。

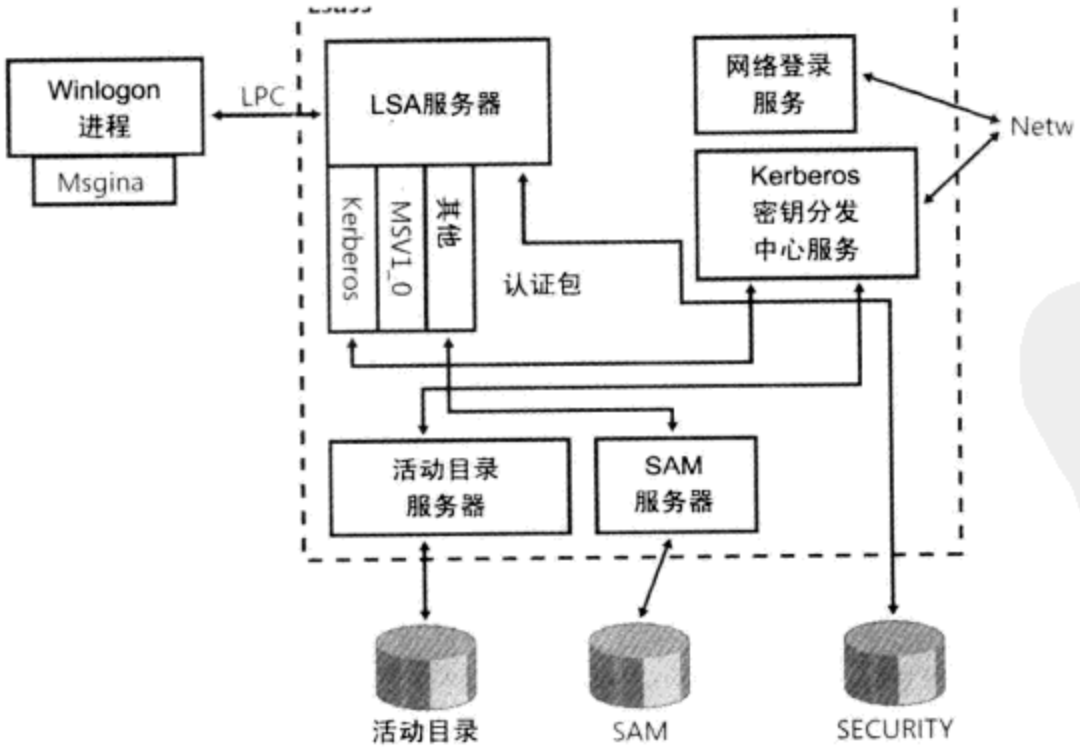


图8.10 登录过程中涉及的组件

www.TopSage.com

除了支持可替换的GINA以外，Winlogon也可以加载那些额外的、用于执行二级认证所需要的网络提供者DLL。这一能力使得多个网络提供者可以在正常的登录过程中一次收集所有的标识和认证信息。一个登录到Windows系统中的用户可能同时也在一台UNIX服务器上进行认证。然后，该用户就能够直接从Windows机器上访问此UNIX服务器中的资源，而无需再要求额外的认证。这样的能力也被认为是“**单点登录**（*single sign-on*）”的一种形式。

Winlogon初始化

在系统初始化过程中，在任何用户应用程序被激活起来以前，一旦系统准备好与用户进行交互了， Winlogon就执行下面的步骤来确保它控制着整个工作站。

1. 创建并打开一个交互式窗口站（例如，对象管理器名字空间中的\Windows\WindowStations\WinSta0）来代表键盘、鼠标和监视器。Winlogon为该窗口站创建一个安全描述符，它有且只有一个ACE，并且此ACE只包含了System SID。这一特殊的安全描述符确保了：除非得到Winlogon的明确许可，否则没有其他的进程可以访问此窗口站。
2. 创建和打开两个桌面：一个应用程序桌面（\Windows\WinSta0\Default，也被称为交互式桌面）和一个Winlogon桌面（\Windows\WinSta0\Winlogon，也被称为安全桌面）。Winlogon桌面上的安全性被建立起来：只有Winlogon才能访问此桌面。另一个桌面既允许Winlogon，也允许用户访问它。这种安排意味着，不管任何时候，当Winlogon桌面是活动桌面时，没有任何其他的进程可以访问与该桌面相关联的任何活动代码或数据。Windows使用此特性来保护与口令相关的安全操作，以及锁定桌面和解除桌面锁定这样的操作。
3. 在任何人登录到一台计算机上以前，可视的桌面正是Winlogon桌面。当有一个用户登录时，他一旦按下了Ctrl+Alt+Delete，就从默认（Default）桌面切换到Winlogon桌面（这也解释了为什么当他按下Ctrl+Alt+Delete时，他的交互式桌面上的所有窗口就一下子消失了，然后，当他解除了Windows Security对话框时，这些窗口又回来了）。因此，SAS总是带出一个由Winlogon控制的安全桌面。
4. 与Lsass的*LsaAuthenticationPort*建立一个LPC连接。此连接将被用于在登录、注销和口令操作过程中交换信息，它是通过调用*LsaRegisterLogonProcess*而建立起来的。

然后，Winlogon执行以下Windows操作，以建立起窗口环境。

5. 它初始化并登记一个窗口类数据结构，通过该窗口类数据结构将Winlogon窗口过程与它随后创建的窗口关联起来。

www.TopSage.com

6. 它登记SAS，将它与刚刚创建的窗口关联起来，从而保证：无论何时当用户按下SAS时，Winlogon的窗口过程就会被调用。这种办法可防止特洛伊木马程序在用户按下SAS时获得对屏幕的控制。
7. 它登记该窗口，因而当用户注销登录，或者屏幕保护程序超时的时候，与该窗口关联的窗口过程能够被调用到。Windows子系统会进行检查，以检验此“请求获得通知”的进程是Winlogon进程。

SAS是如何实现的

SAS是安全的，因为没有应用程序能够截取Ctrl+Alt+Delete键击组合，也无法阻止Winlogon接收此键击组合。Winlogon使用一个文档化的Windows API，即*RegisterHotKey*，来保留Ctrl+Alt+Delete键组合，因此，无论何时当Windows输入系统看到此组合时，它给Winlogon创建的一个窗口发送一个特殊的消息，让它接收这样的通知。凡是被映射到一个已登记热键的键击都只被发送到登记该键击的那个进程，而不再发送至任何其他的进程，而且，只有登记热键的那个线程才可以注销该热键（通过*UnregisterHotKey* API），因此，特洛伊木马应用程序不可能注销Winlogon对于SAS的所有权。

Windows函数*SetWindowsHook*允许一个应用程序安装一个钩子过程，因而每次当一个键被按下的时候（甚至在热键被处理以前），钩子过程就会被调用；它也允许该钩子将用户的键击动作抑制住。然而，Windows的热键处理代码包含了针对Ctrl+Alt+Delete的专门处理，对于此键击序列，它禁止任何钩子，所以，此键击序列不会被截取。而且，如果交互式桌面被锁住的话，则只有Winlogon拥有的热键才被处理。

在初始化过程中一旦Winlogon桌面被创建，它就成为活动的桌面。当Winlogon桌面是活动桌面时，它总是被锁住的。Winlogon解除其桌面的锁定，仅仅是为了切换到应用程序桌面，或者屏幕保护程序的桌面（只有Winlogon进程才可以锁定一个桌面，或者解除桌面的锁定）。

用户登录步骤

当用户按下SAS（Ctrl+Alt+Delete）时，登录便开始了。当SAS被按下以后，Winlogon调用GINA来获得一个用户名和口令。Winlogon也为该用户创建一个惟一的本地登录SID，这是它分配给此桌面实例（键盘、屏幕和鼠标）的SID。Winlogon在*LsaLogonUser*调用中将此SID传递给Lsass。如果该用户成功登录，则此SID将被包含在登录进程的令牌中——这一步骤保护了对桌面的访问。例如，在另一个系统上登录到同一个账户下将不能对第一台机器的桌面进行写操作，因为第二个登录不是在第一个登录的桌面令牌中。

www.TopSage.com

当用户名和口令被输入时，Winlogon通过调用Lsass的函数*LsaLookupAuthenticationPackage*来获得一个指向认证包的句柄。在注册表的HKLM\SYSTEM\CurrentControlSet\Control\Lsa下面列出了系统中的认证包。Winlogon通过*LsaLogonUser*将登录信息传递给认证包。一旦认证包认证了一个用户，Winlogon便继续该用户的登录过程。如果没有一个认证包指示这是一次成功的登录，则登录过程以失败而告终。

对于交互式登录，Windows使用两个标准的认证包：Kerberos和MSV1_0。在一个单独的Windows系统上默认的认证包是MSV1_0 （\Windows\System32\Msv1_0.dll），这是一个实现了“LAN Manager 2”协议的认证包。Lsass也在域成员计算机上使用MSV 1_0来认证Windows 2000之前的域和未能找到域控制器来进行认证的计算机（与网络断开连接的笔记本计算机也归入后一类）。 Kerberos认证包（\Windows\System32\Kerberos.dll）被用在Windows域的成员计算机上。Windows的Kerberos认证包，连同域控制器上运行的Kerberos服务，两者联合起来支持Kerberos协议的版本5和修订版6。该协议是以Internet RFC 1510为基础的（有关Kerberos标准的细节信息，请访问IETF[Internet Engineering Task Force] Web站点www.ietf.org）。

MSV 1_0认证包接受用户名和经过散列计算后的口令值，并且向本地的SAM发送一个请求，以获取有关账户的信息，其中包括口令、用户所属的组，以及该账户的任何限制。MSV 1_0首先检查该账户的限制，比如允许访问的小时时间段或者类型。如果该用户由于SAM数据库中的限制而不能登录，那么这次登录失败，MSV 1_0向LSA返回一个失败状态。

然后，MSV 1_0将用户名和散列后的口令值，分别与SAM中存储的用户名和散列后的口令值进行比较。如果这是一个缓存的域登录，则MSV 1_0使用Lsass的有关函数来访问缓存的信息；通过Lsass的这些函数，可以从LSA数据库（注册表的SECURITY储巢[hive]）中获得这些“秘密”，或者将“秘密”存储到LSA数据库中。如果用户输入的信息与SAM中的信息相符，则MSV 1_0为该登录会话生成一个LUID，并且通过调用Lsass来创建此登录会话；它也将此惟一标识符与该会话关联起来，而且，在调用Lsass时传递必要的信息以便最终为该用户创建一个访问令牌（前面提到过，一个访问令牌包含用户的SID、组SID和分配的特权）。



> **注** MSV1_0并没有将用户完整的口令散列值缓存在注册表中，因为这将使得任何能通过物理方式访问该系统的人很容易就能危及该用户的域账户，并且访问到该用户的加密文件，以及任何该用户有权访问的网络资源。相反地，MSV 1_0缓存了此散列值的一半。这缓存的一半散列值，对于验证用户口令的正确性来说，已经足够了；但要想访问EFS密钥，或者证明自己是一个域中的用户，则仍然不够，因为这些操作要求整个散列值。

www.TopSage.com

如果MSV 1_0需要使用一个远程系统来认证身份，比如当用户登录到一个Windows 2000之前的可信域中时，MSV 1_0使用Netlogon服务来跟远程系统上的Netlogon实例进行通信。远程系统上的Netlogon与该系统上的MSV 1_0认证包进行交互，然后将认证结果送回到当前正在执行登录的系统上。

Kerberos认证的基本控制流与MSV 1_0的控制流相同。然而，在绝大多数情况下，域登录是在成员工作站或成员服务器上执行的（而不是在域控制器上执行的），因此，在认证过程中，认证包必须进行网络通信。Kerberos认证包的做法是，通过Kerberos TCP/IP端口（端口88）与域控制器上的Kerberos服务进行通信。Kerberos的密钥分发中心（Kerberos Key Distribution Center）服务（\Windows\System32\Kdcsvc.dll）实现了Kerberos认证协议，它运行在域控制器上的LSASS进程中。

Kdcsvc利用活动目录服务器\Windows\System32\Ntdsa.dll，通过活动目录的用户账户对象来验证散列的用户名和口令信息以后，向LSASS返回域凭证（domain credential），而LSASS又通过网络，向当前正在执行登录的系统返回认证的结果和用户的域登录凭证（如果登录成功的话）。



> **注** 这里关于Kerberos认证的描述是高度简化的，但是它突出了其中涉及的各个组件。虽然Kerberos认证协议在Windows的分布式域安全性中扮演了一个非常关键的角色，但是它的细节内容超出了本书的范围。

一次登录已经通过认证以后，LSASS在本地策略数据库中查找该用户有哪些允许的登录方式，包括交互方式、网络方式、批方式或服务方式。如果当前所请求的登录不符合数据库中允许的访问方式，则此次登录被终止。LSASS通过清除其所有数据结构的方式来删除新建的登录会话，然后向Winlogon返回失败，而Winlogon又依次向用户显示一个适当的消息。如果当前请求的登录是允许的，则LSASS加入一些额外的安全ID（比如Everyone、Interactive之类的）。然后它在策略数据库中检查该用户所有ID的特权，并且将这些已授予的特权加入到用户的访问令牌中。

当LSASS已经积累了所有这些必要的信息时，它调用执行体来创建访问令牌。执行体为交互方式登录或服务方式登录创建一个主访问令牌，为网络方式登录创建一个模仿令牌。在成功地创建访问令牌以后，LSASS复制此令牌，创建一个可被传递给Winlogon的句柄，然后关闭它自己的句柄。如果有必要，此次登录操作被审计下来。在这个点上，LSASS向Winlogon返回成功，同时还返回一个指向该访问令牌的句柄、此登录会话的LUID，以及认证包返回的轮廓信息（如果有的话）。

www.TopSage.com



实验：列出活动的登录会话

对于一个给定的登录会话LUID，只要存在至少一个令牌，Windows就认为该登录会话是活动的。你可以使用*www.sysinternals.com*的LogonSessions工具来列出活动的登录会话，该工具用到了*LsaEnumerateLogonSessions*函数（在Platform SDK中有文档）。示例如下：

```
 C:\>logonsessions
LogonSessions 1.0
Copyright (C) 2004 Bryce Cogswell
Sysinternals - www.sysinternals.com

[0] Logon session 00000000:000003e7:
    User name:    AUSTIN\MARKLAP$
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          S-1-5-18
    Logon time:   5/13/2004 9:05:43 AM
    Logon server:
    DNS Domain:
    UPN:
[1] Logon session 00000000:0000d37d:
    User name:
    Auth package: NTLM
    Logon type:   (none)
    Session:      0
    Sid:          (none)
    Logon time:   8/17/2004 5:08:10 PM
    Logon server:
    DNS Domain:
    UPN:
[2] Logon session 00000000:000003e4:
    User name:    NT AUTHORITY\NETWORK SERVICE
    Auth package: Negotiate
    Logon type:   Service
    Session:      0
    Sid:          S-1-5-20
    Logon time:   8/17/2004 5:08:18 PM
    Logon server:
    DNS Domain:
    UPN:
...

[8] Logon session 00000000:0079da73:
    User name:    MARKLAP\Administrator
    Auth package: MICROSOFT_AUTHENTICATION_PACKAGE_V1_0
    Logon type:   NetworkCleartext
    Session:      0
    Sid:          S-1-5-21-1787744166-3910675280-2727264193-500
    Logon time:   5/13/2004 12:14:20 PM
    Logon server: MARKLAP
    DNS Domain:
    UPN:
```

- 该工具为每个会话报告的信息有与该会话相关联的用户的SID和名称，以及会话的认证包和登录时间。

www.TopSage.com

注意，在以上输出的登录会话2中，Negotiate认证包是通过Kerberos还是通过NTLM来进行认证，这取决于哪一个认证包对于这次认证请求更为合适。

会话的LUID被显示在每一个会话输出块中的“Logon Session”这一行上，利用Handle工具（同样地，你可以从*www.sysinternals.com*下载），你可以找到代表一个特定登录会话的令牌。例如，要想找到上述例子中显示的登录会话8的令牌，你可以输入以下命令：

```
C:\>handle -a 79da73
  43c: Token          MARKLAP\Administrator:79da73
```

然后，Winlogon查看注册表中HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Userinit的值，并且无论该字符串的值是什么，它都创建一个进程来运行该字符串值（该值可能是几个用逗号来隔开的exe）。默认值是Userinit.exe，它加载该用户的轮廓，然后，创建一个进程来运行HKCU\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell中的字符串值（如果该值存在的话）。在默认情况下，该值并不存在。如果它确实不存在，则Userinit.exe对HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon\Shell中的值同样进行处理，而该值的默认值是Explorer.exe。然后，Userinit退出（这也正是为什么在进程管理器中你看到Explorer.exe没有父进程的原因）。关于用户登录过程中以后的步骤的更多信息，请参见第5章。

8.6 软件限制策略

诸如病毒和蠕虫之类的恶意代码成为越来越严重的问题。Windows XP引入了一种称为软件限制策略（Software Restriction Policy）的机制，它使得管理员可以控制哪些映像和脚本文件可以在他们的系统上运行。安全策略编辑器（Security Policy Editor）的软件限制策略节点也被看作一台机器的代码执行策略（code execution policy）的管理界面，如图8.11所示。不过，利用域的组策略来定义针对每个用户的策略也是可能的。

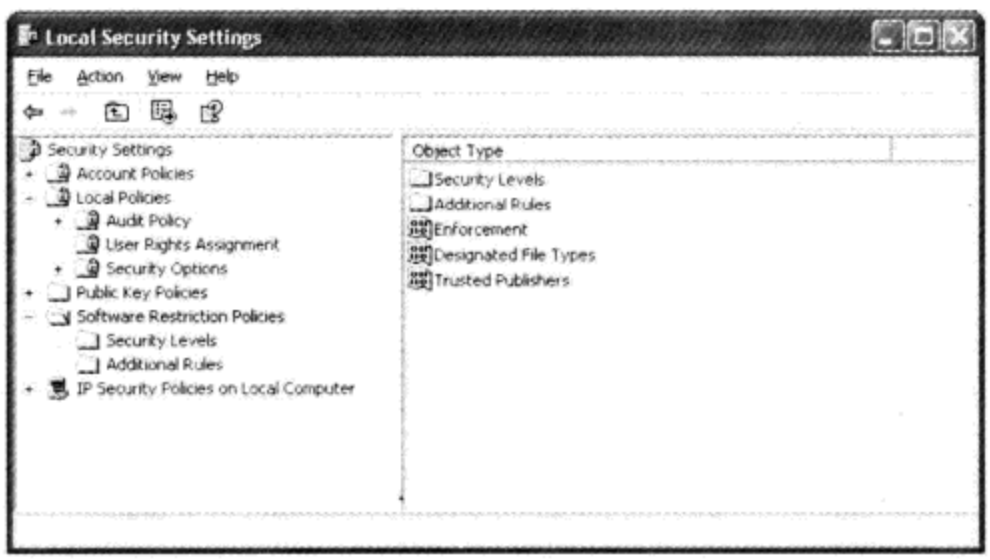


图8.11 软件限制策略的配置界面

www.TopSage.com

在软件限制策略节点的下面有以下几个全局策略设置。

- *Enforcement*（**强制**）策略配置了：这些限制策略是否应用于像DLL这样的库，以及这些策略是仅仅应用于普通用户，还是也应用于管理员；
- *Designated File Type*（指定的文件类型）策略记录了那些被认为是可执行代码的文件扩展名。
- *Trusted Publishers* （**可信的发布者**）控制了谁可以选择哪些证书发布者是可信的。

在为某个特定的脚本或映像文件配置一个策略时，管理员可以告诉系统，用它的路径、它的散列值、它的Internet区域（由Internet Explorer定义）或者它的密码学证书来识别它；而且，管理员也可以指定该文件是否与“**不允许的**（*Disallowed*）”或“**不受限制的**（*Unrestricted*）”安全策略有关联。

软件限制策略的*Enforcement*设置可以在各种组件中起作用。在这些组件中，文件被认为包含了可执行的代码。下面列出一些这样的组件。

- \Windows\System32\Kernel32.dll中的用户模式Windows函数CreateProcess强制为可执行映像文件应用*Enforcement*设置；
- Ntdll的DLL加载代码（\Windows\System32\Ntdll.dll）强制为DLL应用此设置；
- Windows命令提示程序（\Windows\System32\Cmd.exe）强制为批文件的执行应用此设置；
- 启动脚本代码的Windows脚本宿主（Windows Scripting Host）程序强制为脚本的执行应用此设置。其中，Windows脚本宿主程序包括\Windows\System32\Cscript.exe（针对命令行脚本）、\Windows\System32\Wscript.exe （针对UI脚本），以及\Windows\System32\Scrobj.dll（针对脚本对象）。

以上每一个组件都通过读取注册表的值HKEY_LOCAL_MACHINE\Software\Microsoft\Policies\Windows\Safer\CodeIdentifiers\TransparentEnabled来决定软件限制策略是否起作用，如果此注册表值为1，则表明这些策略是有效的。然后，它要确定：它将要执行的代码是否符合CodeIdentifiers键的子键中指定的规则之一；如果符合，此代码是否允许被执行。如果没有相符的规则，则根据CodeIdentifiers键的*DefaultLevel*值中指定的默认策略，来确定此代码的执行是否允许。

从防止代码和脚本被未授权访问的角度而言，软件限制策略是一个功能强大的工具，但只有当正确地使用这些限制策略时，这种强大的能力才能体现出来。除非默认策略被设置为不允许执行，否则，用户只需对一个已被标记为不允许执行的映像做一点很小的改变，就可以绕过这些规则，而执行该映像程序。例如，一个用户可以将一个进程映像中某个无关紧要的字节做一点改变，这样，散列规则就不会识别出它来，或者，用户可以将一个文件拷贝到另一个位置，从而躲开那些基于文件路径的规则。

www.TopSage.com



实验：观察软件限制策略的强制使用情况

当你试图执行一个未被允许的映像文件时，你通过观察对注册表的访问，就可以间接地看到软件限制策略被强制使用的情形。

1. 运行secpol.msc来打开本地安全策略编辑器，并且转到软件限制策略（Software Restriction Policies）节点上。
2. 如果尚没有策略定义的话，则从环境菜单中选择Create New Policies。
3. 针对\Windows\System32\Notepad.exe，创建一个基于路径的不允许限制策略。
4. 运行Regmon，并且设置一个include过滤器来查找“Safer”（关于Regmon的介绍，请参见第4章）。
5. 打开一个命令提示窗口，从命令提示符下运行Notepad。

像这样试图运行Notepad，你应该会得到一个消息，它告诉你不能运行你所指定的程序，而且Regmon应该显示了命令提示程序（cmd.exe）正在查询本地机器的限制策略。

8.7 本章总结

Windows提供了一组广泛的安全功能，既满足了政府部门的关键需求，也满足了商业机构的安全需要。在本章中，我们简要地浏览了作为这些安全特性之基础的各个内部组件。

在下一章中，我们来讨论本书中最后一个大的执行体组件：I/O系统。

www.TopSage.com

www.TopSage.com

I/O System

第9章 I/O系统

Microsoft Windows I/O系统是由几个执行体组件构成的，它们合起来管理硬件设备，并向应用程序和系统提供接口以便操纵这些硬件设备。在本章中，我们首先列出I/O系统的设计目标，这些设计目标已经影响了I/O系统的实现。接着，我们将讨论I/O系统的各个组件，包括I/O管理器、即插即用（PnP）管理器和电源管理器。然后我们将检查I/O系统和各种类型设备驱动程序的结构和组件。我们将看一看一些用来描述设备、设备驱动程序和I/O请求的关键数据结构，之后，我们将介绍在完成I/O请求过程中的必要步骤。最后，我们将介绍设备检测、驱动程序安装和电源管理的工作原理。

9.1 I/O系统组件

Windows I/O系统的设计目标是，为应用程序提供一个有关设备的抽象，包括硬件设备（物理）和软件（虚拟或逻辑）设备。此设备抽象具有以下特性。

- 统一的、跨设备的安全性和命名机制，以保护那些可共享的资源（关于Windows安全模型的描述，请参见第8章）；
- 高性能的、异步的、基于数据包的I/O，以支持那些可伸缩应用程序的实现；
- 提供了一些服务，以支持用高级语言来编写驱动程序，以及易于在不同的体系结构之间移植驱动程序；
- 层次结构和扩展性，以允许加入新的驱动程序，从而在不改变其他驱动程序的情况下，透明地改变这些驱动程序或对应设备的行为；
- 动态地加载和卸载设备驱动程序，因而可以做到按需加载驱动程序，并且当不需要驱动程序的时候，它们不会消耗系统资源；
- 支持即插即用，也就是说，系统可以为新检测到的硬件找到并安装驱动程序，然后为它们分配所需要的资源；
- 支持电源管理，因此，系统和单独的设备可以进入低电源状态；

www.TopSage.com

- 支持多个可安装的文件系统，包括FAT、CD-ROM文件系统（CDFS）、通用磁盘格式（UDF，Universal Disk Format）文件系统和Windows文件系统（NTFS）（有关文件系统类型和结构的更多信息，请参考第12章）；
- 对于Windows管理规范（WMI）的支持和诊断能力，因而可以通过WMI应用程序和脚本来管理和监视驱动程序（第4章介绍了WMI）。

为了实现这些特性，Windows I/O系统包含了以下几个执行体组件以及设备驱动程序，如图9.1中所示。

- I/O管理器是I/O系统的核心。它将“应用程序和系统组件”与“虚拟的、逻辑的和物理的设备”连接起来，而且它定义了基础设施来支持设备驱动程序；
- 设备驱动程序通常为某一种特定类型的设备提供一个I/O接口。设备驱动程序接收来自I/O管理器传送给它们的命令，当这些命令完成时，它们通知I/O管理器。设备驱动程序通常使用I/O管理器来向其他的共享同一设备接口或控制的设备驱动程序转送I/O命令；
- PnP管理器与I/O管理器和一种称为**总线型驱动程序**（*bus driver*）的设备驱动程序紧密地一起工作，以指导硬件资源的分配，以及检测并响应硬件设备的加入和移除。PnP管理器和总线型驱动程序负责在检测到一个设备时，将该设备的驱动程序加载进来。当一个设备被加入到一个缺乏正确的设备驱动程序的系统中时，执行体即插即用组件就会请求用户模式PnP管理器的设备安装服务；
- 电源管理器也与I/O管理器紧密地一起工作，以指导系统和各个单独的设备驱动程序经过电源状态的转变；
- 一组Windows管理规范（WMI）支持例程，它们被称为Windows驱动程序模型（WDM）WMI提供者。设备驱动程序可以使用WDM WMI提供者作为中间媒介与用户模式的WMI服务进行通信，因而它们间接地成了WMI提供者（有关WMI的更多信息，请参见第4章的“Windows管理规范”一节）；
- 注册表被当作一个数据库，保存了当前系统上附载的基本硬件设备的描述信息，以及关于驱动程序初始化和配置的设置信息（有关更多的信息，请参见第4章的“注册表”一节）；

www.TopSage.com

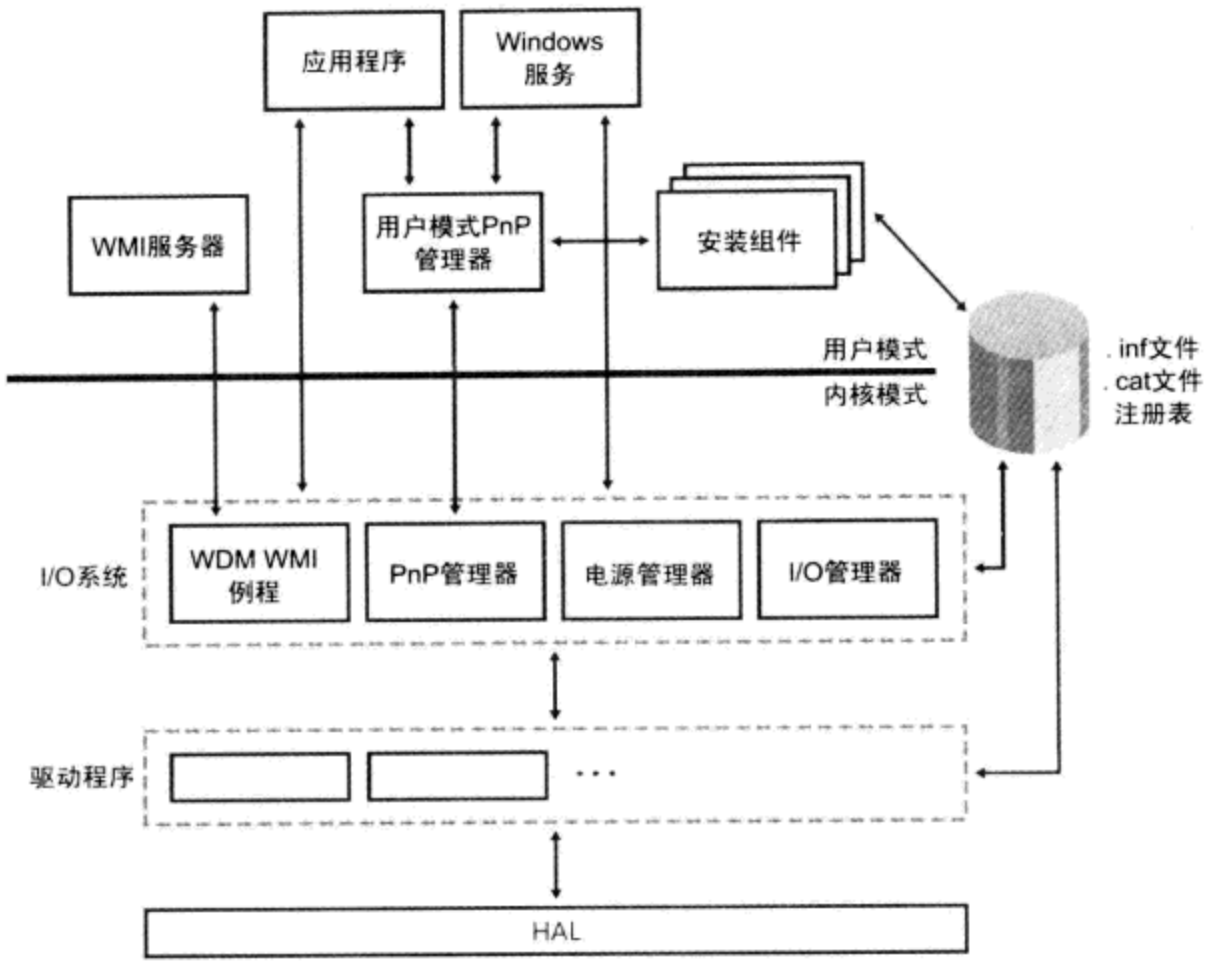


图9.1 I/O系统的组成部件

- INF文件是驱动程序安装文件，它是通过.inf扩展名来指定的。INF文件是一个特定的硬件设备与作为该设备主控制的驱动程序之间的纽带。INF文件是由像脚本类型的指令构成的，这些指令描述了该INF文件所对应的设备、驱动程序文件的源位置和目标位置、安装驱动程序所需要的注册表修改，以及驱动程序的相依性信息。Windows利用数字签名来检验一个驱动程序文件已经通过了Microsoft Windows硬件质量实验室（WHQL，Windows Hardware Quality Lab）的测试，这些数字签名被保存在.cat文件中；
- 硬件抽象层（HAL）通过提供一组API，将不同平台之间的差异隐藏起来，从而使驱动程序与处理器和中断控制器的特殊性相互隔离开。从本质上来说，HAL是“一台计算机的主板上所有不受其他设备控制的设备”的总线型驱动程序。

I/O管理器

I/O管理器是I/O系统的核心，因为它定义了一个非常整齐的框架（或者称为模型），在这个框架内部，I/O请求被递交给设备驱动程序进行处理。I/O系统是**包驱动的**（*packet driven*）。绝大多数I/O请求都是通过一个**I/O请求包**（IRP，*I/O request packet*）来表示的，这些I/O请求包被从一个I/O系统组件传递到另一个I/O系统组件（正如你将在“快速I/O”一节中看到的那样，快速I/O是一个例外：它没有使用IRP）。这种设计使得单个应用程序线程可以并发地管理多个

www.TopSage.com

I/O请求。IRP是一个数据结构，其中包含了用来描述一个I/O请求的完整信息（在“I/O请求包”一节中你将会看到更多有关IRP的信息）。

I/O管理器创建一个IRP来代表一个I/O操作，并且将该IRP传递给正确的驱动程序，当此I/O操作完成时再处理该请求包。相反地，驱动程序接收一个IRP，执行该IRP指定的操作，然后将IRP传回给I/O管理器，告诉它：该操作已完成，或者应该传给另一个驱动程序以作进一步处理。

除了创建和处理IRP以外，I/O管理器也提供了一些对于不同驱动程序都通用的代码，驱动程序可以调用这些代码来完成它们的I/O处理。由于这些通用的任务都被合并到I/O管理器中了，因此，单独的驱动程序变得更为简单，也更为紧凑。例如，I/O管理器提供了这样一个功能：允许一个驱动程序调用其他的驱动程序。它也为I/O请求管理缓冲区，为驱动程序提供超时支持，记录下哪些可安装的文件系统被加载到操作系统中。在I/O管理器中有将近100个不同的例程，可供设备驱动程序调用。

I/O管理器也提供了灵活的I/O服务，通过这些I/O服务，使得环境子系统（比如Windows和POSIX）实现其各自的I/O功能。这些服务包括专门针对异步I/O的复杂服务。开发人员利用这些异步I/O服务，可以建立起可扩展的高性能服务器应用程序。

驱动程序所具有的一致的、模块化的接口使得I/O管理器能够调用任何一个驱动程序，而无需任何有关驱动程序数据结构或内部细节的专门知识。操作系统对待所有的I/O请求就好像它们都是针对一个文件进行操作一样：驱动程序将这些请求从一个针对虚拟文件的请求，转变成一个与硬件相关的请求。驱动程序也可以相互之间调用（使用I/O管理器），从而实现了I/O请求的分层的、独立的处理过程。

除了提供常规的打开、关闭、读和写功能以外，Windows I/O系统还提供了一些高级的特性，比如异步的、直接的、缓冲的，以及分散/聚集的I/O。在本章后面的“I/O的类型”一节中讲述了这些高级特性。

典型的I/O处理过程

绝大多数I/O操作并不涉及I/O系统的所有组件。典型的I/O请求首先从“一个应用程序执行一个与I/O相关的函数（比如从一个设备中读取数据）”开始，此请求将由I/O管理器、一个或多个设备驱动程序和HAL来处理。

正如刚刚提到的，在Windows中，线程在虚拟文件上执行I/O操作。操作系统将所有的I/O请求都抽象成针对一个虚拟文件的操作，从而掩盖了“一个I/O操作的目标可能不是一个文件结构的设备”这样的事实。这一抽象也使得应用程序对待设备的接口变得更为泛化。所谓虚拟文件，是指任何可被当作文件来对待的I/O源或目标（比如文件、目录、管道和邮件槽）。所

www.TopSage.com

有被读或写的数据都被看作一个简单的、位于这些虚拟文件中的字节流。用户模式的应用程序（不管是Windows还是POSIX）都调用文档化的函数，这些函数又依次调用内部的I/O系统函数来读一个文件、写一个文件，或者执行其他的操作。I/O管理器动态地将这些虚拟文件请求引导到正确的设备驱动程序上。图9.2显示了一个典型I/O请求的流程的基本结构。



图9.2 一个典型I/O请求的流程

在下面的章节中，我们将更加细致地看一看这些组件，讨论各种设备驱动程序的类型、它们的结构如何组织、它们如何加载和初始化，以及它们如何处理I/O请求。然后，我们将介绍PnP管理器和电源管理器的操作及其角色。

9.2 设备驱动程序

一个设备驱动程序为了与I/O管理器和其他的I/O系统组件集成在一起，必须符合“与它所管理的设备类型相一致的实现规范”，也要与“它在管理该设备时所扮演的角色”相符。在本节中，我们来看一看Windows所支持的设备驱动程序的类型，以及一个设备驱动程序的内部结构。

设备驱动程序的类型

Windows支持一组范围广泛的设备驱动程序类型和各种编程环境。即使在同一种类型的设备驱动程序的内部，编程环境也可能不尽相同，这取决于一个驱动程序所针对的目标设备的具体类型。对于一个驱动程序，最为宽泛的分类是，它是一个用户模式驱动程序还是一个内核模式驱动程序。

www.TopSage.com

Windows支持以下几种类型的用户模式驱动程序。

- **虚拟设备驱动程序**（VDD）被用于仿真16位MS-DOS应用程序。它们把“MS-DOS应用程序所认为是I/O端口的引用”捕获住，并且将它们转译成原生的Windows I/O函数，然后，通过这些函数传递到实际的设备驱动程序中。因为Windows是一个完全受保护的操作系统，所以，用户模式的MS-DOS应用程序不能直接访问硬件，因此必须通过一个实际的内核模式设备驱动程序才能执行；
- Windows子系统**打印机驱动程序**（*printer driver*）把与设备无关的图形请求转译成与打印机相关的命令。然后这些命令通常被转送到一个内核模式的端口驱动程序中，比如并行端口驱动程序（Parport.sys）或者通用串行总线（USB，universal serial bus）打印机端口驱动程序（Usbprint.sys）。

在本章中，我们把焦点放在内核模式设备驱动程序上。内核模式的驱动程序有很多种类型，它们可以被划分成以下基本的类别。

- **文件系统驱动程序**（*file system driver*）接受针对文件的请求，然后，通过向大容量存储设备（mass storage）或网络设备驱动程序发送它们自己的、更为显式的请求，以此来满足所接收到的I/O请求；
- **即插即用驱动程序**（*Plug and Play driver*）与硬件一起工作，并且与Windows电源管理器和PnP管理器结合起来。它们包含了大容量存储设备、视频适配器、输入设备和网络适配器的驱动程序；
- **非即插即用驱动程序**（*Non-Plug and Play driver*，也称为**内核扩展**）扩展了系统的功能，它们为用户模式的程序提供了“访问内核模式的服务和驱动程序”的途径。它们并不与PnP管理器或电源管理器结合在一起。这种驱动程序的例子有网络API和协议驱动程序。第4章中讲述的Regmon的驱动程序就是这样一个例子。

在内核模式驱动程序的类别之中，还可以进一步根据一个驱动程序所遵从的驱动程序模型和它在处理设备请求过程中所处的角色来进行分类。

WDM驱动程序

WDM驱动程序是指遵从Windows驱动程序模型（WDM，Windows Driver Model）的设备驱动程序。WDM包含了对Windows电源管理、即插即用和WMI的支持，而且，绝大多数即插即用驱动程序都遵从WDM规范。Windows、Windows 98和Windows ME（Millennium Edition）都实现了WDM，所以，在这些操作系统上，WDM驱动程序是源代码兼容的，而且，在许多情况下，它们也是二进制兼容的。WDM驱动程序有以下三种类型。

- **总线型驱动程序**（*bus driver*）管理一个逻辑的或物理的总线。总线的例子有PCMCIA、PCI、USB、IEEE 1394和ISA。一个总线型驱动程序负责检测在它控制的总线上所附载的设备，检测到以后通知PnP管理器，它也管理该总线的电源设置；

www.TopSage.com

- **功能型驱动程序**（*function driver*）管理某一特定类型的设备。总线型驱动程序通过PnP管理器，将设备展示给功能型驱动程序。所谓功能型驱动程序，是指该驱动程序将一个设备的操作接口导出给操作系统。一般而言，它是掌握了最多关于该设备操作的知识的驱动程序；
- **过滤型驱动程序**（*filter driver*）在逻辑上位于功能型驱动程序之上层或下层，增加或者改变一个设备或另一个驱动程序的行为。例如，一个键盘捕获工具可能是通过一个键盘过滤型驱动程序来实现的，该驱动程序位于键盘功能型驱动程序之上层。

在WDM中，没有一个驱动程序负责控制一个特定设备的所有方面。总线型驱动程序负责检测总线上成员的变化（设备加入或移除），帮助PnP管理器列举该总线上的设备，访问与该总线有关的配置寄存器，以及在某些情况下，还要控制该总线上设备的电源。功能型驱动程序通常只是一个访问该设备硬件的驱动程序而已。



> **注** 在Windows 2000、Windows XP和Windows Server 2003中，HAL的角色与它在Windows NT中的角色并不相同。在Windows 2000以前，第三方硬件厂商如果想要为“Windows系统本身并不支持的硬件总线”提供支持，则必须实现一个定制的HAL。Windows 2000、Windows XP和Windows Server 2003允许第三方实现一个总线型驱动程序，为Windows系统本身并不支持的硬件总线提供支持。

分层的驱动程序

系统对硬件单独部分的支持通常被分散在几个驱动程序之中，每个驱动程序提供了一部分功能，这些功能只有联合起来才能使该设备正确地工作。除了WDM总线型驱动程序、功能型驱动程序和过滤型驱动程序以外，对硬件的支持也可能在以下的组件之间划分。

- **类驱动程序**（*class driver*）实现了某一特定类型的设备的I/O处理。这种特定类型的设备如磁盘、磁带、CD-ROM等。在这种特定类型的设备中，硬件接口已经被标准化了，所以，一个驱动程序可以为来自各种生产商的设备提供服务；
- **端口驱动程序**（*port driver*）实现了与某一类型I/O端口相关的I/O请求的处理，比如SCSI，它们被实现为内核模式的函数库，而不是真正的设备驱动程序；
- **小端口驱动程序**（*miniport driver*）将“针对一种类型端口的通用I/O请求”映射为一种适配器类型，比如某种特定的SCSI适配器。小端口驱动程序是真正的设备驱动程序，它们需要导入（import）一个端口驱动程序所提供的函数。

介绍一个例子将有助于说明这些设备驱动程序是如何工作的。文件系统驱动程序接受一个“在某个特殊文件中的特定位置上写入数据”的请求。它把该请求转译成一个“在磁盘的特定‘逻辑’位置上写入特定数量的字节”的请求。然后，它把此请求（通过I/O管理器）传递给一个简单的磁盘驱动程序。该磁盘驱动程序又依次将此请求转译成针对该磁盘某一物理位置（柱面/磁道/扇区）的请求，并且操纵磁头写入数据。这一层次结构如图9.3所示。

www.TopSage.com

该图示出了两个分层的驱动程序之间的劳动分工。I/O管理器接收一个写请求，此请求相对于一个特定文件的开始处。I/O管理器把此请求传递给文件系统驱动程序，该驱动程序将这个写操作从一个相对于文件的操作，转译成一个开始位置（磁盘上的一个扇区边界）和要写入的字节数量。文件系统驱动程序调用I/O管理器，将此请求传递给磁盘驱动程序，然后，磁盘驱动程序把此请求转译成一个物理磁盘位置，并且传送数据。

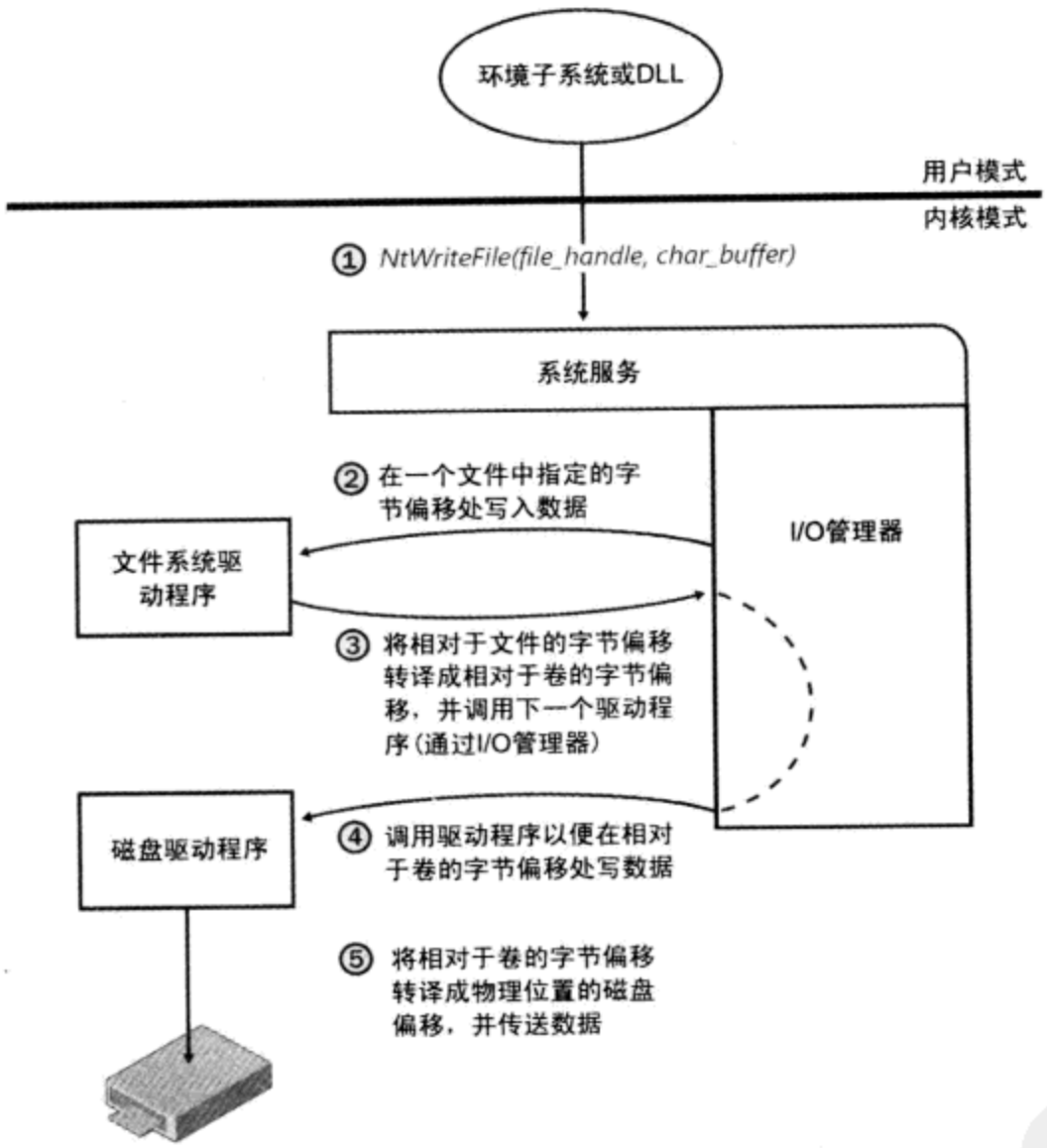


图9.3 一个文件系统驱动程序和一个磁盘驱动程序的分层结构

因为所有的驱动程序——无论是设备驱动程序还是文件系统驱动程序——展示给操作系统的框架都是相同的，所以，另一个驱动程序可以很容易地插入到这一层次结构中，而无需改变已有的驱动程序或者I/O系统。例如，只要加入一个驱动程序，就可以让几个磁盘变得像一个大的单磁盘那样工作。在Windows中有一个这样的驱动程序提供了对容错磁盘的支持（虽然在所有的Windows版本中都有此驱动程序，但是，对容错磁盘的支持仅限于在Windows服务器

www.TopSage.com

版本上才可以使用）。这一逻辑的卷管理器驱动程序位于文件系统驱动程序和磁盘驱动程序之间，如图9.4所示。第10章更加详细地描述了卷管理器驱动程序。

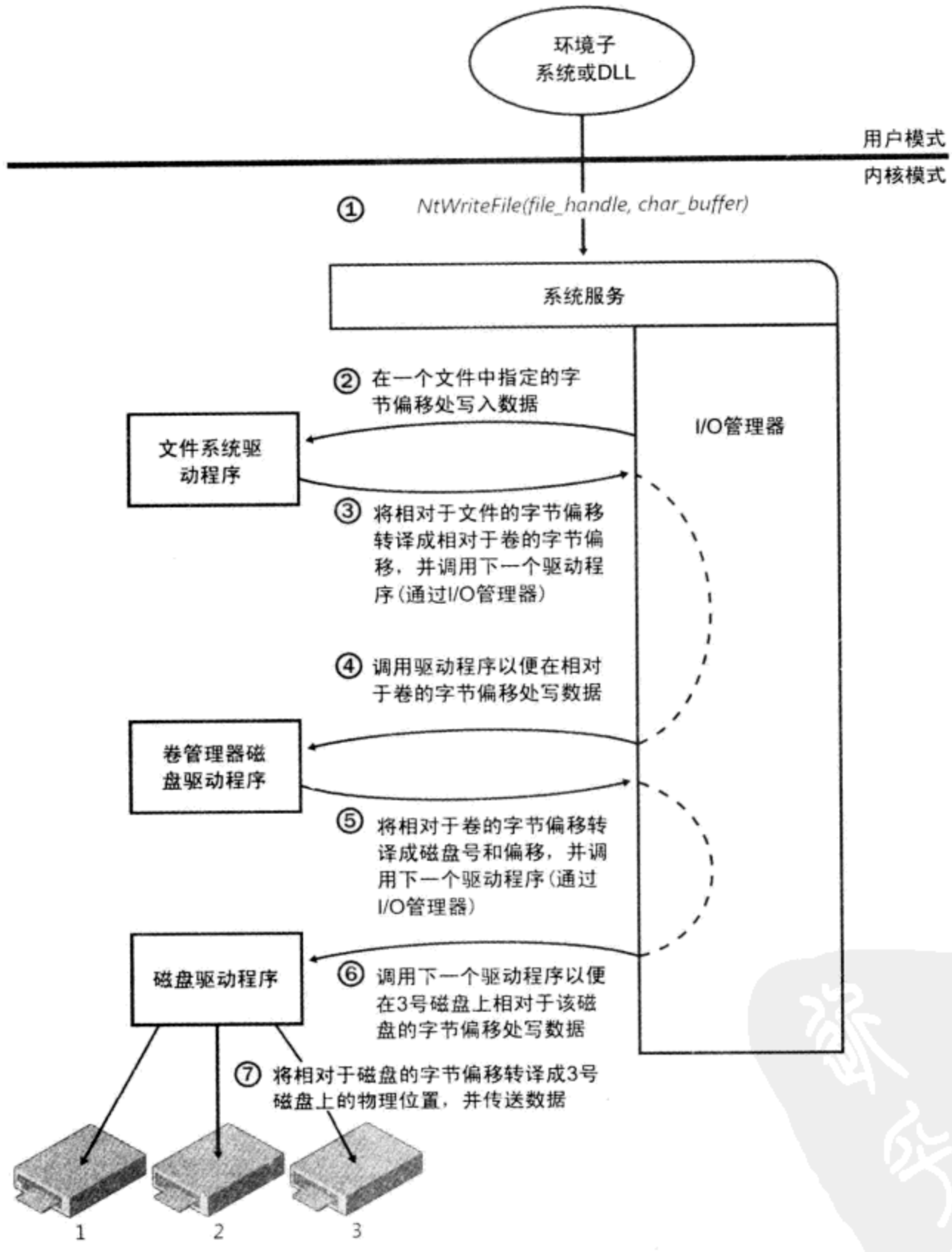


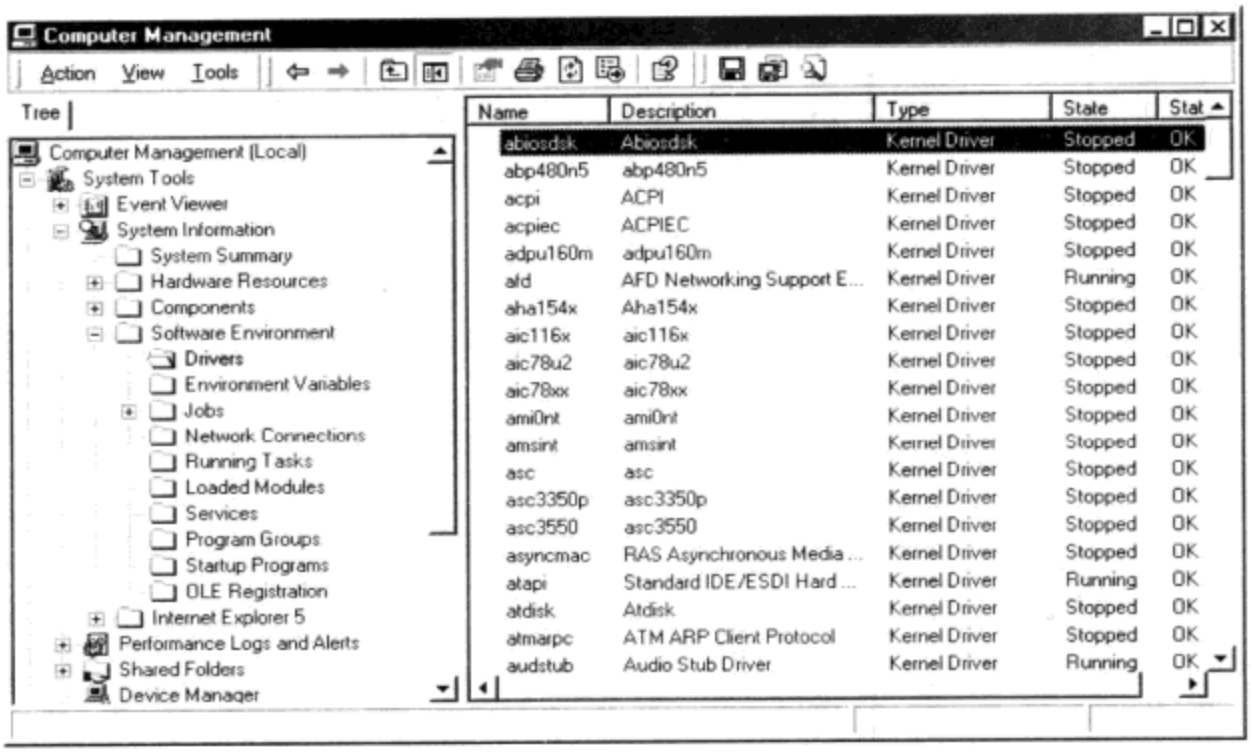
图9.4 加入一个分层驱动程序

www.TopSage.com



实验：查看已加载的驱动程序列表

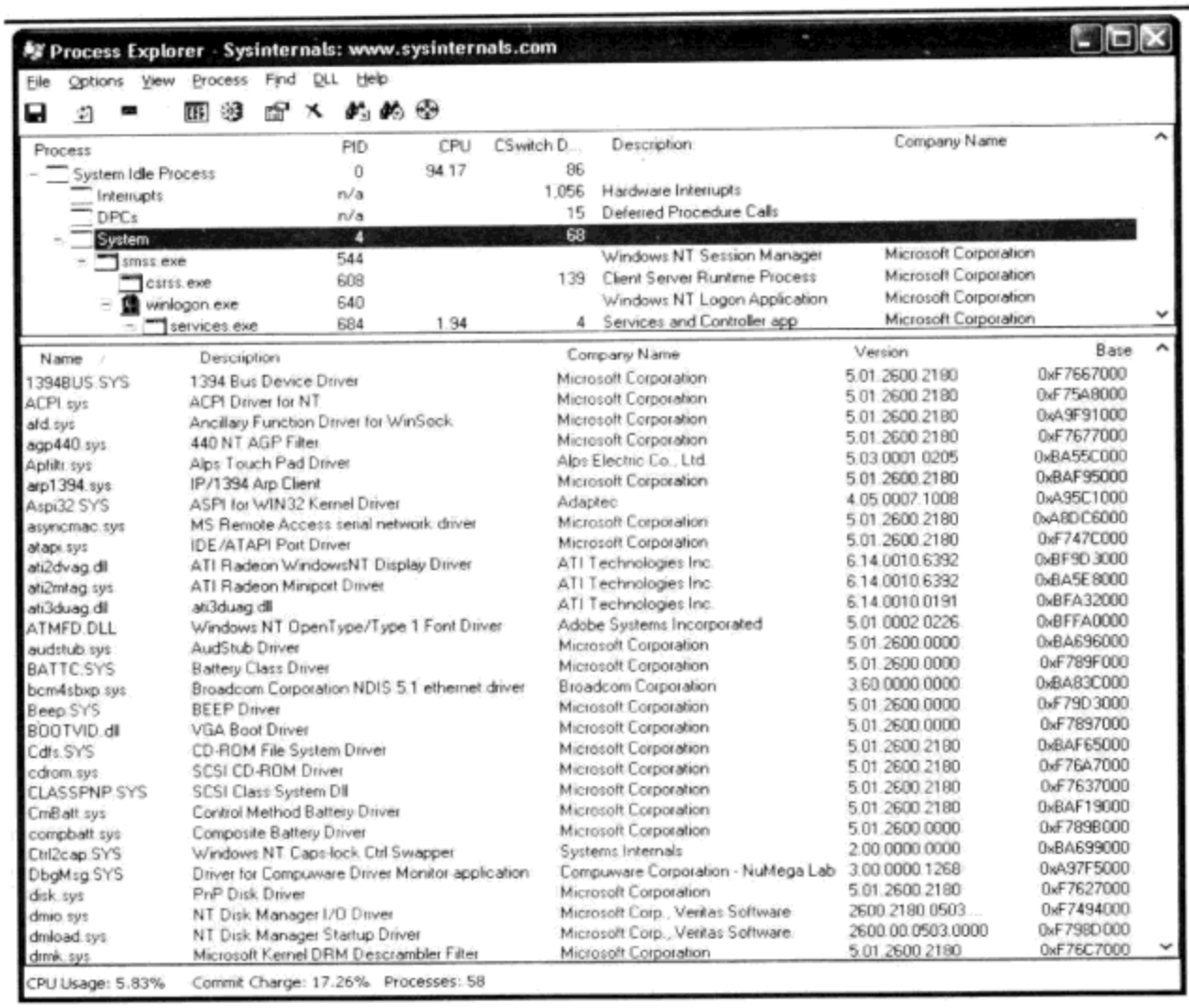
在Windows 2000系统上，你只要在Computer Management（计算机管理）MMC（Microsoft Management Console）加载件中转到Drivers部分，或者在桌面上右键点击My Computer图标，然后从环境菜单中选择Manage，就可以看到已注册驱动程序的列表（计算机管理MMC加载件位于Start菜单的Programs/Administrative Tools文件夹中）。在计算机管理加载件中，为了转到Drivers部分，你只需展开System Tools、System Information、Software Environment，并选中Drivers，如下图所示：



在Windows XP和Windows Server 2003中，你只要在Start菜单的Run对话框中执行Msinfo32.exe工具，就可以得到如同Windows 2000的计算机管理MMC加载件报告的同样信息。在Software Environment下选中System Drivers项，就可以看到当前系统上配置的驱动程序的列表。已经被加载的驱动程序在Started一栏中标明了“Yes”。

你也可以利用*www.sysinternals.com*的进程管理器（Process Explorer）来查看已加载的内核模式驱动程序的列表。运行进程管理器，选中System进程，从View菜单的Lower Pane菜单项中选择DLLs。进程管理器列出已加载的驱动程序、它们的名称、包括公司和说明在内的版本信息，以及加载的地址（假定你已经配置了进程管理器，让它显示相应的栏目），如下图所示：

www.TopSage.com



最后，如果你正在使用内核调试器来检查一个崩溃转储（或者实时系统），那么，通过内核调试器的*lm kv*命令，你可以得到类似的输出：

```
kd> lm kv
start    end        module name
804d4000 806aa280   nt           (pdb symbols)          c:\Symbols\ntoskrnl.pdb\
FB1EDACE71FB4812A5D5132819D72E523\ntoskrnl.pdb
    Loaded symbol image file: ntoskrnl.exe
    Image path: ntoskrnl.exe
    Timestamp: Thu Apr 24 10:57:43 2003 (3EA80977)  Checksum: 001E311B
    ImageSize : 001D6280
    File version:     5.1.2600.1151
    Product version:  5.1.2600.1151
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Windows
    File type:        1.0 App
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft" Windows« Operating System
    InternalName:     ntoskrnl.exe
    OriginalFilename: ntoskrnl.exe
    ProductVersion:   5.1.2600.1151
    FileVersion:      5.1.2600.1151 (xpsp2.030422-1633)
    FileDescription:  NT Kernel & System
    LegalCopyright:    Microsoft Corporation. All rights reserved.
```


www.TopSage.com

```
806ab000 806bde80   hal          (deferred)
    Image path: halacpi.dll
    Timestamp: Thu Aug 29 03:05:02 2002 (3D6DD5AE)  Checksum: 000203BD
    ImageSize : 00012E80
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
a8b8e000 a8bb4e80   kmixer       (deferred)
    Image path: \SystemRoot\system32\drivers\kmixer.sys
    Timestamp: Thu Aug 29 03:32:28 2002 (3D6DDC1C)  Checksum: 00032574
    ImageSize : 00026E80
    Translations:     0000.04b0 0000.04e0 0409.04b0 0409.04e0
```

驱动程序的结构

I/O系统驱动了设备驱动程序的执行。设备驱动程序是由一组例程构成的，在I/O系统处理一个I/O请求的各个不同阶段，这些例程会被调用。图9.5显示了一个驱动程序的关键例程。

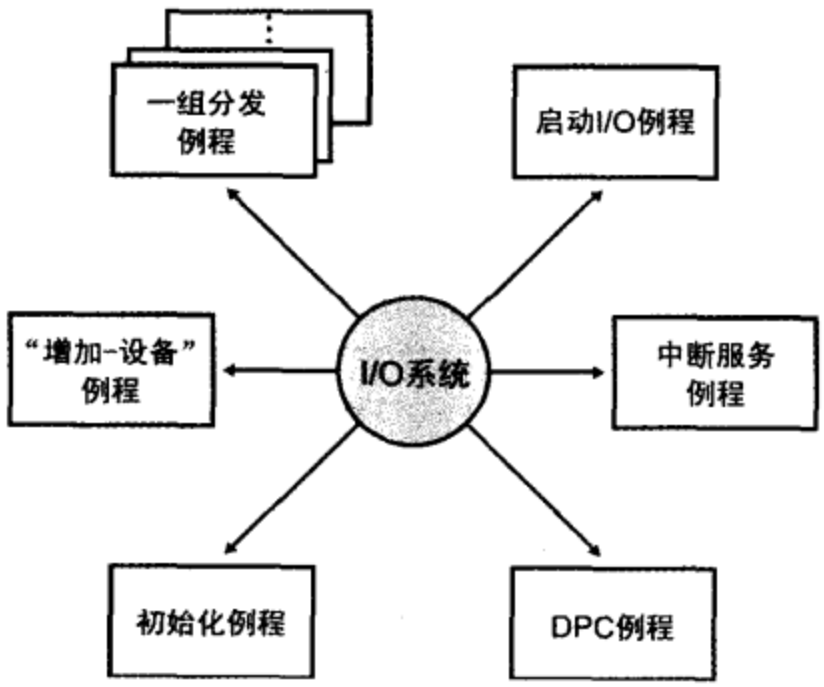


图9.5 主要的设备驱动程序例程

- **初始化例程** 当I/O管理器将一个驱动程序加载到操作系统中时，它执行该驱动程序的初始化例程，此例程通常被命名为*DriverEntry*。此例程填充一些系统数据结构，以便向I/O系统登记该驱动程序其余的例程，同时它也执行任何必要的、有关该驱动程序的全局初始化工作；
- **“增加-设备”例程**（Add-device routine） 一个支持即插即用的驱动程序需要实现一个“增加-设备”例程。无论何时当PnP管理器检测到由该驱动程序来负责的设备时，它通过此例程向该驱动程序发送一个通知。驱动程序在这个例程中，通常要申请一个设备对象（本章后面将会讲述）来表达此设备；

www.TopSage.com

- **一组分发例程** 分发例程是一个设备驱动程序所提供的主要函数。这种函数的例子有打开、关闭、读和写，以及该设备、文件系统或网络所支持的任何其他能力。当I/O管理器被要求执行一个I/O操作时，它生成一个IRP，并且通过该驱动程序的某一个分发例程来调用该驱动程序；
- **启动I/O例程** 驱动程序可以使用一个启动I/O例程来发起一次数据传输（发送至该设备或者从该设备发送至别处）。只有对于那些需要依赖于I/O管理器来对它的进入I/O请求进行排队的驱动程序，才需要定义此例程。I/O管理器确保一个驱动程序每次只处理一个IRP，从而对该驱动程序的IRP进行串行化。大多数驱动程序可以并发地处理多个IRP，但是这种串行化对于某些驱动程序是（比如键盘驱动程序）有意义的；
- **中断服务例程（ISR）** 当一个设备中断时，内核的中断分发器把控制权传递给此例程。在Windows I/O模型中，ISR是在设备中断请求级别（DIRQL，device interrupt request level）上运行的，所以，它们尽可能执行很少的工作，以避免不必要地阻塞低级别的中断（关于IRQL的更多信息，请参见第3章）。ISR把一个延迟过程调用（DPC，deferred procedure call）排到队列中，由它执行余下的中断处理过程；DPC运行在一个更低的IRQL（DPC/Dispatch级别）上（只有那些由中断驱动的设备的驱动程序，才有ISR。譬如，文件系统驱动程序没有ISR）；
- **中断服务的DPC例程** 当一个设备中断发生时，在ISR执行以后，绝大多数处理该中断的工作是由一个DPC例程来完成的。该DPC例程在一个低于设备级别（ISR的执行级别）的IRQL（DPC/Dispatch级别）上执行，以避免不必要地阻塞其他的中断。DPC例程激发“I/O完成”动作，并启动一个设备上的下一个排队的I/O操作；

虽然图9.5中没有显示以下的例程，但是在许多类型的设备驱动程序中都可以看到它们：

- **一个或者多个I/O完成例程** 一个分层的驱动程序可能有一些I/O完成例程，当一个低层的驱动程序完成了一个IRP的处理时，就会通过这些I/O完成例程通知该驱动程序。例如，当一个设备驱动程序完成了一个文件的数据传输时，I/O管理器就会调用文件系统驱动程序的I/O完成例程。该完成例程通知此文件系统驱动程序，该I/O操作是成功的还是失败的，或者被取消了。这使得文件系统驱动程序可以执行一些清理操作；
- **取消I/O例程** 如果一个I/O操作可以被取消，则驱动程序可以定义一个或者多个取消I/O例程。当驱动程序接收到一个IRP并且该IRP代表一个可被取消的I/O请求时，它给该IRP分配一个取消例程。如果发出I/O请求的线程在该请求被完成以前退出了，或者它取消了该操作（譬如，通过Windows函数*CancelIo*），那么，若该IRP被分配了一个取消例程，则I/O管理器执行此取消例程。取消例程的任务是，执行必要的步骤来释放在该IRP处理过程中已经获得的任何资源，以及用一个取消状态来完成此IRP；

www.TopSage.com

- **卸载例程** 卸载例程释放一个驱动程序所用到的任何系统资源，因此，I/O管理器可以将它们从内存中移除出去。任何在初始化例程中获得的资源通常都应该在卸载例程中被释放掉。一个驱动程序可以在系统运行时被加载到系统中，或者从系统中卸载出去；
- **系统停机通知例程** 此例程使得驱动程序可以在系统停机的时候做一些清理工作；
- **“错误-记录”例程** 当意料之外的错误发生（比如，一个磁盘块坏了）时，驱动程序的“错误-记录”例程会注意到该事件的发生，并且通知I/O管理器。I/O管理器将这些信息写到一个错误日志文件中。



注 绝大多数设备驱动程序都是用C来编写的。强烈不鼓励使用汇编语言，因为它会引入复杂性，并且使一个驱动程序难以在不同的硬件体系结构（比如x86、x64和IA64）之间进行移植。

驱动程序对象和设备对象

当一个线程打开一个指向文件对象的句柄（本章后面的“打开设备”一节中讲述）时，I/O处理器必须根据该文件对象的名称来确定：它应该调用哪个驱动程序（或哪些驱动程序）来处理该请求。更进一步，当下一次一个线程使用同样的文件句柄时，I/O管理器必须能够找到这些信息。下面的系统对象满足这一要求。

- **驱动程序对象**（*driver object*）代表了系统中一个单独的驱动程序。I/O管理器从该驱动程序对象中获得其每一个分发例程（入口点）的地址；
- **设备对象**（*device object*）代表了系统中一个物理的或逻辑的设备，并且描述了它的特征，比如它要求的缓冲区的对齐特性，以及它的设备队列（用于存放进来的IRP）的位置。

当一个驱动程序被加载到一个系统中时，I/O管理器创建一个驱动程序对象，然后它调用该驱动程序的初始化例程（例如，*DriverEntry*），该例程会利用该驱动程序的入口点来填充此对象的属性。

驱动程序被加载进来以后，它可以在任何时候通过调用*IoCreateDevice*或*IoCreateDeviceSecure*来创建一些代表设备（或甚至该驱动程序的接口）的设备对象。然而，绝大多数即插即用驱动程序是在PnP管理器通知它们，它们负责管理的设备出现之时，才通过它们的“增加-设备”例程来创建设备对象的。另一方面，非即插即用的驱动程序则通常是在I/O管理器调用它们的初始化例程时创建设备对象的。当一个驱动程序的最后一个设备对象已被删除，并且不再存在指向该驱动程序的引用时，I/O管理器卸载该驱动程序。

www.TopSage.com

一个驱动程序创建一个设备对象时，它可以有选择地为该设备分配一个名称。设备对象的名称使得它能够置身于对象管理器名字空间中；驱动程序可以显式地定义一个名称，也可以让I/O管理器自动生成一个（第3章讲述了对象管理器名字空间）。按照约定，设备对象被放在名字空间的\Device目录中，应用程序通过Windows API是无法访问该目录的。



> **注** 有些驱动程序把设备对象放在\Device以外的其他目录中。例如，Windows逻辑磁盘管理器（Windows Logical Disk Manager）卷管理器把代表磁盘分区的设备对象放在Device\HarddiskDmVolumes目录中。关于存储体系结构的介绍，包括存储驱动程序使用设备对象的方法，请参见第10章。

如果一个驱动程序需要让应用程序能够打开该设备对象，那么，它必须要在\Global??（或者在Windows 2000中为\??）目录中创建一个符号链接，并且使它指向\Device目录中该设备对象的名称（有关\??的更多信息，请参见第3章）。非即插即用驱动程序和文件系统驱动程序通常用一个知名的名称来创建一个符号链接，比如\Device\Hardware2。因为这些知名的名称在硬件动态变化（一会儿出现，一会儿消失）的环境中并不能很好地工作，所以，PnP驱动程序通过调用*IoRegisterDeviceInterface*函数，并且指定一个GUID（全局惟一标识符）来暴露一个或者多个接口。这里指定的GUID代表了将要暴露的功能的类型。GUID是一个128位值，你可以利用DDK和Platform SDK中包含的一个称为Guidgen的工具来生成。考虑到128位所能表达的值的范围，从统计意义上几乎可以保证，Guidgen生成的每个GUID将永远都是全局惟一的。

*IoRegisterDeviceInterface*决定了与一个设备实例相关联的符号链接。然而，在I/O管理器真正创建该链接以前，驱动程序必须先调用*IoSetDeviceInterfaceState*，以激活该设备的接口。通常驱动程序是在“PnP管理器通过向驱动程序发送一个**启动-设备**（*start-device*）命令来启动该设备”时做这件事情的。

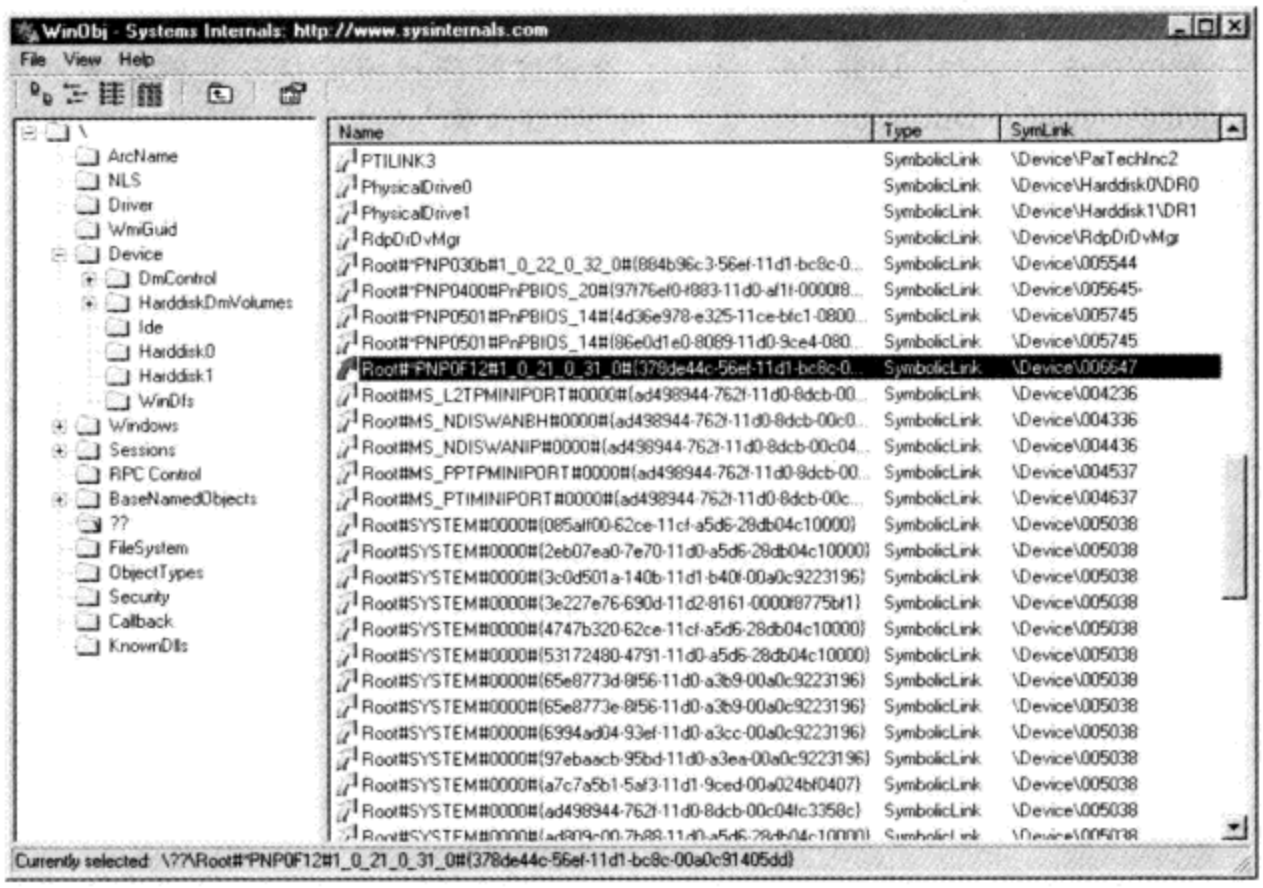
若一个应用程序想要打开一个由GUID来代表的设备对象，则它可以调用用户空间中的即插即用安装函数（比如*SetupDiEnumDeviceInterfaces*）来列举一个特定GUID上出现的接口，以及可用于打开这些设备对象的符号链接的名称。对于*SetupDiEnumDeviceInterfaces*报告的每一个设备，应用程序可以执行*SetupDiGetDeviceInterfaceDetail*以便获得有关该设备的额外信息，比如自动生成的名称。从*SetupDiGetDeviceInterfaceDetail*获得了一个设备的名称以后，应用程序可以执行Windows函数*CreateFile*来打开该设备，并得到一个句柄。

www.TopSage.com



实验：看一看\Device目录

你可以使用来自*www.sysinternals.com*的Winobj工具，或者内核调试器命令*!object*来查看对象管理器名字空间中\Device下面的设备名称。下面的屏幕截图显示了一个由I/O管理器分配的符号链接，它指向\Device中的一个设备对象，其名称是自动生成的。



当你运行内核调试器的命令*!object*并指定\Device目录时，你应该看到类似如下的输出：

```
kd> !object \device
Object: e100c4a0  Type: (8a4f3178) Directory
    ObjectHeader: e100c488
    HandleCount: 0  PointerCount: 301
    Directory Object: e10011e8  Name: Device
    65535 symbolic links snapped through this directory

    Hash Address  Type          Name
    ---- -------  ----          ----
     00  8a437398 Device        KsecDD
         8a4a56f0 Device        Ndis
         8a0ed5c0 Device        ProcExp
         8a1ddb40 Device        Beep
         8a336d38 Device        0000008e
         8a4ed730 Device        00000032
         8a4ee4f0 Device        00000025
         8a4b5030 Device        00000019
```


www.TopSage.com

```
01  8a2303e8 Device        Netbios
        8a258030 Device        0000008f
        8a4ed4f0 Device        00000033
        8a4ee2b0 Device        00000026
    02  8a1756d8 Device        KSENUM#00000001
        8a4ec730 Device        00000040
        8a20fd58 Device        Ip
        8a2ef660 Device        RDP_CONSOLE0
        8a4ed2b0 Device        00000034
        8a4b4030 Device        00000027
    03  8a4ec4f0 Device        00000041
        8a15bf18 Device        0000009e
        8a2947d8 Device        {EFF45047-C948-4D32-86B5-736480DDBB9C}
        8a1b38e0 Device        Fips
        8a0b6038 Device        Video0
        8a288b48 Device        RDP_CONSOLE1
        8a2501f8 Device        KeyboardClass0
        8a4b3030 Device        00000035
        8a4b4df0 Device        00000028
    04  8a181030 Device        NDProxy
        8a4ec2b0 Device        00000042
        8a00b038 Device        Video1
        8a189030 Device        KeyboardClass1
  §
```

当你执行*!object*并指定一个对象管理器目录对象时，内核调试器根据对象管理器的内部组织方式，将该目录的内容转储出来。为了快速查找，目录对象根据这些对象名称的散列值，将这些对象存放在一个散列表中，所以上面的输出显示了该目录的散列表的每个桶中存储的对象。

如图9.6所示，设备对象指回到它的驱动程序对象上，这正是当I/O管理器接收到一个I/O请求时，它如何知道该调用哪一个驱动程序例程的原因。它利用该设备对象来找到一个驱动程序对象，此对象代表了负责为该设备提供服务的驱动程序。然后，它利用原始请求中所提供的功能代码，索引到该驱动程序对象中；每个功能代码对应于驱动程序的一个入口点（图9.6中显示的功能代码，将在本章后面的“IRP栈单元”一节中讲述）。

驱动程序对象通常有多个与之关联的设备对象。这些设备对象的列表代表了该驱动程序所控制的物理和逻辑设备。例如，一个硬盘的每个分区都有一个单独的设备对象，该设备对象包含了与此分区相关的信息。然而，同样的硬盘驱动程序被用于访问所有这些分区。当一个驱动程序被从系统中卸载时，I/O管理器使用设备对象的队列来决定哪些设备将会受到“该驱动程序被移除”的影响。

www.TopSage.com

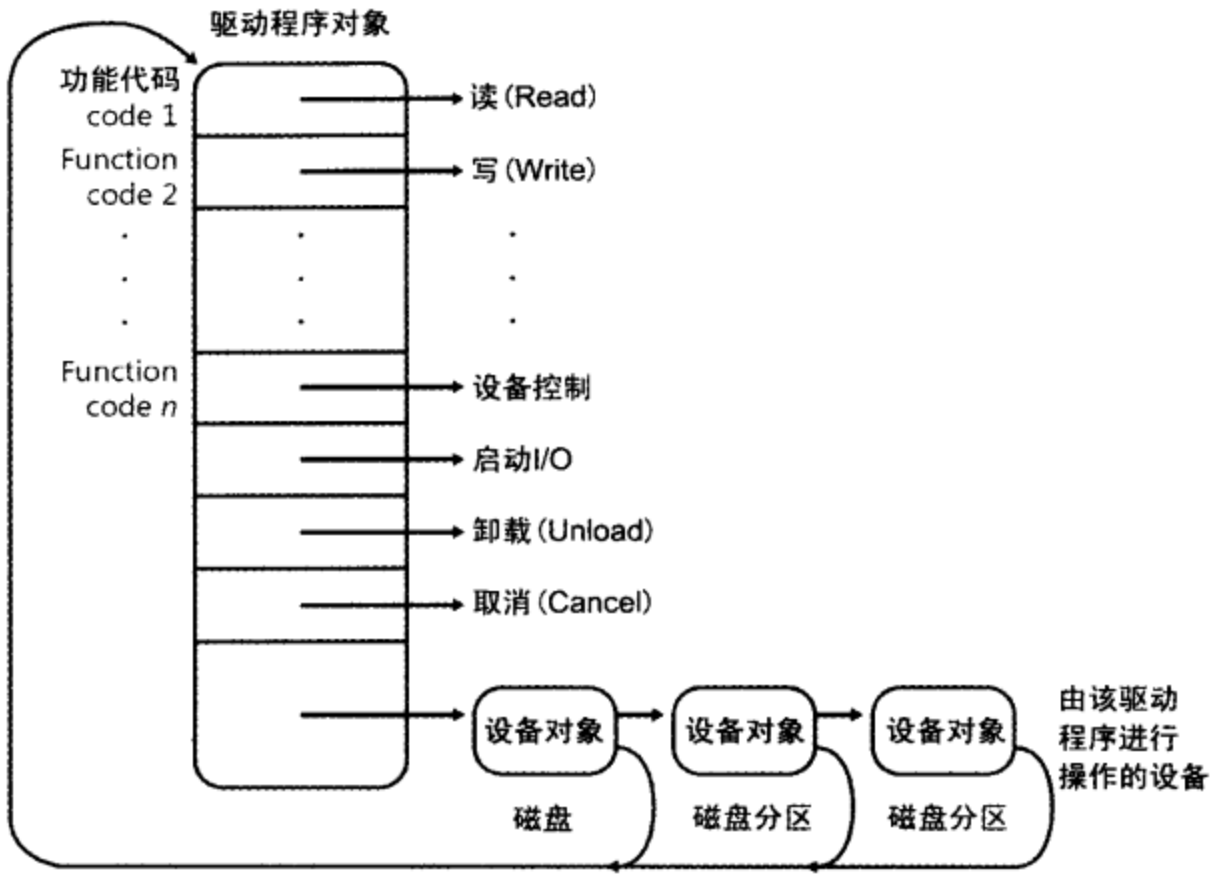


图9.6 驱动程序对象



实验：显示驱动程序和设备对象

利用内核调试器的*!drvobj*和*!devobj*命令，你可以显示驱动程序和设备对象。下面的例子检查了键盘类驱动程序的驱动程序对象，及其惟一的设备对象：

```
kd> !drvobj kbdclass
Driver object (81869cb0) is for:
 \Driver\Kbdclass
Driver Extension List: (id , addr)

Device Object list: 81869310
kd> !devobj 81869310
Device object (81869310) is for:
 KeyboardClass0 \Driver\Kbdclass DriverObject 81869cb0
Current Irp a57a0e90 RefCount 0 Type 0000000b Flags 00002044
DevExt 818693c8 DevObjExt 818694b8
ExtensionFlags (0000000000)  AttachedDevice (Upper) 818691e0 \Driver\Ctrl2cap
AttachedTo (Lower) 81869500 \Driver\i8042prt
Device queue is busy -- Queue empty.
```

注意，*!devobj*命令也向你显示了你想要查看的对象之下的任何设备对象的地址和名称（*AttachedTo*行），以及位于你所指定的对象之上的设备对象的地址和名称（*AttachedDevice*行）。

www.TopSage.com

使用对象来记录有关驱动程序的信息，这意味着I/O管理器不必知道有关单个驱动程序的细节。I/O管理器只需顺着一个指针就能找到一个驱动程序，因而提供了一层灵活性，并且使新的驱动程序很容易被加载进来。用不同的对象来代表设备和驱动程序，这也使得当系统配置发生变化时，I/O系统很容易分配相关的驱动程序来控制额外的或者不同的设备。

打开设备

文件对象是文件句柄或设备句柄的内核模式结构。很显然，文件对象符合Windows中关于对象的标准：它们是两个或者多个用户模式进程可以共享的系统资源，它们可以有名称，它们也受到Windows中基于对象的安全机制的保护，而且它们支持同步。虽然Windows中绝大多数的共享资源都是基于内存的资源，但是，I/O系统管理的绝大多数共享资源要么位于物理设备上，要么代表了实际的物理设备。尽管存在这样的差异，但是，I/O系统中的共享资源，如同Windows执行体的其他组件中的共享资源一样，也是按对象来操作的（关于对象管理器的描述，请参见第3章；关于对象安全性的信息，请参见第8章）。

文件对象为那些“符合以I/O为核心接口的资源”提供了一个以内存为基础的表示法；通过这种以I/O为主的接口，文件对象中的数据可以被读取或者写入。表9.1列出了文件对象的一些属性。有关特定域的声明和大小，请参考Ntddk.h中FILE_OBJECT的结构定义。

表9.1 文件对象的属性

属 性	用 途
文件名	标识了该文件对象所引用的物理文件
当前字节偏移	标识了文件中的当前位置（仅对于同步I/O才有效）
共享模式	指明了当当前调用者正在使用该文件对象时，其他的调用者是否可以打开该文件来进行读、写或者删除操作
打开模式标志	指明了该文件的I/O操作是同步的还是异步的，是缓存的还是非缓存的，是顺序的还是随机的，等等
指向设备对象的指针	指明了该文件所在的设备的类型
指向卷参数块（VPB，volume parameter block）的指针	指明了该文件所在的卷或者分区
指向内存区对象指针的指针	指明了一个根结构（此根结构描述了一个映射文件）
指向私有缓存表的指针	标识了该文件的哪些部分被缓存管理器缓存起来了，以及它们驻留在缓存中什么地方

www.TopSage.com



实验：查看文件对象数据结构

利用内核调试器的dt命令，你可以查看内核模式的文件对象数据结构的内容，如下所示：

```
kd> dt nt!_file_object
nt!_FILE_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Vpb              : Ptr32 _VPB
   +0x00c FsContext        : Ptr32 Void
   +0x010 FsContext2       : Ptr32 Void
   +0x014 SectionObjectPointer : Ptr32 _SECTION_OBJECT_POINTERS
   +0x018 PrivateCacheMap  : Ptr32 Void
   +0x01c FinalStatus      : Int4B
   +0x020 RelatedFileObject : Ptr32 _FILE_OBJECT
   +0x024 LockOperation    : UChar
   +0x025 DeletePending    : UChar
   +0x026 ReadAccess       : UChar
   +0x027 WriteAccess      : UChar
   +0x028 DeleteAccess     : UChar
   +0x029 SharedRead       : UChar
   +0x02a SharedWrite      : UChar
   +0x02b SharedDelete     : UChar
   +0x02c Flags            : Uint4B
   +0x030 FileName         : _UNICODE_STRING
   +0x038 CurrentByteOffset : _LARGE_INTEGER
   +0x040 Waiters          : Uint4B
   +0x044 Busy             : Uint4B
   +0x048 LastLock         : Ptr32 Void
   +0x04c Lock             : _KEVENT
   +0x05c Event            : _KEVENT
   +0x06c CompletionContext : Ptr32 _IO_COMPLETION_CONTEXT
```

当一个调用者打开一个文件或者一个简单设备时，I/O管理器返回一个指向文件对象的句柄。图9.7示出了当一个文件被打开时发生的情形。

在这个例子中，①一个C程序调用运行时库函数*fopen*，该函数依次②调用Windows的*CreateFile*函数。Windows子系统DLL（在这个例子中为Kernel32.dll），然后③调用Ntdll.dll中的原生函数*NtCreateFile*。Ntdll.dll中的例程包含了适当的指令，以转换到内核模式中，并到达系统服务分发器，然后，系统服务分发器④调用Ntoskrnl.exe中实际的*NtCreateFile*例程（关于系统服务分发的更多信息，请参见第3章）。

www.TopSage.com

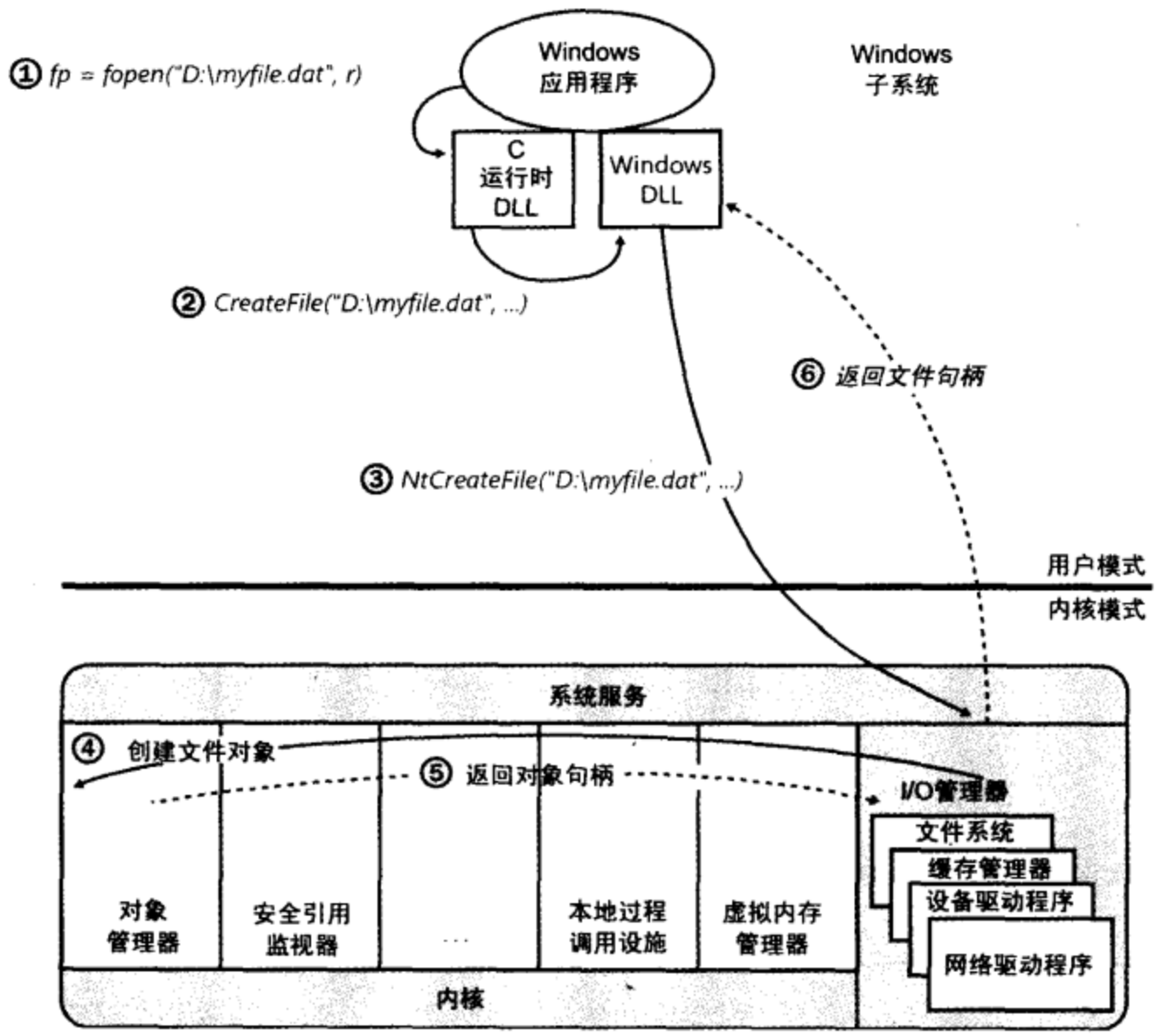


图9.7 打开一个文件对象



> 注 文件对象表达的是文件的已打开实例，而并非文件本身。UNIX系统使用*vnode*，与此不同的是，Windows并没有定义一个文件的表示形式；Windows系统的驱动程序定义它们自己的表示形式。

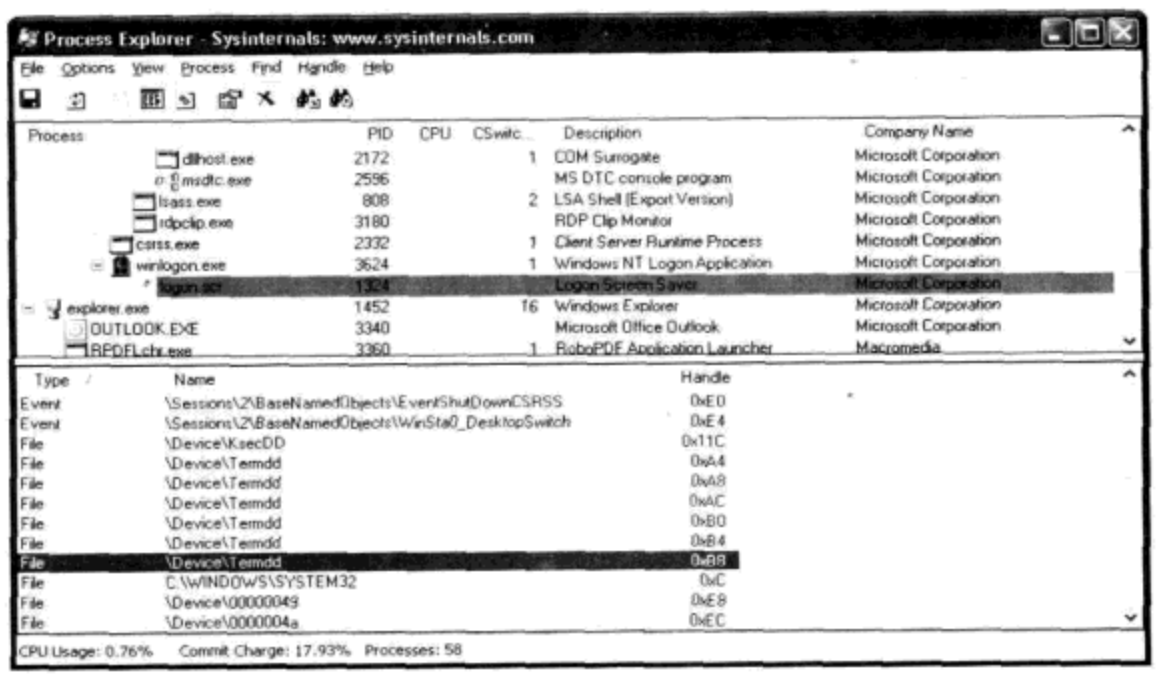
如同其他的执行体对象一样，文件也是由一个安全描述符来保护的，该安全描述符包含了一个访问控制列表（ACL）。I/O管理器需要通过安全子系统的检查，以确定该文件的ACL是否允许一个进程按照它的线程所请求的方式来访问该文件。如果允许访问，则⑤、⑥对象管理器授予此访问权，并且将授予的访问权限与它返回的文件句柄关联起来。如果该进程中的这个线程或者另外的线程需要执行额外的、在原来的请求中没有指定的操作，则该线程必须打开另一个句柄，以执行另一次安全检查（有关对象保护的更多信息，请参见第8章）。

www.TopSage.com



实验：查看设备句柄

任何一个进程，如果它有一个指向设备的已打开句柄，那么，在它的句柄表中有一个文件对象对应于此已打开的实例。你可以利用*www.sysinternals.com*的进程管理器来查看这些句柄。做法如下：选中一个进程，然后在View菜单中选中Show Lower Pane，并且在View菜单的Lower Pane View子菜单中选择Handles。按Type栏进行排序，并且滚动到你看到了代表文件对象的句柄为止。其Type栏显示为“File”，如下图所示：



在这个例子中，Csrss进程中有一些已打开的句柄指向了代表设备实例的文件对象（这些设备具有自动生成的名称）；也有一些已打开的文件对象句柄指向了属于Terminal Server Driver（终端服务器驱动程序）的设备的已打开实例。在内核调试器中，你只要知道了一个特定文件对象的地址，就可以查看该对象的信息。下面的命令显示了上述屏幕截图中被加亮显示的句柄（句柄值0xB8）的有关信息，该句柄位于Csrss.exe进程中，并且Csrss进程的进程ID为2332（ox91c）：

```
0: kd> !handle b8 f 91c
processor number 0
Searching for Process with Cid == 91c
PROCESS 86a6c020  SessionId: 0  Cid: 091c    Peb: 7ffde000  ParentCid: 028c
    DirBase: 1158a000  ObjectTable: e1b5d080  HandleCount: 643.
    Image: csrss.exe

New version of handle table at e2b44000 with 643 Entries in use
00B8: Object: 866ae9e8  GrantedAccess: 0012019f
Object: 866ae9e8  Type: (86fe8ad0) File
    ObjectHeader: 866ae9d0
        HandleCount: 1  PointerCount: 3
```

因为该对象是一个文件对象，所以，你可以利用*!fileobj*命令来获得有关该对象的信息：

www.TopSage.com

```
0: kd> !fileobj 866ae9e8

File object name:

Device Object: 0x86d0c8b8    \Driver\TermDD
Vpb is NULL

Flags:  0x40000
        Handle Created

FsContext: 0x866a31d8    FsContext2: 0x00000001
CurrentByteOffset: 0
```

因为文件对象是一个可共享资源的内存表示，而并非该资源本身，所以，它与其他的执行体对象有所不同。文件对象仅仅包含了特定于一个对象句柄的数据，而文件本身包含了要被共享的数据或代码。每次当一个线程打开一个文件句柄的时候，一个新的文件对象被创建出来，它具有一组新的与该句柄相关的属性。例如，“当前字节偏移”属性指的是下一个用该句柄来执行的读操作或写操作将在该文件中的哪个位置上进行。每一个指向文件的句柄都有一个私有的字节偏移，即使底层的文件是共享的。除非一个进程复制了一个文件句柄，并把它给另一个进程（通过使用Windows的*DuplicateHandle*函数），或者一个子进程从父进程中继承了一个文件句柄，否则，文件对象只属于一个进程。在这两种例外情况下，两个进程都有单独的句柄，但它们指向同样的文件对象。

虽然一个文件句柄可能只属于一个进程，但底层的物理资源并非如此。因此，如同对于任何共享的资源一样，线程必须对可共享的文件、文件目录和设备的访问进行同步。例如，如果一个线程正在写一个文件，那么，它应该在打开该文件句柄的时候指定互斥的写访问，以防止另外的线程同时也执行写操作。或者，此线程可以使用Windows的*LockFile*函数，在写一个文件的时候锁住该文件的一部分。

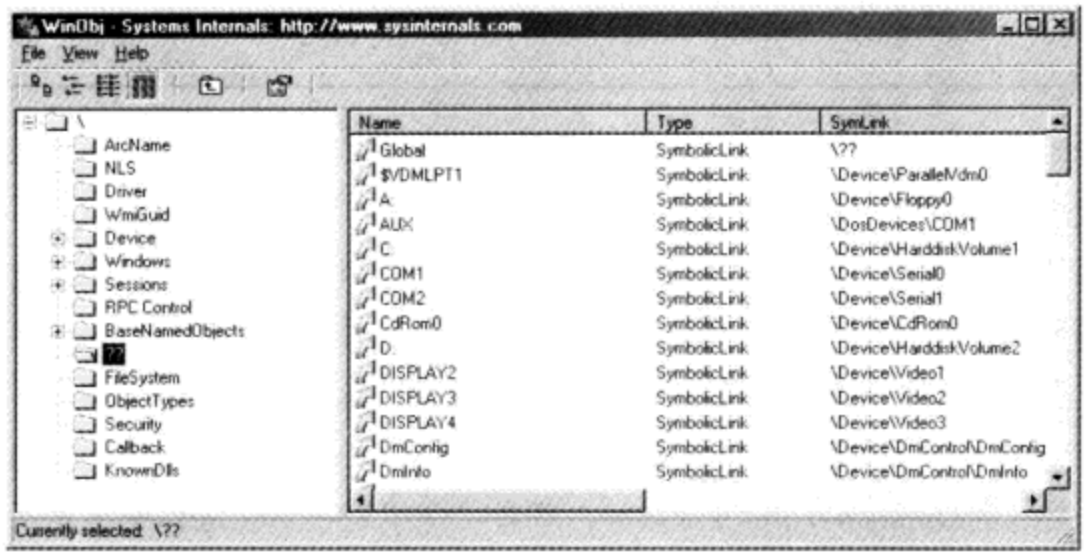
当一个文件被打开时，指定的文件名包含了该文件所在的设备对象的名称。例如，名称\Device\Floppy0\Myfile.dat指的是软盘驱动器A上的文件Myfile.dat。子串\Device\Floppy0是Windows内部代表软盘驱动器的设备对象的名称。当打开Myfile.da时，I/O管理器创建一个文件对象，并且在该文件对象中保存一个指向Floppy0设备对象的指针，然后向调用者返回一个文件句柄。因此，当调用者使用该文件句柄的时候，I/O管理器可以直接找到Floppy0设备对象。记住，在Windows应用程序中是不能使用Windows内部设备名称的。相反地，此内部设备名称必须出现在对象管理器名字空间的特定目录中（在Windows 2000上是\??，在Windows XP和Windows Server 2003上是\Global??）。此目录包含的是一些符号链接，它们指向了Windows内部实际的设备名称。设备驱动程序负责创建此目录中的链接，因而它们的设备能够被Windows应用程序访问。你可以利用Windows的*QueryDosDevice*或*DefineDosDevice*函数，用编程的方式检查这些链接，甚至改变它们。

www.TopSage.com

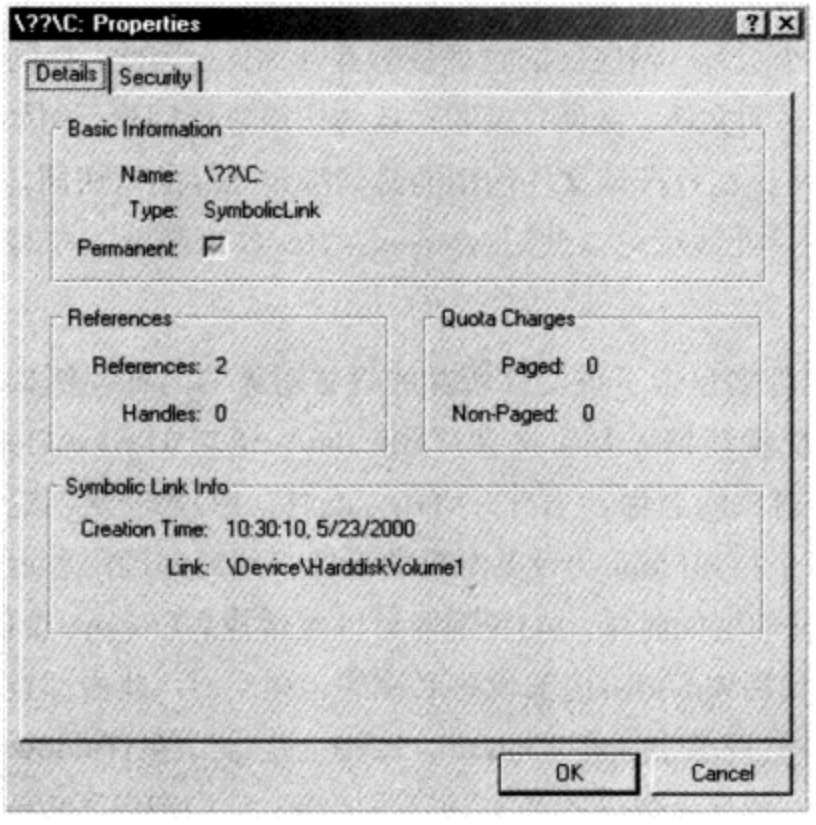


实验：查看Windows设备名称之间的映射

利用来自*www.sysinternals.com*的Winobj实用工具，你可以检查Windows设备名字空间的符号链接。运行Winobj，如果在Windows 2000上，则点击\??目录；如果在Windows XP或Windows Server 2003上，则点击\Global??。如下图所示：



注意到右侧的符号链接。试着在驱动器C:上双击。你应该看到类似如下的界面：



C:是一个指向内部设备名称\Device\HarddiskVolume1，或者，系统中第一个硬盘上第一个卷的符号链接。在Winobj中的COM1项是一个指向\Device\Serial0的符号链接，等等。你可以试着在命令提示符下利用*subst*命令来创建你自己的链接。

www.TopSage.com

9.3 I/O处理

我们已经讨论了驱动程序的结构和类型，以及相应的数据结构，现在，我们来看一看I/O请求是如何在系统中流动的。I/O请求在处理过程中要经过几个可预知的阶段。这些阶段不尽相同，这取决于该请求的目标是一个由单层驱动程序来进行操作的设备，还是一个要经过多层次的驱动程序才能到达的设备。进一步的处理过程也不尽相同，这取决于调用者指定的是同步I/O还是异步I/O，所以，我们从这两种I/O类型开始讨论，然后再转到有关其他内容的讨论。

I/O类型

应用程序在发出I/O请求时有几种选择。例如，它们可以指定同步的或异步的I/O，或者另外的I/O方式：将一个设备的数据映射到应用程序的地址空间中以便可以通过应用程序虚拟内存而不是通过I/O API来访问该设备的数据，或者通过单个请求在一个设备和非连续的应用程序缓冲区之间传送数据。而且，I/O管理器为驱动程序提供了一个实现快捷I/O接口的选择，快捷I/O接口通常可以减轻I/O处理过程中分配IRP的负担。在这一小节中，我们将讨论这些I/O类型。

同步I/O和异步I/O

应用程序发出的大多数I/O操作是**同步的**。也就是说，在设备执行数据传输过程中，应用程序一直等待，直到当I/O完成时，设备返回一个状态码。然后，该程序可以继续运行，并且立即可以访问这些被传输的数据。Windows的*ReadFile*和*WriteFile*函数的最简单使用形式就是按照同步方式来执行的。它们在完成了I/O操作以后才将控制权返回给调用者。

异步I/O使得一个应用程序可以先发出一个I/O请求，然后在设备传输数据的过程中继续执行。这种I/O类型可以提高一个应用程序的吞吐量，因为它使得该应用程序的线程在一个I/O操作进行过程中继续处理其他的工作。为了使用异步I/O，你必须在调用Windows的*CreateFile*函数时指定FILE_FLAG_OVERLAPPED标志。当然，在发出了一个异步I/O操作以后，该线程必须非常小心：在设备驱动程序完成数据传输以前，不能访问该I/O操作中的任何数据。该线程必须监视一个“可在该I/O操作完成时能被用信号通知到的同步对象”（可以是一个事件对象、I/O完成端口或者文件对象本身）的句柄，以便将它的执行过程与I/O请求的完成同步起来。

不管I/O请求是哪种类型，在内部，代表该应用程序而被发送给某个驱动程序的I/O操作都是按异步方式来执行的。也就是说，一旦一个I/O请求已经被发出了，设备驱动程序就立即返回到I/O系统。I/O系统是否立即返回到调用者，完全取决于该文件是以同步I/O方式还是以异步I/O方式来打开的。图9.8示出了当一个读操作被激发时的控制流程。注意，如果一个等待完成

www.TopSage.com

了（这取决于该文件对象上的重叠标志[overlapped flag]），则它是由*NtReadFile*在内核模式中完成的。

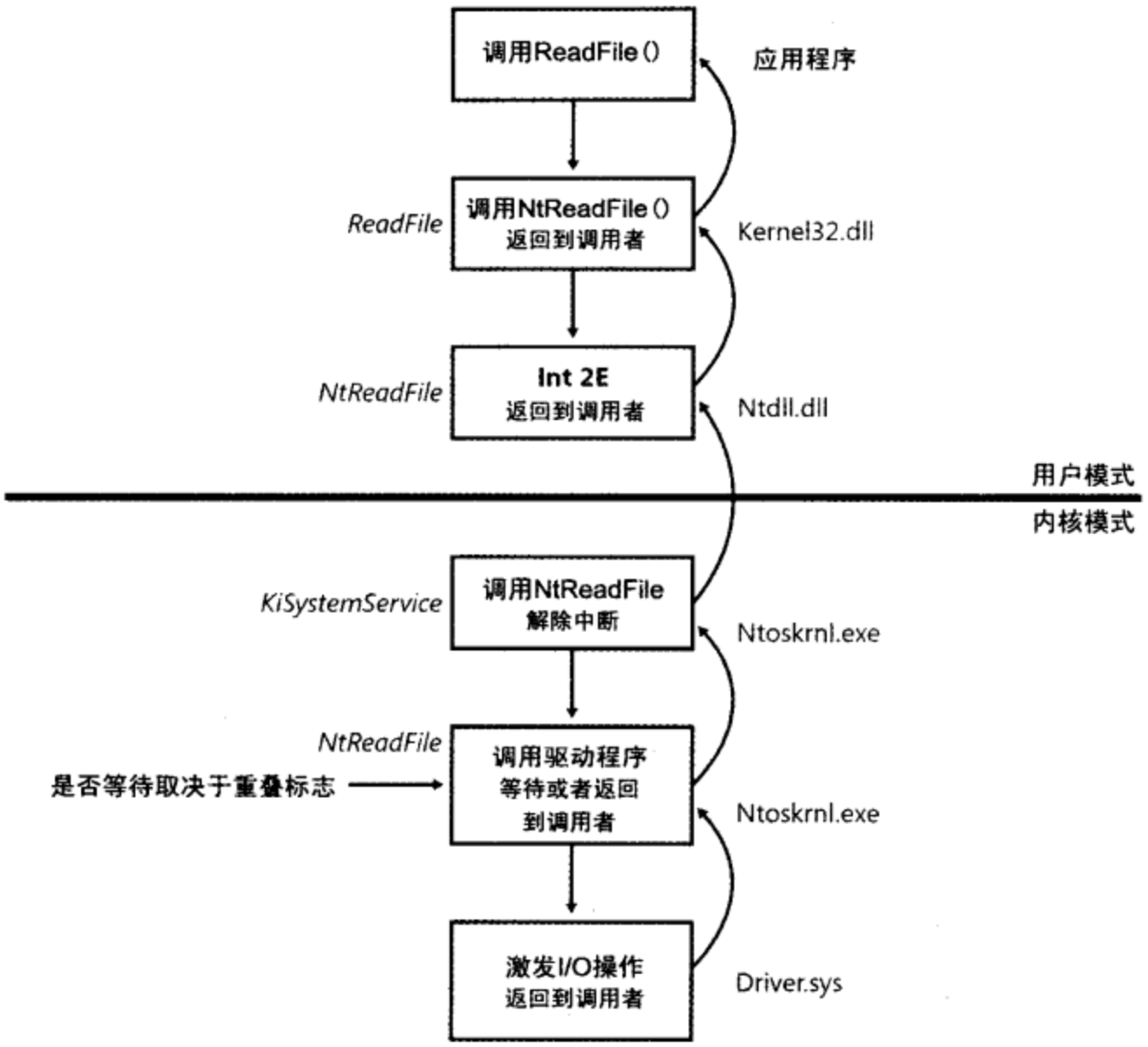


图9.8 一个I/O操作的控制流

你可以利用Windows的*HasOverlappedIoCompleted*函数来测试一个正在进行过程中的异步I/O的状态。如果你正在使用I/O完成端口（在本章后面的“I/O完成端口”一节中介绍），那么你可以使用*GetQueuedCompletionStatus*函数。

快速I/O

快速I/O是一种特殊的机制，它使得I/O系统可以避免生成一个IRP，而是直接到达文件系统驱动程序或者缓存管理器，以便完成一个I/O请求（在第11章和第12章中将会详细地介绍快速I/O）。一个驱动程序通过将它的快速I/O入口点放在其驱动程序对象的PFAST_IO_DISPATCH指针所指的结构中，以登记它的这些快速I/O入口点。

www.TopSage.com



实验：查看一个驱动程序登记的快速I/O例程

利用内核调试器命令*!drvobj*可以列出一个驱动程序登记在它的驱动程序对象中的快速I/O例程。然而，通常只有文件系统驱动程序才使用了快速I/O例程。下面的输出显示了NTFS文件系统驱动程序对象的快速I/O表：

```
kd> !drvobj \filesystem\ntfs 2
Driver object (8a4372a0) is for:
 \FileSystem\Ntfs
DriverEntry:   f7bd7398 Ntfs!DriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f7b76390        Ntfs!NtfsFsdCreate


Fast I/O routines:
FastIoCheckIfPossible                   f7b74a0b        Ntfs!NtfsFastIoCheckIfPossible
FastIoRead                              f7b77bbc        Ntfs!NtfsCopyReadA
FastIoWrite                             f7b8a9cc        Ntfs!NtfsCopyWriteA
FastIoQueryBasicInfo                    f7b7cd5e        Ntfs!NtfsFastQueryBasicInfo
FastIoQueryStandardInfo                 f7b7779e        Ntfs!NtfsFastQueryStdInfo
FastIoLock                              f7b8b738        Ntfs!NtfsFastLock
FastIoUnlockSingle                      f7b8b66c        Ntfs!NtfsFastUnlockSingle
FastIoUnlockAll                         f7ba5cd6        Ntfs!NtfsFastUnlockAll
FastIoUnlockAllByKey                    f7bcdab2        Ntfs!NtfsFastUnlockAllByKey
AcquireFileForNtCreateSection           f7b77771        Ntfs!NtfsAcquireForCreate
Section
ReleaseFileForNtCreateSection           f7b77758        Ntfs!NtfsReleaseForCreate
Section
FastIoQueryNetworkOpenInfo              f7bbec06        Ntfs!NtfsFastQueryNetworkOpen
Info
AcquireForModWrite                      f7b8663d        Ntfs!NtfsAcquireFileFor
ModWrite
MdlRead                                 f7bbed20        Ntfs!NtfsMdlReadA
```

上述输出显示出，NTFS已经登记了它的*NtfsFastIoCheckIfPossible*例程，并把它作为快速I/O表的*FastIoCheckIfPossible*项。正如该快速I/O项的名字所暗示的那样，I/O管理器有时候在发出一个快速I/O请求以前调用此函数，因而当一个文件上的快速I/O操作不可行的时候，驱动程序有机会来指明这一事实。

www.TopSage.com

映射文件I/O和文件缓存

映射文件I/O是I/O系统的一个重要特性，也是I/O系统和内存管理器联合产生的（关于如何实现映射文件的细节知识，请参见第7章）。**映射文件I/O**指的是“将磁盘上的文件看作进程的虚拟内存中的一部分”的能力。一个程序可以按照大数组的形式来访问一个文件，它无需缓冲数据或者执行磁盘I/O。该程序直接访问内存，内存管理器利用它的换页机制，将正确的页面从磁盘文件中加载进来。如果应用程序对它的虚拟地址空间进行写操作，则内存管理器通过正常换页的方式，将数据变化写回到该文件中。

在用户模式下，通过Windows的*CreateFileMapping*和*MapViewOfFile*函数，就可以使用映射文件I/O。在操作系统内部，映射文件I/O被用于一些重要的操作，比如文件缓存、映像激活（加载并运行可执行程序）。映射文件I/O的另一个主要的客户是缓存管理器。文件系统利用缓存管理器，将文件数据映射到虚拟内存中，以便为那些以I/O为限制（I/O-bound）的程序提供更好的响应时间。当调用者使用该文件的时候，内存管理器将被访问的页面带到内存中。虽然绝大多数缓存系统申请了固定数量的字节，并用其把文件缓存在内存中，但是，Windows的缓存会根据当前可用内存的数量而增加或缩减。这种大小变化是有可能的，因为缓存管理器依赖于内存管理器，利用第7章中介绍的正常的工作集机制，来自动扩张（或缩减）缓存的大小。缓存管理器充分利用了内存管理器的换页系统，从而避免了重复实现内存管理器已经完成的工作（第11章中详细地解释了缓存管理器的工作）。

分散/聚集I/O

Windows也支持一种特殊的高性能I/O，它称为**分散/聚集**（*scatter/gather*）。通过Windows的 *ReadFileScatter*和*WriteFileGather*函数，就可以使用这种I/O。这些函数使得应用程序只要发出单个读或写请求，就可以在虚拟内存中的多个缓冲区与磁盘文件的一个连续区域之间传输数据，而无需为每个缓冲区发出单独的I/O请求。为了使用分散/聚集I/O，该文件必须以非缓存I/O的方式来打开，所使用的用户缓冲区必须是页面对齐的，并且用户发出的I/O必须是异步的（可重叠的）。而且，如果该I/O是针对大容量存储设备的，那么，该I/O必须对齐到设备扇区边界，并且其长度值是扇区大小的整数倍。

I/O请求包

I/O请求包（IRP，I/O request packet）是I/O系统为了处理一个I/O请求而用来存储必要信息的地方。当一个线程调用一个I/O服务的时候，I/O管理器构造一个IRP，用于在I/O系统处理此请求的过程中代表此请求。如果有可能的话，I/O管理器从两个非换页的IRP预读列表之一中分配IRP。针对每个处理器都有两个预读列表，它们是：小-IRP预读列表保存的是只有一个栈单元（stack location，稍后介绍栈单元）的IRP；大-IRP预读列表保存的是有多个栈单元的IRP。

www.TopSage.com

在默认情况下，系统在大-IRP预读列表中保存具有8个栈单元的IRP，但是，系统每分钟一次，根据已被请求过的栈单元的数量，来调整所分配的栈单元的数量。如果一个IRP请求的栈单元的数量超过了大-IRP预读列表上的IRP中包含的栈单元，那么，I/O管理器从非换页内存池中申请IRP。在分配并初始化一个IRP以后，I/O管理器将一个指向调用者的文件对象的指针保存在IRP中。



> **注** DWORD注册表值HKLM\System\CurrentControlSet\Session Manager\I/O System\LargeIrpStackLocations如果已经定义的话，则它指定了在大-IRP预读列表上的IRP中包含的栈单元的数量。

图9.9显示了一个示例性的I/O请求，它示出了IRP与上一节中讲述的文件、设备和驱动程序对象之间的关系。虽然这个例子显示的是一个针对单层设备驱动程序的I/O请求，但绝大多数I/O操作并不是这么直接的，它们要涉及一个或者多个分层的驱动程序（本节后面将会显示这种情形）。

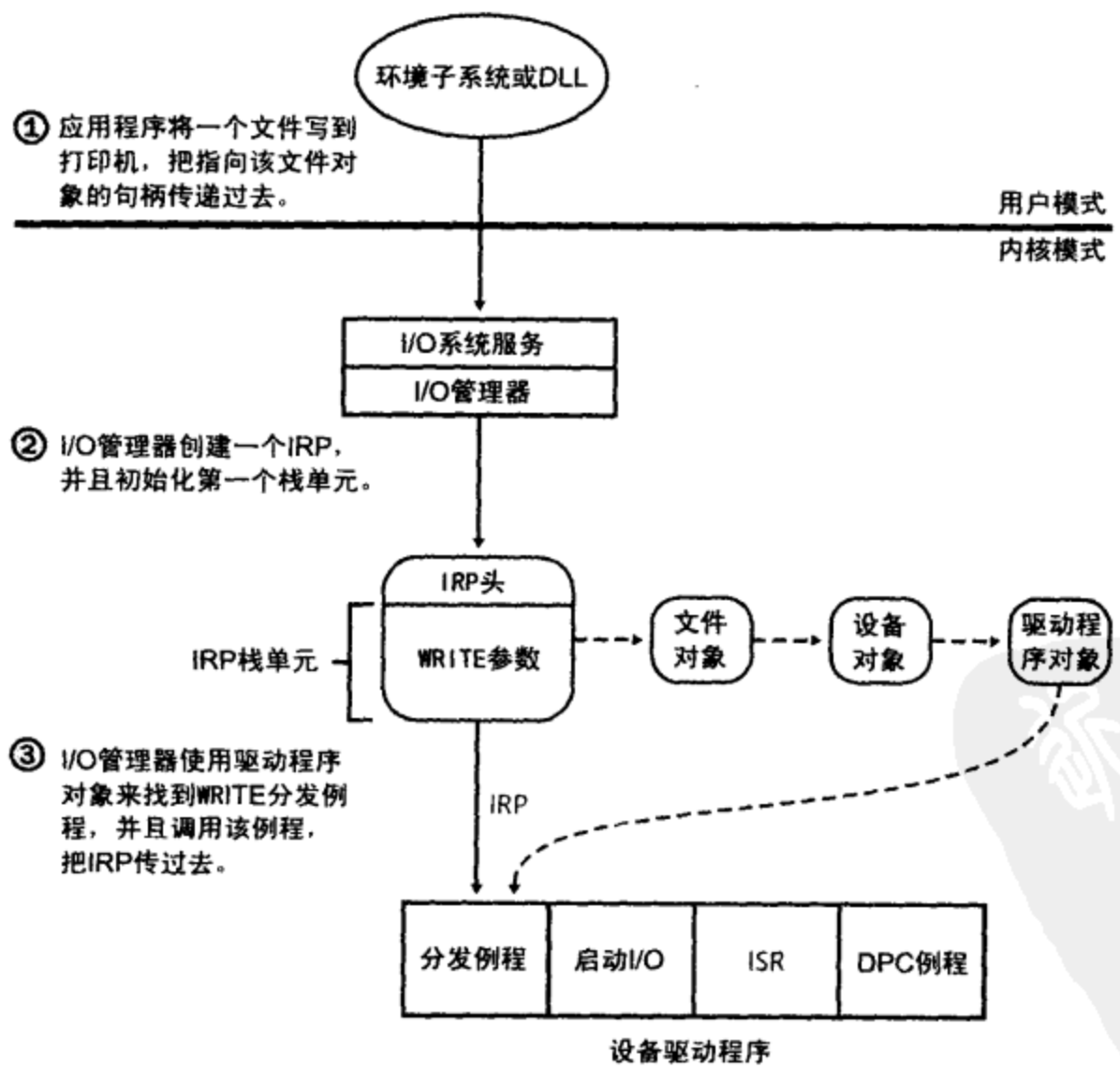


图9.9 在一个单层驱动程序的I/O请求中涉及的数据结构

www.TopSage.com

IRP栈单元

IRP是由两部分组成的：一个固定的头（通常被称为IRP的*体*[*body*]）以及一个或者多个栈单元。固定部分包含一些诸如以下的信息：该请求的类型和大小，该请求是同步的还是异步的，一个用于缓冲类型I/O的缓冲区指针，以及一些在该请求处理过程中会被改变的状态信息。IRP栈单元包含一个功能代码（由一个主代码和一个次代码组成）、与功能相关的参数，以及一个指向调用者文件对象的指针。**主功能代码**（*major function code*）标识了当I/O管理器将一个IRP传递给一个驱动程序时，它应该调用该驱动程序的哪个分发例程。**次功能代码**（*minor function code*）是可选的，它有时候被用于主功能代码的一个修饰符。电源命令和即插即用命令总是有次功能代码。

驱动程序为它将要处理的主功能代码指定相应的分发例程。然而，绝大多数驱动程序只处理所有可能的主功能代码中的一个子集，包括创建（打开）、读、写、设备I/O控制、电源、即插即用、系统（针对WMI命令），以及关闭。（关于完整的主功能代码列表，请参考下面的实验）。文件系统驱动程序是一个几乎填充了绝大多数或者全部分发入口点的驱动程序例子。对于驱动程序没有填充的分发入口点，I/O管理器将它们设置成指向它自己的*IopInvalidDeviceRequest*，该函数向调用者返回一个错误码，指明了该驱动程序中这一特定的功能是无效的。



实验：查看驱动程序的分发例程

在内核调试器命令*!drvobj*中，你只要在驱动程序对象的名称（或地址）后面输入一个7，就可以获得该驱动程序已经定义的分发例程的函数列表。下面的输出显示了驱动程序支持28种IRP类型。

```
kd> !drvobj kbdclass 7
Driver object (8a238900) is for:
 \Driver\Kbdclass
Driver Extension List: (id , addr)

Device Object list:
8a189030  8a2501f8

DriverEntry:   f7822d22 kbdclass!DriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f781fd3b        kbdclass!KeyboardClassCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           804eef8e        nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       f781ff4c        kbdclass!KeyboardClassClose
[03] IRP_MJ_READ                        f7820ba5        kbdclass!KeyboardClassRead
[04] IRP_MJ_WRITE                       804eef8e        nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           804eef8e        nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             804eef8e        nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    804eef8e        nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      804eef8e        nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               f781fcbe        kbdclass!KeyboardClassFlush
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    804eef8e        nt!IopInvalidDeviceRequest
```


www.TopSage.com

```
[0b] IRP_MJ_SET_VOLUME_INFORMATION      804eef8e    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           804eef8e    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         804eef8e    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              f7821829    kbdclass!KeyboardClassDevice
                                                    Control
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     f7821200    kbdclass!KeyboardClassPass
                                                    Through
[10] IRP_MJ_SHUTDOWN                    804eef8e    nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                804eef8e    nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     f781fc84    kbdclass!KeyboardClassCleanup
[13] IRP_MJ_CREATE_MAILSLOT             804eef8e    nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              804eef8e    nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                804eef8e    nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       f7821f51    kbdclass!KeyboardClassPower
[17] IRP_MJ_SYSTEM_CONTROL              f7821649    kbdclass!KeyboardClassSystem
                                                    Control
[18] IRP_MJ_DEVICE_CHANGE               804eef8e    nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 804eef8e    nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   804eef8e    nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         f78206c1    kbdclass!KeyboardPnP
```

当一个IRP有效时，它通常被存储在与发出此I/O请求的线程相关联的IRP列表中。这种安排使得当一个线程主动终止或者被动终止时，如果尚有I/O请求未完成的话，则I/O系统可以找到并取消任何未完成的IRP。



实验：查看一个线程的未完成IRP

当你使用*!thread*命令时，它打印出任何与该线程相关联的IRP。运行内核调试器，启动实时调试模式，在*!process*命令生成的输出中，找到服务控制管理器进程（Services.exe）的信息：

```
lkd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
…
PROCESS 8a238da8  SessionId: 0  Cid: 02a8    Peb: 7ffdf000  ParentCid: 027c
    DirBase: 14fac000  ObjectTable: e1c3e008  HandleCount: 365.
    Image: SERVICES.EXE
…
```

然后，针对该进程对象执行*!process*命令，转储出该进程的线程。你应该看到很多线程，而且对于大多数线程，在线程信息的IRP列表区域中报告了一些IRP（注意，如果一个线程的未完成I/O请求超过了17个，那么调试器只显示前17个IRP）：

```
kd> !process 8a238da8
PROCESS 8a238da8  SessionId: 0  Cid: 02a8    Peb: 7ffdf000  ParentCid: 027c
    DirBase: 14fac000  ObjectTable: e1c3e008  HandleCount: 365.
    Image: SERVICES.EXE
    VadRoot 8a1be328 Vads 88 Clone 0 Private 346. Modified 37. Locked 0.
    DeviceMap e10087c0
…
```


www.TopSage.com

```
     THREAD 8a124870  Cid 02a8.0338  Teb: 7ffd8000 Win32Thread: 00000000 WAIT:
(WrQueue) UserMode Non-Alertable
            8a2dc620  Unknown
            8a124960  NotificationTimer
        IRP List:             8a2c2c00: (0006,0094) Flags: 00000900  Mdl: 00000000
    8a20f770: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a437780: (0006,0094) Flags: 00000900  Mdl: 00000000
            89b1de68: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a0e6058: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a0f1550: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a3b3c18: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a429190: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a49f008: (0006,0094) Flags: 00000900  Mdl: 00000000
            8a227bc0: (0006,0094) Flags: 00000900  Mdl: 00000000
...
```

选择一个IRP，并且用*!irp*命令来检查此IRP：

```
lkd> !irp 8a2c2c00
Irp is active with 1 stacks 1 is current (= 0x8a2c2c70)
 No Mdl Thread 8a124870:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[  3, 0]   0  1 8a0e5680 8a26e4b8 00000000-00000000    pending
            \Driver\Npfs
             Args: 00000400 00000000 00000000 00000000
```

此IRP的主功能代码为3，对应于IRP_MJ_READ。它有一个栈单元，其目标是一个被Npfs（命名管道文件系统，Named Pipe File System）驱动程序所拥有的设备。（第13章讲述了Npfs）。

IRP缓冲区管理

当一个应用程序或者设备驱动程序间接地利用*NtReadFile*、*NtWriteFile*或*NtDeviceIoControlFile*系统服务（或者对应于这些服务的Windows API函数，即*ReadFile*、*WriteFile*和*DeviceIoControl*）来创建一个IRP时，I/O管理器确定它是否需要参与到调用者的输入或输出缓冲区的管理中。I/O管理器执行以下三种类型的缓冲区管理。

- **缓冲的I/O** I/O管理器从非换页内存池中申请一个缓冲区，其大小等同于调用者的缓冲区大小。对于写操作，I/O管理器在创建IRP时，将调用者缓冲区中的数据拷贝到申请得到的缓冲区中。对于读操作，I/O管理器在该IRP完成时，将数据从申请的缓冲区中拷贝到用户的缓冲区中，然后释放这一申请的缓冲区。
- **直接I/O** 当I/O管理器创建此IRP时，它将用户的缓冲区锁在内存中（使它变成不可换页的）。当I/O管理器用完了此IRP时，它解除该缓冲区的锁定。I/O管理器以MDL（*memory descriptor list*，内存描述符列表）的形式保存了此块内存的一个描述。MDL指定了一个缓冲区所占用的物理内存（有关MDL的更多信息，请参见Windows DDK）。执行直接内存访问（DMA）的设备只需要缓冲区的物理描述，所以，对于这些设备的操作而言，MDL已经足够了（支持DMA的设备直接在设备和计算机的内存之间传输数据，而不需要使用CPU）。 然而，如

www.TopSage.com

果一个驱动程序必须要访问一个缓冲区的内容，那么，它可以将该缓冲区映射到系统的地址空间中。

- **两者皆非I/O** I/O管理器并不执行任何缓冲区管理。相反，缓冲区管理的任务被留给设备驱动程序自行处理；对于在其他缓冲区管理类型中I/O管理器所执行的各种步骤，设备驱动程序可以选择手工来完成。

对于每一种缓冲区管理类型，I/O管理器把输入和输出缓冲区的位置的可用引用放在IRP中。I/O管理器到底执行哪一种类型的缓冲区管理，要取决于驱动程序为每一种操作类型所请求的缓冲区管理类型。驱动程序针对读和写操作，在代表目标设备的设备对象中登记了它所期望的缓冲区管理类型。“设备I/O控制”操作（即由*NtDeviceIoControlFile*执行的操作）是通过由驱动程序定义的I/O控制代码来指定的，每个控制代码包含了关于缓冲区管理的描述：当发出包含此代码的IRP时，I/O管理器将会使用此种类型的缓冲区管理。

当调用者的传输请求小于一个页面（在x86处理器上为4KB）时，驱动程序通常使用缓冲的I/O；但对于大一点的请求，则使用直接I/O。一个页面的大小，差不多正好是缓冲I/O的拷贝操作与直接I/O执行内存锁定的开销两者达到平衡点的缓冲区大小。文件系统驱动程序通常使用“两者皆非I/O”，因为数据被从文件系统的缓存拷贝到调用者的原始缓冲区中，并没有招致任何缓冲区管理开销。绝大多数驱动程序之所以不使用“两者皆非I/O”，是因为，只有当调用者进程的线程正在执行时，指向调用者缓冲区的指针才是有效的。如果一个驱动程序必须在一个ISR或者DPC例程中将数据传输给一个设备或者从设备接收数据，那么，它必须确保，调用者的数据在任何一个进程环境中都是可以访问的。这意味着该缓冲区必须有一个系统虚拟地址。

如果驱动程序使用“两者皆非I/O”来访问可能位于用户空间中的缓冲区，则它们必须特别小心，以确保缓冲区地址既是有效的，也没有引用内核模式内存。若未能保证这一点，则可能会导致系统崩溃，或者安全漏洞。因为在这种情况下应用程序可以访问到内核模式的内存，或者在内核中注入代码。内核提供给驱动程序使用的*ProbeForRead*和*ProbeForWrite*函数，能够验证一个缓冲区是否完全驻留在地址空间的用户模式部分。为了避免因为引用了一个无效用户模式地址而崩溃，驱动程序可以在异常处理代码内部（称为try/except块）访问用户模式缓冲区，如果异常处理代码块捕捉到任何无效的内存错误，则将它们转译成错误代码，然后将错误代码返回给应用程序。

针对单层驱动程序的I/O请求

这一小节跟踪一个针对单层内核模式设备驱动程序的同步I/O请求的过程。处理一个针对单层驱动程序的同步I/O要涉及以下7个步骤。

1. 通过一个子系统DLL传递I/O请求。

www.TopSage.com

2. 子系统DLL调用I/O管理器的*NtWriteFile*服务。
3. I/O管理器申请一个描述此I/O请求的IRP，并且通过调用它自己的*IoCallDriver*函数将此IRP发送给驱动程序（在这个例子中，是一个设备驱动程序）。
4. 该驱动程序将IRP中的数据传输给设备并启动I/O操作。
5. 驱动程序通过中断CPU的方式来通知I/O完成。
6. 当设备完成了操作并且中断CPU时，设备驱动程序为该中断提供服务。
7. 驱动程序调用I/O管理器的*IoCompleteRequest*函数来通知I/O管理器，它已经完成了该IRP中的I/O请求的处理工作，然后I/O管理器完成此I/O请求。

这7个步骤如图9.10中所示。

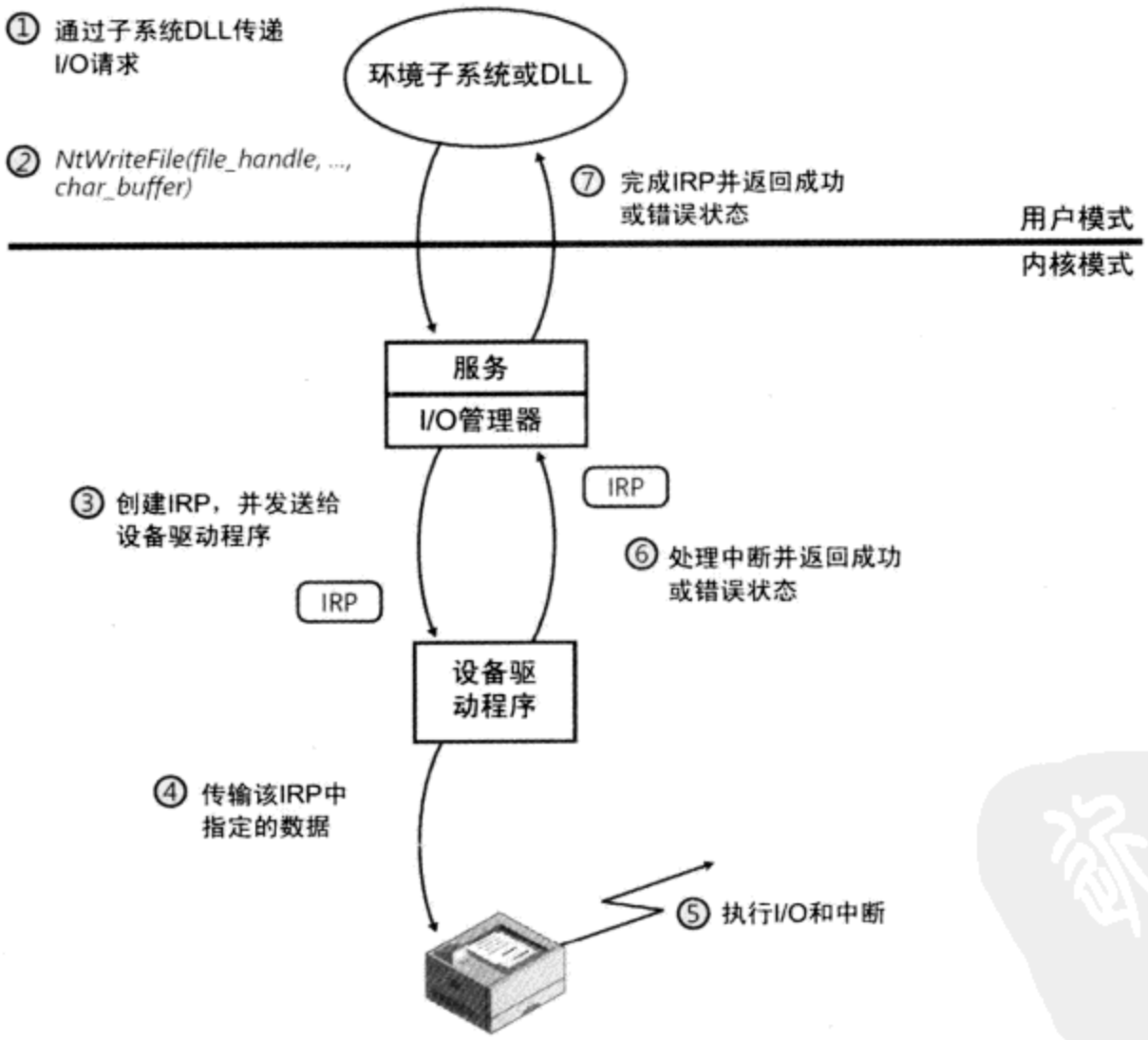


图9.10 排队并完成一个同步I/O请求的过程

我们已经看过了一个I/O是如何被激发的，现在我们更加仔细地看一看中断处理和I/O完成。

www.TopSage.com

为一个中断提供服务

当一个I/O设备完成一次数据传输时，它激发一个中断以要求得到服务，于是，Windows内核、I/O管理器和设备驱动程序便采取相应的动作。图9.11示出了此处理过程的第一个阶段（第3章讲述了中断分发机制，包括DPC。这里我们简单作一个回顾，因为DPC对于I/O处理至关重要）。

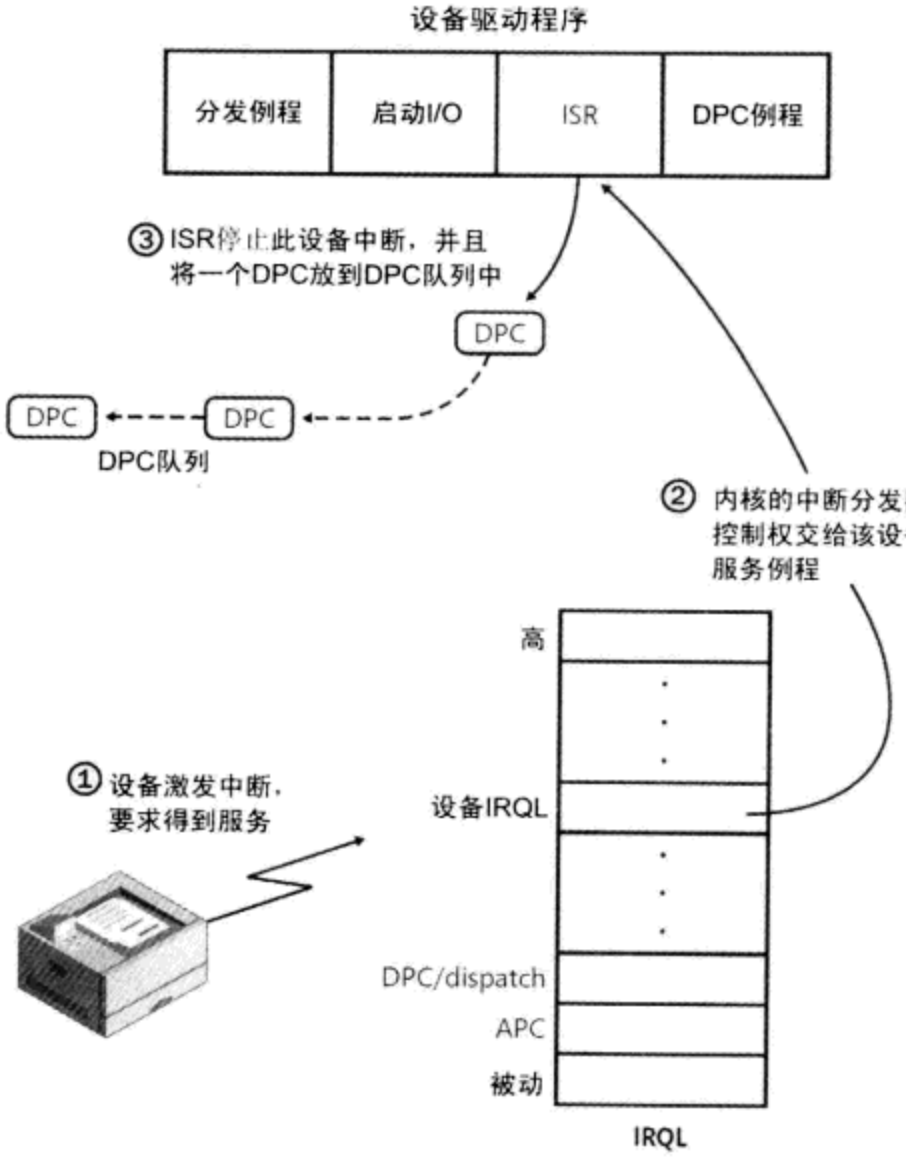


图9.11 为一个设备中断提供服务（阶段1）

当一个设备中断发生的时候，处理器将控制权交给内核的陷阱处理器（trap handler），陷阱处理器根据中断号，在其中断分发表中找到该设备的ISR。Windows中的ISR通常分两个步骤来处理设备中断。当一个ISR首先被调用时，它停留在设备IRQL上的时间通常只够捕获住该设备的状态，然后停止该设备的中断。然后，它把一个DPC放入到队列中，并且退出，从而解除此中断。以后，当该DPC例程被调用时，该设备再完成此中断的处理。在这些都做完以后，设备调用I/O管理器以便完成此I/O，并清除IRP。它也有可能启动下一个正在该设备队列中等待的I/O请求。

www.TopSage.com

使用一个DPC来执行大部分设备服务功能的好处是，优先级处于设备IRQL和DPC/Dispatch IRQL之间的任何阻塞类型的中断都有机会在低优先级DPC被处理之前首先被处理。因此，中间级别的中断更能够获得及时的服务。图9.12示出了一个I/O的第二个阶段（DPC处理过程）。

完成一个I/O请求

在一个设备驱动程序的DPC例程被执行完以后，还有些工作需要做完才能认为此I/O请求已经完成了。这是I/O处理的第三阶段，被称为**I/O完成**（*I/O completion*）。当一个驱动程序调用*IoCompleteRequest*，以通知I/O管理器它已经完成了该IRP（和其中的栈单元）中指定的请求的处理工作时，这第三阶段就开始了。在I/O完成阶段中所涉及的步骤随着不同的I/O操作而不尽相同。例如，所有的I/O服务都将该操作的结果记录在一个**I/O状态块**（*I/O status block*）中，这是一个由调用者提供的数据结构。类似地，有些服务执行的是缓冲的I/O，它们可能要求I/O系统将数据返回给调用线程。

在这两种情形下，I/O系统必须将某些存储在系统内存中的数据拷贝到调用者的虚拟地址空间中。如果此IRP是以同步方式来完成的，则调用者的地址空间是当前可直接访问的。但如果此IRP是以异步方式来完成的，则I/O管理器必须将此IRP的完成延迟到它可以访问调用者的地址空间为止。为了获得对调用者虚拟地址空间的访问，I/O管理器必须“在调用者的线程环境中”传输数据。也就是说，当调用者的线程正在执行的时候，这意味着，调用者的进程是当前进程，它的地址空间在该处理器上是激活的。它的做法是，把一个内核模式的异步过程调用（APC，asynchronous procedure call）排队到该线程中。这一过程如图9.13所示。

正如在第3章中所解释的那样，APC是在某个特定的线程环境中执行的，而DPC可以在任意线程环境中执行。这也意味着，DPC例程不能碰用户模式的进程地址空间。也要记住，DPC比APC有更高的软件中断优先级。

www.TopSage.com

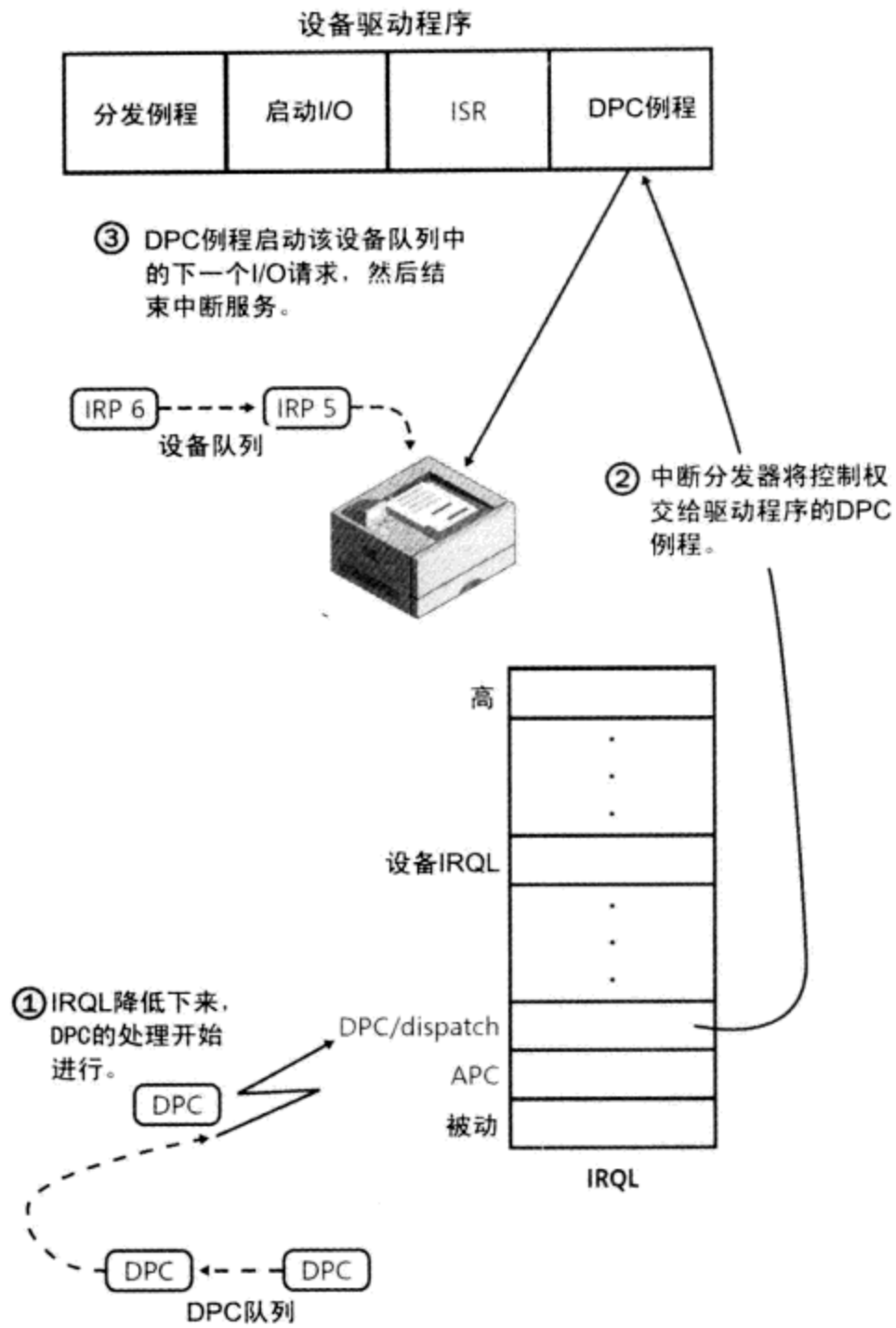


图9.12 为一个设备中断提供服务（阶段2）

www.TopSage.com

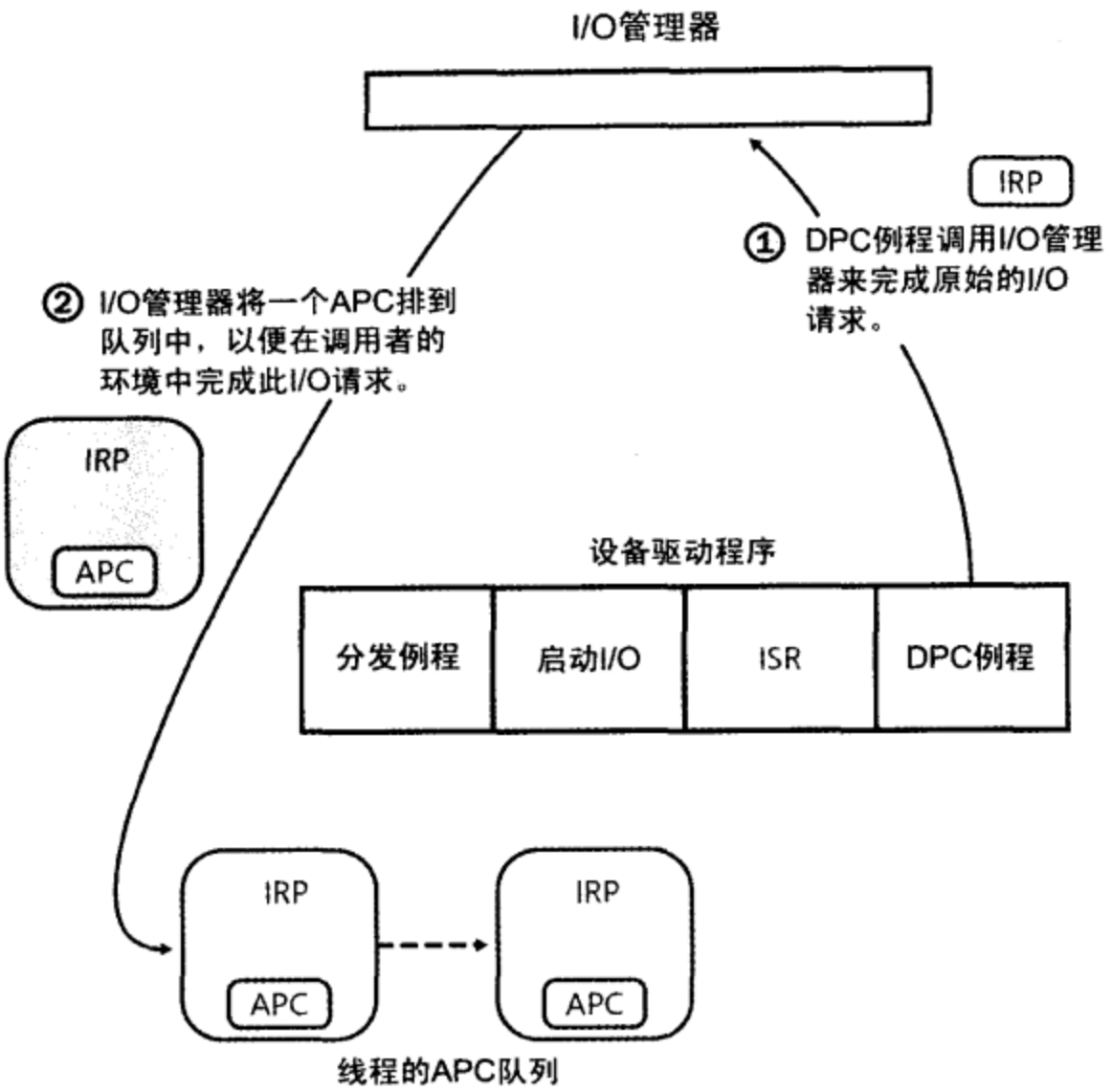


图9.13 完成一个I/O请求（阶段1）

下一次当该线程开始在低的IRQL上执行时，这一待处理的APC得以交付执行。内核将控制权交给I/O管理器的APC例程，该例程把数据（如果有的话）和返回状态拷贝到原来调用者的地址空间中，释放该I/O操作所对应的IRP，并且将调用者的文件句柄（以及任何由调用者提供的事件或I/O完成端口）设置成有信号的状态。现在，该I/O可以被认为已经完成了。原来的调用者或者其他任何正在该文件（或其他对象）句柄上等待的线程都被解除等待状态，并且准备执行。图9.14示出了I/O完成过程的第二个阶段。

www.TopSage.com

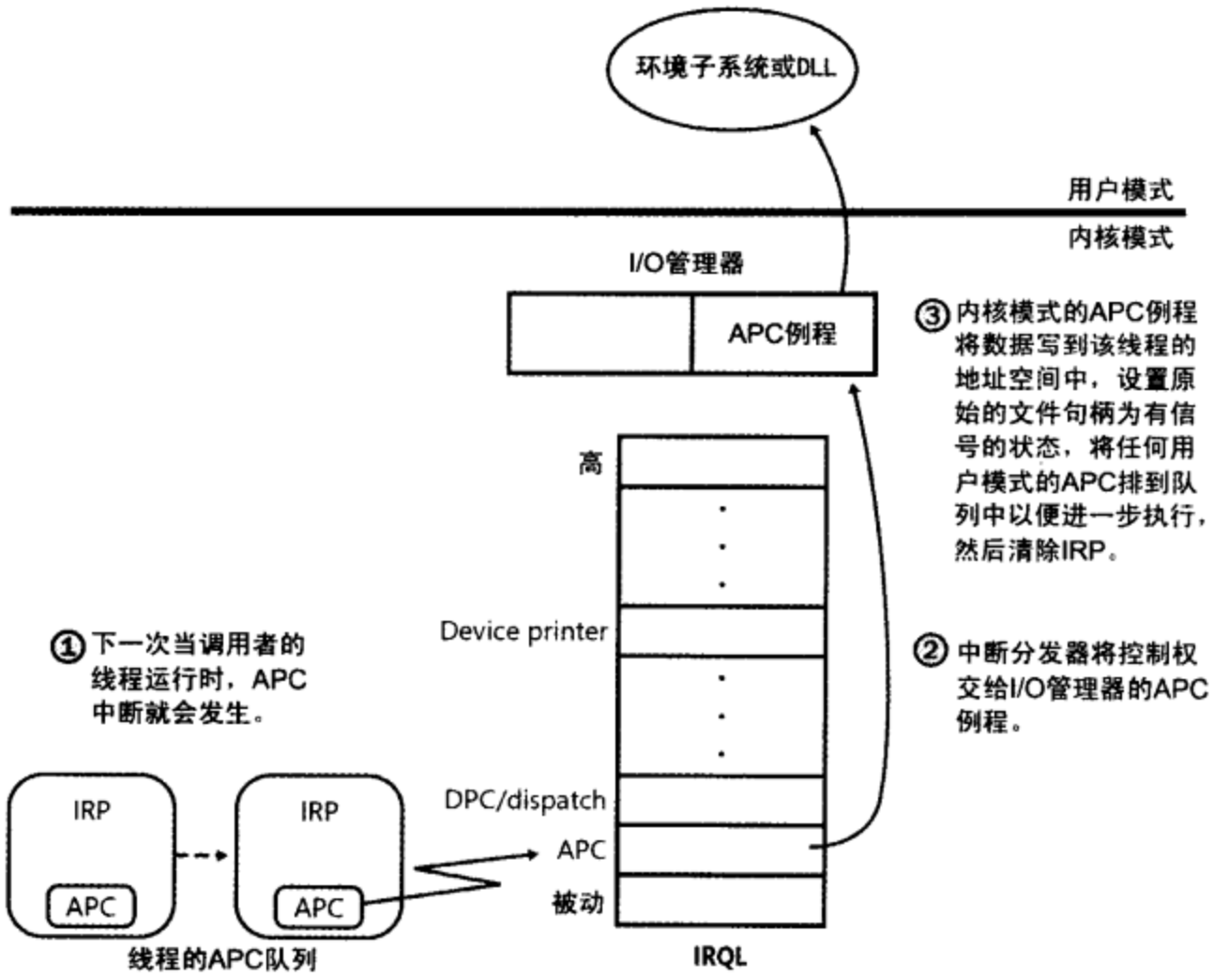


图9.14 完成一个I/O请求（阶段2）

关于I/O完成，最后一个要注意的地方是：异步I/O函数*ReadFileEx*和*WriteFileEx*允许调用者提供一个用户模式的APC作为一个参数。如果调用者确实提供了用户模式的APC，那么，I/O管理器将这个APC排到该调用者的线程APC队列中，作为I/O完成的最后一个步骤。这一特性使得调用者可以指定一个子例程，以便当一个I/O请求完成或者取消的时候，该子例程能被调用。用户模式的APC完成例程是在发出请求的线程的环境中执行的，而且，只有当该线程进入到一种可警告的等待状态（比如调用Windows的函数*SleepEx*、*WaitForSingleObjectEx*或*WaitForMultipleObjectsEx*）时，它们才会被交付。

同步

驱动程序对于全局的驱动程序数据和硬件寄存器的访问必须进行同步，理由有以下两点：

- 一个驱动程序的执行有可能会被高优先级的线程抢占，或者时间片（或时限）到期的时候也可能被强占，或者被中断打断；
- 在多处理器系统上，Windows可以并发地在多个处理器上运行驱动程序代码。

www.TopSage.com

如果没有同步，则有可能发生破坏。例如，当一个调用者发起一个I/O请求时，设备驱动程序的代码运行在被动IRQL级别上，它可能被一个设备中断打断，因此，这有可能造成当一个设备驱动程序正在运行时，它的ISR也得到执行的状况。如果该设备驱动程序正在修改的数据，也是其ISR要修改的数据，比如设备寄存器、堆存储区或者静态数据，那么，当ISR执行时，此数据可能被破坏。图9.15示出了这一问题。

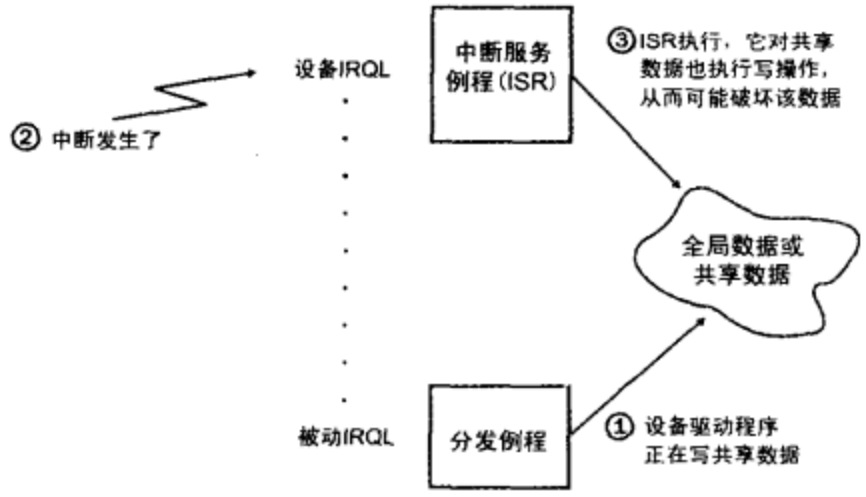


图9.15 将一个异步请求排入到分层的驱动程序中驱动程序的分发例程和ISR都要访问共享数据的情形

为了避免这种情形，Windows的设备驱动程序必须对所有它与其ISR共享的数据的访问进行同步。在试图更新这些共享数据以前，设备驱动程序必须将所有其他的线程（或CPU，如果是多处理器系统的话）锁在外面，以防止它们也在更新同样的数据结构。

Windows内核提供了一个特殊的同步例程，称为*KeSynchronizeExecution*，当设备驱动程序访问那些其ISR也要访问的数据时，它们可以调用该函数。这一内核同步例程使得当共享的数据被访问时，其ISR不会被执行。在单CPU系统上，在更新一个结构以前，该例程将IRQL提升至与该ISR相关联的级别。然而，在一个多处理器系统上，因为一个驱动程序可以同时在两个或多个处理器上执行，因此，这一技术不足以阻止其他的访问者。于是，内核使用另一种机制，即一个**自旋锁**（*spinlock*），来锁住一个结构，以确保此特定的CPU可以独自访问该结构。（在第3章的“同步”一节中解释了自旋锁）。驱动程序也可以使用*KeAcquireInterruptSpinLock*来直接地访问一个中断对象的自旋锁，不过，一般来说，依赖于*KeSynchronizeExecution*与一个ISR进行同步会更加快速一些。

到现在为止，你应该明白，除了ISR要求特别关注以外，一个设备驱动程序所使用的任何数据都有可能被运行在其他处理器上的同一驱动程序所访问。因此，设备驱动程序代码在使用任何全局或共享数据（或者访问物理设备本身）时，务必要进行同步，这是非常关键的。如果ISR也使用该数据的话，则设备驱动程序必须使用*KeSynchronizeExecution*；要不然，设备驱动程序也可以使用标准的内核自旋锁。

www.TopSage.com

针对分层的驱动程序的I/O请求

上一小节介绍了由单个设备驱动程序所控制的简单设备的I/O请求是如何被处理的。对于以文件为基础的设备的I/O处理过程，或者对于其他分层的驱动程序的I/O请求的处理，也都以类似的方式进行。很显然，最大的不同之处是，在整个模型中加入了一个或者多个额外层次的处理。

图9.16显示了一个异步I/O请求是如何经过多个分层的驱动程序的。它使用一个由文件系统来控制的磁盘作为例子。

再说明一次，I/O管理器接收该请求，并创建一个I/O请求包来表达这一请求。然而，这一次，它把I/O请求包传递给一个文件系统驱动程序。在这个点上，文件系统驱动程序对于此I/O操作有强大的控制力。根据调用者所发出的请求的类型情况，文件系统可能将同样的IRP发送给磁盘驱动程序，也可能产生额外的一个或者多个IRP，并且将它们单独发送给磁盘驱动程序。



实验：查看一个设备栈

内核调试器命令*!devstack*显示了与某个特定设备对象相关联的分层设备对象的设备栈。下面的例子显示了与一个设备对象\device\keyboardclass0相关联的设备栈，此设备对象归属于键盘类驱动程序：

```
lkd> !devstack keyboardclass0
  !DevObj   !DrvObj            !DevExt   ObjectName
  8a266d28  \Driver\Ctrl2cap   8a266de0
> 8a09a030  \Driver\Kbdclass   8a09a0e8  KeyboardClass0
  8a2672b0  \Driver\nmfilter   8a267368  0000008c
  8a09ba78  \Driver\i8042prt   8a09bb30
  8a4adce0  \Driver\ACPI       8a4ab9c8  0000006b
!DevNode 8a4acee8 :
  DeviceInst is "ACPI\PNP0303\4&61f3b4b&0"
  ServiceName is "i8042prt"
```

在上面的输出中，“>”前缀突出显示了与KeyboardClass0相关联的那一项。在这一行之上的那些项都是位于键盘类驱动程序之上的驱动程序，这一行之下的那些项在驱动程序层次中也位于它的下面。一般而言，IRP从设备栈的最上边流动到最底下。

如果文件系统接收到的请求被转译成一个直接对于某个设备的请求，那么，文件系统最有可能重用该IRP。例如，如果一个应用程序发出一个请求，要读取软盘上一个文件中的前512字节，那么，FAT文件系统只是简单地调用磁盘驱动程序，请它从软盘上该文件的起始位置开始，读取一个扇区。

为了在一个对于分层驱动程序的请求中也能够让多个驱动程序做到这种IRP重用，IRP包含了一系列**IRP栈单元**（*IRP stack location*，请不要与“线程用于存放函数参数和返回地址的栈”

www.TopSage.com

相混淆）。这是一些数据区域，每个将要被调用的驱动程序占用一个区域，它们包含了每个驱动程序用于执行它这一部分工作所需要的信息，比如功能代码、参数和驱动程序环境信息。正如图9.16所示，当IRP被从一个驱动程序传递到下一个驱动程序时，又有额外的栈单元被填充上数据。你可以把IRP想象成像一个栈一样来工作，在它的生命期中数据被加进去，又被移除出来。然而，IRP并不与任何特定的进程关联起来，而且它得到的内存大小也不会增长或缩减。I/O管理器在I/O操作开始的时候，从它的某一个IRP预读列表中，或者非换页的系统内存中申请一个IRP。

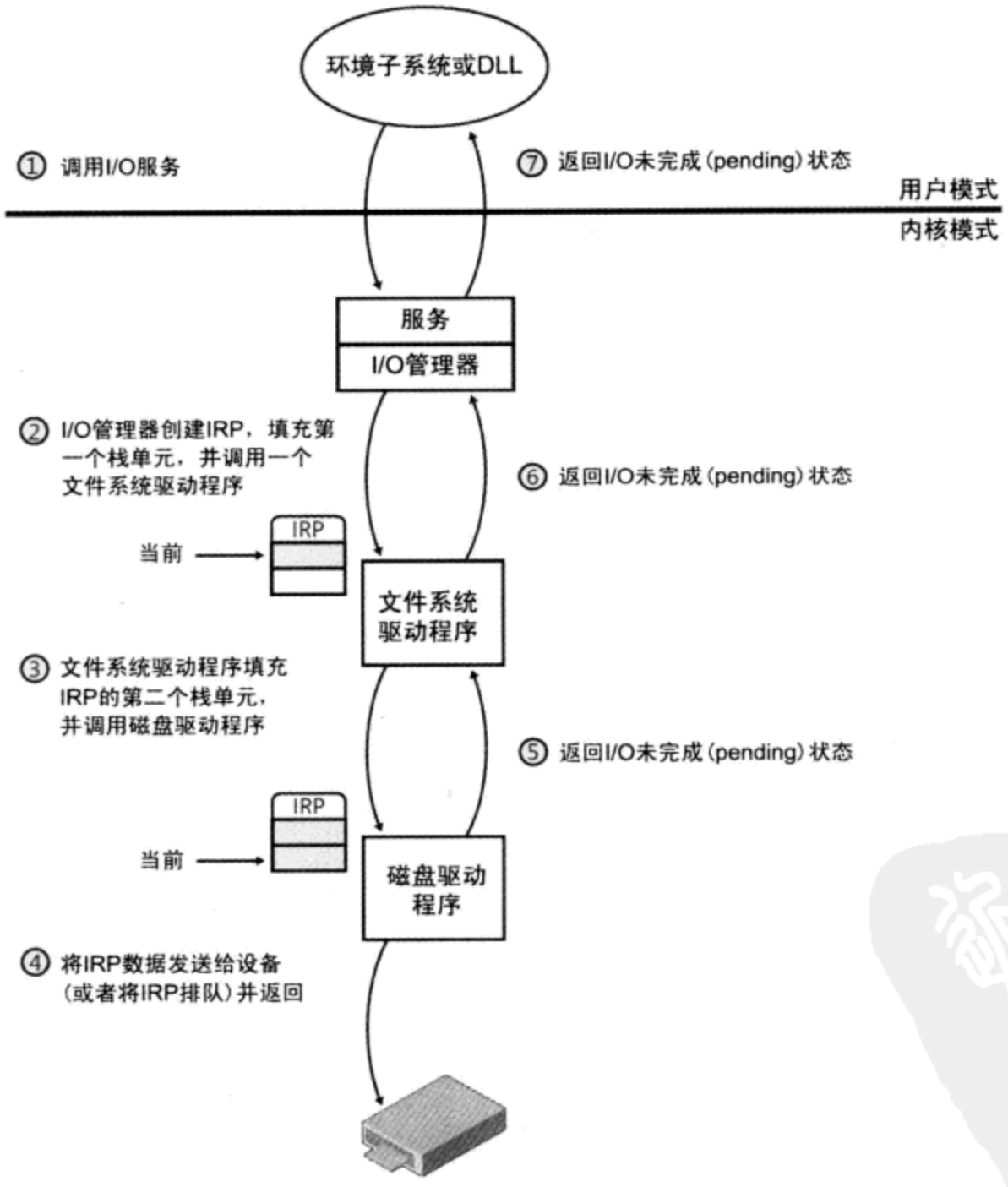


图9.16 · 将一个针对分层驱动程序的异步请求排到队列中

www.TopSage.com



实验：查看IRP

在这个实验中，你将会看到系统中一个未完成的IRP，并且你可以确定此IRP的类型、它所针对的设备、负责管理此设备的驱动程序，以及发出此IRP的线程和该线程所属的进程。

在任何时间点上，系统上至少会有少量的未完成IRP。这是因为，有许多设备，当应用程序向它们发出IRP时，驱动程序只有在某个特定事件（比如，数据已准备好）发生时才能完成这些请求。这样的一个例子是，来自一个网络端点的阻塞方式的读操作。在一个系统上利用内核调试器命令*!irpfind*，你可以看到该系统中尚未完成的IRP：

```
kd> !irpfind
unable to get large pool allocation table - either wrong symbols or pool tagging
 is disabled

Searching NonPaged pool (82502000 : 8a502000) for Tag: Irp?

  Irp    [ Thread ] irpStack: (Mj,Mn)   DevObj  [Driver]
89695868 [00000000] Irp is complete (CurrentLocation 4 > StackCount 3) 0x43776f56
89712008 [8a29d7c0] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
89716008 [8a29d7c0] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
… 89cb3928 [8a3acbc0] irpStack: ( 3, 0)  8a09a030 [ \Driver\Kbdclass]
89cb3c88 [89cb1da8] irpStack: ( c, 2)  8a436020 [ \FileSystem\Ntfs]
89cb4640 [8a165498] irpStack: ( e, 9)  8a19e208 [ \Driver\AFD]
```

在以上输出中，加亮显示的这一项描述了一个针对Kbdclass驱动程序的IRP，所以，它可能是由Windows子系统中专门负责读取键盘输入的原始输入线程（raw input thread）所发出的IRP。利用*!irp*命令检查此IRP，如下：

```
kd> !irp 8a1716f0
Irp is active with 3 stacks 3 is current (= 0x8a1717a8)
 No Mdl System buffer = 8a458180 Thread 8a3acbc0:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                         Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

                         Args: 00000000 00000000 00000000 00000000
>[  3, 0]   0  1 8a1eccb8 8a1262a8 00000000-00000000    pending
               \Driver\Kbdclass
                         Args: 00000078 00000000 00000000 00000000
```

当前活动的栈单元位于最底下。（调试器利用“>”前缀来表示当前活动的栈单元）。其功能代码为3，对应于IRP_MJ_READ。

www.TopSage.com

下一步，在当前活动栈单元中的设备对象地址上执行*!devobj*命令，就可以看到该IRP的目标设备对象是什么：

```
kd> !devobj 8a1eccb8
Device object (8a1eccb8) is for:
 KeyboardClass1 \Driver\Kbdclass DriverObject 8a24bd78
Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002044
Dacl e1ec01e4 DevExt 8a1ecd70 DevObjExt 8a1ece50
ExtensionFlags (0000000000)
AttachedTo (Lower) 8a2e8ac8 \Driver\TermDD
Device queue is not busy.
```

该IRP的目标设备是KeyboardClass1。从上述输出可以看出，附载在此设备对象之下的设备对象是由Termdd驱动程序所拥有的，这说明此设备代表了来自一个终端服务客户程序的键盘输入，而并非物理键盘（以上输出是在一个Windows XP系统上得到的）。

通过使用*!thread*和*!process*命令，我们可以看到关于发出此IRP的线程和进程的细节信息：

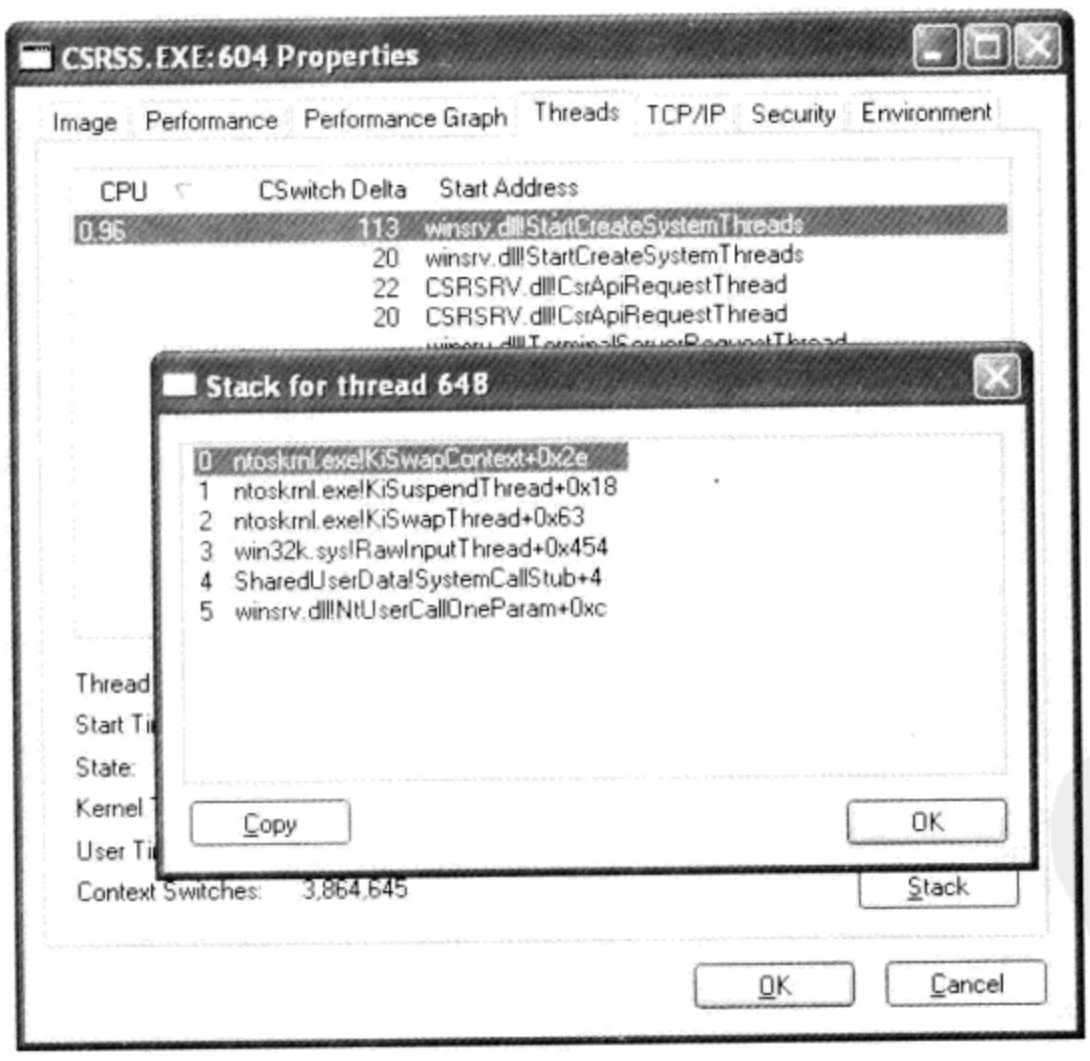
```
kd> !thread 8a3acbc0
THREAD 8a3acbc0  Cid 025c.0288  Teb: 7ffd9000 Win32Thread: e101fab0 WAIT:
rRequest) KernelMode Alertable
    8a3cdb30  SynchronizationEvent
    8a28b4e0  SynchronizationEvent
    8a3cc908  NotificationTimer
    8a294828  SynchronizationEvent
    805453e0  NotificationEvent
    8a2e1830  SynchronizationEvent
    8a45eeb0  SynchronizationTimer
IRP List:
    89cb3928: (0006,01b4) Flags: 00000970  Mdl: 00000000
    8a1716f0: (0006,01b4) Flags: 00000970  Mdl: 00000000
Not impersonating
```


www.TopSage.com

```
DeviceMap                 e10087c0 Owning Process            8a3a08b8
Wait Start TickCount      6844081
Context Switch Count      2130848                 LargeStack
UserTime                  00:00:00.0000
KernelTime                00:00:03.0274
Start Address 0x75b6e8ad
Stack Init baf a0000 Current baf9fa68 Base baf a0000 Limit baf9d000 Call 0
Priority 13 BasePriority 13 PriorityDecrement 0 DecrementCount 16

kd> !process 8a3a08b8 0
GetPointerFromAddress: unable to read from 80543ed4
PROCESS 8a3a08b8  SessionId: 0  Cid: 025c    Peb: 7ffdf000  ParentCid: 0220
    DirBase: 139af000  ObjectTable: e1c0b3d8  HandleCount: 581.
    Image: CSRSS.EXE
```

在进程管理器（来自*www.sysinternals.com*）中，打开Csrss.exe的Properties对话框，然后转到Threads页面，找到该线程，通过其调用栈中的函数名称，可以确认该线程的角色是Windows子系统的一个原始输入线程（raw input thread）。界面如下：



www.TopSage.com

当磁盘驱动程序完成了数据传输以后，磁盘引发中断，I/O完成，如图9.17所示。

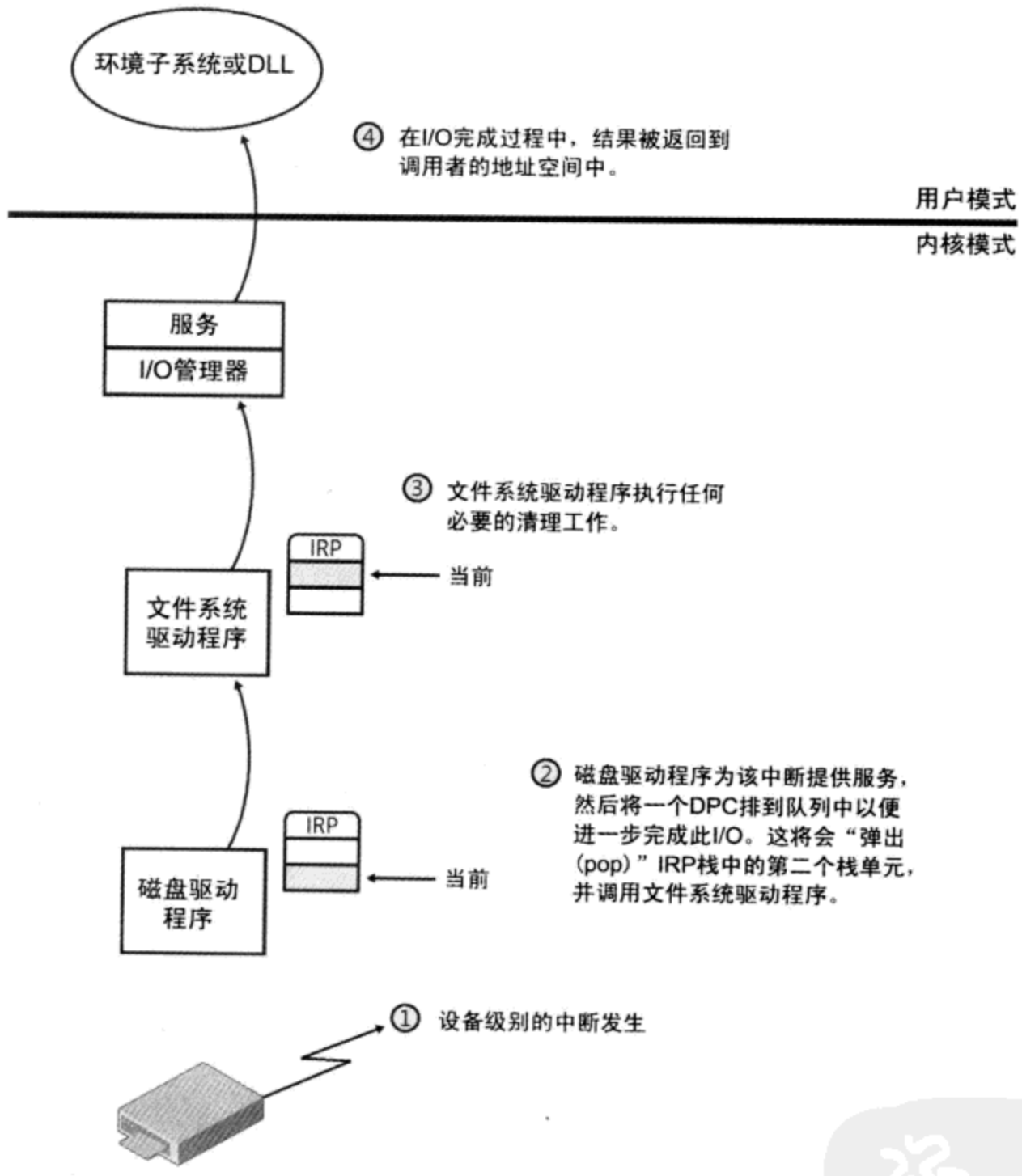


图9.17 完成一个分层的I/O请求

除了“重用同一个IRP”这种做法以外，文件系统还可以建立一组**关联的IRP**，让它们并行地为同一个I/O请求服务。例如，如果从一个文件中读取的数据被分散在磁盘上的不同地方，那么，文件系统驱动程序可以创建多个IRP，每一个IRP从不同的扇区中读取某一部分数据。这一排队过程如图9.18所示。

www.TopSage.com

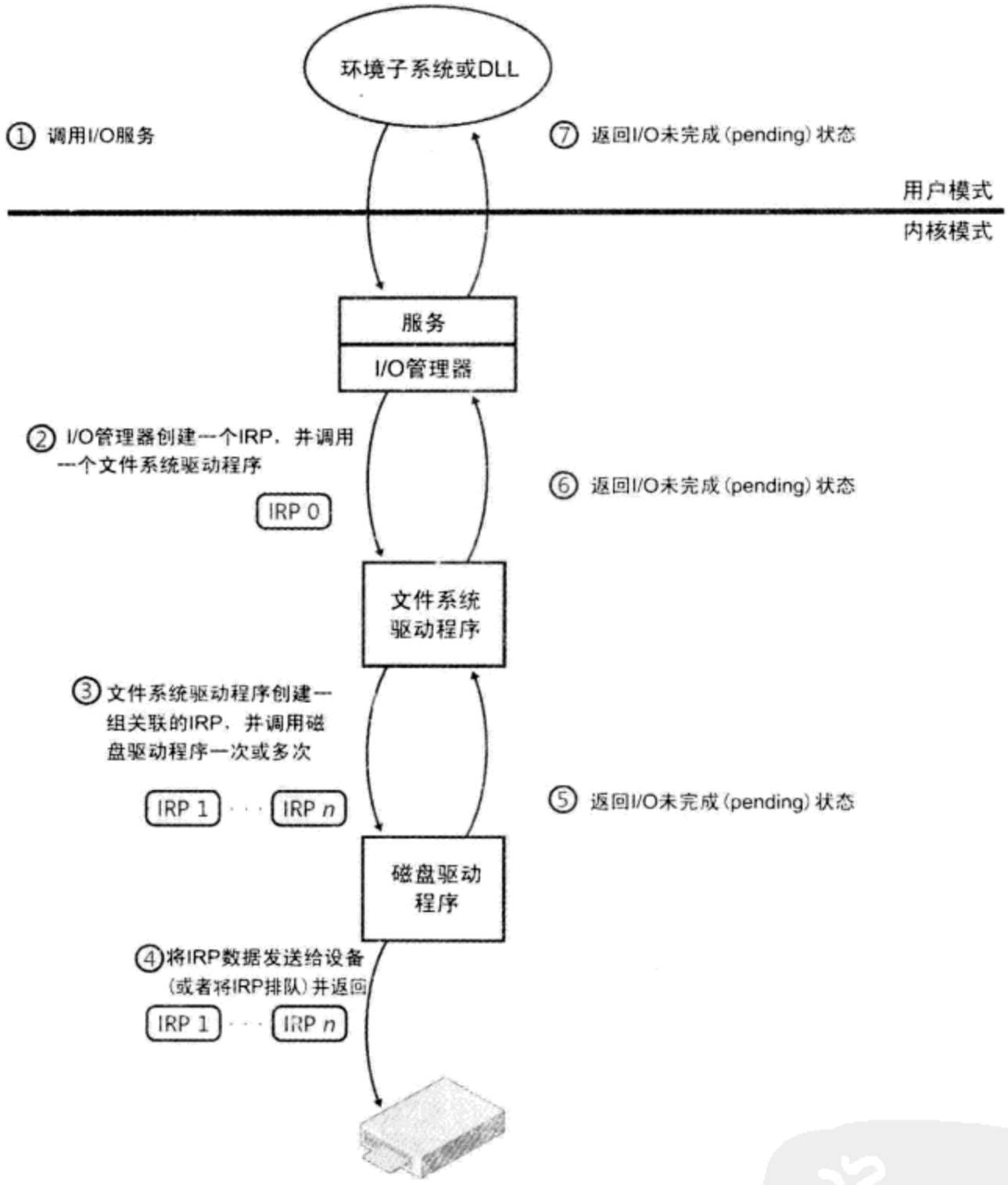


图9.18 关联IRP的排队

文件系统驱动程序将这些关联的IRP传递给设备驱动程序，然后设备驱动程序将它们排到该设备的队列中。这些IRP每次只被处理一个，文件系统驱动程序跟踪记录下所返回的数据。当所有的关联IRP都完成时，I/O系统完成原始的IRP，并返回给调用者，如图9.19所示。

www.TopSage.com

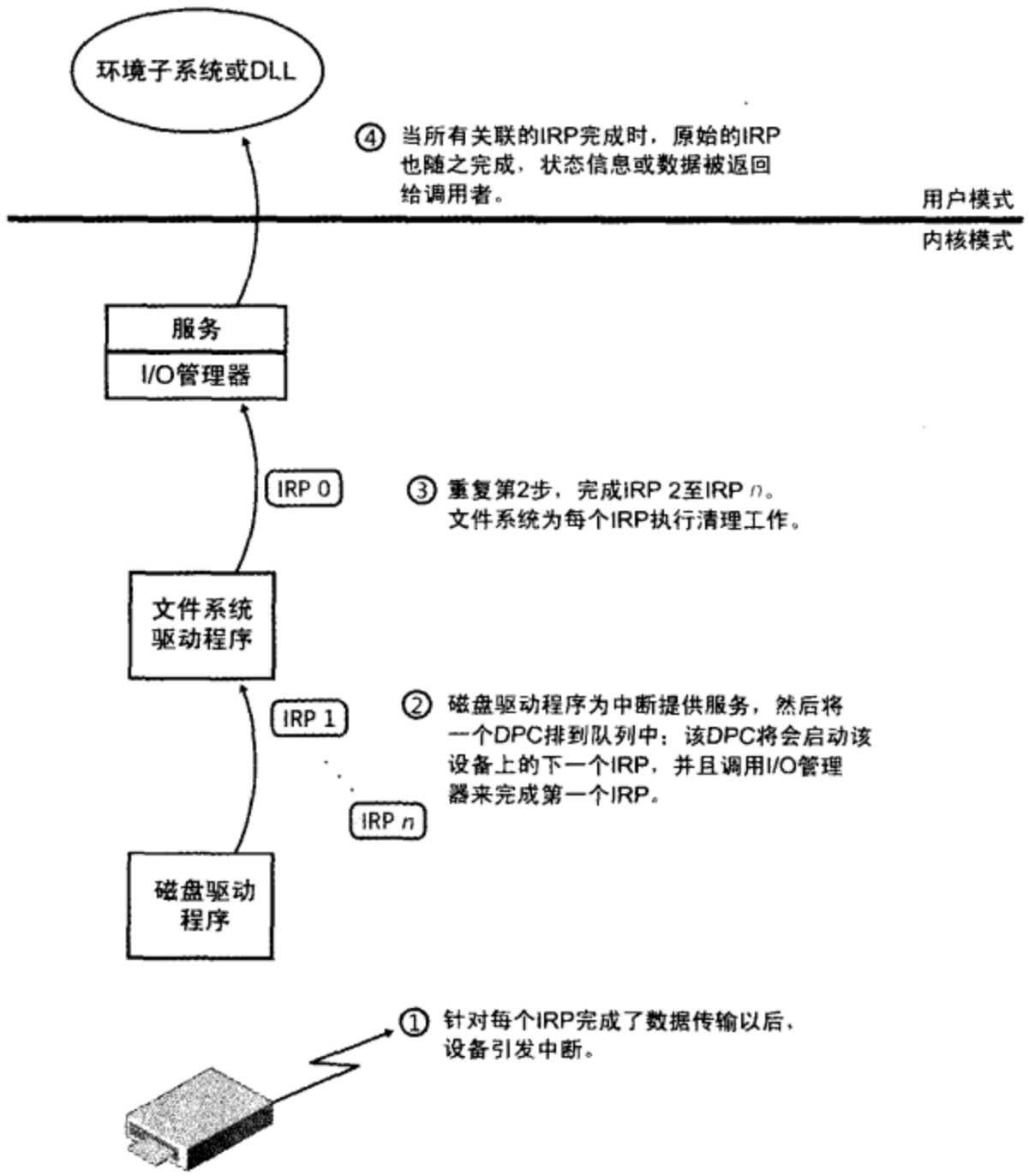


图9.19 完成关联的IRP



> 注 所有负责管理“基于磁盘的文件系统”的Windows文件系统驱动程序，都是某一个至少三层深度的驱动程序栈的一部分：最上边的是文件系统驱动程序，中间的是卷管理器，最底下的是磁盘驱动程序。而且，在这些驱动程序的上面和下面还可以插入任何数量的过滤型驱动程序。为了清楚起见，前述的分层I/O请求的例子只包含一个文件系统驱动程序和一个磁盘设备驱动程序。有关存储管理的更多信息，请参见第10章。

www.TopSage.com

I/O完成端口

要想编写一个高性能的服务器应用程序，必须实现一个高效的线程模型。让太少或者太多的服务器线程来处理客户的请求，都可能会导致性能问题。例如，如果一个服务器创建单个线程来处理所有的请求，那么，客户可能会遭受饥饿，因为服务器同一时刻只能忙于处理一个请求。单个线程也可以并发地处理多个请求，当I/O操作被启动的时候，它可以从一个请求切换到另一个请求，但是，这种结构引入了严重的复杂性，而且也不能很好地发挥多处理器系统的优势。在另一个极端，服务器可以创建一个很大的线程池，因而几乎每一个客户请求都可以由一个专门的线程来处理。这种情形通常会导致线程频繁轮换（thread-thrashing）：大量的线程首先被唤醒、执行一点CPU处理、在等待I/O时被阻塞，然后，在完成了请求处理以后，再一次被阻塞以等待新的请求。如果没有其他请求的话，则太多的线程导致了大量额外的环境切换，这是由于调度器必须将处理器时间在多个活动线程之间划分而引起的。

服务器的目标是尽可能地少引发一些环境切换，它要让它的线程尽量避免不必要的阻塞，同时还要通过使用多个线程来达到最大程度的并行性。理想的情形是，在每个处理器上有一个线程在积极地处理一个客户请求，而且，如果有其他的请求正在等待的话，那么，当处理器上的活动线程完成一个请求时，它们根本不阻塞。然而，为了使这一最优过程能够得以正确地进行，应用程序必须有一种办法，使得一个正在处理客户请求的线程在I/O上阻塞时（比如当它在处理过程中读取一个文件时）能激活另一个线程。

*IoCompletion*对象

应用程序使用*IoCompletion*执行体对象，把它作为与多个文件句柄相关联的I/O操作完成时的焦点。*IoCompletion*执行体对象被导出到Windows中，称为**完成端口**（*completion port*）。一旦一个文件与一个完成端口关联起来，任何在此文件上完成的异步I/O操作都会导致一个完成包（completion package）被排队到此完成端口中。一个线程只需简单地等待一个完成包被排队到此完成端口上，就可以等待到在多个文件上的所有正在进行之中的I/O操作的完成事件。Windows API中的*WaitForMultipleObjects*函数提供了类似的功能，但是，完成端口的好处是，在系统的协助下**并发性**（*concurrency*）是可控的。这里的并发性是指应用程序主动处理客户请求的线程的数量。

当一个应用程序创建一个完成端口时，它指定一个并发性值。该值指示了在任何给定时候正在运行的、与该端口相关联的线程的最大数量。正如前面所说明的，理想情形是，在任何给定时刻，系统中每个处理器都有一个线程在运行。Windows利用与一个端口相关联的并发性值来控制一个应用程序可以有多少个活动的线程。如果与一个端口相关联的活动线程的数量等于此并发性值，那么，正在该完成端口上等待的线程便不允许再运行了。相反地，你可以这样期望，当前活动线程中的某一个线程将会结束它正在处理的请求，并且检查是否有另一个包正在

www.TopSage.com

该端口上等待。如果有的话，该线程只是简单地把这个包抓过来，然后处理该包。在这个过程中，没有环境切换，所以，CPU的所有能力几乎都被利用起来了。

使用完成端口

图9.20从高层次角度示出了完成端口的操作。完成端口是通过调用Windows API函数*CreateIoCompletionPort*来创建的。阻塞在完成端口上的线程都与该完成端口关联起来，而且它们按照后进先出（LIFO）的顺序被唤醒，因而，最近被阻塞的线程正是被给予下一个包的线程。对于长时间被阻塞的线程，它们的栈有可能被换出到磁盘上，所以，如果与一个端口相关联的线程数量超过了当前的处理能力，则最长时间被阻塞的线程的内存印迹可以被最小化。

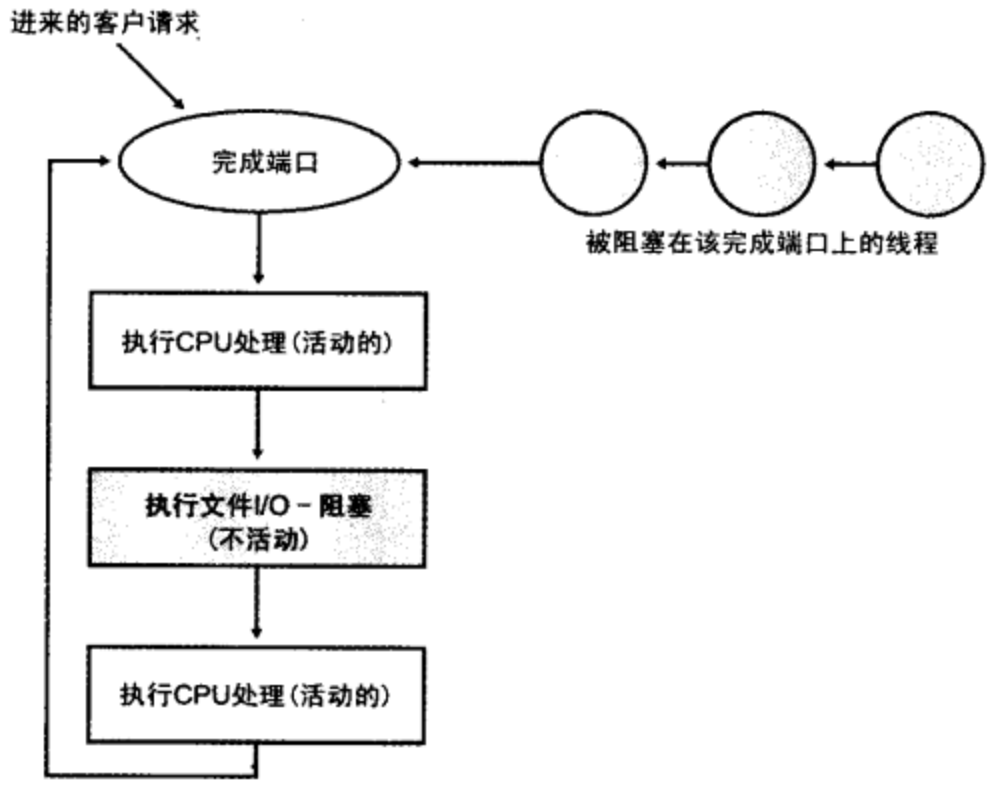


图9.20 I/O完成端口的操作

服务器应用程序往往通过网络端点来接收客户请求，而这些网络端点是由文件句柄来表示的。这样的例子包括Windows Socket 2（Winsock2）套接字或者命名管道。当服务器创建它的通信端点时，它将这些通信端点与一个完成端口关联起来，并且它的线程通过调用*GetQueuedCompletionStatus*来等待此端口上进来的请求。当一个线程在此完成端口上得到一个包时，它便不再等待，开始处理此请求，从而变成一个活动的线程。一个线程在它的处理过程中将会阻塞许多次，比如当它需要从磁盘上的文件中读写数据时，或者当它与其他的线程进行同步时。Windows检测到这一行为，并且意识到该完成端口已经有了至少一个活动线程。因此，当一个线程由于阻塞而变成不活动的时候，如果在队列中有包的话，则唤醒一个正在等待该完成端口的线程。

www.TopSage.com

Microsoft的指导原则是，将并发性值设置成大约等于该系统中处理器的数目。要记住，针对一个完成端口的活动线程的数量是有可能超过此并行性限制值的。请考虑此限制值被指定为1的情形。一个客户请求进来了，然后有一个线程被分发来处理该请求，因而该线程变成活动的。第二个请求又到达了，但是，正在该端口上等待的第二个线程不允许进行处理，因为并发性限制值已经达到了。然后，第一个线程在等待一个文件I/O的时候阻塞了，因此它变成不活动的。于是，第二个线程被解放了，而且，当它尚在活动时，第一个线程的文件I/O完成了，这使得它再一次变成活动的。从这个点开始，一直到这两个线程中有一个被阻塞以前，并发性值始终是2，高于限制值1。在绝大多数时候，活动线程的数量将维持在并发性限制值上，或者刚刚超过一点。

这一完成端口API，也使得服务器应用程序有可能通过*PostQueuedCompletionStatus*函数，将私有定义的完成包排队到一个完成端口中。服务器通常使用此函数来通知它的线程有外部事件发生了，比如需要温和地停机。

I/O完成端口操作

Windows应用程序通过调用Windows API *CreateIoCompletionPort*函数并且指定一个NULL完成端口句柄来创建完成端口。这将导致*NtCreateIoCompletion*系统服务被执行。执行体的*IoCompletion*对象建立在一个称为**队列**（*queue*）的内核同步对象的基础之上。因此，该系统服务创建一个完成端口对象，并且在该端口对象所分配到的内存中初始化一个队列对象。（指向此完成端口的指针也指向此队列对象，因为队列对象位于此端口内存的开始处）。队列对象有一个并发性值，它是当一个线程初始化队列对象时被指定的，在这里的情形下，此并发性值正是被传递给*CreateIoCompletionPort*的并发性值。*NtCreateIoCompletion*调用*KeInitializeQueue*函数来初始化一个端口的队列对象。

当一个应用程序调用*CreateIoCompletionPort*来把一个文件句柄和一个端口关联起来时，*NtSetInformationFile*系统服务被执行，并且该文件句柄是主要的参数。被设置的信息类别是*FileCompletionInformation*，而完成端口的句柄以及*CreateIoCompletionPort*的*CompletionKey*参数则是数据值。*NtSetInformationFile*通过解引用操作（dereference）从该文件句柄获得对应的文件对象，并且申请一个完成环境数据结构。

最后，*NtSetInformationFile*设置该文件对象中的*CompletionContext*域，并使它指向此环境结构。当一个异步I/O操作在一个文件对象上完成时，I/O管理器检查该文件对象中的*CompletionContext*域是否为非NULL。如果是的话，I/O管理器申请一个完成包，并且通过调用*KeInsertQueue*将它排队到此完成端口中，它在调用该函数时指定此端口作为该完成包要插入的队列（注意，完成端口对象和队列对象是同义的）。

当一个服务器线程调用GetQueuedCompletionStatus时，系统服务NtRemoveIoCompletion被执行。在验证了参数，并且将完成端口的句柄转译成一个指向该端口的指针以后，NtRemoveIoCompletion调用KeRemoveQueue。

www.TopSage.com

正如你所看到的，*KeRemoveQueue*和*KeInsertQueue*是完成端口机制背后的两个引擎。这两个函数决定了一个正在等待I/O完成包的线程是否应该被激活。在内部，队列对象维护了当前活动线程的计数值，以及最大的活动线程数量。如果当一个线程调用*KeRemoveQueue*时当前数量等于或者超过了此最大值，那么，该线程将被放到（按照LIFO顺序）一个线程列表中，此列表中的线程都是在等待有机会轮到它来处理一个完成包。此线程列表挂在队列对象的外面。线程的控制块数据结构中有一个指针引用了一个与之相关联的队列的队列对象；如果该指针是NULL，则该线程没有与队列关联。

Windows依赖于线程控制块中的队列指针来跟踪和记录那些“由于被阻塞在除了完成端口之外的其他事情上而变成不活动”的线程。那些有可能会导致一个线程阻塞的调度器例程（比如*KeWaitForSingleObject*、*KeDelayExecutionThread*等等）要检查该线程的队列指针。如果该指针不是NULL，则这些函数调用*KiActivateWaiterQueue*，这是一个与队列相关的函数，它会递减与该队列相关联的活动线程的计数值。如果递减的结果值小于最大数值，并且至少有一个完成包在该队列中，那么，处于该队列的线程列表最前面的那个线程被唤醒，并且把最老的那个包交给它。与此相反，无论何时，与一个队列相关联的线程在阻塞之后被唤醒时，调度器执行*KiUnwaitThread*，它会递增该队列的活动线程计数值。

最后，Windows API函数*PostQueuedCompletionStatus*会导致*NtSetIoCompletion*系统服务被执行。该函数只是简单地利用*KeInsertQueue*，将指定的包插入到该完成端口的队列中。

图9.21显示了一个正在运行中的完成端口对象的例子。尽管有两个线程已经准备好要处理完成包了，但由于并发性值为1，因而使得只有一个与该完成端口相关联的线程可以是活动的，所以这两个线程都被阻塞在该完成端口上。

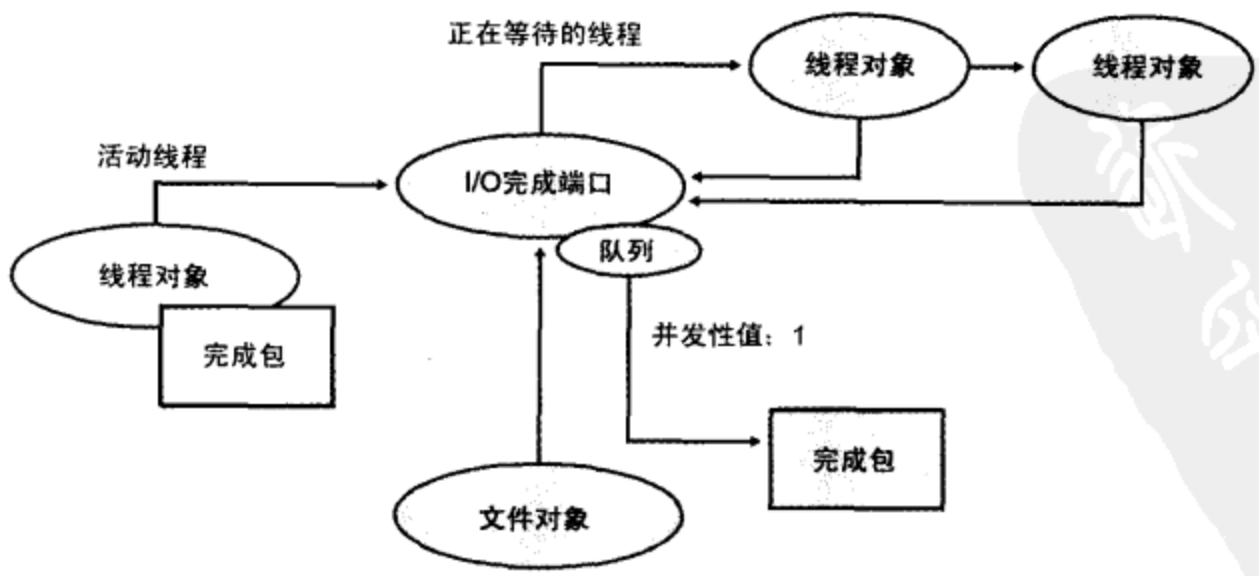


图9.21 I/O完成端口操作

www.TopSage.com

驱动程序检验器（Driver Verifier）

在第7章中曾经介绍过驱动程序检验器。驱动程序检验器包含的几个选项可以检查与I/O相关的操作的正确性。图9.22显示了Windows Server 2003驱动程序检验器管理器的界面，其中与I/O相关的选项都被选中了。

即使你没有选中任何选项，检验器也会监视那些被选中来做检验的驱动程序，寻找许多非法操作，包括在无效的IRQL上调用内核内存池函数、两次释放内存，以及零大小的内存分配请求。

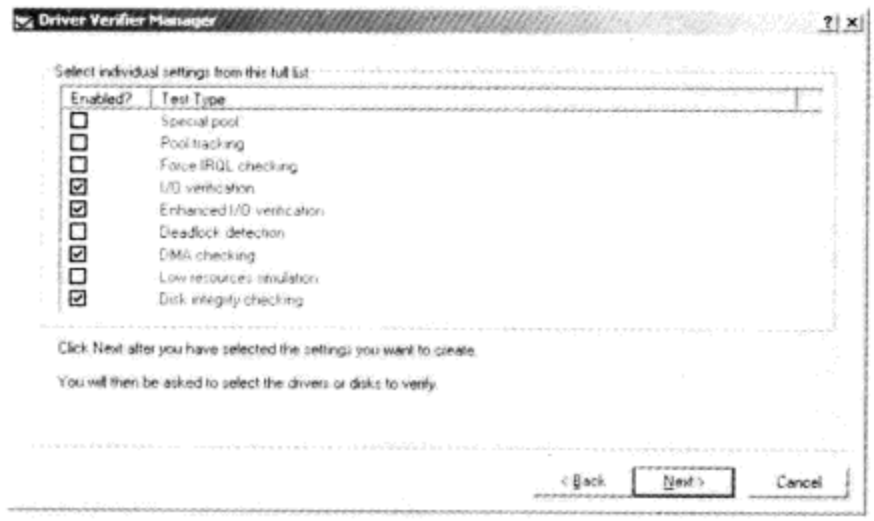


图9.22 驱动程序检验器的I/O相关的选项

由这些I/O相关的选项来启动的额外检查包括以下几个方面。

- **I/O检验** 当这一选项被选中时，I/O管理器为被检验的驱动程序从一个特殊的内存池中分配IRP，而且这些IRP的使用也会被跟踪。若一个IRP在完成时包含了无效的状态，或者当一个无效的设备对象被传递给I/O管理器时，检验器将会使系统崩溃。（在Windows 2000中，这称为I/O检验级别1）
- **I/O检验级别2** 这一选项只有在Windows 2000中才有，它导致对于IRP完成操作和栈用法的更多的严格测试。
- **增强的I/O检验** 这一选项是在Windows XP中引入的，它监视所有的IRP以确保驱动程序在以异步方式来完成这些IRP时能够正确地标记它们；也确保驱动程序正确地管理设备栈单元，以及它们只删除设备对象一次。而且，检验器随机地对驱动程序进行压力测试，它的做法是：给它们发送一些假的电源管理和WMI IRP；改变设备被列举的顺序；当PnP和电源IRP完成时调整这些IRP的状态（以便测试那些从分发例程中返回不正确状态的驱动程序）。
- **DMA检查DMA–直接内存访问** 这是一种硬件支持的机制，它使得无需CPU的介入，设备就可以在它和物理内存之间传输数据。I/O管理器提供了许多函数，驱动程序利用这些函数来调度和控制DMA操作；这一选项启动了对于这些函数的正确用法的检查，以及对于I/O管理器为DMA操作而提供的缓冲区的检查。

www.TopSage.com

- **磁盘完整性检验** 这一选项只有在Windows Server 2003上才可以使用。当你启动了这一选项时，检验器监视磁盘的读和写操作，以及对于相关联数据的校验和检查。当磁盘读操作完成时，它检查是否它原来已经存储了一个校验和，如果新的校验和与老的不匹配，则让系统崩溃，因为这表明在硬件层上磁盘已经破坏了。
- **SCSI检验** 这是在Windows XP中引入的，而且，在驱动程序检验器的选项对话框中是不可见的。然而，当你选中了一个SCSI小端口来做检验，并且至少启动了其他选项之一时，此选项就被打开了。当它被选中时，检验器检查SCSI小端口驱动程序对于SCSI小端口库驱动程序（storport.sys或scsiport.sys）所提供的函数的正确用法。它所做的检查包括：确保驱动程序没有多次完成一个请求；确保它没有传递无效的参数；确保它没有花多于指定的时间来完成I/O操作（关于SCSI小端口驱动程序的更多信息，请参见第10章）。

驱动程序检验器作为一个工具，主要是帮助设备驱动程序开发人员来发现他们的代码中的错误，但是它对于系统管理员分析系统崩溃问题也是一个功能强大的工具。第14章介绍了它在崩溃分析诊断中所提供的能力。

9.4 即插即用（PnP）管理器

在“为了让Windows能够识别和适应硬件配置的变化”而提供的支持之中，PnP管理器是最主要的组件。用户无需理解硬件或手工配置的复杂性，就可以安装和移除设备。例如，PnP管理器允许一台正在运行Windows的便携式电脑在被放到一个插接站（docking station）上时，能够自动检测到该插接站中其他的设备，并且能让用户使用这些设备。

要做到即插即用支持，要求在硬件、设备驱动程序和操作系统层次上紧密合作。用于“列举和标识总线上附载的设备”的工业标准，是Windows即插即用支持的基础。例如，USB标准定义了USB总线上的设备标识其自身的方法。有了这一基础，Windows即插即用支持提供了以下的能力。

- PnP管理器自动识别出已安装的设备，这一过程包括在引导过程中列举出所有附载在系统中的设备，以及在系统执行过程中检测出设备的加入和移除。
- 硬件资源分配是PnP管理器提供的一种能力，它将附载在系统上的设备的硬件资源需求（中断、I/O内存、I/O寄存器，以及与总线相关的资源）收集起来，并且，在一个被称为**资源判优**（*resource arbitration*）的过程中以最优的方式来分配这些资源，因而每个设备都能满足其操作的必要需求。因为硬件设备可以在引导时进行的资源分配完成以后再加入到系统中，所以，PnP管理器也必须能够重新分配资源，以满足动态加入的设备的需要。

www.TopSage.com

- 加载正确的驱动程序是PnP管理器的另一项责任。PnP管理器根据一个设备的标识，来确定该系统中是否已经安装了一个能够管理该设备的驱动程序，如果是的话，则指示I/O管理器加载该驱动程序。如果尚未安装合适的驱动程序，则内核模式的PnP管理器要跟用户模式的PnP管理器进行通信，以便安装该设备，在寻找合适的驱动程序的过程中，可能还要求用户的协助。
- PnP管理器也为硬件配置的变化实现了应用程序和驱动程序的相应机制。应用程序或者驱动程序有时候要求一种特定的硬件设备才能工作，所以Windows为它们提供了一种方法来请求获得有关设备的存在、加入或者移除的通知。

即插即用支持的级别

Windows的目标是提供全面的即插即用支持，但是，支持的级别可能要取决于所附载的设备和所安装的驱动程序。如果一个设备或者驱动程序不支持即插即用，那么，系统的即插即用程度难免要有所下降。而且，一个不支持即插即用的驱动程序可能会妨碍其他的设备被系统所使用。表9.2显示了设备和驱动程序能够和不能够支持即插即用的各种组合的结果。

表9.2 设备和驱动程序的即插即用能力

	驱动程序的类型	
设备的类型	**即插即用**	**非即插即用**
即插即用	完全即插即用	没有即插即用能力
非即插即用	可能有部分即插即用的能力	没有即插即用能力

一个非即插即用兼容的设备是指，它不支持自动监测，比如老的ISA声卡。因为操作系统不知道此硬件物理上位于什么地方，所以，有些特定的操作是不允许的，比如便携式电脑的拔除（undocking）、睡眠和休眠。然而，如果针对该设备手工安装了一个即插即用驱动程序，那么，此驱动程序至少可以为该设备实现“指导PnP管理器进行资源分配”的能力。

非即插即用兼容的驱动程序包括遗留的驱动程序，比如那些在Windows NT 4上运行的驱动程序。尽管这些驱动程序在后来的Windows版本上可能还能继续工作，但是，一旦当需要重新分配资源才能满足动态加入进来的设备的需要时，PnP管理器无法为老驱动程序的设备重新配置资源。例如，一个设备或许能够使用I/O内存范围A和B，在引导过程中PnP管理器为它分配了范围A。如果后来另外一个设备也附载到系统中，但它只能使用A，那么，PnP管理器无法指示第一个设备的驱动程序，让它重新配置成使用范围B。这妨碍了第二个设备得到它所要求的资源，从而导致该设备无法为系统所利用。遗留驱动程序也削弱了一台机器的睡眠和休眠的能力（更多的细节，请参见后面的“电源管理器”一节）。

www.TopSage.com

驱动程序对于即插即用的支持

为了支持即插即用，一个驱动程序必须实现一个即插即用分发例程、一个电源管理分发例程（在本章后面的“电源管理器”一节中介绍），以及一个“增加-设备（add-device）”例程。然而，与功能型或过滤型驱动程序相比，总线型驱动程序必须支持不同类型的即插即用请求。例如，在系统引导过程中，当系统指示PnP管理器列举设备时（详细的细节在本章后面描述），PnP管理器请求总线型驱动程序给出它们各自总线上所发现的对设备的描述。描述的信息包括可惟一标识每个设备的数据，以及设备的资源需求。PnP管理器接受这些信息，并且对于检测到的设备，加载那些已经安装的功能型和过滤型驱动程序。然后，它针对这些驱动程序所负责的每个已安装的设备，调用每个驱动程序的“增加-设备”例程。

功能型和过滤型驱动程序在它们的“增加-设备”例程中，准备管理它们各自的设备，但是，它们并不真的与设备硬件进行通信。相反地，它们等待PnP管理器向它们的即插即用分发例程发送一个针对该设备的**启动-设备**（*start-device*）命令。在发送**启动-设备**命令以前，PnP管理器先执行资源判优（resource arbitration），以确定为一个设备分配哪些资源。**启动-设备**命令包含了PnP管理器在资源判优过程中确定的资源分配。当一个驱动程序接收到一个**启动-设备**命令时，它可以配置它的设备，以便使用指定的资源。如果一个应用程序试图打开一个尚未完成启动的设备，那么它会接收到一个表明该设备不存在的错误。

在一个设备启动起来以后，PnP管理器可以向驱动程序发送额外的即插即用命令，包括与“该设备从系统中移除”有关的命令，或者与资源重新分配有关的命令。例如，当用户调用移除/弹出（remove/eject）设备工具时（如图9.23所示，你只要在任务栏的PC卡图标上右键点击，再选择Unplug Or Eject Hardware，就可以访问此功能），PnP管理器为了告诉Windows要弹出一块PCMCIA卡，给任何已经注册的要接收该设备即插即用通知的应用程序发送一个**询问-移除**（*query-remove*）通知。应用程序通常会为了它们的一些句柄而注册这一通知，它们要在**询问-移除**通知过程中，关闭这些句柄。如果没有应用程序否定这一**询问-移除**请求，则PnP管理器向拥有此弹出设备的驱动程序发送一个**询问-移除**命令。在这个点上，驱动程序有机会拒绝此移除请求，或者，确保任何涉及该设备的未完成I/O操作已经完成，并且拒绝任何后续的针对此设备的I/O请求。如果驱动程序同意此移除请求，而且没有任何已打开的句柄指向此设备，那么，PnP管理器接下来向驱动程序发送一个**移除**（*remove*）命令，请求该驱动程序不再继续访问此设备，以及释放它替该设备申请的任何资源。

www.TopSage.com

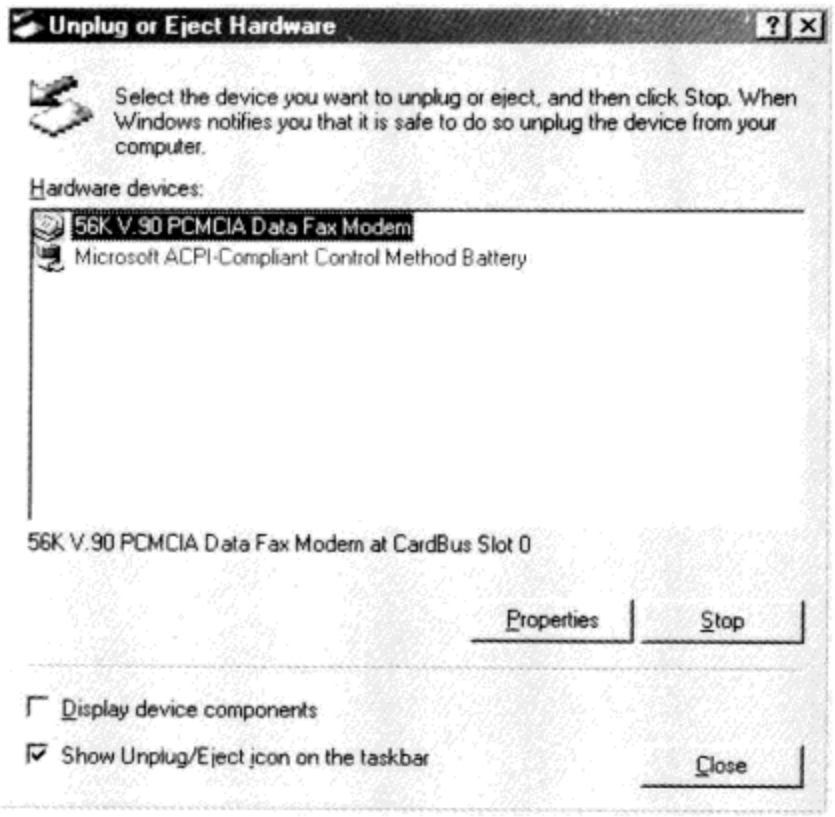


图9.23 PC卡移除/弹出工具

当PnP管理器需要重新分配一个设备的资源时，它首先向驱动程序发送一个**询问-停止**（*query-stop*）命令，以此来询问该驱动程序，它是否可以临时中止该设备上的进一步动作。驱动程序如果觉得这样做不会导致数据丢失或者造成破坏，则同意此请求；或者它拒绝此请求。如同**询问-移除**命令一样，如果驱动程序同意此请求，则它完成当前正在进行中的I/O操作，而且，对于不具备“中止之后再重启”能力的设备，不再激发进一步的I/O请求。驱动程序通常将新的I/O请求放到一个队列中，以便使得资源重分配过程对于当前正在访问该设备的应用程序来说是透明的。然后，PnP管理器向驱动程序发送一个**停止**（*stop*）命令。在这个点上，PnP管理器可以指示驱动程序给设备分配不同的资源，并且再次向驱动程序发送一个针对此设备的**启动-设备**（*start-device*）命令。

在本质上，这些即插即用命令是在指导一个设备经历各种操作状态，它们构成了一个定义良好的状态转移表，图9.24显示了此状态转移表的一种简化形式（为了清楚起见，有几种可能的状态转移和即插即用命令被省略了。而且，这里显示的状态图是由功能型驱动程序实现的。总线型驱动程序实现了一个更为复杂的状态图）。该图中显示的一个状态，即由PnP管理器的**突然-移除**（*surprise-remove*）命令所产生的状态，我们还没有讨论过。当用户在毫无警告的情况下移除一个设备时，或者当一个用户未使用移除/弹出工具来弹出一块PCMCIA卡时，或者设备失败的时候，这一命令就会发生。**突然-移除**命令告诉驱动程序：立即停止所有与该设备的交互操作，因为该设备已经不再附载在系统中了；同时也要取消所有未完成的I/O请求。

www.TopSage.com

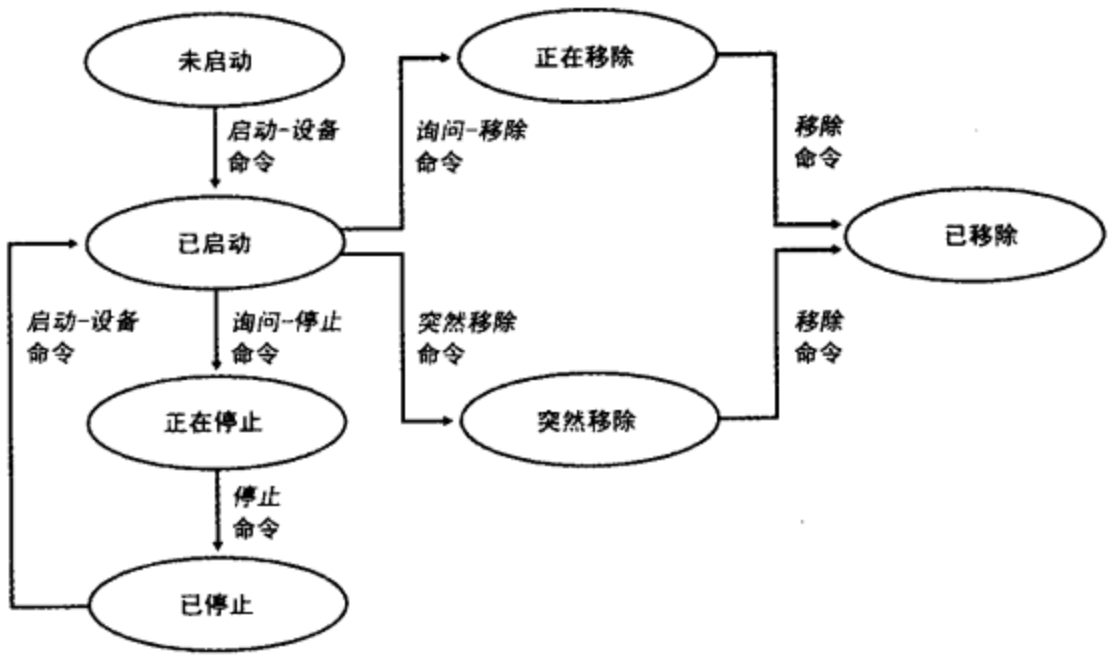


图9.24 设备的即插即用状态转移图

驱动程序加载、初始化和安装

在Windows上，驱动程序的加载和初始化是由两种加载类型构成的：显式的加载和基于列举的加载。显式加载是由注册表的HKLM\SYSTEM\CurrentControlSet\Services分支来控制的，其过程如第4章中的“服务应用”一节所描述。当PnP管理器动态地为“总线型驱动程序在总线列举过程中报告的设备”加载驱动程序时，基于列举的加载就会发生。

Start值

在第4章中，我们解释了每一个驱动程序和Windows服务在当前控制集的Services分支下面都有一个注册表键。该键中包含的值指定了映像的类型（比如，Windows服务、驱动程序和文件系统）、驱动程序或者服务的映像文件的路径，还有一些值控制了驱动程序或者服务的加载顺序。在“显式的设备驱动程序加载”和“Windows服务的加载”之间有以下两个主要的区别：

- 只有设备驱动程序才可以指定引导-启动（0）或系统-启动（1）的Start值；
- 设备驱动程序可以使用Group和Tag值来控制它们在引导阶段内的加载顺序，但是，与Windows服务不同的是，它们不能指定DependOnGroup或DependOnService值。

第5章描述了引导过程的各个阶段，也解释了驱动程序的Start值为0意味着操作系统加载器会加载此驱动程序。Start值为1意味着I/O管理器在执行体子系统完成初始化以后加载此驱动程序。I/O管理器按照驱动程序在引导阶段被加载的顺序来调用这些驱动程序的初始化例程。如同Windows服务一样，驱动程序也在它们的注册表键中使用Group值来指定它们所属的组；注册表值HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List决定了驱动程序组在引导阶段被加载的顺序。

www.TopSage.com

一个驱动程序可以进一步精确地指定它的加载顺序，做法是，在它的注册表键下包含一个Tag值，以控制它在组内的加载顺序。I/O管理器对于每个组内的驱动程序，根据它们的注册表键中定义的Tag值来排列它们的顺序。没有定义Tag值的驱动程序被放在组内驱动程序列表的尾部。你可能会认为，I/O管理器先处理低Tag值的驱动程序，再处理高Tag值的驱动程序，但这样的假设不一定正确。注册表键HKLM\SYSTEM\CurrentControlSet\Control\GroupOrderList定义了组内的Tag优先顺序。通过此键，Microsoft和设备驱动程序开发人员可以很方便地重新定义这一整数系统。

下面是一些关于驱动程序如何设置其Start值的指导原则。

- 非即插即用驱动程序设置其Start值，以反映出它们想要在引导阶段加载到系统中。
- 那些必须在系统引导过程中由引导加载器来加载的驱动程序，包括即插即用的和非即插即用的驱动程序，必须指定其Start值为引导-启动（0）。这样的驱动程序例子包括系统总线驱动程序和引导文件系统驱动程序。
- 如果一个驱动程序对于引导系统来说并不是必要的，并且它检测到一个系统总线驱动程序无法列举的设备，那么它应该指定其Start值为系统-启动（1）。这种驱动程序的一个例子是串行口驱动程序，它通知PnP管理器，标准PC串行口是存在的（它们是Setup程序检测到并记录在注册表中的）。
- 在系统引导时可以不必出现的非即插即用驱动程序或文件系统驱动程序，指定其Start值为自动-启动（2）。这种驱动程序的一个例子是多UNC（Universal Naming Convention，统一命名规范）提供者（MUP，Multiple UNC Provider）驱动程序，它为远程资源提供了UNC路径名支持（例如，\\REMOTECOMPUTERNAME\SHARE）。
- 对于引导系统并非必要的即插即用驱动程序指定其Start值为按需-启动（3）。这样的驱动程序的例子包括网络适配器驱动程序。

对于即插即用驱动程序以及可列举设备的驱动程序，Start值的惟一用途是，确保操作系统加载器加载该驱动程序——如果此驱动程序对于系统成功引导是必需的话。除此以外，PnP管理器的设备列举过程（接下来介绍）会确定即插即用驱动程序的加载顺序。

设备列举

PnP管理器从一个称为Root的虚拟总线型驱动程序开始设备列举过程，Root代表了整个计算机系统，也担当了所有非即插即用驱动程序和HAL的总线型驱动程序。HAL也被认为是一个总线型驱动程序，它可以列举所有直接附载在主板上的设备，以及诸如电池之类的系统部件。实际上，HAL并不是真的执行列举操作，相反，它依赖于Setup进程记录在注册表中的硬件描述来检测系统主总线（绝大多数情况下是PCI总线）和诸如电池和风扇之类的设备。

www.TopSage.com

主总线驱动程序列举其总线上的设备，可能还会发现其他的总线，PnP管理器也会为这些总线初始化驱动程序。这些驱动程序又依次检测其他的设备，包括其他的辅助总线。这一递归的列举过程、驱动程序加载（如果该驱动程序尚未被加载的话），以及进一步的列举过程不断进行，直至当前系统上所有的设备都已经被检测到，并且被配置好为止。

随着总线型驱动程序不断地向PnP管理器报告其检测到的设备，PnP管理器创建了一棵内部树，称为**设备树**（*device tree*），它代表了设备之间的关系。树中的节点称为*devnode*，每个devnode包含了有关该设备所对应的设备对象的信息，以及其他的、由PnP管理器存储在该devnode中的、与即插即用相关的信息。图9.25显示了一棵简化的设备树的例子。该系统是ACPI兼容的，所以，一个ACPI兼容的HAL承担了主总线列举器的角色。该系统的主总线是一个PCI总线，它连接了USB、ISA和SCSI总线。

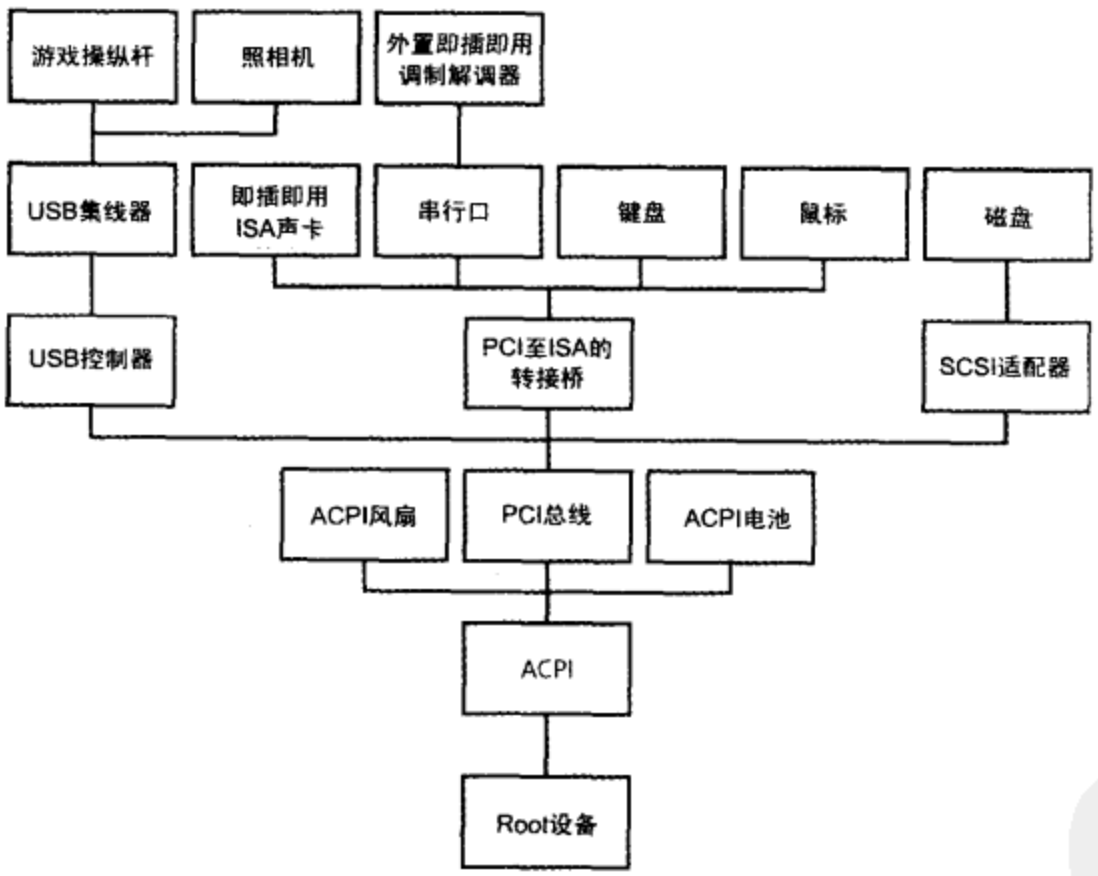


图9.25 设备树示例

设备管理器（Device Manager）工具在它的默认配置下，显示了当前系统上出现的设备的一个简单列表。你只需通过Start菜单的Programs/Administrative Tools文件夹，在计算机管理加载件（Computer Management snap-in）中就可以访问设备管理器工具；或者，你也可以在控制面板中通过System工具的Hardware页面来访问设备管理器工具。你从设备管理器的View菜单中选择“Devices By Connection”选项，就可以看到这些设备在设备树中的相对位置。图9.26显示了设备管理器的“Devices By Connection”视图的一个例子。

www.TopSage.com

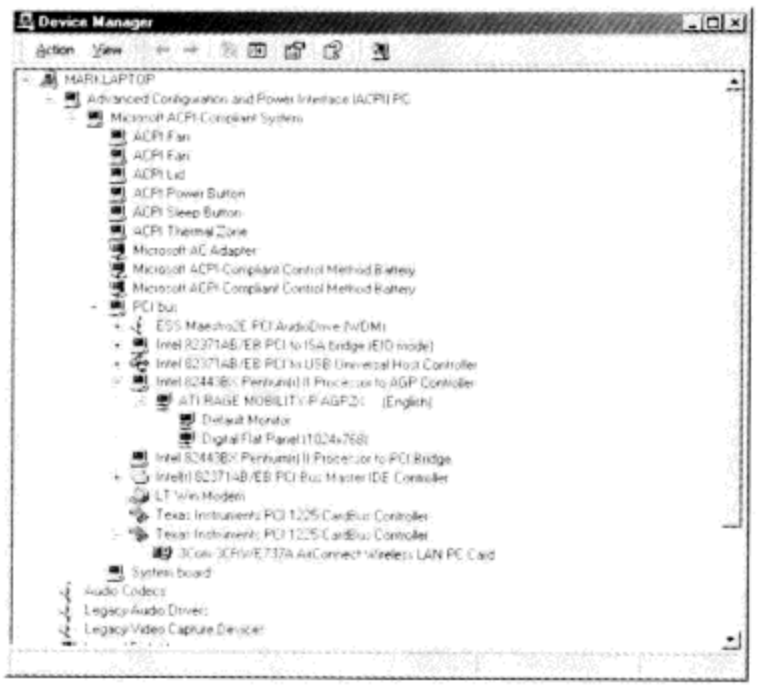


图9.26 设备管理器显示了设备树

考虑到设备列举的因素，驱动程序的加载和初始化顺序如下。

1. I/O管理器调用每个引导-启动的驱动程序的入口例程。如果一个引导驱动程序有子设备，那么，I/O管理器列举这些设备，并且将这些设备的存在事实报告给PnP管理器。如果子设备的驱动程序是引导-启动的驱动程序，那么，这些子设备也被配置和启动。如果一个设备有一个非引导-启动的驱动程序，那么PnP管理器为该设备创建一个devnode，但是不启动它或加载它的驱动程序。
2. 在引导-启动的驱动程序被初始化以后，PnP管理器检查一遍设备树，将第1步中没有加载的devnode的驱动程序加载进来，并且启动它们的设备。在每个设备启动时，PnP管理器列举它的子设备（如果它有子设备的话，则启动这些设备的驱动程序，并且根据需要执行它们的子设备的列举过程）。PnP管理器在这一步中为所有检测到的设备加载驱动程序，而**根本不管驱动程序的Start值**。（一个例外是，如果Start值被设置为“禁止启动[disabled]”）。到这一步的最后，除了那些不可被列举的设备以及它们的子设备以外，所有的即插即用设备都已经加载了相应的驱动程序，并且被启动起来了。
3. PnP管理器将任何尚未被加载的、Start值为系统-启动的驱动程序加载进来。这些驱动程序检测并报告其不可被列举的设备。PnP管理器为这些设备加载驱动程序，直至所有被列举的设备都已经被配置和启动为止。
4. 服务控制管理器加载那些被标记为自动-启动的驱动程序。

当PnP管理器和电源管理器向设备发出即插即用IRP和电源IRP时，它们可以使用设备树作为指导。一般而言，IRP顺着一个devnode的最上边向最底下流动，在有些情形下，一个devnode中的驱动程序会创建新的IRP，并发送给其他的devnode，IRP总是向着根（Root）的方向流动。在本章后面还要进一步介绍即插即用IRP和电源IRP的流动。

www.TopSage.com



实验：将设备树转储出来

一种查看比设备管理器显示的设备树更为详细信息的方法是使用内核调试器命令*!devnode*。指定*0 1*作为命令选项就可以转储出内部设备树的devnode结构，缩进的格式显示了它们的层次关系，如下所示：

```
lkd> !devnode 0 1
Dumping IopRootDeviceNode (= 0x8a4b7ee8)
DevNode 0x8a4b7ee8 for PDO 0x8a4b7020
  InstancePath is "HTREE\ROOT\0"
  State = DeviceNodeStarted (0x308)
  Previous State = DeviceNodeEnumerateCompletion (0x30d)
  DevNode 0x8a4b7a50 for PDO 0x8a4b7b98
    InstancePath is "Root\ACPI_HAL\0000"
    State = DeviceNodeStarted (0x308)
    Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0x8a4af448 for PDO 0x8a4eb2c8
      InstancePath is "ACPI_HAL\PNP0C08\0"
      ServiceName is "ACPI"
      State = DeviceNodeStarted (0x308)
      Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x8a4af198 for PDO 0x8a4b1350
        InstancePath is "ACPI\GenuineIntel_-_x86_Family_6_Model_9\_0"
        ServiceName is "gv3"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
      DevNode 0x8a4e8008 for PDO 0x8a4a8950
        InstancePath is "ACPI\ThermalZone\THM_"
        State = DeviceNodeStarted (0x308)
        Previous State = DeviceNodeEnumerateCompletion (0x30d)
    DevNode 0x8a4e82b8 for PDO 0x8a4eb640
      InstancePath is "ACPI\ACPI0003\2&daba3ff&0"
      ServiceName is "CmBatt"
```

§

针对每个devnode所显示的信息包括InstancePath和ServiceName。这里InstancePath是该设备的列举注册表键（位于HKLM\SYSTEM\CurrentControlSet\Enum的下面）的名称，而ServiceName对应于该设备的驱动程序注册表键（位于HKLM\SYSTEM\CurrentControlSet\Services）。要想看到分配给每个devnode的资源，比如中断、端口和内存，你可以在*!devnode*命令中指定*0 3*作为命令选项。

关于“自从系统安装以来所有被检测到的设备”的纪录被存储在HKLM\SYSTEM\CurrentControlSet\Enum注册表键下。该键的子键都是<Enumerator>\<DeviceID>\<InstanceID>的形式。这里Enumerator是一个总线型驱动程序，DeviceID是一种设备类型的惟一标识符，InstanceID则惟一标识了同一硬件的不同实例。

www.TopSage.com

Devnode

图9.27显示了一个devnode是由以下的至少2个（有时候更多）设备对象构成的。

- 一个**物理设备对象**（PDO，*physical device object*） 当总线型驱动程序在列举过程中报告了在它的总线上存在一个设备时，PnP管理器指示该总线型驱动程序创建一个物理设备对象。PDO代表了该设备的物理接口。

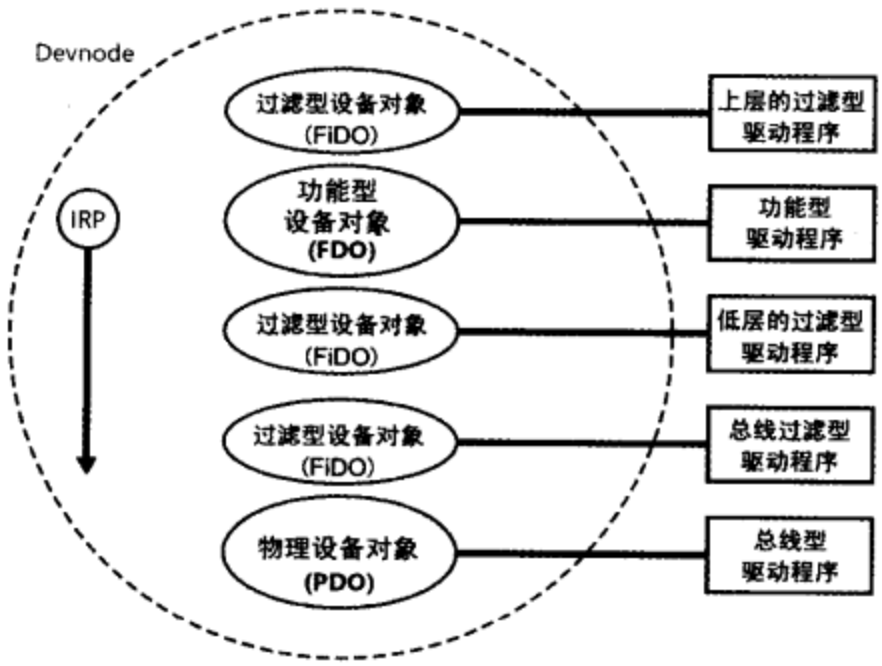


图9.27 Devnode的内部结构

- 在PDO和FDO（接下来讲述）之间，可以有一个或多个可选的过滤型设备对象（FiDO，filter device object），它们是由**总线过滤型驱动程序**（*bus filter driver*）创建的。
- 在PDO和FDO之间，在总线过滤型驱动程序创建的任何FiDO之上，可以有一个或多个可选的FiDO，它们是由**低层的过滤型驱动程序**（*lower-level filter driver*）创建的。
- 一个**功能型设备对象**（FDO，*functional device object*），由一个**功能型驱动程序**（*function driver*）来创建。PnP管理器通过加载此功能型驱动程序来管理一个被检测到的设备。FDO代表了一个设备的逻辑接口。如果多个设备被附载到这一由FDO代表的设备上，则该功能型驱动程序也担当起总线型驱动程序的角色。功能型驱动程序通常创建一个跟所对应的PDO之间的接口（如前面所描述），因而应用程序和其他的驱动程序可以打开该设备，并且与它进行交互。有时候，功能型驱动程序被分成单独的类/端口驱动程序和小端口驱动程序，它们联合起来管理该FDO的I/O。
- 在FDO之上，可以有一个或多个可选的FiDO，它们是由**上层的过滤型驱动程序**（*upper-level filter driver*）创建的。

www.TopSage.com

devnode是由底向上建立起来的，它依赖于I/O管理器的层次功能，所以，IRP从一个devnode的顶部向底部流动。然而，devnode中的任何一层都可以选择来完成一个IRP。例如，功能型驱动程序可以直接处理一个读请求，而不把IRP传递给总线型驱动程序。只有当功能型驱动程序要求得到总线型驱动程序的帮助，以便执行与总线相关的处理时，此IRP才会全程流动到底部，然后进入到包含此总线型驱动程序的devnode中。

Devnode的驱动程序加载

到现在为止，我们一直在避免回答两个重要的问题："针对一个特定的设备，PnP管理器如何确定要加载哪个功能型驱动程序？"以及"过滤型驱动程序如何注册它们自身，从而在创建一个devnode过程中它们会在适当的时候被加载到系统中？"

这两个问题的答案在于注册表中。当一个总线型驱动程序执行设备列举时，它将检测到的设备的标识符报告给PnP管理器。这些标识符是跟特定的总线相关的，对于USB总线，标识符是由制造该设备的硬件厂商的**厂商ID**（VID，*vendor ID*）和该厂商分配给该设备的**产品ID**（PID，*product ID*）组成的。（有关设备ID格式的更多信息，请参考DDK）。 这些ID合起来构成了即插即用过程中的**设备ID**（*device ID*）。PnP管理器也向总线型驱动程序询问一个**实例ID**（*instance ID*），以帮助它区分同一硬件的不同实例。此实例ID可以描述一个与总线相关的位置（比如USB端口），或者描述一个全局惟一的描述符（比如，一个序列号）。设备ID和实例ID组合起来构成了一个**设备实例ID**（DIID，*device instance ID*），PnP管理器利用此ID来定位到注册表的列举分支（HKLM\SYSTEM\CurrentControlSet\Enum）中该设备的键上。图9.28显示了一个键盘的列举子健的例子。该设备的键包含了描述性的数据，也包含了名为Service和ClassGUID的值（可以从一个驱动程序的INF文件中得到），它们可以帮助PnP管理器找到该设备的驱动程序。

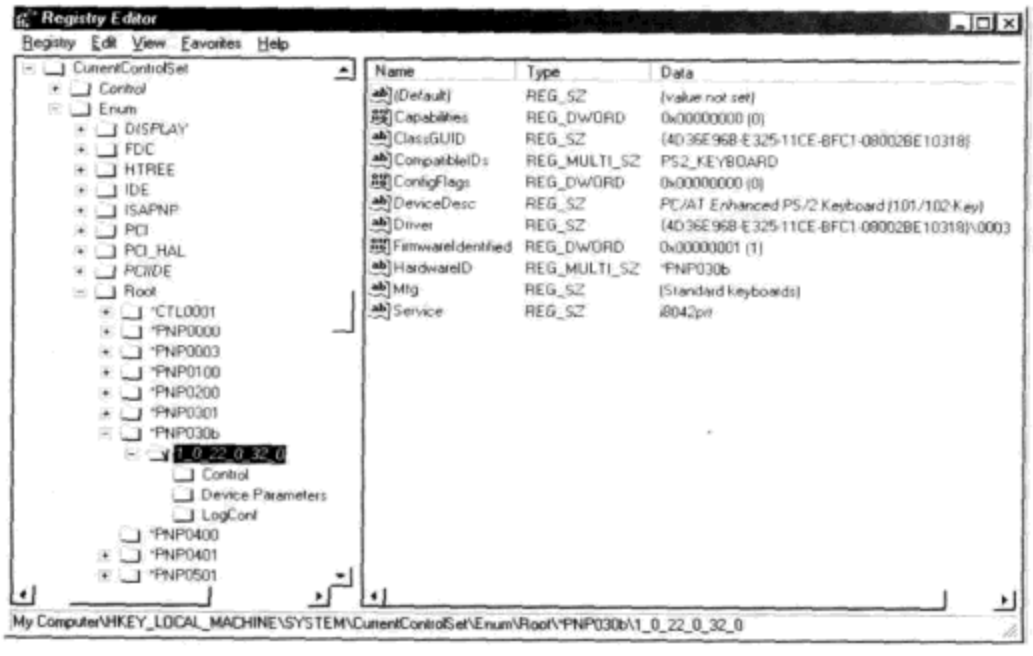


图9.28 键盘列举键

www.TopSage.com



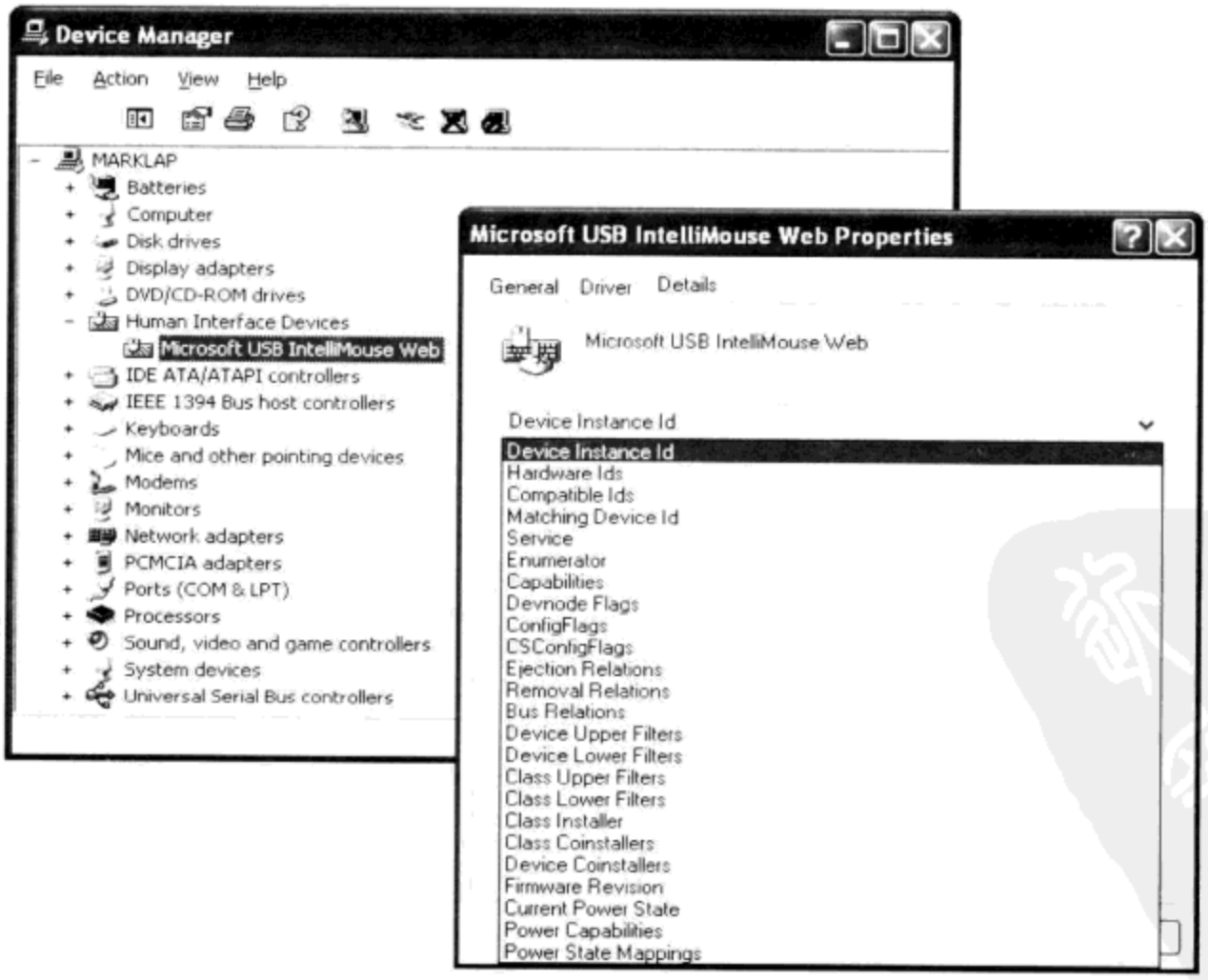
实验：在设备管理器中查看详细的Devnode信息

在默认情况下，设备管理器（Device Manager）小程序（从控制面板的System小程序的Hardware页面上你可以访问设备管理器）并不会显示有关一个设备节点的细节信息。然而，在Windows XP和Windows Server 2003中，你只要创建devmgr_show_details环境变量，并将其设置为1，就可以在设备管理器的节点信息界面上多增加一个称为*Details*的标签页面。该标签页面允许你查看各种域信息，包括devnode的设备实例ID、硬件ID、服务名称、过滤型驱动程序，以及电源能力。

要想启动带Details标签页的设备管理器，最简单的办法是，打开一个命令提示窗口，并且执行下面的命令：

```
C:\>set devmgr_show_details=1
C:\>devmgmt.msc
```

下面的屏幕截图显示了Details页面的选择组合框已被展开，给出了你可以访问的信息的类型：



www.TopSage.com

PnP管理器利用ClassGUID值，可以在注册表的HKLM\SYSTEM\CurrentControlSet\Control\Class下找到该设备的类键。键盘类键如图9.29所示。列举键和类键为PnP管理器提供了所需要的信息，以便让它为该设备的devnode加载必要的驱动程序。这些驱动程序按照下面的顺序来加载。

1. 在该设备列举键的LowerFilters值中指定的任何低层过滤型驱动程序加载。
2. 在该设备类键的LowerFilters值中指定的任何低层过滤型驱动程序加载。
3. 在该设备列举键的Service值中指定的功能型驱动程序加载。该值被解释为在HKLM\SYSTEM\CurrentControlSet\Services下面此驱动程序的键。
4. 在该设备列举键的UpperFilters值中指定的任何上层过滤型驱动程序加载。
5. 在该设备类键的UpperFilters值中指定的任何上层过滤型驱动程序加载。

在所有的情况下，这些驱动程序都是按照在HKLM\SYSTEM\CurrentControlSet\Services下面的键的名称来引用的。

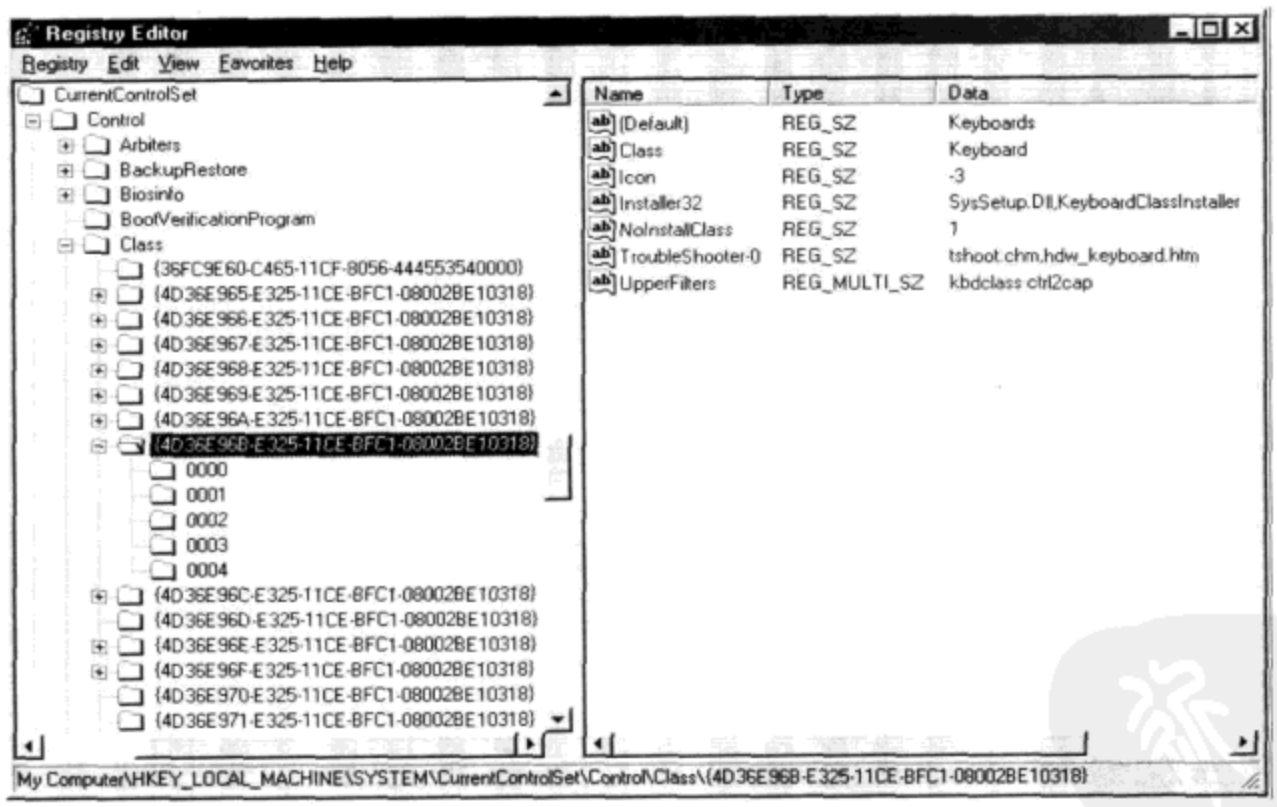


图9.29 键盘类键



> 注 Windows DDK将一个设备的列举键称为*硬件键*（*hardware key*），而将其类键称为*软件键*（*software key*）。

如图9.28和图9.29所显示的键盘设备没有低层过滤型驱动程序。其功能型驱动程序是i8042prt驱动程序，在键盘类键中指定了两个上层过滤型驱动程序：kbdclass和ctrl2cap。

www.TopSage.com

驱动程序安装

如果PnP管理器碰到一个设备，它还没有安装相应的驱动程序，那么，它依赖于用户模式的PnP管理器来指导其安装过程。如果该设备是在系统引导过程中被检测到的，那么，系统为该设备定义一个devnode，但是其加载过程被推迟到用户模式的PnP管理器启动之后再进行（用户模式的PnP管理器是在\Windows\System32\Umpnpmgr.dll中实现的，作为Services.exe进程中的一个服务来运行）。

在安装一个驱动程序时所涉及的组件如图9.30所示。图中的阴影对象对应于通常由系统提供的组件，而非阴影对象则包含在一个驱动程序的安装文件中。首先，总线型驱动程序利用一个DIID来通知PnP管理器它列举到了一个设备①。PnP管理器检查注册表，看是否存在对应的功能型驱动程序，如果它没有找到，则通知用户模式的PnP管理器发现新的设备，并且告诉它DIID②。用户模式PnP管理器首先试着在没有用户干预的情况下执行一次自动安装。如果此安装过程涉及要弹出对话框来执行用户交互操作，并且当前登录的用户具有管理员特权，那么，③用户模式PnP管理器激发Rundll32.exe应用程序（与宿纳控制面板实用工具的应用程序相同），以便执行硬件安装向导（\Windows\System32\Newdev.dll）。如果当前登录的用户没有管理员特权（或者没有用户登录到系统中），而且该设备的安装过程需要用户交互，那么，用户模式PnP管理器将此安装过程推迟到有一个特权用户登录进来再进行。硬件安装向导使用Setup和CfgMgr（配置管理器，Configuration Manager） API函数来找到与检测到的设备相兼容的驱动程序所对应的INF文件。这一过程可能会需要用户插入一片包含厂商提供的INF文件的安装介质，或者，安装向导可能会在.cab文件“\Windows\Driver Cache\i386\Driver.cab”中找到一个合适的INF文件，此cab（Cabinet）文件包含了随Windows一起发行的各种驱动程序。

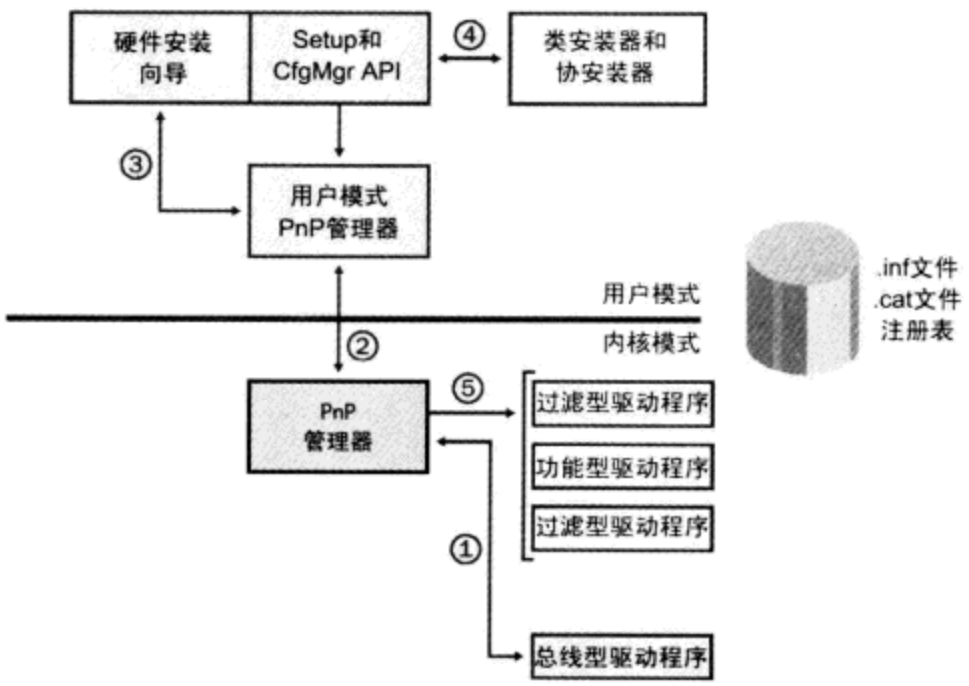


图9.30 驱动程序安装时所涉及的组件

www.TopSage.com

为了找到新设备的驱动程序，安装进程从总线型驱动程序中获得一个**硬件ID**（*hardware ID*）和**兼容ID**（*compatible ID*）的列表。这些ID描述了在一个驱动程序安装文件（.inf）中标识此硬件的各种可能方式。这些列表是排序的，因而，关于与该硬件最为相关的描述位于列表的前面。如果在多个INF文件中都找到了匹配，则较为精确的匹配优先于不那么精确的匹配；经过数字签名的INF优先于未经过签名的INF；较新的经过签名的INF优先于较老的经过签名的INF。如果某个匹配是根据一个兼容ID来找到的，则硬件安装向导可以选择提示用户输入介质位置，以防万一该硬件随带了一个更新的驱动程序。

INF文件指定了功能型驱动程序的文件的位置，它也包含了相应的命令以填充该驱动程序的列举键和类键；INF文件也可能指示硬件安装向导④启动类或设备的协安装器DLL，由它们来执行与特定的类或设备相关的安装步骤，比如显示配置对话框以便让用户为一个设备指定专门的设置信息。



实验：检查一个驱动程序的INF文件

当一个驱动程序或其他的带INF文件的软件被安装时，系统将它的INF文件拷贝到\Windows\Inf目录中。有一个文件，即Keyboard.inf，总是在这个目录下，因为它是键盘类驱动程序的INF文件。在Notepad中打开该文件就可以查看它的内容，你应该可以看到类似如下的信息：

```
; Copyright (c) 1993-1996, Microsoft Corporation

[version]
signature="$Windows NT$"
Class=Keyboard
ClassGUID={4D36E96B-E325-11CE-BFC1-08002BE10318}
Provider=%MS%
LayoutFile=layout.inf
DriverVer=07/01/2001,5.1.2600.1106

[ClassInstall32.NT]
AddReg=keyboard_class_addreg
…
```

如果你在该文件中搜索“.sys”，就会看到“指示用户模式PnP管理器安装i8042prt.sys和kbdclass.sys驱动程序”的内容：

```
…
[STANDARD_CopyFiles]
i8042prt.sys
kbdclass.sys
…
```


www.TopSage.com

用户模式PnP管理器在真正安装一个驱动程序以前，首先检查当前系统的驱动程序-签名策略。如果管理员指定了一个系统全局范围的策略，则该策略被存储在注册表键HKLM\SOFTWARE\Microsoft\Driver Signing\Policy下；如果只有针对每个用户的策略，则该策略被存储在HKCU\Software\Microsoft\Driver Signing\Policy键下。该策略是可以配置的，你可以通过控制面板的System工具的Hardware页面，打开Driver Signing Options对话框来配置该策略，如图9.31所示。如果该对话框中的设置指定了系统应该阻止或警告未签名驱动程序的安装，则用户模式PnP管理器就会在驱动程序的INF文件中查找一个指定了某个目录（catalog）文件（以.cat扩展名结尾，其中包含了该驱动程序的数字签名）的项目。

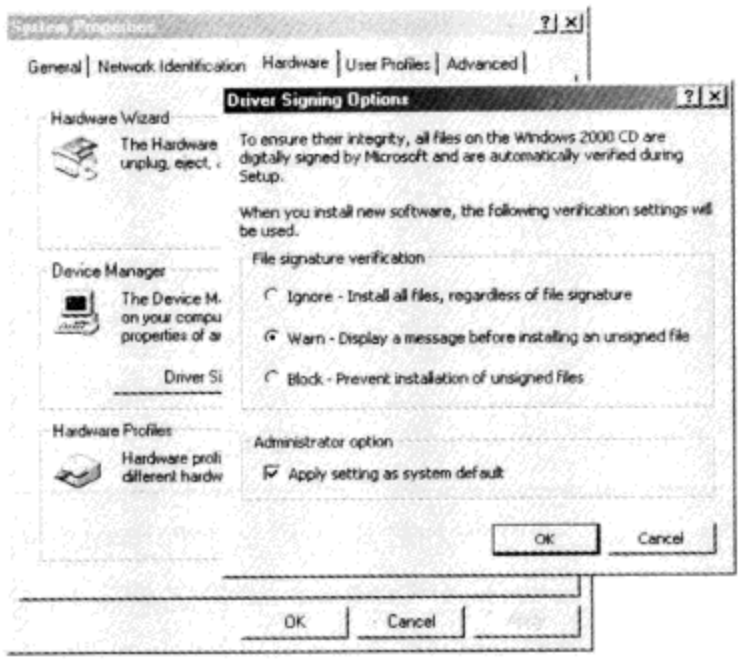


图9.31 驱动程序-签名策略选项

Microsoft的WHQL测试了Windows中包含的驱动程序，以及硬件厂商提交的驱动程序。当一个驱动程序通过了WHQL测试时，Microsoft就为它进行“签名”。这意味着，WHQL得到了一个代表该驱动程序中所有文件的**散列值**（hash）或惟一值，然后用Microsoft专门为驱动程序做签名的私钥，对此散列值进行密码学签名。经过签名的散列值或者被存放在一个目录文件中，放到Windows的安装介质上，或者被返回给提交此驱动程序的厂商，将来随驱动程序一起发行。



实验：查看目录（catalog）文件

当你安装一个内含目录文件的组件，比如驱动程序时，Windows将该目录文件拷贝到\Windows\System32\Catroot下的一个目录中。在Windows Explorer中，你跳转到此目录中，就可以找到包含.cat文件的子目录。例如，Nt5.cat和Nt5inf.cat保存了Windows系统文件的签名。

如果打开某一个目录文件，则会显示一个两页面的对话框。标记为General的页面显示了有关该目录文件中签名的信息，而Security Catalog页面则显示了那些由该目录文件来签名的组件的散列值。

www.TopSage.com

下面的屏幕截图是针对ATI视频驱动程序的目录文件，它显示了视频适配器的小端口驱动程序的散列值。该目录中的其他散列值则是与其他的“随该驱动程序一起发行的支持DLL”相关联的。



当用户模式PnP管理器在安装一个驱动程序时，它从驱动程序的目录文件中提取出它的签名，利用Microsoft的专用于驱动程序签名的公/私钥对中的公钥部分，将签名解密出来，然后将解密的结果散列值与它将要安装的驱动程序文件的散列值进行比较。如果两个散列值匹配，则该驱动程序通过检验，说明它确实通过了WHQL测试。如果一个驱动程序未能通过签名检验，则用户模式PnP管理器根据系统的驱动程序-签名策略来作出反应，或者提示安装失败，或者警告用户该驱动程序未被签名，或者继续安装该驱动程序。



> 注 如果驱动程序是由“手工配置注册表，并且将驱动程序文件直接拷贝到系统中”这样的安装程序（setup programs）来安装的，而且驱动程序文件是由应用程序动态加载的，那么，这样的驱动程序不执行签名检查。只有通过INF文件来安装的驱动程序才需要根据系统的驱动程序-签名策略进行验证。

在一个驱动程序被安装以后，内核模式的PnP管理器启动该驱动程序（图9.30中的步骤⑤），并且调用它的“增加-设备”例程，以便通知该驱动程序，它所负责管理的设备已出现在系统中。然后，devnode的构造继续进行，如前所述。



> 注 在Windows XP和Windows Server 2003上，用户模式PnP管理器还要检查：它将要安装的驱动程序是否在Windows Update维护的*受保护的驱动程序列表*（*protected driver list*）中。如果是的话，则停止安装过程，并且给用户一个警告：那些已知有不兼容性问题或者错误的驱动程序被加入到此列表中，因而在安装过程中被阻止。你可以在*www.microsoft.com/whdc/winlogo/drvsign/drv_protect.mspx*上找到更多的信息。

www.TopSage.com

9.5 电源管理器

正如Windows的即插即用特性要求一个系统的硬件必须提供相应的支持那样，Windows的电源管理能力也要求其硬件遵从ACPI（高级配置和电源接口，Advanced Configuration and Power Interface）规范（可以从*www.teleport.com/~acpi/spec.htm*获得）。此要求的一个直接结果是，计算机的BIOS（Basic Input Output System，基本输入输出系统），即计算机开机时运行的代码，也必须符合ACPI标准。自从1998年底以来生产的绝大多数x86计算机都是ACPI兼容的。



注 有些计算机，特别是好几年前的计算机，并不符合ACPI标准。相反地，它们通常遵从老的APM（高级电源管理，Advanced Power Management）标准，该标准要求的电源管理能力比ACPI要弱一些。Windows只为APM系统提供了很有限的电源管理，但是，在这里，我们不会探究此话题的细节。在本书中，我们将注意力集中于Windows在ACPI机器上的行为。

ACPI标准为一个系统和各种设备定义了各种电源级别。表9.3列出了六种系统电源状态。这些状态称为从S0（**完全打开或工作**）到S5（**完全关闭**）。每一种状态都有以下的特征：

- **电源消耗** 计算机消耗的电源数量
- **软件恢复** 当转移到“电源更加打开”的状态时，计算机可以恢复的软件状态
- **硬件延迟** 计算机返回到完全工作状态所需要的时间长度

表9.3 系统电源状态的定义

状 态	电源消耗	软件恢复	硬件延迟
S0（完全打开）	最大	不适用	无
S1（睡眠）	小于S0，大于S2	系统恢复到原来停止的地方（返回到S0）	小于2秒
S2（睡眠）	小于S1，大于S3	系统恢复到原来停止的地方（返回到S0）	大于等于2秒
S3（睡眠）	小于S2；处理器是关闭的	系统恢复到原来停止的地方（返回到S0）	与S2相同
S4（休眠）	只有电源按钮和唤醒电路才有点滴电流	系统从原来保存的休眠文件中重新启动，并且恢复到休眠之前停止的状态（恢复到S0）	很长，未定义
S5（完全关闭）	只有电源按钮才有点滴电流	系统引导	很长，未定义

www.TopSage.com

S1至S4之间的状态都是睡眠状态，计算机看起来是关闭的，因为电源消耗很低。然而，计算机仍然维持着足够的信息（既有内存中的，也有磁盘上的），以便转移到S0状态。对于状态S1至状态S3，要求有足够的电源以便维持计算机内存中的内容，因而当状态转移到S0时（当用户或某个设备唤醒了该计算机时），电源管理器可以从该计算机被挂起之前停止的地方继续执行。当系统状态转移到S4时，电源管理器将内存中的内容压缩以后，保存到一个名为Hiberfil.sys的休眠文件中，该文件位于系统卷的根目录中，它必须足够大以便容纳解压缩以后的内存内容。（之所以采用压缩，是因为要尽可能地减少磁盘I/O，以及提高休眠和“从休眠中恢复”的性能）。电源管理器在完成了内存保存操作以后，关闭计算机的电源。之后，当用户再打开计算机时，正常的引导过程继续进行，不过，Ntldr检查并检测是否在休眠文件中保存了一个有效的内存映像。如果该休眠文件包含了保存下来的系统状态，则Ntldr将文件的内容读入到内存中，然后根据在休眠文件中记录的值，在内存中恰当的点上恢复执行。

计算机永远不会直接在S1和S4之间进行状态转移，相反地，它必须首先转移到状态S0。如图9.32所示，当系统从S1至S5之间的任何一个状态转移到状态S0时，它被称为**唤醒**（*waking*）；当它从状态S0转变到S1至S5之间的任何一个状态时，它被称为**睡眠**（*sleeping*）。

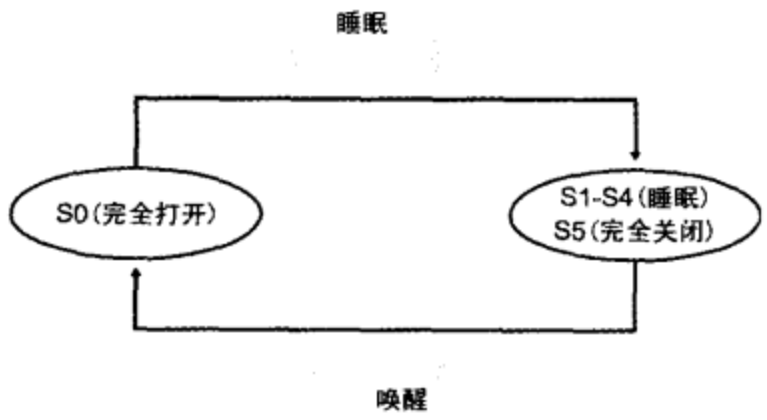


图9.32 系统的电源状态转移

虽然系统可以处于六种电源状态之一，但是ACPI规定，设备只能处于4种电源状态之一，从D0至D3。状态D0是**完全打开**，状态D3是**完全关闭**。ACPI标准将状态D1和D2的含义留给单独的驱动程序和设备来定义，它只是要求：状态D1所消耗的电源数量少于或等于状态D0所消耗的电源数量，而且当设备处于状态D2中时，它消耗的电源必须少于或等于在状态D1中消耗的电源数量。Microsoft连同一些大的硬件OEM（原始设备制造商），已经定义了一系列电源管理参考规范（可以通过Microsoft Web站点获得，位于*www.microsoft.com/whdc/resources/respec/specs/pmref*），这些规范指定了对于某一特定类别（大的设备类别有显示器、网络、SCSI，等等）的所有设备都要求支持的设备电源状态。对于某些设备，在完全打开和完全关闭之间没有中间电源状态，因而这些状态没有被定义。

www.TopSage.com

电源管理器的操作

Windows中的电源管理策略被分成两部分：一部分针对电源管理器；另一部分针对单独的设备驱动程序。电源管理器是系统电源策略的所有者。这一所有关系意味着，在任何给定的点上，由电源管理器来决定哪一种系统电源状态是恰当的，而且，当接到睡眠、休眠或者停机请求时，电源管理器指示该系统中具有电源能力的设备执行适当的系统电源状态转移。电源管理器在决定何时有必要执行一次系统电源状态转移时，考虑以下诸多因素：

- 系统的活动级别；
- 系统的电池级别；
- 来自应用程序的停机、休眠或者睡眠请求；
- 用户的动作，比如按下了电源按钮；
- 控制面板的电源设置。

当PnP管理器执行设备列举时，在它接收到的有关一个设备的信息中，有一部分是关于其电源管理能力的信息。驱动程序向PnP管理器报告，它的设备是否支持设备状态D1和D2，它也可以有选择地报告其设备从状态D1至D3转移到D0的延迟或所需时间。为了帮助电源管理器决定何时执行系统电源状态转移，总线型驱动程序也会返回一张表，该表实现了从每一种系统电源状态（S0至S5）到该设备所支持的电源状态之间的映射。该表针对每一种系统状态列出了最低可能的设备电源状态，它直接反映了当机器睡眠或休眠时的各种电源等级状态。例如，一条支持所有4种设备电源状态的总线可能返回如表9.4所示的映射表。绝大多数的设备驱动程序在离开S0时都将它们的设备完全关闭（D3），以便当机器不在使用时电源的消耗最小化。然而，有些设备，比如网络适配器卡，支持“从睡眠状态中唤醒系统”的能力。这种能力，连同该能力所处的最低设备电源状态，也是在设备列举过程中被报告的。

表9.4 从系统电源到设备电源之间映射关系的一个例子

系统电源状态	设备电源状态
S0 （完全打开）	D0 （完全打开）
S1 （睡眠）	D2
S2 （睡眠）	D2
S3 （睡眠）	D2
S4 （休眠）	D3 （完全关闭）
S5 （完全关闭）	D3 （完全关闭）

www.TopSage.com

驱动程序的电源操作

当电源管理器决定要在系统电源状态之间进行转移时，它向驱动程序的电源分发例程发送电源命令。虽然可以有多个驱动程序负责管理一个设备，但是在这些驱动程序之中，只有一个驱动程序被指定为该设备的电源-策略所有者。该驱动程序根据系统的状态来决定一个设备的电源状态。例如，如果系统在状态S0和S1之间进行转移，则驱动程序可能决定将一个设备的状态从D0转移到D1。一个设备的电源-策略所有者并不是直接通知其他那些共享同一设备管理任务的驱动程序，而是通过*PoRequestPowerIrp*函数，请电源管理器向其他驱动程序的电源分发例程发出一个设备电源命令，以告知这些驱动程序。这种做法使得电源管理器在任何给定的时刻，都可以控制一个系统中当前有效的电源命令的数量。例如，系统中的有些设备可能要求相当数量的电流才能供电启动（power up）。电源管理器确保这样的设备不会被并发地供电启动。



实验：查看一个驱动程序的电源映射关系

在Windows XP和Windows Server 2003中，通过设备管理器你可以看到一个驱动程序的“从系统电源状态到驱动程序电源状态”之间的映射关系。打开一个设备的Properties对话框，并且从Details页面的下拉列表中选择Power State Mappings，就可以看到这一映射关系。（在默认情况下，设备管理器不显示Details标签页。要想知道如何打开此页面，请参见本章前面的实验“在设备管理器中查看详细的Devnode信息”。）

下面的磁盘驱动程序的映射关系表明，除了完全打开（D0）和完全关闭（D3）以外，它还为S1支持中间状态D1。这可能代表磁盘停止旋转（disk spin-down）的电源状态。



www.TopSage.com

许多电源命令有对应的查询命令。例如，当系统转移到一种睡眠状态时，电源管理器首先询问该系统上的设备，这种状态转移是否可以接受。如果一个设备正在忙着执行紧急的操作，或者正在与设备硬件打交道，那么，它可能会拒绝该命令，因而导致系统维持其当前的系统电源状态设置。



实验：查看系统的电源能力和策略

利用内核调试器命令*!pocaps*，你可以查看一台计算机的系统电源能力。下面是在一台运行Windows Professional的ACPI兼容的便携式电脑上运行该命令时得到的输出：

```
kd> !pocaps
PopCapabilities @ 0x8046adc0
  Misc Supported Features:  PwrButton SlpButton Lid S1 S3 S4 S5
                            HiberFile FullWake
  Processor Features:       Thermal Throttle (MinThrottle = 03, Scale = 08)
  Disk Features:            SpinDown
  Battery Features:         BatteriesPresent
    Battery 0 - Capacity: 00000000  Granularity: 00000000
    Battery 1 - Capacity: 00000000  Granularity: 00000000
    Battery 2 - Capacity: 00000000  Granularity: 00000000
  Wake Caps
    Ac OnLine Wake:         Sx
    Soft Lid Wake:          Sx
    RTC Wake:               S3
    Min Device Wake:        Sx
    Default Wake:           Sx
```

“Misc Supported Features”这一行报告了除了S0（完全打开）以外，该系统还支持系统电源状态S1、S3、S4和S5（它没有实现S2），并且有一个有效的休眠文件（所以，当它休眠时[状态S4]，它可以把系统内存保存在此休眠文件中）。

下一页显示的“Power Options Properties”对话框（在控制面板中选择Power Options就可以打开此对话框）允许你配置当前系统的电源策略的各个方面。到底你能够配置哪些属性，要取决于系统的电源能力，上面我们刚刚检查过。

在一台ACPI兼容的便携式电脑上，Windows Professional和Home系统（比如我们捕捉以下屏幕截图所使用的系统）通常提供了绝大多数电源管理特性。在这样的系统上，你可以设置各种空闲检测超时间隔，这些值控制了系统何时关闭监视器、何时停止磁盘旋转、何时进入到待机模式（转移到系统电源状态S1），以及何时休眠（系统转移到电源状态S4）。而且，Power Options中的Advanced页面允许你指定，当你按下电源按钮或睡眠按钮，或者合上一台便携式电脑的盖子时该系统中的一些与电源相关的行为。

www.TopSage.com



你在Power Options中配置的这些设置直接影响了系统的电源策略中的值。通过调试器命令*!popolicy*你可以显示这些值。下面是该命令在同一个系统上的输出：

```
kd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x80469180
  PowerButton:          Off  Flags: 00000003   Event: 00000000   Query UI
  SleepButton:        Sleep  Flags: 00000003   Event: 00000000   Query UI
  LidClose:       Hibernate  Flags: 00000001   Event: 00000000   Query
  Idle:                None  Flags: 00000001   Event: 00000000   Query
  OverThrottled:      Sleep  Flags: c0000004   Event: 00000000   Override
                    NoWakes Critical
  IdleTimeout:     00000000  IdleSensitivity:        32
  MinSleep:              S1  MaxSleep:              S3
  LidOpenWake:           S0  FastSleep:             S1
  WinLogonFlags:   00000000  S4Timeout:         00000000
  VideoTimeout:    00000000  VideoDim:              6e
  SpinTimeout:     00000708  OptForPower:           01
  FanTolerance:          64  ForcedThrottle:        64
  MinThrottle:           19
```

以上显示的前面几行对应于在Power Options的Advanced页面上指定的按钮行为，在这个系统上，电源开关被解释为关机开关，睡眠按钮将系统转移至一种睡眠状态，而合上便携式电脑的盖子则导致系统休眠。

在以上输出的末尾显示的超时值是按秒来解释的，它们被显示成十六进制格式。这里报告的超时值直接对应于你在Power Options屏幕截图中看到的设置（该便携式电脑是插上电源的）。例如，视频超时值（VideoTimeout）为0，意味着监视器永远不关闭；硬盘停止旋转的超时值（SpinTimeout）为0x708，该值对应于1800秒，或者30分钟。

www.TopSage.com

驱动程序对于设备电源的控制

驱动程序除了响应与系统电源状态转移有关的电源管理器命令以外，还可以单方面控制其设备的电源状态。在有些情形下，当一个驱动程序所控制的设备已经有一段时间不活动时，此驱动程序可能想要降低该设备的电源消耗。这样的例子有，支持暗淡模式（dimmed mode）的监视器和支持停止旋转（spin-down）的磁盘。驱动程序既可以自己检测到空闲的设备，也可以利用电源管理器提供的设施。如果该设备使用了电源管理器，那么，驱动程序通过调用*PoRegisterDeviceForIdleDetection*函数就可以向电源管理器登记该设备。此函数告诉电源管理器，用于检测一个设备为空闲的超时值是多少，以及当电源管理器检测到该设备为空闲时应该使用哪一种设备电源状态。驱动程序指定两个超时值：一个在用户已经配置该计算机为节能模式时使用；另一个在用户已经配置该计算机为最优性能模式时使用。驱动程序在调用了*PoRegisterDeviceForIdleDetection*以后，无论何时当设备活动时，它必须调用*PoSetDeviceBusy*函数来通知电源管理器。

9.6 本章总结

I/O系统定义了Windows平台上I/O处理的模型，它完成一些公共的功能，或者说对于多个驱动程序都需要的功能。它的首要任务是创建IRP来响应I/O请求，并且负责在各个驱动程序之间传递这些请求包，然后当I/O完成时将结果返回给调用者。I/O管理器利用I/O系统对象（包括驱动程序对象和设备对象）来找到各种驱动程序和设备。在内部，Windows I/O系统是以异步方式来工作的，以便获得尽可能高的性能，但是，它为用户模式应用程序提供了同步和异步两种I/O能力。

设备驱动程序不仅包括传统的硬件设备驱动程序，而且也包括文件系统、网络和分层的过滤型驱动程序。所有的驱动程序都有一个共同的结构，它们使用共同的机制与I/O管理器进行通信，或者相互之间进行通信。I/O系统的这种接口模型使得这些驱动程序可以用高级语言来编写，因而缩短开发时间和增强它们的移植性。因为所有的驱动程序展示给操作系统的是共同的结构，所以，它们可以彼此之间按层次叠加起来，从而实现模块化能力和减少驱动程序之间的重复工作。而且，所有的Windows设备驱动程序都应该被设计成可以在多处理器系统上正确地工作。

最后，PnP管理器的角色是：与设备驱动程序一起工作以便动态地检测硬件设备；建立一棵内部树，由这棵树来指导硬件设备列举和驱动程序安装。电源管理器与设备驱动程序一起工作，以便当需要节约能源和延长电池使用寿命时可以将设备转移到低电源消耗状态。

接下来的四章涉及与I/O系统相关的其他一些话题：存储管理、文件系统（包括NTFS文件系统的细节）、缓存管理器和网络。

www.TopSage.com

www.TopSage.com

Storage Management

第10章 存储管理

存储管理定义了一种操作系统与非易失性存储设备和介质打交道的方式。术语“**存储**（*storage*）”涵盖许多种不同的设备，包括磁带驱动器、光学介质、CD唱机、软盘、本地硬盘和存储区域网络（SAN，storage area network）。Microsoft Windows为这些存储介质中的每一种类型都提供了专门的支持。因为在本书中我们的焦点集中在Windows的内核组件上，所以，在本章中，我们将仅仅关注于Windows中硬盘存储子系统的基础。Windows对于可移除介质和远程存储体（离线存档）的支持的主体部分是在用户模式中实现的。

在本章中，我们将检查内核模式设备驱动程序如何将文件系统驱动程序与磁盘介质连接起来，讨论磁盘是如何被分区的，讲述卷管理器如何抽象和管理多个卷，介绍Windows中多分区磁盘管理特性是如何实现的，包括文件系统的数据如何被复制和划分到多个物理磁盘中以实现可靠性和性能增强。最后，我们将讲述文件系统驱动程序如何挂载（mount）它们所负责管理的那些卷。

10.1 有关存储的术语

为了完全理解本章剩余部分的内容，你需要熟悉以下一些基本的术语。

- **磁盘**（*disk*）是一个物理的存储设备，比如硬盘、3.5英寸软盘，或者CD-ROM；
- 一个磁盘被分成**扇区**（*sector*），这些扇区是一些固定大小的、可寻址的存储块。扇区的大小是由硬件来决定的。大多数硬盘的扇区是512字节，CD-ROM的扇区通常是2048字节；
- **分区**（*partition*）是磁盘上一组连续扇区的集合。分区表或者其他的磁盘管理数据库保存了一个分区的起始扇区、分区的大小和其他的特征，它与该分区处在同一个磁盘上；
- **简单卷**（*simple volume*）是指这样的对象：它描述了单个分区中的扇区，文件系统驱动程序将它当作一个单元来管理；
- **多分区卷**（*multipartition volume*）是指这样的对象：它描述了多个分区中的扇区，文件系统驱动程序将它当作一个单元来管理。多分区卷提供了性能、可靠性和尺寸变化方面的特性，这是简单卷无法提供的。

www.TopSage.com

10.2 磁盘驱动程序

在管理一个特定存储设备的过程中所涉及的设备驱动程序，合起来称为一个存储栈。图10.1显示了可能出现在一个栈中的每一种驱动程序类型，其中也包括了有关其用途的简短描述。本章讲述了在存储栈中位于文件系统层以下的设备驱动程序的行为（第12章讲述了文件系统驱动程序的操作）。

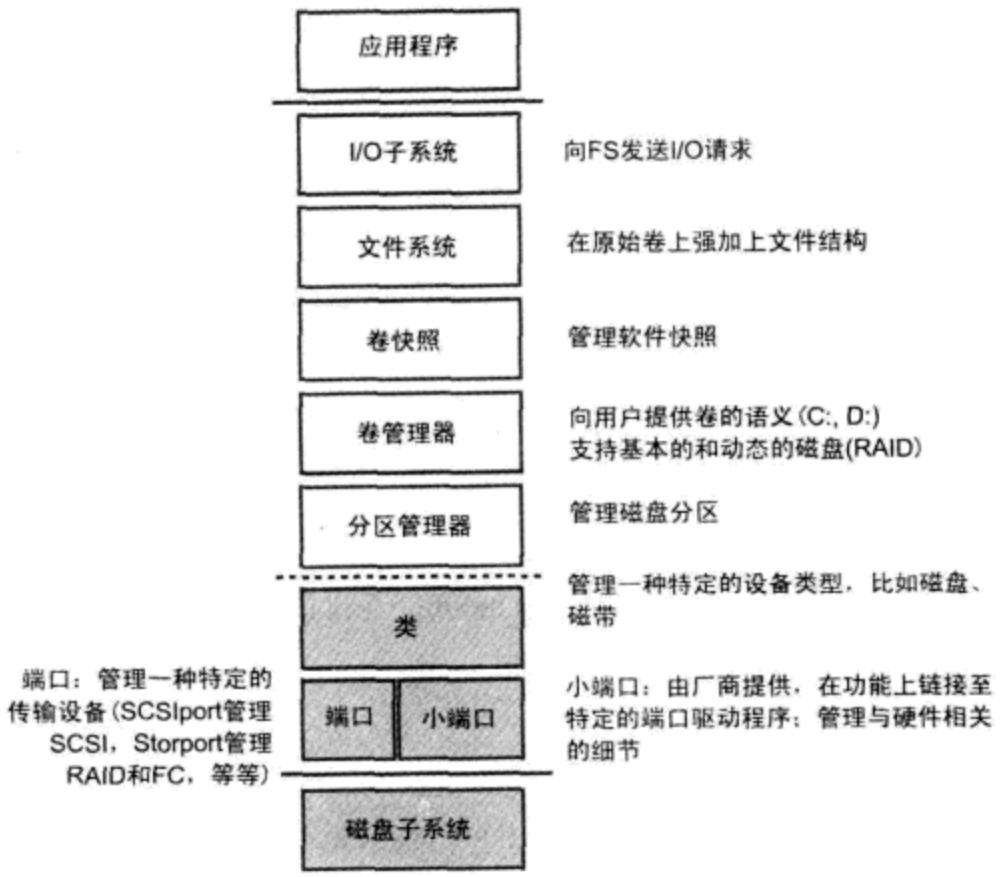


图10.1 Windows存储栈

Ntldr

正如你在第5章中看到的那样，Ntldr是一个Windows操作系统文件，它控制了Windows引导过程的最初部分。虽然从技术上讲，Ntldr并不是存储栈的一部分，但是，它也涉及了存储管理，因为在Windows的I/O系统起作用以前，是它提供了访问磁盘设备的支持。Ntldr驻留在系统卷上；系统卷上的引导扇区代码执行Ntldr。Ntldr从系统卷中读入Boot.ini文件，将该计算机的引导选择项展示给用户。Boot.ini指定的分区名称是按照multi(0)disk(0)rdisk(0)partition(1)的形式来指定的。这些名称是ARC（Advanced RISC Computing）名称，因为它们是Alpha固件和其他RISC处理器使用的标准分区命名方案的一部分。Ntldr将用户选择的Boot.ini引导项的名称转译成正确的引导分区，然后加载Windows的系统文件（从注册表、Ntoskrnl.exe和引导驱动程序开始）到内存中，以继续此引导过程。在所有情况下，Ntldr都使用计算机固件来读取系统

www.TopSage.com

卷所在的磁盘，但是，正如第5章中所描述的那样，有时候，Ntldr依赖于一个磁盘小端口驱动程序中的函数来从引导卷的磁盘中读取数据。

磁盘类、端口和小端口驱动程序

在初始化过程中，Windows I/O管理器启动硬盘的存储驱动程序。Windows中的存储驱动程序符合“类/端口/小端口”结构：Microsoft提供一个存储类驱动程序，它实现了对于所有的存储设备都共同的功能；Microsoft也提供一个存储端口驱动程序，它实现了对于特定的总线而言共同的功能，比如SCSI（Small Computer System Interface）总线或者IDE（Integrated Device Electronics）系统；OEM厂商提供小端口驱动程序，它们插入到端口驱动程序中，将Windows与特定的实现连接起来。

在磁盘的存储驱动程序体系结构中，只有类驱动程序符合标准的Windows设备驱动程序的接口。小端口驱动程序使用端口驱动程序的接口，而不使用设备驱动程序的接口，而端口驱动程序只是简单地实现了一组设备驱动程序支持例程，由这组例程将小端口驱动程序与Windows连接起来。这种做法简化了小端口驱动程序开发人员的任务，因为Microsoft提供了与操作系统相关的端口驱动程序，因而赋予了小端口驱动程序在Microsoft Windows 98、Microsoft Windows Millennium Edition和Windows这些操作系统之间的二进制移植性。

Windows包含了Disk驱动程序（\Windows\System32\Drivers\Disk.sys），这是一个类驱动程序，它实现了对于所有磁盘都共同的功能。Windows也提供了许多磁盘端口驱动程序。例如，Scsiport是针对SCSI总线上的磁盘的端口驱动程序，Atapi.sys是针对IDE系统的端口驱动程序。Windows Server 2003还引入了Storport.sys端口驱动程序，并将它作为Scsiport.sys的一个替代。Storport.sys的设计目标是，实现硬件RAID和光纤通道（Fibre Channel）适配器的高性能。Storport的模型类似于Scsiport，这使得厂商很容易将已有的SCSI小端口驱动程序移植到Storport上。开发人员编写的使用Storport的小端口驱动程序可以充分发挥Storport的几项性能增强特性，包括对于在多处理器系统上并行地执行“I/O激发请求”和“完成请求”的支持，多个可控制的I/O请求队列的结构，以及在更低的IRQL上执行多份代码以便使硬件中断的屏蔽时间最小化。

Scsiport.sys和Atapi.sys驱动程序都实现了一个磁盘调度算法的版本，称为C-LOOK。这两个驱动程序把磁盘I/O请求放到排序的列表中，这些列表按照每个I/O请求所指的第一个扇区（也称为逻辑块号或LBN[Logical Block Number]）进行排序。它们使用KeInsertByKeyDeviceQueue和KeRemoveByKeyDeviceQueue函数（在Windows DDK中有文档介绍），将I/O请求表示成队列中的项目，并使用每个请求的起始扇区作为这些函数所要求的键。当驱动程序处理这些请求时，它们从最低的扇区向最高的扇区遍历整个列表。当它们到达列表的末尾时，驱动程序又回到起始处重新开始，因为在此过程中新的请求可能又被插进来了。如果磁盘请求被散布在

www.TopSage.com

整个磁盘空间中，则这种做法可以让磁头连续地从接近于磁盘起始处的地方向磁盘的末尾移动。Storport.sys并没有实现磁盘调度，因为它通常被用来管理针对存储阵列的I/O，在这种情形下，没有明确定义的磁盘起始和末尾的概念。

Windows随带发行了几个小端口驱动程序，其中包括Aha154x.sys，这是针对Adaptec的1540 SCSI控制器族的小端口驱动程序。在至少有一个基于ATAPI的IDE设备的系统上，Pciidex.sys和Pciide.sys合起来提供小端口的功能。绝大多数安装的Windows系统都包含这里提到的一个或者多个驱动程序。

iSCSI驱动程序

iSCSI是作为一个磁盘传输协议来开发的，它集成了SCSI协议与TCP/IP网络的能力，所以，计算机可以通过IP网络，与块存储设备（包括磁盘）进行通信。存储区域网络（SAN）通常是建立在光纤通道（Fibre Channel）网络连接的基础之上的，但管理员也可以利用iSCSI，通过诸如千兆以太网这样的网络技术来创建相对不昂贵的SAN，以提供可伸缩性、灾难保护、高效的备份，以及数据保护。Windows对于iSCSI的支持是以Microsoft iSCSI软件发起器（Software Initiator）的形式来提供的，你可以从Microsoft的Web站点下载到iSCSI软件发起器，它可以在Windows 2000、Windows XP和Windows Server 2003上工作。

Microsoft iSCSI软件发起器包括以下几个组件。

- **发起器**（Initiator） 这是一个可选的组件，它由一个iSCSI端口驱动程序（\Windows\System32\Drivers\Iscsiprt.sys）和一个小端口驱动程序（\Windows\System32\Drivers\Msiscis.sys）构成，它使用TCP/IP驱动程序来实现基于标准以太网适配器和TCP/IP数据外送（offloaded）网络适配器的软件iSCSI；
- **发起器服务**（Initiator service） 此服务是在\Windows\System32\Iscsiexe.exe中实现的，它管理所有iSCSI发起器的发现和安全性，以及会话初始化和终止。iSCSI设备发现功能是在\Windows\System32\Iscsium.dll中实现的，符合iSNS（Internet Storage Name Server，Internet存储名称服务）协议；
- **管理应用程序** 这包括Iscsicli.exe和控制面板中对应的应用程序。Iscsicli.exe是一个命令行工具，它管理iSCSI设备的连接和安全性。

有些厂商生产的iSCSI适配器将iSCSI协议下移到硬件上。发起器服务可以与这些适配器一起工作，但它们必须支持iSNS。因此，所有的iSCSI设备，包括由发起器服务发现的设备和由iSCSI硬件发现的设备，都可以通过标准的Windows界面来识别和管理。

www.TopSage.com

多路径I/O（MPIO）驱动程序

大多数磁盘设备在它们和计算机之间有一条**路径**（*path*）或者由适配器、线缆和交换机构成的一个序列。要求高级别可用性的服务器使用**多路径**（*multipathing*）方案，也就是说，在计算机和磁盘之间存在不止一组连接硬件，所以，如果一条路径失败了，系统通过另一条路径仍然可以访问该磁盘。然而，如果没有操作系统或者磁盘驱动程序的支持，具有两条路径的磁盘会被看成两个不同的磁盘。Windows包含了多路径I/O的支持，以便把多路径磁盘按照单个磁盘来管理，但是它依赖于被称为DSM（*device-specific module*，**与设备相关的模块**）的第三方驱动程序来管理有关路径管理的细节，比如负载平衡策略（选择哪条路径来转发请求）和错误检测机制（当一条路径失败时通知Windows）。在Windows 2000 Server、Advanced Server和Datacenter Server，以及Windows Server 2003中，对MPIO的支持是以Microsoft MPIO Driver Development Kit的形式来提供的，软硬件厂商可以申请许可权。

在一个Windows MPIO存储栈中，如图10.2所示，多路径磁盘驱动程序的版本（\Windows\System32\Drivers\Mpdev.sys）取代了标准的Disk.sys类驱动程序的功能。Mpdev.sys负责宣称那些代表多路径磁盘的设备对象的所有权，这样它可以确保只创建一个设备对象来表示这些磁盘；它也负责找到适当的DSM来管理那些通向该设备的路径。多路径总线驱动程序（\Windows\System32\Drivers\Mpio.sys）管理计算机和设备之间的连接，包括该设备的电源管理。Mpdev.sys将该设备的存在告诉Mpio.sys，以便让它进行管理。最后，多路径端口过滤器驱动程序（\Windows\System32\Drivers\Mpsfltr.sys）叠加在一个多路径磁盘的端口驱动程序之上，以管理所有通过此设备栈传递的信息。

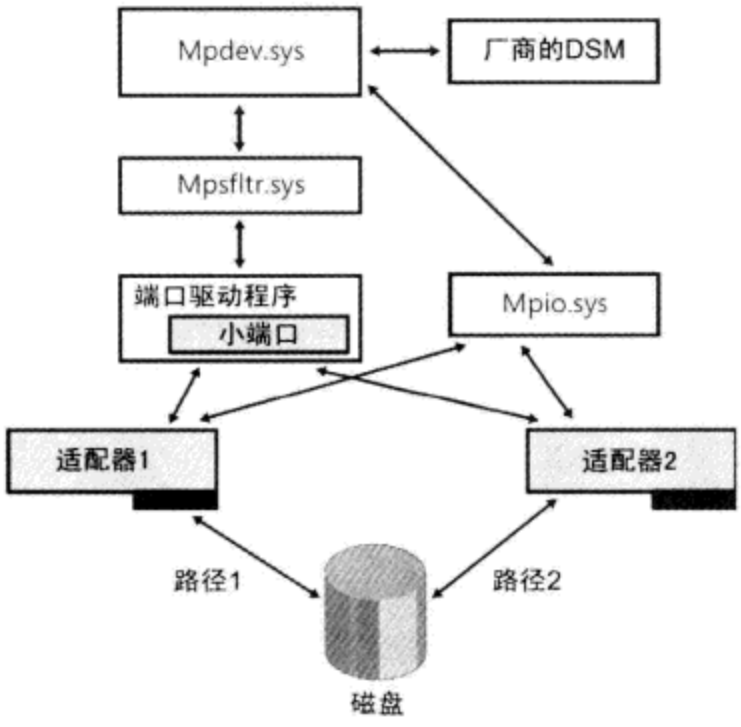


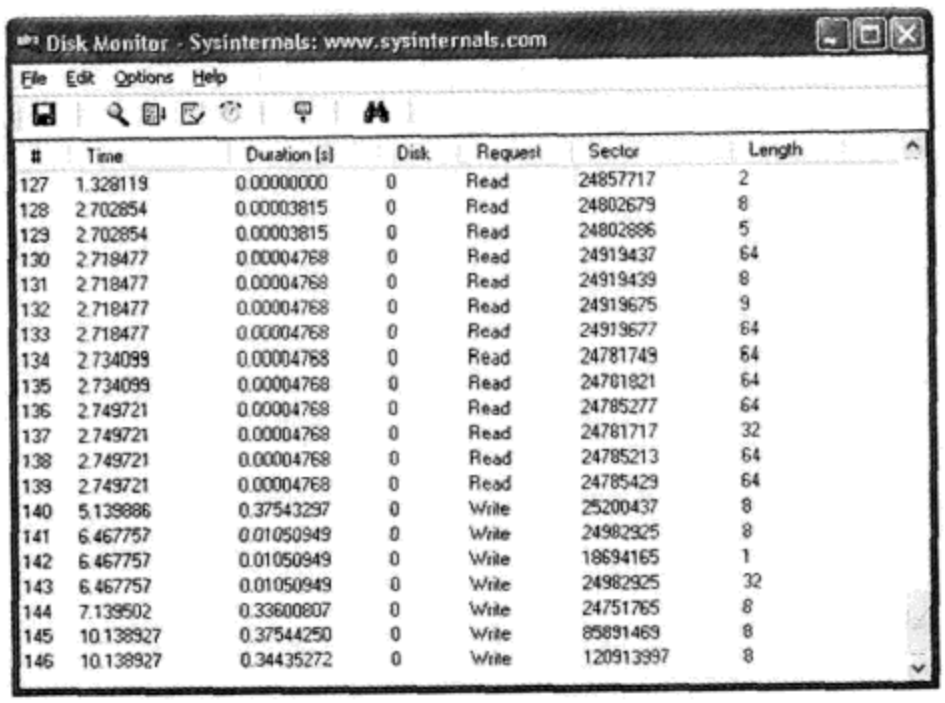
图10.2 Windows MPIO存储栈

www.TopSage.com



实验：观察物理磁盘I/O

Sysinternals的Diskmon使用磁盘类驱动程序的ETW（Event Tracing for Windows，第3章中介绍）设施来监视物理磁盘的I/O活动，并且将这些活动显示在一个窗口中。Diskmon每一秒钟更新一次新数据。对于每个操作，Diskmon显示了时间、持续的时间长度、目标磁盘号、类型和偏移，以及长度。



#	Time	Duration (s)	Disk	Request	Sector	Length
127	1.328119	0.00000000	0	Read	24857717	2
128	2.702854	0.00003815	0	Read	24802679	8
129	2.702854	0.00003815	0	Read	24802886	5
130	2.718477	0.00004768	0	Read	24919437	64
131	2.718477	0.00004768	0	Read	24919439	8
132	2.718477	0.00004768	0	Read	24919675	9
133	2.718477	0.00004768	0	Read	24919677	64
134	2.734099	0.00004768	0	Read	24781749	64
135	2.734099	0.00004768	0	Read	24781821	64
136	2.749721	0.00004768	0	Read	24785277	64
137	2.749721	0.00004768	0	Read	24781717	32
138	2.749721	0.00004768	0	Read	24785213	64
139	2.749721	0.00004768	0	Read	24785429	64
140	5.139886	0.37543297	0	Write	25200437	8
141	6.467757	0.01050949	0	Write	24982925	8
142	6.467757	0.01050949	0	Write	18694165	1
143	6.467757	0.01050949	0	Write	24982925	32
144	7.139502	0.33600807	0	Write	24751765	8
145	10.138927	0.37544250	0	Write	85891469	8
146	10.138927	0.34435272	0	Write	120913997	8

磁盘设备对象

Windows磁盘类驱动程序创建了代表磁盘和分区的设备对象。代表磁盘的设备对象具有形如\Device\ HarddiskX\DRX的名称；名称中的两个X由标识特定磁盘的编号来替代。磁盘类驱动程序使用I/O管理器的*IoReadPartitionTable*函数（在Windows 2000中）或*IoReadParititionTableEx*函数（在Windows XP和Windows Server 2003中）来标识分区，并创建代表这些分区的设备对象。当小端口驱动程序向磁盘类驱动程序报告它们在引导过程早期标识出来的磁盘时，磁盘类驱动程序为每个磁盘调用*IoReadPartitionTable*或*IoReadPartitionTableEx*函数。这些函数又调用类、端口和小端口驱动程序提供的扇区级别的磁盘I/O，以读取一个磁盘的MBR风格或GPT风格的分区表（本章后面介绍），并且为该磁盘的硬分区构造一个内部表示形式。磁盘类驱动程序创建一些设备对象来表示它从*IoReadPartitionTable*或*IoReadPartitionTableEx*调用中得到的每一个主分区（包括扩展分区内部的逻辑驱动器）。下面是一个示例性的分区对象名称：

\Device\Harddisk0\DP(1)0x7e000-0x7ff50c00+2

www.TopSage.com

该名称标识了一个系统的第一个磁盘上的第一个分区。前两个十六进制数值（0x7e000和0x7ff50c00）指定了该分区的起始位置和长度，最后的数字是一个内部标识符，是类驱动程序分配给该分区的。

为了保持与那些仍然使用Windows NT 4命名规范的应用程序的兼容性，磁盘类驱动程序创建了Windows NT 4格式命名的符号链接，让这些符号链接指向此驱动程序创建的设备对象。例如，Disk类驱动程序创建的符号链接\Device\Harddisk0\Partition0指向\Device\Harddisk0\DR0，以及\Device\Harddisk0\Partition1指向第一个磁盘的第一个分区设备对象。磁盘类驱动程序也创建了与“Windows NT 4系统中代表物理驱动器的符号链接”相同的Windows符号链接。因此，举例来说，链接\??\PhysicalDrive0代表了\Device\Harddisk0\DR0。在图10.3中，Sysinternals的Winobj工具显示了一个基本磁盘的Harddisk目录的内容。你可以在右侧的窗格中看到物理磁盘和分区的设备对象。

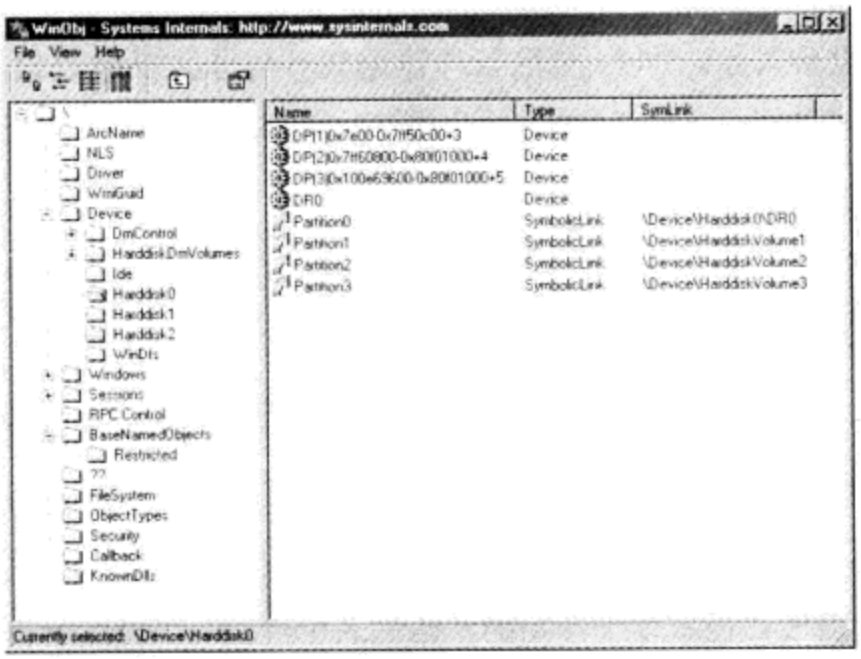


图10.3 Winobj显示了一个基本磁盘的Harddisk目录

正如你在第3章中所看到的，Windows API并不知道Windows的对象管理器名字空间。Windows保留了两个不同的名字空间子目录，以供自己使用，其中一个是\Global??（在Windows 2000上为\??）子目录。（另一个是\BaseNamedObjects子目录，参见第3章的介绍）。在这个子目录中，Windows暴露了Windows应用程序赖以打交道的设备对象，包括COM和并行端口，以及磁盘。因为磁盘对象实际上驻留在另外的子目录中，所以，Windows使用符号链接，将\Global??下面的名称与名字空间中其他地方的对象连接起来。对于系统中的每一个物理磁盘，I/O管理器创建一个\Global??\PhysicalDrive*X*链接，指向\Device\Harddisk*X*\Partition0（这里的*X*是数字，从0开始）。直接与磁盘扇区打交道的Windows应用程序，通过调用Windows的*CreateFile*函数，并指定\\.\PhysicalDrive*X*（这里*X*是磁盘号）名称作为一个参数，就可以打开该磁盘。Windows应用层在把该名称交给Windows对象管理器以前，先将它转换成\Global??\PhysicalDrive*X*。

www.TopSage.com

分区管理器

分区管理器（\Windows\System32\Drivers\Partmgr.sys）负责通知即插即用（PnP）管理器当前有哪些分区，因而，卷管理器驱动程序（本章后面讲述）可以接收到有关分区创建和删除的通知。

分区管理器为了知晓这些分区，它充当了磁盘类驱动程序所创建的磁盘设备对象的功能型驱动程序。当系统引导的时候，它读入所附载的磁盘的分区表（在Windows 2000上，通过内核的*IoReadPartitionTable*函数；在Windows XP和Windows Server 2003上，通过内核的*IoReadPartitionTableEx*函数），并且将它们报告给PnP管理器。卷管理器设备驱动程序接收到它们所管理的磁盘的分区通知，它们考察所有构成其卷的分区时定义相应的卷对象。因为分区管理器设备对象过滤了磁盘设备对象，所以，分区管理器监视着与分区表修改相关的I/O请求包（IRP），因此，它可以更新它的内部分区表，并且将分区的创建和删除事件通知PnP管理器。

10.3 卷的管理

Windows引入了**基本**（*basic*）磁盘和动态（*dynamic*）磁盘的概念。Windows将那些只依赖于MBR风格或GPT分区方案的磁盘称为**基本磁盘**（*basic disk*）。动态磁盘是在Windows 2000中新引入的，它们实现了一种比基本磁盘更为灵活的分区方案。基本磁盘和动态磁盘之间的根本区别是，动态磁盘支持新的多分区卷。在本章刚开始的术语列表中提到，多分区卷提供了性能、尺寸变化和可靠性这些特性，这是简单卷所不支持的。Windows将所有的磁盘都按照基本磁盘来管理，除非你手工创建动态磁盘，或者将已有的基本磁盘（有足够的空闲空间）转换为动态磁盘。Microsoft推荐你使用基本磁盘，除非你需要动态磁盘的多分区功能。



> **注** 基本磁盘只支持从Windows NT 4系统升级携带过来的多分区卷，不过在Windows Server 2003上是一个例外，它不支持基本磁盘上的多分区卷。出于诸多因素的考虑，譬如，便携式电脑通常只有一个磁盘，而且便携式电脑的磁盘通常不容易在计算机之间移动，所以，Windows在便携式电脑上只使用基本磁盘。此外，只有固定的磁盘才可以是动态的，位于IEEE 1394或USB总线上的磁盘。共享的集群服务器磁盘总是基本磁盘。

www.TopSage.com

Windows卷管理的演变

Windows中存储管理的演变过程是从Microsoft的第一个操作系统，即MS-DOS开始的。随着硬盘变得越来越大，MS-DOS需要能够适应大的硬盘。为了做到这一点，Microsoft首先采取的步骤之一是，让MS-DOS在一个物理磁盘上创建多个分区或者逻辑磁盘。MS-DOS可以将每个分区格式化成不同的文件系统类型（FAT12或FAT16），并且为每个分区分配一个不同的驱动器字母。MS-DOS版本3和版本4对于它们所能创建的分区的大小和数量有非常严格的限制，但是，在MS-DOS 5中，分区方案完全成熟了。MS-DOS 5能够将一个磁盘划分成任何数量、任何大小的分区。

Windows NT借用了MS-DOS中已经发展成熟的分区方案，一方面可以提供它与MS-DOS和Windows 3.x的磁盘兼容性，另一方面也可以让Windows NT开发组依赖于业经考验的磁盘管理工具。在Windows NT中，Microsoft扩展了MS-DOS磁盘分区的基本概念，以支持企业级操作系统所要求的存储管理特性：磁盘跨展（disk spanning）和容错（fault tolerance）。从Windows NT的第一个版本（3.1版本）开始，系统管理员已经能够创建由多个分区构成的卷，这使得大的卷能够包含来自多个物理磁盘的分区，也可以通过基于软件的数据冗余来实现容错机制。

在Windows 2000之前的Windows NT版本中，虽然这种MS-DOS风格的分区方案足够灵活以支持大多数存储管理任务，但是，它仍然有一些缺点。其中的一个缺点是，绝大多数的磁盘配置的改变都要求重新引导以后才能生效。在今天的环境中，服务器必须要连续在线几个月，甚至几年，任何的重新引导，即使是有计划的重新引导，也是极大的不便。另一个缺点是，Windows NT 4的注册表为MS-DOS风格的分区保存了多分区的磁盘配置信息。这种安排意味着：当你在系统之间移动磁盘时，移动配置信息是非常麻烦的；当你需要重新安装操作系统时，你很容易就会丢失配置信息。最后，每个卷都要求A和Z之间的一个单独的驱动器字母，这一要求也使得Windows 2000之前的Microsoft操作系统的用户备受困扰，他们能创建的本地和远程卷的数量有一个上限。

Windows包含三种类型的分区，这使得它可以在必要的时候克服以上提到的限制：MBR（Master Boot Record，主引导纪录）风格、GPT（GUID Partition Table，GUID分区表）和LDM（Logical Disk Manager，逻辑磁盘管理器）。

www.TopSage.com

基本磁盘

这一节讲述两种分区类型：MBR风格和GPT。Windows在基本磁盘上使用这两种类型来定义卷。本节也介绍FtDisk，这是卷管理器驱动程序，它把卷的抽象展示给文件系统驱动程序。Windows 2000磁盘管理器向你推荐将任何未分区的磁盘做成一个动态磁盘，但是，Windows Server 2003，像Windows XP和Windows 2000 Professional一样，默认将所有的磁盘定义成基本磁盘。

MBR风格的分区

x86硬件使用的标准BIOS规定了Windows的分区格式必须符合一个要求，即主磁盘的第一个扇区包含MBR（Master Boot Record，主引导纪录）。当一个x86处理器引导时，计算机的BIOS读入MBR，并且将MBR中的一部分内容看作是可执行代码。当BIOS完成了该计算机硬件的基本配置以后，BIOS调用此MBR代码来启动操作系统的引导过程。在Microsoft操作系统中，包括Windows，MBR也包含了一个**分区表**（*partition table*）。分区表是由4个条目组成的，这4个条目定义了该磁盘上至多4个**主分区**（*primary partition*）的位置。分区表也记录了一个分区的类型。预定义的分区类型有许多种；一个分区的类型指定了该分区包含的是哪一种文件系统。例如，针对FAT32和NTFS的分区类型。一种特殊的分区类型，称为**扩展分区**（*extended partition*），它在其自己的分区表中包含了另一个MBR。在一个扩展分区中，与主分区等价的分区称为**逻辑驱动器**（*logical drive*）。Microsoft通过使用扩展分区，克服了每个磁盘只有4个分区的明显限制。一般而言，扩展分区允许的递归层次可以无限继续，这意味着一个磁盘上可能的分区的数目不存在上限。Windows引导过程对于主驱动器和逻辑驱动器作了明确的区分。系统必须将主磁盘的一个主分区标记为活动的。MBR中的Windows代码将“存储在活动分区的第一个扇区中的代码”加载到内存中，然后将控制权交给此代码。由于主分区中第一个扇区在引导过程中所扮演的角色，Windows将任何分区的第一个扇区指定为引导扇区。第5章曾经提到过，每一个已被格式化成一个文件系统的分区都有一个引导扇区，其中保存了该分区中的文件系统的结构信息。

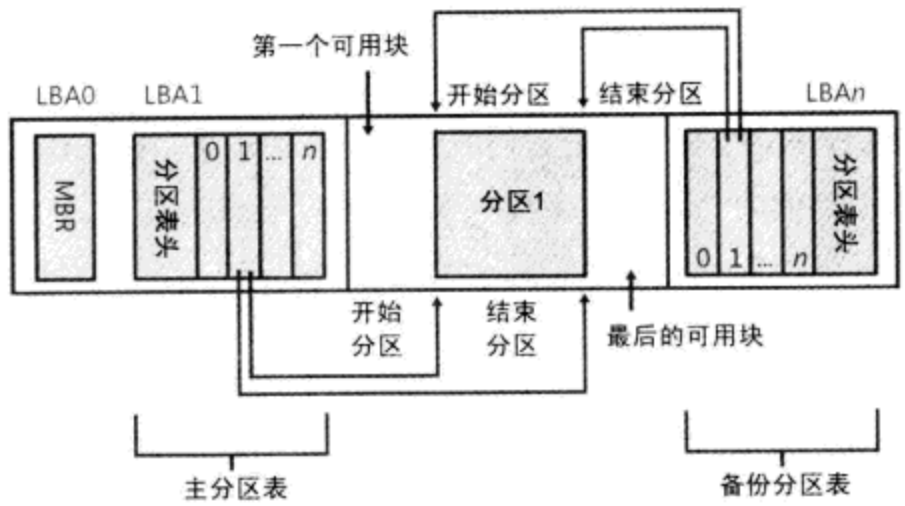
GPT（GUID分区表）分区方案

作为初始化过程的一部分，为了提供一个标准化的、可扩展的固件平台，以便于操作系统在它们的引导过程中使用，Intel设计了EFI（Extensible Firmware Interface，可扩展固件接口）规范。EFI包含了一个在固件（通常是ROM）中实现的小操作系统环境；在系统引导过程的早期，操作系统利用它来加载系统诊断程序和引导代码。EFI首先针对的目标之一是IA-64，所以，Windows的IA-64版本使用了EFI（你仍然可以使用老式的MBR分区风格来创建磁盘）。你可以在*http://developer.intel.com/technology/efi*上找到有关EFI的完整描述。

www.TopSage.com

EFI定义了一种分区方案，称为GUID（Globally Unique Identifier，全局惟一标识符）分区表（GPT，GUID Partition Table），它克服了MBR风格分区方案的某些缺点。例如，GPT分区结构使用的扇区地址是64位宽，而不是32位宽。对于访问只有2TB（terabyte，兆兆字节）的存储体，32位扇区地址已经足够了，而GPT允许磁盘大小的编址能力在可预见的将来都是足够的。GPT方案的其他优势包括：它使用循环冗余检验和（CRC，cyclic redundancy checksum）来确保分区表的完整性，以及它维护了分区表的一个备份拷贝。GPT之所以得此名称，是因为，它除了为每个分区保存一个36字节的Unicode分区名以外，还为每个分区分配了一个GUID。

图10.4显示了一个示例性的GPT分区布局图。如图MBR分区方案一样，GPT磁盘的第一个扇区是一个MBR，它的作用是保护GPT分区（万一从一个非GPT感知的操作系统中访问该磁盘）。然而，该磁盘的第二个扇区和最后一个扇区保存了GPT分区表的头，实际的分区表跟在第二个扇区的后面，在最后一个扇区的前面。GPT分区方案采用了这种可扩展的分区链表，所以它不要求像MBR分区那样使用嵌套的分区。



注：LBA=逻辑块地址

图10.4 GPT分区布局示例



> 注 因为Windows不支持在基本磁盘上创建多分区卷，所以，一个新的基本磁盘分区等同于一个卷。由于这个原因，当你在一个基本磁盘上创建一个卷的时候，磁盘管理MMC加载件（Disk Management MMC snap-in）仍然使用术语*分区*（*partition*）。

基本磁盘卷管理器

FtDisk驱动程序（\Windows\System32\Drivers\Ftdisk.sys）创建的磁盘设备对象代表了基本磁盘上的卷；该驱动程序在管理所有的基本磁盘卷（包括简单卷）的任务中扮演了不可或缺的角色。对于每个卷，FtDisk创建一个形如\Device\ HarddiskVolumeX的设备对象，其中X是一个标识该卷的数字（从1开始）。

www.TopSage.com

FtDisk实际上是一个总线型驱动程序，因为它负责列举基本的磁盘，以便检测基本卷的存在情况，并且将它们报告给PnP管理器。为了实现这种列举功能，FtDisk充分利用PnP管理器，同时在分区管理器驱动程序（Partmgr.sys）的帮助下，确定到底存在哪些基本磁盘分区。分区管理器向PnP管理器作了登记，所以，无论何时当磁盘类驱动程序创建一个分区设备对象时，Windows可以通知分区管理器。分区管理器通过一个私有的接口，将有关新分区对象的情况告诉FtDisk；它也创建过滤器设备对象，然后将它们附载到分区对象上。过滤器对象的存在，使得当一个分区设备对象被删除时，Windows可以通知分区管理器，因而分区管理器可以更新FtDisk。当在磁盘管理MMC加载件中一个分区被删除时，磁盘类驱动程序删除一个分区设备对象。随着FtDisk了解到有关分区的情况，它使用基本磁盘配置信息来确定从分区到卷的对应关系，并且，当它从一个卷的描述中知道了该卷所有分区都存在时，它创建一个卷设备对象。

Windows的卷驱动器字母的分配过程（稍后介绍）创建了一些驱动器字母符号链接，并将它们放在\Global??（在Windows 2000中为\??）对象管理器目录中。这些符号链接指向FtDisk创建的卷设备对象。当系统或者应用程序第一次访问一个卷的时候，Windows执行一个挂载（mount）操作，让文件系统驱动程序对于那些“用它们所管理的文件系统来格式化的卷”，有机会识别并宣称其所有权（本章后面的“卷的挂载”一节中描述了挂载操作）。

动态磁盘

正如我们前面所说明的，动态磁盘是指Windows中创建多分区卷（比如镜像卷、条带卷和RAID-5，本章后面将会介绍）所必需的磁盘格式。动态磁盘使用LDM（Logical Disk Manager，逻辑磁盘管理器）分区方案来进行分区。Windows中的LDM子系统包含了用户模式组件和设备驱动程序组件，并负责监管动态磁盘。Microsoft从VERITAS Software公司获得LDM的许可，这家公司最初为UNIX系统开发了LDM技术。通过与Microsoft的紧密合作，VERITAS将它的LDM移植到Windows上，从而为Windows操作系统提供更为稳定的分区方案和多分区卷的能力。在LDM的分区方案与MBR和GPT分区方案之间，一个重要的差别是，LDM维护一个统一的数据库，它保存了系统中所有动态磁盘的分区信息，包括多分区卷的配置信息。



> 注 LDM的UNIX版本结合了*磁盘组*（*disk group*）的概念，系统分配给一个磁盘组的所有动态磁盘共享同一个数据库。VERITAS针对Windows提供的商业卷管理软件也包含磁盘组，但是，Windows的LDM实现只包含一个磁盘组。

www.TopSage.com

LDM数据库

LDM数据库驻留在每个动态磁盘末尾的1MB预留空间中。为什么你在将一个基本磁盘转换为动态磁盘以前，Windows要求此基本磁盘的末尾必须有空闲空间，其原因正在于此。LDM数据库由四个区域构成（如图10.5所示）：一个头扇区，LDM称其为私有头（Private Header）；内容表区；数据库纪录区；以及一个事务型日志区（图10.5中显示的第5个区域只不过是私有头的一个拷贝而已）。私有头扇区位于动态磁盘末尾之前1MB的地方，它将LDM数据库的位置确定下来。随着你在Windows平台上花的时间越多，你将很快注意到，它使用GUID来标识几乎一切事物，磁盘也不无例外。GUID（全局惟一标识符）是一个128位值，Windows中的各种组件利用它来惟一地标识各种对象。LDM为每个动态磁盘分配一个GUID，私有头扇区将它所在的动态磁盘的GUID记录下来。因此，私有头的目的就在于记录这些对于磁盘来说私有的信息。私有头也保存了磁盘组的名称，它是计算机的名称再拼接上Dg0（例如，如果计算机的名称为Dary1，则磁盘组的名称为Dary1Dg0）；私有头还保存了一个指向数据库内容表起始处的指针（正如前面所提到的，LDM的Windows实现只包含一个磁盘组，所以，磁盘组的名称总是以Dg0结尾）。 为了可靠起见，LDM在磁盘的最后扇区中保存了私有头的一份拷贝。

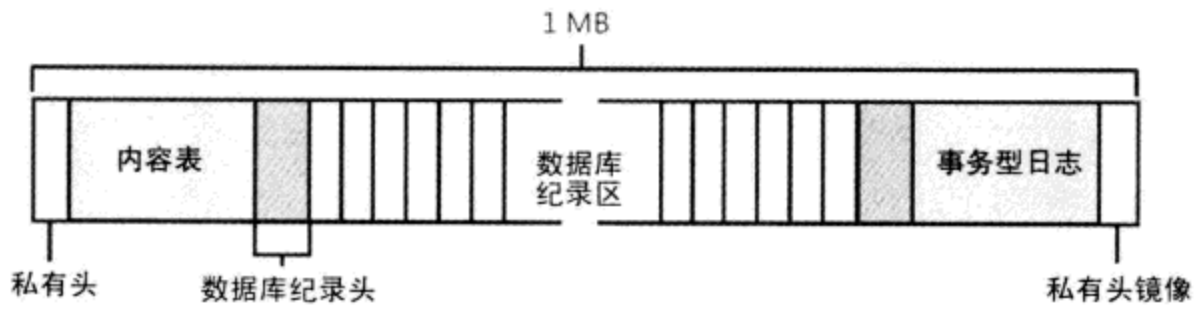


图10.5 LDM数据库布局结构

数据库的内容表是16个扇区大小，它包含了有关数据库布局结构的信息。在内容表之后紧跟着的是数据库纪录区，刚开始的一个扇区被当作数据库纪录头。该扇区存储了有关数据库纪录区的信息，包括它所包含的纪录的数量、与该数据库相关的磁盘组的名称和GUID，以及LDM为该数据库中的下一个纪录将使用的序列号标识符。跟在数据库纪录头后面的那些扇区包含了128字节固定大小的纪录，这些纪录中保存的信息描述了该磁盘组的分区和卷。

数据库中的纪录项可以是四种类型：（分区、磁盘、组件和卷）之一。LDM使用这些数据库纪录类型来标识三种描述卷的层次。LDM将这些纪录项与内部对象标识符关联起来。在最低层次上，**分区纪录项**（*partition entry*）描述了软分区，这是指磁盘上的连续区域；存储在分区纪录项中的标识符将该纪录项与一个组件和磁盘纪录项链接起来。

www.TopSage.com

磁盘纪录项（*disk entry*）代表了一个动态磁盘（属于磁盘组的一部分），它包含了该磁盘的GUID。**组件纪录项**（*component entry*）的用途是，作为一个或多个分区纪录项与每个分区所关联的卷纪录项之间的一个连接器。**卷纪录项**（*volume entry*）保存了卷的GUID、卷的总大小和状态，以及一个驱动器字母提示。大于一个数据库纪录的磁盘纪录项可以扩展到多个纪录中。分区、组件和卷纪录项很少跨越多个纪录。

LDM要求三个纪录项来描述一个简单的卷：一个分区、组件和卷纪录项。下面的列表显示了一个简单LDM数据库的内容，它定义了一个只包含一个分区的200MB大小的卷：

```
Disk Entry      Volume Entry    Component Entry    Partition Entry
Name: Disk1     Name: Volume1   Name: Volume1-01   Name: Disk1-01
GUID: XXX-XX... ID: 0x408       ID: 0x409          ID: 0x407
Disk ID: 0x404  State: ACTIVE   Parent ID: 0x408   Parent ID: 0x409
                Size: 200MB                        Disk ID: 0x404
                GUID: XXX-XX...                    Start: 300MB
                Drive Hint: H:                     Size: 200MB
```

分区纪录项描述了一个磁盘上系统分配给该卷的那部分区域，组件纪录项将分区纪录项和卷纪录项连接起来，卷纪录项包含了Windows内部用于标识该卷的GUID。多分区卷要求三个以上的纪录项。例如，一个条带卷（本章后面讲述）包含了至少两个分区纪录项、一个组件纪录项和一个卷纪录项。若一个卷类型有多于一个组件纪录项，则该卷类型必定是一个镜像卷；镜像卷有两个组件纪录项，每一个代表了该镜像的一半。LDM为镜像卷使用了两个组件纪录项，所以，当你打破一个镜像卷时，LDM可以在组件层次上将它分开，创建两个卷，每个卷都有一个组件纪录项。

LDM数据库的最后区域是事务型日志区域，它是由少量扇区构成的，当数据库信息被修改时，LDM将备份信息保存在这里。这样的设计是为了防止数据库万一遭遇崩溃或电源失败，因为LDM可以利用日志使数据库回到一致的状态。



实验：使用LDMDump来查看LDM数据库

你可以使用Sysinternals的LDMDump来查看有关LDM数据库内容的细节信息。LDMDump接受一个磁盘号作为它的一个命令行参数，它的输出通常要超过几个屏幕，所以，你应该将它的输出通过管道定向到一个文件中，以便在文本编辑器中查看 —— 例如，*ldmdump /d0 > disk.txt*。下面的例子显示了LDMDump输出的一段节选。首先显示的是LDM数据库头，然后是LDM数据库纪录，它描述了一个4GB硬盘，只有一个4-GB简单卷。该卷的数据库纪录项被列出为Volume1。在输出的末尾，LDMDump列出了它在数据库中找到的卷的软分区及其定义。

www.TopSage.com

```
C:\>ldump /d0
Logical Disk Manager Configuration Dump v1.03
Copyright (C) 2000-2002 Mark Russinovich

PRIVATE HEAD:
Signature           : PRIVHEAD
Version             : 2.11
Disk Id             : b78e4169-2e39-4a57-a98b-afd7131392f9
Host Id             : 1b77da20-c717-11d0-a5be-00a0c91db73c
Disk Group Id       : 2ea3c400-c92a-4172-9286-de46dda1098a
Disk Group Name     : Vmware03Dg0
Logical disk start  : 3F
Logical disk size   : 7FF54B (4094 MB)
Configuration start: 7FF7E0
Configuration size  : 800 (1 MB)
Number of TOCs      : 1
TOC size            : 7FE (1023 KB)
Number of Configs   : 1
Config size         : 5AC (726 KB)
Number of Logs      : 1
Log size            : DC (110 KB)

TOC 0:
Signature           : TOCBLOCK
Sequence            : 0x9
Config bitmap start: 0x11
Config bitmap size : 0x5AC
...
VBLK DATABASE:
0x000004: [000004] <Disk>
          Name         : Disk1
          Object Id    : 0x0403
          Disk Id      : b78e4169-2e39-4a57-a98b-afd7131392f9

0x000005: [000002] <DiskGroup>
          Name         : Vmware03Dg0
          Object Id    : 0x0401
          GUID         : 2ea3c400-c92a-4172-9286-de46dda1098a
0x000006: [000006] <Volume>
          Name         : Volume1
          Object Id    : 0x0406
          Volume state: ACTIVE
          Size         : 0x007FB68A (4086 MB)
          GUID         : e6ee6edc-d1ba-11d8-813e-806e6f6e6963
          Drive Hint   : C:

...
PARTITION LAYOUT:
Disk Disk1:
        Disk1-01 Offset: 0x00000000 Length: 0x007FB68A (4086 MB)
VOLUME DEFINITIONS: Volume1 Size: 0x007FB68A (4086 MB)
    Volume1-01 -
      Disk1-01   VolumeOffset: 0x00000000 Offset: 0x00000000 Length: 0x007FB68A
```


www.TopSage.com

LDM和GPT或MBR风格的分区方案

当你在一台计算机上安装Windows的时候，你首先要做的一件事情是，在系统的主物理磁盘上创建一个分区。Windows在该分区上定义系统卷，以存放那些在引导过程早期它需要调用的文件。而且，Windows Setup程序要求你创建一个分区作为引导卷所在的分区，它把Windows的系统文件安装于其上，并且在该分区上创建系统目录（\Windows）。系统卷和引导卷可以是同一个卷，在这样的情况下，你不必为引导卷创建新的分区。Microsoft定义的系统卷和引导卷这两个术语多少有点让人混淆。系统卷是Windows放置引导文件的地方，包括引导加载器程序（Ntldr）和Ntdetect；而引导卷是Windows存放系统文件的地方，比如核心的内核文件Ntoskrnl.exe。

虽然一个动态磁盘的分区数据驻留在LDM数据库中，但是，LDM实现了MBR风格的分区方案或者GPT分区方案，因而，当系统卷和引导卷位于动态磁盘上的时候，Windows的引导代码可以找到它们（例如，Ntldr和IA64固件对于LDM分区方案一无所知）。如果一个磁盘包含了系统卷或引导卷，那么，MBR或GPT分区表中的分区信息描述了这些卷的位置。否则的话，有一个分区包括了该磁盘的整个可用区域。LDM将这个分区标记为类型“LDM”，这是Windows 2000新引入的分区类型。这一占位性质的MBR风格或GPT的分区方案，所包括的区域也正是LDM创建分区（并且由LDM数据库来组织这些分区）的范围。在采用MBR分区风格的磁盘上，LDM数据库驻留在磁盘末尾的隐藏扇区中，而在采用GPT分区方案的磁盘上，在接近于磁盘起始处的地方，有一个LDM元数据分区，其中包含了LDM数据库。

LDM要创建一个MBR风格或GPT分区表的另一个原因是，这样做以后，那些遗留下来的磁盘管理工具，包括那些在双引导环境中既可在Windows下运行，也可在其他操作系统下运行的磁盘管理工具，它们不会错误地认为，动态磁盘是未分区的。

因为在一个磁盘的MBR风格或GPT分区表中并没有描述LDM分区，所以，它们称为**软分区**（*soft partition*）；而MBR风格的分区或GPT分区称为**硬分区**（*hard partition*）。图10.6显示了在一个按MBR风格进行分区的磁盘上的动态磁盘布局结构。

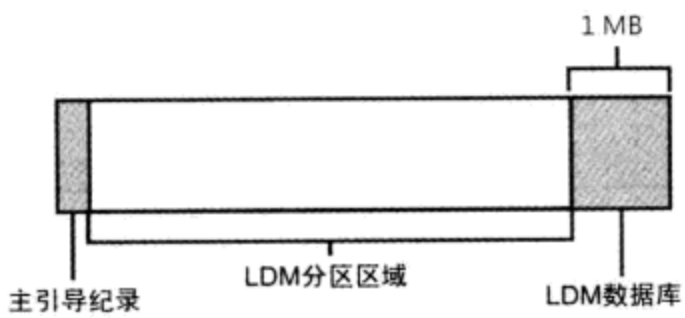


图10.6 动态磁盘的内部组织结构

www.TopSage.com

动态磁盘的卷管理器

磁盘管理（Disk Management）MMC加载件DLL（DMDiskManager，位于\Windows\System32\Dmdskmgr.dll，如图10.7所示）使用DMAdmin来创建和改变LDM数据库的内容。这里DMAdmin是LDM磁盘管理服务（\Windows\System32\Dmadmin.exe）。当你打开磁盘管理MMC加载件时，DMDiskManager加载到内存中，如果DMAdmin尚未运行的话，则启动DMAdmin。DMAdmin从每个磁盘中读入LDM数据库，并且将它获得的信息返回给DMDiskManager。如果DMAdmin检测到一个数据库来自于另一台计算机的磁盘组，那么，它注明，该磁盘上的卷是外部的，如果你想要使用这些卷的话，你可以将它们导入到当前计算机的数据库中。当你改变了动态磁盘的配置时，DMDiskManager将这些变化通知DMAdmin，DMAdmin则更新其数据库在内存中的拷贝。当DMAdmin提交变化时，它把更新过的数据库传递给DMIO设备驱动程序（Dmio.sys）。DMIO在动态磁盘环境中的角色等价于FtDisk在基本磁盘环境中的角色，所以，它控制了对于磁盘上的数据库的访问，并且它负责创建那些“代表动态磁盘上的卷”的设备对象。当你退出磁盘管理MMC加载件时，DMDiskManager停止运行，并卸载DMAdmin服务。

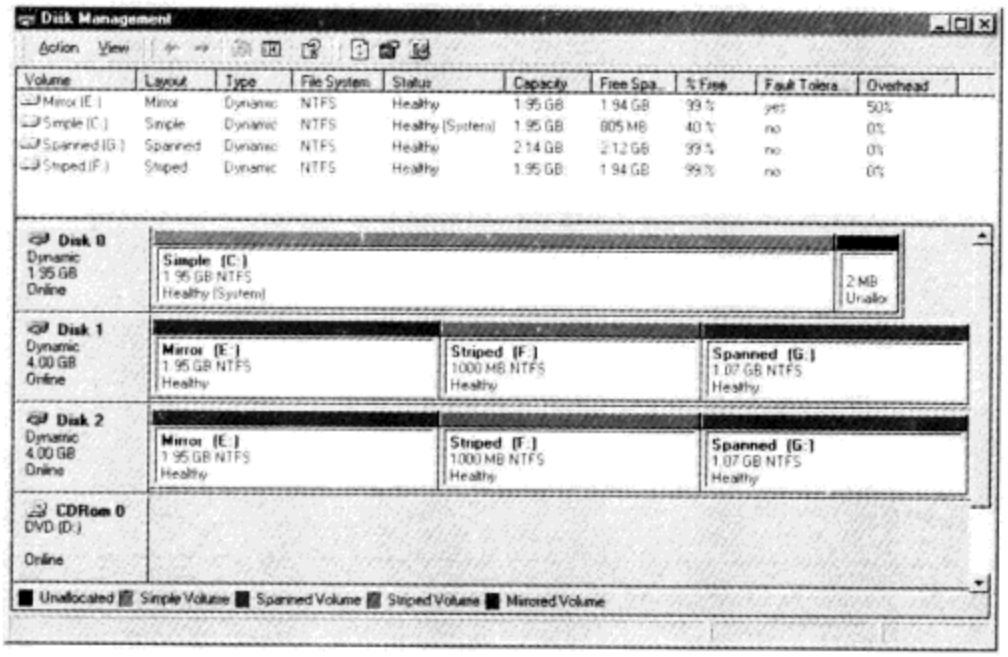


图10.7 磁盘管理MMC加载件

DMIO并不知道如何解释它所监管的数据库。DMConfig（\Windows\System32\Dmconfig.dll，一个由DMAdmin加载的DLL）和另一个设备驱动程序DMBoot（Dmboot.sys）负责解释该数据库。DMConfig知道如何读和更新数据库；DMBoot只知道如何读数据库。如果另一个LDM驱动程序DMLoad（Dmload.sys）确定当前系统上至少有一个动态磁盘，则在引导过程中DMBoot被加载到系统中。DMLoad通过询问DMIO来做出这一决定。如果当前系统上至少有一个动态磁盘，则DMLoad启动DMBoot，DMBoot会扫描LDM数据库。DMBoot告诉DMIO，它所碰到的每一个卷是如何构成的，所以，DMIO可以创建设备对象来代表这些卷。DMBoot在完成了这一扫描过程以后，立即从内存中卸载。因为DMIO中没有关于如何解释数据库的代码逻辑，所以，它相对比较小。它的小尺寸是非常有好处的，因为DMIO总是被加载在内存中。

www.TopSage.com

如同FtDisk一样，DMIO是一个总线型驱动程序，它为每一个动态磁盘卷创建一个名为\Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume*X*的设备对象，这里*X*是DMIO分配给该卷的标识符。而且，DMIO还创建另一个名为\Device\HarddiskDmVolumes\PhysicalDmVolumes\RawVolume*X*的设备对象，它代表了这个卷中的原始（未结构化的）I/O。图10.8显示了在一个有三个动态磁盘卷的系统上，DMIO所创建的设备对象。DMIO也在对象管理器名字空间中为每个卷创建许多符号链接，对于每个卷，从形如\Device\HarddiskDmVolumes\ *ComputerName*Dg0\Volume*Y*的链接开始。DMIO用计算机的名称来替代*ComputerName*，用卷的标识符（不同于DMIO分配给设备对象的内部标识符）来替代*Y*。这些链接指向PhysicalDmVolumes目录下的块设备对象。

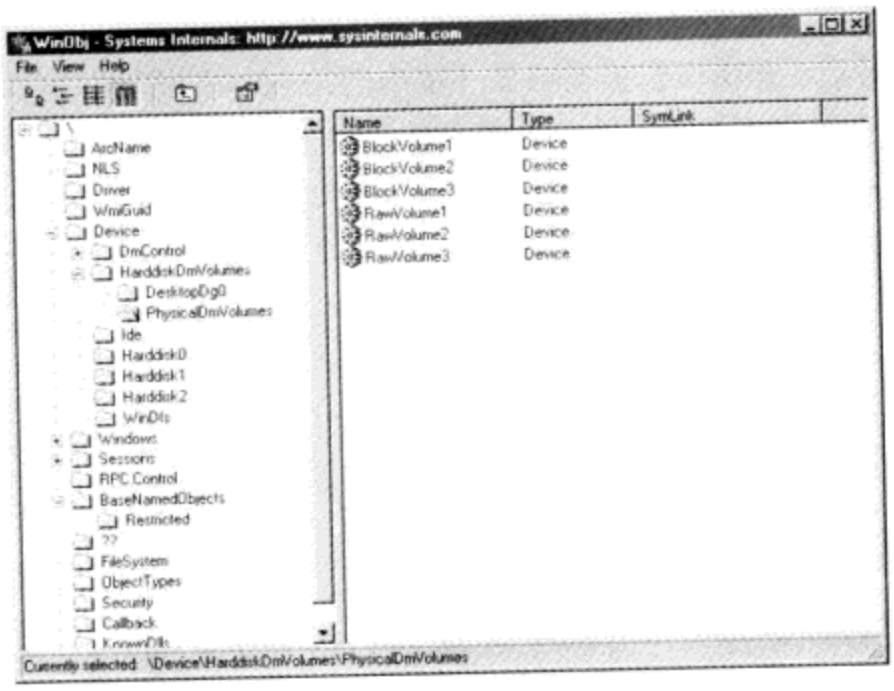


图10.8 DMIO驱动程序设备对象



注 在Windows 2000中，另一个驱动程序是DiskPerf（磁盘性能驱动程序，位于\Windows\System32\Drivers\Diskperf.sys）。DiskPerf将它的设备对象附载到代表物理磁盘的设备对象（例如，\Device\Harddisk0\Partition0）上，所以，它可以监视针对这些磁盘的I/O请求，并生成与性能有关的统计信息，供性能工具展示出来。这样的统计信息包括每秒钟读和写的字节、每秒钟的传输次数，以及花在执行磁盘I/O上的时间。在Windows XP和Windows Server 2003中，DiskPerf的功能是在分区管理器驱动程序中实现的，因为它已经过滤了物理磁盘设备对象来实现它的其他功能了（如本章前面所描述）。

多分区卷的管理

FtDisk和DMIO负责将卷展示给文件系统驱动程序，让它们来管理其中的数据；同时FtDisk和DMIO也负责将针对卷的I/O请求映射到底层构成这些卷的分区中。对于简单的卷，这个过程非常直截了当：卷管理器只需将相对于卷的偏移加上该卷的起始磁盘偏移，就可以确保，相对

www.TopSage.com

于卷的偏移被转译为相对于磁盘的偏移。

多分区卷更为复杂一些，因为构成一个卷的多个分区可以是不连续的分区，甚至在不同的磁盘上。有些类型的多分区卷使用了数据冗余，所以，它们在进行“从卷到磁盘的偏移”的转译过程中要牵扯到更多的工作。因此，DMIO必须处理所有针对它所管理的多分区卷的I/O请求，它要决定每个I/O最终影响到哪些分区。

在Windows中，以下类型的多分区卷是可以使用的：

- 跨距卷（spanned volume）；
- 镜像卷（mirrored volume）；
- 条带卷（striped volume）；
- RAID-5卷（RAID-5 volume）。

在讲述了每一种多分区卷类型的分区配置和逻辑操作以后，我们将介绍FtDisk和DMIO是如何处理从文件系统驱动程序发送给多分区卷的IRP的。在解释多分区卷的整个过程中，术语“**卷管理器**（*volume manager*）”被用来代表DMIO，因为正如本章前面所提到的，FtDisk只支持从NT 4迁移过来的多分区卷类型。

跨距卷

跨距卷（*spanned volume*）是一个由至多32个空闲分区构成的逻辑卷，这些空闲分区可以在一个或多个磁盘上。Windows的磁盘管理MMC加载件将这些分区组合成为一个跨距卷，然后该跨距卷可以被格式化成任何一种Windows支持的文件系统。图10.9显示了一个100MB的跨距卷，由驱动器字母D:标识，该跨距卷是根据第一个磁盘的后三分之一和第二个磁盘的前三分之一来创建的。在Windows NT 4中，跨距卷称为**卷集**（*volume set*）。

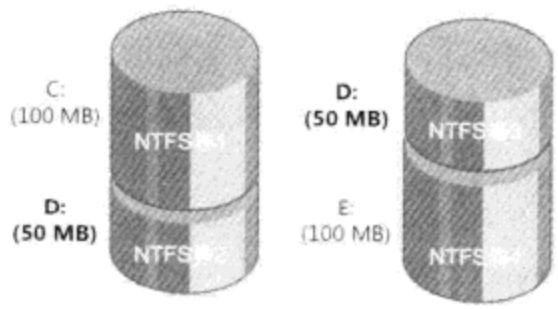


图10.9 跨距卷

跨距卷对于将空闲磁盘空间中的许多小区域合并成一个大的卷是非常有用的，或者对于将两个或多个小磁盘创建成单个大的卷，也是非常有用的。如果跨距卷已经被格式化成NTFS，则它还可以被扩展，以便将新的空闲区域或新的磁盘加入进来，而不会影响到该卷上已经保存

www.TopSage.com

的数据。这种扩展性是“将NTFS卷上的所有数据都按照文件来描述”而带来的最大好处之一。NTFS可以动态地增加一个逻辑卷的大小，因为记录该卷的分配状态的位图只是另一个文件——位图文件。该位图文件可以被扩展，以便将新加入该卷的任何空间都包含进来。另一方面，要想动态地扩展一个FAT卷，则要求FAT本身能被扩展，而这将导致磁盘上的其他所有数据都被弄乱。

卷管理器将磁盘的物理配置隐藏起来，从而Windows上安装的文件系统看不到磁盘的物理配置情况。例如，NTFS把图10.9中的卷D:看作一个普通的100MB的卷。NTFS参照它的位图来确定该卷中的哪些空间是空闲的，以便可以分配出去。然后，它调用卷管理器来读或写“从该卷的某个特定的字节偏移处开始”的数据。卷管理器将跨距卷中的物理扇区看作是顺序编号的，从第一个磁盘上的第一个空闲区域开始，一直到最后磁盘上的最后空闲区域为止。它需要确定的是，哪个磁盘上的哪个物理扇区对应于所提供的字节偏移。

条带卷

条带卷（*striped volume*）是由一系列分区组成的单个逻辑卷。条带卷中至多有32个分区，并且每个磁盘一个分区。条带卷也称为*RAID 0*（*RAID 0*级）卷。图10.10显示了一个由三个分区构成的条带卷，三个磁盘中每个都有一个分区（条带卷中的分区并不一定要跨越整个磁盘，惟一的限制是，每个磁盘上的分区必须同样大小）。

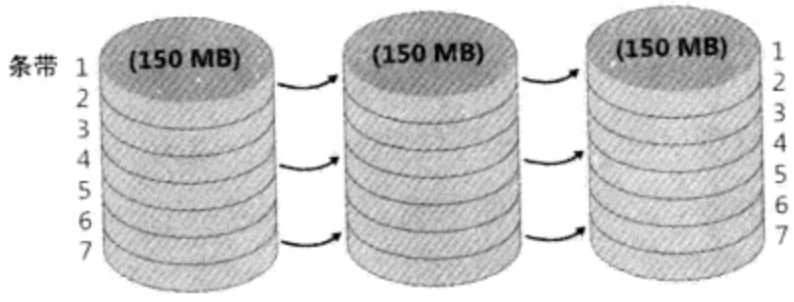


图10.10 条带卷

对于一个文件系统，此条带卷看起来像是单个450MB卷，但是，卷管理器通过调整卷数据在这些物理磁盘上的分布，来优化数据存储和信息获取时间。卷管理器访问磁盘的物理扇区，就好像它们在磁盘的条带中是顺序编号的一样，如图10.11所示。

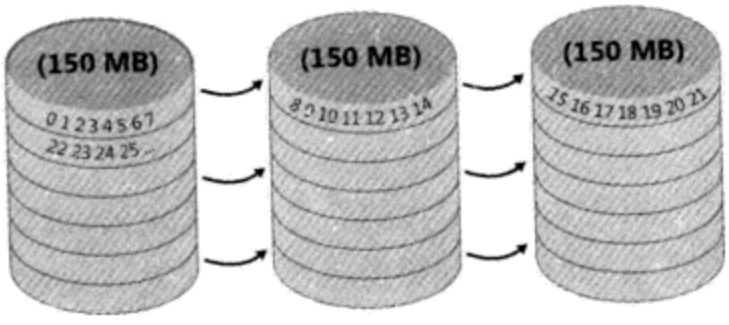


图10.11 条带卷中的物理扇区的逻辑编号

www.TopSage.com

因为每个条带是相对比较窄的64KB（选择这个值，是为了防止单独的读和写操作访问两个磁盘），所以，数据能够比较均匀地分布在这些磁盘上。因此，条带也增加了多个未完成的读和写操作被绑定在不同磁盘上的概率。因为所有这三块磁盘上的数据都能够被并发地访问，所以，磁盘I/O的延迟时间常常能被减少，尤其是在负载很重的系统上。

跨距卷使得管理磁盘卷非常方便，而条带卷使得I/O负载散布到多个磁盘上。然而，如果一个磁盘失败了，则这两种卷管理特性并不能提供恢复数据的能力。对于数据恢复，卷管理器实现了三种冗余存储方案：镜像卷、RAID-5卷和扇区备用（sector sparing）。（第12章讲述了扇区备用和NTFS对于扇区备用的支持）。这些特性可以通过Windows的磁盘管理工具建立起来。

镜像卷

在**镜像卷**（*mirrored volume*）中，一个磁盘上的分区的内容被复制到另一个磁盘上相等大小的分区中。镜像卷有时候也称为RAID-1（*RAID 1级*）。图10.12显示了一个镜像卷。

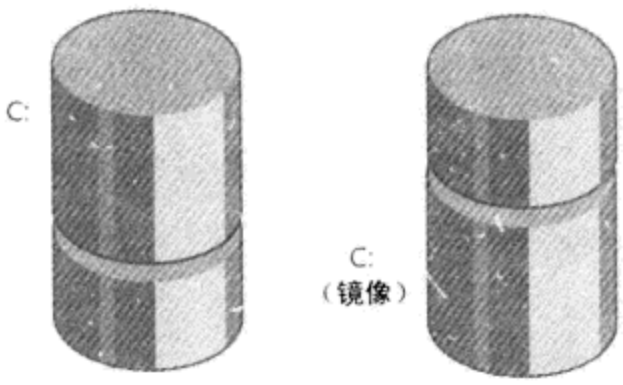


图10.12 镜像卷

当一个程序向驱动器C:中写数据时，卷管理器将同样的数据也写到镜像分区上同样的位置处。如果第一个磁盘或者它的C:分区上的任何数据由于硬件或软件的失败而变得不可访问了，那么，卷管理器自动地访问镜像分区上的数据。镜像卷可以被格式化成Windows支持的任何一种文件系统。文件系统驱动程序仍然是独立的，它们不受卷管理器的镜像行为的任何影响。

在负载很重的系统上，镜像卷可以帮助提高读I/O吞吐量。当I/O活动很频繁的时候，卷管理器在主分区和镜像分区之间，对它的读操作进行平衡（考虑每一个磁盘上的未完成且正在进行之中的I/O请求的数量）。两个读操作可以并发地进行，因此理论上可以在一半的时间内完成。当一个文件被修改时，此镜像集的两个分区都必须被写入，但是磁盘写是异步完成的，所以，用户模式程序的性能通常不会由于额外的磁盘更新操作而受到影响。

www.TopSage.com

镜像卷是惟一一种系统卷和引导卷都可以支持的多分区卷。其理由是：Windows的引导代码，包括MBR代码和Ntldr，并没有包含“为了理解多分区卷而需要的”复杂逻辑；镜像卷是一个例外，因为引导代码将它们看作简单卷，它只是从镜像卷的一个分区（在MBR风格的分区表中被标记为引导的或系统的驱动器）中读取代码或数据。因为引导代码并不修改该磁盘，所以，它可以安全地忽略镜像卷的另一半。



实验：观察镜像卷的I/O操作

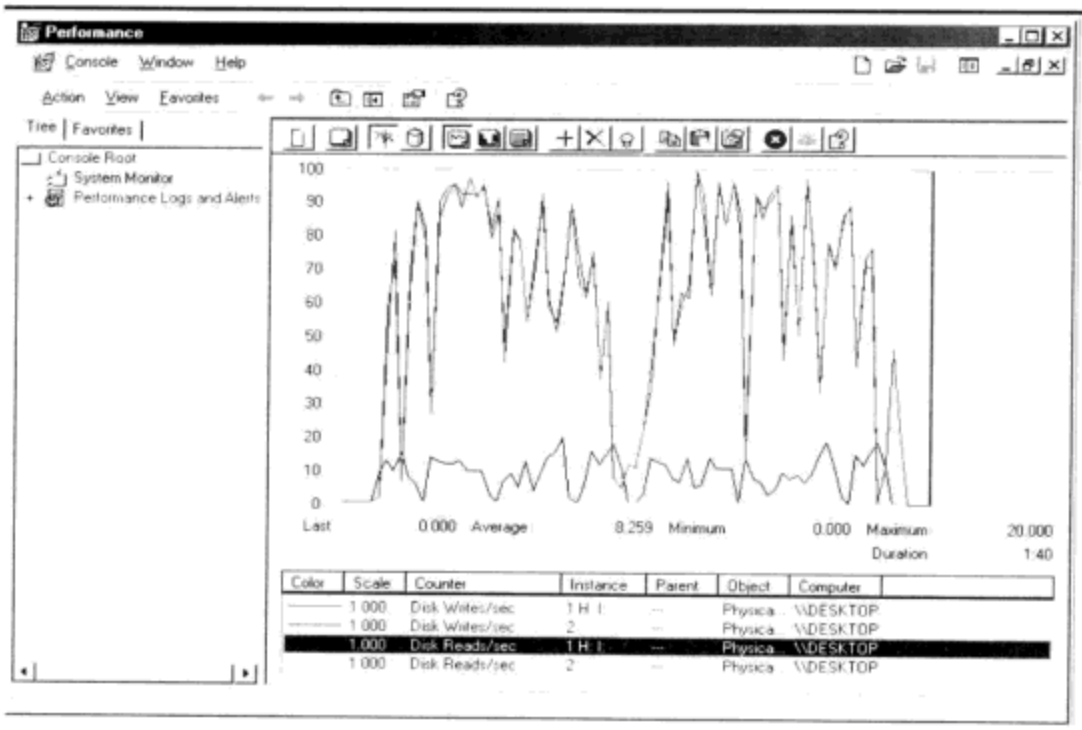
利用Windows的性能工具，你可以检验一下，针对镜像卷的写操作同时将数据拷贝到该镜像的两个磁盘中，而对于读操作，如果相对不频繁的话，则主要发生在该卷的一个分区上。这个实验要求有三个硬盘和一个Windows 2000服务器或Windows Server 2003系统。如果你没有三个磁盘或者一个服务器系统，那么，你可以跳过实验操作步骤，直接看一下本实验中的性能工具屏幕截图，它说明了本实验的结果。

使用磁盘管理MMC加载件来创建一个镜像卷。为了做到这一点，执行下面的步骤。

1. 启动计算机管理（Computer Management）程序，展开Storage树，并选项Disk Management（或者，在一个MMC控制台中，插入Disk Management作为一个加载件），从而运行Disk Management。
2. 在一个驱动器的未分配空间上右键点击，选择“Create Volume”。
3. 按照Create Volume向导的指示，创建一个简单卷（确保在另一个磁盘上有足够的空间，以便你可以创建一个同样大小的分区）。
4. 在新的卷上右键点击，从环境菜单中选择“Add Mirror”。

一旦你已经有了一个镜像卷，就运行性能工具。并且，针对这个镜像卷的两个分区所属的两个磁盘实例，加入PhysicalDisk性能对象的计数器。为每个实例选择“Disk Reads/Sec”和“Disk Writes/Sec”计数器。从第三个磁盘（不属于镜像卷）上选择一个大的目录，将它拷贝到镜像卷中。在拷贝操作进行过程中，性能工具的输出窗口应该看起来类似如下：

www.TopSage.com



上面的两条线几乎整个重叠在一起了，它们是每个磁盘的“Disk Writes/Sec”计数器。底下的两条线是“Disk Reads/Sec”线。上面的屏幕截图表明了卷管理器（这里是DMIO）把被拷贝的数据写到该卷的两个分区上，而主要从一个分区中读取数据。读数据的行为之所以会如此（不平衡），是因为，在拷贝过程中，正在进行中的I/O操作的数量并没有确保“卷管理器应该执行更为主动的读操作负载平衡”。

RAID-5卷

*RAID-5卷*是常规条带卷的一种容错版本的变种。RAID-5卷实现了*RAID 5级*。它们也称为**支持奇偶校验的条带卷**（*striped volume with parity*），因为它们建立在条带卷所采用的条带方法基础之上。容错是这样实现的：预留等价于一个磁盘的容量，用于存储每个条带的奇偶校验。图10.13是一个RAID-5卷的可视表示。

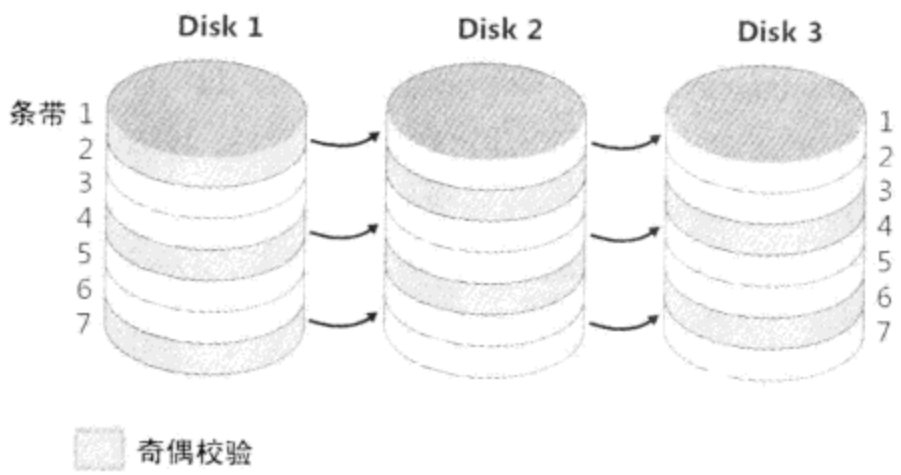


图10.13 RAID-5卷

www.TopSage.com

在图10.13中，条带1的奇偶校验存储在磁盘1上。它包含了磁盘2和3上第一个条带的字节对字节的逻辑和（XOR）。条带2的奇偶校验存储在磁盘2上，条带3的奇偶校验存储在磁盘3上。按照这种循环方式，将奇偶校验分布到磁盘上，这是一种I/O优化技术。每次当数据被写到一个磁盘时，对应于被修改数据的奇偶校验字节也必须被重新计算，并重新写到磁盘上。如果奇偶校验数据总是被写到同样的磁盘上，那么，这个磁盘将会一直很忙，有可能成为一个I/O瓶颈。

要想从RAID-5卷的一个失败磁盘中恢复过来，可依赖于简单的算术原则：在一个含有n个变量的等式中，如果你知道了$n-1$个变量的值，那么，你可以通过减法确定那一漏掉变量的值。例如，在等式 $x+y=z$中，z代表奇偶校验条带，则卷管理器计算$z-y$来确定x的内容；如果要找到y，则它计算$z-x$。卷管理器使用类似的逻辑来恢复丢失的数据。如果RAID-5卷中的一个磁盘失败了，或者如果一个磁盘上的数据变得不可读了，那么，卷管理器通过使用XOR操作（按位的逻辑加法）来重构丢失的数据。

如果图10.13中的磁盘1失败了，那么，它的条带2和5的内容可以通过将磁盘3上的对应条带与磁盘2上的奇偶校验条带进行XOR运算而重新计算得到；类似地，磁盘1上的条带3和6的内容，可以通过将磁盘2上的对应条带与磁盘3上的奇偶校验条带进行XOR运算而得到。为了创建一个RAID-5卷，要求至少有3个磁盘（或者说，3个磁盘上3个相等大小的分区）。

卷名字空间

驱动器字母分配是存储管理的一个方面，从Windows NT 4到Windows 2000，这部分管理有了重要的变化。虽然如此，Windows包含了相应的支持以便将Windows NT 4系统中的驱动器字母分配迁移到升级后的Windows中。Windows NT 4驱动器字母分配保存在HKLM\SYSTEM\Disk中。在升级过程中，这些信息被读出来，并保存到Windows系统所使用的位置上，所以，升级以后，系统不再引用Disk注册表键。

挂载管理器

挂载管理器设备驱动程序（Mountmgr.sys）为以下设备分配驱动器字母：在Windows安装以后创建的动态磁盘卷和基本磁盘卷、CD-ROM、软盘驱动器和可移除设备。Windows将所有的驱动器字母分配信息存储在HKLM\SYSTEM\MountedDevices中。如果你在注册表中查看该键，将会看到诸如\??\Volume{X}（这里X是一个GUID）这样的名称，以及诸如\DosDevices\C:这样的值。每一个卷都有一个卷名项，但它并不一定要分配一个驱动器字母。图10.14显示了一个挂载管理器注册表键的例子。注意，MountedDevices键，如同Windows NT 4中的Disk键一样，并不被包含在控制集中，所以，它并不受到“最后已知的好配置”引导选项的保护。（关于控制集和最后已知的好配置引导选项，请参见第4章的“接受当前引导和最后已知的好控制集”一节）。

www.TopSage.com

注册表中保存的关于基本磁盘卷驱动器字母和卷名称的数据有Windows NT 4风格的磁盘标识特征（disk signature），以及与该卷相关联的第一个分区的起始偏移位置。注册表中保存的关于动态磁盘卷的数据包括该卷的DMIO内部GUID。挂载管理器在引导过程中初始化时，它向Windows即插即用子系统进行登记，从而当FtDisk或DMIO创建一个卷的时候，它总是能够接收到通知。挂载管理器接收到这样一个通知时，它确定新卷的GUID或者磁盘标识特征，并使用该GUID或标识特征作为指导来查看其内部数据库，此内部数据库反映了MountedDevices注册表键的内容。然后，挂载管理器确定它的内部数据库是否包含了驱动器字母分配。如果该卷在数据库中没有相应的条目，则挂载管理器请FtDisk或DMIO（看谁创建了该卷），给出一个推荐的驱动器字母分配，并保存在数据库中。FtDisk并不返回任何推荐方案，而DMIO在该卷的数据库条目中查看驱动器字母提示（drive-letter hint）。

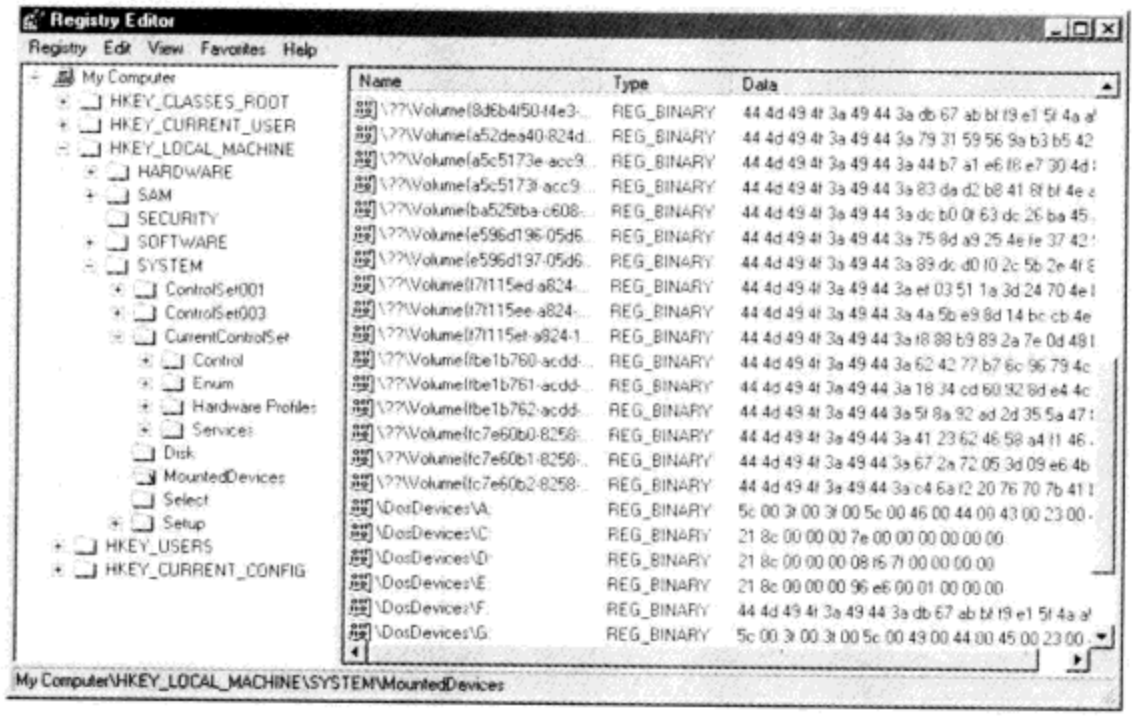


图10.14 在挂载管理器的注册表键中列出的已挂载设备

如果针对该卷不存在推荐的驱动器字母分配，则挂载管理器使用第一个未分配的驱动器字母（如果存在这样的字母），并定义一个新的字母分配，再为此字母分配创建一个符号链接（例如，在Windows XP和Windows Server 2003上为\Global??\D:，在Windows 2000上为\??\D:），并更新MountedDevices注册表键。如果没有可使用的驱动器字母，则不再进行驱动器字母分配，同时，挂载管理器创建一个卷符号链接（即\Global??\Volume{X}）。如果该卷尚未有GUID的话，则此符号链接定义一个新的卷GUID。此GUID不同于DMIO内部使用的卷GUID。

挂载点

挂载点（*mount point*）使得你可以通过NTFS卷上的目录，将多个卷链接起来，这使得没有分配到驱动器字母的卷也能够被访问。例如，你命名为C:\Projects的NTFS目录可以挂载另一

www.TopSage.com

个包含了你的工程目录和文件的卷（NTFS或FAT）。如果你的工程卷有一个文件，你命名为\CurrentProject\Description.txt，那么，你可以通过路径C:\Projects\CurrentProject\Description.txt来访问该文件。挂载点之所以成为可能，是因为它有一项支撑技术，此项技术称为**重解析点**（*reparse point*）（第12章更加详细地讨论了重解析点）。

重解析点是一个具有固定头部的任意数据块，Windows将它与一个NTFS文件或目录关联起来。应用程序或系统定义了一个重解析点的格式和行为，包括一个惟一的“用于标识该重解析点属于此应用程序或系统”的重解析点标记值；它也指定了一个重解析点的数据部分的大小和含义。（数据部分可以达到16KB大小）。重解析点将它们的惟一标记值存储在一个固定的段中。任何一个应用程序，如果它实现了一个重解析点，则它必须提供一个文件系统过滤型驱动程序，以观察那些在NTFS卷上执行的文件操作的、与重解析相关的返回代码，当它检测到这样的代码时，驱动程序必须采取适当的措施。当NTFS处理一个文件操作，并遇到一个与重解析点相关联的文件或目录时，它返回一个重解析状态代码。

Windows NTFS文件系统驱动程序、I/O管理器以及对象管理器都部分地实现了重解析点的功能。对象管理器通过I/O管理器与文件系统驱动程序连接起来，从而它可以激发路径名的解析操作。因此，对象管理器对于那些“I/O管理器返回了重解析状态代码”的操作，必须进行重试。I/O管理器实现了挂载点和其他重解析点可能要求的路径名修改动作，NTFS文件系统驱动程序必须将重解析点数据与文件和目录关联起来，并且能够识别这些重解析点。因此，你可以认为，对于Microsoft定义的许多重解析点，I/O管理器就相当于重解析点文件系统过滤型驱动程序。

重解析点的一个应用例子是HSM（层次型存储管理，Hierarchical Storage Management）系统，比如Windows 2000 Server和Windows Server 2003包含的Windows远程存储服务（RSS，Remote Storage Service），它使用重解析点来规定，管理员可以将哪些文件移到离线的磁带存储体中。当一个用户试着要访问一个离线文件时，HSM过滤型驱动程序检测到NTFS返回的重解析状态代码，它跟一个用户模式的服务进行通信，以便从离线存储体中获取该文件，然后它从该文件中删除重解析点，当用户模式服务获取到该文件以后，它再重试该文件操作。这正是RSS过滤型驱动程序（Rsfilter.sys）使用重解析点的做法。

如果I/O管理器从NTFS接收到一个重解析状态代码，但是，此NTFS返回代码所对应的那个文件或目录并没有与某一个内置的Windows重解析点关联在一起，那么，没有过滤型驱动程序声明过此重解析点。然后，I/O管理器向对象管理器返回一个错误，对象管理器又将此错误当作一个“系统不能访问此文件（file cannot be accessed by the system）”错误传递给发起这次文件或目录访问的应用程序。

挂载点是把卷名称（\Global??\Volume{*X*}）当作重解析数据保存起来的重解析点。当你使用磁盘管理MMC加载件来分配或移除卷的路径分配时，实际上你正在创建挂载点。你也可以使用内置的命令行工具*Mountvol.exe*（\Windows\System32\Mountvol.exe）来创建和显示挂载点。

www.TopSage.com

挂载管理器在每一个NTFS卷上，维护一个**挂载管理器远程数据库**（*Mount Manager remote database*），其中记录了针对该卷而定义的所有挂载点。此数据库文件$MountMgrRemoteDatabase驻留在NTFS根目录下。当一个磁盘从一个系统移动到另一系统中时，或者当它处在一个双引导的环境中（即在多个Windows系统之间进行引导）时，挂载点也会跟着移动，这是因为有了挂载管理器远程数据库的原因。NTFS也在NTFS元数据文件\$Extend\$Reparse中记录了挂载点（NTFS并没有让它的任何一个元数据文件可被应用程序查看）。NTFS将挂载点信息保存在元数据文件中，所以，当Windows应用程序，比如磁盘管理程序，要求得到挂载点的定义时，Windows可以很容易地将一个卷的挂载点列举出来。



实验：递归的挂载点

本实验使用www.sysinternals.com的Filemon工具来显示一个递归的挂载点所引起的有趣行为。递归挂载点是指这样一个挂载点：它链接回到其自身所在的卷。在一个递归挂载点上执行一个递归的目录列示命令，这样得到的文件访问痕迹，可以让我们清楚地看到NTFS是如何处理挂载点的。

要想创建并查看一个挂载点，请执行下面的步骤。

1. 打开一个命令提示窗口，或者Windows Explorer窗口，在一个NTFS驱动器上创建一个名为\Recurse的目录。
2. 在磁盘管理MMC加载件中，在相应的卷上右键点击，并选择“Change Drive Letter And Path”。
3. 当Add/Remove对话框出现的时候，输入你所创建的目录路径（例如，*I:\Recurse*）。
4. 启动Filemon，在Drivers菜单中，除了你刚才创建挂载点所在的那个卷以外，不再选择其他的任何卷。

现在你已经准备好生成一份递归挂载点的痕迹了。打开一个命名提示窗口，执行命名“*dir /s I:/Recurse*”。如果你看到了在后续文件操作的Filemon痕迹中所有引用Recurse的文件访问，那么，你将会注意到，此命令提示窗口第一次访问I:\Recurse，然后是I:\Recurse\Recurse，等等，递归越来越深。

这里的应用程序试图在递归的每一级上列出一个目录的内容，但无论何时，当它碰到挂载点的时候，它就会钻进去试图执行另一个目录列示操作。NTFS返回一个重解析状态代码，这告诉对象管理器倒退一级，再试一次。最后，当它退回到根目录的时候，应用程序检查它在挂载点递归中已经发现的文件或目录。由于对象管理器指示的重试活动，应用程序永远不会接收到一个重解析代码。对象管理器处理此重解析代码，就好像它是在执行目录来查找时从NTFS接收到重解析代码一样。

www.TopSage.com

Filemon将请求类型表示成它们的原生IRP类型，所以，目录或文件的打开操作被表示成一个*IRP_MJ_CREATE*请求。文件或目录的关闭操作是*IRP_MJ_CLOSE*，目录查询是一个*IRP_MJ_DIRECTORY_CONTROL*请求，同时在Filemon的Other栏中显示为*FileBothDirectoryInfo*。

File Monitor - Systems Internals: http://www.sysinternals.com

File Edit Options Drives Help

#	Process	Request	Path	Result	Other
83	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
84	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	NO MORE FILES	FileBothDirectoryInfo...
85	cmd.exe	IRP_MJ_CLEANUP	I:\	SUCCESS	
86	cmd.exe	IRP_MJ_CLOSE	I:\	SUCCESS	
87	cmd.exe	FSCTL_IS_VOLUME_MOUNTED	I:\	SUCCESS	
88	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse\recurse\recurse\	REPARSE	Attributes: Any Optio...
89	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse\recurse	REPARSE	Attributes: Any Optio...
90	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse	REPARSE	Attributes: Any Optio...
91	cmd.exe	IRP_MJ_CREATE	I:\recurse	REPARSE	Attributes: Any Optio...
92	cmd.exe	IRP_MJ_CREATE	I:\	SUCCESS	Attributes: Any Optio...
93	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
94	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
95	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	NO MORE FILES	FileBothDirectoryInfo...
96	cmd.exe	IRP_MJ_CLEANUP	I:\	SUCCESS	
97	cmd.exe	IRP_MJ_CLOSE	I:\	SUCCESS	
98	cmd.exe	FSCTL_IS_VOLUME_MOUNTED	I:\	SUCCESS	
99	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse\recurse\recurse\recurse\	REPARSE	Attributes: Any Optio...
100	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse\recurse\recurse	REPARSE	Attributes: Any Optio...
101	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse\recurse	REPARSE	Attributes: Any Optio...
102	cmd.exe	IRP_MJ_CREATE	I:\recurse\recurse	REPARSE	Attributes: Any Optio...
103	cmd.exe	IRP_MJ_CREATE	I:\recurse	REPARSE	Attributes: Any Optio...
104	cmd.exe	IRP_MJ_CREATE	I:\	SUCCESS	Attributes: Any Optio...
105	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
106	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	SUCCESS	FileBothDirectoryInfo...
107	cmd.exe	IRP_MJ_DIRECTORY_CONTROL	I:\	NO MORE FILES	FileBothDirectoryInfo...

为了防止缓冲区溢出和无限循环，命令提示程序和Windows Explorer会在目录深度到达32或路径名超过256个字符（取决于哪个条件先满足）时停止递归。

卷的挂载

Windows把一个驱动器字母分配给一个卷，这并不意味着该卷包含的数据是按照Windows能识别的一种文件系统格式来组织的。卷识别过程是由“一个文件系统声称对于某个分区具有所有权”构成的；此过程发生在内核、设备驱动程序或者应用程序第一次访问该卷上的一个文件或目录的时候。在文件系统驱动程序用信号通知了它对一个分区的责任以后，I/O管理器将所有针对该卷的IRP都转送给此所有者驱动程序。Windows中的挂载操作是由三个部件组成的：文件系统驱动程序的注册、卷参数块（VPB）和挂载请求。



> 注 在Windows Server 2003 Enterprise和Datacenter Edition中，当卷管理器报告其卷的存在情况时，卷的自动挂载功能是禁止的，这是为了防止系统太过于主动地把卷附载到一个SAN（System Area Network）上。你可以使用Diskpart命令行工具（Windows Server 2003中包含了该工具）来允许和禁止自动挂载的行为，你也可以使用系统中包含的Mountvol工具来挂载卷。

www.TopSage.com

I/O管理器监管着挂载过程，并且它知道所有当前可用的文件系统驱动程序。因为所有的文件系统驱动程序在初始化的时候都已经向I/O管理器注册过了。I/O管理器提供了*IoRegisterFileSystem*函数，让本地磁盘（不是网络的）文件系统驱动程序完成它们的注册工作。当一个文件系统驱动程序注册时，I/O管理器把该驱动程序的一个引用保存在一个列表中，从而可在挂载操作过程中使用。

每个设备对象都包含了一个VPB数据结构，但是I/O管理器认为这些VPB只对于卷设备对象才有意义。VPB的作用是，作为一个卷设备对象和代表该卷被挂载以后的文件系统实例（由某个文件系统驱动程序创建）之间的一个链接纽带。如果一个VPB的文件系统引用是空的，则说明没有文件系统挂载了该卷。无论何时，若当前正在执行的打开API指定了一个卷设备对象上的文件名或目录名时，则I/O管理器就会检查该卷设备对象的VPB。

例如，如果卷管理器将驱动器字母D分配给系统上的第二个卷，那么，它创建\Global??\D:符号链接，代表设备对象\Device\HarddiskVolume2。若一个Windows应用程序试图打开D:驱动器上的\Temp\Test.txt文件，则它指定名称D:\Temp\Test.txt；Windows子系统在调用内核的文件打开例程*NtCreateFile*以前，将该名称转换为\Global??\D:\Temp\Test.txt。*NtCreateFile*使用对象管理器来解析该名称，对象管理器碰到\Device\ HarddiskVolume2设备对象，且路径\Temp\Test.txt仍未被解析。在这个点上，I/O管理器检查\Device\HarddiskVolume2的VPB，看它是否引用了一个文件系统。如果它没有引用，那么，I/O管理器通过一个挂载请求，询问每一个已注册的文件系统驱动程序，它们是否认得该卷的格式作为其自身的文件系统格式。



实验：查看VPB

利用内核调试器命令*!vpb*，你可以查看一个VPB的内容。因为VPB是通过一个卷的设备对象来指示的，所以，你必须首先找到一个卷设备对象。为了做到这一点，你必须转储出卷管理器的一个设备对象，找到代表某个卷的设备对象，并显示该设备对象，从而可以得到其VPB域。

如果你的系统有一个动态磁盘，那么，你可以针对DMIO驱动程序来使用*!drvobj*驱动程序对象查看命令；否则，你需要指定FtDisk驱动程序。下面是一个例子：

```
kd> !drvobj ftdisk
Driver object (818aec50) is for:
 \Driver\Ftdisk
Driver Extension List: (id , addr)

Device Object list:
818a5290  817e96f0  817e98b0  817e9030
818a73b0  818a7810  8182d030
```


www.TopSage.com

*!drvobj*命令列出一个驱动程序所拥有的设备对象的地址。在这个例子中，共有7个设备对象。它们中的某一个代表了该设备驱动程序的编程接口，其余的都是卷设备对象。因为这些对象是按照它们被创建的逆序来列出的，并且该驱动程序首先创建设备驱动程序接口对象，所以，你知道，第一个被列出的设备对象是一个卷的设备对象。现在，在该卷设备对象的地址上执行内核调试器命令*!devobj*：

```
kd> !devobj 818a5290
Device object (818a5290) is for:
 HarddiskVolume6 \Driver\Ftdisk DriverObject 818aec50
Current Irp 00000000 RefCount 3 Type 00000007 Flags 00001050
Vpb 818a5da8 Dacl e1000384 DevExt 818a5348 DevObjExt 818a53f8 Dope 818a50a8
  DevNode 818a5ae8
ExtensionFlags (0xa8000000)  DOE_RAW_FDO, DOE_DESIGNATED_FDO
                             Unknown flags 0x08000000
AttachedDevice (Upper) 86b52b58 \Driver\VolSnap
Device queue is not busy.
```

*!devobj*命令显示了该卷设备对象的VPB域。（这里显示的设备对象被命名为HarddiskVolume6）。现在你可以执行*!vpb*命令了：

```
kd> !vpb 818a5da8
Vpb at 0x818a5da8
Flags: 0x1 mounted
DeviceObject: 0x850dcac0
RealDevice:   0x818a5290
RefCount: 3
Volume Label:     BACKUP
```

该命令表明了，此设备对象是由一个文件系统驱动程序挂载的，并且该文件系统驱动程序已经分配给该卷的名称为BACKUP。VPB中的*RealDevice*域又指回了卷设备对象，而*DeviceObject*域则指向被挂载的文件系统设备对象。

文件系统如何识别“按其格式来挂载的卷”呢？它所遵从的约定是，检查该卷的引导纪录，即该卷的第一个扇区中保存的信息。Microsoft文件系统的引导纪录包含一个域，其中存储了文件系统的格式类型。文件系统驱动程序往往检查该域，如果该域指明了这是它所管理的格式，那么，它继续查看引导纪录中存储的其他信息。这些信息往往包含一个文件系统的名称域，以及足够的数据，可以让文件系统驱动程序定位到该卷上关键的元数据文件。例如，只有当类型域是NTFS，名称域是“NTFS”，而且引导纪录描述的关键元数据文件也相符的时候，NTFS才会识别一个卷。

如果一个文件系统驱动程序做了肯定的答复，那么，I/O管理器填充此VPB，并且将含有剩余路径（即\Temp\Test.txt）的打开请求传递给该文件系统驱动程序。文件系统驱动程序利用它的文件系统格式来解释该卷上存储的数据，从而完成此请求。

www.TopSage.com

在一次挂载操作填充了一个卷设备对象的VPB以后，I/O管理器就把随后的针对该卷的打开请求都传递给这一被挂载的文件系统驱动程序。如果没有文件系统驱动程序声明对一个卷负责，那么，Ntoskrnl.exe中内置的一个文件系统驱动程序Raw接受该卷，并且，所有打开该分区上的文件的请求都失败。图10.15显示了一个简化的例子（也就是说，图中省略了文件系统驱动程序与Windows缓存和内存管理器之间的交互），其中说明了针对挂载卷的I/O操作所遵从的路径。

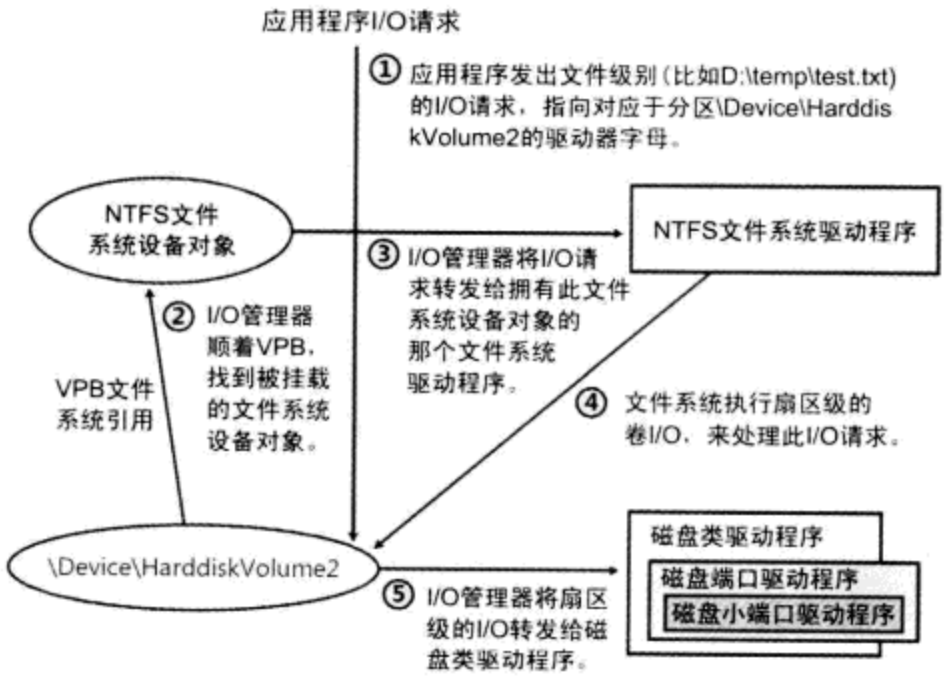


图10.15 挂载卷的I/O的流程

Windows为了使内存使用量尽可能地最小化，它不是让每一个文件系统驱动程序都加载进来（不管它们是否真的管理了哪个或哪些卷），相反地，它使用一个名为**文件系统识别器**（*File System Recognizer*）的代理驱动程序（\Windows\System32\Drivers\Fs_rec.sys），来执行初步的文件系统识别工作。文件系统识别器知道足够多的关于Windows所支持的每一个文件系统格式的信息，因而能够检查一个引导纪录并确定它是否跟某一个Windows文件系统驱动程序关联得起来。当系统引导时，文件系统识别器作为一个文件系统驱动程序向系统注册；当I/O管理器在一个新卷的文件系统挂载操作过程中调用文件系统识别器时，如果与引导纪录相对应的文件系统驱动程序尚未被加载进来，则文件系统识别器加载此文件系统驱动程序。在加载了文件系统驱动程序以后，文件系统识别器将此挂载IRP转发给该驱动程序，让该文件系统驱动程序声明它对该卷的所有权。

对于引导卷，驱动程序是在内核初始化的时候挂载的，除此以外，对于其他绝大多数卷，文件系统驱动程序是在引导序列过程中当Chkdsk文件系统一致性检查应用程序运行的时候挂载的。引导时刻的Chkdsk版本是一个原生应用程序（相对于Windows应用程序），名为Autochk.exe（\Windows\System32\Autochk.exe）。会话管理器（\Windows\System32\Smss.exe）

www.TopSage.com

之所以运行它，是因为在HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute值中，它被指定为一个引导时运行的程序。Chkdsk检查每一个驱动程序字母，以查看与该字母关联的卷是否要求做一致性检查。

针对同一个磁盘发生多次挂载的一种情形是，与可移除介质一起工作之时。Windows文件系统驱动程序通过查询该磁盘的卷标识符来响应介质的变化。如果驱动程序看到卷标识符改变了，则它解除该磁盘已有的挂载，并试图重新挂载该磁盘。

卷的I/O操作

文件系统驱动程序管理在卷上存储的数据，但是它们依赖于卷管理器来跟存储体驱动程序打交道，因而能够传输数据至一个卷所在的磁盘，或者从磁盘上读取数据。文件系统驱动程序通过挂载过程来获得一个卷管理器的卷对象的引用，然后，通过卷对象来发送卷管理器请求。当应用程序想要直接操作一个卷的数据时，它们也可以发送卷管理器请求，以绕过文件系统驱动程序。文件删除恢复（file-undelete）程序是这种程序的一个例子，Windows资源工具箱中的DiskProbe工具也是。

无论何时，当一个文件系统驱动程序或者应用程序向一个代表某个卷的设备对象发送I/O请求时，Windows I/O管理器将该请求（在一个自包含的请求包——IRP中）转发给创建此目标设备对象的卷管理器。因此，如果一个应用程序想要读取系统中第二个卷的引导扇区（在这个例子中是一个简单卷），那么，它打开设备对象\Device\HarddiskVolume2，然后向该对象发送一个请求，要求读取该设备上从零偏移处开始的512字节。I/O管理器将该应用程序的请求，以IRP的形式发送给拥有此设备对象的卷管理器，并通知它，此IRP是针对HarddiskVolume2设备的。

因为卷提供了逻辑上的方便，Windows利用卷来代表一个或多个物理磁盘上的连续区域，所以，卷管理器必须将相对于一个卷的偏移量转译为相对于一个磁盘开始处的偏移量。如果Volume2所包含的分区是从磁盘中的第4096个扇区开始的，那么，卷管理器在把IRP请求传递给磁盘类驱动程序以前，先要利用该值来调整IRP的参数，以指定一个适当的偏移。磁盘类驱动程序使用一个小端口驱动程序来完成物理磁盘I/O，并且将所请求的数据读入到该IRP指定的应用程序缓冲区中。

通过卷管理器的一些例子，可以有助于澄清它在处理针对多分区卷的请求时所扮演的角色。如果一个条带卷是由两个分区组成的，则不妨称为分区1和分区2，它们由设备对象\Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume3来表示，如图10.16所示，如果管

www.TopSage.com

理员为该条带卷分配了驱动器字母D:，那么，I/O管理器定义链接\Global??\D:，指向\Device\HarddiskDmVolumes*ComputerName*Dg0\Volume3。这里*ComputerName*是计算机的名称。回忆前面曾经讲述过的，该链接也是一个符号链接，它指向PhysicalDmVolumes目录中对应的卷设备对象（在这个例子中是BlockVolume3）。DMIO设备对象截取了针对\Device\HarddiskDmVolumes\PhysicalDmVolumes\BlockVolume3的文件系统磁盘I/O，并且DMIO在把此I/O请求传递给磁盘类驱动程序以前，首先调整该请求。DMIO所做的调整是，配置该请求，让它指向该请求的目标条带（在分区1或分区2上）的正确偏移位置。如果此I/O跨越了该卷的两个分区，则DMIO必须发出两个辅助I/O请求，每个磁盘各一个。

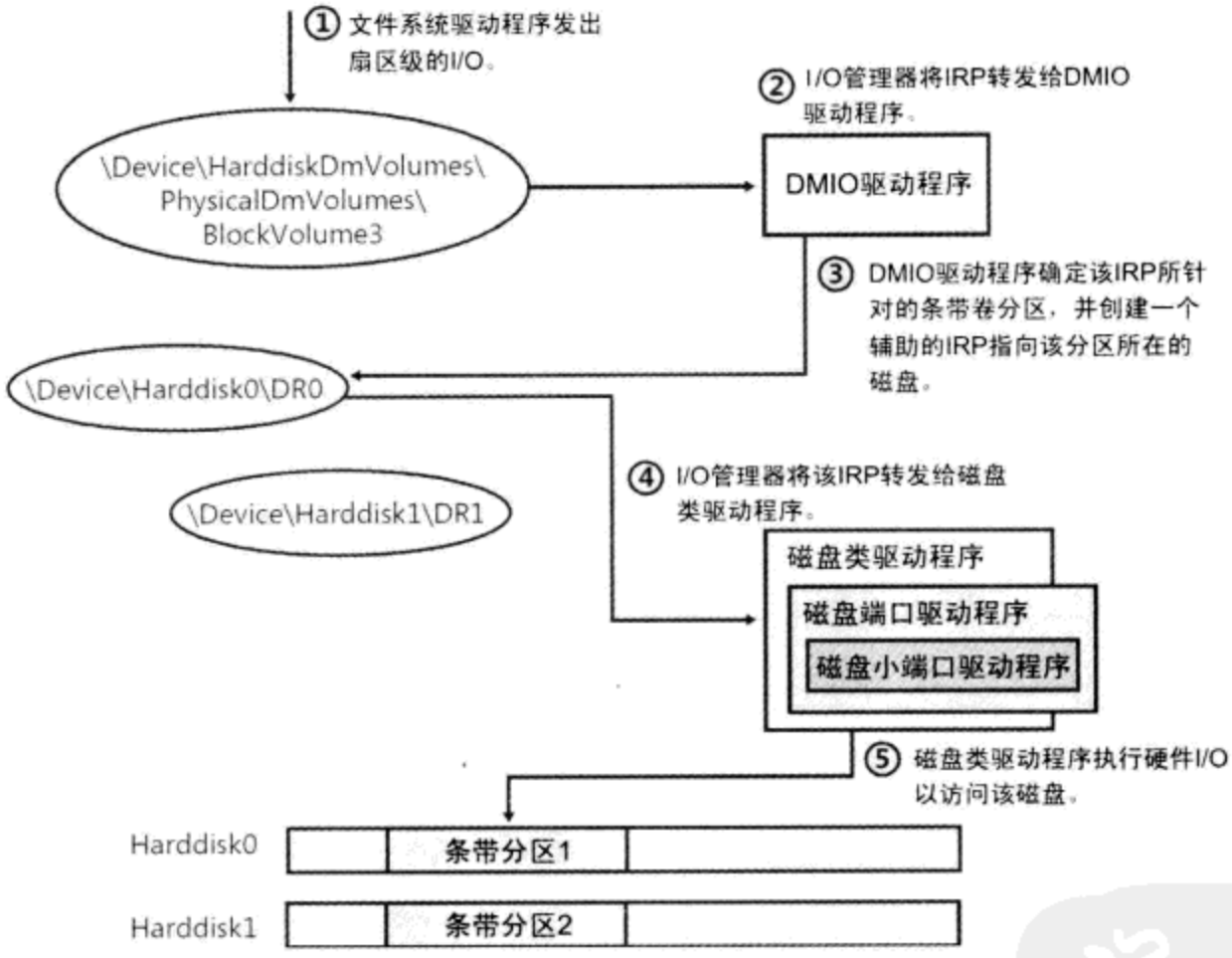


图10.16 DMIO I/O操作

如果是写一个镜像卷，则DMIO把每个请求分裂开，因而，镜像卷的每一半都会接收到此写操作。对于镜像卷的读操作，DMIO只从镜像卷的一半执行读操作，当一个读操作失败时，再依赖另一半。

www.TopSage.com

虚拟磁盘服务

做存储产品（比如RAID适配器、硬盘或者存储阵列）的公司必须实现自定义的应用程序来安装和管理它们的设备。为不同的存储设备使用不同的管理应用程序，从系统管理的角度来说，这样做有很明显的缺点。这些缺点包括：用户需要学习多个用户界面、不能使用标准的Windows存储管理工具来管理第三方存储设备。

Windows Server 2003引入了虚拟磁盘服务（或者VDS，位于\Windows\System32\Vds.exe）。VDS提供了统一的高层存储接口，因而，管理员可以通过同样的用户界面来管理不同厂商的存储设备。图10.17显示了VDS。VDS导出了一个基于COM的API，这使得应用程序和脚本能够创建和格式化磁盘，能够查看和管理硬件RAID适配器。例如，一个工具可以使用VDS API来查询那些“映射到某个RAID逻辑单元号（LUN，Logical Unit Number）的物理磁盘”的列表。Windows的各种磁盘管理工具，包括磁盘管理MMC加载件，以及Diskpart和Diskraid（在Windows Server 2003 Deployment Kit中可以使用）命令行工具，都使用了VDS API。

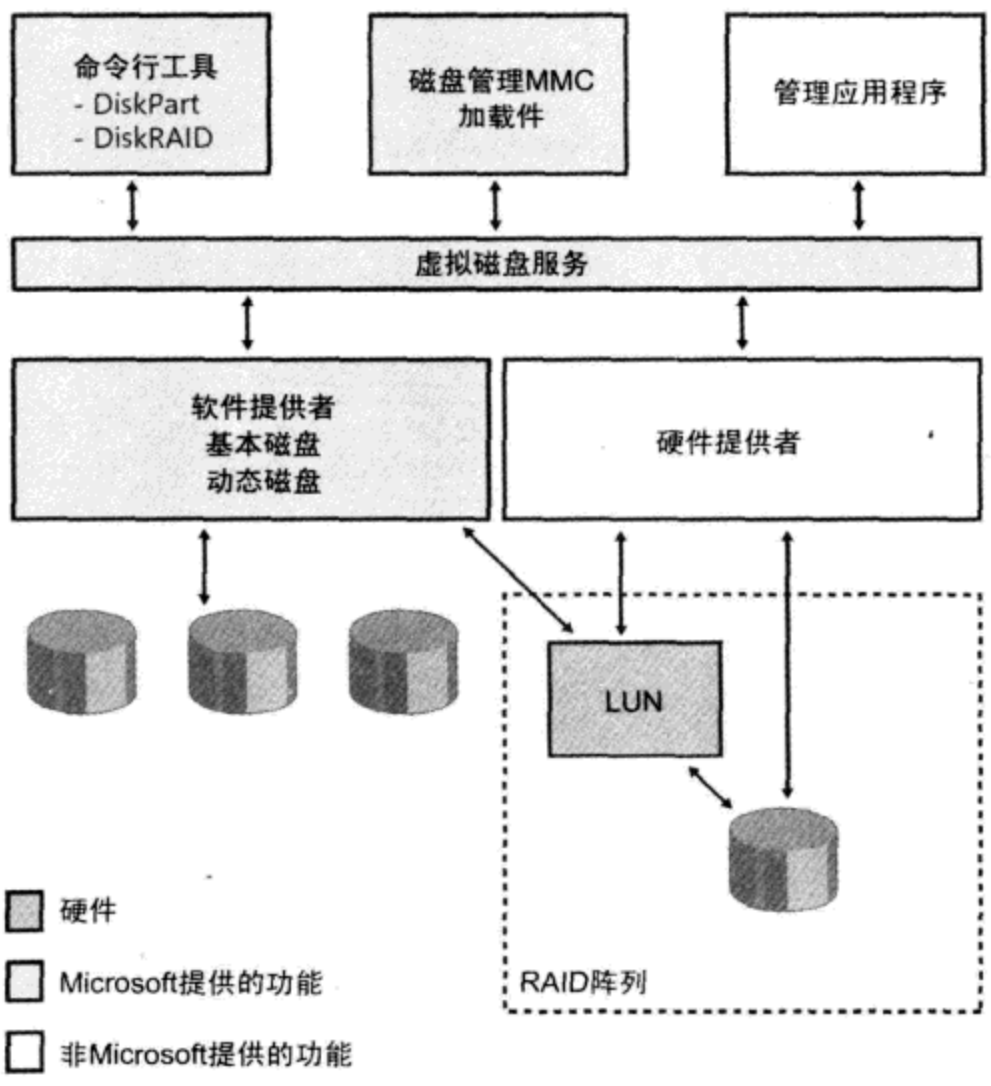


图10.17 VDS服务体系结构

www.TopSage.com

VDS提供了两个接口：一个针对软件提供者（software provider）；另一个针对硬件提供者（hardware provider）。

- 软件提供者实现了针对高层存储抽象（比如磁盘、磁盘分区和卷）的接口。譬如，这些接口支持的操作有：创建、扩展和删除卷；加入或打破镜像卷；格式化和分配驱动器字母等。VDS在HKLM\System\CurrentControlSet\Services\Vds\SoftwareProviders中寻找已注册的软件提供者。Windows Server 2003包含了VDS动态磁盘提供者（\Windows\System32\Vdsdyndr.dll，以连接（或管理）动态磁盘；也包含了VDS基本磁盘提供者（\Windows\System32\Vdsbas.dll），以连接（或管理）基本磁盘。
- 硬件厂商以DLL的形式实现VDS硬件提供者，它们注册在HKLM\System\CurrentControlSet\Services\Vds\HardwareProviders下面；它们负责将与设备无关的VDS命令翻译成针对其硬件的命令。硬件提供者考虑到了存储子系统（比如硬件RAID阵列或者适配器卡）的管理所支持的操作，包括创建、扩展、删除、屏蔽（masking）和暴露（unmasking）LUN。

当应用程序发起一个它跟VDS API之间的连接，并且VDS服务尚未被启动时，宿纳RPC服务的Svchost进程启动VDS加载器进程（\Windows\System32\Vdsldr.exe），该进程启动VDS服务进程，然后退出。当VDS API的最后一个连接关闭时，VDS服务进程即退出。

卷影像（shadow）拷贝服务

许多备份工具有一个限制，它与已打开的文件有直接关系。如果一个应用程序已经以独占访问方式打开了一个文件，那么，备份工具无法获得对该文件内容的访问权。即使备份工具能够访问一个已打开的文件，它仍然要承担“创建一个内容不一致的备份”的风险。考虑这样一个应用程序，它先在一个文件的开始处更新该文件，然后在结束处更新该文件。如果备份工具在此操作过程中保存该文件，那么，它记录下来的文件映像可能反映了应用程序做修改之前的文件开始部分，以及修改之后的结束部分。如果该文件后来又被恢复，那么，应用程序可能认为整个文件已被破坏，因为它打算要处理的情形可能是开始部分已被修改，结束部分未被修改，而并非相反的情形。这两个问题说明了为什么绝大多数备份工具跳过那些已打开的文件。

Windows XP引入了卷影像拷贝服务（\Windows\System32\Vssvc.exe），这使得内置的备份工具能够记录下所有文件的一致的视图，包括那些已打开的文件。卷影像拷贝服务可以被看作是一个可扩展备份核心的命令中心，它允许独立软件厂商（ISV）插入“写者（writer）”和“提供者（provider）”。写者是一个软件组件，它使得那些能感知影像拷贝（shadow-copy-aware）的应用程序可以接收**冻结**和**解冻**通知，从而确保其数据文件的备份拷贝是内部一致的；而提供者使得那些具有特殊存储方案的ISV能够与影像拷贝服务集成在一起。例如，一个镜像存储设备的ISV可能将影像拷贝定义成一个分裂的镜像卷中已冻结的那一半。图10.18显示了影像拷贝服务、写者和提供者之间的关系。

www.TopSage.com

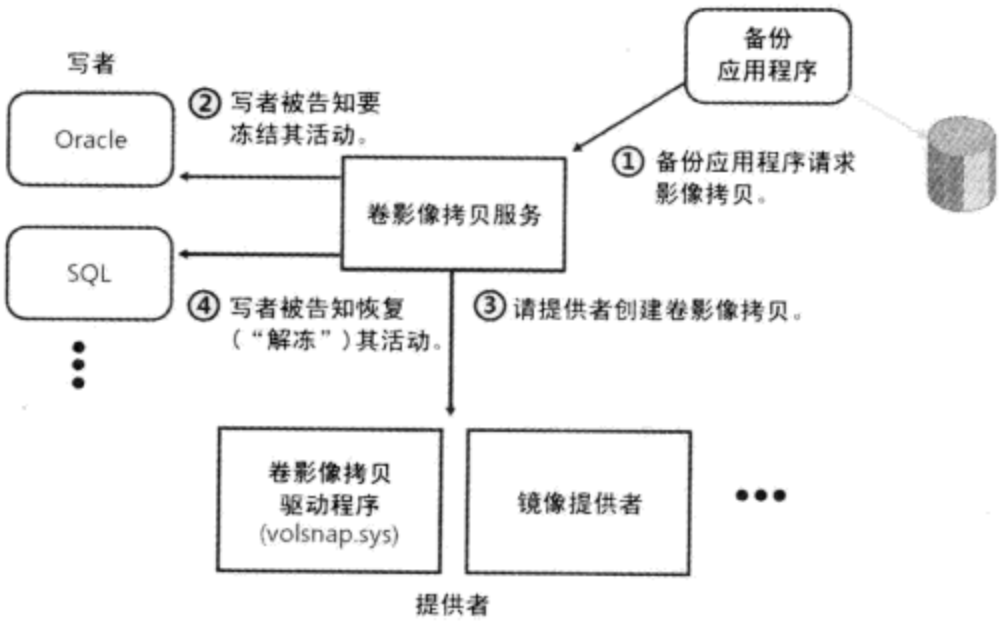


图10.18 影像拷贝服务、写者和提供者

Microsoft影像拷贝提供者（\Windows\System32\Drivers\Volsnap.sys）是一个随Windows一起发布的提供者，目的是实现基于软件的卷快照功能。它是一种被称为存储过滤型驱动程序（storage filter driver）的驱动程序，位于文件系统驱动程序和卷驱动程序之间（卷驱动程序展示了“一个逻辑驱动器所对应的磁盘扇区”的视图），因而，它可以看到指向卷的I/O请求。备份工具开始一个备份操作时，指示Microsoft影像拷贝提供者驱动程序，对于那些包含了所要记录的文件和目录的卷，创建一个卷影像拷贝。该驱动程序冻结住所有指向目标卷的I/O，并为每一个卷创建一个影像卷。例如，如果一个卷在对象管理器名字空间中的名称是\Device\HarddiskVolume0，那么，影像卷将有一个类似于\Device\HarddiskVolumeShadowCopy*N*这样的名称，这里*N*是一个惟一ID。



实验：查看Microsoft影像拷贝提供者的设备对象

在Windows XP和Windows Server 2003系统中，你可以在内核调试器中看到附载在每个卷设备上的Microsoft影像拷贝提供者驱动程序的设备对象。每个系统至少有一个卷，下面的命令显示了一个系统中第一个卷的设备对象：

```
0: kd> !devobj \device\harddiskvolume1
Device object (86b9daf8) is for:
 HarddiskVolume1 \Driver\Ftdisk DriverObject 86b6af38
Current Irp 00000000 RefCount 2 Type 00000007 Flags 00001050
Vpb 86b90008 Dacl e1000384 DevExt 86b9dbb0 DevObjExt 86b9dc98 Dope 86ba33e8 DevN
ode 86b71320
ExtensionFlags (0xa8000000)  DOE_RAW_FDO, DOE_DESIGNATED_FDO
                             Unknown flags 0x08000000
AttachedDevice (Upper) 86b687f8 \Driver\VolSnap Device queue is not busy.
```


www.TopSage.com

*!devobj*命令的输出中的AttachedDevice域显示了一个设备对象的地址，以及所有者驱动程序（附载到[过滤]所要检查的设备对象）的名称。对于卷设备对象，你应该会看到一个属于Volsnap驱动程序的设备对象，如上面的例子输出中所示。

备份工具并非在原始的卷上打开文件进行备份，而是打开影像卷上的文件。影像卷代表了一个卷在某个时间点上的视图，所以，无论何时，卷影像拷贝驱动程序看到一个针对原始卷的写操作时，它把将要被改写的那些扇区的一份拷贝，读入到一个与对应的影像卷相关联的、由页面文件支撑的内存区中。对于已修改扇区的影像卷读操作，它从这一内存区读取数据；对于未修改区域的读操作，它从原始卷读取数据。因为备份工具并不保存页面文件或者每个卷上用于系统管理的\System Volume Information目录的内容，所以，快照驱动程序使用碎片整理API来确定这些文件和目录的位置，而不记录它们的变化情况。Windows XP和Windows Server 2003上的备份工具借助于这一影像拷贝设施，克服了与打开文件相关的备份问题。

图10.19示出了当应用程序访问一个卷，以及备份应用程序访问该卷的影像卷时的行为。一个应用程序在快照时间过了以后写一个扇区时，Volsnap驱动程序做一份备份拷贝，就像在图中它对于卷C:的扇区a、b和c所做的那样。随后，当一个应用程序读取扇区c时，Volsnap直接将读操作交给卷C:，但是，当备份应用程序读取扇区c时，Volsnap从快照中读取该扇区。当一个读操作是针对未修改的扇区（比如扇区d）时，Volsnap直接将读操作交给该卷。

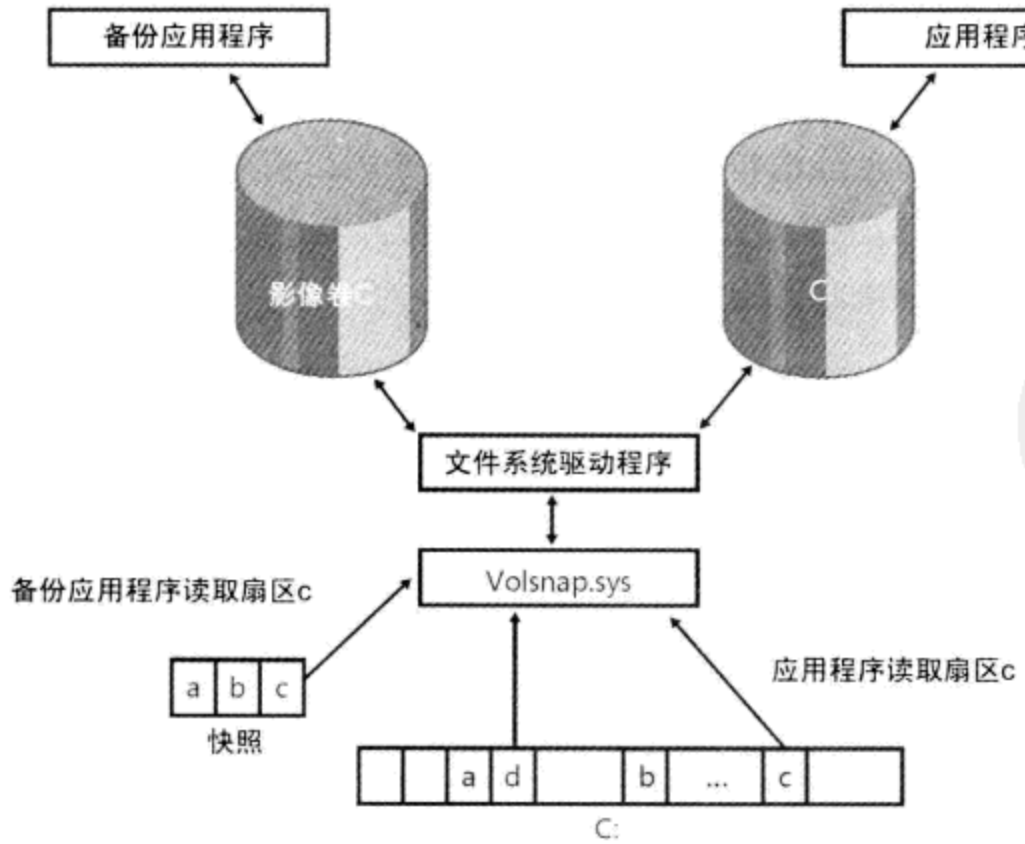


图10.19 Volsnap的操作

www.TopSage.com

实验：查看影像卷的设备对象

通过如下操作，你可以在对象管理器名字空间中看到影像卷设备对象的存在：运行Windows备份应用程序（在Start菜单的Accessories文件夹的System Tools中），并选择足够多的备份数据，以使得备份过程要花费足够长的时间，从而你可以运行Winobj，并且在\Device子目录中看到这些对象。



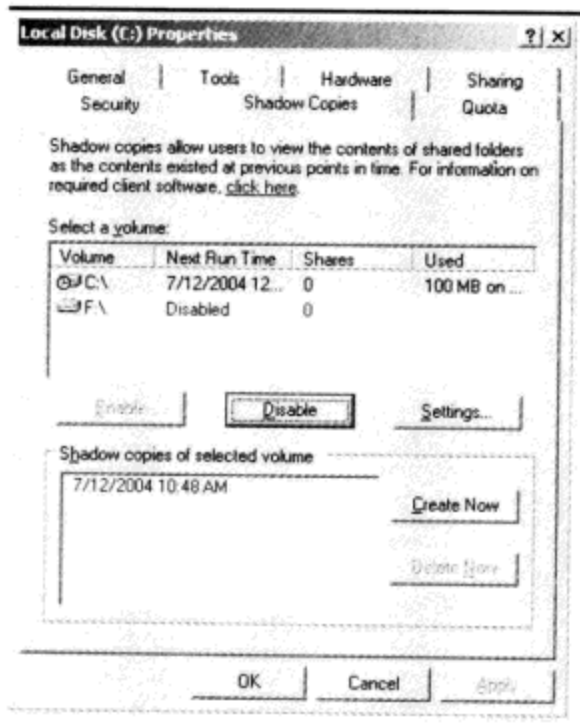
文件系统驱动程序必须处理两个与影像拷贝相关的I/O控制请求（IOCTL），以帮助确保快照的一致性：IOCTL_VOLSNAP_FLUSH_AND_HOLD_WRITES和IOCTL_VOLSNAP_RELEASE_WRITES。这两个IOCTL的名称已经自己解释了其含义。影像拷贝API将这些IOCTL发送给要取快照的逻辑驱动器，因而，当取了影像拷贝时，在快照之前发起的任何修改操作都已经完成了，从而使得从影像拷贝中记录下来的文件数据在时间上是一致的。

共享文件夹的影像拷贝

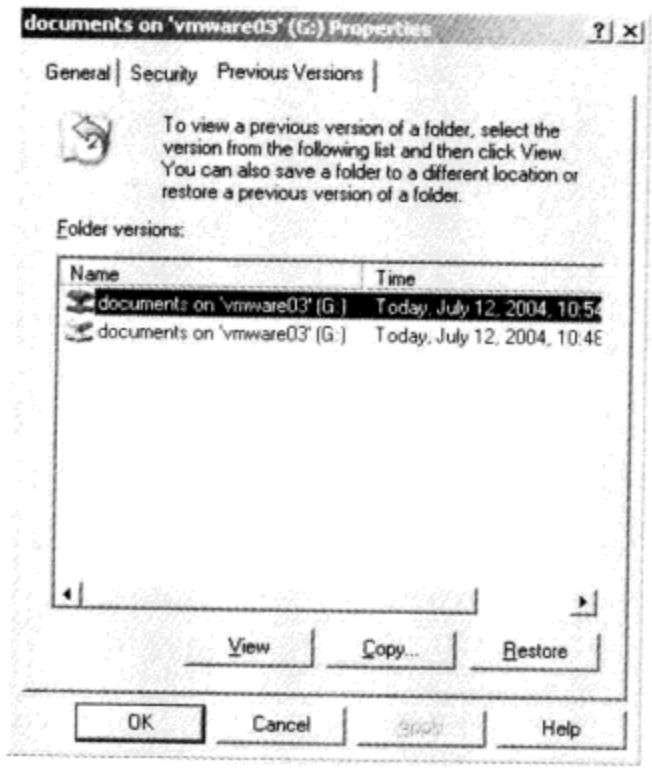
Windows Server 2003利用卷影像拷贝来提供这样一种功能：允许最终用户访问卷的备份版本，从而他们可以恢复那些可能已经被删除或修改的文件和文件夹的老版本。这一特性减轻了系统管理员的负担，要不然的话，他们必须将备份介质加载进来并代表最终用户访问以前的版本。

在Windows Server 2003系统上，卷的属性对话框包含一个名为“Shadow Copies”的标签页，在这个页面中，管理员可以为一个卷创建预定的快照，如下图中的屏幕截图所示。管理员也可以限制这些快照所消耗的空间数量，因而，系统删除掉老的快照以便符合空间的限制。

www.TopSage.com



想要利用共享文件夹的影像拷贝能力的客户系统必须安装一个称为“Previous Versions Client”的Explorer扩展组件，它随Windows Server 2003一起发行，位于\Windows\System32\Clients\Twclient目录下，从Microsoft的Web站点上也可以下载得到（在Windows XP Service Pack 2和更高版本上，随带了此扩展组件）。当一个安装了Explorer影像拷贝扩展组件的客户Windows系统映射了一个共享体，并且该共享体位于一个存在快照的卷上时，此共享体中的文件夹和文件的属性对话框上，将会出现一个名为Previous Versions的标签页。Previous Versions页面显示了该服务器上存在的快照列表，它允许用户查看或者拷贝一个文件或文件夹在以前的快照中的数据。



www.TopSage.com

10.4 本章总结

在本章中，我们回顾了Windows磁盘存储管理的组件和操作，以及在磁盘上的组织结构。在下一章中，我们将深入研究Windows的缓存管理器，这是一个执行体组件，它对于文件系统驱动程序（如果它们挂载了本章中展示的卷类型）的操作，是一个不可或缺的组成部分。

www.TopSage.com

Cache Manager

第 11 章　缓存管理器

缓存管理器是一组内核模式的函数和系统线程，它们与内存管理器协同工作，为所有的Windows文件系统驱动程序（本地的和网络的）提供了数据缓存的能力。在本章中，我们将解释：缓存管理器是如何工作的，包括它的关键内部数据结构和函数；在系统初始化时刻它是如何确定大小的；它如何跟操作系统的其他部件进行交互；你如何通过性能计数器来观察它的行为。我们也将讲述Windows的*CreateFile*函数中的5个会影响文件缓存的标志。



注　除了在解释缓存管理器的工作原理时涉及一点深度介绍以外，本章中并没有对缓存管理器的内部函数进行专门的描述。缓存管理器的编程接口文档位于Windows可安装文件系统（IFS，Installable File System）开发工具箱中。有关IFS开发工具箱的更多信息，请参见*http://www.microsoft.com/whdc/devtools/ifskit*。

11.1　缓存管理器的关键特性

缓存管理器具有以下几个关键的特性。

- 支持所有的文件系统类型（包括本地的和网络的），因此无需每一个文件系统单独实现其自己的缓存管理代码。
- 使用内存管理器来控制哪些文件的哪些部分位于物理内存中（用户进程和操作系统对于物理内存的需求之间的一种平衡）。
- 以虚拟块（即在一个文件内部的偏移）为基础来缓存数据——相对于许多其他的缓存系统，它们以逻辑块（即在一个磁盘卷内部的偏移）为基础来缓存数据，这使得可以在不涉及文件系统驱动程序的情况下进行智能预读和高速访问缓存（这种缓存方法称为**快速*I/O***，本章后面将会讲述）。
- 允许应用程序在打开文件时传递“提示（hint）”（比如随机还是顺序的访问、临时文件的创建，等等）。
- 支持可恢复的文件系统（例如，使用事务日志的文件系统），从而在系统失败以后还可以恢复数据。

www.TopSage.com

虽然我们将在本章中通篇谈论如何在缓存管理器中使用这些特性，但是，在这一节中，我们将向你介绍这些特性背后的一些概念。

单个中心化的系统缓存

有些操作系统依赖于每一个单独的文件系统来缓存数据，这种做法要么导致在操作系统中有重复的缓存管理代码和内存管理代码，要么导致可被缓存的数据种类有所限制。相反地，Windows提供了一个中心化的缓存设施，它可以缓存所有外部保存的数据，无论它们是在本地硬盘、软盘、网络文件服务器上，还是在CD-ROM上。任何数据都可以被缓存，不管它是用户数据流（某一个文件的内容，以及对该文件的正在进行中的读和写活动），还是文件系统**元数据**（*metadata*）（比如目录和文件头）。你在本章后面将会发现，Windows访问缓存的方法取决于被缓存的数据的类型。

内存管理器

缓存管理器的一个非同寻常的方面是，它从来不知道有多少被缓存的数据真的在物理内存中。这一声明可能听起来非常怪异，因为一般而言，缓存的用途是将频繁被访问的数据的一个子集记录在物理内存中，作为提高I/O性能的一种方法。缓存管理器之所以不知道有多少数据在物理内存中，是因为，它访问数据的做法是将文件的视图映射到系统虚拟地址空间中，它使用了标准的**内存区对象**（*section object*）（按照Windows API的术语来说，是**文件映射对象**[*file mapping object*]）。（内存区对象是内存管理器的基本原语，在第7章中有详细的解释）。随着这些映射视图中的地址陆续被访问，内存管理器把那些不在物理内存中的数据块换到物理内存中。当内存需求吃紧时，内存管理器又把缓存中的数据换出去，让它们回到在缓存中已被打开的（被映射的）文件中。

通过使用映射文件并且在虚拟地址空间的基础上进行缓存，缓存管理器避免产生读或写I/O请求包（IRP）来访问当前正在缓存的文件的数据。相反地，它简单地使用缓存文件被映射部分的虚拟地址进行数据拷贝，并且依赖于内存管理器，在必要的时候将数据换入（或换出）内存。这一过程使得内存管理器可以进行全局平衡："有多少内存给予系统缓存，有多少内存给予用户进程"（缓存管理器也会发起I/O，比如本章后面将要讲述的延迟写操作；然而，它调用内存管理器来写页面）。而且，正如你在下一节中将要学到的那样，这种设计使得那些打开了被缓存文件的进程可以看到同样的数据，就如同它们将同样的文件映射到它们的用户地址空间中一样。

缓存一致性

缓存管理器的一个重要功能是，确保任何进程在访问被缓存的数据时，总是得到该数据最新的版本。当一个进程打开文件（因此该文件被缓存了）而另一个进程直接将该文件映射到它的地址空间中（使用Windows的*MapViewOfFile*函数）时，问题可能会发生。这一问题在Windows

www.TopSage.com

下不会发生，因为缓存管理器和映射该文件到用户地址空间中的那些应用程序使用了同样的内存管理文件映射服务。因为内存管理器保证，每个被映射的文件，不管有多少内存区对象或映射的视图，它都只有一份内部表示，所以，它把一个文件的所有视图（即使它们有重叠）都映射到物理内存中的一组页面上，如图11.1所示（有关内存管理器如何与映射文件一起工作的更多信息，请参见第7章）。

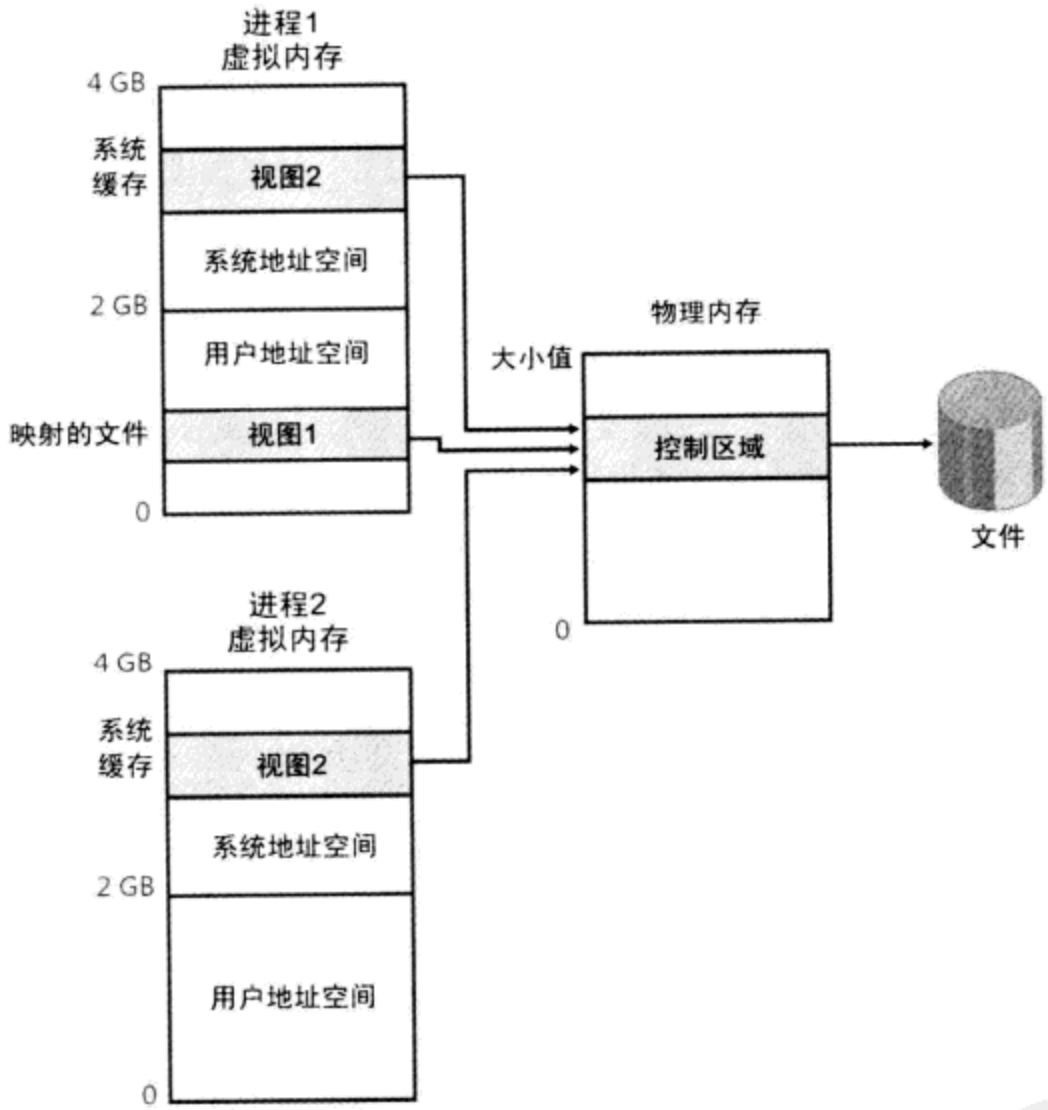


图11.1 一致的缓存方案

所以，举例来说，如果进程1有一个视图，它指向其用户地址空间中一个已被映射的文件，同时，进程2通过系统缓存也在访问此同样的视图，那么，进程2将会看到进程1所做的任何改变，而且是在改变发生的时候，而并非这些变化被刷新之时。内存管理器不会刷新**所有的**用户映射的页面，而只刷新那些它知道已经被写过的页面（因为这些页面已经被设置了修改位）。因此，在Windows中，任何一个进程，它在访问一个文件时，总是看到该文件的最新版本，即便是有些进程通过I/O系统来打开该文件，而其他进程使用Windows的文件映射函数来把该文件映射到它们的地址空间中。

www.TopSage.com



> **注** 缓存一致性对于网络重定向器比对于本地文件系统，要更加困难，因为网络重定向器必须实现额外的刷新和清除操作，以确保在访问网络数据时候的缓存一致性。参见第12章关于机会锁（opportunistic locking）的描述，机会锁是Windows的分布式缓存一致性机制。

虚拟块缓存

许多操作系统的缓存管理器（包括Novel NetWare、OpenVMS和老的UNIX系统）都在**逻辑块**（*logical block*）的基础上缓存数据。通过这种方法，缓存管理器跟踪、记录下一个磁盘分区的哪些块位于缓存中。相反地，Windows的缓存管理器使用了一种称为**虚拟块缓存**（*virtual block caching*）的方法，它跟踪、记录了哪些**文件**的哪些部分位于缓存中。缓存管理器能够监视这些文件部分；它使用内存管理器中专门的系统缓存例程，把文件的256KB的视图映射到系统虚拟地址空间中。这种做法有以下关键的好处。

- 它使得智能预读成为可能。因为缓存管理器记录了哪些文件的哪些部分位于缓存中，所以，它能够预测调用者接下来可能要往哪儿读。
- 它使得I/O系统能够避免“到文件系统中请求那些已经在缓存中的数据”（快速I/O）。因为缓存管理器知道哪些文件的哪些部分已经在缓存中，所以，它可以返回已被缓存的数据的地址，以满足一个I/O请求，而无需再调用文件系统。

有关智能预读和快速I/O是如何工作的，本章后面提供了详细的说明。

流式缓存机制

缓存管理器也被设计成可进行**流式缓存**（*stream caching*），以区别于文件缓存。流是指一个文件内部的字节流。有些文件系统，比如NTFS，允许一个文件包含多个流。缓存管理器能够独立地缓存每一个流，从而可以适应这样的文件系统。NTFS充分利用了这一特性，它将主文件表（master file table，在第12章中讲述）组织成流，并且缓存这些流。实际上，与其说缓存管理器缓存的是文件，不如说它缓存的是流（所有的文件至少有一个数据流），这些流通过一个文件名和一个流名（如果该文件中存在多个流的话）来标识。

对可恢复文件系统的支持

像NTFS这样的可恢复文件系统被设计成，在经历了一次系统失败以后能够重构磁盘卷结构。这种能力意味着，在系统失败时尚未完成的那些I/O操作，在系统重新启动时，必须要么整个操作都完成，要么整个从磁盘上退回去。半完成的I/O操作可能会破坏一个卷，甚至导致

www.TopSage.com

整个卷变得不可访问。为了避免这一后果，一个可恢复的文件系统需要维护一个日志文件，它在每次做一个涉及文件系统结构（文件系统的元数据）的修改以前，都要在日志文件中记录下此更新操作。如果系统失败了，正在进行之中的卷修改操作被打断了，那么，这一可恢复文件系统利用日志中保存的信息，重新发出这些卷更新操作。



> **注** 术语“*元数据*（*metadata*）”仅应用于文件系统结构中的以下变化：文件和目录的创建、重命名和删除。

为了保证卷恢复能够成功地进行，日志文件中每一个描述了卷更新操作的纪录必须在此更新操作本身被应用到该卷上以前被完整地写到磁盘上。因为磁盘写操作是被缓存的，所以，缓存管理器和文件系统必须协同工作，以确保下面的动作按照以下顺序进行。

1. 文件系统写一条日志文件纪录，此记录描述了它将要进行的卷更新动作。
2. 文件系统调用缓存管理器，以便将日志文件纪录刷新到磁盘上。
3. 文件系统将卷的更新数据写到缓存中。也就是说，它修改其被缓存起来的元数据。
4. 缓存管理器将更改的元数据刷新到磁盘上，更新卷的结构（实际上，日志文件纪录在被刷新到磁盘上以前是成批处理的，它们也是对卷的修改）。

一个文件系统将数据写到缓存中时，它可以提供一个**逻辑序列号**（LSN，*logical sequence number*），以标识出“在其日志文件中对应于此次缓存更新”的纪录。缓存管理器跟踪这些编号，记录下与缓存中每个页面相关联的最低和最高LSN（分别代表了最老的和最新的日志文件纪录）。而且，受事务日志纪录保护的数据流被NTFS标记为“不可写出（no write）”，因而，映射页面写出器不会在这些页面对应的日志纪录被写到磁盘上以前，将这些页面写出去（当映射页面写出器看到有这种标记的页面时，它将该页面移到一个特殊的列表中，以后缓存管理器会在恰当的时候刷新该页面列表，比如当延迟写出器活动的时候）。

当缓存管理器准备把一组脏页面刷新到磁盘上时，它确定与“这些要被刷新的页面”关联的最高LSN，并且将该编号报告给文件系统。然后，文件系统可以往回调用缓存管理器，并指示它刷新日志文件数据，直至此LSN所代表的那一点。缓存管理器刷新日志文件到达此LSN**以后**，它再把对应的卷结构更新数据刷新到磁盘上，因此确保在实际动作发生之前记录下了将要发生的动作。文件系统和缓存管理器之间的这些交互过程保证了磁盘卷在经历一次系统失败之后还能够恢复过来。

www.TopSage.com

11.2 缓存的虚拟内存管理

因为Windows系统的缓存管理器是按照虚拟地址为基础来缓存数据的，所以，它被赋予系统虚拟地址空间中的一个区域（而不是物理内存区域）来进行管理。然后，缓存管理器将每个地址空间区域分成256KB的槽，称为**视图**（*view*），如图11.2所示（关于系统空间布局结构的详细描述，请参见第7章）。

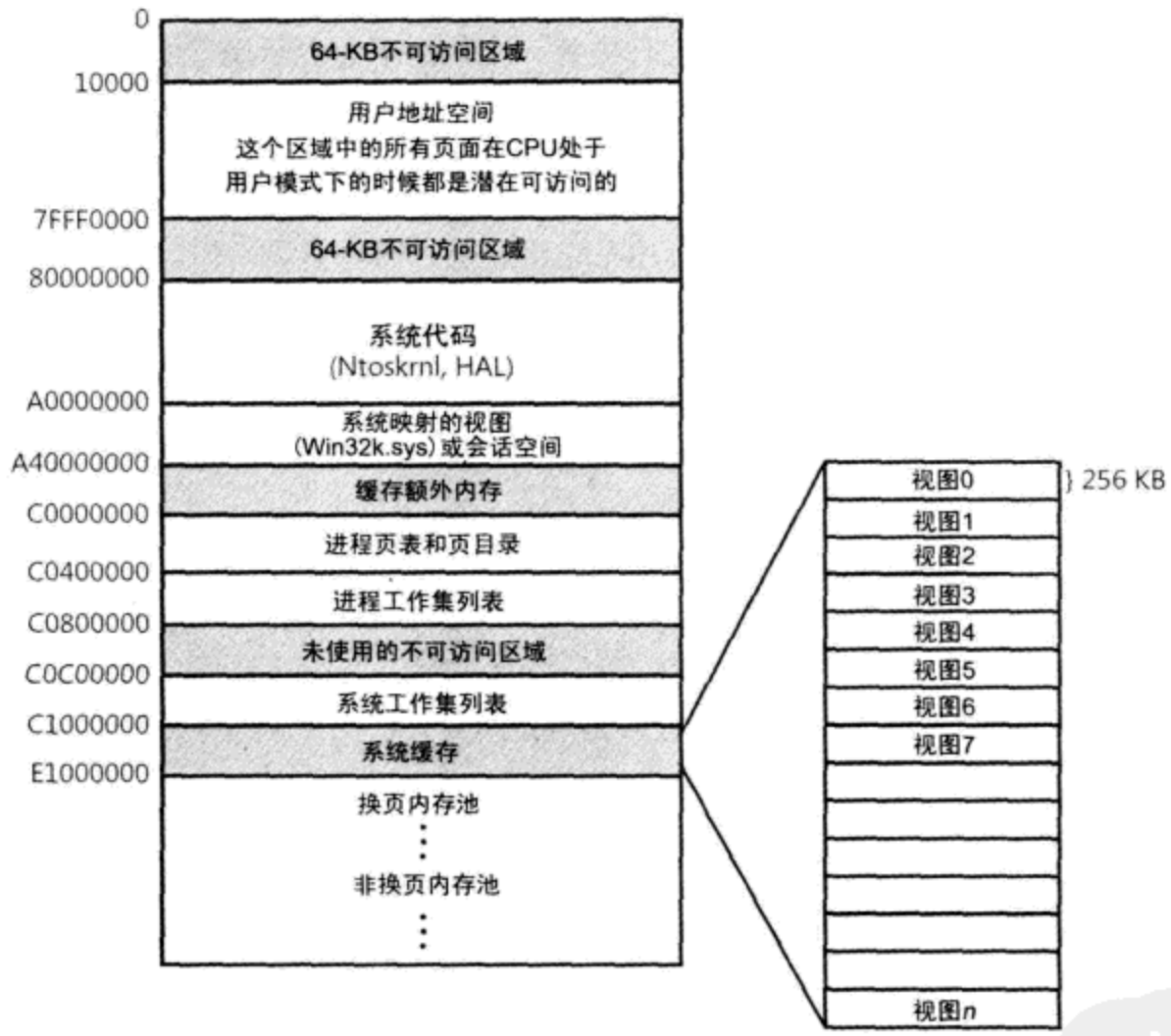


图11.2 系统缓存地址空间

在一个文件的第一个I/O（读或写）操作时，缓存管理器把该文件中包含所请求数据的256KB对齐的区域，映射到系统缓存地址空间中的一个空闲槽中。例如，如果从300 000字节偏移开始，有10个字节被读入到一个文件中，那么，被映射的视图将从偏移262 144处开始（即该文件第二个256KB对齐的区域），视图的宽度为256KB。

缓存管理器按照轮循（round-robin）原则将文件的视图映射到系统缓存的地址空间中。也就是说，第一个请求的视图被映射到第一个256KB的槽中，第二个视图被映射到第二个256KB的槽中，以此类推，如图11.3所示。在这个例子中，文件B被首先映射，其次是文件A，再是

www.TopSage.com

文件C，所以，文件B被映射的数据块占据了缓存中的第一个槽。注意，文件B只有前面的256KB部分已被映射，这是由于该文件只有这部分已被访问过；而且，虽然文件C只有100KB（小于系统缓存中一个视图的大小），但它也要求在缓存中有它自己的256KB的槽。

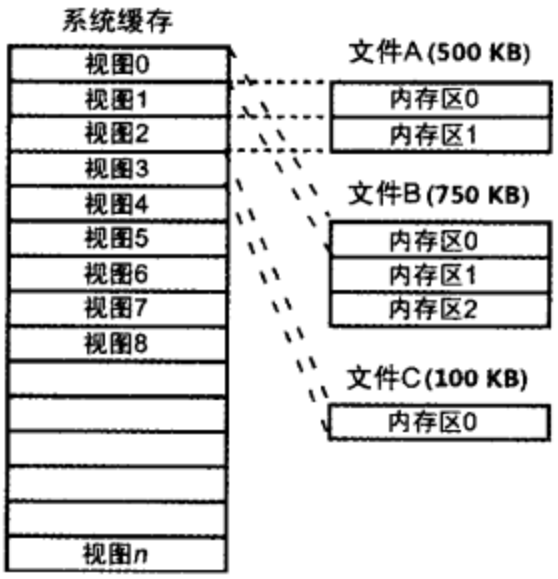


图11.3 各种不同大小的文件被映射到系统缓存中

缓存管理器保证，只有当一个视图是活动的（active）时候，它才被映射（不过，当这些视图变成不活动以后，它们仍然是被映射的）。然而，只有在一个文件的读或写操作过程中，其视图才被标记为活动的。除非一个进程在打开文件、调用*CreateFile*时指定了FILE_FLAG_RANDOM_ACCESS标记，否则，缓存管理器在为该文件映射新的视图时，它将解除不活动视图的映射关系。这些被解除映射的视图的页面被转送到备用列表或修改列表中（取决于它们是否已经被修改），而且，因为内存管理器专门为缓存管理器导出了一个特殊的接口，所以，缓存管理器可以指示这些页面被放在这些列表的末尾或开头。如果在打开文件时设置了FILE_FLAG_SEQUENTIAL_SCAN标记，则这些文件的视图所对应的页面被移到列表的开头，而所有其他的页面被移到列表的末尾。这种方案可以鼓励重用那些属于“被顺序执行读操作的文件”的页面，并且也特别防止大文件的拷贝操作影响到较大区域的物理内存。

如果缓存管理器需要映射一个文件的视图，而系统缓存中又没有足够的空闲槽，那么，它将解除最早被映射的不活动视图，并使用该槽。如果没有视图可使用，则返回一个I/O错误，表明没有足够的系统资源可以用来完成该操作。既然只有在读或写操作过程中视图才被标记为活动的，那么，这种情形绝对是不可能的，因为要想让这种情形发生，数千个文件必须被并发访问。

www.TopSage.com

11.3 缓存的大小

在下面的小节中，我们将解释Windows如何计算系统缓存的大小，包括虚拟的和物理的。由于大多数计算都跟内存管理有关，因此，系统缓存的大小依赖于许多因素，包括内存的大小，以及正在运行的Windows是哪个版本。

LargeSystemCache

你将会看到，注册表值HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\Memory Management\LargeSystemCache同时影响到了系统缓存的虚拟大小和物理大小。在Windows 2000 Professional和Windows XP上，该注册表值默认为0；在Windows服务器系统上，该值默认为1。在Windows 2000服务器系统上，你可以通过修改文件服务器服务的属性，以GUI的方式来调整该注册表值（打开一个网络连接的属性，双击File and Printer Sharing for Microsoft Networks）。尽管在Windows 2000 Professional上该服务也存在，但是，它的参数无法调整。图11.4显示了在Windows 2000 Server系统上，你可以使用该对话框来修改“在服务器网络服务中，为本地和网络应用程序所分配的内存数量”。

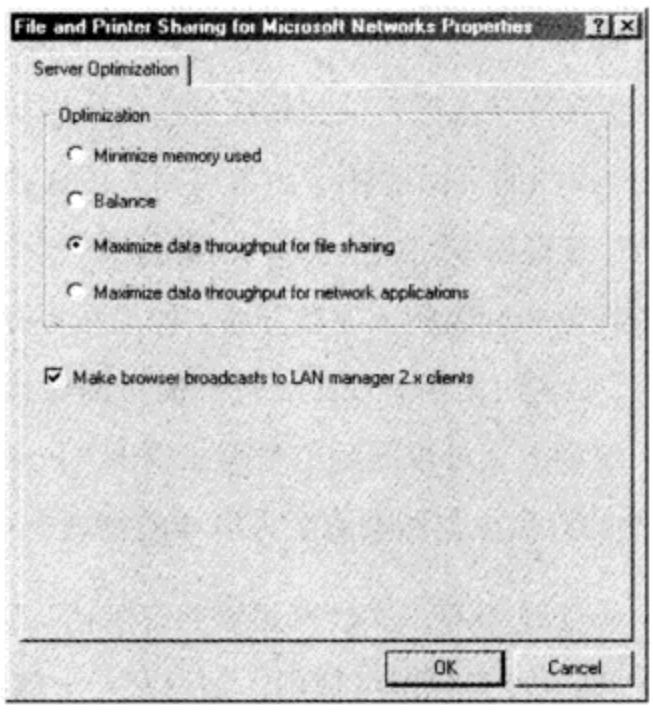


图11.4 File and Printer Sharing for Microsoft Networks Properties对话框，可用于修改Windows 2000服务器网络服务的属性

对于安装了终端服务的服务器系统，图11.4中显示的设置“Maximize data throughput for file sharing”是默认的，此时LargeSystemCache值为1。

www.TopSage.com

选择任何其他的设置都将导致LargeSystemCache值为0。虽然File and Printer Sharing for Microsoft Networks Properties对话框的Optimization部分的四个选项中的每一个都影响到系统缓存的行为，但它们同时也改变了文件服务器服务的行为。

在Windows XP和Windows Server 2003上，你可以在Performance Options对话框中修改LargeSystemCache。要想访问该对话框，你可以在控制面板的System小程序的Advanced页面上，点击Performance部分的Settings按钮，如图11.5所示。在Advanced页面上的Memory Usage区域中选择System Cache，则可以将LargeSystemCache值设置为1，而选择Programs则设置LargeSystemCache为0。

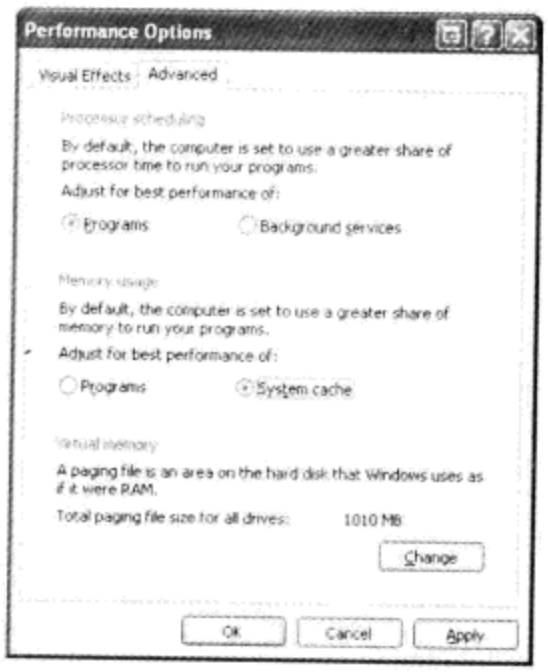


图11.5 在Windows XP和Windows Server 2003上的LargeSystemCache设置

缓存的虚拟大小

系统缓存的虚拟大小是所安装的物理内存数量的一个函数。默认大小为64MB。如果系统的物理内存页面超过了4 032个（16MB物理内存），那么，缓存的大小被设置为128MB，并且每额外4MB物理内存再加上64MB虚拟大小。使用这一算法，对于一台具有64MB物理内存的计算机，其系统缓存的虚拟大小将是：

*128 MB + （64 MB - 16 MB） / 4 MB * 64 MB = 896 MB*

表11.1显示了在不同系统上，系统缓存的最小虚拟大小和最大虚拟大小，以及开始地址和结束地址。如果它在x86系统上运行，并且系统计算得到的缓存虚拟大小超过了512MB，那么，此虚拟大小被限制为512MB，除非LargeSystemCache注册表值为1，在这种情况下，缓存最多可分配到960MB虚拟内存（其中包括一段被称为**缓存额外内存**的地址范围）。为缓存分配更多的虚拟内存，其好处是，一般而言，随着不同的文件、文件的不同视图逐渐被访问，越大的虚拟内存可以导致越少的解除映射（unmap）和重新映射（remap）操作。

www.TopSage.com

表11.1 系统数据缓存的大小和位置

平 台	地址范围	最小的/最大的虚拟大小
x86 2-GB系统空间	0xC1000000~E0FFFFFF, 0xA4000000~BFFFFFFF	64~960 MB
x86 1-GB系统空间	0xC1000000~DBFFFFFF	64~432 MB
x86 1-GB系统空间(支持终端服务)	0xC1000000~DCFFFFFF	64~448 MB
x64	0xFFFFF98000000000~FFFFFA8000000000	64 MB~1024 GB
IA64	0xE000006000000000~E000010600000000	64 MB~1024 GB

表11.2列出的系统变量包含了系统缓存的虚拟大小和地址。

表11.2 与系统缓存的虚拟大小和地址相关的系统变量

系统变量	说 明
MmSystemCacheStart	缓存的开始虚拟地址
MmSystemCacheEnd	缓存的结束虚拟地址
MiSystemCacheStartExtra	如果缓存大小超过了512MB,缓存额外内存的开始虚拟地址(仅适用于x86系统)
MiSystemCacheEndExtra	如果缓存大小超过了512MB,缓存额外内存的结束虚拟地址(仅适用于x86系统)
MmSizeOfSystemCacheInPages	缓存的虚拟大小,按页面计数



实验:查看缓存的虚拟大小

缓存的虚拟大小并不能通过性能计数器来获得,所以,要想知道该值,惟一的办法是检查存有该值的内核变量:*MmSizeOfSystemCacheInPages*:

```
kd> dd MmSizeOfSystemCacheInPages l 1
80539a8c  00020000
```

以上例子是从一个运行Windows XP的x86系统上得到的,并且它的LargeSystemCache值被设置为0。在这个例子中,缓存的大小是0x20000个虚拟页面。因为在x86上一个页面是4KB,因此,此虚拟缓存为512MB,它是从系统的2GB虚拟地址空间中分配而得到的。

www.TopSage.com

缓存的工作集大小

正如前面所提到的，在Windows缓存管理器的设计中，它区别于其他操作系统的一个关键点是，它把物理内存的管理委托给了全局的内存管理器。由于这一原因，现有的处理工作集扩充和修剪的代码，以及管理修改列表和备用列表的代码也被用于控制系统缓存的大小，从而在进程和操作系统对于物理内存的需求之间达到动态的平衡。

系统缓存并没有它自己的工作集，而是与系统的其他部分共用同一个工作集，此工作集包含缓存数据、换页内存池、可换页的Ntoskrnl代码，以及可换页的驱动程序代码。正如在第7章的“系统工作集”一节中所解释的那样，这一工作集在内部称为**系统缓存工作集**（*system cache working set*），尽管系统缓存只不过是其中的一部分。从本书的角度，我们把这一工作集简单地称为**系统工作集**（*system working set*）。第7章中也解释了这样的事实：如果LargeSystemCache注册表值是1，那么，相对于当前系统上正在运行的进程的工作集来说，内存管理器更加偏向于系统工作集。

通过检查表11.3中列出的性能计数器或系统变量，你可以检查系统缓存的物理大小和总系统工作集的大小，以及系统工作集上的页面错误信息。

表11.3 与系统缓存物理大小和页面错误信息相关的系统变量

性能计数器（按字节）	系统变量（按页面）	说 明
Memory: System Cache Resident Bytes	*MmSystemCachePage*	系统缓存所耗的物理内存
Memory: Cache Bytes	*MmSystemCacheWs.WorkingSetSize*	系统工作集的总大小（包括缓存、换页内存池、可换页的代码，以及系统映射的视图）并不像其名称所指示的那样，而仅仅是系统缓存的大小
Memory: Cache Bytes Peak	*MmSystemCacheWs.Peak*	系统工作集的尖峰大小
Memory: Cache Faults/Sec	*MmSystemCacheWs.PageFaultCount*	系统工作集（不仅仅是系统缓存）中的页面错误

www.TopSage.com



实验：查看系统缓存的工作集

调试器命令*!filecache*可以转储出以下信息：系统缓存当前正在使用的物理内存的情况、工作集的当前大小和尖峰大小、与视图相关联的有效页面的数量，以及被映射到视图中的文件的名称（如果有适当名称的话）。（文件系统驱动程序利用未命名的文件流来缓存元数据，比如目录结构和卷的位图）。 此命令的输出如下所示：

```
lkd> !filecache
***** Dump file cache******
  Reading and sorting VACBs ...
  Removed 657 nonactive VACBs, processing 1389 active VACBs ...
File Cache Information
  Current size 132308 kb
  Peak size    145764 kb
  1389 Control Areas
Skipping view @ c1040000 - no VACB, but PTE is valid!
  Loading file cache database (100% of 131072 PTEs)
  SkippedPageTableReads = 0
  File cache has 21510 valid pages


  Usage Summary (in Kb):
Control Valid Standby/Dirty Shared Locked Name
89be09c8   272      0     0      0  default_message_database.db
8a267ad8     4      0     0      0  $Directory
8a37d140  3504   1196     0      0  $Mft
8a35c928  4320   1676     0      0  outlook.pst
8a2acbb8     4      0     0      0  $Directory
8a42c908   564      0     0      0  $BitMap
8a42bd28    12      0     0      0  $Mft
8a29ed88     4      0     0      0  $Directory
8a467008    48      0     0      0  $Directory
8a42b828   124      0     0      0    No Name for File
8a467c58     8      0     0      0  $Directory
8a357f18   136     20     0      0  archive.pst
8a4a7248    52      0     0      0  $Directory
8a42a928   388      0     0      0  $Directory
8a457a10     4      0     0      0  $Directory
8a3f17a8     4      0     0      0  WIASERVC.LOG
89ede0e8     4      0     0      0  $Directory
8a491708 11900    136     0      0  $Mft
8a429fa0     4      0     0      0  $Directory
8a4318d8  2304  26624     0      0  $LogFile
8a438bc0     4      0     0      0  $Directory
8a4286a8     4      0     0      0  $Directory
8a491c50     8      0     0      0  $Directory
8a455de0     4      0     0      0  $Directory
8a3d0f00     4      0     0      0  $Directory
8a44d7a0  3420      0     0      0  $Mft
8a35ba10  1312      0    16      0  SHDOCVW.DLL
8a324130     4      0     0      0  $Directory
8a42c110     4      0     0      0  $Directory
8a41dd80     4      0     0      0  $Directory
88dfff38     4      0     0      0  $Directory
8a4a7640     4      0     0      0  $Directory
8a464d00     4      0     0      0  $Directory
```


www.TopSage.com

缓存的物理大小

虽然系统工作集包含了那些“被映射至缓存虚拟地址空间中的视图”的物理内存的数量，但是，它未必反映了被缓存在物理内存中的文件数据的总数量。在这两个值之间可以有一些差异，因为有一些文件数据可能在内存管理器的备用列表或修改页面列表中。

回顾第7章中曾经讲到过的，在工作集修剪或者页面替换过程中，内存管理器可以将脏页面从一个工作集中移到备用列表或者修改页面列表中，这取决于该页面是否包含了“在被重用以前，必须写到页面文件或其他文件中”的数据。如果内存管理器并没有实现这些列表，那么，任何时候当一个进程访问此前已经从其工作集中移除的数据时，内存管理器必须以硬页面错误的形式，将它从磁盘中取回来。反之，如果被访问的数据位于这些列表中的任何一个，那么，内存管理器只需简单地以软页面错误的形式，将该页面放回到该进程的工作集中。因此，对于保存在页面文件、可执行映像文件或数据文件中的数据来说，这些列表就相当于是这些数据在内存中的缓存。所以，在一个系统中，被缓存的文件数据的总数量不仅包含了系统工作集，也包含了备用列表和修改页面列表联合起来的大小。

用一个例子可以说明缓存管理器如何能够让“超过系统工作集容纳能力”的文件数据被缓存到物理内存中。考虑一个专用做文件服务器的系统。该系统有8GB物理内存，缓存的虚拟大小是960MB（这是x86系统上的最大值）。因此，能够直接被映射到系统缓存虚拟内存中的文件数据的最大数量是960MB。一个客户应用程序跨越网络和服务器来访问文件数据，文件服务器驱动程序（\Windows\System32\Drivers\Srv.sys，在第12章中介绍）使用缓存管理器接口，来代表该客户读写文件数据。如果客户读取几千个1MB大小的文件，那么，缓存管理器在映射第961个文件时，将不得不重新使用它的视图。对于之后的每一个文件读操作，缓存管理器都要解除原有的视图映射，并且为新的文件重新映射视图。当缓存管理器解除一个视图的映射时，内存管理器并没有丢掉在系统缓存的工作集中对应于该视图的文件数据，它只是把这些数据移到备用列表中。如果不存在任何其他的物理内存需求，那么，备用列表几乎可以消耗掉系统工作集之外的所有物理内存。换句话说；该服务器的8GB物理内存几乎可以全部被用于缓存文件数据，如图11.6所示。

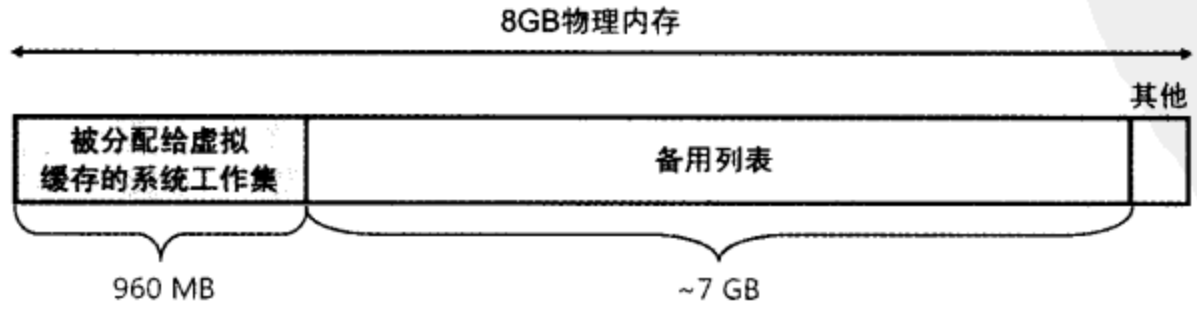


图11.6 绝大多数物理内存被用于文件缓存的一个例子

www.TopSage.com

因为被缓存的文件数据的总数量包括系统工作集、修改页面列表和备用列表，并且，这些部分的大小都是由内存管理器来控制的，因此，从某种意义上说，内存管理器才是真正的缓存管理器。缓存管理器子系统只不过是为“通过内存管理器来访问文件数据”提供了一些便利的接口而已，不过，它也扮演了重要的角色，它通过其预读（read-ahead）和滞后写（write-behind）策略来影响“内存管理器将哪些数据保留在物理内存中”。

为了测试并精确地反映出一个系统上被缓存的文件数据的总数量，任务管理器（Task Manager）和进程管理器（Process Explorer）都在其性能视图中显示了一个名为System Cache的值，它反映了系统工作集、备用列表和修改页面列表这三者合起来的大小。图11.7显示了进程管理器的系统信息视图，在该图中间的Physical Memory区域中显示了System Cache值。

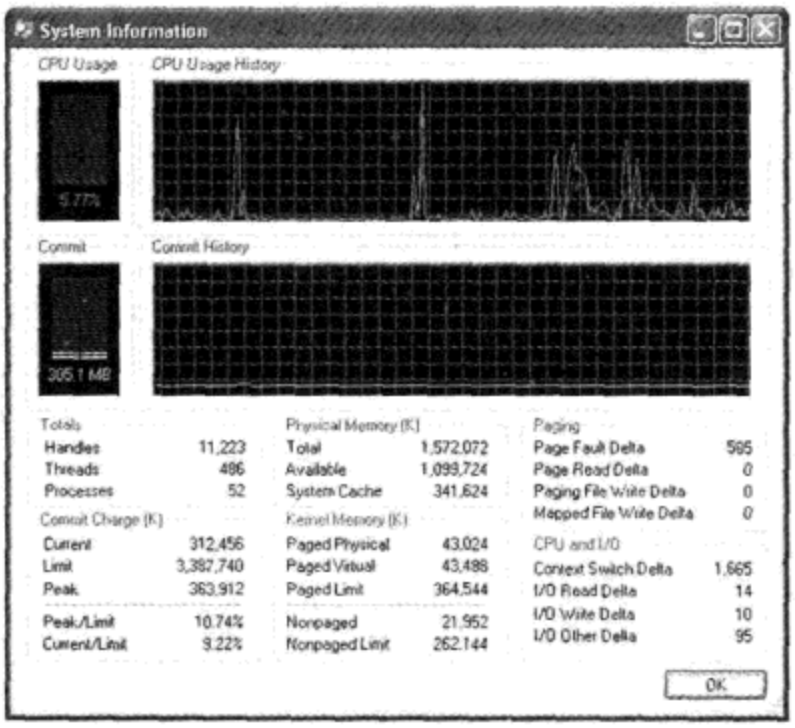


图11.7 进程管理器的System Information对话框

11.4 缓存的数据结构

缓存管理器使用下面的数据结构来记录被缓存的文件：

- 系统缓存中的每一个256KB的槽都是通过一个VACB来描述的；
- 每一个被单独打开的、被缓存的文件都有一个私有的缓存表，其中包含了用于控制预读（本章后面将会讨论）的信息；
- 每一个被缓存的文件都有一个共享的缓存表结构，它指向系统缓存中包含有该文件映射视图的那些槽。

下面的小节描述了这些结构及其相互之间的关系。

www.TopSage.com

系统范围的缓存数据结构

缓存管理器利用一个数据结构数组来跟踪系统缓存中视图的状态。此数组中的数据结构称为**虚拟地址控制块**（VACB，*virtual address control block*）。在系统初始化过程中，缓存管理器分配一个非换页的内存块，以存储所有为描述系统缓存而需要的VACB。它将VACB数组的地址存储在变量*CcVacbs*中。每个VACB代表了系统缓存中的一个256KB视图，如图11.8所示。VACB的结构如图11.9所示。

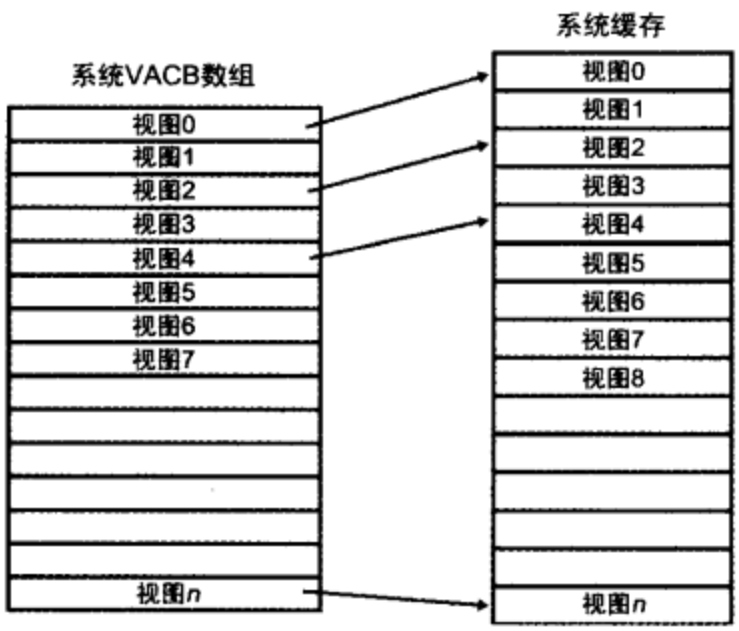


图11.8 系统VACB数组

数据在系统缓存中的虚拟地址
指向共享缓存表的指针
文件偏移
活动计数

图11.9 VACB结构

正如你在图11.9中所看到的那样，VACB中的第一个域是数据在系统缓存中的虚拟地址。第二个域是一个指针，指向共享的缓存表结构，它标识了哪个文件被缓存了。第三个域标识了该视图在文件内部的偏移（总是以256KB粒度为基础）。最后，VACB包含了该视图的引用数目，也就是说，有多少活动的读或写操作正在访问该视图。在一个文件的I/O操作进行过程中，该文件的VACB引用计数被递增，然后，当该I/O操作结束时，它又被递减。当该引用计数为非零时，此VACB是**活动的**（*active*）。对于文件系统元数据的访问，活动计数代表了有多少个文件系统驱动程序使该视图中的页面锁定在内存中。

www.TopSage.com

针对每个文件的缓存数据结构

每一个打开的文件句柄都有一个对应的文件对象（第9章详细地解释了文件对象）。如果该文件被缓存了，那么，该文件对象指向一个**私有的缓存表**（*private cache map*）结构，此表结构包含了最近两次读的位置，所以，缓存管理器可以执行智能预读（在“智能预读”一节中讲述）。而且，针对同一个文件的已打开实例的所有私有缓存表被链接在一起。

每一个被缓存的文件（相对于文件对象）都有一个共享的缓存表结构，它描述了被缓存的文件的状态，包括它的大小和（出于安全的原因）有效数据长度（在“回写缓存和延迟写（Write-Back Caching and Lazy Writing）”一节中解释了有效数据长度域的功能）。共享缓存表也指向了由内存管理器所维护的**内存区对象**（它描述了该文件被映射到虚拟内存的情况）、与该文件相关联的私有缓存表的列表，以及任何描述了该文件当前在系统缓存中的映射视图的VACB（有关内存区对象指针的更多信息，请参见第7章）。这些针对每个文件的缓存数据结构之间的关系如图11.10所示。

当缓存管理器接到从某个特定文件中读数据的请求时，它必须决定以下两个问题的答案：

1. 该文件在缓存中吗？
2. 如果是的话，哪个VACB（若存在的话）指向被请求的位置？

换句话说，缓存管理器必须弄明白，该文件的期望地址处的视图是否已经被映射到系统缓存中了。如果没有VACB包含所期望的文件偏移，那么，所请求的数据当前尚未被映射到系统缓存中。

对于给定一个文件，为了跟踪它的哪些视图已被映射到系统缓存中了，缓存管理器维护了一个VACB指针数组，称为*VACB索引数组*（*VACB index array*）。VACB索引数组中的第一项指向该文件的第一个256KB，第二项指向第二个256KB，以此类推。图11.11中的框图显示了三个不同文件的四个不同内存区，它们当前被映射到系统缓存中了。

当一个进程访问一个特定文件的特定位置时，缓存管理器在该文件的VACB索引数组中寻找正确的数组项，以确定所请求的数据是否已经被映射到系统缓存中了。如果该数组项是非零（因此包含一个指向VACB的指针），那么，该文件的被引用区域位于缓存中。并且，该VACB指向系统缓存中对应于该文件这一视图被映射的位置。如果该数组项为零，那么，缓存管理器必须在系统缓存中找到一个空闲槽（因此是一个空闲的VACB），以映射所请求的视图。

www.TopSage.com

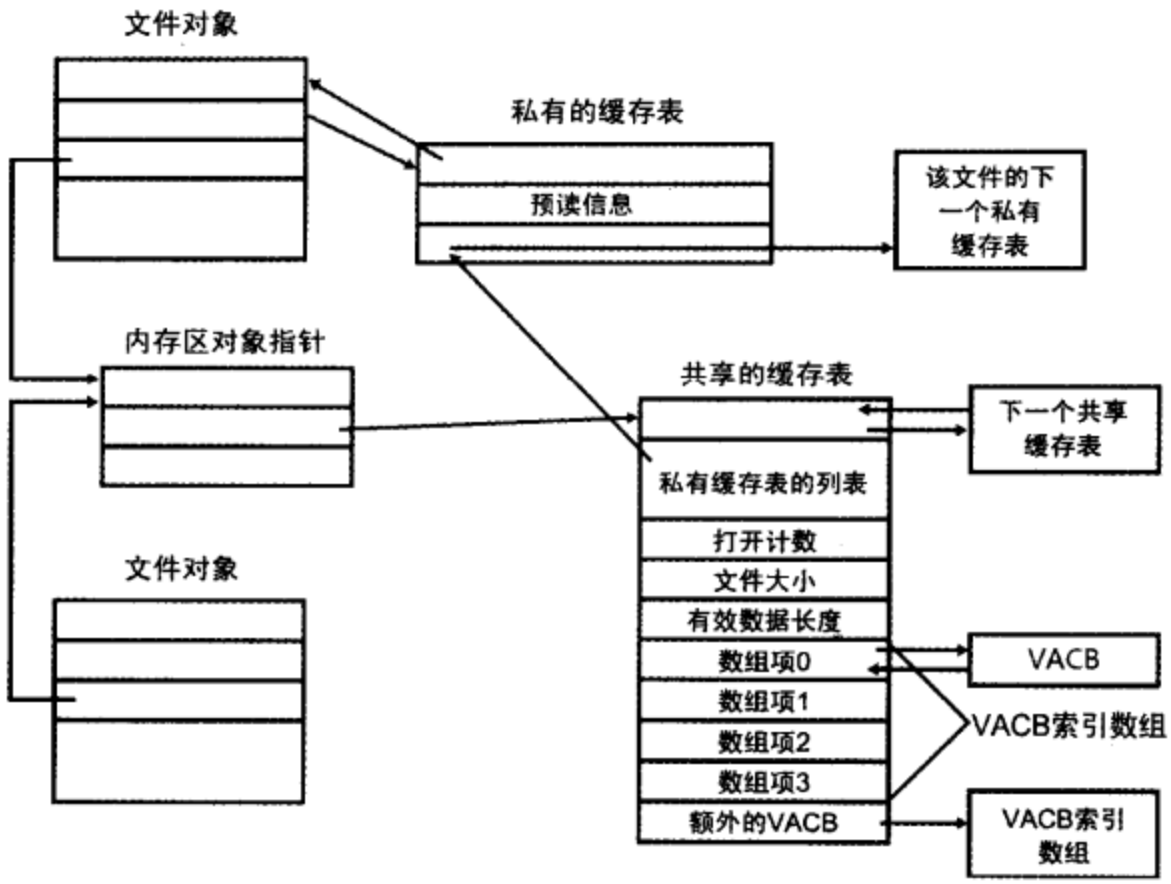


图11.10 针对每个文件的缓存数据结构

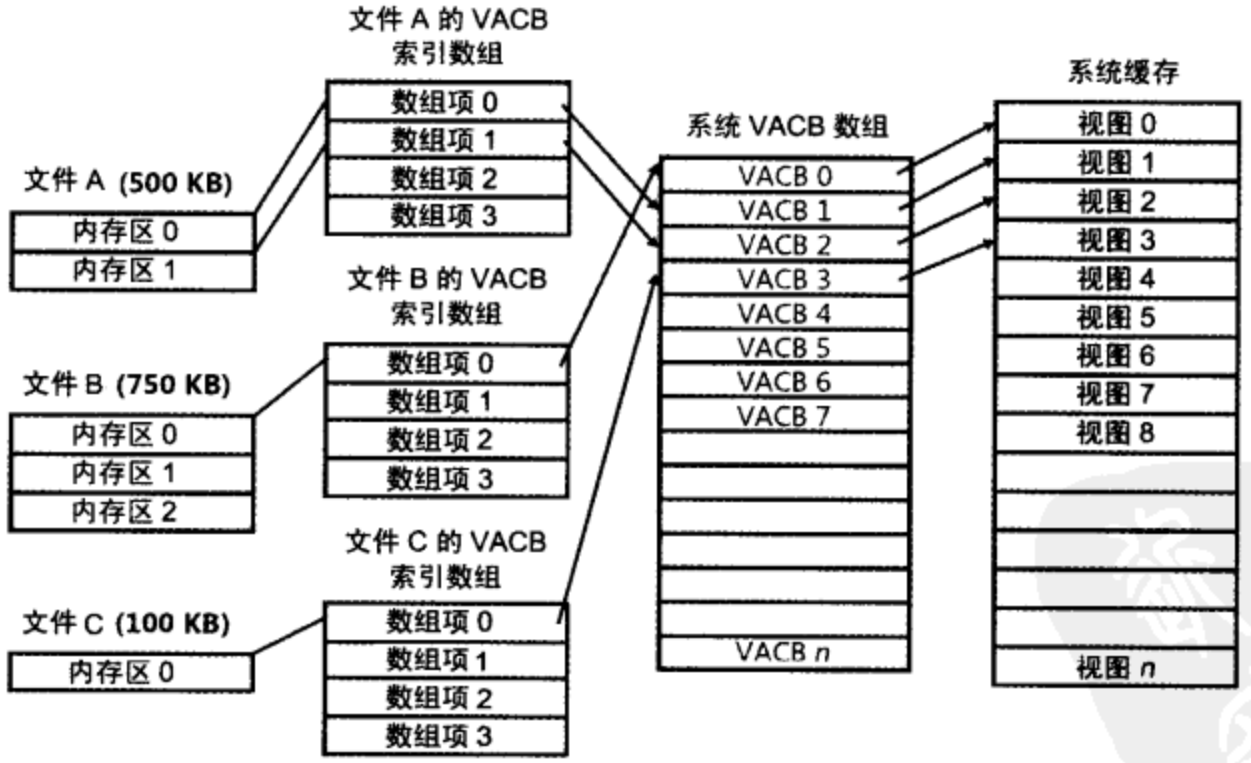


图11.11 VACB索引数组

作为一种空间优化，共享的缓存表包含了一个VACB索引数组，其中含有4个数组项。因为每个VACB描述了256KB，所以，在这个较小的、且又是固定大小的索引数组中，其数组项所指的VACB总共可以描述最大可达1MB的文件。如果一个文件大于1MB，则需要从非换页的

www.TopSage.com

内存池中分配一个单独VACB索引数组，其大小为该文件的大小除以256KB，如果有剩余的话则往上进一。然后，共享的缓存表指向此单独的结构。

作为进一步的优化，如果文件大于32MB，那么，从非换页的内存池中分配的VACB索引数组变成一个稀疏的多级索引数组，此时每个索引数组包含128项。你可以利用以下公式，计算一个文件所需要的级数：

（表示文件大小所需要的位数-*18*） /*7*

将此公式的结果向上取整数。公式中的18源于“每个VACB代表了256KB”这一事实，而256KB等于2^{18}。值7源于“数组中的每一级有128项”这一事实，并且2^7是128。因此，如果一个文件的大小最大可以用63位来描述（这也是缓存管理器所支持的最大尺寸），那么，它只要求7级。该数组是稀疏的，因为缓存管理器只为那些“最底下一级索引数组上具有活动视图”的分支节点分配VACB索引数组。图11.12显示了一个多级VACB数组的例子，它描述了一个足够大的、要求三级索引数组的稀疏文件。

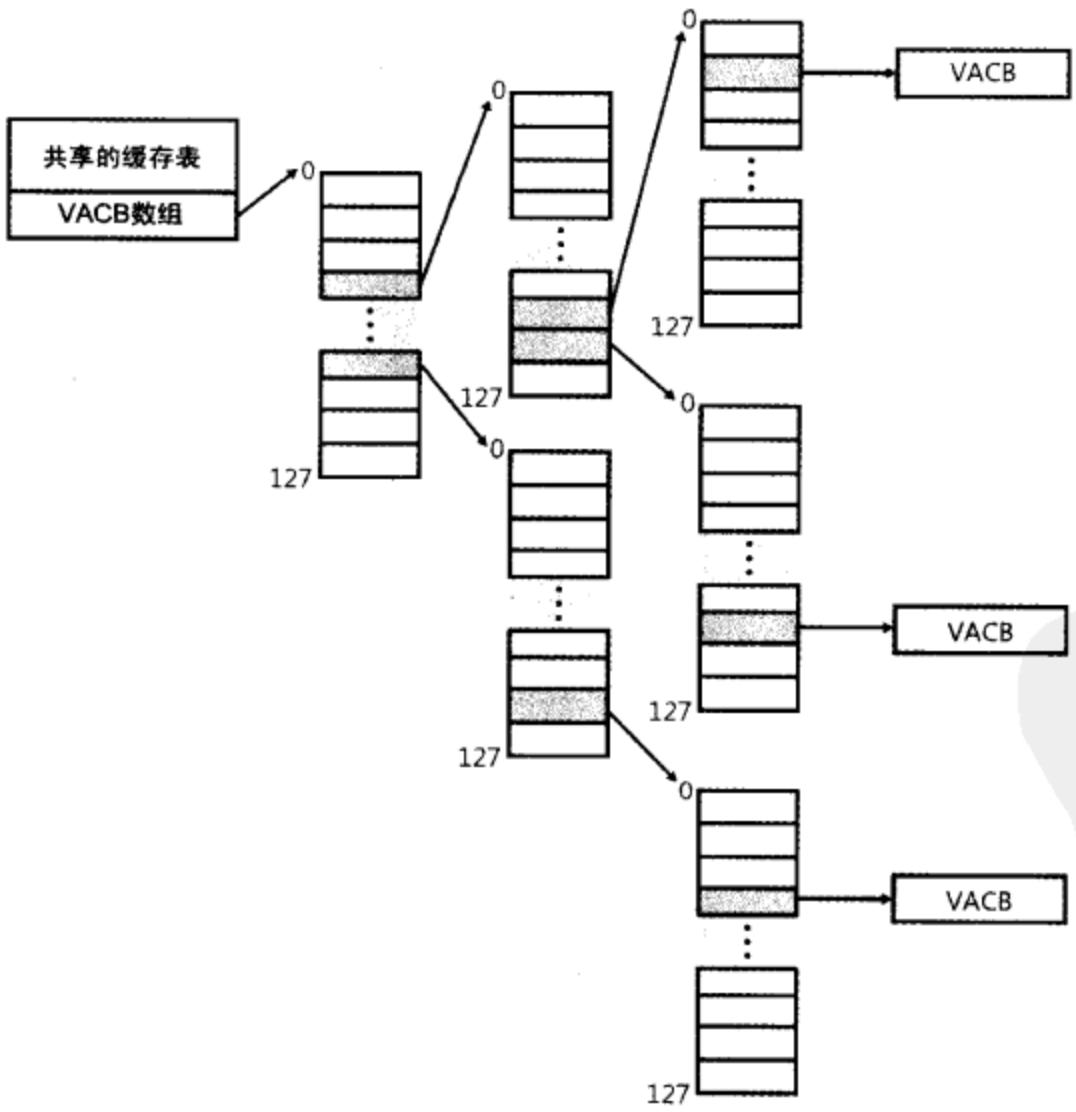


图11.12 多级VACB数组

www.TopSage.com

对于“虽然尺寸很大，却只有一小部分有效数据”的文件，为了有效地处理这样的稀疏文件，以上的方案是必要的，因为只需为VACB数组分配足够的索引数组就可以处理一个大文件的当前被映射的视图了。例如，对于一个32GB的稀疏文件，它只有256KB被映射到系统缓存的虚拟地址空间中，那么，它所要求的VACB数组只包含三个已分配的索引数组，因为该VACB数组只有一个分支有活动视图，而32GB（2^{35}字节）大小的文件只要求一个三级数组。如果缓存管理器没有为该文件使用多级VACB数组的优化，那么，它将必须分配一个包含128 000项的VACB数组，或者等价于1 000个索引数组。



实验：查看共享的和私有的缓存表

你可以使用内核调试器的*dt*命令来查看共享的和私有的缓存表数据结构的定义，并且在一个实际系统上也可以检查这些数据结构。首先，执行*!filecache*命令，并且根据一个你认识的文件名来找到VACB输出中的某一项。在这个例子中，目标文件是一个Microsoft Outlook数据文件：

```
8653c828  120  160    0    0 debugger.chm
```

第一个地址是一个控制区域数据结构的地址，内存管理器用控制区域来跟踪一个地址范围（有关更多的信息，请参见第7章）。控制区域中保存了一个指向文件对象的指针，该文件对象与缓存中的视图相对应。每个文件对象标识了一个已打开文件的实例，在这个例子中，是Windows调试工具箱的帮助文件的一个实例。利用以上你标识出来的这一项的控制区域中的地址，执行下面的命令，你可以看到控制区域的结构：

```
lkd> !ca 8653c828

ControlArea @8653c828
  Segment:    e1bce428     Flink               0   Blink               0
  Section Ref          1  Pfn Ref            46   Mapped Views        4
  User Ref             0  WaitForDel          0   Flush Count         0
  File Object 85d927a8    ModWriteCount       0   System Views        0

  Flags (8008080) File WasPurged Accessed

  File: \Program Files\Debugging Tools for Windows\debugger.chm
```

接下来，利用以下命令，查看该控制区域所引用的文件对象：

```
lkd> dt _File_object 85d927a8
nt!_FILE_OBJECT
   +0x000 Type                   : 5
   +0x002 Size                   : 112
   +0x004 DeviceObject           : 0x86b78e30
   +0x008 Vpb                    : 0x86b90d80
   +0x00c FsContext              : 0xe3b629e0
   +0x010 FsContext2             : 0xe3b62b38
   +0x014 SectionObjectPointer   : 0x85e7a334
   +0x018 PrivateCacheMap        : 0x862f5a50
```


www.TopSage.com

```
   +0x01c FinalStatus       : 0
   +0x020 RelatedFileObject : (null)
   +0x024 LockOperation     : 0 ''
…
```

私有缓存表位于偏移0x18处：

```
lkd> dt _private_cache_map 0x862f5a50
nt!_PRIVATE_CACHE_MAP
   +0x000 NodeTypeCode      : 766
   +0x000 Flags             : _PRIVATE_CACHE_MAP_FLAGS
   +0x000 UlongFlags        : 0x2fe
   +0x004 ReadAheadMask     : 0xffff
   +0x008 FileObject        : 0x85d927a8
   +0x010 FileOffset1       : _LARGE_INTEGER 0x10b28
   +0x018 BeyondLastByte1   : _LARGE_INTEGER 0x12206
   +0x020 FileOffset2       : _LARGE_INTEGER 0x10b28
   +0x028 BeyondLastByte2   : _LARGE_INTEGER 0x12206
   +0x030 ReadAheadOffset   : [2] _LARGE_INTEGER 0x0
   +0x040 ReadAheadLength   : [2] 0
   +0x048 ReadAheadSpinLock : 0
   +0x04c PrivateLinks      : _LIST_ENTRY [ 0x862f5a10 - 0x862f5a10 ]
```

最后，你可以在文件对象的SectionObjectPointers结构中找到共享的缓存表，然后查看其内容：

```
lkd> dt _Section_object_pointers 0x85e7a334
nt!_SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : 0x8653c828
   +0x004 SharedCacheMap    : 0x862f5978
   +0x008 ImageSectionObject : (null)
lkd> dt _shared_cache_map 0x862f5978  nt!_SHARED_CACHE_MAP
   +0x000 NodeTypeCode      : 767
   +0x002 NodeByteSize      : 304
   +0x004 OpenCount         : 1
   +0x008 FileSize          : _LARGE_INTEGER 0x19970a
   +0x010 BcbList           : _LIST_ENTRY [ 0x862f5988 - 0x862f5988 ]
   +0x018 SectionSize       : _LARGE_INTEGER 0x1c0000
   +0x020 ValidDataLength   : _LARGE_INTEGER 0x19970a
   +0x028 ValidDataGoal     : _LARGE_INTEGER 0x19970a
   +0x030 InitialVacbs      : [4] (null)
   +0x040 Vacbs             : 0x85d42c00  -> 0x86bba610
   +0x044 FileObject        : 0x85d927a8
   +0x048 ActiveVacb        : 0x86bba610
…
```

11.5 文件系统接口

当一个文件的数据第一次被读操作或写操作访问时，文件系统驱动程序负责决定该文件的某一部分是否被映射到系统缓存中。如果决定不这样做，那么，文件系统驱动程序必须调用*CcInitializeCacheMap*函数，以建立起上一节所介绍的针对每个文件的数据结构。

www.TopSage.com

一旦一个文件已经被建立起缓存方式的访问环境，文件系统驱动程序就调用几个函数中的某一个，来访问该文件中的数据。要访问被缓存的数据，有三种主要方法，每一种方法各自适用于特定的情形：

- 拷贝方法，在系统空间的缓存缓冲区和用户空间的进程缓冲区之间拷贝用户数据；
- 映射（mapping）和锁定（pinning）的方法，使用虚拟地址来直接读写数据到缓存缓冲区中；
- 物理内存访问方法，使用物理地址来直接读写数据到缓存缓冲区中。

文件系统驱动程序必须提供两个文件读操作版本（缓存的和非缓存的），以避免当内存管理器处理页面错误时导致无限循环。内存管理器通过调用文件系统来从文件中获取数据（当然是通过设备驱动程序），以处理一个页面错误时，它必须在IRP中设置“非缓存（no cache）”标志，以指定非缓存的读操作。

图11.13示出了缓存管理器、内存管理器与文件系统驱动程序在响应用户读或写文件I/O时的典型交互过程。文件系统通过拷贝接口（*CcCopyRead*和*CcCopyWrite*路径）来调用缓存管理器。譬如，为了处理*CcFastCopyRead*或*CcCopyRead*读操作，缓存管理器在缓存中创建一个视图，以便将文件中被读取的那一部分映射到该视图中，然后从视图中拷贝数据，以便将文件数据读到用户缓冲区中。此拷贝操作访问到视图中每一个原先无效的页面时，它就会产生页面错误。相应地，内存管理器向文件系统驱动程序激发非缓存的I/O请求，以获取到被映射至此错误页面的那一部分文件数据。

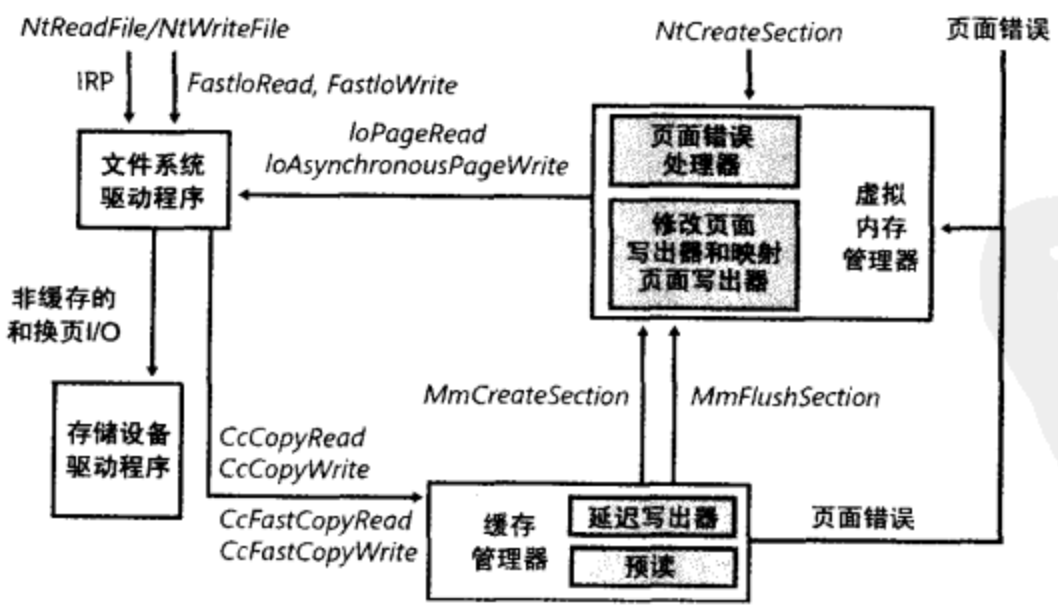


图11.13 文件系统与缓存管理器和内存管理器之间的交互

下面的三小节解释了这些缓存访问机制、它们的用途，以及它们的用法。

www.TopSage.com

从缓存中来回拷贝数据

因为系统缓存位于系统空间中，因此，它被映射到每个进程的地址空间中。然而，如同所有的系统空间页面一样，系统缓存中的页面是无法从用户模式中访问的，因为那将会导致潜在的安全漏洞（例如，一个进程可能无权读一个文件，但该文件的数据当前被包含在系统缓存的某一部分中）。因此，用户应用程序若要读写被缓存的文件，则必须通过内核模式的例程来获得服务：这些内核模式例程在“系统空间的缓存缓冲区”和“驻留在进程地址空间中的应用程序缓冲区”之间拷贝数据。被文件系统驱动程序用来执行这一操作的函数如表11.4所示。

表11.4 从缓存中来回拷贝数据的内核模式函数

函 数	说 明
CcCopyRead	将指定的字节范围从系统缓存拷贝到用户缓冲区中
CcFastCopyRead	CcCopyRead的一个变形，更加快速，但也有限制：32位文件偏移，以及同步读操作
CcCopyWrite	将指定的字节范围从用户缓冲区拷贝到系统缓存中
CcFastCopyWrite	CcCopyWrite的一个变形，更加快速，但也有限制：32位文件偏移，以及同步的、非直写（non-write-through）的写操作（NTFS使用这种写模式，FAT不使用）

你可以通过表11.5中所列的性能计数器或系统变量，来检查缓存的读操作行为。

表11.5 可用于检查缓存读操作行为的系统变量

性能计数器（频度）	系统变量（计数）	说 明
Cache: Copy Read Hits %	（*CcCopyReadWait* + *CcCopyReadNoWait*）/（*CcCopyReadWait* + （*CcCopyReadWaitMiss* + *CcCopyReadNoWait*）+ *CcCopyReadNoWaitMiss*）	对于缓存中的文件部分的拷贝读（copy read）操作的百分比（拷贝读操作仍然可能产生换页I/O——“Memory: Cache Faults/Sec”计数器报告的是系统工作集的页面错误行为，但是它同时包含了硬的和软的页面错误，所以，此计数器并不表明实际的“由缓存的页面错误而引起的”换页I/O）
Cache: Copy Reads/Sec	*CcCopyReadWait* + *CcCopyReadNoWait*	缓存中总的拷贝读操作的次数
Cache: Sync Copy Reads/Sec	*CcCopyReadWait*	缓存中的同步拷贝读操作的次数
Cache: Async Copy Reads/Sec	*CcCopyReadNoWait*	缓存中的异步拷贝读操作的次数

www.TopSage.com

通过映射和锁定接口进行缓存

当用户应用程序读和写磁盘文件中的数据时，文件系统驱动程序需要读和写那些描述这些文件本身的数据（元数据，或者卷结构数据）。然而，由于文件系统驱动程序运行在内核模式下，因此，如果缓存管理器被正确地通知到的话，这些文件系统驱动程序就可以直接修改系统缓存中的数据。为了允许这种优化，缓存管理器提供了表11.6中所示的函数。这些函数使得文件系统驱动程序可以找到文件系统的元数据驻留在虚拟内存中的什么地方，因此允许直接进行元数据修改，而无需使用中间缓冲区。

表11.6 寻找元数据位置的函数

函 数	说 明
CcMapData	为了读访问而映射一段字节范围
CcPinRead	为了读/写访问而映射一段字节范围，并锁定这段范围
CcPreparePinWrite	为了写访问（读访问不是有效的）而映射并锁定一段字节范围
CcPinMappedData	锁定一个原先已被映射的缓冲区
CcSetDirtyPinnedData	通知缓存管理器，此数据已被修改了
CcUnpinData	释放这些页面，因而它们可以被从内存中移除

如果一个文件系统驱动程序需要读缓存中的文件系统元数据，那么，它调用缓存管理器的映射接口来获得目标数据的虚拟地址。缓存管理器寻找所有被请求的页面，并把它们带入到内存中，然后将控制返回给文件系统驱动程序。于是，文件系统驱动程序可以直接访问这些数据。

如果文件系统驱动程序需要修改缓存中的页面，那么，它调用缓存管理器的锁定服务，该服务会将这些被修改的页面保持在内存中。这些页面并不真的被锁在内存中（比如当设备驱动程序为了直接内存访问传输而锁住页面时所做的那样）。在绝大多数时间，文件系统驱动程序将它的元数据流标记为“不可写出（no write）”，这指示内存管理器的映射页面写出器（在第7章中解释）不要将这些页面写出到磁盘上，直到明确地通知它可以写出为止。当文件系统驱动程序解除锁定（释放）这些页面时，缓存管理器将所有的变化刷新到磁盘上，并且释放这些元数据所占据的缓存视图。

映射和锁定接口解决了一个在实现文件系统时很棘手的问题：缓冲区管理。文件系统若不能直接操作被缓存的元数据，那么，它必须预测“当它更新一个卷的结构时”所需缓冲区的最大数量。由于允许文件系统直接访问和更新缓存中的元数据，因此，缓存管理器消除了对缓冲区的需求，文件系统只需简单地更新内存管理器所提供的虚拟内存中的卷结构即可。文件系统所碰到的惟一限制是，有多少内存可供使用。

www.TopSage.com

你可以通过表11.7中所列出的性能计数器和系统变量来检查系统缓存中的锁定和映射行为。

表11.7 可用于检查锁定和映射行为的系统变量

性能计数器（频度）	系统变量（计数）	说 明
Cache: Data Map Hits %	(*CcMapDataWait* + *CcMapDataNoWait*) / (*CcMapDataWait* + *CcMapDataNoWait*) + (*CcMapDataWaitMiss* + *CcMapDataNoWaitMiss*)	对于缓存中的文件部分的数据映射操作的百分比（拷贝读操作仍然可能会产生换页I/O）
Cache: Data Maps/Sec	*CcMapDataWait* + *CcMapDataNoWait*	缓存中总的数据映射的次数
Cache: Sync Data Maps/Sec	*CcMapDataWait*	缓存中的同步数据映射的次数
Cache: Async Data Maps/Sec	*CcMapDataNoWait*	缓存中的异步数据映射的次数
Cache: Data Map Pins/Sec	*CcPinMappedDataCount*	请求“锁定已映射数据”的次数
Cache: Pin Read Hits %	(*CcPinReadWait* + *CcPinReadNoWait*) / (*CcPinReadWait* + *CcPinReadNoWait*) + (*CcPinReadWaitMiss* + *CcPinReadNoWaitMiss*)	对于缓存中的文件部分的锁定读（pinned read）操作的百分比（拷贝读操作仍然可能会产生换页I/O）
Cache: Pin Reads/Sec	*CcPinReadWait* + *CcPinReadNoWait*	缓存中总的锁定读的次数
Cache: Sync Pin Reads/Sec	*CcPinReadWait*	缓存中的同步锁定读的次数
Cache: Async Pin Reads/Sec	*CcPinReadNoWait*	缓存中的异步锁定读的次数

通过直接内存访问接口进行缓存

除了映射和锁定接口可被用于直接访问缓存中的元数据以外，缓存管理器还提供了第三个访问缓存中的数据的接口：**直接内存访问**（DMA，*direct memory access*）。通过DMA函数，可以直接读或写系统缓存的页面，而无需经由缓冲区的介入，比如当一个网络文件系统正在通过网络进行传输的时候。

DMA接口将被缓存的用户数据的物理地址（而不是虚拟地址，映射和锁定接口返回的是虚拟地址）返回给文件系统，然后，这些物理地址可以直接被用于在物理内存和网络设备之间传输数据。虽然少量的数据（1KB至2KB）可以使用通常的基于缓冲区的拷贝接口，但是，对于更大的数据传输，DMA可以为网络服务器在处理远程系统的文件请求时，带来显著的性能改进。

www.TopSage.com

为了描述这些对于物理内存的引用，需要用到**内存描述符列表**（MDL，*memory descriptor list*）（第7章中简单介绍了MDL）。表11.8中描述的四个单独的函数建立起了缓存管理器的DMA接口。

表11.8 建立起DMA接口的四个函数

函　数	说　明
CcMdlRead	返回一个描述了指定字节范围的MDL
CcMdlReadComplete	释放MDL
CcMdlWrite	返回一个“描述一指定字节范围（可能包含了零）”的MDL
CcMdlWriteComplete	释放MDL，并将该范围标记为已被修改

你可以通过表11.9中所列出的性能计数器和系统变量来检查系统缓存中的MDL行为。

表11.9 可用于检查系统缓存中的MDL行为的系统变量

性能计数器（频度）	系统变量（计数）	说　明
Cache: MDL Read Hits %	（*CcMdlReadWait* + *CcMdlReadNoWait*）/（*CcMdlReadWait* + *CcMdlReadNoWait*）+（*CcMdlReadWaitMiss* + *CcMdlReadNoWaitMiss*）	对于缓存中的文件部分的MDL读操作的百分比（引用那些由MDL读操作来满足的页面可能仍然会产生换页I/O）
Cache: MDL Reads/Sec	*CcMdlReadWait* + *CcMdlReadNoWait*	缓存中总的MDL读操作的次数
Cache: Sync MDL Reads/Sec	*CcMdlReadWait*	缓存中的同步MDL读操作的次数
Cache: Async MDL Reads/Sec	*CcMdlReadNoWait*	缓存中的异步MDL读操作的次数

11.6 快速I/O

无论何时，只要有可能，对于被缓存的文件的读和写操作都是由一种称为**快速I/O**（*fast I/O*）的高速机制来处理的。快速I/O是一种对于被缓存文件的读写方法，它无需经过“生成一个IRP”这样的环节（如第9章中所描述）。通过快速I/O，I/O管理器调用文件系统驱动程序的快速I/O例程，以确定该I/O是否可以直接由缓存管理器来满足，而无需生成一个IRP。

因为缓存管理器是建立在虚拟内存子系统之上的，因此，文件系统驱动程序只需简单地对于“映射到实际被引用文件”的页面进行数据拷贝，就可以使用缓存管理器来访问文件数据，这样做也避免了生成一个IRP所需要的开销。

www.TopSage.com

快速I/O并不总是发生的。例如，对一个文件的第一次读或写操作要求建立起该文件的缓存访问环境（将文件映射到缓存中，建立缓存数据结构，如前面的“缓存的数据结构”一节中介绍的那样）。而且，如果调用者指定了一个异步读或写，则快速I/O就不会被用到，因为那样的话，调用者可能在换页I/O操作（为了满足缓冲区和系统缓存之间的数据拷贝，换页I/O是必需的）过程中被停滞，从而并未真正提供所要求的异步I/O操作。即使在一个同步I/O上，文件系统驱动程序也可以决定该I/O操作不能用快速I/O机制来处理，比如说，如果所涉及的文件有一个锁定的字节范围（调用Windows的*LockFile*和*UnlockFile*函数的结果），而缓存管理器又不知道哪些文件的哪些部分被锁定了，那么，文件系统驱动程序必须要检查读或写的有效性，而这要求生成一个IRP。快速I/O的决策树如图11.14所示。

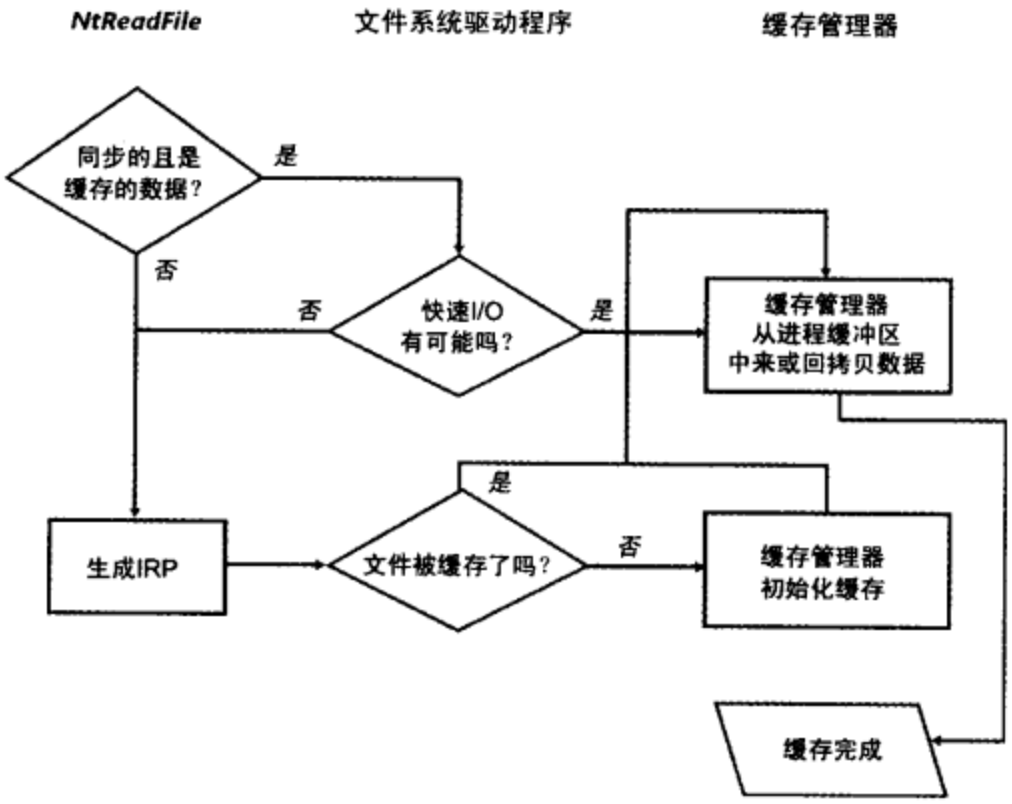


图11.14 快速I/O的决策树

在用快速I/O来服务一个读或写操作的时候，会涉及以下这些步骤。

1. 线程执行一个读或写操作。
2. 如果该文件已被缓存，并且I/O是同步的，则该请求被传递给文件系统驱动程序的快速I/O入口点。如果该文件未被缓存，则文件系统驱动程序建立起该文件的缓存访问环境，因而下一次可以使用快速I/O来处理一个读或写请求。
3. 如果文件系统驱动程序的快速I/O例程确定快速I/O是可能的，那么，它调用缓存管理器的读或写例程来直接访问缓存中的文件数据（如果快速I/O不可能的话，则文件系统驱动程序返回至I/O系统，然后，I/O系统为该I/O生成一个IRP，并最终调用文件系统的常规读例程）。

www.TopSage.com

4. 缓存管理器将所提供的文件偏移转译成缓存中的虚拟地址。
5. 对于读操作，缓存管理器将数据从缓存拷贝到发出该请求的进程的缓冲区中；对于写操作，它将数据从进程缓冲区拷贝到缓存中。
6. 以下动作之一发生：
   - 对于读操作，若文件被打开时未指定FILE_FLAG_RANDOM_ACCESS，则调用者的私有缓存表中的预读信息被更新；
   - 对于写操作，缓存中任何已修改页面的脏位被设置，因而延迟写出器（lazy writer）将知道该把这些页面刷新到磁盘上；
   - 对于直写的文件（write-through file），任何修改都被刷新到磁盘上。



> **注** 快速I/O的路径并不局限于“当所请求的数据已经驻留在物理内存中”的场合。正如在上述列表的第5和6步中所说明的那样，缓存管理器只是简单地访问“它认为数据所在的那个已打开文件”的虚拟地址。如果发生缓存未命中（cache miss），则内存管理器动态地将数据换入到物理内存中。

表11.10中所列出的性能计数器和系统变量可被用于检查系统中的快速I/O的行为。

表11.10 可用于确定快速I/O行为的系统变量

性能计数器（频度）	系统变量（计数）	说 明
Cache: Sync Fast Reads/Sec	*CcFastReadWait*	作为“快速I/O请求”来处理的同步读操作
Cache: Async Fast Reads/Sec	*CcFastReadNoWait*	作为“快速I/O请求”来处理的异步读操作（总是零，因为Windows中并不执行异步快速读操作）
Cache: Fast Read Resource Misses/Sec	*CcFastReadResourceMiss*	由于资源冲突而未能被满足的快速I/O操作（这种情形可能在FAT中发生，但不会在NTFS中发生）
Cache: Fast Read Not Possibles/Sec	*CcFastReadNotPossible*	未能被满足的快速I/O操作（由文件系统驱动程序来决定。例如，有字节范围锁的文件不能使用快速I/O）

www.TopSage.com

11.7 预读（Read Ahead）和滞后写（Write Behind）

在这一节中，你将会看到，缓存管理器如何实现“代表文件系统驱动程序来读和写文件数据”。记住，只有当一个文件被打开时未设置FILE_FLAG_NO_BUFFERING标记，并且之后用Windows I/O函数（例如，使用Windows的*ReadFile*和*WriteFile*函数）来读或写数据，缓存管理器才会介入到文件I/O中。被映射的文件（mapped file）并不经过缓存管理器，在打开文件时设置了FILE_FLAG_NO_BUFFERING标记的文件也不经过缓存管理器。

智能预读

缓存管理器使用空间局部性的原则来执行智能预读，其做法是，根据调用进程当前所读的数据，来预测接下来它可能会读哪些数据。因为系统缓存是建立在虚拟地址基础之上的，而对于一个特定的文件来说，其虚拟地址是连续的，因此，它不用关心一个文件的数据在物理内存中是否并列放置。逻辑块缓存系统的文件预读更为复杂，它要求文件系统驱动程序和块缓存之间的紧密合作，因为这样的缓存系统是以磁盘上被访问数据的相对位置为基础的，当然，文件在磁盘上并不必须连续存放。你可以使用“Cache: Read Aheads/Sec”性能计数器或*CcReadAheadIos*系统变量来检查系统的预读行为。

按顺序访问模式来读取文件的下一块，这样做可以获得明显的性能提高。为了将预读的优势扩展到条带式数据访问（strided data accesses，前向和后向访问一个文件）的情形，缓存管理器对于被访问的文件句柄，在其私有缓存表中维护了最近两个读请求的历史信息，这种方法称为**带历史信息的异步预读**（*asynchronous read-ahead with history*）。如果从调用者看似随机的读请求中能够确定出一种模式，则缓存管理器推断出该模式。例如，如果调用者先读页面4 000，然后读页面3 000，那么，缓存管理器假设调用者下一个将要请求的页面是2 000，从而将它预读进来。



> **注** 虽然调用者必须至少发出三个读操作才能建立起一个可预测的序列，但是，在私有缓存表中只存储两个读操作。

为了使预读更为高效，Windows的*CreateFile*函数提供了一个标志来指明前向的、顺序的文件访问：FILE_FLAG_SEQUENTIAL_SCAN。如果该标志被设置了，则缓存管理器不再为调用者保留供预测使用的读操作历史，而是执行顺序的预读。然而，随着文件被读入到系统缓存的工作集中，缓存管理器解除那些不再活动的文件视图的映射关系，并且，如果它们未被修改，则指示内存管理器把属于这些解除映射的视图的页面放到备用列表的头部，因而它们将很快被重用。它也会提前读取两倍的数据（例如，读入128KB而不是64KB）。随着调用者连续执行读操作，缓存管理器预读额外的数据块，总是保持比调用者提前完成一次读操作（其大小等于当前读操作的大小）。

www.TopSage.com

缓存管理器的预读是异步的，因为它是在一个独立于调用者线程的线程环境中执行的，并且与调用者的执行是并发进行的。当调用缓存管理器来获取被缓存的数据时，缓存管理器首先访问被请求的虚拟页面以满足该请求，然后在一个系统辅助线程中排队一个额外的I/O请求，以获取额外的数据。此辅助线程然后在后台执行，预先读取额外的数据，期望这些数据正是调用者下一次请求要读取的数据。在程序继续执行时，这些预读的页面被换入到内存中，因此，当调用者请求这些数据的时候，它们已经在内存中了。

若应用程序没有可预测的读模式，则它们可以在调用*CreateFile*函数的时候指定FILE_FLAG_ RANDOM_ACCESS标志。此标志告诉缓存管理器，不要企图预测应用程序接下来将在哪个位置上读取数据，所以，此标志关闭了预读特性。此标志也阻止缓存管理器在该文件被访问的过程中主动地解除文件视图的映射关系，因而，应用程序重新访问该文件的各个部分时，可以使该文件的映射/解除映射的行为尽可能地最小化。

回写缓存（Write-Back Caching）和延迟写（Lazy Writing）

缓存管理器利用延迟写的做法实现了一种回写缓存。这意味着，被写入文件的数据首先存储在缓存页面的内存中，以后再被写到磁盘上。因此，写操作可以累积一段短的时间，然后一次将所有累积的数据刷新到磁盘上，因而减少了磁盘I/O操作的总数。

缓存管理器必须显式地调用内存管理器来刷新缓存页面，否则的话，只有当对于物理内存的需求超过了当前供给量时，内存管理器才会将内存的内容写到磁盘上（这种做法对于易失性数据是非常合适的）。然而，被缓存的文件数据代表了非易失性磁盘数据。如果一个进程修改了被缓存的数据，那么，用户期望这些内容能被及时地反映到磁盘上。

以什么样的频率来刷新缓存，这是一个重要的决定。如果缓存被刷新得太频繁，则由于不必要的I/O，系统性能将变慢。如果缓存被刷新得太不频繁，则你要冒“万一失败以后丢失已修改的文件数据”（如果用户知道自己已经请应用程序保存修改了，再要丢失数据就尤其会激怒他们）和“用完物理内存（已修改的页面也要额外占据一部分内存）”的风险。

为了平衡这些考虑，缓存管理器的延迟写出器（*lazy writer*）函数在一个系统辅助线程上每秒钟执行一次，同时将系统缓存中八分之一的脏页面排入到队列中，以便被写到磁盘上。如果脏页面产生的数量超过了延迟写出器已确定的写出数量，则延迟写出器根据它的计算，将一部分额外数量的脏页面写出去，以便跟脏页面产生的数量相匹配。实际的I/O操作是由“系统全局范围的关键辅助线程池”中的系统辅助线程来完成的。



注 缓存管理器为文件系统驱动程序提供了一种方法来跟踪“何时以及多少数据已被写到一个文件中”。在延迟写出器将脏页面刷新到磁盘上以后，缓存管理器通知文件系统，指示它更新该文件的有效数据长度的视图（缓存管理器和文件系统分别单独地跟踪一个文件在内存中的有效数据长度）。

www.TopSage.com

你可以通过表11.11中所列出的缓存性能计数器和系统变量来检查延迟写出器的行为。

表11.11 可用于检查延迟写出器行为的系统变量

性能计数器（频度）	系统变量（计数）	说 明
Cache: Lazy Write Flushes/Sec	*CcLazyWriteIos*	延迟写出器的刷新次数
Cache: Lazy Write Pages/Sec	*CcLazyWritePages*	被延迟写出器写出的页面数量

禁止一个文件的延迟写出行为

如果你在调用Windows的*CreateFile*函数时指定了FILE_ATTRIBUTE_TEMPORARY标志，以此来创建一个临时文件的话，则延迟写出器不会把脏页面写到磁盘上，除非存在严重的物理内存短缺，或者该文件被显式地刷新。延迟写出器的这一特征提高了系统的性能——延迟写出器并不立即把最终可能要丢弃的数据写到磁盘上。应用程序通常在关闭了临时文件以后，很快把它们删除。

强迫缓存被直写（write-through）到磁盘上

因为有些应用程序不能容忍在“写一个文件”和“看到磁盘上最新的数据”之间那怕有些微的延迟，所以，缓存管理器也支持在单个文件对象的基础上进行缓存直写：一旦数据被改变了，则应尽快被写到磁盘上。为了打开缓存直写功能，应用程序可以在调用*CreateFile*函数时设置FILE_FLAG_WRITE_THROUGH标志。此外，一个线程执行到某个点上，需要将数据写到磁盘上时，它可以利用Windows的*FlushFileBuffers*函数来显式地刷新一个打开的文件。你可以通过表11.12中所列出的性能计数器或系统变量，来观察直写I/O请求或显式的*FlushFileBuffers*调用所导致的缓存刷新操作。

表11.12 可用于查看缓存刷新操作的系统变量

性能计数器（频度）	系统变量（计数）	说 明
Cache: Data Flushes/Sec	*CcDataFlushes*	由于直写，或由于延迟写出器而显式地刷新缓存页面的次数
Cache: Data Flush Pages/Sec	*CcDataPages*	由于直写，或由于延迟写出器而显式被刷新的页面数

刷新被映射的文件

如果延迟写出器必须从一个已经被映射到另一进程地址空间的视图中，将数据写到磁盘上，则情形变得有点复杂，因为缓存管理器只知道它曾经修改过的那些页面（被另一个进程修改过的页面，只有该进程才知道，因为这些已修改页面的页表项中的修改位只被记录在进程的私有页表中）。

www.TopSage.com

为了处理这种情形，当一个用户映射一个文件时，内存管理器会通知缓存管理器。当这样一个文件在缓存中被刷新（例如，由于调用了Windows的*FlushFileBuffers*函数而导致）时，缓存管理器将缓存中的脏页面写到磁盘上，然后检查该文件是否被另一个进程映射了。缓存管理器看到该文件已被映射时，刷新该内存区的整个视图，以便将第二个进程可能已经修改的那些页面也写出去。如果一个用户映射了一个文件的视图，并且该文件在缓存中也是打开的，那么，当该视图被解除映射时，已修改的页面被标记为脏的，因而以后当延迟写出器线程刷新该视图时，这些脏页面将被写到磁盘上。只要此动作序列按下面的顺序来发生，则这一过程就可以正确无误。

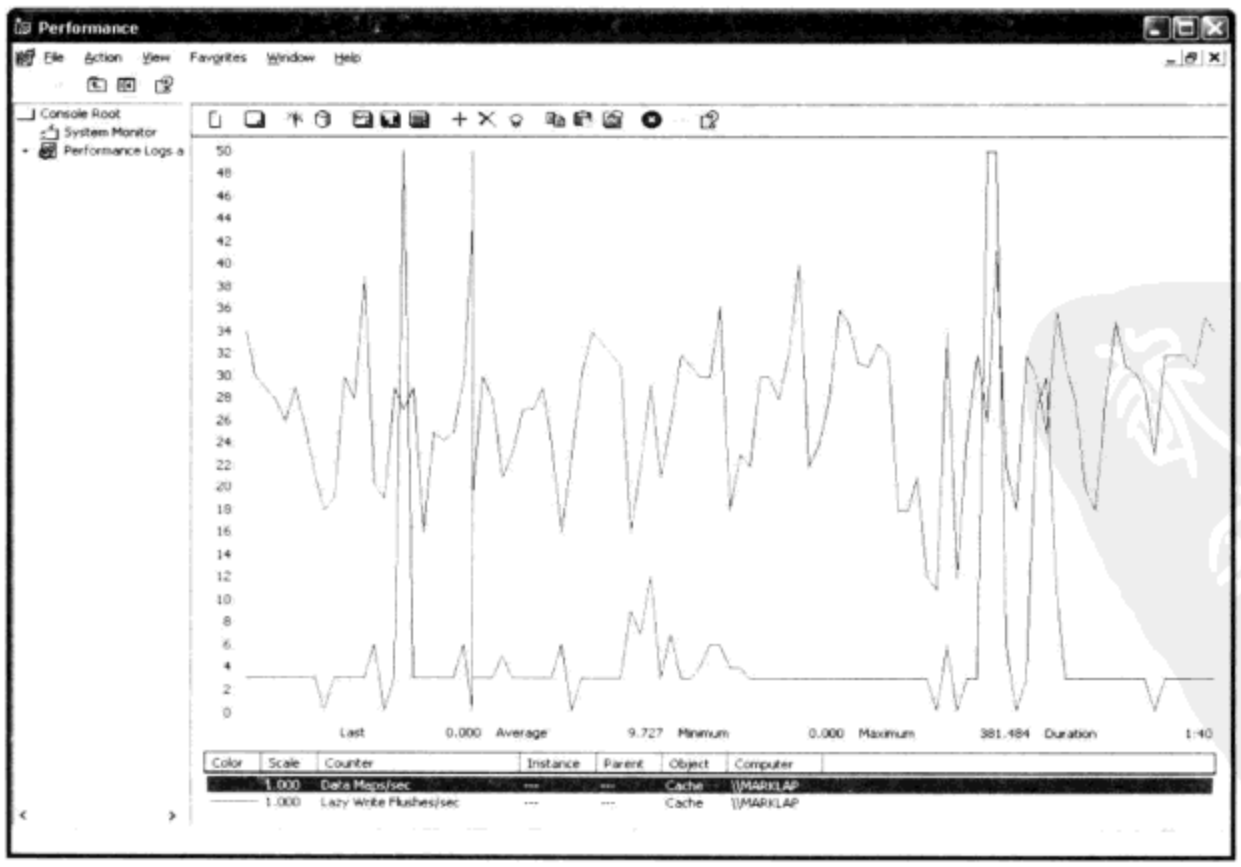
1. 一个用户解除该视图的映射。
2. 一个进程刷新文件缓冲区。

如果此顺序不能被遵从，那么，你将无法预测哪些页面将被写到磁盘上。



实验：观察缓存的刷新

利用下面的做法你可以看到“缓存管理器将视图映射到系统缓存中并且将页面刷新到磁盘上”：运行性能工具，加入“Data Maps/sec”和“Lazy Write Flushes/sec”计数器，然后将一个大文件从一个位置拷贝到另一个位置。在下面的屏幕截图中，往往较高的那条线是Data Maps/sec，另一条线是Lazy Write Flushes/sec。



www.TopSage.com

写节流（Write Throttling）

文件系统和缓存管理器必须确定，一个被缓存的写请求是否会影响系统的性能，然后再来调度和安排任何被延迟的写操作。首先，文件系统通过*CcCanIWrite*函数询问缓存管理器：现在立即写出特定数量的字节是否会损害性能；如果有必要的话，阻塞这次写操作。然后，文件系统通过调用*CcDeferWrite*来建立一个回调，以便当再次允许写操作时，缓存管理器可以自动写出这些字节数据。一旦缓存管理器被通知有一个马上要进行的写操作时，它确定在系统缓存中有多少脏页面，以及有多少物理内存可以使用。如果空闲的物理页面非常少，则缓存管理器将请求“往缓存中写数据的文件系统线程”临时阻塞住。缓存管理器的延迟写出器把一些脏页面刷新到磁盘上，然后允许刚刚被阻塞住的文件系统线程继续执行。当文件系统或者网络服务器发出一个大的写操作时，这种“写节流（write throttling）”可以防止系统性能退化。



> **注** 写节流对于系统的影响是全局的，因为它所依据的资源——可利用的物理内存——对于系统来说是全局的。这意味着，如果由于一个慢设备有很重的写动作从而触发了写节流机制，那么，到其他设备的写操作也将被节流。

脏页面阈值（*dirty page threshold*）是指，在对被缓存的写操作进行节流控制以前，系统缓存中允许的脏页面的数量。此值是在系统初始化时刻计算得到的，它依赖于物理内存的大小和前面描述过的注册表值LargeSystemCache。

表11.13示出了用于计算脏页面阈值的算法。如果系统最大工作集的大小值超过4MB（通常是这样），那么，表11.13中的计算方法也随之改变（参见第7章可知道内存管理器如何选择系统工作集的大小值。也就是说，它如何确定此值是小的、中等的或者大的）。当工作集最大值超过4MB时，脏页面阈值被设置为系统工作集最大值减去2MB。

表11.13 计算脏页面阈值的算法

系统内存大小	脏页面阈值
小	物理页面数/ 8
中等	物理页面数/ 4
大	以上两个值的和

对于在慢速通信线路上传输数据的网络重定向器，写节流机制也是很有用的。例如，假设一个本地进程通过一条9600波特率的线路，向一个远程文件系统发送大量的数据。在缓存管理器的延迟写出器刷新系统缓存以前，这些数据不会被写到远程磁盘上。

www.TopSage.com

如果重定向器累积了大量的、需要一次被刷新到磁盘上的脏页面，那么，接收方可能会在数据传输完成以前接收到一个网络超时。通过使用*CcSetDirtyPageThreshold*函数，缓存管理器允许网络重定向器设置一个可容忍的脏缓存页面的数量界限。通过限制脏页面的数量，重定向器可保证，缓存的刷新操作不会引起网络超时。



> **注** 在Windows XP和以后的版本上，网络重定向器并不设置脏页面阈值，相反地，它们依赖于默认的系统值。



实验：查看写节流参数

内核调试器命令*!defwrites*能够转储出缓存管理器所使用的内核变量的值，包括“当确定是否应该对写操作进行节流时”文件缓存中脏页面的数量（*CcTotalDirtyPages*）：

```
kd> !defwrites
*** Cache Write Throttle Analysis ***

        CcTotalDirtyPages:                 758 (    3032 Kb)
        CcDirtyPageThreshold:              770 (    3080 Kb)
        MmAvailablePages:                42255 (  169020 Kb)
        MmThrottleTop:                     250 (    1000 Kb)
        MmThrottleBottom:                   30 (     120 Kb)
        MmModifiedPageListHead.Total:      689 (    2756 Kb)

        CcTotalDirtyPages within 64 (max charge) pages of the threshold, writes
          may be throttled

        Check critical workqueue for the lazy writer, !exqueue 16
```

以上输出显示了脏页面的数量已接近于可触发写节流机制的数值（*CcDirtyPage-Threshold*），所以，如果在该实验进行时刻有一个进程试图写12个页面（48KB）以上的数据，那么，该写操作将被延迟，直到延迟写出器减少了脏页面的数量为止。

系统线程

正如前面所提到的，缓存管理器执行的延迟写操作和预读I/O操作，是通过向“公共的关键系统辅助线程池（common critical system worker thread pool）”提交请求来实现的。然而，它限制这些线程的使用数量，对于小内存系统和中等内存系统而言，可使用的线程数比关键的辅助系统线程的总数少一个（对于大内存系统，比总数少两个）。

www.TopSage.com

在内部，缓存管理器将它的工作请求组织成以下两个列表（虽然它们是由同一组执行体辅助线程来提供服务的）：

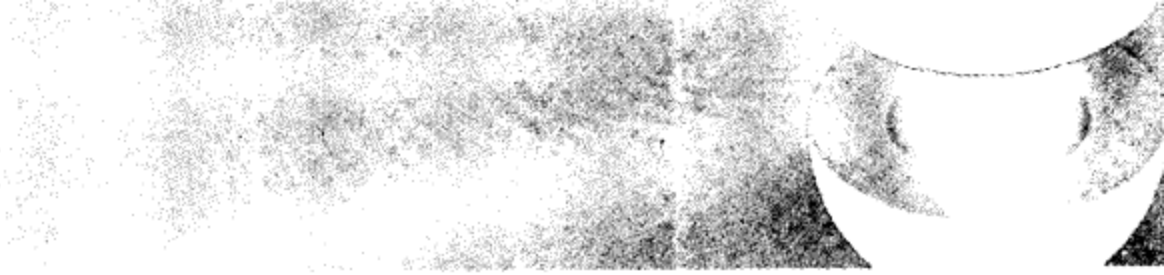
- **快速队列**（*express queue*），用于预读操作；
- **常规队列**（*regular queue*），用于延迟写扫描（寻找脏数据以执行刷新）、滞后写和延迟关闭。

为了跟踪这些辅助线程需要执行的工作项目，缓存管理器创建了其自己内部的**针对每个处理器的预读列表**（*per-processor look-aside list*），这是一个固定长度的列表，每个处理器有一个，表中每一项是一个工作队列项目结构（第7章讨论了预读列表）。工作队列项目的数量取决于系统的大小：对于小内存系统为32，中等内存系统为64，大内存的Windows Professional系统为128，大内存的Windows Server系统为256。

11.8 本章总结

缓存管理器提供了一种高速的、智能的机制来减少磁盘I/O数量和增加系统总体吞吐量。缓存管理器通过在虚拟块的基础上进行数据缓存，从而能够执行智能预读。它依赖于全局内存管理器的映射文件的原语来访问文件数据，从而能够提供特殊的快速I/O机制来减少读和写操作所需要的CPU时间，同时也把所有跟物理内存管理相关的事项留给Windows的全局内存管理器，因此避免了代码的重复，也提高了效率。

www.TopSage.com



File Systems

第12章 文件系统

在本章中，我们首先介绍Microsoft Windows所支持的文件系统格式的总体情况。然后，我们介绍文件系统驱动程序的类型，以及它们的基本操作，包括它们如何与其他的系统组件（比如内存管理器和缓存管理器）打交道。接下来描述如何使用Filemon（可从*www.sysinternals.com*获得）来诊断各种各样的文件系统访问问题。在本章的余下部分，我们关注Windows的原生文件系统（称为NTFS）在磁盘上的结构，以及NTFS的高级特性，比如压缩、可恢复性、配额和加密。

为了全面地理解这一章的内容，你应该熟悉第10章中引入的名词术语，包括术语**卷**（*volume*）和**分区**（*partition*）。你也需要对以下这些额外的术语有所了解。

- **扇区**（*sector*）是存储介质上可由硬件来寻址的块。x86系统的硬盘几乎总是定义512字节的扇区大小。因此，如果操作系统想要修改一个磁盘上的第632个字节，那么，它必须把一个512字节的数据块写到磁盘上的第2个扇区中；
- **文件系统格式**（*File system format*）定义了文件数据在存储介质上的存储方式，它们影响了一个文件系统的各种特性。例如，如果一种格式不允许用户的许可权限与文件和目录相关联，则它就不可能支持安全性。文件系统格式也强制规定了文件大小的限制，以及它所支持的存储设备的大小限制。最后，有些文件系统格式可以非常有效地实现对于大文件或小文件的支持，或者对于大磁盘或小磁盘的支持；
- **簇**（*cluster*）是许多文件系统格式使用的可寻址数据块。簇的大小总是扇区大小的整数倍，如图12.1所示。文件系统格式使用簇来有效地管理磁盘空间；簇的大小大于扇区的大小，它把一个磁盘分成许多可管理的块。簇越大，则潜在的消耗是更多被浪费的磁盘空间，或者称为内部碎片，这是由于文件的大小通常并不恰好是簇大小的整数倍而导致的；

www.TopSage.com

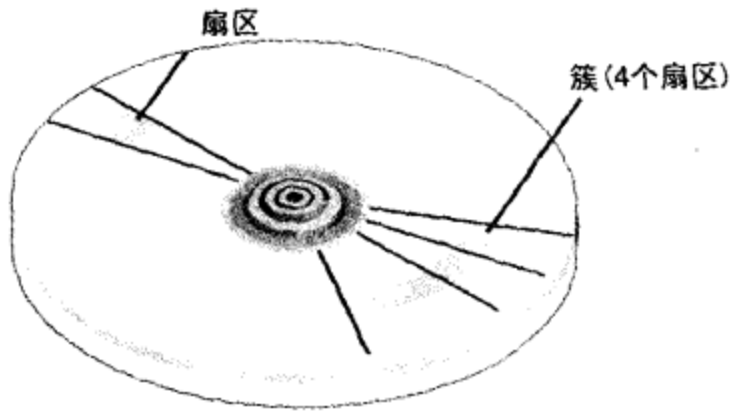


图12.1 一个磁盘上的扇区和簇

- **元数据**（*Metadata*）是为了支持文件系统格式管理而存储在一个卷上的数据。对于应用程序，它通常是不可访问的。例如，元数据包含了一个卷上的文件和目录的放置信息。

12.1 Windows文件系统格式

Windows包含对以下文件系统格式的支持：

- CDFS；
- UDF；
- FAT12、FAT16和FAT32；
- NTFS。

这些格式中的每一种都特别适合于某些特定的环境，在下一节中你将会看到这一点。

CDFS

CDFS （\Windows\System32\Drivers\Cdfs.sys）或CD-ROM文件系统是一个只读文件系统驱动程序，它支持ISO-9660格式的一个超集，以及Joliet磁盘格式的一个超集。ISO-9660格式相对比较简单，它有一些限制，比如最大长度为32字符的ASCII大写名称。而Joliet更为灵活，例如，它支持任意长度的UNICODE名称。如果这两种格式的结构同时出现在一个磁盘上（以便提供最大程度的兼容性），则CDFS使用Joliet格式。CDFS有以下一些限制：

- 文件大小最大为4GB；
- 最多65 535个目录。

CDFS被认为是一种遗留的格式，因为工业界已经采用UDF（Universal Disk Format，通用磁盘格式）作为只读介质的标准。

www.TopSage.com

UDF

Windows的UDF文件系统实现是与OSTA（Optical Storage Technology） UDF兼容的（UDF是ISO-13346格式的一个子集，该格式对于像CD-R和DVD-R/RW之类的格式进行了扩展）。OSTA于1995年针对磁-光存储介质（主要DVD-ROM）定义了UDF，作为一种替代ISO-9660的格式。UDF也被包含在DVD规范中，它比CDFS更加灵活。在Windows 2000上，UDF驱动程序支持UDF 1.02和1.5版；在Windows XP和Windows Server 2003上，它支持2.0和2.01版。UDF文件系统有以下显著特点：

- 目录和文件名可以达到254 ASCII字符或127 UNICODE字符长；
- 文件可以是稀疏的（本章后面定义了稀疏文件）；
- 文件的大小用64位来指定。

对UDF格式虽然在设计时考虑了可重写的介质，但是，Windows UDF驱动程序（\Windows\System32\Drivers\Udfs.sys）仅仅提供了只读支持。而且，Windows没有实现对于其他UDF特性的支持，包括命名的流、访问控制列表或扩展的属性。

FAT12、FAT16和FAT32

Windows对于FAT文件系统的支持主要是为了能够从Microsoft Windows的其他版本升级过来，也是为了在多引导系统中与其他的操作系统保持兼容，以及作为一种软盘格式。Windows FAT文件系统驱动程序是在\Windows\System32\Drivers\Fastfat.sys中实现的。

每一种FAT格式的名称都包含一个数值，指明了该格式用于标识磁盘上的簇所需要的位数。FAT12的12位簇标识符限定了一个分区最多只能存储2^{12}（4096）簇。Windows使用的簇的大小是从512字节至8KB，这限定了一个FAT12卷的大小最多为32MB。因此，Windows使用FAT12作为所有5英寸软盘和3.5英寸软盘的格式，这些软盘最多存储1.44MB数据。



> **注** 所有的FAT文件系统类型都保留了一个卷的前两簇和最后16簇，因此，对于一个FAT12卷，可用簇的数量略少于4096。

FAT16使用16位的簇标识符，它可以处理2^{16}（65 536）个簇。在Windows上，FAT16的簇的大小是从512字节（扇区的大小）至64KB，这限定了FAT16卷的大小最多只能是4GB。Windows使用的簇的大小取决于一个卷的大小。各种大小值如表12.1所示。如果你使用*format*命令或者磁盘管理MMC加载件，将一个小于16MB的卷格式化成FAT，那么，Windows将使用FAT12而不使用FAT16。

www.TopSage.com

表12.1 Windows中FAT16的默认簇大小

卷的大小	簇的大小
0～32 MB	512字节
33 MB～64 MB	1 KB
65 MB～128 MB	2 KB
129 MB～256 MB	4 KB
257 MB～511 MB	8 KB
512 MB～1023 MB	16 KB
1024 MB～2047 MB	32 KB
2048 MB～4095 MB	64 KB

FAT卷被分成几个区域，如图12.2所示。文件分配表（file allocation table）是FAT文件系统格式之所以得名“FAT”的由来。对于卷上的每一簇，文件分配表中都有一个相应的表项。因为文件分配表对于成功地解释一个卷的内容是非常的关键，所以，FAT格式维护了该表的两份拷贝，因此，如果一个文件系统驱动程序或者一致性检查程序（比如Chkdsk）不能访问其中之一（比如，由于磁盘扇区已坏），那么它可以从另一个文件分配表中读取信息。

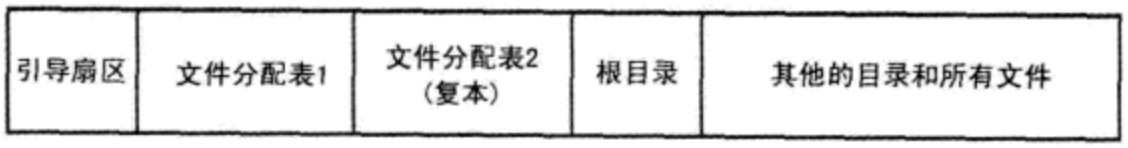


图12.2 FAT格式的组织

文件分配表中的表项定义了文件和目录的文件分配链（file-allocation chain），如图12.3所示。在每条文件分配链上的链接是指向该文件的下一簇位置。文件的目录项存放了该文件的起始簇。文件分配链的最后一项被保留为值0xFFFF（对于FAT16）和0xFFF（对于FAT12）。未使用的簇的FAT项包含0值。在图12.3中你可以看到，FILE1被分配了簇2、3和4；FILE2已被打碎，它使用了簇5、6和8；FILE3只使用簇7。从FAT卷中读取一个文件，将会涉及读取一个文件分配表的大部分，以便经历该文件的分配链。

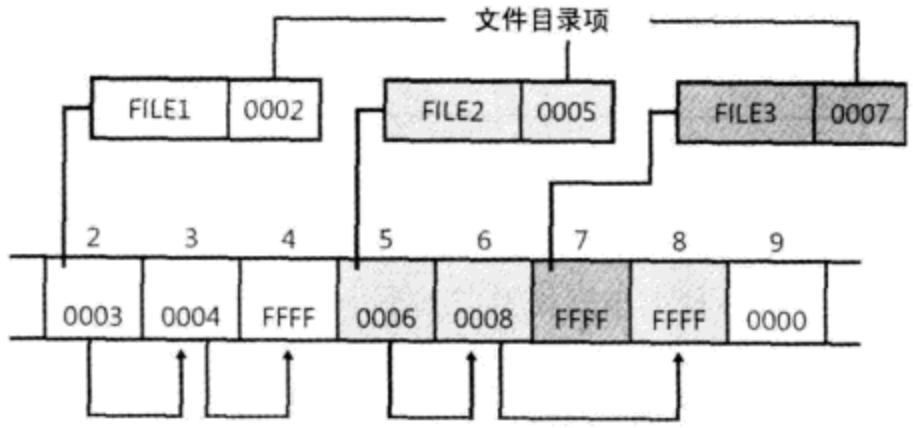


图12.3 示例性的FAT文件分配链

www.TopSage.com

FAT12和FAT16卷的根目录是预先分配的，位于一个卷的开始处，它被分配了足够的空间来存放256个目录项，这使得在根目录中能存储的文件和目录的数量有一个上限（在FAT32的根目录中没有预分配的空间或者大小限制）。每个FAT目录项是32字节，它保存了一个文件的名称、大小、起始簇和时间戳信息（最后访问时间、创建时间，等等）。如果一个文件有一个UNICODE名称，或者不符合MS-DOS 8.3命名规范，则分配额外的目录项来存放此长文件名。补充的目录项位于文件的主项之前。图12.4显示了一个名为“The quick brown.fox”的文件的目录项。系统已经为该名称创建了一个“THEQUI~1.FOX”8.3表示（也就是说，你不会在目录项中看到“.”，因为假设在第8个字符后面总是跟一个“.”），并且使用两个目录项来存储此UNICODE长文件名。图中的每一行包含16个字节。

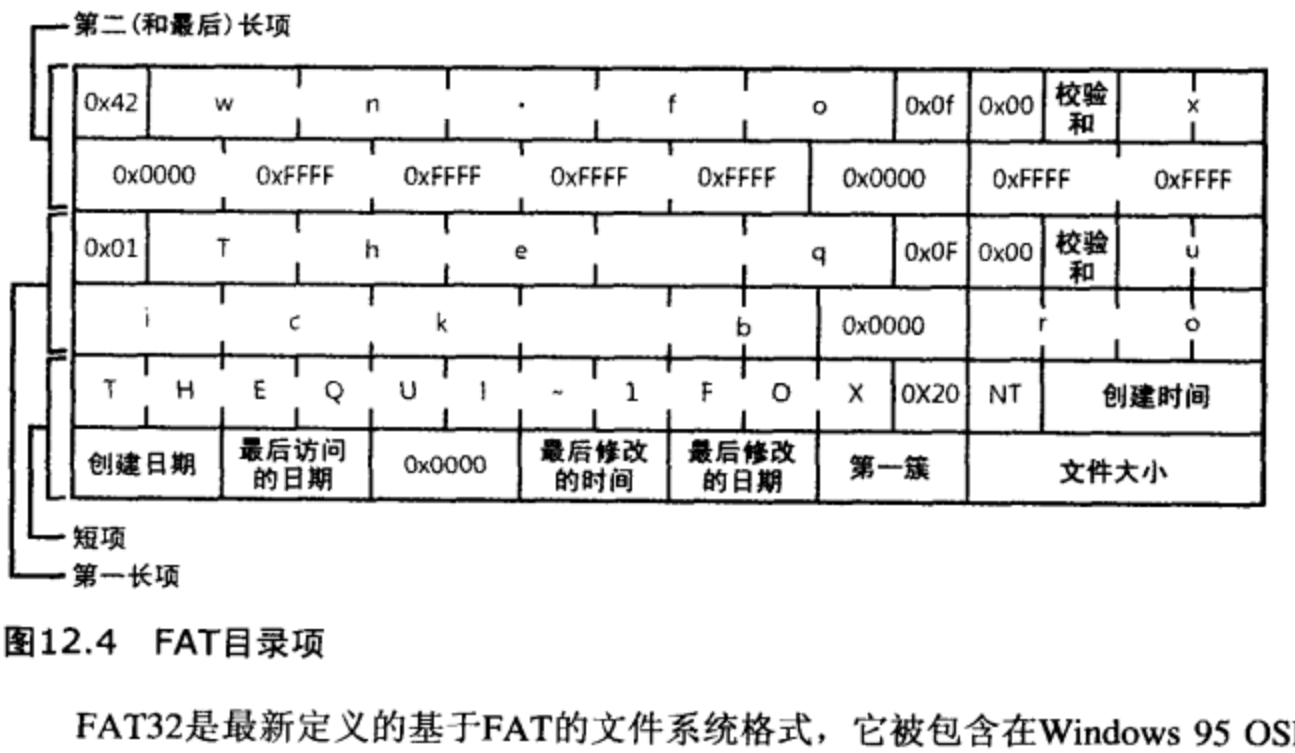


图12.4 FAT目录项

FAT32是最新定义的基于FAT的文件系统格式，它被包含在Windows 95 OSR2、Windows 98和Windows ME（Millennium Edition）中。FAT32使用32位簇标识符，但是保留了高4位，所以实际上它只有28位簇标识符。因为FAT32的簇大小可以大到32KB，所以，FAT32理论上能够处理8TB（terabyte，兆兆字节）的卷。虽然Windows可以与“已有的更大尺寸的FAT32卷（在其他的操作系统中被创建的）”一起工作，但是它限制新的FAT32卷最大只能为32GB。FAT32比FAT16有更高的簇编号，这使得它能够更加有效地管理磁盘；它可以利用512字节的簇来处理多达128MB的卷。表12.2显示了FAT32卷的默认簇大小。

表12.2 FAT32卷的默认簇大小

分区的大小	簇的大小
32 MB ~ 8 GB	4 KB
8 GB ~ 16 GB	8 KB
16 GB ~ 32 GB	16 KB
32 GB	32 KB

www.TopSage.com

除了簇编号的限制值被加大了以外，FAT32胜过FAT12和FAT16的其他优势还包括以下事实：FAT32根目录并不存储在卷的预定义位置处；根目录的大小没有上限；FAT32保存了引导扇区的第二份拷贝（为了可靠性）。FAT32也具有与FAT16一样的一个限制，即文件的最大尺寸是4GB，因为在目录项中文件的大小值被保存为一个32位的值。



注 Windows XP引入了对于DVD-RAM设备的FAT32支持。

NTFS

正如在本章开始时所说的那样，NTFS文件系统是Windows的原生文件系统格式。NTFS使用64位簇编号。这一容量使得NTFS能够处理多达16艾字节（16个10亿GB）的卷；然而，Windows限制NTFS卷的大小为“可用32位簇来寻址的卷大小”，这稍小于256TB（使用64KB的簇）。表12.3显示了NTFS卷的默认簇大小（当你格式化一个NTFS卷时，你可以改变此默认值）。NTFS也支持在每个卷上存放$2^{32}-1$个文件。NTFS格式允许文件的大小可以达到16艾字节，但是，NTFS的实现限制了文件的最大尺寸为16TB。

表12.3 NTFS卷的默认簇大小

卷的大小	默认的簇大小
512 MB或更小	512字节
513 MB～1024 MB （1 GB）	1 KB
1025 MB～2048 MB （2 GB）	2 KB
大于2048 MB	4 KB

NTFS包含了许多高级特性，比如文件和目录的安全性、磁盘配额、文件压缩、基于目录的符号链接，以及加密。最重要的特性之一是**可恢复性**（*recoverability*）。如果一个系统被意外地停止了，则FAT卷的元数据可以处于一种不一致的状态，从而导致大量的文件和目录数据被破坏。NTFS以事务方式记录下元数据的改变情况，所以，文件系统结构可以被修复到一致的状态，因而文件或目录结构的信息不会丢失（然而，文件数据可能会丢失）。

我们将在本章中详细地描述NTFS的数据结构和高级特性。

12.2 文件系统驱动程序总体结构

文件系统驱动程序（FSD）管理文件系统格式。虽然FSD运行在内核模式下，但它们在许多方面与标准的内核模式驱动程序有所不同。可能最重要的是，它们必须向I/O管理器注册成一个FSD，并且它们要大量地与内存管理器打交道。为了提高性能，文件系统驱动程序也常常要依赖于缓存管理器的服务。因此，它们使用的Ntoskrnl导出函数是标准驱动程序使用的Ntoskrnl导出函数的一个超集。为了联编（build）标准的内核模式驱动程序，你需要Windows

www.TopSage.com

DDK。然而，为了联编文件系统驱动程序，你必须使用Windows IFS（可安装文件系统，Installable File System）开发工具箱（有关DDK的更多信息，请参见第1章；有关IFS开发工具箱的更多信息，请参见*www.microsoft.com/whdc/devtools/ifskit*）。

Windows有以下两种不同类型的文件系统驱动程序：

- **本地FSD**管理那些直接连接至计算机的卷；
- **网络FSD**允许用户访问那些连接至远程计算机的数据卷。

本地FSD

本地FSD包括Ntfs.sys、Fastfat.sys、Udfs.sys、Cdfs.sys和Raw FSD（集成在Ntoskrnl.exe中）。图12.5示出的一个简化的视图，说明了本地FSD如何与I/O管理器和存储设备驱动程序打交道。正如在第10章的“卷的挂载”一节中所讲述的，本地FSD有责任向I/O管理器注册。一旦FSD已被注册，I/O管理器就可以在应用程序或系统最初访问卷的时候，调用它来执行卷的识别工作。卷的识别过程涉及检查一个卷的引导扇区，通常也要检查文件系统元数据（以此来检查系统的一致性）。

Windows支持的每个文件系统格式的第一个扇区被保留为该卷的引导扇区。引导扇区包含了足够的信息，所以本地FSD既可以识别出“该扇区所在的卷确实包含了此FSD所管理的一种格式”，又可以定位到任何对于“识别出该卷元数据所在位置”极为必要的其他元数据。

一个本地FSD识别出一个卷时，它创建一个“代表此挂载的文件系统格式”的设备对象。I/O管理器通过卷参数块（VPB）在该卷的设备对象（是由存储设备创建的）和FSD创建的设备对象之间建立一个连接这一VPB的连接导致了I/O管理器将本该发送给卷设备对象的I/O请求，现在重定向到FSD设备对象（有关VPB的更多信息，请参见第10章）。

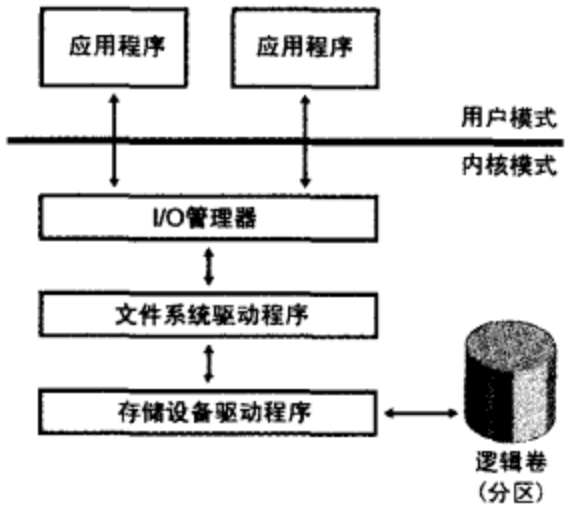


图12.5 本地FSD

www.TopSage.com

为了提高性能，本地FSD通常使用缓存管理器来缓存文件系统的数据，包括元数据。它们也结合了内存管理器，因而，映射的文件也能够得以正确地实现。例如，当一个应用程序试图截断一个文件时，本地FSD为了确保没有其他的进程也映射了该文件截断点之后的部分，必须询问内存管理器。Windows不允许那些“已被一个应用程序映射的文件数据”，再以文件截断或删除的方式被删除。

本地FSD也支持文件系统卸载（dismount）操作，这使得系统可以断开FSD跟卷对象之间的连接。当一个应用程序请求以原始方式访问（raw access）一个卷的磁盘内容时，或者与一个卷相关联的介质发生变化时，卸载就会发生。在卸载以后，当应用程序第一次访问该介质时，I/O管理器会重新为该介质发起一个卷挂载操作。

远程FSD

远程FSD是由两个部件组成的：客户和服务器。客户方的远程FSD允许应用程序访问远程的文件和目录。客户FSD接受来自应用程序的I/O请求，并且将这些请求转译成网络文件系统协议的命令，该FSD将这些命令通过网络发送给服务器方的部件，通常是一个远程FSD。服务器方的FSD监听来自网络连接的命令，并且通过向本地FSD发送I/O请求来实现这些命令，此本地FSD负责管理该命令的目标文件或目录所在的卷。

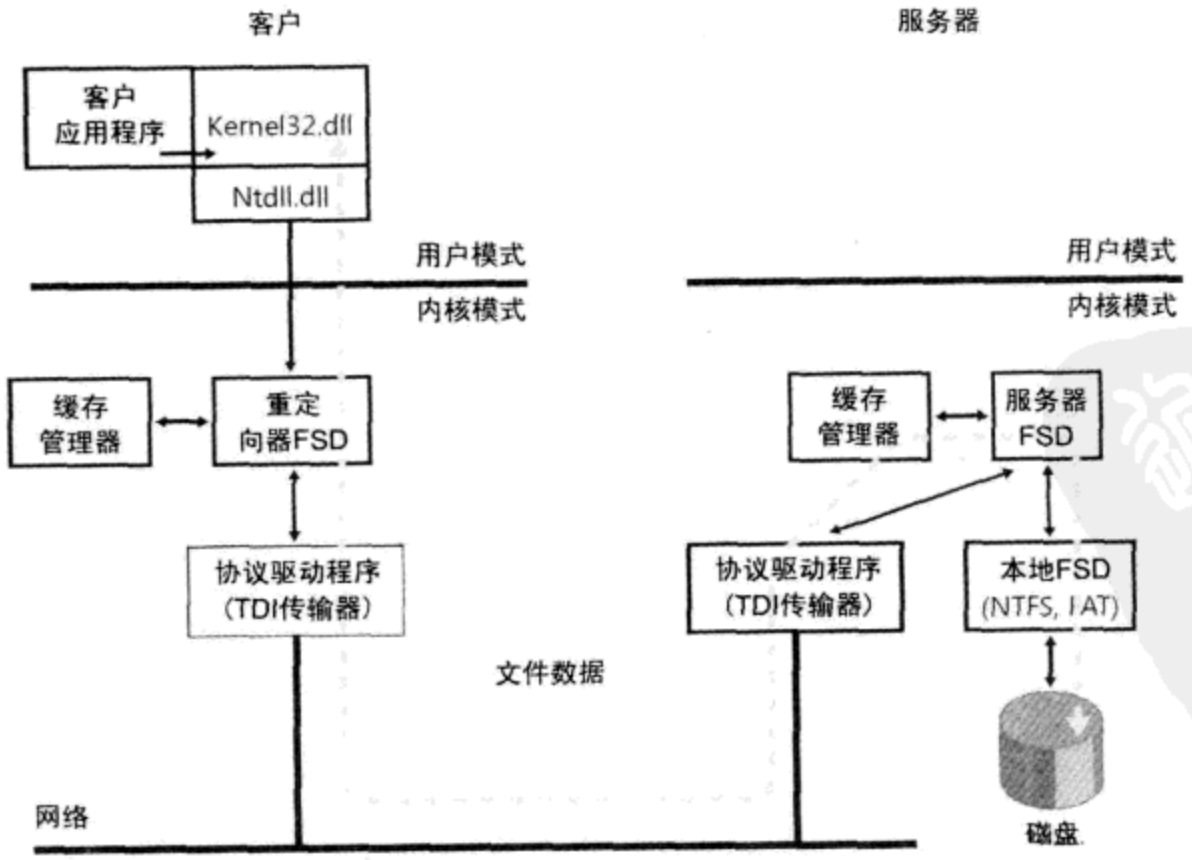


图12.6 CIFS文件共享

www.TopSage.com

Windows包含一个名为“LANMan Redirector”（一个重定向器）的客户方远程FSD和一个名为“LANMan Server”的服务器方远程FSD（\Windows\System32\Drivers\Srv.sys）。该重定向器被实现为一对“端口/小端口驱动程序”组合，其中端口驱动程序（\Windows\System32\Drivers\Rdbss.sys）被实现为一个驱动程序子例程库，而小端口驱动程序（\Windows\System32\Drivers\Mrxsmb.sys）用到了端口驱动程序所实现的服务。另一个重定向器小端口驱动程序是WebDAV（\Windows\System32\Drivers\Mrxdav.sys），它实现了“通过HTTP进行文件访问”的客户部分。这种端口/小端口模型简化了重定向器的开发，因为端口驱动程序是所有远程FSD小端口驱动程序所共享的，它处理了客户方远程FSD与Windows I/O管理器之间打交道的诸多琐碎细节。除了FSD组件以外，LANMan Redirector和LANMan Server都包含了名为Workstation和Server的Windows服务。图12.6显示了客户通过重定向器和服务器FSD来远程访问文件时两者之间的关系。

Windows依赖于公共Internet文件系统（CIFS，Common Internet File System）协议来格式化重定向器和服务器之间交换的消息CIFS是Microsoft的SMB（Server Message Block，服务器消息块）协议的一个版本（有关CIFS的更多信息，请访问*www.cifs.com*）。

如同本地FSD一样，客户方的远程FSD通常使用缓存管理器的服务，将“属于远程文件和目录的文件数据”缓存在本地。然而，客户方远程FSD必须实现一个称为*oplocks*（opportunistic locking，机会锁）的分布式缓存一致协议，因此，当一个应用程序访问远程文件时所看到的数据，与其他计算机上正在运行的、也在访问此相同文件的应用程序所看到的数据是相同的。虽然服务器方远程FSD参与到维护其客户之间的缓存一致性的过程之中，但是，它们并不缓存来自本地FSD的数据，因为本地FSD会缓存它们自己的数据。

当一个客户想要访问一个服务器文件时，它必须首先请求一个oplock。客户指明它可以根据服务器授予的oplock的类型来执行相应的缓存。

有以下三种主要的oplock类型。

- 级别I oplock（Level I oplock）。当客户要对一个文件进行独占访问时，它需要这种锁。如果客户针对一个文件持有这种类型的oplock，那么，它可以在客户系统上缓存读和写操作；
- 级别II oplock（Level II oplock）代表一个共享的文件锁。持有级别II oplock的客户可以缓存读操作，但是对文件的写操作使该级别II oplock锁无效；
- 批oplock（Batch oplock）是最为宽容的一种oplock。持有这种oplock的客户既可以缓存对文件的读和写操作，也可以在不请求其他额外oplock的情况下打开和关闭该文件。批oplock通常只是被用来支持批文件的执行，当这些文件执行时，它们可以反复地打开和关闭一个文件。

如果客户没有oplock，那么，它不可以在本地缓存读和写的数据，相反，它必须从服务器获取数据，并将所有的修改直接发送给服务器。

www.TopSage.com

图12.7中所示的例子有助于表现oplock的操作。服务器自动向第一个客户授予一个级别I oplock，让它打开一个服务器文件来访问。客户方的重定向器将该文件的读和写的数据缓存在客户机器的文件缓存中。如果第二个客户打开了该文件，那么，它也请求一个级别I oplock。然而，因为现在有两个客户在访问同一个文件，所以，服务器必须采取相应的措施，以便向两个客户展示一致的文件数据视图。如果第一个客户已经写了该文件，像图12.7中所示的那样，那么，服务器撤回它的oplock，对两个客户都不授予oplock。当第一个客户的oplock被撤回（或者被**打破**）时，该客户将其缓存的该文件的数据全都刷新到服务器上。

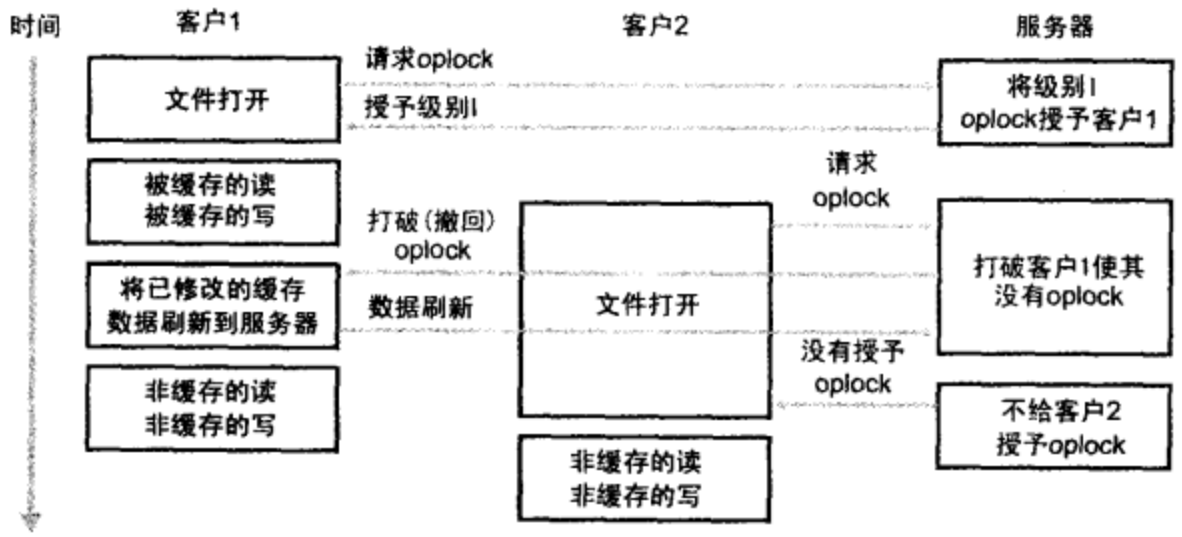


图12.7 Oplock例子

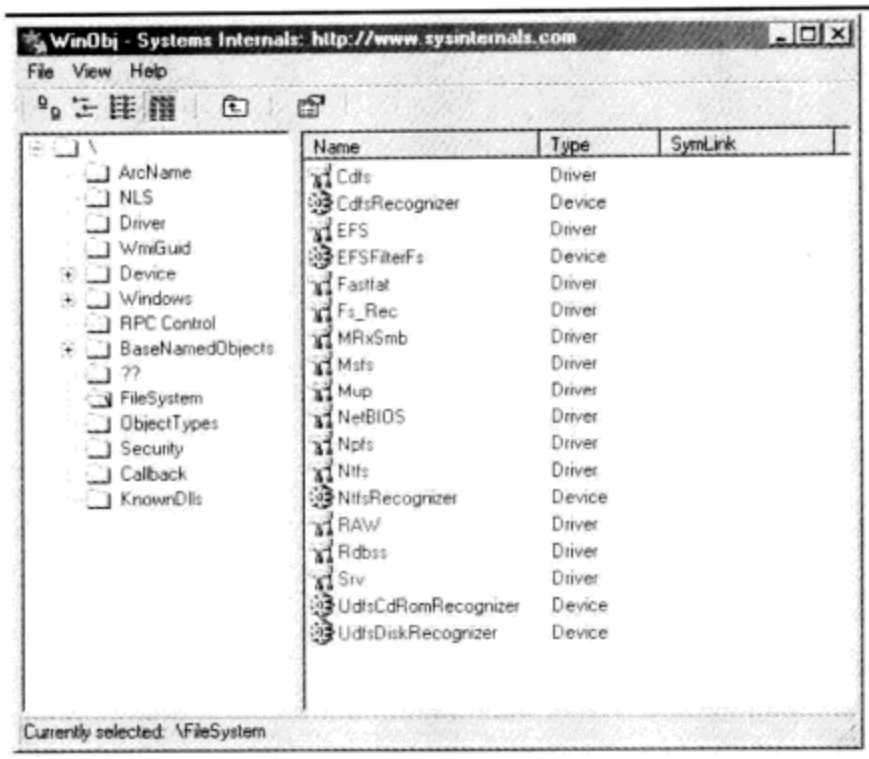
如果第一个客户还没有写过该文件，那么，第一个客户的oplock将被突变为级别II oplock，这与服务器授予第二个客户的oplock是相同的类型。现在，两个客户都可以缓存读操作，但是，如果其中任何一个对文件进行写操作，则服务器撤回它们的oplock，以便开始非缓存的操作。一旦oplock已被打破，对于一个文件的同一个打开实例，不会再授予oplock。然而，如果客户关闭一个文件，然后再重新打开该文件，则服务器根据其他哪些客户也打开了该文件，以及它们是否至少有一个已经写过该文件，来重新评估应该授予此客户哪一级别的oplock。



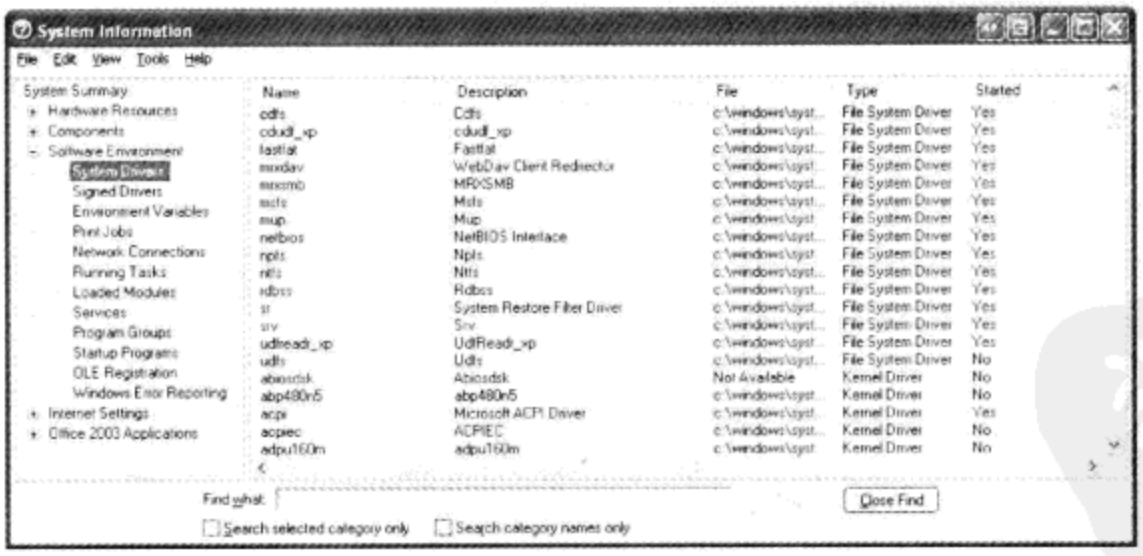
实验：查看已注册文件系统的列表

当I/O管理器将一个设备驱动程序加载到内存中时，它通常对它所创建的、用来代表该驱动程序的驱动程序对象进行命名，因而该驱动程序对象可以被放到\Drivers对象管理器目录中。对于I/O管理器加载进来的、凡是Type属性值为SERVICE_FILE_SYSTEM_DRIVER （2）的驱动程序，I/O管理器将它的驱动程序对象放在\FileSystem目录中。因此，使用一个诸如Winobj（来自*www.sysinternals.com*）这样的工具，你可以看到系统中已经注册的文件系统，如下面的屏幕截图所示（注意，有些文件系统驱动程序也把设备对象放在\FileSystem目录中）。

www.TopSage.com



另一种查看已注册文件系统的办法是，运行System Information查看器。在Windows 2000上，运行计算机管理MMC加载件（Computer Management MMC snap-in），在System Information节点的Software Environment下选择Drivers；在Windows XP和Windows Server 2003上，从Start菜单的Run对话框中运行Msinfo32，并且在Software Environment下选择System Drivers。点击Type栏，可以对驱动程序的列表进行排序，于是，Type属性为SERVICE_FILE_SYSTEM_DRIVER的驱动程序都聚集到一起。



注意，仅仅因为一个驱动程序注册成一个文件系统驱动程序类型，并不意味着它是一个本地或远程FSD。例如，上图中可以看到的Npfs（Named Pipe File System，命名管道文件系统）是一个支持命名管道，但实现了一个私有名字空间的网络API驱动程序，因此，它在某些方面像一个文件系统驱动程序。第13章有一个实验说明了如何显示出Npfs名字空间。

www.TopSage.com

文件系统操作

应用程序和系统通过两种方法来访问文件：通过文件I/O函数（比如ReadFile和WriteFile）直接进行；通过读写其地址空间中代表一个映射文件内存区的内存间接地进行（有关映射文件的更多信息，请参见第7章）。图12.8是一个简化图，它显示了在这些文件系统操作中涉及的各个组件，以及它们之间的交互方式。正如你所看到的，FSD可以通过几条路径来调用：

- 通过一个正在执行显式文件I/O的用户或系统线程来调用；
- 通过内存管理器的修改页面写出器和映射页面写出器来调用；
- 通过缓存管理器的延迟写出器来间接调用；
- 通过缓存管理器的预读线程来间接调用；
- 通过内存管理器的页面错误处理器来调用。

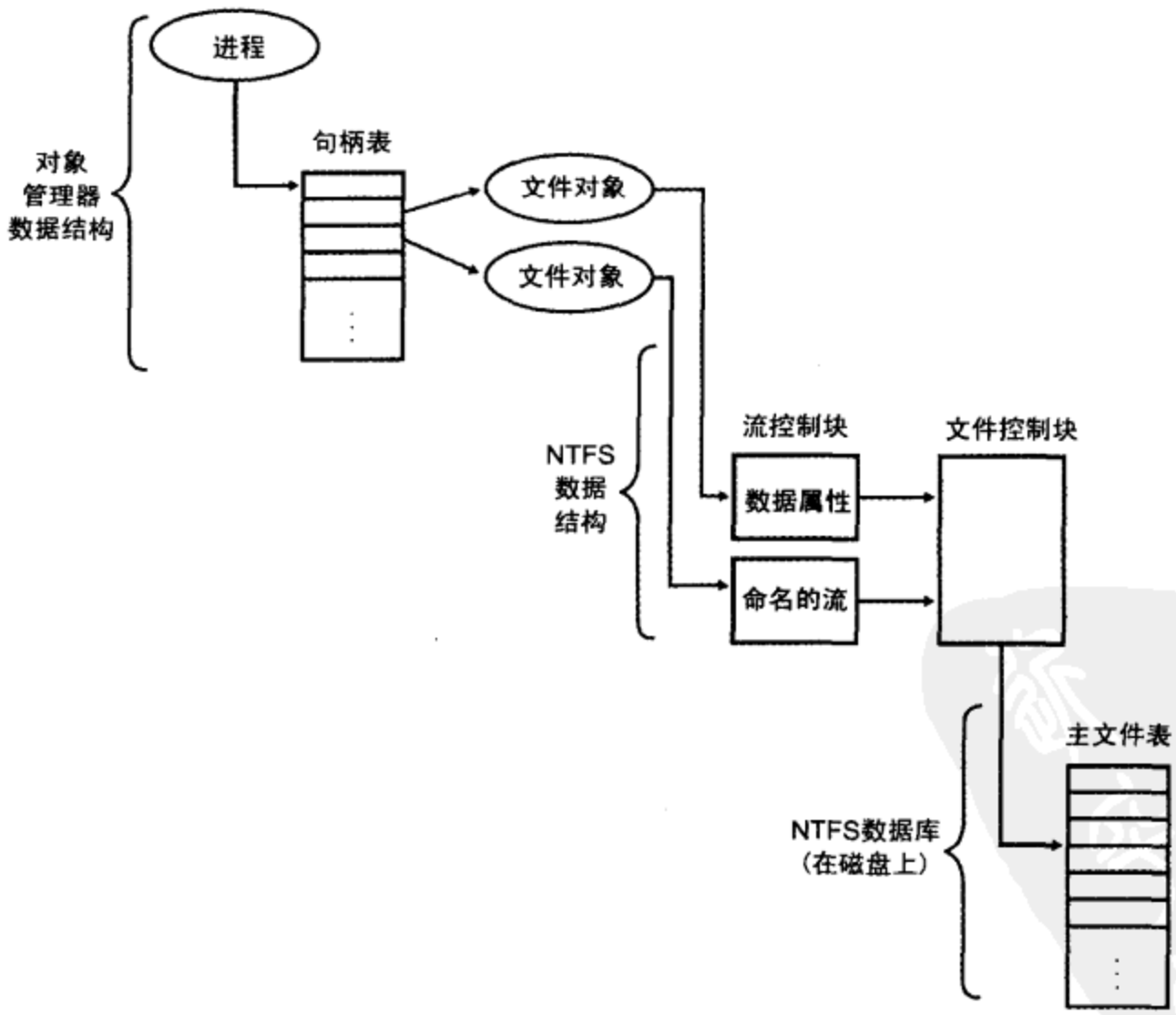


图12.8 文件系统I/O中涉及的组件

www.TopSage.com

下面的小节讲述了以上每一种情形的具体情况，以及FSD为了响应每一种调用而通常采取的步骤。你将会看到，FSD多么地依赖于内存管理器和缓存管理器。

显式文件I/O

应用程序访问文件的最显然途径是，调用Windows的I/O函数，比如*CreateFile*、*ReadFile*和 *WriteFile*。应用程序通过*CreateFile*来打开一个文件，然后将*CreateFile*返回的句柄传递给其他的Windows函数，从而对该文件进行读、写或者删除操作。*CreateFile*函数是在Kernel32.dll（Windows的客户方DLL）中实现的，它调用了原生函数*NtCreateFile*；对于应用程序传递给它的路径，它形成一个完整的、相对于根目录的路径名（需要处理路径名中的“.”和“..”符号），并且在前面加上“\??”（例如，形成路径“\??\C:\Daryl\Todo.txt”）。

*NtCreateFile*系统服务使用*ObOpenObjectByName*来打开该文件，它解析以对象管理器根目录作为开始的名称和路径名的第一个部分（“\??”）。第3章包含了有关对象管理器名称解析的完整描述，以及进程设备表的使用。在这里我们将焦点集中在卷驱动器字母的查找过程上，重新看一下所遵从的步骤。

对象管理器采取的第一个步骤是，将\??翻译为该进程对象的设备表结构的DosDevicesDirectory域所引用的、针对每个会话的名字空间目录。在不安装终端服务的Windows 2000系统上，DosDevicesDirectory域引用了\??目录；在安装了终端服务的Windows 2000系统上，该设备表引用了一个针对每个会话的目录，其中存储了代表所有有效卷驱动器字母的符号链接。然而，在Windows XP和Windows Server 2003上，通常只有网络共享体的卷名称才被存储在针对每个会话的目录中，所以在这些系统上，当一个名称（在这个例子中为C:）没有出现在此会话目录中时，对象管理器在与会话目录相关联的设备表的GlobalDosDevicesDirectory域所引用的目录中重新开始它的搜索。GlobalDosDevicesDirectory总是指在\Global??目录上，这正是Windows XP和Windows Server 2003保存本地卷的卷驱动器字母的地方（更多的信息，请参见第3章中的“会话名字空间”一节）。

卷驱动器字母的符号链接指向\Device下的卷设备对象，所以，对象管理器碰到卷对象时，将路径名的余下部分交给“I/O管理器为设备对象而注册的解析函数”*IopParseDevice*（对于动态磁盘上的卷，符号链接指向一个中间符号链接，此中间符号链接又指向一个卷设备对象）。图12.9显示了如何通过对象管理器名字空间来访问卷对象。该图显示了在不安装终端服务的Windows 2000上，符号链接\??\C:如何指向卷设备对象\Device\HarddiskVolume1。

www.TopSage.com

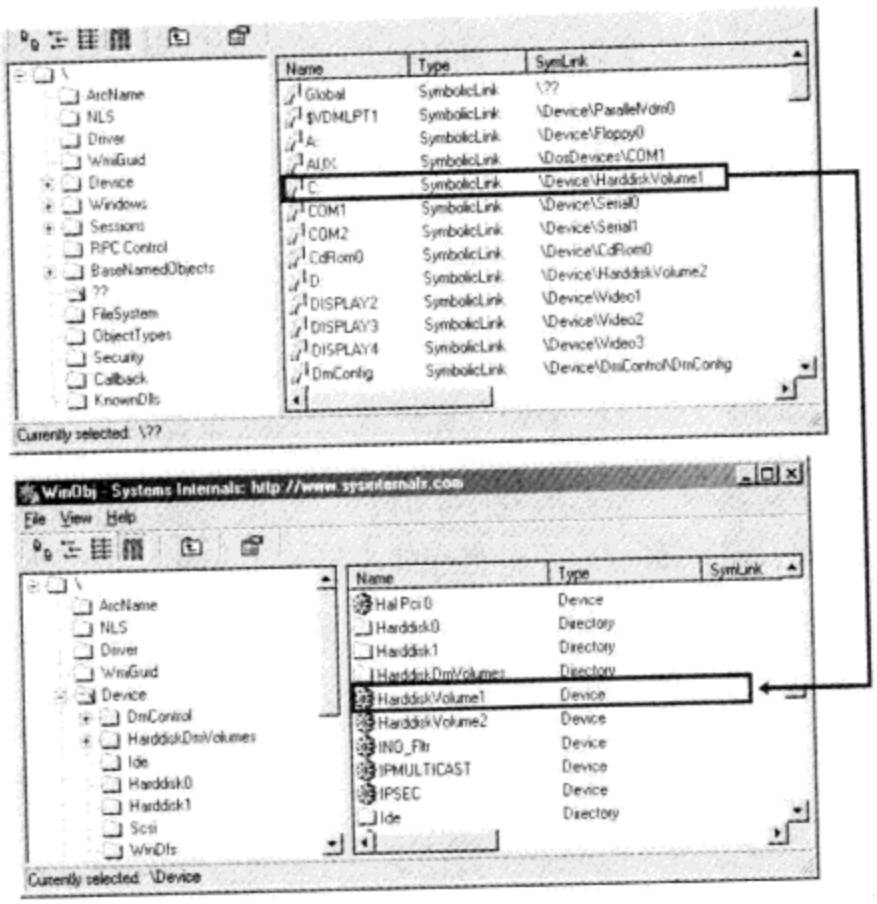


图12.9 驱动器字母的名称解析

在锁住了调用者的安全环境并且从调用者的令牌中获得了安全信息以后，*IopParseDevice*创建一个类型为IRP_MJ_CREATE的I/O请求包（IRP），创建一个文件对象来存储被打开文件的名称，根据卷设备对象的VPB找到该卷的已挂载的文件系统设备对象，并且使用*IoCallDriver*，将该IRP传递给拥有该文件系统设备对象的文件系统驱动程序。

当一个FSD接收到一个类型为IRP_MJ_CREATE的IRP时，它查找此特定的文件，执行安全验证，而且，如果该文件存在，并且用户确实具有他所请求的访问许可，则返回一个成功代码。对象管理器在该进程的句柄表中创建一个指向该文件对象的句柄，然后通过调用链将该句柄传递回去，最后，该句柄作为*CreateFile*的返回参数到达应用程序中。如果文件系统未能成功地完成创建操作，则I/O管理器删除它所创建的文件对象。

我们跳过了有关“FSD如何在一个卷上定位到被打开的文件”的细节，但是，*ReadFile*函数调用操作也共用了“FSD与缓存管理器和存储驱动程序之间的许多交互过程”。从*ReadFile*调用至内核的路径与*CreateFile*调用至内核的路径相同，但是，*NtReadFile*系统服务并不需要执行名称查询——它通过对象管理器，将*ReadFile*传递过来的句柄翻译成一个文件对象指针。如果该句柄指明了“当该文件被打开时，调用者获得了读该文件的许可”，那么，*NtReadFile*继续进行，创建一个IRP_MJ_READ类型的IRP，并且将它发送给该文件所在的FSD。*NtReadFile*获得此FSD的设备对象（存储在文件对象之中），调用*IoCallDriver*；I/O管理器从设备对象中找到此FSD，并且将IRP交给此FSD。

www.TopSage.com

如果被读的文件可以被缓存的话（也就是说，在该文件被打开时，传递给*CreateFile*的标志中没有FILE_FLAG_NO_BUFFERING），那么，FSD查看该文件对象的缓存特性是否已被启动。如果一个文件对象的缓存特性已被启动，则该文件对象中的*PrivateCacheMap*域指向一个私有的缓存表数据结构（在第11章中我们曾经描述过）。如果FSD尚未初始化该文件对象的缓存特性（当一个文件对象被第一次读或写的时候，确实是这样的），则*PrivateCacheMap*域将为空。FSD调用缓存管理器的*CcInitializeCacheMap*函数来初始化缓存特性，此函数会导致缓存管理器创建一个私有的缓存表，同时，如果另一个文件对象也指向同一个文件，而且它也没有启动缓存特性，则还要创建一个共享的缓存表和一个内存区对象。

FSD在检查了该文件的缓存特性已被打开以后，它把所请求的文件数据从缓存管理器的虚拟内存中拷贝到该线程通过*ReadFile*函数传递进来的缓冲区中。文件系统在一个try/except块中执行此拷贝操作，所以，它可以捕获到任何由于无效的应用程序缓冲区导致的错误。文件系统用来执行此拷贝操作的函数是缓存管理器的*CcCopyRead*函数。*CcCopyRead*接受一个文件对象、文件偏移和长度作为参数。

缓存管理器执行*CcCopyRead*时，获得一个指向共享缓存表的指针，该指针被存储在文件对象中。在第11章中曾经提到过，共享的缓存表保存了指向虚拟地址控制块（VACB）的指针，文件的每个256-KB块对应一个VACB。如果该文件被读部分的VACB指针是空，则*CcCopyRead*申请一个VACB，在缓存管理器的虚拟地址空间中保留一个256-KB的视图，并且利用*MmMapViewInSystemCache*将该文件的这一特定部分映射到该视图中。然后，*CcCopyRead*简单地把文件数据从映射的视图中拷贝到传递进来的缓冲区中（最初传递给*ReadFile*的那个缓冲区）。如果这些文件数据不在物理内存中，则此拷贝操作会产生页面错误，进而*MmAccessFault*会处理这些页面错误。

当一个页面错误发生时，*MmAccessFault*检查引发此错误的虚拟地址，并且在引发该错误的进程的VAD树中找到相应的虚拟地址描述符（VAD）（有关VAD树的更多信息，请参见第7章）。在这里的情形下，此VAD描述了缓存管理器中该文件被读部分的映射视图，所以，*MmAccessFault*调用*MiDispatchFault*来处理一个有效虚拟内存地址上的页面错误。*MiDispatchFault*找到控制区域（VAD指向此控制区域），并且通过控制区域再找到代表此已打开文件的文件对象（如果该文件被打开不只一次，则可能有一个文件对象列表，通过其私有缓存表中的指针链接起来）。

有了文件对象以后，*MiDispatchFault*调用I/O管理器函数*IoPageRead*来构建一个IRP（类型为IRP_MJ_READ），并且将该IRP发送给此文件对象指向的设备对象所属的FSD。因此，文件系统被重入，以便读取到它通过*CcCopyRead*所请求的数据，但是，这一次的IRP被标记为非缓存的换页I/O。这些标志告知FSD，它应该直接从磁盘上获取文件数据。FSD的做法是，确定磁盘上的哪些簇包含了所请求的数据，并且向负责“该文件所在的卷设备对象”的卷管理器发送IRP。该FSD的设备对象中的卷参数块（VPB）域指向了此卷设备对象。

www.TopSage.com

虚拟内存管理器等待FSD完成此IRP的读操作，然后将控制权返回给缓存管理器，缓存管理器又继续执行刚才被页面错误打断的拷贝操作。当*CcCopyRead*完成时，FSD将控制权返回给刚才调用*NtReadFile*的线程，所请求的文件数据已经被拷贝到该线程的缓冲区中了（在缓存管理器和虚拟内存管理器的帮助下）。

*WriteFile*的路径也类似，只不过*NtWriteFile*系统服务产生一个类型为IRP_MJ_WRITE的IRP，并且FSD调用的是*CcCopyWrite*，而不是*CcCopyRead*。而*CcCopyWrite*如同*CcCopyRead*一样，也保证了文件中被写的部分首先被映射到缓存中，然后将*WriteFile*接受的缓冲区拷贝到缓存中。

如果一个文件的数据已经被存储在系统的工作集中，那么，我们刚才讨论的情形有一些变化形式。如果一个文件的数据已经被存储在缓存中了，那么，*CcCopyRead*不会招致页面错误。而且，在特定的条件下，*NtReadFile*和*NtWriteFile*调用一个FSD的快速I/O入口点，而不是立即构建IRP并发送给FSD。以下是其中一些这样的条件：文件中被读的部分必须驻留在文件的前4GB中，文件可以没有锁，文件中被读或写的部分必须落在该文件当前已被分配的大小范围以内。

对于绝大多数FSD，快速I/O读和写的入口点调用了缓存管理器的*CcFastCopyRead*和*CcFastCopyWrite*函数。标准拷贝例程的这些变化函数需要确保：在拷贝操作执行以前，文件的数据已被映射到文件系统缓存中。如果这个条件不满足的话，*CcFastCopyRead*和*CcFastCopyWrite*会指示快速I/O不可能实现。当快速I/O不可能实现时，*NtReadFile*和*NtWriteFile*又回到创建IRP的途径上（关于快速I/O的完整描述，请参见第11章中的“快速I/O”一节）。

内存管理器的修改页面写出器和映射页面写出器

内存管理器的修改页面写出器和映射页面写出器线程定期被唤醒过来，或者当可用内存变得很低时也会被唤醒，以便将那些已被修改的页面刷新到磁盘上。这些线程调用*IoAsynchronousPageWrite*来创建IRP_MJ_WRITE类型的IRP，并且将页面写到一个页面文件中，或者写到一个被映射后又做了修改的文件中。如同*MiDispatchFault*创建的IRP一样，这些IRP也被标志为非缓存的换页I/O。因此，FSD绕过文件系统的缓存，直接向一个存储驱动程序发出IRP，从而将内存数据写到磁盘上。

缓存管理器的延迟写出器（Lazy Writer）

缓存管理器的延迟写出器线程也在写这些已修改页面的任务中扮演了一个角色，因为它定期地对它所知道的、在缓存中已被映射的并且已经变脏的文件区域视图执行刷新操作。缓存管理器通过调用*MmFlushSection*来执行此刷新操作，而且，此刷新操作也触发内存管理器将所刷新区域部分的所有已修改页面写到磁盘上。如同修改页面写出器和映射页面写出器一样，*MmFlushSection*使用*IoSynchronousPageWrite*，将数据发送给FSD。

www.TopSage.com

缓存管理器的预读线程

缓存管理器包含一个从文件中预读数据的线程，其职责是：在一个应用程序、驱动程序或者系统线程显式地请求文件数据以前，试图提前从文件中读取数据。预读线程利用一个文件上过去已经执行过的读操作历史（这些信息保存在文件对象的私有缓存表中），来确定应该读取多少数据。该线程执行预读操作时，只是简单地将它想要读取的文件部分映射到缓存中（必要时分配VACB），并且访问一下此映射的数据即可。这些内存访问所引起的页面错误，会调用到页面错误处理器，而页面错误处理器将会把这些页面读入到系统的工作集中。

内存管理器的页面错误处理器

我们讲述了如何在显式文件I/O和缓存管理器预读操作的环境下使用页面错误处理器来读取数据，但是，应用程序在访问一个映射文件的视图所在的虚拟内存时，如果碰到此文件这些部分所处的页面尚未成为该应用程序工作集的一部分，则页面错误处理器也会被调用。内存管理器的*MmAccessFault*处理器也按照“当缓存管理器从*CcCopyRead*或*CcCopyWrite*中产生一个页面错误时所采取的步骤”那样，遵从同样的步骤，通过*IoPageRead*向该文件所在的文件系统发送IRP。

文件系统过滤型驱动程序

叠加在一个文件系统驱动程序之上的过滤型驱动程序称为**文件系统过滤型驱动程序**（*file system filter driver*）。（有关过滤型驱动程序的更多信息，请参见第9章。）它们能够看到所有的文件系统请求，并且可以修改或完成这些请求，这种能力促成了各种各样的应用，比如远程文件复制服务、文件加密、高效地备份，以及文件许可控制。每一种商业的、在访问时进行检查的病毒扫描程序都包含一个文件系统过滤型驱动程序，它截取到当应用程序打开一个文件时发出的IRP_MJ_CREATE命令所附载的IRP。在把该IRP传递给此命令所指向的文件系统驱动程序以前，病毒扫描程序检查被打开的那个文件，以确保它是无病毒的干净文件。如果该文件是干净的，则病毒扫描程序传递此IRP，但如果该文件已被病毒感染，则病毒扫描程序与它所关联的Windows服务进程进行通信，以便隔离或清理此文件。如果该文件不可能被清理干净，则驱动程序使该IRP失败（通常伴随一个访问被拒绝的错误），因而病毒不会被激活。

在这一节中，我们将描述两个特殊的文件系统过滤型驱动程序的操作，即Filemon和系统恢复（System Restore）操作。Filemon是一个可以从*www.sysinternals.com*下载的文件系统活动监视工具，本书几乎通篇都在使用该工具，它是一个被动的过滤型驱动程序。也就是说，它不会修改应用程序和文件系统驱动程序之间的IRP流。而系统恢复（System Restore）则是Windows XP中引入的一项特性，它使用一个文件系统过滤型驱动程序来观察关键的系统文件上的变化，并且做好备份，因而，这些文件可以被恢复到它们之前在某个特定时间点上的状态，此时间点被称为“**恢复点**（*restore point*）”。

www.TopSage.com



> 注 Windows XP Service Pack 2和Windows Server 2003包含了文件系统过滤器管理器（Filesystem Filter Manager，\Windows\System32\Drivers\Fltmgr.sys），作为文件系统过滤型驱动程序的端口/小端口模型的一部分，并且，此文件系统过滤器管理器对于Windows 2000也将可以使用。文件系统过滤器管理器把过滤型小端口驱动程序与Windows I/O系统连接起来；对于查询文件名、附载到卷，以及与其他过滤型驱动程序打交道这样的任务，它都提供了相应的服务，因此，文件系统过滤器管理器可以极大地简化过滤型驱动程序的开发。软件厂商，包括Microsoft在内，可以编写新的文件系统过滤型驱动程序，也可以把现有的过滤型驱动程序移植到文件系统过滤器管理器所提供的框架中。

Filemon

Filemon的工作方式是：当你在系统引导后第一次运行它时，它从可执行映像（Filemon.exe）中提取出一个文件系统过滤型设备驱动程序（Filem.sys），并且将该驱动程序安装到内存中，然后从磁盘上删除此驱动程序。通过Filemon的GUI，你可以指示它监视那些已被分配了驱动器字母的本地卷、网络共享体、命名管道（named pipe）和邮件槽（mail slot）上的文件系统活动。该驱动程序接收到一个命令来启动对一个卷的监视操作时，创建一个过滤型设备驱动程序，并且将其附载到代表“该卷上的一个已挂载文件系统”的设备对象上。例如，如果NTFS驱动程序已经挂载了一个卷，则Filemon的驱动程序利用I/O管理器函数*IoAttachDeviceToDeviceStackSafe*，将它自己的设备对象附载到NTFS的设备对象上。在附载操作以后，I/O管理器将那些目标指向此底下设备对象（在这个例子中，即NTFS设备对象）的IRP，重定向到拥有此附载设备的驱动程序上，在这个例子中即Filemon。

Filemon驱动程序截取一个IRP时，在一个非换页的内核缓冲区中记录下有关此IRP的命令的信息，包括目标文件名和其他与该命令相关的参数（比如读和写的长度和偏移）。Filemon GUI每秒钟两次向Filemon的接口设备对象发送一个IRP，请求拷贝一份包含最新文件系统活动的缓存区数据，并且在它的输出窗口中显示这些活动。有关Filemon的用法，在本章后面的“诊断文件系统的问题”一节中有进一步的描述。

系统恢复

系统恢复（System Restore）机制最初以一种甚为初级的形式出现在Windows ME（Millennium Edition）中，现在，它提供了一种办法，可把一个Windows XP系统恢复到以前的一个已知是好的点上，如若不然，你就得重新安装一个应用程序，甚至整个操作系统（Windows 2000或Windows Server 2003不提供系统恢复机制）。例如，如果你安装了一个或多个应用程序，或者做了其他的系统文件改变，或者改变了注册表，或者同时改变了这两者，因而引起应用程序失败，那么，你可以使用系统恢复机制，使得系统文件和注册表恢复到改变之前的状态。当你安装了一个应用程序，而该应用程序所做的修改是你所不希望的时候，系统恢复机制尤为有用。与Windows XP兼容的安装程序可以跟系统恢复机制协同工作，它们在安装开始以前先创建一个“恢复点”。

www.TopSage.com

系统恢复机制的核心位于一个名为SrService的服务当中，它执行的是一个DLL（\Windows\System32\Srsvc.dll），运行于一个通用服务宿主（\Windows\System32\Svchost.exe）进程实例中（关于Svchost的描述，请参见第4章）。该服务的任务有两个方面：一是自动创建恢复点；二是导出一个API，从而诸如安装程序这样的应用可以手动地激发恢复点的创建操作。系统恢复机制从HKLM\Software\Microsoft\System Restore中读取它的配置参数，包括必须有多少磁盘空间它才能够工作，以及每隔多长时间创建一次自动的恢复点。在默认情况下，在一个未签名的设备驱动程序被安装以前该服务会创建一个恢复点，并且，它试着每隔24小时创建一个自动的检查点（checkpoint）（有关驱动程序签名的信息，请参见第9章）。如果在系统恢复机制的参数键HKLM\System\CurrentControlSet\Services\SR\Parameters的下面，设置了DWORD注册表值RPGlobalInterval，则它将改变默认的时间间隔，并且以秒为单位指定了自动恢复点之间的最小时间间隔值。

系统恢复服务在创建一个新的恢复点的同时，创建一个恢复点目录，然后对一组关键的系统文件做一个“快照”，其中包括系统储巢（hive）和用户轮廓储巢、WMI配置信息、IIS元基（metabase）文件（如果安装了IIS），以及COM注册数据库。然后，系统恢复文件系统过滤型驱动程序（\Windows\System32\Drivers\Sr.sys）开始跟踪文件和目录的变化；对于被删除或修改的文件，在恢复点中保存一份拷贝；在恢复点痕迹日志（tracking log）中记录下其他的变化，比如目录的创建和删除。

恢复点数据是以每个卷为基础来维护的，所以，记录的日志和保存的文件都被存储在这些文件的原始卷的\System Volume Information_restore{*XX-XXX-XXX*}目录下（这里X代表了系统分配给该计算机的GUID）。恢复目录包含了恢复点子目录，其名称的形式为RP*n*，这里*n*是恢复点的惟一标识符。构成一个恢复点初始快照的那些文件被存储在该恢复点的Snapshot目录下。

系统恢复（System Restore）驱动程序所拷贝的备份文件都被赋予了惟一名称，比如A0000135.dll，它们位于某个适当的恢复点目录下，这些文件名反映了标识符的分配，同时保留了原来的文件扩展名。一个恢复点可以有多个痕迹日志（tracking log），每个日志有一个像change.log.*N*这样的名称，其中*N*是惟一的痕迹日志ID。痕迹日志中的纪录包含了有关一个文件或目录上所做修改的信息，这些信息足够被用来执行撤销（undo）操作。例如，如果一个文件被删除了，则针对该操作的痕迹日志项保存了该文件的一份拷贝在恢复点中的名称（例如A0000135.dll），以及该文件的原始长文件名和短文件名。每次当当前的痕迹日志增长到超过1MB时，系统恢复驱动程序就会开始一个新的痕迹日志。图12.10显示了当系统恢复驱动程序为了响应修改操作而更新一个恢复点时的文件系统请求流程。

www.TopSage.com

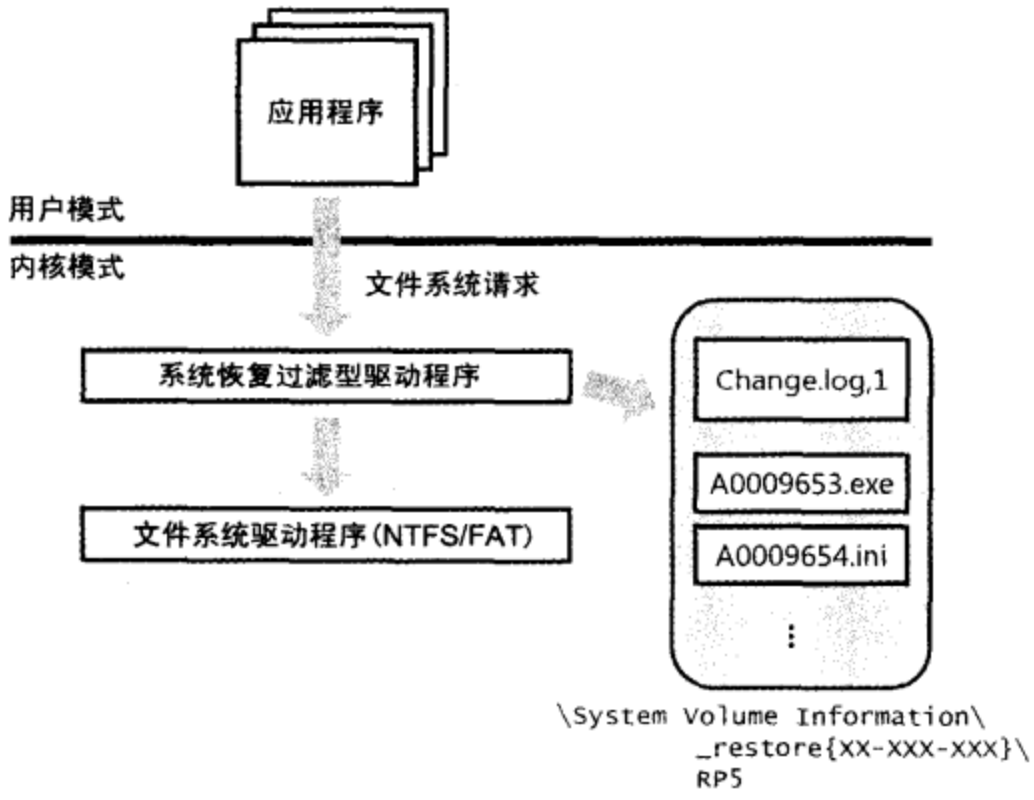


图12.10 系统恢复过滤型驱动程序的操作

图12.11显示了一个系统恢复目录的屏幕截图，其中包含了几个恢复点子目录，以及对应于恢复点1的子目录的内容。注意，通过用户账户是无法访问\System Volume Information目录的，甚至管理员账户也不可访问，但是，本地系统账户可以访问。为了查看此文件夹的内容，请利用*www.sysinternals.com*的PsExec工具，并遵照以下步骤：

```
C:\WINDOWS\SYSTEM32>psexec -s cmd

PsExec v1.55 - Execute processes remotely
Copyright (C) 2001-2004 Mark Russinovich
Sysinternals - www.sysinternals.com


Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\WINDOWS\system32>cd \system*

C:\System Volume Information>cd _rest*

C:\System Volume Information\_restore{987E0331-0F01-427C-A58A-7A2E4AABF84D}>
```

一旦已经到了系统恢复目录下，你就可以使用DIR命令来检查该目录的内容，或者转到与恢复点相关联的对应目录中。

www.TopSage.com

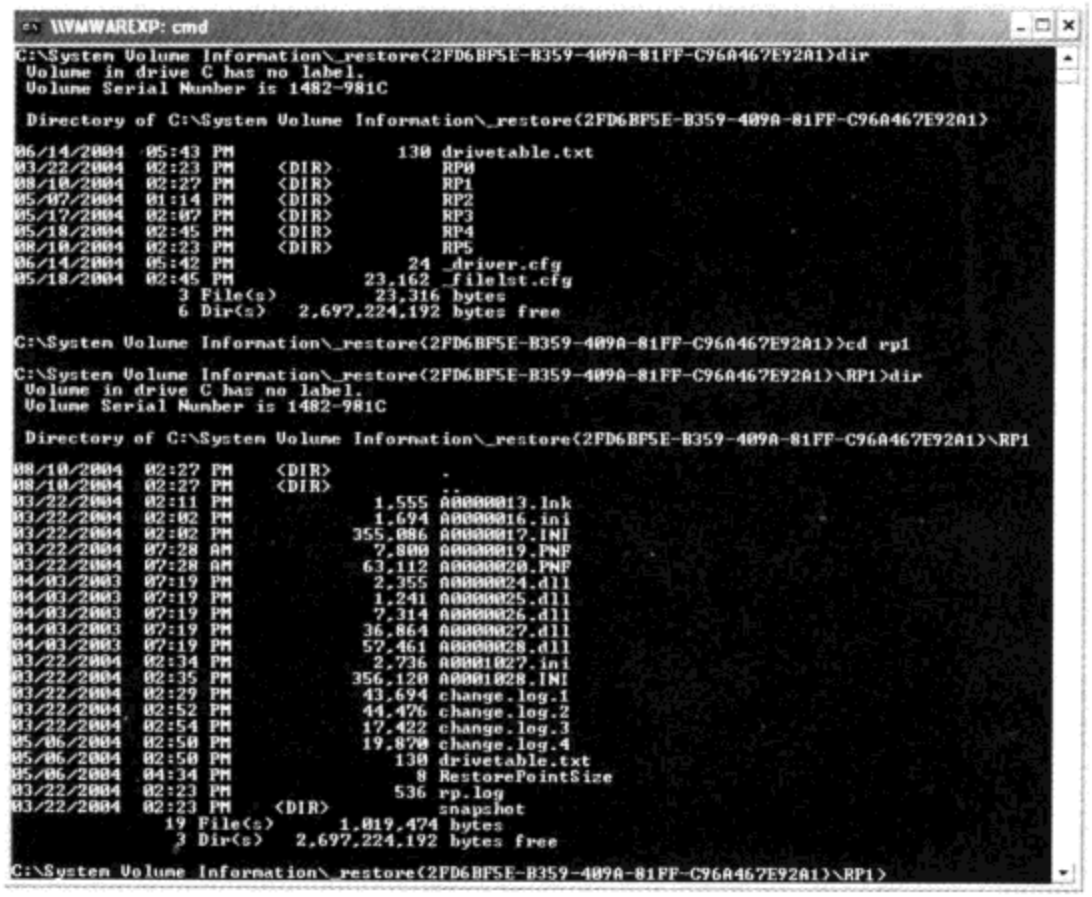


图12.11 系统恢复目录和恢复点的内容

引导卷上的恢复点目录也保存了一个名为_filelst.cfg的文件，它是一个二进制文件，其中包含了那些“在修改之后应该被存储到恢复点中的文件”的扩展名，以及那些“应该被忽略其变化的目录（比如专门存储临时文件的目录）”的一个列表。这一列表（在Platform SDK中有文档）指示系统恢复机制只记录下非数据文件。例如，你并不希望发生这样的情况：你仅仅为了改正一个应用程序配置问题而回滚了系统，却导致一个重要的Microsoft Word文档被删除了。



实验：查看系统恢复过滤型设备对象

为了监视文件和目录的变化，系统恢复过滤型驱动程序必须将过滤型设备对象附载到代表相应卷的FAT和NTFS设备对象上。而且，它还把一个过滤型设备对象附载到代表文件系统驱动程序的设备对象上，因而，当新的卷被文件系统挂载进来时它可以知道这些新的卷，然后它可以将过滤型设备对象附载到这些卷上。利用内核调试器，你可以看到系统恢复驱动程序的设备对象：

```
lkd> !drvobj \filesystem\sr
Driver object (81543850) is for:
 \FileSystem\sr
Driver Extension List: (id , addr)

Device Object list:
814ee370  81542dd0  81543728
```


www.TopSage.com

在以上的示例输出中，系统恢复驱动程序有三个设备对象。列表中的最后一个名为*SystemRestore*，所以，它被当作系统恢复机制的用户模式组件发送命令的接口：

```
lkd> !devobj 81543728
Device object (81543728) is for:
 SystemRestore \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e128feac DevExt 00000000 DevObjExt 815437e0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
Device queue is not busy.
```

第一和第二个设备对象被附载到NTFS文件系统设备对象上：

```
lkd> !devobj 814ee370
Device object (814ee370) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 814ee428 DevObjExt 814ee570
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedTo (Lower) 81532020 \FileSystem\Ntfs
Device queue is not busy.
lkd> !devobj 81542dd0
Device object (81542dd0) is for:
  \FileSystem\sr DriverObject 81543850
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 81542e88 DevObjExt 81542fd0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedTo (Lower) 815432e8 \FileSystem\Ntfs
Device queue is not busy.
```

其中一个NTFS设备对象是NTFS文件系统驱动程序的接口设备，因为它的名称是NTFS：

```
lkd> !devobj 815432e8
Device object (815432e8) is for:
 Ntfs \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 1 Type 00000008 Flags 00000040
Dacl e1297154 DevExt 00000000 DevObjExt 815433a0
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 81542dd0 \FileSystem\sr
Device queue is not busy.
```

另一个设备对象代表了已挂载在C:上的NTFS卷，这是系统的惟一卷，所以它没有名称：

```
lkd> !devobj 81532020
Device object (81532020) is for:
  \FileSystem\Ntfs DriverObject 81543410
Current Irp 00000000 RefCount 0 Type 00000008 Flags 00000000
DevExt 815320d8 DevObjExt 81532880
ExtensionFlags (0x80000000)  DOE_DESIGNATED_FDO
AttachedDevice (Upper) 814ee370 \FileSystem\sr
Device queue is not busy.
```


www.TopSage.com

当用户指示系统执行一次恢复时，系统恢复向导（System Restore Wizard，\Windows\System32\Restore\Rstrui.exe）在系统恢复机制的参数键下面创建一个名为*RestoreInProgress*的DWORD值，并且将它设置为1。然后，它调用Windows的*ExitWindowsEx*函数，以激发系统停机重引导动作。在重新引导起来以后，WinLogon进程（\Windows\System32\Winlogon.exe）意识到它应该执行一次恢复操作，于是，它把保存的文件从恢复点的目录拷贝到它们原来的位置上，并且利用日志文件来恢复文件系统对于文件和目录所做的修改。当这个过程完成时，系统继续引导。除了保证恢复过程更加安全以外，即使为了激活被恢复出来的注册表储巢，重新引导也是必要的。

Platform SDK包含了两个与系统恢复有关的API的文档，即*SRSetRestorePoint*和*SRRemoveRestorePoint*。安装程序可以使用这两个API，而且，开发人员应该检查其应用程序所使用的文件扩展名是否在系统恢复机制的保护范围内。存放用户数据的文件不应该使用系统恢复机制所保护的扩展名；否则的话，当系统回滚到一个恢复点的时候，用户可能会丢失数据。

12.3 诊断文件系统的问题

第4章讲述了“系统和应用程序将数据存储在注册表中”的这种做法，这使得与注册表有关的问题（比如错误配置的安全设置和遗漏的注册表值和键）成为许多系统失败和应用程序失败的根源。系统和应用程序也使用文件来存放数据，它们访问可执行映像文件和DLL映像文件。因此，错误配置的NTFS安全性和遗漏的文件和目录也是系统失败和应用程序失败的一个常见根源。这是因为，系统和应用程序通常假设它们能够访问哪些文件或目录，然后，当这些假设不成立的时候，它们就会以非预期的方式做出任何意外之事。

Filemon实时地显示了所有的文件活动，这使得它成为一个“诊断与文件系统相关的系统失败和应用程序失败”的理想工具。Filemon的用户界面几乎等同于Regmon的界面，并且Filemon包含了与Regmon同样的过滤、加亮显示和搜索功能。要想第一次在一个系统上运行Filemon，当前账户必须具有与运行Regmon同样的特权：加载驱动程序（Load Driver）和调试（Debug）。在加载进来以后，驱动程序仍然驻留在内存中，所以，以后再次执行时只要求调试（Debug）特权即可。

Filemon的基本和高级模式

当你运行Filemon时，Filemon在基本模式下启动起来，在此模式下，Filemon显示了对于诊断最为有用的文件系统活动。Filemon在基本模式下时，它的显示输出忽略了某些特定的文件系统操作，包括：

- 对NTFS元数据文件的访问；
- 发生在System进程中的活动；
- 页面文件的I/O；

www.TopSage.com

- 由Filemon进程生成的I/O；
- 快速I/O失败。

Filemon在基本模式下时，它也使用友好名称来报告文件I/O操作，而不是代表这些I/O操作的IRP类型。例如，IRP_MJ_WRITE和FASTIO_WRITE这两种操作都被显示成Write，而对于IRP_MJ_CREATE操作，如果它代表一个打开操作，则被显示为Open；如果它代表的是创建新的文件，则被显示成Create。



实验：在一个空闲系统上查看文件系统的活动

Windows文件系统驱动程序实现了对于**文件变化通知**（*file change notification*）的支持，这使得应用程序可以请求获得有关文件系统变化的通知，而无需主动地询问文件系统的变化。实现文件变化通知的Windows函数包括*ReadDirectoryChangesW*，以及*FindFirstChangeNotification*和 *FindNextChangeNofification*函数对。因此，当你在一个空闲的系统上运行Filemon时，你应该不会看到对于文件和目录的重复访问，因为这种行为会毫无必要地影响一个系统的总体性能。

运行Filemon，几秒钟以后，检查它的输出日志，看看你是否能够找到主动询问文件系统变化的行为。在与这样的行为相关联的输出行上右键点击，从环境菜单中选择Process Properties，以查看执行此种行为的进程的细节信息。

Filemon诊断技巧

如同Regmon的诊断技巧一样，Filemon也有两种基本的诊断技巧：在Filemon的痕迹数据中，检查一个应用程序在失败以前最后所做的事情；或者，将一个失败应用程序的Filemon痕迹数据跟一个正常工作的系统中的痕迹数据进行比较。有关这些技术的更多信息，请参见第4章中的“Regmon诊断技巧”一节。

在Filemon痕迹数据中，凡是在Result栏中具有FILE NOT FOUND、NO SUCH FILE、PATH NOT FOUND、SHARING VIOLATION和ACCESS DENIED值的项目都是你应该探查的对象。一个应用程序或者系统试图打开一个不存在的文件或目录时，会得到前面三种值。在许多情况下，这些错误并不代表一个严重问题。例如，当你从Start菜单的Run对话框中执行一个程序，但又没有指定它的全路径时，Explorer将会在系统的PATH环境变量所列出的目录中，搜索该映像文件，直至它找到此文件，或者搜遍所有列出的目录为止。每次试图在一个不包含此文件的目录中找到它，都会导致在Filemon中输出类似如下的一行：

```
5:28:26 PMEXPLORER.EXE:1568FASTIO_QUERY_OPENC:\Documents and Settings\mark.AUSTIN\Start
Menu\test.exe    FILE NOT FOUND    Attributes: Error
```

访问拒绝（access-denied）错误是许多与文件系统有关的应用程序失败的常见根源，当一

www.TopSage.com

个应用程序没有许可权限来按照它所期望的访问模式打开文件或目录时，这种错误就会发生。有些应用程序并不检查错误代码或者执行错误恢复，它们的失败方式是直接崩溃或进程终止；其他有的应用程序显示一些误导性的错误消息，这些消息掩盖了真正的错误根源。

缓存区溢出漏洞一旦被挖掘出来，就是一个严重的安全问题，但是，结果代码BUFFER OVERFLOW只不过是“一个文件系统驱动程序向应用程序指明，它所指定的用于存储结果数据的缓冲区太小了，从而无法容纳要读取的数据”的一种方法而已。应用程序开发人员利用这种行为来确定一个缓冲区应该要多大，因为文件系统驱动程序也会返回“为了存放此数据而要求的缓冲区的大小”。在一个带有缓冲区溢出结果代码的操作之后，通常跟着一个同样的操作，但后者带有成功的结果代码。

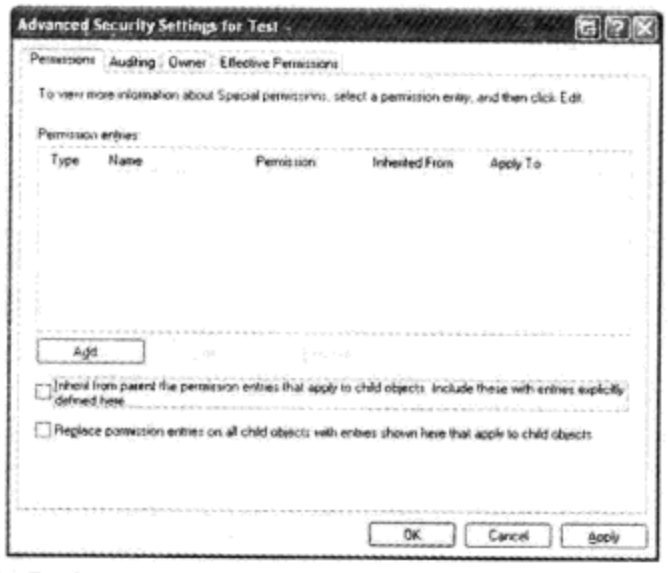


实验：用Filemon来检查一个错误的根源

应用程序有时候在响应一个错误条件时，会给出一些并不揭示该错误根源的错误消息。这些错误消息可能反而会起到妨碍作用，因为它们可能导致你花许多时间来诊断或解决一些并不存在的问题。如果该错误消息涉及一个文件系统问题，则Filemon将可以在错误消息出现以前，先向你显示出底层可能发生的错误。

在这个例子中，你将设置一个目录的许可权限，然后在Notepad中执行一个文件保存操作，该操作会导致一个误导性的错误消息。Filemon的痕迹数据显示了实际的错误，以及在Notepad错误对话框中所显示的消息的来源。

1. 运行Filemon，设置Include过滤器为“notepad.exe”。
2. 打开Explorer，在某个NTFS卷的一个目录中创建一个名为“Test”的目录（在这个例子中，用到了根目录）。
3. 编辑Test目录的安全许可属性，去掉所有的访问权限。这可能要求你打开Advanced Security Settings对话框，并使用Permissions标签页上的设置来去掉继承的安全性。



www.TopSage.com

当你应用（Apply）已修改的安全性时，Explorer应该会警告你，这将导致无人可以访问该文件夹了。

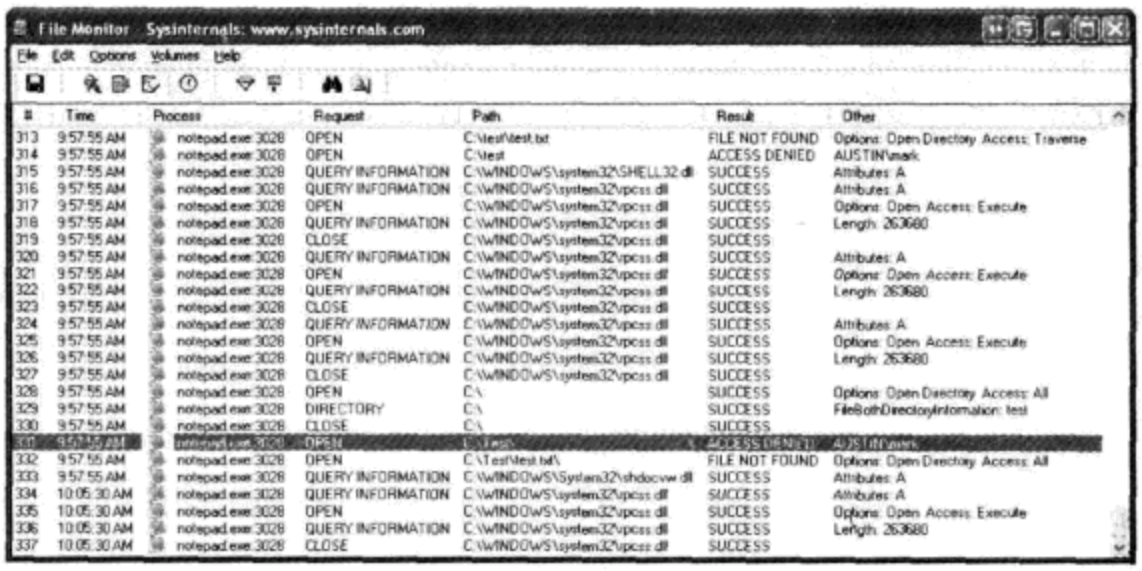
4. 运行Notepad，在它的窗口中输入一些文本。然后从File菜单中选择Save菜单项。在Save对话框的File Name域中，输入**c:\test\test.txt**（假定你刚才创建的文件夹位于C:卷上）。

5. Notepad将显示如下的错误消息：



6. 该消息意味着C:\Test不存在。

7. 你看到的Filemon痕迹数据应该类似于如下：



该图中底部的输出是在出现错误消息之前那时刻产生的。第331行显示了：Notepad试图打开C:\Test，但得到了一个访问被拒绝（access-denied）的错误。在之后的第332行，它试图打开C:\Test\Test.txt目录（如该行的Other栏中的Options所指示），但接收到一个“文件未找到（file-not-found）”的错误，因为该目录不存在。Notepad所显示的错误消息“Path does not exist”与“文件未找到（file-not-found）”错误是一致的，但跟“访问被拒绝（access-denied）”错误不一致。所以，看起来是这样的：Notepad首先试图打开该目录（C:\Test），在该操作失败以后，出于某种原因，它假定“C:\Test\Test.txt”是一个目录的名称，而不是一个文件的名称。当Notepad无法打开此目录的时候，它出示了前面的错误消息，但这里的根源是“访问被拒绝”错误，Filemon揭示了此错误根源。

Filemon已经被广泛地应用于Microsoft和其他组织的内部，以解决一些困难的或者几乎不可能诊断的问题。用Filemon来诊断问题的一个例子是，用它来揭示出Windows安装器（Windows Installer）服务所产生的误导性错误消息的根源。在这个例子中，当用户试图使用Windows安

www.TopSage.com

装器软件包（Windows Installer Package）文件来安装一个应用程序时，Windows安装器服务报告出如图12.12所示的错误，这表明该服务不能写Temp文件夹。而用户经过验证却发现，与该错误消息所说明的相反，他的轮廓中的临时目录（在控制台窗口中通过输入“**set temp**”可以获得）位于一个有足够空闲空间的卷上，而且具有默认的访问权限。

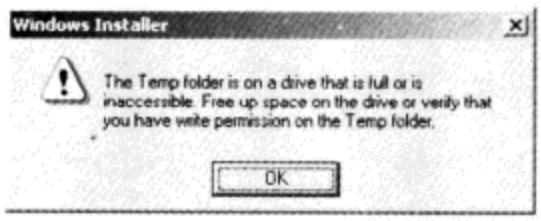


图12.12 Microsoft Windows安装器的错误消息

用户运行Filemon来捕获到导致发生这一错误的文件系统活动的痕迹数据，并且找到了该错误的根源，如图12.13中被加亮显示的这一行所示。用户从该痕迹数据中得出的结论是：Windows安装器服务正在引用\Windows\Installer，而不是引用其轮廓中的临时目录（如错误消息中所指的Temp文件夹）。此输出中的“访问被拒绝（ACCESS DENIED）”这一行报告了：Windows安装器服务是在本地系统账户下运行的。因此，用户修改\Windows\Installer上的许可权限，以允许本地系统账户具有写访问权，于是解决了该问题。



图12.13 Microsoft Windows安装器服务的Filemon痕迹数据

另一个用Filemon来诊断问题的例子跟Microsoft Word有关。在这个例子中，用户可以启动Word，但只有几秒钟时间来输入数据，然后Word窗口在毫无通知的情况下被关闭。图12.14显示了此情形下的Filemon痕迹数据的一部分。这表明了Word在退出以前那时刻，反复地从一个名为Mssp3es.lex的文件中读取同一部分数据（当一个进程退出时，系统关闭其所有的句柄，在这个例子中，你可以从25 460行中看到这一过程正在发生）。用户搞清楚了.lex文件与Microsoft Office Proofing Tools有关，于是他重新安装了该组件，解决了问题。



图12.14 Microsoft Word正在读一个.lex文件

www.TopSage.com

在另一个例子中，用户碰到的问题是：每次当他启动Microsoft Excel时，就会出现如图12.15所示的错误消息。用户在Excel的启动过程中捕获到如图12.16所示的Filemon痕迹，它反映了Excel正在从Microsoft Office的安装目录下的Xlstart目录中，从一个名为59403e20的文件中读取数据。用户检查了此问题，从Excel的文档中得知，Excel总是自动地试图打开Xlstart目录中的文件。然而，在痕迹数据中看到的文件并不是一个Excel文件，因而导致产生这一错误消息。他删除了此文件，于是该错误不再出现了。

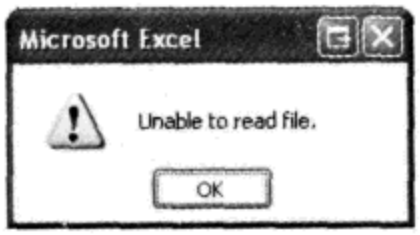


图12.15 Microsoft Excel启动时的错误消息

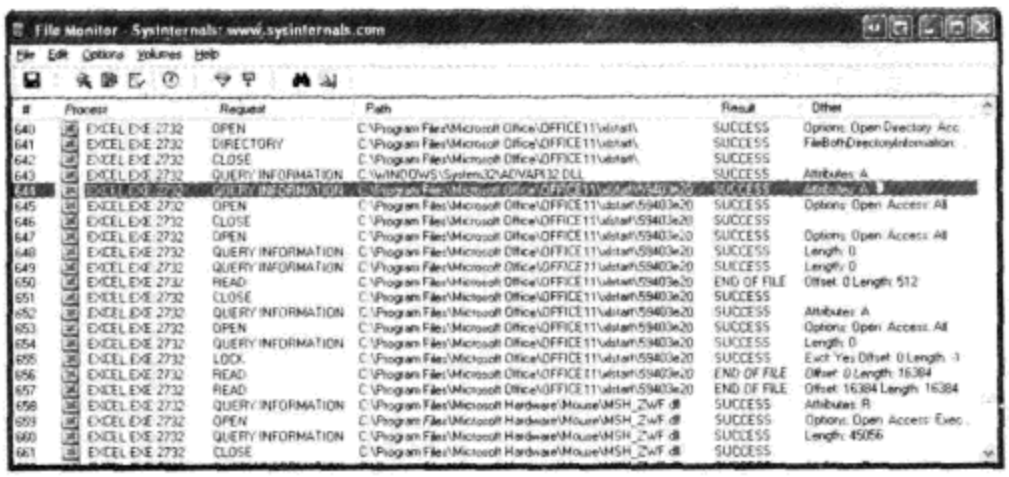


图12.16 Microsoft Excel启动时的Filemon痕迹数据

用Filemon来诊断文件的最后一个例子涉及过时的DLL。一个特殊的用户在运行了Microsoft Access 2000以后，当他试图导入一个Microsoft Excel文件时Access 2000停止运行（hang）了。用户试着在另一个也安装了Microsoft Access 2000的系统上导入同样的文件，则可以成功地完成导入过程。用户在这两个系统上捕获到了导入操作的痕迹数据，并将痕迹数据保存到日志文件中，之后，他使用Windiff来比较这两份日志。比较的结果如图12.17所示。

在去除了一些不重要的差别（比如19行中看到的临时文件名的不同，以及26行中文件名大小写的不同）以后，用户发现，两份痕迹数据之间的第一个实质性的差别是37行。在导入失败的系统上，Microsoft Access从\Winnt\System32目录中加载Accwiz.dll，而在导入成功的系统上，它从\Progra~1\Files\Microsoft\Office中读取Accwiz.dll。用户检查了\Winnt\System32中的Accwiz.dll拷贝，发现它来自于Microsoft Access的一个老版本，但系统的DLL搜索顺序导致了系统找到的是该实例，而不是Microsoft Access安装目录中的实例。删除该拷贝并注册了正确的拷贝以后，即修正了问题。

www.TopSage.com

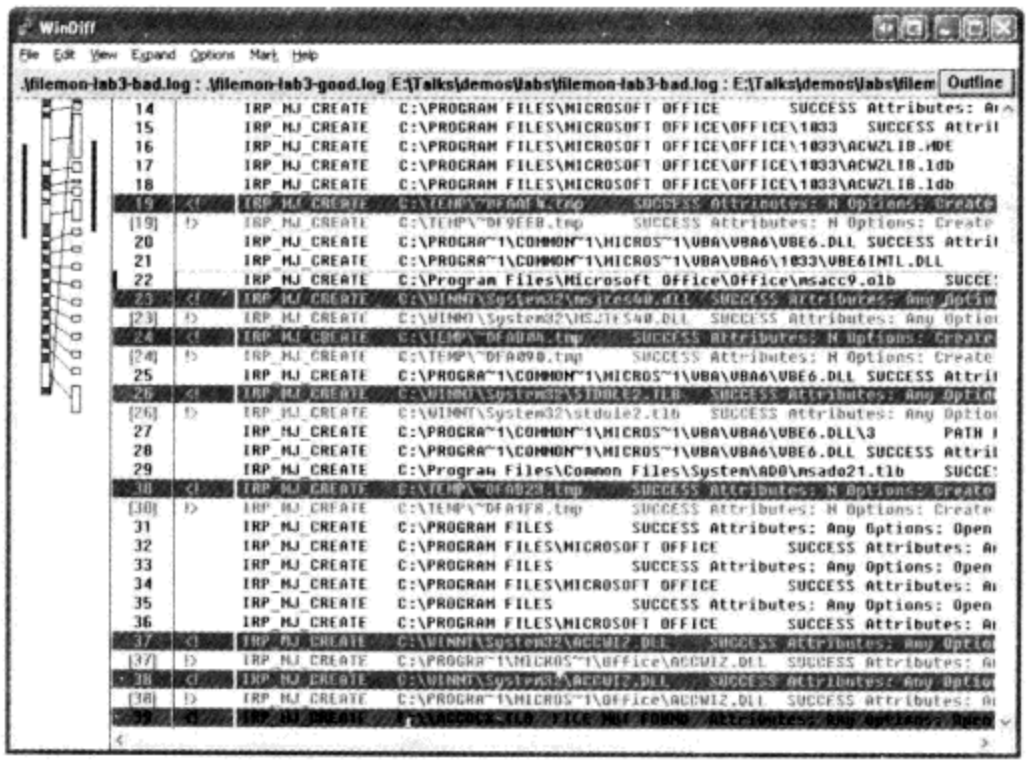


图12.17 Microsoft Access痕迹日志的比较

以上只是列举少量的例子来说明：可通过Filemon在应用程序未能清楚地报告文件系统问题的情况下，发现这些文件系统问题的根源。本章剩余的部分将集中介绍Windows的原生文件系统：NTFS。

12.4 NTFS设计目标和特性

在下一小节中，我们将介绍那些驱动了NTFS设计的需求。然后，在随后的小节中，我们将讨论NTFS的高级特性。

高端（High-End）文件系统的需求

从一开始，NTFS的设计目标就包含了作为一个企业级文件系统所必须具备的各种特性。为了在面临意外的系统停电或崩溃时数据的损失尽可能地最少，一个文件系统必须确保在任何时候都能保证其元数据的完整性。而且，为了保护敏感的数据避免遭受未授权的访问，文件系统必须有一个综合的安全模型。最后，一个文件系统必须允许使用“基于软件的数据冗余机制”作为对于“用硬件冗余来保护用户数据”的一种廉价替代。在这一节中，你将会明白，NTFS如何实现这些能力。

可恢复性

为了解决“可靠的数据存储和数据访问”的需求，NTFS在“**原子性事务**（*atomic transaction*）”概念的基础上提供了文件系统恢复功能。原子性事务是一种用来处理数据库修改的技术，其目

www.TopSage.com

标是，系统的失败不会影响到数据库的正确性或完整性。原子性事务的基本原则是，有些称为**事务**（*transaction*）的数据库操作，是“要么全部完成，要么全部不做（all-or-nothing）”的任务（**事务**被定义为一个“改变了文件系统数据或者改变了卷目录结构”的I/O操作）。 构成一个事务的那些独立磁盘更新操作必须**按原子方式**来执行，也就是说，一旦事务开始执行，它的所有磁盘操作就必须被完成。如果一次系统失败中断了该事务，那么，那些已经被完成的部分必须被撤销（undo），或者**回滚**（*roll back*）。回滚操作使得数据库又回到原先已知的、一致的状态，就好像该事务从未发生过。

NTFS使用原子性事务来实现它的文件系统恢复特性。如果一个程序发出了一个I/O操作，并且该操作修改了一个NTFS卷的结构，比如说，改变了目录结构、扩展了一个文件、为一个新文件分配了空间，等等，那么，NTFS将该操作作为一个原子性事务来对待。它保证，要么该事务得以完成，要么，如果在执行该事务时系统失败，则该事务被回滚。有关NTFS如何做到这一点的细节信息，请参见本章后面“NTFS的恢复支持”一节中的介绍。

而且，对于关键的文件系统信息，NTFS使用了额外的存储区，所以，如果磁盘上的一个扇区坏了，NTFS仍然可以访问该卷的关键文件系统数据。文件系统数据的这种冗余与FAT文件系统和HPFS文件系统（OS/2的原生文件系统格式）在磁盘上的结构（on-disk structure）形成了鲜明的对比，这些文件系统只有一份扇区包含了关键的文件系统数据。在这些文件系统上，如果这些扇区中的一个发生了读错误，则整个卷丢失了。

安全性

NTFS中的安全性直接来源于Windows的对象模型。文件和目录都是被保护的对象，可以避免遭受未授权用户的访问（有关Windows安全性的更多信息，请参见第8章）。 一个打开的文件被实现为一个文件对象，其安全描述符被作为该文件的一部分存储在磁盘上。在一个进程能够打开一个指向任何对象（包括文件对象）的句柄以前，Windows安全系统会检验该进程是否有恰当的权限可以这么做。安全描述符以及“用户登录到系统上并提供一个标识其身份的口令”这一需求，两者结合起来可以确保：除非系统管理员或者文件的所有者给予特定的许可权来访问一个文件，否则任何进程都无法访问该文件（有关安全描述符的更多信息，请参见第8章中的“安全描述符和访问控制”一节；有关文件对象的更多细节信息，请参见第9章中的“文件对象”一节）。

数据冗余和容错能力

除了文件系统数据的恢复能力以外，有些用户要求他们自己的数据不能因为电源断电或者灾难性的磁盘失败而遭受损害。NTFS的恢复能力确实保证了一个卷上的文件系统仍然是可以访问的，但是，没有为用户文件的完全恢复提供保证。对于那些不能承受文件数据丢失的应用程序，其保护是通过数据冗余机制来提供的。

www.TopSage.com

用户文件的数据冗余机制是通过Windows的分层驱动程序模型（在第9章中介绍）来实现的，该模型提供了容错磁盘的支持。为了把数据写到磁盘上，NTFS与一个卷管理器进行通信，而卷管理器又跟一个硬盘驱动程序进行通信。卷管理器可以将数据从一个硬盘复制或**镜像**（*mirror*）到另一个磁盘上，因而总是存在冗余的拷贝。这种支持一般称为*RAID 1*级（*RAID level 1*）。卷管理器也允许数据被写到跨越三个或四个磁盘的**条带**（*stripe*）中，并且利用等价于一个磁盘的开销来维护奇偶校验信息。如果一个磁盘上的数据丢失或不可访问了，则驱动程序可以通过异或操作的办法来重构该磁盘的内容。这种支持称为*RAID 5*级（*RAID level 5*）。（有关条带卷、镜像卷和RAID-5卷的更多信息，请参见第10章。）

NTFS的高级特性

NTFS除了针对关键的业务系统具有可恢复性、安全性、可靠性和高效性以外，它还包含下面的高级特性，这些高级特性使得它可以支持范围广泛的各种应用。其中有些特性通过API的形式被导出来，以供应用程序使用，而其他一些则是内部特性：

- 多数据流；
- 基于Unicode的名称；
- 通用的索引设施；
- 动态的坏簇重新映射；
- 硬链接和交接（junction）；
- 压缩文件和稀疏文件；
- 变化日志；
- 针对每个用户的卷配额；
- 链接跟踪；
- 加密；
- POSIX支持；
- 碎片整理；
- 只读支持。

下面的小节对这些特性作一般性的介绍。

多数据流

在NTFS中，与一个文件相关联的每一个信息，包括其名称、所有者、时间戳、内容，等等，都被实现为一个文件属性（NTFS对象属性，NTFS object attribute）。每个属性是由一个流（stream）构成的，也就是说，由一个简单的字节序列构成。这种一般化的实现方式使得很容易为一个文件加入更多的属性（因而也加入了更多的流）。因为一个文件的数据“只不过是该文件的一个属性”而已，而且，新的属性还能够被加入进来，所以，NTFS文件（和文件目录）可以包含多个数据流。

www.TopSage.com

NTFS文件有一个默认的数据流，此默认的数据流没有名称。应用程序可以创建附加的、有名称的数据流，并且可以通过其名称来访问这些数据流。为了避免改变原来的Microsoft Windows I/O API函数（它们只带一个字符串作为文件名参数），数据流的名称是通过在文件名的后面再加上一个冒号“:”来指定的。因为冒号是一个保留字符，所以，它可以用做文件名称和数据流名称之间的一个分隔符，如下面的例子所示：

```
myfile.dat:stream2
```

每个流有一个单独的分配大小（该值定义了有多少磁盘空间为它而保留）、实际大小（定义了调用者已经使用了多少字节），以及有效数据的长度（该流有多少已经被初始化了）。而且，每个流也被赋予一个单独的文件锁，利用该锁可以锁住字节范围，以及允许并发访问。

在Windows中，一个使用多数据流的组件是，Windows Server随带的Apple Macintosh文件服务器支持。Macintosh系统为每个文件使用两个流：一个用于存储数据；另一个用于存储资源信息，比如文件类型和代表该文件的图标。因为NTFS允许多数据流，所以，Macintosh用户可以将一个完整的Macintosh文件夹拷贝到Windows服务器上，而另一个Macintosh用户则可以从该服务器上拷贝此文件夹，并且此过程不会丢失资源信息。

Windows Explorer是另一个使用多数据流的应用程序。当你在一个NTFS文件上右键点击并选择Properties（属性）时，结果对话框中的Summary（摘要）标签页允许你将一些信息（比如标题、主题、作者和关键字）与该文件关联起来。 Windows Explorer把这些信息保存在该文件的另一个流中，名为Summary Information。

其他的应用程序也可以使用多数据流特性。例如，一个备份工具可以使用额外的数据流来保存有关被备份文件的、与备份相关的时间戳信息。或者，一个存档工具可以实现多级存储结构，其中，在某个特定日期以前的文件，或者一段时间以来尚未被访问过的文件都被转移到磁带上。该工具可以把这样的文件拷贝到磁带上，并且将文件的默认数据流设置为0，再加入一个数据流来指定该文件存储在磁带上的名称和位置。



实验：查看数据流

大多数Windows应用程序都没有被设计为可支持命名的数据流，但*echo*和*more*命令却支持命名数据流。因此，一种查看多数据流机制的简单办法是，使用*echo*创建一个命名的流，然后使用*more*来显示该流。下面的命令序列创建一个名为test的文件，该文件中有一个名为stream的流：

```
C:\>echo hello > test:stream
C:\>more < test:stream
hello
C:\>
```


www.TopSage.com

如果你执行一次列目录的命令，test的文件大小并不反映出额外数据流中存储的数据，因为NTFS对于文件查询操作，只返回未命名数据流的大小，列目录命令也不例外。

```
C:\>dir test
 Volume in drive C is WINDOWS
 Volume Serial Number is 3991-3040

 Directory of C:\

08/01/00  02:37p                 0 test
               1 File(s)              0 bytes
                          112,558,080 bytes free
```

你可以利用*www.sysinternals.com*的streams工具来确定，在你的系统上哪些文件和目录具有额外的数据流，如下所示：

```
C:\>streams test

Streams v1.5 - Enumerate alternate NTFS data streams
Copyright (C) 1999-2003 Mark Russinovich
Sysinternals - www.sysinternals.com
 C:\Temp\test:
          :stream:$DATA 8
```

基于Unicode的名称

如同Windows本身一样，NTFS完全支持Unicode，它使用Unicode字符来存储文件、目录和卷的名称。Unicode是一种16位的字符编码方案，它允许世界上每一种主要语言的每一个字符都可以被惟一表示，从而有助于很容易地将数据从一个国家转移到另一个国家。Unicode是传统的国际字符表示法的一个改进（传统的国际字符表示法使用双字节编码方案，有些字符存储在8位中，而其他的字符存储在16位中，该技术要求加载各种代码页以便建立起可用的字符集）。因为Unicode针对每个字符有一个惟一的表示，所以，它不依赖于哪个代码页已被加载。路径中的每个目录和文件名可以长达255个字符，可以包含Unicode字符、内嵌的空格和多个句点号。

通用的索引设施

NTFS的总体结构是高度结构化的，从而允许对一个磁盘卷上的文件属性进行索引。这一特性使得文件系统可以高效地定位到符合某种准则的文件，例如，某个特定目录中的所有文件。FAT文件系统对文件名编了索引，但是并不对它们进行排序，因而使得在大的目录中查找文件非常慢。

NTFS的几个特性用到了通用的索引机制，包括统一的安全描述符。在一个卷上，文件和目录的安全描述符被保存在单个内部流中，因而可以去除重复的安全描述符，并且它使用NTFS定义的内部安全标识符进行索引。在本章后面的“NTFS在磁盘上的结构”一节中描述了这些特性使用索引机制的做法。

www.TopSage.com

动态的坏簇重新映射

一般地，如果一个程序试图从一个坏磁盘扇区中读取数据，则读操作会失败，这一已分配的簇中的数据变得不可访问。然而，如果该磁盘被格式化成一个容错的NTFS卷，则Windows的容错驱动程序将会动态地获得一份原本存储在此坏扇区中的数据的好的拷贝，然后发送一个警告，告诉NTFS该扇区已经坏了。于是NTFS分配一个新的簇，替换掉坏扇区所在的簇，然后把数据拷贝到新的簇中。它对坏的簇做上标记，以后不再使用该簇。这种数据恢复和动态的坏簇重新映射的机制，对于文件服务器和容错系统，或者对于不能承受数据丢失的任何应用程序来说，都是一种特别有用的特性。如果当一个扇区变坏时，该卷管理器尚未被加载，则NTFS仍然替换掉坏簇，不再使用该簇，但它不能恢复坏簇上原来的数据。

硬链接（link）和交接（Junction）

硬链接使得多个路径指向同一个文件（在目录上，不支持硬链接）。如果你创建了一个名为C:\Users\Documents\Spec.doc的硬链接，它指向一个已有的C:\My Documents\Spec.doc，则这两个路径链接到同一个磁盘文件，你可以使用任何一个路径来改变此文件。进程可以利用Windows的*CreateHardLink*函数或者POSIX *ln*函数来创建硬链接。

除了硬链接以外，NTFS还支持另一种称为**交接**（*junction*）的重定向机制。交接也称为符号链接（symbolic link），它使得一个目录可以将文件和目录路径名的翻译重定向到另一个目录上。例如，路径C:\Drivers是一个重定向到C:\Windows\System32\Drivers上的交接，若应用程序读取C:\Drivers\Ntfs.sys文件，那么，它实际上读取的文件是C:\Windows\System\Drivers\Ntfs.sys。交接是一种很有用的方法，它可以把目录树深处的目录提升到一个更加方便的深度上，同时又不会破坏原来的目录树的结构或内容。在刚刚引用的例子中，drivers目录被提升到该卷的根目录下，当通过交接来访问Ntfs.sys文件时，它的目录深度从3级降低到1级。你不能使用交接机制来链接到远程的目录，你只能链接到本地卷的目录中。

交接机制建立在一种称为**重解析点**（*reparse point*）的NTFS机制的基础之上。（关于重解析点，在本章后面的“重解析点”一节中有进一步的讨论。） 重解析点是一个文件或目录，它有一个与之相关联的称为“**重解析数据**（*reparse data*）”的数据块。重解析数据是由用户定义的，它包含了有关文件或目录的数据，比如其状态或位置。创建重解析数据的应用程序后，文件系统过滤型驱动程序或者I/O管理器，都可以从解析点读取重解析数据。当NTFS在一个文件或目录的查找过程中碰到一个重解析点时，它返回一个**重解析状态代码**（*reparse status code*），此状态代码告知附载在该卷上的文件系统过滤型驱动程序和I/O管理器，应该检查一下重解析数据。每一种重解析点类型都有一个惟一的**重解析标记**（*reparse tag*）。此重解析标记使得负

www.TopSage.com

责翻译重解析数据的组件，可以在不必检查重解析数据的情况下就能够识别出重解析点。重解析标记的所有者，无论是一个文件系统过滤型驱动程序，还是I/O管理器，在识别重解析数据时，可以选择以下的选项之一。

- 重解析标记的所有者可以操纵在文件I/O操作（若这些I/O操作要跨越该重解析点）中指定的路径名，并且让该I/O操作重新用修改后的路径名再发出一遍。例如，交接（junction）利用这种方法来重定向一个目录查询操作；
- 重解析标记的所有者可以从文件中去掉重解析点，以某种方式改变该文件，然后重新发出该文件I/O操作。Windows的多级存储管理（HSM，Hierarchical Storage Management）系统按照这种方式来使用重解析点。HSM存档文件的做法是，将文件的内容转移到磁带上，并且在其中留下对应的重解析点。当一个进程访问一个已经被存档的文件时，HSM过滤型驱动程序（\Windows\System32\Drivers\Rsfilter.sys）从文件中去掉重解析点，并且从存档介质中读入该文件的数据，然后重新发出此访问操作。因此，此获取离线数据的过程，对于访问这一已存档文件的进程来说是透明的。

没有专门用于创建重解析点的Windows函数。相反地，进程必须在Windows的*DeviceIoControl*函数中使用FSCTL_SET_REPARSE_POINT文件系统控制代码。进程可以使用FSCTL_GET_REPARSE_POINT文件系统控制代码来查询一个重解析点的内容。在重解析点的文件属性中，FILE_ATTRIBUTE_REPARSE_POINT标志被设置上了，所以，应用程序可以利用Windows的*GetFileAttributes*函数来检查重解析点。



实验：创建一个交接（Junction）

Windows没有包含任何可用来创建交接的工具，但是，你可以利用*www.sysinternals.com*的Junction工具（网站上包含了其源代码），或者Windows资源工具箱中的工具Linkd来创建一个交接。Linkd工具也让你查看已有的交接的定义，而Junction工具让你查看有关交接和其他重解析点标记的信息。

压缩文件和稀疏文件

NTFS支持文件数据的压缩功能。因为NTFS透明地执行压缩和解压缩过程，所以，应用程序无需任何修改，就可以利用这一特性。目录也可以被压缩，这意味着，该目录中随后创建的任何文件都是经过压缩的。

应用程序压缩和解压缩文件的做法是，将FSCTL_SET_COMPRESSION文件系统控制代码传递给*DeviceIoControl*。它们可以通过FSCTL_GET_COMPRESSION文件系统控制代码来查询一个文件或目录的压缩状态。如果一个文件或目录是压缩的，则它的属性中设置了FILE_ATTRIBUTE_COMPRESSED标志，所以，应用程序可以通过*GetFileAttributes*来确定一个文件或目录的压缩状态。

www.TopSage.com

第二种压缩类型称为**稀疏文件**（*sparse file*）。如果一个文件被标记为稀疏的，则NTFS并不为“该文件中被应用程序指定为空的部分”分配磁盘卷空间。当一个应用程序从一个稀疏文件的空区域中读取数据的时候，NTFS返回以0填充的缓冲区。对于实现“循环缓冲区方式的日志（circular-buffer logging）”的客户/服务器应用程序来说，这种压缩类型可能会非常有用。在这种应用程序中，服务器将信息记录到一个文件中，而客户异步地从文件中读取信息。因为服务器写入的信息在客户读过以后，就不再需要了，所以，不必将这些信息继续保存在文件中。通过将这样的文件做成一个稀疏文件，客户可以指定它读过的文件部分变成空，从而释放相应的卷空间。服务器可以继续在文件后面追加新的信息，而无需担心该文件会一直增长并消耗掉该卷上所有的可用空间。

如同压缩文件一样，NTFS透明地管理稀疏文件。应用程序只需将FSCTL_SET_SPARSE文件系统控制代码传递给*DeviceIoControl*，就可以指定一个文件的稀疏状态。要想设置文件中的一段范围为空的，应用程序可以使用FSCTL_SET_ZERO_DATA代码；应用程序还可以利用FSCTL_QUERY_ALLOCATED_RANGES控制代码来询问NTFS，以获得一份关于“文件中哪些部分是稀疏的”描述。稀疏文件的一个应用是接下来将要介绍的NTFS**变化日志**（NTFS *change journal*）。

变化日志

许多类型的应用程序需要监视一个卷中的文件和目录的变化。例如，一个自动备份程序可能先执行一个初始的完全备份，然后根据文件的变化来执行增量备份。为了允许一个应用程序监视一个卷的变化，一种显然的做法是让它扫描该卷，记录下文件和目录的状态，然后在后续的扫描中检测两次扫描的差异。然而，这一过程可能会不可避免地影响系统的性能，特别是在具有成千上万个文件的计算机上。

另一种做法是，让应用程序使用Windows的*FindFirstChangeNotification*或*ReadDirectoryChangesW*函数，注册一个目录通知。应用程序指定它想要监视的目录的名称作为一个输入参数，一旦该目录的内容发生了变化，函数便返回。虽然这种做法比扫描卷的做法更加高效，但是，它要求应用程序始终在运行。使用这些函数可能也要求一个应用程序扫描该目录，因为*FindFirstChangeNotification*并不指明什么被改变了——它只是说明，该目录中的某些东西被改变了。应用程序可以传递一个缓冲区给*ReadDirectoryChangesW*，FSD会把变化纪录填充到该缓冲区中。然而，如果该缓冲区溢出了，则应用程序必须准备好退回到扫描该目录的做法。

NTFS提供了第三种做法，它可以克服前两种做法的缺点。这种做法是：应用程序可以使用*DeviceIoControl*函数的FSCTL_CREATE_USN_JOURNAL文件系统控制代码，来配置NTFS的变化日志设施，让NTFS把有关文件和目录变化的信息记录到一个称为**变化日志**（*change journal*）的内部文件中；变化日志文件通常要足够大，以便保证应用程序有机会毫无遗漏地处理这些变化。应用程序使用FSCTL_QUERY_USN_JOURNAL文件系统控制代码，从一个变化日志中读取纪录。它们也可以指定：直到有了新的纪录以后*DeviceIoControl*函数才完成。

www.TopSage.com

针对每个用户的卷配额

系统管理员通常需要跟踪或限制用户在共享的存储卷上的磁盘空间使用量，所以，NTFS包含了配额管理支持。NTFS的配额管理支持允许为每个用户指定相应的配额限制，这对于“跟踪使用量”和“跟踪用户何时到达警告和限制阈值”是极为有用的。NTFS也可以被配置成：如果一个用户超过了他的警告限制值，则在系统的事件日志（Event Log）中记录一个事件来指明这一情况。类似地，如果一个用户试图使用超过其配额限制所允许的卷存储空间，则NTFS可以在系统的事件日志中记录一个事件，并且使引发此配额违例的应用程序文件I/O操作失败，其错误代码为“磁盘已满”。

NTFS基于以下事实来跟踪一个用户的卷使用量：它利用文件和目录的创建者用户的安全ID（SID）来标记这些文件和目录（关于SID的定义，请参见第8章）。 一个用户拥有的文件和目录的逻辑大小被计算在管理员为该用户定义的配额限制中。因此，用户不可能通过创建一个空的、超过其配额的稀疏文件来骗过NTFS的配额限制，然后再用非零的数据来填充该文件。类似地，虽然一个50KB的文件可以被压缩至10KB，但整个50KB都被用于配额测算。

在默认情况下，卷没有打开配额跟踪功能。你需要使用一个卷的属性对话框的“Quota”标签页（见图12.18）来打开配额功能，并指定默认的警告和限制阈值，以及配置“当一个用户达到了警告和限制阈值时”NTFS的行为。你可以从该对话框中启动Quota Entries工具，管理员通过该工具可以为每个用户指定不同的限制值和行为。如果应用程序想要与NTFS的配额管理组件打交道，它可以使用COM配额接口，包括*IDiskQuotaControl*、*IDiskQuotaUser*和*IDiskQuotaEvents*。

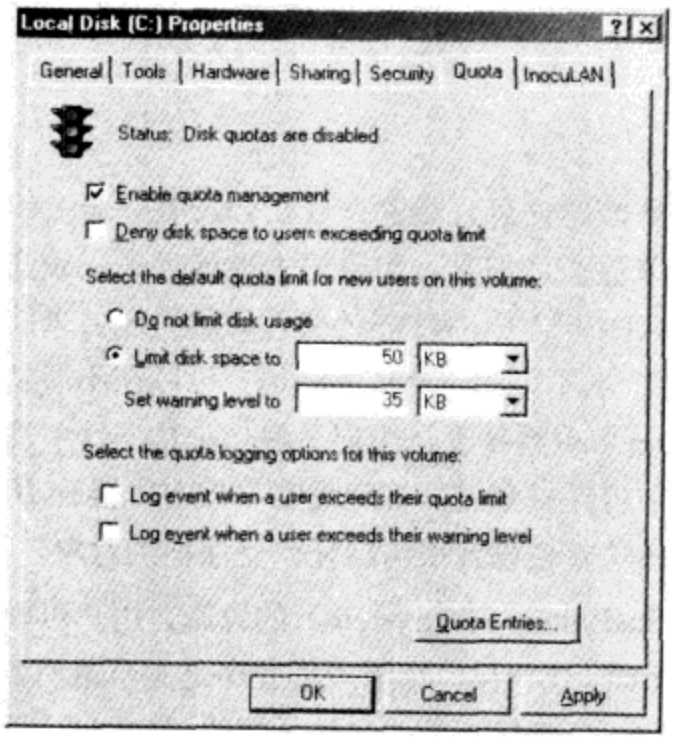


图12.18 卷的属性对话框

www.TopSage.com

链接跟踪

外壳（shell）快捷方式允许用户把一些文件放在他们的外壳名字空间中（例如，在它们的桌面上），这些文件实际上又链接到文件系统名字空间中的其他文件上。Windows Start菜单大量地使用了外壳快捷方式。类似地，对象链接和嵌入（OLE，object linking and embedding）中的链接使得一个应用中的文档可以被透明地嵌入到其他应用的文档中。Microsoft Office套件中的产品，包括PowerPoint、Excel和Word，都使用了OLE链接机制。

虽然外壳和OLE链接提供了一种便捷的方式，利用这种方式可以将一个文件跟另一个文件连接起来，或者将文件跟外壳名字空间连接起来。但是，在过去，它们很难管理。在Windows NT 4、Windows 95或者Windows 98中，如果一个用户移动了一个外壳或OLE链接的源（链接源是指一个链接所指向的文件或目录），那么，该链接将被打破，系统不得不依赖于启发式的过程来试图找到该链接的源。Windows的NTFS包含了一个称为分布式链接-跟踪（distributed link-tracking）的服务应用程序的支持，当外壳和OLE链接的目标移动时，它会维护这些链接的完整性。利用NTFS的链接-跟踪支持，如果位于一个NTFS卷上的链接源被移动到原始卷所在域内的任何另一个NTFS卷中，则链接-跟踪服务可以透明地跟随此移动过程，并更新该链接以反映出相应的变化。

NTFS链接-跟踪支持是建立在一个称为**对象ID**（*object ID*）的可选文件属性的基础之上的。应用程序可以利用FSCTL_CREATE_OR_GET_OBJECT_ID（如果一个文件尚未被分配对象ID，则分配一个对象ID）和FSCTL_SET_OBJECT_ID文件系统控制代码，为一个文件分配一个对象ID。通过FSCTL_CREATE_OR_GET_OBJECT_ID和FSCTL_GET_OBJECT_ID文件系统控制代码，应用程序可以查询对象ID。而FSCTL_DELETE_OBJECT_ID文件系统控制代码则允许应用程序从文件中删除对象ID。

加密

企业的用户通常在他们的计算机上存放一些敏感信息。尽管存放在公司服务器上的数据通常有适当的网络安全设置和物理访问控制机制来保护，但是，当笔记本电脑丢失或被盗时，存放在笔记本电脑上的数据有可能泄露出去。在这种情况下，NTFS的文件许可机制并未能提供保护，因为通过NTFS文件阅读软件（这样的软件根本不要求运行Windows）可以在不涉及安全性的情况下访问NTFS的卷。而且，当在另一个Windows系统上从管理员账户来访问这些文件时，NTFS的文件许可机制也同样变得无效。第8章中曾经提到过，管理员账户具有“接管-所有权”特权和备份特权，这两者都使得可通过改写对象的安全设置来访问任何受保护的对象。

NTFS包含一个称为加密文件系统（EFS，Encrypting File System）的设施，用户可以利用该设施来加密敏感数据。EFS的操作，如同文件压缩操作一样，对于应用程序是完全透明的，这意味着：当一个应用程序读文件数据时，若它是在一个有权查看该文件数据的用户账户下运行的，则这些文件数据被自动解密；当一个授权的应用程序改变文件数据时，文件数据被自动加密。

www.TopSage.com



> **注** NTFS不允许系统卷的根目录或\Windows目录中的文件被加密，因为这些目录中的许多文件对于引导过程是必需的，而在引导过程中EFS尚未被激活。

EFS依赖于Windows提供的、用户模式下的密码学服务，所以，它是由一个内核模式组件和一些用户模式DLL组成的，其中内核模式组件跟NTFS紧密地集成在一起，而用户模式DLL负责跟本地安全权威子系统（LSASS，Local Security Authority Subsystem）和密码学DLL进行通信。

被加密的文件只能通过一个账户的EFS私钥/公钥对中的私钥才能访问，而私钥则通过一个账户的口令被锁起来。因此，在被丢失或被偷走的笔记本电脑上，EFS加密的文件也无法通过任何途径（除了使用密码学上的强力攻击以外）被访问到，除非获得了有权查看此数据的某个账户的口令。

应用程序可以使用*EncryptFile*和*DecryptFile* Windows API函数来加密和解密文件，使用*FileEncryptionStatus*来获得一个文件或目录的与EFS有关的属性，比如该文件或目录是否已被加密。

POSIX支持

正如在第2章中所解释的那样，Windows的早期要求之一是完全支持POSIX 1003.1标准。在文件系统领域，POSIX标准意味着要求支持大小写敏感的文件和目录名称、穿越目录的许可（在确定一个用户是否有权访问一个文件或目录时，要用到该路径中每个目录的许可权限）、“文件改变时间”时间戳（不同于MS-DOS的“最后修改时间”时间戳），以及硬链接（多个目录项指向同一个文件）。NTFS实现了这些特性。

碎片整理

自从引入NTFS以来，许多人持有的一种常见误解是，它自动地优化磁盘上文件的位置，所以不会使文件变成碎片。虽然NTFS确实做了很多努力来保持文件的连续性，但是，一个卷的文件还是有可能会随着时间的推移而变碎，尤其是当只有有限的空闲空间之时。如果一个文件的数据占据了不连续的簇时，则该文件便变碎了。例如，图12.19显示了一个变碎的文件，它由三块碎片组成。然而，如同绝大多数文件系统（包括Windows上的各个FAT版本）一样，NTFS并没有做特殊的努力来使得文件保持连续，除了为MFT（主文件表，master file table）保留一个特殊的区域（称为MFT区域）以外（当卷释放空间变得很慢时，NTFS让其他的文件从MFT区域中分配空间）。为MFT保留一个空闲的区域，这可以帮助它保持连续性，但是，该区域也可能变碎（有关MFT的更多信息，请参见本章后面的“主文件表”一节）。

www.TopSage.com

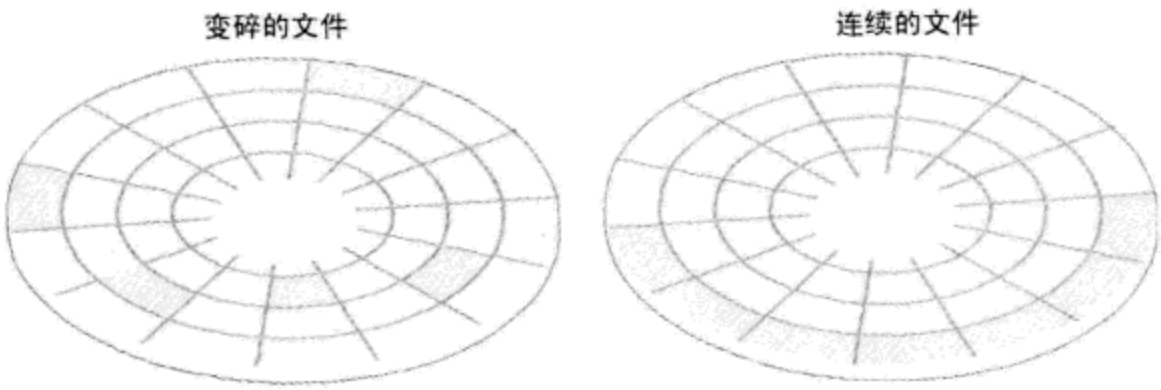


图12.19 变碎的文件和连续的文件

为了支持第三方开发磁盘碎片整理工具，Windows包含了一个碎片整理API。这一类工具可以利用该API来移动文件数据，从而使文件占据连续的簇。此API是由一些文件系统控制代码（FSCTL_GET_VOLUME_BITMAP）构成的。它让应用程序获得一个卷的空闲簇（free cluster）和在使用簇（in-use cluster）的映射表；FSCTL_GET_RETRIEVAL_POINTERS，获得一个文件的簇使用情况的映射表；FSCTL_MOVE_ FILE，移动一个文件。

Windows包含一个内置的碎片整理工具，通过使用Disk Defragmenter工具（\Windows\System32\Dfrg.msc）可以访问到它。对此内置的碎片整理工具有许多限制，在Windows 2000上，它不能从命令提示符下运行，也不能被安排自动运行。在Windows XP和Windows Server 2003上，此碎片整理工具有一个命令行界面\Windows\System32\Defrag.exe，所以你可以交互式地运行它，或者安排自动运行，但是它不会产生细节的报告信息，也不提供对整理过程的控制能力（比如排除一些文件或目录）。第三方磁盘碎片整理产品通常会提供更加丰富的功能集合。

在Windows XP中，NTFS中的碎片整理支持被重写了，以便减少原本在Windows 2000中的一些限制。例如，在Windows 2000中，NTFS中的碎片整理支持依赖于缓存管理器，这带来了许多限制，比如不能移动一个文件中跨越256KB边界的数据块，也不能整理NTFS的元数据文件（比如主文件表）。（注意，256KB正好是缓存管理器的视图的大小。有关缓存管理器的更多信息，请参见第11章。）在Windows XP和Windows Server 2003中，NTFS的碎片整理功能所强加的惟一限制是，页面文件和NTFS日志文件不能整理碎片。

只读支持

在Windows XP以前，NTFS文件系统驱动程序有这样的要求：它所挂载的卷必须位于可写介质上，这样它可以重置其事务型日志文件。Windows XP和Windows Server 2003的NTFS驱动程序可以挂载只读介质上的卷，对于具有只读的基本文件系统映像的嵌入式系统来说，这一功能是必要的。

www.TopSage.com

12.5 NTFS文件系统驱动程序

正如第9章中所描述的，在Windows I/O系统的框架中，NTFS和其他的文件系统是运行于内核模式下的可加载设备驱动程序。使用Windows或其他I/O API（比如POSIX）的应用程序间接地调用这些设备驱动程序。如图12.20所示，Windows环境子系统调用Windows的系统服务，而Windows的系统服务又找到正确的已加载的驱动程序并调用它们（有关系统服务分发的详细描述，请参见第3章中的“系统服务分发”一节）。

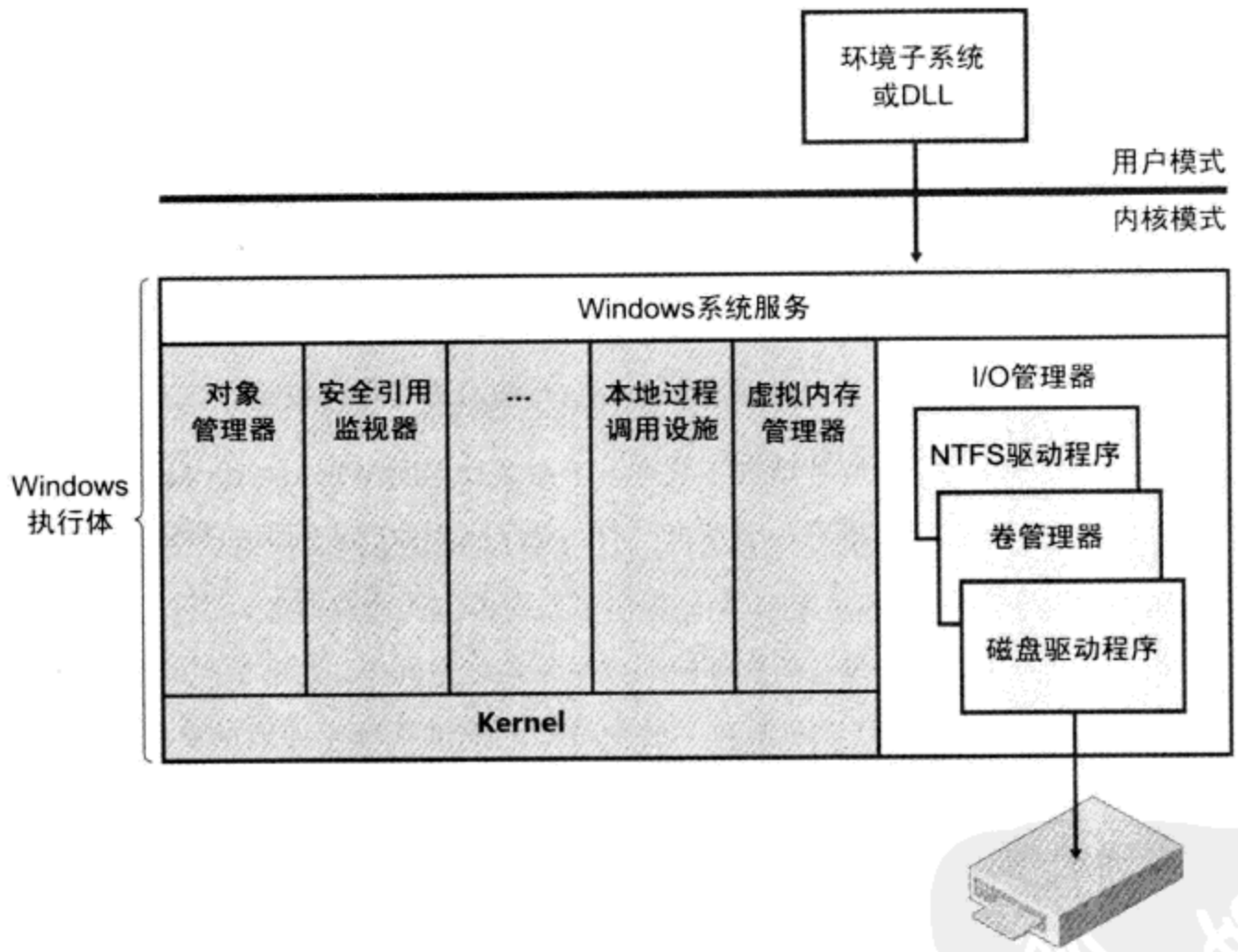


图12.20 Windows I/O系统的组件

分层的驱动程序通过调用Windows执行体的I/O管理器，将I/O请求传递给另一个驱动程序。依赖于I/O管理器作为一个中间载体，使得每个驱动程序可以维护其独立性，因而它可以在不影响其他驱动程序的情况下被加载进来或卸载出去。而且，NTFS驱动程序与其他三个Windows执行体组件打交道，如图12.21的左侧所示，这三个执行体组件与文件系统紧密相关。

www.TopSage.com

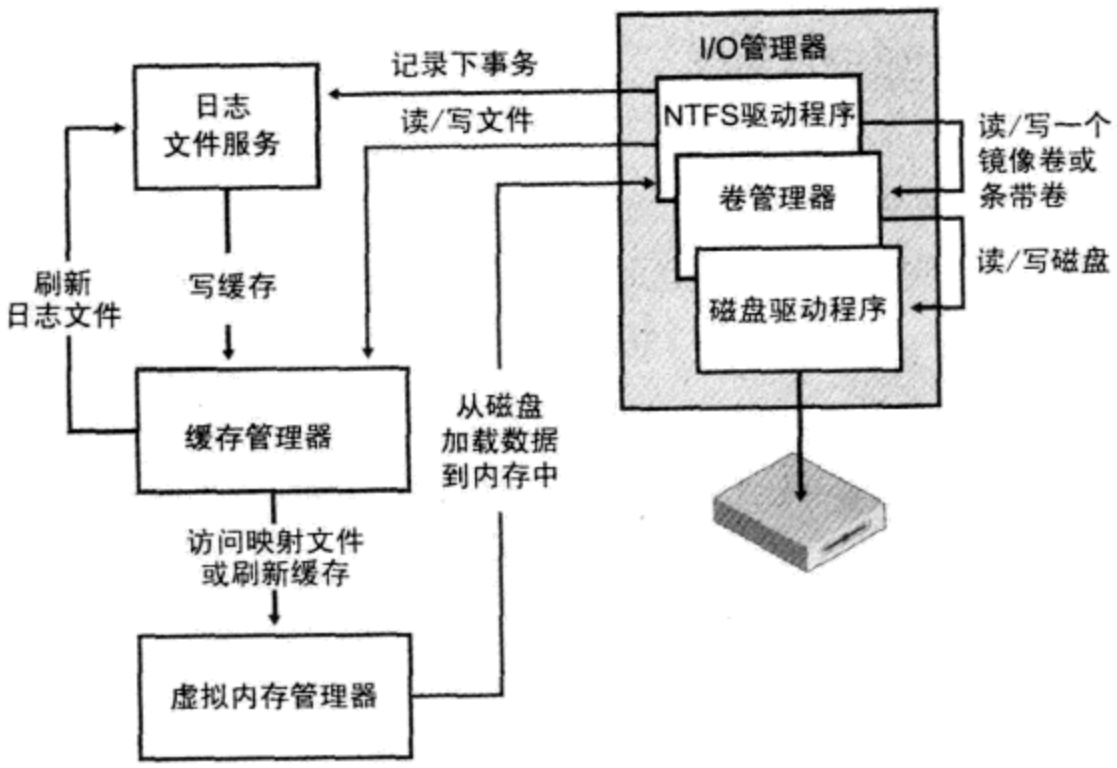


图12.21 NTFS和相关的组件

日志文件服务（LFS，log file service）是NTFS的一部分，它提供了相应的服务来维护关于磁盘写操作的日志。万一系统失败的话，LFS写的日志文件可被用来恢复一个NTFS格式的卷（参见“日志文件服务（LFS）”一节）。

缓存管理器是Windows执行体的组件，它为NTFS和其他的文件系统驱动程序（包括网络文件系统驱动程序[服务器和重定向器]）提供了系统范围的缓存服务。所有为Windows而实现的文件系统都通过“先把被缓存的文件映射到系统地址空间中，然后访问此虚拟内存”的做法来访问被缓存的文件。缓存管理器为了这个目的，提供了一个专门的文件系统接口来访问Windows内存管理器。当一个程序试图访问一个文件中尚未被加载到缓存中的部分（**缓存未命中**，*cache miss*）时，内存管理器调用NTFS来访问磁盘驱动程序，从磁盘中获取此文件内容。缓存管理器优化了磁盘I/O，它利用其延迟写出器线程（lazy writer thread）来调用内存管理器，通过后台活动将缓存的内容刷新到磁盘上（异步磁盘写操作）。（有关缓存管理器的完整描述，请参见第11章。）

NTFS将文件实现为对象，从而参与到Windows的对象模型中。这种文件实现方式使得文件可以被共享，也可以通过对象管理器（负责管理所有执行体层次的对象的Windows组件）来实施保护（第3章中的“对象管理器”一节讲述了对象管理器）。

www.TopSage.com

应用程序创建和访问文件的方式就如同它创建和访问其他Windows对象的做法一样：通过对象句柄来进行。当一个I/O请求到达NTFS时，Windows对象管理器和安全系统已经检验过，此调用进程有权按照它所期望的方式来访问该文件对象。安全系统已经比较过调用者的访问令牌与该文件对象的访问控制列表中的项目（有关访问控制列表的更多信息，请参见第8章）。I/O管理器也已经把文件句柄转变成一个指向文件对象的指针。NTFS利用文件对象中的信息来访问磁盘上的文件。

图12.22显示了相应的数据结构，它们将一个文件句柄与文件系统在磁盘上的结构链接起来。

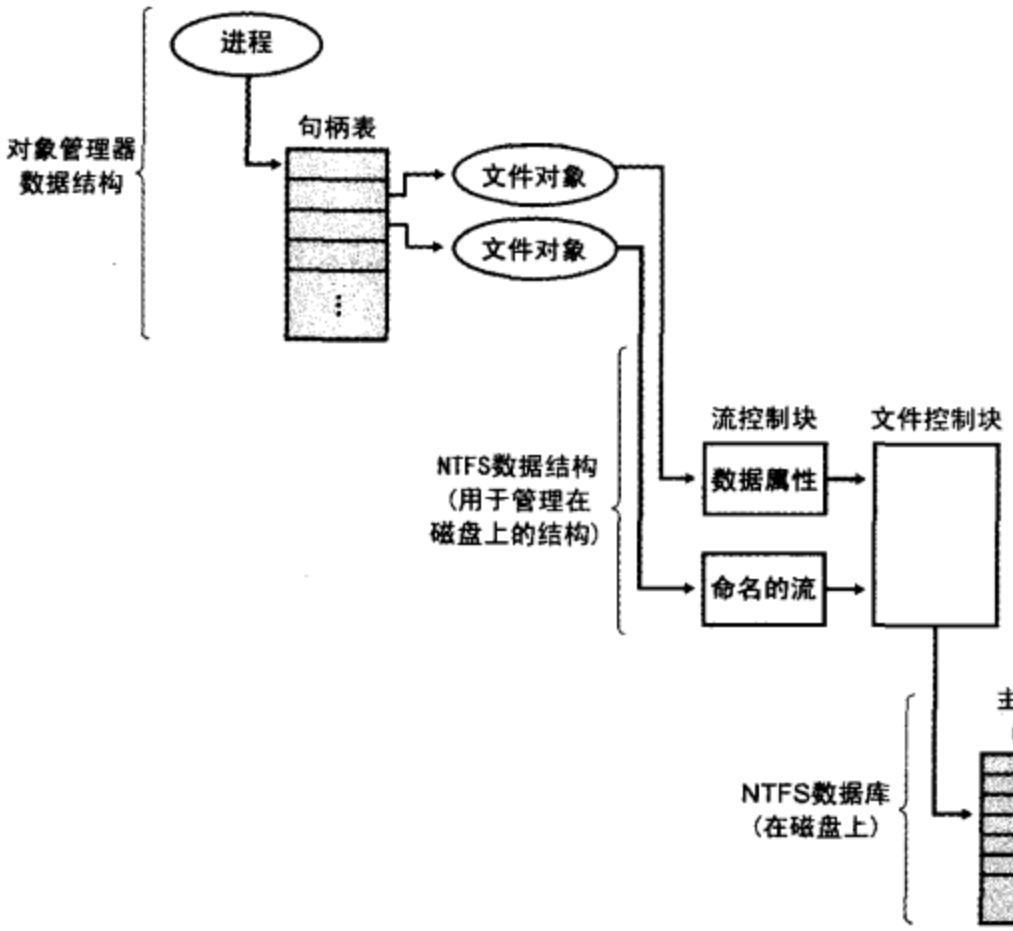


图12.22 NTFS数据结构

NTFS顺着几个指针，就可以从文件对象到达磁盘上该文件所在的位置。在图12.22中，文件对象代表了对于“打开文件（open-file）系统服务”的一次调用，它指向调用者想要读或写的文件属性所对应的**流控制块**（SCB，*stream control block*）。在该图中，一个进程既打开了此文件的未命名的数据属性，也打开了它的一个命名的流（另外的数据属性）。SCB代表单独的文件属性，它包含了有关如何找到一个文件内部的某些特定属性的信息。一个文件的所有SCB指向一个公共的数据结构，称为**文件控制块**（FCB，*file control block*）。FCB包含一个指针（实际上，是一个文件引用，参见本章后面的“文件引用号”一节中的解释），指向该文件在磁盘的主文件表（MFT）中的纪录（关于MFT的详细描述，请参见下一节）。

www.TopSage.com

12.6 NTFS在磁盘上的结构

这一节讲述NTFS卷在磁盘上的结构，包括磁盘空间是如何划分并组织成簇的，文件是如何被组织成目录的，实际的文件数据和属性信息是如何保存在磁盘上的，以及NTFS数据压缩是如何实现的。

卷（volume）

NTFS的结构首先从卷开始。**卷**（*volume*）对应于磁盘上的一个逻辑分区，当你将一个磁盘或者磁盘的一部分格式化成NTFS时，卷被创建起来。你也可以利用Windows的磁盘管理MMC加载件来创建一个跨越多个磁盘的RAID卷。

一个磁盘可以只有一个卷，也可以有几个卷。NTFS独立地处理每一个卷。图12.23中显示了一个150MB硬盘的三种示例性磁盘配置。

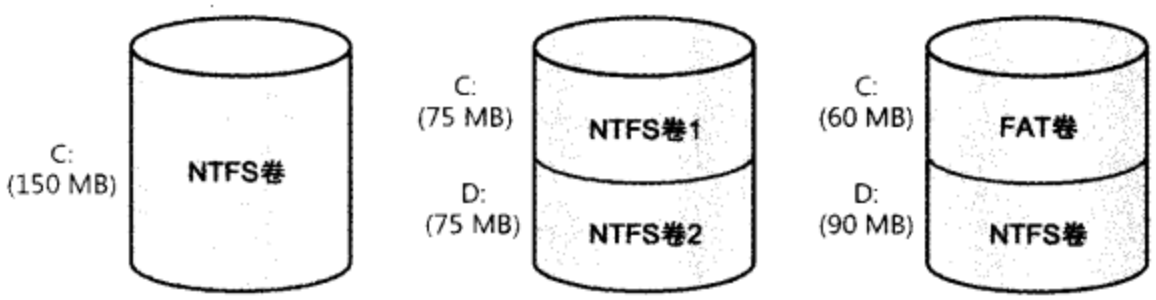


图12.23 示例性的磁盘配置

卷是由一系列文件以及该磁盘分区上剩余的、尚未分配的空间构成的。在FAT文件系统中，卷也包含了一些经过特殊格式化的、由文件系统使用的区域。然而，NTFS卷把所有的文件系统数据都存储为普通的文件，比如位图和目录，甚至系统引导程序。



> 注 在Windows 2000上，NTFS卷的磁盘格式为3.0版本，在Windows XP和Windows Server 2003上，由于对该格式有了些微变化，因此，NTFS卷的格式为3.1版本。一个卷的版本号被保存在它的$Volume元数据文件中。

簇（cluster）

在一个NTFS卷上，簇的大小，或者说**簇因子**（*cluster factor*），是在用户通过*format*命令或磁盘管理MMC加载件格式化该卷的时候建立起来的。默认的簇因子随着卷大小的不同而不同，但它是物理扇区的整数倍，而且总是2的幂（1个扇区，2个扇区，4个扇区，8个扇区，依此类推）。簇因子被表示成一个簇中的字节数，比如512字节、1KB或2KB。

www.TopSage.com

NTFS在内部只引用到簇一级（然而，NTFS提供了以扇区对齐的低级I/O操作，其长度是扇区大小的倍数）。NTFS使用簇作为它的分配单位，从而保持它与物理扇区大小之间的独立性。这种独立性使得NTFS只需使用一个更大的簇因子就可以有效地支持非常大的磁盘，或者支持非标准的磁盘（其扇区大小并非512字节）。在一个大的卷上，使用更大的簇因子可以减少碎片，并提高分配速度，但从被浪费的磁盘空间而言，则需要付出一点小小的代价。无论是Windows命令提示符下所使用的*format*命令，或是磁盘管理MMC加载件中Action菜单上All Tasks选项下的Format菜单项，都根据卷大小来选择一个默认的簇因子，但是你可以改变此大小值。

NTFS依赖于**逻辑簇号**（LCN，*logical cluster number*）来引用物理位置。LCN只不过是从一个卷的开始直到结束的所有簇的一个编号而已。为了将LCN转换成一个物理磁盘地址，NTFS将LCN乘以簇因子，从而得到该卷上的物理字节偏移，这正符合磁盘驱动程序接口的要求。NTFS依赖于**虚拟簇号**（VCN，*virtual cluster number*）来引用一个文件内部的数据。VCN是这样形成的：它对于同属于一个文件的所有簇进行编号，从0到m。然而，VCN不必是物理上连续的；它们可以被映射到该卷上的任何LCN编号。

主文件表（MFT）

在NTFS中，一个卷上存储的所有数据都被包含在文件中，包括定位和获取文件所使用的数据结构、引导程序的数据，以及记录整个卷分配状态的位图（NTFS元数据）。将所有的一切都保存在文件中，这使得文件系统很容易找到数据和维护这些数据，而且，每个单独的文件可以通过一个安全描述符来保护。另外，如果磁盘的某一特定部分损坏了的话，则NTFS可以重新找到元数据文件，以防止该磁盘变得不可访问。

MFT是NTFS卷结构的核心所在。MFT被实现为一个文件纪录阵列。每个文件纪录的大小是固定的，不管簇的大小如何，它都是1KB（关于文件纪录的结构，请参见本章后面的“文件纪录”一节）。从逻辑上讲，MFT为该卷上的每个文件都包含了一个纪录，其中也包括针对MFT本身的一个纪录。除了MFT以外，每个NTFS卷还包含一组元数据文件，这些元数据文件包含了用于实现此文件系统结构的信息。这些NTFS元数据文件中的每一个都有一个以美元符号（$）打头的名称，只不过该符号被隐藏起来了。例如，MFT的文件名是$Mft。NTFS卷上的其余文件都是普通的用户文件和目录。MFT文件纪录如图12.24所示。

www.TopSage.com



图12.24 在MFT中，针对NTFS元数据文件的文件纪录

通常，每个MFT纪录对应于一个不同的文件。然而，如果一个文件有大量的属性，或者变得极其碎，则有可能针对单个文件需要多个纪录。在这种情况下，第一个MFT纪录存储了其他MFT纪录的位置，它被称为**基本文件纪录**（*base file record*）。



实验：查看MFT

OEM支持工具箱（Windows调试工具箱的一部分，通过*support.microsoft.com/support/kb/articles/Q253/0/66.asp*可以下载）中包含的Nfi工具允许你把一个NTFS卷的MFT的内容转储出来，也可以把一个卷的簇号或者物理磁盘扇区号（仅针对非RAID卷）翻译为包含该簇或扇区的文件（如果该簇或扇区属于某个文件的话）。MFT的前16项被保留用于元数据文件，但是可选的元数据文件(只有当一个卷用到了相关的特性时才会出现的元数据文件)落在该区域以外：\$Extend\$Quota、\$Extend\$ObjId、\$Extend\$UsnJrnl和\$Extend\$Reparse。在下面的转储输出中，实验所用的卷使用了重解析点（$Reparse）、配额（$Quota）和对象ID（$ObjId）这些特性：

```
C:\>nfi G:\
NTFS File Sector Information Utility.
Copyright (C) Microsoft Corporation 1999. All rights reserved.
```


www.TopSage.com

```
File 0
Master File Table ($Mft)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 32-52447 (0x20-0xccdf)
    $BITMAP (nonresident)
        logical sectors 16-23 (0x10-0x17)

File 1
Master File Table Mirror ($MftMirr)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048728-2048735 (0x1f42d8-0x1f42df)

File 2
Log File ($LogFile)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048736-2073343 (0x1f42e0-0x1fa2ff)
File 3
DASD ($Volume)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $OBJECT_ID (resident)
    $SECURITY_DESCRIPTOR (resident)
    $VOLUME_NAME (resident)
    $VOLUME_INFORMATION (resident)
    $DATA (resident)

File 4
Attribute Definition Table ($AttrDef)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 512256-512263 (0x7d100-0x7d107)

File 5
Root Directory
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $INDEX_ROOT $I30 (resident)
    $INDEX_ALLOCATION $I30 (nonresident)
        logical sectors 2073416-2073423 (0x1fa348-0x1fa34f)
    $BITMAP $I30 (resident)

File 6
Volume Bitmap ($BitMap)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
```


www.TopSage.com

```
    $DATA (nonresident)
        logical sectors 2073424-2073675 (0x1fa350-0x1fa44b)
File 7
Boot Sectors ($Boot)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 0-15 (0x0-0xf)
File 8
Bad Cluster List ($BadClus)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (resident)
    $DATA $Bad (nonresident)

File 9
Security ($Secure)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA $SDS (nonresident)
        logical sectors 2073932-2074447 (0x1fa54c-0x1fa74f)
        logical sectors 523160-523163 (0x7fb98-0x7fb9b)
    $INDEX_ROOT $SDH (resident)
    $INDEX_ROOT $SII (resident)
    $INDEX_ALLOCATION $SDH (nonresident)
        logical sectors 1876152-1876159 (0x1ca0b8-0x1ca0bf)
    $INDEX_ALLOCATION $SII (nonresident)
        logical sectors 24-31 (0x18-0x1f)
    $BITMAP $SDH (resident)
    $BITMAP $SII (resident)

File 10
Upcase Table ($UpCase)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073676-2073931 (0x1fa44c-0x1fa54b)

File 11
Extend Table ($Extend)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $I30 (resident)

File 12
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)
File 13
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
```


www.TopSage.com

```
    $DATA (resident)
File 14
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 15
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 24
\$Extend\$Quota
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)
    $INDEX_ROOT $Q (resident)

File 25
\$Extend\$ObjId
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)

File 26 \$Extend\$Reparse
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $R (resident)
```

当NTFS第一次访问一个卷时，它必须**挂载**（*mount*）该卷。也就是说，从磁盘上读入元数据并构建起内部的数据结构，从而它能够处理应用程序的文件系统访问请求。为了挂载该卷，NTFS查询引导扇区，以找到MFT的物理磁盘地址。MFT自己的文件纪录是该表中的第一项；第二个文件纪录指向位于该磁盘中间位置处的一个文件，称为MFT**镜像**（*MFTmirror*，文件名为$MftMirr），它包含了MFT的前几行数据的一份拷贝。如果MFT文件的一部分由于某种原因无法读取时，通过这一部分拷贝仍可以找到元数据文件。

一旦NTFS找到了MFT自己的文件纪录，它就从该文件纪录的数据属性中获得有关VCN-LCN映射关系的信息，并且将这些信息存放在内存中。每一个**行串**（*run*）都有一个VCN-LCN映射项，以及一个行串长度，因为要从一个VCN找到其对应的LCN，这是所有必要的信息。这一映射信息让NTFS知道，构成此MFT的行串位于磁盘上的什么位置处（在本章后面的“驻留的和非驻留的属性”一节中解释了行串的概念）。然后，NTFS处理其他一些元数据文件的MFT纪录，并打开这些文件。接下来，NTFS执行它的文件系统恢复操作（在“恢复”一节中讲述）。最后，它打开其余的元数据文件。现在，该卷已经准备好接受用户的访问请求了。

www.TopSage.com



> 注 为了清楚起见，本章中的文字和图表将行串描述成包含有“一个VCN、LCN和行串长度”。NTFS实际上在磁盘上将这些信息压缩成一个“LCN/下一个VCN”对。知道了一个起始VCN，NTFS将下一个VCN减去起始VCN，就可以得到一个行串的长度。

在系统运行的时候，NTFS写另一个重要的元数据文件，即**日志文件**（*log file*，文件名为$LogFile）。NTFS使用此日志文件来记录所有影响到NTFS卷结构的操作，包括文件创建和任何会改变目录结构的命令，比如*Copy*。在一次系统失败以后，利用日志文件可以恢复一个NTFS卷，本章后面的“恢复”一节中讲述了这一过程。

MFT中的另一项被保留用于**根目录**（*root directory*，即人们熟知的“\”）。它的文件纪录包含了NTFS目录结构的根下面存放的文件和目录的索引。当NTFS被第一次请求打开一个文件时，它在根目录的文件纪录中开始对该文件的搜索过程。在打开了一个文件以后，NTFS将该文件的MFT文件引用、保存起来，从而当它以后读或写该文件时，它可以直接访问该文件的MFT纪录。

NTFS把一个卷的分配状态记录在**位图文件**（*bitmap file*，文件名为$Bitmap）中。位图文件的数据属性包含一个位图，其中的每一位代表了该卷上的一簇，指明该簇是空闲的，或是已经被分配给一个文件了。

安全文件（*security file*，文件名为$Secure）包含了整个卷范围的安全描述符数据库。NTFS的文件和目录有单独的、可设置的安全描述符，但是，为了节省空间，NTFS将这些设置存储在一个公共的文件中，这使得具有相同安全设置的文件和目录可以指向同样的安全描述符。在绝大多数环境中，整个目录树具有相同的安全设置，所以，这种优化带来了显著的空间节省。

另一个系统文件，即**引导文件**（*boot file*，文件名为$Boot），存储了Windows的引导代码。为了让系统能够引导起来，此引导代码必须位于某一特定的磁盘位置处。然而，*format*命令在格式化过程中，通过为该区域创建一个文件纪录，把该区域定义成一个文件。引导文件的创建，使得NTFS遵从了“把磁盘上的一切都做成文件”的原则。引导文件以及其他的NTFS元数据文件可以分别通过安全描述符这种适用于所有Windows对象的机制来加以保护。使用这种“磁盘上的一切皆文件”的模型，也意味着可通过常规的文件I/O来修改引导程序，尽管引导文件是受保护不能随意编辑的。

NTFS也维护了一个**坏簇文件**（*bad-cluster file*，文件名为$BadClus），此文件用于记录该磁盘卷上所有的坏点；另外，它还维护了一个名为**卷文件**（*volume file*，文件名为$Volume）的文件，其中包含卷名称、用于格式化该卷的NTFS版本，以及一个特殊的位（当该位被设置的时候，表示发生了一次磁盘损坏，必须由Chkdsk工具来修复）。（有关Chkdsk工具的细节，本章后面还有进一步介绍。） **大写字符文件**（*uppercase file*，文件名为$UpCase）包含一张翻译表，用于小写字符和大小字符之间的翻译。NTFS维护的另一个文件（文件名为$AttrDef）包含了一张**属性定义表**（*attribute definition table*），此属性表定义了该卷上支持的属性类型，指明了这些属性是否可被索引，它们在系统恢复操作过程中是否能被恢复，等等。

www.TopSage.com

NTFS在元数据目录extensions（目录名$Extend）中存储了几个元数据文件，包括对象标识符文件（object identifier file，文件名为$ObjId）、配额文件（quota file，文件名为$Quota）、变化日志文件（change journal file，文件名为$UsnJrnl）和重解析点文件（reparse point file，文件名为$Reparse）。这些文件存储的信息与NTFS的可选特性有关。对象标识符文件存储了文件对象ID；在支持配额的卷上，配额文件存储了配额限制和行为信息；变化日志文件记录了文件和目录的变化；重解析点文件存储了关于“该卷上哪些文件和目录中包含重解析点”的信息。



实验：查看NTFS信息

在Windows 2000中，你可以使用*www.sysinternals.com*的NTFSInfo工具来查看一个NTFS卷的信息，包括MFT和MFT区域的位置和大小；在Windows XP和Windows Server 2003中，你可以使用内置的Fsutil.exe命令行程序：

```
C:\Windows\System32>fsutil fsinfo ntfsinfo c:
NTFS Volume Serial Number :       0xe82828e72828b68a
Version :                         3.1
Number Sectors :                  0x0000000001e461b7
Total Clusters :                  0x00000000003c8c36
Free Clusters  :                  0x00000000000164c8
Total Reserved :                  0x00000000000001b0
Bytes Per Sector  :               512
Bytes Per Cluster :               4096
Bytes Per FileRecord Segment    : 1024
Clusters Per FileRecord Segment : 0
Mft Valid Data Length :           0x0000000006413800
Mft Start Lcn  :                  0x00000000000c5294
Mft2 Start Lcn :                  0x00000000000002f427
Mft Zone Start :                  0x00000000003bf7e0
Mft Zone End   :                  0x00000000003bf800
```

文件引用号

在NTFS卷上，文件可以通过一个称为**文件引用**（*file reference*）的64位值来标识。此文件引用是由一个文件号和一个序列号构成的。文件号对应于该文件在MFT中的文件纪录的位置减去1（或者，如果该文件有多个文件纪录的话，则是基本文件纪录的位置减去1）。文件引用序列号则随着每次一个MFT文件纪录位置被重新使用而增加1，这使得NTFS可以执行内部的一致性检查。图12.25显示了一个文件引用的结构。

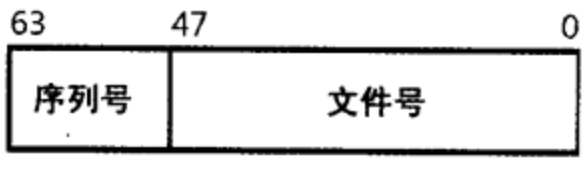


图12.25 文件引用

www.TopSage.com

文件纪录

NTFS并不是把文件仅仅看作文本数据或二进制数据的一个存储仓库，相反地，它以一组“属性/值”对的形式来存储文件，其中的一对“属性/值”是该文件所包含的数据（该属性称为**未命名的数据属性**[*unnamed data attribute*]）。构成一个文件的其他属性还有文件名、时间戳信息，以及其他额外的命名的数据属性。图12.26示出了一个小文件的MFT纪录。

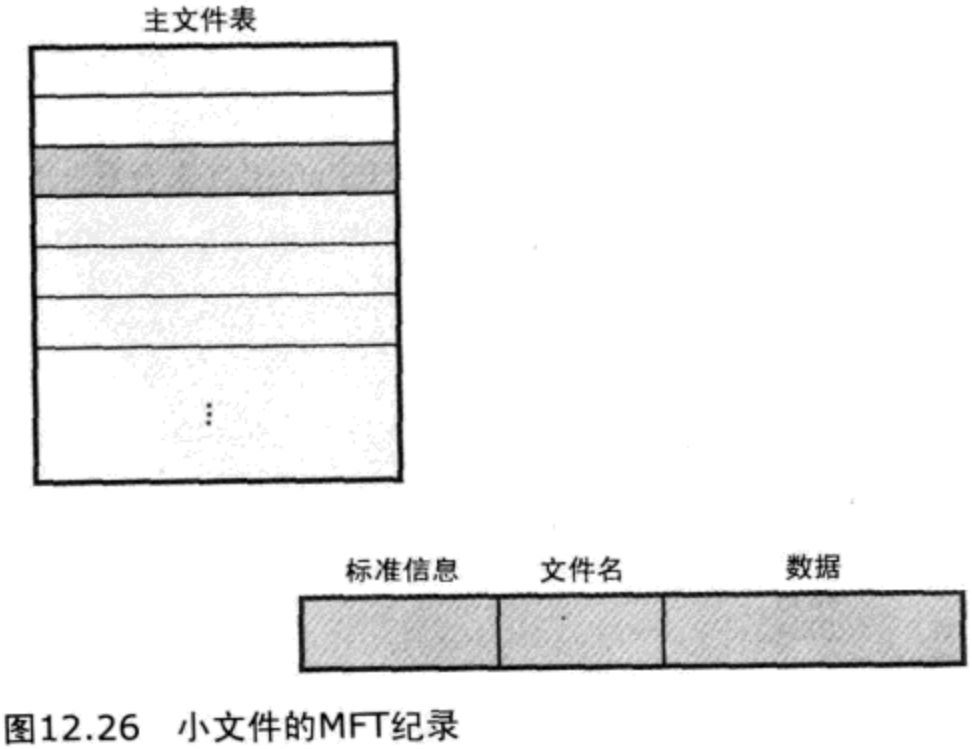


图12.26 小文件的MFT纪录

每个文件属性都被存储为该文件内部的一个独立字节流。严格来说，NTFS并不读和写文件，它读写的是属性流。NTFS提供了以下这些属性操作：创建、删除、读（字节范围）和写（字节范围）。读和写操作通常是在一个文件的未命名的数据属性上进行的。然而，调用者可以利用命名的数据流的语法来指定另一个不同的数据属性。

表12.4列出了一个NTFS卷上的文件所使用的属性（并不是每个文件中都出现所有这些属性）。

表12.4 NTFS文件的属性

属 性	属性名称	说 明
卷信息	$VOLUME_INFORMATION, $VOLUME_NAME	这些属性只出现在$Volume元数据文件中。它们保存了卷的版本和标注（label）信息
标准信息	$STANDARD_INFORMATION	诸如只读、存档等文件属性；时间戳，包括该文件何时被创建，最后一次修改是在何时；以及多少目录指向该文件（它的**硬链接计数**[*hard link count*]）

www.TopSage.com

续表

属 性	属性名称	说 明
文件名	$FILE_NAME	该文件的名称，以Unicode字符来表示。一个文件可以有多个文件名属性。当一个文件存在硬链接时，或者当一个具有长名称的文件有一个自动生成的“短名称”（为了让MS-DOS和16位Microsoft Windows应用程序访问）时，它有多个文件名属性
安全描述符	$SECURITY_DESCRIPTOR	此属性的出现是为了与NTFS的以前版本保持向后兼容。Windows版本的NTFS把所有的安全描述符都存储在$Secure元数据文件中，所以，具有相同安全设置的文件和目录共享同样的描述符。以前版本的NTFS将私有的安全描述符信息与每个文件和目录存储在一起
数据	$DATA	文件的内容。在NTFS中，一个文件只有一个默认的未命名的数据属性，但可以有额外的多个命名的数据属性，也就是说，一个文件可以有多个数据流。目录没有默认的数据属性，但可以有多个可选的命名的数据属性
索引根、索引分配和索引位图	$INDEX_ROOT, $INDEX_ALLOCATION, $BITMAP	这三个属性用来实现大目录的文件名分配和位图索引（仅适用于目录）
属性列表	$ATTRIBUTE_LIST	一个属性列表，其中包括构成该文件的这些属性，以及每个属性所在的MFT文件纪录的文件引用。只有一个文件要求多个MFT文件纪录时，此属性才会出现。它很少使用
对象ID	$OBJECT_ID	一个文件或目录的64字节标识符，其中低16字节（128位）对于该卷来说是惟一的。链接-跟踪服务将对象ID分配给外壳快捷方式和OLE链接源文件。NTFS提供了相应的API，因而文件和目录可以通过其对象ID，而不是通过其文件名来打开
重解析信息	$REPARSE_POINT	此属性存储了一个文件的重解析点数据。NTFS的交接（junction）和挂载点包含此属性
扩展属性	$EA, $EA_INFORMATION	扩展属性现在已不再被主动使用，之所以提供这些属性是因为要与OS/2应用程序保持向后兼容
EFS信息	$EFS	EFS存储在此属性中的数据被用于管理一个文件的加密操作，比如为了解密该文件而需要的密钥（此密钥本身是被加密过的），以及一组被授权可以访问该文件的用户的列表

www.TopSage.com

表12.4显示了属性名称。然而，属性实际上对应于数值化的类型代码，NTFS利用这些类型代码来对一个文件纪录内部的属性进行排序。一个MFT纪录中的文件属性是按照这些类型代码来排序的（按照数值的升序），其中有些属性类型会出现多次——譬如，若一个文件具有多个数据属性或者多个文件名。

一个文件纪录中的每个属性都是通过其属性类型代码来标识的，它有一个值和一个可选的名称。属性的值即为构成此属性的字节流。例如：$FILE_NAME属性的值是该文件的名称；$DATA属性的值是用户存储在该文件中的所有字节。

绝大多数属性没有名称，不过，与索引有关的属性和$DATA属性通常有名称。在一个文件所包含的同一类型的多个属性之间，通过名称可以对它们进行区分。例如，若一个文件有一个命名的数据流，则它有两个$DATA属性：一个未命名的$DATA属性存储默认的未命名数据流；一个命名的$DATA属性具有另一个流的名称，并且存储此命名的流的数据。

文件名

NTFS和FAT都允许一个路径中的每个文件名可以长达255个字符。文件名可以包含Unicode字符，以及多个句点和内嵌空格。然而，随MS-DOS提供的FAT文件系统限定了它的文件名不超过8个（非Unicode）字符，后面跟一个句点和一个3字符的扩展名。图12.27给出了Windows支持的不同文件名字空间的一个可视化表示形式，其中也显示了它们的交集。

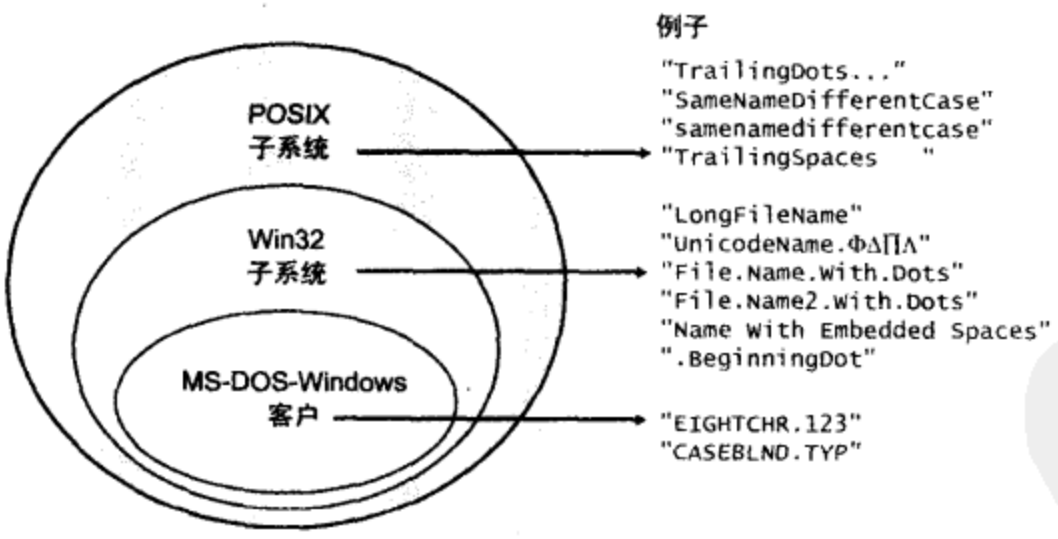


图12.27 Windows文件名字空间

POSIX子系统要求最大的名字空间，涵盖了Windows支持的所有应用程序执行环境，因此，NTFS名字空间等价于POSIX名字空间。POSIX子系统可以创建对于Windows和MS-DOS应用程序不可见的名称，包括具有结尾句点和结尾空格的名称。通常，使用大的POSIX名字空间来创建一个文件不是一个问题，因为只有当你有意让POSIX子系统或POSIX客户系统来使用该文件时，你才会这么做。

www.TopSage.com

然而，32位Windows应用程序和MS-DOS Windows应用程序之间是一层更近的关系。图12.27中的Windows区域代表了“在一个NTFS卷上，Windows子系统可以创建、但MS-DOS和16位Windows应用程序却无法看见的文件名”。这一组包括了比MS-DOS 8.3格式还要长的文件名、内含Unicode（国际）字符的文件名、内含多个句点字符或一个起始句点的文件名，以及包含内嵌空格的文件名。当用这样一个名称来创建一个文件时，NTFS自动为该文件生成一个可替代的MS-DOS风格的文件名。当你在*dir*命令中使用*/x*选项时，Windows会显示这些短名称。

这些MS-DOS文件名是对应NTFS文件的功能性的别称，它们被保存在与长文件名相同的目录中。图12.28显示了一个文件的MFT纪录，它具有自动生成的MS-DOS文件名。

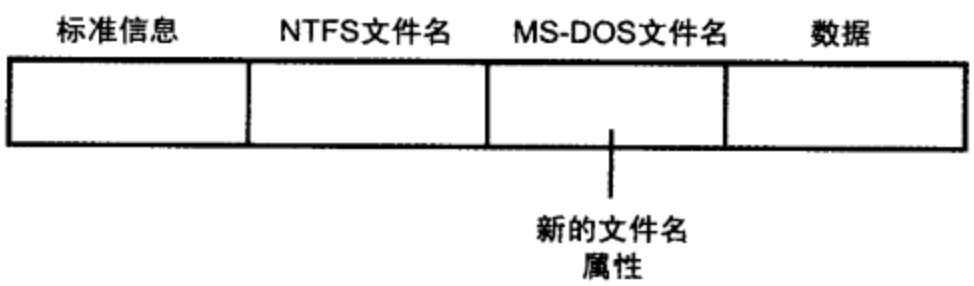


图12.28 具有MS-DOS文件名属性的MFT文件纪录

NTFS名称和自动生成的MS-DOS名称被存储在同样的文件纪录中，因此，它们指向同一个文件。利用MS-DOS名称可以打开、读、写或者拷贝该文件。如果用户使用长文件名或短文件名来重新命名该文件，则新的名称将同时替代这两个已有的名称。如果新的名称不是一个有效的MS-DOS名称，则NTFS为该文件生成另一个MS-DOS名称。



> **注** POSIX硬链接也是以一种类似的方式来实现的。当一个指向某个POSIX文件的硬链接被创建时，NTFS在该文件的MFT文件纪录中加入另一个文件名属性。然而，在一个方面，这两种情形并不相同。当用户删除一个具有多个名称（硬链接）的POSIX文件时，其文件纪录和文件本身仍然存在。只有当最后一个文件名（硬链接）被删除的时候，该文件和它的纪录才被删除。然而，如果一个文件同时具有NTFS名称和自动生成的MS-DOS名称，那么，用户通过任何一个名称都可以删除该文件。

NTFS利用下面的算法从一个长文件名生成一个MS-DOS名称。

1. 从长文件名中去掉任何在MS-DOS名称中非法的字符，包括空格和Unicode字符。去掉前导的和结尾处的句点。除了最后一个内嵌句点，去掉所有其他的内嵌句点。
2. 将句点以前（如果存在句点的话）的字符串截短成6个字符，并附加上字符串“*~n*”（这里*n*是一个数字，从1开始，用于区分那些被截短成相同名称的不同文件）。将句点以后（如果存在句点的话）的字符串截短成3个字符。

www.TopSage.com

3. 将结果转变成大写字母。MS-DOS是与大小写无关的，所以，这一步保证NTFS不会生成一个仅仅在大小写方面区别于已有文件名的新名称。
4. 如果生成的名称与目录中某个已有的名称重复的话，则递增~*n*字符串。

表12.5显示了图12.26中的长Windows文件名，以及NTFS生成的对应的MS-DOS名称。这里描述的算法，以及图12.26中的例子应该可以让你了解到NTFS生成的MS-DOS风格的文件名到底是什么样子。



> 注 你只需将注册表中的DWORD值HKLM\System\CurrentControlSet\Control\FileSystem\NtfsDisable8dot3NameCreation设置为1，就可以关闭生成短名称的功能，但是，这种做法一般不推荐，因为这可能会导致与那些“依赖于短名称的应用程序”不兼容。

表12.5 NTFS生成的文件名

Windows长名称	NTFS生成的短名称
LongFileName	LONGFI~1
UnicodeName.	UNICOD~1
File.Name.With.Dots	FILENA~1.DOT
File.Name2.With.Dots	FILENA~2.DOT
Name With Embedded Spaces	NAMEWI~1
.BeginningDot	BEGINN~1

驻留的和非驻留的属性

如果一个文件比较小，那么，所有它的属性和它们的值（例如它的数据）都适合于放在文件纪录中。当一个属性的值被直接存储在MFT中时，该属性称为**驻留的属性**(*resident attribute*)。（例如，在图12.27中，所有的属性都是驻留的。）有一些属性被定义为总是驻留的，所以NTFS可以通过它们找到非驻留的属性。例如，标准信息和索引根属性总是驻留的。

每个属性都是以一个标准头作为开始，此标准头包含了有关该属性的信息。NTFS以一种统一的方式，利用标准头的信息来管理属性。此头部总是驻留的，它记录了该属性的值是驻留的还是非驻留的。对于驻留的属性，此头部也包含了从头部到该属性的值之间的偏移，以及该属性的值的长度，图12.29显示了文件名属性的情形。

www.TopSage.com

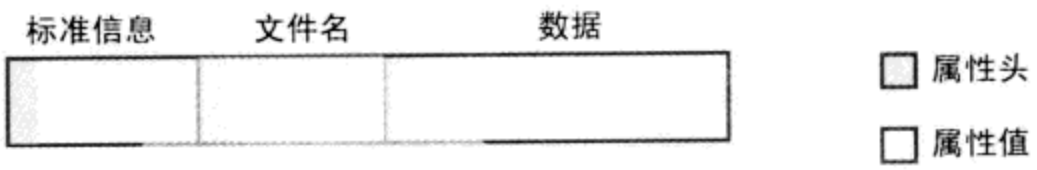


图12.29 驻留的属性的头和值

当一个属性的值被直接存储在MFT中时，NTFS访问该值所需要的时间大大地减少了。NTFS不需要先在一张表中查找一个文件，然后读入一组连续的存储单元以找到该文件的数据（譬如，FAT文件系统正是这样做的）。相反地，它访问磁盘一次就可以立即获得数据。

小目录以及小文件的属性可以驻留在MFT中，如图12.30所示。对于一个小目录，其索引根（index root）属性包含了有关该目录中的文件和子目录的文件引用的一份索引。

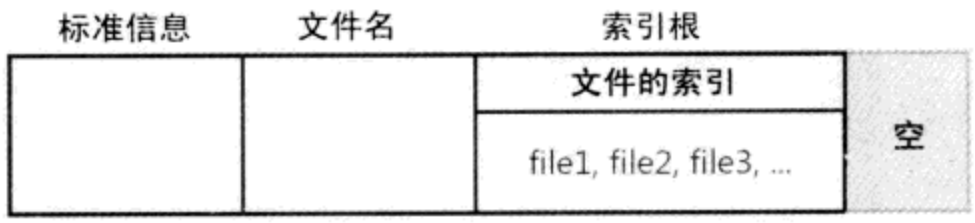


图12.30 小目录的MFT文件纪录

当然，许多文件和目录并不能被挤到一个1KB固定大小的MFT纪录中。如果某个特别的属性，比如一个文件的数据属性，因为太大而不能被包含到一个MFT文件纪录中，那么，NTFS为该属性的数据分配专门的簇，这些簇不在MFT中。此区域被称为一个**行串**（*run*，或者称为*extent*，**范围**）。如果此属性的值后来又增长了（例如，用户向该文件又追加了数据），则NTFS为追加的数据再分配另一个行串。如果属性的值被存储在行串中，而不是存储在MFT中，则这样的属性称为**非驻留的属性**（*nonresident attribute*）。文件系统会决定一个属性是驻留的，还是非驻留的；数据的位置对于访问该数据的进程来说是透明的。

如果一个属性是非驻留的，比如一个大文件的数据属性可能就是非驻留的，则它的头部包含了有关的信息，以便让NTFS能够在磁盘上找到该属性的值。图12.31显示了一个非驻留的数据属性，它被存储在两个行串中。

www.TopSage.com

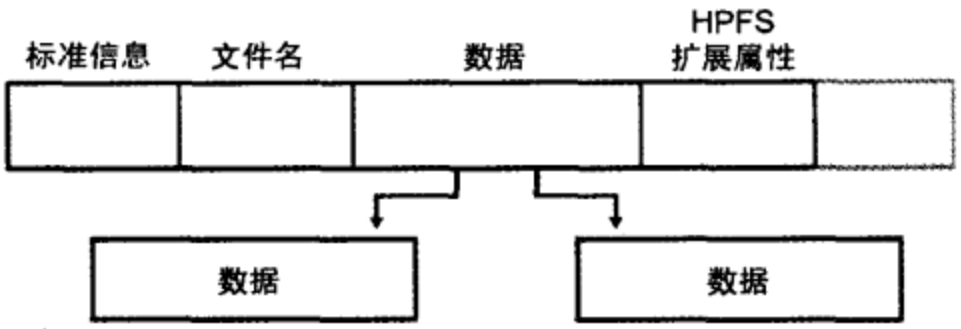


图12.31 具有两个数据行串的大文件的MFT文件纪录

在标准属性之中，只有那些可能会增长的属性才可能是非驻留的。对于文件，可能会增长的属性是数据和属性列表（在图12.31中没有显示）。标准信息和文件名属性总是驻留的。

大的目录也可能有非驻留的属性（或者有些属性的一部分是非驻留的），如图12.32所示。在这个例子中，MFT文件纪录没有足够的空间来存储此大目录的所有文件的索引信息。一部分索引被存放在索引根（index root）属性中，其余的索引被存放在称为**索引缓冲区**（*index buffer*）的非驻留行串中。图中显示的索引根、索引分配以及位图这三个属性都是简化形式。下一节更加详细地讲述了这些属性。标准信息和文件名属性总是驻留的。对于目录来说，头部，以及索引根属性值中的至少一部分总是驻留的。

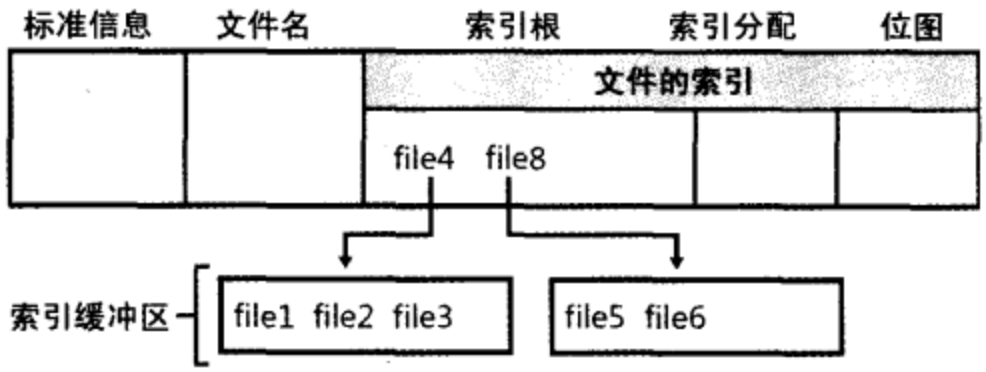


图12.32 具有非驻留文件名索引的大目录的MFT文件纪录

当一个文件（或者一个目录）的属性在一个MFT文件纪录中放不下，而需要独立的存储空间时，NTFS按照“VCN-LCN映射”对的方式记录下这些行串。LCN代表整个卷上簇的序列，从0到n。VCN是对所有属于某个特定文件的簇的编号，从0到m。例如，在一个非驻留数据属性的行串中，簇的编号如图12.33所示。

www.TopSage.com

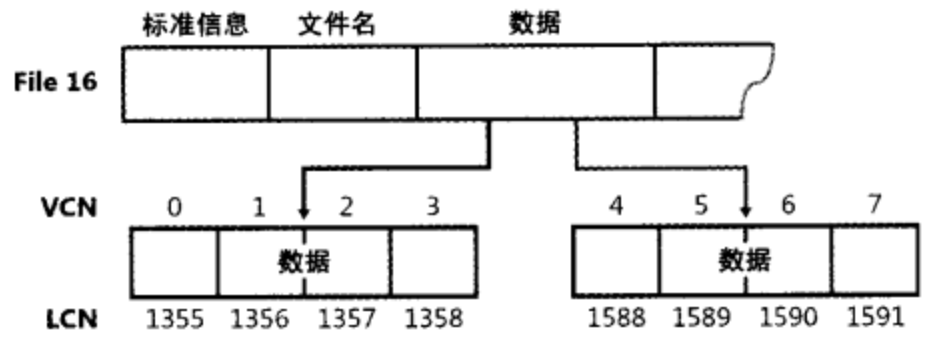


图12.33 一个非驻留数据属性的VCN

如果该文件有超过两个行串的话，则第三个行串的编号将从VCN 8开始。如图12.34所示，数据属性的头部包含了这里两个行串的VCN-LCN映射项，这使得NTFS很容易找到磁盘上的数据单元的位置。

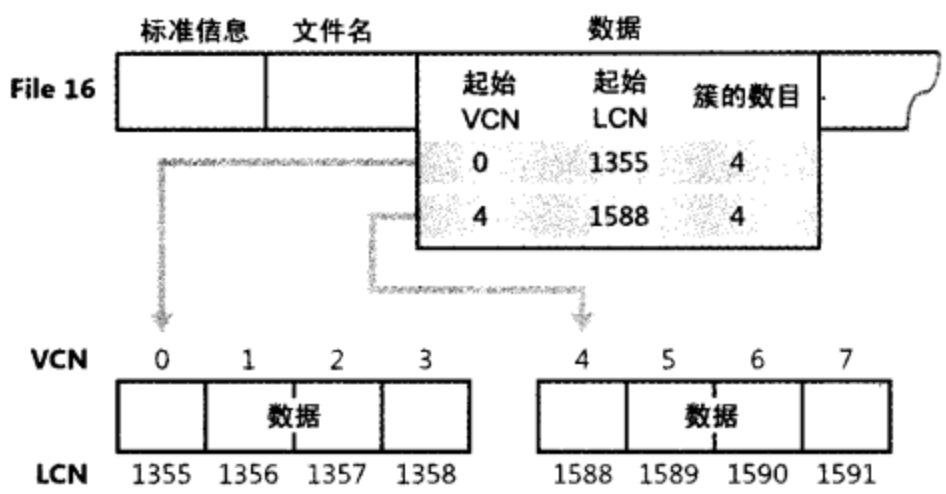


图12.34 一个非驻留数据属性的VCN-LCN映射表

图12.33显示的仅仅是数据行串，对于其他的属性，如果在MFT文件纪录中没有足够的空间来存放其值，则它们也可以被存放到行串中。而且，如果一个文件有太多的属性，而在 MFT纪录中放不下，那么，可以使用第二个MFT纪录来包含额外的属性(或者非驻留属性的属性头)。在这种情况下，需要加入一个称为**属性列表**（*attribute list*）的属性。属性列表属性包含了该文件的每个属性的名称和类型代码，以及该属性所在的MFT纪录的文件引用。之所以提供属性列表属性，是因为要应对这样的情形：由于文件增长得太大或者变得太碎，因而单个MFT纪录无法包含“为了找到所有行串而需要的诸多VCN-LCN映射项”。如果一个文件有超过200个行串，则它往往要求使用一个属性列表属性。

数据压缩和稀疏文件

NTFS支持以每个文件、每个目录或每个卷为基础的压缩功能（NTFS的压缩只在用户数据上执行，而不在文件系统元数据上执行）。你通过Windows的*GetVolumeInformation*函数，可以知道一个卷是否是压缩的。为了得到一个文件实际被压缩之后的大小值，你可以使用

www.TopSage.com

Windows的*GetCompressedFileSize*函数。最后，为了检查或者改变一个文件或者目录的压缩设置，你可以使用Windows的*DeviceIoControl*函数（请参见文件系统控制代码FSCTL_GET_COMPRESSION和FSCTL_SET_COMPRESSION）。记住，虽然通过设置一个文件的压缩状态，可以立即压缩（或者解压缩）该文件，但是，设置一个目录或卷的压缩状态并不会导致立刻压缩或解压缩。相反地，设置一个目录或卷的压缩状态，只是设置一个默认的压缩状态，将来在该目录或卷内新创建的文件或子目录会被赋予此默认的新状态。

下面的小节通过讨论一个压缩稀疏数据的简单例子来介绍NTFS的压缩功能。随后的小节将进一步讨论普通文件和稀疏文件的压缩。

压缩稀疏数据

稀疏数据（*sparse data*）通常非常大，但相对于其总体大小而言，它只包含很小一部分非零数据。稀疏矩阵是一个这样的例子。正如前面所描述，NTFS使用VCN，从0到m，来列举一个文件的所有簇。每个VCN映射到一个对应的LCN，而LCN标识了该簇在磁盘上的位置。图12.35显示了一个普通的非压缩文件的行串（磁盘分配），包括它的VCN及其对应的LCN。

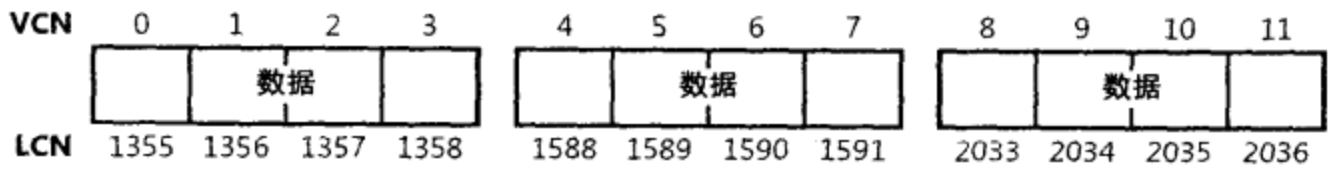


图12.35 非压缩文件的行串

该文件被存储在3个行串中，每个行串是4个簇，所以，总共有12簇。图12.36显示了该文件的MFT纪录。正如前面所描述，为了节省空间，此MFT纪录的数据属性（其中包含了VCN-LCN映射表）只为每个行串记录一个映射项，而并非为每一簇都记录一个映射项。然而，请注意，从0至11的每个VCN都有一个与之相关联的对应LCN。第一项从VCN 0开始，包含4个簇，第二项从VCN 4开始，也包含4个簇，依此类推。对于非压缩文件来说，这每一项的格式是非常典型的。

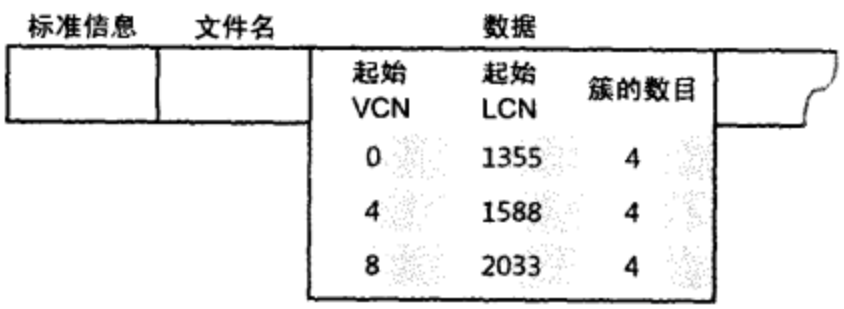


图12.36 非压缩文件的MFT纪录

当用户从NTFS卷上选择一个文件来压缩时，一种NTFS压缩技术是，从文件中去掉长长的

www.TopSage.com

零串。如果该文件的数据是稀疏的，那么，这通常会大大地缩短该文件，使得它只包含本该所需要的那部分磁盘空间。在随后接到对该文件的写请求时，NTFS只分配包含非零数据的行串空间。

图12.37显示了一个包含稀疏数据的压缩文件的行串。注意，该文件中某些特定范围的VCN（1631，以及64127）没有分配磁盘空间。

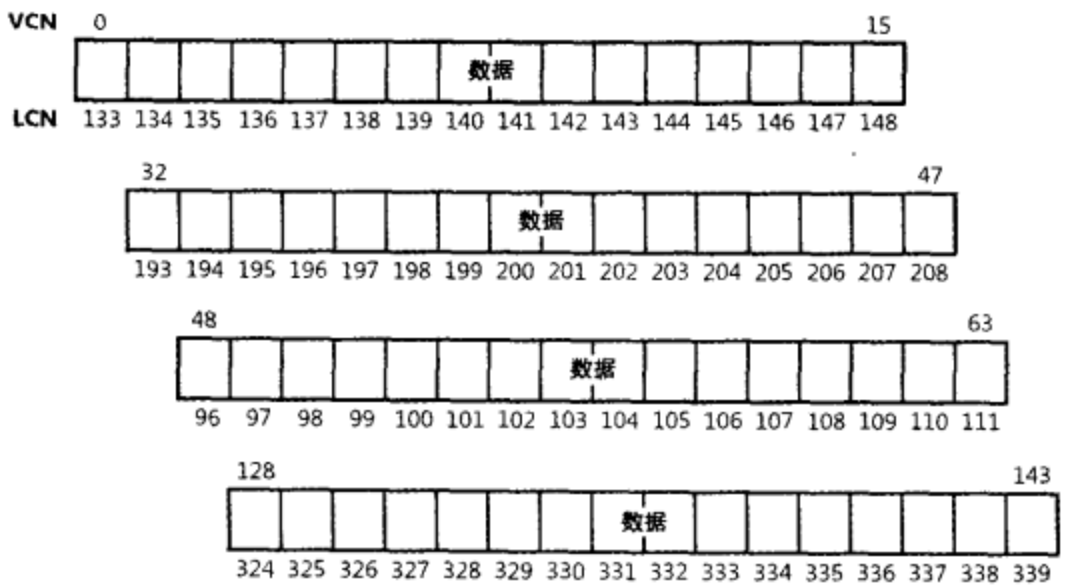


图12.37 一个包含稀疏数据的压缩文件的行串

此压缩文件的MFT纪录省略了那些包括零数据的VCN的块，因此，没有为它们分配物理存储空间。例如，图12.38中的第一个数据项从VCN 0开始，包含16个簇。第二项跳到VCN 32，包含16簇。

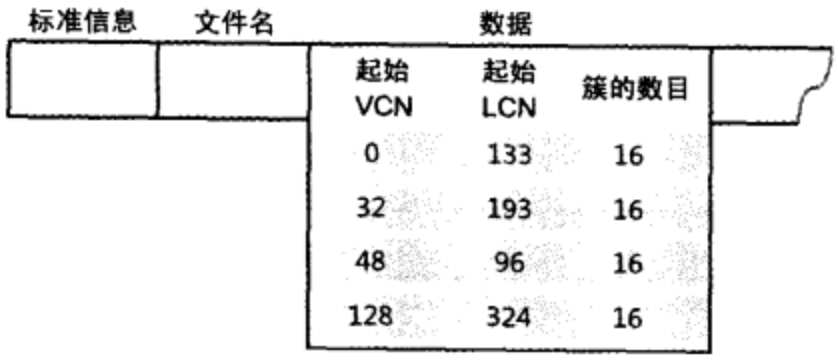


图12.38 一个包含稀疏数据的压缩文件的MFT纪录

当一个程序从一个压缩文件中读入数据时，NTFS检查MFT纪录，以确定是否有一个VCN-LCN映射项包含了所要读取的位置。如果此程序是从该文件中一个未被分配的“洞”中读取数据，则意味着，该文件的这一部分包含的数据全是零，所以，NTFS无需访问磁盘就返回零。如果一个程序将非零的数据写到一个“洞”里，则NTFS直接分配磁盘空间，然后写入数据。这一技术对于包含大量零数据的稀疏文件非常有效。

www.TopSage.com

压缩非稀疏数据

上面讲述的压缩稀疏文件的例子多少有一点人为编造的意思。它描述的“压缩”所针对的情形是，一个文件的某些区域整个被填充了零，但该文件余下的数据并未受到压缩的影响。绝大多数文件中的数据不是稀疏的，但是这些数据仍然可以通过采用一个压缩算法来加以压缩。

在NTFS中，用户可以为单独的文件指定压缩特性，或者为一个目录中的所有文件指定压缩特性（在一个被标记为压缩的目录中，新创建的文件自动被压缩——已有的文件必须被单独压缩）。当NTFS压缩一个文件时，它把该文件的未处理数据分成16簇长的**压缩单元**（*compression unit*，譬如，对于512字节的簇来说，此压缩单元等于8KB）。一个文件中某些特定的数据序列可能不会被压缩太多。因此，对于该文件中的每一个压缩单元，NTFS要确定压缩该单元是否能节省至少1簇的存储空间。如果压缩该单元未能释放出至少1簇，那么，NTFS分配一个16簇的行串，直接将该单元中的数据写到磁盘上，不对它做压缩。如果此16簇的单元中的数据能被压缩到15或更少的簇中，那么，NTFS只分配所需数量的簇，然后将压缩后的数据写入到磁盘中。图12.39示出了一个包含4个行串的文件的压缩情形。该图中未加阴影的区域代表了该文件被压缩之后所占用的实际存储位置。第一、第二和第四个行串都是压缩的；第三个行串未被压缩。尽管该文件有一个未压缩的行串，但是，压缩该文件仍然节省了26簇磁盘空间，或者说40%的空间。

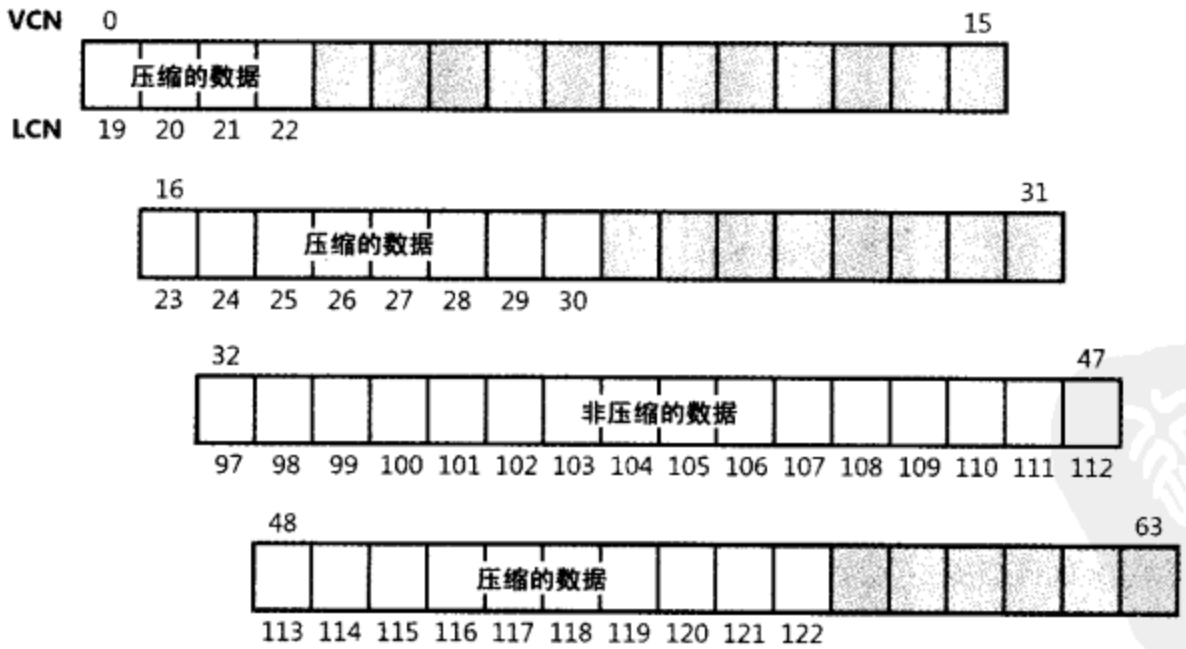


图12.39 一个压缩文件的数据行串



> 注 虽然本章的图中显示了连续的LCN，但实际上，一个压缩单元并不必须存储在物理上连续的簇中。占用非连续簇的行串将会导致所生成的MFT纪录比图12.39中显示的MFT纪录略微复杂一些。

www.TopSage.com

当NTFS向一个压缩文件写入数据时，它确保每一个行串都是从一个实际的16簇的边界开始的。因此，每一个行串的起始VCN都是16的倍数，并且这些行串不会比16簇还要长。每次当NTFS访问压缩的文件时，它读或写至少一个压缩单元。然而，当它写压缩数据时，NTFS试图将压缩单元存储在物理上连续的位置中，以便将来仅用一个I/O操作就可以把它们读出来。NTFS压缩单元的大小为16簇，这样选取的目的是为了减少内部碎片：压缩单元越大，存储这些数据所需要的总磁盘空间越小。选择16簇作为压缩单元的大小，这代表了在“生成更小的压缩文件”和“对于随机访问这些文件的程序，其读操作变慢”两者之间的一种权衡。每次缓存未命中时，必须把相当于16簇的数据解压缩出来（在随机的文件访问过程中，缓存未命名的情形更有可能发生）。图12.40显示了图12.39中所示的压缩文件的MFT纪录。

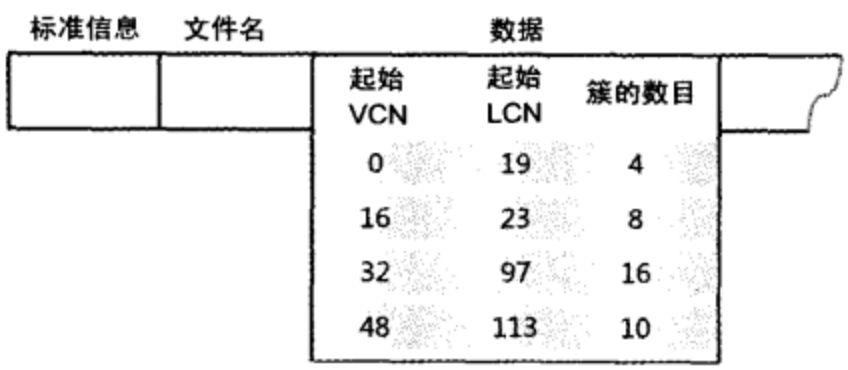


图12.40 一个压缩文件的MFT纪录

在以上压缩文件与前述的包含稀疏数据的压缩文件例子两者之间的一个区别是，这个文件中的三个被压缩的行串都小于16簇长。从一个文件的MFT文件纪录中读取到这一信息，NTFS就可以知道该文件中的数据是否经过了压缩。任何短于16簇的行串都包含了压缩之后的数据，当NTFS将数据读到缓存中时它必须执行解压缩操作。恰好等于16簇长的行串不包含压缩数据，因此无需解压缩操作。

如果一个行串中的数据已经被压缩了，则NTFS将该数据解压缩到一个临时使用的缓冲区中，然后把数据拷贝到调用者的缓冲区中。NTFS也会把解压缩的数据加载到缓存中，这使得后续的从同一个行串中读取数据的操作，能够像任何其他的被缓存的读操作一样快。NTFS把文件的所有更新数据写到缓存中，由延迟写出器负责对已修改的数据进行压缩，并且异步地写到磁盘上。这种策略确保了：相对于对非压缩文件的写操作，对压缩文件的写操作不会产生更为显著的延迟。

NTFS尽可能地保持一个压缩文件的磁盘分配区域是连续的。在图12.38中，根据LCN的指示，压缩文件的前两个行串是物理上连续的，最后两个也是。当两个或多个行串是连续的时候，NTFS执行磁盘预读，就好像针对其他文件中的数据所做的预读一样。因为对于连续文件数据的读取和解压缩操作是在程序请求数据以前异步执行的，所以，后续的读操作可以直接从缓存中获得这些数据，从而极大地提高了读操作的性能。

www.TopSage.com

稀疏文件

稀疏文件（NTFS的一种文件类型，相对于前面所描述的包含稀疏数据的文件）本质上是压缩的文件，NTFS并没有将压缩过程应用在该文件的非稀疏数据上。然而，NTFS管理稀疏文件MFT纪录的行串数据的做法，跟它管理由稀疏数据和非稀疏数据组成的压缩文件的做法是相同的。

变化日志文件

变化日志文件\$Extend\$UsnJrnl是一个稀疏文件，只有当一个应用程序激活（enable）了变化日志的记录功能时，NTFS才会创建该文件。该日志将变化项存储在$J数据流中。这些变化项包含了以下有关一个文件或目录变化的信息：

- 此次变化的时间；
- 变化的类型（删除、重命名、大小扩展，等等）；
- 文件或目录的属性；
- 文件或目录的名称；
- 文件或目录的文件引用号；
- 文件的父目录的文件引用号。

该日志是稀疏的，因而，它永远不会溢出。当该日志在磁盘上的空间大小超过了针对该文件定义的最大值时，NTFS的做法很简单，它把一个变化信息窗口（其大小等于最大日志尺寸）之前的文件数据都变成零，如图12.41所示。当一个应用程序不断地导致超越该日志的大小时，为了避免频繁地改变大小，NTFS只有当该日志的大小超过了最大配置值再加上一个应用程序定义的值的两倍时，才会缩减该日志。

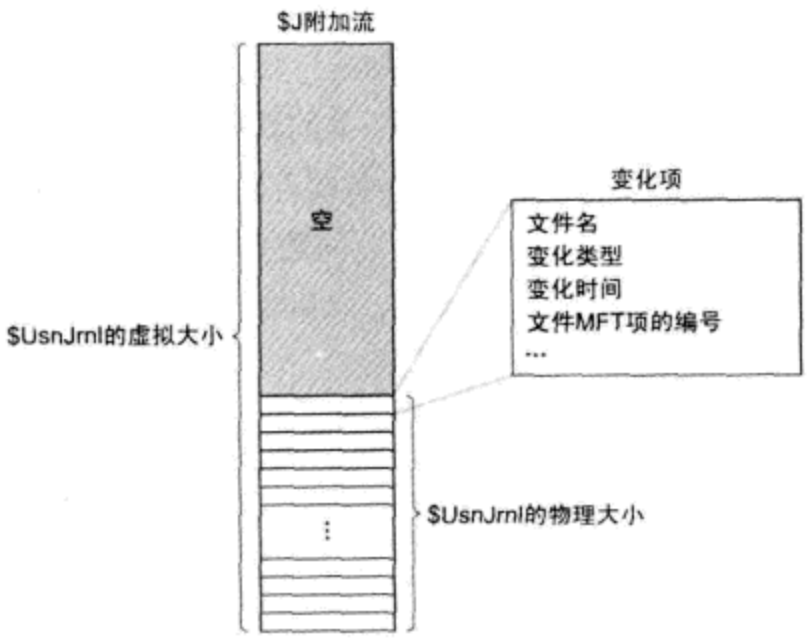


图12.41 变化日志（$UsnJrnl）的空间分配

www.TopSage.com

索引

在NTFS中，文件目录只不过是一组文件名的索引而已。也就是说，一组文件名（以及它们的文件引用）按照一种特殊的方式组织起来以便于快速访问。为了创建一个目录，NTFS对于该目录中的文件的文件名属性进行了索引。一个卷的根目录的MFT纪录如图12.42所示。

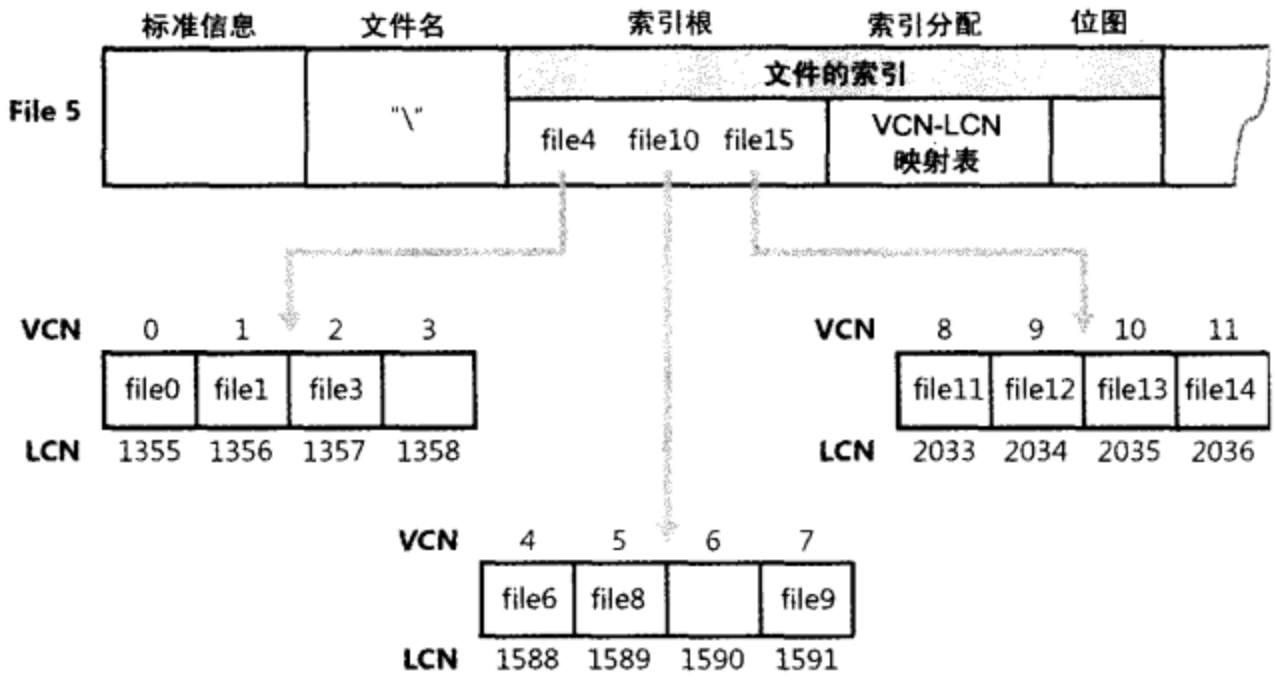


图12.42 一个卷的根目录的文件名索引

从概念上讲，一个目录的MFT项在其索引根属性中包含了该目录中的文件的一个排序列表。然而，对于大的目录，这些文件名实际上被存储在一些4KB固定大小的索引缓冲区中，这些缓冲区真正包含和组织这些文件名。索引缓冲区实现了一个*b+*树数据结构，这使得在查找一个特定的文件时所需要的磁盘访问次数尽可能地最少，尤其对于大的目录。索引根属性包含b+树（根子目录）的第一层，并指向下一层节点（更多的子目录，可能还有文件）所在的索引缓冲区。

图12.42只显示了索引根属性中的文件名和索引缓冲区中的文件名（例如*file6*），但是，索引中的每一项也包含了该文件所对应的MFT中的文件引用，以及该文件的时间戳和文件大小的信息。NTFS从该文件的MFT纪录中复制了时间戳和文件大小的信息。FAT和NTFS都使用了这项技术，它要求把更新的信息写到两个地方。即便如此，它对于目录浏览操作是一个重要的速度优化，因为这使得文件系统无需打开目录中的每一个文件就可以显示每个文件的时间戳和大小。

www.TopSage.com

索引分配（index allocation）属性将这些索引缓冲区行串的VCN映射到LCN，因而通过这些LCN可以知道索引缓冲区驻留在磁盘上的什么地方；而位图属性则记录了在索引缓冲区中哪些VCN已经被使用了，哪些VCN还是空闲的。图12.42显示了每个VCN（即每一簇）一个文件项，但实际上，多个文件名项被包装起来后放到每一簇中。每个4KB索引缓冲区可以包含大约20至30个文件名项。

b+树数据结构是一种平衡树；要将排序的数据组织到磁盘上，它是非常理想的，因为它使得查找一个数据项所需要的磁盘访问次数最少化。在MFT中，一个目录的索引根属性包含多个文件名，这些文件名又被看作b+树第二层的索引。索引根属性中的每个文件名有一个可选的、与之关联的指针，指向一个索引缓冲区。它所指的索引缓冲区中包含的文件名，均小于它自己的文件名（按字典序）。例如，在图12.42中，*file4*是b+树中的第一层节点项。它指向的索引缓冲区中的文件名都小于（按字典序）它自己的文件名——文件名*file0*、*file1*和*file3*。注意，此例子中使用的文件名*file1*、*file2*等并不是字面意义上的文件名，但是，如此命名的意图是，这些文件的字典顺序与它们的相对摆放位置是一致的。

将文件名存储在b+树中，这带来了几方面的好处。首先，目录查询非常快，因为文件名是按照排序的顺序来存储的。其次，当高层软件列举一个目录中的文件时，NTFS返回已经排序的名称。最后，因为b+树倾向于往宽度增长，而不是向深度增长，所以，NTFS的快速查找时间并不会随着目录的增长而变慢。

NTFS也提供了索引其他数据（非文件名）的一般性支持，而且，NTFS的几项特性——包括对象ID、配额跟踪和一致的安全性——使用索引机制来管理内部数据。

对象ID

NTFS除了把分配给一个文件或目录的对象ID存储在该文件或目录的MFT纪录的$OBJECT_ID属性中以外，还在\$Extend\$ObjId元数据文件的$O索引中记录了对象ID和它们的文件引用号之间的对应关系。此索引根据对象ID来排列其中的项，从而使得NTFS可以快速地根据一个文件的ID来找到它。这一特性使得应用程序可以使用一个文件或目录的对象ID来打开它（通过未文档化的原生API功能可以做到）。图12.43示出了$Objid元数据文件和MFT纪录中的$OBJECT_ID属性之间的对应关系。

www.TopSage.com

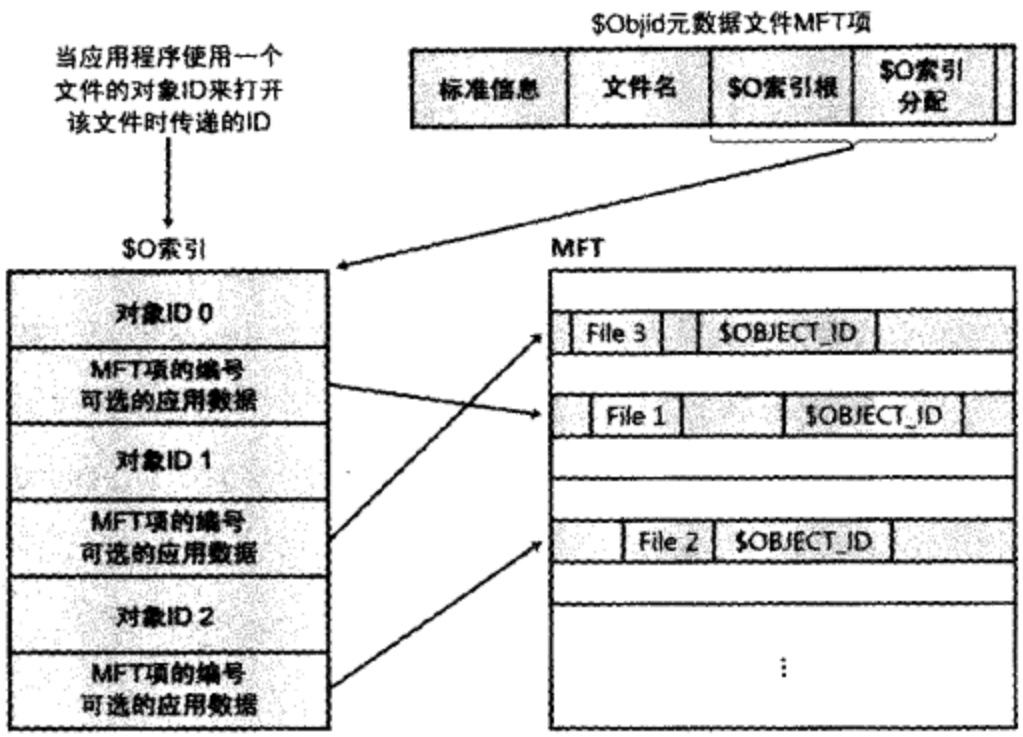


图12.43 $Objid和$OBJECT_ID之间的关系

配额跟踪

NTFS将配额信息存储在\$Extend\$Quota元数据文件中，此元数据文件是由索引$O和$Q组成的。图12.44显示了这些索引的组织结构。正如NTFS为每个安全描述符分配一个惟一的内部安全ID一样，NTFS给每个用户分配一个惟一的用户ID。当管理员为一个用户定义配额信息时，NTFS分配一个用户ID，它对应于该用户的SID。在$O索引中，NTFS创建一个从SID到用户ID的映射项，并按照用户ID对该索引进行排序；在$Q索引中，NTFS创建一个配额控制项。配额控制项包含了该用户的配额限制值，以及该用户在该卷上已经使用的磁盘空间数量。

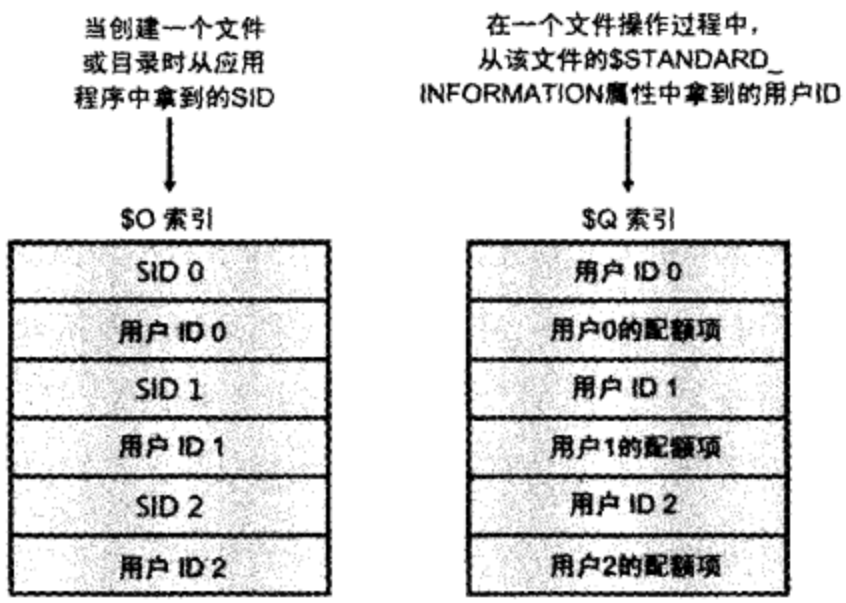


图12.44 $Quota索引

www.TopSage.com

当一个应用程序创建一个文件或目录时，NTFS获得该应用程序用户的SID，并且在$O索引中查找与该SID相关联的用户ID。NTFS在新文件或目录的$STANDARD_INFORMATION属性中记录下此用户ID，这也使得所有分配给该文件或目录的磁盘空间都计算在这个用户的配额中。然后，NTFS在$Q索引中查找该用户的配额项，以确定这次新的空间分配是否会导致该用户的磁盘空间使用量超过他或她的警告或限制阈值。当一次新的空间分配导致用户超出某个阈值时，NTFS采取适当的步骤，比如在系统事件日志中记录一个事件，或者不允许用户创建该文件或目录。当一个文件或目录改变大小时，NTFS也会更新与在$STANDARD_INFORMATION中保存的用户ID相关联的配额控制项。NTFS使用通用索引技术，以便很有效地将用户ID与账户SID关联起来，并且，给定一个用户ID，可以高效地找到一个用户的配额控制信息。

统一的安全性

NTFS一直以来都支持安全性，这使得管理员可以指定哪些用户能够或者不能够访问单独的文件和目录。在Windows 2000之前的NTFS，每个文件和目录将它的安全描述符存放在其自己的安全属性中。在绝大多数情况下，管理员会把同样的安全设置应用在一个完整的目录树上，这使得此安全设置所适用的所有文件和子目录中存在大量重复的安全描述符。在一个多用户环境中，这种重复性可能会显著地消耗磁盘空间，比如，在Windows 2000 Server终端服务环境下，安全描述符中可能包含很多个账户的访问信息。Windows 2000及其以后版本的NTFS对于安全描述符的磁盘使用作了优化，其做法是，用一个名为$Secure的中心元数据文件来存储该卷上每一个安全描述符的实例（每个安全描述符仅保存一份实例）。

$Secure文件包含两个索引属性即$SDH和$SII，以及一个名为$SDS的数据流属性，如图12.45所示。NTFS为一个卷上的每一个惟一的安全描述符分配一个内部NTFS安全ID（不要跟SID混淆，SID惟一地标识了计算机和用户的账户），并且根据一个简单的散列算法计算该安全描述符的散列值。此散列值是该描述符的一个简略表示形式，它可能不是惟一的。$SDH索引中的每一项把安全描述符散列值映射到该安全描述符在$SDS数据属性内部的存储位置上；$SII索引项把NTFS安全ID映射到该安全描述符在$SDS数据属性内部的存储位置上。

www.TopSage.com

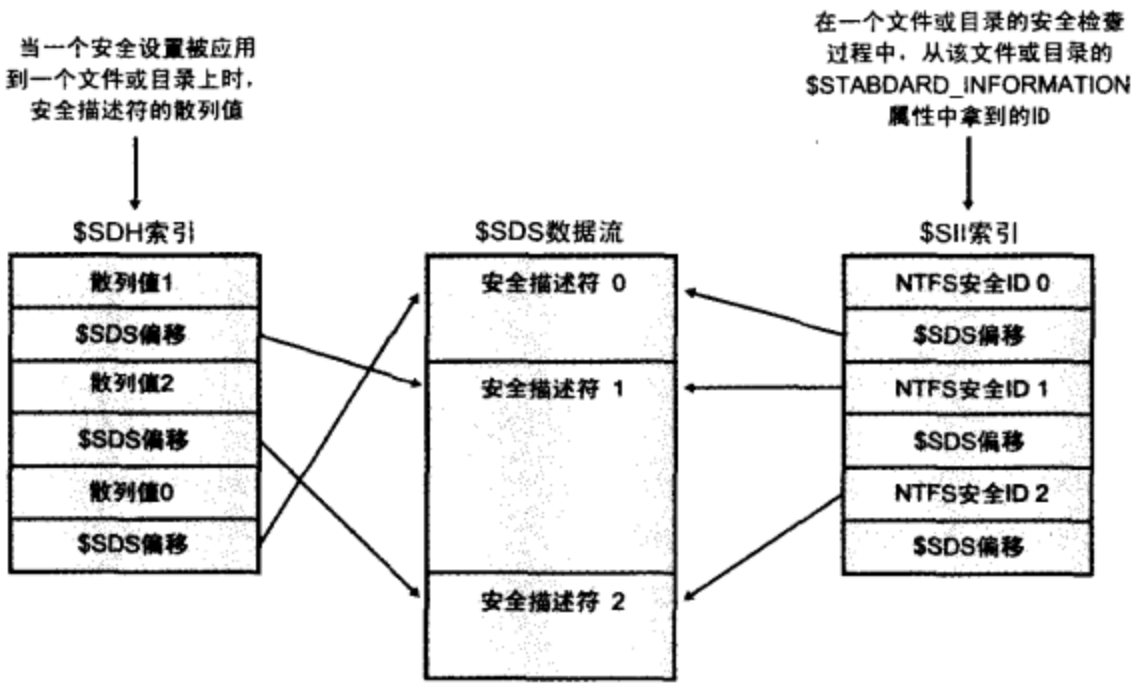


图12.45 $Secure索引

当你将一个安全描述符应用到一个文件或目录上时，NTFS获得该描述符的散列值，然后在$SDH索引中查找匹配的项。NTFS根据安全描述符的散列值来排列$SDH索引项的顺序，并且将这些索引项存储在一个b+树中。如果NTFS在$SDH索引中找到了该描述符的匹配项，则从该匹配项的偏移值中获得该匹配项的安全描述符的偏移量，并且从$SDS属性中读取到该安全描述符。如果散列值匹配了，但是安全描述符不匹配，那么，NTFS在$SDH索引中查找另一个匹配项。当NTFS找到一个精确的匹配项时，你当前正在操作的文件或目录就可以引用$SDS属性中这一现成的安全描述符了。NTFS实现这一引用的做法很简单，只需从$SDH的项中读取NTFS安全标识符，然后将它保存到该文件或目录的$STANDARD_INFORMATION属性中即可。在NTFS中，所有的文件和目录都有$STANDARD_INFORMATION属性，它存储了有关一个文件的基本信息，包括它的属性、时间戳信息，以及文件的安全标识符。

如果NTFS在$SDH索引中没有找到任何一项与你当前正在应用的安全描述符相匹配，那么，你正在应用的描述符对于该卷来说还是惟一的，于是NTFS为该描述符分配一个新的内部安全ID。NTFS内部安全ID是32位值，而SID通常是它的几倍长，所以，用NTFS安全ID来代表SID可以节省$STANDARD_INFORMATION属性的空间。NTFS把该安全描述符加入到$SDS属性中，$SDS属性被按照NTFS安全ID的顺序组织成一颗b+树，然后它在$SDH和$SII索引中分别加入相应的项，让它们引用该描述符在$SDS数据属性中的偏移量。

当一个应用程序试图打开一个文件或目录时，NTFS使用$SII索引来查找该文件或目录的安全描述符。NTFS从MFT项的$STANDARD_INFORMATION属性中读取到该文件或目录的内部安全ID。然后，它使用$Secure的$SII索引来找到该ID在$SDS属性中的项。由于得到了该描述

www.TopSage.com

符在$SDS属性中的偏移量，因此NTFS可以读取到安全描述符，并完成安全检查。NTFS将最近访问过的32个安全描述符以及它们的$SII索引，存储在一个缓存中，所以，只有当一个$SII索引未被缓存时，NTFS才需要访问$Secure文件。

NTFS并不删除$Secure文件中的数据项，即使该卷上不再有文件或目录引用一个数据项。不删除这些数据项并不会严重地减少磁盘空间，因为绝大多数卷，即使那些已经使用了很长时间的卷，也只有相对少量的惟一安全描述符。

NTFS使用通用索引技术，这使得那些具有相同安全设置的文件和目录可以非常高效地共享安全描述符。$SII索引使得NTFS在执行安全检查时可以快速地在$Secure文件中查找一个安全描述符，而$SDH索引使得NTFS可以快速地确定，一个正要被应用于某个文件或目录上的安全描述符是否已经存储在$Secure文件中，以及是否能被共享。

重解析点

本章前面已讲述，**重解析点**（*reparse point*）是一个可达16KB大小的、由应用程序定义的重解析数据块，以及一个32位的重解析标记，它们被存储在一个文件或目录的$REPARSE_POINT属性中。无论何时，当一个应用程序创建或删除一个重解析点时，NTFS都会更新\$Extend\$Reparse元数据文件，该元数据文件中存储的数据项标识了那些“内含重解析点的文件和目录”的文件纪录号。以中心化的方式来存储这些纪录，这使得NTFS能够为应用程序提供接口，以列举一个卷的所有重解析点，或者仅仅某些特定类型的重解析点，比如挂载点（mount point）。（有关挂载点的更多信息，请参见第10章。） \$Extend\$Reparse文件使用了NTFS的通用索引设施，它通过重解析点标记来核查一个文件的项（在一个名为$R的索引中）。

12.7 NTFS的恢复支持

NTFS的恢复支持可确保：如果发生电源失败或者系统失败，文件系统操作（事务）将不会遗留在未完成的状态，而且，磁盘卷的结构仍然完好无损，无需运行磁盘修复工具。NTFS的Chkdsk工具可被用来修复由于I/O错误（例如，坏磁盘扇区、电气方面的异常，或磁盘失败）或者软件错误（software bug）而引起的灾难性磁盘损坏。但是，NTFS具备了恢复能力以后，Chkdsk很少再需要了。

正如前面所提到的（在“可恢复性”一节中），NTFS使用一种事务-处理方案来实现可恢复性。这种策略确保即使对于非常大的磁盘，整个磁盘的恢复也会绝对快速（在秒的数量级内完成）。NTFS的恢复过程仅限于对文件系统数据，其最起码的目的是为了保证：用户不会由于文件系统的破坏而丢失一个卷。然而，除非应用程序采取特别的动作（比如把被缓存的文件刷新到磁盘上），否则NTFS不会保证：若发生崩溃的话，用户数据是完全更新的。

www.TopSage.com

在Windows平台上可以通过大多数数据库产品（比如Microsoft SQL Server）来获得对用户数据的事务方式保护。Microsoft之所以选择不在文件系统层次上实现用户数据的恢复支持，是因为应用程序通常实现了它们自己的数据恢复方案，这些方案针对它们所存储的数据的类型做了特别的定制。

下面的小节描述了文件系统可靠性的演变（以此作为可恢复文件系统的一个总体介绍），详细地讨论了NTFS用来记录文件系统数据结构修改信息的事务记录方案，还解释了当系统失败时NTFS如何恢复一个卷。

文件系统设计的演变

可恢复文件系统的发展是文件系统设计演进历程中的一大步伐。在过去，在构建一个文件系统的I/O和缓存支持时，有两种技术是常用的：**谨慎写**（*careful write*）和**延迟写**（*lazy write*）。DEC（Digital Equipment Corporation，现在是Compaq）、 VAX/VMA和其他一些私有操作系统的文件系统采用了一种谨慎写的算法，而OS/2 HPFS和绝大多数老的UNIX文件系统则使用了一种延迟写的文件系统方案。

下面的两个小节简要地回顾了这两种类型的文件系统，以及它们在保险性（safety）和性能方面所做的一些内在的取舍。随后的小节将介绍NTFS的恢复方法，并解释了它与其他两种策略之间的差异。

谨慎写（Careful Write）文件系统

当一个操作系统崩溃或者掉电时，正在进行中的I/O操作立即被中断，而且通常是永久性的中断。根据这些正在进行中的操作的类型，以及它们的进展程度，这样的突然停止有可能会导致文件系统不一致。在这种环境下的不一致是一种文件系统损坏。例如，某个文件名出现在目录列表中，但是，文件系统并不知道该文件在哪儿，或者根本不能访问该文件。最严重的文件系统损坏可能导致整个卷无法访问。

谨慎写文件系统并不试图阻止这种文件系统不一致。相反地，它对它的写操作进行排序，因而，即使在最差的情况下，系统崩溃也只会产生一些可预见的、非关键性的不一致，而对于这种不一致，文件系统可以在空闲的时候进行修复。

任何种类的文件系统在接收到一个更新磁盘的请求时，必须先执行几个子操作，然后此更新操作才真正完成。在一个使用了谨慎写策略的文件系统中，这些子操作总是被顺序地写到磁盘上。例如，在为一个文件分配磁盘空间时，文件系统首先在它的位图中设置某些位，然后将磁盘空间分配给该文件。如果当这些位刚刚被设置上的时候电源就掉电了，那么，谨慎写文件系统就丢失了对某些磁盘空间的访问权——即这些刚刚被设置的位所代表的空间——但是原有的数据并未遭到破坏。

对写操作进行序列化，也意味着I/O请求是按照其接收到的顺序被填充的。如果一个进程请求分配磁盘空间，很快地另一个进程要创建一个文件，那么，谨慎写文件系统在开始创建文件以前先完成磁盘分配操作，因为交叉地处理两个I/O请求的子操作可能会导致不一致的状态。

www.TopSage.com



> **注** FAT文件系统使用了一种“*直写（write-through）*”算法，这使得对磁盘的修改能够立即被写到磁盘上。与谨慎写的做法不同的是，直写技术并不要求文件系统对它的写操作进行排序来防止不一致。

谨慎写文件系统的主要优势在于，即使系统失败，整个卷仍然处于一致的和可用的状态，无需立即运行一个慢速的卷修复工具。为了纠正那些由于系统失败而导致可预见的、非毁灭性的磁盘不一致，这样的修复工具是必要的。该工具可以在方便的时候运行，通常是在系统重新引导的时候运行。

延迟写文件系统

谨慎写文件系统以牺牲速度来换取它所提供的保险性（safety）。延迟写文件系统利用一种“**回写**（*write-back*）”缓存策略来提高性能。也就是说，它把文件的修改写到缓存中，然后以一种优化的方式将缓存中的内容刷新到磁盘上。它通常是以后台活动的方式来进行的。

与延迟写的缓存技术相关联的性能提高有几种形式。第一，总的磁盘写的次数减少了。因为现在不需要顺序化的、直接的磁盘写操作，所以，一个缓冲区的内容在写到磁盘上以前可以被多次修改。第二，处理应用程序请求的速度可以大大提高。因为文件系统无需等待磁盘写操作完成，就可以将控制权返回给调用者。第三，延迟写策略忽略了一个文件卷上的不一致的中间状态（当两个或多个I/O请求的子操作被交叉处理时，这种不一致的中间状态是有可能发生的）。因此，文件系统很容易被做成多线程模型，从而使得多个I/O操作可以同时进行。

延迟写技术的缺点是：它制造了一些危险间隔，一个卷在这些间隔中处于太不一致的状态，因而文件系统可能无法纠正这种不一致。因此，延迟写文件系统必须自始至终跟踪该卷的状态。一般而言，延迟写文件系统相比较于谨慎写系统，还是获得了性能上的好处。但其代价是要冒更大的风险，如果系统失败了，用户会很不方便。

可恢复的文件系统

像NTFS这样的可恢复文件系统试图既要超过谨慎写文件系统的保险性（safety），同时也要获得延迟写文件系统的性能优势。可恢复文件系统使用最初为事务处理而发展起来的**日志技术**（logging technique，有时候也称为*journaling*）来确保卷的一致性。如果操作系统崩溃了，则可恢复文件系统通过执行一个恢复过程来恢复其一致性，并且，此恢复过程需要访问日志文件中所存储的信息。因为文件系统已经记录了它的磁盘写操作，所以，恢复过程只需要几秒钟时间，跟卷的大小无关。

www.TopSage.com

一个可恢复文件系统的恢复过程是非常精确的，它要保证该卷被恢复到一致的状态。在NTFS中，不会发生与延迟写文件系统相关的不完全恢复。

为了提供这种保险性，可恢复文件系统也招致一些开销。每一个修改卷结构的事务都要求在日志文件中，为该事务的每一个子操作写入一个纪录。这一写纪录的开销可以通过该文件系统的“日志纪录**批处理**（*batching* of log records）”来加以改进。也就是说，用单个I/O操作将多条纪录写到日志文件中。而且，可恢复文件系统可以采用延迟写文件系统的优化技术。它甚至可以加大缓存、刷新之间的间隔长度，因为如果在缓存变化被刷新到磁盘上以前系统崩溃的话，文件系统是可以被恢复的。这种建立在“延迟写文件系统的缓存性能”之上的收益，补偿了（通常也超过了）可恢复文件系统的日志记录开销。

无论是谨慎写文件系统，还是延迟写文件系统，都无法承诺对用户文件数据的保护。如果当一个应用程序正在写一个文件的时候系统崩溃了，那么，该文件可能丢失或破坏。更糟的是，这种系统崩溃可能会破坏一个延迟写文件系统，毁坏已有的文件，或者导致整个卷变得不可访问。

NTFS可恢复文件系统实现了几个策略，以便在传统文件系统的基础上进一步改进它的可靠性。第一，NTFS的可恢复性确保其卷结构不会被破坏，因此，在系统失败以后所有的文件仍然是可访问的。

第二，尽管NTFS并不承诺对用户数据在系统崩溃情况下的保护（有些变化可能会从缓存中丢失），但是，应用程序可以利用NTFS的直写（write-through）和缓存刷新能力，来确保文件的修改能够以适当的间隔被记录到磁盘上。文件系统的**缓存直写**（*cache write-through*，即强迫写操作被立即记录到磁盘上）和**缓存刷新**（*cache flushing*，强迫缓存的内容被写到磁盘上）这两种操作都是非常高效的。NTFS不必做额外的磁盘I/O来把对文件系统的修改刷新到几个不同的文件系统数据结构中，因为这些数据结构的变化都被记录到日志文件中了，而且是在单个写操作中进行的。如果系统失败并且丢失了缓存内容的话，则可以从日志中恢复对文件系统的修改。而且，与FAT文件系统不同的是，NTFS保证：在一次直写操作或者缓存刷新以后，用户数据将是一致的，并且立即可以使用，即使系统后来失败了也是如此。

日志记录

NTFS通过一种称为**日志记录**（*logging*）的事务处理技术来提供文件系统的可恢复能力。在NTFS的日志记录技术中，如果一个事务需要修改重要的文件系统数据结构，那么，它的所有子操作在真正被作用到磁盘上以前，都事先被记录在一个日志文件中。通过这种方法，如果系统崩溃了，则当系统又回到正常活动状态时，部分已经完成的事务可以被重新做一遍（redo），或者被撤销（undo）。在事务处理过程中，这种技术称为**预写日志记录**（*write-ahead logging*）。NTFS中的事务包括往磁盘上写数据或者删除一个文件，它们可以是由几个子操作构成的。

www.TopSage.com

日志文件服务（LFS）

日志文件服务（LFS，log file service）是指NTFS驱动程序内部的、用于访问日志文件的一系列内核模式例程。虽然最初设计LFS的目的是为多个客户提供日志记录和恢复服务，但是，LFS仅仅被NTFS使用。在这种情况下，NTFS是调用者，它传递给LFS一个指针，指向一个已打开的文件对象，该指针指定了要访问的日志文件。LFS要么初始化一个新的日志文件，要么调用Windows的缓存管理器以便通过缓存来访问已有的日志文件，如图12.46所示。

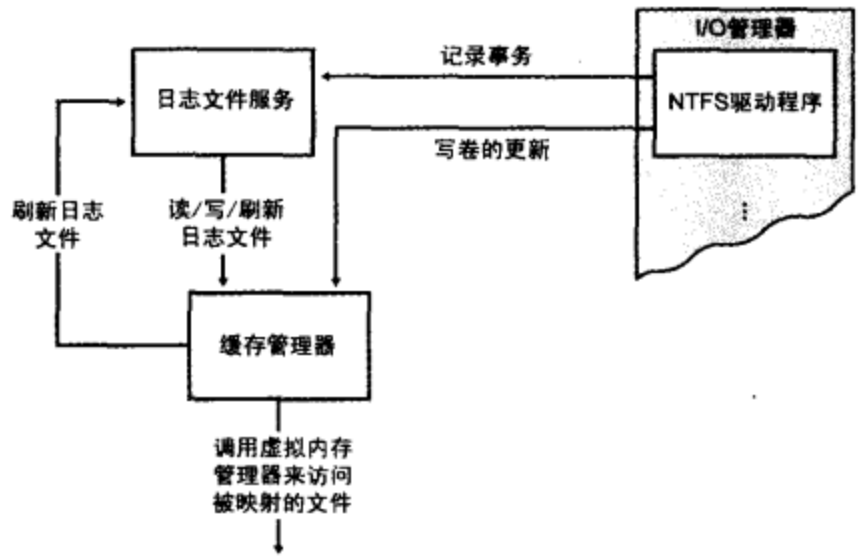


图12.46 日志文件服务（LFS）

LFS将日志文件分成两个区域：**重启动区域**和一个“无限的”**日志记录区域**，如图12.47所示。

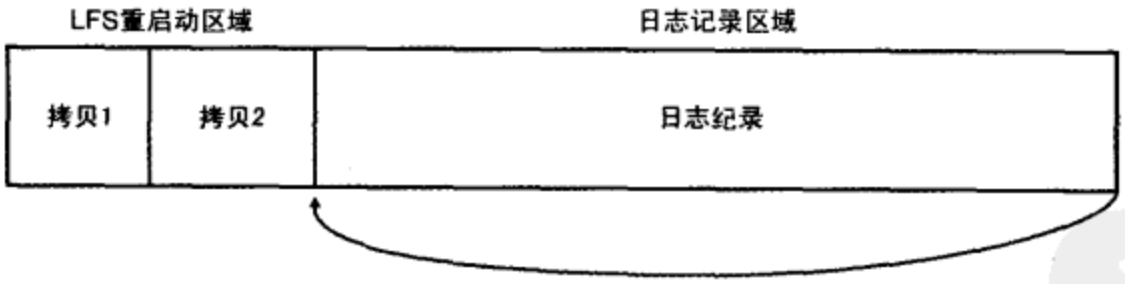


图12.47 日志文件区域

NTFS调用LFS来读和写重启动区域。NTFS使用重启动区域来存储一些环境信息，比如在一次系统失败以后NTFS在恢复过程中从哪个位置开始读取日志记录区域。LFS为重启动数据维护了第二份拷贝，以防万一第一份拷贝被破坏或者无法访问。日志文件的剩余部分是日志记录区域，其中包含了NTFS写入的事务纪录，因而当系统失败时，NTFS可以利用这些事务纪录来恢复一个卷。LFS通过循环地重复使用该日志文件（同时也保证，它不会覆盖掉所需要的信息），从而使得该日志文件看起来像是无限可增长似的。LFS使用**逻辑序列号**（LSN，*logical*

www.TopSage.com

sequence number）来标识这些被写到日志文件中的纪录。随着LFS循环地访问该文件，它相应地增加LSN的值。NTFS使用64位来表达LSN，所以，可能赋予的LSN值的范围是如此之大，以至于可以被认为是无限的。

NTFS从来不直接从日志文件中读取事务，也不直接写事务到日志文件中。LFS提供了相应的服务，NTFS调用这些服务来打开日志文件、写日志纪录、按向前或向后的顺序读取日志纪录、刷新日志纪录直至某个特定的LSN，或者将日志文件的开始点设置为某一个更高的LSN。在恢复过程中，NTFS调用LFS来执行以下的动作：向前读取日志纪录，以便重新完成任何“已经被记录在日志文件中，但是在系统失败时尚未来得及刷新到磁盘上”的事务；向后读取日志纪录，以便撤销或回滚任何“在崩溃之前尚未完全记录下来的”事务；当NTFS不再需要该日志文件中老的事务纪录时，将该日志文件的开始点设置成一个具有更高LSN的纪录。

系统如何保证一个卷总能够被恢复？以下是NTFS的做法。

1. NTFS首先调用LFS，将任何需要修改卷结构的事务记录到（被缓存的）日志文件中。
2. NTFS修改该卷（也可能在缓存中）。
3. 缓存管理器提示LFS刷新日志文件到磁盘上（LFS实现刷新的做法是，回调到缓存管理器，告诉它哪些内存页面需要刷新。前面的图12.46中显示了此调用序列）。
4. 缓存管理将日志文件刷新到磁盘上以后，它再将卷的变化（元数据操作本身）刷新到磁盘上。

采取这些步骤可以确保：如果文件系统的修改最终未能成功完成的话，那么，在文件系统恢复过程中，对应的事务可以从日志文件中获得，并且，这些事务要么被重新完成，要么被撤销。

文件系统的恢复是在系统重新引导以后第一次使用一个卷的时候自动开始的。NTFS首先做检查：在系统崩溃以前被记录在日志文件中的事务是否已经被应用到该卷上，如果它们尚未被应用，则重新执行这些事务。NTFS也保证，在系统崩溃以前没有被完整地记录下来的事务会被撤销，因而它们不会出现在卷上。

日志纪录类型

LFS允许它的客户将任何种类的纪录写到他们的日志文件中。NTFS写入几种类型的纪录。这里描述两种类型：**更新纪录**（*update record*）和**检查点纪录**（*checkpoint record*）。

更新纪录 在NTFS所写的日志文件纪录之中，更新纪录是最常见的纪录类型。每一条更新纪录包含以下两种信息。

www.TopSage.com

- **重做（redo）信息** 若一个事务已被完整记录下来（“已提交”），但是在该事务被从缓存刷新到磁盘上以前系统失败了，则此信息描述了如何将该事务的一个子操作重新应用到该卷上。
- **撤销（undo）信息** 若一个事务在系统失败时只被部分记录下来（“未被提交”），则此信息描述了如何逆转该事务的一个子操作。

图12.48显示了在日志文件中有三条更新纪录。这里的事务是创建一个新文件，每条纪录代表了该事务的一个子操作。每条更新纪录中的Redo项向NTFS指明了应该如何将此子操作重新应用到该卷上，而Undo项则向NTFS指明了应该如何回滚（撤销）该子操作。

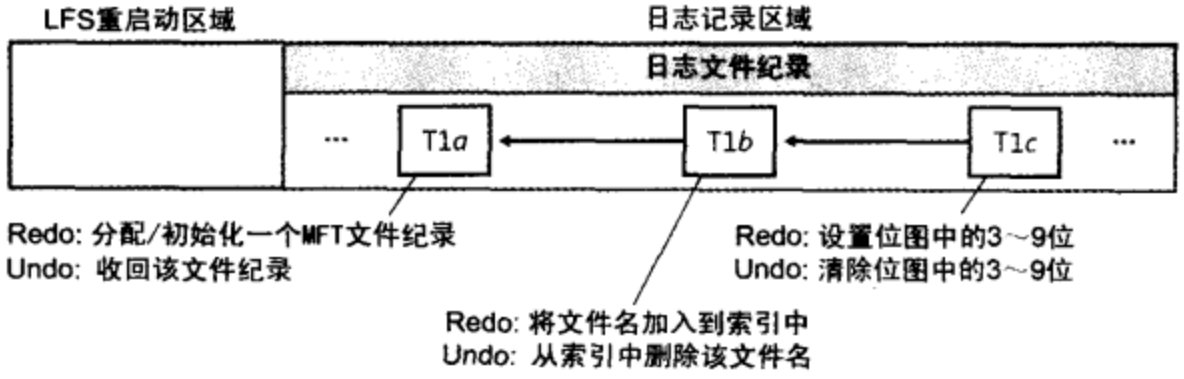


图12.48 日志文件中的更新纪录

在记录了一个事务（在这个例子中，通过调用LFS将三条更新纪录写到日志文件中）以后，NTFS在该卷上执行这些子操作（在缓存中）。当NTFS完成了对缓存的更新以后，它在日志文件中写入另一条纪录，记录下整个事务已经完成——此子操作被称为**提交**（*committing*）一个事务。一旦一个事务已被提交， NTFS就保证这整个事务一定会出现在该卷上，即使后来操作系统失败了。

若系统失败了，那么，当NTFS执行恢复时，它读取此日志文件，并且重做每一个已提交的事务。尽管NTFS在系统失败以前已经完成了这些已提交的事务，但是，它并不知道缓存管理器是否已经及时地把对卷的修改刷新到磁盘上了。当系统失败时，这些更新信息可能已经从缓存中丢失了。因此，NTFS只是简单地再次执行这些已提交的事务，以确保磁盘是最新的状态。

在文件系统恢复过程中，当NTFS重做了已提交的事务以后，它在日志文件中找到那些在失败时刻尚未被提交的所有事务，并且将已被记录下来的每个子操作执行其回滚动作（即撤销）。在图12.48中，NTFS将首先撤销T1*c*子操作，然后沿着向后的指针找到T1*b*，并撤销该子操作。它继续沿着向后的指针，撤销这些子操作，直至到达该事务的第一个子操作。NTFS通过沿着这些指针可以知道，它必须撤销多少个更新纪录以及哪些更新纪录，才能回滚一个事务。

www.TopSage.com

重做和撤销信息既可以按物理来解释，也可以按逻辑来解释。物理的描述指定了“一块磁盘上要被改变或移动（或其他处理方式）的特定字节范围”，以此来指定卷的更新。逻辑的描述则通过诸如“删除文件A.dat”之类的操作来解释卷的更新。NTFS作为维护文件系统结构的最底层软件，它用物理描述方式来写更新纪录。然而，对于事务处理或者其他的应用层软件，则可能用逻辑的方式来写更新纪录更有好处，因为用逻辑方式表达的更新纪录比用物理方式表达的更新纪录更为简洁、紧凑。这些逻辑的描述不可避免地要依赖于NTFS，才能理解所涉及的操作（比如删除一个文件）。

NTFS为以下每一个事务写相应的更新纪录（通常有几条）：

- 创建一个文件；
- 删除一个文件；
- 扩充一个文件；
- 截短一个文件；
- 设置文件信息；
- 重命名一个文件；
- 改变一个文件的安全性。

一个更新纪录中的重做和撤销信息必须经过精心设计，因为尽管NTFS可以撤销一个事务，可以从系统失败中恢复过来，抑或正常地运行，但它也可能会试着重做一个已经完成了的事务，或者撤销一个从未发生过的或已经被撤销了的事务。类似地，NTFS可能会试着重做或撤销一个由几条更新纪录构成的事务，并且这些纪录在磁盘上只有一部分是完整的。更新纪录的格式必须确保，执行多余的重做或撤销操作是**幂等的**（*idempotent*），也就是说，没有累积效应。例如，设置一个已经被设置的位不会有任何影响，但翻转一个已经被翻转的位则有影响。文件系统必须正确地处理中间的卷状态。

检查点纪录 除了更新纪录以外，NTFS还周期性地在日志文件中写入一个检查点纪录，如图12.49所示。



图12.49 日志文件中的检查点纪录

www.TopSage.com

检查点纪录帮助NTFS做出这样的决定：如果系统立即崩溃的话，那么，哪些处理过程对于恢复一个卷是必需的。NTFS利用检查点纪录中存储的信息，就可以知道诸如“在日志文件中必须往回走多远后再开始其恢复过程”这样的信息。NTFS写了一个检查点纪录后，将该纪录的LSN存储在重启动区域中，这样，在系统崩溃以后，当NTFS开始其文件系统恢复过程时，它可以快速地找到最近写入的检查点纪录。

尽管LFS展示给NTFS的是一个看似无限大的日志文件，但实际上它并不是这样的。日志文件需要记录相对大量的内容，以及频繁地写入检查点纪录（此操作通常会释放日志文件中的空间），这两者使得有可能会填满一个（远程的）日志文件。LFS通过跟踪以下一些数值来计算这种可能性：

- 当前可用的日志空间；
- 写一个进来的日志纪录和撤销这次写操作所需要的空间数量，这应该是必需的；
- 回滚所有活动的（未提交的）事务所需要的空间数量，这也应该是必需的。

如果日志文件中没有包含足够的可用空间来容纳上述后两项的总和，则LFS返回一个“日志文件已满”错误，NTFS会引发一个异常。NTFS的异常处理器回滚当前的事务，把它放到一个队列中，以便稍后重新执行。

为了释放日志文件中的空间，NTFS必须立即阻止进一步的文件事务。为了做到这一点，NTFS阻塞文件创建和删除操作，然后请求对于所有系统文件的独占访问，以及对于所有用户文件的共享访问。逐渐地，活动的事务要么成功地完成，要么收到“日志文件已满”的异常。NTFS将这些接收到此异常的事务回滚，并把它们放到队列中。

一旦NTFS按照上述描述的办法阻塞了文件的事务行为，它就调用缓存管理器，把尚未写出的数据刷新到磁盘上，其中也包括尚未写出的日志文件数据。在所有这些都已经被安全地刷新到磁盘上以后，NTFS不再需要日志文件中的数据了。它将日志文件的开始点设置到当前位置上，这使得该日志文件变成一个“空的”日志文件。然后，它重新执行那些刚刚被排队的事务。除了在I/O处理过程中有短暂的停顿以外，“日志文件已满”错误对于正在执行的程序没有任何影响。

这种情形也正好说明：NTFS不仅利用日志文件来完成文件系统恢复，而且也用于在正常操作过程中的错误恢复。在下一节中你将会看到更多有关错误恢复的内容。

www.TopSage.com

恢复

在系统被引导起来以后，当第一次有程序访问一个NTFS卷时，NTFS自动执行磁盘恢复（如果不需要做恢复，则此过程无关紧要）。此恢复过程依赖于NTFS在内存中维护的以下两张表。

- **事务表**（*transaction table*） 此表记录了那些已经被启动，但是尚未提交的事务。在恢复过程中，这些活动事务的子操作必须要从磁盘上去除；
- **脏页表**（*dirty page table*） 此表记录了在缓存中哪些页面包含了有关文件系统结构的修改信息（并且这些修改尚未被写到磁盘上）。在恢复过程中这些数据必须被刷新到磁盘上。

NTFS每隔5秒钟写一个检查点纪录到日志文件中。NTFS在写检查点纪录以前，调用LFS，将事务表和脏页表的当前内容存储到日志文件中。然后，NTFS在检查点纪录中记录下包含这两张表内容的日志纪录的LSN。若系统失败，则当NTFS开始执行恢复过程时，它调用LFS，找到包含最近检查点纪录的日志纪录，以及包含最近的事务表和脏页表内容的日志纪录。然后，它把这两张表拷贝到内存中。

在最后的检查点纪录的后面，日志文件通常还包含了更多的更新纪录。这些更新纪录代表了在“最后的检查点纪录被写到日志文件中”以后发生的卷修改操作。NTFS必须更新事务表和脏页表，使其包含这些操作。在NTFS更新了这些表以后，它利用这些表以及日志文件的内容来更新卷本身。

为了实现卷的恢复，NTFS扫描日志文件三遍，在第一遍扫描过程中将日志文件加载到内存中，以便使磁盘I/O最小化。每一遍都有如下的特定目的：

1. 分析。
2. 重做（redo）事务。
3. 撤销（undo）事务。

分析扫描（Analysis Pass）

在**分析扫描**过程中，如图12.50所示，NTFS从最后的检查点操作的开始处向前扫描日志文件，以便找到更新纪录，并使用这些更新纪录来更新它已经拷贝到内存中的事务表和脏页表。注意，在此图中，检查点操作在日志文件中存储了三个纪录，而这三个纪录中间有可能会夹杂一些更新纪录。因此，NTFS必须从检查点操作的开始处执行此扫描过程。

www.TopSage.com

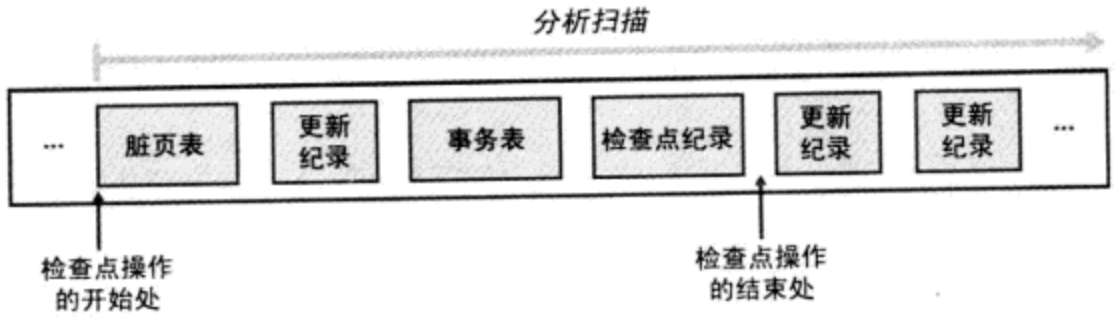


图12.50 分析扫描

在检查点操作开始以后出现在日志文件中的绝大多数更新纪录，都代表了对于事务表或脏页表的一个修改。例如，如果一个更新纪录是一个“事务已提交（transaction committed）”纪录，那么，必须将该纪录所代表的事务从事务表中去除。类似地，如果更新纪录是一个“页面更新（page update）”纪录，它修改了文件系统的一个数据结构，那么，脏页表必须被更新，以便反映出这一变化。

一旦这两张表在内存中已得到更新，NTFS就扫描这两张表，以确定最早的一个“记录了某一个尚未被写到磁盘上的操作”的更新纪录的LSN。事务表包含了尚未提交（未完成）的事务的LSN，而脏页表则包含了在缓存中尚未被刷新到磁盘上的纪录的LSN。NTFS在这两张表中找到的最早的更新纪录的LSN，并决定重做扫描（redo scan）该从哪里开始。然而，如果最后的检查点纪录比它还要早的话，则NTFS将从最后的检查点处开始重做扫描。

重做扫描（Redo Pass）

在**重做扫描**过程中，如图12.51所示，NTFS从最早的更新纪录的LSN（这是在分析扫描过程中找到的）开始，在日志文件中向前扫描。它寻找“页面更新”纪录，这些纪录中包含了在系统失败以前已经执行的卷修改操作，但它们可能尚未被刷新到磁盘上。NTFS在缓存中重做这些更新操作。

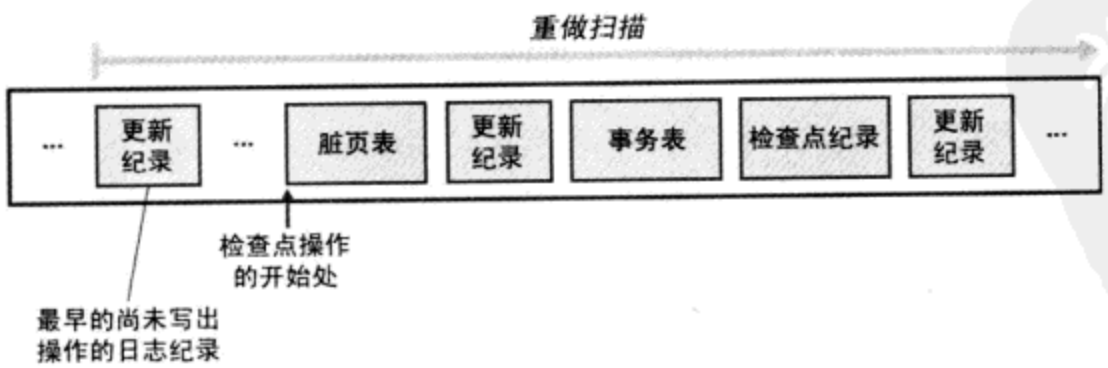


图12.51 重做扫描

当NTFS到达了日志文件的末尾时，它已经利用这些必要的卷修改信息更新了缓存，于是，缓存管理器的延迟写出器可以在后台把缓存的内容写到磁盘上。

www.TopSage.com

撤销扫描（Undo Pass）

在完成了重做扫描以后，NTFS开始其**撤销扫描**过程，在此过程中，它对于在系统失败时未提交的任何事务都执行回滚操作。图12.52显示了日志文件中的两个事务，其中事务1在电源失败以前被提交了，但事务2未能被提交。NTFS必须撤销事务2。

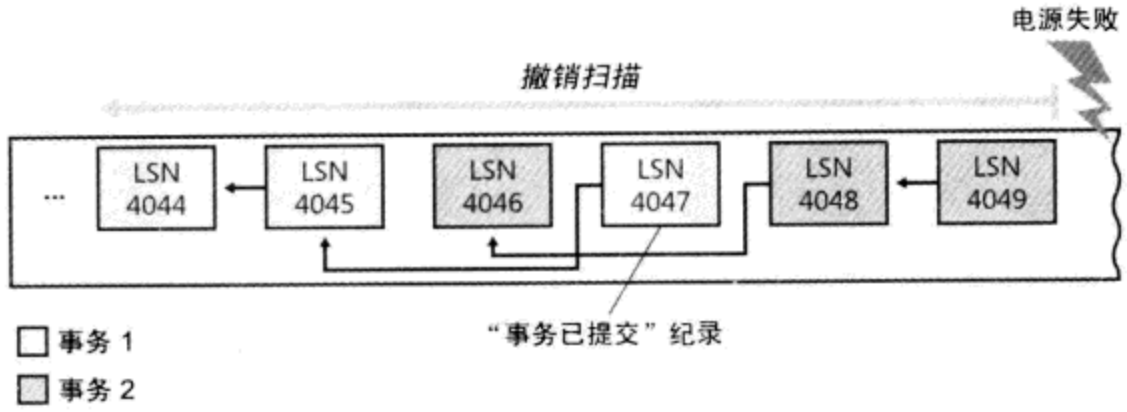


图12.52 撤销扫描

假设事务2创建一个文件，此操作是由三个子操作构成的，每个子操作都有其自己的更新纪录。在日志文件中，一个事务的多个更新纪录通过后向指针链接起来，因为它们往往不是连续存放的。

NTFS事务表列出了每个未提交事务的最后一个更新纪录的LSN。在这个例子中，事务表指明了LSN 4049是事务2的最后一个更新纪录。正如图12.53中从右至左所显示，NTFS回滚事务2。

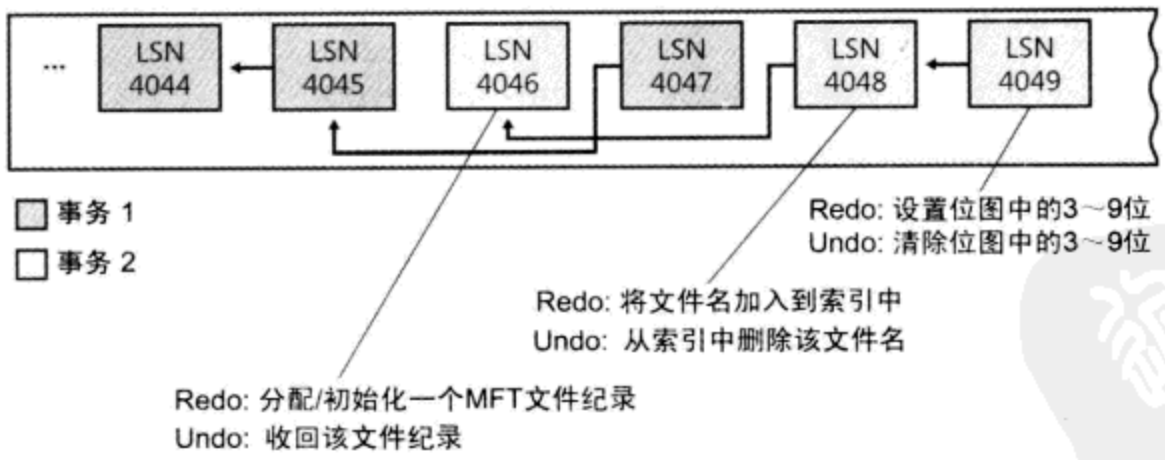


图12.53 撤销一个事务

在定位到LSN 4049以后，NTFS找到撤销信息，并执行撤销动作，清除其分配位图中的第3位至第9位。然后，NTFS沿着后向指针到达LSN 4048，此项LSN指示NTFS从适当的文件名索引中去除新的文件名。最后，它沿着最后的后向指针，并且根据LSN 4046中的更新纪录所指

www.TopSage.com

示的，收回为该文件而保留的MFT文件纪录。现在，事务2被回滚了。如果还有其他未提交的事务需要被回滚，则NTFS采用同样的过程将它们回滚。因为撤销事务会影响到一个卷的文件系统结构，所以，NTFS必须将撤销操作记录在日志文件中。毕竟，在恢复过程中电源可能会再次失败，如果这样的话，NTFS将不得不重做它的这些撤销操作。

在完成了恢复过程的撤销扫描以后，该卷已经被恢复到一致的状态。在这个点上，NTFS将缓存中的变化刷新到磁盘上，以确保该卷是最新的状态。然后，NTFS写一个“空的”LFS重启动区域，以表明该卷是一致的，如果系统立刻失败的话，无需再做恢复操作了。至此，恢复过程已经完成。

NTFS保证，恢复过程将使一个卷回到原先某个一致的状态。但NTFS不能保证，使一个卷正好回到系统失败以前的一致状态。因为，出于性能的考虑，它使用了一个“延迟提交”算法，这意味着，每次写一个“事务已提交”纪录时，日志文件并不立即被刷新到磁盘上。相反地，许多“事务已提交”纪录被批量处理，它们一起被写到磁盘上，或者是“当缓存管理器调用LFS，将日志文件刷新到磁盘上的时候”，或者是“当LFS将一个检查点纪录写到日志文件中（每5秒钟一次）的时候”。被恢复的卷可能无法恢复到最新状态的另一个原因是，当系统崩溃的时候，可能有几个事务同时是活动的（未提交的），并且有些事务的“事务已提交”纪录已经被写到磁盘上，而其他一些事务的“事务已提交”纪录可能还没有被写到磁盘上。恢复过程所得到的卷是一致的，凡是“事务已提交”纪录已被写到磁盘上的卷更新操作，都已经被包含在恢复卷上；而“事务已提交”纪录尚未被写到磁盘上的卷更新操作，无一出现在恢复卷上。

NTFS使用日志文件来恢复一个经历过系统失败的卷，但是，它也可以充分利用因为记录了事务而得来的一项重要的“免费赠品”。文件系统不可避免地要包含大量的、专用于恢复“在正常文件I/O操作过程中发生的文件系统错误”的代码。因为NTFS记录了每一个需要修改卷结构的事务，所以，当发生文件系统错误时，它可以使用日志文件来执行恢复，从而能够极大地简化它的错误处理代码。前面讲述的“文件日志已满”错误正是一个使用日志文件来执行错误恢复的例子。

一个程序接收到的绝大多数I/O错误并不是文件系统错误，因此不能完全由NTFS来解决。例如，当调用NTFS来创建一个文件时，NTFS可能先在MFT中创建一个文件纪录，然后在一个目录索引中输入此新文件的名称。然而，当它试图在位图中为该文件分配空间时，它可能发现磁盘已经满了，因此创建文件的请求未能完成。在这种情况下，NTFS使用日志文件中的信息来撤销它已经完成的那部分操作，收回它为该文件保留的数据结构。然后，它给调用者返回一个“磁盘已满”错误，而调用者必须采取适当的措施来响应此错误。

www.TopSage.com

NTFS的坏簇恢复

Windows中包含的卷管理器，即FtDisk（针对基本磁盘）和逻辑磁盘管理器（LDM，Logical Disk Manager，针对动态磁盘），能够恢复一个容错卷上的坏扇区中的数据。但是，如果此硬盘并没有使用SCSI协议或者用完了备用扇区，那么，卷管理器无法执行扇区备用（sector sparing）功能来替换此坏扇区。（有关卷管理器的更多信息，请参见第10章。） 当文件系统从该扇区读取数据时，卷管理器恢复该扇区中的数据，并且给文件系统返回一个警告，说明这里只有一份数据拷贝了。

FAT文件系统并不响应卷管理器的这一警告。而且，这些文件系统，以及卷管理器都不记录这些坏簇，所以，用户必须运行Chkdsk或者Format工具来防止一个卷管理器重复地为文件系统恢复数据。从“去除坏扇区”这一用途而言，Chkdsk和Format并不是理想的工具。Chkdsk可能要花很长时间才能找到并去除坏扇区，而Format则擦掉了它正在做格式化的分区上的所有数据。

卷管理器的扇区备用功能在NTFS文件系统中也有对应的功能，NTFS动态地替换掉包含坏扇区的簇，并且跟踪记录这一坏簇，以便将来不会再重用它（以前曾经说过，NTFS保持移植性的做法是，它对逻辑簇而非物理扇区进行编址）。当卷管理器不能执行扇区备用功能时，NTFS执行这些功能。当一个卷管理器返回一个坏扇区警告，或者当硬盘驱动程序返回一个坏扇区错误时，NTFS分配一个新的簇来替换掉包含此坏扇区的簇。NTFS把卷管理器已经恢复出来的数据拷贝到新的簇中，从而再次建立数据冗余。

图12.54显示了一个用户文件的MFT纪录，该文件的一个数据行串中有一个坏簇（该文件在此簇变坏以前就退出了）。当NTFS接收到一个坏扇区错误时，它将包含该扇区的簇重新分配给它的坏簇文件。这可以防止该坏簇再被分配给另外的文件。然后，NTFS为该文件分配一个新簇，并且改变该文件的VCN-LCN映射关系，以便指向新簇。图12.55示出了这一坏簇重映射过程（本章前面已经介绍过了）。编号为1357的簇包含一个坏扇区，它被新的编号为1049的簇所替代。

www.TopSage.com

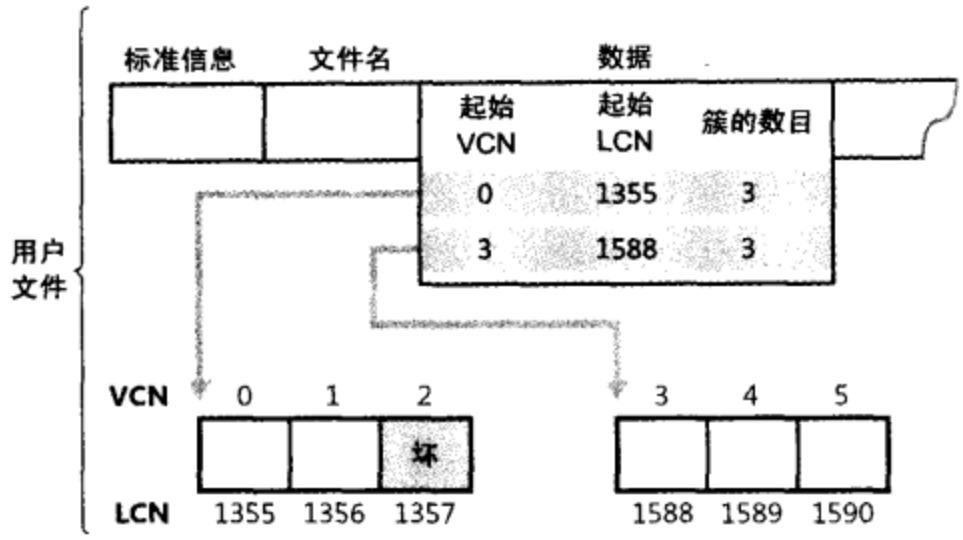


图12.54 包含一坏簇的用户文件的MFT纪录

坏扇区错误是人们不希望看到的，但是，当它们真的发生时，NTFS和卷管理器联合起来提供了最佳的可能解决方案。如果此坏扇区是在一个冗余的卷上，则卷管理器恢复此数据，并且在可能的情况下替换该扇区。如果它不能替换该扇区，则向NTFS返回一个警告。于是，NTFS替换此包含坏扇区的簇。

如果该卷并没有被配置成一个冗余卷，则坏扇区中的数据无法被恢复。当该卷被格式化成FAT卷，并且卷管理器不能恢复数据时，从坏扇区中读取数据的操作将会得到不确定的结果。如果文件系统有一些控制结构位于坏扇区中，则整个文件或者一组文件（甚至有可能整个磁盘）将会丢失。在最好的情况下，受影响的文件中有一些数据（通常是该文件中坏扇区之后的所有数据）将会丢失。而且，FAT文件系统有可能将此坏扇区重新分配给该卷上的同一个文件或者其他的文件，从而使得该问题再次发生。

如同其他的文件系统一样，NTFS如果没有得到卷管理器的帮助，也不能恢复一个坏扇区中的数据。然而，NTFS可以极大地抑制住一个坏扇区可能造成的损失。如果NTFS在一个读操作过程中发现了坏扇区，那么，它重新映射该扇区所在的簇，如图12.55所示。如果该卷没有被配置成一个冗余卷，那么，NTFS给调用程序返回一个“数据读”错误。虽然这一簇中的数据丢失了，但是该文件的其余数据，乃至该文件系统的其余数据，仍是完好无损的。调用程序可以采取适当的措施来响应此数据丢失错误，而且，在将来的簇分配中，这一坏簇不会再被重用。如果NTFS是在一个写操作而不是读操作中发现了坏簇，那么，NTFS在写数据以前重新映射该簇，因此既不会丢失数据，也不会产生错误。

www.TopSage.com

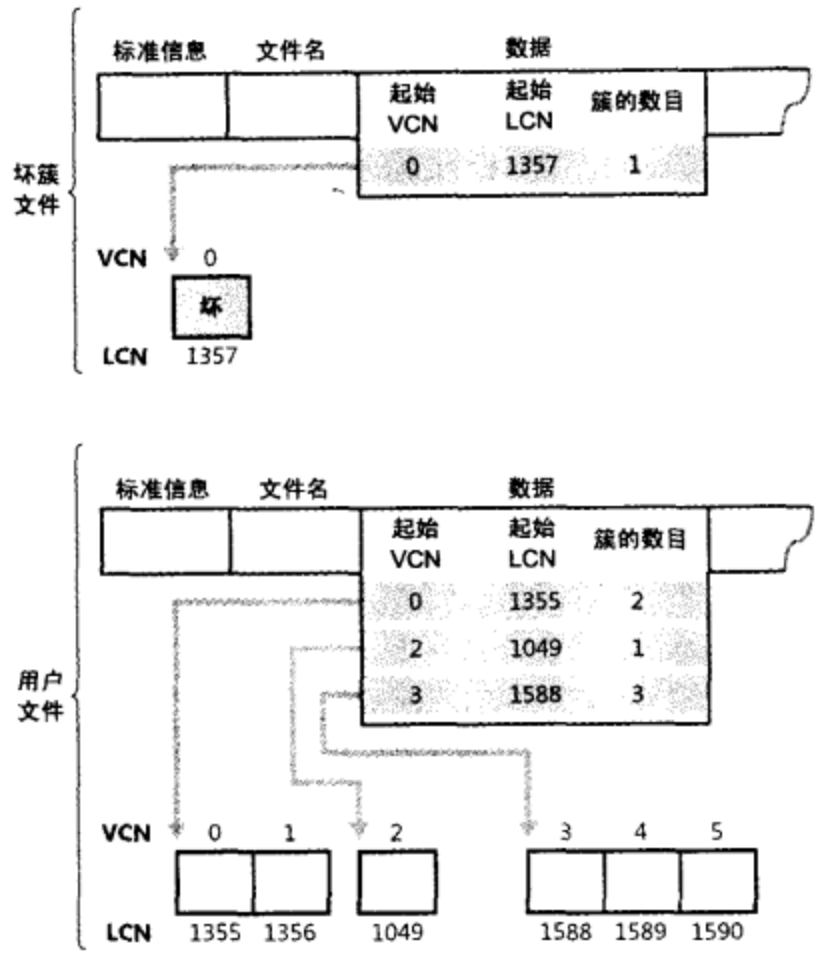


图12.55 坏簇重映射

如果文件系统数据被存储在一个已变坏的扇区中，则NTFS遵从同样的恢复过程。如果该坏扇区是在一个冗余卷上，则NTFS利用卷管理器恢复出来的数据，动态地替换该簇。如果该卷不是冗余的，则数据无法被恢复，NTFS设置该卷文件中的一位，以指明该卷上出现了数据破坏。当系统下一次重新引导时，NTFS的Chkdsk工具检查这一位，如果这一位被设置了的话，则Chkdsk执行，通过重构NTFS元数据来修复文件系统的破坏。

在极少数情况下，即使在一个容错的磁盘配置上，文件系统破坏的情况也可能发生。一个双重错误可能既毁坏文件系统数据，也使得重构文件系统的机制不能工作。如果当NTFS正在写一个MFT文件纪录（比如日志文件的MFT文件纪录，或一个文件名索引的MFT文件纪录）的镜像拷贝时系统崩溃了，那么，此文件系统数据的镜像拷贝可能没有被完全更新。如果系统被重新引导了，并且在主磁盘上有一个坏扇区恰好处在与镜像磁盘的未完成写操作同样的位置上，那么，NTFS将无法从磁盘镜像中恢复出正确的数据来。为了检测出文件系统数据中像这样的破坏情况，NTFS实现了一种特殊的方案。如果它发现了有一个不一致，那么，它设置该卷文件中的破坏位（corruption bit），这将使得在下一次系统重新引导的时候Chkdsk重构NTFS元数据。因为在容错磁盘配置的情况下，文件系统破坏的情况很少发生，所以，很少需要Chkdsk。提供这种方案，只是作为一种预防手段，而并非作为首要的数据恢复策略。

www.TopSage.com

Chkdsk在NTFS上的使用同它在FAT文件系统上的使用完全不同。FAT在向磁盘上写东西以前，首先设置所操作卷的脏位（dirty bit），在修改完成以后再重置该位。如果当系统崩溃时任何一个I/O操作正在进行之中，那么，此脏位处于被设置的状态，当系统重新引导起来时，Chkdsk就会运行。而在NTFS上，只有当发现了意外的，或不可读的文件系统数据，并且NTFS未能从一个冗余卷或单个卷上的冗余文件系统结构中恢复此数据时，Chkdsk才会运行（系统引导扇区是有复制的，用于引导系统和运行NTFS恢复过程的MFT部分也是有复制的。这种冗余确保了NTFS将总能引导，并恢复其自身）。

根据本节中我们已经描述过的各种条件，表12.6概要地展示了当一个磁盘卷上的扇区变坏时所发生的情形。假设此磁盘卷已被格式化成Windows所支持的一种文件系统。

表12.6 有关NTFS数据恢复情形的一个摘要

情 形	有空余扇区的SCSI磁盘	一个非SCSI扇区，或者磁盘上没有备用的扇区*
容错卷+	1. 卷管理器恢复此数据 2. 卷管理器执行扇区备用功能（替换掉坏扇区） 3. 文件系统对于此错误并不知晓	1. 卷管理器恢复此数据 2. 卷管理器向文件系统发送此数据，以及一个坏扇区错误 3. NTFS执行簇重映射操作
非容错卷	1. 卷管理器不能恢复此数据 2. 卷管理器向文件系统发送一个坏扇区错误 3. NTFS执行簇重映射操作。数据丢失*	1. 卷管理器不能恢复此数据 2. 卷管理器向文件系统发送一个坏扇区错误 3. NTFS执行簇重映射操作。数据丢失‡

* 在下面两种情况下，卷管理器都不会执行扇区备用功能：(1) 对于不使用SCSI协议的硬盘，没有标准的接口可用来提供扇区备用功能；(2) 有些硬盘没有提供扇区备用功能的硬件支持，而且，对于提供了扇区备用功能的SCSI硬盘，如果大量的扇区变坏的话，最终也可能会用光备用扇区。

\+ 容错卷是指下面的卷之一：镜像卷集合或者RAID-5卷集合。

‡ 在写操作中，数据不会丢失：NTFS在写之前重映射该簇。

如果坏扇区所在的卷是一个容错卷（镜像卷或RAID-5卷），并且，如果此硬盘是支持扇区备用的（而且它还没有用完所有备用的扇区），那么，不管你当前正在使用的文件系统是什么（是FAT还是NTFS），都不会有问题。卷管理器会替换掉坏扇区，无需用户或文件系统的干预。

如果坏扇区位于一个不支持扇区备用的硬盘上，那么，文件系统负责替换（重映射）此坏扇区，或者若在NTFS的情况下，替换此坏扇区所在的簇。FAT文件系统并没有提供扇区或簇

www.TopSage.com

的重映射功能。NTFS的簇重映射功能的好处是，文件中的坏点可能被修复，并且不会伤害到文件（或者不会伤害到文件系统，这也是有可能的），坏扇区也不会被重新分配给同一个文件或另外的文件。

12.8 加密文件系统（EFS）安全性

EFS安全性依赖于密码学支持。当第一次一个文件被加密时，EFS为执行此次加密的用户账户分配一对私钥/公钥，以便在文件加密过程中使用。用户可以通过Windows Explorer来加密文件，做法是，打开一个文件的属性（Properties）对话框，按下Advanced按钮，选中“Encrypt Contents To Secure Data”选项，如图12.56所示。用户也可以通过一个名为*cipher*的命令行工具来加密文件。对于已被指定为加密目录的目录，Windows自动加密其中的文件。当一个文件被加密时，EFS为该文件生成一个随机数，EFS称其为该文件的文件加密密钥（FEK，file encryption key）。EFS使用此FEK来加密该文件的内容，它使用的加密算法是DES（Data Encryption Standard）算法的一个更强的变形——在Windows 2000上是DESX，在Windows XP（SP1及以后）和Windows Server 2003上是DESX、3重DES（3DES）或AES（Advanced Encryption Standard）。EFS将该文件的FEK与该文件存储在一起，但是它通过RSA公钥加密算法，利用该用户的EFS公钥对FEK进行了加密。当EFS完成了这些步骤以后，该文件是安全的：其他用户没有解密以后的文件FEK，所以无法解密文件的数据；他们没有该用户的私钥，所以也无法解密FEK。

EFS FEK密钥强度

在Windows 2000和Windows XP上，默认的FEK加密算法是DESX，而在Windows XP（SP1及更高版本）和Windows Server 2003上，默认的FEK加密算法是AES。在准许出口到美国以外其他国家和地区的Windows版本上，EFS驱动程序实现了一个56位密钥的DESX加密算法。对于仅限于美国市场的Windows版本，以及随带128位加密包的出口版本，此密钥是128位长。Windows AES算法使用256位密钥。采用3DES可允许访问更长的密钥，所以，如果你要求更大的密钥强度，你可以通过以下两种方法之一来启动3DES加密功能：将其作为对所有的系统密码服务都适用的算法，或者仅限于EFS使用。

要想让3DES成为对所有的系统密码服务都适用的加密算法，请从Start菜单的Run对话框中输入secpol.msc，以打开本地安全策略编辑器（Local Security Policy Editor）；再打开Local Policies下面的Security Options节点。查看“System Cryptography: Use FIPS Compliant Algorithms For Encryption, Hashing And Signing ”的属性，并启动（enable）此项设置。

要想仅仅针对EFS启动3DES，请创建DWORD注册表值HKEY_LOCAL_MACHINE\SOFTWARE\ icrosoft\Windows NT\CurrentVersion\EFS\AlgorithmID，将它设置为0x6603，然后重新引导。

www.TopSage.com

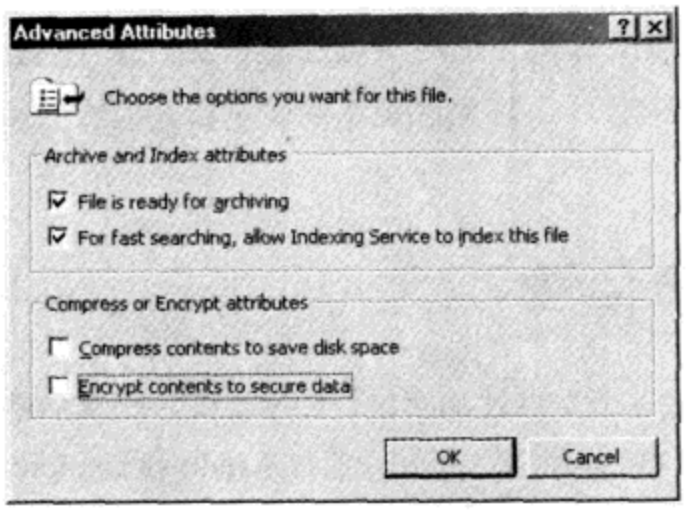


图12.56 使用“Advanced Attributes”对话框来加密文件

EFS使用一个私钥/公钥算法来加密FEK。为了加密文件数据，EFS使用DESX、AES或者3DES，因为这些算法都是**对称加密算法**，这意味着它们使用同样的密钥来加密和解密数据。通常而言，对称加密算法非常快速，这使得它们非常适合于加密大量的数据，比如文件数据。然而，对称加密算法有一个缺点：如果你得到了密钥，你就可以绕过它们的安全性。如果多个用户想要共享一个仅仅由DESX、AES或3DES算法来保护的加密文件，那么，每个用户都要求能访问到该文件的FEK。让FEK不经加密无疑是一个安全问题，但是一旦加密FEK以后，则要求所有的用户都共享同样的FEK解密密钥——这是另一个潜在的安全问题了。

让FEK保持安全是一个难题，EFS利用其加密体系结构中的另一半——基于公钥的技术来解决这一难题。为每个需要访问该文件的用户加密该文件的FEK，这样可使得多个用户能够共享一个加密的文件。EFS可以用每个用户的公钥来加密一个文件的FEK，并且将每个用户的加密FEK与该文件存储在一起。任何人都可以访问一个用户的公钥，但是，对于使用该公钥来加密的数据，没有人能够利用该公钥进行解密。用户解密此文件的惟一办法是通过他们的私钥，操作系统必须访问此私钥。用户的私钥可以用来解密该文件的FEK加密拷贝。基于公钥的算法通常很慢，但是EFS只使用这些算法来加密FEK。把一个公开可用的公钥和一个私钥分开来进行管理，这使得其密钥管理比对称加密算法的密钥管理更加容易，并且解决了“如何保持FEK安全”的难题。

Windows把一个用户的私钥保存在它的用户轮廓目录（通常在\Documents and Settings下面）的Application Data\Microsoft\Crypto\RSA子目录中。为了保护这些私钥，Windows用一个随机的对称密钥，即该用户的主密钥（master key）来加密RSA文件夹中的所有文件。主密钥有64字节长，是通过一个强随机数生成器得到的。主密钥也存储在Application Data\Microsoft\Protect目录下该用户的轮廓中，是用3DES来加密的，并且加密密钥是部分根据该用户的口令生成的。当一个用户改变其口令时，主密钥也自动被解密，并利用新的口令重新加密。

www.TopSage.com

EFS包含几个组件，它们联合起来协同工作，如图12.57中所显示的EFS总体结构框图。在Windows 2000上，EFS被实现为一个设备驱动程序，它运行在内核模式下，与NTFS文件系统驱动程序紧密地连接在一起。但是在Windows XP和Windows Server 2003上，EFS的功能被合并到NTFS驱动程序中。无论何时当NTFS碰到一个加密文件时，NTFS执行EFS的函数，这些函数是当EFS内核模式代码初始化时向NTFS注册的。当应用程序访问加密文件时，EFS的这些函数负责加密和解密文件数据。虽然EFS把一个FEK与文件数据保存在一起，但是该FEK被通过用户的公钥做了加密。为了加密或解密文件数据，EFS必须在密码服务（用户模式下）的帮助下解密该文件的FEK。

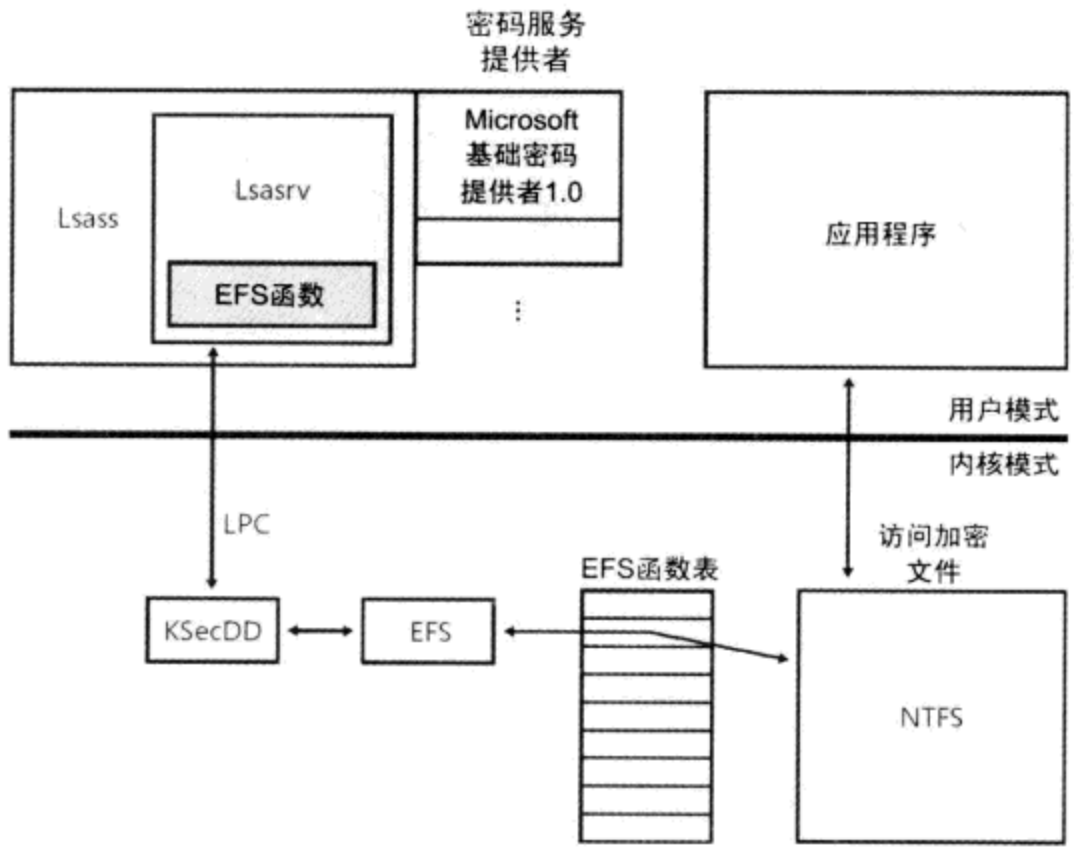


图12.57 EFS总体结构

本地安全权威子系统（LSASS - \Windows\System32\LSASS.exe）不仅负责管理登录会话，而且也处理与EFS密钥管理相关的杂务。例如，当EFS驱动程序需要解密一个FEK以便解密用户想要访问的文件数据时，EFS驱动程序向LSASS发送一个请求。EFS通过一个本地过程调用（LPC）发送这一请求。KSecDD（\Windows\System32\Drivers\Ksecdd.sys）设备驱动程序导出了一些函数，供其他的需要向LSASS发送LPC消息的驱动程序使用。LSASS的本地安全权威服务器（Lsasrv - \Windows\System32\Lsasrv.dll）组件在监听远过程调用（RPC）请求，它把“解密FEK”的请求传递给Lsasrv中某一个适当的与EFS相关的解密函数。Lsasrv使用Microsoft CryptoAPI（也称为CAPI）中的函数来解密此FEK（EFS驱动程序刚才将此FEK以加密形式传递给LSASS了）。

www.TopSage.com

CryptoAPI包含了密码服务提供者（CSP，cryptographic service provider）DLL，正是这些CSP DLL使得各种密码服务（比如加密/解密和散列计算）能够为应用程序所使用。例如，CSP DLL管理用户私钥和公钥的获取过程，因此，Lsasrv无需操心诸如“这些密钥是如何被保护的”之类的细节，甚至也不用关心加密算法的细节。EFS使用了Microsoft增强密码提供者（\Windows\System32\Rsaenh.dll）所提供的RSA加密算法。在Lsasrv解密了一个FEK以后，Lsasrv通过一个LPC回复消息将该FEK返回给EFS驱动程序。在EFS接收到此解密的FEK以后，它可以使用DESX来为NTFS解密文件数据。下面我们来详细地看一看EFS如何与NTFS集成在一起，以及Lsasrv如何使用CryptoAPI来管理密钥。

第一次加密一个文件

当NTFS驱动程序碰到一个加密的文件时，它要调用EFS的函数。一个文件的属性记录了该文件是加密的，其记录方式就如同它记录了“该文件是压缩的”（本章前面讨论过）一样。NTFS和EFS有专门的接口用来将一个文件从非加密的形式转换成加密的形式，但主要由用户模式组件来驱动这一过程。正如前面所描述的，Windows允许你通过两种方式来加密一个文件：使用cipher命令行工具，或者在Windows Explorer中打开一个文件的“Advanced Attributes”对话框，并选中“Encrypt Contents To Secure Data”复选框。Windows Explorer和cipher命令都依赖于Advapi32.dll（Advanced Windows APIs DLL，高级Windows APIs DLL）导出的EncryptFile Windows API函数。Advapi32加载另一个DLL，Feclient.dll （File Encryption Client DLL，文件加密客户DLL）来获得一些API，从而它可以使用这些API来调用（通过LPC）Lsasrv中的EFS接口。

当Lsasrv接收到来自Feclient的RPC消息，要求加密一个文件时，它使用Windows的模仿设施来模仿当前运行该应用程序（*cipher*或Windows Explorer，它正在加密该文件）的用户。（第8章讲述了模仿。）这一过程使得Windows可以将Lsasrv执行的文件操作，看作同“当前想要加密该文件的用户正在执行这些操作”一样。Lsasrv通常运行在System账户下。（第8章介绍了System账户。）实际上，如果Lsasrv不模仿该用户，那么它通常没有许可权来访问所要操作的文件。

Lsasrv接下来在卷的System Volume Information目录中创建一个日志文件，Lsasrv将在此日志文件中记录加密过程的进度。该日志文件的名称通常是Efs0.log，但如果其他的文件也正在进行加密之中，那么，递增后面的数值以替换名称中的0，直至创建一个独一无二的日志文件名称为止。

由于CryptoAPI依赖于一个用户的注册表轮廓中存储的信息，所以，如果此轮廓尚未被加载到系统中的话，Lsasrv接下来使用Userenv.dll（用户环境DLL）的*LoadUserProfile* API函数，将此轮廓加载到它当前正在模仿的用户的注册表中。通常，此用户轮廓已经被加载进来了，因为当一个用户登录的时候Winlogon会加载一个用户的轮廓。然而，如果一个用户使用Windows的*RunAs*命令来登录到另一个不同的账户中，那么，当该用户试图从那个账户下访问加密的文件时，该账户的轮廓可能还没有被加载进来。

然后，Lsasrv利用Microsoft基础密码提供者1.0 CSP的RSA加密设施，为该文件生成一个FEK。

www.TopSage.com

构建密钥环

到这时候，Lsasrv有了一个FEK，现在它可以构建出与该文件存储在一起的EFS信息，包括该FEK的一个加密版本。Lsasrv读取到当前正在执行此加密任务的用户的HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash值，以便获得该用户的公钥签名（注意，直到有一个文件或文件夹被加密的时候，此键才会出现在注册表中）。Lsasrv使用此签名来访问该用户的公钥，以及加密FEK。

现在Lsasrv可以构建出与该文件存储在一起的EFS信息了。EFS在该加密文件中只存储一个块的信息；对于共享此文件的每个用户，该信息块都包含了一个数据项。这些数据项称为**密钥项**（*key entry*），EFS将它们存储在该文件的EFS数据的数据解密域（DDF，Data Decryption Field）部分。多个密钥项的集合称为一个**密钥环**（*key ring*），因为正如前面所提到的，EFS允许多个用户共享加密的文件。

图12.58显示了一个文件的EFS信息格式和密钥项格式。EFS在密钥项的第一部分存储了足够的信息，以便精确地描述一个用户的公钥。这部分数据包括该用户的安全ID（SID）（注意，SID并不保证总是会出现）、存储此密钥的容器名称、密码提供者的名称，以及私钥/公钥对的证书散列值。解密过程仅仅使用了私钥/公钥对的证书散列值。密钥项的第二部分包括FEK的一份加密版本。Lsasrv通过CryptoAPI，利用RSA算法和该用户的公钥来加密此FEK。

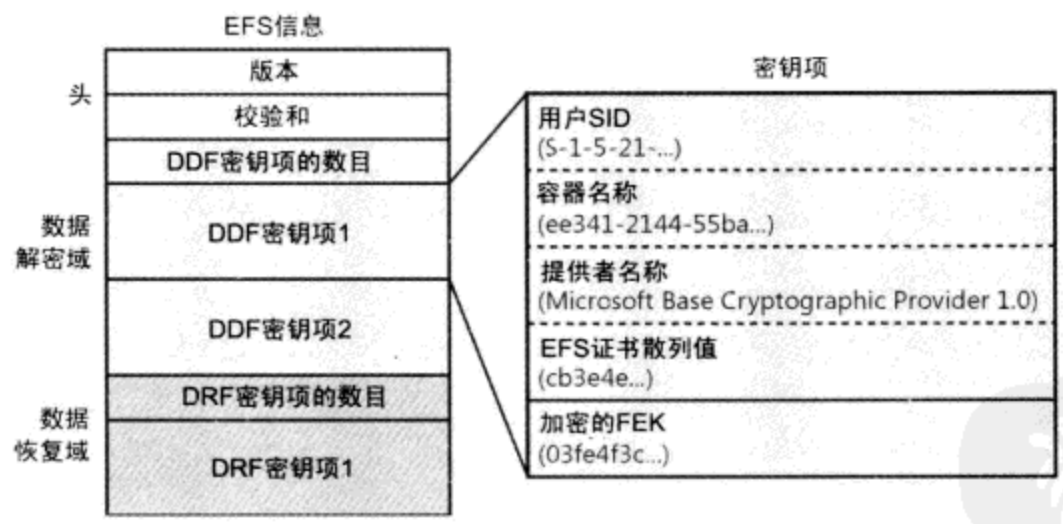


图12.58 EFS信息格式和密钥项格式

接下来，Lsasrv创建另一个密钥环，它包含的是恢复密钥项。EFS将有关恢复密钥项的信息存储在一个文件的数据恢复域（DRF，Data Recovery Field ）中。DRF项的格式与DDF项的格式完全相同。DRF的用途是，当管理权威人员必须访问该用户的数据时，指定的账户，或者称为恢复代理（Recovery Agent），能够解密该用户的文件。例如，假设一个公司的员工忘记了他或她的登录口令。管理员可以重置该用户的口令。但是，如果没有恢复代理，则没有人能够恢复该用户的加密数据。

www.TopSage.com

恢复代理是由本地计算机或者域的“Encrypted Data Recovery Agents”安全策略来定义的。通过“Group Policy（组策略）”MMC加载件，你可以看到此策略，如图12.59所示。当你使用恢复代理向导（Recovery Agent Wizard，右键点击Encrypted Data Recovery Agents，从New菜单项中选择Encrypted Recovery Agent）时，你可以添加恢复代理，并指定此恢复代理使用哪些私钥/公钥对（由它们的证书来指定）来进行EFS恢复。当Lsasrv初始化时，以及当Lsasrv接收到恢复代理的变化通知时，Lsasrv会解释此恢复策略。EFS使用“专门为EFS恢复而注册的密码提供者”，为每一个恢复代理创建一个DRF密钥项。默认的恢复代理提供者是Microsoft基础密码提供者1.0的RSA加密设施——与Lsasrv为用户密钥使用的密码提供者相同。

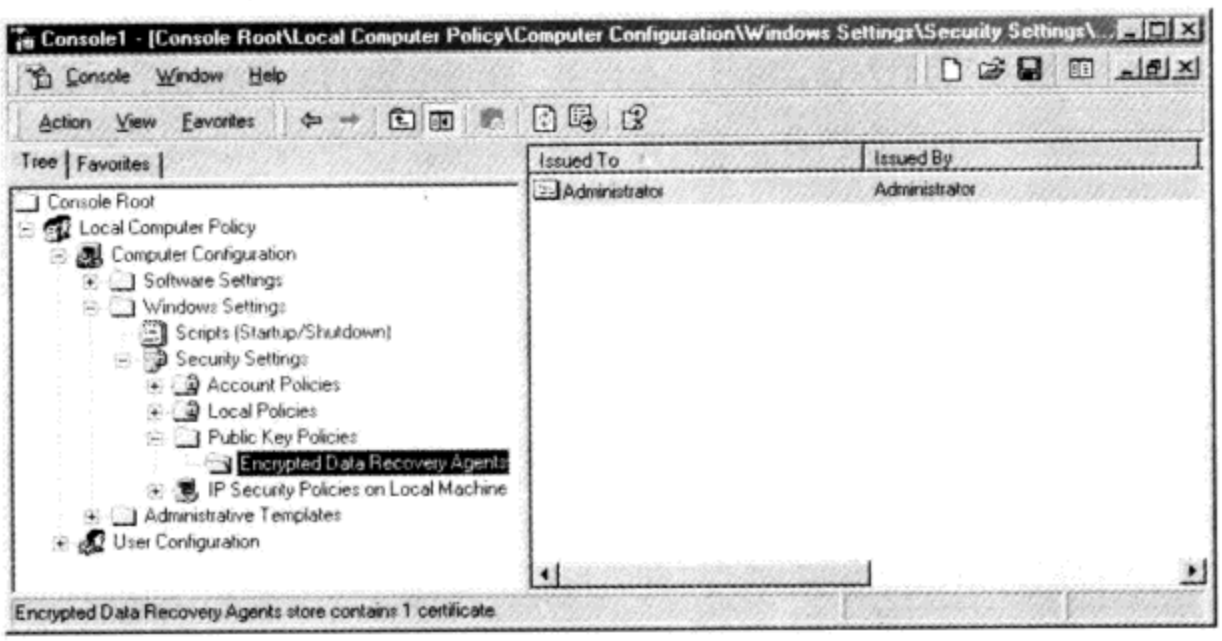


图12.59 “Encrypted Data Recovery Agents”组策略

在为一个文件创建EFS信息的最后步骤中，Lsasrv利用Microsoft基础密码提供者1.0中的MD5散列设施，为DDF和DRF计算一个校验和。Lsasrv将校验和的结果存储在EFS信息头中。EFS在解密过程中需要引用此校验和，以确保一个文件的EFS信息的内容没有被破坏，或者没有被篡改。

加密文件数据

图12.60显示了加密过程的流程。Lsasrv为用户想要加密的文件构建了必要的信息以后，便开始加密此文件。Lsasrv为当前正在进行加密的文件，创建一个备份文件Efs0.tmp（如果同时还存在其他的备份文件，则在备份文件名中使用更大的编号）。Lsasrv在包含此加密文件的目录中创建此备份文件。Lsasrv将一个很严格的安全描述符应用在此备份文件上，这样，只有System账户才可以访问该文件的内容。接下来，Lsasrv初始化它在加密过程的第一阶段中创建的日志文件。最后，Lsasrv在日志文件中记录下“备份文件已创建”。只有在原始的文件已经被完整地备份好以后，Lsasrv才加密该文件。

www.TopSage.com

接下来Lsasrv向EFS内核模式的代码发送一个命令（通过NTFS），以便将刚刚创建的EFS信息添加到原始文件中。在Windows 2000上，NTFS接收到该命令，但是因为NTFS并不理解EFS命令，所以NTFS调用EFS驱动程序。EFS内核模式代码获得Lsasrv发送过来的EFS信息，并使用NTFS的导出函数，将此信息应用到文件中。这些被导出的NTFS函数允许EFS在NTFS文件中加入$EFS属性。执行权又返回到Lsasrv，Lsasrv将正在执行加密的文件的内容拷贝到备份文件中。当此备份拷贝完成（包括所有额外的数据流）时，Lsasrv在日志文件中记录下“备份文件已更新”。然后，Lsasrv给NTFS发送另一个命令，告诉NTFS，现在可以加密原始文件的内容。

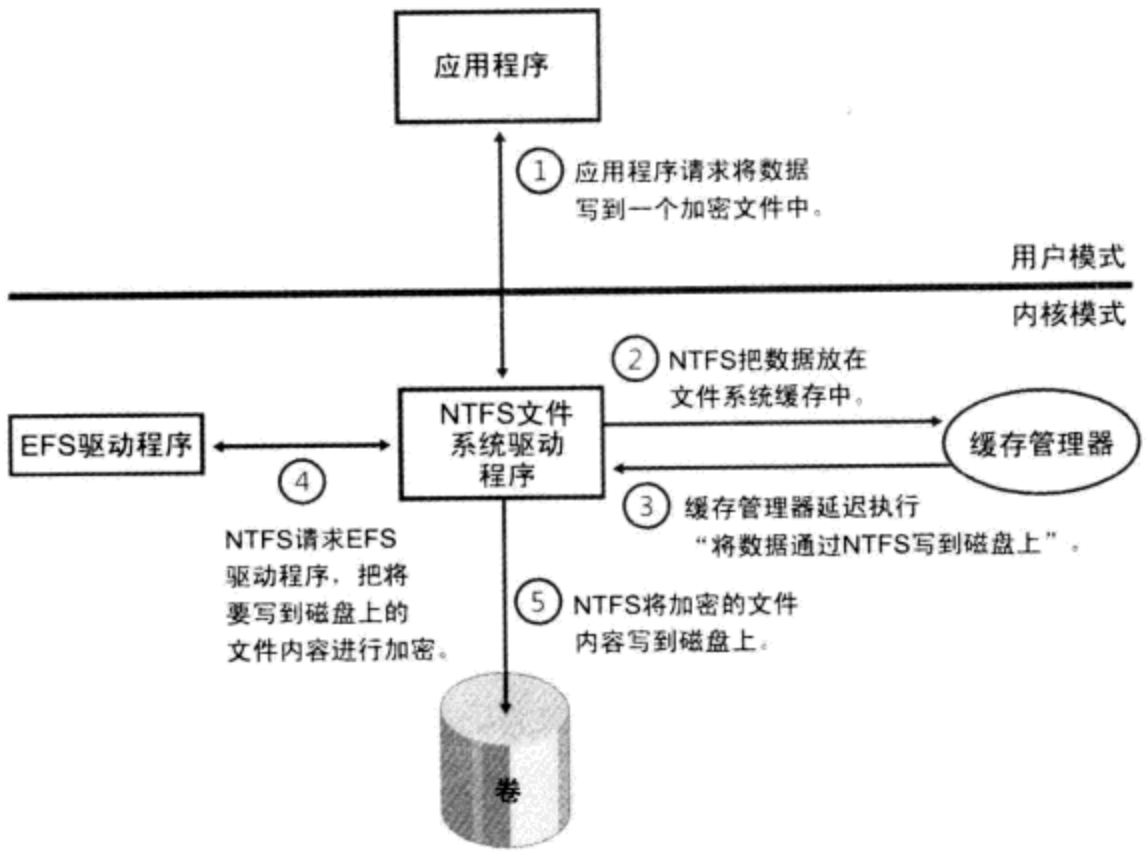


图12.60 EFS的流程

当NTFS接收到EFS命令要加密该文件时，NTFS删除原始文件的内容，并且把备份数据拷贝到该文件中。NTFS拷贝了该文件的每一个区域以后，刷新该区域的数据（数据还在文件系统缓存中），这将提示缓存管理器，告诉NTFS将该文件的数据写到磁盘上。因为该文件被标记为加密文件，所以，在写文件过程中，NTFS在把数据写到磁盘上以前，调用EFS来加密这些数据。EFS使用NTFS传递给它的未加密的FEK，来执行该文件的DESX、AES或3DES加密操作，每次一个扇区（512字节）。

在EFS加密了该文件以后，Lsasrv在日志文件中记录下“加密过程已成功”，并删除该文件的备份拷贝。最后，Lsasrv删除日志文件，将控制权返回给请求加密该文件的那个应用程序。

www.TopSage.com

加密过程总结

下面的列表概括了在加密一个文件时EFS所执行的步骤。

1. 如果有必要的话，加载用户轮廓。
2. 在System Volume Information目录下创建一个日志文件，名为Efs*x*.log，这里*x*是一个惟一号（例如，Efs0.log）。随着后续的步骤逐渐被执行，该日志文件中就会留下纪录，所以，万一在加密过程中系统失败的话，该文件可以被恢复。
3. 通过基础密码提供者1.0为该文件生成一个随机的128位FEK。
4. 生成或者获得一个用户的EFS私钥/公钥对。HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\CertificateHash标识了该用户的密钥对。
5. 为该文件创建一个DDF密钥环，其中有一项是为该用户而建立的。此密钥项包含该FEK的一份拷贝，但已经用该用户的EFS公钥进行了加密。
6. 为该文件创建一个DRF密钥环。对于系统中的每个恢复代理都有一个对应的密钥项，并且每一项所包含的FEK都用相应的恢复代理的EFS公钥进行了加密。
7. 在被加密文件所在的目录中，创建一个备份文件，其名称的形式为Efs0.tmp。
8. 将DDF和DRF密钥环添加到头部，作为该文件的EFS属性。
9. 将原始文件标记为加密文件，并将其内容拷贝到备份文件中。
10. 原始文件的内容被销毁，备份文件再被拷回到原始文件。这次拷贝操作导致原始文件中的数据被加密，因为现在该文件被标记为加密文件。
11. 删除备份文件。
12. 删除日志文件。
13. 卸载用户轮廓（如果它是在第1步中被加载进来的）。

如果在加密过程中系统崩溃，则要么原始文件仍然原封未动，要么备份文件包含了一份完整的拷贝。在系统崩溃以后，当Lsasrv初始化时，它在系统的每个NTFS卷的System Volume Information子目录下寻找日志文件。如果Lsasrv找到一个或多个日志文件，那么，它检查它们的内容，以确定如何执行恢复。如果在系统崩溃时刻原始文件未被修改，则Lsasrv删除日志文件和对应的备份文件，否则，Lsasrv用备份文件覆盖掉已被部分加密的原始文件，然后删除日志文件和备份文件。在Lsasrv处理完日志文件以后，从加密的角度而言，文件系统又将处于一致的状态，用户数据没有丢失。

www.TopSage.com

解密过程

当一个用户打开一个加密的文件时，解密过程就开始了。NTFS在打开该文件时检查其属性，然后执行EFS驱动程序中的一个回调函数。EFS驱动程序读入与此加密文件相关联的$EFS属性。为了读取此属性，驱动程序调用了NTFS导出给EFS使用的EFS支持函数。NTFS完成必要的步骤以便打开该文件。EFS驱动程序确保：当前正在执行打开操作的用户有权访问该文件的加密数据（也就是说，在DDF或DRF密钥环中有一个加密的FEK对应于与该用户相关联的私钥/公钥对）。在EFS执行了此项验证以后，EFS获得该文件的解密的FEK，以便用于后续的数据操作（用户随后可能要对该文件执行一些数据操作）。

EFS不能解密一个FEK，它要依赖于Lsasrv（它可以使用CryptoAPI）来执行FEK解密操作。EFS通过Ksecdd.sys驱动程序给Lsasrv发送一个LPC消息，请Lsasrv获得“在该文件的$EFS属性数据（EFS数据）中对应于当前正在执行打开操作的用户的FEK的解密形式”。

当Lsasrv接收到此LPC消息时，Lsasrv执行Userenv.dll（用户环境DLL）中的*LoadUserProfile* API函数，以便把该用户的轮廓加载到注册表中（如果该用户的轮廓尚未被加载的话）。Lsasrv依次处理EFS数据中的每一个密钥域，用该用户的私钥试着解密每一个FEK。对于每一个密钥，Lsasrv试图解密一个DDF或DRF密钥项的FEK。如果一个密钥域中的证书散列值并没有引用该用户所拥有的密钥，则Lsasrv向前移动到下一个密钥域。如果Lsasrv不能解密任何一个DDF或DRF密钥域的FEK，则该用户不能获得该文件的FEK。因此，EFS拒绝此应用程序打开和访问该文件。然而，如果Lsasrv找到一个散列值，而且此散列值对应于该用户所拥有的一个密钥，那么，它利用CryptoAPI以及该用户的私钥来解密此FEK。

因为Lsasrv在解密一个FEK时既处理DDF密钥环，也处理DRF密钥环，所以，它自动执行文件恢复操作。如果一个恢复代理没有被赋予访问一个加密文件的权限（也就是说，在DDF密钥环中没有对应的域），但它试着访问一个文件，则EFS将允许该恢复代理获得访问权，因为该代理可以访问DRF密钥环中的一个密钥域的密钥对。

解密后的FEK的缓存

从EFS驱动程序到达Lsasrv，再回到EFS需要一段相对较长的时间——在一个典型的系统上，在解密一个FEK的过程中，CryptoAPI用到了2000多个注册表API调用和400多个文件系统访问操作的结果。EFS驱动程序在NTFS的帮助下，使用一个缓存来试图避免这份昂贵的开销。

www.TopSage.com

解密文件数据

一个应用程序打开了一个加密的文件以后，它可以从该文件中读数据，或者向文件写数据。NTFS从磁盘上读取数据时，它先调用EFS驱动程序来解密文件数据，再把解密后的文件数据放到文件系统缓存中。类似地，当应用程序向一个文件写数据时，文件数据在文件系统缓存中一直是非加密的形式，直到应用程序或者缓存管理器通过NTFS将数据刷新到磁盘上。当一个加密文件的数据被从缓存中写回到磁盘上的时，NTFS调用EFS驱动程序来加密这些数据。

正如前面所说明的，EFS驱动程序以512字节为单位执行加密和解密操作。对于该驱动程序来说，512字节是最为方便的，因为磁盘读和写都是以512字节扇区的倍数为单位来进行的。

加密文件的备份

在任何一个文件加密设施的设计中，一个很重要的方面是，除了“通过加密设施来访问加密文件”的应用程序以外，其他应用程序是无法得到非加密形式的文件数据的。这一限制显著地影响了备份工具，因为它们要把文件存储到存档介质中。EFS处理这个问题的做法是，专门为备份工具提供一个设施，让备份工具能够在文件处于加密状态下进行备份和恢复操作。因此，备份工具不必具备解密文件数据的能力，在其备份过程中无需解密文件数据。

在Windows中，备份工具使用EFS API函数OpenEncryptedFileRaw、ReadEncryptedFileRaw、WriteEncryptedFileRaw和CloseEncryptedFileRaw来访问一个文件的加密的内容。Advapi32.dll库提供了这些API函数，所有这些函数都使用LPC来调用Lsasrv中对应的函数。例如，在备份工具在一次备份操作过程中打开了一个文件以便进行原始数据访问（raw access）以后，该工具调用ReadEncryptedFileRaw来获得文件数据。Lsasrv函数EfsReadFileRaw向NTFS驱动程序发出控制命令（这些命令被通过EFS会话密钥和DESX、AES或3DES算法进行了加密），首先读取该文件的EFS属性，然后读取该文件的加密的内容。

为了读取一个大的文件，*EfsReadFileRaw*可能必须执行多个读操作。当*EfsReadFileRaw*读取此类大文件的每一部分时，Lsasrv向Advapi32.dll发送一个RPC消息，此RPC消息导致执行一个由备份程序指定的回调函数，此回调函数是备份程序在调用*ReadEncryptedFileRaw* API函数时指定的。*EfsReadFileRaw*把刚刚读取到的加密的数据交给此回调函数，而回调函数可以把这些数据写到备份介质中。备份工具也采用类似的方式来恢复被加密的文件。备份工具调用*WriteEncryptedFileRaw* API函数，当Lsasrv的*EfsWriteFileRaw*函数在恢复该文件的内容时，它会调用备份程序中的一个回调函数，以便从备份介质中获得加密的数据。

www.TopSage.com



实验：查看EFS信息

EFS还有大量其他的API函数，应用程序可以使用这些函数来操纵被加密的文件。例如：通过*AddUsersToEncryptedFile* API函数，应用程序可以把对加密文件的访问权赋予其他额外的用户；通过*RemoveUsersFromEncryptedFile*函数，可以撤销其他用户对于一个加密文件的访问权。应用程序可以使用*QueryUsersOnEncryptedFile*函数来获得与一个文件相关联的DDF和DRF密钥域的信息。*QueryUsersOnEncryptedFile*返回SID、证书散列值，以及每个DDF和DRF密钥域所包含的显示信息。下面的输出来自于EFSDump工具（你可以从*www.sysinternals.com*获得该工具），这里，在命令行参数中指定了一个加密的文件：

```
C:\>efsdump test.txt
EFS Information Dumper v1.02
Copyright (C) 1999 Mark Russinovich
Systems Internals - http://www.sysinternals.com

test.txt:
DDF Entry:
    DARYL\Mark:
         CN=Mark,L=EFS,OU=EFS File Encryption Certificate
DRF Entry:
    DARYL\Administrator
         OU=EFS File Encryption Certificate, L=EFS, CN=Administrator
```

你可以看到，文件test.txt有一个针对用户Mark的DDF项，和一个针对Administrator的DRF项，在此系统上当前只有Administrator被注册为恢复代理。

你也可以使用OEM支持工具箱（OEM Support Tools）中的Nfi工具，来看到加密文件中确实存在$EFS属性：

```
C:\>nfi test.txt
NTFS File Sector Information Utility.Copyright (C) Microsoft Corporation 1999. All rights
reserved.
\Temp\README.TXT    $STANDARD_INFORMATION (resident)    $FILE_NAME (resident)    $DATA (
nonresident)        logical sectors 2728124-2728135 (0x29a0bc-0x29a0c7)    Attribute Type
 0x100 $EFS (nonresident)        logical sectors 2156984-2156987 (0x20e9b8-0x20e9bb)
```

12.9 本章总结

Windows支持各种各样的文件系统格式，无论是本地系统，还是远程客户都可以访问它们。文件系统过滤型驱动程序的结构框架提供了一种非常整洁的途径来扩展和增强对于文件系统的访问。NTFS为本地文件系统存储体提供了一种可靠的、安全的、可扩展的文件系统格式。在下一章中，我们将讨论Windows中的网络结构。

www.TopSage.com

www.TopSage.com

Networking

第13章 网络

Microsoft Windows从开始设计时候就考虑了网络，它包含了广泛的网络支持，并且这些网络支持与I/O系统和Windows API集成在一起。网络软件的四种基本类型是服务、API、协议，以及网络适配器设备驱动程序，而且，每一种类型依次叠加在下一种类型之上，从而构成了一个网络栈。Windows为每一层都定义了良好的接口，所以，除了使用Windows本身随带的各种API、协议和适配器设备驱动程序以外，第三方也可以通过开发自己的API、协议和设备驱动程序，来扩展Windows操作系统的网络能力。

在本章中，我们将带领你从Windows网络栈的顶部浏览到底部。首先，我们展示了Windows的网络软件组件与开放系统互联（OSI，Open System Interconnection）参考模型之间的映射关系。然后，我们简要地讲述了在Windows上可供使用的各种网络API，并且解释它们是如何实现的。你将会了解到多重定向器支持和名称解析是如何工作的，协议驱动程序是如何实现的。在介绍了网络适配器设备驱动程序的实现以后，我们再来看一看“**绑定**（*binding*）”机制，这是将服务、协议栈和网络适配器连接起来的粘合剂。

13.1 Windows的网络总体结构

网络软件的目标是，从一台机器上的应用程序中取出一个请求（通常是一个I/O请求），将它传递到另一台机器上，在远程机器上执行此请求，然后将结果返回到第一台机器上。在这个过程中，该请求必须被沿着这条路径传输几次。一个比较高层次的请求，比如“从机器z的文件y中读取x个字节”，要求网络软件能够确定如何到达机器z，以及该机器能理解什么样的通信软件。然后，该请求还必须能被改变，以便可以跨网络传输——例如，分成短的信息包。当该请求到达另一端的时候，它还必须能够被检查完整性、被解码，以及被发送给正确的操作系统组件以便执行。最后，回复信息也必须被编码，以便被通过网络发送回去。

OSI参考模型

为了帮助不同的计算机厂商能够标准化和集成他们的网络软件，1983年，国际标准化组织（ISO，International Organization for Standardization）针对在机器之间发送消息的问题定义了

www.TopSage.com

一个软件模型。其结果就是OSI参考模型（Open Systems Interconnection[OSI] reference model，开放系统互联参考模型）。该模型定义了七层软件，如图13.1所示。

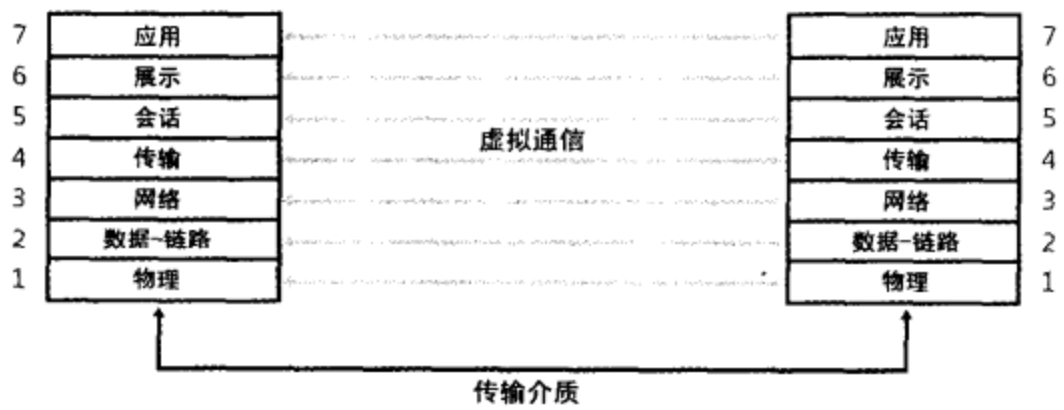


图13.1 OSI参考模型

OSI参考模型是一个理想化的方案，很少有系统完全按照它来实现，但是它通常被用来作为网络原理讨论的框架。一台机器上的每一层都假设它在跟另一台机器上的同一层“通话”。在同一层次上，两台机器“说”的是同一种语言或协议。然而，实际上，每一次网络传输必须要向下传递，通过客户机器的每一层，再传输到网络上，然后在目标机器上向上传递，直至到达某一可以理解和执行该请求的层。

在OSI模型中，每一层的目的是：向更高的层提供服务，并且对如何在低层上实现这些服务进行抽象。详细讲解每一层的目的，显然超出了本书的范围，但下面列出对每一层的简短描述。

- **应用层** 处理两个网络应用程序之间的信息传输，包括诸如安全检查、标识参与的机器，以及发起数据交换之类的功能。
- **表示层** 处理数据的格式化，包括行末是有回车/换行（CR/LF）还是只有回车（CR）、数据是否要压缩或加密，等等。
- **会话层** 管理协作应用程序之间的连接，包括高层次的同步，以及监视哪个应用程序正在“通话”、哪个应用程序正在“监听”。
- **传输层** 在客户端，这一层将消息分成数据包（packet），并且为它们分配序列号，以确保它们全都按照正确的顺序被接收到。在目标端，这一层将已经接收到的数据包组装起来。它也使会话层避免受到硬件上变化的影响。
- **网络层** 创建包头，以及处理路由、拥塞控制和网络互联。它是理解网络**拓扑结构**（*topology*）的最高层。这里的网络拓扑结构指的是网络中的机器的物理配置，带宽的限制，等等。

www.TopSage.com

- **数据-链路层** 传输低层的数据帧，等待确认消息以表明这些数据帧已被接收到，以及重传那些在不可靠线路上被丢失的帧。
- **物理层** 把数据位传送到网络线缆或其他的传输介质上。

图13.1中的虚线代表了在将一个请求传输到远程机器上时所用到的协议。正如前面所说明的，这一层次结构中的每一层都假设它在跟另一台机器上的同一层进行通话，两边使用的是同一个协议。一个请求在它自己这一边往下传递，而在网络另一边向上传递到同样的层次上，这一过程所经过的协议集合被称为一个**协议栈**（*protocol stack*）。

Windows网络组件

图13.2给出了Windows网络组件的一个总体视图，它显示了每个组件如何对应到OSI参考模型中、层与层之间使用了哪些协议。OSI层和网络组件之间的映射关系并不是精确的，这也正是为什么有些组件跨越了多个层次的原因。各种组件如以下所列。

- **网络API**（*Networking API*） 它为应用程序提供了一种与协议无关的方法来跨网络进行通信。网络API既可以在用户模式下实现，也可以同时在用户模式和内核模式下实现，而且，在有些情况下，网络API是另一个“实现了某一特定编程模型或提供了额外服务”的网络API的封装而已（注意，术语**网络***API*也描述了与网络相关的软件所提供的任何编程接口）。
- **传输驱动程序接口**（TDI，*Transport Driver Interface*）**客户** 它是内核模式的设备驱动程序，这些设备驱动程序通常实现了一个网络API具体实现的内核模式部分。TDI客户之所以得此名称，是因为，它们发送给协议驱动程序的I/O请求包（IRP）是根据Windows的传输驱动程序接口标准（在DDK中有文档）来格式化的。该标准为内核模式的设备驱动程序指定了一种公共的编程接口（有关IRP的更多信息，请参见第9章）。
- **TDI传输器**也被称为**传输器**、**网络驱动程序接口规范**（NDIS，*Network Driver Interface Specification*）**协议驱动程序**和**协议驱动程序** 它们是内核模式的协议驱动程序。它们接受来自TDI客户的IRP，并且处理这些IRP所表达的请求。这一处理过程可能要求与一个对等体之间进行网络通信，促使该TDI传输器在此IRP所传递的数据中加入与协议相关的头，比如TCP、UDP或IP，以及使用NDIS函数（也在DDK中有文档）与适配器驱动程序进行通信。TDI传输器通常透明地执行一些消息操作，比如分段和重组、加入序列号、确认和重传，从而为应用程序的网络通信提供便利的设施。
- **NDIS库**（Ndis.sys） 它封装了适配器驱动程序，为它们隐藏了Windows内核模式环境的各种细节。NDIS库导出一些供TDI传输器使用的函数，以及一些供适配器驱动程序使用的支持函数。

www.TopSage.com

■ **NDIS小端口驱动程序** 它们是一些内核模式驱动程序，它们负责为TDI传输器与特定的网络适配器建立连接。NDIS小端口驱动程序被Windows NDIS库包装起来了。这一封装提供了与Microsoft Consumer Windows之间的跨平台兼容性。NDIS小端口驱动程序并不处理IRP，相反地，它们向NDIS库注册一个调用表接口，表中包含的函数指针对应于“NDIS库导出给TDI传输器”的那些函数。NDIS小端口驱动程序利用NDIS库函数（被解析为硬件抽象层[HAL]的函数）与网络适配器进行通信。

正如图中所示，OSI层并不对应于实际的软件。例如，TDI传输器常常要跨越几个边界。实际上，底下四层软件通常合起来称为“传输软件（the transport）”。上面三层中的软件组件通常称为“传输软件的用户（users of the transport）”。

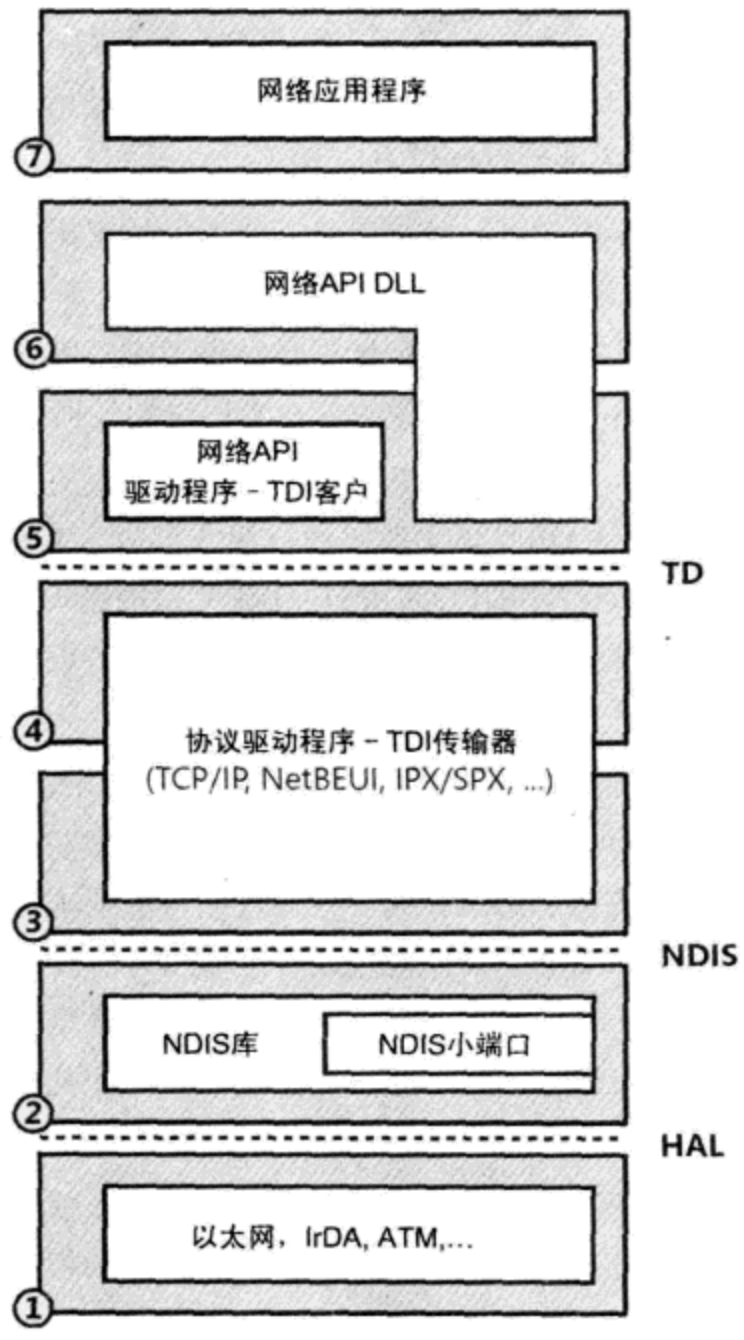


图13.2 OSI模型和Windows网络组件

www.TopSage.com

在本章剩余部分，我们将讨论图13.2中所显示的网络组件（以及图中没有显示的其他组件），看一看这些组件是如何一起工作的，以及它们作为一个整体如何与Windows搭建起关系的。

13.2 网络API

Windows实现了多个网络API，以便为遗留应用程序提供支持，并且与工业标准保持兼容。在本节中，我们将简要地讨论这些网络API，并且介绍一下应用程序是如何使用它们的。要记住的很重要一点是，应用程序应该使用哪个API，这取决于这个API的特征，比如，该API可以在哪些协议上工作、该API是否支持可靠的或者双向的通信，以及该API在其他Windows平台上（此应用程序将来可能要运行于其上）的移植性。我们将讨论下面的网络API：

- Windows套接字（Winsock）；
- 远过程调用（RPC）；
- Web访问API；
- 命名管道和邮件槽；
- NetBIOS；
- 其他的网络API。

Windows套接字（Windows Sockets）

最初的Windows套接字（Winsock）（1.0版本）是Microsoft的BSD套接字（Berkeley Software Distribution Sockets）的实现。BSD套接字是一个编程API，自从20世纪80年代以来它变成了UNIX系统在Internet上进行通信的标准。在Windows上提供对于套接字的支持，这使得“移植UNIX网络应用程序到Windows平台上”这一任务变得相对比较简单。Winsock的现代版本包含了BSD套接字的绝大多数功能，但是它也包含了一些Microsoft特定的增强，而且，这些增强仍然在不断演进。Winsock既支持可靠的、面向连接的通信，也支持不可靠的、无连接的通信。Windows提供了Winsock 2.2。对于所有当前的Windows版本，Winsock 2.2或者直接被包含在Windows中，或者可以通过加载件（add-on）的形式被使用。Winsock 2.2加入了大量的特性，它们远远超出了BSD套接字的规范，比如Winsock 2.2加入了一些可充分利用Windows异步I/O的功能，它们提供了比直接BSD套接字编程更好的性能和伸缩性。

Winsock包含下列特性。

- 支持分散-聚集和异步应用程序I/O。
- 服务质量（QoS）规范，因而当底层网络支持QoS时，上层应用程序可以协商延迟需求和带宽需求。

www.TopSage.com

- 扩展性，因而除了Windows要求Winsock支持的协议以外，Winsock还可以与其他的协议一起被使用。
- 支持集成的名字空间，而不仅仅是应用程序正在使用的Winsock协议所定义的名字空间。例如，一个服务器应用程序可以将它的名称发布到活动目录中，而客户利用名字空间扩展，可以在活动目录中查找服务器的地址。
- 对多点消息的支持，即从单个源并发地传输给多个接收者。

我们将首先讨论典型的Winsock操作，然后讲述一些扩展Winsock的方法。

Winsock客户的操作

Winsock应用程序采取的第一个步骤是调用一个初始化函数，以便初始化Winsock API。在Windows 2000系统上，Winsock应用程序的下一个步骤是创建一个**套接字**（*socket*），该套接字将代表一个通信端点。该应用程序通过调用*gethostbyname*来获得它想要连接的服务器的地址。Winsock是一个与协议无关的API，所以，应用程序可以针对当前系统上任何一个已安装的协议（TCP/IP、基于IPv6的TCP/IP、IPX）来指定一个地址。在获得了服务器的地址以后，面向连接的客户程序使用*connect*并指定此服务器地址，试图与该服务器建立连接。

在Windows XP和Windows Server 2003上，应用程序应该使用*getaddrinfo*，而不是使用*gethostbyname*来获得服务器的地址。*getaddrinfo*返回一个地址列表，其中的地址都是分配给该服务器的，客户试着依次连接到每一个地址上，直至能够与其中之一建立连接。这样做可以确保一个只支持IPv4（IP版本4）的客户能够连接到一个可能同时支持IPv4和IPv6（IP版本6）的服务器，因为服务器同时被分配了两个地址，而客户能够连接到正确的IPv4地址上。

当一个连接被建立起来以后，客户就可以使用*recv*和*send*，在它的套接字上发送和接收数据了。一个无连接的客户在无连接API中指定远程地址，比如说，*send*和*recv*的无连接对应函数，以及*sendto*和*recvfrom*。

Winsock服务器的操作

服务器应用程序的操作步骤的顺序与客户应用程序有所不同。服务器在初始化Winsock API以后，首先创建一个套接字，然后利用*bind*将它绑定到一个本地地址上。同样地，所指定的地址类型，无论是TCP/IP、基于IPv6的TCP/IP，还是其他的地址类型，都取决于服务器应用程序。

如果该服务器是面向连接的，那么，它在该套接字上执行一个*listen*操作，指明它可以为该套接字提供支持的连接数目。然后，它执行一个*accept*操作，以允许客户连接到该套接字上。

www.TopSage.com

如果当前有一个连接请求正在等待，则accept调用立即完成；否则，它要等到有一个连接请求到达才能完成。当一个连接被建立时，*accept*函数返回一个新的套接字，它代表了该连接的服务器一端。服务器可以使用诸如*recv*和*send*这样的函数来实现接收和发送操作。图13.3显示了一个Winsock客户和服务器之间的面向连接的通信。

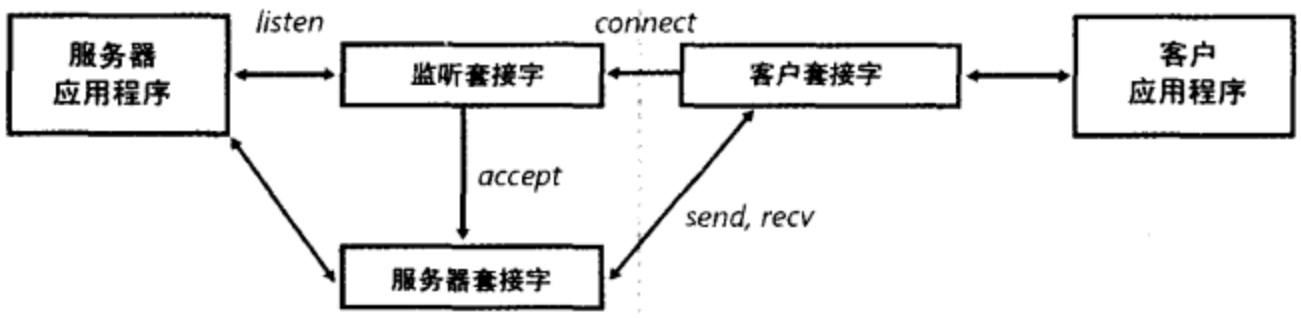


图13.3 面向连接的Winsock操作

对于无连接的服务器，它在绑定了一个地址以后，与无连接的客户并无二致：它只需简单地在每个操作中指定远程地址就可以在套接字上发送和接收数据。绝大多数无连接协议都是不可靠的，一般来说，发送方并不知道目标方是否真的接收到了发送过去的数据包，这种数据包也称为**数据报**（*datagram*）。数据报协议对于快速消息传递是非常理想的，因为在这种应用情形下，建立连接的开销并不大，而且可靠性也不是必需的（不过，应用程序可以在这种协议之上再建立可靠性）。

Winsock的扩展

从Windows编程的观点来看，Winsock API的一个显著的特性是，它与Windows消息集成在一起。Winsock应用程序可以充分利用这种特性来实现异步的套接字操作，它们可以通过标准的Windows消息或执行回调函数来接收一个操作的完成通知。这种能力简化了Windows应用程序的设计，因为应用程序不必采用多线程或管理同步对象，就可以做到既执行网络I/O，同时又响应用户的输入或响应窗口管理器的请求以更新应用程序的窗口。

除了支持这些直接对应于BSD套接字中实现的函数以外，Microsoft也加入了一大堆不属于Winsock标准的函数。其中两个函数*AcceptEx*和*TransmitFile*值得描述一下，因为Windows上的许多Web服务器使用它们来获得很高的性能。*AcceptEx*是*accept*函数的一个版本，它在与客户建立连接的过程中，返回客户的地址和客户的第一个消息。*AcceptEx*使得服务器应用程序可以预先发出多个accept操作，因而它可以处理大量的进入连接请求。利用这个函数，Web服务器可以避免本来必不可少的多个Winsock函数的执行过程。

www.TopSage.com

在与客户建立了一个连接以后，Web服务器通常要给客户发送一个文件，比如一个Web页面。*TransmitFile*函数的实现与Windows缓存管理器集成在一起，所以，服务器可以直接从文件系统缓存中发送一个文件。按照这种方式来发送数据称为“**零拷贝**（*zero-copy*）”，因为服务器不必触及到文件数据就可以发送该文件；它只需简单地指定一个文件句柄和要发送的文件范围。而且，*TransmitFile*也允许一个服务器在文件数据之前或之后插入数据，因而服务器可以发送头信息（可能包含了Web服务器的名称）和一个专门的域（向客户指明当前所发送的消息的大小）。Windows捆绑的Web服务器，IIS（Internet Information Services），使用了*AcceptEx*和*TransmitFile*。

Windows XP和Windows Server 2003加入了许多其他的多功能API，包括*ConnectEx*、*DisconnectEx*和*TransmitPackets*。*ConnectEx*建立一个连接，并且在该连接上发送第一个消息。*DisconnectEx*关闭一个连接，并且允许该连接的套接字句柄在*AcceptEx*和*ConnectEx*调用中被重复使用。最后，*TransmitPackets*类似于*TransmitFile*，只不过它除了发送文件数据以外，也允许发送内存中的数据。

扩充Winsock

在Windows上，Winsock是一个可扩展的API，因为第三方可以加入一个**传输服务提供者**（*transport service provider*），由该传输服务提供者将Winsock与叠加在已有协议之上的其他协议或层连接起来，以便提供诸如代理（proxying）之类的功能。第三方也可以加入一个**名字空间服务提供者**（*namespace service provider*），以增强Winsock的名称解析功能。这些服务提供者利用Winsock的**服务提供者接口**（SPI，*service provider interface*）插入到Winsock中。当一个传输服务提供者被登记到Winsock中时，Winsock对于该传输服务提供者指明的它所实现的地址类型，使用该提供者来实现套接字函数，比如*connect*和*accept*。至于传输服务提供者如何实现这些函数，并没有专门的限制，不过，它实现的这些函数通常会涉及要与内核模式的一个传输驱动程序进行通信。

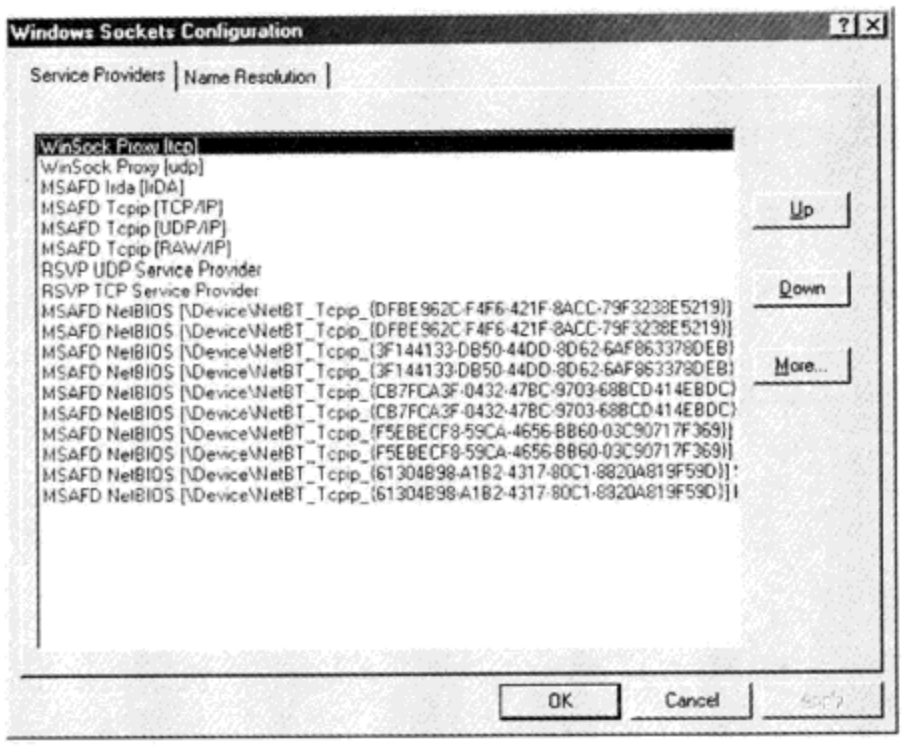
任何Winsock客户/服务器应用程序的一个基本需求是，服务器必须让它的地址可被客户所使用，这样客户才能够连接至该服务器。在TCP/IP协议上执行的标准服务使用“知名的地址”来确保它们的地址能被客户使用。只要浏览器知道了一个Web服务器当前运行所在的计算机的名称，它就可以通过指定该Web服务器的知名地址（该服务器的IP地址再拼接上:80，这里80是HTTP所使用的端口号），来连接到该Web服务器上。名字空间服务提供者使得服务器有可能用其他的方式来登记它们的存在状态。例如，一个名字空间服务提供者的工作方式可能是这样的：在服务器端，它将服务器的地址登记到活动目录中；在客户端，它在活动目录中查找该服务器的地址。名字空间服务提供者通过实现标准的Winsock名称解析函数，比如*getaddrinfo*（*gethostbyname*的一个替代）和*getnameinfo*，来为Winsock提供这种名称服务功能。

www.TopSage.com



实验：查看Winsock服务提供者

Platform SDK中包含的Windows套接字配置工具（Sporder.exe）显示了当前已经登记的Winsock传输服务提供者和名字空间服务提供者，它也允许你改变这些传输服务提供者被列举的顺序。例如，如果有两个TCP/IP传输服务提供者，则对于那些使用TCP/IP协议进行通信的Winsock应用程序而言，第一个列出的是默认提供者。下面是Sporder的一个屏幕截图，它显示了已经登记的传输服务提供者。



Winsock的实现

Winsock的实现如图13.4所示。它的应用程序接口是由一个API DLL构成的，即Ws2_32.dll（\Windows\System32\Ws2_32.dll）。Ws2_32.dll为应用程序提供了对于Winsock函数的访问，它通过调用名字空间服务提供者和传输服务提供者的服务，来执行名称和消息操作。Mswsock.dll库是一个传输服务提供者，它实现了Microsoft在Winsock中所支持的协议，并且，Mswsock.dll库利用*Winsock*辅助库（*Winsock helper*，辅助库是与特定的协议相关的）来跟内核模式的协议驱动程序进行通信。例如，Wshtcpip.dll是TCP/IP辅助库。Mswsock.dll（\Windows\System32\ Mswsock.dll）实现了Microsoft Winsock扩展函数，比如*TransmitFile*、*AcceptEx*和*WSARecvEx*。随Windows一起发行的有针对以下协议的辅助DLL：TCP/IP、基于IPv6的TCP/IP、AppleTalk、IPX/SPX、ATM和IrDA（Infrared Data Association，红外数据协会），以及针对以下协议的名字空间服务提供者：DNS （TCP/IP）、活动目录和IPX/SPX。

www.TopSage.com

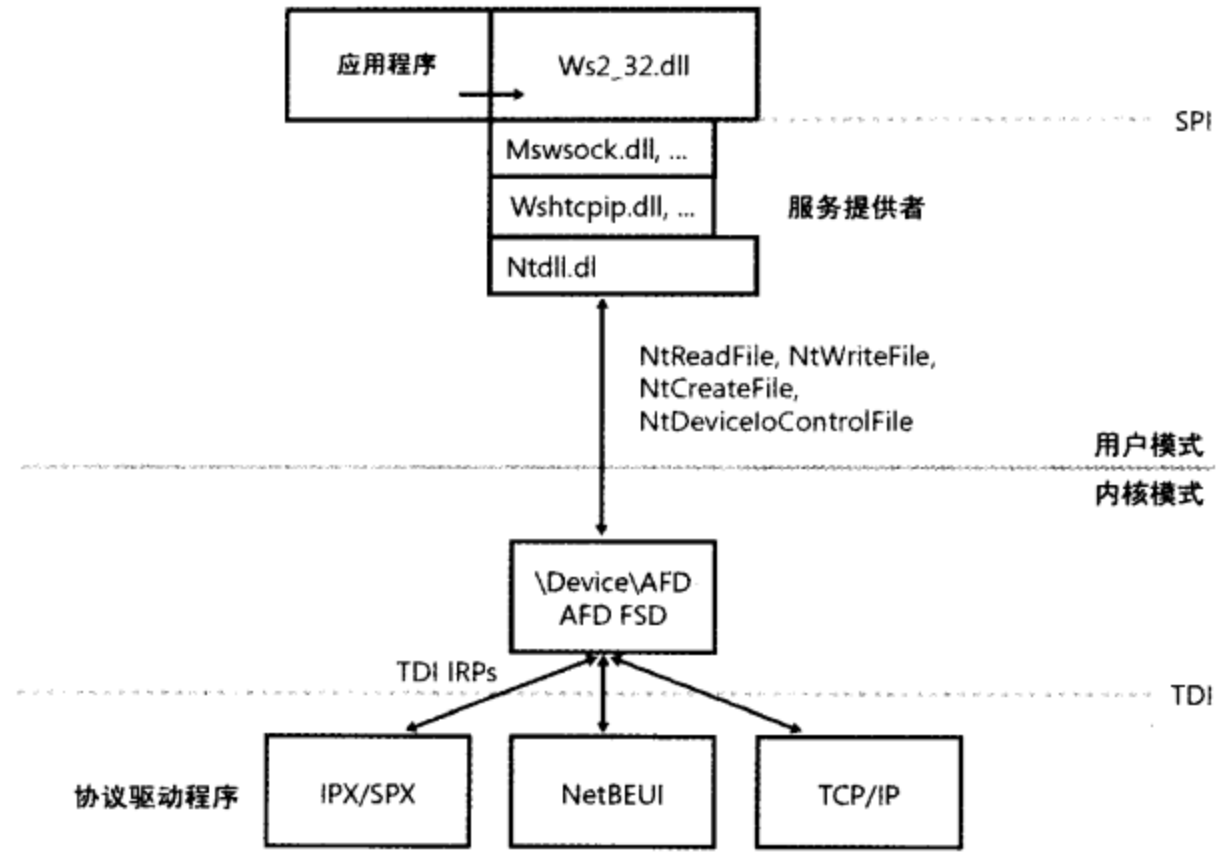


图13.4 Winsock的实现

如同命名管道和邮件槽API（在本章后面讲述）一样，Winsock也跟Windows I/O模型集成在一起，它使用文件句柄来表示套接字。这一支持要求得到一个内核模式文件系统驱动程序的帮助，所以Msafd.dll使用**辅助功能驱动程序**（AFD，*Ancillary Function Driver*，\Windows\System32\Drivers\Afd.sys）来实现基于套接字的函数。AFD是一个TDI客户，它通过向协议驱动程序发送TDI IRP来执行网络套接字操作，比如发送和接收消息。AFD并没有被编写成使用某些特定的协议驱动程序；相反地，Msafd.dll将每个套接字所使用的协议名称通知AFD，因而AFD可以打开代表该协议的设备对象。

Windows套接字直接访问

Windows套接字直接访问（WSD，Windows Sockets Direct）是一个接口，它使得Winsock应用程序可以在不需要修改的情况下充分发挥系统区域网络（SAN，System Area Network）的优势，这里的SAN也称为RDMA（Remote Direct Memory Access，远程直接内存访问）结构。SAN的高性能特征使得这种结构在从分布式计算到三层电子商务结构（见图13.5）中的应用都非常理想。在图13.5所描述的系统中，SAN将前端Web服务器与业务逻辑服务器连接起来，同时也将业务逻辑服务器与后端数据库服务器连接起来，从而为数据处理层提供高速数据传输服务。在Windows 2003和Windows 2000 Data Center Server或Windows 2000 Advanced Server SP2或更高的版本中，WSD已经可以使用了。

www.TopSage.com

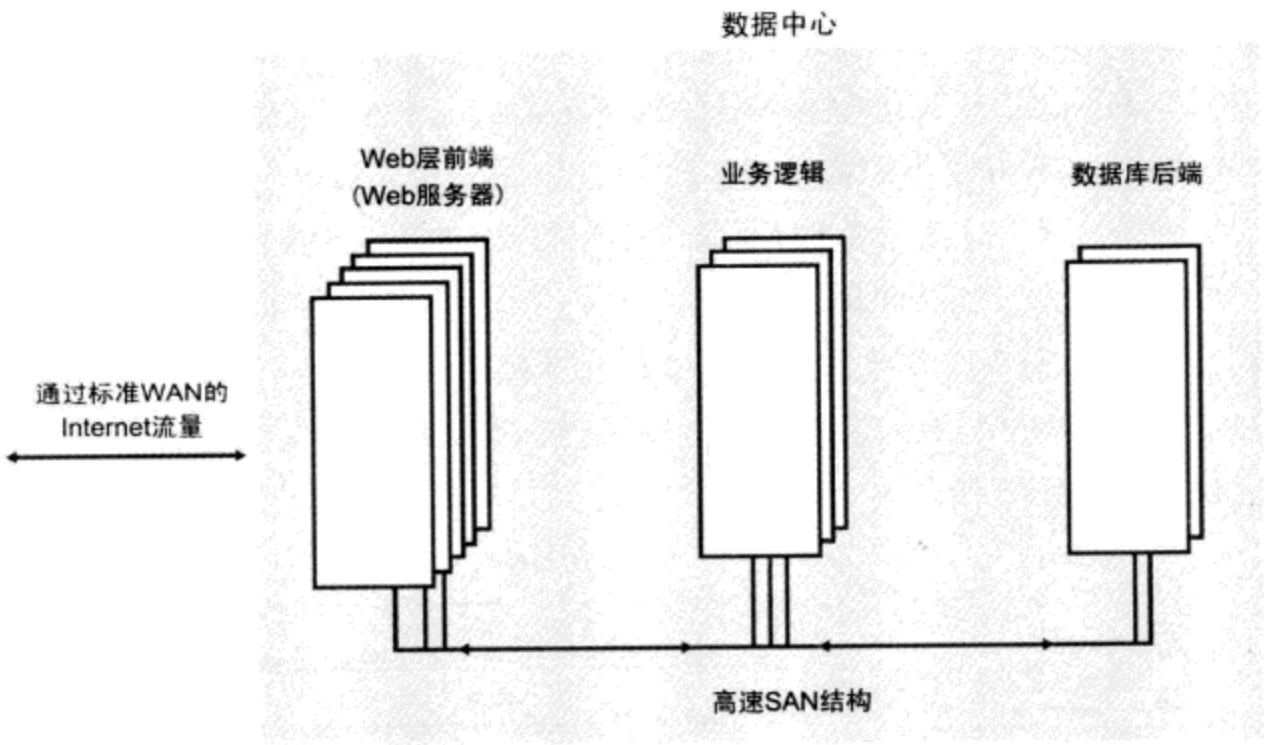


图13.5 在三层电子商务结构中SAN的使用

SAN连接 SAN通常依赖于特殊的网络连接和交换硬件来获得其很高的性能特征。常见的SAN连接类型有InfiniBand、千兆以太网、FiberChannel和一些私有的方案。在两台计算机之间共享的物理内存也可以被当作一种SAN连接方式。

SAN的交换硬件实现了一个不可路由的协议，它提供了类似于TCP的保证，比如可靠的、按序的消息递交。它们也支持一种称为远程直接内存访问（RDMA，Remote Direct Memory Access）的SAN特性，允许被传输的消息直接从源计算机的物理内存转移到目标计算机的物理内存中，通常在消息接收端发生的中间内存拷贝操作在这里不再需要。因此，RDMA免除了一次拷贝操作所消耗的CPU和内存总线带宽。

SAN的实现也允许应用程序绕过内核模式的组件，直接发送数据给用户应用程序，或者从用户应用程序接收数据。这使得应用程序执行的系统调用的数量尽可能地减到最少，从而进一步减少了在操作系统网络代码中所花的时间。

WSD结构 绝大多数SAN实现都要求应用程序做一些修改，以便与SAN网络协议建立接口连接，并且充分利用SAN硬件实现的网络协议和诸如RDMA之类的SAN特性，但是WSD（Windows套接字直接访问）允许任何基于Winsock的、使用TCP协议的应用程序可以在毫不修改的情况下利用SAN的特性。WSD得此名称的由来是，它使得应用程序可以绕过TCP/IP协议栈，直接访问SAN硬件。这一通向网络硬件的捷径给应用程序带来了2至2.5倍的性能提高。

www.TopSage.com

WSD之所以能获得这一通向SAN硬件的捷径，是因为它使用了Winsock DLL底下的一个软交换，如图13.6所示。这一软交换将SAN网络活动转发给一个由厂商提供的SAN Winsock服务提供者（WSP）。此WSP担当了NDIS驱动程序的用户模式等价模块，它能够将SAN硬件寄存器映射到用户模式内存中，然后无需经过内核模式组件就可以操纵SAN硬件。有些网络活动确实需要得到一个内核模式组件的帮助（例如，为了将硬件映射到用户模式内存中），此内核模式组件也是由SAN厂商来提供的。最后，SAN厂商也提供一个NDIS小端口驱动程序，从而将TCP/IP协议栈与SAN硬件连接起来，这对于那些用到了SAN本身不支持的Winsock网络特性的应用程序来说是很有必要的。

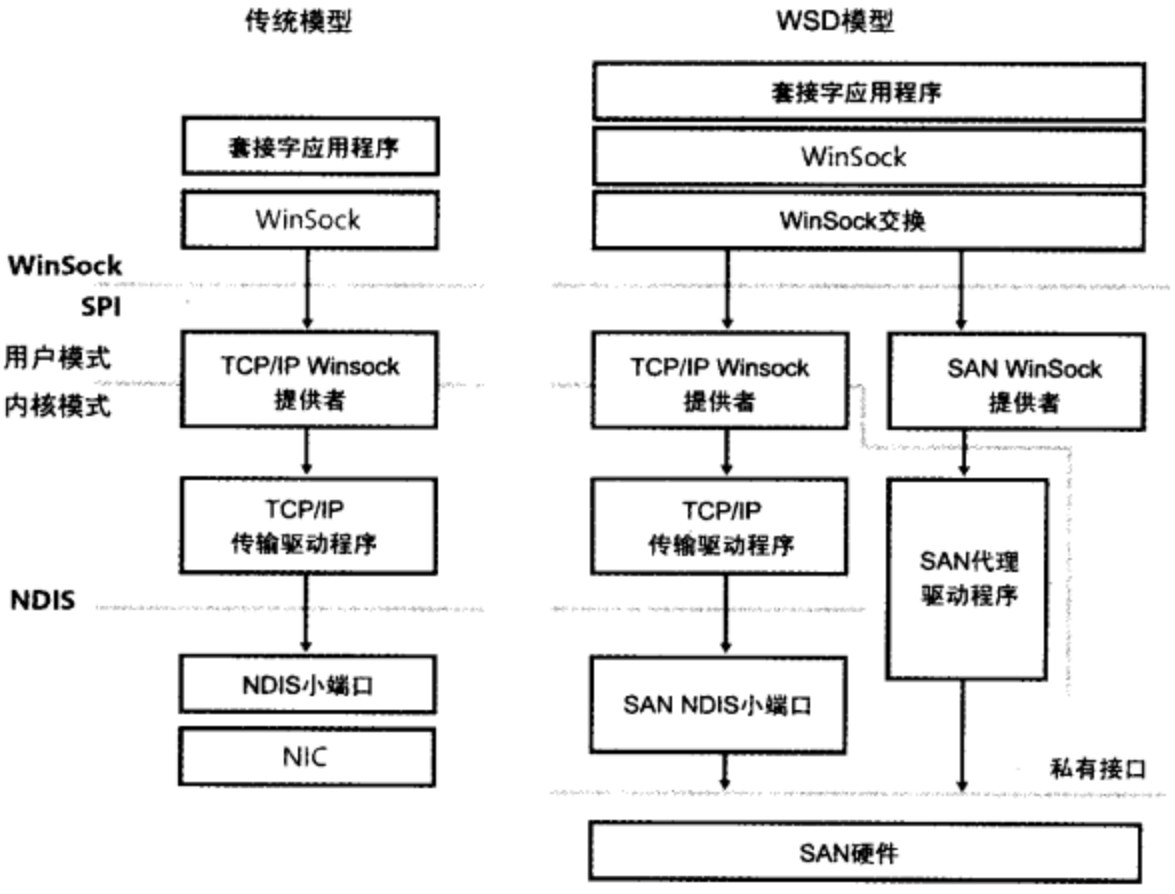


图13.6 传统的Winsock模型与WSD模型

远过程调用

远过程调用（RPC，Remote Procedure Call）是一个网络编程标准，最初是在20世纪80年代早期发展起来的。开放软件基金会（Open Software Foundation，现在是The Open Group）建立了分布式计算环境（DCE，Distributed Computing Environment）的分布式计算标准的RPC部分。尽管还存在另一个RPC标准，即SunRPC，但是，Microsoft的RPC实现是与OSF/DCE标准相兼容的。RPC建立在其他的网络API基础之上，比如命名管道或者Winsock，其目的是提供另一种“能在某种程度上向应用程序开发人员隐藏有关网络编程细节”的编程模型。

www.TopSage.com

RPC的操作

RPC设施是指这样的设施，它允许程序员创建一个包含许多过程的应用程序，其中有些过程在本地执行，其他的过程通过网络在远程的计算机上执行。RPC为各种网络操作提供了一种过程化的视图，而不是以传输为中心的视图，因而简化了分布式应用的开发。

在传统上，网络软件是以I/O处理模型为结构来组织的。例如，在Windows中，当一个应用程序发出一个远程I/O请求时，它就会激发一个网络操作。操作系统酌情进行处理，它将此请求转发给一个**重定向器**（*redirector*）。此重定向器担当起一个远程文件系统的角色，它使得客户可以与“对客户来说不可见的远程文件系统”进行交互。重定向器将该操作传递给远程文件系统，当远程文件系统处理完该请求并且返回结果以后，本地网卡激发一个中断。内核处理该中断，原来的I/O操作完成，处理结果被返回给调用者。

RPC采用了一种完全不同的方法。RPC应用程序就好像其他的结构化应用程序一样，它有一个主程序，主程序调用过程或过程库来完成特定的任务。然而，RPC应用程序和普通应用程序之间的区别在于，RPC应用程序中的有些过程库是在远程计算机上执行的（见图13.7），而其他的过程库则是在本地执行的。

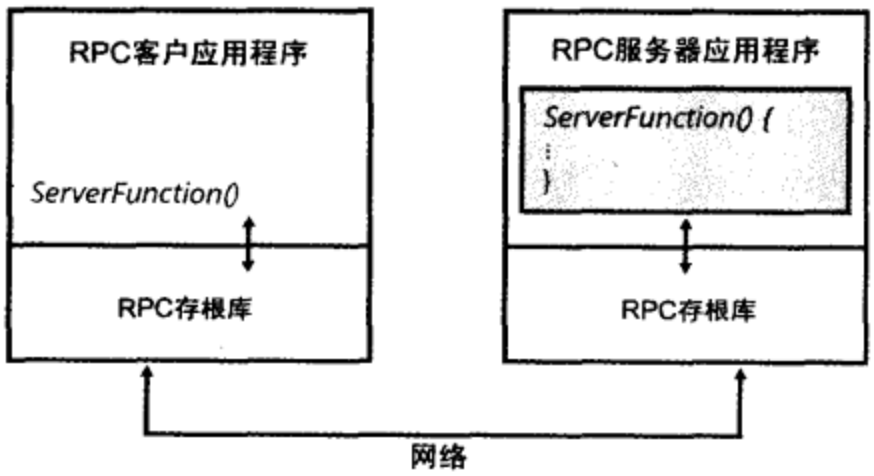


图13.7 RPC操作

对于RPC应用程序，所有的过程看起来都是在本地执行的。换句话说，不需要程序员主动地编写代码来跨网络传输计算类型的请求或者与I/O相关的请求，而是由RPC软件自动地完成诸如“处理网络协议、处理网络错误、等待结果”之类的任务。Windows的RPC设施可以在任何当前已被加载到系统中的传输器上进行工作。

为了编写一个RPC应用程序，程序员要确定哪些过程将在本地执行，哪些将在远程执行。例如，假设一台普通的工作站有一个网络连接，可以连接至一台Cray超级计算机，或者连接至一台专门为高速矢量操作而设计的机器。如果程序员正在编写的应用程序要操作大的矩阵，那么，从性能角度来看，程序员在编写此RPC应用程序时，将数学计算任务转移到远程计算机上显然是十分有意义的。

www.TopSage.com

RPC应用程序的工作方式如下：当应用程序运行时，它既调用本地过程，也会调用不在本地机器上的过程。为了处理后一种情形，应用程序被链接到一个包含**存根过程**（*stub procedure*）的本地静态链接库或者DLL，其中每个远程过程都有一个对应的存根过程。对于简单的应用程序，存根过程以静态方式与应用程序链接在一起，但是对于大的组件，存根被包含在单独的DLL中。在本章后面要讲述的DCOM中，通常采用后一种方法。存根过程与远程过程具有同样的名称，而且也使用同样的接口，但是，存根过程并不执行所请求的操作，相反地，它将传递给它的参数取出来，按照规范把它们转变成一个数据流（这一处理过程称为*marshal*）以便通过网络传输过去。将参数转变成一个数据流，意味着要按照特定的方式将参数排序并包装起来，以便适合于在网络链路上传输，比如把引用解析出来、为指针附加一份所指数据结构的拷贝。

然后存根调用RPC运行时过程，这些RPC运行时过程可以找到远程过程所在的计算机、确定该计算机使用哪些传输机制，以及利用本地传输软件将请求发送给远程计算机。当远程服务器接收到此RPC请求时，它将参数从数据流中解析出来（marshal的反过程，称为*unmarshal*），重新构造原来的过程调用，并且调用该过程。当服务器完成以后，它执行相反的序列，将结果返回给调用者。

除了以上描述的同步的函数调用接口以外，Windows RPC也支持**异步*RPC***（*asynchronous RPC*）。异步RPC使得一个RPC应用程序在执行一个函数时，不必“等待该函数完成才能继续向前执行”。相反地，该应用程序可以执行其他的代码，以后，当来自服务器的应答到达时，RPC运行时库会通知客户该操作已经完成。RPC运行时库使用客户所要求的通知机制。如果客户使用一个事件同步对象作为通知目标，那么它调用*WaitForSingleObject*或*WaitForMultipleObject*函数，以等待该事件对象变成有信号状态。如果客户提供一个异步过程调用（APC，Asynchronous Procedure Call），那么，RPC运行时库将该APC排队到执行此RPC函数的线程中。如果客户程序使用I/O完成端口作为它的通知机制，那么，它必须调用*GetQueuedCompletionStatus*来查询该函数的完成情况。或者另一种做法是，客户可以调用*RcpAsyncGetCallStatus*来主动查询远程过程的完成情况。

· Microsoft的RPC设施除了包含RPC运行时库以外，还包含一个编译器，称为MIDL（*Microsoft Interface Definition Language*）编译器。MIDL编译器大大地简化了一个RPC应用程序的创建工作。程序员编写一系列普通的函数原型（这些函数原型描述了远程例程，假设这是一个C或C++应用程序），并且将它们放到一个文件中。然后，程序员在这些原型上加入一些额外的信息，比如针对例程包（package）的网络惟一标识符和版本号，为例程中的参数指定输入、输出或者“输入/输出”属性。这些经过修饰的原型构成了开发人员的IDL（接口定义语言，Interface Definition Language）文件。

www.TopSage.com

一旦IDL文件已经被创建，程序员就可以利用MIDL编译器来编译该文件，从而生成前面提到过的客户端和服务器的存根例程，以及一些可被包含到应用程序中的头文件。当客户端应用程序被链接至这些存根例程文件时，所有的远程过程引用都将被解析到存根例程上。然后，远程过程也通过一个类似的过程被安装到服务器机器上。如果一个程序员只想调用一个已有的RPC应用程序，那么，它只需编写该软件的客户端，然后将客户端应用程序链接至本地的RPC运行时库设施即可。

RPC运行时库使用了一个通用的**RPC传输提供者接口**来跟一个传输协议进行通话。此提供者接口作为RPC设施与传输器之间的一个薄层，将RPC操作映射到由该传输器提供的函数上。Windows RPC设施实现了针对命名管道、NetBIOS、SPX、TCP/IP和UDP的传输提供者DLL。Windows Server 2003去掉了NetBIOS传输提供者，但加入了一个针对HTTP的传输提供者。利用类似的方式，RPC设施被设计为可与不同的网络安全设施一起工作。



> **注** 在Windows 2000中，你可以编写新的提供者DLL来支持额外的传输协议，但到了Windows XP以后，这种增加额外提供者DLL的支持被去掉了。

绝大多数Windows网络服务都是RPC应用程序，这意味着无论是本地进程，还是远程机器上的进程，都可以调用这些网络服务。因此，远程客户计算机可以调用服务器的服务来列出共享件、打开文件、写到打印队列中，或者激活服务器上的用户，或者它也可以调用消息投递服务（messenger service）来给你传递消息（当然，所有这一切都以服从安全限制为前提）。

服务器名称发布（*server name publishing*）是指一个服务器能够将它的名称注册到一个可供客户查找的场所，RPC也具有这种能力，并且与活动目录集成在一起。如果没有安装活动目录的话，则RPC名称定位器服务退回到使用NetBIOS广播的方式。这一行为确保了可与Windows NT 4系统进行互操作，而且也允许RPC可以在独立的服务器和工作站上使用。

RPC的安全性

Windows RPC实现了与安全支持提供者（SSP）集成的能力，所以，RPC客户和服务器可以使用经过认证的或者加密的通信。当一个RPC服务器想要安全的通信时，它告诉RPC运行时库，将哪一个认证服务加入到可供客户使用的**认证服务**（*authentication service*）列表中。当一个客户想要使用安全的通信时，它绑定到该服务器上。此时，它必须将它想要的认证服务以及**认证级别**（*authentication level*）告诉RPC运行时库。存在各种认证级别的目的是：确保只有授权的客户才可以连接到一个服务器，验证服务器接收到的每个消息都是从一个授权的客户发出的，检查RPC消息的完整性以检测消息是否被篡改，甚至加密RPC消息数据。很显然，越是高级的认证级别，越是要求更多的处理。客户也可以有选择地指定服务器的**安全个体名称**（*principal name*）。安全个体是指RPC安全系统识别的一个实体。服务器必须向一个SSP登记它的**安全个体名称**，此名称是与特定的SSP相关的。

www.TopSage.com

SSP不仅为RPC，同时也为Winsock负责处理在网络通信认证和加密过程中的细节。Windows包含了许多内置的SSP，包括Kerberos SSP和SChannel，其中Kerberos SSP实现了Kerberos版本5认证协议，而SChannel（安全通道，Secure Channel）实现了SSL（安全套接字层，Secure Sockets Layer）和TLS（传输层安全，Transport Layer Security）协议。如果指定的SSP不存在，则RPC软件使用底层传输器的内置安全性。有些传输器，比如命名管道和本地RPC（Local RPC），它们有内置的安全性。其他的传输器，像TCP，则没有安全性，在这种情况下，若指定的SSP不存在，则RPC执行不安全的调用。

RPC的另一个安全特性是，服务器通过*RpcImpersonateClient*函数，可以模仿客户的安全身份。当服务器完成了它代表该客户而执行的模仿的操作以后，它通过调用*RpcRevertToSelf*或*RpcRevertToSelfEx*函数，又回到它自己的安全身份下（关于模仿的更多信息，请参见第8章）。

RPC的实现

图13.8描述了RPC的实现，它显示了一个基于RPC的应用程序跟RPC运行时DLL（\Windows\System32\Rpcrt4.dll）链接在一起。RPC运行时DLL提供了参数打包（marshaling）和解包（unmarshaling）函数，供应用程序的RPC函数的存根使用；RPC运行时DLL也提供了相应的函数用于发送和接收打包后的数据。RPC运行时DLL也包含了一些支持例程，以处理基于网络的RPC和一种称为**本地*RPC***（*Local RPC*）的形式。本地RPC可被用于在同一个系统上的两个进程之间进行通信，RPC运行时DLL使用内核模式中的本地过程调用（LPC）设施作为本地网络API（有关LPC的更多信息，请参见第3章）。当RPC建立在非本地的通信机制基础之上时，RPC运行时DLL使用Winsock、命名管道或消息队列（Message Queuing，稍后讲述）API。



注 Windows Server 2003不支持将消息队列（Message Queuing）作为一种传输机制。

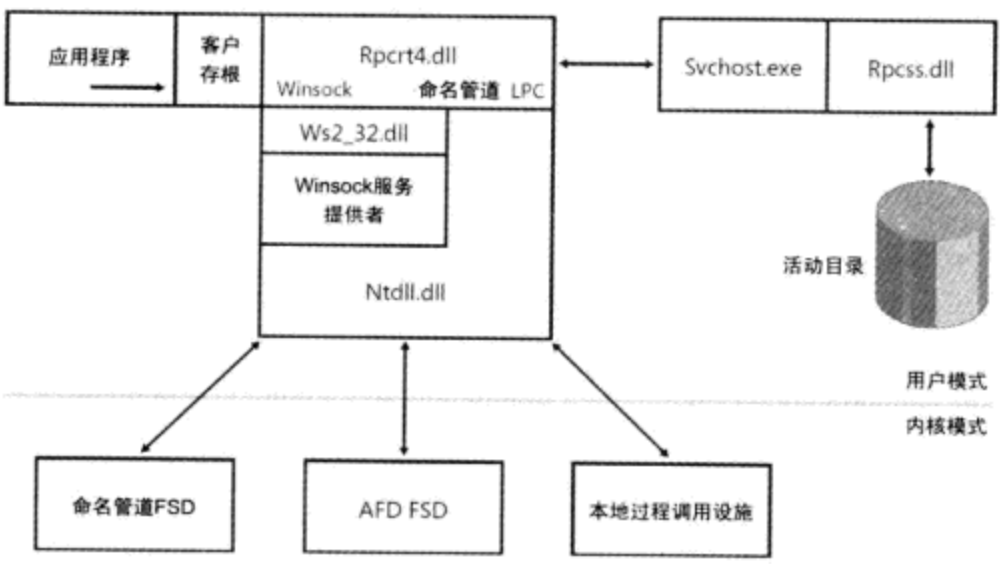


图13.8 RPC的实现

www.TopSage.com

RPC子系统（RPCSS，\Windows\System32\Rpcss.dll）被实现为一个Windows服务。RPCSS本身是一个RPC应用程序，它与另一个系统上的RPCSS实例进行通信，来完成名称查找、注册和动态端点映射（为了清楚起见，图13.8没有显示出RPCSS也链接了RPC运行时DLL）。

Web访问API

为了使Internet应用程序的开发更为方便，Windows同时提供了客户Internet API和服务器Internet API。应用程序利用这些API，无需掌握Gopher、FTP和HTTP协议的复杂细节知识，就可以提供和使用对应的服务。客户API包括Windows Internet和WinHTTP。Windows Internet也称为WinInet，它使得应用程序能够与Gopher、FTP和HTTP协议进行交互；WinHTTP使得应用程序可以与HTTP协议进行交互，在某些特定场合下它比WinInet更为合适。HTTP是一个服务器端API，它是在Windows Server 2003中引入的，目的是为了支持Web服务器应用程序的开发。

WinInet

WinInet支持Gopher、FTP和HTTP 1.0及1.1协议。该API被分解成一些子API集，分别针对每个具体的协议。应用程序使用与FTP相关的API，比如*InternetConnect*，与FTP服务器建立连接，使用*FtpFindFirstFile*和*FtpFindNextFile*可以列举一个FTP目录中的内容，使用*FtpGetFile*和*FtpPutFile*可以接收和发送文件。因此，应用程序的开发人员避免了有关与FTP服务器建立连接的细节，也不用关心如何将TCP/IP消息格式化成FTP协议。与Gopher相关的API以及与HTTP相关的API提供了类似的抽象层次，其中与HTTP相关的API提供了cookie持久性、自动拨号服务、客户端文件缓存，以及自动凭证对话处理的能力。Windows的核心组件，比如Explorer和Internet Explorer使用了WinInet。

WinHTTP

WinHTTP API的当前版本为5.1，在Windows 2000（安装了Service Pack 3）、Windows XP和Windows Server 2003上可以使用。它为HTTP客户应用程序提供了HTTP/1.1协议的抽象，类似于与WinInet HTTP相关的API所提供的协议抽象。然而，WinInet HTTP的目标是针对用户交互型的客户端应用程序，而WinHTTP API的设计目标是针对与HTTP服务器进行通信的服务器应用程序。服务器应用程序通常被实现为Windows服务，它没有提供用户界面，所以也不期望像WinInet API那样显示出对话框。而且，WinHTTP API更具伸缩能力，它也提供了安全功能，比如线程模仿，这是WinInet API所不具备的。

HTTP

服务器应用程序利用HTTP API可以进行注册，以接收来自某些特定URL的HTTP请求，接收HTTP请求，以及发送HTTP应答。Windows Server 2003实现了HTTP API。HTTP API包含了

www.TopSage.com

安全套接字层（SSL）的支持，所以，应用程序可以在安全的HTTP连接上交换数据。该API包含了服务器端的缓存能力、同步和异步I/O模型，以及IPv4和IPv6两种编址方式。IIS（Internet Information Services）版本6，即随Windows Server 2003一起发行的IIS版本，使用了HTTP API。

应用程序通过Httpapi.dll库可以访问HTTP API，而HTTP API依赖于内核模式的Http.sys驱动程序。当一个系统上第一次有应用程序调用*HttpInitialize*时，Http.sys便按需启动。应用程序使用*HttpCreateHttpHandle*来创建一个私有的请求队列，并且使用*HttpAddUrl*来指定它想要处理的URL。Http.sys使用请求队列以及为这些请求队列而注册的URL，从而允许多个应用程序在同一个给定的端口（比如端口80）上处理HTTP请求，每一个应用程序分别处理URL名字空间中的不同部分。

*HttpReceiveHttpRequest*接收针对已注册URL的进来HTTP请求，*HttpSendHttpResponse*发送HTTP应答。这两个函数都提供了异步操作支持，所以，应用程序可以使用*GetOverlappedResult*或I/O完成端口来确定何时一个操作已经结束。

应用程序可以使用Http.sys来将数据缓存在非换页的物理内存中，其做法是，调用*HttpAddToFragmentCache*并且在被缓存的数据上关联一个**段名称**（*fragment name*）。Http.sys调用内存管理器函数*MmAllocatePagesForMdl*来申请未映射的物理页面。当Http.sys要求为缓存中某一项所描述的物理内存执行一个虚拟地址映射时（比如，当它拷贝数据到缓存中，或者从缓存中发送数据时），它使用*MmMapLockedPagesSpecifyCache*，然后在完成了访问以后使用*MmUnmapLockedPages*。Http.sys一直维护着这些被缓存的数据，直到一个应用程序让它变成无效，或者一个由应用程序选择的、与该数据相关联的可选超时值到期为止。Http.sys也会在一个辅助线程中修剪被缓存的数据，当低内存通知事件变成有信号状态时，该线程被唤醒（有关低内存通知事件的信息，请参见第7章）。当一个应用程序在*HttpSendHttpResponse*调用中指定了一个或者多个段名称时，Http.sys传递一个指针（指向物理内存中被缓存的数据）给TCP/IP驱动程序，从而避免一次拷贝操作。

命名管道和邮件槽

命名管道和邮件槽是Microsoft最初为OS/2 LAN Manager开发的编程API，后来又被移植到Windows NT上。命名管道提供了可靠的双向通信，而邮件槽则提供了不可靠的单向数据传输。邮件槽的一个优势是，它们支持广播。在Windows中，这两个API都利用了Windows的安全性，因而使得服务器可以精确地控制哪些客户能够与它连接。

服务器分配给命名管道和客户的名称遵从Windows统一命名规范（UNC，Universal Naming Convention），这是一种与协议无关的标识Windows网络上资源的方法。本章后面将介绍UNC名称的实现。

www.TopSage.com

命名管道的操作

命名管道的通信是在一个命名管道服务器和一个命名管道客户之间进行的。命名管道服务器是指创建命名管道的应用程序，客户可以将它链接到它所创建的命名管道上。命名管道的名称的格式是\\Server\Pipe\PipeName。名称中的*Server*部分指定了此命名管道服务器当前执行所在的计算机（命名管道服务器不能创建远程系统上的命名管道）。此名称可以是一个DNS名称（比如，*mspress.microsoft.com*）、一个NetBIOS名称（*mspress*）或者一个IP地址（131.107.0.1）。名称中的*Pipe*部分必须是字符串“Pipe”，而*PipeName*则是分配给此命名管道的惟一名。命名管道名称中的惟一名部分可以包含子目录。\\MyComputer\Pipe\MyServerApp\ConnectionPipe是一个包含子目录的命名管道名称的例子。

命名管道服务器使用Windows函数*CreateNamedPipe*来创建一个命名管道。该函数的输入参数之一是一个指向命名管道名称的指针，其格式为\\.\Pipe\PipeName。这里“\\.\”是一个由Windows定义的别名，代表“此计算机”。该函数接受的其他参数有：一个可选的安全描述符用于保护对该命名管道的访问，一个标志指定了该命名管道应该是双向的还是单向的，一个值指明了该管道支持的并发连接的最大数目，以及一个标志指定了该管道应该按**字节模式**（*byte mode*）还是按**消息模式**（*message mode*）来工作。

绝大多数网络API只在字节模式下工作，这意味着，用一次发送操作来发送的消息可能要求接收方执行多次接收操作，然后由这些分段的数据建立起完整的原始消息。命名管道可以在消息模式下工作，从而简化了接收方的实现，因为在发送和接收操作之间有一对一的对应关系。因此，接收方每完成一次接收操作，就得到了一个完整的消息，它无需考虑与“跟踪和记录消息分段”相关的事宜。

针对某个特定名称的第一个*CreateNamedPipe*调用创建了该名称的第一个实例，也建立了所有具有该名称的命名管道的行为基础。服务器通过额外的*CreateNamedPipe*调用，可以创建额外的实例，直至达到第一个调用中指定的最大值为止。一个服务器在创建了至少一个命名管道实例以后，它执行Windows函数*ConnectNamedPipe*，这使得该服务器创建的命名管道可以与客户建立连接。*ConnectNamedPipe*既可以被同步执行，也可以被异步执行，而且，一直到有客户跟这一实例建立连接（或者有错误发生），此函数才完成。

命名管道客户使用Windows函数*CreateFile*或*CallNamedPipe*，以及指定服务器所创建的管道的名称，来跟服务器建立连接。如果该服务器已经执行了一个*ConnectNamedPipe*调用，那么，该客户的安全轮廓，以及它所请求的对该管道的访问模式（读、写），需要针对该命名管道的安全描述符进行验证（关于Windows所使用的安全检查算法的更多信息，请参见第8章）。如果该客户被授予了对一个命名管道的访问，那么它接收到一个句柄（代表了此命名管道连接的客户端），同时服务器的*ConnectNamedPipe*调用也随之完成。

www.TopSage.com

当一个命名管道连接被建立起来以后，客户和服务器可以使用Windows函数*ReadFile*和*WriteFile*来从该管道中读取数据，或者写数据到管道中。命名管道支持同步的和异步的消息传输操作。图13.9显示了一个服务器和客户正在通过一个命名管道实例进行通信。

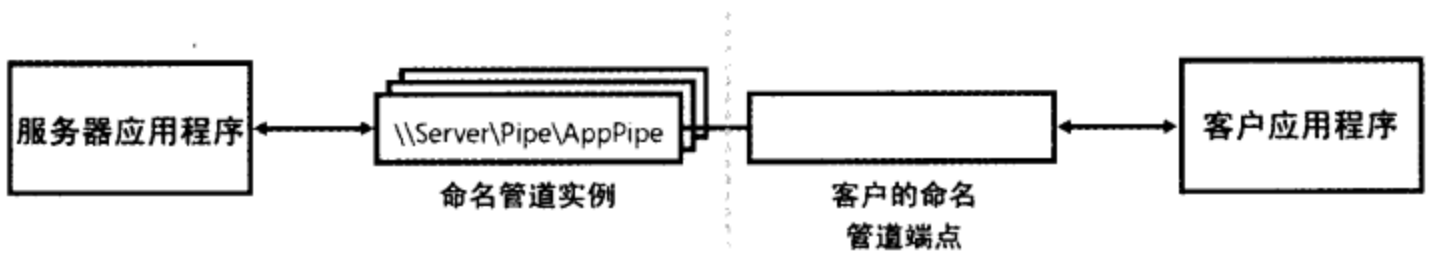


图13.9 命名管道的通信

命名管道网络API独有的一个特征是，它允许一个服务器通过使用*ImpersonateNamedPipeClient*函数来模仿一个客户。有关在客户/服务器应用程序中如何使用模仿的讨论，请参见第8章的“模仿”一节。

邮件槽的操作

邮件槽提供了一种不可靠的单向广播机制。一个使用这种通信类型的应用程序例子是时间同步服务，它可能每隔几秒钟在域范围内广播一次源时间。对于网络上的每台计算机来说，接收源时间消息并不是那么关键，所以这是一个很好的利用邮件槽作为通信机制的应用。

如同命名管道一样，邮件槽也跟Windows API集成在一起。邮件槽服务器使用*CreateMailslot*函数来创建一个邮件槽。*CreateMailslot*接受形如“\\.\Mailslot\MailslotName”的名称作为一个输入参数。而且，如同命名管道一样，邮件槽服务器只能创建它当前所在机器上的邮件槽，并且它分配给邮件槽的名称也可以包含子目录。*CreateMailslot*也接受一个安全描述符，并将其用于控制客户对该邮件槽的访问。*CreateMailslot*返回的句柄是**可重叠的**（*overlapped*），这意味着在该句柄上执行的操作，比如发送和接收消息，是异步的。

因为邮件槽是单向的和不可靠的，所以，*CreateMailslot*不像*CreateNamedPipe*那样带有那么多参数。服务器在创建了一个邮件槽以后，它只是简单地在代表该邮件槽的句柄上执行*ReadFile*函数，以便监听来自客户的消息。

邮件槽客户使用一种类似于命名管道客户所使用的命名格式，但稍有一些变化，以便有可能向客户所在域或者某个指定域中符合某一给定名称的所有邮件槽广播消息。为了向一个邮件槽的特定实例发送消息，客户调用*CreateFile*，并指定一个与特定计算机相关的名称。这种邮件槽名称的一个例子是“\\Server\Mailslot\MailslotName”（客户可以指定“\\.\”来代表本地计算机）。如果客户想要得到一个代表它所在域中符合某个给定名称的所有邮件槽的句柄，那么，

www.TopSage.com

它可以按照格式“*\Mailslot\MailslotName”来指定该名称；如果客户想要给另一个不同域中某个给定名称的所有邮件槽广播消息，则它使用的格式是“\\DomainName\ Mailslot\ MailslotName”。

在得到了代表邮件槽客户端的句柄以后，客户通过调用*WriteFile*，就可以发送消息了。由于邮件槽的实现方式的原因，只有小于425字节的消息才可以被广播。如果一个消息大于425字节，那么，邮件槽使用了可靠的通信机制，而这种可靠的通信机制要求端到端的客户/服务器连接，从而使得消息广播不再可能。而且，邮件槽的实现还有另一个怪癖，即它将任何425或426字节的消息截短成424字节。这些限制使得邮件槽通常不适合于广播大于424字节的消息。图13.10显示了一个客户向某个域内的多个邮件槽服务器广播消息的情形。

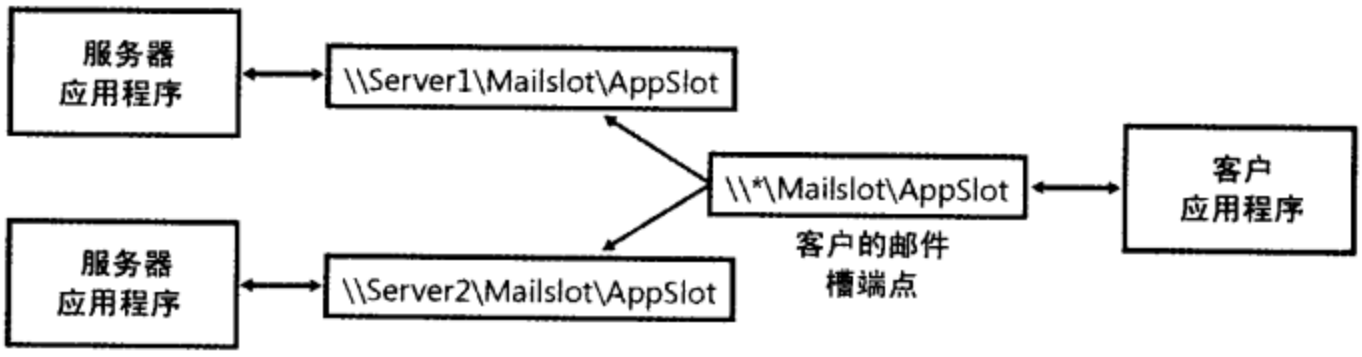


图13.10 邮件槽广播

命名管道和邮件槽的实现

命名管道和邮件槽与Windows紧密集成的一个最明显的证据是，它们的函数都是在Kernel32.dll（即Windows客户端DLL）中实现的。应用程序利用命名管道和邮件槽来发送和接收消息时所使用的函数，即*ReadFile*和*WriteFile*，也是主要的Windows I/O例程。而客户用来打开命名管道或邮件槽的*CreateFile*函数，也是一个标准的Windows I/O例程。然而，命名管道和邮件槽应用程序所指定的名称，则分别指定了由命名管道文件系统驱动程序（\Windows\System32\Drivers\Npfs.sys）和邮件槽文件系统驱动程序（\Windows\System32\Drivers\Msfs.sys）来管理的文件系统名字空间，如图13.11所示。命名管道文件系统驱动程序创建一个名为\Device\NamedPipe的设备对象，和一个名为\Global??\Pipe （在Windows 2000上为\??\Pipe）的符号链接指向该对象；邮件槽文件系统驱动程序创建一个名为\Device\Mailslot的设备对象，和一个名为\Global??\Mailslot的符号链接指向该对象（关于\Global??对象管理器目录的介绍，请参见第3章）。传递给*CreateFile*的名称形如\\.\Pipe\...和\\.\Mailslot\...，其前缀\\.\被翻译为\Global??\，所以这些名称被通过一个符号链接解析到一个设备对象上。两个专门的函数*CreateNamedPipe* 和 *CreateMailslot* 分别使用了对应的原生函数 *NtCreateNamedPipeFile* 和 *NtCreateMailslotFile*。

www.TopSage.com

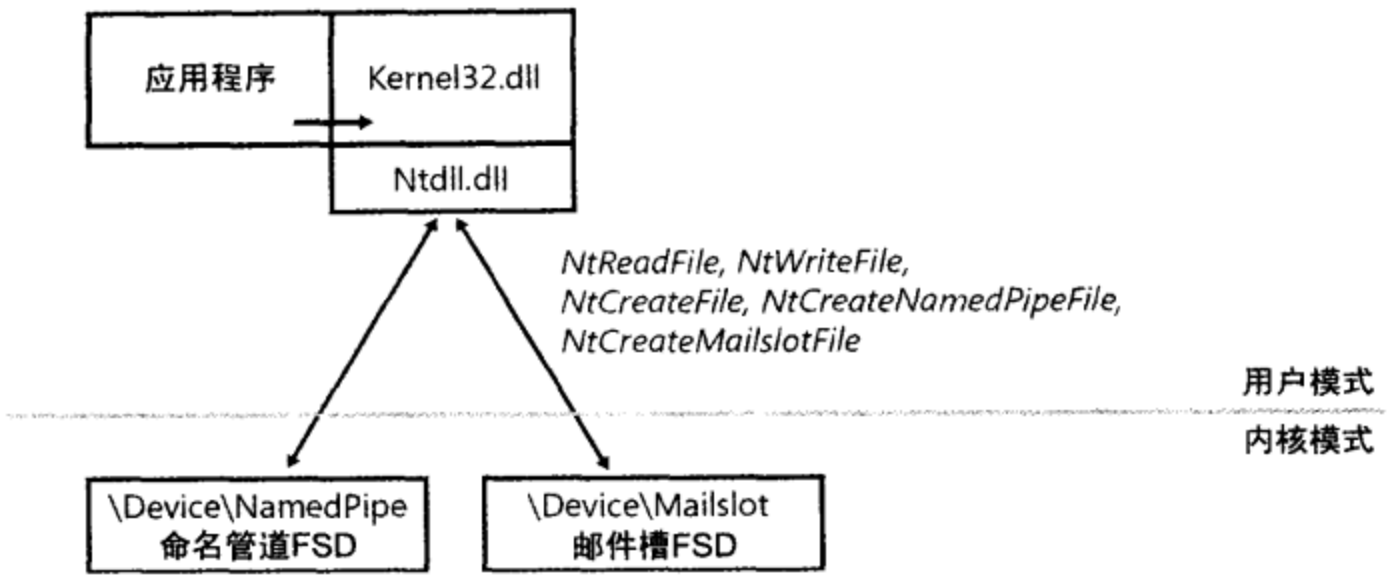


图13.11 命名管道和邮件槽的实现

在本章后面，我们将讨论：当一个指向远程命名管道和邮件槽的名称被解析到一个远程系统时，重定向器文件系统驱动程序是如何介入其中的。然而，当服务器创建或客户打开一个命名管道或邮件槽时，该命名管道或邮件槽所在机器上的某一个适当的文件系统驱动程序（FSD）最终将会被调用到。为什么要由内核模式的FSD来实现命名管道和邮件槽呢？有几个原因，其中主要的一个是，它们与对象管理器名字空间集成在一起，并且可以使用文件对象来表示被打开的命名管道和邮件槽。这种集成性可以带来以下几方面的好处：

- FSD使用内核模式的安全函数来为命名管道和邮件槽实现标准的Windows安全性；
- 应用程序可以使用*CreateFile*来打开一个命名管道或邮件槽，因为FSD与对象管理器名字空间集成在一起；
- 应用程序可以使用诸如*ReadFile*和*WriteFile*这样的Windows函数来跟命名管道和邮件槽打交道；
- 对于代表命名管道和邮件槽的文件对象，FSD依赖于对象管理器来跟踪其句柄和引用计数；
- FSD可以实现它们自己的命名管道和邮件槽名字空间，包括对子目录的支持。

因为命名管道和邮件槽的名称解析过程使用了重定向器FSD来跨网络进行通信，所以，它们间接地依赖于CIFS（Common Internet File System，公共Internet文件系统）协议。CIFS是通过TCP/IP、基于IPv6的TCP/IP和IPX协议来工作的，所以，只要在运行该应用程序的系统上至少有一个协议位列其中，应用程序就可以使用命名管道和邮件槽。有关CIFS的更多信息，请参见第12章。

www.TopSage.com



实验：列出命名管道名字空间，并观察命名管道的活动

要想使用Windows API来打开命名管道FSD的根，然后再列出该目录中的内容，这是不可能的，但是，你可以使用原生的API服务来做到这一点。来自*www.sysinternals.com*的PipeList工具可以向你显示出一台计算机上已定义的所有命名管道的名称，和每个名称的已创建的实例数目，以及在服务器调用*CreateNamedPipe*时指定的最大实例数目。以下是PipeList的一个例子输出：

```
C:\>pipelist

PipeList v1.01
by Mark Russinovich
http://www.sysinternals.com

Pipe Name                          Instances      Max Instances
---------                          ---------      -------------
TerminalServer\AutoReconnect           1              1
InitShutdown                           2             -1
lsass                                  4             -1
protected_storage                      2             -1
ntsvcs                                58             -1
scerpc                                 2             -1
net\NtControlPipe1                     1              1
net\NtControlPipe2                     1              1
ExtEventPipe_Service                   1             30
net\NtControlPipe3                     1              1
Winsock2\CatalogChangeListener-37c-0   1              1
net\NtControlPipe4                     1              1
net\NtControlPipe0                     1              1
DhcpClient                             1             -1
ProfMapApi                             2             -1
winlogonrpc                            2             -1
net\NtControlPipe5                     1              1
SfcApi                                 2             -1
net\NtControlPipe6                     1              1
Winsock2\CatalogChangeListener-4c4-0   1              1
atsvc                                  2             -1
epmapper                               2             -1
net\NtControlPipe7                     1              1
spoolss                                2             -1
wkssvc                                 3             -1
DAV RPC SERVICE                        2             -1
net\NtControlPipe8                     1              1
keysvc                                 2             -1
net\NtControlPipe9                     1              1
```

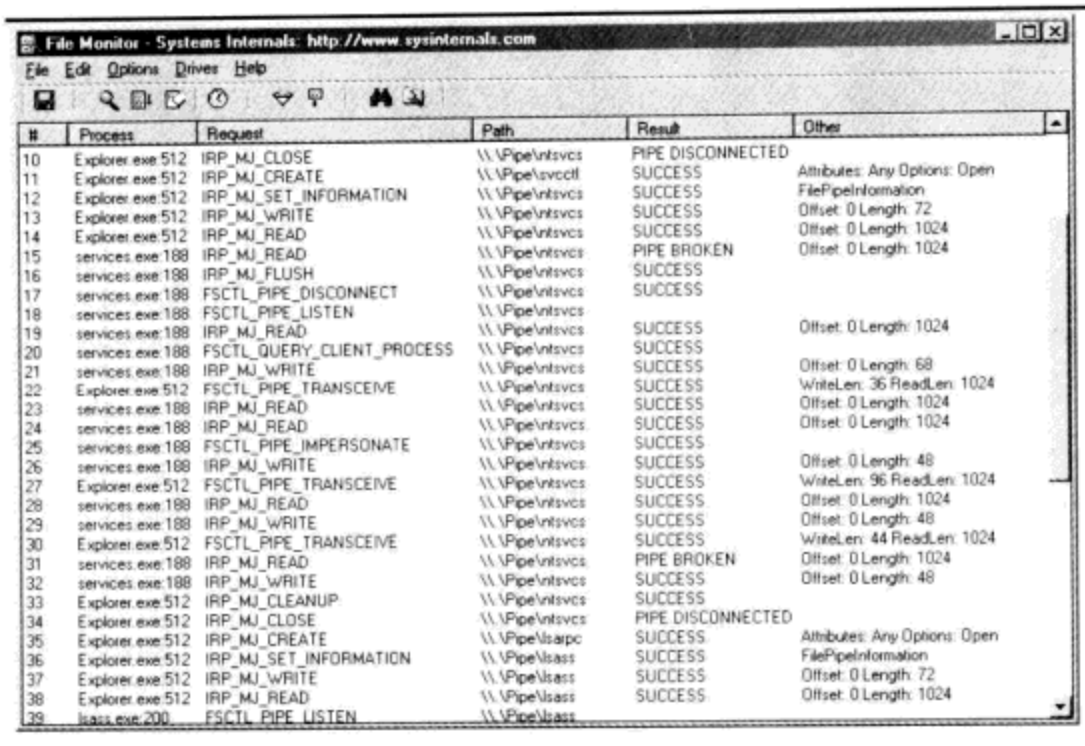

www.TopSage.com

```
PCHHangRepExecPipe                    1                8
PCHFaultRepExecPipe                   1                8
net\NtControlPipe10                   1                1
000000ac.000                          2               -1
Winsock2\CatalogChangeListener-ac-0   1                1
net\NtControlPipe11                   1                1
net\NtControlPipe12                   1                1
winreg                                2               -1
SECLOGON                              2               -1
srvsvc                                3               -1
ipsec                                 2               -1
trkwks                                2               -1
Winsock2\CatalogChangeListener-ac-1   1                1
net\NtControlPipe13                   1                1
vmware-authdpipe                      1               -1
W32TIME                               2               -1
net\NtControlPipe14                   1                1
net\NtControlPipe15                   1                1
Winsock2\CatalogChangeListener-e8-0   1                1
INETINFO                              2               -1
SMTPSVC                               2               -1
browser                               3               -1
net\NtControlPipe16                   1                1
Ctx_WinStation_API_service            2               -1
ExtEventPipe_s0                       1               30
ExtEventPipe_s0_poller                1               30
net\NtControlPipe17                   1                1
interbas\server\gds_db                1               -1
ROUTER                                8               -1
PIPE_EVENTROOT\CIMV2SCM EVENT PROVIDER 2              -1
net\NtControlPipe20                   1                1
```

从以上的输出中很清楚地看出，几个系统组件使用命名管道作为它们的通信机制。例如：*InitShutdown*管道是由Winlogon创建的，目的是为了接受远程停机命令；SecLogon管道是由SecLogon服务创建的，目的是代表Runas工具来执行登录操作。你可以在进程管理器（Process Explorer，可从*www.sysinternals.com*下载）中利用对象搜索功能来确定哪个进程打开了以上的每一个管道。请注意，最大实例数（Max Instances）为-1表明此名称的管道实例数量没有上限。

从*www.sysinternals.com*下载而得到的Filemon文件系统过滤型驱动程序（有关Filemon的更多信息，请参见第12章）也能够附载到Npfs.sys或Msfs.sys文件系统驱动程序上，从而可以看到系统中发生的所有命名管道或邮件槽的活动。从Filemon的Drivers菜单中选择Named Pipes或Mail Slots菜单项，就可以让Filemon附载到对应的驱动程序上。下面的屏幕截图显示了当桌面上的“My Network Places”图标被打开时Filemon捕获到的命名管道的活动。请注意观察在LSASS和工作站服务这两个命名管道上传输的消息。

www.TopSage.com



NetBIOS

在20世纪90年代，NetBIOS（网络基本输入/输出系统，Network Basic Input/Output System）编程API就已经成为PC上最为广泛使用的编程API。NetBIOS同时支持可靠的面向连接的通信和不可靠的无连接的通信。Windows支持NetBIOS是为了遗留下来的应用程序。Microsoft不鼓励应用程序开发人员再使用NetBIOS，因为其他的API，比如命名管道和Winsock，比它更加灵活和便于移植。在Windows上，NetBIOS是由TCP/IP和IPX/SPX协议支持的。

NetBIOS名称

NetBIOS依赖于一种专门的命名规范，在这种命名规范中，计算机和网络服务被赋予一个16字节的名称，称为NetBIOS名称。NetBIOS名称的第16个字节被当作一个修饰符，它可以指定该名称是一个单独的名称，还是某个组的一部分。只有单独的NetBIOS名称的实例才可以被赋予一个网络，但是多个应用程序可以分配同样的组名。客户可以通过给一个组名发送消息来实现广播。

为了支持与Windows NT 4系统和Consumer Windows的互操作，Windows自动为一个域定义一个NetBIOS名称，它是由管理员分配给该域的DNS（Domain Name System）名称的最左边部分的至多前15个字节构成的。例如，如果一个域被命名为*mspress.microsoft.com*，则该域的NetBIOS名称为*mspress*。类似地，Windows要求管理员在每台机器安装的时候为该机器分配一个NetBIOS名称。

NetBIOS用到的另一个概念是LANA（LAN adapter，LAN适配器）编号。LANA编号被分配给每一个“在一块网络适配器之上层并且与NetBIOS兼容”的协议。例如，如果一台计算机

www.TopSage.com

有两块网络适配器卡，并且TCP/IP和NWLink可以使用这两块适配器卡中的任何一块，则共有四个LANA编号。LANA编号很重要，因为NetBIOS应用程序必须显式地将它的服务名称分配给它所希望的每一个LANA，以便通过此LANA接受客户的连接。如果该应用程序在一个特定的名称上监听来自客户的连接，则客户只能通过“已经注册了该名称的网络适配器卡”上的协议来访问该名称。

本章后面的“Windows Internet名称服务”一节中讲述了从NetBIOS名称到TCP/IP地址的解析过程。

NetBIOS的操作

NetBIOS服务器应用程序使用NetBIOS API来列举一个系统上的LANA，并且将一个代表该应用程序所提供服务的NetBIOS名称分配到每一个LANA上。如果该服务器是面向连接的，那么它执行一个NetBIOS监听命令，以等待客户的连接请求到来。当客户被连接上以后，服务器执行NetBIOS函数来发送和接收数据。无连接的通信也类似，但服务器只是简单地读取消息，无需事先建立连接。

面向连接的客户使用NetBIOS函数来跟一个NetBIOS服务器建立连接，然后进一步执行NetBIOS函数来发送和接收数据。一个已被建立起来的NetBIOS连接也称为一个**会话**(*session*)。如果客户想要发送无连接的消息，那么，它在发送函数中只需指定该服务器的NetBIOS名称即可。

NetBIOS包含了许多函数，但是它们都通过同一个接口，即*Netbios*来转发。这种转发机制是一种历史遗留的结果，因为最初时，NetBIOS在MS-DOS上被实现为一个MS-DOS中断服务。那时候，NetBIOS应用程序执行一个MS-DOS中断，将一个数据结构传递给NetBIOS实现；该数据结构指定了将要被执行的命令的每一个方面。结果是，Windows中的*Netbios*函数也只带一个参数，该参数的数据结构包含了与该应用程序请求的服务相关的所有参数信息。



实验：使用*Nbtstat*来查看NetBIOS名称

你可以使用Windows中包含的*Nbtstat*命令来列出一个系统中活动的会话、被缓存在计算机上的从NetBIOS到TCP/IP的名称映射，以及在一台计算机上定义的NetBIOS名称。下面是一个使用*Nbtstat*命令的例子，其中的*-n*选项指定了列出该计算机上定义的NetBIOS名称：

```
C:\>nbtstat -n
Local Area Connection:
Node IpAddress: [10.1.23.147] Scope Id: []
                    NetBIOS Local Name Table
    Name               Type          Status
---------------------------------------------
      MARKLAP          <00>  UNIQUE       Registered
   Registered        MARKLAP            <20>  UNIQUE     Registered
   WORKGROUP         <1E>  GROUP        Registered
   Registered     ..__MSBROWSE__.<01> GROUP
Wireless Network Connection:
Node IpAddress: [0.0.0.0] Scope Id: []
    No names in cache
```


www.TopSage.com

NetBIOS API的实现

实现NetBIOS API的组件如图13.12所示。*Netbios*函数是通过\Windows\System32\Netapi32.dll被导出给应用程序的。Netapi32.dll打开一个指向名为*NetBIOS仿真器*（*NetBIOS emulator*）的内核模式驱动程序（\Windows\System32\Drivers\Netbios.sys）的句柄；它代表应用程序向该驱动程序发出Windows *DeviceIoControl*文件命令。NetBIOS仿真器将应用程序发过来的NetBIOS命令转译成TDI命令，再发送给协议驱动程序。

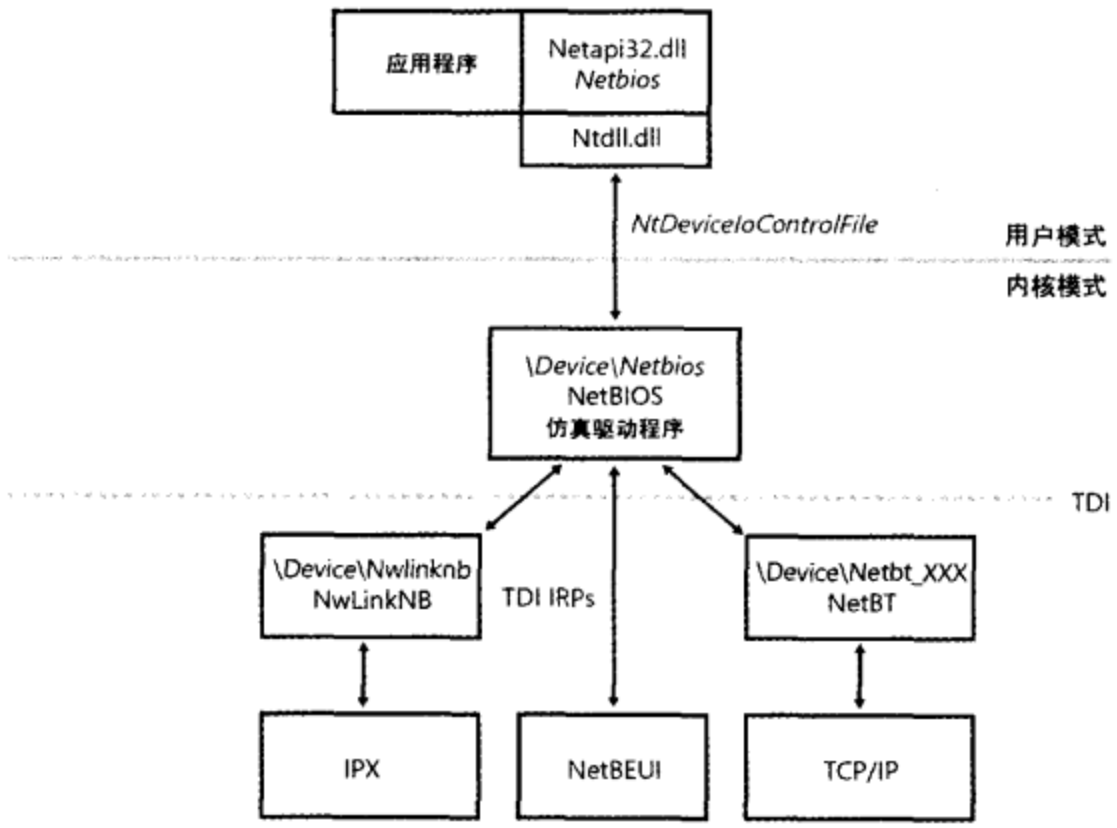


图13.12 NetBIOS API的实现

如果应用程序想要在TCP/IP协议上使用NetBIOS，则NetBIOS仿真器要求NetBT驱动程序（\Windows\System32\Drivers\Netbt.sys）必须存在。NetBT也称为“基于TCP/IP的NetBIOS（NetBIOS over TCP/IP）”驱动程序，它负责支持NetBIOS语义。NetBIOS的语义更加符合NetBIOS扩展用户接口（NetBEUI，NetBIOS Extended User Interface）协议（包含在以前的Windows版本中），而并不符合TCP/IP协议。例如，NetBIOS依赖于NetBEUI的消息模式传输设施和NetBIOS名称解析设施，所以，NetBT驱动程序在TCP/IP协议上实现这些设施。类似地，NwLinkNB驱动程序在IPX/SPX协议上实现NetBIOS语义。

其他的网络API

Windows还包含了其他一些网络API，它们用得没有那么频繁，或者它们建立在前面已经

www.TopSage.com

讲述的那些API层次之上。（这已经超出了本书的范围）。然而，其中四个API，即实时通信（RTC，Real-Time Communications）、DCOM（Distributed Component Object Model，分布式组件对象模型）、消息队列（Message Queuing）和通用即插即用（UPnP，Universal Plug and Play），对于Windows系统和许多应用程序的运行相当重要，因而值得简短地描述一下。

实时通信（RTC）

Windows XP和Windows Server 2003提供了RTC客户API，它使得开发人员可以创建应用程序以建立集成的多模型通信。通过开发这样的应用程序，可以使PC变成家庭或商务通信的中心。音频和视频呼叫，以及即时消息（IM，Instant Messaging）和协作应用，都可以集成到PC上的单个通信会话中。除了PC到PC的会话以外，该API也可以建立PC到电话的呼叫、电话到电话的呼叫，或者仅仅文本的IM会话。在PC到PC的会话上，也可以使用应用程序共享和白板的功能。

RTC支持在场信息（presence information），以便允许客户通过一个注册服务器来呼叫联系人（或好友），注册服务器维护了当前可以联系的成员的场所信息。联系人的场所可以是一台PC，或者一部电话，在将来，还可以是无线电话、寻呼机或者手持设备。例如，如果应用程序试图连接到一个联系人的工作场所，而在场信息表明了该联系人可以通过其家里的一台PC联系到，那么，RTC自动将该连接重定向到此场所。RTC API也允许用户维护其隐私，它可以阻止某些特定的呼叫者获得他们的在场信息。

DCOM

Microsoft的COM API使得应用程序可以是由不同的组件构成的，每个组件是一个可替换的自包含模块。COM对象导出一个面向对象的接口，其中包含了用于维护该对象内部数据的方法。因为COM对象呈现给外部的是定义良好的接口，所以，开发人员可以实现新的对象来扩展已有的接口，并且动态地利用新的支持来更新应用程序。

DCOM扩展了COM，它允许一个应用程序的组件驻留在不同的计算机上。这意味着应用程序不需要关心“一个COM对象可能在本地计算机上，而另一个COM对象可能要跨越LAN才能访问”。因此，DCOM提供了位置透明性，从而简化了分布式应用程序的开发。DCOM不是自包含的API，它需要依赖于RPC来完成其工作。

消息队列（Message Queuing）

消息队列是一个用于开发分布式应用程序的通用平台，它充分利用了松耦合的消息传递机制。因此，消息队列既是一个API，也是一个消息传递设施。它的灵活性来自于如下事实：它的队列被用做消息仓库，发送者可以替接收者将消息排队到仓库中，而接收者可以自行决定将消息提取出来。发送者和接收者不需要建立连接就可以使用消息队列，它们甚至不需要同时在执行，这使得发送者和接收者可以在非连接的情况下进行异步消息交换。

www.TopSage.com

关于消息队列机制，一个值得注意的特性是，它与Microsoft Transaction Server（MTS）和SQL Server集成在一起，所以，它可以参与到由MS DTC（Microsoft Distributed Transaction Coordinator）协调的事务中。利用MS DTC和消息队列，你可以为三层应用程序开发出可靠的事务功能。

UPnP

通用即插即用（Universal Plug and Play）是一种针对智能家电、设备和**控制点**（*control point*）的对等网络连接的体系结构。它的设计目标是，将易用的、灵活的、基于标准的连接特性带入到ad-hoc的、受管理的或不受管理的网络中，无论这些网络是在家庭里，小企业中，还是直接连接到Internet上。通用即插即用是一个分布式的、开放的网络体系结构，它使用现有的TCP/IP和Web技术来实现无缝的邻近网络连接，以及网络设备之间的控制传递和数据传输。

通用即插即用为来自各种厂商的各种设备类型，提供了零配置、对不可见联网和自动发现的支持。这使得一个设备可以动态地加入一个网络，获得一个IP地址，以及根据需要报告它的能力。然后，其他的控制点可以利用UPnP技术和控制点API（Control Point API）来获得其他设备的在场信息和能力信息。当一个设备不再使用时，它可以平稳地、自动地离开一个网络。

13.3 多重定向器支持

应用程序可以用两种方法来检查或访问远程系统上的资源：第一种方法是，使用UNC标准与Windows函数来直接操作一个远程资源；第二种方法是，使用WNet（Windows Networking）API来列举计算机，以及这些计算机导出来供共享的资源。这两种途径都使用了重定向器的能力来找到它们的网络路径（第12章讲述了CIFS）。正如我们以前所说明的，为了从客户处访问CIFS服务器，Microsoft提供了一个CIFS重定向器，它有一个称为重定向器FSD（redirector FSD）的内核模式组件和一个称为工作站服务（Workstation service）的用户模式组件。Microsoft也提供了另一个重定向器，它可以访问由Novell NetWare服务器共享出来的资源。第三方可以将他们自己的重定向器加入到Windows中。在这一节中，我们将讨论这样的软件：当远程I/O请求被发出时，如何确定应该调用哪个重定向器。下面是所涉及的组件。

- **多提供者转发器**（MPR，*Multiple provider router*） 它是一个DLL（\Windows\System32\ Mpr.dll），它确定当一个应用程序使用Windows的WNet API来浏览远程文件系统时应该访问哪个网络。
- **多UNC提供者**（MUP，*Multiple UNC Provider*） 它是一个驱动程序（\Windows\System32\Drivers\Mup.sys），它确定当一个应用程序使用Windows I/O API来打开远程文件时，应该访问哪个网络。

www.TopSage.com

多提供者转发器

Windows WNet函数允许应用程序（包括Windows Explorer的My Network Places）连接到诸如文件服务器和打印机这样的网络资源上，也可以浏览任何类型的远程文件系统的内容。因为通过调用WNet API可以做到在不同网络之间使用不同传输协议进行工作，所以，必须存在相应的软件，以便正确地发送请求到网络上，以及理解远程服务器返回的结果。图13.13显示了重定向器软件负责这些任务。

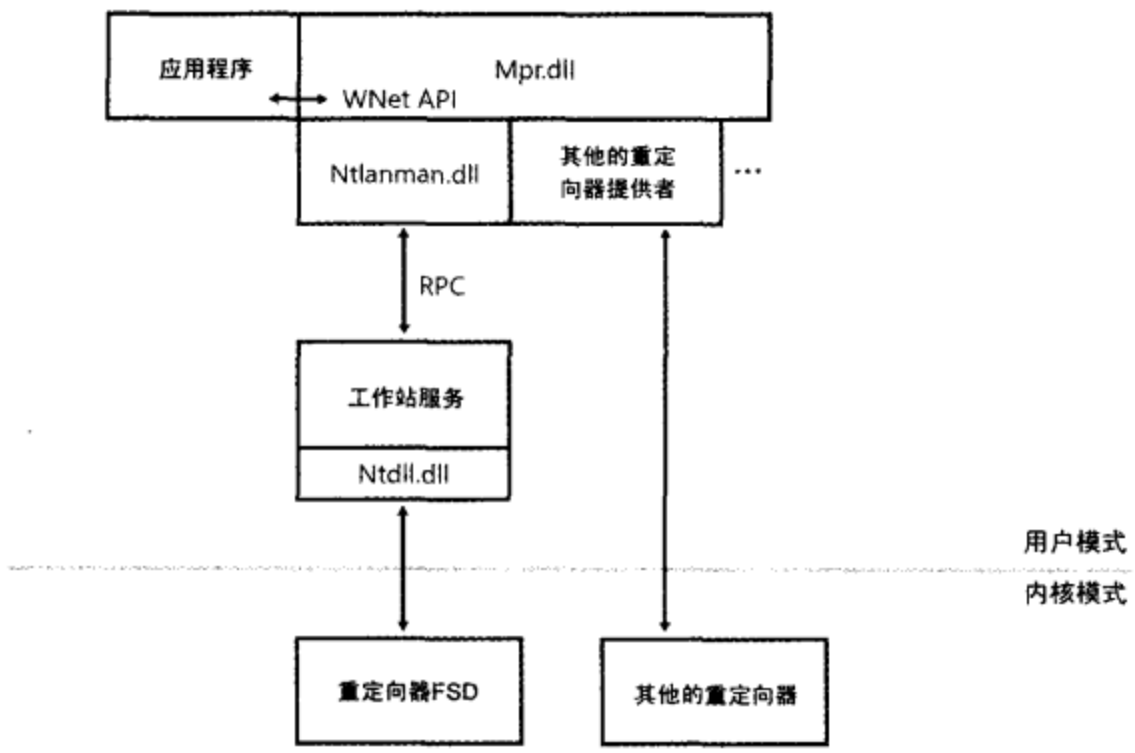


图13.13 MPR组件

这里的**提供者**（*provider*）使得当前Windows系统成为远程网络服务器的一个客户，它建立起相应的客户环境。WNet提供者执行的一部分操作包括建立和断开网络连接、远程打印，以及传输数据。内置的WNet提供者是由一个DLL、工作站服务和重定向器组成的。其他的网络厂商只需要提供一个DLL和一个重定向器。

当应用程序调用一个WNet例程时，该调用被直接传递给MPR.DLL。MPR接受该调用，并确定哪个WNet提供者能识别将要被访问的资源。MPR下面的每个提供者DLL都提供了一组标准函数，这些函数合起来称为**提供者接口**（*provider interface*）。该接口使得MPR可以确定当前的应用程序正在试图访问哪一个网络，以及将客户的请求转发给正确的WNet提供者软件。工作站服务的提供者是\Windows\System32\Ntlanman.dll，这是由HKLM\SYSTEM\CurrentControlSet\Services\lanmanworkstation\NetworkProvider注册表键下的ProviderPath值指示的。

www.TopSage.com

当通过调用*WNetAddConnection* API函数来连接到一个远程网络资源时，MPR检查HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\HwOrder\ProviderOrder注册表值，以确定应该加载哪些网络提供者。它按照这些网络提供者在注册表中被列出的顺序，每次询问一个，直到有一个重定向器能识别该资源，或者所有可用的提供者都被询问过为止。通过如图13.14所示的Advanced Settings对话框，你可以改变ProviderOrder值（此屏幕截图所在的计算机系统上，只安装了一个提供者）。如果你的系统是Windows 2000，或者将Start菜单配置成Classic设置，那么，从Network Connections文件夹的Advanced菜单中可以访问此对话框。你也可以右键点击桌面上的My Network Places，然后从弹出菜单中选择Properties，就可以访问Network Connections文件夹；或者，你可以从Start菜单的Settings选项中选择Network Connections文件夹。访问此对话框的另一种方法是，在控制面板中打开Network Connections小程序，点击你想要查看的连接，然后点击Advanced菜单中的Advanced Settings。

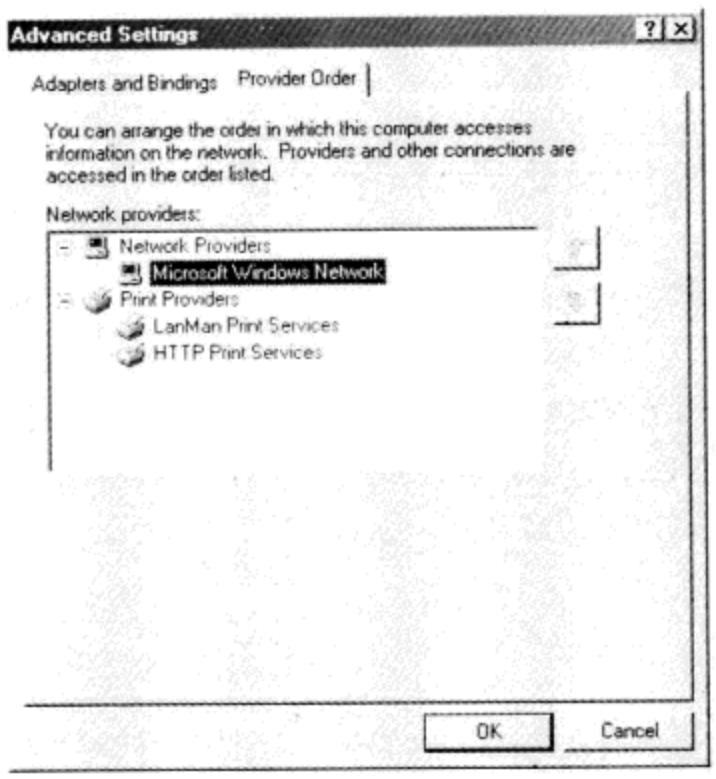


图13.14 提供者顺序编辑器

*WNetAddConnection*函数也可以为一个远程资源分配一个驱动器字母或者设备名称。当*WNetAddConnection*函数被调用来做这件事情的时候，它将该调用转发给正确的网络提供者。然后，该提供者相应地在对象管理器名字空间中创建一个符号链接对象，将所定义的驱动器字母映射到针对该网络的重定向器（即远程FSD）上。

图13.15显示了一个Windows 2000系统上的\??目录（关于\??目录的信息，请参见第3章），你可以看到，有几个驱动器字母代表了远程文件共享体的连接。此图表明：该重定向器创建了一个名为\Device\LanmanRedirector的设备对象；在符号链接值中包含的附加文字向重定向器指

www.TopSage.com

明了该驱动器字母对应于哪一个远程资源。当一个用户打开X:\Book\Chap13.doc时，重定向器接收到用符号链接表达的路径的未被解析部分，即“;X:0\dual\e\Book\Chap13.doc”。重定向器注意到被访问的资源位于服务器*dual*的共享体E上。

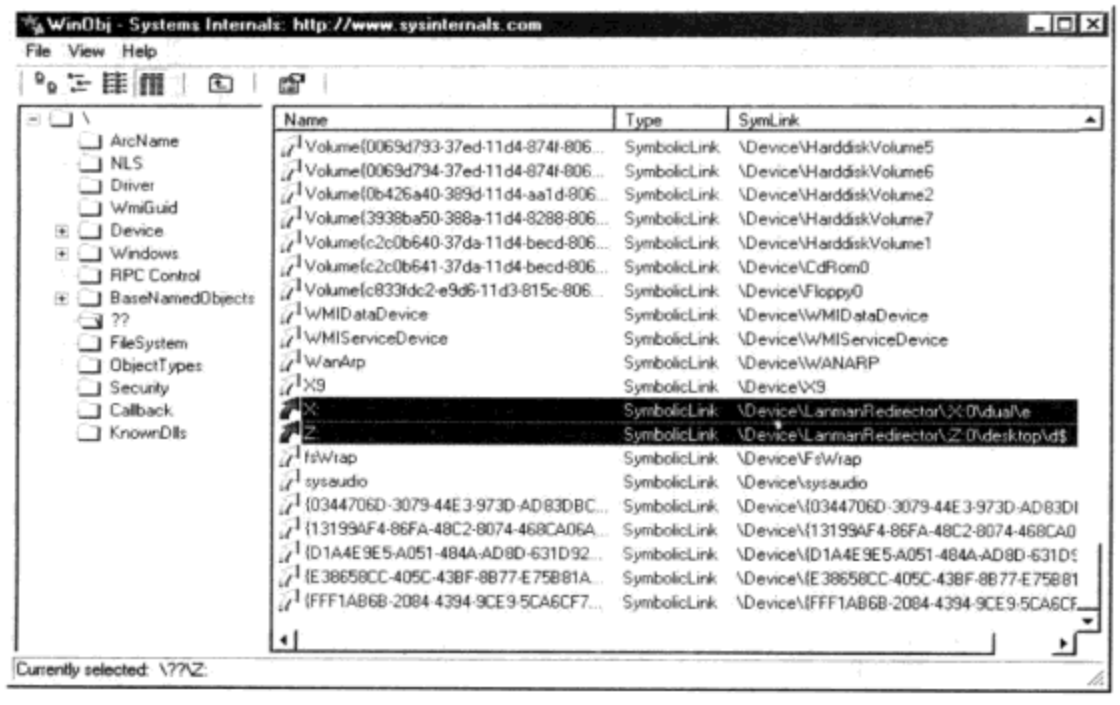


图13.15 解析一个网络资源名称

如同内置的重定向器一样，其他的重定向器在被加载到系统中并执行初始化时，也在对象管理器名字空间中创建一个设备对象。然后，当WNet或者其他的API调用对象管理器来打开一个位于其他网络上的资源时，对象管理器利用该设备对象作为跳转点，进入到远程文件系统中。它调用一个与该设备对象相关联的I/O管理器Parse方法，以便找到可以处理该请求的重定向器FSD（有关文件系统驱动程序的更多信息，请参见第12章）。

多UNC提供者

多UNC提供者（MUP）是一个类似于MPR的网络组件。它的处理范围是，针对“具有UNC名称（以字符\\开头的名称，表明了该资源位于网络上）的文件和设备”的I/O请求。MUP接受这样的请求，并且如同MPR一样，确定本地哪个重定向器可以识别此远程资源。但是，与MPR不同的是，MUP是一个设备驱动程序（在系统引导时刻被加载到系统中），它给低层的驱动程序发出I/O请求。在这里，低层的驱动程序是重定向器，如图13.16所示。Mup.sys也包含了分布式文件系统（Distributed File System，本章后面将讲述DFS）的客户端实现。DFS客户在默认情况下是启用的，但是，只要将DWORD注册表值HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Mup\DisableDfs设置为1就可以禁用DFS客户。

当MUP在引导过程中被加载进来时，它创建一个名为\Device\Mup的设备对象。当一个网络重定向器，比如CIFS，加载进来时，它创建一个命名的设备对象（例如\Device\LanmanRedirector），

www.TopSage.com

并且通过调用*FsRtlRegisterUncProvider*函数，向MUP注册其自身作为一个UNC提供者。如果该重定向器是第一个注册的重定向器，并且MUP的DFS客户支持被禁用的话，那么，FsRtlRegisterUncProvider创建一个符号链接名\??\UNC，并使它指向该重定向器的设备对象；否则，MUP建立符号链接名\Global??\UNC（在Windows 2000上为\??\UNC），并使它指向其自己的设备对象\Device\MUP。

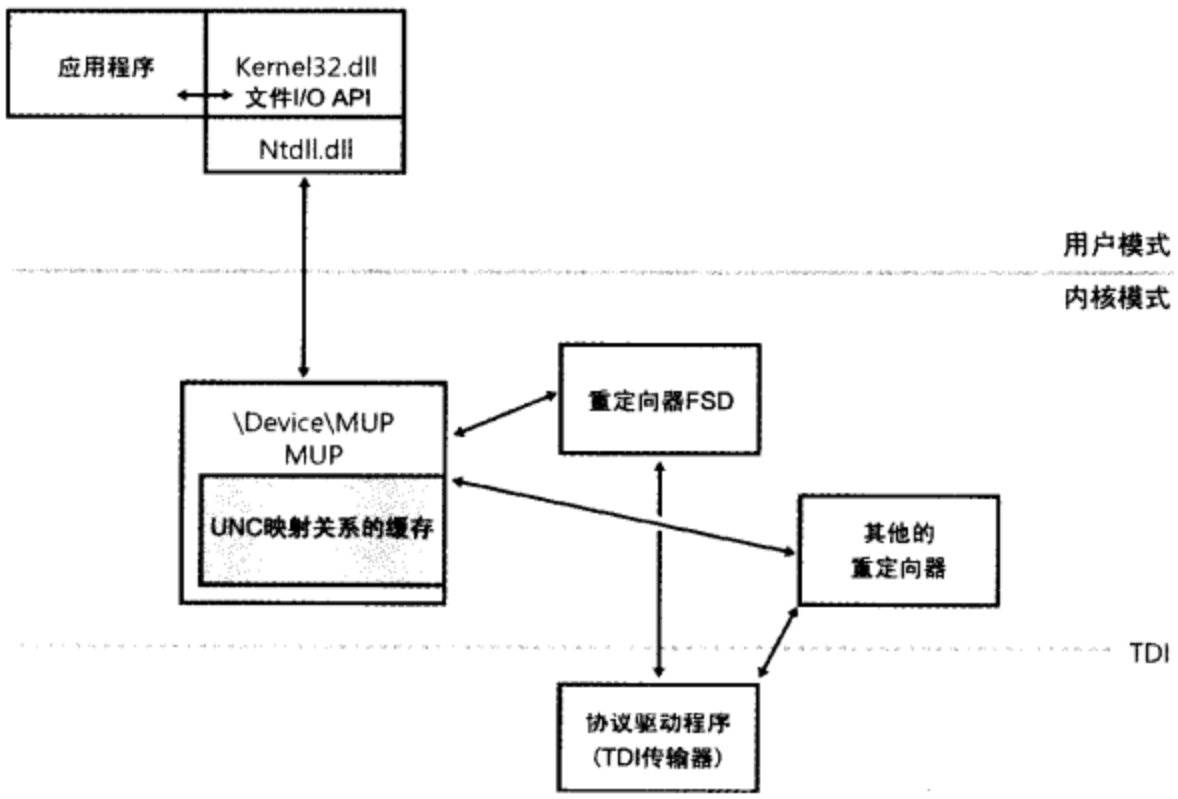


图13.16 多UNC提供者（MUP）

当应用程序第一次试图通过指定一个UNC名称（而不是指定一个重定向的驱动器字母，如前面所述）来打开一个远程文件或设备时，MUP驱动程序被激活。当Windows客户端DLL Kernel32.dll（它导出了与文件I/O相关的API）接收到一个带有UNC路径的文件I/O请求时，该子系统将UNC路径附加到字符串\Global??\UNC后面，然后调用NtCreateFile系统服务来打开该文件。

如果只有一个注册的网络提供者，那么\Global??\UNC被解析到代表该驱动程序的设备对象，由该驱动程序处理此请求。如果有多个注册的提供者，那么\Global??\UNC被解析到\Device\MUP，并且MUP必须确定应该由哪个提供者来处理此请求。

当MUP驱动程序接收到一个I/O请求时，并且DFS客户也是启用的， MUP首先确定I/O请求中的路径是否引用了一个DFS路径（DFS路径也是UNC格式的），如果是的话，则它自己处理I/O请求。如果DFS客户是禁用的，或者该路径不是一个DFS路径，那么，MUP使用HKLM\SYSTEM\CurrentControlSet\Control\NetworkProvider\Order\ProviderOrder注册表值来确定那些利用FsRtlRegisterUncProvider向它注册的网络提供者的优先级顺序。它按照向它注册的网络提供者在注册表值中被列出的顺序，每次询问其中一个，直到有一个提供者表明它能够识别该路径，或者所有可用的提供者都已经被询问为止。MUP忽略掉那些虽然列在注册表值中但没有注册的重定向器。

www.TopSage.com

当一个重定向器识别出一个路径时，它必须指明该路径有多少部分对于它来说是独有的。例如，如果该路径为\\WIN2K3SERVER\PUBLIC\Windowsinternals\Chap13.doc，那么，该重定向器也许能识别出它，并且声明其前缀\\WIN2K3SERVER\PUBLIC是它独有的。MUP驱动程序将这些信息缓存起来，以后，再碰到以此前缀作为开始的请求时，可以直接将它发送给该重定向器，从而跳过了轮询操作。MUP驱动程序的缓存有一个超时特性，所以，过了一段时间以后，一个前缀与某个特定重定向器的关联也随之过期。

13.4 名称解析

名称解析是指这样一个过程：基于字符的名称，比如Mycomputer或www.microsoft.com，被转译成一个可被协议栈识别的数值地址，比如192.168.1.1。这一节讲述Windows提供的两个与TCP/IP相关的名称解析协议：域名系统（DNS，Domain Name System）和WINS（Windows Internet名称服务，Windows Internet Name Service）。

域名系统

域名系统（DNS，Domain Name System）是一个标准。通过此标准，Internet名称（比如*www.microsoft.com*）可以被转译为对应的IP地址。如果网络应用程序想要将一个DNS名称解析成一个IP地址，则它利用TCP/IP协议向一个DNS服务器发送一个DNS查询请求。DNS服务器实现了一个由“名称/IP地址”对构成的分布式数据库，这些服务器利用数据库中的信息来完成地址转译。每个服务器维护了某一特定“**区域**（*zone*）”的地址转译。关于DNS的细节描述，超出了本书的范围，但是，DNS是Windows中命名机制的基础，所以，它是主要的Windows名称解析协议。

Windows DNS服务器被实现为一个Windows服务（\Windows\System32\Dns.exe），它被包含在Windows的服务器版本中。标准的DNS服务器实现方式依赖于一个文本文件作为地址转译数据库，但是，Windows DNS服务器可以被配置成将区域信息存储在活动目录中。

Windows Internet名称服务

一个被称为*Windows Internet*名称服务（WINS，*Windows Internet Name Service*）的网络服务维护了“NetBIOS名称-IP地址”之间的映射关系，对于以NetBIOS为基础的TCP/IP应用程序来说，此映射关系是需要用到的。如果没有安装WINS，那么，对于一个子网上的名称操作，NetBIOS使用本地广播消息。注意，对于Windows套接字应用程序，NetBIOS名称相对于DNS名称来说只是辅助的；计算机名称首先是通过DNS来注册和解析的，只有当DNS名称解析失败时，Windows才退回到NetBIOS名称。

www.TopSage.com

13.5 协议驱动程序

网络API驱动程序必须接受API请求，并且将这些请求转译成低层网络协议的请求，以完成跨网络传输的任务。API驱动程序依赖于内核模式的传输协议驱动程序来完成实际的转译工作。将API与底层协议分离开，这样做可使得网络体系结构更加灵活，允许每个API使用许多不同的协议。Windows包含的协议驱动程序包括TCP/IP、基于IPv6的TCP/IP、NWLink，以及AppleTalk协议。以下分别对这些协议进行简短的描述。

- Internet的急剧增长以及对于TCP/IP协议的依赖，使得TCP/IP成为Windows中最为突出的协议。DARPA（The Defense Advanced Research Projects Agency，国防部高级研究工程机构）于1969年专门开发了TCP/IP，并将其作为Internet的基础。因此，TCP/IP具有一些对于WAN非常友好的特征，比如可路由性（routability）和很好的WAN性能。TCP/IP是首选的Windows协议。它是默认安装的，并且不能被删除。
- 标准TCP/IP栈使用的是IPv4协议，而IPv4协议使用的是4字节网络地址，这限制了公共IP地址的数量大约只能是40亿个。随着越来越多的设备，比如移动电话和PDA，都要获得一个Internet场所标识，IP地址的数量与日俱增。由于这个原因，使用16字节地址的IPv6将会得到越来越广泛的采用。Windows XP（SP1及以后）和Windows Server 2003包含一个实现了IPv6的TCP/IP栈（\Windows\System32\Drivers\Tcpip6.sys）。Windows IPv6栈也通过采用隧道机制来支持与IPv4网络的共存。
- NWLink是由Novell的IPX和SPX协议组成的。Windows包含NWLink是为了与Novell NetWare服务器进行互操作。
- AppleTalk协议是由Apple McIntosh网络使用的，由于Windows提供了对该协议的支持，所以Windows可以与AppleTalk网络中的文件和打印服务进行互操作。

Windows中的TDI传输器通常实现了与它们的主协议相关联的所有协议。例如，IPv4 TCP/IP驱动程序（\Windows\System32\Drivers\Tcpip.sys）实现了TCP、UDP、IP、ARP、ICMP和IGMP。TDI传输器通常会创建一些代表其特定协议的设备对象，所以，客户可以获得一个代表某个协议的文件对象，并且利用IRP来向该协议发出网络I/O。TCP/IP驱动程序创建了几个设备对象，它们代表了一些可供TDI客户访问的协议，这些设备对象是\Device\Tcp、\Device\Udp、\Device\Ip，以及在Windows XP和Windows Server 2003上的\Device\Rawip和\Device\Ipmulticast。

www.TopSage.com



实验：查看TCP/IP的设备对象

利用内核调试器来查看一个实际运行的系统，你可以检查TCP/IP的设备对象。首先，执行*!drvobj*命令以便看到该驱动程序的每个设备对象的地址，然后，执行*!devobj*来查看每个设备对象的名称和其他细节信息。

```
lkd> !drvobj tcpip
Driver object (8a01ada0) is for:
 \Driver\Tcpip
Driver Extension List: (id , addr)

Device Object list:
8a0dbc88  8a0dc958  8a0dcd80  8a0eff18
8a0f32a0

lkd> !devobj 8a0dbc88
Device object (8a0dbc88) is for:
 RawIp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 3 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dbd40
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0dc958
Device object (8a0dc958) is for:
 Udp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 41 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dca10
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0dcd80
Device object (8a0dcd80) is for:
 Tcp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 302 Type 00000012 Flags 00000050
Dacl e1c3a1ac DevExt 00000000 DevObjExt 8a0dce38
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0eff18
Device object (8a0eff18) is for:
 IPMULTICAST \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 0 Type 00000012 Flags 00000040
Dacl e100d19c DevExt 00000000 DevObjExt 8a0effd0
ExtensionFlags (0000000000)
Device queue is not busy.

lkd> !devobj 8a0f32a0
Device object (8a0f32a0) is for:
 Ip \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 48 Type 00000012 Flags 00000050
Dacl e1c3a1ac DevExt 00000000 DevObjExt 8a0f3358
ExtensionFlags (0000000000)   Device queue is not busy.
```


www.TopSage.com

所以，网络API驱动程序并不需要为每个它可能想要使用的传输协议都使用各种接口，Microsoft建立了传输驱动程序接口（TDI，Transport Driver Interface）标准。正如本章前面曾经提到的，在本质上，TDI接口只是一种“将网络请求格式化成IRP，以及申请网络地址和数据通信”的做法规范而已。遵从TDI标准的传输协议向它们的客户导出了TDI接口，它们的客户包括了诸如AFD之类的网络API驱动程序，以及重定向器。按照Windows设备驱动程序的形式来实现的传输协议也称为TDI传输器。因为TDI传输器是设备驱动程序，所以，它们把接收到的客户请求格式化成IRP。

在\Windows\System32\Drivers\Tdi.sys库中的支持函数，连同开发人员在它们的驱动程序中包含的这些定义，构成了TDI接口。TDI编程模型非常类似于Winsock的编程模型。TDI客户执行下面的步骤来跟远程服务器建立一个连接。

1. 客户申请并格式化一个“**地址打开**（*address open*）”TDI IRP，以便申请一个地址。TDI传输器返回一个代表客户所请求的地址的文件对象，它也称为**地址对象**（*address object*）。这一步等价于在Winsock中使用*bind*函数。
2. 然后，客户申请并格式化一个“**连接打开**（*connection open*）”TDI IRP；TDI传输器返回一个代表此连接的文件对象，它也被称为**连接对象**（*connection object*）。这一步等价于在Winsock中使用*socket*函数。
3. 客户利用一个“**地址关联**（*associate address*）”TDI IRP，将此连接对象与地址对象关联起来（在Winsock中没有对应的步骤）。
4. 接受远程连接的TDI客户发出一个“**监听**（*listen*）”TDI IRP，并指定一个连接对象所支持的连接数目。然后发出一个“**接受**（*accept*）”TDI IRP。当一个远程系统建立一个连接（或者一个错误发生）时，此accept请求就会完成。这些操作等价于在Winsock中使用*listen*和*accept*函数。
5. 想要与远程服务器建立连接的TDI客户发出一个“**连接**（*connect*）”TDI IRP，并指定连接对象。当一个连接被建立起来（或者一个错误发生）时，TDI传输器完成此connect请求。发出一个“**连接**（*connect*）”TDI IRP就等同于在Winsock中使用*connect*函数。

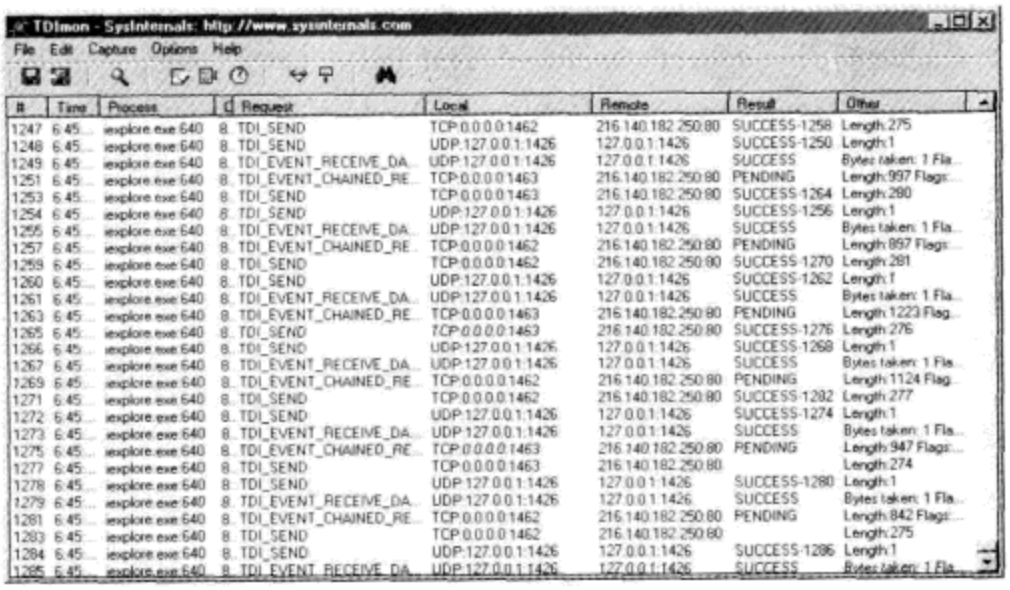
TDI也为像UDP这样的无连接协议，提供了无连接通信的支持。而且，TDI也提供了一种方法，让TDI客户可以向TDI传输器注册“**事件回调**（*event callback*）”（也就是说，被直接调用的函数）。例如，当TDI传输器接收到来自网络上的数据时，它可以调用一个已注册的客户接收回调函数。TDI的这种基于事件的回调特性使得TDI传输器能够将网络事件通知它的客户，而依赖于事件回调机制的客户在接收网络数据时不需要预分配像缓冲区这样的资源，因为它们可以看得到由TDI协议驱动程序提供的缓冲区的内容。

www.TopSage.com



实验：观察TDI的活动

来自*www.sysinternals.com*的一个实用工具TDImon是一种过滤型驱动程序，它附载在TCP/IP驱动程序创建的设备对象\Device\Tcp和\Device\Udp上。这样附载以后，TDImon就可以看到TDI客户给这些协议发出的每一个IRP。通过截取TDI客户事件回调的注册，它也可以监视事件回调。TDImon驱动程序将有关TDI活动的信息发送给它的GUI，以便显示出来，所以，在GUI中，你可以看到一个操作的时间、所发生的TDI活动的类型、一个TCP连接的本地和远程地址或者一个UDP端点的本地地址、IRP或者事件回调的结果状态代码，以及其他一些信息，比如发送或接收的字节数。下面是TDImon的一个屏幕截图，它正在观察Microsoft Internet Explorer浏览一个Web页面时所产生的TDI活动。



#	Time	Process	Request	Local	Remote	Result	Other
1247	6:45...	iexplore.exe:640	8..TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80	SUCCESS-1258	Length:275
1248	6:45...	iexplore.exe:640	8..TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1250	Length:1
1249	6:45...	iexplore.exe:640	8..TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1251	6:45...	iexplore.exe:640	8..TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1463	216.140.182.250:80	PENDING	Length:997 Flags:...
1253	6:45...	iexplore.exe:640	8..TDI_SEND	TCP:0.0.0.0:1463	216.140.182.250:80	SUCCESS-1264	Length:280
1254	6:45...	iexplore.exe:640	8..TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1256	Length:1
1255	6:45...	iexplore.exe:640	8..TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1257	6:45...	iexplore.exe:640	8..TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1462	216.140.182.250:80	PENDING	Length:897 Flags:...
1259	6:45...	iexplore.exe:640	8..TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80	SUCCESS-1270	Length:281
1260	6:45...	iexplore.exe:640	8..TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1262	Length:1
1261	6:45...	iexplore.exe:640	8..TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1263	6:45...	iexplore.exe:640	8..TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1463	216.140.182.250:80	PENDING	Length:1223 Flag...
1265	6:45...	iexplore.exe:640	8..TDI_SEND	TCP:0.0.0.0:1463	216.140.182.250:80	SUCCESS-1276	Length:276
1266	6:45...	iexplore.exe:640	8..TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1268	Length:1
1267	6:45...	iexplore.exe:640	8..TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1269	6:45...	iexplore.exe:640	8..TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1462	216.140.182.250:80	PENDING	Length:1124 Flag...
1271	6:45...	iexplore.exe:640	8..TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80	SUCCESS-1282	Length:277
1272	6:45...	iexplore.exe:640	8..TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1274	Length:1
1273	6:45...	iexplore.exe:640	8..TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1275	6:45...	iexplore.exe:640	8..TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1463	216.140.182.250:80	PENDING	Length:947 Flags:...
1277	6:45...	iexplore.exe:640	8..TDI_SEND	TCP:0.0.0.0:1463	216.140.182.250:80		Length:274
1278	6:45...	iexplore.exe:640	8..TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1280	Length:1
1279	6:45...	iexplore.exe:640	8..TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...
1281	6:45...	iexplore.exe:640	8..TDI_EVENT_CHAINED_RE...	TCP:0.0.0.0:1462	216.140.182.250:80	PENDING	Length:842 Flags:...
1283	6:45...	iexplore.exe:640	8..TDI_SEND	TCP:0.0.0.0:1462	216.140.182.250:80		Length:275
1284	6:45...	iexplore.exe:640	8..TDI_SEND	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS-1286	Length:1
1285	6:45...	iexplore.exe:640	8..TDI_EVENT_RECEIVE_DA...	UDP:127.0.0.1:1426	127.0.0.1:1426	SUCCESS	Bytes taken: 1 Fla...

TDI操作本质上是异步的，作为这一事实的一个证据，你可以看到，在Result一列中的PENDING代码说明一个操作已经被激发，但是定义该操作的IRP尚未完成。为了精确地显示出这些IRP的完成时刻相对于其他操作开始点的顺序关系，每个IRP的发起点或事件回调被标记一个序列号。如果在一个IRP完成以前其他的IRP被发出或者完成了，那么，该IRP的完成也将被标记一个序列号，并显示在Result一列中。例如，在以上屏幕截图中，序列号为1278的IRP在序列号为1279的IRP发出以后才完成，所以，你在IRP 1278的Result一列中看到了1280。

TCP/IP的扩展

其他的Windows网络服务通过一些附加的驱动程序扩展了TCP/IP协议驱动程序的基本网络特性。这些附加的驱动程序利用私有的接口来与TCP/IP协议驱动程序集成在一起。这些扩展网络服务包括网络地址转译（NAT，network address translation）、IP过滤、IP钩子和Internet协议安全性（IPSec）。图13.17显示了这些扩展服务如何与TCP/IP驱动程序结合在一起。

www.TopSage.com

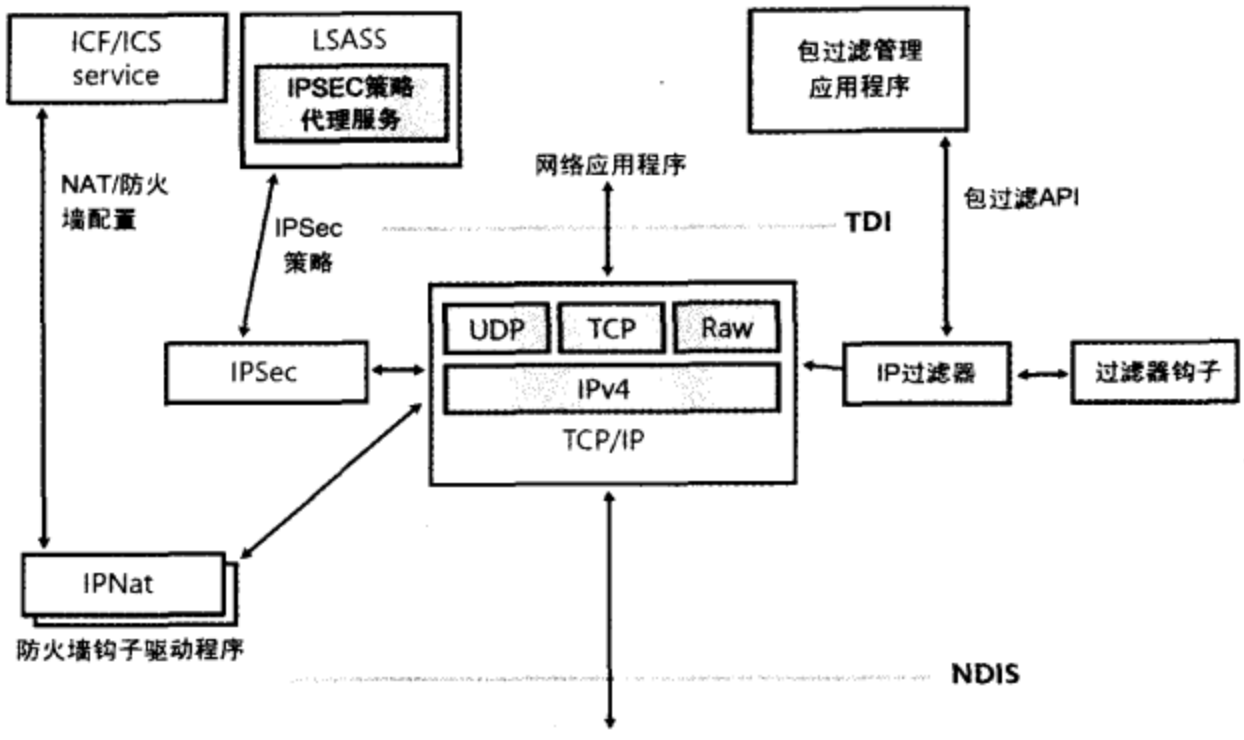


图13.17 TCP/IP的扩展结构

网络地址转译

网络地址转译（NAT，network address translation）是一种路由服务，它允许多个私有IP地址被映射到单个公共IP地址上。如果没有NAT，那么一个LAN中的每台计算机必须被分配一个公共的IP地址，才能通过Internet进行通信。NAT允许一个LAN中只有一台机器被分配一个公共的IP地址，而其他的计算机使用私有的IP地址，并且通过那一台计算机连接到Internet上。NAT会根据需要，在私有IP地址和公共IP地址之间进行转译，并且在LAN计算机和Internet之间转发数据包。

Windows上的NAT组件是由一个NAT设备驱动程序（\Windows\System32\Drivers\Ipnat.sys）和一些编辑器构成的，其中，设备驱动程序将NAT与TCP/IP栈连接起来，而这些编辑器可以执行除了地址和端口转译以外其他更多的包处理任务。你可以利用“Routing And Remote Access”MMC加载件将NAT安装成一个路由协议组件，或者，你在使用Network Connections文件夹来配置Internet Connection Sharing（ICS）时安装NAT（不过，当使用“Routing And Remote Access”MMC加载件来安装NAT时，可以有更多的配置项）。

IP过滤

Windows 2000、Windows XP和Windows Server 2003都包含了一项非常基本的IP过滤功能，通过此项功能，用户可以选择只允许特定的端口或者IP协议。虽然这一功能可以被用来保护一台计算机避免遭受未授权的网络访问，但是，它的缺点是，它是静态的，它无法为该计算机上正在运行的应用程序所发起的网络流量自动地创建新的过滤器。

www.TopSage.com

Windows XP还引入了个人防火墙功能，这种防火墙称为Windows防火墙（Windows Firewall），它超出了上面描述的基本过滤功能。Windows防火墙实现了一个**有状态的防火墙**（*stateful firewall*），它可以跟踪网络流量，所以，它能够区分本地LAN上发起的TCP/IP流量和Internet上不请自来的流量。当一个网络接口（interface）上启用了Windows防火墙时，在默认情况下，该计算机接收到的所有不请自来的流量都被丢弃。用户或应用程序可以定义一些例外，因而，该计算机上运行的服务，比如文件和打印共享或者Web站点，可以被其他的计算机访问到。

Windows防火墙服务是在Svchost进程中执行的，它将Windows防火墙的用户界面中定义的例外规则传递给IPNat驱动程序。在内核模式中，Windows防火墙与NAT（网络地址转译）是在同一个驱动程序（\Windows\System32\Drivers\Ipnat.Sys）中实现的。NAT驱动程序向TCP/IP驱动程序注册成一个**防火墙钩子**驱动程序（*firewall hook* driver）。TCP/IP驱动程序在处理进来的或者出去的IP包时，执行每一个已注册的防火墙钩子的回调函数。回调函数可以修改一个包的源地址和目标地址，从而提供NAT的功能；或者回调函数向TCP/IP返回一个状态代码，要求TCP/IP丢掉该包和停止对该包的进一步处理，从而实现一个防火墙。

虽然Ipnat利用TCP/IP的防火墙钩子接口实现了Windows防火墙，但是，Microsoft推荐第三方厂商利用NDIS中间驱动程序（本章后面将会讲述）的形式来实现包过滤功能。

IP过滤器和过滤器钩子

Windows XP和Windows Server 2003包含了一个用户模式包过滤API和一个IP过滤器驱动程序（\Windows\System32\Drivers\Ipfltrdrv.sys），这两者合起来可以让应用程序很方便地管理一个系统的进来和出去的数据包流。IP过滤器驱动程序也允许至多有一个驱动程序可注册成一个**过滤器钩子**驱动程序（*filter hook* driver）。TCP/IP按照一种类似于它与防火墙钩子驱动程序的交互方式，来执行一个由IP过滤器驱动程序指定的函数，从而允许IP过滤器丢弃或者修改数据包。IP过滤器依次又调用一个由过滤器钩子驱动程序指定的回调函数，进而将对数据包的修改，或者“丢弃数据包”的请求，传递给TCP/IP驱动程序。

系统提供的过滤器钩子功能使得第三方也可以在系统中加入地址转译、防火墙、日志记录或者其他的功能。

Internet协议安全性（IPSec）

IPSec（Internet Protocol security）与Windows TCP/IP协议栈集成在一起，它可以保护单播（unicast）数据避免遭受各种攻击，比如偷听、嗅探器（sniffer）攻击、数据修改、IP地址欺骗和中间人攻击。你可以使用IPSec来提供深度的防御，以对抗各种类型的攻击：来自于不可信计算机的网络攻击；可能导致“应用程序、服务或网络的拒绝服务（denial-of-service）”的攻击；导致数据破坏、数据被窃、用户凭证被窃的攻击；导致“对服务器、其他计算机和网络进行管理性控制”的攻击。IPSec通过基于密码学的安全服务、安全协议和动态密钥管理来帮助对抗各种基于网络的攻击。

www.TopSage.com

IPSec针对在可信主机之间发送的单播IP包提供了下面的属性。

- **数据源认证** 验证IP包的来源，确保未认证的其他方不能访问数据。
- **数据完整性** 保护IP数据包，以避免IP包在途中遭到篡改却未被检测到。
- **数据保密性** 在传输之前加密IP包的有效载荷数据。数据保密性可确保：只有与一台计算机正在进行IPSec通信的对方才能够阅读和解释数据包的内容。
- **反重放攻击**（或者重放保护） 确保每一个IP包都是惟一的，因而不可能被重用。这一属性可防止攻击者在截取到IP包以后，将修改的数据包插入到源计算机和目标计算机之间的一个数据流中。采用了重放保护，攻击者就不能通过重放原先捕获的消息来建立一个会话，或者获得“未经授权而访问数据”的能力。

通过“配置基于主机的IPSec包过滤功能”和“强制使用可信的通信”，你可以使用IPSec来加强防御，以避免遭受基于网络的攻击。当你使用基于主机的IPSec包过滤功能时，IPSec可以根据源地址和目标地址的组合，以及特定的协议和特定的端口，允许或阻止特定类型的单播IP流量。因为IPSec与TCP/IP协议栈的IP层（第3层）集成在一起，而且它被应用于所有的应用程序，所以，你不需要为每个使用TCP/IP的应用程序配置单独的安全性。

在活动目录环境中，你可以利用组策略（Group Policy）来配置域（domain）、站点（site）和组织单元（OU，organizational unit），然后，你可以根据需要为组策略对象（GPO，Group Policy Object）指定IPSec策略。类似地，你也可以配置和指定本地IPSec策略。基于活动目录的IPSec策略被保存在活动目录中，当前策略的一份副本被保存在一个缓存中，该缓存位于本地注册表中。本地IPSec策略被保存在本地系统的注册表中。

为了建立起可信的通信，IPSec使用双向认证，它支持下列认证方法：Kerberos版本5，计算机的X.509版本3公钥证书，或者一个预先共享的密钥。

Windows的IPSec实现建立在IPSec RFC（Requests for Comments）的基础之上。Windows IPSec的总体结构包括了IPSec策略引擎、IKE（Internet Key Exchange）协议和一个IPSec驱动程序。

- **IPSec策略引擎** IPSec策略引擎被作为一个Windows服务来运行，位于LSASS（Local Security Authority Subsystem，本地安全权威子系统）进程中（关于LSASS的信息，请参见第8章）。在MMC（Microsoft Management Console，Microsoft管理控制台）的Services加载件中，IPSec策略引擎位于计算机服务列表中的IPSEC服务名下。IPSec策略引擎从活动目录域或者本地注册表中获得IPSec策略，然后把IP地址过滤器传递给IPSec驱动程序，把认证和安全设置传递给IKE。

www.TopSage.com

- **IKE** IKE等待从IPSec驱动程序发送过来的协商SA（security association，安全关联）的请求，然后协商SA，再把协商得到的SA设置送回给IPSec驱动程序。SA是一些双方都同意的IPSec策略设置和密钥的组合（这些策略设置定义了安全的服务、机制，而这些密钥被用于保护IPSec个体之间的通信）。每个SA是一个单向或单工的连接，它只保护一个方向的流量。当IKE接收到IPSec驱动程序的请求时，它协商主模式SA（main mode SA）和快模式SA（quick mode SA）。IKE主模式（或者ISAKMP）SA保护IKE的协商过程。快模式（或者IPSec）SA保护应用程序的流量。
- **IPSec驱动程序** IPSec驱动程序是一个设备驱动程序（\Windows\System32\Drivers\Ipsec.sys），它被绑定到TCP/IP驱动程序上，并且处理流经TCP/IP驱动程序的数据包。IPSec驱动程序监视和保护发送出去的单播IP流量，并且，它监视、解密和验证接收进来的单播IP包。该驱动程序接收来自IPSec策略引擎的过滤器，然后根据需要，允许、阻止或者保护数据包。为了保护流量，IPSec驱动程序使用当前活动的SA设置，或者它请求创建新的SA。

你可以使用MMC中的“IP安全策略管理（IP Security Policy Management）”加载件来创建和管理IPSec策略。利用该加载件，你可以创建、修改和存储本地的IPSec策略或者基于活动目录的IPSec策略，也可以修改远程计算机上的IPSec策略。在Windows XP和Windows Server 2003中，在建立起由IPSec来保护的通信以后，你可以利用MMC中的“IP安全监视器（IP Security Monitor）”加载件，来监视本地计算机和远程计算机的IPSec信息。

13.6 NDIS驱动程序

当一个协议驱动程序想要按照其协议的格式从网络上读写消息时，该驱动程序必须通过一个网络适配器才能做到这一点。因为期望协议驱动程序能够理解市场上的每一款网络适配器的细微差别（各种厂商的网络适配器的数量有数千之多）是不太现实的，所以，网络适配器厂商提供了能够通过该厂商的专属硬件来接受和传输网络消息的设备驱动程序。1989年，Microsoft和3Com联合开发了网络驱动程序接口规范（NDIS，Network Driver Interface Specification），因而协议驱动程序可以以一种与设备无关的方式来跟网络适配器驱动程序进行通信。遵从NDIS的网络适配器驱动程序称为NDIS驱动程序，或者NDIS小端口驱动程序。随Windows 2000一起发行的NDIS的版本是NDIS 5，而随Windows XP和Windows Server 2003一起发行的版本是NDIS 5.1。

NDIS库（\Windows\System32\Drivers\Ndis.sys）实现了TDI传输器（在通常情况下）和NDIS驱动程序之间存在的NDIS边界。如同Tdi.sys一样，NDIS库是一个辅助库，NDIS驱动程序的客户利用它，对将要发送给NDIS驱动程序的命令进行格式化。NDIS驱动程序与该库相互协作，以便接收请求和送回应答。图13.18显示了与NDIS相关的各个组件之间的关系。

www.TopSage.com

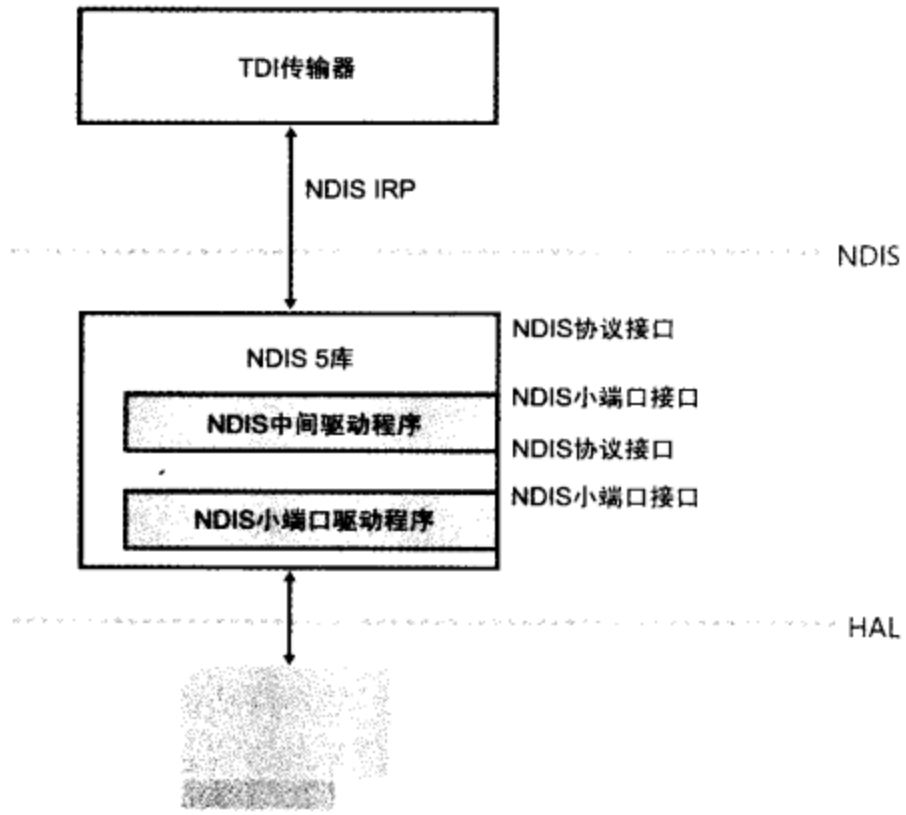


图13.18 NDIS的组件

Microsoft设计此网络结构的目标之一是，让网络适配器的厂商能够很容易地开发NDIS驱动程序，以及将驱动程序的代码在Consumer Windows和Windows 2000之间转换。因此，NDIS库不是简单地提供一些NDIS边界辅助例程，而是为NDIS驱动程序提供了一个完整的执行环境。NDIS驱动程序不是真正的Windows驱动程序，因为如果没有NDIS库给予它们的封装，它们是不能正常工作的。这一隔离层将NDIS驱动程序完全包装起来，因而NDIS驱动程序并不接受和处理IRP。相反地，TDI协议驱动程序调用NDIS库中的一个函数*NdisAllocatePacket*，再通过调用NDIS库的另一个函数（比如*NdisSend*）把数据包传给NDIS小端口。在默认情况下，NDIS驱动程序也不用担心重入。所谓重入是指，在NDIS驱动程序完成上一个请求的服务以前，NDIS库又调用该驱动程序来处理新的请求。避免重入意味着NDIS驱动程序的编写者不需要担心复杂的同步（在多处理器上由于有可能存在并行执行的情形，所以同步会变得更加棘手）。



> **注** NDIS库使用NDIS包而不是使用IRP来表示I/O请求。TDI传输器通过调用*NdisAllocatePacket*来申请一个NDIS包，然后调用一个NDIS库函数（比如*NdisSend*）把该包传递给一个NDIS小端口。

虽然NDIS库对于NDIS驱动程序的请求的序列化简化了开发过程，但是，这种序列化也妨碍了多处理器扩展性。标准的NDIS 4驱动程序（Windows NT 4的NDIS库版本）在多处理器平台上，对于某些特定的操作无法很好地扩展。Microsoft在NDIS 5中，给开发人员提供了一个非序列化的操作选项。NDIS 5驱动程序可以向NDIS库表示，它们不希望被序列化。于是，NDIS

www.TopSage.com

库每当接收到IRP时，就会尽可能快地将IRP中的请求转发给NDIS驱动程序。排队和管理多个并发请求的任务落到了NDIS驱动程序的头上，但是，非序列化带来的好处是更高的多处理器性能。

NDIS 5也包含以下特性。

- NDIS驱动程序可以报告它们的网络介质是否是激活的，这使得Windows可以在任务栏上显示一个网络已连接/已断开的图标。这一特性也使得协议和其他应用程序能够确切地获知这一状态，并依据情况作出响应。例如，TCP/IP传输器将使用这一信息来确定何时应该重新计算来自DHCP的地址信息。
- TCP/IP任务外移（task offloading），这使得一个小端口可以利用网络适配器的高级特性来执行诸如包检验和（packet checksum）以及IPSec（Internet Protocol security）这样的操作。这种任务外移能够将CPU从这些操作中解脱出来，从而提高了系统的性能。
- 从LAN唤醒（Wake-on-LAN），这使得一块支持“从LAN唤醒”的网络适配器可以将Windows从挂起的电源状态唤醒过来。能触发网络适配器向系统发信号的事件包括：介质连接（比如将一条网络线缆插到适配器上）、接收到与协议相关的数据模式（由某个协议注册的模式，例如，TCP/IP传输器要求在接收到ARP[Address Resolution Protocol，地址解析协议]请求时被唤醒），以及，对于以太网适配器，接收到一个*magic*包（一个网络数据包包含了该适配器以太网地址的16份连续拷贝）。
- 面向连接的NDIS，这使得NDIS驱动程序可以管理面向连接的介质，比如ATM（Asynchronous Transfer Mode，异步传输模式）设备（稍后将更加详细地讲述面向连接的NDIS）。

NDIS库提供了一些接口函数给NDIS驱动程序，以便它们与网络适配器硬件打交道，这些函数直接转译成HAL中对应的函数。



实验：列出已加载的NDIS小端口

Ndiskd内核调试器扩展库包含了*!miniports*和*!miniport*命令，这两个命令可以让你在内核调试器中列出已加载的小端口，而且，在给定一个小端口块（Windows用于记录小端口的一个数据结构）的地址的情况下，你还可以查看有关该小端口驱动程序的详细信息。下面的例子显示了如何利用*!miniports*和*!miniport*来列出所有的小端口，然后对于负责连接系统与PCI以太网适配器的小端口，列出其详细的信息。（注意，WAN小端口驱动程序与电话拨号连接一起工作。）

```
kd> .load ndiskd
Loaded ndiskd extension DLL

kd> !miniports
Driver verifier level: 0
Failed allocations: 0
Miniport Driver Block: 817aa610
```


www.TopSage.com

```
  Miniport: 817b1130 RAS Async Adapter
Miniport Driver Block: 81a1ef30
  Miniport: 81a1ea70 Direct Parallel
Miniport Driver Block: 81a21cd0
  Miniport: 81a217f0 WAN Miniport (PPTP)
Miniport Driver Block: 81a23290
  Miniport: 81a22130 WAN Miniport (L2TP)
Miniport Driver Block: 81a275f0
  Miniport: 81a25130 Intel 8255x-based PCI Ethernet Adapter (10/100)

kd> !miniport 81a25130
 Miniport 81a25130 : Intel 8255x-based PCI Ethernet Adapter (10/100)
    Flags           : 20413208
                      BUS_MASTER, INDICATES_PACKETS, IGNORE_REQUEST_QUEUE
                      IGNORE_TOKEN_RING_ERRORS, NDIS_5_0,
                      RESOURCES_AVAILABLE, DESERIALIZED, MEDIA_CONNECTED,
                      NOT_SUPPORTS_MEDIA_SENSE,
    PnPFlags        : 00010021
                      PM_SUPPORTED, DEVICE_POWER_ENABLED, RECEIVED_START
    CheckforHang interval: 2 seconds
        CurrentTick    : 0001
        IntervalTicks : 0001
    InternalResetCount : 0000
    MiniportResetCount : 0000

    References: 3
    UserModeOpenReferences: 0

    PnPDeviceState : PNP_DEVICE_STARTED
    CurrentDevicePowerState : PowerDeviceD0
    Bus PM capabilities
    DeviceD1:1
    DeviceD2:1
    WakeFromD0:0
    WakeFromD1:1
    WakeFromD2:0
    WakeFromD3:0

    SystemStateDeviceState
    PowerSystemUnspecifiedPowerDeviceUnspecified
    S0D0
    S1D1
    S2PowerDeviceUnspecified
    S3PowerDeviceUnspecified
    S4D3
    S5D3
    SystemWake: S1
    DeviceWake: D1

    WakeupMethodes Enabled 6:
    WAKE_UP_PATTERN_MATCH  WAKE_UP_LINK_CHANGE
    WakeUpCapabilities of the miniport
    MinMagicPacketWakeUp: 4
    MinPatternWakeUp: 4
    MinLinkChangeWakeUp: 4
```


www.TopSage.com

```
Current PnP and PM Settings:          : 00000030
                DISABLE_WAKE_UP, DISABLE_WAKE_ON_RECONNECT,

Allocated Resources:
    Memory: f4100000, Length: 1000
    IO Port: 1440, Length: 40
    Memory: f4000000, Length: 100000
    Interrupt Level: 9, Vector: 9
Translated Allocated Resources:
    Memory: f4100000, Length: 1000
    IO Port: 1440, Length: 40
    Memory: f4000000, Length: 100000
    Interrupt Level: 12, Vector: 39
MediaType       : 802.3
DeviceObject    : 81a25030, PhysDO : 81a93cd0  Next DO: 81a63030
MapRegisters    : 819fc000
FirstPendingPkt: 0
SingleWorkItems:
  [0]: 81a254e8 [1]: 81a254f4 [2]: 81a25500 [3]: 81a2550c
  [4]: 81a25518 [5]: 81a25524
DriverVerifyFlags       : 00000000
Miniport Open Block Queue:
  8164b888: Protocol 816524a8 = NBF, ProtocolContext 81649030
  8191f628: Protocol 81928d88 = TCPIP, ProtocolContext 8191f728
 Miniport Interrupt 81a00970
```

以上列出的小端口的Flags域指明：该小端口支持非序列化操作（DESERIALIZED），该介质当前是激活的（MEDIA_CONNECTED），而且，它是一个NDIS 5小端口驱动程序（NDIS_5_0）。另外列出的还有该适配器的“从系统到设备的电源状态映射关系”和即插即用管理器分配给该适配器的总线资源（有关电源状态映射的更多信息，请参见第9章的“电源管理器”一节）。

NDIS小端口的变化形式

NDIS模型也支持混合的TDI传输器-NDIS驱动程序，此驱动程序称为*NDIS*中间驱动程序（*NDIS intermediate driver*）。这样的驱动程序位于TDI传输器和NDIS驱动程序之间。对于NDIS驱动程序来说，一个NDIS中间驱动程序看起来就好像是一个TDI传输器；而对于一个TDI传输器来说，NDIS中间驱动程序看起来好像是一个NDIS驱动程序。NDIS中间驱动程序可以看到发生在一个系统中的所有网络流量，因为它们位于协议驱动程序和网络驱动程序之间。为网络适配器提供容错和负载平衡特性的软件，比如Microsoft的网络负载平衡提供者（Network Load Balancing Provider），都是建立在NDIS中间驱动程序的基础之上的。

面向连接的NDIS

对于面向连接的网络硬件（比如ATM），Windows提供了内置的支持，这构成了Windows网络体系结构中关于连接管理和建立的标准。

www.TopSage.com

面向连接的NDIS驱动程序使用了标准NDIS驱动程序也使用的许多同样的API；然而，面向连接的NDIS驱动程序通过已建立的网络连接来发送数据包，而不是将数据包放在网络介质上。

除了针对面向连接介质而提供的小端口支持以外，NDIS 5也包含下列定义，以进一步支持面向连接的小端口驱动程序。

- 呼叫管理器（call manager）是那些为面向连接的客户（稍后介绍）提供呼叫建立和呼叫断开服务的NDIS驱动程序。呼叫管理器使用一个面向连接的小端口来跟其他的网络实体（比如网络交换机或者其他的呼叫管理器）交换信令消息。一个呼叫管理器支持一个或者多个信令协议，比如ATM用户网络接口（UNI，User-Network Interface）3.1。
- 集成的小端口呼叫管理器（MCM）是一个面向连接的小端口驱动程序，它也为面向连接的客户提供呼叫管理器服务。MCM本质上是一个内置了呼叫管理器的NDIS小端口驱动程序。
- 面向连接的客户使用一个呼叫管理器或者MCM的呼叫建立和断开服务，同时也使用一个面向连接的NDIS小端口驱动程序的发送和接收数据的服务。面向连接的客户可以向网络栈的上层提供它自己的协议服务，或者它也可以实现一个仿真层从而将一个无连接的遗留协议与面向连接的介质结合起来。由一个面向连接的客户来实现仿真层的一个例子是LAN仿真（LANE，LAN emulation），它将ATM的面向连接特征隐藏起来，给上层协议呈现的是一个无连接的介质（比如以太网）。

图13.19显示了这些部件之间的关系。

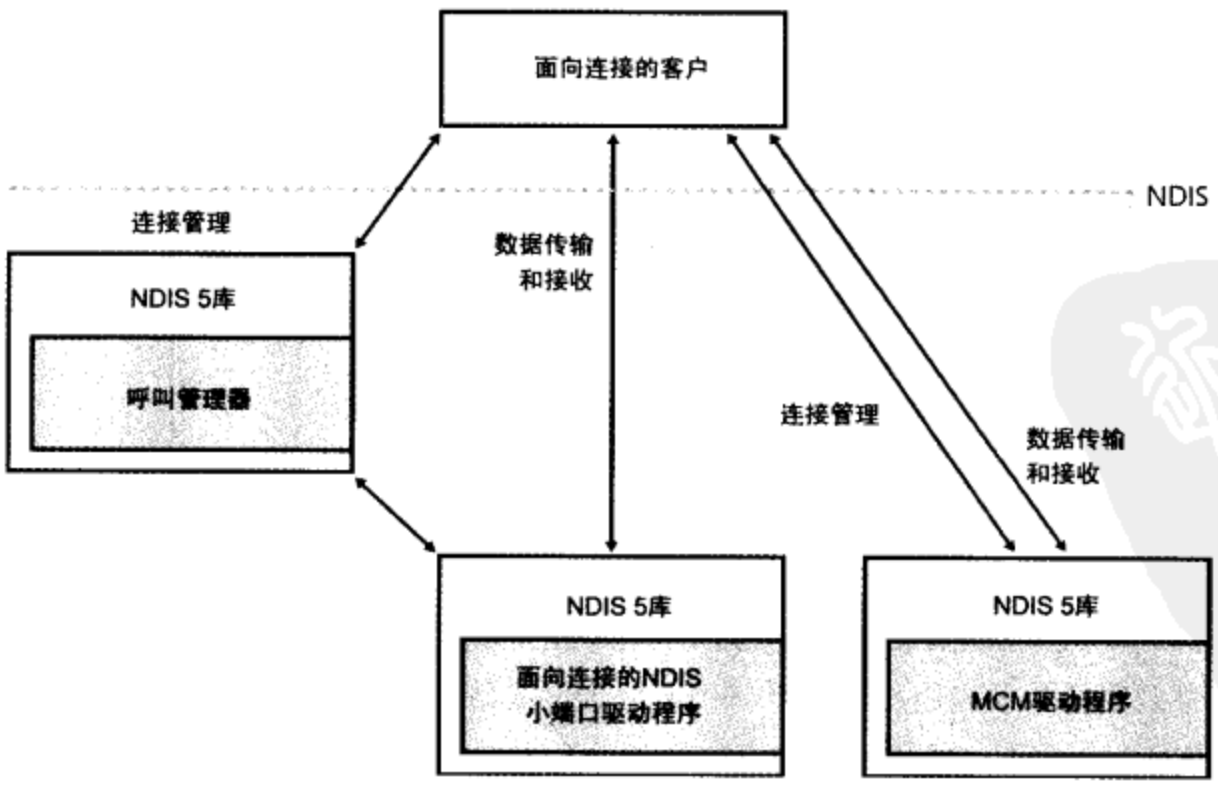


图13.19 面向连接的NDIS驱动程序

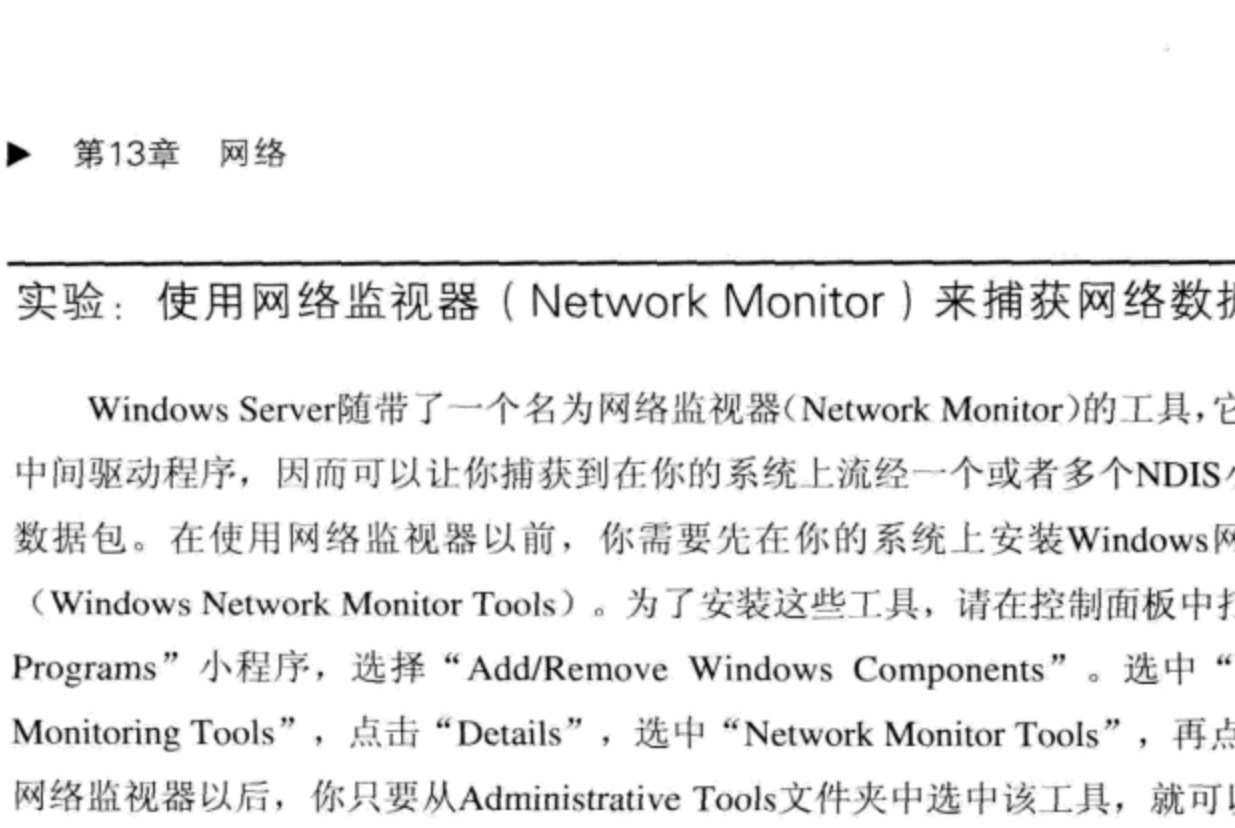
www.TopSage.com



实验：使用网络监视器（Network Monitor）来捕获网络数据包

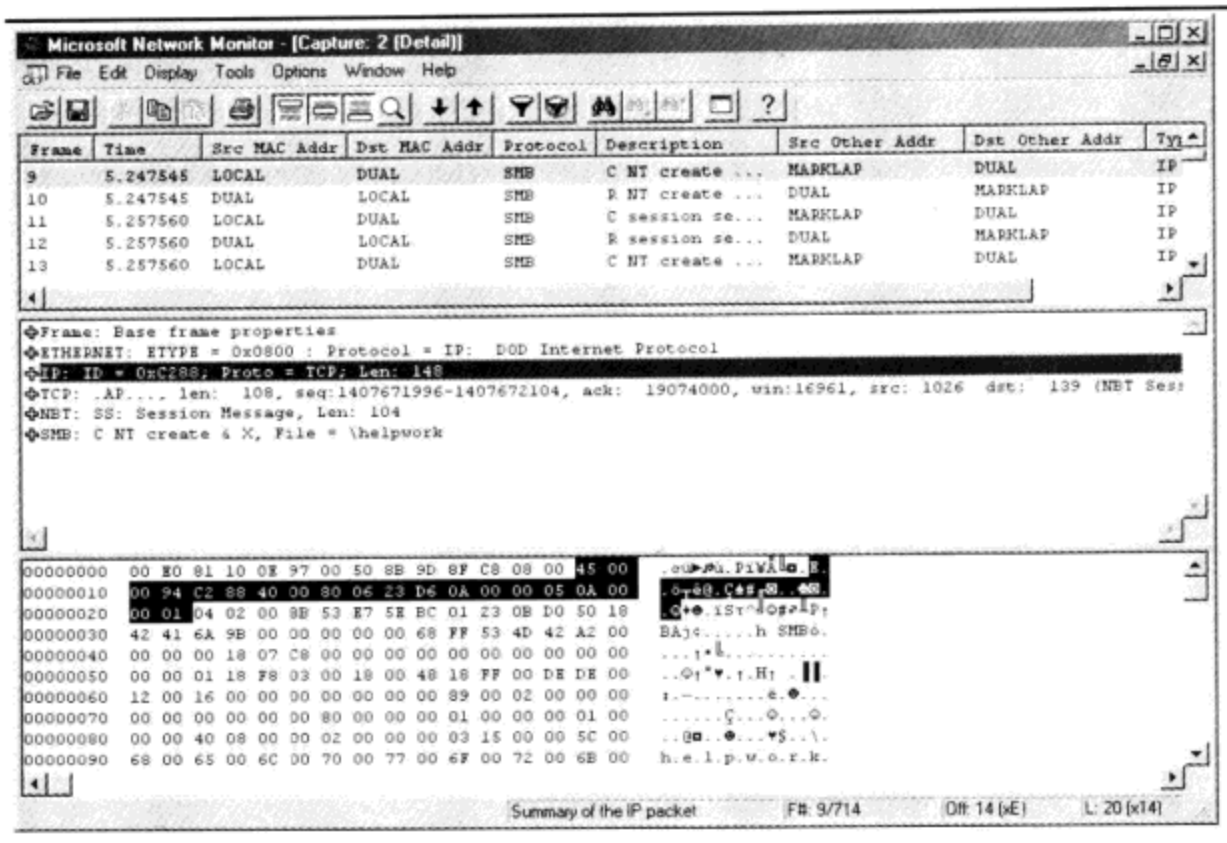
Windows Server随带了一个名为网络监视器(Network Monitor)的工具，它安装了一个NDIS中间驱动程序，因而可以让你捕获到在你的系统上流经一个或者多个NDIS小端口驱动程序的数据包。在使用网络监视器以前，你需要先在你的系统上安装Windows网络监视器工具箱（Windows Network Monitor Tools）。为了安装这些工具，请在控制面板中打开“Add/Remove Programs”小程序，选择“Add/Remove Windows Components”。选中“Management And Monitoring Tools”，点击“Details”，选中“Network Monitor Tools”，再点击OK。在安装了网络监视器以后，你只要从Administrative Tools文件夹中选中该工具，就可以启动网络监视器了。

网络监视器可能会问你想要监视哪个网络连接。你在选择了一个网络连接以后，只需按下工具栏上的“Start Capture”按钮，就可以开始监视了。然后，执行一些操作以便在你监视的连接上产生一些网络活动。在你看到网络监视器已经捕获到数据包以后，点击“Stop And View Capture”按钮（旁边有一副眼镜的停止按钮）以停止监视。网络监视器将会显示这些捕获到的数据的一个视图，如下所示：



以上屏幕截图显示了当用户从该系统中访问远程文件时网络监视器捕获到的SMB（CIFS）包。如果在某一行上双击，网络监视器将切换到针对该包的一个视图，此视图显示出各层次的应用和协议头，如下面的屏幕截图所示：

www.TopSage.com



网络监视器也包含了许多其他的特性，比如捕获触发器（capture trigger）和过滤器，这使得它成为一个强大的诊断网络问题的工具。

外接NDIS（Remote NDIS）

在外接NDIS被设计出来以前，某个厂商，如开发USB网络设备的厂商，必须提供这样一个驱动程序：该驱动程序作为一个小端口驱动程序与NDIS打交道，同时它也要跟USB WDM总线驱动程序打交道，如图13.20所示。如果该厂商的硬件还要支持其他的总线，比如IEEE 1394，那么，它们还要实现相应的驱动程序以便与每一种特定的总线类型打交道。

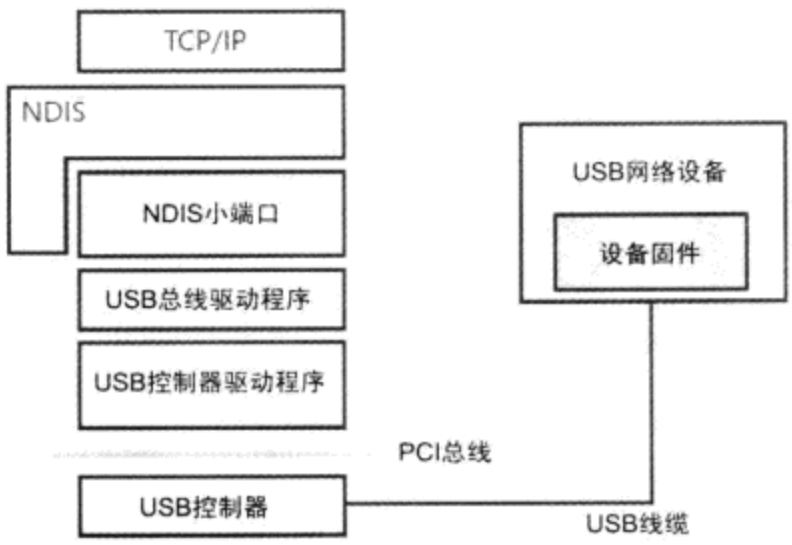


图13.20 针对USB网络设备的NDIS小端口驱动程序

www.TopSage.com

外接NDIS是一个针对“位于动态即插即用总线（比如USB、IEEE 1394和Infiniband）上的网络设备”的规范。该规范彻底消除了硬件厂商编写NDIS小端口驱动程序的必要性，它的做法是，定义一些与总线无关的消息，以及定义一套机制，让这些消息可以在各种总线上传输。外接NDIS消息反映了NDIS接口，包含针对以下各种任务的消息：初始化和重置一个设备、传输和接收数据包、设置和查询设备的参数，以及指示介质链路状态。

图13.21显示了外接NDIS的体系结构，它依赖于一个由Microsoft提供的NDIS小端口驱动程序，即\Windows\System32\Drivers\Rndismp.sys。该驱动程序负责转译NDIS命令，并且将这些命令转发给一个设备所在总线的总线传输驱动程序。该结构使得单个NDIS小端口驱动程序可以被所有的外接NDIS驱动程序和（对于每一种所支持的总线）单个总线传输驱动程序所使用。

当前，针对USB设备的外接NDIS被包含在Windows XP和Windows Server 2003中，而对于Windows 2000，则可以从Microsoft的网站上下载而得到。虽然针对IEEE 1394的外接NDIS已经被完全规范化了，但Windows尚未支持它。

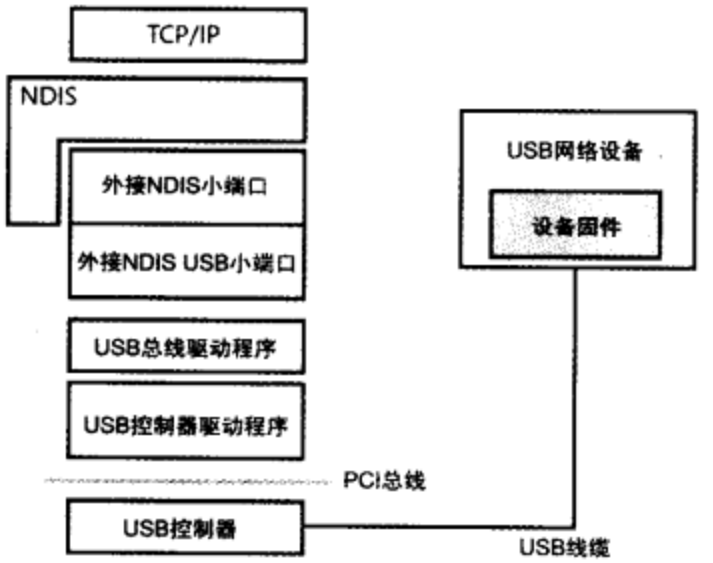


图13.21 针对USB网络设备的外接NDIS体系结构

QOS

如果没有指定特别的度量标准的话，那么，IP流量在通过一个网络时，按照“先到达先服务”的原则被递交。应用程序无法控制其消息的优先级，它们可能会经历**突发性**的网络行为，在此过程中，它们有时候获得很高的吞吐量和很低的延迟，但在其他情况下却获得很差的网络性能。虽然这种服务等级在大多数情况下是可以接受的，但是，越来越多的网络应用程序要求更加一致的服务等级，或者称为**服务质量**（QOS，*quality of service*）保证。视频会议、流媒体和企业资源计划（ERP，enterprise resource planning）这些实际的应用例子都要求有很好的网络性能。QOS使得一个应用程序可以指定最小的带宽和最大的延迟，而这样的条件只有当发送者和接收者之间的所有网络软件和硬件组件都支持QOS标准（比如IEEE 802.1P）时才有可能

www.TopSage.com

被满足。IEEE 802.1P是一个工业标准，它规定了QOS包的格式，以及OSI第2层的设备（交换机和网络适配器）如何响应这些QOS包。

Windows对于QOS的支持是建立在Microsoft定义的诸多Winsock API基础之上的。通过这些API，一个应用程序可以指定在它的Winsock套接字上传输的流量有什么样的QOS要求。例如，一个应用程序可以使用*WSCInstallQOSTemplate*来安装一个QOS模板，并且在QOS模板中指定它所期望的带宽和延迟。另一个API，即流量控制（TC，Traffic Control）API，允许一个管理性的应用程序更加精确地控制流经该计算机所在网络的流量。

Windows QOS实现的核心是RSVP（Resource Reservation Protocol，资源预留协议） Win32服务（\Windows\System32\Rsvp.exe），如图13.22所示。RSVP Winsock服务提供者（\Windows\System32\Rsvpsp.dll）通过RPC向RSVP服务传达有关应用程序QOS的请求信息。然后，RSVP服务使用TC API来控制流量。TC API是在\Windows\System32\Traffic.dll中实现的，它向GPC（Generic Packet Classifier，通用的包分类器）驱动程序（\Windows\System32\Msgpc.sys）发送I/O控制命令。GPC驱动程序又与QOS包调度器NDIS中间驱动程序（\Windows\System32\Psched.sys）紧密地合作，以便对从该计算机发送到网络上的数据包流进行控制，因此，向特定应用程序承诺的QOS等级能够被满足，并且，在期望得到QOS的包中可以放上适当的QOS头。



> 注　在Windows XP和Windows Server 2003中，RSVP信令功能已经被去掉了。虽然RSVP服务仍然存在，但是，它的存在只是被当作应用程序和流量控制组件之间的一个管道而已。

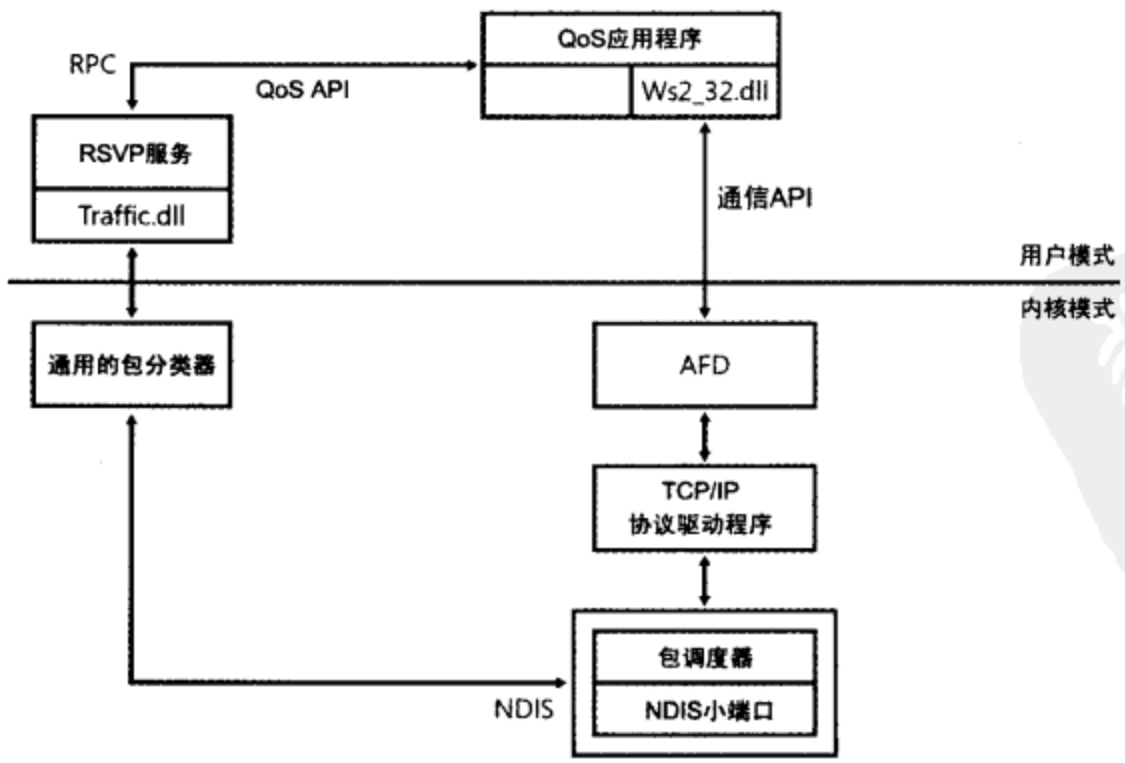


图13.22　QOS体系结构

www.TopSage.com

13.7 绑定

Windows网络体系结构中的最后一个难解之谜是，各个层次（网络API层、TDI传输驱动程序层、NDIS驱动程序层）上的组件相互之间是以什么方式找到对方的。将这些层次连接起来的过程的名称是“**绑定**（*binding*）”。如果你曾经通过在网络连接（Network Connections）文件夹中添加或删除一个网络组件，以此来改变你的系统的网络配置的话，那么，你一定已经见证过绑定过程了。

当你安装一个网络组件时，你必须提供一个针对该组件的INF文件（第9章讲述了INF文件）。此文件包含了一些指令，Setup API例程必须遵从这些指令才能安装和配置该组件。另外，此文件也包含了绑定相依性或绑定关系。开发人员可以为其专属的组件指定绑定相依性，因而服务控制管理器（SCM，第4章中讲述了服务控制管理器）将不仅按照正确的顺序来加载该组件，而且，只有当该专属组件依赖的其他组件出现在系统上时它才加载该组件。绑定关系是由绑定引擎利用一个组件的INF文件中的额外信息来确定的，这些绑定关系建立起各个层上的组件之间的连接。这些连接指定了一个层上的某个网络组件可以利用其底下层上的哪些组件。

例如，工作站服务（重定向器）将自动地绑定到TCP/IP和NWLink协议上。其绑定的顺序决定了绑定的优先级。你可以在“Advanced Settings”对话框的“Adapters And Bindings”标签页上看到绑定的顺序，如图13.23所示（关于如何启动“Advanced Settings”对话框的指示，请参见本章前面的“多重定向器支持”一节）。此重定向器在接收到一个访问远程文件的请求的同时，将该请求提交给这两个协议驱动程序。当应答回来时，重定向器一直在等待，直至接收到来自任何高优先级协议驱动程序的应答为止。只有到那时，重定向器才会将结果返回给调用者。因此，重新排列这些绑定的顺序可能是有好处的，这样，高优先级的绑定也是性能最有效的，或者最适合于你的网络上的绝大多数计算机。你也可以通过“Advanced Settings”对话框手工删除绑定关系。

www.TopSage.com

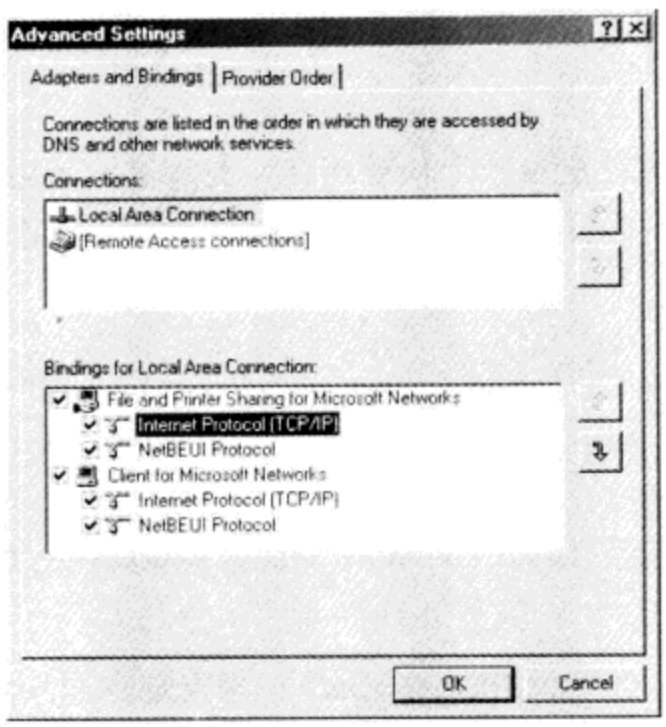


图13.23 使用Advanced Settings对话框来编辑绑定关系

在一个网络组件的注册表配置键的Linkage子键中，Bind值存放了有关该组件的绑定信息。例如，如果你检查HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Linkage\Bind值，你将会看到工作站服务（Workstation service）的绑定信息。

13.8 分层的网络服务

Windows包含的一些网络服务建立在本章前面介绍的API和组件的基础之上。对这些服务的功能和内部实现的详细描述，超出了本书的范围。这一节只是概要地说明远程访问（Remote Access）、活动目录、网络负载平衡、文件复制服务（FRS，File Replication service）和分布式文件系统（DFS，Distributed File System）。

远程访问（Remote Access）

在Windows Server中，远程访问是通过“路由和远程访问（Routing and Remote Access）”服务来提供的，此服务使得远程访问客户可以连接到远程访问服务器上，并且使访问诸如文件、打印机和网络服务之类的网络资源，就好像该客户被通过物理方式连接到远程访问服务器的网络上一样。Windows提供了以下两种远程访问类型。

- **拨号远程访问** 客户通过电话或者其他的电信设施来连接到远程访问服务器的时候需要用到这种类型的远程访问。其中的电信介质被用于在客户和服务器之间创建一个临时的、物理的或者虚拟的连接。

www.TopSage.com

- **虚拟私有网络（VPN）远程访问** 它允许VPN客户与服务器通过一个如Internet这样的IP网络来建立一个虚拟的点到点的连接。

远程访问不同于远程控制，因为远程访问充当一个代理连接，允许客户连接至一个Windows网络，而远程控制软件是在服务器上执行应用程序，并且向客户出示一个用户界面。

活动目录

活动目录是LDAP（Lightweight Directory Access Protocol，轻量的目录访问协议）目录服务的Windows实现。活动目录建立在一个数据库的基础之上，该数据库中保存的对象代表了在一个Windows网络中由各种应用程序定义的资源。例如，Windows域的结构和成员关系，包括用户账户和口令信息，都被保存在活动目录中。

对象类和描述对象各种特征的属性是由一个“**表结构**（*schema*）”来指定的。活动目录中的对象是按照层次结构来组织的，这非常类似于注册表的逻辑组织方式，其中容器对象可以存储其他的对象，包括其他的容器对象（关于容器对象的更多信息，请参见第8章）。

活动目录支持许多API，客户可以利用这些API来访问一个活动目录数据库中的对象。下面列出了这些API。

- **LDAP CAPI** 它是一个使用LDAP网络协议的C语言API。用C或C++语言来编写的应用程序可以直接使用此API，而使用其他语言来编写的应用程序则可以通过转译层来访问此API。
- **活动目录服务接口**（ADSI，Active Directory Service Interfaces） 它是一个COM接口，它专门针对在LDAP基础上实现的活动目录；它抽象了LDAP编程的细节。ADSI支持多种语言，包括Microsoft Visual Basic、C和Microsoft Visual C++。Microsoft WSH（Windows Script Host）应用程序也可以使用ADSI。
- **MAPI**（Messaging API） 它是为了跟Microsoft Exchange客户和Outlook Address Book客户应用程序保持兼容而支持的一个API。
- **安全账户管理器**（SAM）API 它建立在活动目录之上，其目的是为诸如MSV1_0（\Windows\System32\Msv1_0.dll，遗留下来的NT LAN Manager认证协议也使用此认证包）和Kerberos（\Windows\System32\Kdcsvc.dll]）这样的登录认证包提供一个接口。
- **Windows NT 4网络API**（Net API） Windows NT 4客户使用此API来获得对活动目录的访问（通过SAM）。
- **NTDS API** 它被用于在一个活动目录中查找SID和GUID（主要通过DsCrackNames()），以及活动目录的主要管理功能和活动目录复制。有一些第三方厂商通过编写应用程序来利用这些API来监视活动目录。

www.TopSage.com

活动目录被实现为一个数据库文件，在默认情况下，该数据库文件被命名为\Windows\Ntds\Ntds.dit；并且，该数据库文件被在一个域的多个域控制器之间进行复制。“活动目录”目录服务是一个Windows服务，在LSASS进程中执行；它使用一些DLL来管理该数据库，这些DLL实现了该数据库在磁盘上的结构，并且提供了基于事务的更新操作以保护数据库的完整性。在Windows 2000上，活动目录数据库的存储建立在ESE（Extensible Storage Engine，可扩展的存储引擎）数据库的一个版本的基础之上，Microsoft Exchange Server 5.5版本的客户/服务器消息机制和群件也用到了此版本的ESE数据库；在Windows 2003 Server上，活动目录数据库的存储建立在Microsoft Exchange Server 2000所使用的ESE数据库版本的基础之上。图13.24显示了活动目录的体系结构。

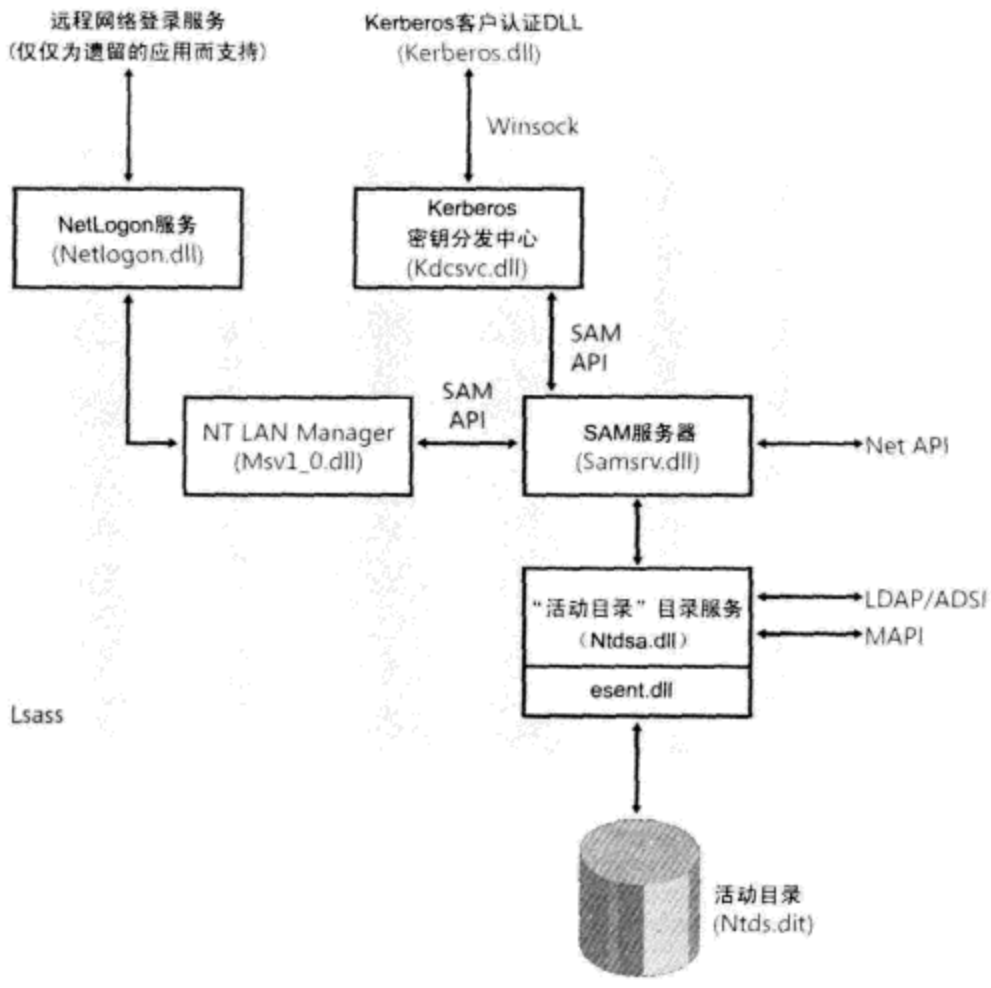


图13.24 活动目录的体系结构

网络负载平衡

正如本章前面所说明的，网络负载平衡是建立在NDIS中间驱动程序的技术基础之上的。网络负载平衡被包含在Windows Advanced Server中，它允许创建一个包含多达32台计算机的集群，这些计算机被称为**集群主机**（*cluster host*）。

www.TopSage.com

每个集群维护了单个虚拟IP地址，此地址是对外公开的，以便于客户来访问。客户的请求可到达该集群中所有的计算机上。然而，只有一个集群主机响应此请求。网络负载平衡NDIS驱动程序可以有效地将客户空间，按照一种分布式的方式，在当前可用的集群主机之间进行划分。按照这种方式，每个主机只处理属于它那一部分的客户请求，而每一个客户请求总是由一个而且只有一个主机进行处理。对于一个客户请求，确定由自己来处理该请求的那一个集群主机允许该请求被传递至TCP/IP协议驱动程序，最终被传递至一个服务器应用程序中，而其他的集群主机不会如此向上传递该请求。如果一个集群主机失败了，则该集群中剩下的主机意识到此集群主机已不再是处理客户请求的一个候选主机了，于是将进来的客户请求重新分配到余下的集群主机中。因此，新的客户请求不再被发送至已停机的集群主机上。另一个集群主机可以被加入到集群中，作为已失败主机的替代者，然后，它将无缝地开始处理客户请求。

网络负载平衡并不是一个通用的集群方案，因为客户与之通信的服务器应用程序必须具备一定的特征：首先，它必须是基于TCP/IP的；其次，它必须在网络负载平衡集群中的任何一个系统上处理客户的请求。这第二个要求通常意味着，应用程序为了处理客户的请求，必须访问共享状态，而且它必须自己来维护此共享状态——网络负载平衡服务并不包含“可以自动地在集群主机之间分发共享状态”的服务。适合于这种网络负载平衡机制的理想应用包括提供静态内容服务的Web服务器、Windows媒体服务器，以及终端服务。图13.25显示了一个网络负载平衡操作的例子。

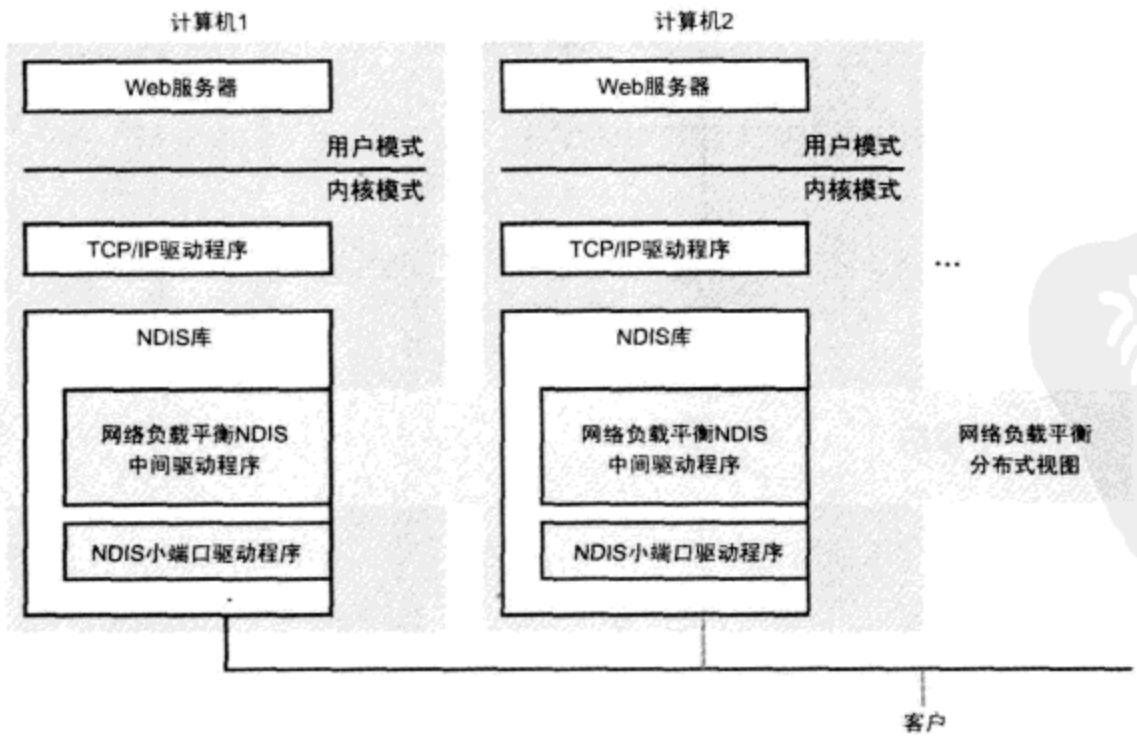


图13.25 网络负载平衡的操作

www.TopSage.com

文件复制服务

文件复制服务（FRS，File Replication service）被包含在Windows Server系统中。它的主要用途是复制域控制器的\SYSVOL目录中的内容，而该目录正是Windows域控制器存储登录脚本和组策略的地方（组策略使得管理员可以为同属于一个域的计算机定义使用策略和安全策略）。而且，FRS也可以被用来在系统之间复制DFS（Distributed File System，分布式文件系统）共享体。FRS使用了分布式的多主复制（multimaster replication）结构，这使得任何服务器都可以执行复制操作。当一个复制的目录或文件被改变时，这些变化信息被传播到其他的域控制器上。

FRS中的一个基本概念是**复制集**（*replica set*）。一个复制集是由两个或多个系统组成的：它们会根据管理员定义的时间安排和拓扑结构，来相互之间复制一个目录树的内容。只有NTFS卷的目录才可以被复制，因为FRS依赖于NTFS的变化日志来检测一个复制集的目录中的文件的变化情况。因为FRS是基于多主复制的，所以，理论上它可以支持上百个，甚至上千个系统作为一个复制集的一部分，而且，一个复制集中的计算机可以通过任意的网络拓扑结构被连接起来（比如环形的、星形的，或者网状的）。任何一台计算机也可以同属于多个复制集。

FRS是以Windows服务的形式来实现的（\Windows\System32\Ntfrs.exe），在多个不同计算机上的Ntfrs.exe实例之间，FRS使用经过认证和加密的RPC进行通信。而且，因为活动目录包含了它自己的复制能力，所以，FRS利用活动目录API，从一个域的活动目录中获取FRS配置信息。

分布式文件系统

分布式文件系统（DFS，Distributed File System）是一个位于工作站服务（Workstation service）的上层的服务，它将多个文件共享体（file share）连接到一起，形成单个名字空间。这些文件共享体既可以驻留在同一台计算机上，也可以驻留在不同的计算机上，DFS为客户提供了一种位置透明的方式来访问这些资源。DFS名字空间的根一定是在某一台Windows服务器上定义的一个文件共享体。

除了实现一个统一的网络资源名字空间以外，DFS还通过DFS复制集（DFS replica set）提供了其他一些好处。管理员可以根据两个或者多个共享体来创建一个DFS复制集，并且利用一种复制机制（比如FRS），在复制集的多个共享体之间拷贝数据，以便使这些共享体的内容保持同步。DFS提供了几种形式的负载平衡，它的做法是：对一个复制集中的成员进行排序，以及/或者选择这些成员，来完成一个客户对于该复制集上的数据的请求。另外，当一个复制集的某个成员不能工作时，DFS通过将客户请求转发到另一个或另一些可以正常工作的成员上，以达到很高的可用性。

构成DFS体系结构的组件如图13.26所示。DFS的服务器端实现是由一个Windows服务（\Windows\System32\Dfssvc.exe）和一个设备驱动程序（\Windows\System32\Drivers\Dfs.sys）构成的。

www.TopSage.com

该DFS服务负责导出DFS拓扑结构管理接口，以及在注册表（在无活动目录服务的系统上）或者活动目录中维护DFS的拓扑结构。DFS驱动程序在接收到客户请求时执行拓扑结构查询操作，因而它可以将客户指引到目标文件所在的系统上。

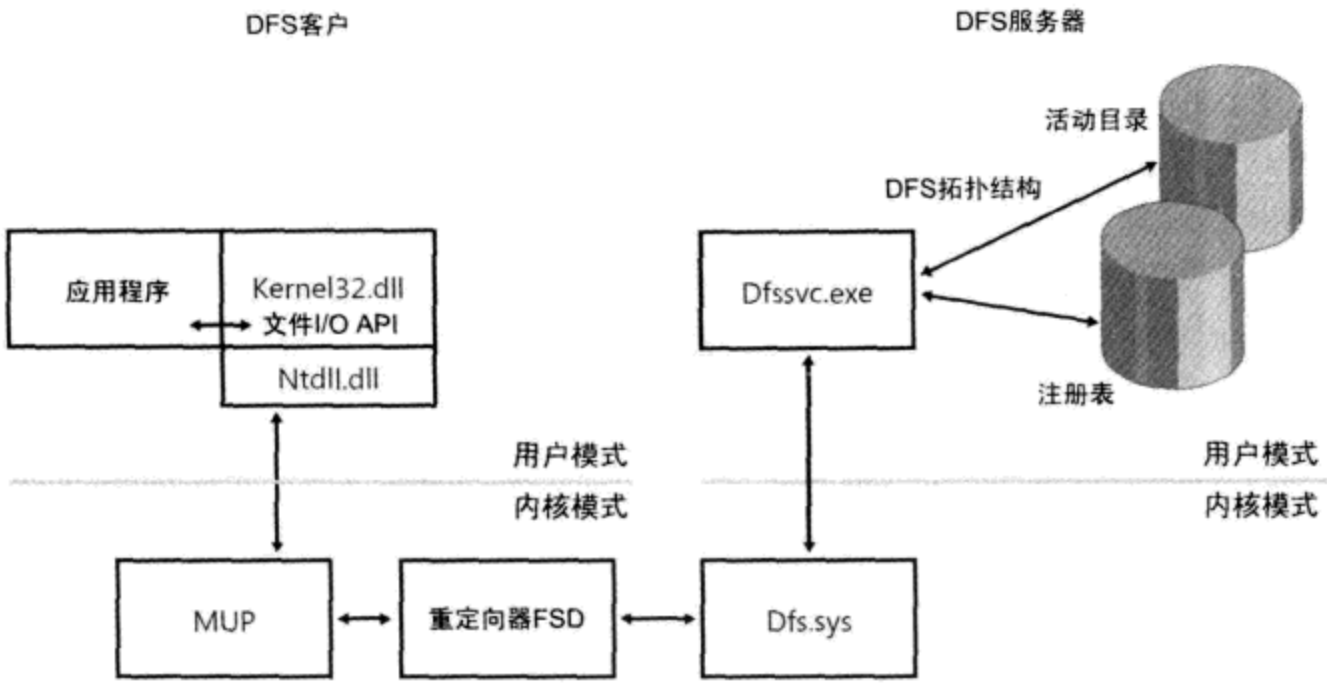


图13.26 DFS的组件

在客户端，正如早先的描述，对DFS的支持是在MUP驱动程序中实现的，而且，此驱动程序使用CIFS重定向器来跟DFS服务器进行内部通信。DFS客户提供者是在\Windows\System32\Ntlanman.dll中实现的。当一个客户发出一个文件I/O请求，并且该请求指定了DFS名字空间中的一个文件时，客户端的MUP驱动程序利用此重定向器来跟目标文件服务器进行通信。

13.9 本章总结

Windows的网络结构为网络API、网络协议驱动程序和网络适配器驱动程序提供了一个灵活的基础设施。Windows网络结构充分发挥了I/O层次机制的优势，为网络支持提供了很强的扩展能力，使得当计算机网络技术进化时，Windows的网络支持也能随之而进化。当新的协议出现时，开发人员通过编写一个TDI传输器，就可以在Windows上实现该协议。类似地，新的API可以结合已有的Windows协议驱动程序。最后，在Windows上实现的各种网络API为网络应用程序开发人员提供了各种可能的实现，每一种实现都有不同的编程模型和协议支持。

www.TopSage.com

Crash Dump Analysis

第 14 章　崩溃转储分析

几乎每一位Windows用户都听说过臭名昭著的“蓝屏死机（blue screen of death）”，有的甚至还亲身经历过。这一不祥的术语是指当Microsoft Windows崩溃或停止执行（由于灾难性的错误或者内部条件阻止系统继续运行下去）时所显示的蓝色屏幕。

在本章中，我们将讨论那些引起Windows崩溃的基本问题，讲述一下在蓝屏上显示的信息，同时还将解释各种用于创建**崩溃转储**（*crash dump*）的配置选项。这里的崩溃转储是指在系统崩溃时刻的系统内存的纪录，它可以帮助你找出是哪个组件导致了这次系统崩溃。安排这一部分内容的意图并不是提供有关如何分析一次Windows系统崩溃的详细诊断信息。本章还将向你展示如何分析一个崩溃转储来识别出一个错误的驱动程序或者组件。为执行基本的崩溃转储分析而需要做的工作是非常少的，只需要几分钟就够了。即使每5个或10个崩溃转储中只有一个通过崩溃转储分析能探查到有问题的驱动程序，这种分析仍然是值得去做的：一次成功的分析可以避免将来的数据丢失、系统宕机和应用失败。

14.1　Windows为什么会崩溃

Windows崩溃（停止执行并显示蓝屏）有以下一些原因。

- 运行在内核模式下的设备驱动程序或者操作系统函数引发了一个未被处理的异常，比如内存访问违例（由于企图写一个只读页面或者企图读一个当前未被映射[因而不是一个有效内存位置]的地址而引起）。
- 调用一个内核支持例程，导致一次重新调度，比如当中断请求级别（IRQL）为DPC/Dispatch级别或更高级别时等待一个处于无信号状态的分发器对象（关于IRQL的细节，请参见第3章）。
- 在DPC/Dispatch级别或更高的IRQL级别时，在“由页面文件或内存映射文件中的数据来支撑的内存”上发生了一个页面错误（这将要求内存管理器必须等待一个I/O操作发生，但正如上面刚刚所说的，在DPC/Dispatch级别或更高级别上不能够进行等待，因为那将要求一次重新调度）。

www.TopSage.com

- 一个设备驱动程序或操作系统函数显式地使系统崩溃（通过调用系统函数*KeBugCheckEx*）。因为它检测到一个内部条件表明了数据遭受破坏，或者表明了其他的情形：在不冒险破坏数据的情况下系统无法继续执行。
- 发生硬件错误，比如机器检查（machine check）或者不可屏蔽中断（NMI）。

对于Windows XP用户提交给Microsoft OCA（Microsoft Online Crash Analysis，在线崩溃分析，本章后面将会介绍）Web站点的崩溃转储，Microsoft的分析表明了崩溃原因的分类如图14.1所示（这里显示的数据是2004年4月份生成的）。

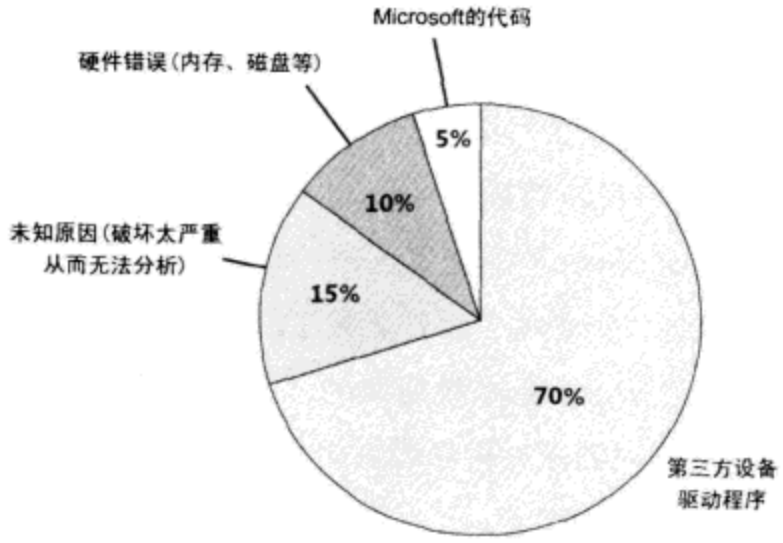


图14.1 提交给OCA的崩溃的原因分类

当一个内核模式设备驱动程序或者子系统引发一个非法的异常时，Windows面临着困难的抉择。此时，它检测到了在操作系统中，有能力访问任何硬件设备和任何有效内存的系统代码部分做了一些不应该做的事情。

但是，为什么这意味着Windows必须崩溃呢？难道它就不能够忽略该异常，让设备驱动程序或者子系统继续往下执行，就好像什么也没有发生过吗？该错误能被隔离出来的可能性是存在的，并且该组件将来能够通过某种方式被恢复过来的可能性也是存在的。但是，更有可能的是，被检测到的异常来源于更深层的问题——比如，由于内存的常规破坏（general corruption），或者由于硬件设备不能正常工作。允许系统继续运行下去将可能导致更多的异常，而且，存储在磁盘或其他外设中的数据可能也会遭受破坏，这样招致的风险太高了。

14.2 蓝屏

不管系统崩溃的原因是什么，真正执行崩溃的函数是*KeBugCheckEx*（Windows DDK中有文档）。该函数接受一个**停止代码**（*stop code*，有时候也称为**错误检查码**[*bug check code*]），以及四个根据停止代码来解释的参数。*KeBugCheckEx*屏蔽了该系统的所有处理器上的所有中

www.TopSage.com

断以后，将显示器切换到低分辨率的VGA图形模式（Windows支持的所有视频卡都实现了这种模式）下，绘制一个蓝色背景，然后显示此停止代码，后面紧跟一些文本来建议用户应该怎么办。最后，*KeBugCheckEx*调用任何已注册的设备驱动程序错误检查回调函数（通过调用*KeRegisterBugCheckCallback*函数来注册的回调函数），从而让驱动程序有机会停止它们的设备（有可能系统数据结构已经被破坏得太严重，以至于蓝屏都显示不出来）。图14.2显示了Windows XP蓝屏的一个例子。



> 注 Windows XP Service Pack 1和更高的版本，以及Windows Server 2003引入了*KeRegisterBugCheckReasonCallback*函数，以允许设备驱动程序在一个崩溃转储的后面附加一些数据，或者将崩溃转储的信息写到其他的设备中。

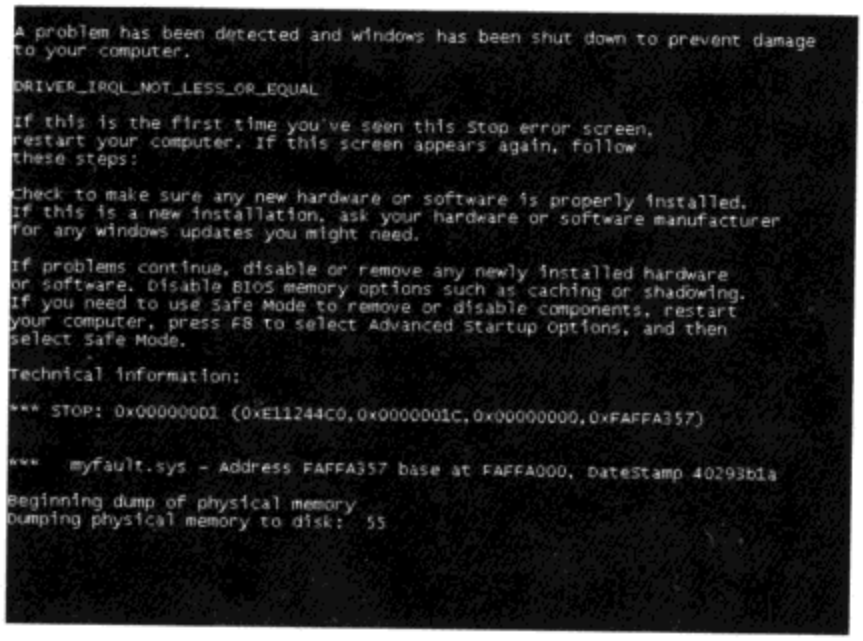


图14.2 蓝屏的例子

*KeBugCheckEx*在蓝屏顶部的附近显示了停止代码的文本表示，在Windows 2000蓝屏的顶部，也显示了停止代码数值以及四个参数，但是，在Windows XP和Windows 2003蓝屏上，这些数值被显示在蓝屏底部。

第一行列出了传递给*KeBugCheckEx*的停止代码和四个参数。屏幕顶部附近的文本提供了该停止代码数值标识符的文本表示。根据图14.2中的例子，停止代码0x000000D1是一个DRIVER_IRQL_NOT_LESS_OR_EQUAL崩溃。当其中一个参数包含了操作系统或设备驱动程序代码的地址时（如同图14.2中显示的那样），Windows显示出该地址所处的模块的基地址、日期戳，以及该设备驱动程序的文件名。仅仅这些信息就有可能帮助你查明错误的组件。

虽然总共有一百多个不同的停止代码，但是，绝大多数代码很少在产品系统上发生（尽管也有）。相反地，少数常见的停止代码表达了大多数的Windows系统崩溃实例。

www.TopSage.com

而且，四个附加参数的含义也取决于停止代码（并不是所有的停止代码都有扩展的参数信息）。然而，查看停止代码和参数（如果适用的话）的含义至少有可能帮助你诊断出当前正在经受失败的组件（或者引发系统崩溃的硬件设备）。

你可以在Windows调试工具箱（Windows Debugging Tools）帮助文件的“Bug Checks （Blue Screens）”一节中找到关于停止代码的信息（有关Windows调试工具箱的信息，请参见第1章）。你也可以在Microsoft的在线Knowledge Base（*http://support.microsoft.com*）中查找某一个停止代码和可疑的硬件或应用程序的名称。你可能会找到有关如何规避此问题、如何更新软件的信息，或者找到关于修正此问题的服务补丁包（service pack）的信息。Windows DDK中的Bugcodes.h包含了150个左右停止代码的一个完整列表，并且给出了有关某些停止代码的原因的额外信息。

通常你是在安装了一个新的软件产品或者硬件设备以后才开始看到蓝屏的。如果你刚刚添加了一个驱动程序，再重新引导，然后在系统初始化过程的早期看到了一个蓝屏，那么，你可以重新启动机器，在出现提示时按下F8键，然后选择“Last Known Good Configuration（最后已知的好配置）”。选择了最后已知的好配置选项，可以让Windows将注册表的设备驱动程序注册键（HKLM\SYSTEM\CurrentControlSet\Services）从最近一次成功引导的配置（在你安装该驱动程序以前）中恢复一份拷贝。从最后已知好配置的角度来看，一次成功的引导是指，所有的服务和驱动程序都已经完成了加载，并且至少有一次登录已经成功了（第5章更加详细地讲述了最后已知的好配置）。

如果你仍然看到蓝屏，那么，一种显而易见的办法是，卸掉在你第一次看到蓝屏之前刚刚加入的那些组件。如果从你上次加入新组件以来已经过了一段时间，或者你差不多同时加入了好多东西，那么，你需要将任何一个参数中引用到的设备驱动程序的名称记录下来。如果你识别出这些名称中有的名称跟你刚刚加入的组件有关联（比如，如果你安装了一个新的SCSI驱动器，你正好看到了Scsiport.sys），那么，你可能已经找到了罪魁祸首。

许多设备驱动程序有隐藏的名称，但是有一种办法可以让你找到哪个应用程序或者硬件设备是跟一个名称相关联的，那就是，通过在注册表的HKLM\SYSTEM\CurrentControlSet\Services键下面搜索该设备驱动程序的名称，你可以找到与该设备驱动程序相关联的服务的名称。注册表的这一分支正是Windows保存当前系统中每个设备驱动程序注册信息的地方。如果你搜索到了一个匹配，则寻找名为DisplayName和Description的值。有些驱动程序通过填充这些值来描述设备驱动程序的用途。例如，你可能会在DisplayName值中找到字符串“Virus Scanner”，这表明你已经运行了反病毒软件。你可以在计算机管理（Computer Management）工具（从Start菜单中，选择Programs/Administrative Tools/Computer Management）中显示驱动程序的列表。在Computer Management中，展开System Tools，System Information和Software Environment，然后选择Drivers即可。

然而，停止代码和四个关联的参数有时候还不足以诊断一次系统崩溃。例如，你可能需要

www.TopSage.com

检查内核模式的调用栈，才能查明触发此次崩溃的驱动程序或者系统组件。而且，因为Windows系统上的默认行为是在系统崩溃时自动重新引导的，因此，你可能没有时间记录下在蓝屏上显示的信息。所以，Windows在默认情况下，要试图将有关系统崩溃的信息记录到磁盘上，以便于事后进行分析。这也将我们带入到下一个话题：崩溃转储文件。

14.3 崩溃转储文件

在默认情况下，所有的Windows系统都被设置为：当系统崩溃时，系统总是试图记录下有关当前系统的状态的信息。你可以通过用户界面看到这些设置，做法是，打开控制面板中的System工具，然后在System Properties对话框中，点击Advanced标签，然后点击Startup And Recovery按钮。Windows XP Professional系统的默认设置如图14.3所示。

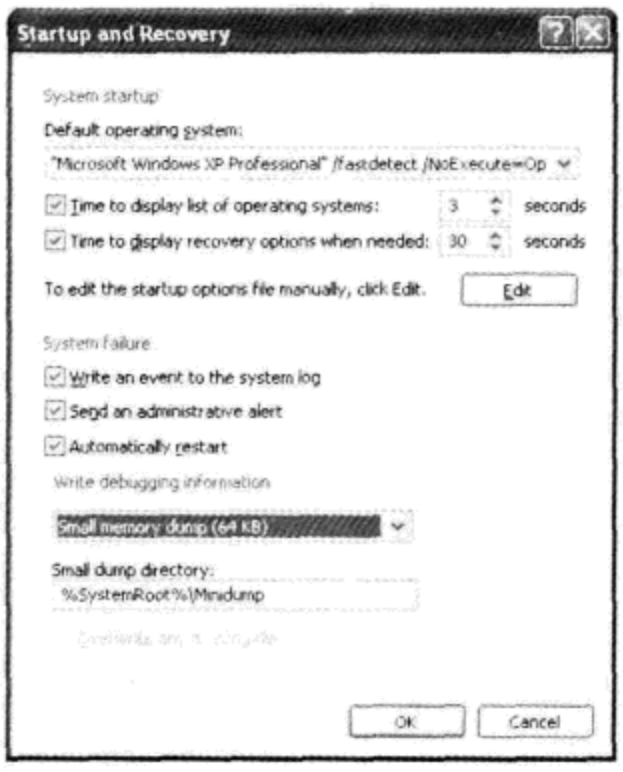


图14.3 崩溃转储设置

对于一次系统崩溃，以下三种层次的信息可以被记录下来。

- **完全内存转储** 完全内存转储包含了在崩溃时刻所有的物理内存。这种转储类型要求页面文件至少是物理内存的大小再加上1MB（用于记录头信息）。因为在大内存的系统上，它要求一个非常大的页面文件，所以，这种类型的转储文件是最少见的设置。Windows NT 4只支持这种类型的崩溃转储文件。这也是Windows Server系统的默认设置。
- **内核内存转储** 内核内存转储只包含了在崩溃时刻位于物理内存中的内核模式读/写页面。这种转储类型并不包含属于用户进程的页面。然而，因为只有内核模式的代码才可以直接导致Windows崩溃，所以，用户进程页面通常对于崩溃调试来说是不必要的。而且，

www.TopSage.com

所有与崩溃转储分析有关的数据结构，包括当前运行进程的列表、当前线程的栈，以及已加载驱动程序的列表，都被保存在非换页的内存中，它们也被记录在内核内存转储中。没有很好的办法来预测内核内存转储的大小，因为它的大小取决于该机器上操作系统和所有活动驱动程序已经分配的内核模式内存的数量。

- **小内存转储** 小内存转储（在Windows Professional上，这是默认设置）也称为**小转储**（*minidump*）或**分类优先转储**（*triage dump*），它的大小为64KB（在64位系统上为128KB）。它包含了停止代码和参数、已加载的设备驱动程序的列表、描述当前进程和线程的数据结构（称为EPROCESS和ETHREAD，见第6章的描述），以及引起此次崩溃的线程的内核栈。

虽然完全内存转储是其他选项的超集，但是，它的缺点是，其大小达到了一个系统上所有物理内存的总量，因而变得很不实用。大的服务器系统有几GB物理内存的情形并不罕见，这样会导致崩溃转储文件太大，以致无法上载到FTP服务器上，或者被刻到CD上。因为在绝大多数的崩溃分析过程中，用户模式的代码和数据并没有被使用到（因为系统崩溃是由于内核内存中存在问题而引起的，并且，系统数据结构驻留在内核内存中），所以，在完全内存转储中保存的许多数据与分析并不相关，因而它们纯粹浪费了转储文件的空间。最后一个缺点是，引导卷（\Windows目录所在的卷）上的页面文件的大小必须至少等于该系统中物理内存的数量再加上1MB。由于在一般情况下，所要求的页面文件的大小与实际物理内存的数量成反比例，因此，这一需求可能强迫页面文件变得不必要的大。你应该考虑小内存转储和内核内存转储这两种选项所提供的好处。

小转储的一个好处是它的尺寸非常小，这使得它非常便于交换，比如通过电子邮件来交换。而且，每一次崩溃都会在\Windows\Minidump目录中生成一个文件，该文件有一个惟一的文件名，是由字符串“Mini”加上日期和一个序列号（比如Mini082604-01.dmp）构成的。小转储的一个缺点是，为了对它们进行分析，你在分析时必须使用与生成该转储的系统上用到的相同映像文件（最起码，Ntoskrnl.exe必须是相符的版本，以便进行最基本的分析）。如果你想在一个不同于生成某个转储的系统上分析该转储，那么这可能是一个问题。然而，Microsoft的符号服务器包含了Windows XP系统及以后版本的映像（和符号），所以，你可以在调试器中设置映像路径，使它指向符号服务器，于是，调试器就会自动下载所需要的映像（当然，Microsoft的映像服务器并不包含你所安装的第三方驱动程序的映像）。

一个更重要的缺点是，由于在转储文件中只保存了很有限的数据，这可能会使得有效的分析难以进行。另外，即使你配置一个系统，并让它生成内核崩溃转储或完全崩溃转储，你也可以获得小转储的好处，你只需用Windbg来打开一个大的崩溃转储，然后用“**.dump /m**”命令提取出一个小转储。注意，在Windows XP和Windows Server 2003上，小转储是自动创建的，即使系统被设置为完全转储或内核转储。

www.TopSage.com



> **注** 你可以在Livekd中，利用.dump命令生成当前活动系统的一个内存映像，因而你无需停止系统就可以离线地分析该系统。若一个系统虽然出现了一个问题但仍然在提供服务，而且你想在不打断该服务的情况下诊断此问题，那么，此时这种方法非常有用。结果得到的崩溃映像并不是完全一致的，因为不同内存区域的内容反映的是不同时间点上的情形，但是它可能包含了对于分析非常有用的信息。

内核内存转储方案提供了一种更切合实际的中间选择。因为它包含了所有的属于内核模式的物理内存，所以，它与完全内存转储具有同样层次的可分析数据，但是它省略了通常不相关的用户模式数据和代码，因此，它可以比完全内存转储小得多。举例来说，在一个具有256MB RAM的Windows XP系统上，一个内核内存转储的大小为34MB；在另一个具有1.5GB RAM的Windows XP系统上，一个内核内存转储达到了72MB。

当你配置内核内存转储的时候，系统检查页面文件是否足够大（见表14.1），但是，这些只是估计的大小值，因为没有办法可以预测一个内核内存转储的大小。之所以无法预测一个内核内存转储的大小，是因为它的大小取决于在崩溃时刻该机器的操作系统和所有活动驱动程序正在使用的内核模式内存的数量。

表14.1 针对内核转储的默认最小页面文件大小

系统内存的大小	针对内核转储的最小页面文件大小
< 128 MB	50 MB
< 4 GB	200 MB
< 8 GB	400 MB
>= 8 GB	800 MB

因此，在崩溃发生时，因为页面文件太小而无法容纳一个内核转储，这是有可能的。如果你想要看一看自己系统上一个内核转储的大小，那么，你可以手工强制来一次崩溃，做法是，配置相应的选项以便你可以从控制台发起一次手工系统崩溃，或者使用本章后面将要讲述的Notmyfault工具（本章后面讲述了这两种方法）。当你重新引导时，你可以进行检查，以确保一个内核转储已经生成了，而且，你也可以检查它的大小以估算出你的引导卷页面文件应该设置为多大。为保险起见，在32位系统上，你可以选择2GB加1MB大小的页面文件，因为2GB是所有可用内核模式地址空间的最大值。

最后，即使系统成功地在崩溃时刻将崩溃转储记录到页面文件中了，它还必须有足够的空闲磁盘空间来提取此转储文件。如果没有足够的磁盘空间，那么，崩溃转储就丢失了，因为在页面文件中用于保存崩溃转储的空间被释放了，随着系统开始使用此页面文件，这部分空间也将被覆盖掉。如果你在引导卷上没有足够的空间来保存memory.dmp文件，那么，你可以通过如图14.3中显示的对话框来选择任何其他本地硬盘上的某一个位置。

www.TopSage.com

崩溃转储的生成

当系统引导时，它读取注册表值HKLM\System\CurrentControlSet\Control\CrashControl，以检查当前配置的崩溃转储选项。如果已经配置了一个转储，那么，它把用来写引导卷的磁盘小端口驱动程序在内存中做一份拷贝，并且赋予它一个与小端口相同的名称，前面加上“dump_”作为前缀。对于在写崩溃转储过程中涉及的组件，系统也计算它们的校验和，并且保存起来。这样的组件包括刚刚被拷贝的磁盘小端口驱动程序、执行崩溃转储写操作的I/O管理器函数，以及引导卷的页面文件在磁盘上的映射表。当*KeBugCheckEx*执行时，它再次计算这些组件的校验和，并且将新得到的校验和与引导时得到的校验和进行比较。如果两者不相符，那么它不再写崩溃转储文件，因为这样做可能会失败，或者破坏磁盘。如果两个校验和相匹配，那么，*KeBugCheckEx*直接将转储信息写到由页面文件占据的磁盘扇区中，从而绕过了文件系统驱动程序（它可能已被破坏，甚至正是它导致了系统崩溃）。

当SMSS在引导过程中启用换页机制时，系统检查引导卷的页面文件，看是否存在一个崩溃转储，并且将页面文件中被崩溃转储所占据的那部分保护起来。这使得在引导的早期阶段，引导卷页面文件的部分或全部是不可用的，从而可能会导致发出通知以显示“系统当前的虚拟内存过低”。当然，这一条件只是暂时的。后来，仍然在引导过程中，Winlogon通过调用未文档化的*NtQuerySystemInformation* API来确定该页面文件中是否存在一个崩溃转储，如果存在崩溃转储，则激发Savedump进程（\Windows\System32\Savedump.exe），以便从该页面文件中提取出此崩溃转储，并且将它拷贝到最终的位置上。这些步骤如图14.4所示。

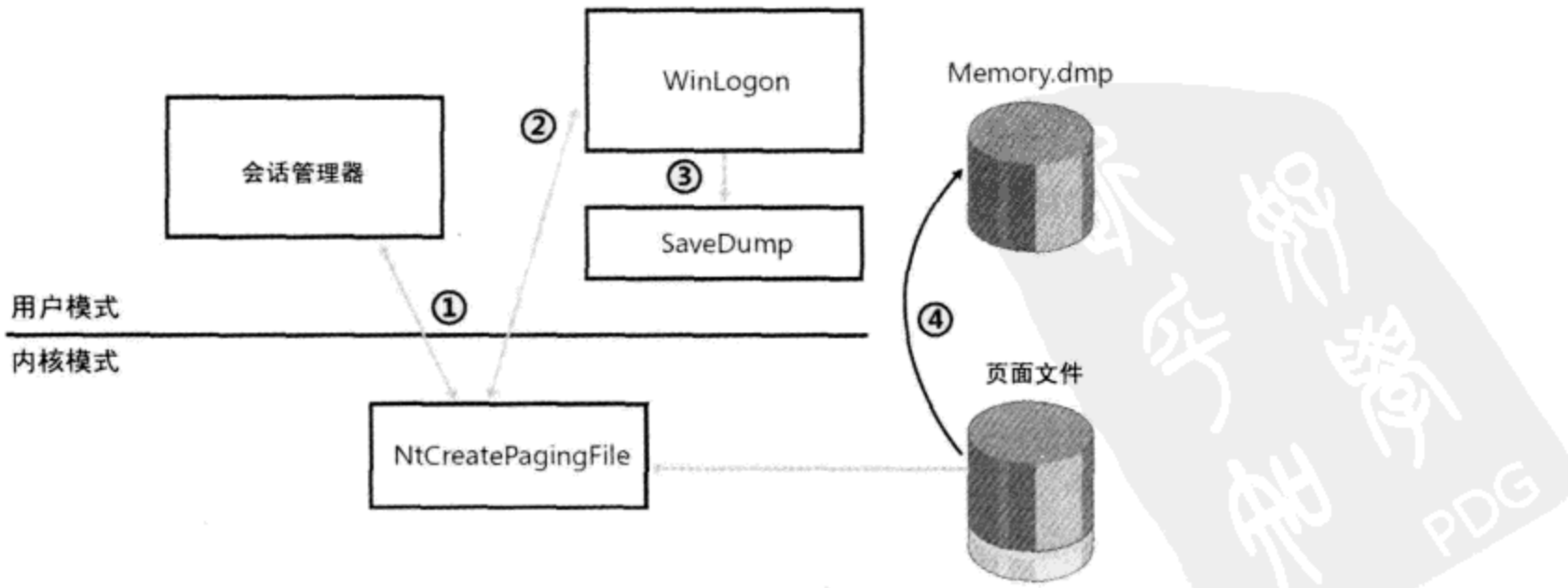


图14.4 崩溃转储文件的生成

www.TopSage.com

14.4 Windows错误报告

正如在第3章中所提到的，Windows XP和Windows Server 2003包含了一个称为Windows错误报告的设施，它可以自动帮助将进程和系统崩溃提交给Microsoft（或者一个内部错误报告服务器）进行分析。这一特性是默认打开的，但是通过改变Savedump的行为，它是可以被修改的；Savedump采取额外的步骤来确定该系统是否已被配置为“在崩溃之后重新引导起来时将崩溃转储发送给Microsoft（或者一个私有服务器，在本章后面的‘在线崩溃分析’一节中有进一步的解释）进行分析”。图14.5显示了错误报告配置对话框，你可以通过控制面板的System小程序的Advanced标签页访问此对话框。此对话框允许你配置系统的错误报告设置，这些设置信息被保存在注册表的HKLM\Software\Microsoft\PCHealth\ErrorReporting下面。

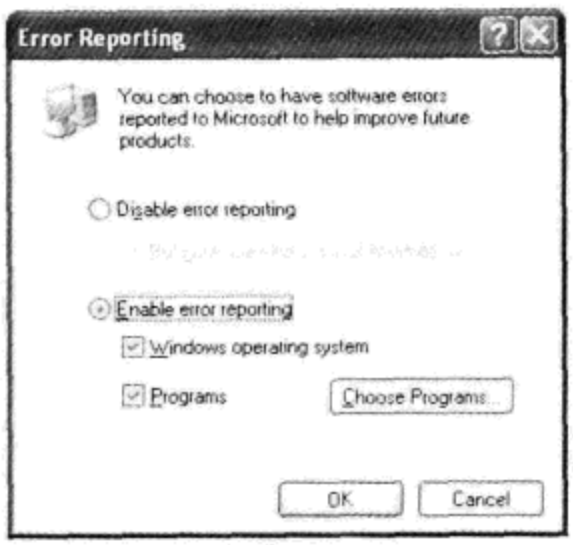


图14.5 错误报告配置对话框

在崩溃之后系统重新引导起来时，Savedump检查ErrorReporting键下的几个值，包括Showui、DoReport和IncludeKernelFaults。如果所有这三个值都是true，则Savedump使用下面的步骤，以便做好准备“向Microsoft OCA（Online Crash Analysis）站点（或者一个内部错误报告服务器，如果已经这样配置的话）发送一个崩溃转储报告”。

1. 如果它生成的转储类型不是一个小转储，那么，它从转储文件中提取出一个小转储，并且将其保存在\Windows\Minidumps的默认位置处。
2. 它将小转储文件的名称写到HKLM\Software\Microsoft\PCHealth\ErrorReporting\KernelFaults中。
3. 它在HKLM\Software\Microsoft\Windows\CurrentVersion\Run中加入一条命令以执行Dumprep （\Windows\System32\Dumprep.exe），所以，在第一个用户登录到该系统的过程中，Dumpreg将被执行。

www.TopSage.com

14.5 在线崩溃分析

如上所述，Savedump配置了要启动Dumprep工具，作为这一配置的结果，当Dumprep工具执行时，它也要检查那三个被Savedump参考过的注册表值，看一看该系统是否已被配置为“当系统从一次崩溃中重新引导起来时，向Microsoft发送一个错误报告”。如果是，则Dumprep生成一个XML格式的文件，其中包含了有关该系统的基本描述信息，包括操作系统的版本、机器上已安装的驱动程序的列表，以及在崩溃时刻系统中已加载的即插即用驱动程序的列表。然后，它出示一个如图14.6所示的对话框，询问用户是否愿意给Microsoft发送一个错误报告。如果用户选择了发送错误报告，则除非被组策略所改变，否则Dumprep将此XML文件和小转储发送至*http://Watson.Microsoft.Com*，此Web服务器又将数据转发给一个服务器场（server farm）以进行自动化分析。下一节将介绍此分析过程。管理员可以通过组策略来配置他们所管辖的系统，使这些系统将错误报告发送至一个内部的错误报告网络共享体，以便事后利用Microsoft Corporate Error Reporting（CER）工具箱进行处理。此工具箱只有符合条件的Microsoft Software Assurance（Microsoft软件保障）客户才可以使用（有关更多的信息，请参见*http://www.microsoft.com/resources/satech/cer*）。

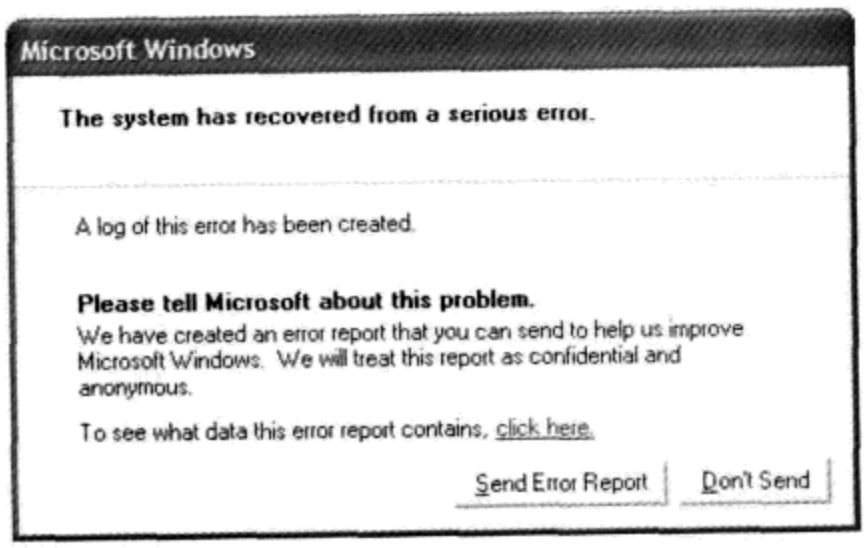


图14.6 崩溃转储错误报告对话框

服务器场的自动化分析过程所使用的分析引擎，也正是当你在Microsoft内核调试器中加载一个崩溃转储文件时（稍后介绍）内核调试器所使用的同一个分析引擎。此分析过程会得出一个**桶***ID*（*bucket ID*），这是一个用于标识某个特定崩溃类型的特征值。服务器场使用此桶ID来查询一个数据库，以确定是否有一个解决方案适用于此次崩溃；而且，它给Dumprep送回一个指向OCA Web站点（*http://oca.microsoft.com*）的URL。Dumprep启动Internet浏览器，打开OCA Web站点上包含初步的崩溃分析结果的页面。如果有一个解决方案可供使用的话，则该页面指示用户到哪儿获得一个热补丁（hotfix）、服务补丁包（service pack）或者第三方驱动程序的更新版本；否则，用户可以接受这样的选项：通过电子邮件来跟踪该崩溃分析的进展。

www.TopSage.com

对于未连接至Internet的站点，或者不想把崩溃转储自动发送至Microsoft的站点，它们可以使用组策略来配置为：错误数据被存储在一个内部的错误报告网络共享体上，以便事后利用前面提到的Microsoft CER工具箱进行处理。

14.6 基本的崩溃转储分析

如果OCA未能提供一个解决方案，或者你不能将崩溃信息提交给OCA（譬如，如果你有一个由Windows 2000生成的崩溃转储文件，这种文件不支持OCA），那么，一种可供选择的做法是，你自己来分析这些崩溃转储。正如前面所提到的，当你在Windbg和Kd中加载一个崩溃转储文件时，它们执行的分析引擎与OCA使用的分析引擎是相同的；有时候，基本的分析也能够查明问题所在。所以，你可能会很幸运，通过自动的分析就能够解开一次崩溃转储之谜。但如果你没有那么幸运，那么，也有一些直接的技术可以让你试着解决系统崩溃的问题。

这一节解释如何执行基本的崩溃分析步骤，在这一节之后将介绍一些诊断技巧，让你利用驱动程序检验器（Driver Verifier，在第7章中介绍）来捕捉有错误的驱动程序（当它们破坏系统的时候），因而通过崩溃转储分析能查出这些驱动程序。

Notmyfault

你可以使用来自*www.sysinternals.com/windowsinternals*的Notmyfault工具来产生这里所描述的崩溃。Notmyfault是由一个名为Notmyfault.exe的可执行文件和一个名为Myfault.sys的驱动程序组成的。当你运行Notmyfault可执行程序时，它加载该驱动程序，并且显示一个如图14.7所示的对话框，该对话框允许你以各种方式来崩溃系统，或者让驱动程序泄漏换页内存池。这里提供的崩溃类型代表了Microsoft产品支持服务组（Microsoft Product Support Services）看到的最常见的系统崩溃。在对话框中选择一个选项，并点击Do Bug按钮，这将使得此可执行程序通过使用*DeviceIoControl* Windows API，来告诉驱动程序应该触发哪种类型的错误。注意，你应该在一个测试系统上，或者在一个虚拟机中执行Notmyfault崩溃，因为毕竟还是有一点小小的风险：Notmyfault破坏的内存将被写到磁盘上，从而导致文件或磁盘被破坏。

www.TopSage.com

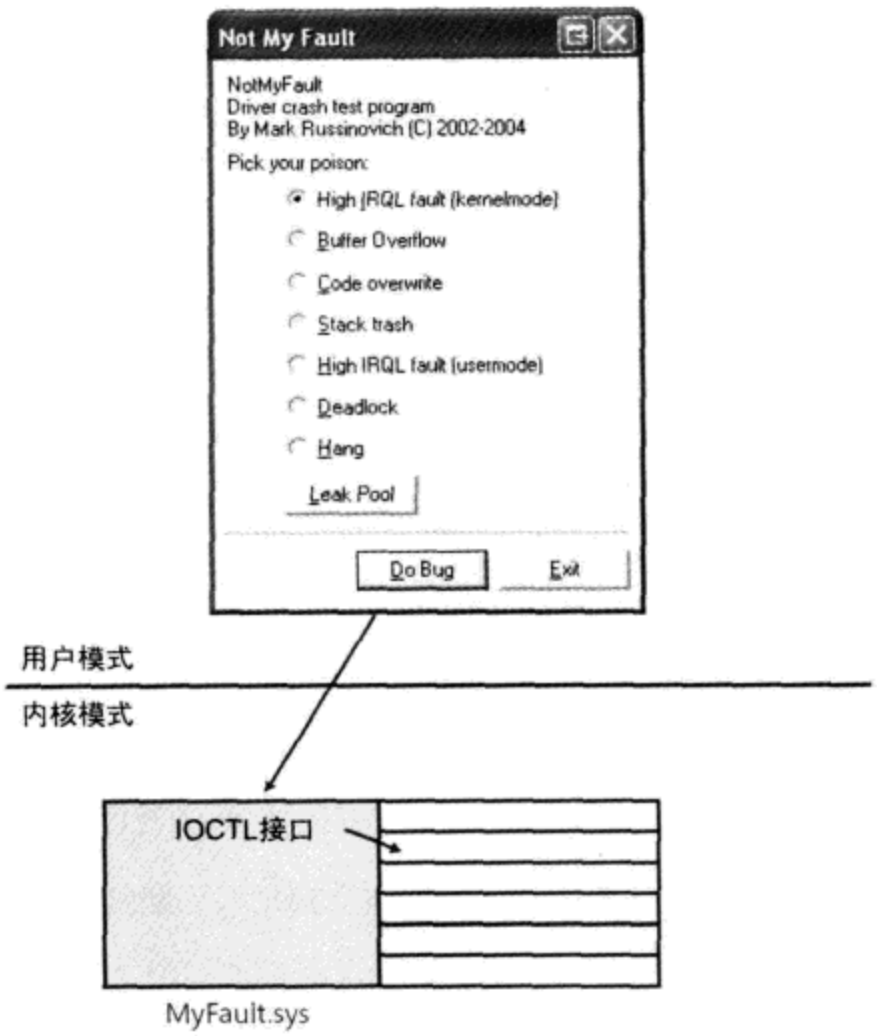


图14.7 Notmyfault



> 注 Notmyfault可执行文件的名称（“不是我的错”）和驱动程序的名称（“我的错”）也正是突出了“用户模式不可能直接导致系统崩溃”这一事实。Notmyfault可执行文件只有通过“加载一个驱动程序，以便帮它在内核模式下执行一个非法操作”才能引发系统崩溃。

基本的崩溃转储分析

最直接的Notmyfault崩溃是通过选择High IRQL Fault （Kernelmode）选项并点击Do Bug按钮而引起的崩溃。这使得驱动程序从换页池中申请一个页面，再释放该页面，并且将IRQL提升到DPC/Dispatch级别以上，然后访问刚刚被释放的页面（有关IRQL的更多信息，请参见第3章）。如果这样做还不会引起系统崩溃，则进程仍然往下执行，它越过该页面的尾部继续读内存，直至由于访问无效页面而引起系统崩溃。结果是，该驱动程序执行了以下几个非法操作。

1. 它引用了不再属于它的内存。
2. 它在DPC/Dispatch级别或更高级别上引用了换页内存池，这是非法的，因为当处理器的IRQL是DPC/Dispatch级别或更高级别时，页面错误是不允许的。

www.TopSage.com

3. 当它越过了它所申请的内存的尾部时，它试图引用一段可能无效的内存。

第一次页面引用可能不会引发系统崩溃，其原因是，如果驱动程序释放的页面仍然在系统工作集中，那么它不会产生一个页面错误（有关系统工作集的信息，请参见第7章）。

当你将一个由该错误产生的崩溃转储加载到Kd中时，它的分析将显示类似如下的信息：

```
Microsoft (R) Windows Debugger  Version 6.3.0011.2
Copyright (c) Microsoft Corporation. All rights reserved.


Loading Dump File [c:\windows\memory.dmp]
Kernel Summary Dump File: Only kernel address space is available

Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/download/symbols

Executable search path is:
Windows 2000 Kernel Version 2195 UP Free x86 compatible
Product: LanManNt, suite: Enterprise TerminalServer
Kernel base = 0x80400000 PsLoadedModuleList = 0x8046a4c0
Debug session time: Mon Apr 05 18:28:44 2004
System Uptime: 0 days 0:09:20.105
Loading Kernel Symbols
...............................................................................
..............
Loading unloaded module list
No unloaded module list present
Loading User Symbols
..........
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.

BugCheck D1, {bead1800, 1c, 0, ec1fb357}

*** ERROR: Module load completed but symbols could not be loaded for myfault.sys

*** WARNING: Unable to verify checksum for NotMyfault.exe
*** ERROR: Module load completed but symbols could not be loaded for NotMyfault.exe
Probably caused by : myfault.sys ( myfault+357 )

Followup: MachineOwner
---------

kd>
```


www.TopSage.com

注意到的第一件事情是，Kd报告错误：它不能加载Myfault.sys和Notmyfault.exe的符号。这些错误是意料之中的，因为它们的符号文件不在符号文件路径（它被配置为指向Microsoft的符号服务器）上。对于不随操作系统一起发行的第三方驱动程序和可执行程序，你也会看到类似的错误。

分析文本本身非常简洁，仅仅显示了数值化的停止代码和错误检查参数，后面跟着“probably caused by”行，显示了分析引擎对于有错误驱动程序的最佳猜测。在这个例子中，它切中要害，直接指出了Myfault.sys，所以，无需再进行手工分析了。

“Followup”行通常并没有用，这是专门给Microsoft内部使用的，调试器在“Debugging Tools for Windows（Windows调试工具箱）”安装目录的Triage目录下的Triage.ini文件中查找该模块的名称。此文件的Microsoft内部版本列出了负责处理“在某一个特定驱动程序中发生的崩溃”的开发人员或开发组，因而调试器在适当的时候将他们的名字显示在Followup这一行中。

详细的分析

尽管Notmyfault崩溃的基本分析已经标识出了有错误的驱动程序，但你总是应该让调试器执行一次详细的分析，做法很简单，输入下面的命令：

!analyze –v

在详细分析和默认分析之间的第一个显然的区别是有关停止代码及其参数的描述。当在上述的崩溃转储上执行详细分析命令时，输出如下：

```
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
Arguments:
Arg1: bead1800, memory referenced
Arg2: 0000001c, IRQL
Arg3: 00000000, value 0 = read operation, 1 = write operation
Arg4: ec1fb357, address which referenced memory
```

这可以让你省下打开帮助文件来查找同样这些信息的麻烦，有时候，这些文本描述给出了有关诊断步骤的建议，在下一节有关高级崩溃转储分析的介绍中你将会看到一个例子。

在一个详细分析中，其他可能有用的信息是，崩溃时刻在发生崩溃的处理器上正在执行的线程的栈痕迹。以下是前面的转储中的栈痕迹信息：

```
STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be wrong.
```


www.TopSage.com

```
b7e6dc34 804ac5de ff7c4040 beb4bf68 ff80a9c8 myfault+0x357
b7e6dd00 804a8f1e 00000070 00000000 00000000 nt!IopXxxControlFile+0x5e4
b7e6dd34 80461691 00000070 00000000 00000000 nt!NtDeviceIoControlFile+0x28
b7e6dd34 77f96be2 00000070 00000000 00000000 nt!KiSystemService+0xc4
0012f4a0 77e84c9b 00000070 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xb
0012f504 004017c3 00000070 83360018 00000000 KERNEL32!DeviceIoControl+0x100
000200ac 00000000 00000000 00000000 00000000 NotMyfault+0x17c3
```

上面的栈痕迹显示了Notmyfault可执行映像（如栈的底部所示）调用了Kernel32.dll中的*DeviceIoControl*函数，而*DeviceIoControl*又依次调用了Ntdll.dll中的ZwDeviceIoControlFile，如此等等，直至最后，在Myfault映像中有一条指令执行之后系统发生崩溃。像这样的栈痕迹可能非常有用，因为有时候，崩溃的发生是由于一个驱动程序给另一个驱动程序传递了格式不正确的、已被破坏的、或者非法的参数信息。接受此无效数据的驱动程序可能引发了崩溃，从而在基本分析中被蒙冤，但是，栈痕迹指明了另一个驱动程序也被牵连进来。在以上的例子中，除了Myfault以外，没有其他的驱动程序被列示出来（模块“nt”是Ntoskrnl）。

如果在一次分析中被选中的驱动程序对你来说并不熟悉，那么，使用lm（列出模块）命令可以查看该驱动程序的版本信息。加上k（内核模块）和v（verbose，详细）选项，以及m选项，后面再跟上该驱动程序的名称和一个通配符，如下所示：

```
kd> lm kv m myfault*
start    end        module name
f224d000 f224dbe0   myfault      (deferred)
    Image path: \??\C:\WINNT\system32\drivers\myfault.sys
    Timestamp: Thu Apr 29 14:53:12 2004 (40915D28)  Checksum: 00010090
    ImageSize : 00000BE0
    File version:     2.0.0.0
    Product version:  2.0.0.0
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        3.7 Driver
    File date:        00000000.00000000
    Translations:     0409.04b0
    CompanyName:      Sysinternals
    ProductName:      Sysinternals Myfault
    InternalName:     myfault.sys
    OriginalFilename: myfault.sys
    ProductVersion:   2.0
    FileVersion:      2.0
    FileDescription:  Crash Test Driver
    LegalCopyright:   Copyright (C) M. Russinovich 2002-2004
```

除了通过描述信息来识别出一个驱动程序的用途以外，你还可以使用该文件和产品的版本号来确定所安装的版本是否是最新的可使用版本（例如，你可以检查厂商的Web站点来做到这一点）。如果版本信息没有出现的话（因为在崩溃时刻这些信息可能被换出物理内存了），那么，你可以在发生崩溃的系统上，在Explorer（文件管理器）中检查该驱动程序映像文件的属性。

www.TopSage.com

14.7 使用崩溃诊断工具

在上一节中通过Notmyfault的“High IRQL Fault（Kernelmode）”选项生成的崩溃转储，并没有对调试器的自动化分析带来多大的挑战。不幸的是，大多数崩溃并不是这么容易，而且通常不可能进行调试。按照系统性能退化的程度，有几个严重程度逐渐增加的级别可能使一个系统本来生成不可分析的崩溃转储，变为生成可被分析的崩溃转储。如果在你配置了一个级别并且重新引导以后，所生成的崩溃转储并没有表露出崩溃的原因，则试一试下一个级别。

1. 如果你正在怀疑有一个或者多个驱动程序可能是崩溃的源头，因为它们是最近才被引入到系统中的，或者它们最近刚刚被更新过，或者崩溃的场景暗示着它们可能是罪魁祸首，那么，对这些驱动程序使用驱动程序检验器（Driver Verifier）进行检验，除了低资源模拟选项以外，检查所有其他的选项。

2. 对于系统中未签名的所有驱动程序，启用第1级别中的检验选项。或者，如果你正在运行Windows 2000，而在Windows 2000上，驱动程序检验器并不区分签名的和未签名的驱动程序，那么，你可以针对所有的非Microsoft的驱动程序，启用第1级别中的检验选项。

3. 对于系统中的所有驱动程序，启用第1级别中的检验选项。为了维持合理的性能，你可能想要将驱动程序分成组，这样，每次重新引导时，只针对一个组启用驱动程序检验器。

显然，在你花时间和精力来调整系统配置和分析崩溃转储以前，首先应该确保你的系统中的内核和驱动程序都是最新的版本，你可以通过Windows Update以及第三方驱动程序支持站点的服务来做到这一点。



> **注** 如果因为驱动程序检验器检测到了一个驱动程序错误并且使系统崩溃，从而导致你的系统变得不可引导，那么，请在安全模式下启动起来（在这种模式下是禁止检验的），运行驱动程序检验器，并删除检验设置。

下面的小节解释了驱动程序检验器如何能够使不可能调试的崩溃变成你可以解决的崩溃。你也应该参考调试工具箱（Debugging Tools）的帮助文件，其中有一些关于高级调试技术的指导。

www.TopSage.com

缓冲区溢出和特殊内存池

到目前为止，在Windows上，最常见的崩溃源头是内存池被破坏。内存池破坏通常发生于一个驱动程序遭受了一个缓冲区上溢或缓冲区下溢错误的时候，也就是说，这样的错误改写了该驱动程序从换页内存池或非换业内存池中申请的缓冲区的开始之前或结束之后的数据。执行体的池跟踪（pool tracking）数据结构驻留在池缓冲区的两端，它们把内存池的缓冲区相互隔开。因此，这些错误会破坏池跟踪数据结构，或者破坏其他驱动程序的缓冲区，或者对两者皆造成破坏。由于内存池破坏而引起的崩溃本质上是不可能调试的，因为系统崩溃发生于破坏的数据被引用之时，而并非发生于数据被破坏之时。



> **注** 为了帮助捕获这些难以捕获的内存破坏，Windows XP （Service Pack 2）及以后的版本总是执行池内存块的尾部检查。因此，缓冲区溢出可能会立刻引发一个BAD_POOL_HEADER崩溃。

你只需运行Notmyfault并选择“Buffer Overflow”错误，就可以产生一个内存池破坏类型的崩溃。这一选项使得Myfault申请一个缓冲区，然后改写该缓冲区之后的40字节。从你点击Do Bug按钮到发生崩溃，两者之间可能会有一段明显的延迟，甚至在崩溃发生之前你可能要运行或操作某些应用程序以便池内存被使用到，这也更加突出了从内存破坏到系统稳定受到影响之间的距离。对这样的崩溃转储的分析，几乎总是报告Ntoskrnl或另一个驱动程序是可能的出错原因，这也说明了详细分析以及它对停止代码的描述是非常有用的：

```
DRIVER_CORRUPTED_EXPOOL (c5)
An attempt was made to access a pageable (or completely invalid) address at an
interrupt request level (IRQL) that is too high.  This is
caused by drivers that have corrupted the system pool.  Run the driver
verifier against any new (or suspect) drivers, and if that doesn't turn up
the culprit, then use gflags to enable special pool.
Arguments:
Arg1: 4f4f4f53, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, value 0 = read operation, 1 = write operation
Arg4: 80467139, address which referenced memory
```

在描述文本中给出的建议是，针对任何新的或可疑的驱动程序运行驱动程序检验器，或者使用GFlags来启用特殊内存池特性。两者完成的事情是一样的：让系统在内存破坏发生的时候就检测到它，并且通过一种专门的方式使系统崩溃，从而允许自动分析过程能够直接指出是哪个驱动程序导致内存破坏。

如果驱动程序检验器的特殊内存池选项被启用，那么，被检验的驱动程序对于略小于一个页面大小的缓冲区申请，都使用特殊的内存池，而不使用换页的或非换页的内存池。从特殊内存池中分配的缓冲区被夹在两个无效页面之间，在默认情况下跟页面的顶端对齐。特殊内存池的例程也会用一些随机模式来填充所分配页面中未被使用的部分。图14.8显示了一个特殊内存池的缓冲区分配。

www.TopSage.com

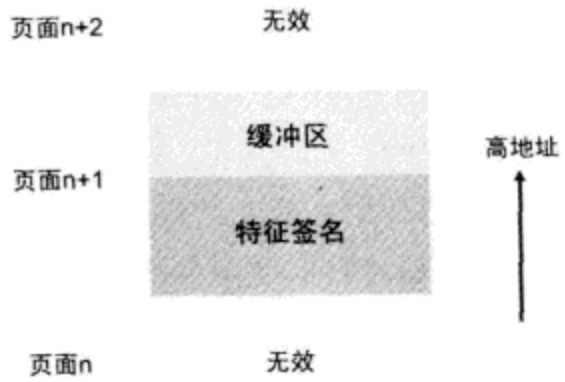


图14.8 特殊内存池的缓冲区分配

对于小于一个页面大小的缓冲区的溢出，系统在溢出发生的时候就会检测到，因为它们导致在缓冲区之后的无效页面发生了页面错误（page fault）。图14.8中的特征签名被用于在驱动程序释放一个缓冲区时捕捉缓冲区下溢的情形，因为在分配该缓冲区时放在这里的模式的完整性可能已经被破坏了。

为了看清楚如何使用特殊内存池来引发一个可让分析引擎很容易诊断的崩溃，请运行驱动程序检验器管理器。在Windows 2000系统上，转到Setting标签页，在页面底部，允许你在指定额外驱动程序的编辑框中输入**myfault.sys**，并选择特殊内存池复选框，应用（apply）这些变化，退出驱动程序检验器管理器，并重新引导。在Windows XP和Windows Server 2003上，在向导的第一个页面上选择“Create Custom Settings （For Code Developers）”选项，在第二个页面上选择“Select Individual Settings From A Full List”，然后选中“Special Pool”。在接下来的页面上选择“Select Drivers From A List”选项，然后在列出驱动程序的页面上，按下按钮以便加入尚未加载的驱动程序，并且在File Find对话框中输入“**myfault.sys**”（你在File Find对话框中不必找到myfault.sys，只需输入其名称即可）。然后，选中myfault.sys驱动程序，退出向导，并重新引导系统。

当你运行Notmyfault，并引发一个缓冲区溢出时，系统将立即崩溃，对崩溃转储的分析报告如下：

```
Probably caused by : myfault.sys ( myfault+3f1 )
```

详细分析描述的停止代码如下：

```
DRIVER_PAGE_FAULT_BEYOND_END_OF_ALLOCATION (d6)
N bytes of memory was allocated and more than N bytes are being referenced.
This cannot be protected by try-except.
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
```


www.TopSage.com

```
Arg1: beb50000, memory referenced
Arg2: 00000001, value 0 = read operation, 1 = write operation
Arg3: ec3473f1, if non-zero, the address which referenced memory.
Arg4: 00000000, (reserved)
```

特殊内存池让一个难以捕捉的错误变成一个可立即暴露其自身的错误，从而使分析工作变得更加容易。

代码改写和系统代码写保护

如果驱动程序有一个可导致破坏或者不正确地解释其自身数据结构的错误，那么，当它把已破坏的数据解释成一个内存指针值的时候，它可能会引用到不属于它自己的内存。该指针所指的目标可以是虚拟地址空间中的任何位置，包括属于其他驱动程序的数据、无效的内存，或者其他驱动程序或内核的代码。对于缓冲区溢出，在数据破坏被检测到（从而致使系统崩溃）以前，要想标识出是哪个驱动程序导致了数据被破坏，通常是不可能的。启用了特殊内存池选项，可以增加捕捉到这种野指针错误的机会，但它无法捕捉到代码被破坏的情形。

当你运行Notmyfault并选择了“Code Overwrite”选项时，Myfault驱动程序破坏掉*NtReadFile*内核函数的入口点。在这时候将会发生两件事情之一：如果你的系统正在运行Windows 2000，并且只有127MB或更少的物理内存，或者你正在运行Windows XP或Windows Server 2003，并且只有255MB或更少的物理内存，那么，你将得到一个崩溃转储，并且崩溃转储分析会指出Myfault.sys。在详细分析中显示的停止代码描述文本将告诉你，Myfault试图对只读内存进行写操作，如下所示：

```
ATTEMPTED_WRITE_TO_READONLY_MEMORY (be)
An attempt was made to write to readonly memory.  The guilty driver is on the
stack trace (and is typically the current instruction pointer).
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: 804bb7fd, Virtual address for the attempted write.
Arg2: 004bb121, PTE contents.
Arg3: b804db60, (reserved)
Arg4: 0000000b, (reserved)
```

然而，如果你运行的Windows 2000有超过127MB的内存，或者Windows XP和Windows Server 2003有超过255MB的内存，那么，你将会得到一种不同类型的崩溃，因为试图破坏内存的那个点不会被捕捉到。*NtReadFile*是一个常常被执行的系统服务，而Win32子系统利用它来读取键盘和鼠标的输入，因此，当一个线程企图执行已被破坏的代码，从而产生一个非法指令错误时，系统几乎立即就会崩溃。由这个错误而产生的崩溃转储的分析总是不正确的，但它可能不尽相同，Windows.sys和Ntoskrnl.exe常常被分析器猜测为应该对该错误负责。这些崩溃转储的错误检查描述文本如下：

```
KMODE_EXCEPTION_NOT_HANDLED (1e)
This is a very common bugcheck.  Usually the exception address pinpoints
the driver/function that caused the problem.  Always note this address
```


www.TopSage.com

```
as well as the link date of the driver/image that contains this address. Arguments:
Arg1: c0000005, The exception code that was not handled
Arg2: 80461885, The address that the exception occurred at
Arg3: 00000000, Parameter 0 of the exception
Arg4: 00000000, Parameter 1 of the exception
```

这里，在不同的配置上有不同的行为，其原因跟Windows 2000中引入的一种称为“**系统代码写保护**（*system code write protection*）”的机制有关。表14.2显示了在默认情况下，在哪些配置上，此系统代码写保护机制是打开的。

表14.2 有关系统代码写保护的配置

	Windows 2000	Windows XP和Windows Server 2003
禁止系统代码写保护	RAM > 127 MB	RAM > 255 MB

如果系统代码写保护是打开的，那么，内存管理器就会使用标准的物理页面（在x86和x64上为4KB、在IA64上为8KB）来映射Ntoskrnl.exe、HAL和所有引导驱动程序。因为映像文件中的保护粒度是标准的页面大小，所以，内存管理器可以对代码页面进行写保护，因而，任何企图修改代码页面的动作都会产生一个访问错误（如同在上面第一个崩溃转储中看到的那样）。然而，当系统代码写保护被禁止时，内存管理器使用大页面（在x86上为4MB，在IA64和x86-64上为16MB）来映射Ntoskrnl.exe，这包括在Windows 2000上RAM超过127MB的系统，以及在Windows XP或Windows Server 2003上RAM超过255MB的系统。它无法保护代码，因为代码和数据可能驻留在同一个页面中。

如果系统代码写保护是关闭的，并且崩溃分析不太可能报告出一次崩溃的原因，或者你怀疑是代码被破坏，那么，你应该启用此保护选项。利用驱动程序检验器来检验至少一个驱动程序，这是启用此选项的最容易方法。你也可以用手工在HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management下面增加两个注册表值来启用此保护选项。指定在多少内存数量的时候内存管理器使用大页面而不使用标准页面来映射Ntoskrnl.exe，你将该值设置成一个实际的无限大值。为了做到这一点，你可以创建一个名为LargePageMinimum的DWORD值，并且将其设置为0xFFFFFFFF。然后，你再加入另一个名为EnforceWriteProtection的DWORD值，并且将其设置为1。在做了这些改变以后，你必须重新引导机器，这些设置才能生效。



注 当调试器访问一个被包含在崩溃转储中的映像文件时，分析过程内部执行*!chkimg*调试器命令来检验该崩溃转储中的映像拷贝是否跟磁盘上的映像相符，并报告出任何的不同之处。请注意，如果你已经启用了驱动程序检验器，那么，*chkimg*总是报告Ntoskrnl.exe中的差异。

14.8 高级的崩溃转储分析

上一节利用驱动程序检验器来创建可让调试器的自动化分析引擎能够解释的崩溃转储。

www.TopSage.com

你可能仍然会碰到一些无法让系统产生很容易分析的崩溃转储的情形，如果是这样，你将需要执行以下的手工的分析来试着确定问题之所在。

- 使用“*!process 0 0*”调试器命令来查看哪些进程正在运行，并确保你了解每一个进程的用途。试着禁止或者卸载掉不必要的应用程序和Windows服务。
- 使用*lm*命令和*kv*选项，列出已加载的内核模式驱动程序。确保你理解任何第三方驱动程序的用途，以及你安装的是它们的最新版本。
- 使用*!vm*命令，看一看该系统是否耗尽了虚拟内存、换页内存池或非换页内存池。如果虚拟内存已被耗尽，则已提交的页面将接近于提交限制（commit limit），所以，你可以检查进程列表，看哪一个进程报告了很高的提交内存量，以期望找到一个可能的内存泄漏错误。如果非换页内存池或换页内存池已被耗尽（也就是说，使用量已接近于最大值），则看一看第7章中的“实验：诊断内存池泄漏”。

还有其他一些调试命令可以提供有用的、但是更加高级的知识，它们对于崩溃转储分析也是很有用的。*!irp*命令就是其中之一。下一小节叙述了通过该命令来找到一个可疑驱动程序的方法。

栈破坏

栈溢出（stack overrun）或栈破坏（Stack Trash）是由于缓冲区上溢或下溢错误造成的。然而，目标缓冲区并不像你在Notmyfault的缓冲区溢出错误中看到的那样位于内存池中，而是位于执行此错误（bug）的线程的栈中。这种类型的错误是另一种很难调试的错误，因为栈是任何崩溃转储分析的基础。

当你运行Notmyfault并选择了Stack Trash时，Myfault驱动程序溢出一个缓冲区，该缓冲区是它在当前执行线程的内核栈中分配的。当Myfault试图将控制权返回给发起此调用的Ntoskrnl函数时，它从栈中读取返回地址，这是该线程应该继续往下执行的地址。该地址已经被栈缓冲区溢出破坏了，所以，该线程在内存中另外某一个地址处继续执行——该地址处可能根本没有包含代码。该线程执行到一条非法CPU指令或者它引用了无效内存时，就会引发一个非法异常并崩溃。

栈溢出的崩溃转储分析把错误的职责归结到哪个驱动程序上，这将随着每次崩溃而有所不同，但是，停止代码几乎总是KMODE_EXCEPTION_NOT_HANDLED。如果你执行一次详细的分析，则栈痕迹看起来如下所示：

```
STACK_TEXT:
b7b0ebd4 00000000 00000000 00000000 00000000 0x0
```

这就像是内核栈已经被用零来改写过一样。不幸的是，像特殊内存池和系统代码写保护这样的机制无法捕捉到这种类型的错误。相反地，你必须采取一些手工的分析步骤，来间接地确定在破坏发生的时候，哪个驱动程序正在执行。一种办法是，对于在栈破坏时刻正在执行的线

www.TopSage.com

程，检查它的正在进行之中的IRP。当一个线程发出一个I/O请求时，I/O管理器将一个指针（指向这一未完成的IRP）保存在该线程的ETHREAD结构的IRP列表上。调试器命令*!thread*可以转储出一个线程的IRP列表。（如果你没有指定一个线程对象地址，则*!thread*转储出该处理器的当前线程。）然后，你可以利用*!irp*命令来查看该IRP。

```
kd> !thread
THREAD ff740020  Cid 8f8.420  Teb: 7ffde000  Win32Thread: a20cdbe8 RUNNING
IRP List:
    bc5a7f68: (0006,0094) Flags: 00000000  Mdl: 00000000
Not impersonating
Owning Process ff75f120
...

kd> !irp bc5a7f68
Irp is active with 1 stacks 1 is current (= 0xbc5a7fd8)
 No Mdl Thread ff740020:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
>[  e, 0]   0  0 ff79e4c0 ff7ac028 00000000-00000000
       \Driver\MYFAULT Args: 00000000 00000000 83360010 00000000
```

上面的输出显示了IRP的当前栈单元（由“>”前缀指明的栈单元，在这里也是惟一的栈单元）属于Myfault驱动程序。如果这是一个实际系统的崩溃，那么，接下来的步骤是，确保所安装的驱动程序版本是最新的版本，如果不是的话，则安装新的版本；如果是的话，则针对该驱动程序启用驱动程序检验器（打开除了低内存模拟以外的所有选项）。

挂起的系统或无响应的系统

如果一个系统变得不再响应了（也就是说，对于你的键盘和鼠标输入，你没有收到任何响应），鼠标被冻结了，或者你虽然可以移动鼠标，但是系统不再响应鼠标点击事件，那么，该系统称为已经**挂起**（hung）了。以下的一些事情可以导致一个系统被挂起。

- 一个设备驱动程序没有从它的中断服务（ISR）例程或者延迟过程调用（DPC）例程中返回。
- 一个高优先级的实时线程抢占了窗口系统驱动程序的输入线程。
- 在内核模式中发生了死锁（两个线程或处理器相互之间持有对方想要的资源，而且都不愿意放弃自己的资源所有权）。

如果你的系统是Windows XP或Windows Server 2003，那么，你可以利用驱动程序检验器中的一个称为“死锁检测（*deadlock detection*）”的选项，来检查死锁。这一死锁检测功能会监视自旋锁、快速互斥体和互斥体的使用情况，查找可能会导致死锁的模式（有关这些以及其他同步原语的更多信息，请参见第3章）。 如果找到了死锁的模式，则驱动程序检验器使当前系统崩溃，并且指明哪个驱动程序引起了死锁。当两个线程都持有对方想要的资源，并且都不愿意放弃自己已有的资源或放弃对期望资源的等待时，最简单形式的死锁就会发生。如果你正

www.TopSage.com

在运行Windows XP或Windows Server 2003，则诊断一个挂起的系统的第一个步骤是，对于可疑的驱动程序启用死锁检测特性，然后对于未签名的驱动程序，再对于所有的驱动程序都启用死锁检测特性，直至你得到一个崩溃转储，它指明了是哪个驱动程序引起了死锁。

如果你正在运行Windows 2000或者已经检验了所有的驱动程序，但系统仍然被挂起，那么，你必须要么手工使此挂起的系统崩溃，并分析所得到的崩溃转储，要么用内核调试器进入系统，采取另外的方法来检查挂起的原因。

有两种方法可允许你进入到一个挂起的系统中，从而你可以应用本章中讲述的手工崩溃诊断技术来确定是哪个驱动程序或组件导致了系统挂起：第一种方法是，让已挂起的系统崩溃，希望得到一个能够进行分析的崩溃转储；第二种方法是，用内核调试器进入该系统，并分析系统的行为。这两种方法都要求事先建立准备环境，再重新引导。你可以在这两种方法中，使用同样的系统状态分析手段，来试着确定挂起的原因。

为了用手工使一个已挂起的系统崩溃，你必须首先加入一个DWORD注册表值HKLM\System\CurrentControlSet\Services\i8042prt\Parameters\CrashOnCtrlScroll，并将其设置为1。在重新引导以后，i8042端口驱动程序（它是PS2键盘输入的端口驱动程序）在它的中断服务例程（ISR，第3章中有进一步的讨论）中监视键盘的键击，寻找两次ScrollLock键击，同时右Ctrl键也被压下。当该驱动程序看到这样的序列，它调用*KeBugCheckEx*，并且停止代码为MANUALLY_INITIATED_CRASH （0xE2），表明这是一个由用户手工触发的崩溃。当系统重新引导起来时，打开崩溃转储文件，并应用本章前面提到的技术，来试着确定系统为什么被挂起（例如，确定当系统挂起时哪个线程正在运行，其内核栈指明了当时正在发生什么事情，等等）。注意，这种做法对于绝大多数挂起的系统是行得通的，但如果i8042端口驱动程序的ISR没有执行，则这种做法就不能生效了（如果所有处理器的IRQL都高于此ISR的IRQL，从而导致它们全部被挂起，或者系统数据结构的破坏波及了与中断相关的代码或数据，那么，i8042端口驱动程序的ISR将不会执行）。



> **注** 利用i8042端口驱动程序中提供的支持来用手工使一个已挂起的系统崩溃，这种做法对于USB键盘行不通。它只能跟PS2键盘一起工作。

另一种触发系统崩溃的方法是，如果你的硬件有一个内置的“崩溃”按钮（有些高端的服务器有这样的按钮），在这种情况下，通过给系统主板的不可屏蔽中断（NMI）针脚发送一个信号，则可以触发崩溃。为了启用这种能力，你可以将DWORD注册表值HKLM\System\CurrentControlSet\Control\CrashControl\NMICrashDump设置为1。然后，当你按下转储开关时，系统就会接到一个NMI，内核的NMI中断处理器调用*KeBugCheckEx*。这种方法比i8042端口驱动程序的机制可以在更多的场合下使用，因为NMI IRQL总是高于i8042端口驱动程序中断的IRQL。有关更多的信息，请参见*http://www.microsoft.com/hwdev/platform/proc/dmpsw.asp*。

www.TopSage.com

如果你不能用手工生成一个崩溃转储，那么，你可以首先在系统引导的时候让系统进入调试模式，再试图进入已挂起的系统。有两种方法可以让系统进入调试模式：你可以在引导的时候按下F8键，并选择Debugging Mode；或者，你可以在Boot.ini中创建一个调试模式引导选项，做法很简单，你只需从系统的Boot.ini中复制一个已有的引导条目，然后加上/DEBUG开关。如果你使用F8键这种做法，那么，系统将使用默认的连接（串行端口COM2和19200波特率）。若使用/DEBUG方案，你必须也要配置在“运行内核调试器的主机系统”和“引导进入调试模式的目标系统”之间的连接机制，然后针对此连接类型配置适当的/Debugport和/Baudrate开关。两种连接类型为：使用串行端口的零调制解调器线缆；或者，对于Windows XP和Windows Server 2003系统，在每个系统上使用1394端口的IEEE 1394（Firewire）线缆。关于如何配置调试主机和目标系统来进行内核调试的详细信息，请参见Windows调试工具箱的帮助文件。

当引导进入调试模式时，系统在引导时刻加载内核调试器，并且准备好跟串行线缆或IEEE 1394线缆另一端的计算机上正在运行的内核调试器建立一个连接。注意，内核调试器的存在并不会影响性能。当系统挂起时，在被连接的系统上运行Windbg或Kd调试器，建立一个内核调试连接，并进入到已挂起的系统中。如果中断已被禁止，或者其内核调试器已经被破坏，那么，这种方法也不能奏效。



> **注** 引导一个系统进入到调试模式下，如果它没有被连接到另一个系统，则并不会影响性能；然而，对于一个已被配置成一旦崩溃即自动重新引导的系统，如果它在引导时启用了内核调试特性，则它不会自动重新引导，因为在崩溃以后，其内核调试器将等待跟另一个系统建立一个连接。

当你执行分析时，你可以不让系统保留在停止的状态，而是使用调试器的“.dump”命令，在主调试器机器上创建一个崩溃转储文件。然后，你可以重新引导被挂起的系统，并且以离线的方式来分析这一崩溃转储（或者提交给Microsoft）。注意，如果你是使用串行的零调制解调器线缆来连接两台计算机的，则可能要花一点时间（相比较于高速的1394连接），所以，你可能会使用“.dump /m”命令，仅仅捕捉一个小转储。另外一种做法是，如果目标主机还有能力写一个崩溃转储，那么，你可以从调试器中发出“.crash”命令，以强迫它写一个崩溃转储。这将使得目标机器在它的本地硬盘驱动器中创建一个崩溃转储，当系统重新引导以后你可以检查此转储。

你可以运行Notmyfault并选择Hang选项，以产生一次挂起情形。这使得Myfault驱动程序在系统的每个处理器上排队一个DPC，并且该DPC执行一个无限循环。因为处理器在执行DPC函数时的IRQL是DPC/Dispatch级别，所以，键盘ISR将会响应特殊的崩溃键击序列。

www.TopSage.com

如果你已经在调试器中进入了一个已挂起的系统，或者加载了一个来自于挂起系统中的、手工生成的转储，那么，你应该执行*!analyze*命令，并带上*-hang*选项。这使得调试器检查该系统中的锁，并试图确定是否存在一个死锁，如果有死锁的话，确定哪个（或哪些）驱动程序被牵连进去。然而，对于像Notmyfault生成的挂起情形，*!analyze*分析命令不会报告任何有用的信息。

如果*!analyze*命令没有查明问题之所在，那么，在该转储的每个CPU环境中执行*!thread*和*!process*，看一看每个处理器正在做什么（利用~命令可切换CPU环境，例如，使用*~1*切换到1号处理器的环境中）。如果由于有一个线程在DPC/Dispatch级别或更高的IRQL上执行一个无限循环，从而挂起了系统，那么，你将会在*!thread*命令的栈痕迹中看到，该线程是在哪个驱动程序模块中被停住了。当你使一个正在经历Notmyfault挂起错误的系统崩溃时，你得到的崩溃转储的栈痕迹看起来如下所示：

```
STACK_TEXT:
f9e66ed8 f9b0d681 000000e2 00000000 00000000 nt!KeBugCheckEx+0x19
f9e66ef4 f9b0cefb 0069b0d8 010000c6 00000000 i8042prt!I8xProcessCrashDump+0x235
f9e66f3c 804ebb04 81797d98 8169b020 00010009 i8042prt!I8042KeyboardInterruptService +0x21c
f9e66f3c fa12e34a 81797d98 8169b020 00010009 nt!KiInterruptDispatch+0x3d
WARNING: Stack unwind information not available. Following frames may be wrong.
ffdff980 8169b288 f9e67000 0000210f 00000004 myfault+0x34a
8054ace4 ffdff980 804ebf58 00000000 0000319c 0x8169b288
8054ace4 ffdff980 804ebf58 00000000 0000319c 0xffdff980
8169ae9c 8054ace4 f9b12b0f 8169ac88 00000000 0xffdff980
...
```

栈痕迹的最上面几行引用了你敲入i8042端口驱动程序的崩溃键序列时所执行的例程。这里出现了Myfault驱动程序，表示它可能要为这次系统挂起负责。

另一个可能有指示作用的命令是*!locks*，它转储出所有执行体资源锁的状态。在默认情况下，该命令只列出那些处于**竞争**条件下的资源。这里竞争的资源是指，它们已被某个线程所持有，而且至少有一个线程正在等着获得它们。用*!thread*命令来检查这些资源的所有者的线程栈，以便看清楚这些线程可能在哪个驱动程序中执行。

当没有崩溃转储时

在这一节中，我们将针对那些由于某种原因而未能记录一个崩溃转储的系统，讨论如何进行诊断。未能记录一个崩溃转储的可能的原因是：引导卷上的页面文件太小，以至于未能容得下此崩溃转储，或者，在重新引导以后，没有足够的空闲磁盘空间来提取出此崩溃转储。这两种情形很容易解决，或者增大页面文件的大小，或者配置系统的崩溃转储设置，让它将此转储保存到有足够磁盘空间的卷上。

未能记录崩溃转储的第三个原因是，在崩溃时刻，用于写崩溃转储的内核代码和数据结构已经被破坏了。

www.TopSage.com

正如前面所描述的那样，当系统引导的时候，此数据是计算了校验和的，如果在崩溃时刻所做的校验和不匹配，则系统根本不考虑保存崩溃转储（以避免可能会破坏磁盘上的数据）。因此，在这种情况下，你需要在系统崩溃的时候，捕捉住系统，然后试着确定崩溃的原因。

最后一个可能的原因是，系统磁盘的磁盘子系统不能够处理磁盘写请求（可能正是这个条件本身触发了系统失败）。这样的条件可能是磁盘控制器中的一个硬件失败，或者也可能是硬盘附近的连接线缆问题。

一种简单的方案是，在“Startup And Recovery”设置中关闭“Automatically Restart”选项，这样，如果系统崩溃了，你可以检查控制台上的蓝屏。然而，只有对最简单的崩溃，才能够仅仅根据蓝屏上的文本信息来加以解决。

为了执行更为深入的分析，你需要使用内核调试器，以便在崩溃的时候查看系统的情况。你只需按照上一小节所讲述的那样在调试模式下引导系统，就可以做到这一点。当一个在调试模式下引导起来的系统崩溃时，它不再绘制蓝屏和试图记录崩溃转储，相反地，它会一直等待，直至有一个主内核调试器连接过来。这样，你可以看到崩溃的原因，可能还可以利用前面讲述的内核调试器命令来执行一些基本的分析。正如在上一小节中所提到的那样，你可以在调试器中使用.dump命令来保存该崩溃系统的内存空间的一份拷贝，以便于将来调试。因此，你可以重新引导此崩溃的系统，并且以离线方式来调试该问题。



实验：蓝屏屏幕保护程序

有一种很巧妙的方法可以提醒你自己，蓝屏应该长什么样，或者也可以用来捉弄你的办公室同事或朋友，那就是，运行Sysinternals的蓝屏屏幕保护程序（Sysinternals Blue Screen screen saver，你可以从www.sysinternals.com下载）。此屏幕保护程序模拟真实的蓝屏，它显示了当前你正在运行的Windows的版本，并且利用实际的系统信息（比如已加载的驱动程序的列表）来生成所有的蓝屏文本。它也会模仿一次自动的重新引导过程，直到出现Windows的启动屏幕（splash screen）。注意，在其他的屏幕保护程序中，一旦鼠标移动就会退出屏幕保护，而与此不同的是，蓝屏屏幕保护程序要求一次键击才能退出。

利用Sysinternals的Psexec工具的以下语法，你甚至可以在另一个系统上运行屏幕保护程序：

```
psexec \\computername -i -d "c:\sysinternals bluescreen.scr" -s
```

该命令要求你在远程系统上具有管理特权（你可以使用Psexec的-u和-p开关来指定其他的凭证）。 不过，你一定要确保你的同事懂得幽默！

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

术语表

access-control list (ACL) The part of a security descriptor that enumerates who has what access to an object. The owner of an object can change the object's ACL to allow or disallow others access to the object. An ACL is made up of an ACL header and zero or more access-control entry (ACE) structures. An ACL with zero ACEs is called a null ACL and indicates that no user has access to the object.

access token A data structure that contains the security identification of a process or a thread, which includes its security ID (SID), the list of groups that the user is a member of, and the list of privileges that are enabled and disabled. Each process has a primary access token that it inherits by default from its creating process.

add-device routine A routine implemented by drivers that supports Plug and Play. The Plug and Play manager sends a driver notification via this routine whenever a device for which the driver is responsible is detected. In this routine, a driver typically allocates a device object to represent the device.

Address Windowing Extensions (AWE) A mechanism in Windows that allows a 32-bit application to allocate up to 128 GB of physical memory and then map views, or windows, into its 2-GB virtual address space. Using AWE puts the burden of managing mappings of virtual-to-physical memory on the programmer but solves the immediate 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.

affinity mask A bitmask that specifies the processors on which the thread is allowed to run. The initial thread affinity mask is inherited from the process affinity mask.

aging A process performed on a page that increments a count indicating that the page hasn't been referenced since the last working set trim scan. On a single-processor system, the working set manager tries to remove pages that haven't been accessed recently. It does this by first clearing the accessed bit in the hardware page table entry (PTE) and then later checking the bit to see whether the page has been accessed. If the bit remains clear, the page wasn't accessed between scans and is aged. Later, the age of pages is used to locate candidate pages to remove from the working set.

alertable wait state A thread state that the thread enters either by waiting on an object handle and specifying that its wait is alertable (with the Windows WaitForMultipleObjectsEx function) or by testing directly whether it has a pending APC (using SleepEx). In both cases, if a user-mode APC is pending, the kernel interrupts (alerts) the thread, transfers control to the APC routine, and resumes the thread's execution when the APC routine completes. User-mode APCs are delivered to a thread only when it's in an alertable wait state.

allocation granularity The granularity with which virtual memory is allocated. Windows aligns each region of reserved process address space to begin on an integral boundary defined by the system allocation granularity value, which can be retrieved from the Windows *GetSystemInfo* function. Currently, this value is 64 KB. This size was chosen so that if support were added for future processors with large page sizes (for example, up to 64 KB), the risk of requiring changes to applications that made assumptions about allocation alignment would be reduced. (Windows kernel-mode code isn't subject to the same restrictions; it can reserve memory on a single-page granularity.)

alternative hive A hive that acts as a backup to the crucial SYSTEM hive. The alternate hive is stored in \Winnt\System32\Config as System.alt. Whenever a hive sync flushes dirty sectors to the SYSTEM hive, the hive sync also updates the System.alt hive. If the configuration manager detects that the

www.TopSage.com

SYSTEM hive is corrupt when the system boots, the configuration manager attempts to load the alternate hive. If that hive is usable, it then uses that alternate to update the original SYSTEM hive.

APC queue A queue in which asynchronous procedure calls (APCs) waiting to execute reside. The APC queues (one for user mode and one for kernel mode) are thread-specific—each thread has its own APC queues (unlike the DPC queue, which is processor-wide).

asymmetric multiprocessing (ASMP) A system of processing on a multiprocessor system that typically selects one processor to execute operating system code while other processors run only user code.

asynchronous I/O An I/O model that allows an application to issue an I/O request and then continue executing while the request is completed. This type of I/O can improve an application's throughput because it allows the application to continue with other work while an I/O operation is in progress.

asynchronous procedure call (APC) A function that provides a way for user programs and system code to execute code in the context of a particular user thread (and hence a particular process address space). An APC can be either kernel mode or user mode. (Kernel-mode APCs don't require "permission" from a target thread to run in that thread's context, as user-mode APCs do.)

asynchronous read-ahead with history A method in which the cache manager maintains a history of the last two read requests in the private cache map for the file handle being accessed.

atomic transaction A technique for handling modifications to a database so that system failures don't affect the correctness or integrity of the database. The basic tenet of atomic transactions is that some database operations, called transactions, are all-or-nothing propositions. The separate disk updates that make up the transaction must be executed atomically; that is, once the transaction begins to execute, all its disk updates must be completed. If a system failure interrupts the transaction, the part that has been completed must be undone, or rolled back. The rollback operation returns the database to a previously known and consistent state, as if the transaction had never occurred. *See also* transaction.

attribute list A special kind of file attribute in an NTFS file header that contains additional attributes. The attribute list is created if a particular file has too many attributes to fit in the master file table (MFT) record. The attribute list attribute contains the name and type code of each of the file's attributes and the file reference of the MFT record where the attribute is located.

authentication packages Dynamic-link libraries (DLLs) that run in the context of the LSASS process and that implement Windows authentication policy. An authentication DLL is responsible for checking whether a given username and password match, and if so, returning to LSASS information detailing the user's security identity. Windows authentication packages include Kerberos and MSV1_0.

automatic working set trimming A technique the memory manager uses when physical memory runs low to increase the amount of free memory available in the system.

bad-cluster file A system file (filename $BadClus) that records any bad spots on the disk volume.

balance set manager A system thread that wakes up once per second to check and possibly initiate various scheduling and memory management–related events.

basic disk A disk that relies on the MS-DOS-style partitioning scheme. *See also* dynamic disk.

bitmap file A system file (filename $Bitmap) in which NTFS records the allocation state of the volume. The data attribute for the bitmap file contains a bitmap, each of whose bits represents a cluster on the volume, identifying whether the cluster is free or has been allocated to a file.

www.TopSage.com

boot code Instructions executed when a system is booted.

boot device drivers Device drivers necessary to boot the system.

boot file A system file (filename $Boot) that stores the Windows bootstrap code.

boot partition The partition that contains core operating system files. The boot partition is identified by the system at startup. The code in a master boot record (MBR) scans the primary partition table until it locates a partition containing a flag that signals the partition is bootable. When the MBR finds at least one such flag, it reads the first sector from the flagged partition into memory and transfers control to code within the partition.

boot sector The first sector of the partition marked as active and from which the MBR boots. The boot sector contains information identifying the partition's file system format and structure.

boot volume The volume that contains the Windows operating system and its support files. The boot volume can be, but does not have to be, the same as the system volume.

bus driver Driver that 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.

cache manager The component of the Windows executive that provides systemwide caching services for NTFS and other file system drivers, including network file system drivers (servers and redirectors).

careful write A technique for constructing a file system's I/O and caching support. *See also* write-through.

change journal An internal file where an NTFS file system can record information that allows applications to efficiently monitor file and directory changes. A change journal is usually large enough to virtually guarantee that applications get a chance to process changes without missing any.

checked build A special debug version of Windows that is available only as part of the MSDN Professional (or Universal) subscription. The checked build is created by compiling the Windows sources with the compile-time flag DEBUG defined.

checkpoint record A record that helps NTFS determine what processing would be needed to recover a volume if a crash were to occur immediately. This record also includes redo and undo information.

class driver A type of kernel-mode device driver that implements the I/O processing for a particular class of devices, such as disk, tape, or CD-ROM.

clock algorithm A working set page replacement policy implemented on uniprocessor systems, similar to a least recently used policy (as implemented in most versions of UNIX).

clock interrupt handler A system routine that updates the system time and then decrements a counter that tracks how long the current thread has run.

cluster factor The cluster size on a volume, which is established when a user formats the volume with either the format command or the Disk Management Microsoft Management Console (MMC) snap-in.

cluster remapping A process in which NTFS dynamically retrieves good data from a cluster with a bad sector, allocates a new cluster, and copies the data to the new cluster.

clustering A method by which the memory manager resolves a page fault by reading into memory several pages near the page explicitly accessed.

clusters Same-size allocation units into which a volume is divided. Each cluster must be uniquely numbered using 16 bits.

collided page fault A fault that occurs when another thread or process faults a page that is currently being in-paged.

commitment The process by which the memory manager keeps track of private committed memory usage on a global basis.

www.TopSage.com

common model A set of classes in the Common Information Model (CIM) that represent objects that are specific to management areas of a system but independent of a particular implementation. These classes are considered an extension of the CIM core model. *See also* core model.

complete memory dump A memory dump that contains all of physical memory at the time of the crash. This type of dump requires that a page file be at least the size of physical memory. Because it can require an inordinately large page file on large memory systems, this type of dump file is the least common. Windows NT 4 supported only this type of crash dump file.

completion port A mechanism to deliver I/O completion notification to threads. Once a file is associated with a completion port, any asynchronous I/O operations that complete on the file result in a completion packet being queued to the completion port. A thread can wait for any outstanding I/Os to complete on multiple files simply by waiting for a completion packet to be queued to the completion port. With completion ports, concurrency, or the number of threads that an application has actively servicing client requests, is controlled with the aid of the system.

configuration manager A major component of the executive that's responsible for implementing and managing the system registry.

container object A namespace object that can hold other objects, including other container objects. Examples of containers are directories in the file system namespace and keys in the registry namespace.

context switch The procedure of saving the volatile machine state associated with a running thread, loading another thread's volatile state, and starting the new thread's execution.

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

core model A set of classes in the Common Information Model (CIM) provided as part of the WBEM standard. These classes are CIM's basic language and represent objects that apply to all areas of management. *See also* common model.

crash dump A record of system memory at the time of a crash that can help you figure out which component caused the crash.

critical sections An intra-process mutual exclusion synchronization primitive.

deadlock detection A Driver Verifier option in Windows XP and Windows Server 2003 that monitors the use of spin locks, fast mutexes, and mutexes, looking for patterns that could result in deadlock.

deferred procedure call (DPC) A routine that performs most of the work involved in handling a device interrupt after the interrupt service routine (ISR) executes. The DPC routine executes at an interrupt request level (IRQL) that is lower than that of the ISR to avoid blocking other interrupts unnecessarily. A DPC routine initiates I/O completion and starts the next queued I/O operation on a device.

deferred procedure call (DPC) object A kernel control object that describes a request to defer interrupt processing to DPC/dispatch level. (*See* interrupt request levels [irqls].) This object isn't visible to user-mode programs but is visible to device drivers and other system code. The most important piece of information the DPC object contains is the address of the system function that the kernel will call when it processes the DPC interrupt.

Deferred ready A state used for threads that have been selected to run on a specific processor but have not yet been scheduled. This new state in Windows Server 2003 exists so that the kernel can minimize the amount of time the systemwide lock on the scheduling database is held.

www.TopSage.com

demand-paging policies A fetch policy that loads a page into physical memory only when a page fault occurs. In a demand-paging system, a process incurs many page faults when its threads first begin executing because the threads reference the initial set of pages they need to get going. Once this set of pages is loaded into memory, the paging activity of the process decreases.

desired access rights The accesses desired by a thread opening an object.

device drivers Loadable kernel-mode modules (typically ending in .sys) that interface between the I/O system and the relevant hardware. Device drivers on Windows don't manipulate hardware devices directly, but rather they call parts of the hardware application layer (HAL) to interface with the hardware.

device ID A device identifier reported to the Plug and Play manager. The identifiers are bus-specific; for a USB bus, an identifier consists of a vendor ID (VID) for the hardware vendor that made the device and a product ID (PID) that the vendor assigned to the device.

device instance ID (DIID) An identifier consisting of a device ID and an instance ID that the Plug and Play manager uses to locate the device's key in the enumeration branch of the registry (HKLM\SYSTEM\CurrentControlSet\Enum).

device object A data structure that represents a physical, logical, or virtual device on the system and describes its characteristics, such as the alignment it requires for buffers and the location of its device queue to hold incoming I/O request packets.

device-specific module (DSM) Third-party drivers used to manage details of the path management, such as load balancing policies that choose which path to route requests and error detection mechanisms to inform Windows when a path fails.

device tree An internal tree the Plug and Play manager creates that represents the relationships between devices. Nodes in the tree are called devnodes. *See also* devnode.

devnode A node in a device tree. A devnode contains information about the device objects that represent the device as well as other Plug and Play–related information the Plug and Play manager stores. *See also* device tree.

direct memory access (DMA) A third interface provided by the cache manager to cached data. The DMA functions are used to read from or write to cache pages without intervening buffers, such as when a network file system is doing a transfer over the network.

dirty page threshold The number of pages that the system cache keeps in memory before waking up the cache manager's lazy writer system thread to write out pages back to the disk. This value is computed at system initialization time and depends on physical memory size and the value of the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargeSystemCache.

discretionary access control Allows the owner of a resource to determine who can access the resource and what they can do with it. The owner grants rights that permit various kinds of access to a user or to a group.

disk group Dynamic disks that share a common database. VERITAS's commercial volume-management software for Windows includes disk groups, but the Windows Logical Disk Manager (LDM) implementation includes only one disk group.

dispatch code Instructions of assembly language code stored in an interrupt object when it is initialized. When an interrupt occurs, this code is executed.

dispatch routines The main functions that a device driver provides. Some examples of dispatch routines are open, close, read, and write, and any other capabilities the device, file system, or network supports. When called on to perform an I/O operation, the I/O manager generates an IRP and calls a driver through one of the driver's dispatch routines.

dispatcher A set of routines in the kernel that implement Windows scheduling. Windows doesn't have a single "scheduler" module or

www.TopSage.com

routine—the code is spread throughout the kernel in which scheduling-related events occur.

dispatcher database A set of data structures the kernel maintains to make thread-scheduling decisions. The dispatcher database keeps track of which threads are waiting to execute and which processors are executing which threads. *See also* dispatcher ready queue.

dispatcher header A data structure that contains the object type, the signaled state, and a list of the threads waiting on that object.

dispatcher objects A set of kernel objects that incorporate synchronization capabilities and 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.

dispatcher ready queue The most important structure in the dispatcher database (located at KiDispatcherReadyListHead). The dispatcher ready queue is really a series of queues, one queue for each scheduling priority. The queues contain threads that are in the ready state, waiting to be scheduled for execution.

display driver Driver that translates device-independent graphics requests into device-specific requests. The device-specific requests are then paired with a kernel-mode video miniport driver to complete video display support. A display driver is responsible for implementing drawing operations, either by writing directly to the frame buffer or by communicating with the graphics accelerator chip on the controller.

driver object Data structure that represents an individual driver in the system and records for the I/O manager the address of each of the driver's dispatch routines (entry points).

driver support routines Routines that device drivers call to accomplish their I/O requests.

dynamic disk A disk that supports multipartition volumes, providing a more flexible partitioning scheme than that of a basic disk. *See also* basic disk.

***dynamic-link library* (DLL)** A set of callable subroutines linked as a binary image that can be dynamically loaded by applications that use them.

environment subsystems User processes and DLLs that expose the native operating system services to user applications, thus providing an operating system environment, or personality. Windows 2000 ships with two environment subsystems: Windows and POSIX (Windows NT 4.0 had one for OS/2 1.2). Windows XP and later only ship with Windows, but an enhanced POSIX subsystem is included with the free Services for Unix product from Microsoft.

event An object with a persistent state (signaled or not signaled) that can be used for synchronization; also, a system occurrence that triggers an action.

exception A synchronous condition that results from the execution of a particular instruction. Running a single program with the same data under the same conditions can reproduce exceptions.

exception dispatcher A kernel module that services all exceptions, except those simple enough to be resolved by the trap handler. The exception dispatcher's job is to find an exception handler that can "dispose of" the exception.

executive The upper layer of Ntoskrnl.exe. (The kernel is the lower layer.) The executive contains the base operating system services, such as the process and thread manager, the virtual memory manager, the memory manager, the security reference monitor, the I/O system, and the cache manager. *See also* kernel.

executive objects Objects implemented by various components of the executive (such as the process manager, memory manager, I/O subsystem, and so on). The executive objects and object services are primitives that the environment subsystems use to construct their own versions of objects and other resources. Because executive objects are

www.TopSage.com

typically created either by an environment subsystem on behalf of a user application or by various components of the operating system as part of their normal operation, many of them contain (encapsulate) one or more kernel objects. *See also* kernel objects.

executive resources Resources that provide both exclusive access (such as a mutex) as well as shared read access (multiple readers sharing read-only access to a structure). Because executive resources are available only to kernel-mode code, they aren't accessible from the Windows API.

executive support routines Functions in Ntoskrnl.exe that provide services to device drivers.

extended partition A special partition type that contains a master boot record (MBR) with its own partition table. By using extended partitions, Microsoft's operating systems overcome the apparent limit of four partitions per disk. In general, the recursion that extended partitions permit can continue indefinitely, which means that no upper limit exists to the number of possible partitions on a disk. *See also* partition.

fast I/O A means of reading or writing a cached file without going through the work of generating an I/O request packet (IRP).

fast LPC A special interprocess communication facility used to send messages between threads.

file mapping objects Windows APO underlying primitives in the memory manager that are used to implement shared memory (called section objects internally). *See also* section object.

file reference A 64-bit value that identifies a file on an NTFS volume. The file reference consists of a file number and a sequence number. The file number corresponds to the position of the file's file record in the master file table minus 1 (or to the position of the base file record minus 1 if the file has more than one file record).

file system driver (FSD) A type of kernel-mode device driver that accepts I/O requests to files and satisfies the requests by issuing its own, more explicit, requests to physical device drivers. *See also* local file system driver (FSD), network file system driver (FSD).

file system filter driver A type of kernel-mode device driver that intercepts I/O requests, performs additional processing, and passes them on to lower-level drivers.

file system format Defines the way that file data is stored on storage media and impacts a file system's features. For example, a format that doesn't allow user permissions to be associated with files and directories can't support security. A file system format can also impose limits on the sizes of files and storage devices that the file system supports.

filter device object (FiDO) A device object that can be part of a devnode. One or more optional FiDOs can layer either between the physical device object (PDO) and the functional device object (FDO) or above the FDO. *See also* devnode, functional device object (FDO), physical device object (PDO).

filter driver *See* file system filter driver.

frame-based exception handlers The mechanism that permits each stack frame in a call stack to have its own exception handler declared.

foreground application The process that owns the thread that owns the window that's in focus.

free build The version of the Windows system that can be purchased as a retail product. It is built with full compiler optimizations turned on and has internal symbol table information stripped out from the images. *See also* checked build.

function driver The main device driver that 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 the driver that knows the most about a particular device and is usually the only driver that accesses device-specific registers.

www.TopSage.com

functional device object (FDO) A device object that is a required part of a devnode. The function driver that the Plug and Play manager loads to manage a detected device creates the FDO. An FDO represents the logical interface to a device. *See also* devnode, filter device object (FiDO), physical device object (PDO).

granted access rights The accesses granted to a thread by the security reference monitor as the result of a successful object open.

Graphical Identification and Authentication (GINA) A user-mode DLL that runs in the Winlogon process and that Winlogon uses to obtain a user's name and password or smart card PIN. The standard GINA is \Winnt\System32\Msgina.dll.

handle An object identifier. A process receives a handle to an object when it creates or opens an object by name. Referring to an object by its handle is faster than using its name because the object manager can skip the name lookup and find the object directly.

handle table A table that contains pointers to all the objects that the process has opened a handle to. Handle tables are implemented as a three-level scheme, similar to the way that the x86 memory management unit implements virtual-to-physical address translation.

hardware abstraction layer (HAL) A loadable kernel-mode module (Hal.dll) that provides the low-level interface to the hardware platform on which Windows is running. The HAL hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms–any functions that are architecture-specific and machine-dependent.

hardware device drivers Device drivers that manipulate hardware 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.

hash A statistically unique value that is generated from a block of data (for example, a file) using cryptographic algorithms. Because different data results in different hashes, hashes can be used to detect changes to data from corruption or tampering. The Windows driver signing facility uses hashes.

heap A region of one or more pages that can be subdivided and allocated in smaller chunks by a set of functions provided by the heap manager.

heap manager A set of functions that allocate and deallocate variable amounts of memory (not on a page-size granularity). The heap manager functions exist in two places: Ntdll.dll and Ntoskrnl.exe. The subsystem APIs (such as the Windows heap APIs) use the copy in Ntdll, and various executive components and device drivers use the copy in Ntoskrnl.

hive One of a number of files stored on disk that contain registry information. Each hive contains a registry tree, which has a key that serves as the root or starting point of the tree.

hyperspace A special region used to map the process working set list and to temporarily map other physical pages for such operations as zeroing a page on the free list (when the zero list is empty and a zero page is needed), invalidating page table entries in other page tables (such as when a page is removed from the standby list), and on process creation setting up a new process's address space.

I/O completion routine A routine implemented by a layered driver that will notify the driver when a lower-level driver finishes processing an I/O request packet (IRP). For example, the I/O manager calls a file system driver's I/O completion routine after a device driver finishes transferring data to or from a file. The completion routine notifies the file system driver about the operation's success, failure, or cancellation, and it allows the file system driver to perform cleanup operations.

I/O request packet (IRP) A data structure that controls how the I/O operation is processed at each stage. Most I/O requests are represented by an IRP, which travels from one I/O system component to another.

www.TopSage.com

I/O subsystem API The internal executive system services (such as NtReadFile and NtWriteFile) that subsystem DLLs call to implement a subsystem's documented I/O functions.

I/O system The Windows executive component that accepts I/O requests (from both user-mode and kernel-mode callers) and delivers them, in a different form, to I/O devices.

ideal processor The preferred processor that a particular thread should run on.

idle summary A bitmask (KiIdleSummary) in which each set bit represents an idle processor.

impersonation A capability that allows threads to have a different access token than that of the process.

initialization routine A driver routine that the I/O manager executes when it loads the driver into the operating system. The initialization routine creates system objects that the I/O manager uses to recognize and access the driver.

in-paging I/O A condition that occurs when a read operation must be issued to a file (paging or mapped) to satisfy a page fault. The in-page I/O operation is synchronous—the thread waits on an event until the I/O completes—and isn't interruptible by asynchronous procedure call (APC) delivery.

instancing The term for making separate copies of the same parts of a namespace. Instancing \DosDevices makes it possible for each user to have different drive letters and Windows objects such as serial ports.

intelligent file read-ahead A technique that predicts what data the calling thread is likely to read next based on the data it's currently reading.

inter-processor interrupt (IPI) An interrupt the kernel issues to request that another processor perform an action, such as dispatching a particular thread for execution or updating its translation look-aside buffer cache.

interrupt An asynchronous event (one that can occur at any time) that is unrelated to what the processor is executing. Interrupts are generated primarily by I/O devices, processor clocks, or timers, and they can be enabled or disabled.

interrupt dispatch table (IDT) A data structure that Windows uses to locate the routine that will handle a particular interrupt. The interrupt request level (IRQL) of the interrupting source serves as a table index, and table entries point to the interrupt-handling routines.

interrupt dispatcher A submodule of the kernel's trap handler that responds to interrupts.

interrupt object A kernel control object that allows device drivers to register interrupt service routines (ISRs) for their devices. An interrupt object contains all the information the kernel needs to associate a device ISR with a particular level of interrupt, including the address of the ISR, the interrupt request level (IRQL) at which the device interrupts, and the entry in the kernel's interrupt dispatch table with which the ISR should be associated.

interrupt request (IRQ) A value identifying an interrupt. On x86 systems, external I/O interrupts come into one of the lines on an interrupt controller. The controller in turn interrupts the processor on a single line. Once the processor is interrupted, it queries the controller to get the interrupt request (IRQ). The interrupt controller translates the IRQ to an interrupt number, uses this number as an index into the interrupt dispatch table (IDT), and transfers control to the appropriate interrupt dispatch routine. At system boot time, Windows fills in the IDT with pointers to the kernel routines that handle each interrupt and exception.

interrupt request levels (IRQLs) An interrupt priority scheme imposed by Windows. The kernel represents IRQLs internally as a number from 0 through 31 (0 to 15 on 64-bit Windows), with higher numbers representing higher-priority interrupts. Although the

www.TopSage.com

kernel defines the standard set of IRQLs for software interrupts, the HAL maps hardware-interrupt numbers to the IRQLs.

interrupt service routine (ISR) A device driver routine that the kernel's interrupt dispatcher transfers control to when a device issues an interrupt. In the Windows I/O model, ISRs run at a high device interrupt request level (IRQL), so they perform as little work as possible to avoid blocking lower-level interrupts unnecessarily. An ISR queues a deferred procedure call (DPC), which runs at a lower IRQL, to execute the remainder of interrupt processing. Only drivers for interrupt-driven devices have ISRs; a file system, for example, doesn't have one.

job object A nameable, securable, shareable object in Windows that controls certain attributes of processes associated with the job. A job object's basic function is to allow groups of processes to be managed and manipulated as a unit. The job object also records basic accounting information for all processes associated with the job and for all processes that were associated with the job but have since terminated.

journaling A logging technique originally developed for transaction processing that a recoverable file system such as NTFS uses to ensure volume consistency.

kernel The lowest layer in Ntoskrnl.exe. The kernel, a component of the executive, determines how the operating system uses the processor or processors and ensures that they are used prudently. The kernel provides thread scheduling and dispatching, trap handling and exception dispatching, interrupt handling and dispatching, and multiprocessor synchronization. *See also* executive.

kernel debugger A tool used to debug drivers, troubleshoot hung systems, and examine crash dumps. It is also a useful tool for investigating Windows internals because it can display internal Windows system information not visible through any standard utility. (The LiveKd tool on the companion CD allows the use of the standard kernel debuggers on a live system.)

kernel handle table A table (referenced internally with the name ObpKernelHandleTable) whose handles are accessible only from kernel mode and in any process context. A kernel-mode function can reference the handles in this table in any process context with no performance impact.

kernel memory dump A type of memory dump (the default on Windows Server systems) that contains only the kernel-mode read/write pages present in physical memory at the time of the crash. A kernel memory dump doesn't contain pages belonging to user processes. Because only kernel-mode code can directly cause Windows to crash, however, it's unlikely that user process pages are necessary to debug a crash. There is no way to predict the size of a kernel memory dump because its size depends on the amount of kernel-mode memory allocated by the operating system and drivers present on the machine.

kernel mode A privileged mode of code execution in a processor in which all memory is totally accessible and all CPU instructions can be issued. Operating system code (such as system services and device drivers) runs in kernel mode. *See also* user mode.

kernel objects A primitive set of objects implemented by the Windows kernel. These objects aren't visible to user-mode code but are created and used only within the executive. Kernel objects provide fundamental capabilities, such as synchronization, on which executive objects are built. *See also* executive objects.

kernel streaming filter drivers Kernel-mode drivers chained together to perform signal processing on data streams, such as recording or displaying audio and video.

kernel-mode device driver The only type of driver that can directly control and access hardware devices.

kernel-mode graphics driver A Windows subsystem display or print device driver that translates device-independent graphics (GDI) requests into device-specific requests.

www.TopSage.com

key A mechanism to refer to data in the registry. Although keys appear in the object manager namespace, the registry manages them in a way similar to how it manages file objects. Zero or more key values are associated with a key object; key values contain data about the key.

key control block A structure that stores the full pathname of a registry key, includes the cell index of the key node that the control block refers to, and contains a flag that notes whether the configuration manager needs to delete the key cell that the key control block refers to when the last handle for the key closes. In Windows, all key control blocks are in an alphabetized binary tree to enable quick searches for existing key control blocks by name. A key object points to its corresponding key control block, so if two applications open the same registry key, each will receive a key object and both key objects will point to a common key control block.

key object An object type the configuration manager defines to integrate the registry's namespace with the kernel's general namespace.

keyed event A synchronization object new to Windows XP.

last known good control set A copy of the critical boot-time information under the registry key HKLM\SYSTEM\CurrentControlSet, made when a user successfully logs on. The last known good control set can be selected at boot time in case configuration changes made to the registry result in the system not being able to boot successfully.

lazy IRQL A performance optimization that avoids Programmable Interrupt Controller (PIC) accesses. When the interrupt request level (IRQL) is raised, the HAL notes the new IRQL internally instead of changing the interrupt mask. If a lower-priority interrupt subsequently occurs, the HAL sets the interrupt mask to the settings appropriate for the first interrupt and postpones the lower-priority interrupt until the IRQL is lowered. Thus, if no lower-priority interrupts occur while the IRQL is raised, the HAL doesn't need to modify the PIC.

lazy writer A set of system threads that call the memory manager to flush cache contents to disk as a background activity (asynchronous disk writing). The cache manager optimizes disk I/O by using its lazy writer.

legacy drivers Device drivers written for Microsoft Windows NT but that run unchanged on Windows 2000 and later. They are differentiated from other Windows drivers in that they don't support power management or work with the Windows Plug and Play manager.

local file system driver (FSD) A driver that manages volumes directly connected to the computer. *See also* file system driver.

local kernel debugging The ability to connect the kernel debugger to the local running system to view internal state (as opposed to connecting to a target system booted in debugging mode using a null modem or 1394 cable).

local procedure call (LPC) An interprocess communication facility for high-speed message passing (not available through the Windows API but rather through an internal mechanism available only to Windows operating system components). LPCs are typically used between a server process and one or more client processes of that server. An LPC connection can be established between two user-mode processes or between a kernel-mode component and a user-mode process.

local security authority (LSA) server A user-mode process running the image \Winnt\System32\LSASS.exe that is responsible for the local system security policy (such as which users are allowed to log on to the machine, password policies, privileges granted to users and groups, and the system security auditing settings), user authentication, and sending security audit messages to the Event Log. The LSA service (Lsasrv - \Winnt\System32\ Lsasrv.dll), a library that LSASS loads, implements most of this functionality.

www.TopSage.com

local security authority (LSA) server policy database A database (stored in the registry under HKLM\SECURITY) that contains the system security policy settings. This database includes such information as what domains are trusted to authenticate logon attempts, who has permission to access the system and how (interactive, network, and service logons), who is assigned which privileges, and what kind of security auditing is to be performed.

Local Security Authority Subsystem (LSASS) The system user-mode process responsible for authentication of accounts accessing a computer.

Local System account A predefined local account that is used to start a service and provide the security context for that service. The name of the account is NT AUTHORITY\System. This account does not have a password, and any password information that you supply is ignored. The Local System account has full access to the system, including the directory service on domain controllers. Because the Local System account acts as a computer on the network, it has access to network resources.

log file A metadata file (filename $LogFile) NTFS uses to record all operations that affect the NTFS volume structure, including file creation or any commands, such as Copy, that alter the directory structure. The log file is used to recover an NTFS volume after a system failure.

log hive A registry hive the configuration manager uses to make sure that a nonvolatile registry hive (one with an on-disk file) is always in a recoverable state. Each nonvolatile hive has an associated log hive, which is a hidden file with the same base name as the hive and a .log extension. *See also* hive.

logging A transaction-processing technique NTFS uses to maintain file system integrity in case of system crashes or other failures. In NTFS logging, the suboperations of any transaction that alters important file system data structures are recorded in a log file before they are carried through on the disk so that if the system crashes, partially completed transactions can be redone or undone when the system comes back on line.

logical cluster numbers (LCNs) The numbers of all clusters from the beginning of the volume to the end with which NTFS refers to physical locations on a disk. To convert an LCN to a physical disk address, NTFS multiplies the LCN by the cluster factor to get the physical byte offset on the volume, as the disk driver interface requires.

logical prefetcher The kernel component that monitors boot and application startup file access and that preemptively reads that data to speed up subsequent booting and application startups.

logical sequence numbers (LSNs) The numbers that NTFS uses to identify records written to the log file.

logon process A user-mode process running Winlogon.exe that is responsible for capturing the username and password, sending them to the local security authority server for verification, and creating the initial process in the user's session.

look-aside list A fast memory allocation mechanism that contains only fixed-sized blocks. Look-aside lists can be either pageable or nonpageable, so they are allocated from paged or nonpaged pool.

LPC facility Local procedure call interprocess communication support in the kernel.

mapped file I/O The ability to view a file residing on disk as part of a process's virtual memory. A program can access the file as a large array without buffering *data or performing* disk I/O. The program accesses memory, and the memory manager uses its paging mechanism to load the correct page from the disk file. If the application writes to its virtual address space, the memory manager writes the changes back to the file as part of normal paging.

mask The process whereby interrupts wait for an executing thread to lower the IRQL before the interrupt is processed. Interrupts from a source with an IRQL above the

www.TopSage.com

current level interrupt the processor, whereas interrupts from sources with IRQLs equal to or below the current level are masked until an executing thread lowers the IRQL.

master file table (MFT) The heart of the NTFS volume structure. The MFT is implemented as an array of file records. The size of each file record is fixed at 1 KB, regardless of cluster size.

memory manager The Windows executive component that implements demand-paged virtual memory, giving each process the illusion that it has a large virtual address space (while mapping a subset of that address space to physical memory).

metadata Data that describes the files on a disk; also called volume structure data.

metadata files A set of files in each NTFS volume that contains the information used to implement the file system structure.

MFT mirror An NTFS metadata file (filename $MFTMirr) located in the middle of the disk called that contains a copy of the first few rows of the master file table.

miniport driver A type of kernel-mode device driver that maps a generic I/O request to a type of port into an adapter type, such as a specific SCSI adapter.

mirrored volume A volume on which the contents of a partition on one disk are duplicated in an equal-sized partition on another disk. Mirrored volumes are sometimes referred to as RAID level 1 (RAID-1).

mirror set A technique by which the contents of a partition on one disk are duplicated in an equal-size partition on another disk.

modified page writer A thread in the virtual memory manager that is responsible for limiting the size of the modified page list by writing pages to their backing store locations when the list becomes too big. The modified page writer consists of two system threads: one to write out modified pages (MiModifiedPageWriter) to the paging file and a second one to write modified pages to mapped files (MiMappedPageWriter).

mount A technique NTFS uses when it first accesses a volume; in this context, to mount means to prepare the volume for use. To mount the volume, NTFS looks in the boot file to find the physical disk address of the master file table.

mount points A mechanism that permits the linking of volumes through directories on NTFS volumes, which makes volumes accessible with no drive-letter assignment. Reparse points in NTFS make mount points possible.

MSDN Microsoft Developer Network, Microsoft's support program for developers. MSDN offers three CD-ROM subscription programs: MSDN Library, Professional, and Universal. For more information, see msdn.microsoft.com.

multipartition volumes Objects that represent sectors from multiple partitions and that file system drivers manage as a single unit. Multipartition volumes offer performance, reliability, and sizing features that simple volumes don't.

multipathing Where more than one set of hardware exists between the computer and a disk so that if a path fails, the system can still access the disk via an alternate path.

mutant Internal name for a mutex.

mutex A synchronization mechanism used to serialize access to a resource.

mutual exclusion A means whereby only one thread or processor is allowed access to a resource at any given time.

name retention The first phase of object retention, which the object manager implements. Name retention is controlled by the number of open handles to an object that exist. Every time a process opens a handle to an object, the object manager increments the open handle counter in the object's header. As processes finish using the object and close their handles to it, the object manager decrements the open handle counter. When the counter drops to 0, the object manager deletes the object's name from its global namespace. This deletion prevents new processes from opening a handle to the object.

www.TopSage.com

native application An application that uses only system service APIs provided by Ntdll and that isn't a client of the Windows subsystem. Smss (Session Manager) is an example of a native application.

network file system driver (FSD) A driver that allows users to access data volumes connected to remote computers. *See also* file system driver.

network logon service A user-mode service inside the Services.exe process that responds to network logon requests. Authentication is handled as local logons are, by sending them to the LSASS process for verification.

network redirectors and servers File system drivers that transmit remote I/O requests to a machine on the network and receive such requests, respectively.

nonpaged pool Memory pool that consists of ranges of system virtual addresses that are guaranteed to be resident in physical memory at all times and thus can be accessed from any address space without incurring paging I/O. Nonpaged pool is created at system initialization and is used by kernel-mode components to allocate system memory.

Ntdll.dll A special system-support library primarily for the use of subsystem DLLs that contains system service dispatch stubs to Windows executive system services and internal support functions used by subsystems, subsystem DLLs, and other native images.

Ntkrnlmp.exe The executive and kernel for multiprocessor systems.

Ntoskrnl.exe The executive and kernel for uniprocessor systems.

object In the Windows executive, a single, run-time instance of a statically defined object type.

object attribute A field of data in an object that partially defines the object's state.

object directory A container object for other objects. The object directory is used to implement the hierarchical namespace within which other object types are stored.

object handle An index into a process-specific handle table, pointed to by the executive process (EPROCESS) block.

object manager The Windows executive component responsible for creating, deleting, protecting, and tracking objects. The object manager centralizes resource control operations that would otherwise be scattered throughout the operating system.

object methods The means for manipulating objects, usually to 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.

object reuse protection A means of preventing users from seeing data that another user has deleted or from accessing memory that another user previously used and then released. Object reuse protection prevents potential security holes by initializing all objects, including files and memory, before they are allocated to a user.

object type A system-defined data type, including services.that operate on instances of the data type and a set of object attributes; sometimes called an *object class*.

page directory A page the memory manager creates to map the location of all page tables for that process. Each process has a single page directory.

page directory entries (PDEs) The page directory is composed of PDEs, each of which is 4 bytes long and describes the state and location of all the possible page tables for that process.

page fault A reference to an invalid page. The kernel trap handler dispatches this kind of fault to the memory manager fault handler (MmAccessFault) to resolve.

page file backed section A section object that is mapped to committed memory.

page file quota A limit on the number of committed pages a process can consume—not necessarily page file space.

page frame database A database that describes the state of each page in physical

www.TopSage.com

memory. Pages are in one of eight states: active (also called valid), transition, standby, modified, modified-no-write, free, zeroed, or bad.

page frame number (PFN) database Describes the state of each page in physical memory.

page table A page of mapping information (made up of an array of page table entries) the operating system constructs that describes the location of the virtual pages in a process address space. Because Windows provides a private address space for each process, each process has its own set of process page tables to map that private address space because the mappings will be different for each process. The page tables that describe system space are shared among all processes.

page table entry (PTE) An entry in a process's page table that contains the address to which the virtual address is mapped. The page can be in physical memory or it can be on disk.

paged pool A region of virtual memory in system space that can be paged in and out of the system process's working set. Paged pool is created at system initialization and is used by kernel-mode components to allocate system memory. Uniprocessor systems have two paged pools; multiprocessor systems have four. Having more than one paged pool reduces the frequency of system code blocking on simultaneous calls to pool routines.

paging The process of moving memory contents to disk, freeing physical memory so that it can be used for other processes or for the operating system itself. Because most systems have much less physical memory than the total virtual memory in use by the running processes (2 GB or 3 GB for each process), the memory manager transfers, or pages, some of the memory contents to disk.

partition A discrete area of the hard disk in Microsoft operating systems. File systems (such as FAT and NTFS) format each partition into a volume. A hard disk can contain up to four primary partitions. *See also* extended partition.

partition table A part of the master boot record (MBR) that consists of four entries that define the locations of as many as four primary partitions on a disk. The partition table also records a partition's type. Numerous predefined partition types exist, and a partition's type specifies which file system the partition includes.

Physical Address Extension (PAE) A memory-mapping mode included in all Intel x86 processors since the Pentium Pro. With the proper chipset, the PAE mode allows access to up to 64 GB of physical memory. When the x86 executes in PAE mode, the memory management unit (MMU) divides virtual addresses into four fields.

physical device object (PDO) A device object that's a required part of a devnode. The PDO represents the physical interface to a device. *See also* devnode.

Plug and Play (PnP) manager A major component of the executive that determines which drivers are required to support a particular device and loads those drivers. The PnP manager 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.

port driver A type of kernel-mode device driver that implements the processing of an I/O request specific to a type of I/O port, such as SCSI.

port object A single executive object a local procedure call (LPC) exports to maintain the state needed for communication.

power manager A major component of the executive that 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

www.TopSage.com

by individual devices are handled by device drivers but are coordinated by the power manager.

printer driver A driver that translates device-independent graphics requests to printer-specific commands. These commands are then typically forwarded to a kernel-mode port driver such as the parallel port driver (Parport.sys) or the USB printer port driver (Usbprint.sys).

private cache map A structure that contains the location of the last two reads so that the cache manager can perform intelligent read-ahead.

process The virtual address space and control information necessary for the execution of a set of thread objects.

process and thread manager The services in the executive that export the process and thread object. These services use primitive functions in the kernel to create processes and threads.

process ID A unique identifier for a process (internally called a client ID).

process working set The subset of a process's virtual address space that is resident and owned by the running process. *See also* system working set.

processor affinity The set of processors a thread is permitted to run on.

processor control register (PCR) A data structure that along with its extension, the processor control block (PRCB), contains information about the state of each processor in the system, such as the current interrupt request level (IRQL), a pointer to the hardware interrupt dispatch table (IDT), the currently running thread, and the next thread selected to run. The kernel and the HAL use this information to perform architecture-specific and machine-specific actions. Portions of the PCR and PRCB structures are defined publicly in the Windows Device Driver Kit (DDK) header file Ntddk.h.

program A static sequence of instructions.

protected-mode A state during the boot process in which no virtual-to-physical translation occurs but a full 32 bits of memory becomes accessible. After the system is in protected-mode, Ntldr can access all of physical memory.

protocol driver A driver that implements a networking protocol such as TCP/IP, NetBEUI, or IPX/SPX.

prototype page table entries (prototype PTEs) A software structure the memory manager relies on to map potentially shared pages when a page can be shared between two processes. An array of prototype PTEs is created when a section object is first created.

quality of service (Qos) A networking technology that ensures that critical network applications receive highest priority.

quantum The length of time a thread is allowed to run before Windows interrupts the thread to find out whether another thread at the same priority level is waiting to run or whether the thread's priority needs to be reduced.

quantum unit A value that represents how long a thread can run until its quantum expires. This value is an integer value, not a length of time.

queue A method for threads to enqueue and dequeue notifications of the completion of I/O operations (called an I/O completion port in the Windows API).

queued spinlock A special type of spinlock that is used only by the kernel and not exported for executive components or device drivers. A queued spinlock scales better on multiprocessor systems than a standard spinlock does. *See also* spinlock.

quota charges In the Windows object manager, the record of how much the object manager subtracts from a process's allotted paged and/or nonpaged pool quota when a thread in the process opens a handle to the object.

RAID-5 volume A fault tolerant variant of a regular stripe volume. Fault tolerance is achieved by reserving the equivalent of one disk for storing parity for each stripe. Also called stripe volume with parity.

www.TopSage.com

ready summary A bit mask (KiReadySummary) that Windows maintains to speed up the selection of which thread to run or preempt.

real-mode An operating mode in which no virtual-to-physical translation of memory addresses occurs, which means that programs that use the memory addresses interpret them as physical addresses and that only the first 1 MB of the computer's physical memory is accessible. Simple MS-DOS programs execute in a real-mode environment.

recoverability An advanced feature of NTFS that allows a system to recover from an unexpected system halt. If a system is halted unexpectedly, the metadata of a FAT volume can be left in an inconsistent state, leading to the corruption of large amounts of file and directory data. NTFS implements recoverability by logging changes to metadata in a transactional manner so that file system structures can be repaired to a consistent state with no loss of file or directory structure information. (File data can be lost, however.)

redo information Information included in the NTFS checkpoint record that explains how to reapply one suboperation of a fully logged (committed) transaction to the volume if a system failure occurs before the transaction is flushed from the cache.

reference count The object manager's record of how many object pointers it has dispensed to operating system processes. The object manager increments a reference count for an object each time it gives out a pointer to the object; when kernel-mode components finish using the pointer, they call the object manager to decrement the object's reference count.

reparse data User-defined data about the file or directory, such as its state or location, that can be read from the reparse point by the application that created the data, a file system filter driver, or the I/O manager.

reparse point An NTFS file or directory that has a block of data called reparse data associated with it.

Reparse tag The reparse tag allows the component responsible for interpreting the reparse point's reparse data to recognize the reparse point without having to check the reparse data.

resident attribute An attribute that is stored directly in the master file table. (If a file is small, all its attributes and their values [its data, for example] fit in the file record.)

resource arbitration A process by which the Plug and Play (PnP) manager optimally assigns hardware resources so that each device meets the requirements necessary for its operation. Because hardware devices can be added to the system after boot-time resource assignment, the PnP manager must also be able to reassign resources to accommodate the needs of dynamically added devices.

restricted token A token created from a primary or impersonation token using the CreateRestrictedToken function. The restricted token is a copy of the token it's derived from, with some possible modifications: Privileges can be removed from the token's privilege array. SIDs in the token can be marked as deny-only. SIDS in the token can be marked as restricted.

ring A privilege level defined in the x86 and x64 processor architectures to protect system code and data from being overwritten either inadvertently or maliciously by code of lesser privilege. Windows on the x86 and x64 platforms use privilege level 0 (or ring 0) for kernel mode and privilege level 3 (or ring 3) for user mode.

safe mode A boot configuration that consists of the minimal set of device drivers and services. By relying on only the drivers and services that are necessary for booting, Windows avoids loading third-party and other nonessential drivers that might crash.

SAM database A database (stored in the registry under HKLM\SAM) that contains the defined users and groups, along with their passwords and other attributes.

scatter/gather I/O A kind of high-performance I/O Windows supports, available via the Windows ReadFileScatter and WriteFileGather functions. These functions allow an application to issue a single read or write

www.TopSage.com

from more than one buffer in virtual memory to a contiguous area of a file on disk. To use scatter/gather I/O, the file must be opened for noncached I/O, the user buffers being used have to be page-aligned, and the I/Os must be asynchronous (overlapped).

section object An object that represents a block of memory that two or more processes can share. A section object can be mapped to the paging file or to another file on disk. The executive uses section objects to load executable images into memory, and the cache manager uses them to access data in a cached file. In the Windows subsystem, a section object is called a file-mapping object.

section object pointers Structure for each open file (represented by a file object) that is the key to maintaining data consistency for all types of file access as well as to providing caching for files. The section object pointers structure points to one or two control areas. One control area is used to map the file when accessed as a data file, and one is used to map the file when it is run as an executable image.

sector A hardware-addressable portion of a physical disk. A hard disk sector on an IBM-compatible PC is typically 512 bytes. Utilities that prepare hard disks for the definition of logical drives, including the MS-DOS Fdisk utility or the Windows Setup program, write a sector of data called a master boot record (MBR) to the first sector on a hard disk. The MBR includes a fixed amount of space that contains executable instructions (called boot code) and a partition table with four entries that define the locations of the primary partitions on the disk. *See also* boot code, partition table.

secure attention sequence (SAS) A keystroke combination that when entered notifies Winlogon of a user logon request.

Secure logon facility Requires that users can be uniquely identified and that they must be granted access to the computer only after they have been authenticated in some way.

Security Accounts Manager (SAM) service A set of subroutines responsible for managing the database that contains the usernames and groups defined on the local machine or for a domain (if the system is a domain controller). The SAM runs in the context of the LSASS process.

security auditing A way in which Windows detects and records important security-related events or any attempts to create, access, or delete system resources. Logon identifiers record the identities of all users, making it easier to trace anyone who performs an unauthorized action.

security descriptor The data structure that specifies who can perform what actions on an object. A security descriptor consists of attributes.

security identifier (SID) A means of uniquely identifying entities that perform actions in a system. A SID is a variable-length numeric value that consists of a SID structure revision number, a 48-bit identifier authority value, and a variable number of 32-bit subauthority or relative identifier (RID) values.

security quality of service (SQOS) The indicator specified by a client opening a server resource that specifies the maximum level of client impersonation the server is allowed.

security reference monitor (SRM) A component in the Windows executive (Ntoskrnl.exe) that enforces security policies on the local computer. It guards operating system resources, performing run-time object protection and auditing.

semaphore A counter that provides a resource gate by allowing some maximum number of threads to access the resources protected by the semaphore.

server processes User processes that are Windows services, such as the Event Log and Schedule services. Many add-on server applications, such as Microsoft SQL Server and Microsoft Exchange Server, also include components that run as Windows services.

session Consists of the processes and other system objects (such as the window station, desktops, and windows) that represent a single user's workstation logon session. Each session has a session-specific paged pool area used by the kernel-mode portion of the Windows subsystem (Win32k.sys) to allocate session-private GUI data structures. In addition,

www.TopSage.com

each session has its own copy of the Windows subsystem process (Csrss.exe) and logon process (Winlogon.exe).

session space A component of system space used to map information specific to a user session. The session working set list describes the parts of session space that are resident and in use.

shared cache map A structure that describes the state of a cached file, including its size and (for security reasons) its valid data length.

shared memory Memory visible to more than one process or that is present in more than one virtual address space.

signal state The state of a synchronization object.

simple volume A set of objects that represent sectors from a single partition that file system drivers manage as a single unit.

single sign-on The ability for logon information to be simultaneously authenticated on more than one system. For example, a user logging on to a Windows system might simultaneously be authenticated on a UNIX server. That user would then be able to access resources of the UNIX server from the machine running Windows without requiring additional authentication.

small memory dump A small memory dump (the default on Windows Professional, 64 KB in size) containing the stop code and parameters, the list of loaded device drivers, the data structures that describe the current process and thread (called the EPROCESS and ETHREAD), and the kernel stack for the thread that caused the crash.

spanned volume A single logical volume composed of a maximum of 32 free partitions on one or more disks. The Windows Disk Management Microsoft Management Console (MMC) snap-in combines the partitions into a spanned volume, which can then be formatted for any of the file systems that Windows supports.

sparse files Files, often large, that contain only a small amount of nonzero data relative to their size.

spinlock The locking mechanism the kernel uses to achieve multiprocessor mutual exclusion. The spinlock gets its name from the fact that the kernel (and thus, the processor) is held in limbo, "spinning," until it gets the lock. Spinlocks, like the data structures they protect, reside in global memory. *See also* queued spinlock.

stack frame Information, representing the activation of a procedure, that is pushed onto the stack when a procedure is invoked. A stack frame can have one or more exception handlers associated with it, each of which protects a particular block of code in the source program.

stream A sequence of bytes within a file.

striped volume Series of up to 32 partitions, one partition per disk, that combines into a single logical volume. Striped volumes are also known as RAID level 0 (RAID-0) volumes. A partition in a striped volume need not span an entire disk; the only restriction is that the partitions on each disk be the same size.

stop code An error code that classifies the type of error detected by a component that crashes the system.

structured exception handling A type of exception handling that allows applications to gain control when exceptions occur. The application can then fix the condition and return to the place the exception occurred, unwind the stack (thus terminating execution of the subroutine that raised the exception), or declare back to the system that the exception isn't recognized, and continue searching for an exception handler that might process the exception.

subsection A structure that describes the mapping information for each section of the file (read-only, read-write, copy-on-write, and so on).

subsystem dynamic-link libraries (DLLs) DLLs that translate a documented function into the appropriate 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.

www.TopSage.com

symbolic link A mechanism for referring to an object name indirectly.

symmetric encryption algorithm An algorithm that uses the same key to encrypt and decrypt data. Symmetric encryption algorithms are typically very fast, which makes them suitable for encrypting large amounts of data, such as file data. They do have a weakness, however; their security can be bypassed if their key is obtained.

symmetric multiprocessing (SMP) A multiprocessing operating system in which there is no master processor—the operating system as well as user threads can be scheduled to run on any processor. All the processors share just one memory space.

synchronization A thread's ability to synchronize its execution by waiting for an object to change from one state to another. A thread can synchronize with executive process, thread, file, event, semaphore, mutex, and timer objects. Section, port, access token, object directory, symbolic-link, profile, and key objects don't support synchronization.

synchronous I/O A model for I/O in which a device performs a data transfer and returns a status code when the I/O is complete. The program can then access the transferred data immediately. When used in their simplest form, the Windows ReadFile and WriteFile functions are executed synchronously. They complete an I/O operation before returning control to the caller.

system access-control list (SACL) Specifies which operations by which users should be logged in the security audit log.

system code write protection The setting of the read-only parts of the kernel as read-only pages.

system audit ACE An access-control entry (ACE) contained by a system access-control list (SACL) that, along with the system audit-object ACE, specifies which operations performed on the object by specific users or groups should be audited. System audit-object ACEs specify a GUID indicating the types of objects or subobjects that the ACE applies to and an optional GUID that controls propagation of the ACE to particular child object types. If an SACL is null, no auditing takes place on the object.

system cache Pages used to map files open in the system cache.

system page table entries (PTEs) Pool of system PTEs used to map system pages such as I/O space, kernel stacks, and memory descriptor lists.

system service dispatch table Table in which each entry contains a pointer to a system service rather than to an interrupt handling routine.

system services (or executive system services) Native functions in the Windows 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.

system support processes User processes, such as the logon process and the Session Manager, that are not Windows services (that is, not started by the service controller).

system thread A kind of thread that runs only in kernel mode. System threads always reside in the system process (always process ID 2). These threads have all the attributes and contexts of regular user-mode threads (such as a hardware context, priority, and so on) but run only in kernel-mode executing code loaded in system-space code, whether that be in Ntoskrnl.exe or in any other loaded device driver. System threads don't have a user process address space and hence must allocate any dynamic storage from operating system memory heaps, such as paged or nonpaged pool.

system worker threads Threads created in the system process during system initialization that exist solely to perform work on behalf of other threads.

system working set The physical memory being used by the system cache, paged pool, pageable code in Ntoskrnl.exe, and pageable code in device drivers. *See also* process working set.

www.TopSage.com

thread An entity within a process that Windows schedules for execution. A thread includes the contents of a set of volatile 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 for use by subsystems, run-time libraries, and DLLs; and a unique identifier called a thread ID (also internally called a client ID).

thread context A thread's volatile registers, the stacks, and the private storage area. Because this information is different for each machine architecture that Windows runs on, this structure is architecture-specific. In fact, the CONTEXT structure returned by the Windows GetThreadContext function is the only public data structure in the Windows API that is machine-dependent.

timer A mechanism that notifies a thread when a fixed period of time elapses.

transaction An I/O operation that alters file system data or changes the volume's directory structure. The separate disk updates that make up the transaction must be executed atomically; that is, once the transaction begins to execute, all its disk updates must be completed. If a system failure interrupts the transaction, the part that has been completed must be undone, or rolled back. The rollback operation returns the database to a previously known and consistent state, as if the transaction had never occurred.

transaction table A table that keeps track of transactions that have been started but that aren't yet committed. The suboperations of these active transactions must be removed from the disk during recovery.

transition A kind of invalid page table entry (PTE) in which the desired page is in memory on either the standby, modified, or modified-no-write list. The page is removed from the list and added to the working set.

translation look-aside buffer (TLB) A CPU cache of recently translated virtual page numbers.

trap A processor's mechanism for capturing an executing thread when an exception or an interrupt occurs, switching it from user mode into kernel mode, and transferring control to a fixed location in the operating system. In Windows, the processor transfers control to the kernel's trap handler.

trap frame A data structure in which the execution state of the interrupted thread is stored. This information allows the kernel to resume execution of the thread after handling the interrupt or the exception. The trap frame is usually a subset of a thread's complete context.

trap handler A module in the kernel that acts as a switchboard, fielding exceptions and interrupts detected by the processor and transferring control to code that handles the condition.

trusted facility management Requires support for separate account roles for administrative functions. For example, separate accounts are provided for administration (Administrators), user accounts charged with backing up the computer, and standard users.

trusted path functionality Prevents Trojan horse programs from being able to intercept users' names and passwords as they try to log in. The trusted path functionality in Windows comes in the form of its Ctrl+Alt+Delete logon-attention sequence, which cannot be intercepted by nonprivileged applications. This sequence of keystrokes, which is also known as the secure attention sequence (SAS), always pops up a logon dialog box, so would-be Trojan horses can easily be recognized. A Trojan horse presenting a fake logon dialog box will be bypassed when the SAS is entered.

type object An internal system object that contains information common to each instance of the object.

Unicode An international character set standard that defines unique 16-bit values for most of the world's known character sets.

update records The most common type of record NTFS writes to the log file. Each

www.TopSage.com

update record contains the information needed to redo an operation that updated the file system structure.

user mode The nonprivileged processor mode that applications run in. A limited set of interfaces is available in this mode, and the access to system data is limited. *See also* kernel mode.

VACB index array An array of pointers maintained by the cache manager to keep track of which views for a given file are mapped to the system cache.

view The portion of the section object required by a process. A section object can refer to files that are much larger than can fit in the address space of a process. (If the paging file backs a section object, sufficient space must exist in the paging file to contain it.) To access a very large section object, a process can map only a view of the section by calling the *MapViewOfFile* function and then specifying the range to map. Mapping views permits processes to conserve address space because only the views of the section object needed at the time must be mapped into memory. *See also* section object.

virtual address control blocks (VACBs) The data structures that the cache manager uses to manage the system address space into which it maps files.

virtual address descriptors (VADs) Data structures the memory manager maintains that keep track of which virtual addresses have been reserved in the process's address space. VADs are structured as a self-balancing binary tree to make lookups efficient.

virtual address space A set of virtual memory addresses that a process can use.

virtual block caching A method the Windows cache manager uses to keep track of which parts of which files are in the cache.

virtual cluster numbers (VCNs) VCNs number the clusters belonging to a particular file from 0 through m. VCNs aren't necessarily physically contiguous, but they can be mapped to any number of logical cluster numbers (LCNs) on a volume.

virtual device drivers (VDDs) Drivers used to emulate 16-bit MS-DOS applications. They trap what an MS-DOS application thinks are references to I/O ports and translate them into native Windows I/O functions. Because Windows is a fully protected operating system, user-mode MS-DOS applications can't access hardware directly and thus must go through a real kernel-mode device driver.

virtual memory manager Implements virtual memory, a memory management scheme that provides a large, private address space for each process that can exceed available physical memory.

volume One or more logical disk partitions that are treated as a single unit.

volume file A system file (filename $Volume) that contains the volume name, the version of NTFS for which the volume is formatted, and a bit that when set signifies that a disk corruption has occurred and must be repaired by the Chkdsk utility.

volume manager A term used to represent both FtDisk and DMIO because both FtDisk and DMIO support the same multipartition-volume types.

volume set A single logical volume composed of a maximum of 32 areas of free space on one or more disks.

wait block A data structure that represents a thread waiting on an object. Each thread that is in a wait state has a list of the wait blocks that represent the objects the thread is waiting on. Each dispatcher object has a list of the wait blocks that represent which threads are waiting on the object.

wait hint A value indicating how long a service should wait before informing the system a shutdown is complete.

WDM drivers Device drivers that adhere to the Windows Driver Model (WDM). WDM includes support for Windows power management, Plug and Play, and Windows Management Instrumentation (WMI). WDM is implemented on Windows 2000 and later, Windows 98, and Windows Millennium Edition, so WDM drivers are source compatible between these operating systems and in

www.TopSage.com

many cases are also binary compatible. There are three types of WDM drivers: bus drivers, function drivers, and filter drivers.

WDM Windows Management Instrumentation routines 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.

Windows API The 32-bit interface to Windows 2000 and both the 32-bit and 64-bit programming interfaces to Windows XP and Windows Server 2003.

Windows services A mechanism to start processes at system startup time that provide services not tied to an interactive user. Services are similar to UNIX daemon processes and often implement the server side of client/server applications.

window station A window station contains desktops, and desktops contain windows. Only one window station can be visible on a console and receive user mouse and keyboard input. In a Terminal Services environment, one window station per session is visible, but services all run as part of the console session.

Windows drivers Device drivers that integrate with the Windows power manager and Plug and Play manager, when required. They include drivers for mass storage devices, protocol stacks, and network adapters.

Windows Management Instrumentation (WMI) manager A component of the executive that enables 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.

work item A unit of work placed on a queue dispatcher object when a device driver or an executive component requests a system worker thread's services by calling the executive functions ExQueueWorkItem or IoQueueWorkItem. Work items include a pointer to a routine and a parameter that the thread passes to the routine when it processes the work item. The routine is implemented by the device driver or executive component that requires passive-level execution.

working set A subset of virtual pages resident in physical memory. There are two kinds of working sets—process working sets and the system working set.

working set manager A routine that runs in the context of the balance set manager system thread to initiate automatic working set trimming to increase the amount of free memory available in the system. Although Windows attempts to keep memory available by writing modified pages to disk, when modified pages are being generated at a very high rate, more memory is required to meet memory demands. The working set manager is called when physical memory runs low (when MmAvailablePages is less than MmMinimumFreePages).

write-back A caching strategy the lazy write file system uses to improve performance. In write-back, the file system writes file modifications to the cache and flushes the contents of the cache to disk in an optimized way, usually as a background activity.

write-through An algorithm the FAT file system uses that causes disk modifications to be immediately written to the disk. Unlike the careful-write approach, the write-through technique doesn't require the file system to order its writes to prevent inconsistencies. *See also* careful write.

write throttling Prevents system performance from degrading because of a lack of memory when a file system or network server issues a large write operation.

zero page thread A kernel-mode system thread (thread 0 in the system process). A zero page thread zeroes out pages on the free list so that a cache of zero pages is available to satisfy future demand-zero page faults.

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引

术语对照表

access-control entry, ACE	访问控制项
access-control list, ACL	访问控制列表
access token	访问令牌
access validation	访问验证
Active Directory	活动目录
Address Windowing Extension, AWE	地址窗口扩展
affinity	亲和性
attach	附载
assertion	断言
asymmetric multiprocessing, ASMP	非对称多处理
asynchronous procedure call, APC	异步过程调用
authentication	认证
balance set manager	平衡集管理器
bin	巢箱
bind, binding	绑定
blue screen	蓝屏
boost	提升
boot	引导
boot sector	引导扇区
boot partition	引导分区
boot volume	引导卷
bucket	桶
bug check	错误检查
bus driver	总线型驱动程序
cable	线缆
cache manager	缓存管理器
cache miss	缓存未命中
Careful Write file system	谨慎写文件系统
catalog file	编目文件
cell	巢室
change journal	变化日志
class driver	类驱动程序
cluster	簇(用于磁盘存储单元)，集群
COM	组件对象模型
Common Internet File System, CIFS	Internet文件系统
commit	提交
committed memory	提交的内存
compatibility	兼容性
concurrency	并发性
context	环境
context switch	环境切换
copy-on-write	写时复制
crash analysis	崩溃分析
crash dump	崩溃转储
credencial	凭证
critical section	临界区
cryptographic service provider, CSP	密码服务提供者
cyclic redundancy checksum, CRC	循环冗余检验和
DCOM	分布式组件对象模型
deadlock	死锁
debugger	调试器
deferred procedure call, DPC	延迟过程调用
defragmenter	碎片整理器
deliver	交付(针对APC&DPC)
Dependency	相依性

www.TopSage.com

desktop	桌面
device driver	设备驱动程序
Device Enumeration	设备列举
Discretionary access control, DAC	自主访问控制
dismount	卸载
disk spanning	磁盘跨展
dispatch/DPC	Dispatch/DPC，指一种IRQL，保留不译
dispatcher	分发器
dispatching	分发
Distributed File System, DFS	分布式文件系统
docking station	插接站
double-freeing	两次释放
DPC/Dispatch	DPC/Dispatch，指一种IRQL，保留不译
drive letter	驱动器字母
Driver Verifier	驱动程序检验器
dynamic link library, DLL	动态链接库
Encrypting File System, EFS	加密文件系统
environment subsystem	环境子系统
event	事件
Event Log	事件日志
ETW, Event Tracing for Windows	Windows事件跟踪
exception	异常
excpetion dispatching	异常分发
exception frame	异常帧
exception handling	异常处理
exception handler	异常处理器
executive	执行体
executive object	执行体对象
extended partition	扩展分区
Extensible Firmware Interface, EFI	可扩展的固件接口
extensibility	扩展性
fast mutex	快速互斥体
fault tolerance	容错
fiber	纤程
fibre channel	光纤通道
file allocation table	文件分配表
file encryption key, FEK	文件加密密钥
file mapping object	文件映射对象
file system driver, FSD	文件系统驱动程序
file system recognizer	文件系统识别器
firmware	固件
filter driver	过滤型驱动程序
footprint	印迹
free list	空闲(页面)列表
function driver	功能型驱动程序
GINA	图形化标识和认证
Group Policy	组策略
guard page	守护页面
GUID	全局惟一标识符
GUID Partition Table, GPT	GUID分区表
handle	句柄
handle table	句柄表
handler	处理器
hardware abstraction layer	硬件抽象层(HAL)
heap manager	堆管理器
hive	储巢
hotfix	热补丁
hyperthreading	超线程
I/O request packet, IRP	I/O请求包
idle process	空闲进程
idle thread	空闲线程
impersonation	模仿
inter-processor interrupt, IPI	处理器间的中断
interrupt	中断
interrupt controller	中断控制器
interrupt dispatch table	中断分发表
interrupt handler	中断处理器
interrupt request level, IRQL	中断请求级别

www.TopSage.com

Interrupt service routine, ISR	中断服务例程
IoCompletion	IO完成
job	作业
junction	交接
keystroke	键击
kernel	内核
kernel mode	内核模式
kernel object	内核对象
key ring	密钥环
native	原生的
last known good, LKG	最后已知的好配置，最后已知的好控制集
lazy writer	延迟写出器
lazy write file system	延迟写文件系统
Lightweight Directory Access Protocol, LDAP	轻量的目录访问协议
local procedure call, LPC	本地过程调用
local security authentication	本地安全认证
logical cluster number, LCN	逻辑簇号
Logical Disk Manager, LDM	逻辑磁盘管理器
logical prefetcher	逻辑预取器
logical sequence number, LSN	逻辑序列号
logon	登录
look-aside list	预读列表
Low Fragmentation Heap, LFH	低碎片堆
LUID	本地惟一标识符
mailslot	邮件槽
managed code	托管代码
manifest	清单
mapped file	映射的文件
master boot record, MBR	主引导纪录
master file table	主文件表
mechanism	机制
memory	内存
memory descriptor list, MDL	内存描述符列表
memory mapped file	内存映射文件
metadata	元数据
microkernel	微内核
minidump	小转储
miniport	小端口
miniport driver	小端口驱动程序
mirrored volume	镜像卷
MMC snap-in	MMC加载件
modified page writer	已修改页面写出器
mount	挂载
mount point	挂载点
mutant	突变体
mutex	互斥体
multimaster replication	多主复制
Multiple Provider Router	多提供者转发器
Multiple UNC Provider, MUP	多UNC提供者
multiprocessing	多处理
multitasking	多任务
named pipe	命名管道
network address translation, NAT	网络地址转译
Network Driver Interface Specification, NDIS	网络驱动程序接口规范
Network Monitor	网络监视器
network share	网络共享体
NUMA(non-uniform memory architecture)	非一致的内存结构
NTFS object attribute	NTFS对象属性
bject linking and embedding, OLE	对象链接和嵌入
object manager	对象管理器
Open System Interconnection, OSI	开放系统互联
opportunistic locking, oplock	机会锁

www.TopSage.com

on-disk structure	磁盘上的结构
overlapped	可重叠的
page	页面
page directory	页目录
page directory entry, PDE	页目录项
page fault	页面错误
page file, paging file	页面文件
page frame number, PFN	页面帧编号
page table	页表
page table entry, PTE	页表项
pageable	换页的
paged pool, nonpaged pool	换页内存池，非换页内存池
pager	换页器
partition	分区
partition table	分区表
payload	有效载荷
Performance Monitor	性能监视器
Performance tool	性能工具
Physical Address Extension, PAE	物理地址扩展
placement policy	放置策略
plug and play	即插即用
pool tracking	内存池跟踪
port driver	端口驱动程序
portability	可移植性
preempt	抢占
preemptive	抢先式
prefetcher	预取器
principal	安全个体
priority boost	优先级提升
priority inversion	优先级倒置
Privilege	特权
procedure	过程
process	进程
Process environment block, PEB	进程环境块
process explorer	进程管理器
process manager	进程管理器(视上下文)
processor	处理器
profile	轮廓，性能剖析
prototype page table entry, prototype PTE	原型页表项
push lock	压栈锁
quality of service, QOS	服务质量
quantum	时限
quantum boosting	时限增大
queue	队列
quota	配额
Quota charges	配额花费
RAM Optimizer	RAM优化器
read-ahead	预读
record	记录(动词)，纪录(名词)
Recovery Console	恢复控制台
redirector	重定向器
redo	重做
reference count	引用计数
reference monitor	引用监视器
registry	注册表
registry key	注册表键
registry value	注册表值
reliability	可靠性
Remote Direct Memory Access, RDMA	远程直接内存访问
Remote NDIS	外接NDIS
remote procedure call, RPC	远过程调用
recoverability	可恢复性
reparse	重解析
reparse point	重解析点
reserved memory	保留的内存
resource accounting	资源记账
resource gate	资源门控
Resource Reservation	资源预留协议

www.TopSage.com

Protocol, RSVP	
robustness	健壮性
routine	例程
run	行串(用于NTFS结构中)
scalability	可伸缩性
scatter/gather I/O	分散/聚集I/O
scheduler	调度器
schema	表结构
secondary object namespace	从属对象名字空间
section, memory section	内存区
section object	内存区对象
sector sparing	扇区备用
secure attention sequence, SAS	安全注意序列
Security Accounts Manager, SAM	安全账户管理器
security credential	安全凭证
Security descriptor	安全描述符
security identifier, SID	安全标识符
security quality of service, SQOS	安全服务质量
Security reference monitor	安全引用监视器
Security Support Provider Interface,SSPI	安全支持提供者接口
security validation	安全验证
Semaphore	信号量
serial port	串行口
server	服务器
service	服务
service descriptor table	服务描述符表
session manager	会话管理器
session namespace	会话名字空间
Service Control Manager, SCM	服务控制管理器
Service Pack	服务补丁包
shared memory	共享的内存
shutdown	停机
socket	套接字
soft fault	软错误
Software Restriction Policy	软件限制策略
spanned volume	跨距卷
spinlock	自旋锁
spin-down	停止旋转
splash screen	启动屏幕
stack frame	栈帧
stack location	栈单元
stack overrun	栈溢出
stack trash	栈破坏
standby list	备用(页面)列表
startup	启动
Starvation	饥饿
storage area network, SAN	存储区域网络
striped volume	条带卷
stub	存根
symbolic link	符号链接
symmetric multiprocessing, SMP	对称多处理
synchronization	同步
system service	系统服务
system service call	系统服务调用
system service dispatching	系统服务分发
system volume	系统卷
targeted DPC	定向的DPC(与特定的CPU相关联)
Task Manager	任务管理器
task offloading	任务外移
TDI transport	TDI传输器
terminal service	终端服务
thread	线程
thread-thrashing	线程频繁轮换
Thread environment block, TEB	线程环境块
thread scheduling	线程调度
tick	嘀嗒
time slice	时间片

www.TopSage.com

timer	定时器
TLS	线程局部存储区
transaction	事务
Translation Look-aside Buffer, TLB	地址转译快查缓冲区
transition	转移，转换
Transport Driver Interface, TDI	传输驱动程序接口
trap	捕获(动词)，陷阱(名词)
trap frame	陷阱帧
trap handler	陷阱处理器
trim	修剪
truncate	截断、截短
undo	撤销
unicast	单播
Universal Disk Format, UDF	通用磁盘格式
Universal Naming Convention, UNC	统一命名规范
Universal Plug and Play, UPnP	通用即插即用
unmanaged code	非托管代码
user mode	用户模式
view	视图(确切含义视上下文)
virtual address control block, VACB	虚拟地址控制块
Virtual address descriptor, VAD	虚拟地址描述符
virtual cluster number, VCN	虚拟簇号
volatile	易失的、易失性
window station	窗口站
Windows Driver Model, WDM	Windows驱动程序模型
Windows Error Reporting	Windows错误报告
Windows File Protection, WFP	Windows文件保护机制
Windows Global Flags	Windows全局标志
Windows Internet Name Service, WINS	Windows Internet名称服务
Windows Management Instrumentation	Windows管理规范
Windows Sockets Direct, WSD	Windows套接字直接访问
working set	工作集
working set trimming	工作集修剪
write throttling	写节流
write-through	直写
write-behind	滞后写
yield	放弃
zero list	零(页面)列表
zero page thread	零页面线程

www.TopSage.com

索引

Symbols and Numbers

: (colon), 720
$ (dollar sign), 733
! (exclamation point), 30
?? (question marks directory), 147, 551, 559, 621, 701
3DES encryption algorithm, 775
16-bit systems, 2
32-bit systems
  execution protection, 390
  system service dispatching, 119–120
  Windows versions, 2, 15
64-bit systems
  address space layouts, 422
  execution protection, 390
  extended systems, 40
  handle tables, 139
  system service dispatching, 120
  Windows versions, 3, 16. *See also* Windows Server 2003; Windows XP

A

abstract classes, WMI CIM, 240
accept function, 793
accept TDI IRP, 823
AcceptEx function, 793
Access, 716
access checks, 493–505. *See also* access control
  access tokens, 497–502
  impersonation, 502–504
  object manager, 493–495
  restricted tokens, 504–505
  security identifiers, 495–497
access control. *See also* security
  access checks, 493–505
  C2 security rating, 486
  security descriptors, 506–515
  Windows and, 23
Access token executive object type, 127
access tokens
  access checks, 497–502
  EPROCESS block, 294
  jobwide, 370
  LSASS, 81
  processes, 6, 14, 309
  restricted, 504–505
  user logon, 531
  viewing, with kernel debugger, 500–502
AccessCheck function, 511
AccessCheckByType function, 511
access-control entries (ACEs)
  relative ordering of, 512
  security descriptors, 506–507
access-control lists (ACLs)
  assignment, 509–511
  file objects, 557
  section objects, 390
  security descriptors, 506–507
access-denied errors, 195, 712
account rights, 516–518. *See also* privileges
  accounts. *See* administrator accounts; service accounts; user accounts
  Accvio.exe tool, 117
Active Directory
  authentication, 531
  DNS, 820
  domain accounts, 190
  IPSec Group Policies, 827
  network service and APIs, 840–841
  RPC, 801
  security, 487, 489
Active Directory Service Interfaces (ADSI), 840
Active Directory Users and Groups MMC snap-in, 516
active logon sessions, 532–533
active pages, 442, 469
active processes, 369
active processor mask, 358
ActiveX controls, WMI, 239
add-device routines, device driver, 548, 592
address objects, TDI, 823
address open TDI IRP, 823
address space. *See* virtual address space
address translation, 425–438
  byte index, 432
  IA64, 437
  large pages, 383
  memory manager task, 375
  page directories, 428–430
  page tables and page table entries, 425–427, 430–432
  Physical Address Extension (PAE), 435–437
  process, 433–434
  translation look-aside buffer, 434
  x64, 438
  x86, 425–434
Address Windowing Extensions (AWE), 15, 399–400
addresses. *See also* virtual address space
  viewing start, for threads, 113
  well-known, 794
AddUsersToEncryptedFile function, 785
  Administrative Tools, 83
administrator accounts
  Recovery Console, 279
  RIDs, 495
  security ratings, 487
Advanced Configuration and Power Interface (ACPI) specification, 607
Advanced Encryption Standard (AES), 775
Advanced Power Management (APM) standard, 607
  Advanced Programmable Interrupt Controllers (APICs), 89, 260
Advanced RISC Computing (ARC) naming convention, 254, 616
Advanced Settings dialog box, 817, 838
Advanced Technology Attachment (ATA) systems, 252
Advapi32.dll, 39, 56, 784
AES (Advanced Encryption Standard), 775
affinity, processor. *See* processor affinity
  affinity manager, 397
Aha154x.sys, 618
alertable wait states, 109
algorithms
  demand-paging, 448, 458
  EFS encryption, 775–778

www.TopSage.com

algorithms, *continued*
multiprocessor thread scheduling, 366–369
security descriptors and access, 511–515
AllocateUserPhysicalPages function, 399
allocation granularity, 385, 394
Alpha AXP systems, 40
alternate accounts, service, 220
alternate hives, 210
AMD processors, 40, 120
analysis pass, NTFS recovery, 767
Ancillary Function Driver (AFD), 796
anonymous access, local service accounts and, 220
ANSI character strings, 25
anti-replay, IPSec, 827
APC objects, 66, 108. *See also* asynchronous procedure calls (APCs)
APC_LEVEL IRQLs, 94, 96, 108, 163
APCs (asynchronous procedure calls), 108–109, 163, 572, 800
APICs (Advanced Programmable Interrupt Controllers), 89, 260
APIs (application programming interfaces)
Active Directory APIs, 841
AuthZ API, 516
CryptoAPI, 777, 783
Data Protection API, 487
networking APIs, 791–815
Performance Data Helper API, 191
Windows API. *See* Windows API
APM (Advanced Power Management) standard, 607
AppleTalk protocol, 821
Application Compatibility Toolkit, 391, 398
application desktops. *See* desktops
application layer, OSI model, 788
application programming interfaces. *See* APIs (application programming interfaces)
applications. *See also* executable images; programs
boot-verification, 230
client. *See* clients
Event Tracing for Windows, 175
locking memory, 385
management, 237
Microsoft iSCSI Software Initiator management, 618
processes, 38, 302
registry settings, 184, 194
Regmon tool troubleshooting techniques, 195–196
restoring, using System Restore, 706
running, as services, 217
security functions, 494
server. *See* servers
service, 212–217
setup. *See* setup programs
side-by-side assemblies, 311
startup and logical prefetcher, 311, 458. *See also* prefetching
tools. *See* tools
viewing registry polling activity, 193
Applied Microsoft .NET Framework Programming, 4
ARC (Advanced RISC Computing) naming convention, 254, 616
architecture, Windows. *See* system architecture, Windows
ASCII characters, 211
ASMP (asymmetric multiprocessing), 41
assemblies, side-by-side, 311–313
assembly programming language, 40, 550
ASSERT macro, 50
assignment, ACL, 509–511
assignment, volume drive-letter. *See* volume namespace
associate address TDI IRP, 823
association classes, WMI, 244
associative memory, TLB, 434
asymmetric multiprocessing (ASMP), 41
asynchronous I/O, 561–562. *See also* I/O system
asynchronous procedure calls (APCs), 108–109, 163, 572, 800
asynchronous read-ahead with history, 682
asynchronous socket operations, Winsock, 793
Asynchronous Transfer Mode (ATM), 830
ATA (Advanced Technology Attachment) systems, 252
Atapi.sys, 617
atomic transactions, 717
attributes
executive object header, 129
file object, 555
NTFS, 719, 738, 740–742, 744–747
object, 22
section object, 450, 451
type object, 133
audio calls, 814
Audit Log, 507
auditing, security, 23, 486, 506, 507, 524–526
authentication. *See also* security
Active Directory APIs, 841
IPSec, 827
packages, 489, 526, 530
password, 23
RPC, 801
AuthZ API, 516
Autochk.exe tool, 645
Automated System Recovery (ASR), 282
automatic startup, programs and, 273
Autoruns tool, 274
auto-start device driver loading, 597
auto-start services. *See* services
AWE (Address Windowing Extensions), 15, 399–400

B

backing up
EFS and, 780, 782, 784
registry, 201
Volume Shadow Copy service, 649–654
Backup tool
compacting registry, 201
restoring corrupted system files, 282
restoring SYSTEM hive, 283
bad pages, 469
bad-cluster recovery, NTFS, 722, 738, 771–775
balance set manager, 77, 354, 376, 464, 466
base blocks, hive, 200
Base Cryptographic Provider 1.0, 778, 782
base file record, NTFS, 734
base named objects, 145
base priority, thread and process, 328
BaseNamedObjects subdirectory, 621

www.TopSage.com

basic disks, 622, 624–626. *See also* disks
  dynamic disks vs., 622
  FtDisk driver volume manager, 625
  GUID Partition Table (GPT) partitioning, 624
  Master Boot Record (MBR) partitioning, 624
batching
  GDI operations, 57
  recoverable file system, 761
binary MOF (BMF) files, 243
bind function, 792
binding, networking and, 838
bins, hive, 201
BIOS (Basic Input Output System)
  ACPI standard, 607
  boot process, 253
bitmap file, NTFS, 738
B-level security rating, 487
blocks, hive, 200
blue screen, crashes and, 846–849
Blue Screen screen saver, 870
bodies, executive object, 129–130
boosting, quantum, 343, 344. *See also* quantum
boosts, priority. *See* priority boosts
boot code, 253, 256
boot device drivers, 263
boot loader. *See* Ntldr (boot loader)
boot log files
  boot option, 257
  initialization, 268
  post-splash screen crashes, 286
  safe mode boots, 277–279
boot menu, 261, 263, 274, 275. *See also* Boot.ini
boot partitions, 253
boot process, 251–286
  Autoruns tool and automatic program startup, 274
  blue screen recovery and last known good configurations, 848
  boot sector corruption problem, 281
  Boot.ini misconfiguration problem, 282
  common problem solutions, 281–286
  crash dump generation, 852
  debugging mode, 868
  device driver loading, 594
  disk configuration changes and rebooting, 623
Driver Verifier , 411, 860
encryption, 727
executable images that start automatically, 273
IA64, 264–265
initializing kernel and executive subsystems, 266–269
last known good (LKG) mode, 274
logical prefetcher, 458
Master Boot Record, 624
Master Boot Record corruption problem, 281
memory manager configuration, 378
NTFS boot file, 738
NTFS recovery, 763
page files, 445
post-splash screen crash or hang problem, 284–286
Recovery Console, 279–284
registry, 184, 190
safe mode boots, 274–279
services, 225–230. *See also* services
Smss, Csrss, Winlogon, 269–273
system file corruption problem, 282–283
SYSTEM hive corruption problem, 283
System Restore, 711
system service dispatching, 120
troubleshooting, 274–286
user address space, 415
virtual memory options, 15
Windows File Protection, 283
x86 and x64 preboot, 251–255
x86/x64 boot sector and Ntldr, 255–264
boot sectors
  boot options, 262
  boot process, 252
  corruption problem, 281
  file system drivers, 695
  NTFS MFT, 737
  NTFS recovery, 774
  Ntldr, 255–264
  volume management, 624, 644
boot volumes, 263, 630, 636
Boot.ini
  boot menu. *See* boot menu
  debugging-mode option, 868
  enabling PAE kernel, 436
  execution protection, 391
  misconfiguration problem, 282
  Ntldr, 256
  options, 257–263
  partition names, 616
  Setup and creation, 254
  virtual memory options, 15
Bootcfg.exe tool, 257, 264
Bootsect.dos, 254, 256
Bootsect.w40, 262
boot-verification programs, 230
breakpoints, debugger. *See also* debugging
  exceptions, 112
  HAL boot option, 257
broadcasts. *See* mailslots
BSD (Berkeley Software Distribution) Sockets API, 791
bucket ID, online crash analysis, 854
buffer management
  cache manager file system interfaces, 677
  IRP, 568–569
buffer overrun, Driver Verifier and, 861–863
buffered I/O, 568
buffer-overflow errors, 195, 713
bug check codes, 846
bus drivers
  device drivers, 71
  device enumeration, 596
  IRQLs, 95
  Plug and Play manager, 538
  Plug and Play support, 592
  WDM, 542
bus filter drivers, 599
Bypass Traverse Checking privilege, 523
byte index, 426, 432
byte mode, named pipe, 805

C

C and C++ programming languages, 37, 40, 550, 840
C2 security rating, 472, 486
cache, PTE, 383, 432, 434
cache, system. *See* cache manager; system cache
cache coherency, 656
cache extra memory, 663
cache flushing. *See* write-through caching
cache manager, 655–688
  cache data structures, 668–674
  cache size, 662–668

www.TopSage.com

cache manager, *continued*
executive, 65
explicit file I/O, 703
fast I/O, 679–681
features, 655–659
file system drivers, 694
file system interfaces, 674–679
initialization, 268
lazy writer and read-ahead threads, 704
mapped file I/O, 564
mapping hive files into memory, 203
memory manager, 656
NTFS defragmentation, 728
NTFS file system driver, 730
NTFS log file service, 762
read ahead and write behind, 682–688
remote FSDs, 697
virtual memory management, 660–661
Winsock, 794
cache size, 662–668
physical size, 667–668
registry, 662
virtual size, 663–664
working set size, 665–666
cache table, 211
cache write-through. See write-through caching
cache-aware push locks, 166
caching. *See also* cache manager; system cache
decrypted FEK, 783
virtual block, 658
call managers, NDIS, 833
callback functions, Winsock, 793
callback mechanism, Regmon and registry, 192
cancel I/O routines, device driver, 549
careful write file systems, 759
Carnegie Mellon University Mach operating system, 36
carriage return/line feed (CR/LF) and carriage return (CR), 788
case-sensitive names, 144
catalog files, 605
CcCanIWrite function, 686
CcCopyRead function, 703
CcCopyWrite function, 704
CcDeferWrite function, 686
CcFastCopyRead and CcFastCopyWrite functions, 704
CcInitializeCacheMap function, 674, 703
CcSetDirtyPageThreshold function, 687
CDFS (CD-ROM file system), 690
cell indexes, hive, 202, 206
cells, hive, 200
central processing units. *See* CPUs (central processing units)
centralized system cache, 656. *See also* cache manager
CfgMgr (Configuration Manager) function, 603
chains, file allocation, 692
change journal, NTFS, 724, 739, 752
change logging, NTFS. *See* logging, NTFS
checked build versions, Windows, 49–51
checked kernel environment, 259
checkpoint records, NTFS, 765–766
Chkdsk tool, 278, 282, 283, 645, 738, 758, 771, 773
ChkReg tool, 284
CIM_FileSystem class, 241
CIM_LogicalFile class, 241
CIM_ManagedSystemElement core class, 240
CIMOM Object Repository, 238, 248
CIMV2 namespace, 245
cipher tool, 775, 778
class drivers, 543, 617
class keys, device, 602
classes
class/port/miniport architecture, 617–619
job-scheduling, 370
process priority, 301, 328, 331, 370
registry and COM, 189
trace, 176
WMI, 239, 240, 243, 244
client communication ports, 174
client ID, 6, 12
client versions, Windows, 47–49. *See also* versions, Windows
clients
connection-oriented NDIS, 833
file system driver, 696
named pipe, 805
TDI, 789
Winsock, 792
client/server programming model, 502
client-side library (Kernel32.dll), 300
clock, system. *See* system clock
clock intervals, quantum and, 340
clock IRQL, 96
Clockres tool, 340
close method, executive object, 134
CloseEncryptedFileRaw function, 784
cluster hosts, 842
clusters
defined, 689
FAT12, FAT16, and FAT32, 691, 693
NTFS, 694, 732
NTFS bad-cluster recovery, 722, 771–775
Cmd.exe, 303, 331
code
boot, 253, 256
overwrite corruption crash dump analysis, 863–864
system, 414
write protection, 860, 863–864
coherency, cache, 656
collided page faults, 444
colon (:), 720
COM API, 814
COM classes, registry and, 189
command prompt safe mode boots, 275
committed memory
committed pages, 384–385
page files, 445. *See also* page files
section objects, 387
committed transactions, NTFS logging and, 764
Common Criteria (CC), 23, 485, 487–488
Common Information Model (CIM), 240–243
Common Information Model (CIM) Object Manager (CIMOM), 238
Common Internet File System (CIFS) protocol, 697, 808
Common Language Runtime (CLR), 4
common model, WMI CIM, 241
compacting registry, 201

www.TopSage.com

compatibility, Windows, 36. *See also* versions, Windows
Compatibility Administrator Tool, 391
compatible IDs, 604
compiler, Microsoft Interface Definition Language, 800
complete memory dump crash dump files, 849
completion ports, I/O. *See* I/O completion ports
component entries, LDM, 628
components, Windows system. *See* system components, Windows
compression units, NTFS, 750. *See also* data compression, NTFS
Computer Management tool, 72, 546–548, 699, 848
Compuware NuMega SoftICE tool, 33
concurrency, I/O completion port, 585, 587
configuration. *See also* configuration manager; registry
  Driver Verifier, 409–411
  hardware. *See* Plug and Play (PnP) manager
  memory manager, 378
  system code write protection, 864
  system memory pools, 401–404
configuration manager, 197–211. *See also* registry
  boot option, 260
  cell maps, 203–207
  executive, 64
  hive size limits, 198–199
  hive structure, 200–203
  hives and registry keys, 197–198
  initialization, 268, 269
  Key executive object type, 128
  manually loading and unloading hives, 198
  registry namespace, 134, 207–209
  registry optimizations, 210–211
  stable storage for hives, 209–210
  viewing hive handles, 199
  viewing hive memory usage, 205
  viewing hive paged pool usage, 204
  viewing key control blocks, 208
connect TDI IRP, 823
ConnectEx function, 794
connection objects, TDI, 823
connection open TDI IRP, 823
connectionless protocols, 793
connection-oriented NDIS, 832
console programs, service applications as, 212
  console sessions, 21, 147
consolidated security descriptors, NTFS, 756–758
consumer applications, event tracing and, 175
context switching
  context swap spinlocks, 358
  I/O completion ports, 585
  kernel, 67
  multiprocessor thread scheduling, 359
  thread scheduling, 327, 347
contexts
  access tokens and security, 497
  thread, 12, 308–309. *See also* context switching
control area data structure, 673
control areas, section object, 453, 454–457
control codes, device I/O, 569. *See also* DeviceIoControl function
control handler functions, 216
control objects, kernel, 66
Control Point API, 815
Controlled Access PP rating, 487
controller applications, event tracing and, 175
ConvertThreadToFiber function, 13
copies, shadow, 649–654. *See also* backing up
copying interfaces, cache manager, 676
copy-on-write page protection, 392–394
core heap layer, 395
core model, WMI CIM, 240
Corporate Error Reporting tool set, 119
CPU Stress tool
  changing image affinity, 364
  watching foreground priority boosts and decays, 351
  watching priority boosts for CPU starvation, 355–357
CPUs (central processing units)
  access modes. *See* kernel mode; user mode
  central processor value, HKLM value, 190
  job objects and time limits, 370
  priority boosts and starvation, 354–357
  processor affinity. *See* processor affinity
  thread scheduling. *See* multiprocessor thread scheduling
crash dump analysis, 845–870
  basic, 855–859
  basic kernel debugger, 856–858
  blue screen function, 846–849
  Blue Screen screen saver, 870
  code overwrite and system code write protection, 863–864
  crash dump absence, 869
  crash dump files, 849–851
  crash dump generation, 852
  crash reasons, 845–846
  crash troubleshooting tools, 860–864
  Driver Verifier special pool option and buffer overruns, 861–863
  generating crashes using Notmyfault tool, 855
  hung or unresponsive systems, 866–869
  levels of severity, 860
  manual kernel debugger, 864–870
  monitoring pool usage, 406
  online crash analysis, 854–855
  stack trashing, 865–866
  verbose kernel debugger, 858–859
  Windows Error Reporting, 853
crash dump files. *See also* crash dump analysis
  Dr. Watson tool, 115–117
  kernel debugging, 29
  LiveKd, 32
  system space, 415
crashes
  blue screen function, 846–849
  boot options, 258
  device driver, 274
  Dr. Watson tool, 115–117
  generating, using Notmyfault tool, 851, 855
  generating, using registry setting, 867
  hive inconsistency, 210
  initialization, 269
  kernel mode, 37
  NTFS design, 718
  NTFS recovery. *See* recovery, NTFS
  post-splash screen, 284–286
  reasons for, 845–846

www.TopSage.com

crashes, *continued*
Session Manager, 80
troubleshooting. *See* crash dump analysis
USER and GDI in kernel mode, 58
Windows Error Reporting, 118–119
Create a token object privilege, 524
CreateFiber function, 13
CreateFile function, 127, 503, 556, 561, 621, 661, 682, 684, 701
CreateFileMapping function, 387, 564
CreateHardLink function, 722
CreateIoCompletionPort function, 586, 587
CreateMailslot function, 806
CreateMemoryResourceNotification function, 479
CreateNamedPipe function, 805
CreateProcess function, 135, 300–313, 328, 534
CreateProcessAsUser function, 228, 498, 503
CreateProcessWithLogon function, 498
CreateRestrictedToken function, 504
CreateService function, 212, 236
CreateThread function, 112, 322, 324
creator processes, 6–8
critical sections
keyed events, 160
synchronization, 150
critical worker threads, 167
CritSecOutOfMemoryEvent keyed event, 160
CryptoAPI, 777, 783
cryptography. *See* Encrypting File System (EFS)
cscript command, 246
Csrss (Windows subsystem process)
boot process, 269–272
exceptions, 112
notifying, about new processes, 309–310
process creation, 300
process data structures, 290
service shutdown, 232
sessions, 414
system shutdown, 286–288
thread data structures, 313
as Windows Client/Server Run-Time Subsystem (Csrss), 53, 55
Ctrl+Alt+Delete key combination, 8, 81, 487, 529
Ctrl+Break key combination, 32
Ctrl+C key combination, 32
Ctrl+H key combination, 20
Ctrl+Shift+ESC key combination, 8
CTRL_LOGOFF_EVENT event, 287
CTRL_SHUTDOWN_EVENT event, 288
current hardware profile, 190
current priority, thread, 328
cyclic redundancy checksums (CRC), 625

D

data
caching. *See* cache manager
compression. *See* data compression, NTFS
configuration. *See* registry
databases. *See* databases
EFS decryption, 784
EFS encryption, 780
hive format, 200
IPSec, 827
NTFS data streams, 719–721
NTFS recovery and user, 758
redundancy, 633, 718, 773
structures. *See* data structures
types. *See data types*
data compression, NTFS, 747–752
compressing nonsparse data, 750–751
compressing sparse data, 748–749
sparse files, 723, 752
Data Decryption Field (DDF), 779
Data Encryption Standard (DES), 775
data execution prevention (DEP), 389, 390–392
Data Protection API, 487
Data Recovery Field (DRF), 779
data redundancy
multipartition volume, 633
NTFS, 718, 773
data structures. *See also* data; databases
cache, 668–674
crash dump files, 850, 869
DDK header files, 34
dispatcher object, 161–163
file object, 556
NTFS, 731
objects vs., 22
process, 289–296
section object, 453
thread, 290, 313–320
viewing type information for kernel, 31–32
data types
hive cell, 200
registry, 185–186
databases
Active Directory, 841
LDM, 627–629
Lsass policy, 489
Mount Manager, 641
multiprocessor thread scheduling dispatcher, 357–359
object manager, 124
page frame number (PFN), 476–479
SAM, 489
service, 223
thread-scheduling dispatcher, 338–340
datagrams, 793
data-link layer, OSI model, 789
DbgPrint function, 50
Dbgview.exe tool, 51
DCOM (Distributed Component Object Model) API, 814
Dcpromo.exe tool, 495
DDK. *See* Device Driver Kit (DDK)
deadlocks
detecting, 166
hung or unresponsive systems, 866
debug objects, 112
Debug programs privilege, 523
DebugActiveProcess function, 29
debugging. *See also* errors; exceptions; troubleshooting
booting into debugging mode, 868
breakpoints, 112, 257
copy-on-write breakpoint support, 393
crashes. *See* crash dump analysis
heap manager features, 398
kernel, 28–33. *See also* kernel debugger
pageheap tool, 398
process creation, 304, 310
unhandled exceptions, 115
user-mode exceptions and debugger ports, 111
Windows debug versions, 49–51

www.TopSage.com

DecryptFile function, 727
decryption process, EFS, 783–784
default cluster factor, NTFS, 732
default exception handlers, 111
default process heap, 395
default security, 493
Defense Advanced Research Projects Agency (DARPA), 821
deferred procedure calls (DPCs)
  device drivers, 549
  interrupts, 104–107
  IRP interrupt servicing, 571
  objects, 66
  system processes, 76
deferred ready state, thread, 338, 359
DefineDosDevice function, 559
defragmentation
  logical prefetcher, 462
  NTFS, 727
  page file, 445
delayed close table, 211
delayed worker threads, 167
deleting
  delayed delete operations, 270
  executive object delete method, 134
  executive objects, 141
  key control blocks, 208
demand zero pages, 384, 440, 442, 469–472
demand-paging algorithm, 448, 458
dependencies
  binding, 838
  service, 226
Dependency Walker tool (Depends.exe)
  viewing image subsystem type, 54
  viewing Ntoskrnl.exe and HAL image dependencies, 69
dereference segment thread, 377
design goals
  NTFS, 717–719
  Windows, 35
  Windows I/O system, 537
desired access rights, executive object, 140
Desktop executive object type, 128
desktops
  process creation, 301
  Winlogon, 528
DESX encryption algorithm, 775
Device directory, 551, 552–553
Device Driver Kit (DDK)
  blue screen stop codes, 848
  executive functions, 63
  PCR and PRCB structures, 94
  tools, 34
device drivers, 69–73, 541–560. *See also* services
  APCs, 108
  auto-start, 225
  boot options, 262, 263
  checked build versions, 50
  control of device power state, 613
  crashes, 845
  disk drivers. *See* disk drivers
  driver objects and device objects, 550–555
  driver power operation, 610–611
  Driver Verifier, 99, 860. *See also* Driver Verifier
  executive functions, 63
  graphics, 56
  hung or unresponsive systems, 866
  initialization, 268
  installation files. *See* INF files
  interrupt objects, 103
  I/O requests to layered, 577–584
  I/O system, 538
  kernel mode, 39
  layered, 543–545
  levels of Plug and Play support, 591
  loading, in safe mode, 275–277
  locking memory, 385
  mapping System process system threads to, 79
  object access, 135
  opening devices using file objects, 555–560
  Plug and Play loading, initialization, and installation of, 594–602
  Plug and Play support, 592–593
  querying for Windows versions, 49
  Recovery Console, 280
  registry, 185, 189, 213–215
  security, 17
  spinlocks, 153
  starting, 224, 225
  structure, 548–550
  synchronous I/O requests to single-layered, 569–576
  types, 69–70, 541–548
  uninstalling, 848
  user-mode Plug and Play manager and installation of, 603–606
  versions of, 859
  viewing dispatch routines, 566–567
  viewing installed, 72
  viewing loaded driver list, 546–548
  Windows Driver Model (WDM), 71, 542
  WMI, 248
device enumeration, Plug and Play, 595–598
device ID and device instance ID (DIID), 600
device interrupt request level (DIRQL), 549
device I/O control operations, 569
device IRQLs, 96
Device Manager
  determining which HAL is running, 68
  disabling drivers, 285
  registry, 189
  viewing device tree, 596
  viewing devnode information, 601
  viewing driver power mappings, 610
  viewing multiprocessor-specific support files on Windows 2000, 43
  viewing resource assignments, 101
device objects
  device drivers, 550–555
  file system driver, 695
  viewing Microsoft Shadow Copy Provider, 650
  viewing shadow volumes, 652
  viewing System Restore filter, 709
  viewing TCP/IP, 822
device power states
  driver control, 613
  mapping system power states to, 609–611
  power manager, 607–608
device stacks, 577
device tree, 596–598
DeviceIoControl function
  change journal, 724
  data compression, 723, 748
  link tracking, 726
  NetBIOS, 813
  reparse points, 723
  system components, 63
DeviceMap structure, 147

www.TopSage.com

devices. *See also* device drivers; Device Manager
levels of Plug and Play support, 591
objects. *See* device objects
opening, using file objects, 555–560
power states. *See* device power states
device-specific modules (DSMs), 619
devnodes
device objects, 599
device tree, 596
driver loading, 600–602
dial-up remote access, 839
digital signatures, device driver, 539
direct I/O, 568
direct memory access (DMA), 568, 589, 678–679
directories
??, 147, 551, 559, 621, 701
Device, 551, 552–553
device object, 621
DosDevices, 147
encryption, 727, 775
Global??, 551, 559, 621
initialization, 267
NTFS change logging, 724
NTFS extensions, 739
NTFS root, 738
object directory objects, 146
page, 428–430
shadow copies for shared, 652
standard object, 144
system, 630
System Restore, 708
WMI, 243, 248
Directory Service, Windows 2000, 488
dirty pages, 376, 686, 767
dirty sector arrays, hive, 209
DisconnectEx function, 794
discretionary access control, 23, 486, 487
discretionary access-control lists (DACLs)
assignment, 509–511
security descriptors, 506–507
Disk Defragmenter tool, 728
disk device objects, 620–621
disk drivers, 616–622. *See also* device drivers; disks
boot process, 256
class/port/miniport architecture, 617–619
disk device objects, 620–621
file system drivers, 543
iSCSI drivers, 618
layered drivers, 584
multipath I/O (MPIO) drivers, 619
Ntldr, 616
partition manager, 622
watching physical disk I/O, 620
Disk Management MMC snap-in
creating mirrored volumes, 636
DMIO driver, 631–632
formatting volumes, 691, 732
partitions and volumes, 625
recursive mount points, 641
spanned volumes, 633
Virtual Disk Service, 648
Disk Probe tool, 202
Diskmon tool, 620
Diskpart tool, 642, 648
DiskPerf (Disk Performance) driver, 632
Diskraid tool, 648
disks. *See also* storage management
basic, 622, 624–626
crash dump files and space on, 869
defined, 615
directories. *See* directories
Driver Verifier disk integrity verification, 590
drivers. *See* disk drivers
dynamic, 622, 626–632
LDM disk entries, 628
network drive letters, 225
recovery. *See* recovery, NTFS
volumes. *See* volume management; volumes
x86 and x64 preboot, 251–255
Disk.sys class driver, 617
dismount operations, file system driver, 696
dispatch code, 99
dispatch interrupts, 104–107
dispatch routines, device driver, 549, 566–567, 592
dispatcher, thread, 327
dispatcher database
multiprocessor thread scheduling, 357–359
thread scheduling, 338–340
dispatcher objects, 156–163
data structures, 161–163
executive objects vs., 126
kernel, 66
KTHREAD blocks, 315
signaled state, 158–160
waiting for, 156–157
dispatcher spinlocks, 358
DPC/dispatch level IRQL, 94, 856, 868
display driver boot options, 257, 260
display I/O, Windows subsystem and, 55
Distributed Component Object Model (DCOM) API, 814
distributed computing environment (DCE) standard, 798
Distributed File System (DFS), 843
Distributed Management Task Force (DMTF), 237
DLLs (dynamic link libraries)
authentication, 489
AuthZ, 516
DLL hell, 311, 388
subsystem, 39, 53–63. *See also* environment subsystems
Windows and, 6
DMA checking option, Driver Verifier, 589
DMAdmin tool, 631
Dmboot.sys, 631
Dmconfig.dll, 631
DMIO device driver volume manager, 631–632, 647
Dmload.sys, 631
dollar sign ($), 733
Domain Name System (DNS)
NetBIOS, 811
network name resolution, 820
domains
Active Directory, 489
local system account, 218
logons, 531
SIDs for, 495, 497
DosDevices directory, 147
DPC objects, 104. *See also* deferred procedure calls (DPCs)
DPC/dispatch-level IRQLs, 96, 99, 150
Dr. Watson tool, 115–117
drive letters, network, 225
drive-letter assignment, volume. *See* volume namespace
driver groups, 275

www.TopSage.com

driver objects
  device drivers, 550–555
  viewing registered fast I/O routines, 563
driver signing, 17, 605
Driver Verifier
  configuration and initialization, 409–411
  deadlock detection, 166, 866
  device driver bugs, 17, 99
  disabling large pages using, 384
  force IRQL checking option, 412
  I/O-related options, 589–590
  low resources simulation option, 413
  pool tracking option, 406, 412
  special pool option, 411–412, 861–863
  system memory pools, 409–413
  verification levels, 860
Driver Verifier Manager, 406, 409, 589
  Driver.cab file, 68
DriverEntry routine, 548
drivers. *See* device drivers; NDIS (Network Driver Interface Specification) drivers; protocol drivers
  Drivers.exe tool, 72
driver-signing policy, 605
Dumprep.exe tool, 853
DuplicateHandle function, 135
DVD-ROM file system formats, 691
dynamic disks, 622, 626–632. *See also* disks
  basic disks vs., 622
  DMIO driver volume manager, 631–632
  LDM and GPT or MBR-style partitioning, 630
  LDM database, 627–629
  Logical Disk Manager (LDM) partitioning, 626
  viewing LDM database, 628
dynamic link libraries (DLLs). *See* DLLs (dynamic link libraries)
dynamic WMI classes, 242
dynamic worker threads, 168

E

EAX register, 120
EBX register, 120
echo command, viewing data streams using, 720
edge-triggered system clock, 258
EDX register, 120
  EFSDump tool, 785
EfsReadFileRaw function, 784
EfsWriteFileRaw function, 784
elevated IRQLs, 151–154
Emergency Management Services (EMS) boot option, 261
emulation layers, NDIS, 833
Encrypted Data Recovery Agents security policy, 780
EncryptFile function, 727, 778
Encrypting File System (EFS), 775–785
  backing up encrypted files, 784
  decryption process, 783–784
  encryption process, 778–782
  file encryption key (FEK) algorithms, 775–778
  NTFS, 726
  security components and, 487, 490
  viewing information about encrypted files, 785
encryption, RPC, 801
EndOfJobTimeAction function, 369
enhanced I/O verification option, Driver Verifier, 589
EnterCriticalSection and LeaveCriticalSection functions, 160
enumeration, Plug and Play device, 595–598
enumeration keys, device, 602
enumeration-based loading, Plug and Play device driver, 594
environment subsystems, 53–63
  Ntdll.dll support library, 63
  OS/2 subsystem, 61–62
  POSIX subsystem, 60–61
  server processes, 38
  stability, 58
  viewing image subsystem type, 54
  viewing POSIX subsystem start, 61
  Windows subsystem, 55–57
epc instruction, 121
EPROCESS blocks
  crash dump files, 850
  executive object handle tables, 137
  object manager session namespace, 147
  process creation, 305
  processes, 289–296
  viewing, 299
errors. *See also* debugging; exceptions; troubleshooting
  access-denied, 195, 712
  buffer-overflow errors, 195, 713
  crashes and hardware, 846. *See also* crashes
  device driver error-logging routines, 550
  exception dispatching and reporting, 118–119
  NTFS recovery and file system, 770
  service startup, 229
  services, 214
  viewing root causes of, using Filemon tool, 713
ETHREAD blocks
  crash dump files, 850
  processes, 289
  threads, 308, 313–320
Event executive object type, 127
Event Log
  NTFS quota management, 725
  provider, 239, 244, 248
  security auditing, 524
  service errors, 229
  service failures, 228
Event Logger, security auditing and, 525
Event Manager, 237
event monitoring. *See* Windows Management Instrumentation (WMI)
  Event Tracing for Windows (ETW), 106, 175–182, 620
Event Viewer tool, 45, 237
events
  keyed, 160
  priority boosts after waiting for, 350
  TDI event callbacks, 823
  thread dispatching, 327
Everyone SID, 496
ExAcquireFastMutex and ExAcquireFastMutexUnsafe function, 163
ExAcquireResourceExclusiveLite function, 164
ExAcquireResourceSharedLite function, 164
ExAcquireSharedStarveExclusive function, 164
ExAcquireWaitForExclusive function, 164
ExAllocatePool function, 401

www.TopSage.com

ExTryToAcquireResourceExclusive-Lite function, 164
Excel, 716
exception dispatching, 109–119. *See also* exceptions
  structured exception handling, 109–112
  unhandled exceptions, 112–117
  viewing start addresses for threads, 113
  viewing unhandled exceptions log file, 117
  Windows Error Reporting, 118–119
exception handlers. *See also* exception dispatching
  frame-based, 110
  NTFS log file service, 766
exceptions. *See also* debugging; *errors*; *troubleshooting*
  dispatching. *See* exception dispatching
  exception port and user-mode, 111
  handlers. *See* exception handlers
  interrupts vs., 85
  unhandled, 112–117
exclusion, mutual, 149–151. *See also* mutexes
ExCreateHandle function, 493
ExecNotificationQuery function, 246
executable images. *See also* applications; programs
  changing processor affinity for, 364
  enabling tracing for, 170
  opening, for processes, 302–304
  processes, 6
  section objects, 453
  service, 212, 214
  Software Restriction Policies, 534
  subsystem types, 53
execution protection, 389, 390–392
  execution states, thread, 334–338
executive
  functions, 63–65
  interlocked operations, 154
  kernel mode, 39
  memory manager, 376
  mutexes, 163
  NTFS file system driver, 729
  objects. *See* executive objects
  process blocks. *See* EPROCESS blocks
  process creation, 300
  resources, 156, 164–165
  subsystems, 55, 266–269
  system services, 5, 63
  thread blocks. *See* ETHREAD blocks
executive objects. *See also* object manager
  executive process objects, 304–308
  executive thread objects, 322
  headers and bodies, 129–130
  kernel objects vs., 126
  methods, 133–135
  names, 143–146
  object handles and process handle tables, 135–140
  resource accounting, 143
  retention, 141–142
  security, 140
  structure, 128
  type objects, 130–133
  types, 126–128
executive process blocks. *See* EPROCESS blocks
executive thread blocks. *See* ETHREAD blocks
Exetype tool, 54
ExInitializeNPagedLookasideList and ExInitializePagedLookasideList functions, 408
ExInterlockedPopEntryList, ExInterlockedPushEntryList, ExInterlockedPushEntryList, and ExInterlockedPushEntryList functions, 154
ExitThread function, 347
ExitWindowsEx function, 232, 286, 711
ExpInitializeExecutive function, 266
explicit file I/O, 701–704
explicit loading, Plug and Play device driver, 594
Explorer.exe
  creating processes using restricted tokens, 504–505
  encrypting files using, 775; 778
  Previous Versions Client extension, 653
  as shell, 82. *See also* shell
  user logon, 533
  viewing catalog files, 605
express queue, cache manager, 688
ExpWorkerInitialization function, 167
ExpWorkerThreadBalanceManager function, 168
ExQueueWorkItem function, 166
extended model, WMI CIM, 241
extended partitions, 252, 624
extensibility
  Windows, 35
  Winsock, 792
Extensible Firmware Interface (EFI) specification, 264, 624
Extensible Storage Engine (ESE) database, 841
extensions
  NTFS directory, 739
  TCP/IP protocol, 824–28
  Winsock, 792, 793

F

F2 key, 282
F8 key, 261, 263, 274, 275, 285, 848, 868
F10 key, 279
facility management, trusted, 487
failures, service, 215, 231
fast I/O, 562–563, 658, 679–681, 704
fast LPC, 57
fast mutexes, 163
fast user switching, 21
FAT12, FAT16, and FAT32 file system formats
  bad-cluster recovery, 771
  boot sectors, 255
  cluster sizes, 691–694
  file allocation table, 692
  write-through algorithm, 760
fault handling, page. *See* page fault handling
fault tolerance, NTFS
  data recovery, 774
  design requirement, 718
  volume management, 623
Federal Information Processing Standard (FIPS) 151
fibers, threads vs., 13
Fibre Channel adapters, 617
file allocation table (FAT), 692. *See also* FAT12, FAT16, and FAT32 file system formats
File And Printer Sharing For Microsoft Networks Properties dialog box, 662

www.TopSage.com

file caching, stream caching vs., 658. *See also* cache manager; system cache
file change notifications, Filemon tool and, 712
file control blocks (FCBs), NTFS, 731
file encryption key (FEK), EFS
  caching decrypted, 783
  encrypting, 776
  encrypting files using, 775
File executive object type, 127
file I/O, 682–688, 700–705. *See also* file systems; I/O system
  cache manager's lazy writer and read-ahead threads, 704
  components and paths, 700
  explicit, 701–704
  intelligent read-ahead, 682–683
  memory manager's modified and mapped page writer threads, 704
  memory manager's page fault handler, 705
  mirrored volume, 636
  multipath (MPIO) drivers, 619
  system worker threads, 687
  volume, 646–647
  write throttling, 686–687
  write-through caching and lazy writing, 683–685
file logging, Event Tracing for Windows, 176
file mapping objects, 13. *See also* section objects
file objects, 555–560
file records, NTFS, 733, 740–742
file references, NTFS, 739
  File Replication Service (FRS), 843
file server driver (Srv.sys), 79
file system control codes. *See* DeviceIoControl function
file system drivers (FSDs), 694–711. *See also* formats, file system
  cache manager and. *See* file system interfaces, cache manager
  device drivers, 70
  disk drivers, 543
  file system filter drivers, 705–711
  file system operation, 700–705
  kernel mode, 542
  layered drivers, 584
  local, 695
  named pipe and mailslot, 807
  NTFS, 729–731
  remote, 696
  viewing list of registered file systems, 698
  volume mounting, 642–646
file system filter drivers, 705–711
  device drivers, 70
  Filemon tool, 706
  System Restore, 706–711
  virus scanners, 705
file system interfaces, cache manager, 674–679
  copying, 676
  direct memory access (DMA), 678–679
  mapping and pinning, 677–678
File System Recognizer (Fs_rec.sys), 645
file systems, 689–785. *See also* file I/O
  boot process, 254, 264
  boot sectors, 255
  file system drivers (FSDs), 694–711
  file system formats, 690–694
  file system namespace, 134
  MS-DOS volume management, 623
  NTFS. *See* NTFS file system
  partitions, 624
  recoverable, 658–659
  troubleshooting problems using Filemon tool, 711–717
FileEncryptionStatus function, 727
Filemon tool
  basic and advanced modes, 711
  as file system filter driver, 706
  named pipes and mailslots, 810
  troubleshooting file system problems using, 711–717
  troubleshooting techniques, 712–717
  viewing file system activity on idle systems, 712
  viewing recursive mount points, 641
  viewing root causes of errors using, 713
  watching prefetch file reads and writes, 461
files. *See also* file systems
  backing up encrypted, 784
  binary MOF (BMF), 243
  catalog, 605
  common system, in Windows versions, 48
  crash dump, 849–851
  delayed rename and delete operations, 270
  disabling lazy writing for temporary, 684
  EFS decryption process, 783–784
  EFS encryption process, 778–782
  errors accessing, 712
  flushing mapped, 684
  hardware abstraction layer, 67
  hive, 209–210
  IA64 boot process, 265
  INF, 539, 603, 604
  mapped. *See* mapped files
  mutiprocessor and uniprocessor versions of, 42
  network binding and INF, 838
  NTFS change journal, 752
  NTFS data compression and sparse, 723
  NTFS filenames, 742–744
  NTFS master file table, 733–739
  NTFS sparse, 752
  page. *See* page files
  prefetch, 460, 461
  privileges, 520
  registry. *See* hives, registry
  registry and filename associations, 189
  Software Restriction Policies, 534
  symbolic link objects, 146
  system file corruption problem, 282–283
  transmitting, using Winsock, 794
  viewing EFS information about encrypted, 785
  viewing multiprocessor-specific, on Windows 2000, 43
  virtual I/O, 540
  Windows system components, 39
  x86 and x64 boot process, 252
Filesystem Filter Manager, 706
filter device objects (FiDOs), 599, 626, 709
filter drivers
  device drivers, 71
  file system. *See* file system filter drivers
  layered drivers, 584
  Plug and Play support, 592
  WDM, 543
filter hook driver, TCP/IP and, 826
FindFirstChangeNotification function, 712, 724

www.TopSage.com

FindNextChangeNotification functions, 712
firewall hook drivers, 826
firmware, operating system in, 624
first in, first out (FIFO) replacement policy, 462
fixed disks. *See* disks
fixed quantum, 344
flags, Windows global, 125, 168–171
FlushFileBuffers function, 684
flushing, cache. *See* write-through caching
FlushViewOfFile function, 384, 450
folders. *See* directories
forcing write-through caching, 684
foreground processes, quantum boosting and, 343, 344
foreground threads, priority boosts for, after waits, 351–353
fork function, POSIX, 394
format command, 691, 732
Format tool, 771
formats, file system, 690–694. *See also* file system drivers (FSDs)
  CDFS, 690
  defined, 689
  FAT12, FAT16, and FAT32, 691–694
  NTFS, 694. *See also* NTFS file system
  UDF, 691
Fpswa.efi, 265
frame-based exception handlers, 110
Framework Class Library (FCL), 4
free checking, heap manager, 398
free kernel environment, 259
free pages, 381, 440, 469, 472
front-end heap layer, 395
Fsutil.exe tool, 739
FtDisk tool
  bad-cluster recovery, 771
  basic disks, 625
FTP protocol, 803
function codes
  driver object, 553
  IRP, 566
function drivers
  device drivers, 71
  devnodes, 599
  Plug and Play support, 592
  WDM, 543
functional device objects (FDOs), 599
functions. *See also* routines; services
  Address Windowing Extensions, 15, 399–400
  API. *See* APIs (application programming interfaces)
  cache manager copy interface, 676
  cache manager DMA interface, 679
  cache manager mapping and pinning interface, 677
  EFS, 785
  executive function name prefixes, 74
  explicit file I/O with, 701–704
  heap, 394
  job-related, 369
  memory manager, 382
  NUMA-related, 362
  process-related, 298–300
  registry, 191
  thread-related, 322, 330, 341, 345
  trap handler, 85
  Windows, 5. *See also* Windows API

G

Gdi32.dll, 39, 56
GdiFlush function, 57
GdiSetBatchLimit function, 57
Generic Packet Classifier (GPC) driver, 837
generic services, executive object, 129
getaddrinfo function, 792, 794
GetCompressedFileSize function, 747
GetDriveType function, 225
GetEffectiveRightsFromAcl function, 511
GetFileAttributes function, 723
gethostbyname function, 792
getnameinfo function, 794
GetProcessHeap function, 395
GetQueuedCompletionStatus function, 370, 562, 586, 587, 800
GetSystemInfo function, 385
GetSystemMetrics function, 277
GetSystemTimeAdjustment function, 340
GetThreadContext function, 12
GetVolumeInformation function, 747
Gflags tool
  enabling pool tagging, 404
  enabling special pool, 861
  heap debugging options, 398
  viewing and changing system global flags, 169–171
GINA (Graphical Identification and Authentication). *See also* Winlogon
  interactive logon, 527
  security, 490
  system processes, 81
global flags, Windows, 125, 168–171
global object manager namespace, 147
global queued spinlocks, 154
Global?? directory, 551, 559, 621
globally unique identifiers. *See* GUIDs (globally unique identifiers)
Gopher protocol, 803
granted access rights, executive object, 141
granted-access masks, 511
granularity, allocation, 385
Graphical Identification and Authentication. *See* GINA (Graphical Identification and Authentication)
graphical user interface. *See* GUI (graphical user interface)
Graphics Device Interface (GDI), 56
group policies
  FRS, 843
  IPSec, 827
  online crash analysis, 854
Group Policy Objects (GPOs), 827
groups
  associated IRPs, 582
  disk, 626
  process. *See* job objects
  registry, 190
  service, 223
  SIDs, 498
guarded mutexes, 163
GUI (graphical user interface)
  functions, 39
  security editors warning, 514
  threads and priority boosts, 353–354
GUID Partition Table (GPT) partitioning
  basic disks, 624
  dynamic disks, 630
Guidgen tool, 551
GUIDs (globally unique identifiers)
  LDM, 627
  partitioning, 624, 630
  Plug and Play drivers, 551

www.TopSage.com

H

HAL (hardware abstraction layer)
  boot options, 257, 258, 259
  boot process, 252, 265
  determining which, is running, 68
  device drivers, 70
  device enumeration, 595
  files, 39
  Halchk.dll, 259
  Hal.dll, 39, 43, 252, 259, 265
  Halmps.dll, 258, 260, 262
  initialization, 266
  I/O system, 539
  IRQLs, 95
  kernel mode, 39
  multiprocessor and uniprocessor versions, 42
  portability, 40
  synchronization and uniprocessor, 153
  system space, 414
  versions, 67–69
  versions of Windows, 543
  viewing base versions included with Windows, 68
  viewing Ntoskrnl.exe image dependencies using, 69
HalGetSystemInterruptVector function, 95
HalInitializeProcessor function, 266
HalInitSystem function, 266, 267
Handle tool
  viewing logon session access tokens, 533
  viewing open object handles, 136
  viewing section objects, 451
handles
  EPROCESS block, 294
  executive object, 135–140
  NTFS file system driver and object, 731
  objects, 22
  processes, 6
  section objects, 387
  viewing, using Process Explorer, 11
  viewing file objects, 558
  viewing hive, 199
  viewing open, 124
hang problems, post-splash screen, *284–286*
hard disks. *See* disks; storage management
hard links, NTFS, 722–723
hard links, POSIX, 743
hard page faults, 474
hard partitions, 630
hard real-time systems, 102
hard working set limits, 463
hardware
  configuration. *See* Plug and Play (PnP) manager
  crash buttons, 867
  crash dump files and failure of, 870
  crashes and errors of, 846
  detection, 252, 258, 265
  device drivers, 70
  device keys, 602
  IDs, 604
  interrupts. *See* hardware interrupts
  I/O. *See* I/O system
  kernel support, 66
  latency and system power states, 607
  memory protection, 389
  no execute page protection, 390
  page table entries, 432
  portability, 40
  profiles, 190, 191
  registry, 189
  SAN, 797
  VDS providers, 649
hardware abstraction layer. *See* HAL (hardware abstraction layer)
Hardware Installation Wizard, 603
hardware interrupts
  exceptions vs., 86. *See also* exception dispatching
  IA64 interrupt controllers, 90
  processing, 87–90. *See also* interrupt dispatching
  viewing interrupt dispatch tables (IDTs), 88
  viewing PIC and APIC configuration, 90
  x64 interrupt controllers, 90
  x86 interrupt controllers, 89
Hardware Profiles dialog box, 191
hashed usernames and passwords, 530
HasOverlappedIoCompleted function, 562
header files, SDK and DDK, 33
headers
  dispatcher, 161
  executive object, 129–130
  object, 131–132
heap manager, 394–399
  debugging features, 398
  functions, 394
  look-aside lists, 396
  Low Fragmentation Heap (LFH), 397
  pageheap debugging tool, 398
  structure, 395
  synchronization, 396
  types of heaps, 395
heaps
  functions, 382
  system space, 414
  types of, 395
HeapSetInformation function, 397
Helper libraries, Winsock, 795
hibernated systems, booting, 256
hibernation file (Hiberfil.sys), 256, 608
Hierarchical Storage Management (HSM) systems, 640, 723
high and low memory notification, 479–480
high IRQL fault analysis, 856–858
High priority class, process, 301
high-IRQL synchronization, 151–154
  executive interlocked operations, 154
  instack queued spinlocks, 154
  interlocked operations, 151
  queued spinlocks, 153–154
  spinlocks, 152–153
  viewing global queued spinlocks, 154
hives, registry. *See also* registry
  cell maps, 203–207
  manually loading and unloading, using Registry Editor tools, 198
  registry keys, 197–198
  registry namespace and operation, 207–209
  size limits, 198–199
  stable storage for, 209–210
  structure, 200–203
  viewing handles, 199
  viewing memory usage, 205
  viewing paged pool usage, 204
HKEY_CLASSES_ROOT (HKCR) registry key, 186, 189
HKEY_CURRENT_CONFIG (HKCC) registry key, 186, 190
HKEY_CURRENT_USER (HKCU) registry key, 186, 187

www.TopSage.com

HKEY_LOCAL_MACHINE(HKLM) registry key, 186, 189–190
HKEY_PERFORMANCE_DATA (HKPD) registry key, 186, 191
HKEY_USERS (HKU) registry key, 186, 188
HKLM(HKEY_LOCAL_MACHINE) registry keys, 24
hot fixes, symbol files and, 28
HTTP protocol, 803
hung systems, crash dump analysis and, 866–869
hung-program dialog box, 287
hypercritical worker threads, 167
hyperspace, 305, 414
hyperthreading, 41, 360, 366

I

i8042 port driver, 867
I8042KeyboardInterruptService function, 100
i82489 Advanced Programmable Interrupt Controller (APIC), 89–90
i8259A Programmable Interrupt Controller (PIC), 89–90
IA64 systems
  address space layout, 422
  address translation, 437
  boot process, 264–265
  interrupt controllers, 90
  IRQLs, 91
  page sizes, 383
  partitioning, 624
  system service dispatching, 121
Ia64ldr.efi, 265
icons, base named object, 145
ideal processors, 366, 367
Idle process, 21, 75, 267
idle processors, 367–368
idle summary mask, 358
idle systems, 193, 712
idle threads, 348
IEEE 1394 channel boot option, 257
image database, service control manager, 227
image subsystem types, 54
Imagecfg tool
  disabling look-aside lists, 397
  setting processor affinity, 363
images, executable. *See* executable images
ImpersonateNamedPipeClient function, 502
ImpersonateSecurityContext function, 503
ImpersonateSelf function, 502
impersonation
  access checks, 502–504
  access tokens, 499, 502
  EFS file encryption, 778
  named pipe, 806
  object security, 492
  RPC, 802
Increase Scheduling Priority privilege, 301, 333
indexing
  hive cell, 201, 202
  NTFS, 721, 746, 753–754
  page mapping, 426
INF files
  Hardware Installation Wizard, 603
  I/O system, 539
  network binding, 838
  viewing, 604
inheritance, access control entry (ACE), 507, 511
initialization
  device driver routine, 548
  Driver Verifier, 409–411
  kernel and executive subsystems, 266–269
  Plug and Play and device driver, 594–602
  process, 310
  services, 213
  thread, 336
  Winlogon, 528–529
initiator component and service, Microsoft iSCSI Software Initiator, 618
in-paging I/O, 443
instack queued spinlocks, 154
  Installable File System (IFS), 63, 655, 695
installation. *See also* setup programs
  multiprocessor and uniprocessor file versions, 42
  Plug and Play and device driver, 594–602
  registry and device driver, 185
  user-mode Plug and Play manager and device driver, 603–606
  viewing installed device drivers, 72
  viewing installed services, 83
  Windows. *See* Setup program, Windows
instance ID, 600
Instant Messaging (IM), 814
Integrated Device Electronics (IDE), 617
Intel 64-bit Extension Technology for x86, 40
Intel Itanium IA-64 systems, 40
intelligent read-ahead, cache manager, 658, 682–683
interactive logon. *See* logon
interactive services, 221–223, 228
Interactive Session Locking and Trusted Path, security rating and, 488
Interface Definition Language (IDL), RPC and, 800
interfaces
  cache manager file system. *See* file system interfaces, cache manager
  examining undocumented, 73–74
  kernel hardware support, 66
  TDI (Transport Driver Interface), 789, 823–824
  VDS, 649
interlocked operations, 151, 154, 396
intermediate drivers, NDIS, 832
Internal Data Transfer Protection, security rating and, 488
Internal Replication, security rating and, 488
Internet APIs, 803–804
Internet Connection Firewall (ICF), 826
Internet Connection Sharing (ICS), 825
Internet Explorer, 196
Internet Key Exchange (IKE) protocol, 828
Internet names, DNS, 820
Internet Protocol security (IPSec), TCP/IP and, 826–828
Internet Storage Name Service (iSNS) protocol, 618
inter-processor interrupt IRQL, 96
interprocessor interrupts (IPIs), 95
interrupt dispatch tables (IDTs), 87, 88
interrupt dispatching, 87–109
  hardware interrupt processing, 87–90. *See also* hardware interrupts
  profiling execution using Kernel Profiler, 96–98

www.TopSage.com

real-time processing, 102
software interrupt request levels (IRQLs), 91–103
software interrupts, 103–109
viewing interrupt dispatch tables (IDTs), 88
viewing PIC and APIC configuration, 90
interrupt objects, 66, 99–103
interrupt request levels (IRQLs), 91–103
crash dump analysis, 856–858
crashes, 845
Driver Verifier checking option, 412
high-IRQL synchronization, 151–154
interrupt objects, 99–103
interrupt prioritization, 91–95
low-IRQL synchronization, 155–166
mapping interrupts to, 95
multiprocessor thread scheduling, 359
predefined, 96–99
priority levels vs., 334
profiling execution using Kernel Profiler, 96–98
real-time processing, 102
uniprocessor dispatcher database synchronization, 339
viewing, 94
viewing interrupt objects, 100
interrupt requests (IRQs), 87, 95
interrupt service routines (ISRs), 99, 102, 549
interrupts. *See also* interrupt dispatching
device driver DPC routines, 549, 571
exceptions, 85, 110
hardware. *See* hardware interrupts
high-IRQL synchronization, 151
initialization, 266, 267
interrupt affinities boot option, 258
mapping, to IRQLs, 95
software. *See* software interrupts
system processes, 76
invalid page table entries (PTEs), 440. *See also* page fault handling
invasive user-mode debugging, 29
I/O completion
device driver routines, 549
layered drivers, 582
ports. *See* I/O completion ports
priority boosts after, 349
single-layered drivers, 572–575
I/O completion ports
IoCompletion objects, 585
IRPs, 585–588
scalability, 46
I/O devices, hardware interrupts and, 87
I/O manager
device driver loading, 597
device drivers, 70
disk drivers, 617
executive, 64
executive object close method, 134
file system drivers, 694
file system namespace, 134
I/O system, 538, 539
initialization, 268
mount points, 640
safe mode boots, 276
volume mounting, 643
I/O operations. *See* file I/O
I/O request packets (IRPs), 564–590
buffer management, 568–569
cache manager, 656
cache manager fast I/O, 679
crash dump analysis, 866
Driver Verifier, 589–590
examining uncompleted, 579–581
explicit file I/O, 702
Filemon tool, 706
I/O completion ports, 585–588
I/O manager, 539
layered drivers, 577–584
look-aside lists, 564
single-layered drivers, 569–576
stack locations, 566–567
TDI, 823
viewing driver dispatch routines, 566–567
viewing outstanding thread, 567
volume management, 622, 642, 646
I/O system, 537–613
components, 537–541
device drivers. *See* device drivers
ETHREAD blocks, 314
fast I/O, 562–563
file I/O. *See* file I/O
I/O manager, 539. *See also* I/O manager
I/O request packets, 564–590. *See also* I/O request packets (IRPs)
mapped file I/O and file caching, 564
Plug and Play (PnP) manager, 590–606
power manager, 607–613
scatter/gather I/O, 564
synchronous and asynchronous I/O, 561–562
types of I/O, 561–564
typical I/O processing, 540
I/O verification option, Driver Verifier, 589
IoAsynchronousPageWrite function, 704
IoAttachDeviceToDeviceStackSafe function, 706
IoCallDriver function, 570, 702
IoCompleteRequest function, 349, 570, 572
IoCompletion executive object type, 128
IoCompletion objects, 585
IoConnectInterrupt and IoDisconnectInterrupt functions, 103
IoCreateDevice and IoCreateDeviceSecure functions, 550
IoPageRead function, 703
IopBootLog function, 278
IopCallDriverAddDevice function, 276
IopCopyBootLogRegistryToFile function, 278
IopInvalidDeviceRequest function, 566
IopLoadDriver function, 276, 278
IopParseDevice function, 701
IopSafeBootDriverLoad function, 276
IoQueueWorkItem function, 166
IoReadPartitionTable and IoReadParititionTableEx functions, 620
IoRegisterDeviceInterface function, 551
IoRegisterFileSystem function, 643
IoSetDeviceInterfaceState function, 551
IP filtering, TCP/IP and, 825, 826

www.TopSage.com

IP Security Monitor MMC snap-in, 828
IP Security Policy Management MMC snap-in, 828
IPSec driver, 828
IPSec Policy Agent, 827
IPSEC service, Windows 2000, 488
IPv4 protocol, 792, 821
IPv6 protocol, 792, 821
IPX/SPX protocol, 813, 821
iretd instruction, 120
IRPs. *See* I/O request packets (IRPs)
IRQLs. *See* interrupt request levels (IRQLs)
iSCSI drivers, 618
Iscsicli.exe, 618
Iscsiexe.exe, 618
Iscsiprt.sys, 618
Iscsium.dll, 618
ISO-9660 format, 690
ISO-13346 format, 691

J

Job executive object type, 127
job objects
  features, 369–374
  job-scheduling classes, 370
  job-wide affinity masks, 363
  processes, threads, and, 6–14. *See also* threads; processes
  thread quantums, 342
Joliet disk format, 690
journaling. *See* logging, NTFS
Junction tool, 723
junctions, NTFS, 722–723

K

Kd tool, 29, 32, 857. *See also* kernel debugger
KeAcquireGuardedMutex function, 164
KeAcquireInStackQueuedSpinlock and KeReleaseInStackQueuedSpinlock functions, 154
KeAcquireInterruptSpinLock function, 576
KeAcquireQueuedSpinlock function, 153
KeAcquireSpinlock and KeReleaseSpinlock functions, 153
KeActiveProcessors mask, 358
KeAddSystemServiceTable function, 122
KeBalanceSetManager, 466
KeBugCheckEx function, 87, 96, 846, 846–849, 852, 867
KeEnterCriticalRegion function, 108, 163
KeEnterGuardedRegion function, 108, 164
KeEnterGuardedRegionThread function, 108
KeInitializeQueue function, 587
KeInitThread function, 309
KeInsertQueue function, 587
KeLowerIrql function, 93
KeNodeBlock variable, 361
KeRaiseIrql function, 93
Kerberos authentication
  Active Directory, 841
  package, 530
  RPC, 802
KeRegisterBugCheckCallback function, 847
KeRegisterBugCheckReasonCallback function, 847
KeRemoveQueue function, 587
kernel
  boot process, 252, 259, 260, 261, 265
  debugging, 28–33. *See also* kernel debugger
  dispatcher objects, 156–163
  dispatcher objects vs. executive objects, 126
  event tracing, 175–182
  exception dispatching. *See* exception dispatching
  extensions, 542
  handle table, 139
  hardware support, 66
  initializing executive subsystems, 266–269
  kernel memory dump crash dump files, 849
  kernel mode, 39. *See also* kernel mode
  logger, 175–182
  multiprocessor and uniprocessor versions, 42
  Ntoskrnl.exe kernel image, 28, 65–67. *See also* Ntoskrnl.exe
  objects, 66
  PAE support, 436
  portability, 40
  process blocks. *See* KPROCESS blocks
  safe mode boots, 276
  service descriptor table, 122
  software interrupts. *See* software interrupts
  streaming filter drivers, 70
  synchronization mechanisms, 155
  thread blocks. *See* KTHREAD blocks
  thread scheduling code, 327
  variables. *See* kernel variables
  Windows Management Instrumentation (WMI), 176
kernel debugger, 28–33
  accounting for physical memory use, 380–381
  basic crash dump analysis, 856–858
  boot options, 257, 260
  checked builds, 50
  debugger extension commands, 30
  determining maximum memory pool sizes, 402–404
  displaying driver objects and device objects, 554
  displaying type information for kernel structures, 31–32
  dumping device tree, 598
  enabling image loader tracing and viewing NtGlobalFlag, 170
  examining IRPs, 579–581
  examining page directories and page directory entries, 429
  examining page table entries (PTEs), 431
  examining PEB, 296
  generating memory dumps, 851
  heap debugging options, 398
  identifying system threads in System process, 77
  Kd too, 29, 32, 857
  listing acquired executive resources, 164
  listing loaded NDIS miniports, 830–832
  listing system worker threads, 168
  LiveKd tool, 32. *See also* LiveKd tool
  manual crash dump analysis, 864–870
  monitoring pool usage, 406
  pool tracking, 412

www.TopSage.com

SoftICE tool, 33
symbols, 28
translating addresses, 433–434
verbose crash dump analysis, 858–859
viewing access tokens, 500–502
viewing cache virtual size, 664
viewing cache working set, 666
viewing control areas, 454–457
viewing Device directory, 552–553
viewing device stacks, 577
viewing driver dispatch routines, 566–567
viewing EPROCESS blocks, 291–292, 299
viewing ETHREAD blocks, 314
viewing file object data structure, 556
viewing file objects and device handles, 558
viewing global queued spinlocks, 154
viewing handle tables, 140
viewing hive memory usage, 205
viewing hive paged pool usage, 204
viewing hyperthreaded systems, 360
viewing interrupt dispatch tables (IDTs), 88
viewing interrupt objects, 100
viewing interrupt request levels (IRQLs), 94
viewing job objects, 371–373
viewing key control blocks, 208
viewing loaded driver list, 546–548
viewing look-aside lists, 408
viewing LPC port objects, 172–174
viewing Microsoft Shadow Copy Provider filter device objects, 650
viewing nonuniform memory access (NUMA) systems, 361
viewing object headers, 131–132
viewing open handles using, 124
viewing outstanding thread IRPs, 567
viewing page frame number (PFN) database, 472
viewing PFN entries, 478
viewing PIC and APIC configuration, 90
viewing PRCB structures, 359
viewing quota structure, 143
viewing ready threads, 326
viewing registered fast I/O routines, 563
viewing security descriptors, 508–509
viewing session space usage, 421
viewing sessions, 419
viewing shared and private cache maps, 673–674
viewing system memory information, 378
viewing system power capabilities and policy, 611–612
viewing System Restore filter device objects, 709
viewing TCP/IP device objects, 822
viewing TEB, 320
viewing thread activity, 325
viewing thread data structures, 316–318
viewing trap frames, 86
viewing virtual address descriptors (VADs), 449
viewing volume parameter blocks (VPBs), 643
viewing wait queues, 162
viewing working set list, 465
viewing write-throttle parameters, 687
Windows Debugging Tools, 29–32
kernel mode
APCs, 108
device drivers, 542
exception dispatching, 111
memory pools, 388
objects, executive objects, device drivers, and, 135
preemptive thread-scheduling, 346
storage management, 615. *See also* storage management
support functions, 5
synchronization mechanisms, 155
system components, 39
system mechanisms. *See* system mechanisms, Windows
system service dispatching, 121
user mode vs., 16–21. *See also* user mode
viewing activity using Performance tool, 19–21
viewing thread activity using QuickSlice tool, 18
Windows model, 36
kernel process blocks. *See* KPROCESS blocks
Kernel Profiler tool, 96–98
Kernel Security Device Driver (KSecDD), 490
kernel thread blocks. *See* KTHREAD blocks
kernel variables. *See also* system variables
maximum working set, 463
memory manager configuration, 378
NUMA system KNODE structures, 361
physical memory, 479
process-related, 297
session space, 419
system memory pool, 402
system space, 417
system working set, 467
thread-related, 320
thread-scheduling, 339
Kernel32.dll, 39, 43, 56, 300, 807
KeServiceDescriptorTable array, 122
KeServiceDescriptorTableShadow array, 122
KeSetEvent function, 353
KeSetEventBoostPriority function, 350
KeSwapProcessOrStack, 466
KeSynchronizeExecution function, 576
key cell data type, 201
key control blocks, 207–209, 211
Key executive object type, 128
key nodes, hive cell, 201
key object object type, 207
keyed events, 160
keys
device, 602
EFS, 779–780
managed object property, 244
registry, 185
registry. *See also* hives, registry
registry. *See also* root keys, registry
KiActivateWaiterQueue function, 588
KiChainedDispatch function, 99, 103
KiContextSwapLock spinlocks, 358
KiDispatcherLock spinlocks, 358
KiIdleSummary mask, 358

www.TopSage.com

KiInitializeKernel function, 266, 267
KiInterruptDispatch function, 99
KiInterruptTemplate, 99
KiSystemService (system service dispatcher), 121
KiSystemStartup function, 266
KiThreadStartup function, 309, 310
KiUnwaitThread function, 351, 588
KiWaitTest function, 156
KNODE structures, 361
Knowledge Base
  blue screen stop codes, 848
  kernel debugging, 32
KPROCESS blocks
  creating, 306
  EPROCESS blocks, 292, 294
  page directory, 428
  quantum, 340
KTHREAD blocks
  APCs, 108
  context switching, 348
  creating, 309
  ETHREAD blocks, 314
  ideal and last processors, 366
  structure, 314–319

L

LAN adapter (LANA) numbers, NetBIOS and, 811
LAN Manager 2 authentication protocol, 530
language editions, Windows, 25
LANMan Redirector, 697
LANMan Server, 697
laptops, 622
large pages, 382–384
large-IRP look-aside list, 564
last known good (LKG) configurations
  boot option, 259
  control set in registry, 190
  crash dump analysis, 848
  overriding successful boot definitions, 272
  post-splash screen crashes, 284
  rebooting services, 214
  recovery, 274
  services, 230–231
last processors, 366, 368
layered device drivers
  components, 543–545
  I/O requests to, 577–584
layered network services. *See* network services
lazy commit algorithm, NTFS, 770
lazy evaluation, 393, 448
lazy IRQL, 93
lazy write operations
  cache manager, 683–685
  file systems, 760
  hive synchronization, 209
  threads and file I/O, 704
LDAP C API, 840
LDMDump tool, 628
LdrInitializeThunk function, 311
leaks, memory
  memory optimizers, 482
  troubleshooting pool, 407
least recently used (LRU) replacement policy, 462
level-sensitive system clock, 258
Lightweight Directory Access Protocol (LDAP) directory services, 840
lightweight threads, fibers as, 13
link objects, symbolic, 146
link tracking, NTFS, 726
links, registry, 186
listen TDI IRP, 823
LiveKd tool. *See also* kernel debugger
  creating junctions, 723
  features, 32
  generating memory dumps, 851
  identifying system threads in, 77
  viewing ready threads, 326
  viewing thread activity, 325
ln POSIX function, 722
Load and unload device drivers privilege, 524
load balancing, network, 842
load driver security privilege, 228
loaded driver list, 546–548
loading, Plug and Play device driver, 594–602
LoadUserProfile function, 228, 778, 783
local accounts, 190
local administrators group, 218–219
local file system drivers (FSDs), 695
local kernel debugging, 30, 32
local object manager namespace, 147
local procedure calls (LPCs)
  EPROCESS blocks, 293
  ETHREAD blocks, 314
  exceptions, 112
  fast LPC, 57
  initialization, 268
  Kernel Security Device Driver and, 490
  LPC facility and executive, 65
  RPC, 802
  as system mechanism, 171–174
local RPC, 802
Local Security Authority (LSA), 517
Local Security Authority Subsystem (Lsass)
  Active Directory, 841
  boot process, 252, 265
  EFS, 777
  initialization, 271
  local procedure calls, 171
  policy database, 489
  security, 489
  security auditing, 525
  security reference monitor (SRM), 491
  service control manager, 224
  service logon, 227
  system processes, 81–82
  user logon, 531
local security policy, security auditing and, 524
Local Security Policy Editor
  assigning privileges, 516
  configuring Recovery Console security, 279
  security auditing, 525
  setting encryption algorithm, 775
  watching Software Restriction Policies enforcement, 535
local service account, 220
local system account, 196, 218–219
LockFile function, 559
locks
  crash dump analysis, 869
  memory, 385
  push, 165
log files. *See* logs
logging, NTFS, 761–766
  change journal, 724
  log file, 738
  log file service (LFS), 730, 762–763
  log record types, 763–766
logical block caching, 658
logical cluster numbers (LCNs), 733
Logical Disk Manager (LDM)
  bad-cluster recovery, 771
  device objects, 551
Logical Disk Manager (LDM) partitioning
  database, 627–629
  dynamic disks, 626, 630
logical drives, 624

www.TopSage.com

logical prefetcher, 65, 458–462. *See also* prefetching
logical sequence numbers (LSNs), 659, 762
logoff, logging activity during, 196
logon
  account rights, 517
  C2 security rating, 486
  components, 526
  dialog box. *See* GINA (Graphical Identification and Authentication)
  listing active sessions, 532–533
  logging activity during, 196
  Recovery Console, 280
  registry, 184
  secure attention sequence (SAS), 81, 487, 529
  security, 526–533
  services, 227, 230
  user logon steps, 529–533
  process, 528–529. *See also* Winlogon
LogonSessions tool, 532–533
LogonUser function, 496, 498, 503
logs
  boot. *See* boot log files
  Dr. Watson tool, 115–117
  NTFS, 738. *See also* logging, NTFS
  recoverable file system, 659
  registry log hives, 209–210
  Regmon, 196
  System Restore tracking, 707
long mode, 263
long quantum, 341, 344
look-aside lists, 395, 396, 408–409, 564
low and high memory notification, 479–480
Low Fragmentation Heap (LFH), 395, 397
low resources simulation option, Driver Verifier, 413
lower-level filter drivers, 599
low-IRQL synchronization, 155–166
  deadlock detection using Driver Verifier, 166
  executive resources, 156, 164–165
  fast mutexes and guarded mutexes, 163
  kernel dispatcher objects, 156–163
  keyed events and critical sections, 160
  listing acquired executive resources, 164
  mechanisms, 155
  push locks, 165
  viewing wait queues, 162
LsaAddAccountRights, LsaRemoveAccountRights, and LsaEnumerateAccountRights functions, 517
LsaAuthenticationPort, 528
LsaEnumerateAccountRights function, 518
LsaEnumerateLogonSessions function, 532
LsaLogonUser function, 227, 517, 529
LsaLookupAuthenticationPackage function, 530
LsaRegisterLogonProcess function, 528
LsaStorePrivateData function, 227

M

machine security identifier (SID), 218
mailslots
  implementation, 807–808
  named pipes vs., 804
  operation, 806
  viewing activity, 810
maintain objects list global flag, 125
major function codes, IRP, 566
Managed Object Format (MOF) language, 239, 240–243
managed objects, WMI, 238
management applications
  Microsoft iSCSI Software Initiator, 618
  WMI, 237
management mechanisms. *See* registry; services; Windows Management Instrumentation (WMI)
manifest files, 312
mapped files. *See also* section objects
  flushing, 684
  I/O, 450, 564
  modified page writer, 475
  pages in, 442
  shared memory, section objects, and, 386–388
mapped page writer, 376, 475
mapped views, system, 414
mapping. *See also* mapped files
  cache manager interface, 677–678
  hives into memory, 203, 206
  System process system threads to device drivers, 79
  virtual memory to physical memory, 14
MapUserPhysicalPages and MapUserPhysicalPagesScatter functions, 399
MapViewOfFile function, 387, 564
marshaling, RPC, 800
masked interrupts, 93
Master Boot Record (MBR)
  boot process, 252
  corruption problem, 281
Master Boot Record (MBR) partitioning
  basic disks, 624
  dynamic disks, 630
master file table (MFT)
  logical prefetcher, 459
  NTFS, 733–739
master keys, EFS, 776
media, NTFS and read-only, 728
memory. *See also* virtual memory
  boot options, 257, 259, 260
  configuration manager and hive bins, 201
  dirty page threshold, 686
  DPC/dispatch IRQL, 99
  dumps. *See* crash dump analysis
  hive size limits, 198
  hives and paged pool, 203–207
  HTTP API, 804
  management. *See* memory management
  optimizers, 480–482
  physical. *See* physical memory
  pools. *See* system memory pools
  registry and volatile data, 24
  resource-allocation policies, 333
  SANs, 797
  Section executive object type, 127
  synchronization and. *See* synchronization
  system code write protection, 864
  type objects, 130
  viewing start addresses for threads, 113
  Windows NT shared, 2
memory descriptor list (MDL), 426, 568, 679
memory management, 375–483. *See also* memory; virtual memory
  address translation, 425–438
  cache manager, 660–661
  KTHREAD blocks, 315

www.TopSage.com

memory management, continued
memory manager and. *See* memory manager
memory optimizers, 480–482
page fault handling, 439–447
page frame number (PFN) database, 469–482
section objects, 450–457
system memory pools, 401–413
virtual address descriptors (VADs), 448–450
virtual address space layouts, 413–422
working sets, 457–468
memory manager, 375–400. *See also* memory management
Address Windowing Extensions (AWE), 399–400
allocation granularity, 385
cache manager, 656
cache working set size, 665
components, 376–377
configuring, 378
copy-on-write page protection, 392–394
data execution prevention, 390–392
examining memory usage, 378–381
executive, 65
executive object delete method, 134
explicit file I/O and virtual, 704
fault handling, 439, 705. *See also* page fault handling
file system drivers, 694
groups of functions, 382
heap manager, 394–399
hive paged pool memory, 198
initialization, 266, 268
internal synchronization, 377
large and small pages, 382–384
locking pages in physical memory, 385
mapped file I/O, 564
memory optimizers, 481
modified and mapped page writer threads, 704
NTFS file system driver, 730
portability, 40
process address space, 306
protecting memory, 388–394
reserving and committing pages, 384–385
services, 382–400
shared memory, mapped files, and section objects, 386–388
standby and modified page lists and cache physical size, 667–668
system code write protection, 864
tasks, 375
virtual memory, 14–16. *See also* virtual memory
memory optimizers, 480–482
memory-mapped file functions, 382
menu, boot, 261, 263, 274, 275. *See also* Boot.ini
Message Queuing API, 802, 814
MessageBox function, 222
messages
local procedure call, 174
named pipe, 805
Winsock multipoint, 792
Messaging API (MAPI), 840
metadata
cache manager mapping and pinning interfaces, 677–678
defined, 690
NTFS files, 733
recoverable file systems, 659
methods, object, 22, 133–135
MFT mirror, 737
MiApplyDriverVerifier function, 411
microkernel-based systems, Windows vs., 36
Microsoft Access, 716
Microsoft Corporate Error Reporting (CER) Toolkit, 854
Microsoft Developer Network. *See* MSDN (Microsoft Developer Network)
Microsoft Distributed Transaction Coordinator (MS DTC), 815
Microsoft Enhanced Cryptographic provider, 778
Microsoft Exchange Server, 841
Microsoft Excel, 716
Microsoft Interface Definition Language (MIDL) compiler, 800
Microsoft Internet Explorer, 196
Microsoft iSCSI Software Initiator, 618
Microsoft MPIO Driver Development Kit, 619
Microsoft Online Crash Analysis (OCA), 119, 846, 853, 854–855
Microsoft Security Response Center, 488
Microsoft Shadow Copy Provider, 650–651
Microsoft SQL Server, 815
Microsoft symbol server, 28, 850
Microsoft TechNet Scripting Center WMI scripts, 246
Microsoft Transaction Server (MTS), 815
Microsoft Windows. *See* Windows
Microsoft Word, 715
MiDispatchFault function, 703
MiMappedPageWriter function, 475
MiModifiedPageWriter function, 475
minidump crash dump files, 850
miniport call manager (MCM), NDIS, 833
miniport drivers, 543, 617. *See also* NDIS (Network Driver Interface Specification) drivers
minor function codes, IRP, 566
MIPS architecture, 40
mirrored volumes, 628, 635–637, 719
MiZeroInParallel function, 377, 473
MKS Toolkit, 60
MmAccessFault function, 439, 703, 705
MmFlushSection function, 704
MmIsThisAnNtAsSystem function, 49
MmPrefetchPages function, 460
MmProbeAndLockPages, MmLockPagableCodeSection, MmLockPagableDataSection, and MmLockPagableSectionByHandle functions, 385
MmProductType global variable, 49
MmSizeOfSystemCacheInPages system variable, 664
MmTrimAllSystemPagableMemory, 413
modified no-write pages, 441, 442, 469
modified page list, memory manager, 441, 442, 474, 475, 667–668
modified page writer, 376, 384, 475
modified pages, 469. *See also* page files
MOF Compiler (Mofcomp.exe) tool, 243
more command, viewing data streams using, 720

www.TopSage.com

Mortice Kern Systems Inc. (www.mkssoftware.com), 60
Motorola PowerPC systems, 40
Mount Manager device driver, 638–639, 641
mount operations, NTFS, 737
mount points, 639–642
mounting, volume, 642–646
Mountvol.exe tool, 640, 642
MoveFileEx function, 271
Mpdev.sys, 619
Mpio.sys, 619
Mpsfltr.sys, 619
Msconfig tool, 273
MSDN (Microsoft Developer Network)
  Device Driver Kit (DDK), 34
  Platform SDK, 33
  resource kits, 28
  Windows API documentation, 3
MS-DOS operating system
  applications and process creation, 302
  boot process, 254, 256
  filenames, 693, 742
  volume management, 623
Msgina.dll, 81, 490, 527
Msinfo32.exe tool, 546–548
Msiscis.sys, 618
MSV1_0 authentication package, 530, 841
Mswsock.dll, 795
multi() syntax, 254
multipartition volume management, 632–638
  basic disks, 622
  mirrored volumes, 635–637
  RAID-5 volumes, 637
  spanned volumes, 633
  striped volumes, 634
  types of volumes, 615, 633
  watching mirrored volume I/O operations, 636
Multipath Bus Driver, 619
Multipath Disk Driver Replacement, 619
multipath I/O (MPIO) drivers, 619
Multipath Port Filter, 619
multiple provider router (MPR), 225, 272, 816–818
multiple redirector support. *See* redirectors, network
Multiple UNC Provider (MUP), 818–819
multipoint messages, Winsock, 792
multiprocessing, symmetric, 41–46, 150
Multiprocessor Specification, 89
multiprocessor systems
  boot options, 261
  memory manager, 377
  synchronization. *See* synchronization
  thread scheduling on. *See* multiprocessor thread scheduling
  Windows NT and, 2
multiprocessor thread scheduling, 357–369
  algorithms, 366–369
  dispatcher database, 357–359
  hyperthreaded systems, 360
  ideal and last processor, 366
  idle processors, 367–368
  nonuniform memory access (NUMA) systems, 360–362
  processor affinity, 362–365
  selecting threads to run on specific CPUs, 368–369
multitasking, 41
mutants, mutexes vs., 128
Mutex executive object type, 127
mutexes
  fast and guarded, 163
  mutants vs., 128
  signaled state, 160
mutual exclusion, 149–151

N

name resolution
  drive-letter, 701
  Domain Name System (DNS), 820
  Windows Internet Name Service (WINS), 820
  Winsock, 794
named pipes, 804–809
  impersonation, 502
  implementation, 807–808
  listing namespace and watching activity, 809
  mailslots vs., 804
  operation, 805–806
  services, 224, 228
names. *See also* name resolution; namespaces
  Advanced RISC Computing (ARC) naming convention, 254
  boot options for kernel image and HAL, 259
  device, 551
  disk device object, 620
  dynamic disk, 632
  executive object, 143–146
  executive object name retention, 141
  executive object parse method, 134
  filename associations and registry, 189
  filenames and file objects, 559
  function name prefixes, 74
  idle thread, 348
  logical prefetcher, 459
  mailslot, 806
  manifest file, 312
  memory information and confusing, 378
  MS-DOS filenames, 693, 742
  multiple provider router (MPR), 818
  named data streams, 720
  named pipe, 805. *See also* named pipes
  NetBIOS, 811, 812
  network name resolution, 820
  NTFS file attribute, 742
  NTFS file system and Unicode-based, 721
  NTFS filenames, 742–744
  NTFS metadata files, 733
  object manager, 125
  objects and system resource, 23
  partition, 616
  registry and device driver, 848
  registry keys, 185
  RPC server name publishing, 801
  service, 82
  System Restore, 707
  Universal Naming Convention (UNC), 804
  viewing device name to device name mappings, 560
  Windows system component filenames, 39
namespaces
  device, 560
  device objects, 621
  DFS and, 843
  listing named pipe, 809
  object manager, 807
  object manager, registry, and file system, 134
  object manager session, 146–149
  registry, 207–209
  viewing object manager, 124

www.TopSage.com

namespaces, *continued*
volume, 638–646
Winsock and, 792, 794
WMI, 243–244
National Computer Security Center (NCSC), 485
native system services, 5
Nbtstat command, viewing NetBIOS names using, 812
NDIS (Network Driver Interface Specification) drivers, 828–837
capturing network packets using Network Monitor, 834
connection-oriented, 832
hybrid intermediate drivers, 832
library and features, 828–830
listing loaded, 830–832
overview, 790
QOS (quality of service), 836
Remote NDIS, 835
NDIS library, 789, 828
NDIS packets, 829
NdisAllocatePacket function, 829
Ndiskd kernel debugger extension library, 830–832
neither I/O, 569
.NET Framework, 4
NetBIOS (Network Basic Input/Output System) API, 811–813
implementation, 813
names, 811
operation, 812
viewing names using Nbtstat, 812
WINS name resolution, 820
NetBIOS Extended User Interface (NetBEUI) protocol, 813
Netbios function, 812, 813
NetBIOS over TCP/IP driver, 813
NetBT driver, 813
Netlogon
authentication, 531
security, 490
network adapters, 790
network address translation (NAT), TCP/IP and, 825
Network Driver Interface Specification (NDIS) drivers. *See* NDIS (Network Driver Interface Specification) drivers; protocol drivers
network file system drivers, 696
network layer, OSI model, 788
Network Load Balancing, 842
network logon service. *See* Netlogon
Network Monitor, 834
network redirectors. *See* redirectors, network
network service account, 220
network services, 839–844. *See also* networking
Active Directory, 840–841
Distributed File System (DFS), 843
File Replication Service (FRS), 843
Network Load Balancing, 842
remote access, 839–840
RPC, 801
networking, 787–844
APIs. *See* networking APIs
architecture, 787–791
binding, 838
components, 789–791
direct memory access (DMA), 678
executive object access, 145
internal error-reporting online crash anlysis, 854
layered network services, 839–844
local system account, 218
multiple redirector support, 815–819
name resolution, 820
NDIS (Network Driver Interface Specification) drivers, 828–837
network drive letters, 225
network service accounts, 220
OSI reference model, 787–789
protocol drivers, 821–828
safe mode boots, 275
networking APIs, 791–815
Distributed Component Object Model (DCOM), 814
Message Queuing, 814
named pipes and mailslots, 804–810
NetBIOS (Network Basic Input/Output System), 811–813
overview, 789
Real-Time Communications (RTC) Client, 814
remote procedure call (RPC), 798–803
Universal Plug and Play (UPnP), 815
Web access APIs, 803–804
Windows Sockets (Winsock), 791–798
Nfi tool
viewing encrypted files, 785
viewing master file table, 734–737
no execute page protection, 258, 389, 390–392
noncached file read operations, 675
noninvasive user-mode debugging, 29
nonpaged memory pool
contents of, 401
size of, 381
system space, 415
non-Plug and Play drivers, 542, 550
nonresident attributes, NTFS, 744–747
nonsignaled state, dispatcher object, 156
nonsparse data, compressing, 750–751
non-uniform memory architecture (NUMA), 41, 360–362, 366
Notepad
locating registry settings for, 194
viewing INF files, 604
notifications
Filemon tool and file change, 712
low and high memory, 479–480
notification event objects, 160
NTFS directory, 724
Plug and Play manager, 592
NotifyBootConfigStatus function, 230, 272
Notmyfault tool, 407, 851, 855, 861
Novell NetWare, 815, 821
NT Kernel Logger, 175–182
Ntbootdd.sys, 252, 256
NtCreateFile function, 127, 556, 643, 701
NtCreateIoCompletion function, 587
NtCreatePagingFile function, 446
NtCreateProcess function, 304
NtCreateThread function, 308, 322
Ntddk.h, 94
Ntdetect.com, 252, 258, 263
NtDeviceIoControlFile function, 568
Ntdll.dll, 39, 43, 63, 123, 176, 394, 534
NTDS API, 841
Ntevt.mof, 248
NTFS file system, 255, 717–785
advanced features, 719–728
bad-cluster recovery, 722
change logging, 724
cluster sizes, 694
data compression and sparse files, 723

www.TopSage.com

data redundancy and fault tolerance, 718
defragmentation, 727
design requirements, 717–719
Encrypting File System (EFS), 726, 775–785
File Replication Service (FRS), 843
file system driver, 729–731
hard links and junctions, 722–723
indexing, 721
link tracking, 726
mount points, 639
multiple data streams, 719–721
on-disk volume structure. *See* volumes, NTFS
per-user volume quotas, 725
POSIX support, 727
read-only media support, 728
recovery. *See* recovery, NTFS
security requirement, 718
Unicode-based names, 721
NTFSInfo tool, 739
NtGlobalFlag variable, 168–171
NtInitializeRegistry function, 230, 278
Ntkrnlmp.exe, 43, 259
Ntkrnlpa.exe, 39, 43, 45, 259, 260, 261, 436
Ntkrpamp.exe, 43, 259
Ntldr (boot loader)
boot process, 252, 616
boot sector, 255–264
boot sector corruption problem, 281
hive size limits, 198
safe mode boots, 276
SYSTEM hive, 204
NtLoadDriver service, 228
NTOSBOOT-B00DFAAD.PF, 460
Ntoschk.exe, 259
Ntoschkpa.exe, 259
Ntoskrnl.exe
boot process, 252, 259, 265
checking running versions of, 45
heap manager, 394
kernel in, 65–67. *See also* kernel
memory manager, 376
service descriptor table, 122
symbol files, 28
as system component, 39
undocumented interfaces in, 73
versions, 43
viewing HAL image dependencies using, 69
Windows executive in, 63–65
Windows Management Instrumentation (WMI) and, 176
NtQuerySystemInformation function, 459, 852
NtReadFile function, 568, 702
NtRemoveIoCompletion function, 587
NtSetEventBoostPriority function, 350
NtSetInformationFile function, 587
NtSetIoCompletion function, 588
NtSetSystemPowerState function, 288
NtShutdownSystem function, 288
NtShutdownSystem service, 230
Ntuser.dat, 187
Ntvdm.exe, 302
NtWriteFile function, 123, 568, 704
NUMA (non-uniform memory architecture), 41, 360–362, 366
NVRAM boot process, 264
NWLink protocol, 821
NwLinkNB driver, 813

O

ObCheckObjectAccess function, 493
object directory objects, 146
object identifier file, NTFS, 739
object IDs, NTFS, 726, 754
object manager, 124–149
access checks, 493–495
configuration manager, registry namespace, and, 207
device objects, 551
EPROCESS block, 294
executive, 65
executive objects, 126–128. *See also* executive objects
explicit file I/O, 701
functions and type of objects, 124–126
initialization, 266, 267
named pipes and mailslots, 807
namespace and device objects, 621
namespace and dynamic disks, 632
NTFS file system driver, 730
object handles and process handle tables, 135–140
object headers and bodies, 129–130
object methods, 133–135
object names, 143–146
object retention, 141–142
object security, 140, 492. *See also* object security
object structure, 128
privileges, 520
resource accounting, 143
section objects, 450
session namespace, 146–149
tools for exploring database of, 124
type objects, 130–133
viewing handle tables with kernel debugger, 140
viewing namespace instancing, 148
viewing object headers and type objects, 131–132
WNet API, 817
object reuse protection, C2 security rating and, 486
object security, 492–516. *See also* security
access checks, 493–505
object manager, 140, 492
security descriptors and access control, 506–515
Windows and, 24
ObjectAttributes parameter, 22
object-oriented design, Windows, 37
objects. *See also* object manager
attributes, 22
disk device, 620–621
executive. *See* executive objects
executive process, 304–308
executive thread, 322
functions and types of, 124–126
handles, 22, 135–140
headers and bodies, 129–130
kernel, 66, 126
methods, 22, 133–135
names, 143–146
object directory, 146
retention, 141–142
security. *See* object security; security descriptors
signaled state, 158
stores and inheritance, 511
structure, 128
type, 130–133
volume, 646
Windows and, 37
WMI managed, 238, 244

www.TopSage.com

ObOpenObjectByName function, 701
ObpCreateHandle function, 493
ObpIncrementHandleCount function, 493
ObpKernelHandleTable, 139
OEM Support Tools, 734–737, 785
Oh.exe tool
  viewing memory resource notification events, 480
  viewing open handles using, 124
on-demand symbol server, 28
Online Crash Analysis (OCA), 119, 846, 853, 854–855
open method, executive object, 134
Open Software Foundation (The Open Group), 798
OpenEncryptedFileRaw function, 784
OpenFileMapping function, 387
Openfiles /query command, viewing open handles using, 124
OpenProcess function, 29
OpenSCManager function, 223, 236
operating system model, Windows, 36–37
operations, interlocked, 151, 154
oplocks (opportunistic locking), 697
optimization, performance. *See* performance
optimizers, memory, 480–482
Orange Book, 485
OS/2 subsystem, 61–62
OSI (Open Systems Interconnection) reference model, 787–789
OSTA (Optical Storage Technology), 691
overrun detection, Driver Verifier, 412

P

packet filtering API, 826
packets, NDIS, 829
PAE (Physical Address Extension), 39, 260, 261, 391, 435–437
page coloring, 482
page directories, 428–430
page directory entries (PDEs), 427, 428
page directory index, 426
page directory pointer table, 436, 437
page fault handling, 439–447
  collided page faults, 444
  crashes, 845
  file I/O, 703, 705
  in-paging I/O, 443
  invalid page table entries, 440
  memory manager, 376
  page fault reasons, 439–440
  page files, 444–447
  prototype page table entries, 441–443
page files. *See also* pages
  crash dump files, 869
  invalid PTEs, 440
  kernel memory dumps and size of, 851
  modified page writer, 475
  page fault handling, 444–447
  page file backed sections, 387
  page file quota, 445
  pages in, 442
page frame number (PFN) database
  data structures, 476–479
  low and high memory notification, 479–480
  memory management, 469–482
  memory optimizers, 480–482
  modified page writer, 475
  page list dynamics, 472
  page states, 469–471
  valid PTEs, 431
  viewing, 472
  viewing page fault behavior, 474
page granularity virtual memory functions, 382
page list dynamics, 471, 472
page maps, 438
page table entries (PTEs)
  address translation and page tables, 425–427
  invalid, 440. *See also* page fault handling
  page frame number (PFN) database, 469, 476
  page tables, 430–432
  prototype, 441–443
  system, 421
  system space and system, 415
  valid. *See* address translation
page tables
  page frame number (PFN) database, 469
  page table entries, 425–427, 430–432
  process virtual address space, 413
page writer threads, memory manager's modified and mapped, 704
paged memory pool
  contents, 401
  hive size limits, 198
  hives, 203–207
  system space, 414
Pagedefrag tool, 445
pageheap tool, 398
pages
  copy-on-write protection, 392–394
  dirty and modified page writer, 376
  initial process address space, 305
  large and small, 382–384
  locking, into physical memory, 385
  memory manager task, 375
  modified. *See* page files
  no execute protection, 389, 390–392
  page states, 469–471
  prototype PTEs and shared, 441–443
  reserving and committing, 384–385
  system working set, 467
paging. *See also* pages
  data to disk, 15
  initialization, 270
  Ntldr, 256
parameter checking, heap manager, 398
parameter marshaling, RPC, 800
parent processes, 6–8, 304
parse method, executive object, 134, 145
partial quantum decay, 341
partition entries, LDM, 627
partition manager
  disk drivers, 622
  FtDisk driver, 626
partition tables
  boot process, 253
  defined, 615
  Master Boot Records, 624
partitions
  boot process, 251
  boot sectors and format of, 255
  defined, 615
  GUID Partition Table (GPT), 624
  Logical Disk Manager (LDM) and dynamic disks, 626–630

www.TopSage.com

Master Boot Record (MBR), 624
MS-DOS, 623
names, 616
partition manager, 622, 626
primary, 620
soft vs. hard, 630
types of, 624
volumes vs., 625
passive IRQLs, 96
passwords
authentication, 23, 530. *See also* authentication
registry, 190
service, 227
paths
key control blocks, 207, 211
MPIO drivers, 619
registry hive, 197
service executables, 214
trusted path functionality, 487
PC card remove/eject device utility, 592
PCI devices boot option, 261
Pciidex.sys and Pciide.sys, 618
PC-phone calls, 814
PE (Preinstallation Environment) boot option, 260
PEB (process environment block)
EPROCESS block, 294–296
process creation, 307
processes, 289
peer-to-peer network connectivity, 815
Pendmoves tool, 271
per-file cache data structures, 670–674
performance
client vs. server Windows versions, 49
lazy IRQL, 93
monitoring. *See* performance counters; Performance Monitor; Performance tool; Windows Management Instrumentation (WMI)
Performance Options dialog box, 199, 663
registry, 191, 210–211
scalability, 46
Windows design goal, 36
performance counters
cache flush, 684
cache manager mapping and pinning activity, 678
cache manager MDL activity, 679
cache physical size and page fault, 665
cache read activity, 676
copy-on-write faults, 394
determining maximum memory pool sizes, 402–404
fast I/O, 681
intelligent read-ahead, 682
lazy writer, 684
memory usage, 378
mode-related, 18
monitoring memory pool usage, 404
page file, 446
performance monitoring with, 237
Performance tool, 27. *See also* Performance tool
process private committed memory, 444
process-related, 297–298
registry, 24, 191
session, 419
system memory pool, 402
system service activity, 124
system working set, 467
thread-related, 321
Performance Data Helper API, 191
Performance Monitor
watching foreground priority boosts and decays, 351
watching priority boosts for CPU starvation, 355–356
watching priority boosts on GUI threads, 353–354
Performance Options dialog box, 199, 663
Performance tool. *See also* performance counters
accounting for physical memory use, 380–381
determining page table entries (PTEs), 422
examining memory usage, 378
features, 27
tracing TCP/IP activity, 177
viewing job objects, 371–373
viewing kernel mode and user mode activity, 19–21
viewing process CPU utilization, 10
viewing process working set sizes, 465
viewing processes, 7
viewing system page table entries (PTEs), 415
viewing system service activity, 124
viewing thread activity, 323
watching cache flushes, 685
watching mirrored volume I/O operations, 636
watching thread-scheduling state changes, 336
permanent objects, 141
permissions editors, warning about GUI, 514
per-processor look-aside list, cache manager, 688
per-user class registration data, 189
per-user preferences, registry and, 184
per-user volume quotas, NTFS, 725
Pfmon tool, 474
Phase1Initialization function, 267
phone calls, 814
Physical Address Extension (PAE), 39, 260, 261, 391, 435–437
physical device objects (PDOs), 599
physical layer, OSI model, 789
physical memory. *See also* working sets
accounting for, 380–381
address translation. *See* address translation
Address Windowing Extensions, 399
kernel variables, 479
locking pages in, 385
mapping virtual memory to, 14
maximum, 375
page states. *See* page frame number (PFN) database
physical size, cache, 667–668
PICs (Programmable Interrupt Controllers), 89–90
pinning interface, cache manager, 677–678
PipeList tool, 809
placement policy, working set, 462
Platform Software Development Kit (SDK)
documentation, 3
Gflags tool. *See* Gflags tool
tools, 33
Pstat tool. *See* Pstat tool
Windows Sockets Configuration tool, 795
Winobj tool. *See* Winobj tool

www.TopSage.com

Plug and Play drivers, 542, 550
Plug and Play (PnP) manager, 590–606
  capabilities, 590–591
  driver loading, initialization, and installation, 594–602
  driver Plug and Play support, 592–593
  executive, 64
  FtDisk driver, 626
  initialization, 267, 268
  I/O system, 538
  IRQLs, 95
  levels of device and driver Plug and Play support, 591
  partition manager, 622
  safe mode boots, 276
  user-mode, and driver installation, 603–606
Plug and Play system, registry and, 185
Pmon.exe tool, 378
PnP_DeviceList function, 224
policies
  device power, 610
  driver-signing, 605
  IPSec, 827
  Lsass policy database, 489
  resource-allocation, 333
  system power, 609, 611–612
polling, registry, 185, 193
pool corruption, Driver Verifier and, 861–863
pool tagging, 404–408
pool tracking Driver Verifier option, 412
Poolmon tool
  checking page pool usage, 203
  monitoring memory pool usage, 404
  troubleshooting pool leaks, 407
pools, system memory. *See* system memory pools
PoRegisterDeviceForIdleDetection function, 613
PoRequestPowerIrp function, 610
port drivers
  disk, 617
  layered drivers, 543
  printer drivers, 542
port objects, LPC, 172–174
port/miniport model, FSDs and, 697
portability, Windows, 35, 40
ports
  debugger and exception, 111
  drivers. *See* port drivers
  I/O completion. *See* I/O completion ports
  LPC, 174
PoSetDeviceBusy function, 613
POSIX
  APCs, 108
  exceptions, 112
  executive objects, 127
  filenames, 742
  fork function, 394
  processes, 302, 306
  NTFS support, 727
  as subsystem, 60–61
  tools for watching POSIX subsystem start, 61
PostQueuedCompletionStatus function, 587, 588
post-splash screen crash or hang problem, 284–286
power consumption, system power states and, 607
power dispatch routines, device driver, 610
power fail IRQL, 96
power manager, 607–613
  driver control of device power state, 613
  driver power operation, 610–611
  executive, 64
  initialization, 267, 269
  I/O system, 538
  system power states and device power states, 607–608
  system power-state transitions, 609
  viewing system power capabilities and policy, 611–612
preboot, x86 and x64, 251–255. *See also* boot process
predefined interrupt request levels (IRQLs), 96–99
preemptive thread scheduling, 346
preferences, registry and, 187
prefetching
  application, 311
  logical prefetcher, 65, 458–462
  looking inside prefetch files, 460
  watching prefetch file reads and writes, 461
PrefetchTracesReady event, 459
prefixes, function name, 74
Preinstallation Environment (PE) boot option, 260
presentation layer, OSI model, 788
Previous Versions Client extension, 653
primary access tokens, 499
primary bus drivers, 596
primary partitions, 620, 624
principal names, RPC, 801
printer drivers, 542
priority boosts
  CPU starvation, 354–357
  for foreground threads after waits, 351–353
  after GUI threads wake up, 353–354
  after I/O completion, 349
  listening to, 357
  quantum boosts, 343
  thread scheduling, 348–357
  after waiting for events and semaphores, 350
priority classes, process
  job objects, 370
  process creation, 301
  start command, 331
  thread priority levels, 328
priority levels
  boosting. *See* priority boosts
  DPC, 105
  examining and specifying process and thread, 331
  interrupt levels vs., 334
  IRQL vs. thread-scheduling, 92
  real-time, 333
  thread scheduling, 327–329
private address space, process, 388, 413
private assemblies, 312
private cache map structures, 670–674
Private Header sectors, LDM, 627
private/public key pairs, EFS. *See* Encrypting File System (EFS)
PrivilegeCheck function, 518
privileges, 518–524
  access control, 23. *See also* access control
  access token privilege arrays, 499
  account rights vs., 516
  Bypass Traverse Checking privilege usage, 523
  enforcement of, 518–522
  local system account, 218–219

www.TopSage.com

super privileges, 523–524
viewing enabling of, 519
x86 system privilege levels, 16
ProbeForRead and ProbeForWrite functions, 569
process control block (PCB). *See* KPROCESS blocks
Process Control Manager tool, 371
process environment block. *See* PEB (process environment block)
Process executive object type, 127
Process Explode (Pview) tool, 76
Process Explorer
determining maximum memory pool sizes, 402–404
examining and specifying process and thread priorities, 331
examining threads and IRPs, 581
exploring object manager using, 124
mapping System process system threads to device drivers, 79
monitoring interrupt and DPC activity, 106
named pipes and processes, 810
setting processor affinity, 363
troubleshooting pool leaks, 407
viewing access tokens, 502
viewing and changing process affinity, 363
viewing cache physical size, 668
viewing enabling of privileges, 519
viewing file objects and device handles, 558
viewing hive handles, 199
viewing job objects, 371–373
viewing loaded driver list, 546–548
viewing mapped files, 387
viewing memory resource notification events, 480
viewing object manager namespace instancing, 148
viewing open object handles, 136
viewing page file usage, 447
viewing process tree using, 272
viewing processes using, 10–12
viewing restricted tokens, 504–505
viewing section objects, 451
viewing service details inside service processes, 84
viewing services running inside processes, 235
viewing start addresses for threads, 113
viewing system process tree, 75
viewing thread activity, 323
viewing Wmiprvse.exe creation, 247
watching POSIX subsystem start, 61
process handle tables, executive object, 135–140
process heap, 395
process ID, 6
process manager
executive, 64
initialization, 267
privileges, 521
process objects
process type objects, 130
signaled state, 158
process properties dialog box, 235
Process Status. *See* Pstat tool
Process Viewer (Pviewer) tool, 76, 77, 331
process working sets
content, 457
list, 414
setting limits, 463
size of, 381
viewing sizes, 465
processes, 289–313
automatic startup of, 273
creating, using CreateProcess function, 300–313
creating executive process objects, 304–308
creating initial thread, 308–309
data structures, 289–296
examining and specifying priorities, 331
file objects, 559
functions, 298–299
kernel variables, 297
memory manager services, 382
notifying Csrss about new, 309–310
opening executable images, 302–304
performance counters, 297–298
performing initialization in context of new, 310
priority classes and thread scheduling, 328, 331
private address space, 388, 413
processor affinity, 362
quantum boosting and foreground, 343, 344
security structures, 526
service, 216
shared service, 233–235
side-by-side assemblies, 311–313
starting execution of initial thread, 310
system, 75–84
threads, jobs, and, 6–14. *See also job objects; threads*
Userinint.exe, 533
user-mode debugging, 29
user-mode types, 38
viewing, using Performance tool and Tlist.exe tool, 7
viewing, using Process Explorer, 10–12
viewing, using Task Manager, 8
viewing and changing process affinity, 363
viewing EPROCESS blocks, 299
viewing services running inside, 235
viewing sessions, 419
virtual size of, 375
processor access modes. *See* kernel mode; user mode
processor affinity
functions, 362
job objects, 370
multiprocessor thread scheduling, 362–365
thread scheduling, 326
processor control blocks (PRCBs), 91, 94, 359
processor control regions (PCRs), 94, 153
processors. *See* CPUs (central processing units)
process/stack swapper, 376
product ID (PID), 600
profile IRQL, 96
profiles. *See* hardware, profiles; system profile; user profiles
Programming Applications for Microsoft Windows , 3, 109, 160
programming languages, 840. *See also* assembly programming language; C and C++ programming languages
Programming the Microsoft Windows Driver Model, Second Edition, 34
programs. *See also* applications; executable images
automatic startup of, 273
processes vs., 6
safe-mode-aware, 277

www.TopSage.com

programs, *continued*
setup. *See* setup programs
tools. *See* tools
protected driver list, 606
protected mode, 256
protection, memory
large pages, 383
memory manager, 388–394
PTE flags, 431
protection, system code write, 860, 863–864
Protection Profile (PP) concept, 487
protocol drivers, 821–828
device drivers, 70
overview, 789, 821
TCP/IP extensions, 824–828
Transport Driver Interface (TDI), 823–824
viewing TCP/IP device objects, 822
watching TDI activity, 824
protocol stack, network, 789
protocol-specific Winsock Helper libraries, 795
prototype page table entries (PTEs), 441–443
prototype registry, 184
providers
cryptographic service, 778
event tracing, 175
VDS, 649
Volume Shadow Copy, 649
WMI, 238, 239–240
WNet, 816
PsCreateSystemThread function, 76
PsExec tool
executing Regmon with, 196
running Blue Screen screen savers on remote systems, 870
running Regedit in local system account with, 190
setting processor affinity, 363
viewing registry security settings, 491
viewing System Restore directory, 708
PsGetSid tool, 496
PspCreateThread function, 308
PspUserThreadStartup function, 310
Pstat tool
identifying system threads in System process, 77
Idle process in, 76
viewing installed device drivers, 72
viewing start addresses for threads, 113
viewing system memory information, 378
Psxss.exe, 61
publisher configurations, 313
push locks, 165
Pview tool, 76
Pviewertool, 76, 77, 331

Q

QOS (quality of service)
NDIS, 836
Winsock and, 791
quantum
accounting, 340
boosting, 343
clock intervals, 340
controlling, 341–343
default initial, 306
expiration, 106
job objects, 370
quantum end thread scheduling, 347
registry settings, 344
thread scheduling, 326, 340–344
QueryDosDevice function, 559
QueryMemoryResourceNotification function, 479
query-remove command, Plug and Play manager, 592
QueryServiceStatus function, 236
query-stop command, Plug and Play manager, 593
QueryUsersOnEncryptedFile function, 785
queue dispatcher objects, 166
queue objects, 587
queued spinlocks, 153–154
queues
associated IRPs, 582
asynchronous requests to layered drivers, 576, 578
synchronous IRPs, 570
QueueUserAPC, 109
QuickSlice tool
Idle process in, 76
viewing thread activity using, 18
quota, page file, 445
quota charges attribute, executive object, 143
Quota Entries tool, 725
quota file, NTFS, 739
quota setting, registry, 198
quota tracking, NTFS, 725, 755

R

RAID adapters
Storport.sys, 617
VDS, 648
RAID level 0 volumes, striped volumes as, 634
RAID level 1, mirrored volumes as, 635–637, 719
RAID level 5 volumes, 719, 637
RAM memory optimizers, 480–482
rating standards, security, 485–488
Common Criteria (CC), 487–488
Trusted Computer System Evaluation Criteria (TCSEC), 485–487
raw (unstructured) I/O device objects, 632
RcpAsyncGetCallStatus function, 800
read-ahead, cache manager, 658, 682–683
read-ahead threads, 705
ReadDirectoryChangesW function, 712, 724
ReadEncryptedFileRaw function, 784
ReadFile function, 701
ReadFileEx function, 109, 575
ReadFileScatter function, 564
read-only media support, NTFS, 728
ReadProcessMemory function, 13, 388
ready queues, thread dispatcher, 339
ready state, thread, 335
ready threads, 326
real mode, boot process and, 256
Real-Time Communications (RTC) Client API, 814
Real-time priority class, process, 301
real-time priority level, thread, 333, 349
real-time processing
interrupt dispatching, 102
Windows and, 333
rebooting. *See* boot process
record types, NTFS log, 763–766
records, NTFS file, 740–742
recoverable file systems, 759–761. *See also* recovery modes
cache manager, 658–659
careful write file systems, 759
lazy write file systems, 760
NTFS as, 760

www.TopSage.com

recovery, NTFS, 758–775
  bad-cluster recovery, 771–775
  design requirement, 658, 717
  evolution of recoverable file systems, 759–761
  logging, 761–766
  tables and process, 767–770
Recovery Agent Wizard, 780
Recovery Agents, 779
Recovery Console, 258, 279–284
recovery modes
  boot options, 259, 261
  last known good (LKG), 274
  safe mode, 274–279
recursive mount points, 641
recv function, 792
redirector file system driver (FSD)
  named pipes and mailslots, 807
  network redirectors, 815
redirectors, network, 815–820
  cache coherency, 658
  multiple provider router (MPR), 816–818
  Multiple UNC Provider (MUP), 818–819
  remote FSD, 697
  RPC, 799
  servers, 70
  write throttling, 686
redo operations, NTFS recovery, 764, 768
redundancy, data. *See* data redundancy
reentrant code, Windows NT and, 2
reference counts
  executive object, 141
  key control block, 208
  objects, 23
  PFN, 476
REG_BINARY data type, 185
REG_DWORD data type, 185
Regedit and Regedt32 tools. *See* Registry Editor tools
Reg.exe tool, 184
RegisterHotKey function, 529
RegisterServiceCtrlHandler function, 216
RegisterServiceProcess function, 225
RegisterServicesProcess function, 232
registry, 183–211
  authentication packages, 530
  boot process, 260, 263
  Boot.ini options, 257
  cache size, 662
  configuration manager, 197–211. *See also* configuration manager
  crash dump generation, 852
  crash generation, 867
  data types, 185–186
  device driver and service types, 72
  device driver load order, 595
  device driver names, 848
  device driver Start value, 594–595
  devnode driver loading, 600
  drive-letter assignments, 638
  Driver Verifier settings, 410, 412
  driver-signing policy, 605
  enabling and disabling prefetching, 459
  encryption algorithms, 775
  execution protection, 391
  executive object quota charges, 143
  explicit device driver loading, 594
  global flags variable, 168
  initial processes, 81
  initial system worker threads, 167
  initialization, 268, 269
  installed devices, 598
  interactive services, 222
  I/O system, 538
  IPSec policies, 827
  IRP stack locations, 565
  kernel versions, 45
  Key executive object type, 128
  large pages for device drivers, 383
  last known good (LKG) mode, 274, 285
  locating application settings, 194
  logging activity in unprivileged accounts or during logon/logoff, 196
  low and high memory notification, 479
  memory allocation addresses, 417
  memory manager, 378
  modified page writer setting, 475
  multipartition information, 623
  multiprocessor setting, 42
  namespace, 134
  network binding information, 839
  network providers, 817
  nonmaskable interrupt, 867
  NTFS short-name generation, 744
  page files, 445, 446
  process creation, 303
  quantum settings, 344
  Regmon troubleshooting techniques, 192–196
  restoring, using System Restore, 706
  root keys, 186–191
  safe mode, 261, 275, 278
  security databases, 489
  security settings, 279, 491
  service control manager, 223
  Service Host process, 233
  services, 212–215
  Session Manager initialization, 80
  session space size, 414, 419
  Software Restriction Policies, 534
  subsystem startup information, 53
  system code write protection, 864
  SYSTEM hive corruption problem, 283
  system memory pool size, 401
  system page table entries (PTEs), 422
  system working set priority, 468
  tools for viewing and changing, 183. *See also* Registry Editor tools
  unhandled exceptions, 115
  usage, 184
  VDS providers, 649
  viewing activity on idle systems, 193
  watching user profile loading and unloading, 189
  Windows and, 24
  Windows Error Reporting, 118, 853
  Windows version setting, 48
  WMI and, 248
Registry Editor tools
  last known good (LKG) mode, 285
  manually loading and unloading hives, 198
  running, in local system account, 190
  unnamed values, 185
  viewing security settings, 491
  watching user profile loading and unloading, 189
  Windows versions and, 184
Registry key object, 207
REG_LINK data type, 186
Regmon tool, 192–196
  internals, 192–194
  locating application registry settings, 194

www.TopSage.com

Regmon tool, *continued*
logging activity in unprivileged accounts or during logon/logoff, 196
troubleshooting techniques, 195–196
viewing registry activity on idle systems, 193
watching Software Restriction Policies enforcement, 535
RegNotifyChangeKey function, 185, 193
RegQueryValueEx function, 191
RegSaveKey and RegReplaceKey functions, 201
REG_SZ data type, 185
regular queue, cache manager, 688
relationships, binding, 838
relative identifiers (RIDs), 495
reliability, Windows, 35
remote access network service, 839–840
remote desktop connection program (Mstsc.exe), 21
Remote Direct Memory Access (RDMA), 797
remote file system drivers (FSDs), 696
Remote NDIS, 835
remote performance monitoring, 191
remote procedure calls (RPCs), 798–803
implementation, 802
local procedure calls, 171
operation, 799–801
security, 801
service control manager, 224
as system mechanism, 65
remote resources. *See* redirectors, network
remote sessions, Terminal services and, 21–22
Remote Storage Service (RSS), 640
removable media, volume mounting and, 646
remove command, Plug and Play manager, 592
RemoveUsersFromEncryptedFile function, 785
rename operations, delayed, 270
reparse point file, NTFS, 739
reparse points
mount points, 640
NTFS, 722, 739, 758
replacement policy, working set, 462
replay protection, IPSec, 827
replica sets, FRS, 843
ReportFault function, 118
Requests for Comments (RFCs), IPSec, 827
requirements. *See* design goals
reserved pages, 384–385
resident attributes, NTFS, 744–747
resource arbitration, Plug and Play manager, 590
resource kits, Windows
Oh.exe tool, 124, 480
QuickSlice tool, 18, 76
registry help, 183
SrvAny tool, 217
tools, 27
WinDiff tool, 195, 285
WMI scripts, 246
Resource Reservation Protocol (RSVP) service, 837
Resource Utilization, security rating and, 488
resources
allocation policies, 333
Driver Verifier low resources simulation option, 413
executive, 156, 164–165
executive object handles, 135
handles, 6
object manager, 125, 143
objects, 23
processes, 13
remote. *See* redirectors, network
restart area, log file service, 762
Restore files and directories privilege, 524
restoring
from backups, 282
registry, 201
System Restore, 705, 706–711
restricted tokens, 504–505
ResumeThread function, 228
retention, executive object, 126, 141–142
rights, account, 516–518. *See also* privileges
roaming profiles, 189
robustness, Windows, 35
rollback operations, NTFS, 718
Rom pages, 469
root directories. *See also* directories
encryption, 727
NTFS, 738
WMI, 243
root keys, registry, 185, 186–191
Root virtual bus driver, 595
routines. *See also* functions
device driver dispatch, 566–567
key device driver, 548–550
Plug and Play support and device driver, 592
viewing registered fast I/O, 563
Windows, 5
Routing And Remote Access MMC snap-in, 825
RPC Subsystem (RPCSS), 803
RpcImpersonateClient function, 802
Rpcrt4.dll, 802
RSA encryption algorithms, 775, 778
RtlAssert function, 50
RtlGetVersion function, 49
Runas command
executing Regmon tool using, 196
watching user profile loading and unloading with, 189
running state, thread, 335
runs, NTFS, 737, 745
run-time library functions, executive, 65
Russinovich, Mark, 34

S

safe mode, 274–279
boot logging, 277–279
boot option, 261
post-splash screen crashes, 285
starting services in, 226
user-mode programs, 277
SANs (System Area Networks), 618, 642, 796–798
SAPICs (Streamlined Advanced Programmable Interrupt Controllers), 90
Savedump process, 852, 853, 854
scalability, Windows, 46
scatter/gather I/O, 564
ScAutoStartServices function, 225, 229
ScCreateServiceDB function, 223
ScGetBootAndSystemDriverState function, 224
scheduling, thread. *See* thread scheduling
schemas, Active Directory, 840
ScLoadDeviceDriver function, 228
ScLogonAndStartImage function, 227, 228

www.TopSage.com

screen saver, Blue Screen, 870
ScRevertToLastKnownGood function, 230
scripts
  Software Restriction Policies, 534
  WMI, 246
ScShutdownAllServices function, 233
SCSI (Small Computer System Interface), 252, 262, 590, 617, 618, 771
scsi() ARC syntax, 255
Scsiport.sys, 617
ScStartService function, 227, 228
SDK. *See* Platform Software Development Kit (SDK)
SeAccessCheck function, 494
secret service passwords, 227
secrets, Lsass, 489, 530
Section executive object type, 127
section objects
  access-control lists (ACLs), 390
  attributes, 450
  cache manager, 656, 670
  data structures, 453
  memory management, 450–457
  process creation, 304
  prototype PTEs, 441–443
  shared memory and mapped files, 386–388
  viewing, 451
  viewing control areas, 454–457
sectors
  bad-cluster recovery and spare, 771
  boot, 624
  boot process, 253
  defined, 615, 689
secure attention sequence (SAS), 81, 487, 529
Secure Channel (SChannel) SSP, 802
secure desktops, Winlogon, 528
Secure Sockets Layer (SSL), 802, 803
security, 485–535
  Access token executive object type, 127
  account rights and privileges, 516–524
  Address Windowing Extensions, 400
  auditing, 524–526
  AuthZ API, 516
  Common Criteria (CC), 487–488
  components, 488–492
  contexts, 497
  driver-signing policy, 605
  Encrypting File System (EFS). *See* Encrypting File System (EFS)
  EPROCESS blocks, 294
  ETHREAD blocks, 314
  executive object, 135, 140
  executive object services, 130
  explicit file I/O, 702
  file object, 557
  IPSec, 826–828
  job objects, 370
  kernel mode, 17
  logging activity in unprivileged accounts, 196
  logon, 526–533
  mailslot, 806
  memory protection, 383, 38894, 431
  named pipe, 805
  NTFS consolidated security descriptors, 756–758
  NTFS design requirement, 718
  NTFS security file, 738
  object manager, 125
  object protection. *See* object security
  processes, 6, 14, 309
  rating standards, 485–488
  Recovery Console, 279
  registry, 188, 190
  Regmon tool, 192
  remote procedure call (RPC), 801
  Security Accounts Manager (SAM). *See* Security Accounts Manager (SAM)
  security descriptors. *See* security descriptors
  security identifiers. *See* security identifiers (SIDs)
  service accounts, 217–223
  services, 212, 215, 236
  Software Restriction Policies, 533–535
  system code write protection, 860, 863–864
  threads, 12
  troubleshooting access-denied errors, 195
  Trusted Computer System Evaluation Criteria (TCSEC), 485–487
  viewing registry security settings, 491–492
  warning regarding GUI security editors, 514
  Windows and, 23–24
  Windows NT file system, 2
  WMI, 248
  zero-initialized pages, 472
Security Accounts Manager (SAM)
  Active Directory and, 841
  privileges, 521
  security, 489
security descriptors
  access control, 506–515
  attributes, 506
  file object, 557
  NTFS consolidated, 756–758
  registry, 210
  service, 215
  viewing, 508–509
  warning regarding GUI security editors, 514
security editors, warning regarding GUI, 514
security identifiers (SIDs)
  access checks, 495–497
  machine, 218
  quota management, 725
  registry, 188
  viewing, using PsGetSid tool, 496
security inheritance support DLL, 511
security method, executive object, 135
security policies, 190
Security Policy Editor, 533
security quality of service (SQOS), 503
security reference monitor (SRM)
  executive, 64
  executive objects, 141
  initialization, 267, 268, 269
  local procedure calls, 171
  Lsass, 491
  security, 488
Security Support Provider Interface (SSPI) packages, 503
security support providers (SSPs), 801
Security Target (ST) concept, 487
security-descriptor cell data type, 201
SeDefaultObjectMethod function, 493
segment structure, 441, 453
SeLsaCommandPort, 491

www.TopSage.com

Semaphore executive object type, 127
semaphores
  executive resources, 164
  priority boosts after waiting for, 350
send function, 792
serial port boot options, 257, 258, 260, 261
serialization, NDIS, 829
SeRmCommandPort, 491
Server Message Block (SMB) protocol, 697
server versions, Windows, 47–49
servers
  file system driver, 696
  named pipe, 805
  ports, 174
  RPC server name publishing, 801
  server farm online crash analysis, 854
  Winsock, 792
service accounts, 217–223
  interactive services, 221–223
  local service account, 220
  local system account, 218–219
  network service account, 220
  running services in alternate, 220
service applications, 212–217
service control manager (SCM). *See also* services
  auto-start device driver loading, 597
  boot process, 252, 265
  initialization, 271
  network binding, 838
  safe mode boots, 277
  service control programs (SCPs), 236
  services, 6, 223–225
  system processes, 82
service control programs (SCPs)
  service applications, 212
  service control manager, 223
  services, 236
Service Host (Svchost) process
  as shared service process, 233
  WMI provider, 247
service packs, Windows
  symbol files, 28
  system service numbers, 121
service provider interface (SPI), Winsock, 794
Service-0x0-3e7$ windows station, 221
services, 211–236. *See also* device drivers
  accepting boot and last know good, 230–231
  components of, 211
  service dispatching. *See* system service dispatching
  executive object generic, 129
  failures, 231
  interactive, 221–223
  listing installed, 83
  memory manager, 376, 382–400
  network. *See* network services
  service accounts, 217–223
  service applications, 212–217
  service control manager (SCM), 82, 223–225
  service control programs (SCPs), 236
  service descriptor tables, 122–124
  service processes, 38
  shared processes, 233–235
  shutdown of, 232–233
  SrvAny tool, 217
  startup errors, 229
  startup of, 225–229
  system. *See* system services
  viewing, running inside processes, 235
  viewing details inside service processes, 84
  Windows and, 5
Services for UNIX, 53, 60, 61
Services MMC snap-in, 232, 236
Services.exe. *See* service control manager (SCM)
SeSecurityPrivilege and SeAuditPrivilege privileges, 524
SeSinglePrivilegeCheck and SePrivilegeCheck functions, 518
session layer, OSI model, 788
Session Manager. *See* Smss (Session Manager)
session space
  sessions, 414
  viewing, 419
  x86 layouts, 418–421
session working sets, 457
sessions
  session zero, 21
  NetBIOS, 812
  Terminal Services and multiple, 21–22
SetHandleInformation function, 135, 139
SetInformationJobObject function, 363
SetPriorityClass function, 328
SetProcessAffinityMask function, 363
SetProcessShutdownParameters function, 286
SetProcessWorkingSetSize function, 385, 463
SetProcessWorkingSetSizeEx function, 333, 463
SetServiceStatus function, 216
SetThreadAffinityMask function, 363
SetThreadIdealProcessor function, 366
SetThreadPriorityBoost function, 351
Setup function, 603
Setup program, Windows
  boot process, 251, 254
  partitioning, 630
  repair installs, 282
  security identifiers (SIDs), 495
setup programs. *See also* installation
  driver signing, 606
  registry settings, 184
  service applications, 212
  Windows. *See* Setup program, Windows
Setup.log
  determining which HAL is running, 68
  kernel versions, 46
  multiprocessor and uniprocessor file versions, 42
Setupdd.sys, 280
SetupDiEnumDeviceInterfaces function, 551
SetupDiGetDeviceInterfaceDetail function, 551
SetWindowsHook function, 529
shadow copies, 649–656
shared assemblies, 312
shared cache map structures, 669, 670–674, 703
shared executive objects, 144
shared folders, 652
shared memory. *See also* section objects
  Section executive object type, 127
  section objects, mapped files, and, 386–388
  sections, 13
  Windows NT, 2

www.TopSage.com

shared resources, 23, 24. *See also* file objects
shared service processes, 233–235
shell
  boot option, 261
  Explorer.exe as, 82. *See also* Explorer.exe
  initialization, 272
  NTFS shortcuts, 726
  safe mode boots, 277
  user logon, 533
  Userinit.exe, 82
short names, NTFS, 743
short quantum, 341, 344
shutdown
  clearing page file, 445
  overview, 286–288
  service, 232–233
  shutdown levels, 286, 310
side-by-side assemblies, 311–313
signaled state, dispatcher object, 156, 158–160
signature() syntax, 255
signatures, hive, 200, 202
simple volumes, 615
single sign-on, 528
single-layered device drivers, 569–576
size
  cache, 662–668
  hive limits, 198–199
SleepEx function, 109
Small Computer System Interface (SCSI), 252, 262, 590, 617, 618, 771
small memory dump crash dump files, 850
small pages, 382–384
small-IRP look-aside list, 564
Smss (Session Manager)
  Autochk.exe, 645
  boot process, 252, 265, 269–272
  Chkdsk, 278
  creating sessions, 414
  initialization, 269
  Recovery Console, 280
  system processes, 80
sockets, Winsock, 792
soft page faults, 474
soft partitions, 630
soft real-time systems, 102
SoftICE tool, 33
software configuration, registry, 187
software data execution prevention (DEP), 392
Software Development Kit. *See* Platform Software Development Kit (SDK)
software interrupt request levels. *See* interrupt request levels (IRQLs)
software interrupts, 103–109
  asynchronous procedure calls (APCs), 108–109
  dispatch or deferred procedure call (DPC) interrupts, 104–107
  interrupt dispatching, 87
  monitoring activity, 106
software keys, device, 602
software providers, VDS, 649
Software Restriction Policies, 302, 533–535
software resumption, system power states and, 607
spanned volumes, 633
spare sectors, 771
sparse data, compressing, 748–749
sparse files, NTFS, 723, 752
Spcmdcon.sys, 280
special pool option, Driver Verifier, 411–412, 861–863
spinlocks
  functions of, 152–153
  instack queued, 154
  IRP synchronization, 576
  kernel-mode mechanisms vs., 155
  multiprocessor thread dispatching, 358
  queued, 153–154
SQL Server, 815
SRSetRestorePoint and SRRemoveRestorePoint functions, 711
SrvAny tool, 217
stability, Windows, 58
stacks
  creating thread, for processes, 308–309
  heap manager stack traces, 398
  inswapping and outswapping, 376
  IRP stack locations, 566–567, 577, 866
  network, 787
  stack frame exception handlers, 111
  stack trashing and crash dump analysis, 865–866
  viewing device, 577
  viewing thread, 325
  Windows storage stack, 616
standby pages, 381, 440, 442, 469, 472, 475, 667–668
standby state, thread, 335
start address, thread, 319, 324
start command, 328, 331
start I/O routine, device driver, 549
Start value, device driver, 594–595
start-device command, Plug and Play manager, 592
start-of-process function, 112
start-of-thread function, 112
StartService function, 212
StartServiceCtrlDispatcher function, 216, 228
startup. *See* boot process
stateful firewalls, 826
states
  page, 469–471
  PFN entry, 476
  shared page, 442
  system power and device power, 607–608, 609–611
  thread scheduling and thread, 334–338
static WMI classes, 242
stop codes, 846
stop command, Plug and Play manager, 593
storage area networking. *See* System Area Networks (SANs)
storage management, 615–654
  disk drivers, 616–622
  terminology, 615
  volume management. *See* volume management
  watching physical disk I/O, 620
Storport.sys, 617
stream caching, cache manager, 658
stream control blocks (SCBs), NTFS, 731
Streamlined Advanced Programmable Interrupt Controller (SAPIC), 90
streams, NTFS data, 719–721
Streams tool, 720
Strings tool
  looking inside prefetch files, 460
  searching for device drivers by pool tags, 406
  SYSTEM hive corruption problem, 283
  troubleshooting pool leaks, 407
striped volumes, 628, 634, 719
striped volumes with parity, 637

www.TopSage.com

structured exception handling, 109–112. *See also* exception dispatching
Structured Query Language (SQL), WMI scripts and, 246
stub procedures, RPC, 800
subkey-list cell data type, 201
subkeys, registry, 185
subsection structures, section object, 453
subsystem DLLs. *See* environment subsystems
super privileges, 523–524
support images, process creation and, 302
surprise-remove command, Plug and Play manager, 593
SvcCtrlMain function, 223, 225
Svchost. *See* Service Host (Svchost) process
swapper, working sets and, 466
SwitchToFiber function, 13
symbol server, Microsoft, 28, 850
Symbolic link executive object type, 127
symbolic links
  device object, 551, 560
  NTFS, 722–723
  objects, 146
  registry key type, 200
symbols
  kernel debugging, 28, 850
  viewing, using Process Explorer, 11
symmetric encryption algorithms, 776
symmetric multiprocessing (SMP), Windows, 41–46, 150
SYNCH_LEVEL IRQLs, 91, 359
synchronization, 149–166
  deadlock detection, 166, 866
  executive, 65
  heap, 396
  high-IRQL, 151–154
  kernel dispatcher objects, 66
  KTHREAD blocks, 315
  low-IRQL, 155–166
  memory manager internal, 377
  multiprocessing, 44
  multiprocessor thread context switching, 359
  Mutex executive object type, 127
  mutual exclusion, 149–151
  scalability, 46
  synchronous I/O requests to single-layered drivers, 575–576
  type object attribute, 133
  uniprocessor dispatcher database, 339
  viewing global queued spinlocks, 154
synchronous I/O, 561–562
syscall instruction, 120
sysenter instruction, 120
sysexit instruction, 120
sysinternals tools
  Accvio.exe, 117
  Blue Screen screen saver, 870
  ChkReg tool, 284
  Clockres tool, 340
  Dbgview.exe tool, 51
  device drivers, 70
  Diskmon tool, 620
  EFSDump tool, 785
  Filemon tool. *See* Filemon tool
  Handle tool. *See* Handle tool
  Junction tool, 723
  LDMDump tool, 628
  LiveKd tool. *See* LiveKd tool
  LogonSessions tool, 532–533
  Msconfig tool, 273
  Notmyfault tool, 407, 851, 855, 861
  NTFSInfo tool, 739
  Pagedefrag tool, 445
  Pendmoves tool, 271
  PipeList tool, 809
  Process Explorer. *See* Process Explorer
  PsExec tool. *See* PsExec tool
  PsGetSid tool, 496
  Regmon tool. *See* Regmon tool
  Streams tool, 720
  Strings tool. *See* Strings tool
  TDImon tool, 824
  Testlimit tool, 138
  Windows and, 34
  Winobj tool. *See* Winobj tool
sysret instruction, 120
system access-control lists (SACLs)
  assignment, 510
  security descriptors, 506–507
System account, 778
system accounts, 190
system architecture, Windows, 37–51
  checked build versions, 49–51
  checking Ntoskrnl versions, 45
  differences between client and server versions, 47–49
  portability, 40
  scalability, 46
  symmetric multiprocessing (SMP), 41–46
  system components, 39. *See also* system components, Windows
  user-mode and kernel-mode processes, 37–39
  viewing multiprocessor-specific support files on Windows 2000, 43
System Area Networks (SANs), 618, 642, 796–798
system audit and system audit-object ACEs, 507
system cache
  centralized, 656. *See also* cache manager
  data structures, 668–674
  extra memory, 663
  system space, 414
system clock
  boot options, 258, 262
  DPCs, 106
  interrupts, 95
  IRQLs, 96
system code, 414
system code write protection, 860, 863–864
system components, Windows, 51–84
  determining which HAL is running, 68
  device drivers, 69–73. *See also* device drivers
  environment subsystems and subsystem DLLs, 53–63
  examining undocumented interfaces, 73–74
  executive, 63–65
  hardware abstraction layer, 67–69. *See also* HAL (hardware abstraction layer)
  kernel, 65–67. *See also* kernel
  Ntdll.dll library, 63
  stability, 58
  system architecture, 39, 51. *See also* system architecture, Windows
  system processes, 75–84
  viewing base HALs included with Windows, 68
  viewing image subsystem type, 54

www.TopSage.com

viewing installed device drivers, 72
viewing Ntoskrnl.exe and HAL image dependencies, 69
system crashes. *See* crashes
system directory, 630
System Disk Image (SDI) file boot options, 261, 262
system files corruption problem, 282–283
SYSTEM hive, 202, 204, 210, 260, 263, 283
System Information viewer tool, 698
system mapped views, 414
system mechanisms, Windows, 85–182
kernel event tracing, 175–182
local procedure calls (LPCs), 171–174
object manager. *See* object manager
synchronization. *See* synchronization
system worker threads, 166–168
trap dispatching. *See* trap dispatching
Windows global flags, 168–171
system memory pools, 401–413
configuring, 401–404
Driver Verifier, 409–413
look-aside lists, 408–409
monitoring usage, 404–408
types of, 401
system monitoring. *See* Performance tool
System object, 124
system page table entries (PTEs), 415
system power states
mapping device power states to, 609–611
power manager, 607–608
System process, 76–80
system processes, 75–84
idle process, 75
interrupts and DPCs, 76
listing installed services, 83
service control manager (SCM), 82
System process and system *threads*, 76–80
viewing service details inside service processes, 84
Winlogon, Lsass, and Userinit, 81–82
system profile, 188
System Restore
as file system filter driver, 706–711
post-splash screen crashes, 285
restoring SYSTEM hive, 284
System Restore Wizard, 711
system service dispatching, 119–124
32-bit, 119–120
64-bit, 120
kernel-mode, 121
KTHREAD blocks, 315
mode transitions, 17
Regmon tool, 192
service descriptor tables, 122–124
viewing activity, 124
system services. *See also* services; system service dispatching
dispatching and Ntdll.dll, 63
exceptions, 86
executive, 63
Windows and, 5
system shutdown, 286–288, 550
system space
components, 414
page table entries (PTEs), 421
x86 layouts, 417
system startup. *See* boot process
system support processes, 38
system threads, 76–80
system timer. *See* system clock
System tool
crash dump file settings, 849
page file configuration settings, 446
system variables. *See also* kernel variables
cache flush, 684
cache manager mapping and pinning activity, 678
cache manager MDL activity, 679
cache physical size and page fault, 665
cache read activity, 676
cache virtual size and address, 664
fast I/O, 681
intelligent read-ahead, 682
lazy writer, 684
System Volume Information directory, 778, 782
system volumes, 253, 630, 636
system worker threads, 78, 166–168, 683, 687
system working sets, 381, 414, 457, 467–468, 665
System.alt, 210
Systematic Flaw Remediation, security rating and, 488
system-call hooking, Regmon and, 192
system-event monitoring. *See* Windows Management Instrumentation (WMI)
System.log, 210
SystemParametersInfo function, 371
System.sav, 210
systemwide cache data structures, 669
systemwide class registration data, 189
systemwide dispatcher spinlock, 359
systemwide settings, registry, 184, 190
SYSVOL directory, 843

T

tagging, heap, 398
tagging, pool, 404–408
tail checking, 398, 861
Take ownership privilege, 524
targeted DPCs, 105
Task List tool. *See* Tlist tool
Task Manager
examining and specifying process and thread priorities, 331
Idle process in, 76
memory optimizers, 481
setting processor affinity, 363
troubleshooting pool leaks, 407
viewing and changing process affinity, 363
viewing cache physical size, 668
viewing kernel mode and user mode activity, 20
viewing page file usage, 447
viewing process information, 8
viewing system memory information, 378
task offloading, NDIS TCP/IP, 830
Task Scheduler, 459, 461
TCP/IP protocol
extensions, 824–828
iSCSI drivers, 618
NDIS task offloading, 830
NetBIOS, 813
protocol driver, 821
tracing activity using kernel logger, 177
viewing device objects, 822
WSD, 797

www.TopSage.com

TDI (Transport Driver Interface), 789, 823–824. *See also* protocol drivers
TDImon tool, 824
TEB (thread environment block)
  creating, 308
  ETHREAD blocks, 313
  fields, 319
TechNet, 28
temporary files, disabling lazy writer for, 684
temporary objects, 141
Terminal Services
  cache size, 662
  hive size limits, 199
  initialization, 270
  multiple sessions, 21–22
  object manager session namespace, 146–149
  Session Manager, 81
  window stations, 221
TerminateThread function, 347
termination, thread, 336, 347
Testlimit tool, 138
thread dispatching, 104, 152, 157
thread environment block. *See* TEB (thread environment block)
Thread executive object type, 127
thread ID, 12
  thread manager, 64
thread objects, 158
thread scheduling, 325–369. *See also* threads
  context switching, 347
  dispatcher database, 338–340
  examining and specifying priorities, 331
  functions, 330
  idle thread, 348
  job objects, 370
  KPROCESS blocks, 306
  KTHREAD blocks, 315
  multiprocessor, 357–369
  overview of, 326–327
  preemptive, 346
  priority boosts, 348–357
  priority levels, 327–329
  quantum, 340–344
  quantum end, 347
  real-time priorities, 333
  scenarios, 345–347
  termination, 347
  thread states, 334–338
  tools, 331–333
  viewing ready threads, 326
  voluntary switching, 345
  Windows System Resource Manager (WSRM), 333
thread-local storage (TLS), 12
threads, 313–369
  cache manager and system worker, 687
  cache manager's lazy writer and read-ahead, 704
  creating, using CreateThread function, 322
  creating initial, for processes, 308–309
  data structures, 290, 313–320
  fibers vs., 13
  functions, 322
  hung or unresponsive systems, 866
  I/O completion ports, 585
  impersonation and security, 492
  IRQLs, 99
  kernel mode vs. user mode, 17
  kernel variables, 320
  lazy writing and system worker, 683
  memory manager, 376
  memory manager modified and mapped page writer, 704
  multiprocessor scheduling, 47
  performance counters, 321
  priority levels, 92
  processes, jobs, and, 6–14. *See also* job objects; processes
scheduling. *See* thread scheduling
security structures, 526
service, 216
starting execution of initial, for processes, 310
synchronization of. *See* synchronization
system, 76–80
system shutdown, 287
system worker, 78, 166–168, 683, 687
tools for viewing activity of, 323–325
unhandled exception handlers, 112
viewing activity using QuickSlice, 18
viewing outstanding IRPs for, 567
viewing start addresses for, 113
  virtual file I/O, 540
  Windows subsystem, 57
timer, system. *See* system clock
Timer executive object type, 127
timer object, 158
Tlist tool
  idle process in, 76
  mapping service process to services, 82
  viewing process tree using, 7
  viewing services running inside processes, 235
  viewing start addresses for threads, 113
  viewing thread activity, 323
  viewing thread information, 318
tokens, access. *See* access tokens
tools, 25–34
  crash troubleshooting, 860–864
  Device Driver Kit (DDK), 34
  examining memory usage, 378–381
  for exploring object manager database, 124
  Idle process names in, 76
  idle thread names, 348
  kernel debugging, 28–33. *See also* kernel debugger
  monitoring memory pool usage, 404–408
  Performance tool, 27. *See also* Performance tool
  Platform Software Development Kit (SDK), 3, 33, 795
  processes, 6–12
  registry, 183
  sources for, 25–26
  sysinternals, 34. *See also* sysinternals tools
  thread scheduling, 331–333
  thread-related, 323–325
  Windows resource kits, 27. *See also* resource kits, Windows
  Windows Support Tools, 27. *See also* Windows Support Tools
  Windows System Resource Manager (WSRM), 333
topology, network, 788
Tracedmp tool, 177
Tracerpt tool, 177
tracing
  enabling image loader, 170
  Event Tracing for Windows, 175–182

www.TopSage.com

Filemon, 712–717
heap manager stack, 398
Regmon tool, 195–196
tracking logs. *See* logs
Traffic Control (TC) API, 837
transaction table, NTFS, 767
transactions, atomic, 717
transition invalid PTEs, 441
transition pages, 442, 469
transition state, thread, 336
translation, address. *See* address translation
translation look-aside buffer (TLB), 383, 432, 434
TransmitFile function, 793
TransmitPackets function, 794
transport layer, OSI model, 788
Transport Layer Security (TLS), 802
transport provider interface, RPC, 801
transport service providers, Winsock, 794
transports. *See* protocol drivers
trap dispatching, 85–124
exception dispatching. *See* exception dispatching
interrupt dispatching. *See* interrupt dispatching
system service dispatching, 119–124
traps and trap handlers, 85–87, 111
triage dump crash dump files, 850
Triage.ini, 858
trimming, working set, 464
Trojan horse programs, 487, 529
troubleshooting. *See also* debugging; errors; exceptions
crashes. *See* crash dump analysis
file system problems with Filemon tool, 711–717
manually loading and unloading hives, 198
pool leaks, 407
registry problems with Regmon tool, 192–196
system threads, 77
Trusted Computer System Evaluation Criteria (TCSEC), 485–487
Trusted Publishers, Software Restriction Policies and, 534
type objects, 127, 128, 130–133
types
heaps, 395
LDM database, 627
NTFS log record, 763–766
NTFS type codes, 742
object, 22, 124–126
services, 214
viewing, for kernel structures, 31–32

U

UDF (Universal Disk Format) file system format, 691
UNC (Universal Naming Convention), 804, 818
undo operations, NTFS update log records and, 764
undo pass, NTFS recovery, 769–770
undocumented interfaces, 73–74
unhandled exceptions, 112–117
Unicode
configuration manager, 211
NTFS and, 721
Windows and, 25
uniprocessor systems, 42, 339
Universal Naming Convention (UNC), 804, 818
Universal Plug and Play (UPnP) API, 815
UNIX
BSD (Berkeley Software Distribution) Sockets API, 791
Logical Disk Manager (LDM) partitioning, 626
POSIX subsystem, 53, 60–61
Windows and, 36
unknown invalid PTEs, 441
unload routines, device driver, 550
unnamed communication ports, 174
unprivileged accounts, logging activity in, 196
unresponsive systems, crash dump analysis and, 866–869
update records, NTFS, 763–765
uppercase file, NTFS, 738
upper-level filter drivers, 599
USB devices, Remote NDIS for, 835
user accounts
rights, 516–518. *See also* privileges
services, 220, 221
user applications. *See* applications
user credentials. *See* access tokens
USER functions, 39, 56
user logon steps, 529–533
user mode
APCs, 108
AuthZ API, 516
basic types of processes, 38
debugging, 29
device drivers, 542
exception dispatching, 110
kernel dispatcher objects, 156
kernel mode vs., 16–21. *See also* kernel mode
memory pools, 388
Plug and Play (PnP) manager and driver installation, 603–606
preemptive thread-scheduling, 346
safe mode boots, 277
type objects, 133
viewing activity using Performance tool, 19–21
viewing thread activity using QuickSlice tool, 18
Windows operating system model, 36
user profiles
EFS, 776, 778, 783
local system account, 218
registry keys, 187, 188
services, 228
watching loading and unloading of, using Regedit tool, 189
User Profiles management dialog box, 188
User State Migration Wizard, 313
user virtual address space layouts, 415
User32.dll, 39, 56
Userenv.dll, 228
Userinit.exe
initialization, 272
safe mode boots, 277
system processes, 81–82
user logon, 533
user-interface limits, job objects and, 371
user-mode process dump files, 29
usernames, 530
users registry key, 187
Usrclass.dat, 189

V

VACB index array, 670
valid page table entries (PTEs). *See* address translation
valid pages, 442, 469

www.TopSage.com

validation, access. *See* access checks
validation, heap, 398
value cell data type, 201
value-list cell data type, 201
values, registry, 185
variable quantum, 344
variables. *See* kernel variables; system variables
VCN-to-LCN mapping, NTFS, 737, 746
VDS Basic Provider, 649
VDS Dynamic Disk Provider, 649
vendor ID (VID), 600
VenturCom, 102
verbose kernel debugger crash dump analysis, 858–859
VerifyVersionInfo function, 49
VERITAS Software, 626
version number, NTFS volume, 732
versions, Windows
  blue screen stop codes, 847
  cache manager worker queue items, 688
  cache size, 662
  checked build versions, 49–51
  configuration manager and hive memory usage, 203
  default initial quantum, 306
  differences between client and server, 47–49
EFS, 777
event tracing, 177
exception dispatching, 112
executive object types, 127
explicit file I/O, 701
FAT file system, 691
handle tables, 137
hardware abstraction layer (HAL), 543
hive data formats, 200
hive handles, 199
hive size limits, 198
hive stable storage, 210
idle processor selection, 367
initial system worker threads, 167
LiveKd tool, 32
memory manager synchronization, 377
multiprocessor dispatcher databases, 358
multiprocessor systems, 41
multiprocessor thread scheduling, 368–369
NDIS, 828
NTFS volumes, 732
object manager session namespace, 147
PEB values, 307
process initialization, 310
prototype PTEs, 441
Registry Editor tools, 184
Regmon device drivers, 192
remote sessions, 21
resource kits, 27
security ratings, 488
system space page table entries (PTEs), 421
temporary page files, 445
thread dispatcher database, 338
thread-scheduling quantum, 340, 341, 342
Unicode and ANSI strings, 25
user address space sizes, 415
user-mode debugging, 29
user-mode memory access, 388
Windows NT code base, 1–2
Windows Support Tools, 27
Winsock operation, 792
WMI implementation, 247
video calls, 814
video driver boot options, 257, 260
views
  Address Windowing Extensions, 399
  cache manager, 660, 670
  section object, 387
  system mapped, 414
virtual address control blocks (VACBs), 669, 703
virtual address descriptors (VADs)
  EPROCESS block, 293
  invalid PTEs, 441
  memory management, 448–450
  processes, 14
virtual address space, 413–422. *See also* virtual memory
  64-bit address space layouts, 422
  basic components, 413–415
  boot options, 257, 262
  pages. *See* pages
  process, 305, 306
  physical memory vs., 375
  system page table entries (PTEs), 421
  x86 session space layouts, 418–421
  x86 system address space layouts, 417
  x86 user address space layouts, 415
virtual address translation. *See* address translation
virtual block caching, 658
virtual cluster numbers (VCNs), 733
virtual device drivers (VDDs), 542
Virtual Disk Service (VDS), 648
virtual files, 540
virtual memory. *See also* virtual address space
  boot options, 257, 262
  cache manager, 660–661
  EPROCESS block, 293
  executive, 65
  job objects, 370
  kernel mode, 17
  management. *See* memory; memory management
  pools. *See* system memory pools
  processes, 6, 14, 305, 306
  threads, 12, 13
  Windows and, 14–16
virtual private network (VPN) remote access, 840
virtual size, cache, 663–664
VirtualAlloc function, 384, 395, 399, 417
VirtualAllocEx function, 384
VirtualFree and VirtualFreeEx functions, 384
VirtualLock function, 385
virus scanners, 705
volatile hives, 197
volume management, 622–654. *See also* volumes
  bad-cluster recovery, 771
  basic disks, 624–626
  basic vs. dynamic disks, 622
  data redundancy and fault tolerance, 719
  DMIO driver, 631–632
  drivers, 545, 584
  dynamic disks, 626–632
  evolution of, 623
  FtDisk driver, 625
  multipartition, 632–638
  Virtual Disk Service (VDS), 648
  volume I/O operations, 646–647
  volume namespace, 638–646
  Volume Shadow Copy service, 649–654
volume namespace, 638–646
  Mount Manager device driver, 638–639
  mount points, 639–642

www.TopSage.com

viewing volume parameter blocks (VPBs), 643
volume mounting, 642–646
volume parameter blocks (VPBs), 642–644, 695
Volume Shadow Copy service, 649–654
volumes
boot, 263
boot and system, 630
boot process, 251
cluster sizes and FAT, 691, 693
LDM entries, 628
management. *See* volume management
multipartition, 615, 632–638
NTFS. *See* volumes, NTFS
partitions vs., 625. *See also* partitions
recoverable file systems, 658
shadow copy service, 649–654
simple, 615
spanned, as volume sets, 633
volume objects, 646
volumes, NTFS, 732–758
change journal file, 752
cluster sizes, 694
clusters, 732
consolidated security descriptors, 756–758
data compression and sparse files, 747–752
disk configuration, 732
fault-tolerant, and data recovery, 774
file records, 740–742
file reference numbers, 739
filenames, 742–744
indexing, 753–754
master file table (MFT), 733–739
mounting, 644
object IDs, 754
quota tracking, 725, 755
recovering. *See* recovery, NTFS
reparse points, 758
resident and nonresident attributes, 744–747
spanned volumes, 633
viewing information about, 739
voluntary switching thread scheduling, 345

W

W32PROCESS blocks, 294, 310
W32THREAD blocks, 313
wait blocks, 161
wait functions, thread, 341, 345
wait hints, service, 233
wait queues, viewing, 162
wait state, thread, 335, 345
wait states, alertable, 109
WaitForMultipleObjects function, 161
WaitForMultipleObjectsEx function, 109
WaitForSingleObject and WaitForMultipleObjects functions, 156
Wake-on-LAN, NDIS, 830
waking states, 608
Watson.Microsoft.Com, 119, 854
WbemTest tool, 243
Web access APIs, 803–804
Web-Based Enterprise Management (WBEM) standard, 237
WebDAV driver, 697
well-known addresses, 794
well-known names, 551
Win16 applications, 302
Win32 API, 5. *See also* Windows API
Win32k.sys (Windows subsystem kernel)
process data structures, 290
service descriptor table, 122
sessions, 414
system space, 414
Windows subsystem, 39, 43, 53, 56, 270
Win32_NTEventlogFile class, 241
Win32_NTLogEvent class, 244
Win32_OperatingSystem class, WMI, 50
WinDbg tool, 29, 32, 116. *See also* kernel debugger
WinDiff tool, 195, 285
window manager, 56
window stations
interactive services, 221, 228
object manager, 147
security rating, 487
Winlogon initialization, 528
windowing
priority boosts after GUI threads wake up, 353–354
Windows subsystem, 39, 55
Windows
API, 3–5. *See also* Windows API
boot process. *See* boot process
cache manager. *See* cache manager
concepts and terms, 3–25
crash dump analysis. *See* crash dump analysis
Device Driver Kit. *See* Device Driver Kit (DDK)
file systems. *See* file systems
I/O system. *See* I/O system
kernel mode vs. user mode, 16–21. *See also* kernel mode; user mode
management mechanisms. *See* registry; services; Windows Management Instrumentation (WMI)
microkernel-based systems vs., 36
.NET Framework and WinFX, 4
networking. *See* networking
objects and handles, 22. *See also* handles; objects
operating system model, 36–37
processes, threads, and jobs, 6–14. *See also* job objects; processes; threads
real-time processing, 102, 333
requirements and design goals, 35
security, 23–24. *See also* security
services, functions, and routines, 5. *See also* functions; routines
shutdown, 286–288
storage management. *See* storage management
subsystem, 39, 53, 55–57. *See also* Csrss (Windows subsystem process); Win32k.sys (Windows subsystem kernel)
system architecture. *See* system architecture, Windows
system components. *See* system components, Windows
system mechanisms. *See* system mechanisms, Windows
Terminal Services and multiple sessions, 21–22
tools, 25–34. *See also* tools
Unicode, 25
versions, 1–2. *See also* versions, Windows
virtual memory, 14–16. *See also* memory; memory management; virtual memory
Windows NT vs. Windows 95, 2

www.TopSage.com

Windows 95, 2
Windows 98, 2
Windows 2000. *See also* versions, Windows
  Common Criteria rating, 487
  event tracing, 177
  handle tables, 137
  initial system worker threads, 167
  object manager session namespace, 147
  Process Control Manager, 371
  remote sessions, 21
  resource kits, 18, 27
  unhandled exceptions, 116
  Unicode, 25
  viewing multiprocessor-specific support files on, 43
  Windows File Protection, 311
  Windows NT and, 1
Windows 2000 Server Operations Guide, 27
Windows API
  memory manager functions, 382
  memory protection options, 389
  security, 24
  Windows and, 3–5
Windows Backup. *See* Backup tool
Windows Debugging Tools, 29–32, 848. *See also* Tlist tool
Windows Driver Model (WDM), 64, 71, 542, 538
Windows Error Reporting, 118–119, 853
Windows Explorer, 45
Windows File Protection (WFP), 282, 283, 311
Windows Hardware Quality Lab (WHQL), 539, 605
Windows Installable File System (IFS), 63, 655, 695
Windows Installer Package problem, 715
Windows Internet Name Service (WINS), 820
Windows Management Instrumentation (WMI), 237–249
  architecture, 237–239
  checking for checked build version using, 50
  class association with association classes, 244
  Common Information Model (CIM) and Managed Object Format (MOF) language, 240–243
  executive, 64
  implementation, 247–248
  initialization, 268
  I/O system, 538
  managing systems using WMI scripts, 246
  mechanisms, 237
  namespace, 243–244
  NT Kernel Logger, 175
  providers, 239–240
  security, 248
  viewing MOF definitions of classes, 243
  Wmic.exe tool, 248
Windows Media Player, 357
Windows Millennium Edition, 2
Windows Networking (WNet) API, 815
Windows NT. *See also* versions, Windows
  C2 security rating, 486
  naming conventions and device objects, 621
  networking APIs, 841
  requirements and design goals, 35
  volume management, 623
  Win32 API, 5
  Windows 95 vs., 2
Windows Script Host (WSH), 246, 534
Windows Security dialog box, 82
Windows Server 2003. *See also* versions, Windows
  deferred ready thread state, 338
  event tracing, 177
  guarded mutexes, 164
  handle tables, 137
  hive lazy writer delay setting, 209
  initial system worker threads, 167
  Message Queuing, 802
  object manager session namespace, 147
  remote sessions, 21
  Shadow Copies for Shared Folders, 652
  Virtual Disk Service, 649
  Windows NT and, 1
  Windows System Resource Manager (WSRM), 333
Windows Server 2003 Deployment Kit, 183
Windows Server Network Monitor Tools, 834
Windows Services MMC snap-in, 220
Windows Sockets Configuration tool, 795
Windows Sockets Direct (WSD), 796–798
Windows Support Tools, 27. *See also* Poolmon tool; Tlist tool
Windows Sustained Engineering Team, 488
Windows System Resource Manager (WSRM), 333
Windows Update, 606
Windows XP. *See also* versions, Windows
  dirty page threshold, 687
  event tracing, 177
  handle tables, 137
  initial system worker threads, 167
  keyed events, 160
  logical prefetcher, 458
  object manager session namespace, 147
  post-splash screen crashes, 285
  push locks, 165
  real-time processing, 102
  remote sessions, 21
  restoring SYSTEM hive, 284
  service accounts, 217
  shared assemblies, 312
  System Restore, 706–711
  Volume Shadow Copy service, 649
  Windows NT and, 1
WindowStation executive object type, 128, 134
WinFX, 4
WinHTTP API, 803
WinInet (Windows Internet) API, 803
Winlogon
  access tokens, 497
  boot process, 252, 265, 269–272
  extracting crash dumps, 852
  initialization, 528–529
  interactive logon, 527
  local procedure calls, 171
  registry, 188
  security, 489
  security rating, 488
  services, 230
  sessions, 414
  SIDs for logon sessions, 496
  system processes, 81–82

www.TopSage.com

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java 一览无余：**Java** 视频教程 | **Java SE** | **Java EE**

.Net 技术精品资料下载汇总：**ASP.NET** 篇

.Net 技术精品资料下载汇总：**C#**语言篇

.Net 技术精品资料下载汇总：**VB.NET** 篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书+视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI 脚本语言编程学习资源下载地址大全

Python 语言编程学习资料(电子书+视频教程)下载汇总

最新最全 **Ruby**、**Ruby on Rails** 精品电子书等学习资料下载

数据库精品学习资源汇总：**MySQL** 篇 | **SQL Server** 篇 | **Oracle** 篇

最强 **HTML/xHTML**、**CSS** 精品学习资料下载汇总

最新 **JavaScript**、**Ajax** 典藏级学习资料下载分类汇总

网络最强 **PHP** 开发工具+电子书+视频教程等资料下载汇总

UML 学习电子资下载汇总 软件设计与开发人员必备

经典 **LinuxCBT** 视频教程系列 **Linux** 快速学习视频教程一帖通

天罗地网：精品 **Linux** 学习资料大收集(电子书+视频教程) **Linux** 参考资源大系

Linux 系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX 操作系统精品学习资料<电子书+视频>分类总汇

FreeBSD/OpenBSD/NetBSD 精品学习资源索引 含书籍+视频

Solaris/OpenSolaris 电子书、视频等精华资料下载索引