Broadview®

Windows
内核原理与实现

潘爱民 著

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

Windows内核原理与实现

Windows操作系统已有20年历史，尽管市面上存在大量讲述Windows技术的文章和书籍，但讲述操作系统底层机理的并不多，本书则从Windows内核源代码出发，诠释其核心技术。

本书从操作系统原理角度组织材料，深度解析了Windows内核如何实现现代操作系统的各个关键部件，并直接以Windows内核源代码（WRK, Windows Research Kernel）为参照。在内容选取方面，侧重Windows内核中基本的系统部件，同时也兼顾作为一个操作系统的完整性。在讲解代码实现的同时并未局限于代码本身，而是着重阐述了Windows内核中运用的算法和机制。

无论您是操作系统课程的学生，还是从事系统软件开发的软件工程师，又或是拥有技术背景的Windows系统管理员，都能通过阅读本书获得对Windows更为全面和深入的理解。

通过本书您可以了解到以下有关Windows的内容：

- 内核发展历史和操作系统基础概念
- 对象管理器、注册表、内核日志记录等公共管理设施
- 进程和线程的管理
- 内存管理
- 并发和同步机制
- I/O模型
- 存储模型
- 系统服务分发和重要的跨进程通信机制
- 网络模型
- Windows子系统
- Windows Vista和Windows 7系统的重要变化

策划编辑：周　筠
技术编辑：卢鹄翔　白　涛
责任编辑：陈元玉
责任美编：杨小勤

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

图书分类：操作系统

ISBN 978-7-121-10528-9

9 787121 105289 >

定价：99.00元

Windows内核原理与实现

潘爱民 著

電子工業出版社
Publishing House of Electronics Industry
北京 • BEIJING

内容简介

本书从操作系统原理的角度，详细解析了 Windows 如何实现现代操作系统的各个关键部件，包括进程、线程、物理内存和虚拟内存的管理，Windows 中的同步和并发性支持，以及 Windows 的 I/O 模型。在介绍这些关键部件时，本书直接以 Windows 的源代码（WRK, Windows Research Kernel）为参照，因而读者可以了解像 Windows 这样的复杂操作系统是如何在 x86 处理器上运行的。

在内容选取方面，本书侧重于 Windows 内核中最基本的系统部件，同时也兼顾到作为一个操作系统的完整性，所以，本书也介绍了像存储体系、网络、Windows 环境子系统等，这些虽然并不位于内核模块但却支撑整个 Windows 运行的重要部件。在本书最后，也介绍了 Windows Server 2003 以后的内核发展和变化。

虽然书中有大量关于 Windows 代码实现的描述，但是本书并没有罗列 WRK 中的代码，即使读者不对照 WRK 的源代码，也可以从这些章节的描述中理解 Windows 的实现机理。在每一个技术专题的介绍中，本书几乎都提供了一个框架图，并且有关键细节的实现分析，这样做的意图是让读者既能够对一项技术有总体上的把握，也通晓关键的实现细节。

Windows 操作系统已经有 20 年历史了，市面上有大量关于 Windows 技术的文档和书籍，但是，真正从源代码来诠释 Windows 底层机理的，本书还是第一次尝试。在本书覆盖的内容中，有相当一部分是第一次以文字形式披露出来的，期望这些内容能消除人们对于 Windows 的神秘感。

写作本书的目的是让对 Windows 有好奇心的人真正了解到 Windows 中的核心机理，让计算机专业的学生和老师，以及系统软件工程师可以快速地领略到 Windows 中先进的系统技术，以及在 Windows 上编写出更加高效的软件。本书也配备了一些小工具，通过这些小工具，读者可以查看内核中的静态或动态的信息，甚至观察系统的行为，可通过 Internet 下载这些工具。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Windows 内核原理与实现 / 潘爱民著.—北京：电子工业出版社，2010.5
ISBN 978-7-121-10528-9
Ⅰ.W… Ⅱ.潘… Ⅲ.窗口软件，Windows—程序设计… Ⅳ.TP316.7
中国版本图书馆 CIP 数据核字（2010）第 043898 号

策划编辑： 周　筠
技术编辑： 卢鸶翔　白　涛
责任编辑： 陈元玉
印　　刷： 北京智力达印刷有限公司
装　　订： 三河市皇庄路通装订厂
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：** 44.75　　**字数：** 850 千字
印　　次： 2010 年 5 月第 1 次印刷
印　　数： 6 000 册　　**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，联系及邮购电话：（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。

序 一
Foreword

The Windows Operating System has been evolving, along with the PC-based computer. Although systems based on UNIX have dominated in the universities since the 1970s, Windows has completely surpassed UNIX in commercial systems. Yet relatively little information about the internals of Windows – the architecture and code of the kernel – has been available outside of Microsoft.

With the support of Microsoft's executive management (particularly Rob Short, Jim Allchin, and Bill Gates), my colleagues Arkady Retik, Chris Fagan, and I set out to make the sources for the Windows kernel available to faculty, researchers, and students throughout the world. We wanted them to have the opportunity to examine the internals of the kernel and understand its implementation and thus be able to decide for themselves what is admirable about the Windows kernel – and what is not.

This project became the Windows Research Kernel (WRK), and is part of the Windows Academic Program (WAP www.microsoft.com/WindowsAcademic). The WAP also includes a large amount of teaching materials in the Curriculum Resource Kit (CRK). Effectively the WRK is the Windows kernel in source form, and the CRK is the Windows kernel in PowerPoint (with exercises, experiments, projects, and other learning aids).

All along we knew that the WRK and CRK by themselves would not make the Windows kernel as accessible as we wanted it to be. It would take books like *Understanding the Windows Kernel*[1] to draw together and organize the ideas in the kernel and explain the source code.

Our intention with the Windows Academic Program has never been to advocate that Windows supplant UNIX in the teaching of operating systems in universities. We believe that both the UNIX and the Windows kernel should be taught, so that it is

[1] *Understanding the Windows Kernel* 为本书英文书名。——编者注

possible to compare and contrast the architecture and design choices that are evident in both systems.

As you will see from this book, Windows has made some very different choices than UNIX. In large part this is because Windows was developed to target a very different system and user environment than UNIX did when it was created. Windows would run on multiprocessors, so threads are the unit of scheduling rather than processes. The systems would have at least 32 bits of pageable virtual address space, so the Windows kernel had the opportunity to build general purpose mechanisms such as the object manager and stackable I/O. Windows expected to run in a very dynamic environment, so the object name space resides outside the file system and is maintained by the kernel itself. Although not a microkernel, Windows did incorporate the Mach idea of multiple operating system personalities in user-mode "subsystem" processes, initially supporting OS/2, POSIX, and Win32 interfaces.

The internal architecture of the kernel reflects learning from UNIX, Mach and VMS, Digital's VAX operating system that was also designed by Dave Cutler, the senior architect of the Windows OS. There is a layer within the kernel which abstracts and schedules the CPUs (as threads, interrupts, traps, etc). This "kernel of the kernel" layer is about 5% of the code. The rest of the kernel is in the "executive" layer (including the device drivers, file systems, and network code which run outside of the kernel proper). The executive is implemented as fairly ordinary multithreaded code, which greatly simplifies the expertise required by kernel-mode developers.

For those who learned operating systems by reading and understanding only the ideas in UNIX, learning about Windows creates a great opportunity to evaluate their own assumptions about what an operating system should be. This is important because the nature of computer systems continues to rapidly evolve. UNIX was designed for 16-bit, low-memory, swapping systems. Windows for 32-bit systems with paged virtual addressing and megabytes of physical memory. But contemporary systems are 64-bit with gigabytes of memory, and many new features like virtualization hardware, solid-state disks, and integrated computational and graphics accelerators. Connectivity has evolved from the dial-up modems at UNIX's birth in 1969 to the 10 Mbps local Ethernet at Windows birth in 1989. And now we have multiple Gbps networks interconnecting computers around the world. Will a new operating system emerge which targets these new hardware environments? Or will existing systems just continue to evolve their capabilities? It is only by having a broader understanding than

either just UNIX or just Windows that software engineers will be able to judge for themselves.

Our colleague, Aimin Pan, is one of the bright stars in the new generation of computer scientists teaching and researching some of these important questions about the future of operating systems. His presentation of the Windows kernel will be a great asset for those who want to also participate in the evolving future of operating systems, as well as those who are just simply curious, or want to better understand the software architecture that they are using daily.

Dave Probert, Ph.D.
compsci@microsoft.com
Windows Kernel Architect – Microsoft
Developer of the Windows Research Kernel release
14 February 2010

序二

Windows 作为用户最多的主流操作系统，备受广大师生和工程技术人员的关注，大家希望能通过深入内核的实验和分析源代码的方式，来透彻地了解 Windows 操作系统。微软亚洲研究院认真地考虑了这一需求，并和微软 Windows 产品部的专家一起做了大量的努力，终于在 2005 年促成了微软公司面向教育界发布的 Windows 学术支持计划，其中就包括了为教育和科研提供 Windows 内核源代码的访问许可 WRK（Windows Research Kernel）。

我们围绕 WRK 进行了一系列的教师培训，大家认为 WRK 的发布是深入了解 Windows 操作系统内核的一个里程碑，但同时也反映出一个问题：WRK 包含大量的源代码，如何阅读和分析它们，并将这些源代码与教学中的知识结构对应起来，是进一步使用 WRK 的瓶颈。

潘爱民博士写的《Windows 内核原理与实现》这本书很好地解决了这一问题。潘博士采取将全局系统框图和关键实现分析相结合的方式，详细解析了 Windows 对现代操作系统的各个关键部件的实现。书中采用了 Windows 操作系统的真实数据结构，并将操作系统的实现机理与 WRK 中的源代码描述对应了起来。书中附带有很多分析工具，使读者可以方便地跟踪和观察操作系统内核的运行过程。

潘爱民博士是软件技术和操作系统方面的专家，他从 2004 年开始在微软亚洲研究院从事系统性能分析的研究工作，在 Windows 操作系统的内核优化方面有着丰富的知识和经验。潘博士承担了《Windows Internals, 4th Edition》一书中文版的翻译工作，他还参与了很多与操作系统有关的高校合作项目，通过教师培训和指导学生，对 Windows 操作系统在教学和科研中的实际需求有着深入的了解。

本书是潘爱民博士对 Windows 操作系统以及 WRK 深入研究的结晶，代表了微软亚洲研究院的科研人员对探究技术和帮助中国教育的极大热情。我很高兴看到本书的顺利出版，它是一本很好的操作系统教学参考书；对广大希望深入了解 Windows 操作系统的读者来说，它是一本很好的学习用书。

微软亚洲研究院 副院长 宋罗兰

2010 年 2 月 13 日

岁月如歌

—出版人感言—

3 年前，潘老师在我的 CSDN 博客上留言：

我对翻译的热情越来越淡了，也希望出版界多做一些本土作家的精品，原文版的图书尽可能直接引进。这对国内工业界是有长远意义的。

当年，潘老师放弃了一本曾经由他翻译的经典 C++著作的新版翻译，选择翻译吃力且未必讨好的《Windows Internals，第四版》，取舍之间，让我再次看到他的得失观和价值观，看到他于国内 IT 业界的长远眼光。

11 年前，潘老师自己的原创作品《COM 原理与应用》就得到了业内的诸多好评，若干年后还是不断有读者提起潘老师的这本处女作，并问潘老师怎么不写新的书。

侯捷老师在一篇文章里，提到翻译和原创技术图书，说：这如同歌手唱歌，翻唱别人的歌，总比写出自己的原唱歌曲要容易。

10 年前，国内好的翻译图书都不多，潘老师和侯捷老师、裘宗燕老师这批人，以自己的勤奋认真，开创了一代有品质的技术图书的写译先河，为后来者树立起高标杆，善莫大焉，功莫大焉。但当时限于积累，原创图书还只能让位于翻译图书。

10 年间，武汉博文团队在成立的 7 年里，一点点践行着潘老师的期望：出版界多做一些本土作家的精品。

去年，由潘老师策划组织的原创好书《程序员的自我修养——链接、装载与库》由武汉博文出版。上市后即受到许多技术高手的好评。该书繁体版 2009 年底在台湾上市后，连续三个月位居畅销榜第一名。侯捷老师认真读过这本书后，给予了高度赞誉。

今年，武汉博文再次得到潘老师的信任，获得了潘老师独立创作的《Windows 内核原理与实现》的出版权。我和我的同事深感责任重大，不敢有丝毫懈怠。为了做好这本书，

我和同事们较早开始做各种准备。我付出的辛苦是最少的，担任责任编辑的陈元玉，担任技术编辑的卢鹄翔和白涛，担任设计任务的小勤和文佳，他们都比我劳累。

然而，出书的辛苦远不能和写书相比。多少个凌晨，我已从梦中醒来，潘老师却才入睡。他也一次次地和我商量着写书的细节，担心自己写的不够好。他对自己书稿的要求，远比编辑要高。我知道他身边还有着依恋爸爸的幼儿，但他牺牲了很多陪伴孩子和太太的时间。

孤灯一盏，孜孜不倦辛勤写作技术性极强而又极有价值的著作，希望弥补国内业界和出版界的空白。这种苦，不是什么人都能品尝的，没有足够的技术积累，没有内心深沉的热情，没有仔细的筹划，没有 Just do it 的执行力，难以坚持到最后一刻。惟其如此，这种种艰辛中的快乐和欣慰，也只有少数人有资格品味。

潘老师常常情不自禁地和我提起他的老师王选，这位以一己之力让一个行业发生了巨变的大侠，有着非凡的人格魅力。从潘老师身上，我看到：一个年轻人，在他初出茅庐之时，若得遇王选那样的智者指路，他的一生将会受到怎样深远的影响。

生命如歌，岁月如歌。

周　筠

2010 年春于武汉

致谢

首先，要感谢在微软工作的人，没有他们的鼎力支持和辛勤努力，这本书的内容和选题便不复存在：

- 感谢 Dave Probert 提供了一套可编译和运行的 Windows 源代码，将 Windows 的最新技术成果拿出来与社会分享，而且，每次到中国来，都会将自己在操作系统领域中的最新工作结果介绍给我。感谢 Arkady Retik 一直在 WAP（Windows Academic Program）项目上辛勤地工作，不遗余力地将 Windows 的教学资源在全球范围内推广。
- 感谢宋罗兰女士，她和她领导的高校关系团队长期以来支持我走到学生中间，让我始终能够贴近高校的氛围，并接触到中国最为优秀的学生。感谢张高和马歆，他们一直在努力让中国的大学生有更多的机会接近微软的技术，并有机会融入到微软的大家庭中。感谢历任 WinCore 工作组的负责人，邹静、李贝、Bill Luan、黄婷婷、景霓和 John Warren，他们的辛勤工作使国内更多教师和学生能获得 WRK 及相关教学资源。
- 感谢我的老板李世鹏博士，允许我在研究工作之余钻研 WRK，并支持我从事 Windows 系统性能方向的研究。感谢我的同事朱斌博士，作为最紧密的工作伙伴，对我所做的每一个项目都给予了最大力的支持。感谢微软亚洲研究院前院长沈向洋博士，他长期以来一直支持和鼓励我做任何感兴趣的研究题目，并把最优秀的学生交给我来指导。
- 感谢微软公司和微软亚洲研究院的众多同仁，让我在浓厚的技术氛围中感受着 Windows 无与伦比的魅力。我有机会经常跟一些同事讨论关于 Windows 核心技术细节，他们是：Stephen Hsiao、Jin Jia、Qiang Chen、Bin Zhao、党映农、刘未鹏。

其次，感谢过去几年中跟着我钻研 Windows 内核并从事 Windows 系统性能研究的实习生们，他们是：李子拓、黄晓晨、韩铮、郭津之、杨晨、武健、龙海、王嘉捷、张开敏、曹家鑫、潘震皓、齐鸣、高翔，其中高翔和曹家鑫编写了本书中用到的 5 个工具，潘震皓实现了附录 B 中介绍的 KInjectToolKit 项目。

成书的过程并不轻松，从立题一直到初稿完成，再到最后的审稿、修改和制作，都是在与出版社编辑人员的紧密合作下完成的。尤其在最后审稿和制作过程中，与几位编辑按流水线方式通力合作，既缩短了编审周期，也保证了终稿质量。

- 感谢周筠女士，她和她的团队是如此地让人信任，因而我一开始就找到她，期望她能出版这本书。事实证明，我的选择是正确的。谢谢周老师为这本书倾注的热情，以及对我一贯的信任。
- 感谢责任编辑陈元玉女士，从最初商定版式，一直到后期制作的所有事宜，她都事无巨细，承担了所有可能提供给我的编辑支持，让我可以安心地关注在本书内容上。
- 感谢技术编辑白涛和卢鹤翔，两位编辑夜以继日地检查书中的每一个细节，使书中的技术不精确程度降低到最小，也纠正了很多笔误。
- 感谢设计师杨小勤，这本书的封面以及书中的部分插图均出自他手。Windows 的内核就像一个交响乐队，复杂、深邃而又和谐，让上层应用程序可以安心和高效地运行。同时感谢为本书设计版式的设计师胡文佳，谢谢她的细心和耐心。

最后，感谢我的家人，我的妻子孙红芳女士和儿子安安在我写作本书的两年间对我的大力支持。这两年正好是安安从一岁到三岁的成长期，我在这本书上所花的时间有相当一部分本该是陪着他们的。

潘爱民

2010 年 3 月于北京西二旗

前言

《Windows 内核原理与实现》是一本讲述 Windows 内核机理，并配合 Windows 的源代码来解释其实现细节的学习用书。全书从操作系统原理的角度来组织内容。本书适合于已经有了操作系统基本概念的读者进一步理解 Windows 操作系统。通过学习本书的内容，读者不仅可以掌握 Windows 的核心机制，也可以理解像 Windows 这样的现代操作系统是如何构建起来的。因此，这是一本 Windows 操作系统内核学习用书，而不是一本指导在 Windows 平台上进行软件开发的书。

对于绝大多数 IT 从业人员及高校学生来说，Windows 已经是一个熟悉得不能再熟悉的操作系统了，而且也有相当多的人能够熟练地开发各种类型的 Windows 应用程序，然而，真正熟悉 Windows 内部机理的人却少而又少。究其根源，很多人将这归咎于 Windows 是一个闭源操作系统，也就是说，除了 Microsoft 内部的员工以外，人们接触不到 Windows 的源代码。这样的解释只能说是部分正确的，因为从历史发展来看，UNIX 类操作系统确实有广泛的群众基础，很多人得益于阅读各种 UNIX 版本的源代码（包括 Linux 和 FreeBSD）。但对于 Windows 操作系统，这似乎并非一个合理的解释，因为事实上，有相当多的人在过去三四年间已经获得了 Windows 的源代码，但是在代码阐释和核心深入理解方面却并未表现出拥有源代码而带来的优势。反倒是在过去几年间基于逆向工程而获得的 Windows 核心知识要深刻得多。

这么说并非指 Windows 源代码没有帮助，而是间接地揭示了一个事实：Windows 的源代码不是那么容易读的，Windows 的核心也不是那么好理解的。本书作者在过去几年间，与国内高校的操作系统课程的老师和学生有着广泛的接触和了解，真切地感受到，要想在操作系统课程中完全融入 Windows 的内容，迫切需要一本按照操作系统基本概念和理论来阐释 Windows 实现机理的书籍。基于这样的动机，本书力图让学习操作系统的读者能够理解 Windows 中的核心机制，并且可以近距离地观察到 Windows 内核中的实现细节。

本书针对的读者

对 Windows 内核有好奇心的读者，无论您是操作系统课程的学生，或是正在从事系统软件开发的软件工程师，或是有技术背景的 Windows 系统管理员，都可以从本书中找到很多问题的答案。虽然本书中描述的大多数细节是以 WRK（Windows Research Kernel）的代码为基础，但是，对于各种核心机制的介绍却是通用的，并不限于 WRK 内核，甚至适用于各种版本的 Windows，包括最新的 Windows Vista、Windows Server 2008 和 Windows 7。

对于有资格获得 WRK 的读者（包括高校老师和学生），本书可以是配套的学习参考用书。本书正文部分列出了 WRK 中一些关键的数据结构定义，而在介绍各种核心机制时，明确地指出了源代码中的函数调用关系、关键数据的含义和用途，甚至在有些地方指明了 WRK 中对应的代码片段。因此，配合 WRK 代码来阅读本书是最理想的阅读方式，就如同本书作者在写作过程中经常查看 WRK 代码一样。

然而，对于因某些原因而无法获得 WRK 的读者，本书仍然有指导意义。首先，本书在讲解 Windows 内核的核心机制时，几乎都以文字或图形的方式描述了这些机制的总体结构，并指出了关键的数据结构或函数。即使没有 WRK 代码的参照，读者也可以想象出这些函数的功能实现。其次，如果读者善用 windbg，并设置 Microsoft 提供的公共符号服务（参考本书附录 A），则可以在内核调试环境中查看到很多实现细节。书中描述的绝大多数核心机制不仅适用于 Windows Server 2003 SP1（即 WRK 内核适用的系统），也适用于其他的 Windows 版本，包括 Windows XP、Windows Vista、Windows Server 2008 和 Windows 7。

本书不是一本轻松的入门书籍，无论读者是否拥有 WRK，都最好在阅读以前先有操作系统的基本知识，以及较为熟练的编程能力，起码要能够轻松地阅读 C 代码。硬件系统结构、汇编语言和编译原理等知识虽然不是必要的，但是有了这些知识，可以让您更加深入地理解各种系统机制。

本书内容组织

本书的内容并没有覆盖 Windows 内核的方方面面，而是从现代操作系统的概念模型出发，重点讨论了进程、线程和内存管理的基本概念以及在 Windows 中的实现细节，也讨论了 Windows 中的并发性和同步处理机制。作为一个对各种硬件设备有广泛兼容性的桌面操作系统，Windows 的 I/O 模型是提供这种兼容能力的基础，因此，Windows 中的 I/O 处理也是本书重点讨论的内容之一。在此基础上，本书还讲述了 Windows 的存储体系结

构，它把 Windows 系统中众多的内核组件，包括缓存管理器、文件系统、磁盘管理驱动程序等，纳入到了一个统一的框架中。

这些内容构成了本书的主体。然而，仅仅这些还不足以形成一个完整的操作系统，Windows 内核中还有其他一些不可或缺的机制，比如内核中的对象管理、配置管理（即注册表服务）、系统内核日志、安全性管理、系统服务分发、LPC（Local Procedure Call，本地过程调用），以及网络和 Windows 子系统等，本书对此也给出了足够详尽的描述，以使读者能够透彻地理解 Windows 操作系统的全貌。另外，Windows 内核的初始化过程也在本书的覆盖范围之中。

在描述以上这些内核机制时，本书以 WRK 为主要的参照系统。WRK 支持 AMD64 和 Intel x86 两种体系结构，本书选择以 Intel x86 作为目标平台，而完全忽略在其他体系结构上的差异。Windows XP (x64)和 Windows Server 2003 SP1 (x86)分别代表了客户端操作系统和服务器端操作系统，它们使用了同一套内核源代码，所以，虽然本书讨论的内容以 Windows Server 2003 SP1 为基准系统，但同样适用于客户端操作系统。

本书讲述的内容显然超越了 WRK 中源代码的范围，对于在 WRK 中未提供源代码的话题，比如即插即用管理器、Windows 子系统和网络等，本书参考了 Microsoft 提供的各种文档，以及通过其他途径提供的源代码，例如 Windows DDK 文档和例子驱动程序。此外，Mark Russinovich 和 David Solomon 合著的《Windows Internals》（第 4、5 版，中文版书名为《深入解析 Windows 操作系统》）也是重要的参考材料。本书在讲解各个话题时注明了所引用资料的出处，对于 Microsoft 提供的参考资料，包括 Windows DDK 文档，本书在参考资料引用条目中注明了这些内容在 MSDN 网站上的链接。

本书内容简介

本书共包括 9 章正文内容和两个附录。内容介绍如下：

第 1 章介绍了操作系统的基础概念、Windows 操作系统的发展历史、Windows 内核的发展，以及学习操作系统的一些前提知识，最后还介绍了操作系统的研究进展。

第 2 章从总体上介绍了 Windows 操作系统。首先讨论现代操作系统的基本模型，以及一些重要部件的职责；然后讨论 Windows 的总体结构，以及有关 WRK 和相应源代码的一些说明。为了让读者对 Windows 内核有一个初步的认识，这一章还介绍了内核中的一些基本概念，包括进程、线程、内存管理、中断、异常、同步等，然后细致地讨论了 Windows 内核中的公共管理设施，包括对象管理器、配置管理器、事件追踪（ETW）和

安全性管理。之所以在这一章中介绍这些公共设施，是为了让读者在学习本书后面章节中的核心概念和机制时，可以方便地直接引用这些管理设施中的功能。这一章的最后还介绍了 Windows 引导过程，包括内核初始化和用户会话建立过程。

第 3 章讨论了 Windows 中的进程和线程管理。首先讨论了进程和线程的基本概念，然后描述了 Windows 中有关进程和线程的重要数据结构，列出了这些数据结构的定义并解释了其中域的含义。接下来介绍了 Windows 进程中的句柄表结构、进程和线程的创建过程和结束处理，以及系统的初始进程和线程。这一章后面还介绍了 Windows 中的线程调度，包括时限管理、环境切换、线程状态转移等内容。

第 4 章介绍了 Windows 中的内存管理。首先介绍了一般性的内存管理知识，包括页式内存管理和段式内存管理。然后介绍了 Windows 的系统内存管理，涉及系统地址空间初始化、换页内存池和非换页内存池的管理算法，以及系统 PTE 区域的管理算法。紧接着介绍了进程地址空间的内存管理，包括地址空间的创建和初始化、虚拟内存的管理数据结构，以及内存区对象（section object）。然后讨论了 Windows 的页面交换机制和物理内存管理。最后讨论了 Windows 的工作集（working set）管理。

第 5 章讨论了 Windows 中的并发和同步机制。这一章首先介绍了有关进程和线程同步的基础知识，然后介绍了 Windows 中的中断和异常处理机制，包括 IRQL（中断请求级别）、中断对象、DPC（延迟过程调用）、定时器管理、APC（异步过程调用），以及异常分发过程。接下来分别介绍了两种类型的同步机制：高 IRQL 时的同步处理，以及基于线程调度的同步机制。Windows 支持大量的同步语义，而且很多同步语义被导出成系统服务、可供应用程序使用。

- 第 6 章讲述了 Windows 的 I/O 模型。首先讨论了 Windows I/O 系统的三个重要部件：I/O 管理器、即插即用管理器和电源管理器，介绍了设备驱动程序的初始化，驱动程序对象、设备对象和文件对象的数据结构，设备的列举过程，以及 Windows 中电源 I/O 请求的处理过程。在此基础上，详细讨论了设备驱动程序，包括驱动程序分类、驱动程序结构模型，并通过一个例子（toaster）说明了驱动程序设备栈结构。最后，介绍了 I/O 处理，包括 IRP（I/O 请求包）的定义、I/O 请求的处理过程以及 I/O 完成处理。

第 7 章讨论了 Windows 的存储管理。这一章分别讨论了 Windows 的缓存管理器、卷的管理以及文件系统。缓存管理器将文件系统和内存管理器有机地结合起来；而文件系统和卷管理驱动程序以及磁盘管理驱动程序联合起来形成了一个存储栈，为 Windows 提供外部存储服务。还讨论了文件系统的过滤机制，以及三个文件系统的实现：RAW（由内核模块提供）、FAT 和 NTFS。

第 8 章讨论了 Windows 的系统服务。这一章首先描述了 Windows 用户模式代码调用内核模式系统服务的详细过程，然后介绍了 Windows 的系统服务分发机制。随后，还介绍了三个重要的跨进程通信系统服务：LPC（Local Procedure Call）、命名管道（Named Pipe）和邮件槽（Mailslot）。

第 9 章介绍了一些高级话题，重点讨论了 Windows 中的网络结构、Windows 子系统，以及 Windows 的内核日志。最后介绍了 Windows 最新版本（Windows Vista、Windows Server 2008 和 Windows 7）中的一些重要变化。

本书正文之后是两个附录。附录 A 讲述了如何编译 WRK，以及在虚拟机环境中运行和调试 WRK。附录 B 介绍了一个基于 WRK 的学生实习项目：KInjectToolKit，KInjectToolKit 允许把一段用户模式代码插入到内核模式下运行，从而提供了在响应内核模式事件时执行用户指定代码的能力。

本书内容与 Windows 系统的版本

本书中介绍的 Windows 核心机制的实现细节以 Windows Server 2003 SP1 为准。由于 Windows 体系结构在不同版本之间基本上没有变化，内核机制的变化也非常微小，所以，本书绝大多数内容适用于自 Windows 2000 以来的各个 Windows 版本，包括 Windows Vista、Windows Server 2008 和 Windows 7。此外，在本书讲解过程中，有些地方特别指出了 Windows Vista 及以后系统的内核变化。

本书的配套工具

为了配合本书中的内容讲解，本书也特别提供了一组工具，读者可以通过本书的主页下载到这些工具，它们不仅可以运行在 WRK 内核的系统中，也可以在其他版本的 Windows 系统上运行。下面是这组工具的简单介绍：

- ProcMon，进程和线程监视工具。它可以实时地显示当前每个处理器正在执行哪些进程和线程。这些信息还可以被转储到一个文本文件中供进一步分析和查看。
- MemMon，内存监视工具。它可以显示当前系统内存空间和每一个进程的用户空间的内存布局。在进程内存空间，MemMon 能够显示进程中的模块、堆、栈等信息。
- DPerfLite，一个可以揭示线程间同步关系的工具。它记录了当前系统中与同步有关的内核操作，允许用户在一个图形界面中直观地观察这些操作，并检查线程之间的各种同步关系。

- IRPMon，IRP 完成情况的监视工具。它在内核中记录下每一个 I/O 请求的完成操作，然后由 IRPMon 用户模式程序定期地从内核中获取这些记录，并显示在一个列表窗口中，供用户查看。
- SdtViewer，SDT（服务描述符表）显示工具。此工具可以显示出当前系统中的 SDT；如果有符号信息支持，它也能够显示 Windows 子系统的 SDT。

本书的支持信息

本书的主页是发布各种与本书内容有关的信息的场所，包括前面提到的一组工具，以及本书的勘误、修订信息。主页地址为：

http://www.phei.com.cn/windowskernelbook

您也可以将有关本书的建议和疑问，包括在阅读本书过程中遇到的问题、发现的错误，通过以下电子邮件地址发送给作者：

winkernelbook@live.com

目 录

第 1 章 概述 1

1.1 操作系统基础 2

1.1.1 计算机系统的硬件资源管理 2

1.1.2 为应用程序提供执行环境 5

1.2 学习操作系统之必备知识 7

1.3 Windows 操作系统发展历史 9

1.4 Windows 内核的版本 11

1.5 操作系统的研究与发展 13

1.6 本章总结 16

第 2 章 Windows 系统总述 17

2.1 现代操作系统的基本结构 17

2.2 Windows 系统结构 18

2.2.1 Windows 内核结构 20

2.2.2 Windows 内核中的关键组件 22

2.2.3 Windows 子系统 32

2.2.4 系统线程和系统进程 35

2.3 关于 Windows 研究内核 37

2.3.1 WRK 包含了什么 38

2.3.2 WRK 源代码说明 39

2.3.3 本书对 WRK 源代码的引用 41

2.4 Windows 内核的基本概念 42

2.4.1 处理器模式 43

2.4.2 内存管理 44

2.4.3 进程和线程管理.....46
2.4.4 中断和异常.....48
2.4.5 同步.....51
2.5 Windows 内核中的公共管理设施.....53
2.5.1 Windows 内核中的对象管理.....53
2.5.2 注册表和配置管理器.....61
2.5.3 事件追踪（ETW）.....72
2.5.4 安全性管理.....75
2.6 Windows 引导过程.....81
2.6.1 内核加载.....82
2.6.2 内核初始化.....85
2.6.3 建立用户登录会话.....90
2.7 本章总结.....96
第 3 章 Windows 进程和线程.....97
3.1 进程基本概念.....97
3.1.1 多进程模型.....98
3.1.2 进程与程序.....99
3.2 线程基本概念.....102
3.2.1 线程模型.....102
3.2.2 线程调度算法.....104
3.2.3 线程与进程的关系.....106
3.3 Windows 中进程和线程的数据结构.....106
3.3.1 内核层的进程和线程对象.....106
3.3.2 执行体层的进程和线程对象.....118
3.4 Windows 的进程和线程管理.....129
3.4.1 Windows 进程的句柄表.....129
3.4.2 获得当前线程或进程.....135
3.4.3 进程和线程的创建过程.....136
3.4.4 进程和线程的结束处理.....146
3.4.5 系统初始进程和线程.....148
3.5 Windows 中的线程调度.....150
3.5.1 线程优先级.....150
3.5.2 线程状态转移.....153

3.5.3 时限管理 163
3.5.4 优先级调度和环境切换 165
3.6 进程和线程运行状态监视工具 171
3.6.1 ProcMon 使用示例 171
3.6.2 ProcMon 实现原理 173
3.7 本章总结 174

第 4 章 Windows 内存管理 175

4.1 内存管理概述 176
4.1.1 页式内存管理 177
4.1.2 段式内存管理 181
4.1.3 内存管理算法介绍 184
4.1.4 Windows 内存管理概述 192
4.2 Windows 系统内存管理 194
4.2.1 系统地址空间初始化 194
4.2.2 系统地址空间内存管理 209
4.2.3 系统 PTE 区域的管理 223
4.3 进程内存管理 229
4.3.1 地址空间的创建和初始化 229
4.3.2 地址空间切换 234
4.3.3 进程地址空间的内存管理 235
4.3.4 内存区对象 241
4.4 内存页面交换 250
4.4.1 Intel x86 中的 PTE 251
4.4.2 软件 PTE：无效 PTE 和原型 PTE 253
4.4.3 页面错误处理 257
4.4.4 Windows 的写时复制 263
4.5 物理内存管理 265
4.5.1 PFN 数据库 266
4.5.2 物理页面的状态变化 272
4.5.3 物理页面链表的管理和操作 275
4.5.4 修改页面写出器 280
4.5.5 进程/栈交换器 282
4.5.6 低内存通知和高内存通知 285

4.6 工作集管理 286
4.6.1 Windows 工作集管理器 286
4.6.2 平衡集管理器 292
4.7 内存监视工具 MemMon 293
4.7.1 MemMon 使用介绍 293
4.7.2 MemMon 实现原理 295
4.8 本章总结 295

第 5 章 Windows 并发和同步 297

5.1 进程和线程的同步基础 297
5.1.1 并发性基础 298
5.1.2 进程或线程之间的通信 301
5.1.3 经典的同步问题 305
5.2 Windows 中断与异常 310
5.2.1 硬件中断的发生和处理 311
5.2.2 中断请求级别（IRQL） 317
5.2.3 中断对象 320
5.2.4 DPC（延迟过程调用） 323
5.2.5 时钟中断和定时器管理 327
5.2.6 APC（异步过程调用） 330
5.2.7 异常分发 336
5.3 不依赖于线程调度的同步机制 343
5.3.1 提升 IRQL 实现数据同步 343
5.3.2 互锁操作 345
5.3.3 无锁的单链表实现 346
5.3.4 自旋锁 349
5.4 基于线程调度的同步机制 354
5.4.1 线程进入等待 354
5.4.2 分发器对象 361
5.4.3 门等待 369
5.4.4 执行体资源（executive resource） 370
5.4.5 推锁（push lock） 373
5.4.6 死锁 378
5.5 观察线程同步关系——DPerfLite 379

5.5.1 DPerfLite 使用示例 379
5.5.2 DPerfLite 实现原理 381
5.6 本章总结 382

第 6 章 Windows I/O 系统 383

6.1 I/O 概述 384
6.1.1 现代计算机系统的 I/O 384
6.1.2 I/O 软件技术 388
6.1.3 Windows I/O 系统结构 390
6.2 I/O 管理器 392
6.2.1 驱动程序初始化 393
6.2.2 驱动程序对象和设备对象 399
6.2.3 文件对象 404
6.2.4 对象生命周期管理 407
6.3 即插即用管理器 408
6.3.1 即插即用的基本要求 409
6.3.2 Windows 中驱动程序的即插即用支持 410
6.3.3 设备列举与设备树 411
6.4 电源管理器 414
6.4.1 电源管理概述 414
6.4.2 Windows 中的电源管理 417
6.5 设备驱动程序 422
6.5.1 设备驱动程序分类 423
6.5.2 例子驱动程序 toaster 425
6.5.3 驱动程序的代码结构 427
6.5.4 toaster 设备的设备栈 432
6.5.5 过滤驱动程序的配置和加载 434
6.5.6 非即插即用驱动程序 437
6.6 I/O 处理 440
6.6.1 I/O 请求包（IRP） 440
6.6.2 针对独立设备对象的 I/O 处理 447
6.6.3 处理 I/O 请求过程中的事项 451
6.6.4 针对设备栈的 I/O 处理 461
6.6.5 I/O 完成端口 465

6.7 I/O 请求监视工具 IRPMon 468
6.7.1 IRPMon 使用介绍 468
6.7.2 IRPMon 实现原理 469
6.8 本章总结 470

第 7 章 Windows 存储管理 471

7.1 存储管理概述 471
7.1.1 硬件存储体系（memory hierarchy） 472
7.1.2 Windows 的存储管理结构 474
7.2 Windows 缓存管理 476
7.2.1 Windows 缓存空间的内存管理 476
7.2.2 缓存管理器的数据访问路径 483
7.2.3 直接使用系统缓存中的数据 486
7.2.4 缓存管理器的预读处理 493
7.2.5 缓存管理器的延迟写 496
7.3 Windows 中卷的管理 499
7.3.1 Windows 存储栈结构 500
7.3.2 卷的挂载 504
7.3.3 卷与文件系统 507
7.3.4 文件对象的 I/O 处理 510
7.4 Windows 文件系统 513
7.4.1 文件系统驱动程序结构 514
7.4.2 RAW 文件系统与 FsRtl 519
7.4.3 文件系统的 I/O 过滤 522
7.4.4 FAT 文件系统 530
7.4.5 NTFS 文件系统 534
7.5 本章总结 542

第 8 章 Windows 系统服务 545

8.1 Windows 系统服务原理 545
8.1.1 Intel x86 的用户模式-内核模式切换 545
8.1.2 Windows 的用户模式-内核模式切换 550
8.1.3 Windows 中的系统服务分发 555
8.1.4 增加系统服务表或表项 562

8.2 LPC（本地过程调用）服务 565
8.2.1 LPC 结构模型 565
8.2.2 LPC 端口和 LPC 消息 567
8.2.3 LPC 通信模型的实现 569
8.2.4 LPC 应用 575
8.3 命名管道（Named Pipe）服务 577
8.3.1 命名管道的名称解析 577
8.3.2 命名管道的通信模型 579
8.3.3 命名管道的实现 581
8.4 邮件槽（Mailslot）服务 584
8.4.1 邮件槽的名称解析 584
8.4.2 邮件槽的通信模型 585
8.4.3 邮件槽的实现 586
8.5 SDT 显示工具 SDTViewer 588
8.5.1 SDTViewer 使用介绍 588
8.5.2 SDTViewer 实现原理 589
8.6 本章总结 590

第 9 章 Windows 系统高级话题 591

9.1 网络 591
9.1.1 Windows 网络体系结构 591
9.1.2 TDI（传输驱动程序接口） 595
9.1.3 NDIS（网络驱动程序接口规范） 599
9.1.4 Windows Vista 及以后版本的网络结构 601
9.2 Windows 子系统 603
9.2.1 Windows 子系统结构 603
9.2.2 Windows 子系统初始化与 GUI 线程 607
9.2.3 窗口管理 610
9.2.4 GDI（图形设备接口） 620
9.2.5 Windows Vista 及以后的子系统变化 627
9.3 内核日志 629
9.3.1 内核日志记录器 629
9.3.2 利用内核日志信息诊断性能问题 632
9.4 Windows Vista/Server 2008/7 的重要变化 640

9.4.1 MinWin 工程 640
9.4.2 进程和线程管理 643
9.4.3 内存管理 645
9.4.4 I/O 处理的改进 647
9.5 本章总结 650

附录 A 建立 WRK 工作环境 651

A.1 编译 WRK 651
A.2 启动 WRK 655
A.3 调试 WRK 658

附录 B 内核代码插入工具 KInjectToolKit 665

B.1 KInjectToolKit 功能介绍 666
B.2 KInjectToolKit 的代码实现 667
B.3 KInjectToolKit 的限制 671

参考资料 673

术语对照表 681

索引 687

1

概述

现代的人们大多是这样认识计算机的：一个人盯着屏幕敲打着键盘，这就是在操纵计算机了。于是，屏幕和键盘就成了计算机的代表物了。更进一步了解计算机的人们知道，屏幕和键盘只不过是计算机的输出和输入而已，计算机的核心在一个机箱里，包括处理器、内存、主板、硬盘和电源等。所有这些构成了计算机的有形部分（实际上，这些部件也在越变越小）。再进一步，真正使用过计算机的人们还知道，光是计算机硬件还不够，还需要各种软件，日常用到的软件包括：办公软件、游戏软件、字典软件、上网软件、聊天软件以及防病毒软件，等等。正是有了这些软件，计算机用起来才这么有趣。所以，虽然软件并不有形（不过，存放软件的介质却是看得见摸得着的），但人们知道它们是切实存在的。那么，操作系统（Operating System）是什么呢?

就本质而言，操作系统也是一种软件，只不过，相对于一般的软件而言，有其特殊性。首先，操作系统往往是随机器配置的，人们常常认为这是机器本身的一部分。只有计算机专业人员或技术维护人员才会安装或更换操作系统。其次，所有的应用软件都建立在操作系统的基础之上，操作系统的发展本身也推动了应用软件的发展。一个众所周知的事实是，Windows 操作系统的发展和普及，为大量的应用软件提供了得以存在的平台，各种家用软件也纷纷随之诞生。

操作系统是专门管理硬件资源的软件，计算机硬件本质上只提供计算和存储的能力，而操作系统则利用硬件的计算和存储能力，建立起一个抽象层。在现代操作系统中，这一抽象层包括任务或进程（或线程）、文件、设备或字节流等诸多概念以及相应的功能。基于操作系统所提供的抽象概念和功能，应用程序能够方便地完成其功能，并且无须直接操纵计算机的硬件。而且，现代操作系统也提供了多个任务共享硬件资源的能力，所以，应

用程序并不独占硬件资源，而是以某些既定的方式来共享，这也为用户使用各种应用软件提供了极大的便利。

对于应用软件开发人员，不同种类操作系统的特殊性甚至决定了他们的职业范围。比如，UNIX 平台的开发人员转到 Windows 平台上，需要经过相当一段时间的再学习过程；反之亦然。直接在操作系统上进行软件开发的程序员，必须非常清楚地了解操作系统所提供的抽象层，才能够编写出行之有效的软件来。在现代软件开发领域，其中一个分支是在系统无关的平台上进行软件开发。这里所谓的系统无关的平台是指，一个公共抽象层供上层软件在其上运行，因而上层软件与底下的操作系统平台无关。典型的例子是 Java 语言及其开发平台。这实际上是软件多层次化的体现。在本书中，我们仅关心直接建立在硬件之上的那一层软件，即传统意义上的操作系统。

本章将首先扼要地介绍操作系统的基础知识，然后介绍学习操作系统的一些必备知识。接下来讲述 Windows 作为一个操作系统的发展历史，以及 Windows 的内核版本。最后介绍操作系统领域的研究和进展。

1.1 操作系统基础

操作系统是一层特殊的软件，它的职责是管理硬件资源，同时为上层应用软件提供各种服务。所以，操作系统直接跟硬件打交道，自然地，它使用机器能理解的语言，即机器语言。因此，操作系统的发行版本一定是与特定的硬件体系结构相关的，比如针对 Intel x86 的操作系统。虽然操作系统的运行版本使用的一定是机器语言，但操作系统本身不必是用机器语言或汇编语言来编写的，目前最常见的是用 C 语言来编写的（主要原因是，C 语言与机器语言有较好的对应性，所以程序员容易控制目标代码）。在本书中，我们主要参考的也是 C 代码，只在个别地方由于硬件相关性，会用到汇编语言。

接下来先讨论操作系统的硬件资源管理职责，再介绍它为应用程序提供的执行环境。

1.1.1 计算机系统的硬件资源管理

操作系统管理哪些硬件资源，分别又是怎么管理的呢？最主要的资源是计算资源和存储资源。计算资源即 CPU（Central Processing Unit，中央处理单元）[1]，现在主流的计算机

[1] GPU（Graphics Processing Unit）也是一种计算资源。目前越来越多的计算机配备了 GPU，但当前的操作系统尚未将 GPU 当做基本的计算资源来考虑。

通常有一个或多个 CPU，或者一个 CPU 中有多个核（即多核 CPU）。从操作系统的角度来看，有多个 CPU 或一个多核 CPU 意味着可以同时执行多个任务。所以，操作系统必须合理地安排和调度任务，使得多个 CPU 或多核尽可能地利用起来，避免出现竞争或闲置的情形。在支持多任务并发的操作系统中，这一职责称为任务调度。在现代操作系统中，由于任务是由进程或线程来完成的，操作系统的这部分功能也称为进程调度或线程调度。因为任务的数量可能超过 CPU 或核的数量，所以，多个任务可能共用同一个 CPU 或核，这就需要有一种硬件机制能够让操作系统在不同的任务之间实现切换，这是任务调度的硬件基础。通常，计算机提供的时钟中断可以让操作系统很方便地做到这一点，也就是说，每隔一定的时间，硬件系统会触发一个中断；操作系统截获此中断，按照某种算法暂停当前正在执行的任务，并选择一个新的任务，从而实现任务的切换；到下一个时钟中断到来时，再继续这样的切换过程。因此，多个任务可以在一个 CPU 或核中被轮流执行。操作系统可以设定时钟中断间隔的长度，也可以选择不同的算法来安排这些任务被先后执行，这样就形成了各种不同的调度算法。本书将在第 3 章讨论 Windows 的线程调度方案。

存储资源通常包括内存（RAM，随机访问存储器）和外存（也称为辅助存储器）。由于外存是通过标准的 I/O（输入/输出）来管理的，而内存是 CPU 直接通过系统总线来访问的，所以，我们将在讨论输入/输出的时候再介绍外存，现在仅考虑内存资源的管理。内存是 CPU 执行一个任务的物质基础，CPU 内部的寄存器具备计算的能力，但计算的数据从哪里来呢？除了寄存器（其本身也是一种存储资源）以外，数据的来源是系统内存。在现代操作系统中，每个任务都有其独立的内存空间，从而避免任务之间产生不必要的干扰，所以操作系统有责任为每个任务提供相对独立的内存空间。把连续编址的物理内存划分成独立的内存空间，典型的做法是段式内存寻址和页式虚拟内存管理。不同的硬件体系结构可能支持不同的方案。Intel x86 体系结构同时支持段式寻址和页式虚拟内存映射，但是，可在 Intel x86 上运行的操作系统几乎都选择了虚拟内存映射作为内存管理的硬件基础。Windows 和 Linux 便是典型的例子。

在支持多任务的系统中，若所有任务的内存需求加起来的总量超过了当前系统的物理内存总量，那么，系统要么停掉一些任务，要么把一些任务转移到外存（如磁盘）中，以后当内存空闲时再把这些任务转换回来。或者系统有选择地把部分不常用的内存转换到外存，并且根据适当的规则将来再慢慢地转换回来。虚拟内存的映射以及物理内存不足时的换出和换入操作，这都是操作系统管理内存资源的重要任务。前者依赖于硬件提供的机制，而后者则更多地由操作系统自己来控制。

除了计算资源和内存资源的管理以外，操作系统对其他资源都通过 I/O 来管理。例如，

上面提到的外存资源，像磁盘，在现代计算机中是不可或缺的部件；另外，键盘和鼠标通常是标准的输入设备，而显示器和打印机往往是标准的输出设备。操作系统为了跟 I/O 设备打交道，需要三方面的技术保障：CPU 通过特定的指令来控制 I/O 设备，I/O 设备通知 CPU 发生了特定的事情，以及在系统主内存和设备之间传输数据。

通常，CPU 直接访问设备的寄存器来操作一个设备，在 Intel x86 系统上，CPU 通过 in 和 out 指令能够做到这一点。设备寄存器是另一个地址空间，CPU 通过 I/O 端口（I/O port）来访问它们。在现代计算机中，I/O 端口的分配跟软件和硬件都有关系。不同的硬件设备会使用不同的端口编号，现代的设备大都可以通过软件方式来设置其端口号，而过去一些老的设备可能需要通过硬件跳线来改变端口号的设置。对硬件设备进行恰当的设置，也是操作系统管理硬件设备的任务之一。另外，CPU 怎么知道或检测设备的工作状态呢？一种做法是，通过不停地查询设备的状态寄存器来获知其工作状态；但是，更有效的做法是，当设备的状态发生变化时，它能够主动地通知 CPU，从而操作系统可以采取相应的措施。这后者即是设备中断机制。比如说，当键盘设备接收到按键动作时，它产生一个中断，告诉 CPU（和操作系统），当前哪个键被按下了。中断也有编号，中断的编号被视为系统全局资源，在早期计算机的中断控制器中，不同设备的中断号不能冲突，否则设备无法正常工作。现代计算机通常允许多个设备共享中断号，操作系统和设备驱动程序可以协商设备的中断号。

计算机的计算处理能力往往仅限于在 CPU 内部寄存器和主内存之间进行，但是为了实现基本算术计算以外的其他各种能力，通常有必要让设备中的数据也参与到计算中来，所以，在设备与 CPU 寄存器或主内存之间传输数据往往是必要的功能。例如，计算机通过磁盘设备可以实现永久存储，通过显示控制器实现彩色显示甚至三维模拟。实现数据传输的方法有多种，如果设备本身的数据量很小，则可以直接通过 in 指令来读取设备中的数据，或通过 out 指令输出到设备中。或者也可以映射一段地址范围到设备中，这样，当 CPU 访问这块地址范围时，实际上是在访问设备的内存，而不是系统的主内存。另外一种适合于大块数据传输的技术是 DMA（Direct Memory Access，直接内存访问）。像硬盘控制器和网络控制器就通常采用 DMA 方式来传输数据，CPU 只须设置好数据传输的方向、位置、数量等信息，就可以启动 DMA 传输了。DMA 传输可以与 CPU 计算同时进行，但是 DMA 使用的总线不能与 CPU 使用的发生冲突，它可以趁 CPU 不用总线的时刻来传送数据，也可能会因此而阻塞 CPU 指令的执行。DMA 传输影响 CPU 执行指令的程度取决于 DMA 控制器的传输策略。

从操作系统的角度来看，考虑到 I/O 设备的多样性和出现新设备的可能性，操作系统有必要定义一个框架来容纳各种各样的 I/O 设备，并且允许操作系统发布之后还能够为新

的设备提供支持。所以，除了专用操作系统以外，现代操作系统都会提供一个 I/O 模型，允许设备厂商按照此模型编写设备驱动程序（Device Driver），并加载到操作系统中。I/O 模型通常具有广泛的适用性，能够支持各种类型的设备，包括对硬件设备的控制能力，以及对数据传输的支持。可以这么来概括 I/O 模型，它对下提供了控制硬件设备的能力，对上为应用程序访问硬件提供了一个标准接口，同时 I/O 模型也必须能够让操作系统有效地管理设备驱动程序。在 Windows 系统中，第三方厂商可以使用 Windows 的 I/O 模型来编写设备驱动程序。Windows 本身在发行时，已经随带了大量主流设备的驱动程序，所以，Windows 系统在安装阶段可以自动将识别出来的设备的驱动程序安装到系统中。因此，用户并不需要手工下载或安装这些驱动程序。另一方面，由于驱动程序需要直接访问硬件设备，它执行的许多指令（包括 in 和 out）是特权指令，所以，驱动程序对于系统的稳定性和安全性有至关重要的影响，操作系统有必要对其执行严格的筛查措施，以避免恶意代码趁机闯入系统中。

1.1.2 为应用程序提供执行环境

操作系统之所以要管理各种硬件资源，是为了更好地为上层应用程序提供服务。应用程序并不直接与机器的各种硬件设备及资源打交道，而是运行在一个抽象层上，即操作系统提供的功能语义层。操作系统提供什么样的语义层，将决定应用程序应该怎么来构建它们的功能逻辑。尽管不同的操作系统提供的这一功能语义层的接口可能不一致，甚至相距甚远，但现代操作系统的语义层在概念上仍然比较一致，下面我们从应用程序的构建需求来介绍一些核心概念。

首先，对应用程序的任务作最基本的抽象，这就是进程和线程的概念。尽管不同的操作系统对于进程和线程的定义不尽相同，但是每个任务都应该有它自己的执行环境，即该任务的控制流以及函数调用的层次递进痕迹（假设 CPU 的指令体系中支持函数调用的语义）。比如，在 UNIX 中，一个进程代表了一个应用任务，它记录了该任务的执行状态；在 Windows 中，一个线程代表了一个任务，其中也记录了该任务的执行状态。现代操作系统都支持多个任务并发执行，任务的数量可以远远大于 CPU 或核的数量，所以，每个任务实际上只是分到了一部分 CPU 执行时间，多个任务可以共享一个 CPU。因此，在这种多任务操作系统模型下，操作系统除了为每个任务（进程或线程）维护好其状态以外，还必须有恰当的算法来决定什么时候让哪个任务执行，什么时候暂停一个任务。正如本章前面所提到的，这便是任务调度，或者称为进程调度或线程调度（取决于操作系统是以进程还是线程的方式来管理任务）。简单地轮流执行每个任务，这种调度算法往往难以满足

实际应用程序的各种需求。通常操作系统会采用基于优先级的抢占式调度算法，甚至根据任务的各种特性进行优先级的动态微调，这使得任务调度算法趋向于复杂化。Windows的线程调度方案属于这一类型。

其次，在支持多任务的基础上，如果这些任务是相互独立的，则操作系统总是可以采用适当的方式让它们获得执行机会，但在实践中，应用程序为了实现各种功能逻辑，不同任务之间往往有一些逻辑上的关联性。比如，任务之间必须强加某种时序关系，才能保证每个任务的状态是有效的；当多个任务竞争某些稀有资源（注意，系统的硬件资源，像打印机和显示器等，是共享的）时，系统必须确保这些任务有序执行。所以，凡是存在共享资源的地方，操作系统都必须提供恰当的方法来同步应用程序对它的访问。现代操作系统通常会提供多种同步机制，例如互斥体（mutex）、信号量（semaphore）、临界区（critical section）等。应用程序可以有选择地使用这些同步机制，以确保多个任务有序地共享资源。

除了任务和同步的概念，每个应用程序还必须有它自己相对独立的执行空间。在现代操作系统中，进程也代表了一个应用程序和它的执行空间。不同进程的空间是相互隔离的，这是现代操作系统的基本需求。操作系统必须在处理器的硬件特性基础之上，实现一套行之有效的空间隔离方案。每个进程有它自己的内存空间，并且无法直接访问其他进程的内存空间。进程之间如果要共享数据，则必须通过操作系统提供的机制来进行。Intel x86体系结构上的操作系统基本上都利用硬件的虚拟内存映射机制来隔离每个进程的内存空间。所以，操作系统的职责是为每个进程维护好从虚拟地址到物理地址的映射关系，并且管理好物理内存的分配和回收。另一方面，除了进程的空间隔离性，操作系统还必须提供相应的机制让不同的进程可以相互通信，毕竟，很多软件需要进程之间的协作来完成一些上层功能。同步机制和跨进程地共享内存是典型的进程间通信（IPC，Inter-Process Communication）手段。

前面提到，现代操作系统往往以统一的框架来管理I/O设备，这也隐含了操作系统向应用程序暴露的I/O接口是统一的这样一层意思。应用程序通过此接口来访问系统的外部设备，而操作系统不仅要管理好应用程序访问外设的各种请求，包括它们的时序，还必须将应用程序的请求发送到对应的设备驱动程序中，最终由设备驱动程序来处理这些请求，而处理的结果也必须以某种方式回送给应用程序。操作系统通常以句柄（handle）来代表一个可访问的抽象设备，抽象设备可能与物理设备连接，也可能并不存在对应的物理设备或资源。操作系统还提供在一个句柄上读（read）、写（write）数据，以及发送控制命令的能力。所以，应用程序与系统设备打交道的方式非常简洁明了：打开设备获得句柄、向设备发送命令、读或写设备，以及关闭设备。操作系统的任务是管理这些设备和驱动程序，以及传递或解释应用程序的命令。

一类特殊的设备是磁盘，磁盘具有随机读写的能力，尽管其随机读写的性能比顺序读写要差很多。通常，磁盘的驱动程序负责操纵磁盘控制器，以便从磁盘中读取数据或者向磁盘上写入数据，但磁盘上的数据如何有效地组织起来以便不同的应用程序可以共享同一个磁盘，这一任务是由文件系统（file system）来完成的。在绝大多数操作系统中，文件系统是不可缺少的，它通过一个树状名字空间来管理磁盘上的存储空间，允许应用程序通过名称来访问磁盘上的数据。这种结构奠定了现代计算机的信息存储系统的基础。

所以，应用程序实际上工作在操作系统提供的抽象层上，它无须管理系统的硬件设备和资源，但必须使用操作系统提供的机制或语义来实现其自身的功能逻辑。这也暗示着，操作系统在某种程度上决定了上层应用程序应该如何使用它所提供的服务才是最有效和方便的，同时，应用程序也有足够的灵活性来完成其自身的上层功能。这就形成了在同一个操作系统上存在大量风格迥异的应用程序的局面，并且，由于不同操作系统提供的抽象层在概念上有一定的互通性，所以，很多应用软件能够便捷地从一个操作系统平台移植到另一个上。

我们通常所指的操作系统的范畴包括上面提到的资源管理功能，以及为应用程序提供的各种抽象概念和抽象接口的具体实现。但是，在应用程序和操作系统核心模块之间，往往还存在很多预封装好的模块，以便于应用开发人员可以复用这些模块，从而高效地开发应用软件，缩短应用软件的开发周期。这些模块既可能是操作系统厂商提供的，也可能是独立软件厂商提供的。它们既可能构成一个通用的中间件（middleware），也可能只是随编译器而提供的一个库模块（静态的或动态的）。例如，Windows 操作系统的发行版本包含大量这样的中间模块（例如 COM/COM+、Winsock2、.NET 等），它们也成为了 Windows 的一部分，但是，在本书中，我们将把焦点集中在 Windows 操作系统的核心功能和模块上，只在必要的地方提及这些中间模块。

1.2 学习操作系统之必备知识

前面提到，操作系统是直接建立在硬件层之上的软件。由于它需要直接跟硬件打交道，而且本身又为其他的软件提供了基本的执行环境，所以，仅仅掌握基本的软件开发技能还不足以理解和掌控一个完整的操作系统。为了便于读者更加有效地学习操作系统的基本知识，以及理解一个像 Windows 这样的复杂操作系统，本节将介绍一些笔者认为必不可少的基础知识，包括体系结构、汇编语言、编译原理和 C/C++语言。如果读者缺乏这方面的背景，建议在阅读本书以前先了解或学习相关的内容。

首先，有关现代计算机处理器的体系结构知识，会帮助读者更好地理解操作系统中的各种机制为什么要这样设计而不是那样设计。例如，操作系统中对于多线程的支持需要考虑处理器体系结构的指令集和寄存器集，而内存管理部分更需要吻合处理器的内存寻址方案，像段式内存寻址或者页式内存管理，等等。另外，硬件上的优化对于系统软件并不一定是透明的，这意味着有些优化需要软件来支持，例如，有些快表机制（TLB）允许操作系统来填充快表中的表项。因此，为了充分发挥硬件的性能，系统软件也有必要做一些针对性的处理。

作为学习操作系统的先学知识，读者并不需要精通体系结构的方方面面。然而，基本的指令流执行过程和指令执行过程中的时序分析，以及内存寻址的实现方案，对于理解操作系统的底层实现是非常有帮助的。

现代操作系统几乎都是用 C 语言来编写的，使用 C 语言有两方面的原因：一是为了可移植性，因为几乎每一种硬件平台上都有 C 语言的开发支持；二是由于 C 语言和底层汇编指令之间有较好的对应性，所以，C 代码的性能较容易控制，同时也易于理解。然而，对于在特定硬件体系结构上实现的一个具体操作系统，除了 C 代码以外，或多或少还会有一些汇编语言代码，也就是直接针对处理器来编写的代码。这些汇编代码，有的是为了优化而绕开了 C 编译器，有的则是为了处理一些 C 编译器和 C 运行库没有提供的、但又不可避免的任务，例如操作系统在处理内核模式和用户模式切换时需要用到特殊的指令，以及维护有关的状态寄存器。

汇编代码通常用于实现一些比较精巧的任务，很少用大量的人工汇编代码来实现复杂的功能，毕竟，编写汇编代码对于程序员的要求甚高，而且，生产效率相对低下，所以，汇编代码在操作系统中通常只用于实现某些特定的任务。但这些代码对于理解整个系统的运行又非常关键，它们常常出现在线程切换、中断处理例程等关键之处。因此，与硬件体系结构相关联的汇编语言知识对于学习和理解操作系统很有帮助。对于本书而言，基本的 Intel x86 汇编语言知识将有助于理解 Windows 操作系统中的一些关键点。本书在适当的地方会引用到 Intel x86 汇编指令。

除了硬件体系结构和汇编语言知识以外，有关编译的知识更是必不可少。操作系统需要负责从系统加电一直到等待接收用户输入这整个过程，它必须逐渐建立起系统执行环境；操作系统必须为应用软件提供完全的执行环境，它需要定义应用软件的二进制规范，需要解决从二进制映像文件到内存代码和数据之间的映射关系，甚至还需要针对二进制代码的一些特点进行适当的优化处理。然而，几乎所有的软件，包括系统模块和应用程序，都是用高级语言编写的，它们经过了编译器的代码生成过程，所以，掌握编译原理有助于理解操作系统是如何加载和运行各种二进制模块的。

最后，C 语言是系统程序员之间进行交流的基本语言。Windows 内核的绝大部分代码是以 C 语言来编写的，大多数其他的内核也如此。本书中列举的代码大多是 C 代码，而书中用到的很多工具则是用 C++来编写的。你可以把 C 看做 C++语言的一个简化，或者把 C++看做 C 语言的面向对象扩展，但这种观点其实并不完全正确。C 和 C++是两种连设计思想都完全不同的语言，本书并不介绍这两种语言，但是很多地方会直接列示 C 代码来讲解一些细节知识，所以，基本的 C 语言知识是必不可少的，如果有 C++的知识，则更有助于理解一些上层的 C++代码。

简而言之，在学习本书以前，读者需要具备基本的硬件体系结构和汇编语言知识，而且对于二进制代码的规范和生成过程有基本的了解。当然，熟练的 C 语言技能更是必不可少的。

1.3 Windows 操作系统发展历史

Microsoft 在操作系统领域中的发展最早开始于 MS-DOS，并于 20 世纪 80 年代后期开始按两个分支发展：一是基于 MS-DOS 的 Windows 开发平台，并发展成 Windows 95/98/Me 这一系列操作系统；另一个分支则是以 Windows NT 为代表的操作系统系列，经历了 Windows NT 3.1/3.5/3.51/4.0、Windows 2000、Windows XP/Server 2003，一直到 Windows Vista/Server 2008 和最近刚刚发布的 Windows 7。后者是按照 20 世纪 80 年代后期的硬件基础全新设计的一款操作系统，其中包含了现代操作系统的各个要素。这也奠定了 Windows 作为现代操作系统典范的基础。下面介绍这一系列操作系统的发展历程。

Windows NT 内核是在 1988 年秋季开始设计的，当时的设计目标是提供一个真正 32 位、抢占式调度和采用虚拟内存的网络操作系统。设计的初衷是可以在多种硬件体系结构和平台上运行，但随着计算机硬件市场的逐步发展，Windows NT 最终仅支持 Intel 兼容的体系结构。另外一个值得一提的设计目标是，兼容已有的操作系统，包括 OS/2、POSIX（针对 UNIX 类操作系统的一个接口标准）和 Windows（基于 MS-DOS 版本的窗口系统）。但其间由于 Windows 3.0 在商业上取得巨大成功，于是 Windows NT 的设计也作了相应调整，以支持 Windows 应用程序的兼容性为主，而把 OS/2 兼容性移到了一个子系统中。

Windows NT 内核在设计之初有清晰的目标，并且 Microsoft 集中了当时最有内核设计经验的专家（由 David Cutler 领头，他之前设计了 VAX/VMS、RSX-11M 和 VAXELN 操作系统）来担当此任，所以，Windows NT 内核有一个非常整齐、先进的体系结构，并且综合考虑了安全性、扩展性等要素。这一结构在接下来的 20 年发展历程中基本保

持不变，即使近几年硬件有了飞速的发展，Windows XP/Vista/7、Windows Server 2003/2008 仍然沿袭了 NT 内核基本结构。

基于 Windows NT 内核的第一个操作系统于 1993 年发布，即 Windows NT 3.1，这一版本支持 Intel i386、Intel i486、MIPS R4000 以及 Digital Alpha 处理器。而后，1994 年秋天发布了 Windows NT 3.5；1995 年 5 月发布了 Windows NT 3.51，在这个版本中加入了对于 IBM PowerPC 处理器的支持。这些版本均实现了 Windows NT 的基本设计目标。然而，从 NT 3.1 到 NT 3.51 版本上的变化，主要是让系统运行得更加可靠、速度更快，并且让系统自身的规模变得更加合理、精巧。

Windows NT 3.1 至 3.51 这些版本采用了 Windows 3.1 窗口系统的界面风格，并且保持了对于 MS-DOS 和 Windows 3.1 应用程序的兼容性。接下来，Windows NT 4.0 于 1996 年 7 月发布，此版本完全融合了 Windows 95 的界面风格，并且对 Windows 子系统作了适当调整，尤其将 GDI 转移到了内核中，从而显著地改进了系统性能。Windows NT 4.0 是一个相对比较成熟和稳定的网络服务器操作系统，并且内置了 Microsoft IIS（Internet Information Server）作为基本的 Web 服务器软件。在随后的 3 年多时间中，Microsoft 又为 NT 4.0 版本发布了一系列补丁，更进一步完善了 NT 4.0 作为企业服务器操作系统的能力。

下一个版本是 Windows 2000，它综合了 Windows NT 4.0 及其补丁中的各种组件，像活动目录和 COM+等。而且，Windows 2000 终结了 Windows 9x 这一分支的发展，也就是说，自 Windows 2000 以后，Microsoft 的客户端操作系统和服务器端操作系统合并为一支。Windows 2000 包含 4 个版本，其中既有客户机版本，也有服务器版本。这些版本共享同样的内核。除了功能上的综合以外，Windows 2000 在安全性方面也有显著的进步，它充分发挥了 NT 系统中的安全框架，为企业网络环境提供全方位的安全服务。

此后，Microsoft 于 2001 年 8 月发布了客户端操作系统 Windows XP，以及于 2003 年 4 月发布了服务器端操作系统 Windows Server 2003。Windows XP 采用了全新的用户界面风格，并且在稳定性和运行效率方面比 Windows 2000 有了较大改进。而且，Windows XP 首次采用软件激活机制来应对软件盗版问题，实践证明这是一种有效的软件保护手段。Windows Server 2003 在稳定性、安全性和易管理性方面都超过了其前身 Windows 2000 服务器版本，是 Microsoft 公司在服务器操作系统领域的一个重要里程碑，并且也是 Microsoft 在倡导可信计算（Trustworthy Computing）以后发布的第一个操作系统。顺便提及，从 Windows XP 和 Windows Server 2003 开始，Windows 操作系统除了发行 32 位版本，也发行 64 位版本。这标志着 Windows 操作系统进入了 64 位计算的领域。

接下来的版本则是 Windows Vista 和 Windows Server 2008，它们分别是 Windows XP 和 Windows Server 2003 的替代版本。Windows Vista 于 2007 年 1 月发布，除了全面更新用户界面风格，也集成和增强了桌面搜索的能力。Windows Vista 比过去任何一个操作系统都更加关注安全性，它避免了用户直接在管理员权限下运行软件，从而防止了因滥用管理员权限而导致入侵事件发生。Windows Server 2008 集成了 Windows Vista 的内核更新，并且在内存和处理器等硬件资源的热插拔方面提供了更好的支持，以减少服务器系统需要重新引导而停机的次数和时间。另外，Windows Server 2008 对于硬件失败和进程崩溃提供了更加全面的报告机制。Windows Server 2008 所有的版本都支持多处理器，不再区分单处理器和多处理器版本。值得一提的是，Windows Server 2008 引入了对于虚拟化的支持，此特性被称为 Hyper-V。另外，Windows Vista 和 Windows Server 2008 改进了 Windows NT 文件系统 NTFS，允许在不重新启动的情况下就可以做到自我修复，而在此之前，必须重新启动之后才能修复数据损坏。

最新的 Windows 操作系统版本是 Windows 7，于 2009 年 10 月发布。相应的服务器版本为 Windows Server 2008 R2。Windows 7 的亮点并不在于它比 Windows Vista 又引入了多少新特性，而是着重于性能的提高，以及与 Windows Vista 的完全兼容性（包括应用程序和设备驱动程序的兼容）。在 Windows 7 的开发过程中，各种性能分析工具被用于测量系统在各个方面的性能，从而定位出导致性能不佳的代码路径，并改进这些细节实现。因此，在 Windows 7 系统上，从系统引导到用户的各种常见动作，性能都有了显著的提高。当然，新特性也引入了不少，例如新的任务栏（称为 Superbar）、Windows Virtual PC 和 XP Mode 等。Windows Server 2008 R2 与 Windows 7 使用同一内核版本，但只发行 64 位版本，它可以支持 64 个物理处理器或 256 个逻辑处理器。

Windows 操作系统的核心——NT 内核经过 20 年的发展，已经非常成熟，其扩展性和安全性等重要的设计目标在这段发展历程中体现得淋漓尽致。Windows 操作系统的每一个重要版本都将这一内核框架推向了新的顶峰。本书第 2 章将介绍 Windows NT 的体系结构，在 9.4 节还将介绍一些最新的发展和变化。

1.4 Windows 内核的版本

正如上一节所介绍，Windows 内核经过了 20 年的发展，其体系结构并没有大的变化。而 Windows 内核中的各个组件在经过了长期发展以后，变得更加优化和成熟。下页表 1.1 列出了 Windows 内核的版本以及相应的操作系统。

表 1.1 Windows 内核的版本列表

NT 内核版本	发布的操作系统	发布日期
NT 3.1	Windows NT 3.1	1993.7
NT 3.5	Windows NT 3.5	1994.9
NT 3.51	Windows NT 3.51	1995.5
NT 4.0	Windows NT 4.0	1996.7
NT 5.0	Windows 2000	1999.12
NT 5.1	Windows XP	2001.8
NT 5.2	Windows Server 2003	2003.3
	Windows XP SP2	2004.8
	Windows XP 64 位版本 Windows Server 2003 SP1	2005.4
NT 6.0	Windows Vista	2007.1
	Windows Server 2008 Windows Vista SP1	2008.2
NT 6.1	Windows 7 Windows Server 2008 R2	2009.10

值得重点提及的是，NT 5.2 版本是一个特殊的版本，其核心代码经过简单的改编之后已经向教育科研领域公开。这份公开源代码的内核称为 Windows Research Kernel（Windows 研究内核）[WRK]，简称 WRK。它包括了 Windows 内核中最重要的组件，例如内存管理器、进程和线程管理、对象管理器、缓存管理器、配置管理器、安全引用监视器和 I/O 管理器等。此内核源代码可以被编译成一个 EXE 可执行文件，然后安装到一个 Windows Server 2003 SP1（x86 系统）或 Windows XP 64 位（AMD）系统中，替换其中的内核模块。因此，如果用户改变了源代码中的实现逻辑，则替换了内核模块之后的 Windows Server 2003 SP1 或 Windows XP 64 位系统可以运行用户的代码逻辑。如果配置了调试环境，则还可以调试 WRK 内核和用户的代码。WRK 是 2006 年 7 月份正式对外发布的，就当时而言，它代表了最新的 Windows 内核技术。

在本书中，我们在讲解 Windows 操作系统各个部分时，重点参考 WRK 中的实现。由于 WRK 仅仅许可给教育机构，并不是每一个读者都可以合法地获得 WRK，因而，对于无法参照 WRK 来理解 Windows 的读者，我们建议他们同样安装一个 Windows Server 2003 SP1 或者 Windows XP 64 位的运行环境，甚至配置好调试环境。本书中讲解的代码逻辑你可以在调试器中检查，有些运行过程也可以在调试器中看到。不过，本书中介绍的工具并不限于在 WRK 内核上运行，它们也可以在 Windows XP 及以后的版本上运行。

虽然 WRK 并不是最新的 Windows 内核，但是，从操作系统学习和研究的角度，WRK 基本上涵盖了 Windows 操作系统 20 年发展的积累，所以，使用 WRK 来学习 Windows 内核技术并无妨碍。个别地方若在 Windows Vista 以后有大的变化，本书也会在讲解的时候指出来。

1.5 操作系统的研究与发展

操作系统是计算机科学发展的一条主线，它与计算机研究领域和工业界紧密相关。每一个成功的操作系统都需要经过几年甚至几十年的发展和检验。尽管在计算机科学发展的几十年间，有大量的操作系统问世，但真正对研究领域和工业界产生影响并能够长久流传下来的并不多。UNIX 是一个典型的代表，它于 1969 年在贝尔实验室问世以来，已经发展出许多分支，形成了各种版本，包括 UNIX System V、BSD、Solaris、HP-UX、AIX 和 IRIX 等。其中有些系统已经公开源代码，而另一些则完全由商业公司控制。UNIX 操作系统经过了 40 年的发展，几乎在各种硬件体系结构上都有它的版本，它已经被广泛应用于各个领域，尤其在大型数据处理和计算领域，始终占据着重要的地位。UNIX 操作系统仍然在发展，除了在技术方面与硬件体系结构和 Internet 的发展保持同步以外，在软件发行和许可证管理方面也在发展，以迎合现代信息社会对系统软件的多样性需求。

在 UNIX 操作系统大家族中，有三种完全开放源代码的 UNIX 类操作系统：Linux、BSD（Berkeley Software Distribution）和 Minix，它们与教育研究领域的关系尤为密切。Linux 是 20 世纪 90 年代初为 Intel 386 体系结构开发的 UNIX 类操作系统，目前已被移植到几乎所有的硬件体系结构平台，它可以运行在个人计算机、PC 服务器、中大型计算机系统上。Linux 也是 Internet 发展的产物，其核心代码来自于自由软件爱好者们的贡献，Linux 的发行遵从 GPL（General Public License）许可，该许可要求凡是在 Linux 源代码基础上扩展的新技术也必须公开源代码。BSD 使用了简化的许可证管理模式，它包含多个版本，其中比较常见的是三个发行版：FreeBSD、NetBSD 和 OpenBSD。FreeBSD 侧重于对个人计算机体系结构的支持，主要运行在 Intel 386 体系结构平台上；NetBSD 支持大量的硬件体系结构，为用户提供一个规模小、可移植性强的操作系统选择；OpenBSD 的重点是系统安全性，其开发人员针对软件错误（bug）和安全漏洞审核源代码，它的目标是用于安全性要求高的场合，比如防火墙。Minix 是一个基于微内核结构的 UNIX 类操作系统，其作者 Andrew S. Tanenbaum 编写 Minix 的目的是用于操作系统教学。Minix 在教学中的运用激发了 Linux 的诞生，并且也影响了 Linux 在构造过程中的许多设计决策。

与 UNIX 类操作系统的发展完全不同的是，Windows 是一个完全由商业公司来管理和控制的操作系统，在 2006 年以前它不开放任何源代码。目前它开放的研究内核源代码（即 WRK）仍然侧重于教育目的。Windows 比 UNIX 晚了整整 20 年，它的定位是个人计算机操作系统。由于硬件技术的飞速发展，Windows 操作系统也适用于服务器系统，甚至构建大型集群系统。目前 Windows 仍然是一个活跃的、发展中的操作系统。

在现代操作系统的发展历程中，先后有多种内核结构被提了出来。最原始的内核结构是大内核（macrokernel 或 monolithic kernel）结构。使用大内核结构的操作系统，其本身是一个程序，所有基本的 OS 服务都运行在同一内存空间中，包括设备驱动程序提供的服务。在 20 世纪 80 年代以前构建的操作系统都是大内核操作系统，即使到今天，大内核仍然是操作系统主流的内核结构，像大多数 UNIX 类操作系统、Windows 9x 和 Linux 系统都属于大内核操作系统。大内核操作系统的优点是，内核组件之间直接通过函数调用进行通信，所以执行效率高，内核中所有的操作都在特权模式下进行；但是它的缺点也不可忽略：系统组件和设备驱动程序运行在同一内存空间中，一旦发生错误（bug），则整个系统崩溃；另一方面，大内核内部组件之间的依赖性容易导致系统难以维护，以及出现意外的错误。

为了解决大内核操作系统日益增长的复杂性及其带来的各种问题，研究人员提出了微内核（microkernel）结构。在微内核操作系统中，只有直接建立在硬件基础上的一组最小系统服务（例如内存管理、多任务支持和进程间通信）由内核来实现，其他常见的服务，包括网络服务、文件系统等，都在用户进程中实现，它们通过内核提供的进程间通信（IPC）服务进行通信。微内核结构的优势是易于维护，小的内核更容易验证和测试其正确性，而更多的系统服务可以以高级语言来编写，并且可以在运行时动态替换或恢复；但是微内核结构达成这些优势的代价是牺牲效率：大量的处理器模式切换以及进程环境切换，使得其系统开销比大内核结构中的函数调用要大得多。纯粹的微内核结构并不多，Minix[MINIX]、QNX[QNX]和 Mach[MACH]是微内核操作系统的典型代表。L4 是一个微内核接口，代表了一族采用此内核接口的操作系统[L4-OS]。在实践中，有些操作系统综合了大内核和微内核的优势，将有些系统组件转移到了用户模式下，但仍然维护一个不小的内核，以达到在内核结构的简化和性能两方面的合理折中，这被称为混合内核（hybrid kernel）结构。Windows NT 和 Linux 虽然常常被当做大内核操作系统，但实际上它们属于混合内核结构。

与微内核结构较为接近但又独树一帜的是 exokernel，称为外核结构[EXOKERNEL]。与传统操作系统理念不同的是，exokernel 提供尽可能少的硬件抽象，从而使应用程序开发人员对于硬件的使用有尽可能多的自主决策权。所以，exokernel 结构中的内核比微内核还要小，本质上它们仅仅提供硬件资源的保护和共享使用。应用程序通过一些库操作系统

（library operating system）来使用系统的硬件资源。Exokernel 使得应用程序可以以最有效的方式来使用系统的硬件，避免因过于通用而带来的性能损失。然而，exokernel 内核结构自 1995 年发表以来，仍然是一个理论模型，尚无重要的商用操作系统采用此模型。

对硬件进行抽象是操作系统的本质任务，而硬件抽象的一个极端是硬件虚拟化（virtualization）。硬件虚拟化的做法是，在给定的硬件平台上运行一个控制程序（称为宿主软件，host software），该程序为一个或多个客体软件（guest software）建立一个模拟环境，此模拟环境称为虚拟机（virtual machine）。所以，硬件虚拟化也称为虚拟机，而控制程序称为 VMM（Virtual Machine Monitor）。虚拟机技术需要系统软件和硬件平台的有机结合才有可能，IBM 在 20 世纪六七十年代已经在 IBM S/370 系统中引入了虚拟机技术[IBM-S370]。在 Intel x86 系统环境中，VMware 在不利用硬件特性支持的情况下，使用“二进制代码翻译（binary translation）”技术自动修改 x86 软件中的指令，达到了完全虚拟化。2005 年，Intel 在 x86 处理器中引入了专门的虚拟化支持（称为 Intel VT），从而使虚拟机模拟环境的执行效率更高。在 Intel x86 平台上，VMware 和 Microsoft Virtual PC 是完全虚拟化的典型代表。本书附录 A 介绍了如何通过 Virtual PC 来调试虚拟机中的 Windows 系统。另一类值得一提的虚拟化是 paravirtualization（仿虚拟化），典型的代表是 Xen 虚拟机[XEN]，它的 VMM 并非全是模拟硬件，而是提供了一个特殊的 API 供客体 OS 使用，因而客体 OS 能够更加高效地使用宿主 OS 中的硬件资源，但是，这种虚拟机技术要求修改客体 OS，这在某种程度上限制了它的应用范围。

从功能角度来看，现代操作系统的发展已经渐趋成熟，但是，随着网络环境的发展，以及硬件平台的多样化，现代操作系统也迫切需要一些新的结构模型或技术来迎合市场和用户的需求，其中安全性是现代操作系统发展过程中越来越被重视的一个特性。Singularity 是 Microsoft 研究人员按照现代软件的安全性需求重新设计的一个操作系统原型[SINGULARITY]，它使用类型安全的编程语言和抽象指令集合使得系统的进程之间有严格的隔离保证，其进程称为 SIP（Software Isolated Process）。每一个程序、设备驱动程序或系统的扩展功能模块都运行在自己独有的 SIP 中，每个 SIP 有自己的对象空间、GC（垃圾回收器）和运行库。另一方面，从计算能力角度来看，现在的个人计算机正朝着多处理器和多核的方向发展，所以，操作系统如何更好地安排这些计算资源，以便于应用程序有机会发挥这些处理器的硬件优势，已成为一个新的研究课题。仅仅在原有操作系统基础上简单地扩展线程调度和进程管理模块，并不一定是最有效的做法，例如 Corey 原型系统证明了让应用程序来控制处理器（或核）之间的数据共享，比使用 Linux 的多任务调度机制有显著的性能提高[COREY]。

操作系统的研究既是一个老的主题，也仍然是一个活跃的主题。近几年随着云计算、

移动网络、智能手机等一些相关研究领域的兴起，操作系统正面临着更多的挑战和机遇。在客户端，操作系统和浏览器技术的结合[GOOGLE-OS]有可能会改变用户使用计算机或手持设备的模式；而在服务器端（或云端），应用程序或服务的运行环境将变得更加复杂，甚至完全独立于实际的硬件环境。有关操作系统的最新研究状况，感兴趣的读者可以参考最近几年在 USENIX、ACM、IEEE Computer Society 等组织的一些有关计算机系统的会议或定期刊物上发表的文章。

1.6 本章总结

操作系统是一种特殊的软件，它一方面管理系统中的硬件资源和设备，另一方面为应用软件提供基本的系统服务。由于它需要直接跟硬件打交道，所以，为了学习像 Windows 这样的复杂操作系统，基本的硬件体系结构知识和汇编语言、C 语言的编程技能是必不可少的。本章也描述了 Windows 操作系统的 20 年发展历程，以及它的内核版本变化，最后还介绍了现代操作系统的研究与发展情况。

从下一章开始，我们将集中在 Windows 操作系统的基本原理与具体实现上，并首先从 Windows 的总体结构开始全面地认识 Windows。

2

Windows 系统总述

这一章首先讲述 Windows 系统的基本结构。作为一个典型的现代操作系统，Windows 有着广泛的用户群体，并且自诞生以来就一直备受关注。在讲述了 Windows 基本结构以后，本章将简要地介绍 WRK（Windows Research Kernel），这是 Microsoft 提供的一套可以编译和运行的 Windows 内核源代码，本书后面章节的绝大多数讲解都将参考 WRK 中的源代码。

本章也将介绍 Windows 系统中的基本概念以及 Windows 内核中的一些公共管理设施，这些知识不仅有助于读者全面认识 Windows 操作系统，而且也便于本书后面的章节直接引用 Windows 内核中的概念和管理机制。最后，本章还将介绍 Windows 的引导过程。从系统启动一直到内核能够正常工作，再到用户登录到系统中，Windows 系统中的各个组件都要完成相应的初始化任务。

2.1 现代操作系统的基本结构

操作系统本身属于软件的范畴，但是它需要紧密地跟硬件打交道，它为上层应用软件或应用系统提供了一层抽象，专门负责硬件资源的管理和分配。应用软件不需要直接跟硬件打交道，它们利用操作系统提供的功能来实现各种应用任务，如果它们要访问硬件，则必须通过操作系统提供的抽象接口来完成。图 2.1 显示了现代操作系统的一般性框架结构。

图 2.1 中的模型可能会有一些变化形式，比如，有些专用机器的系统并没有提供内核扩展性能，所有的硬件驱动模块都链接在系统内核中。但通用机器的系统倾向于提供可动态加载的驱动程序或专门的扩展接口，Windows 和 Linux 就是这类系统的典型代表。另外，在系统服务层之上，应用软件通过一层接口来调用系统提供的服务，而应用程序之间则保

持相对隔离，它们相互间的通信需要通过系统来完成。操作系统提供了一些基本的 IPC（Inter-Process Communication，进程间通信）原语来支持上层应用程序之间的交互操作。系统内核负责管理这些应用程序，把有限的计算资源分配给它们，使它们的任务得以完成。例如，计算机的处理器（即 CPU）、内存和外存（如硬盘）由系统内核统一管理和分配，因而多个应用软件可以同时运行在一个系统中，它们既可能毫不相关，也可能相互依赖，操作系统会协调它们的运行过程。

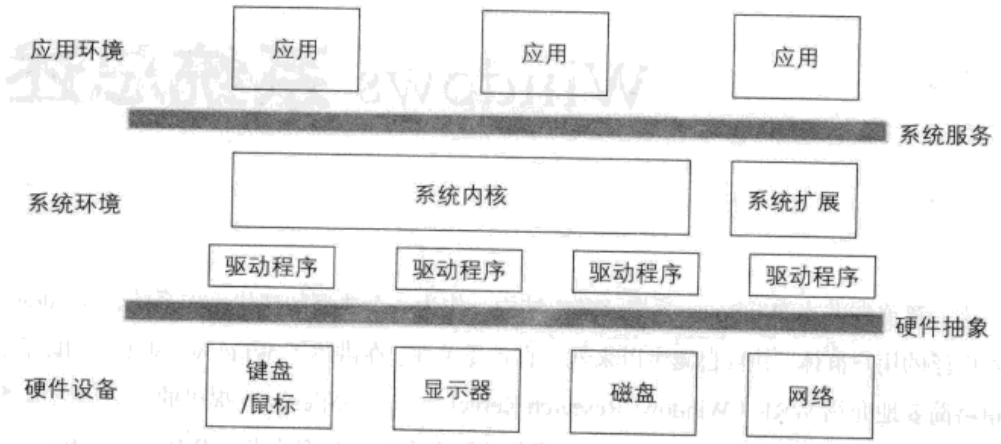


图 2.1 现代操作系统的基本结构

硬件设备，像键盘、显示器、磁盘、打印机和网络等，则通过各自的驱动程序，以一致的方式交由操作系统来处理。系统内核接受应用程序的请求，与硬件设备进行通信；另一方面，硬件设备向计算机发送信号，驱动程序收到信号后，与系统内核一起把信号传递给恰当的应用程序。对于通用的操作系统，为众多外部设备提供支持是系统执行环境的一个重要方面。为了灵活、方便起见，驱动程序往往是可替换或可升级的，它们构成了系统执行环境的一个重要组成部分，在代码量方面甚至超过了系统内核本身。

在现代计算机系统中，仅仅简单地支持外部设备往往还不够，设备的即插即用（plug-and-play）和电源管理越显重要。这一方面要求系统总线能够检测到设备的插入和拔除，另一方面也要求操作系统能够动态地加载或移除驱动程序，同时保持系统状态的完整性。电源管理不仅会影响到操作系统自身的状态管理，也需要获得驱动程序和硬件设备的支持，所以，电源管理自然成了设备驱动程序接口的一部分。

2.2 Windows 系统结构

上一章已经提到，Windows 内核（由于是从 Windows NT 发展起来的，也称为 NT 内

核）从一开始就有良好的设计，其结构具备很好的可扩展性和安全性。所以，Windows 内核在 20 年的发展历程中一直能够很好地适应硬件的发展，在 Windows 操作系统的各个版本中并没有根本性的变化。这一节将介绍 Windows 操作系统的基本框架，这些内容完全适用于 Windows XP/Server 2003 及以后的版本。

图 2.2 显示了 Windows 基本结构。Windows 采用了双模式（dual mode）结构来保护操作系统本身，以避免被应用程序的错误所波及。操作系统核心运行在内核模式（kernel mode）下，应用程序的代码运行在用户模式（user mode）下。每当应用程序需要用到系统内核或内核的扩展模块（内核驱动程序）所提供的服务时，应用程序通过硬件指令从用户模式切换到内核模式中；当系统内核完成了所请求的服务以后，控制权又回到用户模式代码。

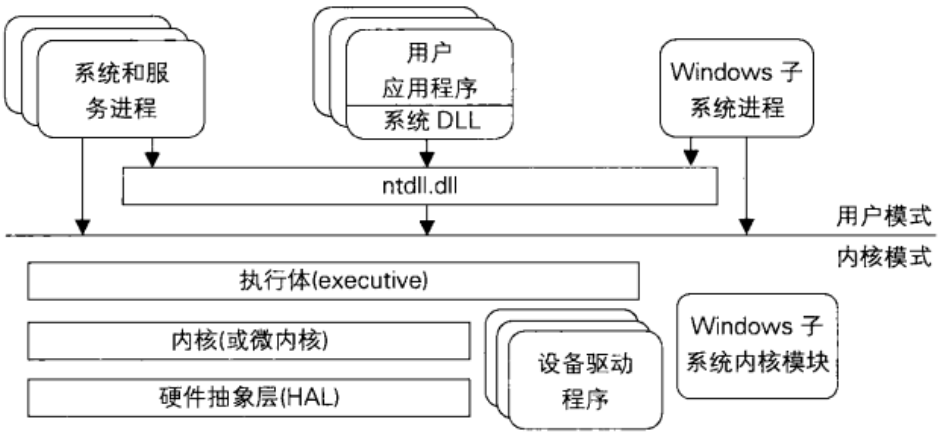


图 2.2 Windows 系统结构图

在 Windows 中，用户代码和内核代码有各自的运行环境，而且它们可以访问的内存空间也并不相同。在 32 位系统中，内核代码可以访问当前进程的整个 4 GB 虚拟地址空间，而用户代码只能访问底端的 2 GB 虚拟地址（或 3 GB，如果打开了内核启动开关/3GB 的话）。

Windows 子系统是 Windows 系统中一个不可缺少的组成部分，它与系统内核一起构成了用户应用程序的执行环境。Windows 的原始设计是一个支持多环境子系统的操作系统，除了 Windows 子系统作为它的原生环境子系统，它还支持 POSIX 和 OS/2 环境子系统，为 UNIX 类应用程序和 OS/2 应用程序提供一个仿真执行环境。随着 Windows 操作系统的发展，自 Windows XP 以后，只有 Windows 子系统随系统一起发行。Windows 子系统既有内核模式部分（图形和窗口管理），也有用户模式部分。用户模式部分包括一个单独的子系统进程和一组链接到各个应用进程中的系统 DLL。

Windows 操作系统还包括一组系统进程和服务进程，它们为操作系统提供了关键的服

务，比如会话管理、用户登录和注销、身份验证，以及打印服务、事件日志和任务调度等。这些系统进程和服务进程也是整个操作系统运行环境的一部分。

2.2.1 Windows 内核结构

正如图 2.2 所示，Windows 内核分为三层，与硬件直接打交道的这一层称为硬件抽象层（Hardware Abstraction Layer，简称 HAL），这一层的用意是把所有与硬件相关联的代码逻辑隔离到一个专门的模块中，从而使上面的层次尽可能做到独立于硬件平台。HAL 之上是内核层，有时候也称为微内核（micro-kernel），这一层包含了基本的操作系统原语和功能，如线程和进程、线程调度、中断和异常的处理、同步对象和各种同步机制。在内核层之上则是执行体（executive）层，这一层的目的是提供一些可供上层应用程序或内核驱动程序直接调用的功能和语义。Windows 内核的执行体包含一个对象管理器，用于一致地管理执行体中的对象。执行体层和内核层位于同一个二进制模块中，即内核基本模块，其名称为 ntoskrnl.exe。

内核层和执行体层的分工是，内核层实现操作系统的基本机制，而所有的策略决定则留给执行体。执行体中的对象绝大多数封装了一个或者多个内核对象，并且通过某种方式（比如对象句柄）暴露给应用程序。这种设计体现了机制与策略分离的思想。图 2.3 显示了 Windows 内核的详细组成结构。

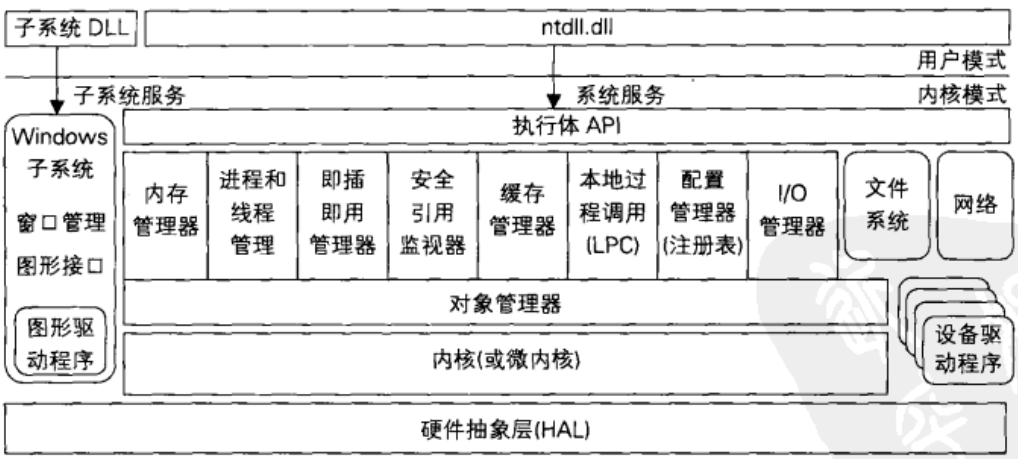


图 2.3 Windows 内核的组成结构

Windows 内核为用户模式代码提供了一组系统服务，供应用程序使用内核中的功能。应用程序通常并不直接调用这些系统服务，而是通过一组系统 DLL，最终通过 ntdll.dll 切换到内核模式下的执行体 API 函数中，以调用内核中的系统服务。Ntdll.dll 是连接用户模式代

码和内核模式系统服务的桥梁。对于内核提供的每一个系统服务，该 DLL 都提供一个相应的存根函数，这些存根函数的名称以“Nt”作为前缀，例如 NtCreateProcess、NtOpenFile 和 NtSetTimer。另外，ntdll.dll 还提供了许多系统级的支持函数，比如映像加载器函数（以“Ldr”为前缀）、Windows 子系统进程通信函数（以“Csr”为前缀）、调试函数（以“Dbg”为前缀）、系统事件函数（以“Etw”为前缀），以及一般的运行支持函数（以“Rtl”为前缀）和字符串支持函数等。

执行体 API 函数接收的参数来自于各种应用程序，因此，为了确保系统的健壮性，以及抵抗来自用户模式的恶意攻击，所有的执行体 API 函数必须保证参数的有效性。这意味着它们必须在恰当的时刻检查参数的值，若是指针的话，还必须保证调用者可以访问指针所指的内存。通常，执行体系统服务函数会在其开始处，对所接收的参数逐一探查它们的可访问性。例如，以下代码就演示了这一做法：

```
PreviousMode = KeGetPreviousMode();
if (PreviousMode != KernelMode) {
   try {
      ProbeForWrite(InputInformation,
            InputInformationLength,
            sizeof(ULONG));
      if (ARGUMENT_PRESENT(ReturnLength)) {
         ProbeForWriteUlong(ReturnLength);
      }
   } except(EXCEPTION_EXECUTE_HANDLER) {
      return GetExceptionCode();
   }
}
```

这里 InputInformation 和 ReturnLength 是该系统服务的直接参数，InputInformation 是个指针参数，它的可用长度由另一个参数 InputInformationLength 来指定，因此，这段代码通过 ProbeForWrite 来探查这一输入数据是否可写；另外，ReturnLength 是一个 ULONG 类型的输出参数，这段代码通过 ProbeForWriteUlong 来验证该参数是否可写。一旦探查函数发生访问违例，则说明这两个参数有问题，于是该系统服务就将直接返回违例异常代码。

正如前文所述，用户模式和内核模式代码所能访问的地址空间有所不同。在 32 位系统上，用户模式代码只能访问 2 GB 以下的虚拟内存地址空间，而内核模式代码可以访问当前进程整个 4 GB 虚拟地址范围。2 GB 以下称为进程地址空间，2 GB 以上称为系统地址空间。实际上，在两者之间有一块特殊的 64 KB 地址空间，位于 0x7fff0000~0x7fffffff，在两种模式下都不可访问。上述代码片段中的 ProbeForWrite<*Xxx*>函数首先检查目标内存地址是否越过了此特殊区域，若越过则访问违例；然后，试图将目标地址处的值赋回该地址，即触发一次该地址处的读和写操作，若该内存地址处当前线程不可写，则引发异常，从而使代码片段中的 except 子句截获控制。Windows 通过这种方法来捕捉到“用户模式

代码传递一个系统空间地址”或者“传递一个无效内存地址”的情形，从而确保执行体函数接收到的参数已被检验过。

然而，执行体 API 函数并不总是要检验参数的有效性，如果在调用该 API 函数以前的模式是内核模式，那么，它不需要检验参数的有效性。执行体不会用坏的参数来调用它自己的服务，只有当执行体 API 函数接收到一个或多个来自用户模式的参数时，它才使用 Probe 函数族检查参数的有效性。每个线程都维护着一个状态值，用以说明它以前的处理器模式。当从用户模式切换到内核模式时，该值将被设置为 UserMode，从而满足上述代码中的 if 条件。

2.2.2 Windows 内核中的关键组件

Windows 操作系统虽然算不上真正意义上的微内核结构，但是它的内核部分有良好的设计以及清晰的模块结构，如前面图 2.3 所示。现在我们来逐一介绍内核部分的各个关键组件，不过 Windows 子系统部分将留到下一小节单独讲述。

HAL（硬件抽象层）

HAL 的设计目的是将硬件的差别隐藏起来，从而为操作系统的上层提供一个抽象的、一致的硬件资源模型，以使 Windows 更容易被移植到不同的硬件平台上。理想的情形是，只要硬件厂商能够提供一个 HAL，Windows 就能够在相应的硬件平台上运行。因此，HAL 使得上层的模块无须考虑硬件的差异，它们通过 HAL 而不是直接访问硬件。

在 Windows 中，HAL 是一个独立的动态链接库。尽管 Windows 随带了多个主流机器的 HAL，但是在系统安装的时候只有一个会被选中，并拷贝和改名为 hal.dll。HAL 提供了一些例程供其他内核模块或设备驱动程序调用，这使得一个驱动程序可以支持同样的设备在各种硬件平台上运行。HAL 不仅涵盖了处理器的体系结构，也涉及了中断控制器、单处理器或多处理器等硬件条件。表 2.1 列出了在 Intel x86 机器上 Windows Server 2003 系统中随带的 HAL。

表 2.1 Windows Server 2003 的 HAL 列表（Intel x86 处理器）

HAL 文件	所支持的硬件系统
Hal.dll	标准 PC
Halacpi.dll	ACPI（高级配置和电源接口）PC
Halapic.dll	APIC（高级可编程中断控制器）PC
Halaacpi.dll	APIC ACPI PC
Halmps.dll	多处理器 PC
Halmacpi.dll	多处理器 ACPI PC

内核（或微内核）

这是大内核中的小内核，将其称为微内核更可以说明它在整个内核模式代码中的地位。它是内核模块 ntoskrnl.exe 中的下层部分（上层部分为执行体），最接近于 HAL 层，负责线程调度和中断、异常的处理。对于多处理器系统，它还负责同步处理器之间的行为，以优化系统的性能。这一层的核心任务是，让系统中的所有处理器尽可能地忙和高效。内核层可在多个处理器上并发执行，它的代码以 C 语言为主，也包含一部分汇编代码。

Windows 的内核实现了抢占式线程调度机制，按照优先级顺序将线程分配到处理器上，并且允许高优先级的线程中断或抢占低优先级的线程。每个处理器上的线程切换也是由内核来完成，它按照调度规则让处理器放弃当前线程，选择下一个要执行的线程。每个线程有一个基本优先级值（base priority），这是由程序在创建线程时指定的；每个线程还有一个动态优先级值，这是在线程执行过程中根据各种条件在基本优先级基础上由内核来调整的，目的是让系统更快地响应用户的动作，以及在系统服务和其他低优先级进程之间平衡处理器资源的分配。

Windows 的内核按照面向对象的思想来设计，它管理两种类型的对象：分发器对象（dispatcher object）和控制对象。分发器对象实现了各种同步功能，这些对象的状态会影响线程的调度。Windows 内核实现的分发器对象包括事件（event）、突变体（mutant）、信号量（semaphore）、进程（process）、线程（thread）、队列（queue）、门（gate）和定时器（timer）。控制对象被用于控制内核的操作，但是不影响线程的调度，它包括异步过程调用（APC）、延迟过程调用（DPC），以及中断对象等。

内核层位于 HAL 之上，但鉴于内核所提供功能与硬件体系结构的紧密关联性，它不可避免地需要引入一些与体系结构相关的代码，例如，在切换线程时，保存和恢复线程的执行环境取决于处理器体系结构。不过，如何选择下一个线程，这是与体系结构无关的。内核有义务将 Windows 所支持的各种硬件体系结构进行抽象，使得体系结构的差异对 Windows 代码的影响尽可能地小，并且有些功能可以通过 HAL 来完成，毕竟 HAL 才是真正的硬件抽象层。例如自旋锁和中断的功能是在 HAL 中实现的，内核只需简单地使用 HAL 的导出函数即可。

执行体

执行体是内核模块 ntoskrnl.exe 的上层部分，它包含 5 种类型的函数：

- 被导出的、可在用户模式下调用的函数。对这些函数的调用接口位于 ntdll.dll 模块中。应用程序通过 Windows API 来间接地调用这些函数。

- 虽已被导出并且可在用户模式下调用，但无法通过任何一个 Windows API 来调用的函数。这样的例子包括 LPC（Local Procedure Call，本地过程调用）函数、各种查询函数（如 NtQueryInformation<*Xxx*>），以及一些专用的函数，比如 NtCreatePagingFile 等。对这些函数的调用需要直接链接 ntdll.dll 来完成。
- 只能在内核模式下调用的导出函数，并且在 Windows DDK 中有关于这些函数的文档。这些函数可以被设备驱动程序调用。
- 供执行体组件之间相互调用，但未被文档化的函数。这包括执行体内部使用的一组支持函数。
- 属于一个组件的内部函数。

以上提到的组件是指执行体内部的组件，从大的方面来看，执行体包含以下组件（参考图 2.3）：

- 进程和线程管理器。负责创建进程和线程，以及终止进程和线程。在 Windows 中，对于进程和线程的底层支持是在内核层提供的，执行体在内核层的进程和线程对象的基础上，又提供了一些语义和功能。
- 内存管理器。此组件实现了虚拟内存管理，既负责系统地址空间的内存管理，又为每个进程提供了一个私有的地址空间，并且也支持进程之间内存共享。内存管理器也为缓存管理器提供了底层支持。
- 安全引用监视器（SRM，Security Reference Monitor）。该组件强制在本地计算机上实施安全策略，它守护着操作系统的资源，执行对象的保护和审计。
- I/O 管理器。它实现了与设备无关的输入和输出功能，负责将 I/O 请求分发给正确的设备驱动程序以便进一步处理。
- 缓存管理器。它为文件系统提供了统一的数据缓存支持，允许文件系统驱动程序将磁盘上的数据映射到内存中，并通过内存管理器来协调物理内存的分配。
- 配置管理器。它负责系统注册表的实现和管理。
- 即插即用管理器。它负责列举设备，并为每个列举到的设备确定哪些驱动程序是必需的，然后加载并初始化这些驱动程序。当它检测到系统中的设备变化（增加或移除设备）时，负责发送恰当的事件通知。
- 电源管理器。它负责协调电源事件，向设备驱动程序发送电源 I/O 通知。当系统电源

状态变化时，通知设备驱动程序处理设备的电源状态。即插即用设备的管理和电源的管理也可以看做是 I/O 管理器的扩展功能。

此外，执行体还包含 4 组主要的支持函数，供以上这些执行体组件调用。差不多有 1/3 的支持函数可以在 Windows DDK 中找到相应的文档，因为设备驱动程序也要调用它们。这 4 类支持函数如下所列：

- 对象管理器。它负责创建、管理和删除 Windows 执行体对象，以及用于表达操作系统资源的抽象数据类型，比如进程、线程和各种同步对象。
- LPC 设施。LPC 设施负责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，远程过程调用，关于网络上客户进程和服务器进程之间通信的工业标准）的一个优化版本。
- 一组运行时库函数。其功能广泛，涵盖字符串处理、算术运算、数据类型转换以及安全结构处理等。
- 执行体支持例程。例如系统内存分配（换页内存池和非换页内存池）、互锁的内存访问，以及对两种特殊类型同步对象（资源和互斥体）的支持。

设备驱动程序

在内核中除了内核模块 ntoskrnl.exe 和 HAL 以外，其他的模块几乎都以设备驱动程序的形式存在。Windows 操作系统中的设备驱动程序，并不一定对应于物理设备；驱动程序既可以创建虚拟设备，也可以完全与设备无关，而仅仅是内核的扩展模块。从软件结构角度而言，我们可以认为设备驱动程序是 Windows 内核的一种扩展机制，系统通过设备驱动程序来支持新的物理设备或者扩展功能。

设备驱动程序是可以动态加载到系统中的模块，其文件扩展名为.sys，其格式是标准的 PE 文件格式。驱动程序中的代码运行在内核模式下，尽管它们可以直接操纵硬件，但理想的情况是，调用 HAL 中的函数与硬件打交道，因此，驱动程序往往用 C/C++语言来编写，从而可以方便地在 Windows 所支持的体系结构之间进行源代码层次上的移植。

PE 文件格式

PE（Portable Executable）[PE-SPEC]是在 Windows NT 设计之初，为了建立一种可移植的、能够适应 32 位操作系统需要的可执行文件格式而设计的。这里“可移植”的目标是指当初 Windows NT 可在多种处理器（x86、MIPS、Alpha 等）上运行，不同体系结构平

台上的Windows NT使用相同的二进制可执行文件格式。Windows发展至今，现在仅支持Intel的处理器，但PE文件格式仍然不失为一种定义良好且具有可扩展性的可执行文件格式，非Windows NT内核的操作系统Windows 9x系列也使用PE文件格式，因此，PE仍然是一种可移植的可执行文件格式。

PE格式扩展了COFF(Common Object File Format)，这是UNIX(AT&T UNIX System V)中引入的用于描述二进制目标文件的格式规范。尽管COFF为目标文件定义了一个很好的框架，比以前的a.out格式改进了很多，但它仍然有一些限制，比如文件中段(section)的数量有最大限制，段的名称也有长度限制，无法支持像C++语言所需要的符号化调试信息等。因此，每一个采用COFF的操作系统厂商几乎都或多或少地对其进行了扩展，比如AT&T自己又在COFF基础上定义了ELF(Extensible Linking Format)，ELF目前被广泛用于各种UNIX类操作系统，包括Linux和FreeBSD；IBM在AIX中使用XCOFF；而Microsoft则在COFF格式规范基础上定义了PE格式。在Windows平台上，可执行文件（扩展名为.exe）、目标文件（扩展名为.obj）、动态链接库（扩展名为.dll）以及设备驱动程序（扩展名为.sys）等多种文件类型使用了PE文件格式。随着64位系统的到来，PE文件格式也相应地有了一个扩展的版本，称为PE32+，允许使用64位地址空间；原来的32位PE格式称为PE32。本书中我们仅涉及32位PE格式。

PE文件的基本结构包括DOS头、PE头、段表，以及段数据，如图2.4所示。DOS头的存在仅仅是为了兼容以前的MS-DOS系统环境，其中嵌了一段代码可在MS-DOS环境下打印出“This program cannot be run in DOS mode”字样的提示信息。DOS头后面是4字节的PE标志“PE\0\0”，接着是PE头部分。PE头分两部分：COFF头和可选头，对于可执行映像文件，可选头也是必需的。PE头部分包含了有关当前文件的一些全局信息，包括段的数量、符号信息、可执行文件的基地址和入口函数地址、代码段和数据段的大小、版本信息、栈和堆的初始大小，以及数据目录（data directory）等信息。PE头中的数据目录共有16项，这些目录项指示了一个可执行文件的导入表、导出表、资源表、重定位表等。

在PE头的后面是段表（section table），段表中的每一项指示了段的名称、段的虚拟地址位置和大小，以及其他一些有关该段的信息。对于可执行映像，段名称的长度不超过8个ASCII字符。段的个数由PE头中的一个域指定。在段表后面是段数据，对于每一个段，在文件中的数据量可能小于该段的大小，在这种情况下，段后面的数据补零。例如，未初始化数据部分无须在文件中包含任何有效数据。另一个值得注意的特点是，段的位置按照PE头中指定的边界对齐，所以，在段数据区，段与段之间可能有空隙。

当 Windows 系统加载一个可执行映像时，如果可执行映像中指定的基地址处已经被占用，那么该可执行映像将不得不被加载到其他的位置。为了让可执行映像中的代码区和数据区在移动位置以后仍然有效，PE 文件格式中的重定位表可以指示代码段和数据段中哪些位置（这是相对位置）的地址引用需要被重新定位。所以，在加载可执行映像文件时，一旦被加载的内存起始地址与 PE 头中指定的基地址不一致，则必须要执行重定位操作。所谓重定位操作，是指遍历重定位段表中的每一个重定位项，将当前映像中的代码指令或数据引用按照新的内存起始地址而非原来指定的基地址来引用。

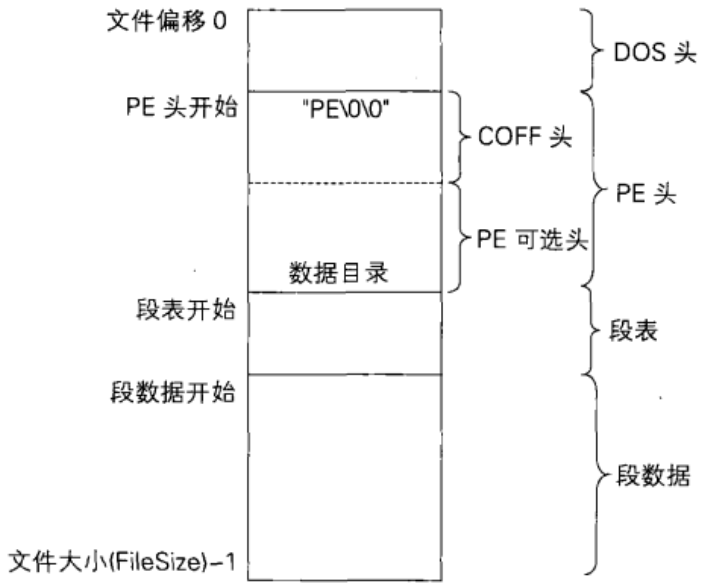


图 2.4 PE 文件的基本结构

有关 PE 文件格式的详细信息，请参考 PE 文件格式规范[PE-SPEC]。另一个有用的参考资料是 Matt Pietrek 写的两篇文章[MSDN-PE1][MSDN-PE2]，他在文章中通过代码和例子介绍了如何解析一个可执行文件，包括 PE 头的解析和重定位的概念等。他提供的 PEDUMP 程序（有源代码）本身也是一份很好的学习和参考材料。

另外，Visual C++提供的工具 dumpbin 可以列示出一个 PE 文件的各种信息，例如，下面显示了 Windows Server 2003 SP1 系统中的文件 notepad.exe 的头信息：

```
C:\>dumpbin D:\Win2k3-Exe\notepad.exe /headers
Microsoft (R) COFF/PE Dumper Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file notepad.exe

PE signature found

File Type: EXECUTABLE IMAGE
```

```
FILE HEADER VALUES
           14C machine (x86)
             3 number of sections
      42435B9A time date stamp Fri Mar 25 08:30:18 2005
             0 file pointer to symbol table
             0 number of symbols
            E0 size of optional header
           10F characteristics
                Relocations stripped
                Executable
                Line numbers stripped
                Symbols stripped
                32 bit word machine

OPTIONAL HEADER VALUES
…… （这里省略了可选头部分的内容）

SECTION HEADER #1
   .text name
    7760 virtual size
    1000 virtual address (01001000 to 0100875F)
    7800 size of raw data
     400 file pointer to raw data (00000400 to 00007BFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
60000020 flags
         Code
         Execute Read

  Debug Directories

Time     Type    Size  RVA       Pointer
-------- ------ ----- -------- --------
42435B9A cv      24    00001910 D10    Format: RSDS, {B4CD0BCE-C210-4
934-8161-DFC07F9870B0}, 1, notepad.pdb

…… （这里省略了 SECTION HEADER #2 和 SECTION HEADER #3 部分的内容）

  Summary

          2000 .data
          9000 .rsrc
          8000 .text
```

概括而言，设备驱动程序有以下三种基本类型：

- 即插即用驱动程序（也称为 WDM 驱动程序，见下文介绍）。这一类驱动程序通常是为了驱动硬件设备而由硬件厂商提供，它们与 Windows 的 I/O 管理器、即插即用管理器和电源管理器一起工作。Windows 自身随带了大量即插即用驱动程序，用于支持各种常见的存储设备、视频适配器、网络适配器、输入设备等。
- 内核扩展驱动程序（也称为非即插即用驱动程序）。这一类驱动程序用于扩展内核的

功能，或者提供访问内核模式代码和数据的一种途径。它们并没有集成到即插即用管理器和电源管理器的框架中。在引入即插即用管理机制以前开发的驱动程序都属于这一类型。现在仍然有大量的内核扩展驱动程序。

- 文件系统驱动程序。这一类驱动程序接收针对文件的请求，再进一步将请求转变成真正对于存储设备或网络设备的 I/O 请求，从而满足原始的文件请求。

Windows 的即插即用管理器是 I/O 系统的一部分，它负责即插即用设备的内核支持，其职责是：自动检测设备的插入和移除；动态地分配硬件资源，例如中断、I/O 端口和 I/O 寄存器；指示 I/O 管理器为设备加载正确的驱动程序；向内核及应用程序提供有关设备插入和移除的通知机制。即插即用管理器根据总线和设备的功能分工，定义了一个驱动程序模型，让总线和设备的驱动程序协作完成设备的列举、插入和拔除等管理工作。支持这一模型的驱动程序称为 WDM（Windows Driver Model）驱动程序，共有三种类型：总线驱动程序、功能驱动程序和过滤驱动程序。总线驱动程序既负责管理总线上的设备（配合即插即用管理器），也为总线上的设备提供了访问总线资源的方法。功能驱动程序负责管理具体的设备，向操作系统提供该设备的功能。过滤驱动程序的用途是监视一个设备的 I/O 请求及其处理过程，甚至增加或改变一个设备或驱动程序的行为。

在 WDM 中，每个硬件设备都有一个设备驱动程序栈（简称设备栈），其中包含一个总线驱动程序和一个功能驱动程序，以及零个或多个过滤驱动程序。即插即用管理器在设备列举过程中，依照总线与设备之间的关系，构建起一棵设备树，其中包含当前系统中所有被检测到的总线和设备。设备树的每一个节点都代表一个实际的设备，该设备的设备栈为其提供软件服务，操作系统（实际上是 I/O 管理器）通过设备栈来访问或操纵设备。

非即插即用驱动程序的用途多种多样，其中内核扩展是最自然的用法。例如，许多系统工具使用内核扩展类型的驱动程序来获得 Windows 内核中的各种系统信息，或者创建系统线程以便在系统进程环境中执行任务。另外，在 Windows 内核中，也有一些模块虽然以“.sys”作为文件扩展名，但它们其实并非设备驱动程序，而是单纯的内核扩展动态链接库，供其他的驱动程序或者内核模块调用。

有关 Windows 中 I/O 管理器、即插即用管理器、电源管理器以及设备驱动程序的进一步介绍，可参考第 6 章。

文件系统/存储管理

在现代操作系统中，文件系统是外部存储设备的标准接口，它为应用程序使用这些设备中的数据提供了统一的抽象，多个应用程序和系统本身可以共享使用这些设备。在

Windows 中，文件系统的接口部分由 I/O 管理器定义和实现，但文件系统的实现部分位于专门的一类驱动程序中。当文件系统接收到 I/O 请求时，它会根据文件系统格式规范，将这些请求转变成更低层的对于外部存储设备的 I/O 请求，通过它们的设备驱动程序来完成原始的 I/O 请求。因此，文件系统的驱动程序定义了外部存储设备中数据的逻辑结构，使得这些数据可直接被操作系统和应用程序使用。

Windows 的原生文件系统是 NTFS（NT File System），其驱动程序为 ntfs.sys。NTFS 是专门为 Windows 设计的文件系统格式，它提供了安全性、可靠性、大容量支持、长文件名支持，以及可恢复性等一系列高级特性，目前广泛应用于 Windows 系统。另一个常用的文件系统格式是 FAT（File Allocation Table），这是从 DOS 时代发展起来的文件系统格式，格式规范相对比较简单，目前仍在使用，主要用于兼容老版本的操作系统，以及用于移动设备以便跨操作系统传送数据。

在 Windows 中，每个文件系统实例有它自己的设备栈，因而通过插入过滤驱动程序可以过滤文件 I/O 请求。Windows 支持两种形式的过滤驱动程序：一种直接插入到设备栈中，从而能够看到每一个经过设备栈的文件 I/O 请求；另一种基于 Windows 提供的过滤管理器驱动程序（FltMgr）的 I/O 过滤框架，称为文件系统小过滤驱动程序，它们并不出现在文件系统设备栈中，而是以回调方式来响应 FltMgr 的事件。

文件系统的底层是对存储设备的管理。大容量存储设备以“分区（partition）”和“卷（volume）”来管理整个存储空间。分区是指存储设备上连续的存储区域（连续的扇区），而卷是指扇区的逻辑集合。一个卷内部的扇区可能来自一个分区，也可能来自多个分区，甚至来自不同的磁盘。文件系统则是卷内部的逻辑结构。因此，Windows 的存储管理形成了一个存储栈，最接近于应用程序的是文件系统，接下来是卷管理部分，最接近于存储设备的是分区管理和磁盘驱动程序。

磁盘设备是典型的即插即用设备，其设备栈和驱动程序符合 WDM 规范。即插即用管理器在设备列举过程中建立起每个磁盘设备的设备栈。设备栈的最底下是总线驱动程序，最上方是一个称为分区管理器的驱动程序，负责通知即插即用管理器当前磁盘上有哪些分区，因而系统中的卷管理器可以接收到有关分区创建和删除的通知。这样，每个物理分区与卷管理器联系起来，卷管理器再将卷与文件系统关联起来，就形成了完整的存储栈。

有关 Windows 中文件系统和存储管理的进一步介绍，可参考第 7 章。

网络

网络虽然并非 Windows 操作系统中必不可少的组成部分，但实际上，它已经成为绝大

多数 Windows 系统的标准配置。Windows 为应用程序提供了多种网络 API，允许应用软件设计人员根据他们的需求适当选择。以下是 Windows 平台上主要的网络 API：

- Windows 套接字，简称 Winsock。它实现并扩展了 BSD 套接字标准。Winsock 2.0 版本支持一些新特性，比如异步网络 I/O、服务质量（QoS）规范、可扩展名字空间，以及多点消息传输等。
- WinInet。这是一个高层网络 API，它支持多个协议，包括 Gopher、FTP 和 HTTP。Microsoft Internet Explorer 使用 WinInet 来完成数据传输。
- 命名管道（named pipe）和邮件槽（mailslot）。用于不同进程之间进行通信。它们支持不同机器上的进程之间相互通信。命名管道支持连接方式的通信模型；邮件槽支持非连接方式的通信模型，客户进程可以发送广播消息。
- NetBIOS。这是一个早期的网络编程 API，Windows 支持 NetBIOS 是为了兼容老的应用程序。NetBIOS 支持有连接的通信和无连接的通信。
- RPC。这是网络编程的一个标准，往往是分布式系统基础设施的重要组件。RPC 建立在其他的网络 API 基础之上，比如命名管道和 Winsock。Windows 的 RPC 支持异步调用方式。

这些网络 API 都提供了用户模式的动态链接库（DLL），当应用程序通过这些 DLL 发出网络 I/O 请求时，它们必须将接收到的请求传递给内核模式下的相应驱动程序。通常，这些网络 API 要么通过专门的系统服务切换到内核模式，比如命名管道和邮件槽就有专门的系统服务；要么通过标准的系统服务接口，比如 NtReadFile、NtWriteFile 和 NtDeviceIoControlFile，由 I/O 管理器和对象管理器将网络请求转送至对应的驱动程序中。

Winsock 是 Windows 最重要的网络 API，它的用户模式部分不仅包含了一个 DLL（即 ws2_32.dll），还定义了一个可扩展的框架，允许第三方插入传输服务提供者和名字空间服务提供者，以支持更多的传输服务和名称解析或地址映射能力。Winsock 默认支持 TCP/IP、IPX/SPX、AppleTalk 和 ATM 等协议，它提供的传输服务和名字空间服务都通过内核模式驱动程序 afd.sys 实现网络通信。

在内核模式部分，网络 API 驱动程序（譬如 afd.sys）通过传输驱动程序接口（TDI，Transport Driver Interface）与协议驱动程序进行通信。TDI 实际上是一组预定义的 I/O 请求，它描述了各种网络请求，包括名称解析、建立连接、发送和接收数据等。网络 API 驱动程序是 TDI 客户，而传输协议驱动程序实现了 TDI 接口，称为 TDI 传输器。TDI 客

户与 TDI 传输器之间是松耦合关系。一个 TDI 传输协议驱动程序可以被多个 TDI 客户使用。例如，TCP/IP 驱动程序为 tcpip.sys，它既可以被 Winsock 驱动程序 afd.sys 使用，也可以被 netbt.sys 驱动程序使用。Windows 不仅实现了基本的 TCP/IP，还支持 NAT（网络地址转译）、IP 过滤以及 IPSec 规范等协议扩展，这些协议扩展也是内核驱动程序，它们通过私有的接口与 tcpip.sys 进行通信。

在 Windows 中，网络协议与网络适配器驱动程序是分开的，协议驱动程序独立于任何一个网络适配器，而真正发送和接收数据是通过网络适配器进行的。协议驱动程序通过统一的接口与适配器驱动程序进行通信，此接口是 NDIS（Network Driver Interface Specification）。符合 NDIS 的网络适配器驱动程序称为 NDIS 驱动程序，或 NDIS 小端口驱动程序（NDIS miniport driver）。Windows 提供了 NDIS 库，即 ndis.sys，作为协议驱动程序与 NDIS 驱动程序两者之间的桥梁。随 Windows XP 和 Windows Server 2003 一起发行的 NDIS 库是 NDIS 5.1；随 Windows Vista 一起发行的 NDIS 库是 NDIS 6。

NDIS 客户（即 TDI 传输器）利用 NDIS 库提供的功能，对将要发送给 NDIS 驱动程序的命令进行格式化，并发送给 NDIS 驱动程序；而 NDIS 驱动程序则利用 NDIS 库，接收请求和送回应答。NDIS 驱动程序并非标准的设备驱动程序，它们通过 NDIS 库与 NDIS 客户进行通信，I/O 管理器并不介入两者之间的通信过程。

有关 Windows 网络体系结构的进一步介绍，可参考 9.1 节。

2.2.3 Windows 子系统

按照 Windows NT 最初的设计，它支持三个环境子系统：OS/2、POSIX 和 Windows（或称为 Win32）。然而，Windows 子系统是必须要运行的，没有它 Windows 系统无法运行，而其他两个子系统则被配置成按需启动。而且，到了 Windows XP 以后，只有 Windows 子系统随 Windows 系统一起发行。这一节将介绍 Windows 子系统的概况，更详细的信息，请参考 9.2 节。

在 Windows 平台上，可执行映像文件的格式为 PE 文件格式[PE-SPEC]（参见上一小节中插入的关于 PE 文件格式的介绍），其头部域 Subsystem 指定了该应用程序将被运行在哪个环境子系统中。Microsoft Visual Studio 的链接器（linker）支持/SUBSYSTEM 命令选项，由它来指定子系统的类型。例如，该域为 2 说明这是一个 Windows GUI 应用程序，为 3 则是 Windows 控制台应用程序（Windows CUI）。

Windows 子系统中既有用户模式部分，也有内核模式部分。内核模式部分的核心是

win32k.sys，虽然它的形式是一个驱动程序，但实际上它并不处理 I/O 请求，相反，它向用户代码提供了大量的系统服务。从功能上讲，它包含两部分：窗口管理和图形设备接口（GDI）。其中窗口管理部分负责收集和分发消息，以及控制窗口显示和管理屏幕输出；图形设备接口部分包含各种形状绘制以及文本输出功能。

用户模式部分包括 Windows 子系统进程（csrss.exe）以及一组动态链接库（DLL）。Csrss.exe 进程主要负责控制台窗口的功能，以及创建或删除进程和线程等。子系统 DLL 则被直接链接到应用程序进程中，包括 kernel32.dll、user32.dll、gdi32.dll 和 advapi.dll 等，负责实现已文档化的 Windows API 函数。除了有些可以直接在用户模式中完成以外，很多 API 函数需要调用执行体 API 或 win32k.sys 模块提供的系统服务。

Win32k.sys 一方面向用户代码提供系统服务，另一方面也跟 Windows 内核紧密地融合在一起。它通过向内核注册一组出调（callout）函数，以便介入到内核的线程和进程管理等处理逻辑中，同时也可以接收电源事件。对于每个线程，一旦它调用了 win32k.sys 的任何一个系统服务，就变成了一个 GUI 线程，从而纳入到了 Windows 子系统的线程和进程管理范畴。Windows 内核的线程和进程数据结构为 Windows 子系统预留了一些域，从而 win32k.sys 可以方便地操纵它的线程和进程。

下面从窗口管理和图形设备接口两方面来介绍 Windows 子系统的核心功能和结构。

窗口管理

Windows 子系统的用户界面管理有一个层次结构，通常应用程序只是在一个默认的桌面上运行。图 2.5 显示了这一层次结构。每个子系统会话都有自己的会话空间，属于某一会话的资源将从该会话空间中分配。当用户登录到 Windows 中时，操作系统将为该用户建立一个会话；即使用户通过远程桌面或者终端服务连接到一个系统中，系统也会为该用户建立一个单独的会话。

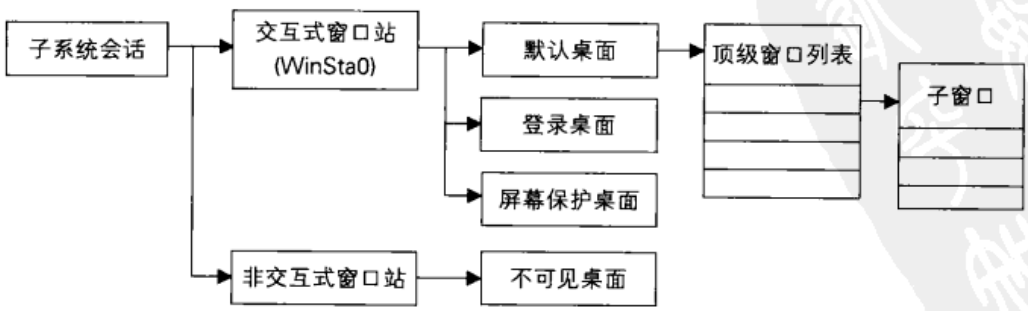


图 2.5 Windows 子系统窗口管理层次结构

在一个会话中，有一个交互式窗口站，可能还有非交互式窗口站。在交互式窗口站中通常有三个桌面：登录桌面、默认桌面和屏幕保护桌面。通常我们运行的图形界面应用程序运行在默认桌面上。交互式窗口站有独立的剪贴板、键盘、鼠标、显示器等，在它的三个桌面中，任一时刻只有一个是激活的，输入输出设备归激活的桌面所有。

在每个桌面中，都有一个顶级窗口列表，这些窗口往往可以相互重叠，有系统菜单、最大化/最小化按钮和滚动条等。通常各个图形界面应用程序的主窗口属于当前桌面的顶级窗口。在 Windows 中，窗口可以有子窗口，子窗口占据父窗口的客户区域。因此，桌面上的窗口形成了一个层次结构。图 2.5 仅仅显示了顶级窗口的子窗口，实际上，一个窗口总是可以构建它自己的子窗口。

Windows 为常用的窗口定义了一些窗口类（window class），因而应用程序可以非常方便地创建这些窗口类的实例。应用程序如果要定义独特的窗口特性，可以生成一个窗口类，这个窗口类既可以是全新的，也可以是在系统已有窗口类的基础上定义得到。窗口类规定了其对象将如何响应各种消息，包括系统发送给它的消息和用户触发的消息。Windows 窗口的编程模型是消息驱动的，每个窗口对象根据其窗口类指定的窗口过程来响应各种消息。

Windows 子系统会话有一个 RIT（Raw Input Thread）线程，负责从输入设备读取原始的输入事件，然后生成消息，寄送到正确的线程消息队列中。每个包含用户界面元素的线程都应该及时地处理这些消息。通常的做法是在一个消息循环中，不停地获取消息，再分发给目标窗口，由目标窗口的窗口过程来响应。有关 Windows 子系统中的窗口管理和消息机制的更多信息，请参考 9.2.3 节。

图形设备接口

Windows 的图形引擎也是在 Windows 子系统中提供的，它有两方面的特点：首先，它提供了一套与设备无关的编程接口，即 GDI，这使得应用程序可以适应各种底层显示设备的差异；其次，应用程序与图形设备驱动程序之间的通信足够高效，从而即使在频繁输出和刷新图形元素的情况下，Windows 也能够为用户提供良好的视觉效果。

在技术上，Windows 子系统定义了一个稳定的图形体系结构，以便于第三方的图形设备硬件厂商可以方便地将他们的视频显示器和打印设备集成到 Windows 中。如图 2.6 所示，win32k.sys 通过 DDI（显示设备驱动程序接口）与显示驱动程序打交道，而显示驱动程序通过 ENG（图形引擎接口）调用 win32k.sys 中图形引擎的功能。

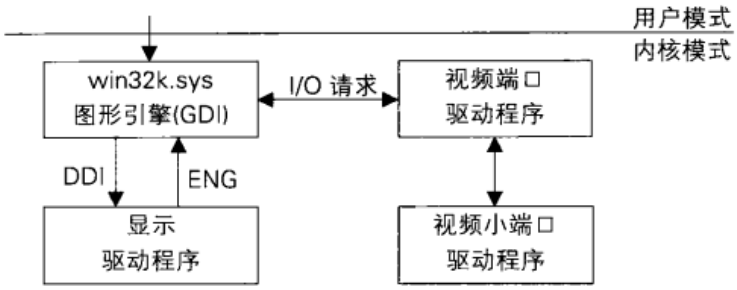


图 2.6 Windows 子系统的图形模块结构

Win32k.sys 的图形引擎实现了基于标准格式位图的图形绘制功能。显示驱动程序在初始化时向图形引擎报告它所支持的物理设备的特征描述。显示驱动程序既可以自己管理图形表面（surface），也可以直接使用图形引擎管理的图形表面。另外，显示驱动程序可以有选择地截取 GDI 的图形绘制操作，也可以将图形绘制操作转交给图形引擎。因此，显示驱动程序在实现功能方面有很大的灵活性。

如图 2.6 所示，显示驱动程序可以帮助图形引擎实现针对特定视频适配器的图形功能，视频小端口驱动程序更是针对视频适配器的硬件特性。两者的分工原则是：显示驱动程序执行一些对用户可见并且性能紧急的图形操作，譬如，它可以直接访问适配器的视频存储区或者寄存器；而视频小端口驱动程序执行一些不常见的图形操作，或者一些不能被中断打断或其他进程抢占的关键图形操作。

视频端口驱动程序由 Windows 操作系统提供，它实际上是一个动态链接库，用于帮助视频小端口驱动程序实现一些公共的、与图形处理有关的功能，以及为小端口驱动程序提供一个与系统内核和执行体打交道的环境。视频小端口驱动程序则负责直接的硬件资源管理和控制。

Windows 子系统的图形系统，除了支持 GDI，还提供了对 DirectX 的显示支持，包括 DirectDraw 和 Direct3D，因而允许像游戏、多媒体播放器等应用软件绕过 GDI 图形引擎，直接操纵显示器硬件，从而获得更快的显示速度，并且避免屏幕抖动。

有关 Windows 子系统图形功能部分的详细信息，请参考 9.2.4 节关于 Windows 显示驱动程序模型的描述，以及 9.2.5 节关于 Windows Vista 以后变化情况的简单介绍。

2.2.4 系统线程和系统进程

在 Windows 中，每个线程代表一个指令执行序列，同时也是一个调度单元；进程定义了一个执行环境，有自己独立的地址空间。每个线程都必定依附于一个进程。Windows 内核除了接受来自应用程序的系统服务调用，它自己也有一些线程用于各种用途。这些线

程运行在一个特殊的进程环境中，称为 System 进程。为区分该进程与本节下文即将介绍的操作系统关键进程，本书将前者称为 System 进程，而将后者称为系统进程。System 进程的线程被称为系统线程，其中有一组系统辅助线程（system worker thread），它们代表操作系统或者其他的应用进程来完成一些特殊的工作。

设备驱动程序或执行体既可以在调用者进程中创建线程，也可以在 System 进程中创建线程。或者，也可以不创建任何一个线程，而是调用执行体函数 ExQueueWorkItem 或 I/O 管理器函数 IoQueueWorkItem，向系统辅助线程请求得到它们的服务。这两个函数负责把一个工作项目（WorkItem）放到一个队列中，而系统辅助线程将从此队列提取工作项目，并执行工作项目中指定的一个例程。因此，工作项目中的例程是在 System 进程的环境中执行的，它不能访问其他进程空间中的数据。

系统辅助线程实际上是一个线程池，Windows 在系统初始化时创建了一定数量的辅助线程，而且，随着辅助线程的负载的变化，执行体也会动态地创建一些辅助线程，以满足系统负载的变化需求。

除了系统辅助线程，内核中的许多组件也会创建系统线程来完成一些必要的工作，例如，内存管理器需要后台系统线程来监视和管理页面的换入和换出。另外，设备驱动程序也可以通过调用 PsCreateSystemThread 函数来创建系统线程，以便完成一些并非依附于调用者进程空间的任务，例如网络驱动程序和文件系统驱动程序可以利用系统线程来完成一些必要的后台处理。注意，通过 PsCreateSystemThread 函数来创建系统线程时也可以指定其他的进程作为系统线程的宿主进程，从而能够访问该进程地址空间中的数据，但默认的宿主进程是 System 进程。在本书后面章节的讲述中，我们会提到一些特定任务的系统线程，甚至可以了解到它们在什么情况下是如何被创建的。

从进程的角度来看，Windows 有一组系统进程对于整个系统的运行来说是必不可少的，包括本章前面介绍过的子系统进程 csrss.exe。当 Windows 操作系统引导起来，并且用户登录到系统中时，Windows 已经经过了一系列初始化过程，除了负责与用户交互的 Shell 进程，还启动了诸多系统进程用于实现各种不同的系统任务。总体而言，以下这些系统进程在 Windows 操作系统中扮演了重要的角色：

- 系统空闲进程（Idle）。该进程的 ID 为 0，其中每个处理器或核对应有一个线程。
- System 进程。在 Windows XP 和 Windows Server 2003 中该进程的 ID 为 4，它包含了内核模式系统线程。正如本节上文所述，系统辅助线程，以及执行体和驱动程序通过 PsCreateSystemThread 创建的线程，都在 System 进程中。

- 会话管理器（Session manager，smss.exe）。这是 Windows 系统中第一个创建的用户模式进程。Smss 在 Windows 启动过程中承担了一些重要的步骤，例如创建环境变量等，尤为重要的是，它启动了子系统进程 csrss.exe 和登录进程 winlogon.exe。另外，会话管理器也负责创建新的终端服务器会话（terminal server session），包括建立会话空间的数据结构，然后为新建的终端服务器会话加载子系统，启动 csrss.exe 和 winlogon.exe。
- 登录进程（winlogon.exe）。它负责处理交互用户的登录和注销。当用户按下 Ctrl+Alt+Del 组合键（称为安全注意序列[SAS，Secure Attention Sequence]）时，winlogon 就会接到登录请求，然后激发安全认证过程，并启动用户会话中的初始进程。在用户会话的任何时候，当用户按下了 SAS 组合键时，winlogon 都会提示一个安全对话框，其中通常包含“注销”、“启动任务管理器”、“锁定计算机”、“更改密码”、“关机”等选项。
- Windows 子系统进程（csrss.exe）。正如本章前文所提到的，Windows 子系统进程负责为用户提供一个子系统环境，包括提供控制台窗口的功能，以及创建或删除进程和线程。
- 本地安全权威子系统进程（lsass.exe）。它负责本地系统安全策略，例如允许哪些用户登录到本地系统中、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；同时也负责认证用户的身份，以及将安全审计消息发送到系统的事件日志（Event Log）中。
- Shell 进程（explorer.exe）。这是 Windows 的默认 Shell，它提供了系统与用户打交道的各种界面，包括开始菜单、任务栏、资源管理器窗口等几乎所有 Windows 用户都熟悉的界面。
- 服务控制管理器（services.exe）。它负责管理 Windows 的系统服务，这里的系统服务是指一些特殊的进程，它们通常并不与登录用户进行交互，因而被配置成可以在系统引导时自动启动起来，无须交互登录过程。Windows 中有很多功能组件是以服务的方式来实现的，比如事件日志、任务调度器和各种网络组件等。

2.3 关于 Windows 研究内核

Windows 并非一个开放源代码的操作系统，但正如上一章所提，Microsoft 开放了一份以 Windows XP x64 和 Windows Server 2003 SP1 为基础的内核源代码，它可以编译和运行，

作为教育科研机构的教学实践和研究的平台使用，称为 WRK（Windows Research Kernel，Windows 研究内核）[WRK]。除了这份源代码本身，WRK 还提供了其他一些材料。这一节将介绍 WRK 所包含的内容、WRK 源代码的说明，以及本书对于 WRK 中代码的引用。

2.3.1 WRK 包含了什么

WRK 的重要目标是辅助教学，让计算机专业的学生能够通过 Windows 内核的源代码来理解和掌握现代操作系统中的基本概念和各种机制。WRK 的可编译源代码于 2006 年 7 月面向全球大学的教职工开放，经过最近三年多的推广以及一些大学采用 WRK 作为操作系统教学和实验平台的经验积累，目前 WRK 已经逐渐形成了源代码、课程参考讲义、上机实验等全方位的教学系统平台。而且，也有一些大学在 Microsoft 的资助下，利用 WRK 从事操作系统的科学研究工作。

无论通过 Internet 在线下载[WRK]，或者申请免费光盘，您都可以获得以下相关内容：

- WRK 内核源代码，涉及进程、线程、内存管理、执行体、对象管理器、缓存管理器、本地过程调用（LPC）、注册表、I/O 管理器、安全引用监视器，以及线程调度、APC（异步过程调用）/DPC（延迟的过程调用）、中断以及异常处理等。随源代码一起提供的还有相应的编译工具，因此，无须额外的编译器即可将 WRK 编译成 Windows Server 2003 SP1 的可执行内核。
- NT 设计文档。这是一组早期的文档，尽管其内容已不完全适用于现在的 Windows 操作系统以及 WRK 中的代码，但是，通过阅读这些文档一方面可以清楚地理解 Windows NT 背后的原始设计思想，另一方面也可以看出 Windows 在这十多年中是如何发展和进化的。这些文档涵盖了 Windows 操作系统的方方面面，甚至包括文件系统设计大纲和内核的调试结构等。
- 软件 Virtual PC 2007，以及 Windows Server 2003 SP1 的虚拟机映像，此虚拟机系统已配置好 WRK 内核。利用此 Virtual PC 2007 和 WRK 系统映像，您可以方便地调试和跟踪 WRK 中的代码。
- 课程资源 CRK（Curriculum Resource Kit），包括一整套 Windows 操作系统讲义，共 15 个单元。每个单元又包含一些专题讲义、习题和上机练习题。对于以 Windows 为教学和实验平台的课程，这是一份非常有帮助的课件。CRK 中也包含了《Windows Internals》（第 4 版）的电子书，以及一组工具（windbg、kernrate 等）。
- ProjectOZ 源代码。ProjectOZ 是一个利用 Windows 内核的 NTAPI 建立起来的 CPU、

MMU 和陷阱机制的操作系统实验环境，其核心是 CPU、MMU 和陷阱机制的 SPACE 抽象[SPACE]。由于此实验平台的下面是一个真实的 Windows 操作系统在处理和操纵硬件，而不是一个模拟器，所以，学生们更有机会学习和感受操作系统算法和数据结构的复杂性。

- 相关的辅助材料和参考资料，包括 WAP（Windows Academic Program，Windows 学院计划）中的一些教学实践项目、《Windows Internals》两位作者的共 12 小时的视频材料，以及 Singularity 项目[SINGULARITY]的一些文章和讲义。另外，还有一份介绍 Windows Vista 内核新特性的演讲稿。

然而，Microsoft 目前并非对所有人开放以上材料，而是以教学和科研为开放目标，仅限于大学的教职员工使用。请在使用 WRK 资料以前，首先阅读 WRK 许可条例，参见 *http://www.microsoft.com/resources/sharedsource/licensing/basics/wrklicense.mspx*。

2.3.2 WRK 源代码说明

正如本章前面所讲，Windows 的内核模块文件是 ntoskrnl.exe，位于 Windows\System32 目录下，它包含了 Windows 体系结构中的执行体和内核（或微内核）部分。WRK 提供的源代码可以编译得到这一内核模块文件，在 WRK 编译环境下针对 Intel x86 处理器的默认生成文件名为 wrkx86.exe。Windows 的引导选项/kernel 可以指定不同于 ntoskrnl.exe 的内核模块文件。参见本书附录 A 关于编译、配置和调试 WRK 的详细介绍。表 2.2 列出了 WRK 源代码的目录结构及其对应的内核组件。

WRK 包含了编译 ntoskrnl.exe 内核模块所需要的绝大部分代码，未公开部分的代码主要包括即插即用设备管理、电源管理、设备驱动程序检验器和虚拟 DOS 机的实现。为了编译 WRK 源代码以得到实际可运行的内核模块，缺失的这部分被以二进制目标代码的形式包含在了 WRK 中，位于 base\ntos\BUILD\PREBUILT\i386（或 base\ntos\BUILD\PREBUILT\amd64）目录。该目录还包含了其他一些需要静态链接的目标文件。尽管如此，WRK 对于学习和理解 Windows 的工作机理仍然是一份极佳的资源。

WRK 的代码是从当时最新的 Windows 产品代码中摘出来的，可以编译并运行于 Windows Server 2003 SP1（x86 处理器版本）和 Windows XP SP2（AMD64 版本）系统中，其内核版本为 5.2。WRK 代码与产品代码几乎一致，主要的变化在于去掉了对服务器的支持，比如与 Intel IA64 有关的代码。

Windows 源代码一致性较好，非常易读。代码本身的逻辑以及各标识符基本上是自

解释性的，重要函数的头部都有详细的注释说明，重要的代码片段也有专门的注释。总体上，代码组织较为清晰，如表 2.2 所示。内核模块内部的每个组件都提供了一些接口函数供其他组件调用，也有一些函数供该组件内部使用。表 2.3 列出了一些常用的标识性前缀。有一些组件内部函数也有规律可循：前缀第一个字母后面跟一个 i，或者在前缀后面跟一个 p，这里 i 代表 internal，即内部的；p 代表 private，即私有的。例如，Ki 和 Mi 分别代表微内核和内存管理器的内部函数，而 Halp、Psp、Iop 分别代表 HAL、进程和线程管理组件、I/O 管理器的内部函数。掌握这些命名规律，有助于快速地理解一个函数的归属。

表 2.2 WRK 目录结构和相应的组件说明

目录	组件说明
public	各个组件共享使用的包含文件，包括 DDK、SDK、HAL 和内核自身的一些头文件
base\ntos	Windows 内核模块的主目录
base\ntos\cache	缓存管理器的源文件
base\ntos\config	注册表的实现代码
base\ntos\dbgk	调试子系统的内核模式部分
base\ntos\ex	执行体函数（内核堆、同步、定时器等）
base\ntos\fsrtl	文件系统运行库
base\ntos\fstub	文件系统引导接口
base\ntos\io	I/O 管理器，不包括即插即用管理器和电源管理器部分
base\ntos\ke	（微）内核，包括线程调度器、CPU 管理，以及低层的同步语义
base\ntos\lpc	本地过程调用（LPC）机制的实现
base\ntos\mm	内存管理器
base\ntos\ob	内核对象管理器
base\ntos\perf	内核的性能日志记录功能
base\ntos\ps	进程和线程
base\ntos\se	安全引用监视器
base\ntos\wmi	Windows 管理规范（WMI，Windows Management Instrumentation）
base\ntos\inc	仅适用于 NTOS 部分的包含文件
base\ntos\raw	RAW 文件系统驱动程序的实现代码
base\ntos\rtl	内核运行时库支持
base\ntos\init	内核启动部分的代码
base\ntos\VDM	虚拟 DOS 机（没有公开代码）
base\ntos\VERIFIER	驱动程序检验器（只有头文件）

表 2.3 WRK 源代码中各组件接口函数的前缀

函数前缀	所属的组件或函数说明
Cc	缓存管理器
Cm	配置管理器（即注册表）
Dbg/Kd	调试支持函数
Ex	执行体函数
FsRtl	文件系统驱动程序运行库函数
Fstub	文件系统引导接口函数
Hal	HAL 提供的接口函数
Io	I/O 管理器
Ke	内核函数
Lpc	本地过程调用（LPC）函数
Mm	内存管理器
Nt	Windows 系统服务
Ob	对象管理器
Perf	日志记录函数
Po	电源管理器
Pp	即插即用管理器
Ps	进程/线程
Raw	RAW 文件系统的函数
Rtl	内核运行库函数
Se	安全函数
Vf	驱动程序检验器函数
Wmi	Windows 管理规范
Zw	与 Nt 前缀同名的一套函数，省去了参数验证的步骤，其他逻辑相同。可以认为以 Nt 前缀为名称的函数针对用户模式的调用者，以 Zw 前缀为名称的函数针对内核模式调用者

2.3.3 本书对 WRK 源代码的引用

本书后面的章节将介绍 Windows 内核中的重要组件，包括进程和线程的管理、内存管理器、同步和并发机制、I/O 管理器，以及存储管理和系统服务等。在讲解这些内容时，将以 WRK 的源代码为主要参照，在必要的地方会列出相应的数据结构或者函数原型，以说明 Windows 中一些关键机制的实现原理。除了 C 语言的代码，有些底层逻辑也可能以 Intel x86 汇编指令的形式列示出来。

在后面的章节引用代码时，为使代码清晰、可读，所摘录的代码会进行简单整理，仅显示原始代码中对应于 Intel x86 编译条件下的部分，注释部分也会做相应处理，目的是使读者能更好地理解相应的逻辑。WRK 的许可条例规定，对代码片段的引用每次不能超过 50 行，但 Windows 有些重要数据结构，例如线程和进程数据结构，超过了 50 行。为此，本书第 3 章采取的做法是，将大的数据结构分成几部分，然后逐部分介绍其成员。

然而，为了保持本书的篇幅不至于过于庞大，笔者将尽可能地避免列出代码，而通过解释关键的逻辑，或者勾画出函数内部或函数之间的控制流图，以说明一些关键机制的实现过程。在讲解一个重要函数的实现时，针对有些关键的步骤，本书可能会指出所对应的代码行。因此，如果读者对照 WRK 来理解这样的函数，建议使用一个可显示行号的编辑器来辅助阅读，这样有助于利用本书帮助阅读源代码。

阅读源代码是掌握实现细节的重要途径，但由于 Windows 内核极其复杂，简单地阅读源代码很容易迷失思路。笔者的观点是，代码本身可能没有那么重要，重要的是 Windows 系统是如何实现一些关键机制的，以及一些关键的控制函数是什么，它们做了哪些重要的操作。这在某种程度上比洞悉每一个实现细节更为重要。本书的一个目标是，帮助读者理清 Windows 各种底层机制的思路和实现机理，而不是简单地把有关的实现细节摸索并展示一遍。根据笔者的经验，建立一个 WRK 调试环境将非常有帮助，这可以让某些控制流更加容易理解。本书附录 A 介绍了 WRK 调试环境的配置办法。

正因为如此，即使读者不能访问 WRK 的代码，也仍然可以阅读本书，譬如，从本书的描述中可以知道 Windows 是如何实现各种系统机制的。配合本书中提到的关键函数的名称以及一些重要步骤的实现逻辑，您可以查阅这些函数的文档，甚至在调试器中跟踪和检查这些函数的实现细节。另一方面，正如本书前言中所讲，即使读者的实验平台并非 Windows Server 2003 SP1 版本，本书的绝大部分内容仍然适用于自 Windows 2000 以来，直至 Windows 7 的系统，因此，抓住 Windows 内核的实现原理比剖析一份源代码更有意义。WRK 仅仅是 Windows 内核实现的一份参照。

2.4 Windows 内核的基本概念

Windows 内核中的各个组件并非单纯的独立模块，相反地，组件之间不可避免地包含着复杂的依赖关系，甚至存在交叉调用。这对于讲述和学习这些组件的原理多少造成了不便。例如，在介绍线程调度算法时，需要理解基本的同步语义；而在介绍同步机制时，又需要明白线程是如何被调度的。为了使读者在阅读后面的章节时思路更加顺畅，这一节将

简要地介绍 Windows 内核中的一些基本概念，从而使读者对于 Windows 内核中的核心机制首先有概念上的初步认识。在本节的正文描述中，这些概念将以粗黑字体的形式突出标示，并尽可能地注明中英文，以便于理解。

2.4.1 处理器模式

我们先来看处理器在运行程序时所处的状态。在 Intel x86 处理器上，段描述符有一个 2 位长度的特权级：0 表示最高特权级，3 表示最低特权级。Windows 只使用 0 和 3 两种特权级(在有些资料上分别称为 0 环和 3 环),通常特权级 0 表示 CPU 处于**内核模式**(kernel mode)，3 表示**用户模式**（user mdoe）。在任何时刻，处理器总是位于这两种模式之一。在本章前面介绍 Windows 体系结构时我们已经看到过内核模式和用户模式的划分。

处理器有许多指令只有在特权级 0 的模式下才可以使用，例如 I/O 指令、操纵内部寄存器（如 GDT、IDT、MSR）的指令等。当处理器位于用户模式时，它处于一种相对隔离的状态：能够执行的指令是受限制的，能够访问的内存也是受限制的。一旦越过这些限制，就会引发处理器异常，因而操作系统可以捕获到这些异常，并决定处理器是否继续执行。因此，操作系统可以有足够的能力来保护自己免受用户模式代码的影响。然而，当处理器位于内核模式时，这一层保护不复存在，任何一个未被捕捉和处理的指令错误都会引起系统崩溃。

在 Windows 中，当处理器位于不同模式下时，它可以访问的内存地址空间也是不一样的。在用户模式下，处理器只能访问当前进程的地址空间（有时也称为用户地址空间）；而在内核模式下，处理器不仅可以访问当前进程的地址空间，还可以访问系统地址空间。因此，正如图 2.2 所示，内核模式下的代码和数据都是共享的，所有的进程一旦其指令流进入到内核模式下，则系统地址空间中的代码和数据都是相同的（有个别例外将在第 4 章中指出）。

一个指令流（即线程）在执行时，在以下情况下会发生模式切换：用户模式代码触发了异常，则控制流进入到内核模式，内核中的异常处理函数可以决定该控制流是否继续执行；用户模式代码在执行时，被一个中断打断（软中断或硬中断），则控制流进入特权模式，等中断处理例程完成以后，它若调用 iret/iretd 指令，则控制流恢复到用户模式下；执行特殊的模式切换指令，例如 Intel x86 的 sysenter 指令，从用户模式切换到内核模式。而为了从内核模式切换到用户模式，通常简单地使用 sysexit、iret/iretd 这样的指令即可。

由于系统空间是所有进程共享的，所以，任何一个进程在执行内核模式的代码时，实际上是在使用操作系统的服务。在 Windows 体系结构中，内核模式向上有一个执行体 API（见图 2.3），尽管它并非文档化的 API，但对于应用程序而言，这便是系统服务。Windows

将这些系统服务组织成了一张表，称为 SDT（Service Descriptor Table，**服务描述符表**）。第 8 章将介绍用户模式与内核模式之间的切换原理，以及系统服务分发过程。

2.4.2 内存管理

Intel x86 体系结构既支持段式内存管理，也支持页式内存管理，然而，Windows 没有使用段式内存管理方案，而只是简单地将 32 位虚拟内存空间按照 0~4 GB 的线性地址空间来看待。任何一个进程都定义了它自己的完整 4 GB 地址空间，但是，其 2 GB~4 GB 之间的部分是所有进程共享的，称为**系统地址空间**；而 0~2 GB 的部分才是它自己私有的，称为**进程地址空间**。对于某些特殊的应用程序，例如数据库程序，它们总是希望其私有空间越大越好。在这种情况下，利用 Windows 的“/3GB”引导选项，这样的进程可以获得 3 GB 进程地址空间，而系统地址空间只占用 1 GB。本书不讨论这样的情形。

为了有效地管理 2 GB 系统地址空间，Windows 在初始化时将这 2 GB 划分成了一些固定的区域，各个区域有专门的用途。而且，Windows 内核使用了一组全局变量来记录每个区域的边界，所以，系统地址空间的初始化实际上是对这些全局变量的初始化，并相应地初始化每个区域。Windows 的一些引导选项以及系统配置（位于注册表）可能会影响到某些区域的位置和大小。系统空间中的主要区域包括：内核模块映像、PFN 数据库、换页内存池、非换页内存池、会话空间、系统缓存区、系统 PTE 区域、系统视图以及页表等。

Windows 使用 Intel x86 的二级或多级页表机制来访问虚拟内存。处理器在执行内存访问指令时，将虚拟地址翻译成物理地址，翻译过程涉及查询页目录和页表，一旦页表项指示一个页面未在物理内存中，则触发**页面错误**（page fault）异常。虚拟内存管理器通过页面错误异常将已被换出到磁盘上的数据或代码重新带入物理内存，供当前活跃的程序访问。另一方面，当物理内存紧张时，它会将不常用的页面换出到磁盘上。Windows 利用 Intel x86 的页表机制，实现了灵活的页面交换算法，可以支持一个或多个页面交换文件；同时也实现了内存页面的**写时复制**（copy-on-write）特性。

在系统地址空间中，不同的区域使用并不完全相同的内存页面管理算法，较为典型的有以下三种：

- 非换页内存池。这部分内存区域在初始化时已经被映射到物理页面，所以，Windows 利用空闲链表的做法，按照不同的粒度（1、2、3、≥4 个页面大小），将空闲页面链接起来。空闲页面本身即链表中的节点，因而这些链表无须额外的内存空间（除了头节点）。申请和释放页面的操作实际上是针对空闲链表来进行的。

- 换页内存池。在换页内存池区域，空闲的页面并没有被映射好物理页面，Windows 使用位图来管理页面的分配。分配连续的多个页面，即从位图中找到连续的零位。
- 系统 PTE 区域。这部分内存区域存放的并非 PTE，而只是表示这部分地址范围是以 PTE 的形式来管理的，即把 PTE 当做资源来管理。当内核需要一段虚拟地址来映射物理页面时，它可以使用系统 PTE 区域中的地址。

以上这些内存区域按照页面粒度来管理其分配情况，Windows 执行体在这些系统内存区域管理的基础上，还提供了一组更小粒度（8 B 的倍数，最小为 8 B）的内存池管理，包括**执行体换页内存池**和**执行体非换页内存池**。这些内存池通过空闲链表记录下每个已申请页面中的空闲内存块；当释放内存时，自动与相邻的空闲内存块合并以构成更大的空闲内存块。内核其他组件或驱动程序通过执行体暴露的 API 函数（例如 ExAllocatePoolWithTag 和 ExFreePoolWithTag）来使用这些内存池。

进程地址空间是随进程一起被创建的，每个进程有它自己的页目录页面，其中有一半页目录项（PDE）是共享的，即系统地址空间部分，余下一半初始化为零。随着进程中的映像文件（包括.exe 文件和各个 DLL 文件）被加载进来，以及各个模块的初始化代码被执行，进程地址空间将被建立起来。

进程地址空间按照其虚拟地址是否被分配或保留来进行管理，用户模式代码通过 Windows API 函数 VirtualAlloc(Ex)和 VirtualFree(Ex)来申请或释放地址范围，而内核中的虚拟内存管理器则通过一棵平衡二叉搜索树来管理进程地址空间被使用的情况。树中的每个节点为 VAD（**虚拟地址描述符**，Virtual Address Descriptor），描述了一段连续的地址范围；但是在整个地址空间中，被使用或保留的地址范围往往是不连续的。之所以选择平衡二叉搜索树，是因为这样可使查找、插入和删除 VAD 节点都可以快速地完成。

在 VAD 树中，有一种重要的节点类型为**内存区对象**（section object），它是 Windows 平台上两个或多个进程之间共享内存的一种常用方式。内存区对象可以被映射到系统的页面文件、可执行映像文件或者其他的数据文件中，也可能被映射到物理内存中，总之，内存区对象代表了一种物理存储资源。

除了对系统地址空间和进程地址空间的管理，内存管理器另一个重要的任务是管理有限的物理内存。在 Windows 的系统地址空间中，专门保留了一个称为 **PFN 数据库**（Page Frame Number Database，**页帧编号数据库**）的区域。每一个物理页面都对应于 PFN 数据库中的一项，此 PFN 项描述了该页面的状态。Windows 支持八种状态：活动、备用（standby）、已修改、已修改但不写出、转移、空闲、零化和坏状态。这里活动状态是

指正在被某个进程或系统空间所使用，有一个对应的 PTE 指向该页面。坏状态说明一个物理页面已被检测到硬件错误。其他的状态说明了一个物理页面已经不再被原来的 PTE 所指，但是其内容可能还有效，或者需要被写到磁盘上，或者内容已经不再需要了，或者页面内容已被零化等多种可能。内存管理器利用 PFN 数据库，按照页面的状态来管理物理内存，并且负责页面的状态转移。

那么，当系统中的进程需要使用大量内存时，内存管理器如何将有限的物理页面分配给那些需要使用内存的进程呢？这是 Windows 工作集管理器要解决的任务。这里**工作集**（working set）是指一个进程当前正在使用的物理页面的集合。Windows 系统中除了进程工作集，还有系统工作集（即系统空间中动态映射的页面集合）和会话工作集（即会话空间中的代码和数据区）。

每个进程都有一个工作集链表，其中的每一项不仅记录了物理页面的编号，还记录了其他的属性，包括它的年龄（2 位），因此，工作集管理器可以根据一些策略来选择要修剪的进程，然后，针对被选中要修剪的进程，选择哪些页面被换出到磁盘中，从而将物理页面腾出来。工作集管理器运行在一个称为**平衡集管理器**（balance set manager）的线程中，每隔 1 s 被触发一次，当可用内存太低时也会被触发。平衡集管理器除了触发工作集管理器以外，也定期触发**进程/栈交换器**（process/stack swapper）。进程/栈交换器是另一个单独的线程，一旦被唤醒，就会将满足特定条件的进程和栈换入内存或换出内存。

Windows 的内存管理综合运用了多种数据结构和算法，较为复杂。关于以上每一个概念的详细描述，请参考第 4 章中的相关章节。

2.4.3 进程和线程管理

前面已经多次提到进程和线程，这一节将主要介绍与它们相关的一些概念。**进程**（process）定义了一个执行环境，包括它自己私有的地址空间、一个句柄表，以及一个安全环境；**线程**（thread）则是一个控制流，有自己的**调用栈**（call stack），记录了它的执行历史。每个进程都包含一个或多个线程，当进程被初始创建时系统为该进程创建第一个线程；当最后一个线程结束时，进程也随之结束。Windows 是一个多进程操作系统环境，但它对处理器资源的分配是按照线程来进行调度的。图 2.7 显示了 Windows 中进程和线程的关系。进程中的线程既可能在用户模式下执行，也可能在内核模式下执行。如果一个进程运行在用户模式下，那么它只能访问进程地址空间；如果运行在内核模式下，那么它将可以访问整个地址空间。

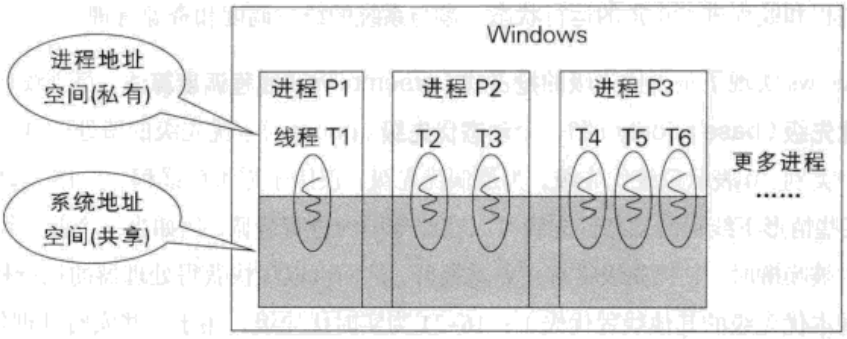


图 2.7 Windows 中的进程和线程

在 Windows 中，进程有父子关系，进程的许多属性在不明确指定的情况下，均继承自父进程。然而，与 UNIX 系统不同的是，Windows 中进程父子关系是一个相对松散的概念，仅仅代表了创建者和被创建者的关系。进程本身是独立的，它仅仅记录了父进程的 ID 而已。

在 Windows 内核结构中，进程和线程的核心机制是在微内核中实现的，而管理机制是在执行体中实现的。这正符合机制与策略分工的原则，例如，线程调度是由微内核来完成，而线程和进程的创建、各种管理属性的设置则由执行体来完成。因此，我们在检查内核函数和阅读 Windows 源代码时，会看到有些关于进程和线程的数据结构和函数属于微内核的范畴，而另外一些则属于执行体的范畴。比较典型的例子是，KPROCESS 和 KTHREAD 是微内核中进程和线程的数据结构，而 EPROCESS 和 ETHREAD 是执行体中进程和线程的数据结构；KeAttachProcess/KeStackAttachProcess 是微内核中将线程附载到指定进程的函数，而 PspCreateThread/PspCreateProcess 是执行体中创建线程和进程的函数。

进程的创建是在内核函数 PspCreateProcess 中完成的，它先创建一个执行体进程对象（EPROCESS），然后初始化该对象的状态，包括建立一个地址空间。类似地，线程的创建是在内核函数 PspCreateThread 中完成的，它先创建一个执行体线程对象（ETHREAD），然后初始化其中的域，并维护好新线程与宿主进程之间的关系。而 Windows 子系统进程的创建过程还要复杂一些，有一部分工作是在子系统 DLL 中完成的，例如，首先要打开指定的可执行文件，创建一个内存区对象，然后才会调用到内核中的 PspCreateProcess 函数。等到进程对象和线程对象建立起来以后，Windows 子系统进程（csrss.exe）还会介入创建过程，以维护子系统内部的进程、线程状态。而且，进程的最后阶段的初始化工作是由初始线程在该进程环境中执行的，所以，当初始线程获得第一次执行权时，总是从某个预定的函数开始执行，然后在这个函数中从内核模式切换到用户模式下，执行基本的进程初始化任务，包括加载必要的动态链接库，并执行每个库的初始化函数。完成了所有的初始化工作以后，最终控制权进入用户指定的线程启动函数或可执行映像文件的入口函数。

至此，进程和线程进入正常的运行状态，参与系统的统一调度和资源管理。

Windows 实现了基于优先级的**抢占式**（preemptive）**线程调度算法**，每个线程都有一个**基本优先级**（base priority）和一个**动态优先级**（priority）。优先级的值处于 0~31 之间，共分三个类别：0 表示系统优先级，为最低优先级，仅用于零页面线程；1~15 为动态优先级，在某些情形下线程的动态优先级可以在此范围内进行微调，例如当一个前台线程从等待状态中被唤醒时，其优先级将有一点点提升，从而可以尽快获得处理器的执行权（至少比相同基本优先级的其他线程优先）；16~31 为实时优先级，用于一些实时处理任务。

每个线程从初始化开始一直到最后终止，其状态会随系统的状态以及它自身的代码逻辑而发生变化。Windows 调度算法中的线程状态转移图比传统的抢占式调度算法略微细致一些。本质上每个线程都处于两种状态之一：满足继续执行的条件，正在排队或者已经在执行；不满足继续执行的条件，处于等待状态，或者它的调用栈甚至所处的进程已被换出内存。在前一种情况下，线程按优先级排队执行；对于多处理器系统，排队过程要复杂一些，调度器不仅要处理多个队列，而且要考虑每个处理器的就绪线程队列的平衡程度。由于各个线程可能存在处理器**亲和性**（affinity），所以，此排队和分发过程要略微复杂一些。而对于不满足执行条件的情形，长时间处于等待状态的线程，其调用栈（甚至整个进程）可能被换出内存，在这种情况下，一旦其执行条件已满足，则首先要被换回内存，然后才能参与排队分发。

还有两个与进程和线程相关的概念也值得提一下：作业和纤程。**作业**（job）是一个执行体支持的内核对象，它使得一个或多个进程被当做一个整体来加以管理和控制。管理程序通过 Windows API 可以控制一个作业消耗系统资源（CPU 或内存）的各种限制，例如用户模式 CPU 时间的限制、进程的处理器亲和性、工作集的最大值和最小值、虚拟内存的使用限制等。**纤程**（fiber）是一种用户线程，它对于内核是不可见的，由 kernel32.dll 实现。应用程序可以在一个线程环境中创建多个纤程，然后手动控制它们的执行。纤程不会被自动执行，应用程序必须显式地选择某个纤程来执行，而且，一旦纤程运行起来，要么一直运行到它退出，要么运行到它显式地切换至另一个纤程。本书不讨论作业和纤程。

有关进程和线程的更多信息，包括它们的数据结构、创建过程以及线程调度算法，请参考第 3 章的讲解。

2.4.4 中断和异常

中断（interrupt）是处理器与外部设备打交道的重要途径；而**异常**（exception）则是处理器的正常指令流在执行过程中产生的一些特殊事件，需要紧急处理才能继续原来的指

令流。它们都会打断一个正常的指令流，但区别在于，中断的发生与当前指令流并无实质联系，而异常则是当前指令流执行的直接结果。而且，中断是异步的，而异常是同步的。

尽管中断和异常的触发来源和方式不相同，Intel x86 处理器内部使用同一套陷阱机制来处理中断和异常，它利用 IDT（Interrupt Descriptor Table，**中断描述符表**），将每个中断或异常与一个处理该中断或异常的服务例程联系起来，因而一旦中断或异常发生，该服务例程即被执行。Windows 在此硬件机制的基础上，又提供了一种更为灵活的软件机制，允许设备驱动程序为特定的中断向量添加它的**中断服务例程**（ISR，Interrupt Service Routine）。一个中断向量允许连接多个**中断对象**（interrupt object），这里中断对象是一种封装了中断服务例程的内核对象。当中断发生时，这些中断对象中的服务例程都有机会处理该中断。通过中断对象机制，设备驱动程序可以在不操纵 IDT 的情况下加入它们的中断服务例程；另一方面，多个硬件设备也可以共享同样的硬件中断向量。

中断控制器（如 APIC）允许设定每一个硬件中断的优先级，但 Windows 并没有使用中断控制器的优先级，而是规定了一套软件中断优先级，称为**中断请求级别**（IRQL，Interrupt Request Level）。在 Intel x86 系统中，Windows 使用 0~31 来表示 IRQL，数值越大，优先级越高。处理器在运行时总是有一个当前 IRQL，如果发生中断时，中断源的 IRQL 等于或者低于当前级别，则该中断被屏蔽，直到处理器的 IRQL 降下来为止。IRQL=0 表示普通线程，称为 PASSIVE_LEVEL 或被动级别，它的优先级最低，可被任何其他级别的中断打断；IRQL=1 表示**异步过程调用**（APC，Asynchronous Procedure Call），称为 APC_LEVEL，它仅仅比 PASSIVE_LEVEL 高，因此，在一个线程中插入一个 APC 对象可以打断该线程的执行；IRQL=2 表示处理器正在做以下两件事情之一：正在进行线程调度，比如选择新的线程；正在处理一个硬件中断的后半部分（不那么紧急的部分），在 Windows 中，这被称为**延迟过程调用**（DPC，Deferred Procedure Call）。因此 IRQL 为 2 也被称为 DISPATCH/DPC 级别，或者简单地称为 DISPATCH_LEVEL。3~26 是设备 IRQL，27~31 是一些特殊的硬件中断，包括时钟中断、电源中断、处理器间中断等，它们都是硬件中断。

DPC 是一个重要的概念，它的 IRQL 等于 DISPATCH_LEVEL，高于 PASSIVE_LEVEL 和 APC_LEVEL，因此它优先于任何一个与线程相关的函数，也屏蔽了线程调度；同时又低于所有的硬件中断，所以它不会屏蔽任何一个硬件中断。之所以称为“延迟的”过程调用，是因为它往往被用来执行一些相对于当前高优先级的任务来说不那么紧急的事情，例如，硬件中断服务例程可以把一些相对不紧急的事情放到一个 DPC 对象中处理，从而缩短处理器停留在高 IRQL 的时间。在 Windows 内核中，DPC 的一个典型用法是**定时器**（timer）的实现，在时钟中断服务例程中，它负责更新中断时间、系统时间，以及当前

线程的时间信息等，并判断系统的定时器数组中是否有定时器到期，若有则发出 DISPATCH_LEVEL 的软中断请求。定时器的到期是在 DPC 交付（deliver）过程中处理的，定时器被当做一种特殊的 DPC 对象而交付执行。

与 DPC 不同的是，APC 是线程相关的例程，它们只能在特定的线程环境中被执行，因而也一定在特定的地址空间中被执行。由于 APC 的 IRQL 高于被动级别，所以 APC 优先于线程本身的指令流。当一个线程获得执行权时，它的 APC 例程会立刻被执行。这一特性使得 APC 非常适合于实现各种异步通知事件，例如，I/O 的完成通知可以用 APC 来实现。

异常是程序指令流执行过程中的同步处理过程，既可以由处理器硬件产生，也可以由指令流软件产生。在 Intel x86 处理器中，异常也是通过 IDT 表分发的。在 IDT 表中，0~0x1f 之间的中断向量是 Intel 保留的，除了 2 号中断向量保留给 NMI（nonmaskable interrupt，不可屏蔽中断）以外，其他已经定义的中断向量都是针对各种条件所引发的异常的。Windows 为所有需要处理的异常都提供了**异常处理器**（exception handler，即异常处理例程），有些异常直接由系统内核来处理，比如页面错误（0x0e 号异常）由虚拟内存管理器接管；有些异常则需要交给当前线程或 Windows 子系统的代码（内核模式或者用户模式）来处理。一旦异常发生的条件消失，则异常处理器返回，原来的指令流继续执行。

内核模式和用户模式的代码都有可能发生异常，根据发生异常时处理器模式的不同，Windows 的**异常分发**（exception dispatch）过程也有所不同。内核模式异常的处理相对简单一些，异常分发器首先将异常交给内核调试器来处理，若不存在内核调试器或者内核调试器没有处理该异常，则尝试分发到一个**基于帧的异常处理器**（frame-based exception handler）。基于帧的异常处理器是一种异常处理技术，它将异常处理器与“栈帧（stack frame）”关联了起来，因而当发生异常时，异常分发器将根据当前栈中的栈帧来查找与之关联的异常处理器。如果未能找到这样的异常处理器，则异常分发器会将该异常再次交给内核调试器，若这一次该异常仍未能被处理，则认为这是一个严重错误，系统崩溃。

用户模式的异常分发过程要复杂一些。异常分发器首先判断进程的调试端口是否有效，若有效，则发送消息至调试端口，然后等待应答，否则将异常交给内核调试器。如果异常仍未得到处理，则将控制转到用户模式下，由用户模式的异常分发器（ntdll.dll 中的 KeUserExceptionDispatcher 函数）寻找一个基于帧的异常处理器。如果在用户模式下该异常仍未得到处理，则控制再次回到内核模式下。这一次，内核模式的异常处理器首先尝试调试端口，若异常仍未被处理，则再尝试当前进程的异常端口。连接进程异常端口的是 Windows 子系统，因此，Windows 子系统在异常处理的最后时刻有机会处理它所属进程的异常。如果它也不能处理此异常，则该进程被终止。

中断和异常是操作系统的底层核心机制，它们既涉及处理器的硬件特性，也跟操作系统的并发性管理和代码的错误处理有关。有关 Windows 中断和异常的更详细信息，请参考 5.2 节。

2.4.5 同步

在现代操作系统中，由于多处理器、多核或者中断等各种**并发性**（concurrency）因素的存在，同样的代码有可能被并发执行，而数据也可能被并发访问。在这些情况下，对于可能被并发访问的数据进行必要的**同步**（synchronization）保护是一种常见的编程实践。Windows 操作系统提供了多种同步机制，使得系统代码或者应用程序代码能够选择恰当的手段来保护其代码和数据。根据执行环境中的 IRQL 值大于 APC_LEVEL 或者等于 PASSIVE_LEVEL，可以将同步机制分为“不依赖于线程调度的同步机制”和“基于线程调度的同步机制”两大类。

当 IRQL 大于 APC_LEVEL 时，Windows 提供了一些方便的同步保护机制，供内核自身或设备驱动程序使用。IRQL 大于 APC_LEVEL 时的典型同步机制如下：

- **提升 IRQL**。根据 IRQL 的定义，当处理器在某个 IRQL 上运行时，它只能被更高 IRQL 的中断打断。这意味着，它不用担心低 IRQL 的过程会抢占掉当前的执行过程。这种做法在单处理器系统上可以工作得很好。但是，在多处理器系统上，仅仅提升 IRQL 还不够，往往还需要结合其他的同步机制（如自旋锁）。
- **互锁操作**。利用 Intel x86 处理器提供的 lock 指令前缀，可以实现基本的对整数操作的保护，确保一个操作以原子方式进行。注意，这种方法只能在小粒度的数据上（可以达到 64 位内存单元）进行同步保护，而且是指令级的保护。
- **无锁的单链表**。链表是 Windows 内核中被广泛使用的数据结构，Windows 利用 64 位互锁指令来实现无锁的单链表数据结构，并且可支持多核、多处理器环境。
- **自旋锁**（spin lock）。自旋锁本质上是一种**忙等待**（busy-wait）。为了获得自旋锁，处理器不停地检测锁的状态，而不做其他的事情，直至锁的状态变成可用为止。在这种情况下，即使有线程正在等着执行，或者有 APC 正在排队，它们也没有机会被执行，所以，自旋锁通常用于 IRQL 大于等于 DISPATCH_LEVEL 的代码。除了标准的自旋锁，Windows 还实现了一些自旋锁扩展：**执行体自旋锁**（支持共享和独占的语义）、**排队自旋锁**（queued spin lock）和**栈内排队自旋锁**（in-stack queued spin lock）。

一段内核代码在使用以上这些同步机制来保护数据时，通常会尽可能在很短的时间内

解除保护，以便不妨碍其他的处理器，并且让正在等待执行的线程尽快有机会得到处理器资源。关于 Windows 中这些同步机制的实现细节，请参考 5.3 节。

另一类常见的同步是 PASSIVE_LEVEL 上线程之间的同步。当一个线程的执行条件不满足时，该线程进入等待状态，系统将控制权交给其他满足执行条件但没有得到处理器资源的线程；以后，当该线程的执行条件满足时，它又有机会继续执行。这里的执行条件正是 Windows 提供的线程同步机制中的语义。Windows 定义了统一的机制来支持各种线程同步原语：**分发器对象**（dispatcher object），其数据结构头部为 DISPATCH_HEADER。

Windows 使用**等待块**（wait block）来描述“一个线程正在等待一个分发器对象变成有信号状态”。对于每个处于等待状态的线程，它有一个等待块链表，链表中的每个节点代表该线程正在等待的一个分发器对象；而对于每个分发器对象，它也有一个等待块链表，链表中的每个节点代表了正在等待该对象的一个线程。所以，当一个分发器对象变成有信号状态时，系统循着此对象的等待块链表，就知道该解除哪个或哪些线程，即唤醒它们。线程进入等待的条件（若等待多个对象，则须指明等待任何其中之一[WaitAny]或等待所有对象[WaitAll]）及分发器对象的状态（解除一个线程或所有线程），决定了等待条件该如何被满足。

Windows Server 2003 实现了以下分发器对象：

- **事件**（event）。分为事件通知对象和事件同步对象，其区别在于，当事件对象变成有信号状态时，是解除所有正在等待该对象的线程，还是只唤醒第一个以 WaitAny 方式等待该对象的线程。
- **突变体**（mutant）。突变体是 Windows 内核中**互斥体**（mutex）的实现。如果突变体对象为无信号状态，则一定被某个线程“占有”；否则可满足任何一个线程的等待要求。突变体记录了“所有者”线程信息，通常可用于实现“锁”。
- **信号量**（semaphore）。信号量有一个计数器，用于控制最多可有多少个线程共享同一个资源。当计数器达到预定的最大值时，信号量将变成无信号状态。
- **队列对象**（queue）。这是 Windows 内核中用于支持线程池的机制，其数据结构为 KQUEUE，通常可用于控制一项任务的并发程度。它的典型用途是 **I/O 完成端口**。
- **进程对象**。Windows 的内核进程对象本身也是一个分发器对象，当进程对象被初始创建时，其状态为无信号；当进程结束时，其状态变成有信号。
- **线程对象**。如同进程对象一样，内核线程对象也是一个分发器对象，当线程对象被初始创建时，其状态为无信号；当线程结束时，其状态变成有信号。

- **定时器对象**。定时器对象既是一个像 DPC 一样的例程，也是一个可等待的分发器对象，当设定的时间到期时，定时器变成有信号状态。
- **门对象**（gate object）。在 Windows 中，门对象和门等待是线程调度器的特殊支持，它绕过了以上分发器对象同步过程中的许多步骤。唤醒一个门等待的线程几乎是以最快捷的方式进行的，线程调度器会直接将线程插入到某个处理器的就绪队列中。

除了以上基于分发器对象的同步机制，Windows 执行体还在此基础上，实现了同步语义更为丰富的一些同步机制，包括：**快速互斥体**（fast mutex）、**守护互斥体**（guarded mutex）、**执行体资源**（executive resource）和**推锁**（push lock）。有关 Windows 同步机制的详细介绍，请参考 5.4 节。

2.5 Windows 内核中的公共管理设施

除了上一节介绍的系统机制，Windows 内核中还有一些公共的管理设施对于 Windows 操作系统的运行也至关重要。这一节将讨论 Windows 内核中的对象管理器、配置管理器、Windows 事件追踪（ETW，Event Tracing for Windows）和安全性管理。

2.5.1 Windows 内核中的对象管理

Windows 内核虽然不是用 C++这样的面向对象的编程语言来编写的，但是，其中的资源管理却完全采纳了面向对象的思想。正如本章前面所讲，Windows 的内核包含执行体、（微）内核和 HAL 三层。对象管理器是执行体中的组件，主要管理执行体对象。但是，执行体对象也可能封装了一个或多个内核对象。这一小节我们来讨论对象管理器的实现。

首先，Windows 对象管理器的基本设计意图是（参考 WRK 附带的 NT 设计文档目录中的 ob.doc）：

- 为执行体的数据结构提供一种统一而又可扩展的定义和控制机制。
- 提供统一的安全访问机制。
- 在无须修改已有系统代码的情况下，加入新的对象类型。
- 提供一组标准的 API 来对对象执行各种操作。
- 提供一种命名机制，与文件系统的命名机制集成在一起。

每一个对象都由两部分构成：对象头和对象体。所有对象的对象头都具有统一的格式，其定义如下（见 base\ntos\inc\ob.h）：

```
typedef struct _OBJECT_HEADER {
    LONG_PTR PointerCount;                          // 引用计数
    union {
        LONG_PTR HandleCount;                       // 指向该对象的句柄数
        PVOID NextToFree;                           // 对象被延迟删除时加入到一条链中
    };
    POBJECT_TYPE Type;                              // 指向对象的类型对象
    UCHAR NameInfoOffset;                           // 名称信息的内存偏移
    UCHAR HandleInfoOffset;                         // 句柄信息的内存偏移
    UCHAR QuotaInfoOffset;                          // 配额信息的内存偏移
    UCHAR Flags;

    union {
        POBJECT_CREATE_INFORMATION ObjectCreateInfo;
        PVOID QuotaBlockCharged;
    };

    PSECURITY_DESCRIPTOR SecurityDescriptor;// 安全描述符
    QUAD Body;                                      // 对象体开始
} OBJECT_HEADER, *POBJECT_HEADER;
```

对象头包含了对象管理所需要的基本信息，包括对象名称、类型、引用计数以及安全描述符等。在 Windows 中，每一种对象都需要有一个对应的类型对象（即 OBJECT_TYPE 对象），其定义为：

```
#define OBJECT_LOCK_COUNT 4

typedef struct _OBJECT_TYPE {
    ERESOURCE Mutex;
    LIST_ENTRY TypeList;
    UNICODE_STRING Name;                            // 为方便起见，拷贝自对象头
    PVOID DefaultObject;
    ULONG Index;                                    // 此类型对象在全局数组中的索引
    ULONG TotalNumberOfObjects;
    ULONG TotalNumberOfHandles;
    ULONG HighWaterNumberOfObjects;
    ULONG HighWaterNumberOfHandles;
    OBJECT_TYPE_INITIALIZER TypeInfo;
#ifdef POOL_TAGGING
    ULONG Key;
#endif //POOL_TAGGING
    ERESOURCE ObjectLocks[ OBJECT_LOCK_COUNT ];
} OBJECT_TYPE, *POBJECT_TYPE;
```

系统定义的对象种类是有限的，WRK 支持 31 种对象，各有一个全局的 POBJECT_TYPE 变量指向其类型对象，表 2.4 列出了这 31 种对象，读者可以在 WRK 源代码中找到它们的定义和用法。

表 2.4 WRK 中的全局类型对象变量

CmpKeyObjectType	ExWindowStationObjectType	ObpDeviceMapObjectType
DbgkDebugObjectType	IoAdapterObjectType	ObpDirectoryObjectType
ExCallbackObjectType	IoCompletionObjectType	ObpSymbolicLinkObjectType
ExDesktopObjectType	IoControllerObjectType	ObpTypeObjectType
ExEventObjectType	IoDeviceHandlerObjectType	PsJobType
ExEventPairObjectType	IoDeviceObjectType	PsProcessType
ExMutantObjectType	IoDriverObjectType	PsThreadType
ExpKeyedEventObjectType	IoFileObjectType	SeTokenObjectType
ExProfileObjectType	LpcPortObjectType	WmipGuidObjectType
ExSemaphoreObjectType	LpcWaitablePortObjectType	
ExTimerObjectType	MmSectionObjectType	

为了使每一种对象都可以在系统内部真正可用，系统通常会在初始化过程中调用 ObCreateObjectType 函数构建起这种对象类型，以完成相应全局变量的初始化。那么，在创建类型对象时，需要指定哪些信息呢？我们先看 ObCreateObjectType 的函数原型：

```
NTSTATUS
ObCreateObjectType (
    __in PUNICODE_STRING TypeName,
    __in POBJECT_TYPE_INITIALIZER ObjectTypeInitializer,
    __in_opt PSECURITY_DESCRIPTOR SecurityDescriptor,
    __out POBJECT_TYPE *ObjectType
);
```

输入参数 ObjectTypeInitializer 值得一提，其类型为指向 OBJECT_TYPE_INITIALIZER 结构的指针。对照 OBJECT_TYPE 的定义，我们可以发现，OBJECT_TYPE 中的 TypeInfo 成员也是此类型。以下是 OBJECT_TYPE_INITIALIZER 结构的定义：

```
typedef struct _OBJECT_TYPE_INITIALIZER {
    USHORT Length;
    BOOLEAN UseDefaultObject;
    BOOLEAN CaseInsensitive;
    ULONG InvalidAttributes;
    GENERIC_MAPPING GenericMapping;
    ULONG ValidAccessMask;
    BOOLEAN SecurityRequired;
    BOOLEAN MaintainHandleCount;
    BOOLEAN MaintainTypeList;
    POOL_TYPE PoolType;
    ULONG DefaultPagedPoolCharge;
    ULONG DefaultNonPagedPoolCharge;
    OB_DUMP_METHOD DumpProcedure;
    OB_OPEN_METHOD OpenProcedure;
    OB_CLOSE_METHOD CloseProcedure;
    OB_DELETE_METHOD DeleteProcedure;
```

```
    OB_PARSE_METHOD ParseProcedure;
    OB_SECURITY_METHOD SecurityProcedure;
    OB_QUERYNAME_METHOD QueryNameProcedure;
    OB_OKAYTOCLOSE_METHOD OkayToCloseProcedure;
} OBJECT_TYPE_INITIALIZER, *POBJECT_TYPE_INITIALIZER;
```

可以看出，在调用 ObCreateObjectType 函数来构建一种新的对象类型时，调用者除了可以指定此种类型对象的一些数据特性以外，还可以指定该类型对象的一些基本操作方法，包括 Dump、Open、Close、Delete、Parse、Security、QueryName 和 OkayToClose。对象管理器正是通过这些方法来统一管理各种类型的对象的。系统有一个全局变量 ObpObjectTypes 数组记录了所有已创建的类型，这是一个静态数组，WRK 限定不超过 48 种对象类型。OBJECT_TYPE 中的 Index 成员记录了一个类型对象在此数组中的索引。

一旦类型对象已经被创建，以后内核代码就可以调用 ObCreateObject 来创建此种类型的对象了，以下是 ObCreateObject 函数的原型：

```
NTSTATUS
ObCreateObject (
    __in KPROCESSOR_MODE ProbeMode,
    __in POBJECT_TYPE ObjectType,
    __in POBJECT_ATTRIBUTES ObjectAttributes,
    __in KPROCESSOR_MODE OwnershipMode,
    __inout_opt PVOID ParseContext,
    __in ULONG ObjectBodySize,
    __in ULONG PagedPoolCharge,
    __in ULONG NonPagedPoolCharge,
    __out PVOID *Object
);
```

注意，在创建对象时，ObjectType 参数只是确定了对象头的部分，对象体部分的大小需要在 ObjectBodySize 参数中指定。该函数返回时，Object 输出参数指向对象体起始位置。对象体的格式是特定于某种对象类型的，由相应类型对象的诸多过程来维护。

下面我们以进程和线程对象为例来说明此过程。首先，在系统初始化阶段，我们可以看到下面的代码（base\ntos\ps\psinit.c 中的 323~362 行）：

```
    RtlZeroMemory (&ObjectTypeInitializer, sizeof (ObjectTypeInitializer));
    ObjectTypeInitializer.Length = sizeof (ObjectTypeInitializer);
    ObjectTypeInitializer.SecurityRequired = TRUE;
    ObjectTypeInitializer.PoolType = NonPagedPool;
    ObjectTypeInitializer.InvalidAttributes = OBJ_PERMANENT |
                                              OBJ_EXCLUSIVE |
                                              OBJ_OPENIF;

    RtlInitUnicodeString (&NameString, L"Process");
    ObjectTypeInitializer.DefaultPagedPoolCharge = PSP_PROCESS_PAGED_CHARGE;
    ObjectTypeInitializer.DefaultNonPagedPoolCharge = PSP_PROCESS_NONPAGED_CHARGE;
    ObjectTypeInitializer.DeleteProcedure = PspProcessDelete;
    ObjectTypeInitializer.ValidAccessMask = PROCESS_ALL_ACCESS;
    ObjectTypeInitializer.GenericMapping = PspProcessMapping;
```

```
if (!NT_SUCCESS (ObCreateObjectType (&NameString,
                                     &ObjectTypeInitializer,
                                     (PSECURITY_DESCRIPTOR) NULL,
                                     &PsProcessType))) {
    return FALSE;
}

RtlInitUnicodeString (&NameString, L"Thread");
ObjectTypeInitializer.DefaultPagedPoolCharge = PSP_THREAD_PAGED_CHARGE;
ObjectTypeInitializer.DefaultNonPagedPoolCharge = PSP_THREAD_NONPAGED_CHARGE;
ObjectTypeInitializer.DeleteProcedure = PspThreadDelete;
ObjectTypeInitializer.ValidAccessMask = THREAD_ALL_ACCESS;
ObjectTypeInitializer.GenericMapping = PspThreadMapping;

if (!NT_SUCCESS (ObCreateObjectType (&NameString,
                                     &ObjectTypeInitializer,
                                     (PSECURITY_DESCRIPTOR) NULL,
                                     &PsThreadType))) {
    return FALSE;
}
```

然后，在新建进程时，执行体可以直接使用进程对象的类型对象 PsProcessType，如下所示（base\ntos\ps\create.c 中的 1 108~1 120 行）：

```
Status = ObCreateObject (PreviousMode,
                         PsProcessType,
                         ObjectAttributes,
                         PreviousMode,
                         NULL,
                         sizeof (EPROCESS),
                         0,
                         0,
                         &Process);

if (!NT_SUCCESS (Status)) {
    goto exit_and_deref_parent;
}
```

同样地，以下是创建线程对象的代码（base\ntos\ps\create.c 中的 370~383 行）：

```
Status = ObCreateObject (PreviousMode,
                         PsThreadType,
                         ObjectAttributes,
                         PreviousMode,
                         NULL,
                         sizeof(ETHREAD),
                         0,
                         0,
                         &Thread);

if (!NT_SUCCESS (Status)) {
    ObDereferenceObject (Process);
    return Status;
}
```

对象管理器使用对象头中包含的信息来管理这些对象，在对象头中，除了对象名称和对象类型，另有两个信息值得一提：指针计数，它记录了内核本身（也包括驱动程序）引

用该对象的次数；句柄计数，它记录了有多少个句柄引用此对象，这些句柄可能出现在不同的进程中。

类型对象并不需要为此种类型的对象提供所有在 OBJECT_TYPE_INITIALIZER 定义中出现的方法。对象管理器提供了一些通用的服务，这些通用服务可以应用在任何类型的对象上。例如，OpenProcedure 和 CloseProcedure 主要用于在已经创建的对象上新建句柄，以指向此对象，这两个操作可以统一实现。在前面关于进程和线程的例子中我们可以看到，只有 DeleteProcedure 成员是显式指定的。实际上，针对所有类型的通用服务，加上特定于某种类型的实现，正好体现了面向对象程序设计中的多态性。

另外，对象的构造是由两部分来完成的：(1)调用 ObCreateObject，根据指定的类型对象来完成对象头的初始化，并且按照指定的大小来分配对象体的内存；(2)完成对象体的初始化。前者可以统一完成，但是后者不可以，因为各种类型的对象有自己不同的初始化逻辑。比如，进程对象的构造是在 PspCreateProcess 函数中完成的，而线程对象的构造是在 PspCreateThread 函数中完成的。本书第 3 章将介绍进程对象和线程对象的创建过程。

除了通过以上通用的服务和专用的初始化代码来管理对象以外，Windows 还提供了其他一些统一的机制来管理对象，比如，类型对象中可以跟踪记录当前所有此种类型的对象。因为所有的对象都是在 ObpAllocateObject 和 ObpFreeObject 中分配和释放的，所以，我们可以在这两个函数中完成此功能。WRK 没有提供这样的能力，读者不妨作为一个练习自己加上。

以上介绍了 Windows 以类型对象来管理对象的机理，除此以外，Windows 还允许以名称的方式来管理对象，这对于跨进程、跨模块的操作带来了极大的方便。试想，如果两个进程想要共享一个对象，那么，它们可以通过名称各自独立地创建或引用此对象，而无须通过其他的途径来传递。为了做到这一点，Windows 必须维护一套全局的名称查询机制。表 2.4 中的 ObpDirectoryObjectType 类型对象正是实现这一机制的关键。

Windows 内部维护了一个对象层次目录（即系统全局名字空间），其根目录对象是由全局变量 ObpRootDirectoryObject 来定义的。在根目录之下，系统内置了一些子目录。在 WRK 中，通过查询 NtCreateDirectoryObject 函数被调用的情况，读者可以看到 Callback、ArcName、Device、Driver、FileSystem、KernelObjects、ObjectTypes、GLOBAL??和 Security 子目录的创建过程。譬如，以下代码创建了 Driver 子目录（参见 base\ntos\io\iomgr\ioinit.c 中的 IopCreateRootDirectories 函数）：

```
// 创建 "\Driver" 目录
RtlInitUnicodeString( &nameString, L"\\Driver" );
InitializeObjectAttributes( &objectAttributes,
                            &nameString,
                            OBJ_PERMANENT|OBJ_KERNEL_HANDLE,
```

```
                                (HANDLE) NULL,
                                (PSECURITY_DESCRIPTOR) NULL );

    status = NtCreateDirectoryObject( &handle,
                                      DIRECTORY_ALL_ACCESS,
                                      &objectAttributes );
    if (!NT_SUCCESS( status )) {
        return FALSE;
    } else {
        (VOID) ObCloseHandle( handle , KernelMode);
    }
```

对象管理器提供了一些基本的操作用于在名字空间中插入、查询和删除目录或目录项。例如，ObpLookupDirectoryEntry 函数的功能是，在一个指定的目录中查找一个名称；ObpInsertDirectoryEntry 函数把一个对象插入到一个目录中；ObpDeleteDirectoryEntry 函数删除刚刚找到的那一项。这三个函数都直接在一个子目录中进行操作。另外有一个重要的操作是 ObpLookupObjectName，它可以从指定的目录或根目录，递归地根据名称来找到一个对象。其函数原型如下：

```
NTSTATUS
ObpLookupObjectName (
    IN HANDLE RootDirectoryHandle,
    IN PUNICODE_STRING ObjectName,
    IN ULONG Attributes,
    IN POBJECT_TYPE ObjectType,
    IN KPROCESSOR_MODE AccessMode,
    IN PVOID ParseContext OPTIONAL,
    IN PSECURITY_QUALITY_OF_SERVICE SecurityQos OPTIONAL,
    IN PVOID InsertObject OPTIONAL,
    IN OUT PACCESS_STATE AccessState,
    OUT POBP_LOOKUP_CONTEXT LookupContext,
    OUT PVOID *FoundObject
    );
```

对象管理器的两个接口函数 ObOpenObjectByName 和 ObReferenceObjectByName，正是通过 ObpLookupObjectName 来完成其打开对象或引用对象的功能的。另一个接口函数 ObInsertObject，它的作用是把一个对象插入到一个进程的句柄表中，它也通过 ObpLookupObjectName 来验证待插入的对象是否在全局名字空间中并不存在，当然，对于无名称的对象并不需要做这样的检查。ObpLookupObjectName 的代码较长，位于 base\ntos\ob\obdir.c 中，下面列出其基本的执行逻辑：

- 参数检查。
- 如果调用者指定了 RootDirectoryHandle 参数，则利用此 RootDirectory 的 Parse 方法来解析对象名称，直到解析成功或者不成功，或者指示从头解析。
- 如果调用者没有指定 RootDirectoryHandle 参数，则系统从全局的根目录 ObpRootDirectoryObject 开始解析。在这种情况下，传递进来的对象名称必须以“\”

开始。如果待查找的名称仅仅是“\”，则执行特殊处理。否则，执行下面的逻辑：

- 首先判断名称是否以“\??\”打头，如果是的话，需要拿到当前进程的 DeviceMap（设备表），以进一步查询。
- 如果名称正好是“\??”，则直接返回当前进程的 DeviceMap 作为结果。
- 调用 ObpLookupDirectoryEntry 函数，层层递进，或者碰到具有 Parse 方法的对象，由它来解析余下的名称字符串，或者碰到子目录对象，从而可以在子目录对象中进一步查询下一级名称。

前面提到过，内核代码在创建对象类型时，可以为新的对象类型指定 Parse 方法。每一种对象都可以有它自己的名称解析方法，这使得 Windows 的名字系统非常强大，既允许以基本的目录方法来管理名称的层次结构，也允许特定类型的对象有它自己的命名和解析策略。譬如，文件（File）对象有它自己的 Parse 方法，从而可以方便地支持我们所熟悉的文件系统中的目录结构。

在 ObpLookupObjectName 的代码逻辑中，我们可以看到进程的设备表（DeviceMap），而且，在目录对象的数据结构 OBJECT_DIRECTORY 中也有一个名为 DeviceMap 的成员，指向一个 DEVICE_MAP。这里简单介绍一下 DeviceMap 的含义，它定义了一个 DOS 设备名字空间，比如驱动器字母（像 C:、D:）和一些外设（如 COM1）。当对象管理器看到一个以“\??\”打头的名称或者像“\??”这样的名称，它会利用进程的 DeviceMap 来获得相应的对象目录，然后进一步解析剩余的名称字符串。在第 7 章介绍卷挂载时我们将会看到“\??”名称的解析过程。

对象管理器中的对象是执行体对象，它们位于系统地址空间中，因而所有的进程都可以访问这些对象。但是，在进程地址空间中运行的用户模式代码不能用指针的方式来引用这些对象，它们在调用系统服务时只能通过句柄来引用执行体对象。句柄是进程范畴的概念，它一定要在特定的进程环境中才有意义。在内核中，将一个句柄转换成对应的对象，可以通过 ObReferenceObjectByHandle 函数（在 base\ntos\ob\obref.c）来完成，该函数负责从当前进程环境或内核环境的句柄表中获得指定的对象引用。本书第 3 章将讨论进程的句柄表结构。

最后介绍一下对象的内存结构和它的生命周期管理。前面提到过，对象有对象头和对象体两部分，对象头的数据结构是 OBJECT_HEADER，而对象体的类型则因对象而异，所以，我们可以看到很多函数在接受一个对象作为参数时其类型为 PVOID。然而，对象头和对象体在内存中是如何存储的呢？通过阅读 ObpAllocateObject 函数的实现代码（位于 base\ntos\ob\obcreate.c 文件中），我们可以知道，对象头和对象体实际上是在同一块内存中，而且，在对象头之前可能还有其他一些可选的内容，包括 QuotaInfo、HandleInfo、NameInfo

和 CreatorInfo 四部分，这每一部分是否出现，取决于对象的类型以及创建对象时所提供的参数。基于这样的内存结构，从对象体转换到对象头不难实现，见下面的代码：

```
#define OBJECT_TO_OBJECT_HEADER( o ) \
   CONTAINING_RECORD( (o), OBJECT_HEADER, Body )
```

而对象的其他信息，例如 QuotaInfo、HandleInfo、NameInfo，则可以通过 OBJECT_HEADER 的 QuotaInfoOffset、HandleInfoOffset 和 NameInfoOffset 来获得（见 OBJECT_HEADER_TO_QUOTA_INFO、OBJECT_HEADER_TO_HANDLE_INFO 和 OBJECT_HEADER_TO_NAME_INFO 函数）。而如果要获得对象的 CreatorInfo 信息，则可以根据这样的内存结构直接往前推算得到（见 OBJECT_HEADER_TO_CREATOR_INFO 函数）。因此，内核代码在任何时候都可以很方便地获得对象的各种信息。

对象是通过引用计数来管理其生命周期的，一旦引用计数为零，则对象的生命周期结束，它所占用的内存也可以被回收。对象的引用计数来源于两个方面。第一个来源是内核中的指针引用。一旦内核中新增了一个对象的引用，则对象的引用计数需要增一；如果一个对象的引用不再有用，则引用计数减一。这两种作用是在 ObReferenceObjectByPointer 和 ObDereferenceObject 函数中完成的。第二个来源是，一个进程打开一个对象并获得一个句柄，它以后通过此句柄来引用此对象。对象头信息中准确地记录了有多少个句柄指向此对象，当一个句柄不再被使用时，其句柄计数减一。这两种作用是在函数 ObpIncrementHandleCount 和 ObpDecrementHandleCount 中完成的。

2.5.2 注册表和配置管理器

Windows 系统的很多组件都是可以配置的，其内核组件通常都支持一些参数，甚至有些完全依赖于系统的配置信息，例如 I/O 管理器和即插即用管理器在初始化阶段依据系统的配置设置来列举和加载设备驱动程序。Windows 操作系统提供了一个称为“注册表（registry）”的中心存储设施作为系统的配置和管理中心，应用程序和内核通过访问注册表来读写各种设置。Windows 同时也提供了一些 API 供应用程序访问注册表，这些 API 函数在接到注册表访问请求以后，将它们转发给内核的系统服务。在内核中，执行体包含一个称为“配置管理器（configuration manager）”的组件，它是注册表的真正实现。注册表是由一组称为储巢（hive）的文件构成的，每个储巢文件的内部都包含了一个树状层次结构。读者可以将每个储巢想象成一个文件系统，类似于我们常常见到的目录和文件结构。从这一层意义来讲，配置管理器差不多等同于一个文件系统驱动程序。这一节我们首先讨论注册表的总体结构，然后介绍储巢的存储结构，最后介绍配置管理器的实现机理以及它所提供的接口。

Windows 注册表是一个树状结构，每个节点是一个键（key）或值（value）。键是一

个容器，好比文件系统中的目录，它可以包含其他的键（称为子键）和值。值存储的是数据，好比是文件系统中的文件。注册表的根是一个键，称为根键。通常而言，注册表对于一般的计算机用户是隐藏的，也就是说，桌面用户不需要知道注册表的存在，更不用说了解注册表结构了。但高级用户可以通过一些工具来访问注册表，比如 Windows 随系统提供的注册表编辑工具（regedit.exe）。软件安装程序和应用程序则调用 Windows API 来读写注册表中的软件设置。图 2.8 是 Windows Server 2003 中的 regedit.exe 界面。

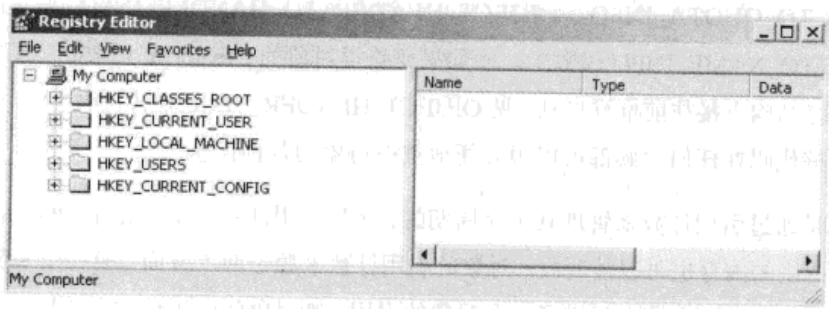


图 2.8 Windows Server 2003 中的 regedit.exe 界面

注册表的值可以有多种类型[MSDN-REG]，绝大多数注册表值的类型为 REG_DWORD（32 位整数）、REG_BINARY（二进制数据）和 REG_SZ（字符串）。有一种特殊的类型为 REG_LINK，即符号链接，允许一个键或值指向另一个键或值。实际上，图 2.8 中所示的注册表顶级结构也是利用符号链接构建起来的。注册表顶级结构下有 8 个根键，它们的缩写和含义如表 2.5 所列。其中，HKEY_PERFORMANCE_<*XXX*>根键在 regedit.exe 工具中是不可见的，这些性能数据并非真正位于注册表的储巢中，它们反映了系统内存中的状态信息，只不过 Windows 为上层应用程序提供了通过注册表 API 来获得性能信息的便捷途径而已。

表 2.5 Windows 注册表的根键

根键	缩写	内容说明
HKEY_USERS	HKU	有关该机器上所有用户账户的信息
HKEY_LOCAL_MACHINE	HKLM	有关当前系统的信息
HKEY_CURRENT_USER	HKCU	有关当前用户的信息，这是一个符号链接，指向 HKU 中对应于当前用户的子键
HKEY_CLASSES_ROOT	HKCR	有关文件关联和 COM 的设置信息，这是 HKLM\SOFTWARE\Classes 和 HKCU\SOFTWARE\Classes 两棵子树经过合并后的一个视图
HKEY_CURRENT_CONFIG	HKCC	有关当前硬件配置的信息，这是一个符号链接，指向 HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current

续表

根键	缩写	内容说明
HKEY_PERFORMANCE_DATA	HKPD	与性能有关的数据，这是系统的运行状态信息
HKEY_PERFORMANCE_TEXT		以美国英语的文本来描述性能计数器
HKEY_PERFORMANCE_NLSTEXT		以系统本地语言的文本来描述性能计数器

除了 HKEY_PERFORMANCE_<*XXX*>以外，在其他的 5 个根键中，真正存放系统设置信息的子树是 HKLM 和 HKU。HKLM 存放有关系统全局的信息，包括 5 个子键，分别为 HARDWARE（硬件设置）、SAM（本地账户和组的信息）、SECURITY（系统全局范围的安全策略和用户权限设置）、SOFTWARE（系统中的全局配置信息，在系统引导时不需要）和 SYSTEM（系统中的全局配置信息，在系统引导时需要，包括设备驱动程序和系统服务等）。HKU 为系统中每个加载过的用户轮廓包含一个子键，也包含一个名为.DEFAULT 的子键，这是系统的默认轮廓，当登录进程 winlogon.exe 为第一次登录到系统中的用户创建轮廓时将以此为基础。

因此，HKLM 存放的是针对当前整个系统的配置信息；而 HKU 以系统中的用户为基础存放各个用户账户的轮廓信息。HKLM 和 HKU 中的结构不随当前登录账户而发生变化，它们代表系统的全局视图；但其他三个根键，HKCU、HKCR 和 HKCC，则利用 HKLM 和 HKU 中的信息重构出一个特定于当前登录用户和当前硬件环境的视图。这使得每个用户既可以有自己独立的配置设置，又能够共享系统的全局配置信息。操作系统和应用软件可以利用这种灵活性，将系统或软件设置应用到整个系统中，或仅仅对当前用户起作用。

了解了注册表的逻辑结构，我们再来看它的存储结构。正如前面所讲，注册表是由一组储巢构成的，每个储巢包含了一个由键和值构成的层次结构。表 2.6 列出了 Windows Server 2003 系统中各个储巢的注册表路径和文件路径。一个系统的储巢列表存放在 HKLM\SYSTEM\CurrentControlSet\Control\hivelist 键下，如图 2.9 所示。当系统初始化时，HKLM\SYSTEM 总是先被加载进来，然后配置管理器找到 hivelist 键，继而加载其他储巢，并创建注册表根键，将这些储巢链接起来，从而建立起完整的注册表结构。

表 2.6 Windows Server 2003 中的储巢以及对应的注册表键

储巢的注册表键	储巢的文件路径
HKLM\HARDWARE	易失储巢，仅存在于内存中
HKLM\SAM	\WINDOWS\system32\config\SAM
HKLM\SECURITY	\WINDOWS\system32\config\SECURITY
HKLM\SOFTWARE	\WINDOWS\system32\config\software

续表

储巢的注册表键	储巢的文件路径
HKLM\SYSTEM	\WINDOWS\system32\config\system
HKU\.DEFAULT	\WINDOWS\system32\config\default
HKU\<用户的 SID>	\Documents and Settings\<用户名>\NTUSER.DAT
HKU\<用户的 SID>_Classes	\Documents and Settings\<用户名>\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat

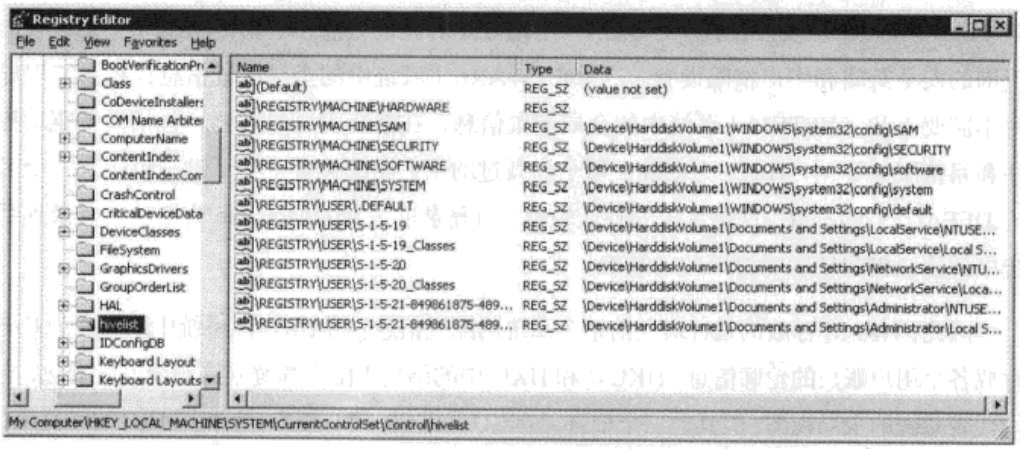


图 2.9 在 Windows Server 2003 系统中的 hivelist 键例子

储巢的内部结构类似于一个文件系统，而储巢相当于是一个磁盘分区。储巢的基本分配单元称为块（block），类似于文件系统定义的簇（cluster）。当储巢为了存储新的数据而需要扩展时，它总是按照块的粒度来增长。在 Windows 中，注册表的块的大小为 4 KB（4 096 B）。储巢的第一个块称为基本块，它包含了储巢文件标识、最新序列号、最后一次写操作的时间戳、储巢格式的版本号、校验和，以及储巢的内部文件名。

储巢中的注册表数据是按照巢室（cell）来组织的。巢室可大可小，具体取决于它的类型和数据，每个巢室可以存放一个键、值、安全描述符、子键列表或者值列表，对应的巢室分别称为键巢室、值巢室、安全描述符巢室、子键列表巢室和值列表巢室。巢室在储巢文件中的偏移称为该巢室的索引（cell index），其他巢室可以利用此巢室索引来引用它，从而建立起巢室之间的关系。比如，父键的巢室包含一个指向子键列表巢室的巢室索引；子键列表巢室中又包含了一组巢室索引，每个代表一个子键。因此，配置管理器利用巢室索引，可以从父键定位到子键。反过来，每个键还包含了父键巢室的巢室索引，因而配置管理器也可以从子键定位到父键。同样地，利用键的值列表巢室索引，配置管理器可以定位到值列表，再根据每一个值的巢室索引，可以定位到一个值。图 2.10 显示了一个键及

其子键和值之间的存储关系，其中每个长方块代表一个巢室，每个箭头线则代表了通过巢室索引建立起来的一个引用关系。

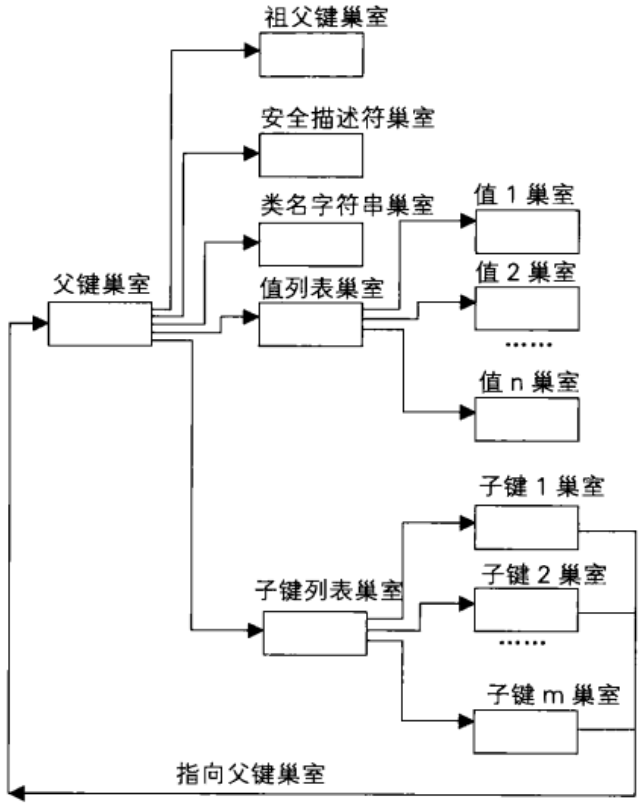


图 2.10 一个键巢室与相关的巢室

基于巢室和巢室索引的关系，如果要从父键访问它的一个子键，那么，首先要找到父键的子键列表巢室，然后在该子键列表巢室指示的所有子键巢室中，定位到符合搜索条件的那个子键巢室。注意，子键列表中的子键巢室索引是根据子键字符串名的顺序来排序的，因此查找子键时可以采用二分查找法，从而将搜索复杂度降低到 $O(\log_2 n)$，然而，值列表中的巢室索引并非是排序的，所以，搜索值的复杂度为 $O(n)$。

巢室的大小并非都相同，配置管理器将根据需要从储巢中申请特定大小的存储空间作为一个巢室。然而，如果配置管理器必须要扩展已有的储巢文件才能分配一个新的巢室，那么，它并非简单地申请此巢室大小的磁盘空间，而是会申请一个称为巢箱（bin）的存储单元。巢箱总是以块为边界，所以，当为一个巢室而扩展储巢文件时，实际申请的存储空间要扩展到下一个块边界。巢箱中未使用的空间被当做空闲空间管理起来，从而可以再分配给其他的巢室。而且，当巢室被删除时，它们的空间会被回收，如果有可能的话，相邻的空闲空间被合并成更大的空闲空间。这样做也可以避免产生很多碎的空闲空间。然而，

储巢文件本质上是越长越大的，只有当尾部的巢箱空闲时，配置管理器才会缩短储巢文件。要想压缩整个储巢文件中的空闲巢箱，只有在原来储巢的基础上重构一个新的储巢文件（通过 RegSaveKey 函数），然后替换或恢复该储巢所在的键（通过 RegReplaceKey 或 RegRestoreKey 函数）。

配置管理器在巢室的基础上又引入了巢箱机制来管理储巢的空间使用，可以避免频繁地操作储巢文件，以及更加有效地管理内存中的巢室，毕竟巢箱的变化比巢室的变化要少得多。

为了方便地访问注册表中的数据，最简单的做法是，将储巢文件读入到内存中，因而，巢室索引可以被解释成相对于该文件在内存中的起始地址的偏移，写操作也需要在适当的时候同步更新到储巢文件中。然而，这种做法需要消耗较大的虚拟地址空间，通常是换页内存池空间。早期版本的 Windows 使用了这种做法，而在 Windows XP/Server 2003 中，配置管理器利用缓存管理器（参考 7.2 节）来映射储巢文件中的数据，它可以有选择地映射所需要的那部分数据，并且也能够解除不再需要访问的数据或者长久不访问的数据，以避免储巢文件占据太多的系统缓存空间。但是，同一个储巢的各个部分被独立地映射到系统缓存中，这样会导致巢室索引不能被直接转译为内存地址。

配置管理器使用了一种类似于 Intel x86 处理器的页表映射的做法（参考 4.1.1 节）来解决巢室地址转译。如图 2.11 所示，一个 32 位的巢室索引被分成四个组成部分：存储类型、巢室目录索引、巢室表索引和块内偏移。存储类型有两种可能：稳定的（stable，最高位用 0 表示）和易失的（volatile，最高位用 1 表示）。每个储巢在内存中有两个巢室目录，分别对应于稳定的和易失的配置数据；每个巢室目录有 1 024 项，每一项指向一个巢室表；每个巢室表包含 512 个表项，每一项指向一个块。由于配置管理器用巢箱来管理内存分配，而巢箱总是以块为边界（4 KB），所以，巢室索引的最后 12 位指定了一个巢室在块内的偏移。

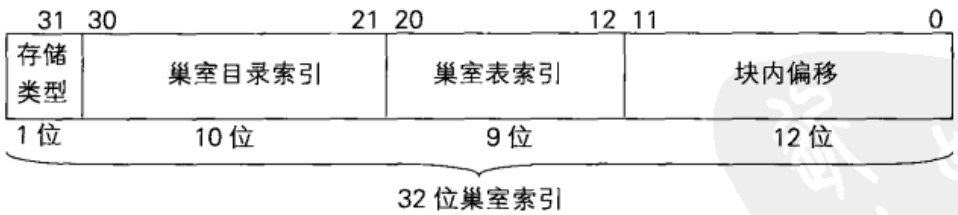


图 2.11 巢室索引的组成结构

基于这样的巢室索引结构，配置管理器将只为每个储巢映射那些需要用到的巢箱，而不是所有的巢箱。巢室目录和巢室表仍然占用换页内存池的空间，但通常情况下，相比于整个储巢文件，它们要小得多。配置管理器通过这种巢室映射的做法，可有效地降低注册表数据的内存使用量。

理解了注册表的逻辑结构和存储结构以后，我们来看 Windows 内核中配置管理器的实现。配置管理器是执行体中的组件，而且，正如前面所讲，它的实现依赖于内存管理器和缓存管理器（以及文件系统），这意味着它必须要在这些组件初始化以后才能正常工作；然而，在系统初始化的早期（比如 I/O 子系统的初始化），Windows 已经需要使用注册表中的配置信息了，但此时配置管理器尚未被初始化。Windows 的做法是，在内核初始化以前，内核加载器（ntldr）已经将整个 HKLM\SYSTEM 储巢作为一个只读文件加载到了内存中，因而配置管理器在完全初始化以前只需直接把巢室索引加上该储巢的内存映像地址，就可以得到巢室的内存地址。这一做法有一个限制，即，在配置管理器完全初始化以前，系统只能访问 HKLM\SYSTEM 中的设置，换句话说，Windows 必须把初始化早期用到的各种设置存放在 HKLM\SYSTEM 中。

下面我们来看一下配置管理器和注册表的初始化过程。配置管理器建立起完全的注册表视图分三个阶段来完成：第一，在内核初始化阶段，建立起 HKLM\SYSTEM 和 HKLM\HARDWARE 储巢；第二，由会话管理器（smss.exe 进程）建立起 HKLM\SAM、HKLM\SECURITY、HKLM\SOFTWARE 和 HKU\.DEFAULT 储巢；第三，当加载用户轮廓时建立起 HKU\<用户的 *SID*>储巢，这是由登录进程（winlogon.exe）来完成的。这里第一阶段可以看做配置管理器的初始化，以及注册表的临时初始化；第二阶段可以看做注册表中系统部分的初始化；第三阶段可以看做注册表中用户部分的初始化。

首先来看第一阶段的初始化，它发生在一个关键点上：在内核初始化过程中，在对象管理器和缓存管理器初始化以后，但在 I/O 子系统初始化以前。内核在这个点以前，不能访问注册表中的任何信息；而在这个点以后，可以访问 HKLM\SYSTEM 和 HKLM\HARDWARE 中的设置。执行这一初始化过程的函数为 CmInitSystem1，它是在内核初始化过程中由 Phase1InitializationDiscard 函数调用的。关于内核的初始化过程请参考 2.6 节的介绍。

CmInitSystem1 函数（参见 base\ntos\config\cmsysini.c 文件）负责完成以下事项：

- 初始化配置管理器的全局变量，包括各种链表和同步对象。
- 创建注册表键的类型对象 CmpKeyObjectType，CmInitSystem1 通过调用 CmpCreateObjectTypes 函数来完成。请参考 cmsysinit.c 文件中 CmpCreateObjectTypes 函数的代码。在这里可以看到键对象的一些方法，例如 Parse 方法为 CmpParseKey。
- 创建主储巢 CmpMasterHive，这是一个易失储巢，代表了注册表的根。创建储巢的函数为 CmpInitializeHive，其代码位于 base\ntos\config\cminit.c 文件中。
- 调用 CmpCreateRegistryRoot 函数建立起注册表的根：在主储巢中创建节点

"\REGISTRY"，并创建一个键对象指向该节点，然后将该对象插入到对象名字空间的根下面。

- 调用 NtCreateKey 函数创建"\REGISTRY\MACHINE"和"\REGISTRY\USER"节点。
- 调用 CmpInitializeSystemHive 函数创建系统储巢。在 CmpInitializeSystemHive 函数中，它根据 ntldr 传递进来的已加载的原始 SYSTEM 储巢映像，来初始化内存中的 SYSTEM 储巢。CmpInitializeSystemHive 函数调用 CmpInitializeHive 来初始化 SYSTEM 储巢，并调用 CmpLinkHiveToMaster 将它链接到主储巢中。
- 调用 CmpCreateControlSet 函数，根据加载信息创建符号链接"\Registry\Machine\System\CurrentControlSet"。
- 调用 CmpInitializeHive，创建 HARDWARE 储巢，这是一个易失储巢。然后调用 CmpLinkHiveToMaster 将它链接到主储巢中。
- 接下来，利用加载块参数，将有关当前这次引导的信息写到注册表中：
  - 调用 CmpInitializeHardwareConfiguration，创建"\Registry\Machine\Hardware"节点，并且把硬件信息设置到注册表中。
  - 调用 CmpInitializeMachineDependentConfiguration 函数，把与机器相关的配置数据设置到注册表 HARDWARE 储巢中。
  - 调用 CmpSetSystemValues，将这次系统启动的信息写到注册表中。
  - 调用 CmpSetNetworkValue，将这次启动的网络信息写到注册表中。

因此，CmInitSystem1 函数将注册表结构初步建立起来，它构造了主储巢、HKLM\SYSTEM 和 HKLM\HARDWARE 三个储巢，并且也建立起与这次启动有关的符号链接和配置信息，为系统的进一步初始化提供了基本的配置信息。

其次，我们再来看注册表的进一步初始化。数组 CmpMachineHiveList 包含 6 个储巢，对应于表 2.6 中的前 6 个储巢。这些储巢（包括 HKLM\SYSTEM 和 HKLM\HARDWARE）是由会话管理器进程（smss.exe）通过 NtInitializeRegistry 系统服务加载和初始化的。参考 base\ntos\config\ntapi.c 文件中 NtInitializeRegistry 函数的代码，在一次正常启动过程中，它调用 CmpCmdInit 函数执行注册表的进一步初始化。在正常启动情形下，CmpCmdInit 函数调用 CmpInitializeHiveList 来初始化储巢列表中的指定储巢，以及建立相应的符号链接。

由于 CmpInitializeHiveList 是在会话管理器进程环境中执行的，而加载和初始化储巢的动作必须在 System 进程中完成，因此，CmpInitializeHiveList 会为储巢列表中的每一个

储巢创建一个系统线程，由该系统线程来初始化该储巢。系统线程的主例程为 CmpLoadHiveThread，参数为每个储巢在 CmpMachineHiveList 数组中的索引。

在 CmpLoadHiveThread 函数中，对于尚未加载的储巢，包括 HKLM\SAM、HKLM\SECURITY、HKLM\SOFTWARE 和 HKU\.DEFAULT，它会调用 CmpInitHiveFromFile 来完成储巢的加载和初始化；而对于已经被初始化的非易失储巢，即 HKLM\SYSTEM，则调用 CmpOpenHiveFiles 打开系统储巢文件，因为在此之前系统储巢文件实际上一直没有被通过文件系统打开过。经过这一步以后，系统储巢被完全初始化。

随着系统的进一步引导，当需要特定于用户的配置信息时，注册表的 HKU 子树下的用户储巢也必须建立起来。这些储巢是按需加载和初始化的，由登录进程（winlogon.exe）在建立起用户运行环境时完成，譬如当用户登录到系统中，或者系统以特定的用户身份来启动一个进程或服务时。Winlogon 通过 NtLoadKey 系统服务将一个储巢文件链接到注册表中，而 NtLoadKey 又进一步调用 CmLoadKey 来完成实际的加载和链接操作。

CmLoadKey 函数的实现比较直截了当，其代码位于 base\ntos\config\cmapi.c 文件中。它首先调用 ObOpenObjectByName 打开注册表中目标键对象的句柄，如果该键对象已经存在，则此调用可以锁住它，以便进行“是否已被加载”的检查。然后，CmLoadKey 调用 CmpCmdHiveOpen 函数打开指定的储巢文件。如同前面提到的 CmpLoadHiveThread 系统线程一样，CmpCmdHiveOpen 调用 CmpInitHiveFromFile 函数来加载并初始化一个储巢。最后，CmLoadKey 调用 CmpLinkHiveToMaster 函数，将新储巢链接到主储巢中恰当之处。

以上讨论了配置管理器的初始化以及 Windows 注册表的建立过程。储巢是配置管理器的核心概念，也是注册表存储结构中的文件实体。WRK 包含了配置管理器的完整代码，储巢的数据类型为 CMHIVE，其内嵌的 HHIVE 成员是它的数据管理结构。正如本节前文所讲，储巢内部的数据管理类似于一个文件系统，它的数据存储单元按照巢箱来分配，而巢箱以块（4 KB 大小）为边界；储巢内部的逻辑数据结构为巢室，巢室有不同的类型，其大小亦不尽相同。在配置管理器的实现中，巢室的数据结构为 HCELL，巢箱的数据结构为 HBIN。空闲的巢箱形成一个空闲链表。实际上，HHIVE 数据结构包含两个 Storage 成员，分别对应于稳定的储巢和易失的储巢；在 Storage 成员中，有空闲巢箱链表，以及一套用于转译巢室索引的巢室目录和巢室表。关于这些数据结构的定义，参见 base\ntos\inc\hivedata.h 文件，本书将不再解释这些数据结构及相应的管理算法。

注册表的层次结构形成了一个名字空间，配置管理器定义了一个以“Key”命名的对象类型，从而将该名字空间与对象管理器的全局名字空间整合起来。正如本节前文所讲，配置管理器在初始化阶段调用 CmpCreateObjectTypes 函数，创建了类型对象全局变量

CmpKeyObjectType。配置管理器充分利用了对象管理器提供的对象管理框架，让注册表中的每个键自动成为对象管理器中的一个对象。对于每个打开的注册表键，配置管理器分配一个键控制块（key control block），其数据结构为 CM_KEY_CONTROL_BLOCK，它包含了该控制块所引用的键节点所在的储巢和巢室索引。配置管理器将所有的键控制块放在一张散列表（全局变量 CmpCacheTable）中，因而可以快速地根据名称来搜索已有的键控制块。散列表 CmpCacheTable 实际上是一个包含 2 048 个元素的数组，散列表的键 ID 是由键控制块所引用的键对象的名称通过计算而获得。每个键控制块然后被放到散列表的相应桶中，放到同一个散列桶中的所有键控制块形成一个链表。关于散列表 CmpCacheTable 的实现和用法，请参考 base\ntos\config\cmsubs.c 中的 CmpInsertKeyHash、CmpRemoveKeyHash 和 CmpCreateKeyControlBlock 函数。

接下来看一看当内核或应用程序访问一个注册表键时对象管理器和配置管理器的名称解析过程。这涉及两个常用的操作：系统服务 NtOpenKey 和 NtQueryValueKey，或者 ZwOpenKey 和 ZwQueryValueKey。根据内核函数的命名约定，我们知道，Nt<*Xxx*>函数供用户模式应用程序使用，而 Zw<*Xxx*>函数供内核代码直接调用。这两组函数的功能是等价的，这里我们仅讨论 NtOpenKey 和 NtQueryValueKey 函数，其原型如下：

```
NTSTATUS
NtOpenKey(
    __out PHANDLE KeyHandle,                              // 返回的句柄
    __in ACCESS_MASK DesiredAccess,                       // 访问模式
    __in POBJECT_ATTRIBUTES ObjectAttributes              // 指定对象的属性
    );

NTSTATUS
NtQueryValueKey(
    __in HANDLE KeyHandle,                                // 要操作的注册表键
    __in PUNICODE_STRING ValueName,                       // 注册表键下值的名称
    __in KEY_VALUE_INFORMATION_CLASS KeyValueInformationClass,// 指定了待获取的信息的类型
    __out_bcount_opt(Length) PVOID KeyValueInformation,      // 接收数据的缓冲区
    __in ULONG Length,                          // KeyValueInformation 缓冲区的长度
    __out PULONG ResultLength                   // 实际返回的长度
    );
```

这两个函数的代码位于 base\ntos\config\ntapi.c 文件中。NtOpenKey 系统服务接收到的对象名称位于 ObjectAttributes.ObjectName 中，它检查 KeyHandle 和对象名称参数是否可以正确地访问，然后将打开注册表键对象的操作全盘交给对象管理器的 ObOpenObjectByName 函数来完成。从这里也可以看出，注册表的接口与实现，都跟对象管理器的框架融合在一起。

根据上一小节关于对象管理器的介绍，我们知道，ObOpenObjectByName 函数通过 ObpLookupObjectName 函数来完成对象打开操作，它层层递进解析一个名称串，若碰到目录对象，则在目录中查询剩余的名称串；若碰到支持 Parse 方法的对象，则交给 Parse

方法来解析剩余的名称串。在 NtOpenKey 的情形中，它的 ObjectAttributes 参数可能已经指定了一个搜索根目录，即 RootDirectory；也可能直接从全局名字空间的根下开始查找，此时调用者应该指定注册表键的全路径名。注册表键的全路径名以“\Registry”作为开始，例如，HKLM\SYSTEM\CurrentControlSet\services 的全路径名为“\Registry\Machine\System\CurrentControlSet\services”。

由于配置管理器已经在全局名字空间的根下创建了一个名为“REGISTRY”的键对象（注意，这不是目录对象，参见 CmpCreateRegistryRoot 函数的代码），所以，当 ObpLookupObjectName 函数解析一个注册表键的全路径名称时，它首先在根目录下找到“REGISTRY”键对象，然后调用键对象类型的 Parse 方法来解析剩余的名称字符串。键对象类型的 Parse 方法为 CmpParseKey 函数。因此，注册表键的名称解析实际上是由 CmpParseKey 函数来完成的，参见 base\ntos\config\cmparse.c 文件中 333~1 212 行代码。CmpParseKey 函数的实现并不难理解，它首先调用 CmpBuildHashStackAndLookupCache 函数，在散列表中查找已经打开的键对象，若能直接找到，则无须进一步名称解析；否则，需要顺序解析剩余的名称串，对于路径上的每一个子键，逐个为它们创建键控制块（通过调用 CmpCreateKeyControlBlock 函数）。最后，CmpParseKey 调用 CmpDoOpen 函数打开此注册表键，并根据需要创建一个键控制块。

ObOpenObjectByName 函数接收到一个指向键对象的句柄，键对象的数据结构为 CM_KEY_BODY，其内部指向一个键控制块。如果两个应用程序打开同一个注册表键的话，它们都会接收到一个键对象，但这两个键对象指向一个公共的键控制块。键控制块有一个引用计数用于跟踪一个键被多少个客户引用。当引用计数为零时，表明该键控制块已不再被使用了，于是配置管理器将它从散列表中移除，并且回收该键控制块。

NtQueryValueKey 函数相对要简单得多，因为它的参数 KeyHandle 已经指示了要查询哪个键，所以，它只需调用 ObReferenceObjectByHandle 函数即可获得目标键的键对象。然后它调用 CmQueryValueKey 函数从目标键中读取指定的值的信息。CmQueryValueKey 调用 CmpFindValueByNameFromCache 函数从键巢室的值列表中查找指定名称的值。配置管理器为每个注册表键在它的键控制块中保存了一个值缓存，用于存放它的值列表。CmpFindValueByNameFromCache 调用 CmpGetValueListFromCache 获得此缓存，如果缓存尚未建立，则 CmpGetValueListFromCache 试图建立此缓存。然后，CmpFindValueByNameFromCache 为值列表中的每个值调用 CmpGetValueKeyFromCache，并检查值名称是否匹配，直至找到指定名称的值或所有的值都不匹配。关于这些操作值列表和值巢室的函数的实现代码，请参考 base\ntos\config\cmtree.c 文件。

除了 NtOpenKey 和 NtQueryValueKey 函数以外，配置管理器还提供了其他一些系统服务，包括 NtCreateKey、NtDeleteKey、NtDeleteValueKey、NtEnumerateKey、NtFlushKey、NtQueryKey、NtRestoreKey、NtSaveKey、NtSetValueKey，以及前面曾经提到过的 NtLoadKey、NtReplaceKey 和 NtInitializeRegistry 等。关于这些系统服务例程的定义和实现，请参考 base\ntos\config\ntapi.c 文件。内核代码调用的是与这些系统服务对应的 Zw<*Xxx*>内核例程，比如，我们可以看到在内核许多地方调用 ZwOpenKey 和 ZwQueryValueKey 来打开注册表键或读取注册表值。

最后值得一提的是，配置管理器提供了注册表键的变化通知机制。应用程序通过调用 NtNotifyChangeKey 或 NtNotifyChangeMultipleKeys 系统服务，可以监视一个或多个注册表键的创建、删除和修改动作。实现注册表键变化通知机制的关键在于，每个键对象都有一个类型为 CM_NOTIFY_BLOCK 的通知块成员，它描述了一个键对象的哪些事件以何种方式被通知到注册方。由于配置管理器提供了这种变化通知能力，因而对于想要监视注册表行为的应用程序，它们无须频繁地检查注册表来判断感兴趣的键是否已被修改。这对于一些安全保护或者注册表行为分析等程序有显著的意义。

2.5.3 事件追踪（ETW）

Windows 提供了统一的跟踪和记录事件的机制，称为 ETW（Event Tracing for Windows）。用户模式应用程序和内核模式驱动程序都可以使用 ETW 来记录事件。ETW 是直接由内核支持的事件记录机制，在它的框架结构中，共有三种组件：

- 控制器（controller）。负责启动、停止或配置事件记录会话。
- 提供者（provider）。负责向 ETW 注册自己的事件类，并接受控制器的命令，以便启动或者停止它们所负责的事件类的记录过程。
- 消费者（consumer）。负责有针对性地读取它们想要的事件数据，选择一个或多个记录会话。它们既可以实时地接收 ETW 缓冲区中的数据，也可以接收日志文件中的事件数据。

事件追踪能力对于诊断系统的性能问题尤为重要，因而 Windows 内置了一个内核日志记录器（kernel logger）作为 ETW 提供者，专门用于记录内核和核心驱动程序的事件。此内核日志记录器是由 WMI（Windows Management Instrumentation，Windows 管理规范）设备驱动程序实现的，也是内核模块 ntoskrnl.exe 的一部分。WMI 驱动程序的名称为“WMIxWDM”，它是在 I/O 管理器初始化过程中调用 WMI 组件的初始化函数（WMIInitialize）而创建的。它除了实现内核日志记录器的功能，也管理用户模式 ETW

事件类型的注册工作。由于它的实现形式是驱动程序，因此，其他的内核例程或设备驱动程序可以通过 I/O 接口与它通信。WRK 包含了 WMI 驱动程序的代码，其初始化例程为 WmipDriverEntry，位于 base\ntos\wmi\wmi.c 文件中。有关该驱动程序的 I/O 请求处理，请参考 base\ntos\wmi 目录下的源代码文件。

值得一提的是，尽管 Windows 早期版本已经提供了像事件管理器（event manager）和性能计数器（performance counter）这些设施来报告错误或指示关键的性能数据，但这些设施缺乏统一的编程接口，并且都不具备良好的扩展能力。随着 Windows 系统越来越复杂，以及各种应用程序都有事件报告和性能分析的需求，Windows 额外提供了一个新的事件管理机制，即 Windows 管理规范（WMI）。WMI 是工业标准 WBEM（Web-Based Enterprise Management）的一个实现。WBEM 标准包含了一套针对企业系统的数据采集和数据管理设施的设计方案，它具有很好的灵活性和可扩展性。自 Windows NT 4.0 (SP4)开始的 Windows 版本均支持 WMI，在 WRK 中我们可以看到 WMI 的内核模式驱动程序 WMIxWDM 的实现代码。根据上面的讨论，我们知道，ETW 的内核日志记录器实际上也是 WMI 设施的一部分。本书不打算讨论 WMI，有关 WMI 的更多信息，请参考 MSDN 中的相关资料[MSDN-WMI]，下面的讨论仅针对内核日志记录器。

内核日志记录器是一个事件提供者，它有一个预定义的 GUID，即内核变量 SystemTraceControlGuid。内核日志记录器支持多种事件类，它采用标志位（flag）来指示是否记录某一类型的事件。而且，它进一步将这些标志位分成了 8 个组，每个组使用一个掩码（29 位）来描述。下面是相关的定义（参见 public\internal\base\inc\ntperf.h 文件）：

```
#define PERF_MASK_INDEX          (0xe0000000)
#define PERF_MASK_GROUP          (~PERF_MASK_INDEX)

#define PERF_NUM_MASKS       8
typedef ULONG PERFINFO_MASK;

typedef struct _PERFINFO_GROUPMASK {
    ULONG Masks[PERF_NUM_MASKS];
} PERFINFO_GROUPMASK, *PPERFINFO_GROUPMASK;
```

当控制器程序指示内核日志记录器记录某些类型的内核事件时，它们需要构造一个 PERFINFO_GROUPMASK 对象，将需要记录的标志位置上。PERFINFO_GROUPMASK 的 Masks 数组包含 8 个 ULONG 成员，每个 ULONG 的最高 3 位（即 29~31 位）是组的索引（0~7），低 29 位（即 0~28 位）为组内的标志位。内核日志记录器的标志位安排是由 Windows 系统定义的，并且允许扩展。Windows Server 2003 仅仅使用了一部分标志位，参见 WRK 的 public\internal\base\inc\ntwmi.h 文件中的 PERF_<*XXX*>宏定义。例如，PERF_CONTEXT_SWITCH 的定义如下：

```
#define PERF_CONTEXT_SWITCH  0x20000004   // Context Switch
```

这说明处理器线程环境切换的事件标志位是第 2 个组（组索引为 1）的低 3 位（0x00000004）。

在 Windows 中，系统的全局组掩码是由全局变量 PerfGlobalGroupMask 定义的。以环境切换事件为例，当 PERF_CONTEXT_SWITCH 标志位被置上时，线程切换时就会记录一个 CSwitch 事件。这发生在 SwapContext 函数中，参见 base\ntos\ke\i386\ctxswap.asm 文件的 766~775 行代码。进一步参考 WmiTraceContextSwap 函数的代码（位于 base\ntos\wmi\callouts.c 文件中），可以看到 CSwitch 事件的记录过程。

缓冲区管理是 WMI 驱动程序的重要职责之一。WMI 驱动程序为环境切换事件定义了一个缓冲区数组（WmipContextSwapProcessorBuffers 变量），让每个处理器使用它自己的缓冲区，从而避免缓冲区冲突。一旦属于某个处理器的当前缓冲区满了（不足以再存放一个 CSwitch 事件），则交由 WMI 刷新该缓冲区，下次记录新的 CSwitch 事件时重新申请一个新的缓冲区作为当前缓冲区。WMI 为每个记录会话管理缓冲区，它在创建记录会话时根据控制器的指示，创建合理数量的缓冲区。读者可参考 base\ntos\wmi\tracesup.c 文件中 WmipStartLogger 和 WmipAllocateTraceBufferPool 函数的代码。

现在我们来看内核日志记录器的控制器和消费者。Microsoft 提供了一个性能工具 xperf，它既是一个控制器，也是一个消费者。Xperf 利用 I/O 接口（NtDeviceIoControlFile 函数）与 WMI 驱动程序进行通信，public\internal\base\inc\wmiumkm.h 文件中包含了 IOCTL_WMI_<*XXX*>次功能码的定义。Windows SDK 提供了事件追踪 API（位于 advapi32.dll 模块中），因而用户模式应用程序可以很方便地操纵和控制 WMI 驱动程序，MSDN 提供了一段例子代码来说明如何配置和启动内核日志记录器[MSDN-KL]。Xperf 直接在 ntdll.dll 的 ETW 接口上工作，并没有使用 advapi32.dll 中的事件追踪 API。

尽管我们在 WRK 中看到了 ETW 和 WMI 的完整实现代码，但是限于内核日志记录器仅仅提供了有限的事件类型，并且缺乏成熟的分析工具，因此，在 Windows Server 2003 及以前版本的 Windows 系统中，ETW 和 WMI 并没有得到广泛的使用。然而，从 Windows Vista 开始，内核日志记录器提供了大量内核事件信息，涉及线程、进程、内存管理、同步机制、文件 I/O 和磁盘 I/O 等系统的各个方面，并且在许多关键事件上可以导出相应的调用栈（包括内核栈和用户栈），而且 xperf 工具可以自由下载，其分析功能更加丰富[MSDN-WPA]。因此，在更新版本的 Windows 系统环境中，利用 ETW 和 WMI 作为性能分析工具，有希望解决大量原先非常棘手或根本无从解决的问题。

另外值得一提的是，Microsoft 提供的 xperf 工具虽然不能直接在 Windows XP/Server 2003 上安装，但实际上它的可执行模块可以在这些系统上运行，只不过受限于内核日

志记录器的能力，它并不能捕获到像在 Windows Vista 以后的系统中那么多的事件。WMI 驱动程序的接口规范以及内核日志记录器支持的事件类型在不同的 Windows 系统中保持一致，这是 xperf 仍然可以在 Windows XP/Server 2003 上运行的基础。但是，在 Windows Vista 及以后的系统中，WMI 驱动程序和内核日志记录器的内部实现有相应调整和变化。

本书 9.3 节将介绍如何利用 ETW 和 xperf 来分析系统的性能。读者如果想了解应用程序或内核代码如何与设备驱动程序进行通信，请参考本书第 6 章的内容。

2.5.4 安全性管理

作为一个现代操作系统，Windows 有严格的安全模型，它既实现了以对象为基础的自主访问控制（discretionary access control），又实现了系统级的强制访问控制（mandatory access control）。在自主访问控制模型中，对象的所有者授权或拒绝哪些人可以访问该对象。而且，Windows 还考虑到了对象所有者丢失的情形，即，当由于某些原因对象的所有者不再有效时，特权用户可以将对象的所有权接管过来，从而可以访问该对象，或授权其他的用户访问它。这在某些情形下是有意义的，比如若一个企业的员工离职了，系统管理员仍然可以访问到该员工的文档，即使这些文档只有该员工的账户才可以访问。基于对象的访问控制并不能保护系统中的所有操作，因为有些操作并没有与一个特定的对象联系在一起，例如系统停机（shutdown）、备份文件和目录、交互式登录到系统中等；甚至有些操作针对整个系统，并不与特定用户账号关联，例如账户的口令策略（通常不允许空口令）、允许或禁止匿名连接等。

在介绍 Windows 如何实现自主访问控制和强制访问控制以前，我们首先来看一下 Windows 操作系统中涉及安全性管理的核心组件，如图 2.12 所示。本章前面 2.2 节曾经提到过这里的三个组件：winlogon、SRM 和 lsass，winlogon 和 lsass 是两个用户模式进程，而 SRM 是 Windows 执行体中的组件。

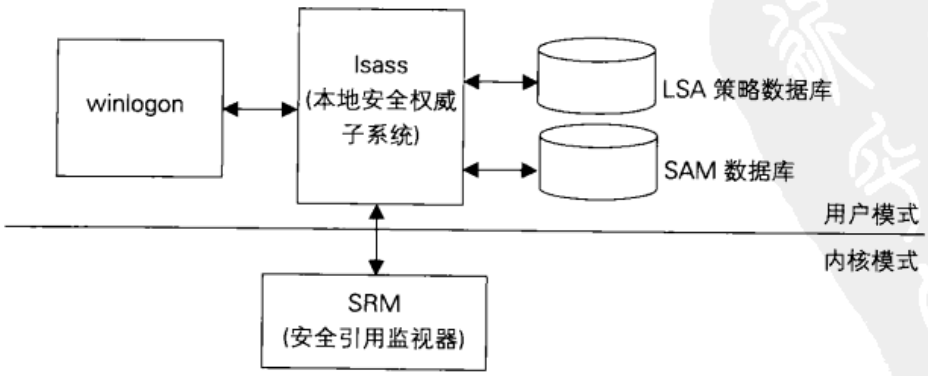


图 2.12 Windows 操作系统的简化安全模型

从安全的角度来看，图 2.12 中所示的安全组件的职责如下所述：

- SRM（安全引用监视器），负责执行对象的安全访问检查、管理用户特权、生成安全审计消息，并且定义了访问令牌数据结构来表示一个安全环境。
- Winlogon，负责响应 SAS（安全注意序列），以及管理交互式登录会话。当用户登录到系统中时，winlogon 创建一个初始进程，并进一步由它创建外壳（shell）进程。
- Lsass（本地安全权威子系统），负责本地系统的安全策略，同时，它也认证用户的身份，以及将安全审计消息发送到系统的事件日志中。
- SAM（安全账户管理器）数据库，包含了本地用户和用户组，以及它们的口令和其他属性。它位于注册表的 HKLM\SAM 下面。由于 HKLM\SAM 键只允许本地系统账户访问，所以，除非用户在 Local System 账户下运行 regedit.exe 工具，否则无法访问 HKLM\SAM 子树。
- LSA 策略数据库，包含了有关当前系统的一些信息，譬如谁允许访问系统以及如何访问（交互式登录、网络登录或者以服务方式登录）；分配给谁哪些特权；安全审计如何进行等。如同 SAM 数据库一样，LSA 策略数据库也存储在注册表中，位于 HKLM\SECURITY 下面。同样地，除了 Local System 账户以外的其他账户均无法访问 HKLM\SECURITY 子树。

简而言之，在 Windows 的安全模型中，winlogon 负责系统登录，包括对用户身份的认证；lsass 负责管理系统本地安全策略，并且将这些策略通知到内核中的 SRM。在内核中，SRM 负责实现基于对象的访问控制以及系统全局安全策略的实施。WRK 包含了 SRM 的代码，位于 base\ntos\se 目录下面。在此模型中，lsass 和 SRM 对于系统的安全性至关重要，一旦这两个组件被恶意代码修改或侵入，则系统的安全防线将不复存在。Windows 对这两个组件有特殊的保护，普通应用程序无法与它们打交道，它们相互之间通过 LPC 进行通信。它们的 LPC 连接在系统初始化时建立，而且，一旦其双向 LPC 连接建立起来，它们的 LPC 端口便不再接受任何其他的连接请求，因而其他程序与它们无法建立 LPC 连接。关于它们的 LPC 连接的建立过程，请参考 8.2.4 节介绍的 LPC 应用例子。

Windows 的自主访问模型与对象管理器集成在一起，每一种对象类型都定义了一个 Security 方法，该方法返回一个对象的安全信息。线程在访问一个对象以前必须先打开这个对象，并获得一个指向该对象的句柄。在打开对象的操作中，对象管理器调用 SRM 的函数，根据调用线程的安全凭证、在打开操作中请求的访问类型（比如读、写、删除等），

以及该类型对象的 Security 方法提供的对象安全信息，来决定此打开操作是否允许。如果可以打开该对象，那么，对象管理器在线程的进程句柄表中创建一个句柄，记录下该对象以及它所请求的访问类型。所以，线程在成功打开对象后获得一个指向该对象的句柄。以后，当该线程访问此对象时，它传递该对象的句柄，而对象管理器将该线程所请求的访问操作与该对象被打开时所获得的访问类型进行比较，如果当前的访问操作允许，则安全检查通过，否则此次对象访问失败。

在这个模型中有两点值得一提。第一，每个线程都有一个安全环境，其中最重要的信息是一个访问令牌，代表了该线程的用户的一次登录；每个对象都有一个自主访问控制列表（ACL），指明了允许谁以何种方式访问该对象，而拒绝哪些用户以何种方式访问它。为了访问一个对象，线程可以不使用它所属进程的安全环境，而是以其他账户身份来运行的安全环境，这称为模仿（impersonation）。第二，同一个进程中的其他线程也可以利用已经得到的句柄来访问该对象。同样地，对象管理器使用调用线程的安全环境和它所请求的访问操作，对照该对象被打开时获得的访问类型进行检查，以决定是否允许该操作。

下面我们来看对象管理器如何使用 SRM 的安全检查功能。当一个进程打开一个对象时，对象管理器在名字空间中查找到目标对象以后，但是在返回句柄给调用者以前，它调用 ObCheckObjectAccess 函数检查访问许可。ObCheckObjectAccess 是对象管理器的函数，它将对象管理器与 SRM 的安全监视机制连接起来。该函数的代码位于 base\ntos\ob\obse.c 文件的 387~564 行。它首先调用 ObGetObjectSecurity 函数以获得目标对象的 SD（安全描述符），数据类型为 SECURITY_DESCRIPTOR。对象的安全描述符包含了对象的所有者的 SID（安全标识符）、自主访问控制列表（DACL）和系统访问控制列表（SACL），以及其他一些描述对象安全特征的属性。这里的自主访问控制列表规定了谁可以或不可以以何种方式访问该对象；系统访问控制列表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。ObGetObjectSecurity 函数的代码也位于 obse.c 文件中，它利用对象头的信息定位到目标对象的类型对象，然后调用类型对象的 SecurityProcedure 函数成员，由它填充一个自包含的安全描述符缓冲区。

对于每一种类型的对象，系统在创建它的类型对象时，可以指定该类对象的 Security 方法，即 SecurityProcedure 函数成员；如果在创建类型对象时不指定该函数成员，那么，对象管理器使用一个默认的安全函数 SeDefaultObjectMethod，参见 ObCreateObjectType 函数的代码。例如，注册表键对象的 Security 方法为 CmpSecurityMethod，文件对象和设备对象的 Security 方法为 IopGetSetSecurityObject，而驱动程序对象使用默认的安全函数。

接着，在获得了对象的安全描述符以后，ObCheckObjectAccess 函数锁住当前的安全环境，并调用 SeAccessCheck 函数执行安全访问许可检查。SeAccessCheck 是 SRM 的一个内核接口函数，真正执行检查任务是在 SepAccessCheck 函数中。它们的代码位于 base\ntos\se\accessck.c 文件中。

线程的安全环境是一个 SECURITY_SUBJECT_CONTEXT 类型的对象，它包含了线程的主令牌（PrimaryToken 成员）、模仿级别、进程审计 ID（直接用进程对象的地址来表示），以及一个可选的客户令牌（ClientToken）。当线程模仿一个客户时，客户令牌不为空，指向被模仿客户的令牌，在这种情况下，SRM 使用客户令牌来检查访问许可。

令牌（token）代表了 SRM 中用到的主体，它描述了一个用户的一次登录，由 winlogon 进程在认证了用户身份以后创建。同一个用户会话中运行的进程都使用同样的令牌。令牌的数据结构为 TOKEN，它包含了令牌 ID、认证 ID、用户账户的 SID 和所属组的 SID、一组与该令牌关联的特权，以及其他一些用于各种安全操作的属性。关于 TOKEN 结构的详细定义和说明可参考 base\ntos\se\tokenp.h 文件中的定义和注释，这里不再一一解释。

理解了线程的安全环境和对象的安全描述符以后，我们来看 SepAccessCheck 函数的实现。SepAccessCheck 函数首先检查调用者是否有 SeSecurityPrivilege 特权，然后从对象的安全描述符中获得 DACL，如果对象没有 DACL，则总是允许访问，返回成功。接下来，如果请求 WRITE_OWNER 操作，则检查调用者是否有 SeTakeOwnershipPrivilege 特权。然后，如果 DACL 为空，则拒绝所有访问。接下来，若调用者请求 MAXIMUM_ALLOWED 操作或者想要检查所有的对象和子对象，则调用 SepMaximumAccessCheck 函数执行一个慢速的检查过程；否则，调用 SepNormalAccessCheck 函数执行正常的许可检查。SepNormalAccessCheck 函数的实现较为直截了当，对 DACL 中的项逐一检查是否允许或拒绝此次访问操作，其代码也在 accessck.c 文件中。

综上所述，当一个线程打开一个对象时所执行的安全检查过程如图 2.13 所示。

一旦线程成功地打开了一个对象，以后当它在该对象上执行操作时，将使用句柄来引用该对象，而对象管理器或其他内核组件在将句柄解析成对象时仍然要检查调用者所请求的操作是否已被授权或拒绝。例如，ObReferenceObjectByHandle 函数使用宏 SeComputeDeniedAccesses 来判断要请求的访问是否被拒绝；ObReferenceFileObjectForWrite 函数使用宏 SeComputeGrantedAccesses 来判断所请求的文件操作是否已被授权。此外，I/O 管理器也使用同样的方法来检查它所接到的访问操作。

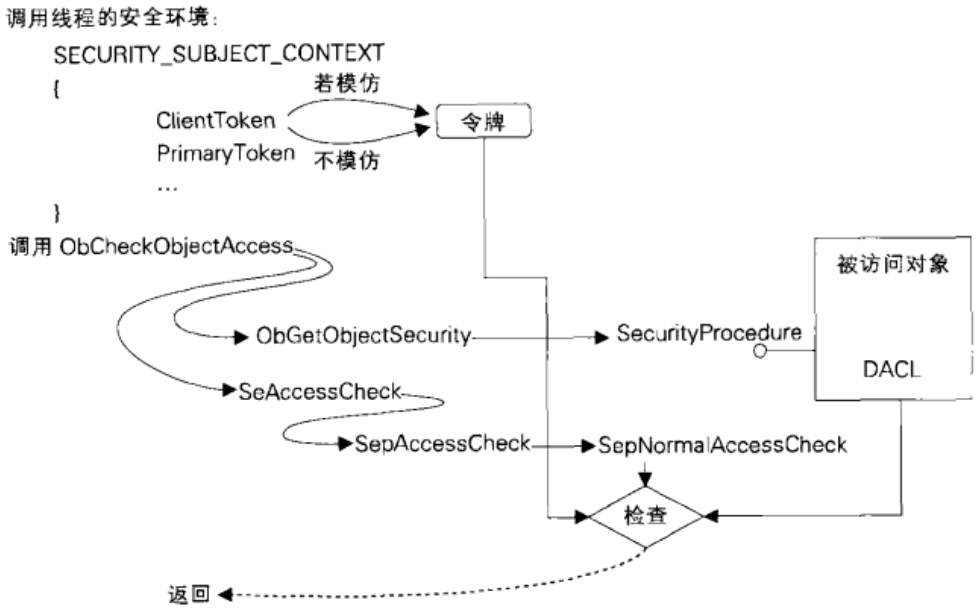


图 2.13 对象管理器打开一个对象时 SRM 所执行的安全检查

至此，我们看到了 SRM 中自主访问控制的实现过程。SRM 的强制访问控制是通过特权检查来实施的，SePrivilegeCheck 函数是 SRM 中用于特权检查的内核接口函数。SePrivilegeCheck 调用 SepPrivilegeCheck 函数执行实际的特权检查。SepPrivilegeCheck 函数的代码位于 base\ntos\se\privileg.c 文件中，其参数 Token 代表了当前线程的安全环境，如同 SepAccessCheck 函数也需要代表调用线程的令牌一样；参数 RequiredPrivileges 和 RequiredPrivilegeCount 合起来指明了调用线程要请求的一组特权；参数 PrivilegeSetControl 定义了如何请求这些特权，即，请求所有指定的特权或其中之一。

在 Windows 中，特权是由 LUID 对象来标识的，LUID 代表一个本地唯一标识符（Locally Unique Identifier），由两个 LONG 成员构成，因此在 32 位系统中 LUID 是一个 64 位的整数。每一个特权都附带一些属性，两者结合起来构成了 LUID_AND_ATTRIBUTES 数据结构，而属性是一个无符号长整数类型。以下是 LUID_AND_ATTRIBUTES 及一组属性的定义：

```
typedef struct _LUID_AND_ATTRIBUTES {
   LUID Luid;
   ULONG Attributes;
   } LUID_AND_ATTRIBUTES, * PLUID_AND_ATTRIBUTES;

#define SE_PRIVILEGE_ENABLED_BY_DEFAULT    (0x00000001L)
#define SE_PRIVILEGE_ENABLED               (0x00000002L)
#define SE_PRIVILEGE_REMOVED               (0x00000004L)
#define SE_PRIVILEGE_USED_FOR_ACCESS       (0x80000000L)
```

在令牌 TOKEN 数据结构的定义中，有一个 Privileges 成员，代表该令牌账户的所有特

权，因此，SepPrivilegeCheck 函数的实现是，用一个二重循环，对调用线程所请求检查的每一个特权，遍历 Token 参数中的 Privileges 数组成员，看是否能匹配上。最后检查总共匹配了多少特权，是否所请求的特权均已匹配上，或至少匹配了一个。检查流程如图 2.14 所示。

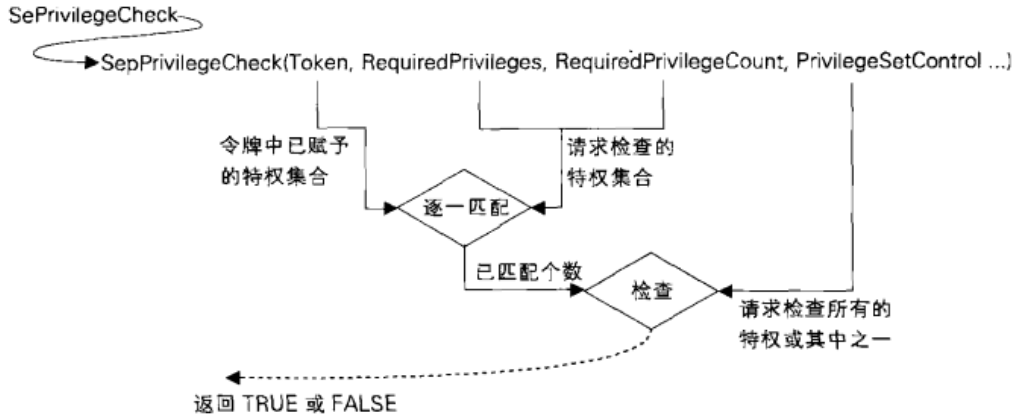


图 2.14 SePrivilegeCheck 函数的特权检查过程

Windows 内核中定义了一组特权，即类型为 LUID 的 Se<*Xxx*>全局变量，其定义位于 base\ntos\se\seglobal.c 文件中，它们的初始化由该文件中的 SepVariableInitialization 函数来完成。这组全局特权为：

```
LUID SeCreateTokenPrivilege;                // 创建令牌对象
LUID SeAssignPrimaryTokenPrivilege;         // 替换一个进程的令牌
LUID SeLockMemoryPrivilege;                 // 锁住内存页面
LUID SeIncreaseQuotaPrivilege;              // 增加一个进程的内存配额
LUID SeUnsolicitedInputPrivilege;           // 已过时，不再使用
LUID SeTcbPrivilege;                        // 操作系统使用
LUID SeSecurityPrivilege;                   // 管理审计和安全日志
LUID SeTakeOwnershipPrivilege;              // 接管文件和其他对象的所有权
LUID SeLoadDriverPrivilege;                 // 加载和卸载设备驱动程序
LUID SeCreatePagefilePrivilege;             // 创建页面文件
LUID SeIncreaseBasePriorityPrivilege;       // 增加调度优先级
LUID SeSystemProfilePrivilege;              // 系统性能分析
LUID SeSystemtimePrivilege;                 // 改变系统时间
LUID SeProfileSingleProcessPrivilege;       // 对单个进程进行性能分析
LUID SeCreatePermanentPrivilege;            // 创建永久对象
LUID SeBackupPrivilege;                     // 备份文件和目录
LUID SeRestorePrivilege;                    // 恢复文件和目录
LUID SeShutdownPrivilege;                   // 系统停机
LUID SeDebugPrivilege;                      // 调试程序
LUID SeAuditPrivilege;                      // 安全审计
LUID SeSystemEnvironmentPrivilege;          // 修改系统的固件环境变量
LUID SeChangeNotifyPrivilege;               // （名字空间中）变化通知，也用于绕过穿越检查
LUID SeRemoteShutdownPrivilege;             // 远程停机
LUID SeUndockPrivilege;                     // 从插接站（docking station）移除计算机
LUID SeSyncAgentPrivilege;                  // 同步代理（目录服务）
LUID SeEnableDelegationPrivilege;           // 允许计算机账户或用户账户被用于委托
LUID SeManageVolumePrivilege;               // 管理卷
```

```
LUID SeImpersonatePrivilege;                    // 模仿
LUID SeCreateGlobalPrivilege;                   // 创建全局对象
```

对于每一种特权，当相关联的操作（即需要此特权才能进行的操作）在内核中被适当的组件激发时，应通过 SePrivilegeCheck 或它的包装函数 SeSinglePrivilegeCheck，来检查调用线程是否具有相应的特权。例如，SepCreateToken 函数在把令牌对象插入到进程的句柄表之前调用 SeSinglePrivilegeCheck 函数来检查当前线程是否具有 SeCreateTokenPrivilege 特权；IopParseDevice 函数在打开一个设备对象时通过 IopCheckBackupRestorePrivilege 函数（它调用 SePrivilegeCheck）来检查调用线程是否具有 SeBackupPrivilege 或 SeRestorePrivilege 特权。读者可以参考这些函数的代码，看一看特权检查的用法。另外值得一提的是 SeMakeSystemToken 函数，它在系统的初始化阶段被调用，它创建了 System 进程的令牌，以本地系统账号为 SID 和组 SID。在 SeMakeSystemToken 函数中我们可以看到一个进程令牌的特权是如何被初始化的；而在另两个函数 SeMakeAnonymousLogonToken 和 SeMakeAnonymousLogonTokenNoEveryone 中，则可以看到它们创建的令牌没有任何特权。

Windows 安全性管理是一个复杂的话题，其安全模型涉及系统的方方面面，而且模型中的许多组件和扩展结构是在用户模式下运行的。本节仅仅讨论了 Windows 安全性管理的核心部分，即内核中的安全引用监视器，包括它的对象访问和特权操作的检查过程。在 Windows 内核中，与安全相关的函数以“Se”作为前缀，有一些安全函数还存在对应的系统服务。这些系统服务函数的名称以“Nt”作为前缀，后面部分与“Se”函数相同。在 WRK 的 public\sdk\inc\ntseapi.h 文件中包含了与安全有关的系统服务函数的原型声明，而 base\ntos\inc\se.h 文件中包含了内核安全函数的原型声明，包括本节前文提到过的众多以“Se”为前缀的函数（注意，以 Sep 为前缀的函数是 SRM 的内部函数）。

2.6 Windows 引导过程

Windows 内核中的各个组件和各种机制在起作用以前，必须首先被初始化。此初始化工作是在系统引导时完成的。当用户打开计算机的电源开关时，计算机便开始运行，但操作系统并不立即获得控制权，而是 BIOS 代码首先获得控制，它执行必要的硬件检测工作，并允许用户通过一些功能键来配置当前系统中的硬件设置，甚至诊断硬件问题，然后才将控制权交给操作系统。这一节我们讨论 Windows 的引导过程，首先介绍 Windows 获得控制以及加载内核模块的过程，然后描述 Windows 内核的初始化，最后介绍 Windows 的会话创建和用户登录过程。

2.6.1 内核加载

在 Intel x86 系统上，Windows 操作系统获得控制首先从硬盘的主引导记录（MBR，Master Boot Record）开始，Windows Setup 程序在安装 Windows 时填充 MBR（其他的磁盘管理程序也可能填充 MBR）。MBR 包含代码和数据，其代码称为引导代码，在系统引导时首先获得控制；MBR 中的数据是一张分区表，指定了每个分区在磁盘上的位置和大小，以及分区的类型。当 MBR 中的引导代码被执行时，它检查分区表中的每一个分区，若找到一个已被标记为可引导的分区（称为引导分区），则将该分区的第一个扇区（称为引导扇区）读到内存中。由于分区表包含了每一个分区的磁盘位置，所以，引导扇区的位置很容易被确定。然后 MBR 的代码将控制权交给引导扇区中的代码。

Windows Setup 程序在确定了要将 Windows 系统安装到哪个分区中以后，除了可能会写入 MBR 以外，还会写入该分区的引导扇区。所以，严格意义上讲，Windows 操作系统的真正入口点应该是引导扇区中的代码。引导分区必须被格式化成 Windows 所支持的文件系统，典型的文件系统格式是 NTFS 和 FAT，其中 NTFS 是 Windows NT 的原生文件系统，而 FAT 则是从 MS-DOS 时代继承和发展过来的。关于 Windows 文件系统的进一步介绍，请参考本书第 7 章。

引导扇区中的代码随硬盘文件系统格式的不同而有所不同，其职责是，给 Windows 提供有关该硬盘上卷的结构和格式方面的信息，并且从该卷的根目录中读入 Windows 的加载程序，即 ntldr 文件；然后将控制权交给 ntldr 的入口函数。为了能够从根目录中读入加载程序，引导扇区包含了能理解文件系统结构和读取文件的代码，这通常只是文件系统极其简单的一部分功能，而并非完整的实现。尽管引导扇区的职责相对简单，但是单个扇区（512 B）的代码和数据往往不足以完成其功能，为此，Windows 的做法是，让引导扇区中的代码读入其他扇区的数据，然后跳转到下一个扇区的代码区。这样就可以不受单个引导扇区长度的限制，这种做法相当于将第一个引导扇区当做一个加载器（loader），而真正完成引导扇区功能的扇区随后被加载进来并执行。这一过程对于 MBR 是透明的，从而保持良好的兼容性。

Intel x86 处理器支持实模式和保护模式，在实模式下，处理器的寄存器都是 16 位的，而且不支持虚拟地址转译，只能访问物理内存空间中最低的 1 MB 内存。计算机系统的 BIOS 工作在实模式下，并且，当 ntldr 获得控制权时，处理器仍然在实模式下运行。Ntldr 文件实际上是由两部分组成的：第一部分是实模式代码，即首先获得控制的代码区；第二部分是一个标准的 Windows 可执行二进制文件（PE 文件格式，请参考本章前文 2.2 节中插入的关于“PE 文件格式”的介绍），在 ntldr 中这部分被称为 os loader。

Ntldr 的实模式代码首先获得控制，它的任务是，完成需在 16 位模式下执行的初始化工作，例如清除键盘缓冲区，然后为切换到保护模式做好基本的环境准备，之后将处理器切换到保护模式（32 位模式）下，这样它就可以访问整个 32 位地址空间了。最后它将控制权交给 os loader。因此，ntldr 中的 os loader 是 Windows 真正的 32 位入口程序。

Os loader 刚接获控制时，处理器虽然已经工作在保护模式下，但是它的虚拟地址转译机制尚未开启，所以，处理器仍然直接使用物理地址。Os loader 首先做的工作是把物理内存管起来，用一个内存描述符（memory descriptor）数组把每一段内存的大小和用途记录下来，然后构造页目录和页表，使得 16 MB 以下的内存能够通过页面映射（paging）机制进行访问，再设置好页目录寄存器，并打开页面映射机制。之后，os loader 继续执行其他的初始化工作，包括 I/O 设备的初始化等。如果它还需要调用 BIOS 中的服务（比如中断 13h、中断 15h 等），则必须保护好保护模式下的设置，并暂时切换回到实模式，待服务完成以后再切换到保护模式，并恢复设置。

接下来，os loader 从系统分区（即引导分区）的根目录下读入 boot.ini 文件。注意，os loader 包含了读取当前文件系统的代码，它能够读取 NTFS 文件系统的子目录。然后，os loader 清除屏幕，并检查在系统分区的根目录下是否有一个有效的 hiberfil.sys 文件，如果存在的话，这一次引导过程转移到休眠系统的恢复过程。因此，os loader 将控制权交给一段能恢复休眠系统的内核代码。

如果当前这次引导不是休眠恢复，那么，os loader 解析 boot.ini 文件，如果该文件中有多个引导选项，则 os loader 显示一个引导选择菜单；如果 boot.ini 文件中只包含一个引导选项，那么，此菜单不显示，而是立即应用该引导选项。

Windows 的引导选项可以用来指示当前这次引导的各种参数，包括内核模块的文件名称、HAL 的文件名称、CPU 参数、各种内存参数、调试参数，等等。关于这些引导选项的全面列表和介绍，读者可参考[MSDN-BOOT]。接下来 os loader 加载并执行 NTDETECT.COM 程序，这是一个 16 位实模式程序，它利用系统的 BIOS 来查询系统的基本设备和配置信息，包括系统的日期和时间、总线的类型、磁盘的信息、输入/输出的接口信息等。这些信息被收集起来，在引导过程的后期被存放到注册表 HKLM\HARDWARE\DESCRIPTION 键的下面。

然后，os loader 加载内核模块映像文件，默认为 ntoskrnl.exe，以及 HAL 映像文件，默认为 hal.dll。再加载注册表的 SYSTEM 储巢，即\WINDOWS\system32\config\system 文件。通过检查 SYSTEM 储巢中的设置信息，它可以知道哪些设备驱动程序必须被加载进来，即被标记为“引导-启动”（SERVICE_BOOT_START）的设备驱动程序。然后它加载所有这

些被标记为“引导-启动”的设备驱动程序，以及访问系统目录所必需的文件系统驱动程序。注意，在此之前 os loader 也可以访问系统分区，但并非通过文件系统驱动程序。

至此，引导系统所需的模块（包括内核映像、HAL，以及必要的设备驱动程序）都已经被加载到内存中。而且，在此过程中 os loader 也已经构造了一个参数块，记录下了这次引导过程中加载器所获得的各种参数信息。参数块的类型为 LOADER_PARAMETER_BLOCK，Windows 内核在初始化过程中将会用到这些参数信息。WRK 中包含有它的定义，如下（见 public\internal\base\inc\arc.h 文件）：

```
typedef struct _LOADER_PARAMETER_BLOCK {
    LIST_ENTRY LoadOrderListHead;                    // 加载的模块链表，每个元素都为
                                                     // KLDR_DATA_TABLE_ENTRY
    LIST_ENTRY MemoryDescriptorListHead;             // 内存描述符链表，每个元素都为
                                                     // MEMORY_ALLOCATION_DESCRIPTOR
    LIST_ENTRY BootDriverListHead;                   // 引导驱动程序链表，每个元素都为
                                                     // BOOT_DRIVER_LIST_ENTRY
    ULONG_PTR KernelStack;                           // 内核栈顶
    ULONG_PTR Prcb;                                  // 进程环境，指向一个进程控制块
    ULONG_PTR Process;                               // 初始进程，EPROCESS
    ULONG_PTR Thread;                                // 初始线程，ETHREAD
    ULONG RegistryLength;                            // 系统储巢的长度
    PVOID RegistryBase;                              // 系统储巢的基地址
    PCONFIGURATION_COMPONENT_DATA ConfigurationRoot;
                                                     // 配置树，包含 ISA、磁盘
                                                     // 和 ACPI 的配置数据
    PCHAR ArcBootDeviceName;                         // 引导分区的 ARC 名称
    PCHAR ArcHalDeviceName;                          // 系统分区的 ARC 名称
    PCHAR NtBootPathName;                            // OS 目录的路径名称，比如“\Windows”
    PCHAR NtHalPathName;                             // OS 加载器的路径名称，比如“\”
    PCHAR LoadOptions;                               // 引导选项，来自 boot.ini
    PNLS_DATA_BLOCK NlsData;                         // 包含 ANSI 代码页、OEM 代码页
                                                     // 和 Unicode 码表
    PARC_DISK_INFORMATION ArcDiskInformation;        // 所有磁盘的签名结构
    PVOID OemFontFile;                               // OEM 字体文件
    struct _SETUP_LOADER_BLOCK *SetupLoaderBlock;    // 网络引导或文字模式安装引导
    PLOADER_PARAMETER_EXTENSION Extension;           // 扩展部分

    union {
        struct I386_LOADER_BLOCK {
            PVOID CommonDataArea;
            ULONG MachineType;                       // Temporary only
            ULONG VirtualBias;                       // 当前内核基地址相对于 2 GB 的偏移量
        } I386;
        // ALPHA_LOADER_BLOCK Alpha;
        // IA64_LOADER_BLOCK Ia64;
    } u;
} LOADER_PARAMETER_BLOCK, *PLOADER_PARAMETER_BLOCK;
```

由上述代码的注解可以看出，LOADER_PARAMETER_BLOCK 参数块中包含了有关这次引导的各种参数信息和系统配置，这里 ARC 名称是指符合 ARC（Advanced RISC Computing）命名规范的字符串，例如“multi(0)disk(0)rdisk(0)partition(1)”是指 0 号磁盘

控制器第一块硬盘上的第一个分区。注意，参数块中的绝大多数信息由 os loader 来填充，而在接下来的内核初始化过程中使用，但也有例外，比如有关线程和进程的信息需要在内核初始化过程中填充。

最后，os loader 将控制权交给内核模块的入口函数，该函数将不再返回，所以，接下来的引导过程由内核模块继续进行，引导扇区和系统加载器（ntldr 或 os loader）的使命已经完成。图 2.15 显示了以上介绍的引导步骤。

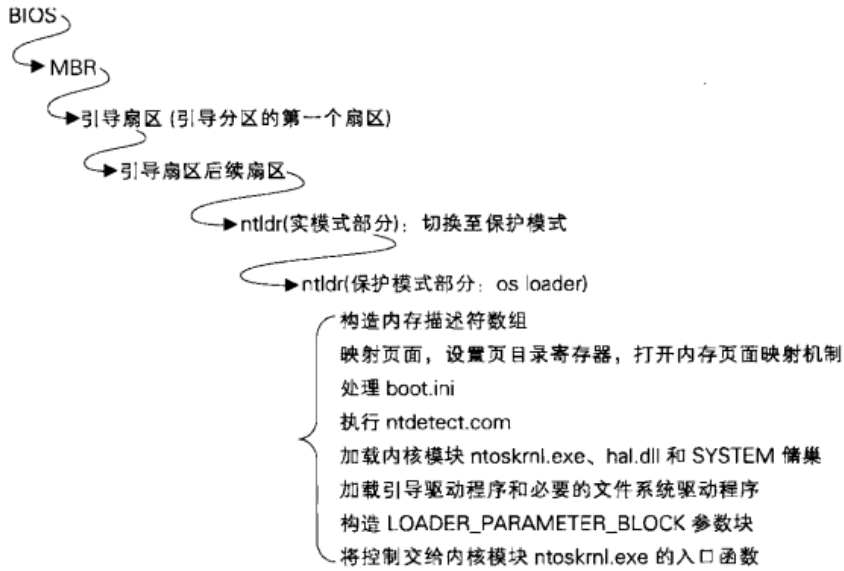


图 2.15 引导扇区和系统加载器的引导执行过程

2.6.2 内核初始化

我们已经看到，ntldr 构造了一个类型为 LOADER_PARAMETER_BLOCK 的参数块，把与系统初始化相关的参数信息包装到此结构中，然后将控制权传递给内核模块 ntoskrnl.exe 的入口函数。因此，内核的初始化从内核模块的入口函数开始，WRK 包含了内核初始化过程的绝大多数代码。此入口函数为 KiSystemStartup，它是一个汇编函数，位于 base\ntos\ke\i386\newsysbg.asm 文件中。

内核的初始化主要是内核各个组件的初始化，但由于这些内核组件之间有紧密的耦合关系，所以它们的初始化并不是简单地顺序执行初始化。为了解决在初始化过程中的相互依赖性问题，内核的初始化分两个阶段进行，称为阶段 0 和阶段 1。大多数内核组件的初始化函数相应地带有一个整数参数，以指明一次调用是阶段 0 初始化还是阶段 1 初始化，而有些组件的初始化函数通过检查一个全局变量 InitializationPhase 的值来判断当前处于哪个阶段。

我们首先来看阶段 0 初始化。阶段 0 初始化的目的是，将阶段 1 初始化所要用到的基本数据结构建立起来。在阶段 0 初始化过程中，中断被禁止，因此处理器可以顺序地执行自己的初始化逻辑。KiSystemStartup 函数首先初始化处理器的状态，包括调整它的 IDT，初始化 TSS（Task State Segment），以及构造 PCR（Processor Control Region）。然后，调用 HalInitializeProcessor 函数，为当前处理器初始化其 HAL 中的 PCR 和处理器间中断向量；接着调用 KiInitializeKernel 函数，执行内核初始化。最后，当前线程蜕变成一个空闲线程。

因此，KiInitializeKernel 函数是实际执行内核初始化的函数，其代码位于 base\ntos\ke\i386\kernlini.c 文件中。它的职责是：初始化内核数据结构、初始化空闲线程和进程对象，初始化 PCR，然后调用执行体初始化函数 ExpInitializeExecutive，最后返回。ExpInitializeExecutive 函数的代码位于 base\ntos\init\initos.c 文件中，它调用 HalInitSystem 以初始化 HAL，调用 ExInitSystem 以初始化执行体组件的各种数据结构，调用 MmInitSystem 以初始化内存管理器和内存池，调用 ObInitSystem 以初始化对象管理器，调用 SeInitSystem 以初始化安全子系统，调用 PsInitSystem 以初始化进程/线程管理器，调用 PpInitSystem 以初始化即插即用管理器，调用 DbgkInitialize 以初始化调试子系统。通常，这些执行体组件的阶段 0 初始化相对简单，以初始化组件的内部状态为主，因而经过阶段 0 初始化以后仅可以提供最基本的服务。

阶段 0 初始化完成以后，系统的线程调度器开始工作。特别值得一提的是，进程管理器在阶段 0 初始化过程（PspInitPhase0）中，除了初始化其内部的状态变量，它也为初始进程创建一个进程对象，并将其命名为“Idle”。另外，它还创建了“System”进程，以及一个系统线程，此系统线程的开始例程为 Phase1Initialization 函数。然而，此线程并不立即被执行，因为在阶段 0 初始化过程中中断是禁止的。

KiInitializeKernel 函数返回以后，KiSystemStartup 启动中断，将当前的中断请求级别（IRQL，Interrupt Request Level）降低到 DISPATCH_LEVEL，从而允许线程调度器选择新的线程。因此，阶段 0 初始化完成以后，阶段 1 初始化例程 Phase1Initialization 得以运行。注意，如果仔细观察 KiSystemStartup 汇编函数的代码，可以发现它在跳转到空闲循环 KiIdleLoop 以前，要经过一个屏障 KiBarrierWait。此屏障对于系统的第一个处理器并不起作用，而仅对后续的处理器起作用。后面我们在讨论其他处理器的初始化过程时将会看到此屏障的影响。

现在我们知道，阶段 1 初始化是在 System 进程的一个系统线程中运行的。Phase1Initialization 函数调用 Phase1InitializationDiscard 执行阶段 1 初始化，然后调用 MmZeroPageThread 函数，从而此线程蜕变成内存管理器的零页面线程（内存管理器中负

责在后台将空闲页面清零的辅助线程，参见 4.5 节关于物理内存管理的描述）。在介绍阶段 1 初始化的逻辑以前，我们先来看看在多处理器或多核系统上的初始化过程。如图 2.16 所示，当第一个处理器（也称为引导处理器，或 0 号处理器，或 P0）执行到阶段 1 初始化时，在 Phase1InitializationDiscard 的一个特定点上，它调用 KeStartAllProcessors 函数，以启动其他的处理器。这些处理器从 KiSystemStartup 函数开始执行。

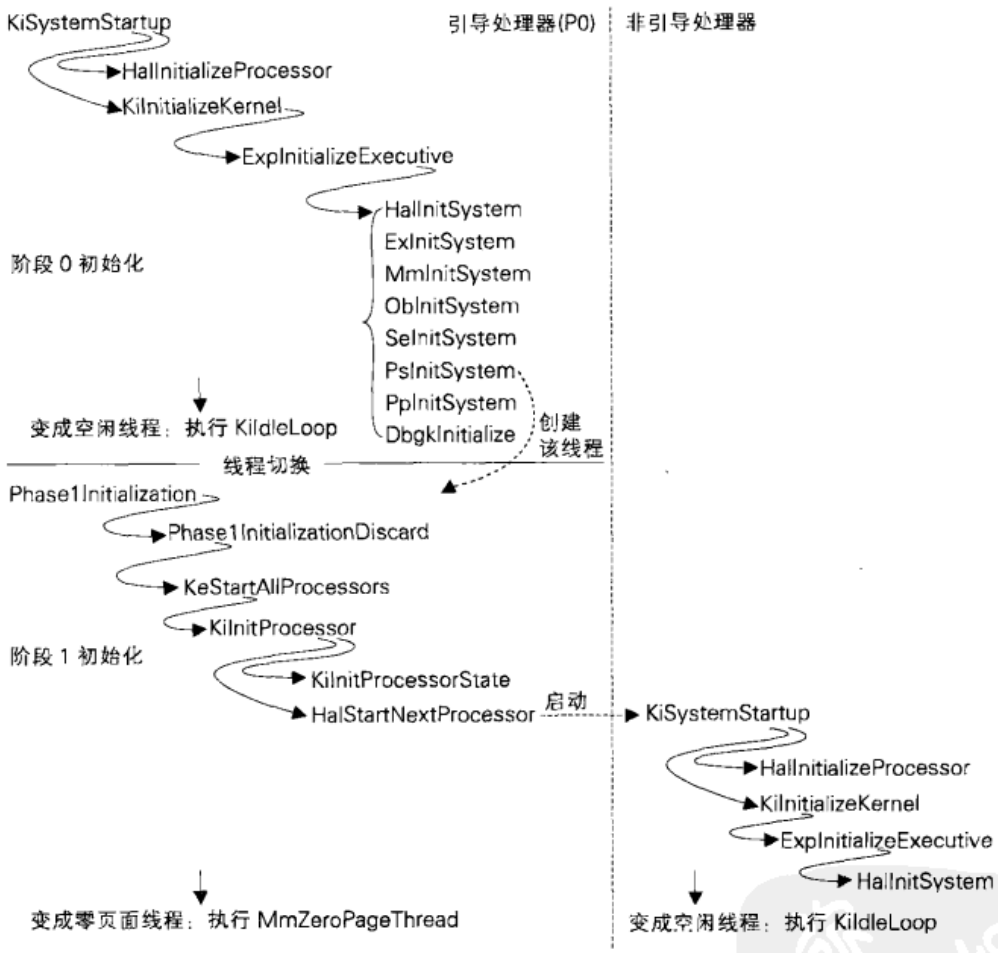


图 2.16 多处理器系统的内核初始化过程

KeStartAllProcessors 函数的代码位于 base\ntos\ke\i386\allproc.c 文件中，它设置好前文提到的位于 KiSystemStartup 函数结束处的屏障 KiBarrierWait，然后依次调用 KiInitProcessor 函数来启动每个处理器。KiInitProcessor 函数为每个处理器构造一份状态信息（KPROCESSOR_STATE 结构），然后调用 HalStartNextProcessor 函数启动该处理器。处理器状态信息是通过调用 KiInitProcessorState 函数来构造的，在该函数的代码中，我们可以看到，新处理器的起始指令地址（即 KPROCESSOR_STATE 结构的 ContextFrame.Eip

成员）为 KiSystemStartup 例程。KeStartAllProcessors 函数在启动了其他所有处理器以后，设置好每个处理器的控制块（PRCB）中的相关信息，并同步所有处理器的性能计数器，最后打开屏障 KiBarrierWait，允许其他处理器进入空闲线程循环。这意味着，此后其他的处理器可以按照线程调度器的选择来运行比空闲线程优先级更高的线程了。所以，屏障 KiBarrierWait 可以看成是引导处理器对于非引导处理器的一个约束，在放任它们参与线程调度以前执行必要的初始处理。

非引导处理器的初始化过程虽然也执行 KiSystemStartup 函数，但其执行逻辑相对要简单很多。同样地，KiSystemStartup 也调用 HalInitializeProcessor 和 KiInitializeKernel 函数来执行 HAL 和内核部分的初始化，并且，KiInitializeKernel 函数也调用 ExpInitializeExecutive，但是，ExpInitializeExecutive 函数仅仅简单地调用 HalInitSystem 而已。在这些函数中，Number 参数代表了当前处理器的序号，如果为 0，则表明这是引导处理器，否则为非引导处理器。这样的逻辑不难理解：有些初始化过程，特别是内核执行体组件（比如内存管理器和对象管理器）的初始化例程，它们的处理逻辑针对整个系统，而并非某个处理器，所以只需在引导处理器上运行一次即可；而另外有些初始化过程，则是针对单个处理器的初始化（包括设置处理器的中断描述符表、PRCB 的状态等），因而需要针对每个处理器都执行一次。在阅读这些函数的代码时，我们可以清楚地看到这一基本差别。

前面我们提到了阶段 1 初始化是在 System 进程的一个系统线程中进行的，它在一个恰当的点上调用 KeStartAllProcessors 函数以启动其他的处理器。阶段 1 初始化是内核的完全初始化，它占据了系统初始化过程中相当一部分时间。在 Windows 2000 系统的引导过程中，可以在屏幕上看到一个进度条，它指示了阶段 1 初始化的进度；而在 Windows XP/Server 2003 及以后的 Windows 系统中，则是一个闪烁的 Windows 标志图案出现在屏幕上，而并非进度条。如前所述，阶段 1 初始化的主函数为 Phase1InitializationDiscard，其代码位于 base\ntos\init\initos.c 文件的 933~1 865 行。从该函数的代码可以看出，阶段 1 的初始化进度仍然有一个百分比估计。该函数按以下步骤完成其主要功能：

(1) 设置全局变量 InitializationPhase 为 1，标志着当前这次系统引导过程进入内核的阶段 1 初始化。

(2) 调用 HalInitSystem 函数，执行 HAL 的阶段 1 初始化。

(3) 初始化图形引导驱动程序，显示 Windows 启动屏幕，设置引导进度条范围(0,100)。虽然在 Windows Server 2003 中已不再显示进度条，但进度指示逻辑仍然保留。

(4) 调用 PoInitSystem，完成电源管理器的阶段 0 初始化。

(5) 调用 HalQueryRealTimeClock，初始化系统时间，这一步必须在 HAL 的阶段 1 初始化以后才能进行。再调用 KeSetSystemTime 以设置内核的系统时间。

(6) 调用 KeStartAllProcessors，启动并初始化其他的处理器。启动了这些辅助处理器以后，重新调用 HalQueryRealTimeClock 以初始化系统时间，并调用 KeSetSystemTime 设置内核的系统时间。

(7) 接下来，调用 ObInitSystem，完成对象管理器的阶段 1 初始化；调用 ExInitSystem，完成执行体组件的阶段 1 初始化；调用 KeInitSystem，完成微内核部分的初始化；调用 KdInitSystem，完成内核调试器的阶段 1 初始化；调用 SeInitSystem，完成安全子系统的阶段 1 初始化。

(8) 调用 InbvUpdateProgressBar 函数，更新进度条至 10%。

(9) 创建符号链接“\SystemRoot”。

(10) 调用 MmInitSystem，完成内存管理器的阶段 1 初始化。

(11) 将国家语言支持（NLS）表映射到系统空间中，然后重置翻译表。

(12) 调用 CcInitializeCacheManager，初始化缓存管理器。

(13) 调用 CmInitSystem1，初始化配置管理器，建立起基本的注册表对象操作，把引导时获得的数据转变成注册表格式。现在，HKLM\SYSTEM 和 HKLM\HARDWARE 已可以使用，这一步必须在 I/O 子系统初始化之前完成。

(14) 调用 CcPfInitializePrefetcher，初始化内核中的预取器（prefetcher）。

(15) 进度条前进到 15%。

(16) 调用 FsRtlInitSystem，初始化文件系统支持库（FsRtl）。

(17) 调用 KdDebuggerInitialize1，即 kdcom.dll 中的调试器初始化。

(18) 调用 PpInitSystem，初始化即插即用管理器。

(19) 进度条前进到 20%。

(20) 调用 LpcInitSystem，初始化 LPC 子系统。

(21) 现在系统时间已经正常运行，调用 ExInitSystemPhase2，再次初始化执行体组件。

(22) 进度条更新范围设置为 25~75%。

(23) 调用 IoInitSystem，初始化 I/O 系统。这是系统引导过程中较为复杂的一部分，将占据进度条 50%的范围。IoInitSystem 函数所做的工作包括：I/O 子系统中的状态变量初始化、驱动程序对象类型和设备对象类型的创建、加载“引导-启动”类型的驱动程序、

加载“系统-启动”类型的驱动程序，以及 WMI 初始化等。详细的执行过程，请参考 IoInitSystem 函数的代码，位于 base\ntos\io\iomgr\ioinit.c 文件中。

(24) 进度条更新范围恢复到 0~100%。

(25) 再次调用 MmInitSystem，将当前已加载内核模块中的 PAGE 段标记为“可换页”。

(26) 进度条前进到 80%。

(27) 调用 PoInitSystem，完成电源管理器的阶段 1 初始化。

(28) 调用 PsInitSystem，完成进程管理器的阶段 1 初始化。

(29) 进度条前进到 85%。

(30) 调用 SeRmInitPhase1，执行安全引用监视器（SRM）的阶段 1 初始化，包括创建安全引用监视器的命令服务线程。该线程创建一个名为“引用监视器命令端口”的 LPC 端口，以接收来自 lsass.exe 进程的命令。参见 SeRmInitPhase1 函数的代码。

(31) 进度条前进到 90%。

(32) 创建会话管理器子系统进程（smss.exe）。首先准备各种参数信息（RTL_USER_PROCESS_PARAMETERS 结构），包括环境参数字符串，然后调用 RtlCreateUserProcess 函数以创建 smss 进程。

(33) 进度条前进到 100%。

(34) 最后，调用 ZwWaitForSingleObject 函数，在 smss 进程的句柄上等待，超时值设置为 5 s。如果等待成功，则意味着在 5 s 内 smss 进程退出了，于是调用“KeBugCheck(SESSION5_INITIALIZATION_FAILED)”，系统崩溃；若等待超时，则认为会话管理器已经正常运行，于是阶段 1 初始化完成，当前线程蜕变成零页面线程。

阶段 1 初始化完成以后，内核已经完全初始化，执行体的各个组件进入正常运行状态。但作为一个操作系统，Windows 的引导过程尚未完成，仅仅内核正常工作还不够，系统必须让应用程序也能够运行起来。接下来的引导工作由刚刚启动起来的 smss 进程继续进行。

2.6.3 建立用户登录会话

Windows 内核在阶段 1 初始化的最后，启动了一个用户模式进程——会话管理器子系统（smss）。Smss 进程是 Windows 操作系统的关键组成部分，它尽管是一个用户模式进程，但有其特殊性：首先，它是可信的，这意味着它可以做一些其他用户进程无法做的事情，比如创建安全令牌；其次，它直接建立在 Windows 内核的基础上，只使用 Windows 内核提供的系统服务，而不依赖于任何一个环境子系统。这不难理解，因为 smss 启动时，

Windows 子系统尚未启动，而且，启动 Windows 子系统本身正是 smss 的任务之一。

Smss 进程启动以后，继续完成系统引导过程。Smss 做的工作有相当一部分依赖于注册表中的设置。在内核的阶段 1 初始化过程中，配置管理器已经初始化，但是直到 smss 进程启动起来，只有 SYSTEM 储巢已被加载到内存中。加载其他的储巢也是 smss 的职责，它通过调用系统服务 NtInitializeRegistry 来初始化注册表。在 NtInitializeRegistry 系统服务中，除了用户轮廓储巢以外的所有其他储巢，均会被加载到系统中，并且注册表树也将建立起来。实际上，加载这些储巢的工作由 CmpInitializeHiveList 函数来完成，它仅仅加载 6 个储巢（包括 SYSTEM 储巢），正好是表 2.6 中的前 6 个储巢。

Smss 在注册表中的主键是 HKLM\SYSTEM\CurrentControlSet\Control\Session Manager，用于指示 smss 在系统初始化阶段做一些必要的工作。图 2.17 显示了在 Windows Server 2003 SP1 系统中该键的一个典型设置。

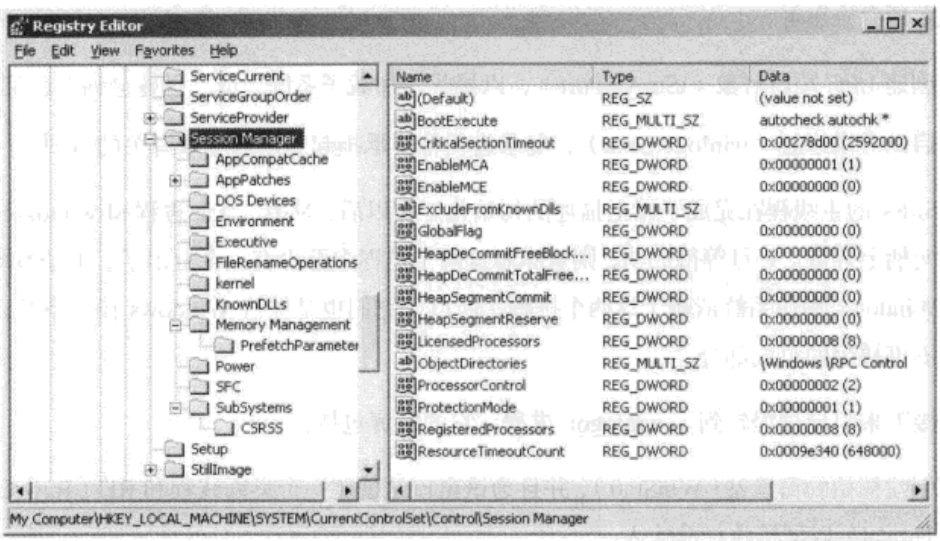


图 2.17 注册表 Session Manager 键的内容

按照 Session Manager 键中的指示，smss 主线程完成以下事项：

- 运行在启动时执行的程序，这些程序由 BootExecute 值指定。
- 执行启动时的文件删除或重命名任务，这由 FileRenameOperations 子键来指定。
- 打开已知的 DLL，并且在对象管理器名字空间的\KnownDlls 目录下创建相应的内存区对象，这些已知 DLL 的列表位于 KnownDLLs 子键中。已知 DLL 是系统全局共享 DLL，包括子系统 DLL 等，各种应用程序通常都会用到这些 DLL。
- 创建页面文件，页面文件列表由 Memory Management 子键中的 PagingFiles 值指定。

- 建立系统的全局环境变量，这些环境变量由 Environment 键下的值指定。
- 加载 Windows 子系统的内核模式模块（win32k.sys），这是通过系统服务 NtSetSystemInformation 来完成的，该函数也会调用 win32k.sys 中的初始化例程（即入口函数）。子系统内核模块的文件路径由 SubSytstems 子键的 Kmode 值指定。关于 Windows 子系统的初始化过程，请参考 9.2 节。
- 启动 Windows 子系统进程（csrss.exe）。子系统进程的命令行字符串由 SubSystems 子键的 Windows 值指定。

如果利用 Microsoft 提供的符号信息来调试 WRK 系统，则可以观察到以上这些行为均在 smss 主模块（smss.exe）的 SmpLoadDataFromRegistry 函数中完成。由此也可以看出，Windows 子系统作为会话（session）的一部分，它的实例由 smss 来启动。Smss 除了依据注册表中的设置来完成必要的系统引导工作以外，还执行以下的步骤，以进一步提供多会话和本地登录服务：

- 创建 LPC 端口对象（\SmApiPort），以接收“加载子系统”或“创建会话”的请求。
- 启动登录进程（winlogon.exe），登录进程将会承担起与用户登录有关的事项。

Smss 的主线程在完成了以上描述的初始化工作以后，将在 csrss 进程和 winlogon 进程的句柄上等待。一旦等待成功，则意味着这两个进程中至少有一个退出了，于是系统崩溃。Windows 操作系统依赖于这两个进程，所以，它们也是保持 Windows 操作系统正常运行不可缺少的组成部分。

接下来引导过程转到了 winlogon 进程。它的职责包括：

- 创建初始的窗口站（WinSta0），并且为该窗口站创建一个桌面线程和 RIT（Raw Input Thread）以便接收标准输入。
- 创建登录桌面和默认桌面。登录桌面只有 winlogon 进程才可以访问，因而也称为 winlogon 桌面；而默认桌面允许其他进程访问。因此，当登录桌面活动时，其他进程无法访问与该桌面关联的代码或数据。Windows 用这种方式来保护与口令相关的操作，以及锁定桌面或解除桌面锁定这样的安全操作。
- 启动服务控制管理器（SCM，Service Control Manager）进程（services.exe）。在启动 Windows 服务的过程中，会有更多的窗口站被创建。SCM 进程加载所有“自动-启动”类型的服务和设备驱动程序。
- 启动本地安全权威子系统（lsass）进程。然后与它建立一个 LPC 连接

（LsaAuthenticationPort 端口），以便在登录、注销和口令操作过程中交换信息。

关于这里提到的窗口站、桌面、桌面线程和 RIT 的详细解释，请参考 9.2 节。它们是 Windows 窗口管理中的重要组成部分，由 Windows 子系统内核模块 win32k.sys 实现。Winlogon 的登录是通过一种称为 GINA（图形化识别和认证，Graphical Identification and Authentication）的可扩展机制来完成的，例如，默认的 GINA 为 msgina，可显示标准的 Windows 登录对话框，指示用户输入用户名和口令。Winlogon 对于 Ctrl+Alt+Del 按键序列做了特殊的保护，以防止其他程序截取此按键序列。第三方厂商可以利用像指纹或虹膜这样的生物特征来识别用户，然后从一个加密的数据库中提取出他们的口令，这样就不需要用户再输入口令了。

当 winlogon 通过 GINA 获得了用户名和口令以后，它首先调用 lsass 函数 LsaLookupAuthenticationPackage 以获得一个认证包的句柄，然后调用 LsaLogonUser 将登录信息传递给认证包。一旦认证包认证了当前用户，则 winlogon 继续该用户的登录过程；否则认证失败。因此，认证过程是由 lsass 来完成的。

在登录过程的最后，winlogon 检查注册表 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit 的值，并创建一个进程来运行该值字符串。该值串的默认值为 userinit.exe 程序的路径。Userinit 进程加载当前登录用户的轮廓，然后检查 HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell 的值，并创建一个进程来运行该值字符串；如果该值不存在，则运行 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell 的值，其默认值为 explorer.exe。然后，userinit 进程退出。由于当前登录会话的 Shell 程序（explorer.exe）已经启动，因此用户可以在桌面上操作了。

至此，引导过程结束，用户登录到系统中，并可通过 explorer.exe 程序的用户界面操作系统中的资源，例如文件系统中的目录和文件；也可以启动各种应用程序。图 2.18 显示了从会话管理器启动一直到 Shell 程序启动之间发生的事项。

在系统引导过程中，有多个进程被创建，包括 smss、csrss、winlogon、SCM（services.exe）、lsass、userinit（登录完成后自动退出）、explorer 等。这些进程都是操作系统的一部分，而且大多数还是可信的。实际上，在系统启动以后，当用户开始在 Shell 程序中操作时，他们通常可以看到更多的进程，这其中有些进程是由 SCM 启动并初始化的。在 Windows 中，SCM 提供了在系统启动时启动进程的机制，这些进程被称为服务（service）进程，它们类似于 UNIX 中的守护进程。例如，Web 服务器或者数据库服务器显然应该以这样的方式来启动，因为无论是否有人登录到系统中，它们都必须被启动。在每个服务进程中，它可以宿纳一个或多个 Windows 服务（Windows service）。

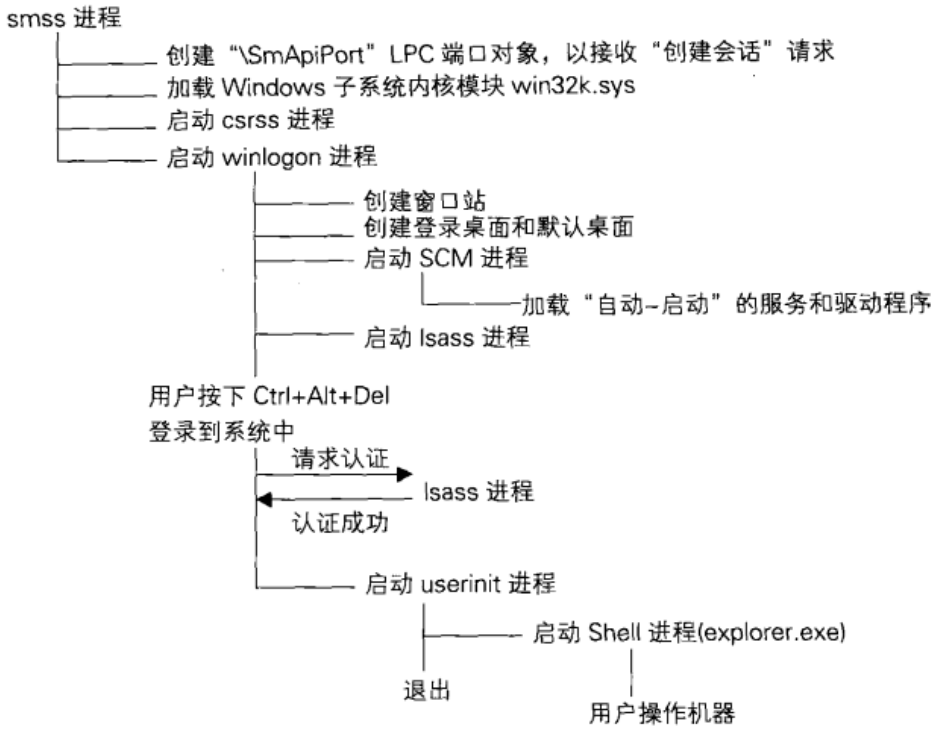


图 2.18 Windows 会话创建和用户登录过程

Windows 服务既可以是用户模式应用程序，也可以是内核模式驱动程序。但只要它们被标记了“自动-启动”（SERVICE_AUTO_START），在引导过程的后期阶段就会被启动或加载到系统中。实际上，Windows 的许多重要组件，包括网络服务、存储管理服务等，都是以 Windows 服务的形式来配置和启动的。本书第 6 章在讨论非即插即用设备驱动程序时，将会进一步介绍如何利用 SCM 来安装、启动或停止一个驱动程序。下面的插入内容“Windows 服务配置”描述了一个 Windows 服务的注册表配置信息。

Windows 服务配置

Windows 服务是 Windows 系统中一种重要的管理机制。Windows 服务控制管理器进程，即 SCM 的职责之一是，在引导过程的后期，将标记为“自动-启动”的服务应用程序或设备驱动程序启动或加载到系统中。SCM 根据注册表键 HKLM\SYSTEM\CurrentControlSet\Services 下的子键来确定如何处理每一个已被安装到系统中的服务。当一个 Windows 服务被安装到系统（通过调用 API 函数 CreateService）中时，SCM 会为它创建一个子键，其中包含了有关该服务的设置信息。图 2.19 显示了 Windows 服务“EventSystem”的注册表键。EventSystem 服务是 Windows 组件子系统（COM+）中用于分发事件的系统服务。

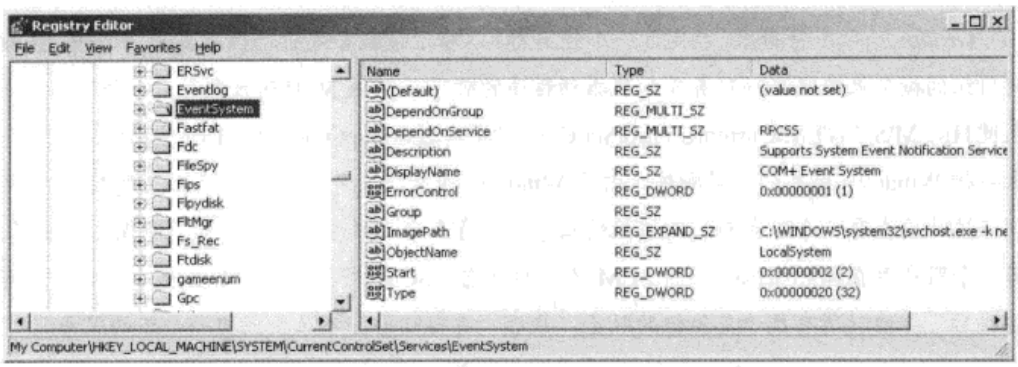


图 2.19 Windows 服务“EventSystem”的注册表键

在 Windows 服务的设置信息中，了解以下几个值将有助于理解一个服务或驱动程序的启动或加载过程：

Type：这是一个 DWORD 值，代表该服务的类型。1、2、8 分别代表设备驱动程序（SERVICE_KERNEL_DRIVER）、文件系统驱动程序（SERVICE_FILE_SYSTEM_ DRIVER）和文件系统识别器驱动程序（SERVICE_RECOGNIZER_DRIVER），这三种类型适用于内核模式驱动程序。16、32、256 分别代表独享进程的服务（SERVICE_ WIN32_OWN_PROCESS）、可共享进程的服务（SERVICE_WIN32_SHARE_PROCESS），以及允许交互式输入/输出的服务（SERVICE_INTERACTIVE_PROCESS），这三种类型适用于 Windows 服务，显然前两者是不相容的。上面例子中的 EventSystem 服务是一个可共享进程的服务。

Start：这也是一个 DWORD 值，代表一个服务或设备驱动程序的启动类型。0 表示“引导-启动”驱动程序（SERVICE_BOOT_START），由 ntldr 加载；1 表示“系统-启动”驱动程序（SERVICE_SYSTEM_START），在内核初始化过程中加载；2 表示“自动-启动”驱动程序或服务（SERVICE_AUTO_START），在引导过程中当 SCM 启动起来以后启动；3 表示“按需-启动”驱动程序或服务（SERVICE_DEMAND_START），由 SCM 根据需要启动。

ImagePath：一个字符串值，指定了该服务或驱动程序的可执行文件路径。

ObjectName：一个字符串值，仅适用于 Windows 服务。此字符串值指定了该服务将在哪个账户下运行。例如，上面例子中的 EventSystem 服务运行在本地系统（LocalSystem）账户下。如果安装服务时没有指定 *ObjectName* 值，那么 SCM 在 LocalSystem 账户下启动该服务。由于不同的服务有可能指定在不同的账户下运行，而每个 Windows 进程都要与一个窗口站关联起来，因此 SCM 在启动服务时可能会创建额外的窗口站，但除了 winlogon 创建的 WinSta0 窗口站，其他窗口站都是不可见的，它们没有输入和输出。因此，这些服务的 *Type* 值中不能标记 SERVICE_INTERACTIVE_PROCESS。

Group: 一个字符串值，指定了一个组名称，以指示该服务或驱动程序属于一个组。SCM 使用组的概念来决定启动服务或加载驱动程序的顺序。当 SCM 启动服务时，它会根据注册表键 HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder 中的 List 值指定的组顺序来安排 Windows 服务的启动顺序。由于 Windows 服务之间可能存在依赖性，因此 SCM 必须谨慎地检查每一个服务与它所依赖的服务是否存在前后顺序矛盾的情形，即循环相依性。若一个服务存在循环相依性，则 SCM 不会启动它。SCM 将在启动了 List 值中列出的所有组以后，再启动所有属于其他组的服务，最后，再启动那些不属于任何一个组的服务。

DependOnGroup、*DependOnService*: 它们是多字符串值，指定了该服务或驱动程序依赖于其他哪些组或服务。SCM 依据这两个值来判断一个服务是否循环依赖于其他的服务。如果按照组的顺序排列下来，一个服务所依赖的组中还没有任何一个服务被启动，那么，SCM 不会启动该服务。所以，在 List 值的组顺序中，前一个组中的服务不能依赖于后面的组或者后面组中的服务。如果一个服务依赖于同一组内的其他服务，并且后者尚未启动，则 SCM 会先跳过此服务，然后继续启动该组内的其他服务，待遍历该组内所有其他服务以后再从头检查每一个服务。所以，对于同一个组内的服务，SCM 利用多遍扫描的方法来检查相依性，直至每一个服务要么被启动，要么被确认存在循环相依性。

关于 Windows 服务的更详细信息，读者可以参考 MSDN 中的文档[MSDN-SERVICE]。

2.7 本章总结

本章从总体上介绍了 Windows 操作系统，包括 Windows 操作系统的组成结构和一些关键组件，以及 Windows 内核中的一些基本概念。另外，本章还基于 WRK 源代码，细致地讨论了 Windows 内核中的公共管理机制，包括对象管理器、配置管理器、事件追踪（ETW）和安全性管理。最后，本章介绍了 Windows 的引导过程，这一过程尽管比较复杂，但仍然有章可循。

Windows 是一个复杂的操作系统，它的内核有良好的设计，内核中的各个组件既有一定的独立性，又能够与其他组件协同工作。本书后面的章节将详细地介绍 Windows 内核中一些最重要的机制。首先，在下一章中我们将讨论 Windows 的进程和线程管理。

3

Windows 进程和线程

这一章介绍 Windows 操作系统中的两个核心概念：进程和线程，包括它们所涉及的内核数据结构和一些关键算法。在 Windows 中，进程和线程有明确的分工与合作：进程是各种资源的容器，它定义了一个地址空间作为基本的执行环境；而线程是一个指令执行序列，它可以直接访问所属进程中的资源。每个进程都至少有一个线程，而每个线程在任一时刻都一定属于某一特定的进程。Windows 内核中的执行体和（微）内核都有大量的代码涉及进程和线程，本章将从它们的基本概念和数据结构讲起，进一步解析 Windows 中进程和线程的管理，以及线程调度算法。

本章在讲解过程中，会列举 WRK 中的很多数据结构和函数，有些地方会不可避免地涉及 Windows 系统的其他内核组件和机制，比如内存管理和中断机制等。建议读者在阅读时，先不要深究这些细节，而把重点放在 Windows 中与进程和线程相关联的数据结构和函数上；等阅读了后面章节的内容以后，自然可以全面领会 Windows 中的进程和线程概念以及相关的管理策略。

3.1 进程基本概念

在这一节中，我们将介绍现代操作系统中的多进程模型，并讨论多个进程在单个处理器上并发执行的情形。然后，从 C 程序的内存模型入手，分析程序与进程之间的关系。程序是一个被广泛使用的术语，在不同的上下文环境中可能有不同的含义；相对而言，进程是一个系统层次上的概念，有较为清晰和严格的定义。最后，将讨论进程的创建和终止，以及进程之间的从属关系。

3.1.1 多进程模型

现代操作系统总是提供可并发执行多个任务的环境，比如，用户可以一边接收电子邮件，一边听音乐，还可以同时跟网络上的朋友聊天。但是，现在主流的计算机只有有限的计算资源，例如只有一个单核处理器，较新的个人计算机可能配有一个双核或四核的处理器。让一个系统同时做几件不同的事情，这是现代操作系统的基本特征。假设计算机只有一个中央处理器，它该如何表现得可以同时处理这些任务呢？基本的做法是把时间细分，然后在某个适当的时间粒度上轮流执行这些任务，让每个任务都有机会被执行到。只要在人所能感知的时间粒度上（差不多是 100~200 ms）每个任务都有机会做一点事情，用户就会认为这些任务是同时在执行。这种并行也被称为**伪并行**，而这样的系统被称为分时系统。有时候，计算机可能配有多个处理器，伪并行也有可能变成**真并行**，因为多个任务可以在多个处理器上真正地并行执行。Windows 操作系统能有效地支持这两种情形，它的任务调度算法可以很好地适应多处理器和多任务的情形。

我们可以简单地把进程（process）理解成一个独立的任务，它不仅是一个正在被执行的程序，有自己的指令流和内存空间，同时，还拥有一颗仅供自己使用的虚拟 CPU。从整个系统的角度来看，多个进程是分时执行的，但是对于每个进程来说，它的指令流按顺序执行，就好像独占一颗 CPU 一样。图 3.1 显示了三个进程在同一个处理器上分时执行的效果。

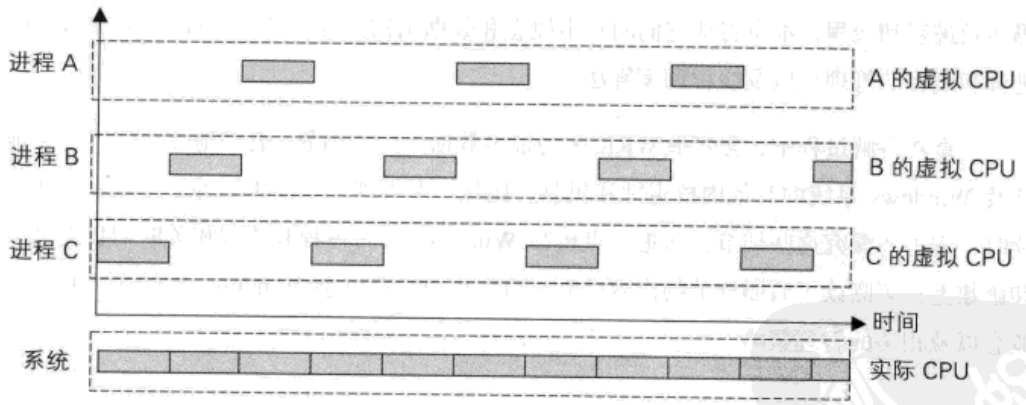


图 3.1 多个进程在一个处理器上分时运行

如图 3.1 所示，每个进程都有自己的运行时间轴，感觉就像在一个单独的处理器上运行一样。系统负责切换这些进程，让它们得以轮流执行。当一个进程执行了一段时间以后，系统通过某种硬件机制或软件机制获得控制权，保留好这个进程的环境信息，包括内存空间、寄存器和指令流信息，然后挑选下一个要被执行的进程，恢复此进程的环境信息，并把控制权交给它，由它使用下一个时间片段。

因此，操作系统需要做的事情是：**维护一个全局的进程表**，记录下当前有哪些进程正在被执行；**把时间分成适当的片段**，在现代处理器结构中，这可以通过设置时钟中断来完成，因而每次时钟中断到来时系统就会获得控制权；**在进程间实施切换**，即保留上一个进程的环境信息，恢复下一个进程的执行环境。现在我们可以明白，每个进程在执行过程中实际上是经常被打断的，但由于系统能够准确地维护好进程的环境信息，所以，进程并不需要考虑自身的执行被频繁地打断这一事实，而是简单地认为自己是在持续地运行。一个聪明的进程通过读取系统的时间信息，可以估算出自己获得了多少真正有效的处理器运行时间。对于大多数进程而言，它们无须考虑这些因素。

3.1.2 进程与程序

理解了进程分时执行的情况以后，我们现在来看一看每个进程如何维护它自己独立的运行环境。在讲述进程环境以前，首先介绍一下**程序**的概念及其与进程之间的关系。在许多讲述 C/C++程序设计的书籍或者材料中，都会提到一个程序的内存布局结构，如图 3.2 所示。实际上，这里提到的程序内存结构与进程有一定的联系。每个程序必定是为了完成特定的功能或任务而存在，所以需要有代码来完成这些任务，但是仅仅有代码还不足以完成这些任务，还需要相应的存储单元来配合，包括内存和寄存器。

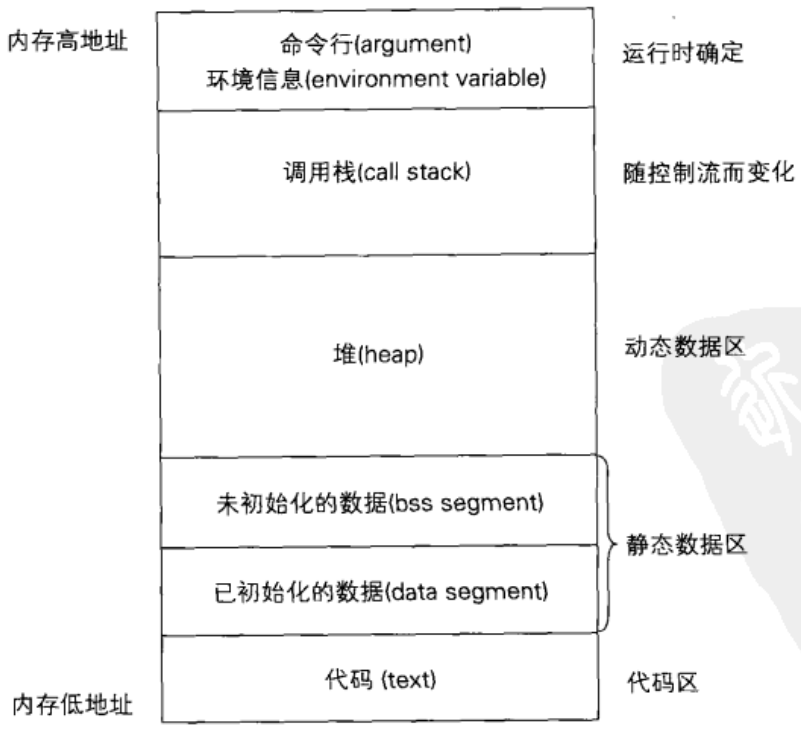


图 3.2 一个 C/C++程序的典型布局结构

在现代程序中用到的内存区域有三种类型：静态数据区、动态数据区，以及维护控制流信息和局部状态的调用栈区域。对于一个 C/C++程序而言，静态数据区存放的是全局变量以及静态变量等信息，这是程序员在编写代码时就确定的数据区域；动态数据区则是在程序执行过程中根据需要而分配或回收内存的区域。调用栈存放的是程序执行过程中的函数状态信息，当控制流从一个函数调用到另一个函数中时，调用栈记录了调用函数中的状态信息，包括所有的局部变量和当前指令流经过此次函数调用的位置。同时调用栈也用于传递函数参数。调用栈是实现函数调用的一种有效手段。

在一个程序的布局结构中，代码和静态数据区是可以预先确定的，命令行和环境信息也可以在启动一个程序的时候确定下来，但动态数据区和调用栈是程序运行过程中动态变化的。调用栈区域的大小通常是在程序启动时或者在程序的二进制属性中预先设置好，如果没有预设的值，则系统使用默认的大小。这些区域确定下来以后，中间的部分全部是动态数据区，由程序在运行过程中分配和管理。

现在我们回到进程的概念上来，如果一个程序是一个完全独立的模块，也就是说，它不依赖于任何其他应用模块就可以完成其既定的功能或任务，那么，图 3.2 实际上也可以代表一个进程的内存布局。对于一个进程来说，它的存在之所以有意义，是因为它能完成特定的功能，这起码需要有一个对应的二进制模块，即前面所说的程序。然而，在实际的应用中，一个程序模块往往需要依赖于其他的模块来完成它的功能，比如某个第三方的二进制库。在这种情况下，图 3.2 中的代码区和静态数据区在一个进程的内存布局结构中会有多份，每份对应于一个二进制模块。从这一层意义上讲，程序的概念相当于一个二进制可执行模块。由于这些被依赖的模块也可以称为程序，所以，程序的概念在不同的上下文环境中可能指代不同的事物。典型情况有两种：

- 程序代表了一个进程，通常用启动该进程的二进制可执行模块的文件名来表示程序的名称。在 Windows 平台上往往是一个.exe 文件。
- 程序代表了一个二进制可执行模块，通常用该模块的文件名来表示程序的名称。在 Windows 平台上，既可以是.exe 文件，也可以是 DLL 文件或者其他后缀的文件。

当然，程序的概念还可以用于其他的情形，甚至指代源代码，这里不一一展开讨论。要记住的一点是，进程的概念是清晰和严格的，相对而言，程序的概念没那么清晰，两者之间既有区别也有联系，有时候甚至混用。所以，读者在阅读书籍或文章时，若看到程序一词，须注意上下文。

现在我们知道了，一个进程包含有代码和数据，以及在执行过程中有关控制流状态的

信息，当它在运行过程中需要更多的内存时，它可以从动态内存区域获取内存，不用的时候再释放这些内存。为了建立起一个进程，需要一个初始的二进制可执行模块，来得到一个图 3.2 所示的基本内存结构。如果该进程还需要额外的二进制模块，则这些模块的代码和静态数据区也会在适当的时候被引入到进程中来，并占据一定的内存空间。

操作系统只维护进程的基本信息，以便对系统中所有的进程实施管理，包括为每个进程分配处理器时间，以及在进程之间进行切换。既然进程是操作系统完成各种任务的基本载体，那么操作系统如何控制进程的创建和终止呢？首先，操作系统在引导过程中，必须把系统运行所必要的进程全部创建起来。所谓创建一个进程是指建立起基本的进程执行环境，然后把它加入到系统的全局进程表中，使它有机会得到处理器时间和其他系统资源，以后，这个进程就可以按照其自身的逻辑独立运行了。所以，操作系统会创建一组初始的进程，这是在系统引导时完成的。

其次，在一个系统的运行过程中，一个进程在必要的时候可能会创建其他的进程。虽然普通进程无法完成创建一个进程所需要的各种步骤，但是它可以请求系统帮它创建一个进程。这实际上建立起了进程之间的父子关系或从属关系，即父进程创建子进程。另外，一个进程也可以为了响应用户的请求而创建一个新的进程。比如用户通过某种方式要求编辑一个文本文件，系统中当前正在运行的进程接收到用户的请求以后，创建一个新的子进程来打开并编辑指定的文本文件。除了系统的初始进程以外，其他的进程都是在系统运行过程中根据需要而创建的。

进程如何终止呢？当一个进程完成了所有预定的功能以后，它的使命便已结束，最后的职责是通知操作系统，自己要终止了。操作系统接到请求以后，清理掉这个进程所占据的各种资源，使它们能被回收，并且将该进程从系统的全局进程表中移除，所以以后不会再给它分配任何处理器时间。这是比较温和、友好的进程终止做法。在其他有些情况下，即使一个进程还没有完成其任务，它也可能被另外的进程或者系统终止。比如，当进程出错或者状态不正确时，系统一旦检测到其错误并且认为这是不可恢复的，则会强制终止此进程。另外，由于进程之间可能存在逻辑上的关联关系，所以，一个进程可能会要求系统终止另一个进程，即俗称的“杀死一个进程（kill a process）”。

前面提到了，一个进程可能会包含多个二进制可执行模块，它们的代码区和静态数据区都被引入到进程的内存空间中，但是，动态数据区可能是系统全局的，也可能每个模块有其特有的动态数据区，这取决于每个模块自己的实现，但从概念上讲，它们的动态数据区应该是进程全局共享的。另外一个没有提及的重要方面是，默认情况下进程只有一个控制流，尽管这个控制流可以跨越多个模块，但是它的调用栈只有一个。在现代操作系统结

构中，进程可以有多个控制流，每个控制流对应一个调用栈。操作系统在创建进程时仅建立一个控制流，但是在该控制流的执行过程中，它又可以创建额外的控制流，这样便形成了一个进程多个控制流的情形。这里的控制流实际上正是下一节要讨论的线程概念。

3.2 线程基本概念

线程不仅仅是一个控制流，它还有更多的内容。线程的调用栈（call stack）记录了它作为控制流的状态信息，包括每一层函数调用和返回的指令地址。线程一定隶属于某个进程，其控制流可以访问这个进程中的资源，包括所有的内存数据以及系统分配给此进程的其他资源。一个进程可以有多个线程，由于这些线程隶属于同一个进程，所以它们之间相互通信要方便得多，毕竟几乎所有的资源（并非全部）对于它们来说都是共享的。因此，线程概念的引入有时候也被称为轻量进程（lightweight process）。接下来我们首先讨论线程的模型，然后介绍典型的线程调度算法，最后介绍进程与线程的关系。

3.2.1 线程模型

在现代计算机结构中，先后提出过两种线程模型：用户级线程（user-level threads）和内核级线程（kernel-level threads）。所谓用户级线程是指，应用程序在操作系统提供的单个控制流的基础上，通过在某些控制点（比如系统调用）上分离出一些虚拟的控制流，从而模拟多个控制流的行为。由于应用程序对指令流的控制能力相对较弱，所以，用户级线程之间的切换往往受线程本身行为以及线程控制点选择的影响，线程是否能公平地获得处理器时间取决于这些线程的代码特征。而且，支持用户级线程的应用程序代码很难做到跨平台移植，以及对于多线程模型的透明。用户级线程模型的优势是线程切换效率高，因为它不涉及系统内核模式和用户模式之间的切换；另一个好处是应用程序可以采用适合自己特点的线程选择算法，可以根据应用程序的逻辑来定义线程的优先级，当线程数量很大时，这一优势尤为明显。但是，这同样会增加应用程序代码的复杂性。有一些软件包（如 POSIX Threads 或 Pthreads 库）可以减轻程序员的负担。

内核级线程往往指操作系统提供的线程语义，由于操作系统对指令流有完全的控制能力，甚至可以通过硬件中断来强迫一个进程或线程暂停执行，以便把处理器时间移交给其他的进程或线程，所以，内核级线程有可能应用各种算法来分配处理器时间。线程可以有优先级，高优先级的线程被优先执行，它们可以抢占正在执行的低优先级线程。在支持线程语义的操作系统中，处理器的时间通常是按线程而非进程来分配，因此，系统有必要维

护一个全局的线程表，在线程表中记录每个线程的寄存器、状态以及其他一些信息。然后，系统在适当的时候挂起一个正在执行的线程，选择一个新的线程在当前处理器上继续执行。这里“适当的时候”可以有多种可能，比如：当一个线程执行某些系统调用时，例如像 sleep 这样的放弃执行权的系统函数，或者像 wait 或 select 这样的阻塞函数；硬中断（interrupt）或异常（exception）；线程终止时，等等。由于这些时间点的执行代码可能分布在操作系统的不同位置，所以，在现代操作系统中，线程调度（thread scheduling）往往比较复杂，其代码通常分布在内核模块的各处。在本章后面，我们将会看到 Windows 中的线程调度是如何实现的。

内核级线程的好处是，应用程序无须考虑是否要在适当的时候把控制权交给其他的线程，不必担心自己霸占处理器而导致其他线程得不到处理器时间。应用线程只要按照正常的指令流来实现自己的逻辑即可，内核会妥善地处理好线程之间共享处理器的资源分配问题。然而，这种对应用程序的便利也是有代价的，即，所有的线程切换都是在内核模式下完成的，因此，对于在用户模式下运行的线程来说，一个线程被切换出去，以及下次轮到它的时候再被切换进来，要涉及两次模式切换：从用户模式切换到内核模式，再从内核模式切换回用户模式。在 Intel 的处理器上，这种模式切换大致需要几百个甚至上千个处理器指令周期。但是，随着处理器的硬件速度不断加快，模式切换的开销相对于现代操作系统的线程调度周期（通常几十毫秒）的比例正在减小，所以，这部分开销是完全可以接受的。

除了线程切换的开销是一个考虑因素以外，线程的创建和删除也是一个重要的考虑指标。当线程的数量较多时，这部分开销是相当可观的。虽然线程的创建和删除比起进程要轻量得多，但是，在一个进程内建立起一个线程的执行环境，例如，分配线程本身的数据结构和它的调用栈，完成这些数据结构的初始化工作，以及完成与系统环境相关的一些初始化工作，这些负担是不可避免的。另外，当线程数量较多时，伴随而来的线程切换开销也必然随之增加。所以，当应用程序或系统进程需要的线程数量可能比较多时，通常可采用线程池技术作为一种优化措施，以降低创建和删除线程以及线程频繁切换而带来的开销。

在支持内核级线程的系统环境中，进程可以容纳多个线程，这导致了**多线程程序设计**（multithreaded programming）模型。由于多个线程在同一个进程环境中，它们共享了几乎所有的资源，所以，线程之间的通信要方便和高效得多，这往往是进程间通信（IPC，Inter-Process Communication）所无法比拟的，但是，这种便利性也很容易使线程之间因同步不正确而导致数据被破坏，而且，这种错误存在不确定性，因而相对来说难以发现和调试。本书第 5 章将会介绍有关线程间同步的话题。

3.2.2 线程调度算法

在图 3.1 中，我们看到了多个进程共享一个处理器时实际上是按照某种规则轮换执行的。当一个进程执行了一段时间以后，下一个进程被选出来占有处理器，继续它的指令流序列。这便是进程调度算法。如果一个操作系统支持内核级线程，那么，处理器资源的调度通常在线程而非进程粒度上进行。本节后面我们将介绍一些典型的线程调度算法（不完全等同于有些书籍中的进程调度或作业调度算法，原因是一些经典的进程或作业调度算法要求预先知道进程或作业的执行时间长度）。

我们首先看一下如何评价一个调度算法。衡量调度算法的准则包括多个方面，第一是公平性，调度算法在选择下一个运行的线程时，要考虑到同等地位的线程必须有相同的机会获得处理器执行权；第二是 CPU 的有效利用，只要有进程或线程还在等待执行，就不能让 CPU 空闲着。另外，不同类型的操作系统对于调度算法会有不同的需求，例如，实时操作系统对于响应时间有最低要求。有时候这些调度需求可能存在矛盾，比如，最大化吞吐量和最小化响应时间可能无法兼顾。

从大的分类来讲，调度算法可以分为非抢占式算法和抢占式算法。在非抢占式系统中，一个线程一旦被选择在处理器上运行，就将一直运行下去，直到阻塞（比如等待 I/O 或等待一个信号量）或者自愿放弃或退出。在这类算法中，如果一个线程陷入一个长时间处理甚至无限循环的过程之中，则系统就将无法再运行其他的线程，从而整个系统可能会挂起，因此系统必须有相应的机制来打破这种可能的停滞。在抢占式系统中，一个线程被选中在处理器上运行以后，允许运行的时间长度有最大限制，一旦达到了这么长时间，就将被迫放弃执行权，交由系统挑选其他的线程来运行，或者若找不到其他的线程，则再把执行权交还给它，让它继续运行。抢占式调度算法需要一个时钟中断来获得对处理器的控制权，而非抢占式算法并不需要时钟中断。

下面介绍三种典型的线程调度算法：

(1) 先到先服务算法。在非抢占式系统中，这一算法比较自然，简单来讲，用一个 FIFO（先进先出）队列就可以满足要求。所有的线程构成一个队列，最先进入队列的线程获得处理器执行权，等到放弃处理器执行权时，又回到队列尾部，下一个线程继续执行。若有新的线程进来，则添加到队列尾部。此算法简单，易于实现，但是，如果每个线程执行的任务单元所需要的时间长短不一的话，则算法的实际效果可能非常不公平。

(2) 时间片轮转调度算法。顾名思义，处理器的时间被分成了最大长度不超过某个值的时间片段，称为时间片，然后，用轮转方法分配给每一个线程。当一个线程获得了处理

器执行权以后，按照自身的逻辑执行下去，直到时间片用完，或者自己主动放弃执行权（比如要等待一个信号量）。系统在获得了处理器控制权以后，用轮转方法找到下一个正在等待运行的线程，让它继续执行。这种线程调度方法实现简单，所有满足运行条件的线程排成一个队列，然后按照时间片的间隔，轮流让每一个线程获得处理器执行权。由于时钟中断每次都要打断一个线程的运行，所以，这种做法存在固有的线程切换开销，而时间片长短的选择会影响到线程切换开销所占的比例。在现代操作系统中，时间片通常设置为几毫秒到几十、上百毫秒。由于现代计算机的指令周期越来越短，线程切换开销（通常几百条指令或几千条指令，取决于算法实现的复杂程度）也在减小。这种算法使用很广泛，它不仅简单，也确实能公平地分配处理器资源。

(3) 优先级调度算法。在时间片轮转算法中，一个基本的假设是所有的线程都同等重要。这一假设在专用计算机上可能是非常合理的，但是，在现代多用途的计算机上，可能难以胜任多种不同类型的应用程序并发执行的实际情形。优先级调度算法是这种算法的一个改进，其基本思路是，每个线程都有一个优先级值，高优先级的线程总是优先被考虑在处理器上执行。操作系统在管理线程时，可以使用一个优先级队列，或者每一个优先级用一个队列来存放所有满足执行条件的线程，这样，当一个线程用完了它的时间片或者自动放弃处理器执行权时，系统选择优先级最高的线程作为下一个要运行的线程。每一个线程在队列中的位置是由它的优先级来决定的。同等优先级的线程使用轮转或先到先执行的策略。

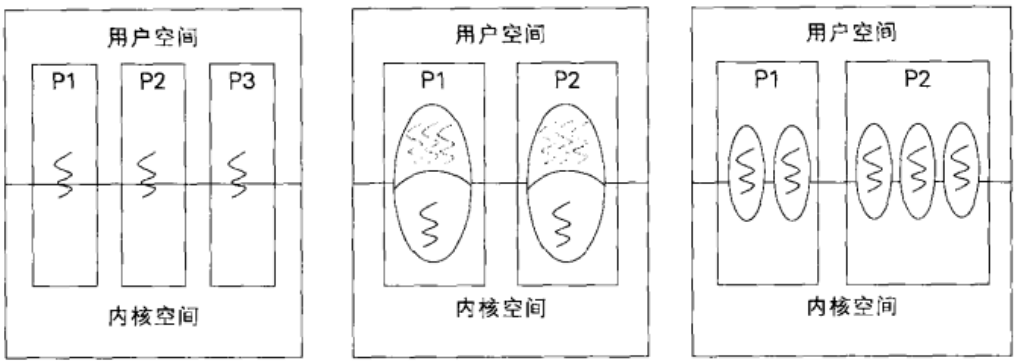
简单优先级算法的潜在问题是，高优先级的线程可能会霸占处理器资源不放，从而导致低优先级的线程一点执行机会都没有。所以，一些变种的优先级算法考虑引入动态优先级，即每个线程有静态的优先级和动态的优先级。所谓动态的优先级是在静态优先级的基础上根据某些特定的条件提升或降低线程的优先级，系统调度器根据线程的动态优先级来安排它们的执行顺序。例如，连续执行了多个时间片的线程可能要降低优先级，而长时间没有得到时间片的低优先级线程可能会得到优先级提升。

Windows 的调度算法是一个抢占式的、支持多处理器的优先级调度算法，它为每个处理器定义了一个链表数组，相同优先级的线程挂在同一个链表中，不同优先级的线程分别属于不同的链表。当一个线程满足了执行条件时，它首先被挂到当前处理器的一个待分配的链表（称为延迟的就绪链表）中，然后调度器会在适当的时候（当它获得了控制权时）把待分配链表上的线程分配到某个处理器的对应优先级的线程链表中。当这个处理器在选择下一个要运行的线程时，会根据优先级准则选择此线程（如果没有同等或更高优先级的线程也在等待运行的话）。Windows 中线程的优先级调整考虑到了很多因素，例如前台线程、等待 I/O 完成后的线程也有轻微的优先级提升，这是一些来自实践经验的设计细节，

它们使得 Windows 操作系统对于交互式应用程序有更好的性能表现。本章 3.5 节将会详细介绍 Windows 的线程调度。

3.2.3 线程与进程的关系

现在，我们来看一下线程与进程之间的关系。在不支持线程语义的系统中，进程提供了一个完全的执行环境，同时也有一个控制流完成其预定的功能，如图 3.3(a)所示。操作系统按进程来分配处理器资源。在这种系统中，对线程的支持可以在用户模式下完成，即用户级线程模型，如图 3.3(b)所示。在支持内核级线程的系统中，进程仅提供一个执行环境，它包含一个或者多个线程，每个线程代表一个单独的指令流，操作系统按线程来分配处理器资源，如图 3.3(c)所示。Windows 中的线程是内核级线程，所以，它与进程之间的关系属于图 3.3(c)所描述的情形。



(a) 进程模型(不支持线程) (b) 用户级线程模型 (c) 内核级线程模型

图 3.3 线程与进程的关系

3.3 Windows 中进程和线程的数据结构

上一章提到，Windows 内核中的执行体层负责各种与管理和策略相关的功能，而内核层（或微内核）实现了操作系统的核心机制。进程和线程在这两层上都有对应的数据结构，本节将列出这些数据结构，并解释其中的域的含义。

3.3.1 内核层的进程和线程对象

首先看（微）内核层的数据结构，这是进程和线程最为基本的数据结构，分别名为 KPROCESS 和 KTHREAD。下面是 WRK 中 KPROCESS 的定义（见 base\ntos\inc\ke.h）：

```
typedef struct _KPROCESS {
    DISPATCHER_HEADER Header;
    LIST_ENTRY ProfileListHead;

    ULONG_PTR DirectoryTableBase[2];

    KGDTENTRY LdtDescriptor;
    KIDTENTRY Int21Descriptor;
    USHORT IopmOffset;
    UCHAR Iopl;
    BOOLEAN Unused;

    volatile KAFFINITY ActiveProcessors;

    ULONG KernelTime;
    ULONG UserTime;

    LIST_ENTRY ReadyListHead;
    SINGLE_LIST_ENTRY SwapListEntry;

    PVOID VdmTrapcHandler;

    LIST_ENTRY ThreadListHead;
    KSPIN_LOCK ProcessLock;
    KAFFINITY Affinity;

    union {
        struct { ..... };
        LONG ProcessFlags;
    };

    SCHAR BasePriority;
    SCHAR QuantumReset;
    UCHAR State;
    UCHAR ThreadSeed;
    UCHAR PowerState;
    UCHAR IdealNode;
    BOOLEAN Visited;
    union {
        KEXECUTE_OPTIONS Flags;
        UCHAR ExecuteOptions;
    };

    ULONG_PTR StackCount;
    LIST_ENTRY ProcessListEntry;
} KPROCESS, *PKPROCESS, *PRKPROCESS;
```

每个 KPROCESS 对象都代表一个进程，反之也成立，即每个进程都有一个 KPROCESS 对象。KPROCESS 对象是如何建立起来的，我们稍后介绍 EPROCESS 对象的时候再讲。现在我们来看 KPROCESS 对象中每个域的含义。

Header 域表明 KPROCESS 对象也是一个分发器对象（dispatcher object），关于分发器对象的定义和用法，请参考 5.4 节关于线程间同步机制的介绍。现在我们只需知道，进程对象本身是可以被等待的，也就是说，如果一个线程调用 Windows 的等待函数，在一个进程对

象上等待，那么，当此进程对象变成有信号状态时（这发生在进程退出时），此等待就会成功。所以，进程对象是一个可等待的对象。*ProfileListHead* 域用于当该进程参与性能分析（profiling）时，作为一个节点加入到全局的性能分析进程列表（内核全局变量 KiProfileListHead）中。

DirectoryTableBase 是一个只有两项的数组，其中第一项指向该进程的页目录表地址，第二项指向该进程的超空间（hyper space）的页目录表地址。关于进程的页目录表和超空间的概念，请参考本书第 4 章的介绍。

接下来的 *LdtDescriptor*、*Int21Descriptor*、*IopmOffset* 和 *Iopl* 这四个域是专门针对 Intel x86 处理器的。其中 *LdtDescriptor* 是该进程的 LDT（局部描述符表）的描述符；*Int21Descriptor* 是为了兼容 DOS 程序，允许它们通过 int 21h 指令来调用 DOS 系统功能；*IopmOffset* 则指定了 IOPM（I/O 权限表，I/O Privilege Map）的位置，内核通过 IOPM 可控制进程的用户模式 I/O 访问权限；*Iopl* 域定义了进程的 I/O 优先级（I/O Privilege Level）。

ActiveProcessors 域记录了当前进程正在哪些处理器上运行。接下来的 *KernelTime* 和 *UserTime* 域分别记录了一个进程对象在内核模式和用户模式下所花的时间。进程的 *KernelTime* 和 *UserTime* 时间值等于其所属线程的对应 *KernelTime* 和 *UserTime* 值的和。但是，由于仅当一个线程结束时才更新其进程的这两个时间值，所以，若一个进程中尚未有任何一个线程结束，则这两个域中的值为 0。

ReadyListHead 是一个双向链表的表头，该链表记录了这个进程中处于就绪状态但尚未被加入全局就绪链表的线程，这个域的意义在于，当一个进程被换出内存以后，它所属的线程一旦就绪，则被挂到此链表中，并要求换入该进程；以后，当该进程被换入内存时，*ReadyListHead* 中的所有线程被加入到系统全局的就绪线程链表中。注意，*ReadyListHead* 链表中的每一项都是一个指向 KTHREAD 对象（后文讲述）的 *WaitListEntry* 域的地址，所以，从链表中的每一项都可以定位到对应的线程对象。例如，下面的代码片段是从 *KiInSwapProcesses* 函数中摘录的，从代码中可以看到 *ReadyListHead* 链表的用法。这段代码的意图是，将链表中的每个线程加入到全局就绪线程链表中（通过调用 *KiReadyThread* 来完成）。

```
Process->State = ProcessInMemory;
NextEntry = Process->ReadyListHead.Flink;
while (NextEntry != &Process->ReadyListHead) {
    Thread = CONTAINING_RECORD(NextEntry, KTHREAD, WaitListEntry);
    RemoveEntryList(NextEntry);
    Thread->ProcessReadyQueue = FALSE;
    KiReadyThread(Thread);
    NextEntry = Process->ReadyListHead.Flink;
}
```

SwapListEntry 域是一个单链表项，当一个进程要被换出内存时，它通过此域加入到以 *KiProcessOutSwapListHead* 为链头的单链表中；当一个进程要被换入内存时，它通过此域加入到以 *KiProcessInSwapListHead* 为链头的单链表中。这里，*KiProcessOutSwapListHead* 和 *KiProcessInSwapListHead* 是定义在 base\ntos\ke\kernldat.c 中的全局变量。

随后的域是 *VdmTrapcHandler*，指向处理 Ctrl+C 中断的函数，仅用于在 VDM（虚拟 DOS 机，Virtual DOS Machine）环境下运行 16 位程序。

ThreadListHead 域指向一个链表头，此链表包含了该进程的所有当前线程。当一个线程被初始创建的时候，被加入到此链表中，在终止的时候被从链表中移除。由此我们可以看出进程和线程之间的从属关系。

接下来是 *ProcessLock* 域，这是一个自旋锁（spin lock）对象，它的用途是保护此进程中的数据成员。在内核代码中关于进程和线程各种操作的很多地方都可以看到对 *ProcessLock* 锁的操作，这可以确保对进程数据结构的修改和访问总是一致的。

Affinity 域指定了该进程的线程可以在哪些处理器上运行，其类型是 KAFFINITY，这是一个 32 位或 64 位整数，其二进制表示的每一位分别对应于当前机器上的一个处理器（或核）。

ProcessFlags 域包括了进程中的几个标志：*AutoAlignment*、*DisableBoost* 和 *DisableQuantum*。*AutoAlignment* 位用于该进程中的内存访问对齐设置，此标志位也会被传递到线程的数据结构中，当一个线程的对齐检查开关打开时，该线程中的未对齐数据访问将会导致对齐错误（alignment fault），不过，Intel x86 并未做任何对齐检查，所以这一位的意义并不重要；*DisableBoost* 和 *DisableQuantum* 位与线程调度过程中的优先级提升和时限（quantum）分配有关，本章 3.5 节在讲述线程调度时将会介绍线程的优先级提升和时限的计算。

KPROCESS 中的 *BasePriority* 和 *QuantumReset* 域是该进程中的线程的调度参数，*BasePriority* 用于指定一个进程中的线程的基本优先级，所有的线程在启动时都会继承进程的 *BasePriority* 值；*QuantumReset* 用于指定一个进程中线程的基本时限重置值，在现代 Windows 版本中此值被设置为 6，本章 3.5 节讲述 Windows 线程调度时将会解释为什么该值为 6 而不是一个基本的时间片单位。*State* 域说明了一个进程是否在内存中，共有六种可能的状态：ProcessInMemory、ProcessOutOfMemory、ProcessInTransition、ProcessOutTransition、ProcessInSwap 和 ProcessOutSwap。所谓一个进程在内存中，或者已被换出，或者正在转移过程中，是指该进程的虚拟地址空间需要占据足够的物理内存，或者虚拟空间中的内容已被换出物理内存，或者正在换入或换出过程之中。

KPROCESS 中的 *ThreadSeed* 域用于为该进程的线程选择适当的理想处理器（Ideal Processor），在每个线程被初始化的时候，都指定此进程的 *ThreadSeed* 值作为它的理想处理器，然后 *ThreadSeed* 域又被设置一个新的值，以便该进程的下一个线程使用。这里的理想处理器是指在多处理器环境下，每个线程都有一个优先选择的处理器。*PowerState* 域用于记录电源状态，关于电源状态管理，请参考 6.4 节的介绍。*IdealNode* 域用于为一个进程选择优先的处理器节点，这是在进程初始化时设定的。这里的处理器节点是 NUMA（非一致的内存访问）结构中的概念。

KPROCESS 中的其余域的含义如下：*Visited* 域，在 WRK 中未使用；*ExecuteOptions* 用于设置一个进程的内存执行选项，这是为了支持 NX（No-Execute，内存不可执行）而引入到 Windows XP/Server 2003 中的一个域；*StackCount* 域记录了当前进程中有多少个线程的栈位于内存中；*ProcessListEntry* 域用于将当前系统中所有具有活动线程的进程串成一个链表，链表头为 *KiProcessListHead*。

简单归纳一下，KPROCESS 对象中记录的信息主要包括两类：一类跟进程的内存环境相关，比如页目录表、交换状态等；另一类是与其线程相关的一些属性，比如线程列表以及线程所需要的优先级、时限设置等。系统中的 KPROCESS 对象通过 *KiProcessListHead* 链表串起来，但这一链表仅用于 AMD64 系统。我们将会看到，系统的总进程链表是在执行体层管理的，在下一节讨论 EPROCESS 对象结构时我们将看到一个总进程链表。

现在讨论内核层线程对象的基本数据结构 KTHREAD，由于 KTHREAD 中包含较多的信息，我们分三部分列出其结构定义。注意，为方便讲述，这里列示的成员顺序跟 base\ntos\inc\ke.h 中 KTHREAD 结构的定义并不完全相同。以下是第一部分：

```
typedef struct _KTHREAD {
   DISPATCHER_HEADER Header;
   LIST_ENTRY MutantListHead;

   PVOID InitialStack;
   PVOID StackLimit;
   PVOID KernelStack;
   PVOID StackBase;

   KSPIN_LOCK ThreadLock;
   union {
      KAPC_STATE ApcState;
      struct { ..... };
   };

   KSPIN_LOCK ApcQueueLock;

   ULONG ContextSwitches;
   volatile UCHAR State;
```

```
    BOOLEAN Alertable;
    BOOLEAN WaitNext;
    UCHAR WaitReason;
    UCHAR NpxState;
    KIRQL WaitIrql;
    KPROCESSOR_MODE WaitMode;
    LONG_PTR WaitStatus;
    union {
        PKWAIT_BLOCK WaitBlockList;
        PKGATE GateObject;
    };

    ..... /* 后面还有两部分 */
        •
} KTHREAD, *PKTHREAD, *PRKTHREAD;
```

由于在 Windows 中，线程是系统处理器调度的基本单元，而且线程调度是在内核层完成的，所以，KTHREAD 的许多域都跟 Windows 的线程调度机制有关。在下面对 KTHREAD 成员的描述中，有些域的含义需结合 Windows 的线程调度机制来理解。

KTHREAD 的第一个域 *Header* 说明了内核层的线程对象也是一个分发器对象，线程可以被等待，当线程结束时，在该对象上的等待可被满足。*MutantListHead* 域指向一个链表头，该链表中包含了所有属于该线程的突变体对象（mutant，对应于 API 中的互斥体[mutex]对象）。由于突变体对象是有所有权的，一旦被某个线程等到，则其所有权归该线程所有，它也被连接到 *MutantListHead* 链表中。

有四个域用于内核栈的维护，它们分别是：*InitialStack*、*StackLimit*、*KernelStack* 和 *StackBase*。栈是从高地址向低地址方向变化的，*InitialStack* 记录了原始的栈位置（高地址）；*StackLimit* 记录了栈的低地址；*KernelStack* 记录了真正内核调用栈的开始位置，由于在内核栈的顶部区域还记录了浮点处理器保存区和一个异常陷阱帧，所以，*KernelStack* 的位置比 *InitialStack* 要低一些（KTRAP_FRAME_LENGTH + sizeof(FX_SAVE_AREA)）。*StackBase* 记录了当前栈的基位置（高地址）。在线程初始化时，*InitialStack* 和 *StackBase* 是相等的，都指向原始的内核栈高地址。

ThreadLock 域是一个自旋锁，用于保护线程数据成员。*ApcState* 是一个结构成员，指定了一个线程的 APC（Asynchronous Procedure Call）信息，包括 APC 链表、是否正在处理 APC 或者是否有内核 APC 或用户 APC 正在等待等信息。KAPC_STATE 中的 *Process* 域指向当前线程所属进程的 KPROCESS 结构。在 *ApcState* 所在的 union 中，*ApcState* 之后还有几个域：*ApcQueueable*、*NextProcessor*、*DeferredProcessor*、*AdjustReason* 和 *AdjustIncrement*，分别指定了以下设置：是否可以插入 APC、关于处理器调度的选择，以及优先级调整原因和调整量。*ApcQueueLock* 也是一个自旋锁，用于保护 APC 队列的操作。

ContextSwitches 域记录了该线程进行了多少次环境切换。*State* 域反映了该线程当前的状态，关于线程的状态，请参考 base\ntos\inc\ke.h 中的 KTHREAD_STATE 枚举类型，在本章 3.5.2 节介绍线程调度算法时我们将讨论线程的状态变化。*NpxState* 反映了浮点处理器的状态，本书不讨论。

Alertable 域说明了一个线程是否可以被唤醒，当一个线程正在等待时，如果它的 *Alertable* 值为 TRUE，则它可以被唤醒。*WaitNext* 域也是一个布尔值，TRUE 值表示这个线程马上要调用一个内核等待函数，它的用途是，在发出了一个信号（比如释放了一个信号量对象）以后，接下来该线程会马上调用等待函数，所以，它不必解除线程调度器锁。*WaitIrql* 域与 *WaitNext* 一起使用，当 *WaitNext* 为 TRUE 时，*WaitIrql* 记录了原先的 IRQL 值。*WaitReason* 域记录了一个线程的等待理由，其值的含义见 base\ntos\inc\ke.h 中的 KWAIT_REASON 枚举类型。*WaitReason* 基本上只是记录了等待的理由，而并不参与到线程调度或决策中。*WaitMode* 域记录了当线程等待时的处理器模式，即内核模式或用户模式的等待。接下来是 *WaitStatus* 域，它记录了等待的结果状态。

WaitBlockList 成员指向一个以 KWAIT_BLOCK 为元素的链表，其中的 KWAIT_BLOCK 对象指明了哪个线程在等待哪个分发器对象。对于一个线程而言，*WaitBlockList* 域以及每个 KWAIT_BLOCK 对象中的 *WaitListEntry* 域构成了一个双链表，指明了该线程正在等待哪些分发器对象；而对于每个分发器对象而言，它又有另一个 KWAIT_BLOCK 链表指明了哪些线程正在等待它。5.4 节介绍 Windows 的线程同步机制时我们将会看到这两个链表。

当一个线程正在等待门对象（Gate Object，也是一种分发器对象）时，*GateObject* 域记录了正在等待的门对象。由于等待门对象和等待其他分发器对象是不会同时发生的，所以，*GateObject* 和 *WaitBlockList* 域构成了一个 union，共用一个指针内存。

接下来列出 KTHREAD 结构定义的第二部分：

```
typedef struct _KTHREAD {

    ..... /* 前面已列出并解释了第一部分 */

    SCHAR Priority;
    SCHAR BasePriority;
    SCHAR PriorityDecrement;
    CHAR Saturation;
    UCHAR EnableStackSwap;
    volatile UCHAR SwapBusy;
    BOOLEAN Alerted[MaximumMode];
    union {
        LIST_ENTRY WaitListEntry;
```

```
        SINGLE_LIST_ENTRY SwapListEntry;
    };

    PRKQUEUE Queue;

    ULONG WaitTime;
    union {
        struct {
            SHORT KernelApcDisable;
            SHORT SpecialApcDisable;
        };

        ULONG CombinedApcDisable;
    };

    PVOID Teb;

    KTIMER Timer;           // 以下 union 的定义与原代码中的定义等价
            union {
                struct {
                    LONG AutoAlignment : 1;
                    LONG DisableBoost : 1;
                    LONG ReservedFlags : 30;
                };
                LONG ThreadFlags;
            };

    union {
        KWAIT_BLOCK WaitBlock[THREAD_WAIT_OBJECTS + 1];
        ……
    };

    LIST_ENTRY QueueListEntry;

    …… /* 后面还有第三部分 */

} KTHREAD, *PKTHREAD, *PRKTHREAD;
```

Priority 域包含了该线程的优先级值，这是指它的动态优先级，即在执行过程中可能由于某些原因而调整过的优先级。*BasePriority* 域是线程的静态优先级，其初始值是所属进程的 *BasePriority* 值，以后可通过 KeSetBasePriorityThread 函数重新设定。*PriorityDecrement* 域记录了一个线程在优先级动态调整过程中的递减值，每次当重新计算优先级值的时候，它会被减掉（参见 base\ntos\ke\ki.h 中的 KiComputeNewPriority 函数）。Windows 中的优先级值被分成两个区域：0~15 是普通线程的优先级，16~31 是实时线程的优先级。而且，线程的优先级在调整时，无论如何其 *Priority* 值不会跨越区域。*Saturation* 域说明了线程的基本优先级相对于进程的基本优先级的调整量是否超过了整个区间的一半，其值为 0、1 或–1。

EnableStackSwap 域是一个布尔值，说明本线程的内核栈是否允许被换出到外存中。*SwapBusy* 域也是一个布尔值，指定了本线程当前是否正在进行上下文环境切换（context

swap），其用法是，在将执行环境切换到其他线程以前设置 *SwapBusy* 域为 TRUE，切换完成以后再设置回 FALSE。*Alerted* 域是一个数组，指定了该线程在每一种警告模式下是否可以被唤醒，在 WRK 中，所谓警告模式实际上也就是内核模式和用户模式，所以，这个数组的含义是指该线程分别在内核模式和用户模式下是否可以被唤醒。

WaitListEntry 和 *SwapListEntry* 分别是一个双链表节点和一个单链表节点，它们是一个 union，分别用于不同的情形。当一个线程正在等待被执行时，*WaitListEntry* 作为一个线程节点加入到某个链表中。例如，本节前文在介绍 KPROCESS 的 *ReadyListHead* 域时曾提到，在进程被换入内存过程中，就绪状态的线程将被加入到以进程的 *ReadyListHead* 域为链表头的双链表中，链表中的节点即为线程的 *WaitListEntry* 域。*SwapListEntry* 域则被用于当线程的内核栈需要被换入时，插入到以全局变量 *KiStackInSwapListHead* 为链表头的单链表中。另外，当一个线程处于 DeferredReady 状态时，其 *SwapListEntry* 将被插入到某个处理器的 *DeferredReadyListHead* 链表中（参见 *KiInsertDeferredReadyList* 和 *KiProcessDeferredReadyList* 内核函数）。

KTHREAD 的 *Queue* 域是一个队列分发器对象，如果不为 NULL，则表示当前线程正在处理此队列对象中的项。关于 Windows 内核中的队列机制，请参考本书第 5 章。*WaitTime* 域记录了一个线程进入等待时刻的时间点（时钟滴答值的低 32 位），主要用于平衡集管理器（参见第 4 章）根据一个线程的等待时间的先后来做一些决策。

接下来是 *KernelApcDisable* 和 *SpecialApcDisable* 域，或者合并成一个 *CombinedApcDisable* 域。*KernelApcDisable* 和 *SpecialApcDisable* 都是 16 位的整数值，0 表示不禁止 APC，负数表示禁止 APC，一个线程在执行过程中可以有多种因素要禁止 APC，这些因素以负值来表示，并累加起来，当因素消除的时候再减去相应的负值。只有当 *KernelApcDisable* 或 *SpecialApcDisable* 为 0 的时候，该线程才允许插入或提交 APC。这两个值分别控制普通的内核 APC 和特殊的内核 APC，关于这两种内核模式 APC 请参考 5.2.6 节。

Teb 域是一个特殊的域，它指向进程地址空间中的一个 TEB（线程环境块）结构。TEB 结构包含了在用户地址空间中需要访问的各种信息，例如与线程相关的 GDI 信息、系统支持的异常，甚至还有 WinSock 的信息，等等。关于 TEB 结构的定义，可以参考 public\sdk\inc\pebteb.h 中的_TEB 定义。base\ntos\mm\procsup.c 文件中的 MmCreateTeb 函数在一个进程中创建并初始化一个 TEB。

接下来是 *Timer* 域，这是附在一个线程上的定时器，当一个线程在执行过程中需要定时器时，比如实现可超时的等待函数（*KeWaitForSingleObject* 或 *KeWaitForMultipleObjects*），

就会用到此定时器对象。紧接着 *Timer* 域之后的是 *AutoAlignment* 和 *DisableBoost* 两个位标记（原始定义把 Timer 放在一个 union 中，本书中列出的代码是等价的声明），线程的这两个标记直接继承自所属进程的同名标记。前面介绍 KPROCESS 成员的时候已经提到过，这里不再赘述。

WaitBlock 域是一个包含 4 个 KWAIT_BLOCK 成员的数组，其中第 4 项专门用于可等待的定时器对象。正如前面介绍 *WaitBlockList* 域时所讲，KWAIT_BLOCK 结构代表了一个线程正在等待一个分发器对象，或者说一个分发器对象正在被一个线程等待，它会被同时加入到两个双链表结构中。*WaitBlock* 域是一个内置数组，内核在实现等待功能的时候，如果一个线程所等待的对象数量小于 4（实际上应该是 3 个分发器对象加上一个定时器对象），则内核无须另外分配 KWAIT_BLOCK 对象内存，只需直接使用 *WaitBlock* 中的数组成员即可。如果等待的对象数量大于 4，则内核必须分配额外的 KWAIT_BLOCK 对象内存。由于等待操作在内核中非常频繁，所以，利用静态数组来满足大多数情况下的内存需求，这一优化非常有意义。关于 KWAIT_BLOCK 结构和分发器对象的 DISPATCHER_HEADER 结构，请参考 5.4 节。

QueueListEntry 域记录了线程在处理一个队列项时加入到队列对象的线程链表中的节点地址。参见 5.4.2 节关于队列对象的介绍。

下面列出 KTHREAD 结构定义的第三部分：

```
typedef struct _KTHREAD {

    ..... /* 前面已列出并解释了第一和第二部分 */

    PKTRAP_FRAME TrapFrame;
    PVOID CallbackStack;
    PVOID ServiceTable;

    UCHAR IdealProcessor;
    BOOLEAN Preempted;
    BOOLEAN ProcessReadyQueue;

    BOOLEAN KernelStackResident;
    KAFFINITY UserAffinity;
    PKPROCESS Process;
    KAFFINITY Affinity;

    UCHAR ApcStateIndex;
    PKAPC_STATE ApcStatePointer[2];
    union {
        KAPC_STATE SavedApcState;
        struct { ..... };
    };
```

```
    PVOID Win32Thread;
    union {
        KAPC SuspendApc;
        ......
    };

    union {
        KSEMAPHORE SuspendSemaphore;
        struct {
            UCHAR SuspendSemaphorefill[KSEMAPHORE_ACTUAL_LENGTH];
            ULONG SListFaultCount;
        };
    };

    LIST_ENTRY ThreadListEntry;
    PVOID SListFaultAddress;

} KTHREAD, *PKTHREAD, *PRKTHREAD;
```

TrapFrame 域是一个线程中最为关键的部分了。在 Windows 中，线程是系统调度的基础，它代表了一个进程中的一个控制流。当一个线程离开运行状态时，其当前的执行状态，比如现在的指令指针（IP）在哪里，各个寄存器中的值是什么，等等，都必须保存下来，以便下次再轮到这个线程运行时，可以恢复原来的执行状态。*TrapFrame* 域正是记录控制流状态的数据结构，它是一个指向 KTRAP_FRAME 类型的指针。对于 Intel x86 体系结构，KTRAP_FRAME 的定义位于 base\ntos\inc\i386.h 文件中，它包含了 Intel x86 的所有常用寄存器，读者可以参考此文件中的定义。*CallbackStack* 域包含了线程的回调（callback）栈地址，此栈在该线程从内核模式调用到用户模式时使用。

ServiceTable 域指向该线程使用的系统服务表（全局变量 *KeServiceDescriptorTable*），如果这是一个图形用户界面（GUI）线程，此域将指向另一个影子系统服务表（全局变量 *KeServiceDescriptorTableShadow*），参见 8.1 节关于 Windows 系统服务分发的介绍。*IdealProcessor* 域指明了在多处理器的机器上该线程的理想处理器。

Preempted 域是一个布尔值，说明这个线程是否被高优先级的线程抢占了，只有当一个线程正在运行或者正在等待运行而被高优先级线程抢占的时候，此值才会置成 TRUE。在其他情况下，该值总是 FALSE。*ProcessReadyQueue* 域也是一个布尔值，说明一个线程是否在所属进程 KPROCESS 对象的 *ReadyListHead* 链表中，TRUE 表示在此链表中，FALSE 表示不在。*KernelStackResident* 域也是一个布尔值，说明该线程的内核栈是否驻留在内存中，当内核栈被换出内存时，该值将被置成 FALSE；当换入内存时，再置成 TRUE。

Affinity 域指定了线程的处理器亲和性，此值初始时继承自进程对象的 *Affinity* 值。为线程指定的处理器集合必须是其进程的亲和性处理器集合的子集。在线程执行过程中，其

Affinity 值可能有两种设置：一是系统亲和性，当该线程执行系统任务时通过 KeSetSystemAffinityThread 函数来设置；二是线程本身的亲和性，称为用户亲和性，通过 KeRevertToUserAffinityThread 函数来设置。*UserAffinity* 域是线程的用户亲和性，此值初始时也继承自进程对象的 *Affinity* 值，以后可通过内核函数 KeSetAffinityThread 改变。*Process* 域指向线程的进程对象，在线程初始化时指定，见 base\ntos\ke\thredobj.c 中的 KeInitThread 函数。

ApcStateIndex 域是一个索引值，它指明了当前的 APC 状态在 *ApcStatePointer* 域中的索引。由于 *ApcStatePointer* 是一个只有两个元素的数组，所以，*ApcStateIndex* 的值为 0 或者 1。*ApcStatePointer* 数组元素的类型是指向 KAPC_STATE 的指针，其两个元素分别指向线程对象的 *ApcState* 和 *SavedApcState* 域，而这两个域分别位于两个 union 中。关于这两个域的含义和用法，请参考本书 5.2.6 节关于 APC 机制的介绍。

Win32Thread 域是一个指针，指向由 Windows 子系统管理的区域。*SuspendApc* 和 *SuspendSemaphore* 两个 union 域相互之间有联系，*SuspendApc* 被初始化成一个专门的 APC。当该 APC 被插入并交付时，*KiSuspendThread* 函数被执行，其执行结果是在线程的 *SuspendSemaphore* 信号量上等待，直到该信号量对象有信号，然后线程被唤醒并继续执行。线程的挂起（suspend）操作正是通过这一机制来实现的，参见 base\ntos\ke\thredobj.c 中的 KeSuspendThread 和 KeFreezeAllThreads 函数。自然地，线程的恢复（resume）操作则是通过控制 *SuspendSemaphore* 信号量的计数来实现的，参见 base\ntos\ke\thredobj.c 中的 KeResumeThread 等函数。

ThreadListEntry 域代表了一个双链表上的节点，当一个线程被创建时，它会被加入到进程对象的 *ThreadListHead* 链表中，参见本节前文关于 KPROCESS 的 *ThreadListHead* 域的介绍。*SListFaultAddress* 域与用户模式互锁单链表 POP 操作（KeUserPopEntrySListFault 函数）的错误处理有关，它记录了上一次用户模式互锁单链表 POP 操作发生页面错误的地址；*SuspendSemaphore* 域所在的 union 中还有一个 *SListFaultCount* 域也与此相关，它记录了在地址 *SListFaultAddress* 上发生页面错误的次数。

从以上介绍的 KPROCESS 和 KTHREAD 数据结构我们可以看出，内核层的进程和线程对象只包含了系统资源管理和多控制流并发执行所涉及的基本信息，而没有包含与应用程序相关联的信息，如进程映像文件和线程启动函数地址等。由于 Windows 的线程调度算法比较复杂（在抢占式调度算法的基础上，还支持优先级局部调整等），且需要支持某些硬件结构特性，所以，在 KPROCESS 和 KTHREAD 结构中，有些成员的引入直接跟这些特性有关，在本书后文我们将会陆续看到这些成员的用法。尽管如此，从概念上讲，内

核层上的进程和线程对象仍然十分清晰，即，进程对象提供了线程的基本执行环境，包括进程地址空间和一组进程范围内公用的参数；线程对象提供了为参与线程调度而必需的各种信息及其维护控制流的状态。

3.3.2 执行体层的进程和线程对象

执行体层位于内核层之上，它侧重于提供各种管理策略，同时为上层应用程序提供基本的功能接口。所以，在执行体层的进程和线程数据结构中，有些成员直接对应于上层应用程序中所看到的功能实体。下面先讨论执行体层进程对象的数据结构 EPROCESS，其定义位于文件 base\ntos\inc\ps.h 中，共分四个部分来介绍，以下是第一部分的定义：

```
typedef struct _EPROCESS {
   KPROCESS Pcb;

   EX_PUSH_LOCK ProcessLock;
   LARGE_INTEGER CreateTime;
   LARGE_INTEGER ExitTime;

   EX_RUNDOWN_REF RundownProtect;
   HANDLE UniqueProcessId;

   LIST_ENTRY ActiveProcessLinks;

   SIZE_T QuotaUsage[PsQuotaTypes];
   SIZE_T QuotaPeak[PsQuotaTypes];
   SIZE_T CommitCharge;
   SIZE_T CommitChargeLimit;
   SIZE_T CommitChargePeak;

   SIZE_T PeakVirtualSize;
   SIZE_T VirtualSize;

   LIST_ENTRY SessionProcessLinks;

   PVOID DebugPort;
   PVOID ExceptionPort;
   PHANDLE_TABLE ObjectTable;

   EX_FAST_REF Token;

   ..... /* 后面还有三部分 */

} EPROCESS, *PEPROCESS;
```

Pcb 域即上节介绍的 KPROCESS 内嵌结构体，因此，在系统内部，一个进程的 KPROCESS 对象的地址和 EPROCESS 对象的地址是相同的。*ProcessLock* 域与 KPROCESS 中的自旋锁同名，但它们的类型不同，保护的成员也不同。这里的 *ProcessLock* 域是一个推锁（push lock）对象，用于保护 EPROCESS 中的数据成员。*CreateTime* 和 *ExitTime* 域分别是进程的创建时间和退出时间。

RundownProtect 域是进程的停止保护锁，当一个进程到最后被销毁时，它要等到所有其他进程和线程已经释放了此锁，才可以继续进行。*UniqueProcessId* 域是进程的唯一编号，在进程创建时设定。*ActiveProcessLinks* 域是一个双链表节点，在 Windows 系统中，所有的活动进程都连接在一起，构成一个双链表，表头是全局变量 PsActiveProcessHead。当一个进程被创建时，其 *ActiveProcessLinks* 域将被作为节点加入到此链表中；当进程被删除时，则从链表中移除。如果 Windows 需要枚举所有的进程，直接操纵此链表即可。

QuotaUsage 和 *QuotaPeak* 域是指一个进程的内存使用量和尖峰使用量。这两个域是数组，其中的元素分别对应于非换页内存池、换页内存池和交换文件中的内存使用情况。这两个数组是在 PspChargeQuota 函数中计算的。*CommitCharge* 域包含了一个进程的虚拟内存已提交的页面数量；*CommitChargePeak* 域则是尖峰时刻的已提交页面数量；*CommitChargeLimit* 域是已提交页面数量的限制值，如果是 0 则表示没有限制。在 WRK 中，默认的限制值为 0。*VirtualSize* 域是指一个进程的虚拟内存大小，*PeakVirtualSize* 域则是指虚拟内存大小的尖峰值。

SessionProcessLinks 域是一个双链表节点，当进程加入到一个系统会话中时，其 *SessionProcessLinks* 域将作为一个节点加入到该会话的进程链表中。*DebugPort* 和 *ExceptionPort* 域是两个句柄，分别指向调试端口和异常端口。当一个进程中的线程发生用户模式异常时，内核的异常处理例程在处理异常过程中，将向该进程的异常端口或调试端口发送消息，从而使这些端口的接收方（调试器或 Windows 子系统）能够处理该异常。5.2.7 节将介绍 Windows 的异常分发过程。*ObjectTable* 域是进程的句柄表，句柄是一个抽象概念，代表了进程已打开的一个对象。句柄表包含了所有已被该进程打开的那些对象的引用。Windows 进程的句柄表并不是简单的一张表，它有层次结构，详见 3.4.1 节。*Token* 域是一个快速引用，指向该进程的访问令牌，用于该进程的安全访问检查。

接下来看 EPROCESS 结构的第二部分：

```
typedef struct _EPROCESS {

    ..... /* 前面已列出并解释了第一部分 */

    PFN_NUMBER WorkingSetPage;
    KGUARDED_MUTEX AddressCreationLock;
    KSPIN_LOCK HyperSpaceLock;

    struct _ETHREAD *ForkInProgress;
    ULONG_PTR HardwareTrigger;

    PMM_AVL_TABLE PhysicalVadRoot;
    PVOID CloneRoot;
    PFN_NUMBER NumberOfPrivatePages;
```

```
    PFN_NUMBER NumberOfLockedPages;
    PVOID Win32Process;
    struct _EJOB *Job;
    PVOID SectionObject;

    PVOID SectionBaseAddress;

    PEPROCESS_QUOTA_BLOCK QuotaBlock;

    PPAGEFAULT_HISTORY WorkingSetWatch;
    HANDLE Win32WindowStation;
    HANDLE InheritedFromUniqueProcessId;

    PVOID LdtInformation;
    PVOID VadFreeHint;
    PVOID VdmObjects;
    PVOID DeviceMap;

    ..... /* 后面还有第三、四部分 */

} EPROCESS, *PEPROCESS;
```

WorkingSetPage 域是指包含进程工作集的页面。*AddressCreationLock* 域是一个守护互斥体锁（guarded mutex），用于保护对地址空间的操作。当内核代码需要对虚拟地址空间进行操作时，它必须在 *AddressCreationLock* 上执行锁操作，完成以后再解锁。base\ntos\mm\mi.h 中的宏 LOCK_ADDRESS_SPACE 和 UNLOCK_ADDRESS_SPACE 可用于简化代码。*HyperSpaceLock* 是一个自旋锁，用于保护进程的超空间。

ForkInProgress 指向正在复制地址空间的那个线程，仅当在地址空间复制过程中，此域才会被赋值，在其他情况下为 NULL。参见 base\ntos\mm\forksup.c 中的 MiCloneProcess-AddressSpace 函数。*HardwareTrigger* 域用于记录硬件错误性能分析次数，在 WRK 中没有使用。

PhysicalVadRoot 域指向进程的物理 VAD 树的根，它并不总是存在，只有当确实需要映射物理内存时才会被创建。*CloneRoot* 也指向一个平衡树的根，当进程地址空间复制时，此树被创建，创建出来以后，一直到进程退出的时候才被销毁。*CloneRoot* 域完全是为了支持 fork 语义而引入。*NumberOfPrivatePages* 和 *NumberOfLockedPages* 域分别是指进程私有页面的数量和被锁住页面的数量。

Win32Process 域是一个指针，指向由 Windows 子系统管理的进程区域，如果此值不为 NULL，说明这是一个 Windows 子系统进程（GUI 进程）。对于 *Job* 域，只有当一个进程属于一个 Job（作业）的时候，它才会指向一个_EJOB 对象。本书不讨论 Job 对象。*SectionObject* 域也是一个指针，代表进程的内存区对象（进程的可执行映像文件的内存区对象）。*SectionBaseAddress* 域为该内存区对象的基地址。*QuotaBlock* 域指向进程的配额

块，其类型为 EPROCESS_QUOTA_BLOCK，Windows 系统中的配额块相互串起来构成了一个双链表，每个配额块都可以被多个进程共享，所以有一个引用计数值用来说明当前有多少个进程在使用这一配额块。配额块中主要定义了非换页内存池、换页内存池和交换文件中的内存配额限制。系统的默认配额块为 *PspDefaultQuotaBlock*，所有配额块构成的双链表的表头为 *PspQuotaBlockList*。

WorkingSetWatch 域用于监视一个进程的页面错误，一旦启用了页面错误监视功能（由全局变量 *PsWatchEnabled* 开关来控制），则每次发生页面错误都会将该页面错误记录到 *WorkingSetWatch* 域的 WatchInfo 成员数组中，直至数组满为止（参见 PsWatchWorkingSet 函数的代码）。*Win32WindowStation* 域是一个进程所属的窗口站的句柄。由于句柄的值是由每个进程的句柄表来决定的，所以，两个进程即使同属于一个窗口站，它们的 *Win32WindowStation* 值也可能不同，但指向的窗口站对象是相同的。窗口站是由 Windows 子系统来管理和控制的。*InheritedFromUniqueProcessId* 域说明了一个进程是从哪里继承来的，即父进程的标识符。

LdtInformation 域用来维护一个进程的 LDT（局部描述符表）信息。*VadFreeHint* 域指向一个提示 VAD（虚拟地址描述符）节点，用于加速在 VAD 树中执行查找操作。*VdmObjects* 域指向当前进程的 VDM 数据区，其类型为 VDM_PROCESS_OBJECTS，进程可通过 NtVdmControl 系统服务来初始化 VDM。*DeviceMap* 域指向进程使用的设备表，通常情况下同一个会话中的进程共享同样的设备表，本书第 7 章将介绍此设备表的用法。

接下来看 EPROCESS 结构的第三部分：

```
typedef struct _EPROCESS {

    ..... /* 前面已介绍了第一、二部分 */

    PVOID Spare0[3];
    union {
        HARDWARE_PTE PageDirectoryPte;
        ULONGLONG Filler;
    };
    PVOID Session;
    UCHAR ImageFileName[ 16 ];

    LIST_ENTRY JobLinks;
    PVOID LockedPagesList;

    LIST_ENTRY ThreadListHead;

    PVOID SecurityPort;

    PVOID PaeTop;
    ULONG ActiveThreads;
```

```
    ACCESS_MASK GrantedAccess;
    ULONG DefaultHardErrorProcessing;
    NTSTATUS LastThreadExitStatus;
    PPEB Peb;
    ..... /* 后面还有第四部分 */
} EPROCESS, *PEPROCESS;
```

Spare0 域是一个备用域，在 WRK 中没有使用。*PageDirectoryPte* 域是顶级页目录页面的页表项。*Session* 域指向进程所在的系统会话，在 EPROCESS 的定义中其类型为 PVOID，实际上它是一个指向 MM_SESSION_SPACE 的指针，其定义位于 base\ntos\mm\mi.h 中。每个进程在初始创建地址空间时会加入到当前的系统会话中。*ImageFileName* 域包含了进程的映像文件名，仅包含最后一个路径分隔符之后的字符串，不超过 16 个字符（含最后的结尾空字符）。

JobLinks 域是一个双链表节点，通过此节点，一个 Job 中的所有进程构成了一个链表。在 Windows 中，所有的 Job 构成了一个双链表，其链表头为全局变量 *PspJobList*；每个 Job 中的进程又构成了一个双链表。*LockedPagesList* 域是一个指向 LOCK_HEADER 结构的指针，该结构包含了一个链表头，Windows 通过此链表来记录哪些页面已被锁住。base\ntos\mm\iosup.c 中有一组函数用于管理此链表，例如 MiAddMdlTracker、MiFreeMdlTracker 和 MiUpdateMdlTracker 等。

ThreadListHead 域是一个双链表的头节点，该链表包含了一个进程中的所有线程。在上一小节我们看到，KPROCESS 结构中也有一个 *ThreadListHead* 域，它所指的链表包含了各个子线程的 KTHREAD 结构中的 *ThreadListEntry* 节点；同样地，在执行体层也有这样的包含关系，即 EPROCESS 中的 *ThreadListHead* 域的链表包含了各个子线程的 ETHREAD 结构中的 *ThreadListEntry* 节点（本节稍后将看到此域）。这样的组织虽然有冗余，但是保持了内核层和执行体层的相对独立关系。

SecurityPort 域是一个安全端口，指向该进程与 lsass 进程之间的跨进程通信端口。*PaeTop* 域用于支持 PAE 内存访问机制。*ActiveThreads* 域记录了当前进程有多少个活动线程。当该值减为 0 时，所有的线程都将退出，于是进程也退出。*GrantedAccess* 域包含了进程的访问权限，访问权限是一个位组合，各种权限定义参见 public\sdk\inc\ntpsapi.h 中的宏 PROCESS_<*XXX*>，例如 PROCESS_VM_READ 代表虚拟内存读权限。*DefaultHardErrorProcessing* 域指定了默认的硬件错误处理，默认为 1。*LastThreadExitStatus*

域记录了刚才最后一个线程的退出状态。*Peb* 域是一个进程的进程环境块（PEB，Process Environment Block），这是一个位于进程地址空间的内存块，其中包含了有关进程地址空间中的堆和系统模块等信息。

接下来看 EPROCESS 结构的最后一部分：

```
typedef struct _EPROCESS {

    ..... /* 前面已介绍了第一、二、三部分 */

    EX_FAST_REF PrefetchTrace;

    LARGE_INTEGER ReadOperationCount;
    LARGE_INTEGER WriteOperationCount;
    LARGE_INTEGER OtherOperationCount;
    LARGE_INTEGER ReadTransferCount;
    LARGE_INTEGER WriteTransferCount;
    LARGE_INTEGER OtherTransferCount;

    PVOID AweInfo;

    SE_AUDIT_PROCESS_CREATION_INFO SeAuditProcessCreationInfo;

    MMSUPPORT Vm;

    LIST_ENTRY MmProcessLinks;

    ULONG ModifiedPageCount;

    ULONG JobStatus;

    union {
        ULONG Flags;
        struct { ..... };
    };

    NTSTATUS ExitStatus;

    USHORT NextPageColor;
    union {
        struct {
            UCHAR SubSystemMinorVersion;
            UCHAR SubSystemMajorVersion;
        };
        USHORT SubSystemVersion;
    };
    UCHAR PriorityClass;
    MM_AVL_TABLE VadRoot;
    ULONG Cookie;
} EPROCESS, *PEPROCESS;
```

PrefetchTrace 域是一个快速引用，指向与该进程关联的一个预取痕迹结构，以支持该进程的预取。本书不讨论进程预取特性。*ReadOperationCount* 和 *WriteOperationCount* 域

记录了当前进程 NtReadFile 和 NtWriteFile 系统服务被调用的次数；而 *OtherOperationCount* 域则记录了除读和写操作以外的其他 I/O 服务的次数。*ReadTransferCount* 和 *WriteTransferCount* 域分别记录了 I/O 读操作和写操作完成的次数；*OtherTransferCount* 域则记录了非读写操作完成的次数。*AweInfo* 域是一个指向 AWEINFO 结构的指针，其目的是支持 AWE（Address Windowing Extension，地址窗口扩展）。

SeAuditProcessCreationInfo 域包含了创建进程时指定的进程映像全路径名，前面提到的 *ImageFileName* 域实际上是从这里提取出来的。*Vm* 域是 Windows 为每个进程管理虚拟内存的重要数据结构成员，其类型为 MMSUPPORT，参见 base\ntos\inc\ps.h 中的定义。本书第 4 章介绍 Windows 工作集管理时将会进一步讲述 MMSUPPORT 结构的用途。*MmProcessLinks* 域代表一个双链表节点，所有拥有自己地址空间的进程都将加入到一个双链表中，链表头是全局变量 *MmProcessList*。当进程地址空间被初始创建时，*MmProcessLinks* 节点被加入到此全局链表中；当进程地址空间被销毁时，该节点脱离此链表。此全局链表的存在使得 Windows 系统可以方便地执行一些全局的内存管理任务。*ModifiedPageCount* 域记录了该进程中已修改页面的数量。

JobStatus 域记录了进程所属 Job 的状态。*Flags* 域包含了进程的标志位，这些标志位反映了进程的当前状态和配置，关于这些标志位的定义，参见 base\ntos\inc\ps.h 中的宏定义 PS_PROCESS_FLAGS_<*XXX*>。*ExitStatus* 域包含了进程的退出状态，从进程的退出状态通常可以获知进程非正常退出的大致原因。反映退出状态的一些宏定义位于 public\sdk\inc\ntstatus.h 文件中。*NextPageColor* 域用于物理页面分配算法。

SubSystemVersion 域中的 *SubSystemMajorVersion* 和 *SubSystemMinorVersion* 分别记录了一个进程的子系统主版本和次版本号，它们的值来源于进程映像文件的对应版本信息（二进制可执行映像文件的头部包含了此信息）。*PriorityClass* 域是一个单字节值，它说明了一个进程的优先级程度，在 WRK 中包括以下类别：空闲类别（值为 1）、普通类别（2）、高优先级类别（3）、实时类别（4）、普通之下类别（5）、普通之上类别（6），以及未知类别（0）。读者可以从 public\skd\inc\ntpsapi.h 中找到 PROCESS_PRIORITY_CLASS_<*XXX*>宏定义。

最后，*VadRoot* 域指向一个平衡二叉树的根，用于管理该进程的虚拟地址空间。*Cookie* 域存放的是一个代表该进程的随机值，当第一次通过 NtQueryInformationProcess 函数获取此 *Cookie* 值的时候，系统会生成一个随机值，以后就用该值代表此进程。

在介绍了 EPROCESS 结构的成员构成以后，我们再来看执行体层上的线程对象的数

据结构 ETHREAD。相对而言，它的成员比 EPROCESS 少一些，其定义也位于文件 base\ntos\inc\ps.h 中。接下来我们分两部分介绍 ETHREAD 结构，以下列出第一部分：

```
typedef struct _ETHREAD {
    KTHREAD Tcb;

    LARGE_INTEGER CreateTime;

    union {
        LARGE_INTEGER ExitTime;
        LIST_ENTRY LpcReplyChain;
        LIST_ENTRY KeyedWaitChain;
    };
    union {
        NTSTATUS ExitStatus;
        PVOID OfsChain;
    };

    LIST_ENTRY PostBlockList;

    union {
        PTERMINATION_PORT TerminationPort;
        struct _ETHREAD *ReaperLink;
        PVOID KeyedWaitValue;
    };

    KSPIN_LOCK ActiveTimerListLock;
    LIST_ENTRY ActiveTimerListHead;

    CLIENT_ID Cid;

    union {
        KSEMAPHORE LpcReplySemaphore;
        KSEMAPHORE KeyedWaitSemaphore;
    };

    union {
        PVOID LpcReplyMessage;  // -> Message that contains the reply
        PVOID LpcWaitingOnPort;
    };

    PPS_IMPERSONATION_INFORMATION ImpersonationInfo;

    LIST_ENTRY IrpList;

    ULONG_PTR TopLevelIrp;  // either NULL, an Irp or a flag defined in FsRtl.h
    struct _DEVICE_OBJECT *DeviceToVerify;

    ..... /* 后面还有第二部分 */

} ETHREAD, *PETHREAD;
```

如同 EPROCESS 结构包含了内核层的 KPROCESS 对象一样，ETHREAD 结构也内嵌了 KTHREAD 对象作为第一个数据成员，所以，一个指向 ETHREAD 对象的指针同时也

是一个指向 KTHREAD 对象的指针。*CreateTime* 域包含了线程的创建时间，它是在线程创建时被赋值的。*ExitTime* 域包含了线程的退出时间，它是在线程退出函数中被赋值的。*LpcReplyChain* 和 *KeyedWaitChain* 是双链表的节点，分别用于跨进程通信（LPC）和带键事件的等待链表。由于这三个域在时间上不重叠，所以它们组成了一个 union。

ExitStatus 是线程的退出状态，*OfsChain* 域在 WRK 中没有用到。*PostBlockList* 域是一个双链表头节点，该链表中的各个节点类型为 PCM_POST_BLOCK，它被用于一个线程向配置管理器登记注册表键的变化通知。*TerminationPort* 域是一个链表头，当一个线程退出时，系统会通知所有已经登记过要接收其终止事件的那些端口。*ReaperLink* 是一个单链表节点，它仅在线程退出时使用。当线程被终止时，该节点将被挂到 *PsReaperListHead* 链表上，所以，在线程回收器（reaper）的工作项目（WorkItem）中该线程的内核栈得以收回。*KeyedWaitValue* 域是带键事件的键值。它和 *TerminationPort*、*ReaperLink* 域构成了一个 union，它们的有效时间不会重叠。

ActiveTimerListHead 域是一个双链表的头，链表中包含了当前线程的所有定时器。*ActiveTimerListLock* 域是操作此链表的自旋锁。*Cid* 是一个常用的域，其中包含了线程的唯一标识符，其类型为 CLIENT_ID。CLIENT_ID 包括两部分：UniqueProcess 和 UniqueThread，UniqueProcess 部分等于所属进程的 UniqueProcessId，而 UniqueThread 部分则等于此线程对象在进程句柄表中的句柄。

接下来的两个 union 域与 LPC 有关。*LpcReplySemaphore* 和 *KeyedWaitSemaphore* 域是两个信号量对象：*LpcReplySemaphore* 用于 LPC 应答通知；*KeyedWaitSemaphore* 用于处理带键的事件。*LpcReplyMessage* 域是一个指向 LPCP_MESSAGE 的指针，其中包含了 LPC 应答消息；*LpcWaitingOnPort* 域说明了在哪个端口对象上等待。*LpcReplyMessage* 和 *LpcWaitingOnPort* 两个域虽然共用同一个 union 空间，但是其最低位可用来区分应该用哪个，具体做法参见 base\ntos\lpc\lpcp.h 中的 LpcpGetThreadMessage 和 LpcpGetThreadPort 宏。关于 LPC 机制的详细描述，请参考本书 8.2 节。

ImpersonationInfo 域指向线程的模仿信息，Windows 允许一个线程在执行过程中模仿其他的用户来执行一段功能，这样可以实现更为灵活的访问控制安全特性。*IrpList* 域是一个双链表头，其中包含了当前线程所有正在处理但尚未完成的 I/O 请求（IRP 对象）。*TopLevelIrp* 域指向线程的顶级 IRP，它或者指向 NULL 或一个 IRP，或者包含了 fsrtl.h 中定义的标记 FSRTL_FAST_IO_TOP_LEVEL_IRP 或 FSRTL_FSP_TOP_LEVEL_IRP。仅当一个线程的 I/O 调用层次中最顶级的组件是文件系统时，*TopLevelIrp* 域才指向当前 IRP。*DeviceToVerify* 域指向一个待检验的设备对象，当磁盘或 CD-ROM 设备的驱动程序发现自

从上一次该线程访问该设备以来，该设备似乎有了变化，就会设置线程的 *DeviceToVerify* 域，从而使最高层的驱动程序，比如文件系统，可以检测到设备的变化。

下面列出 ETHREAD 结构的第二部分：

```
typedef struct _ETHREAD {

    ..... /* 前面已介绍了第一部分 */

    PEPROCESS ThreadsProcess;
    PVOID StartAddress;
    union {
        PVOID Win32StartAddress;
        ULONG LpcReceivedMessageId;
    };

    LIST_ENTRY ThreadListEntry;

    EX_RUNDOWN_REF RundownProtect;

    EX_PUSH_LOCK ThreadLock;

    ULONG LpcReplyMessageId;  // MessageId this thread is waiting for reply to

    ULONG ReadClusterSize;

    ACCESS_MASK GrantedAccess;

    union {
        ULONG CrossThreadFlags;
        struct { ..... };
    };

    union {
        ULONG SameThreadPassiveFlags;
        struct { ..... };
    };

    union {
        ULONG SameThreadApcFlags;
        struct { ..... };
    };

    BOOLEAN ForwardClusterOnly;
    BOOLEAN DisablePageFaultClustering;
    UCHAR ActiveFaultCount;
} ETHREAD, *PETHREAD;
```

ThreadsProcess 域指向当前线程所属的进程，这是在线程初始创建时赋值的。通过此域，可以很方便地从一个线程访问到它所属的进程。base\ntos\inc\ps.h 中的宏 THREAD_TO_PROCESS 正是利用此成员来实现的。*StartAddress* 域包含了线程的启动地址，这是真正的线程启动地址。*Win32StartAddress* 域包含的是 Windows 子系统的启动地

址。*StartAddress* 域包含的通常是系统 DLL 中的线程启动地址，因而往往是相同的（譬如 kernel32.dll 中的 BaseProcessStart 或 BaseThreadStart 函数）；而 *Win32StartAddress* 域中包含的才真正是 Windows 子系统接收到的线程启动地址，即 CreateThread API 函数接收到的线程启动地址。所以，*Win32StartAddress* 不仅对于 Windows 子系统有意义，对于诊断和分析不同线程的行为更有重要的意义。*LpcReceivedMessageId* 域包含了接收到的 LPC 消息的 ID，此域仅当 *SameThreadApcFlags* 域中的 *LpcReceivedMsgIdValid* 位被置上的时候才有效。由于 *LpcReceivedMessageId* 和 *Win32StartAddress* 域组成了一个 union，所以，当一个 Windows 子系统线程在接收 LPC 消息时，它的 *Win32StartAddress* 域也会被修改。

ThreadListEntry 域是一个双链表的节点，每个线程都会加入到它所属进程 EPROCESS 结构的 ThreadListHead 双链表中。正如前面所述，Windows 执行体也维护了进程与线程之间的从属关系。

RundownProtect 域是线程的停止保护锁，对于跨线程引用 TEB 结构或者挂起线程的执行等操作，需要获得此锁才能进行，以避免在操作过程中线程被销毁。*ThreadLock* 域是一把推锁，用于保护线程的数据属性，例如 PspLockThreadSecurityExclusive 和 PspLockThreadSecurityShared 利用该域来保护线程的安全属性。

LpcReplyMessageId 域指明了当前线程正在等待对一个消息的应答。*ReadClusterSize* 域指明了在一次 I/O 操作中读取多少个页面，用于页面交换文件和内存映射文件的读操作。*GrantedAccess* 域包含了线程的访问权限，这里的访问权限是一个位组合，各种权限定义参见 public\sdk\inc\ntpsapi.h 中的宏 THREAD_<*XXX*>，例如 THREAD_TERMINATE 代表终止线程的权限。

接下来三个 union：*CrossThreadFlags*、*SameThreadPassiveFlags* 和 *SameThreadApcFlags* 域包含了三组位标志。*CrossThreadFlags* 域是一些针对跨线程访问的标志位，包括 *Terminated*（线程已执行终止操作）、*DeadThread*（创建失败）、*HideFromDebugger*（该线程对于调试器不可见）、*ActiveImpersonationInfo*（线程正在模仿）、*SystemThread*（是一个系统线程）、*HardErrorsAreDisabled*（对于该线程，硬件错误无效）、*BreakOnTermination*（调试器在线程终止时停下该线程）、*SkipCreationMsg*（不向调试器发送创建消息）和 *SkipTerminationMsg*（不向调试器发送终止消息）。*SameThreadPassiveFlags* 域是一些只有在最低中断级别（被动级别）上才可以访问的标志位，并且只能被该线程自身访问，所以对这些标志位的访问不需要互锁操作。*SameThreadApcFlags* 域是一些在 APC 中断级别上被该线程自身访问的标志位，同样地，对这些标志位的访问也不需要互锁操作。

最后三个域与线程中发生的页面错误处理有关。*DisablePageFaultClustering* 域用于控制页面交换的聚集与否，*ForwardClusterOnly* 域指示是否仅仅前向聚集，*ActiveFaultCount* 域包含了正在进行之中的页面错误数量。

显然，执行体层的进程和线程对象涉及操作系统各个方面，它们不仅内嵌了内核层上的进程和线程对象，而且也包括更多涉及进程与线程管理的信息。在 EPROCESS 进程对象中，进程的唯一标识、映像文件、配额限制、一些重要的锁、与进程的物理内存和虚拟内存相关联的各种成员，还有各种标志，以及与安全有关的属性，这些构成了执行体层进程对象的主体。此外，进程对象也包含一条链表指向其所属的线程，从而与执行体线程对象关联起来。而在 ETHREAD 线程对象中，线程的唯一标识、启动地址、执行时间信息，以及在进行 LPC 跨进程通信时涉及的各种数据结构（包括锁），还有各种标志和安全属性，这些构成了执行体层线程对象的主体。

因此，不同于内核层的进程和线程对象偏重于基本的功能和机制，执行体层的进程和线程对象更侧重于管理和策略。下一节我们在介绍进程和线程管理时会涉及很多本节介绍的数据结构成员。

3.4 Windows 的进程和线程管理

我们已经知道了进程和线程在 Windows 内核中是如何表示的，在内核层和执行体层分别包含了哪些信息。现在我们来看 Windows 执行体是如何管理进程和线程的，包括进程和线程的创建和终止过程，以及系统的初始进程和线程是如何建立起来的。我们首先看一下在 Windows 进程中如何引用系统对象。

3.4.1 Windows 进程的句柄表

上一章介绍了 Windows 内核中的对象管理器，Windows 执行体实现了一套对象机制来管理各种资源或实体。每种对象都有一个类型对象，类型对象定义了该类对象的一些特性和方法。对象管理器也定义了一个全局名字空间，提供了根据名称来解析对象的统一机制（参考 ObpLookupObjectName 函数）。类型对象通过提供自定义的 Parse 方法可以扩展此名字空间。对象管理器中的对象是执行体对象，它们位于系统空间；在进程空间不能通过地址来引用它们。Windows 使用句柄（handle）来管理进程中的对象引用。

当一个进程利用名称来创建或打开一个对象时，将获得一个句柄，该句柄指向所创建或打开的对象。以后，该进程无须使用名称来引用该对象，使用此句柄即可访问。这样做

可以显著地提高引用对象的效率。句柄是一个在软件设计中被广泛使用的概念。例如，在 C 运行库中，文件操作使用句柄来表示，每当应用程序创建或打开一个文件时，只要此创建或打开操作成功，则 C 运行库返回一个句柄。以后应用程序对文件的读写操作都使用此句柄来标识该文件。而且，如果两个应用程序以共享方式打开了同一个文件，那么，它们将分别得到各自的句柄，且都可以通过句柄操作该文件。尽管两个应用程序得到的句柄的值并不相同，但是这两个句柄所指的文件却是同一个。因此，句柄只是一个对象引用，同一个对象在不同的环境下可能有不同的引用（句柄）值。

在 Windows 中，句柄是进程范围内的对象引用，换句话说，句柄仅在一个进程范围内才有效。一个进程中的句柄传递给另一个进程后，句柄值将不再有效。实际上，Windows 支持的句柄是一个索引，指向该句柄所在进程的句柄表（handle table）中的一个表项。在上一节我们看到，EPROCESS 数据结构的 *ObjectTable* 域指向进程的句柄表。句柄表的第一个索引为 4，第二个为 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。下面我们来看句柄表的实现结构。

在 Windows Server 2003 中，句柄表是一个多层结构。*ObjectTable* 的类型为 HANDLE_TABLE，每个句柄项的结构为 HANDLE_TABLE_ENTRY，它们的定义参见 base\ntos\inc\ex.h 文件。HANDLE_TABLE_ENTRY 的大小为 8 B，而且 Windows 执行体在分配句柄表内存时按页面（4 KB 大小）来申请内存。因此，执行体每申请一个新的页面来存放句柄项，则句柄表的容量增加 512。下面是 HANDLE_TABLE 结构的定义：

```
typedef struct _HANDLE_TABLE {
   ULONG_PTR TableCode;                        // 指向句柄表的存储结构，见下文介绍

   struct _EPROCESS *QuotaProcess;             // 句柄表的内存资源记录在此进程中
   HANDLE UniqueProcessId;                     // 创建进程的 ID，用于回调函数

   EX_PUSH_LOCK HandleTableLock[HANDLE_TABLE_LOCKS]; // HANDLE_TABLE_LOCKS=4
                                                     // 句柄表锁，仅在句柄表扩展时使用

   LIST_ENTRY HandleTableList;                 // 所有的句柄表形成一个链表
                                               // 链表头为全局变量 HandleTableListHead
   EX_PUSH_LOCK HandleContentionEvent;         // 若在访问句柄时发生竞争，则在此推锁上等待
   PHANDLE_TRACE_DEBUG_INFO DebugInfo;         // 调试信息，仅当调试句柄时才有意义

   LONG ExtraInfoPages;                        // 审计信息所占用的页面数量
   ULONG FirstFree;                            // 空闲链表表头的句柄索引，见下文介绍
   ULONG LastFree;                             // 最近被释放的句柄索引，用于 FIFO 类型空闲链表

   ULONG NextHandleNeedingPool;                // 下一次句柄表扩展的起始句柄索引
   LONG HandleCount;                           // 正在使用的句柄表项的数量

   union {
      ULONG Flags;                             // 标志域
```

```
        BOOLEAN StrictFIFO : 1;                    // 是否使用 FIFO 风格的重用，即先释放先重用
    };
} HANDLE_TABLE, *PHANDLE_TABLE;
```

HANDLE_TABLE 中的 *TableCode* 域是一个指针，指向句柄表的最高层表项页面，它的低 2 位的值代表了当前句柄表的层数。也就是说，如果 *TableCode* 的最低 2 位为 0，说明句柄表只有一层，此种情况下该进程最多只能容纳 512 个句柄（宏定义 LOWLEVEL_THRESHOLD）；如果 *TableCode* 的最低 2 位为 1，则说明句柄表有两层，此种情况下该进程可容纳的句柄数是 512×1 024（宏定义 MIDLEVEL_THRESHOLD），即 *TableCode* 指向一个中间层句柄表页面，该页面包含 1 024 个指针，每个指向一个最低层句柄表页面；如果 *TableCode* 的最低 2 位为 2，则说明句柄表有三层，此种情况下三层树结构可容纳的句柄数是 512×1 024×1 024（宏定义 HIGHLEVEL_THRESHOLD），但 Windows 执行体限定每个进程的句柄数不得超过 2^{24}=16 777 216（宏定义 MAX_HANDLES）。图 3.4 显示了这三种情形。实际上，在每个最低层句柄表页面中，第一个句柄表项都有特殊用途（参见本节后文介绍），真正供进程使用的句柄表项是 511 个。

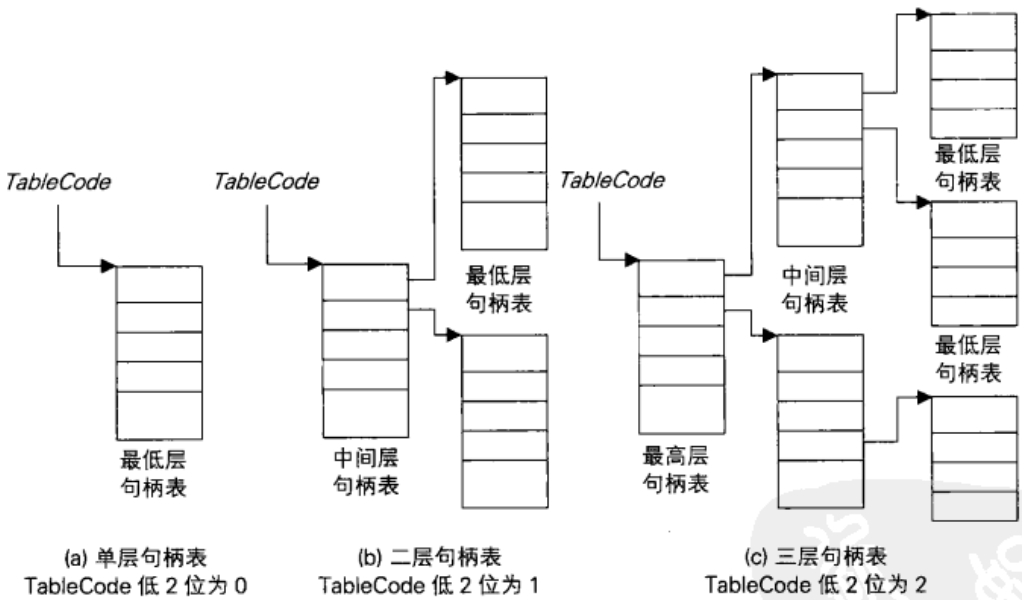


图 3.4 Windows 进程的句柄表结构

执行体在创建进程时，首先为新进程分配一个单层句柄表。句柄表的创建工作是通过调用 ExCreateHandleTable 函数来完成的，该函数调用 ExpAllocateHandleTable 来构造初始的句柄表。随着进程中句柄数量的增加，单层句柄表被扩展为二层句柄表，再进一步被扩展为三层句柄表。句柄表的扩展是由函数 ExpAllocateHandleTableEntrySlow 来实现的。这些函数的实现代码位于 base\ntos\ex\handle.c 文件中。

在句柄表 HANDLE_TABLE 结构中，*FirstFree* 域记录了当前句柄表中的空闲句柄链，这是一个单链表，但并非通过指针链接起来，而是用句柄索引值来链接。句柄索引值按 HANDLE_VALUE_INC 逐个递增，这是个宏定义，其值为 4。*FirstFree* 域指示了链表头的句柄索引值，HANDLE_TABLE_ENTRY 结构中的 *NextFreeTableEntry* 成员等于下一个空闲句柄的句柄索引值。因此，当进程在执行过程中需要创建新的句柄时，执行体可以直接从空闲句柄链表头得到一个句柄，新的链表头变成原来链表头的 *NextFreeTableEntry*，参见 ExpAllocateHandleTableEntry 函数的代码；而当释放句柄时，将待释放的句柄索引赋给 *FirstFree*，且该句柄项的 *NextFreeTableEntry* 赋为原来的 *FirstFree*，参见 ExpFreeHandleTableEntry 函数的代码。另外，HANDLE_TABLE 结构的 *NextHandleNeedingPool* 域记录了下一次句柄表扩展的起始句柄索引，相当于当前句柄表中所有已分配页面都满了以后下一个页面的起始句柄索引。所以，Windows 进程的句柄表只是简单地线性增长，但只有当确实不够用的时候才会增长。

理解了 Windows 进程句柄表的结构以后，我们来看一下句柄表项的内容和句柄本身。句柄表项的类型为 HANDLE_TABLE_ENTRY，虽然只包含两个 32 位的 union，但是其含义非常丰富。HANDLE_TABLE_ENTRY 的定义如下：

```
typedef struct _HANDLE_TABLE_ENTRY {
    union {
        PVOID Object;                          // 指向句柄所代表的对象
        ULONG ObAttributes;                    // 最低三位有特别含义，参见
                                               // OBJ_HANDLE_ATTRIBUTES 宏定义
        PHANDLE_TABLE_ENTRY_INFO InfoTable;    // 各个句柄表页面的第一个表项
                                               // 使用此成员指向一张表
        ULONG_PTR Value;
    };

    union {
        union {
            ACCESS_MASK GrantedAccess;         // 访问掩码

            struct {                           // 当 NtGlobalFlag 中包含
                                               // FLG_KERNEL_STACK_TRACE_DB 标记时使用
                USHORT GrantedAccessIndex;
                USHORT CreatorBackTraceIndex;
            };
        };
        LONG NextFreeTableEntry;               // 空闲时表示下一个空闲句柄索引
    };
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

第一个 union 的关键成员是 *Object* 指针，指向句柄所代表的内核对象，它的最低 3 位有特别含义：第 0 位 OBJ_PROTECT_CLOSE，表示调用者是否允许关闭该句柄；第 1 位 OBJ_INHERIT，指示该进程所创建的子进程是否可以继承该句柄，即是否将该句柄项拷

贝到它们的句柄表中；第 2 位 OBJ_AUDIT_OBJECT_CLOSE，指示关闭该对象时是否产生一个审计事件。在第二个 union 中，如果句柄表项指向一个有效的对象，那么，*GrantedAccess* 成员记录了该句柄的访问掩码；如果这是一个空闲的句柄表项，那么，*NextFreeTableEntry* 成员将加入到句柄表的空闲单链表中。

一种特殊的情形是，对于每个最低层句柄表页面中的第一个句柄表项，它的 *NextFreeTableEntry* 成员等于 EX_ADDITIONAL_INFO_SIGNATURE，即–2，并且第一个 union 中的 *InfoTable* 成员指向一张 HANDLE_TABLE_ENTRY_INFO 表，其中每一项代表了该页面中对应序号处的句柄表项的审计掩码。

另外值得一提的是，在 WRK 代码中，关于 *Object* 域的最低位，即第 0 位，作为 OBJ_PROTECT_CLOSE 来解释有一点微妙。实际上，关闭保护位被转移到了 *GrantedAccess* 成员中，而 *Object* 域的第 0 位变成了句柄表项的锁标志。如何将关闭保护位转移到 *GrantedAccess* 成员中呢？可以参考宏 ObpEncodeProtectClose 和 ObpGetHandleAttributes。因此，事实上，当 *Object* 域的第 0 位为 1 时，表示此句柄表项已被锁住；否则表示该项未被锁住。

现在我们来看如何将一个句柄解析成相应的内核对象。首先，一个有效的句柄有 4 种可能：

- –1，代表当前进程。
- –2，代表当前线程。
- 负值，其绝对值为内核句柄表中的索引。仅限于内核模式的函数可以引用。
- 不超过 226 的正值，当前进程的句柄表中的索引。

这里的内核句柄表是指系统空间中的全局句柄表，在 WRK 中即变量 ObpKernelHandleTable，也是 System 进程的句柄表。ObpKernelHandleTable 中的句柄只有在内核模式下才可以被引用，但可以位于任何一个进程环境中。所以，这些句柄实际上是跨进程环境的。有些系统组件和设备驱动程序可以利用这样的设施来创建或打开可在任何进程环境下访问的句柄，但是又不让用户模式代码引用它们。

解析句柄的基本函数是 ObReferenceObjectByHandle，其代码位于 base\ntos\ob\obref.c 文件中的 1 060~1 351 行。该函数首先对接收到的 Handle 参数进行判断，看属于以上四种情形的哪一种。对于前两种情形，无须查找句柄表，在简单地检查进程或线程的访问权限以后，即可返回所引用的进程对象或线程对象。对于后两种情形，首先调用执行体函数

ExMapHandleToPointerEx，按照句柄表的结构，找到该句柄的索引值所指的句柄表项，然后检查句柄表项中的访问权限是否满足要求，并根据需要对这次对象访问进行审计（调用 ObpAuditObjectAccess），最后返回对象体。ExMapHandleToPointerEx 函数调用 ExpLookupHandleTableEntry 来查找句柄表项，而 ExpLookupHandleTableEntry 则直截了当地根据图 3.4 所示的结构查找到最低层句柄表页面中的表项。

将一个对象插入到句柄表中的函数是 ObInsertObject，其代码位于 base\ntos\ob\obinsert.c 文件中。它首先对参数做各种检查，然后调用 ObpCreateHandle 函数，为要插入的对象创建一个句柄。ObpCreateHandle 函数又调用 ExCreateHandle 来创建一个句柄表项，并填充新建的表项。这些函数的代码比较直截了当，这里不再赘述。

另外值得一提的是对象的引用计数。正如 2.5.1 节的最后部分所讲，对象的引用有两种来源：第一，在内核中直接通过对象地址来引用，这是通过 ObReferenceObjectByPointer 来记录一次新的引用；第二，通过句柄来引用对象，这是由 ObpIncrementHandleCount 函数来检查并记录一次句柄引用。对于一个句柄，它的生命周期从被插入到句柄表中开始，一直到它被关闭，在此过程中，对象的引用计数中包含有它的一份引用。在 WRK 代码中，我们可以看到，ObpCreateHandle 调用了 ObpIncrementHandleCount，标志着新创建的句柄引用了该对象；ObpCloseHandle 函数调用了 ObpCloseHandleTableEntry，而后者又进一步调用了 ObpDecrementHandleCount，标志着一个句柄结束了相应对象的引用。另外，在一个句柄上调用了 ObReferenceObjectByHandle 函数以后，若该对象指针不再使用，则必须调用 ObDereferenceObject 函数。

句柄表除了作为对象引用的容器以外，还有另一个用法：作为分配进程和线程的唯一 ID 的有效手段。进程有一个唯一 ID，称为 *UniqueProcessId*；线程有一个 CLIENT_ID 成员 *Cid*，其中包含了所属进程的唯一 ID 和线程自身的唯一 ID。这些唯一 ID 是怎么生成的呢？它们是通过调用 ExCreateHandle 函数，在一个全局的句柄表 PspCidTable 中创建的句柄索引值。此句柄表也称为 CID 句柄表（Client ID handle table），它没有被加入到系统的句柄表链表中。CID 句柄表中的每个句柄表项都包含了进程或线程的对象地址。

因此，进程和线程的唯一 ID 值都是 4 的倍数，其中 4 是第一个句柄索引值，它被分配给 System 进程的唯一 ID（因为 System 进程是第一个通过 PspCreateProcess 函数创建的进程）。0 是专门保留给空闲进程的，它并非通过 ExCreateHandle 函数而获得。CID 句柄表严格按照 FIFO 来重用句柄表项，所以，一个句柄表项被释放以后，要等到其他的空闲句柄表项都被重用一遍以后才会被再次使用。由于 CID 句柄表项保存了进程或线程的对象地址，所以，在内核中，根据进程或线程的唯一 ID 值，总是可以很方便地找到相应的

对象地址，函数 PsLookupProcessThreadByCid、PsLookupProcessByProcessId 和 PsLookup-ThreadByThreadId 正是利用了 CID 句柄表的这一能力，参见 base\ntos\ps\pscid.c 文件中的代码。

3.4.2 获得当前线程或进程

在阅读 WRK 源代码时，我们会频繁地看到许多有关当前线程或进程的基本操作。内核层函数 KeGetCurrentThread 是一个重要的函数，它返回当前处理器上正在运行的线程的 KTHREAD 结构指针。通过此结构信息，可以进一步得到 KPROCESS、ETHREAD 和 EPROCESS 结构。在 WRK 中，KeGetCurrentThread 是这样实现的：

```
FORCEINLINE
struct _KTHREAD *
NTAPI KeGetCurrentThread (VOID)
{
#if (_MSC_FULL_VER >= 13012035)
   return (struct _KTHREAD *) (ULONG_PTR) __readfsdword (FIELD_OFFSET
       (KPCR, PrcbData.CurrentThread));
#else
   __asm { mov eax, fs:[0] KPCR.PrcbData.CurrentThread }
#endif
}
```

WRK 本身带的编译器（tools\x86\cl.exe）版本（13.10.4035）大于 13012035，所以，上面的代码取第一个条件分支，它展开来实际上只是一条指令。在 Intel x86 处理器的 Windows 系统中，fs 寄存器指向一块被称为处理器控制区（PCR，Processor Control Region）的内存，其数据类型为 KPCR。KPCR 有一个类型为 KPRCB 的数据成员 PrcbData，这是当前处理器的控制块（Processor Control Block），其中包含了指向当前线程的 KTHREAD 结构的指针。关于 KPCR 和 KPRCB 结构的定义，请参考 base\ntos\inc\i386.h 中的代码。因此，上面的 KeGetCurrentThread 代码实际上是一条访问 fs 寄存器加上特定偏移的指令。

获得了当前线程的 KTHREAD 结构指针以后，便可以很方便地获得 ETHREAD 结构的指针，以及进程 KPROCESS 或 EPROCESS 结构的指针。在执行体层上获得当前线程和进程的函数分别是 PsGetCurrentThread 和 PsGetCurrentProcess，代码如下：

```
#define _PsGetCurrentProcess() (CONTAINING_RECORD \
     (((KeGetCurrentThread())->ApcState.Process),EPROCESS,Pcb))

#define _PsGetCurrentThread() ((PETHREAD)KeGetCurrentThread())

PEPROCESS
PsGetCurrentProcess(
   VOID
   )
```

```
{
    return _PsGetCurrentProcess();
}

PETHREAD
PsGetCurrentThread(
    VOID
    )
{
    return _PsGetCurrentThread();
}
```

我们看到，_PsGetCurrentProcess 宏从当前线程 KTHREAD 结构的 ApcState 成员中获得当前线程所属进程的 KPROCESS 结构。这里之所以从 ApcState 成员中获得进程结构指针，而不是从 KTHREAD 的 Process 域或 ETHREAD 的 ThreadsProcess 域获取进程结构指针，是因为即使当前线程附载到其他的进程中（通过 KeAttachProcess 函数），或者又回到原先的进程中（通过 KeDetachProcess 函数），这种做法也总是能够获得正确的当前进程结构的指针。请参考 5.2.6 节关于 APC 环境的描述。

3.4.3 进程和线程的创建过程

在内核中，Windows 创建一个进程的过程是从 NtCreateProcess 函数开始的，它首先创建一个执行体进程对象，即 EPROCESS 对象，然后创建一个初始线程，为初始线程建立一个栈，并设置好它的初始执行环境。完成这些工作以后，该线程就可以参与系统的线程调度了。然而，通过 Windows API 函数创建的进程也要接受 Windows 子系统的管理，在这种情况下，仅仅内核部分的工作还不够，系统在创建进程过程中，还需要跟子系统打交道。另外，建立起独立的内存地址空间是 Windows 创建进程过程中不可避免的步骤，关于进程地址空间的创建，请参考 4.3.1 节。

NtCreateProcess 函数的代码位于 base\ntos\ps\create.c 文件中（815~850 行），它只是简单地对参数稍作处理，然后把创建进程的任务交给 NtCreateProcessEx 函数（位于同一文件中，852~917 行），所以我们来看 NtCreateProcessEx 的原型及其流程。

```
NTSTATUS
NtCreateProcessEx(
    __out PHANDLE ProcessHandle,
    __in ACCESS_MASK DesiredAccess,
    __in_opt POBJECT_ATTRIBUTES ObjectAttributes,
    __in HANDLE ParentProcess,
    __in ULONG Flags,
    __in_opt HANDLE SectionHandle,
    __in_opt HANDLE DebugPort,
    __in_opt HANDLE ExceptionPort,
    __in ULONG JobMemberLevel
    );
```

NtCreateProcessEx 的参数并不复杂，ProcessHandle 是一个输出参数，如果创建成功，则它包含了所创建的进程的句柄。DesiredAccess 包含了对新进程的访问权限。ObjectAttributes 是一个可选的指针参数（意指可以为 NULL），它指定了进程对象的属性。ParentProcess 指向父进程的句柄，这是一个必需的参数，不能为 NULL，并且调用者必须对该进程具有 PROCESS_CREATE_PROCESS 访问权限。Flags 是创建标志，其中有一个标志 PROCESS_CREATE_FLAGS_INHERIT_HANDLES 特别值得一提：NtCreateProcess 专门有一个布尔参数指定是否该标志为 TRUE，表明新进程的对象句柄表是否要复制父进程的句柄表，或者初始设置为空。SectionHandle 是一个可选的句柄，指向一个内存区对象，代表了该进程的映像文件，调用者对于此内存区对象必须具有 SECTION_MAP_EXECUTE 访问权限。DebugPort 也是一个可选的句柄，指向一个端口对象，如果此句柄参数不为 NULL，则此端口被赋为新进程的调试端口，否则，新进程没有调试端口。而且，调用者对于调试端口对象必须具有 PORT_WRITE 和 PORT_READ 访问权限。类似地，ExceptionPort 也是一个可选的句柄，指向一个端口对象，如果此句柄参数不为 NULL，则此端口被赋为新进程的异常端口，否则，新进程没有异常端口。调用者对于异常端口对象必须具有 PORT_WRITE 和 PORT_READ 访问权限。JobMemberLevel 指定了要创建的进程在一个 Job 集中的级别。

NtCreateProcessEx 函数的代码只是简单地检查 ProcessHandle 参数代表的句柄是否可写，然后把真正的创建工作交给 PspCreateProcess 函数，所以，PspCreateProcess 才是真正创建进程的函数，它也位于文件 base\ntos\ps\create.c 中（966~1 758 行）。PspCreateProcess 被三个函数调用，它们是 NtCreateProcessEx、PsCreateSystemProcess 和 PspInitPhase0。其中 PspInitPhase0 是在系统初始化的早期被调用的，它创建的进程（即 System 进程）的句柄保存在全局变量 PspInitialSystemProcessHandle 中，进程对象存放于全局变量 PsInitialSystemProcess 中（参考后面 3.4.5 节的介绍）。PsCreateSystemProcess 可用于创建系统进程对象，它创建的进程都是 PsInitialSystemProcess 的子进程。所以，PspCreateProcess 函数负责创建系统中的所有进程，包括 System 进程。下面介绍此函数的基本流程。

(1) 如果父进程句柄不为 NULL，则通过 ObReferenceObjectByHandle 获得父进程对象的 EPROCESS 指针，放在 Parent 局部变量中，同时也获得父进程的 Affinity 设置。新进程的工作集最大/最小值被初始化为全局变量 PsMinimumWorkingSet 和 PsMaximumWorkingSet。如果父进程句柄为 NULL，则 Affinity 设置为全局变量 KeActiveProcessors，即系统中当前的可用处理器。因为新进程对象尚未创建，所以这些设置都保存在局部变量中。

(2) 调用 ObCreateObject 函数，创建一个类型为 PsProcessType 的内核对象，置于局部变量 Process 中，其对象体为 EPROCESS 数据结构。

(3) 把 Process 对象中的所有域置为 0，然后初始化其中部分成员（见 1 131~1 153 行）。

(4) 检查内存区句柄参数 SectionHandle，对于系统进程，此参数为 NULL，此时，除非父进程为 PsInitialSystemProcess，否则内存区对象继承自父进程，并且不得为 NULL。如果此参数不为 NULL，则利用此句柄参数调用 ObReferenceObjectByHandle 获得内存区对象的指针。所以，新进程对象的内存区对象已经完成初始化。

(5) 接下来根据 DebugPort 参数来初始化新进程对象的 DebugPort 成员。

(6) 同样地，根据 ExceptionPort 参数来初始化新进程对象的 ExceptionPort 成员。

(7) 如果指定的父进程不为 NULL，则创建一个全新的地址空间；否则（对于系统进程），让新进程的句柄表指向当前进程的句柄表，并且利用空闲线程的地址空间来初始化新进程的地址空间。

(8) 然后调用 KeInitializeProcess 函数来初始化新进程对象的基本优先级、Affinity、进程页表目录和超空间的页帧号。

(9) 通过 PspInitializeProcessSecurity 函数初始化新进程的安全属性，主要是从父进程复制一个令牌。

(10) 接下来设置新进程的优先级类别。如果父进程不为 NULL，则拷贝父进程的优先级类别，并且初始化新进程的句柄表，若 Flags 参数中包含了句柄继承标志，则把父进程句柄表中凡是有继承属性的对象拷贝到新进程句柄表中。

(11) 接下来初始化新进程的进程地址空间。有三种可能性：

a. 新进程有新的可执行映像内存区对象。调用 MmInitializeProcessAddressSpace 函数，根据指定的内存区对象来初始化进程地址空间。

b. 没有指定映像内存区对象，但指定了父进程，而父进程并非 PsInitialSystemProcess。也调用 MmInitializeProcessAddressSpace 函数，但根据父进程来初始化进程地址空间。并且把父进程的映像名称字符串拷贝到新进程对象的数据结构中。

c. 没有指定映像内存区对象，但指定了 PsInitialSystemProcess 作为父进程。这对应于系统进程的情形。调用 MmInitializeProcessAddressSpace，不指定内存区和父进程，直接初始化。同样地，把父进程的映像名称字符串拷贝到新进程对象的数据结构中。

实际上还有第四种可能性，即没有指定映像内存区对象，也没有指定父进程。这对应于系统的引导进程（后蜕化成空闲进程），它的地址空间是在 MmInitSystem 执行过程中初始化的，由 MiInitMachineDependent 函数调用 MmInitializeProcessAddressSpace 来完成。

(12) 创建进程 ID。利用 ExCreateHandle 函数在 CID 句柄表中创建一个进程 ID 项。

(13) 对这次进程创建行为进行审计。

(14) 如果父进程属于一个作业对象，则也加入到父进程所在的作业中。

(15) 对于通过映像内存区对象来创建进程的情形，创建一个 PEB；对于进程拷贝（fork）的情形，则使用继承的 PEB。

(16) 把新进程加入到全局的进程链表 PsActiveProcessHead 中。

(17) 调用 ObInsertObject 函数，把新进程对象插入到当前进程的句柄表中。

(18) 接下来计算新进程的基本优先级和时限重置值，并且设置进程的内存优先级。这是通过调用 PspComputeQuantumAndPriority 函数来完成的。

(19) 然后设置进程的访问权限，即 GrantedAccess 域。由于新进程已经被加入到句柄表中，所以它现在能够被终止了（PROCESS_TERMINATE 权限）。对于有父进程但父进程不是 PsInitialSystemProcess 的进程，首先执行访问检查，然后计算进程的访问权限。如果是 PsInitialSystemProcess 的子进程，则授予所有的访问权限。

(20) 最后，设置进程的创建时间。并把新进程的句柄赋到输出参数 ProcessHandle 中，从而创建者可以获得新进程的句柄。

以上是创建并初始化一个进程对象的过程。然而，经过 PspCreateProcess 函数以后，新建的进程中并没有任何线程对象，所以，它现在还只是一个死的进程空间，也就是说，其中的代码并没有被真正运行起来。接下来我们讨论线程对象的创建和初始化。

类似于进程的创建过程，线程的创建是从 NtCreateThread 函数开始的，它也位于 base\ntos\ps\create.c 文件中（77~169 行）。其原型如下：

```
NTSTATUS
NtCreateThread(
    __out PHANDLE ThreadHandle,
    __in ACCESS_MASK DesiredAccess,
    __in_opt POBJECT_ATTRIBUTES ObjectAttributes,
    __in HANDLE ProcessHandle,
    __out PCLIENT_ID ClientId,
    __in PCONTEXT ThreadContext,
    __in PINITIAL_TEB InitialTeb,
    __in BOOLEAN CreateSuspended
    );
```

NtCreateThread 所做的事情很简单，首先，对于非内核模式传递过来的调用，检查几个参数是否可写，包括输出参数 ThreadHandle、ClientId 和输入参数 ThreadContext 及 InitialTeb；其次，处理 InitialTeb 参数，将它放到局部变量 CapturedInitialTeb 中。这些处理工作是在一个 try 块中完成的，所以，其中发生的异常能够被捕捉住，以保证内核的健壮性。完成了这些参数处理以后，NtCreateThread 调用真正创建线程的函数 PspCreateThread（位于同一文件中，242~813 行），以下是 PspCreateThread 的原型：

```
NTSTATUS
PspCreateThread(
   OUT PHANDLE ThreadHandle,
   IN ACCESS_MASK DesiredAccess,
   IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
   IN HANDLE ProcessHandle,
   IN PEPROCESS ProcessPointer,
   OUT PCLIENT_ID ClientId OPTIONAL,
   IN PCONTEXT ThreadContext OPTIONAL,
   IN PINITIAL_TEB InitialTeb OPTIONAL,
   IN BOOLEAN CreateSuspended,
   IN PKSTART_ROUTINE StartRoutine OPTIONAL,
   IN PVOID StartContext
   );
```

PspCreateThread 的参数虽然多，但是并不复杂。ThreadHandle 是一个输出参数，如果创建成功，则它包含了所创建的线程的句柄。DesiredAccess 包含了对新线程的访问权限。ObjectAttributes 是一个可选的指针（意指可以为 NULL），它指定了新线程对象的属性。ProcessHandle 指向一个进程的句柄，新线程将运行在此进程的环境中。ProcessPointer 参数指向所属进程的 EPROCESS 对象，此参数仅当创建系统线程时才会指向全局的 PsInitialSystemProcess 对象，其他情况下为 NULL。ClientId 参数返回新线程的 CLIENT_ID 结构。ThreadContext 参数提供了新线程的执行环境，它代表了用户模式线程的初始执行环境，如果该参数为 NULL，则说明此次调用将创建一个系统线程。InitialTeb 参数为新线程的 TEB 结构提供初始值。CreateSuspended 参数是一个布尔值，指明了新线程被创建起来以后是否被挂起。如果 CreateSuspended 参数为 TRUE，则意味着新线程创建完成以后并不立即执行，以后必须通过 NtResumeThread 函数让它开始运行。StartRoutine 参数指定了系统线程启动函数的地址。StartContext 参数指定了系统线程启动函数的执行环境。

PspCreateThread 函数仅仅被 NtCreateThread 和 PsCreateSystemThread 这两个函数调用，分别用于创建用户线程和系统线程对象。在 PspCreateThread 函数的参数中，ThreadContext 和 InitialTeb 参数针对用户线程的创建操作，而 StartRoutine 和 StartContext 参数则针对系统线程的创建操作。下面分析 PspCreateThread 函数的执行流程，读者可以参考 base\ntos\ps\create.c 文件中的代码。

(1) 首先获得当前线程对象，以及此次创建操作来自于内核模式还是用户模式（即 PreviousMode）。然后根据进程句柄参数 ProcessHandle 获得相应的进程对象，放到局部变量 Process 中。

(2) 调用 ObCreateObject 函数创建一个线程对象 ETHREAD。并把整个 ETHREAD 结构清零。

(3) 对一些基本的域进行初始化，包括 RundownProtect、ThreadsProcess（指向由进程句柄参数解出来的 EPROCESS 对象）、Cid（包括 UniqueProcess 和 UniqueThread 成员）。这里 Cid.UniqueProcess 是从 Process 对象中来的，而 Cid.UniqueThread 则是通过调用 ExCreateHandle 函数在 CID 句柄表中创建一个句柄表项而获得的。

(4) 继续初始化新线程对象 ETHREAD 结构中的一些域（409~441 行），包括 ReadClusterSize、LpcReplySemaphore、LpcReplyChain、IrpList、PostBlockList 成员，以及线程锁成员 ThreadLock 和 ActiveTimerListLock，并且初始化新线程的定时器链表 ActiveTimerListHead 成员。

(5) 获得进程的 RundownProtect 锁，以避免在创建过程中该进程被停掉（rundown）。直到该线程被插入到进程的线程链表中（通过调用 KeStartThread 函数）以后，PspCreateThread 才释放该锁。

(6) 如果 ThreadContext 非 NULL，则此次创建的是用户模式线程，于是创建一个 TEB，并且用 InitialTeb 进行初始化。然后，利用 ThreadContext 中的程序指针（Eip 寄存器）来设置线程的启动地址 StartAddress 域，并且将 ThreadContext 中的 Eax 寄存器设置到线程的 Win32StartAddress 域。完成了这些操作以后，调用 KeInitThread 函数，根据进程对象中的信息来初始化新线程的一些属性，包括跟同步相关的各个域，例如同步头（Header 域）、WaitBlock 的初始化，等等。线程的系统服务表（ServiceTable 域）以及与 APC 和定时器相关的域也在这个函数中被初始化。此外，线程的内核栈也在这个函数中被创建并初始化。最后，KeInitThread 函数根据所提供的参数信息调用 KiInitializeContextThread，以便完成与特定处理器相关的执行环境的初始化。因此，当这个函数返回时，新线程的状态是“已初始化（Initialized）”。

(7) 否则，则是系统线程。首先在 CrossThreadFlags 标志中设置系统线程位。然后将线程的启动地址设置为 StartRoutine 参数。最后同样地调用 KeInitThread 函数。所以，KeInitThread 函数既可以被用来初始化用户模式线程，也可以被用于初始化系统线程。

(8) 接下来锁住进程，并确保此进程并不是在退出或终止过程中。

(9) 然后进程的活动线程数加 1，并且将新线程加入到进程的线程链表中。接着调用 KeStartThread 函数，初始化剩余的域，尤其是跟调度相关的域，比如优先级、时限设置、处理器亲和性，等等。经过这一步以后，新线程就可以开始运行了。注意，在 PspCreateThread 和 KeStartThread 这两个函数中，我们都可以看到"InsertTailList (&Process->ThreadListHead, &Thread->ThreadListEntry);"这样的调用，实际上，这是分别在执行体层和内核层维护线程与进程之间的关系。也就是说，这里 Process 和 Thread 分别对应于不同的结构类型，即 EPROCESS 或 KPROCESS，以及 ETHREAD 或 KTHREAD。

(10) 接下来，若这是进程中的第一个线程，则触发该进程的创建通知（见 576~596 行）。

(11) 如果新线程的进程在一个作业中，则需要做特定的处理（见 607~625 行）。

(12) 通知那些接收线程创建事件的出调例程（callout routine）（见 633~649 行）。

(13) 线程对象的引用计数加 2，一个针对当前的创建操作，另一个针对要返回的线程句柄。

(14) 如果 CreateSuspended 参数指示新线程立即被挂起，则调用 KeSuspendThread 挂起新线程。

(15) 根据指定的期望访问权限，调用 SeCreateAccessStateEx 函数创建一个访问状态结构（ACCESS_STATE）。

(16) 调用 ObInsertObject 函数把新线程对象插入到当前进程的句柄表中。ObInsertObject 调用如果不成功，则终止新线程；否则，设置好输出参数 ThreadHandle 和 ClientId，准备返回。

(17) 设置线程的创建时间。

(18) 然后设置线程的访问权限，即 GrantedAccess 域（见 754~807 行）。

(19) 最后，调用 KeReadyThread 函数，使新线程进入"就绪（ready）"状态，准备马上执行；或者，若此时进程尚未在内存中，则新线程的状态为"转移（transition）"，以等待换入内存后再执行。

(20) 引用计数减 1，当前操作完成。返回。

我们从上述步骤可以看到，一旦 PspCreateThread 函数返回，新线程对象的状态已经完全设置好，它可被马上执行。因为线程的创建是在进程已经创建完成以后才做的动作，所以，线程创建是一个相对简单的过程。而完整的进程创建过程其实并不像前面介

绍的步骤那么直截了当。譬如，我们在 PspCreateProcess 中根本没有看到任何创建线程的动作，甚至，我们也没有看到进程的可执行映像文件是怎么打开的。不过我们在 PspCreateProcess 的参数中看到了对应于可执行映像文件的内存区对象的句柄。在 WRK 中并不能看到完整的进程创建过程，但是理解这一过程仍然是非常必要的，下面我们从上层应用程序的角度来讨论进程的创建全过程。

为了创建一个进程，在 Windows 中最常见的手段是调用某个 API 函数，比如 CreateProcess，此函数一旦成功返回，则新的进程便已建立起来。CreateProcess 只是 kernel32.dll 中的一个函数而已，当应用程序调用此函数时，它不仅会调用执行体层的进程对象创建函数（即前面介绍的 NtCreateProcess 或 NtCreateProcessEx 函数），还需要跟 Windows 子系统打交道，以便让 Windows 子系统参与进程的管理。我们在前面介绍进程和线程数据结构时已经看到了有些域是专门为 Windows 子系统预留的。接下来我们讨论创建 Windows 子系统进程的完整步骤：

(1) 首先用 CreateProcess（实际上是 CreateProcessW）打开指定的可执行映像文件，并创建一个内存区对象。注意，内存区对象并没有被映射到内存中（由于目标进程尚未建立起来，不可能完成内存映射），但它确实是打开了。

(2) 接下来调用内核中的 NtCreateProcessEx 系统服务，实际的调用过程是这样的：kernel32.dll 中的 CreateProcessW 调用 ntdll.dll 中的存根函数 NtCreateProcessEx，而 ntdll.dll 的 NtCreateProcessEx 利用处理器的陷阱机制切换到内核模式下；在内核模式下，系统服务分发函数 KiSystemService 获得控制，它利用当前线程指定的系统服务表，调用到执行体层的 NtCreateProcessEx 函数。然后，执行体层的 NtCreateProcessEx 函数执行前面介绍的进程创建逻辑，包括创建 EPROCESS 对象、初始化其中的域、创建初始的进程地址空间、创建和初始化句柄表，并设置好 EPROCESS 和 KPROCESS 中的各种属性，如进程优先级、安全属性、创建时间等。到这里，执行体层的进程对象已经建立起来，进程的地址空间已经初始化，并且 EPROCESS 中的 PEB 也已初始化。

(3) 现在，虽然进程对象已经建立起来，但是它没有线程，所以，它自己还不能做任何事情。接下来需要创建一个初始线程，在此之前，首先要构造一个栈以及一个可供运行的环境。初始线程的栈的大小可以通过映像文件获得，而创建线程则可以通过调用 ntdll.dll 中的 NtCreateThread 函数来完成。Ntdll.dll 中的 NtCreateThread 又把任务转发给执行体层的 NtCreateThread，即前面刚刚介绍的线程创建逻辑，包括创建 ETHREAD 对象、初始化其中的域、生成线程 ID、建立 TEB 和设置线程的安全属性等工作。进程的第一个线程的启动函数是 kernel32.dll 中的 BaseProcessStart 函数。然而，这里新创建的

线程不会立即运行，它处于挂起状态，要等到进程完全初始化以后才真正开始运行。

(4) 到现在，从内核角度来看，进程对象和第一个线程对象已经创建起来了，但是，从子系统的角度而言，进程创建才刚刚开始。Kernel32.dll 给 Windows 子系统发送一个消息，消息中包括进程和线程的句柄、进程创建者的 ID 等必要的信息。Windows 子系统进程（csrss.exe）接收到此消息，执行以下一系列动作：

a. 保留一份句柄。

b. 设定新进程的优先级类别。

c. 在子系统中分配一个内部进程块。

d. 设置新进程的异常端口，从而子系统可以接收到该进程中发生的异常。

e. 对于正在被调试的进程，设置它的调试端口，从而子系统可以接收到该进程的调试事件。

f. 分配并初始化一个内部线程块，并插入到进程的线程列表中。

g. 窗口会话中的进程计数增 1。

h. 设置进程的停机级别为默认级别。

i. 将新进程插入到子系统的进程列表中。

j. 分配并初始化一块内存供子系统的内核模式部分使用（W32PROCESS 结构）。

k. 显示应用程序启动光标。

(5) 到这时候，进程环境已经建好，其线程将要使用的资源也分配好了，Windows 子系统已经知道并登记了此进程和线程。所以，初始线程被恢复执行，余下部分的初始化工作是初始线程在新进程环境中完成的。在内核中，新线程的启动例程是 KiThreadStartup 函数，这是当 PspCreateThread 调用 KeInitThread 函数时，KeInitThread 函数调用 KiInitializeContextThread（参见 base\ntos\ke\i386\thredini.c 文件）来设置的。现在回到 WRK 中，读者可以看到 KiThreadStartup 函数的代码。这是一段简单的汇编代码，如下所示：

```
cPublicProc _KiThreadStartup    ,1

        xor     ebx,ebx                 ; clear registers
        xor     esi,esi                 ;
        xor     edi,edi                 ;
        xor     ebp,ebp                 ;
        LowerIrql APC_LEVEL             ; KeLowerIrql(APC_LEVEL)
```

```
        pop     eax                     ; (eax)->SystemRoutine
        call    eax                     ; SystemRoutine(StartRoutine, StartContext)

        pop     ecx                     ; (ecx) = UserContextFlag
        or      ecx, ecx
        jz      short kits10            ; No user context, go bugcheck

        mov     ebp,esp                 ; (bp) -> TrapFrame holding UserContext

        jmp     _KiServiceExit2

kits10: stdCall _KeBugCheck, <NO_USER_MODE_CONTEXT>

stdENDP _KiThreadStartup
```

KiThreadStartup 函数首先将 IRQL 降低到 APC_LEVEL，然后调用系统初始的线程函数 PspUserThreadStartup。这里的 PspUserThreadStartup 函数是 PspCreateThread 函数在调用 KeInitThread 时指定的，参见 base\ntos\ps\create.c 的 490 行代码。注意，PspCreateThread 函数在创建系统线程时指定的初始线程函数为 PspSystemThreadStartup（见 base\ntos\ps\create.c 的 514 行代码）。线程启动函数被作为一个参数传递给 PspUserThreadStartup，在这里，它应该是 kernel32.dll 中的 BaseProcessStart。

(6) PspUserThreadStartup 函数被调用，其代码位于 base\ntos\ps\create.c 的 2 002~2 147 行。逻辑并不复杂，但是涉及异步函数调用（APC）机制。以下是它的基本流程：

a. 获得当前线程和进程对象。

b. 是否由于创建过程中出错而需要终止本线程。

c. 如果需要，通知调试器。

d. 如果这是进程中的第一个线程，则判断系统是否支持应用程序预取的特性，如果是，则通知缓存管理器预取可执行映像文件中的页面（见 2 106 行的 CcPfBeginAppLaunch 调用）。所谓应用程序预取，是指将该进程上一次启动的前 10 s 内引用到的页面直接读入到内存中。

e. 然后，PspUserThreadStartup 把一个用户模式 APC 插入到线程的用户 APC 队列中，此 APC 例程是在全局变量 PspSystemDll 中指定的，指向 ntdll.dll 的 LdrInitializeThunk 函数。

f. 接下来填充系统范围的一个 Cookie 值。

(7) PspUserThreadStartup 函数返回以后，KiThreadStartup 函数返回到用户模式，此时，PspUserThreadStartup 插入的 APC 被交付，于是 LdrInitializeThunk 函数被调用，这是映像加载器（image loader）的初始化函数。LdrInitializeThunk 函数完成加载器、堆管理器等初

始化工作，然后加载任何必要的 DLL，并且调用这些 DLL 的入口函数。最后，当 LdrInitializeThunk 返回到用户模式 APC 分发器时，该线程开始在用户模式下执行，调用应用程序指定的线程启动函数，此启动函数的地址已经在 APC 交付时被压到用户栈中。

(8) 至此，进程已完全建立起来，开始执行用户空间中的代码。

现在我们理解了 Windows 系统中一个用户进程的整个创建过程，虽然有一部分工作是由 Windows 子系统来完成的，但是从操作系统内核的角度，我们依然可以清楚地看到，Windows 为了支持进程和线程的概念，是如何以对象的方式来管理它们，并创建和初始化进程和线程，使它们变成真正可以工作的功能实体。

3.4.4 进程和线程的结束处理

在上一节中我们看到了进程和线程的创建过程，其中不仅涉及进程和线程中数据结构成员的初始化，而且，还需要为进程建立地址空间以及从父进程继承有关的信息，然后为线程建立起基本的执行环境。这一节将简要地讨论进程和线程的结束过程。相比之下，结束过程要容易和简便得多。由于结束一个进程首先要结束其中的每一个线程，所以，我们先讨论线程的结束处理。

在执行体层上，线程的终止函数是 NtTerminateThread，它调用 PspTerminateThreadByPointer 函数来完成真正的终止处理。系统线程的终止函数则是 PsTerminateSystemThread，它也调用 PspTerminateThreadByPointer 来完成真正的终止处理。以下是这三个函数的原型：

```
NTSTATUS
NtTerminateThread(
    __in_opt HANDLE ThreadHandle,
    __in NTSTATUS ExitStatus
    );

NTSTATUS
PsTerminateSystemThread(
    __in NTSTATUS ExitStatus
    );

NTSTATUS
PspTerminateThreadByPointer(
    IN PETHREAD Thread,
    IN NTSTATUS ExitStatus,
    IN BOOLEAN DirectTerminate
    );
```

NtTerminateThread 函数的逻辑非常简单，它首先把参数 ThreadHandle 所指的引用解出来，然后调用 PspTerminateThreadByPointer 来结束指定的线程。如果要终止的线程

是当前线程，则 PspTerminateThreadByPointer 的 DirectTerminate 参数被设置为 TRUE，表示此函数永不返回（因为线程已经结束了）；否则，此参数设置为 FALSE，并且 PspTerminateThreadByPointer 返回以后，线程的句柄数减 1。PsTerminateSystemThread 函数更加简单，直接调用 PspTerminateThreadByPointer 函数，并且指定不再返回。

在 PspTerminateThreadByPointer 函数中，如果当前线程被终止，则设置结束标志 PS_CROSS_THREAD_FLAGS_TERMINATED，并调用 PspExitThread 退出线程，此函数执行实际的终止过程，并且不返回。如果不是当前线程，则在该线程中插入一个内核模式 APC，为它指定 PsExitSpecialApc、PspExitApcRundown 和 PspExitNormalApc 例程，由它们完成真正的终止过程。关于 APC 对象的交付过程，即它指定的这些例程是如何被执行的，请参考本书 5.2.6 节。最终 PsExitSpecialApc 函数被执行，它也调用 PspExitThread 函数来完成终止过程，所以下面只讨论 PspExitThread 函数的处理过程。

PspExitThread 函数的代码位于 base\ntos\ps\psdelete.c 的 792~1 342 行。它首先获得进程对象，并判断线程当前的状态是否允许终止。然后关掉各种跨线程的引用，通知那些已注册的线程删除事件接收者。进程的活动线程计数减 1，如果这是最后一个线程，则必须等到该进程的线程链表中所有的线程都退出才能继续往下进行。接下来，如果有必要，则发送调试信息。

然后，处理 TerminationPort，向所有已经登记过要接收终止通知的线程发送终止消息。如果 TerminationPort 为空，但异常端口存在，则向异常端口发送终止消息。之后，通知 Windows 子系统当前线程退出。如果这是进程的最后一个线程，还要通知 Windows 子系统当前进程退出。

接下来，取消该线程的 I/O，取消该线程的定时器，以及清除任何尚未完成的注册表通知请求，并且释放当前线程所拥有的突变体内核对象（mutant）。然后释放 TEB，调用 LpcExitThread 以便让 LPC 组件来处理 LPC 应答消息（LpcReplyMessage 域中的消息）。接下来设置线程的退出时间（ExitTime）和退出状态（ExitStatus）。如果这是进程的最后一个线程，则还要调用 PspExitProcess 以退出进程，并设置进程退出的相关属性，以及销毁句柄表和解除对进程的内存区对象的引用。

然后强制恢复线程运行，以便处理可能有的 APC。遍历用户模式 APC，如果此 APC 有一个终止函数（RundownRoutine），则执行此函数，否则直接扔掉此 APC。接着，如果这是进程的最后一个线程，则清除当前进程的地址空间。

到这时候，可以收尾了，因为线程的所有代码已经执行完了（也不会再有内核 APC 了）。而且，如果这是进程的最后一个线程，则进程的地址空间也不存在了。在这种情况下，PspExitThread 将进程对象的状态变成有信号状态，以唤醒那些可能在等待此进程对象的线程。最后，PspExitThread 调用 KeTerminateThread 函数（在 base\ntos\ke\thredobj.c 文件的 2 217~2 412 行），设置线程对象的状态为有信号状态，并设置线程的调度状态为“已终止（Terminated）”，然后选择一个新的线程来运行。KeTerminateThread 函数永不返回，所以，PspExitThread 也不返回。处理器将执行新选择的线程，以后调度算法再也不会选择这一已终止的线程了。

在了解了线程的结束处理过程以后，我们实际上已经知道了进程的结束处理，因为最后一个线程的终止处理实际上就是进程的终止步骤中的最后一步。执行体层上的进程终止函数是 NtTerminateProcess（在 base\ntos\ps\psdelete.c 文件的 273~393 行），它的逻辑比较简单：首先，对进程中的所有线程使用一个 for 循环进行迭代，凡是非当前线程，调用 PspTerminateThreadByPointer 函数令其终止；然后，如果要终止的是当前进程，则调用 PspTerminateThreadByPointer 函数终止自己，此调用不再返回。

另一个函数 PsTerminateProcess 也是完成进程终止任务的，比如在系统停机的时候会被调用到。它调用了 PspTerminateProcess 函数（在 base\ntos\ps\psdelete.c 文件中的 404~467 行），其逻辑跟 NtTerminateProcess 很类似，只不过它更加简单，不考虑是否在终止自己，只是简单地用一个循环来调用 PspTerminateThreadByPointer 以删除指定进程中的所有线程。如果该进程没有任何线程，则只是清除进程的句柄表。

3.4.5 系统初始进程和线程

以上我们看到了进程的创建和结束过程，应用程序或者系统内核组件可以在执行过程中调用 API 函数或者执行体层的函数来创建进程或线程。这一节我们来讨论系统的初始进程和线程是怎样建立起来的，以及进程和线程管理子系统是如何初始化的。

2.6 节介绍了 Windows 的引导过程，当系统加载器 ntldr 把控制权交给 Windows 内核以后，内核执行一系列的初始化工作。内核的初始化分两个阶段来完成。进程和线程管理子系统的初始化函数 PsInitSystem 在这两个阶段中被分别调用，以完成其初始化工作。下面分别介绍与这两个阶段相关的初始化事项。

Windows 内核在接收到 ntldr 转交过来的 P0 处理器控制权时，内核环境尚未建立起来，所以，这时还根本没有进程和线程的概念，内核得到的只是一个基本的控制流。内核的入

口点是汇编函数_KiSystemStartup，它调用 C 函数 KiInitializeKernel 以执行内核层的初始化。在 KiInitializeKernel 函数中，调用 KeInitializeProcess 函数，以初始化当前进程对象，并设置相关的参数，这便是空闲进程（idle process），其进程 ID 为 0。然后 KiInitializeKernel 又调用 KeInitializeThread 函数以便初始化当前线程对象，而 KeInitializeThread 则调用 KeInitThread 来设置线程的执行环境和各种参数，再进一步调用 KeStartThread 将线程插入到当前进程环境中。在阶段 0 初始化完成以后，当前线程蜕变成 P0 处理器上的空闲线程。这样，空闲进程和 P0 处理器上的空闲线程被初始建立起来。注意，这里提到的进程和线程初始化函数调用均发生在执行体初始化以前，此时执行体本身尚未被初始化。

执行体在阶段 0 初始化过程中，其初始化函数 ExpInitializeExecutive 调用 PsInitSystem 函数，而 PsInitSystem 将控制转给 PspInitPhase0 函数。在 PspInitPhase0 中，除了完成全局初始化工作以外，进一步完成空闲进程的初始化，并且创建 System 进程和用于运行阶段 1 初始化的 Phase1Initialization 线程。

在阶段 1 初始化过程中，Phase1InitializationDiscard 函数也调用了 PsInitSystem 函数，而 PsInitSystem 将控制转给 PspInitPhase1 函数。在 PspInitPhase1 中，调用 PspInitializeSystemDll 函数以初始化系统 DLL（即 ntdll.dll），并找到系统 DLL 中的入口函数，以及其他一些负责异常分发和用户 APC 分发或回调至用户模式的函数的地址，参见 PspLookupKernelUserEntryPoints 函数的代码。执行阶段 1 初始化的线程最终蜕变成系统的零页面线程。

经过阶段 1 初始化以后，进程和线程管理子系统已经可以正常工作。值得一提的是，空闲进程和 System 进程在整个系统中只有一个，它们的进程对象分别是 PsIdleProcess 和 PsInitialSystemProcess，其 ID 分别为 0 和 4。零页面线程也只有一个，但空闲线程则每个处理器对应有一个。除了 P0 处理器的初始线程蜕变成空闲线程以外，其他处理器在阶段 1 初始化过程中被启动起来以后，也从_KiSystemStartup 函数开始执行，并且在完成了该处理器上的初始化任务以后转到空闲循环，蜕变成该处理器的空闲线程。

System 进程容纳了所有的系统线程，内核在调用 PsCreateSystemThread 函数时若不指定目标进程，则新建的线程运行在 System 进程空间中。有一些不依赖于任何其他进程环境的操作可以放在 System 进程中运行。DPC 例程或线程调度器可以将一些需要访问换页内存或等待分发器对象的代码逻辑转移到系统线程中。此外，System 进程中有一组称为系统辅助线程的线程池，专门运行各种工作项目（WorkItem），因此，Windows 内核或驱动程序可以不创建任何线程，而是将要做的工作包装成一个工作项目，然后交给 System 进程中的系统辅助线程来完成。

经过内核初始化以后，尽管系统的引导过程还有很多工作要做，包括会话创建和用户登录等，但是，内核的进程和线程管理已经起作用，后面的工作由线程调度器安排和运行。由于所有的处理器在初始时都从空闲循环进入线程调度器，因此，这里简单提一下空闲循环所做的事情，其代码在 base\ntos\ke\i386\ctxswap.asm 文件中（KiIdleLoop）：

- 它快速地开一下中断，然后又关上中断以便处理后续的 DPC 和线程检查。
- 检查当前处理器上是否有 DPC（延迟的过程调用，通常是中断处理例程登记的过程）正在等待执行，如果有 DPC 在等待，则清除软中断，并交付 DPC。
- 然后检查是否有一个线程已被选出来作为该处理器上运行的下一个线程，如果有，则分发该线程，使它运行。否则，判断该处理器是否已被置上空闲调度（idle schedule）标志，若是，则调用 KiIdleSchedule，以查找在该处理器上运行的下一个线程；否则回到空闲循环开始处。

3.5 Windows 中的线程调度

如同本章 3.1 节和 3.2 节所介绍的那样，一旦系统中存在多个线程，则这些线程将共享物理的处理器资源，它们实际上只得到一部分处理器时间。如果线程的个数大于处理器的数量，则自然地，哪些线程先执行，哪些线程后执行，这需要由系统的线程调度器来决定。Windows 的线程调度器建立在线程的优先级基础之上，它允许一个高优先级的线程将 CPU 资源从低优先级的线程上抢过来，而线程的优先级并不是绝对一成不变的，在系统执行过程中有些线程的优先级会动态地变化。另外，有些线程具有处理器亲和性，即只能在某些处理器上运行；而有些线程对特定的处理器更具有倾向性。所有这些都导致了 Windows 线程调度机制非常复杂，而且，有关线程调度的代码散布在内核层各处，系统的很多执行逻辑都需要涉及线程调度，这使得即使从源代码层次来看，线程调度机制也难以理解。

在 Windows 中，线程调度的职能是在内核层上完成的，并不涉及执行体层。这一节我们将分别从线程优先级、线程状态转移、处理器的时限管理，以及如何执行线程切换这些角度来讨论 Windows 内核中的线程调度。

3.5.1 线程优先级

我们首先来看 Windows 是如何定义和划分线程优先级的。Windows 内部使用 32 个优先级别，其值从 0 至 31。这 32 个值又被分成三种类别：

- 实时类别：16~31。
- 动态类别：1~15。
- 系统类别：0。

进程对象的 KPROCESS 结构的 BasePriority 域指定了一个进程的基本优先级，由于进程本身并不参与调度，所以这个值的意义在于该进程中的每个线程在初始化时都可以直接从该进程对象中获得基本优先级值。而线程对象的 KTHREAD 结构的 BasePriority 和 Priority 域则分别定义了一个线程的静态和动态优先级。进程和线程的基本优先级是保持不变的，除非通过调用 KeSetPriorityAndQuantumProcess 或 KeSetBasePriorityThread 函数显式地改变。

运行在动态优先级类别中的线程，它们的实际优先级可能会根据特定的情形而作调整，所以，KTHREAD 结构的 Priority 域记录了一个线程的当前实际优先级。它往往是在 BasePriority 域的基础上提升一定的优先级增量。但是，无论怎么调整，Priority 域值的范围不会越过 1~15。KiComputeNewPriority 是计算 Priority 值的函数，代码如下：

```
FORCEINLINE
SCHAR
KiComputeNewPriority (
   IN PKTHREAD Thread,
   IN SCHAR Adjustment              // 线程优先级调整量
   )
{

   SCHAR Priority;

   //
   // Compute the new thread priority.
   //

   ASSERT((Thread->PriorityDecrement >= 0)
                  && (Thread->PriorityDecrement <= Thread->Priority));

   ASSERT((Thread->Priority < LOW_REALTIME_PRIORITY) ? TRUE
                  : (Thread->PriorityDecrement == 0));

   Priority = Thread->Priority;
   if (Priority < LOW_REALTIME_PRIORITY) {
      Priority = Priority - Thread->PriorityDecrement - Adjustment;
      if (Priority < Thread->BasePriority) {
         Priority = Thread->BasePriority;
      }

      Thread->PriorityDecrement = 0;
   }

   ASSERT((Thread->BasePriority == 0) || (Priority != 0));
```

```
    return Priority;
}
```

从代码中可以看出，线程的优先级重新计算只针对非实时优先级类别的线程，并且新的优先级不会低于基本优先级。另外，在提升一个线程的优先级时，新的优先级值不会超越到实时线程的优先级范围（即 16~31）。而实时线程的优先级不会调整，它们的当前优先级总是等于其基本优先级。

在 Windows 平台上，用户可以通过像任务管理器（task manager）这样的工具来改变进程的优先级。在任务管理器的优先级菜单中，列出了 6 个选择项：实时（Realtime）、高（High）、普通之上（Above Normal）、普通（Normal）、普通之下（Below Normal）和低（Low）。实际上，这是执行体层提供的优先级分类，执行体允许按这 6 个类别来设置进程的优先级。这些选择项所对应的优先级分别是 24、13、10、8、6 和 4。执行体在创建进程时，调用 PspComputeQuantumAndPriority 函数（见 base\ntos\ps\psquery.c）来计算优先级，而 PspComputeQuantumAndPriority 函数根据进程 EPROCESS 对象中的 PriorityClass 域来查表以便获得内核层的优先级定义。此对照表定义在全局变量 PspPriorityTable 中。以下是一些相关的定义：

```
// 下面的宏定义位于文件 public\sdk\inc\ntpsapi.h 文件中
#define PROCESS_PRIORITY_CLASS_UNKNOWN              0
#define PROCESS_PRIORITY_CLASS_IDLE                 1
#define PROCESS_PRIORITY_CLASS_NORMAL               2
#define PROCESS_PRIORITY_CLASS_HIGH                 3
#define PROCESS_PRIORITY_CLASS_REALTIME             4
#define PROCESS_PRIORITY_CLASS_BELOW_NORMAL         5
#define PROCESS_PRIORITY_CLASS_ABOVE_NORMAL         6

// 下面的全局变量定义位于文件 base\ntos\ps\psquery.c 中
const KPRIORITY PspPriorityTable[PROCESS_PRIORITY_CLASS_ABOVE_NORMAL+1]
                        = {8,4,8,13,24,6,10};
```

所以，执行体层的优先级类别经过全局表 PspPriorityTable 变换后，就变成了内核层上的 0~31 优先级值。另外，在执行体层上，除了通过优先级类别的方式来设置一个进程的基本优先级以外，还可以通过 NtSetInformationProcess 函数对基本优先级进行微调，并非绝对是 PspPriorityTable 表中指定的优先级值。

从这里对进程优先级的控制我们也可以看到，执行体层的职能是提供管理和策略支持，并为上层应用程序提供服务，而内核层提供的是基本的线程调度机制。这正是策略与机制相分离的设计思想。

最后我们看一下在 Windows 中优先级提升的一些情形。首先，我们可以注意到，有不少函数带一个“KPRIORITY Increment”参数，它代表了优先级提升量。显然，优先级

提升应该发生在该线程被作出调度决定（即插入到对应优先级的调度链表中）以前。内核提供的带 Increment 参数的函数包括：KeInsertQueueApc、KePulseEvent、KeSetEvent、KeReleaseMutant 、 KeReleaseSemaphore 、 KeSetProcess 、 KeBoostProirityThread 和 KeTerminateThread。

下面列出优先级提升的典型情形：

- 当一个 I/O 操作完成时，正在等待此 I/O 的线程有更多的机会被立即执行。IoCompleteRequest 函数有一个参数是优先级提升值，至于具体的提升量则由负责 I/O 的驱动程序来决定。
- 一个线程在等待到事件和信号量以后，其优先级得到提升。参见 NtSetEvent、NtPulseEvent 、 NtReleaseSemaphore 、 NtReleaseMutant 、 NtReleaseKeyedEvent 、NtWaitForKeyedEvent、NtSignalAndWaitForSingleObject 和 NtSetEventBoostPriority 中的代码。
- 前台线程从等待状态中醒来时，有一点小小的优先级提升。参见 KiDeferredReady-Thread 函数中关于 PsPrioritySeparation 的用法。
- 在平衡集管理器系统线程中，扫描那些已进入就绪状态 4 s 但尚未被执行的线程，将它们的优先级提升至 15（参见 KiScanReadyQueues 的代码）。其用意在于避免优先级反转情形的长时间出现。
- 窗口线程因窗口的活动（比如窗口消息到来）而醒来时，会得到一个额外的优先级提升。这是窗口管理系统在调用 KeSetEvent 函数时设置的。

3.5.2 线程状态转移

KTHREAD 的 State 域是一个跟线程调度有关的成员，反映了线程的当前调度状态，其类型为 C 枚举类型 KTHREAD_STATE，下面是它的定义，在 base\ntos\inc\ke.h 文件中：

```
typedef enum _KTHREAD_STATE {
   Initialized,
   Ready,
   Running,
   Standby,
   Terminated,
   Waiting,
   Transition,
   DeferredReady,
   GateWait
} KTHREAD_STATE;
```

每个线程从初始化开始一直到最后终止，其状态既可能随着系统状态而变化，也可能随自身的代码逻辑而发生变化，并非总是一直执行下去，直到结束。为了更好地理解线程调度器的工作原理，我们有必要了解 KTHREAD_STATE 枚举类型中的状态的含义，以及在 WRK 中这些状态是如何转移的。下面列出这些状态的说明：

- 已初始化（Initialized）：说明一个线程对象的内部状态已经初始化，这是线程创建过程中的一个内部状态，此时线程尚未加入到进程的线程链表中，也没有启动。
- 就绪（Ready）：代表该线程已经准备就绪，等待被调度执行。当线程调度器选择一个线程来执行时，它只考虑处于就绪状态的线程。此时，线程已被加入到某个处理器的就绪线程链表中。
- 运行（Running）：线程正在运行。该线程一直占有处理器，直至分到的时限结束，或者被一个更高优先级的线程抢占，或者线程终止，或者主动放弃处理器执行权，或者进入等待状态。
- 备用（Standby）：处于备用状态的线程已经被选中作为某个处理器上下一个要运行的线程。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。然而，一个处于备用状态的线程在真正被执行以前，有可能被更高优先级的线程抢占。
- 已终止（Terminated）：表示线程已经完成任务，正在进行资源回收。KeTerminateThread 函数用于设置此状态。
- 等待（Waiting）：表示一个线程正在等待某个条件，比如等待一个分发器对象变成有信号状态，也可以等待多个对象。当等待的条件满足时，线程或者立即开始运行，或者回到就绪状态。
- 转移（Transition）：处于转移状态的线程已经准备好运行，但是它的内核栈不在内存中。一旦它的内核栈被换入内存，则该线程进入就绪状态。
- 延迟的就绪（DeferredReady）：处于延迟的就绪状态的线程也已经准备好可以运行了，但是，与就绪状态不同的是，它尚未确定在哪个处理器上运行。当有机会被调度时，或者直接转入备用状态，或者转到就绪状态。因此，此状态是为了多处理器而引入的，对于单处理器系统没有意义。
- 门等待（GateWait）：线程正在等待一个门对象。此状态与等待状态类似，只不过它是专门针对门对象而设计。关于门对象和门等待，请参考 5.4.3 节。

与传统的操作系统中的调度状态相比，这里可能多了后三个状态。这些状态之间的转移图并不复杂，图 3.5 显示了 WRK 中的线程状态转移图。

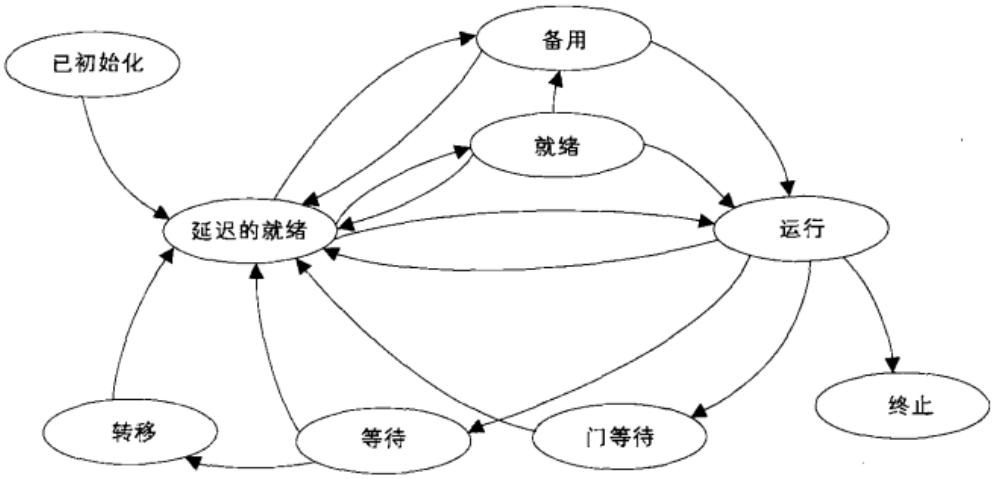


图 3.5 WRK 中的线程状态转移图

在一个线程的生命周期中，它从完成初始化开始进入到线程调度器的视野中，之后，一直到完成所有预定的功能，最后终止并被销毁。在此过程中，为了获得系统的处理器资源，它必须要经历图 3.5 中的状态转移。下面首先介绍一个对于线程的状态转移起关键作用的函数 KiReadyThread。当一个线程从其自身逻辑的角度而言，已经满足了可执行条件（比如，等待的对象变成了有信号状态）时，此函数就会被调用。其代码位于 base\ntos\ke\thredsup.c 中（894~981 行），如下所示：

```
VOID
FASTCALL
KiReadyThread (
    IN PKTHREAD Thread
    )
{

    PKPROCESS Process;

    // 如果所属的进程不在内存中（已被换出），则把此线程插入到进程对象的就绪链表中，
    // 并且通知交换线程（KiSwappingThread）执行换入操作。
    // 设置线程的状态为就绪状态。

    Process = Thread->ApcState.Process;
    if (Process->State != ProcessInMemory) {
        Thread->State = Ready;
        Thread->ProcessReadyQueue = TRUE;
        InsertTailList(&Process->ReadyListHead, &Thread->WaitListEntry);
        if (Process->State == ProcessOutOfMemory) {
            Process->State = ProcessInTransition;
            InterlockedPushEntrySingleList(&KiProcessInSwapListHead,
                                           &Process->SwapListEntry);
```

```
            KiSetInternalEvent(&KiSwapEvent, KiSwappingThread);
        }

        return;

    } else if (Thread->KernelStackResident == FALSE) {

        // 如果线程的内核栈未在内存中，则增加进程的栈计数，将线程的状态设置为转移状态。
        // 并且把线程插入到需要换入内核栈的线程链表中，通知交换线程执行换入操作。

        ASSERT(Process->StackCount != MAXULONG_PTR);

        Process->StackCount += 1;

        ASSERT(Thread->State != Transition);

        Thread->State = Transition;
        InterlockedPushEntrySingleList(&KiStackInSwapListHead,
                                       &Thread->SwapListEntry);

        KiSetInternalEvent(&KiSwapEvent, KiSwappingThread);
        return;

    } else {

        // 把指定的线程插入到延迟就绪链表中。在函数 KiInsertDeferredReadyList 中，
        // 线程的状态被设置为延迟的就绪状态。

        KiInsertDeferredReadyList(Thread);
        return;
    }
}
```

对于一个线程来说，KiReadyThread 函数被调用是一个转折点，这意味着它进入了备选的通途；而在此之前，此线程是不会被考虑执行的。在 WRK 中，KiReadyThread 函数在以下情况下被调用：

- 当一个线程被解除等待时，见 KiUnwaitThread 函数。这通常发生在一个等待对象变成有信号状态的时候。
- 当一个内核队列对象（KQUEUE）中有新的成员插入并且满足必要的条件时，也会导致队列中的等待线程被调用 KiReadyThread 函数。参见 KiInsertQueue 函数。
- 当一个线程被附载（attach）到一个新的进程中时，如果目标进程在内存中，则目标进程的就绪链表中的线程被调用 KiReadyThread 函数。这种情形发生在内存管理任务中。参见 KiAttachProcess 函数。
- 在 KeSetEventBoostPriority 函数中，如同第一种情形一样。在特定情形下，它直接调用 KiReadyThread 函数，而不是通过 KiUnwaitThread 函数来就绪一个线程。这是对

同步类型的事件对象的一种特殊处理。本质上而言，它与 KiUnwaitThread 的情形一致。

- 在处理换入进程链表时，KiInSwapProcesses 函数会针对每个已换入进程中的就绪线程调用 KiReadyThread 函数。
- 在处理换出进程链表时，KiOutSwapProcesses 函数会检查每一个待换出的进程，如果它的就绪线程链表非空，则不换出该进程，并针对此进程中的就绪线程调用 KiReadyThread 函数，让它们尽快得到执行。
- 在创建线程的函数（PspCreateThread）中，通过 KeReadyThread 函数来间接地调用 KiReadyThread 函数，这很容易理解，一旦一个线程创建成功，就需要激活该线程，让它得以执行。

现在我们来看一个延迟的就绪状态的线程如何得到执行。首先，我们看到，KiReadyThread 函数调用 KiInsertDeferredReadyList 函数把一个线程插入到当前处理器 PRCB 中的延迟就绪链表中。以后当调度器获得控制权（比如硬件中断，或者在内核的某些控制点上）时，延迟就绪链表中的线程被逐个处理，这发生在 KiProcessDeferredReadyList 函数中。KiProcessDeferredReadyList 函数的逻辑非常简单，只是遍历当前处理器的延迟就绪链表，对每个线程调用 KiDeferredReadyThread 函数，使它有机会变成就绪或备用状态。

KiDeferredReadyThread 函数位于 base\ntos\ke\thredsup.c 中（112~682 行），它首先根据各种情形调整待处理线程的优先级（193~395 行），然后选择处理器（428~592 行）。这一段选择处理器的代码正说明了 Windows 操作系统对于多处理器的调度支持。如果能找到一个空闲处理器满足线程亲和性要求，则线程的状态被置成备用状态，它被选中接下来在那个处理器上运行，函数返回。否则，选择该线程的理想处理器，试图抢占此处理器上正在运行的线程（606~644 行）。如果确实能抢占，即线程的优先级大于此处理器上的备用线程的优先级，则抢占此线程，让它回到延迟就绪状态，新线程成为该处理器上下一个执行的线程，然后函数返回（616~631 行）。如果该处理器上下一个要执行的线程尚未选出，并且线程的优先级大于该处理器上当前正在执行的线程的优先级，则抢占该处理器上的当前线程，新线程成为该处理器上下一个要执行的线程，然后返回（632~643 行）。最后，如果未能抢占选中的处理器上的线程，则把线程的状态置成就绪状态，并且将线程插入到所选中的处理器的就绪链表中（663~681 行）。

在 KiDeferredReadyThread 的代码中，我们看到了延迟就绪和备用状态之间的相互变化，这正体现了高优先级线程抢占低优先级线程的基本概念。同时我们也看到了延迟就绪状态转变成就绪状态的情形。因此，延迟就绪意味着一个线程尚未确定被安排到哪个处理

器上执行，而就绪状态意味着已经被分配到某个处理器上，这是两种就绪状态之间的根本区别。另外，我们也看到了，处于备用状态的线程也有可能被抢占掉，从而回到延迟就绪的状态。

接下来，我们关心的是，一个就绪状态或者备用状态的线程如何被选出来运行。我们首先了解一下每个处理器的全局控制块数据结构 KPRCB，以下是针对 Intel x86 的 KPRCB 定义，这里只列出几个数据成员，完整的定义请参考 base\ntos\inc\i386.h 文件。

```
typedef struct _KPRCB {
    ……

    struct _KTHREAD *CurrentThread;
    struct _KTHREAD *NextThread;
    struct _KTHREAD *IdleThread;
    ……
    volatile BOOLEAN IdleSchedule;

    ULONG ReadySummary;
    LIST_ENTRY DispatcherReadyListHead[MAXIMUM_PRIORITY];
    SINGLE_LIST_ENTRY DeferredReadyListHead;
    ……

} KPRCB, *PKPRCB, *RESTRICTED_POINTER PRKPRCB;
```

在 KPRCB 数据结构中，CurrentThread、NextThread 和 IdleThread 域分别指向该处理器上当前正在运行的线程（处于运行状态）、下一个要运行的线程（处于备用状态），以及该处理器的空闲线程（即该处理器的初始线程完成了初始化以后蜕变过来的）。DispatcherReadyListHead 域是一个链表数组，其中每一个成员链表都对应于特定优先级（等于数组下标）的就绪线程。而 DeferredReadyListHead 链表包含了所有在该处理器执行过程中进入到延迟就绪状态的线程，ReadySummary 记录了 DispatcherReadyListHead 中哪些优先级的链表非空。所以，我们现在更清楚了，KiInsertDeferredReadyList 函数实际上是把一个线程插入到当前处理器的 DeferredReadyListHead 链表中，而 KiDeferredReadyThread 函数则是给 DeferredReadyListHead 链表中的线程找到一个合适的处理器，并把它插入到该处理器所对应的 KPRCB 的 DispatcherReadyListHead 数组的相应优先级的链表中。

所以，现在的问题是，DispatcherReadyListHead 数组中的线程如何转移到 NextThread 或 CurrentThread 中。在 WRK 中，有两个辅助函数用于从 DispatcherReadyListHead 数组中找到恰当的线程：KiSelectReadyThread 和 KiFindReadyThread。KiSelectReadyThread 函数返回某一优先级之上的最优先的线程；KiFindReadyThread 函数对指定 KPRCB 的链表数组进行遍历，从高优先级开始，直至找到第一个可以在指定处理器上运行的线程。

可以想象，让一个线程从就绪状态变成运行状态，这是由系统的调度器来完成的。系统调度器并不是单个函数，它分散在各个调度点上，可能的调度点包括硬件中断以及当前线程在执行时碰到了某种条件等。在 WRK 中，我们可以看到以下一些调度点，在这些点上，一个线程有机会变成运行状态：

- 在处理器的空闲循环中，若处理器的空闲调度标志（即 KPRCB 的 IdleSchedule 域）被置为 TRUE，则调用 KiIdleSchedule 函数(base\ntos\ke\idsched.c 文件的 26~246 行)。如果处理器上的备用线程（NextThread 域）不为空，则让备用线程运行，否则，调用 KiSelectReadyThread 找到要运行的线程。如果在当前处理器的链表中找不到合适的线程，则再从其他的处理器上寻找适合在当前处理器上运行的线程（通过调用 KiFindReadyThread）。一旦找到满足条件的线程，则该线程进入运行状态。
- 另一重要的调度点是 KiSwapThread 函数（base\ntos\ke\thredsup.c 文件的 1 689~2 027 行），类似地，它先判断 NextThread 域，然后调用 KiSelectReadyThread 函数在当前处理器上搜索，如果没找到，则通过 KiFindReadyThread 从其他处理器的就绪链表中寻找。新找到的线程变成运行状态。一个例外是，如果新找到的线程既不是当前处理器的空闲线程，也不是当前线程，并且新线程当前正在被切换过程中（其 SwapBusy 域为 TRUE），则将新找到的线程置成当前处理器的备用线程，而把空闲线程置成当前线程，从而让空闲线程负责切换到新线程。这样做可以避免死锁。
- 当前线程的时限已用完，这发生在 KiQuantumEnd 函数中。该函数首先计算当前线程的新优先级，并调用 KiSelectReadyThread 找到大于等于新优先级的线程，如果找到，则新线程变成运行状态，而老线程回到就绪状态，排队等待下次被调度。
- 一个线程主动放弃执行权，这发生在 NtYieldExecution 函数中。该函数调用 KiSelectReadyThread 在当前处理器的就绪链表中找到一个新的线程，老的线程回到就绪状态，新的线程变成运行状态。
- 除了以上这些调度点，当线程的优先级、亲和性发生变化或时限调整时，也同样会导致线程状态发生变化。这包括 KiSetPriorityThread、KeRevertToUserAffinityThread、KeSetSystemAffinityThread 和 KiAdjustQuantumThread 函数。在 KiSetPriorityThread 函数中，如果线程原来的状态是就绪、备用或运行状态，那么在改变了线程优先级以后，若它不能再保持原来的状态，则由于它本身的运行条件仍然是满足的，所以，它又回到延迟就绪状态，重新等待分配到某个处理器上。这是一种相对简单的处理方法。

因此，一个线程在自身运行条件满足的情况下，总是处于延迟的就绪、就绪、备用或

运行这四种状态之一。上面在讨论过程中我们实际上已经看到了这些状态之间的相互转移，如图 3.6 所示。尽管这些讨论并没有涵盖所有可能的状态转移，但线程获得处理器执行权的基本途径已经一一列举出来。由于每个处理器上只有一个线程在运行，所以，当一个线程获得了处理器执行权时，也就意味着该处理器上原来运行的线程失去了执行权。

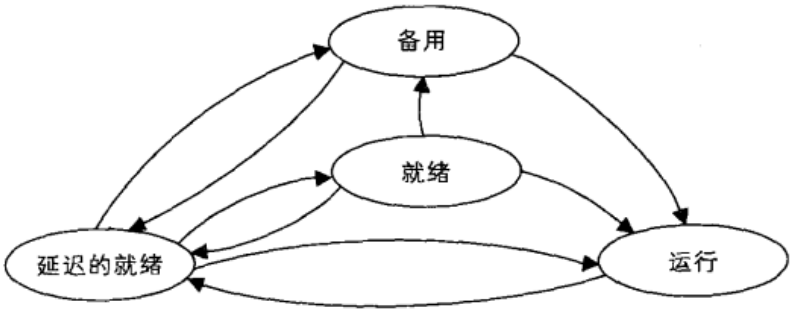


图 3.6 线程在满足执行条件情况下的状态转移图

接下来我们看一下线程等待的情形。一个线程在执行过程中，可能需要依赖某一特定条件，比如等待一个信号量变成有信号状态，或者等待一个系统事件。这些条件都会导致该线程无法继续执行，这时它会进入等待状态，直到所要求的条件满足以后才能继续执行。在 WRK 中，等待是通过 KeDelayExecutionThread、KeWaitForSingleObject 或 KeWaitFor-MultipleObjects 函数来实现的。当一个线程的控制流到达这三个函数之一时，除非等待条件可立即满足，否则，该线程将会进入等待状态，一直到等待的条件满足为止。关于等待的条件如何满足，以及等待机制的内部数据结构等信息，请参考 5.4 节。这里，我们只关心线程的状态转移，如图 3.7 所示。

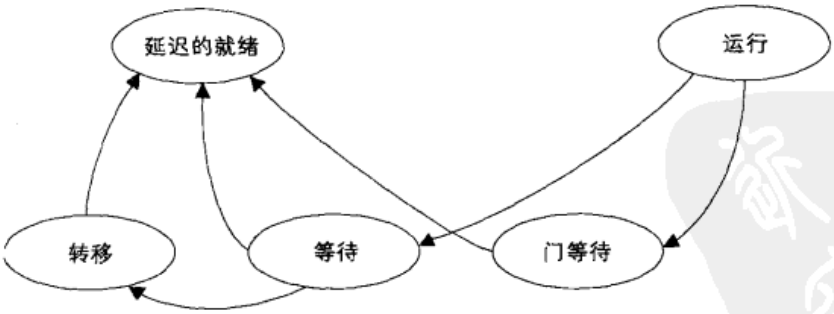


图 3.7 线程在等待时的状态转移图

当一个线程执行到 KeDelayExecutionThread、KeWaitForSingleObject 或 KeWaitFor-MultipleObjects 函数时，可能会进入等待状态，下面是这三个函数的基本流程：

- KeDelayExecutionThread 的功能是让当前线程延迟一段时间后再执行，参见 base\ntos\ke\wait.c 文件的 233~474 行代码。它利用 KTHREAD 结构中的 Timer 域来设置一个定时器，并调用 KiInsertOrSignalTimer 函数把定时器对象插入到系统的定时器表中。同时把线程的状态设置成等待，然后调用 KiSwapThread 交出控制权。在等待期间，也允许有异步过程（APC）被执行，但是，在定时器到期以前，将一直处于等待状态。关于定时器机制，参见本书 5.2.5 节。
- KeWaitForSingleObject 的功能是等待一个对象变成有信号状态，参见 base\ntos\ke\wait.c 文件的 999~1 296 行代码。它允许调用者指定超时值，所以，如果在超时以前，对象仍然处于无信号状态，则等待也将结束。它同样利用 KTHREAD 结构的 Timer 域来设置定时器，并把定时器对象加入到自己的等待链表中。除非此等待可以立即满足，否则，将设置线程的状态为等待，然后调用 KiSwapThread 交出控制权。在等待期间，允许 APC 对象被交付执行。
- KeWaitForMultipleObjects 函数与 KeWaitForSingleObject 类似，只不过它允许等待多个对象，在参数处理上要复杂一些，参见 base\ntos\ke\wait.c 中的 547~963 行代码。

一旦线程的状态被置成等待，并且控制权交出去以后，什么时候再次获得控制权呢？当内核在检测到等待的条件已经满足以后，会调用 KiReadyThread 函数，让线程进入延迟的就绪状态，从而最终有机会再次获得控制权，从以前停止的地方继续执行。有三个函数可以测试线程的等待条件是否满足：KiWaitTest、KiWaitTestSynchronizationObject 和 KiWaitTestWithoutSideEffects。5.4 节将会介绍线程与被等待对象之间的数据结构关系，以及等待条件的测试算法。一旦等待条件已经满足，这些等待测试函数就会调用 KiUnwaitThread 函数，而 KiUnwaitThread 又会进一步调用 KiReadyThread 函数。那么，谁会调用这些等待测试函数呢？当然是那些改变对象信号状态的函数，比如 KeSetEvent、KeReleaseSemaphore、KeReleaseMutant 等。

另外，在 KeRemoveQueue 函数中，如果一个队列对象中没有项可用或者队列对象中的线程数量达到了设定的最大值，则调用此函数的线程也会阻塞，直至有队列项可用或者有线程让出所占用的计数（调用 KiActivateWaiterQueue 函数），或者超时。其行为类似于 KeWaitForSingleObject。一旦出现这样的情形，则当前线程加入到队列对象的等待链表中，并进入等待状态，然后调用 KiSwapThread 函数交出处理器控制权。以后，当有新的队列项插入进来（通过 KiInsertQueue 函数），或者其他线程调用 KiActivateWaiterQueue 函数的时候，队列对象的等待链表中的线程就有机会被唤醒并继续执行（同样通过 KiReadyThread 函数）。队列对象是内核提供的线程池机制，关于它的工作原理，请参考本书 5.4.2 节的介绍。

在线程进入等待状态的每个点上，我们都会看到类似这样的代码：

```
CurrentPrcb = KeGetCurrentPrcb();
Thread->State = Waiting;
if (StackSwappable != FALSE) {
  InsertTailList(&CurrentPrcb->WaitListHead, &Thread->WaitListEntry);
}
```

其中，StackSwappable 是通过调用 KiIsKernelStackSwappable 函数得到的，此函数根据线程对象的 EnableStackSwap 域（并且此线程当前运行于用户模式下，而且优先级小于 25）来确定是否允许将内核栈换出。从这段代码可以看出，每个处理器的控制块 KPRCB 结构中都维护了一个等待线程链表，此链表中的线程都有机会在内存紧张的时候将其内核栈换出到外存中。待必要的时候再换回来。换出和换入的过程是由内存管理器中一个称为平衡集管理器的线程来完成的，参见 KeSwapProcessOrStack 函数。它调用 KiOutSwapKernelStacks 来处理内核栈的换出过程。而 KiOutSwapKernelStacks 则调用 MmOutPageKernelStack 函数把所有满足条件的线程的内核栈换出内存。注意，此时线程的状态仍然是等待状态，但是线程对象的 KernelStackResident 域为 FALSE，说明内核栈不在内存中。当此线程的等待条件被满足，即针对该线程调用 KiReadyThread 时，它的状态被置成转移（Transition），参见本节开始处列出的 KiReadyThread 函数的代码。当内核栈被换入时，线程的状态被置成延迟的就绪（通过调用 KiInsertDeferredReadyList 函数），参见 KiInSwapKernelStacks 中的代码。

因此，一个等待的线程，如果满足了内核栈换出条件，则可能会经历“等待-转移-延迟的就绪”的状态变化过程。内核栈和进程的换入/换出是由 Windows 内存管理器的平衡集管理器来完成的，参见本书 4.5 节、4.6 节中关于物理内存和工作集管理的详细描述。这里我们只需关心，处于等待状态的线程可能会经历这样一个过程。

在图 3.7 中，还有一个分支尚未介绍，即一个正在运行的线程进入到门等待的状态，然后又回到延迟就绪状态。当一个线程调用了 KeWaitForGate 函数时，这一状态变化过程就可能会发生。在 Windows 中，门对象是一种分发器对象，等待门对象的线程的内核栈不会被换出内存，而且一旦门对象变成有信号状态，等待门对象的线程就能获得优先级提升，所以，它们有更好的机会被立即执行。因此，相比普通的分发器同步对象，门对象是一种优化和简化，在 WRK 中，守护互斥体（guarded mutex）和推锁（push lock）使用门对象来实现互斥和共享访问。门对象仅限于在内核层中使用，它没有被暴露到内核以外。

当一个线程调用 KeSignalGateBoostPriority 函数来释放一个门对象时，会唤醒门对象的等待链表中的第一个等待线程，从而使该线程的状态从门等待状态转变为延迟就绪状态，参见 base\ntos\ke\gateobj.c 中的 KeSignalGateBoostPriority 函数的代码。

至此，我们已经全面了解了 Windows 中一个线程的状态变化情况，这些状态变化有些是线程本身的逻辑造成的，而有些则是系统出于更有效地管理可用资源的目的而强加的。当线程数量较多时，系统不得不在某些特定的点上让一个正在运行的线程放弃它的处理器执行权，从而允许其他已经就绪的线程有机会获得处理器资源。在现代计算机系统中，处理器资源是按照一定的粒度（也称为时间片）来管理的，下一节我们将讨论处理器资源的时间管理。

3.5.3 时限管理

如果一个线程在它的执行过程中，既没有进入等待状态，也没有被其他高优先级的线程抢占，那么，它将会获得一段完整的处理器资源，这称为时限（quantum）。当时限结束时，线程调度器将会介入，按照公平性原则，接下来的时限将会分配给同等优先级或者更高优先级的其他线程。在实际系统中，时限的管理要复杂得多，线程可能在时限用完之前就已经主动放弃了处理器执行权，这时候其他线程有机会提前获得处理器资源，这样，时限未必是按整个单位来计算。Windows 实现了一套公平、有效的时限管理机制，尽可能地让线程在各种情况下都能够公平地使用处理器资源。

首先，每个线程对象 KTHREAD 结构中都有两个域与时限分配有关：Quantum 记录了该线程的当前时限中还剩下多少时间单位；QuantumReset 记录了这个线程的时限重置值，即一个完整时限的时间单位。在 KTHREAD 中，这两个域与 SuspendApc 域重叠。根据 KTHREAD 的定义，它们实际上被插入在 SuspendApc 域中。这种重叠是安全的，因为 SuspendApc 域用于当一个线程被挂起的时候，而此时它已不再参与处理器时限分配。

线程的 QuantumReset 值继承自进程对象，在客户系统中，此值默认为 6，而在服务器系统中，此值默认为 36。如果一个线程在一次时钟中断的处理中获得了执行权，即变成了运行状态，并且分配到了一次新的完整时限，即它的 Quantum 等于 QuantumReset 值，那么，当下一次时钟中断到来时，它要从它的当前时限数量中减掉一部分，作为这一次时钟中断间隔的消耗。在 WRK 中，此减量为 3，这也代表了每次时钟中断的时限计量为 3 个单位。在 base\ntos\inc\ke.h 中，我们可以看到以下两个常量定义：

```
#define CLOCK_QUANTUM_DECREMENT 3
#define WAIT_QUANTUM_DECREMENT 1
```

这里 CLOCK_QUANTUM_DECREMENT 代表了一次时钟中断的时限减少量，WAIT_QUANTUM_DECREMENT 代表了一个线程从等待状态中醒过来时的时限减少量。在时钟中断的处理函数 KeUpdateSystemTime（位于 base\ntos\ke\i386\clockint.asm 文件）

中，它调用 KeUpdateRunTime 函数，将当前线程的 Quantum 减少 CLOCK_QUANTUM_DECREMENT，如果 Quantum 值小于等于 0，则触发线程调度（可参考 base\ntos\ke\amd64\runtime.c 文件中 KeUpdateRunTime 函数的 C 代码实现）。

当一个线程从等待状态恢复时，它的 Quantum 将减少 WAIT_QUANTUM_DECREMENT，这发生在该线程进入延迟就绪状态之后 KiDeferredReadyThread 函数被调用时。根据上一小节关于线程状态转移的分析，我们知道，一个线程从等待状态醒过来时，必定进入延迟的就绪状态，然后才有机会被选中到某个处理器上执行。只要一个线程在时钟中断到来之前进入等待状态，那么它的时限值就会减 1；这并不是一种精确的做法，但是，如果不减的话，线程将有可能利用这一空当偷取系统的处理器资源，而逃避系统对它的资源消耗进行计量。然而，由于计量的不精确性，每次等待只扣除了 1/3 时钟中断的时限计量。如果要精确地计量时限值的使用，则必须使用该线程实际消耗的处理器时间。在 WRK 中，我们看到的是，每次等待醒来时时限值减 1。

Windows 在分配处理器资源时，可以通过两种方式向一个进程或线程有所倾斜，一种方式是，临时提升一个线程的优先级，比如等待门对象的线程就有这样的机会（线程对象的 AdjustReason 域等于 AdjustBoost，见 KiDeferredReadyThread 函数中的处理）；另一种方式是，给一个线程分配更大的时限值，这样它就能按比例地比相同优先级的其他线程获得更多的处理器资源。在 Windows 中，服务器进程或者前台进程的线程就有这样的优势。

在 WRK 中，我们可以看到在 base\ntos\ps\psinit.c 中定义了一个全局数组变量 PspForegroundQuantum。在系统初始化阶段，PspInitPhase0 调用 PsChangeQuantumTable 函数设置此 3 元素数组指向内部全局表 PspFixedQuantums 或 PspVariableQuantums 的前三个元素或后三个元素。具体的取值取决于 Windows 系统的配置（由系统注册表值 HKLM\SYSTEM\CurrentControlSet\Control\Priority\Win32PrioritySeparation 决定）。当系统创建进程时，PspCreateProcess 函数调用 PspComputeQuantumAndPriority 来设置基本优先级和时限重置值（相当于时限长度）。PspComputeQuantumAndPriority 根据全局变量 PsPrioritySeparation 来决定时限值应该取 PspForegroundQuantum 数组的哪一项。参见 base\ntos\ps\psquery.c 中的 PspComputeQuantumAndPriority 函数代码。因此，Windows 提供的灵活的时限分配机制，允许前台线程按比例地获得更多的处理器资源。

注册表值 Win32PrioritySeparation 对应于 PsChangeQuantumTable 函数的 PrioritySeparation 参数。根据 PsChangeQuantumTable 函数的代码（位于 base\ntos\ps\psinit.c 文件的 936~1 065 行）不难看出，Win32PrioritySeparation 的第 0~1 位对应于 PsPrioritySeparation 的值，它决定了前台线程的时限重置值取自于 PspForegroundQuantum 数组的哪一项；Win32PrioritySeparation 的第 2~3 位决定了 PspForegroundQuantum 数组

应该从 PspFixedQuantums 数组还是 PspVariableQuantums 数组中复制；而 Win32PrioritySeparation 的第 4~5 位则决定了应该复制前三项还是后三项。无论是 PspFixedQuantums 数组，还是 PspVariableQuantums 数组，后三项正好是前三项的 2 倍，所以，我们可以认为第 4~5 位代表了长时限值还是短时限值。在 Windows 系统中，Win32PrioritySeparation 的默认值为 2。

对于实际的 Windows 系统，Windows Server 2003 是一个服务器系统，它的 PspForegroundQuantum 数组为{36, 36, 36}，PsPrioritySeparation 值为 2，所以，它的前后台线程的时限重置值都是 36。而对于 Windows XP/Vista 以及 Windows 7 系统，它的 PspForegroundQuantum 数组为{6, 12, 18}，PsPrioritySeparation 值为 2，所以，前台线程的时限重置值为 18，后台线程的时限重置值为 6，前台线程有明显的时限长度优势。

时限值与实际时间长度之间的关系是，每 CLOCK_QUANTUM_DECREMENT 个单位对应于一个时钟中断间隔。在实际系统中，时钟中断的间隔差不多是 10~20 ms（由 HAL 设置），例如，在多核的 Windows XP/Vista 系统中，时钟中断间隔大约为 15 ms，而在单核系统中，此值为 10 ms 左右。因此，我们可以计算出来，在多核的 Windows XP/Vista 系统中，默认情况下，后台线程的时限长度为 30 ms，前台线程的时限长度为 90 ms。类似地，在多核的服务器系统中，线程的时限长度总是 180 ms。

3.5.4 优先级调度和环境切换

有了前面介绍的关于线程状态转移和时限管理的基础，现在我们可以把 Windows 的优先级调度算法串起来了。首先，系统维护了一个全局的处理器数组 KiProcessorBlock，其中每个元素对应于一个处理器的 KPRCB 对象。其次，另有一个全局变量 KiIdleSummary 记录了哪些处理器当前是空闲的。所谓一个处理器是空闲的，是指该处理器正在执行空闲循环（见 3.4.5 节的介绍）。KiIdleSummary 的每一位对应于一个处理器，在 32 位系统中，由于 KiIdleSummary 是 32 位的（DWORD），所以只支持至多 32 个处理器。

当一个处理器没有找到合适的线程来运行时，它会调用 KiSetIdleSummary 函数来设置 KiIdleSummary 中相应的位，参见 KiSelectNextThread 和 KiSwapThread 函数中的代码。相反，如果一个空闲处理器被分配了一个线程来运行，则需要通过调用 KiClearIdleSummary 函数来清除 KiIdleSummary 中相应的位。当 KiDeferredReadyThread 函数把一个延迟就绪的线程分配到一个空闲处理器上时，就会执行这一动作，参见 KiDeferredReadyThread 函数的代码。同样地，如果在 KiSwapThread 函数中找到了接下来需要运行的线程，也会调用 KiClearIdleSummary 函数。

在每个处理器的 KPRCB 结构中，我们已经看到过，都有一个单链表 DeferredReadyListHead 记录了那些在该处理器上处于延迟就绪状态的线程，以及一个双链表数组 DispatcherReadyListHead 记录了在每个优先级上已经被分配到此处理器上的就绪状态的线程。另有一个 32 位摘要位码 ReadySummary 反映了哪些优先级的就绪链表是非空的，所以，KiSelectReadyThread 和 KiFindReadyThread 函数利用链表数组和摘要位码信息，能很快地找到满足条件的线程。这实际上也正是 Windows 的优先级调度算法的核心所在，不同优先级的线程加入到不同的链表中，调度器在选择待执行的线程时，高优先级的线程链表首先被扫描，所以其中的线程先被执行。图 3.8 显示了这些数据结构之间的关系。

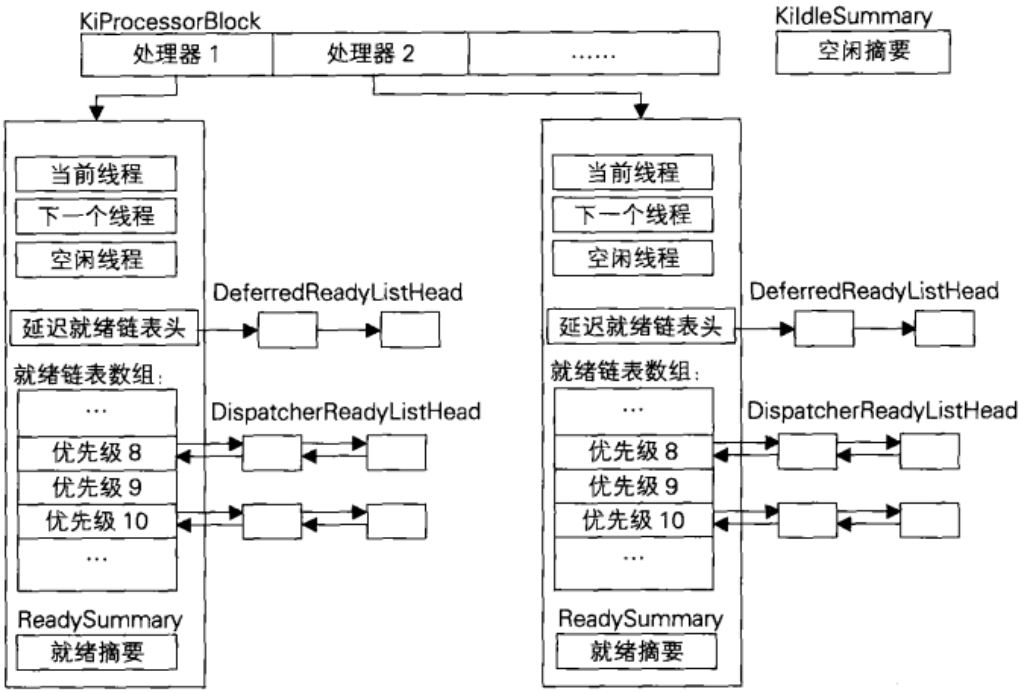


图 3.8 用于线程调度的数据结构和全局信息

接下来我们关心的问题是，一个正在运行的线程在哪些情况下要被切换，而切换又是如何完成的。大体而言，我们可以从两个角度来看一个线程的执行情况。首先，如果一个线程在执行过程中，主动放弃对于当前处理器的执行权，那么，此时当前处理器的控制流就会转移到系统的线程调度器（也称为 dispatcher），由线程调度器作出调度决定。譬如，本章前面分析线程的状态转移时，我们提到过，当一个线程进入等待状态时，将调用 KeDelayExecutionThread、KeWaitForSingleObject 或 KeWaitForMultipleObjects 函数，若等待的条件尚未满足，则线程会放弃当前处理器。其次，一个线程在执行过程中，由于系统的某些原因，将不得不放弃对于当前处理器的执行权。典型的情况是，当一个线程用完了

它的当前时限时，只要还存在优先级大于等于它的其他线程，那么它就必须交出处理器的执行权。我们可以把这两种情形分别称为自愿放弃执行权和被迫放弃执行权。

我们先讨论自愿放弃执行权的情形。当一个线程主动放弃当前处理器的执行权时，会在内核函数中调用 KiSwapThread 函数，该函数的两个参数分别指向当前调用此函数的线程对象和当前处理器的 KPRCB 对象。我们可以这样来理解，此时该线程把控制权交给了线程调度器。因此，KiSwapThread 函数是线程调度器获得控制权的一条重要途径。

KiSwapThread 函数的代码位于 base\ntos\ke\thredsup.c 中（1 689~2 027 行），关于它的代码逻辑，我们在前面 3.5.2 节介绍线程状态转移的时候曾经解释过，这里不再赘述。这里需要提及的是，在 thredsup.c 文件的 1 999 或 2 004 行，它调用 KiSwapContext 函数来完成线程切换。KiSwapContext 函数是一个小的包装函数（位于 base\ntos\ke\i386\ctxswap.asm 文件的 108~135 行），它调用 SwapContext 汇编函数完成实际的线程切换工作。本节最后我们再讨论 SwapContext 函数的代码逻辑。

KiSwapContext 函数的参数很简单，分别是老线程和新线程对象的指针。所以，它的任务仅仅是简单的线程切换。KiSwapThread 函数在调用 KiSwapContext 以前，已经按照本节前面讲述的数据结构选择好了新的线程，并把新线程的状态置成了运行状态。

进入等待状态并不是调用 KiSwapThread 函数的唯一情形，以下列出了所有通过 KiSwapThread 函数进行线程调度的可能性：

- 线程自愿进入等待状态，即 KeDelayExecutionThread、KeWaitForSingleObject 和 KeWaitForMultipleObjects 函数。
- 线程进入门等待状态，这发生在 KeWaitForGate 函数中。
- 一个线程终止了，它完成了终止处理事项以后，调用 KiSwapThread 函数把控制权交给线程调度器。这发生在 KeTerminateThread 函数中，此函数在调用了 KiSwapThread 函数以后，就再也不会得到控制权了。所以，KeTerminateThread 函数永不返回。
- 在队列对象中，如果一个线程调用 KeRemoveQueue，但又不能马上获得队列中的项，该线程也会进入等待状态，并且调用 KiSwapThread 函数把控制权交给线程调度器。此情形类似于 KeWaitForSingleObject。
- 当一个线程附载到（attach）一个进程的地址空间中时，若目标进程尚未被换入内存，则尽管当前线程是就绪的，但是它不能继续执行，所以，它调用 KiSwapThread 函数，把控制权交给线程调度器。自己等待将来进程换入内存以后再继续执行。这种情形发生在 KiAttachProcess 函数中。

接下来我们讨论一个线程被迫放弃执行权的情形。这实际上包括两种可能性：时限用完或者线程被抢占。线程被抢占发生在 KiDeferredReadyThread 函数中，本章 3.5.2 节在介绍此函数时曾经提到，一个线程若抢占另一个正在运行的线程，则它自己将成为目标处理器的下一个线程，即变成备用状态（参见 base\ntos\ke\thredsup.c 中 621~628、634~642 行代码），所以，当目标处理器下一个时钟中断到来时，它就有机会得到执行。时限用完的情形直接发生在时钟中断中。因此，我们现在从时钟中断来解析一个线程被迫放弃执行权的情形。

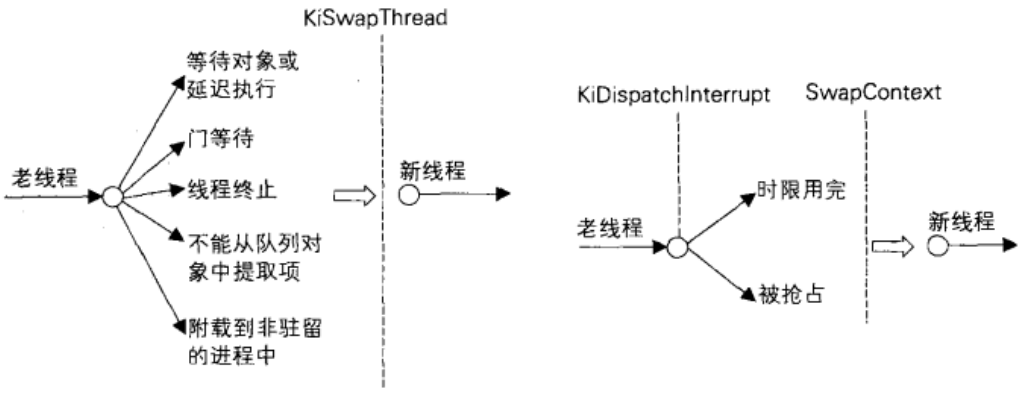
当时钟中断到来时，线程调度器获得控制权并不是直接在时钟中断处理函数中，而是在 KiDispatchInterrupt 函数中。实际上，KiDispatchInterrupt 是 DISPATCH_LEVEL 软件中断的处理函数，它也负责交付系统中的 DPC 对象。关于 Windows 的中断处理和 DPC 机制，请参考本书 5.2 节。

KiDispatchInterrupt 函数的代码位于 base\ntos\ke\i386\ctxswap.asm 文件的 168~317 行。其基本逻辑如下：

- 检查当前处理器上的 DPC 和定时器链表，只要这两个链表不为空，就调用 KiRetireDpcList 函数，交付这些 DPC 或定时器。
- 判断当前处理器的 KPRCB 对象中的 QuantumEnd 标志是否已置上。在系统时钟中断服务例程调用到 KeUpdateRunTime 函数时，若当前线程的时限已用完，则此标志被置上（base\ntos\ke\i386\clockint.asm 的 472~484 行）。如果此标志非零，则清除此标志，并调用 KiQuantumEnd 函数切换线程，然后返回，见 ctxswap.asm 的 313~315 行代码。
- 否则，判断当前处理器的 KPRCB 对象中的 NextThread 是否已被选出，即是否有处于备用状态的线程，如果没有，则函数返回。如果 NextThread 已被选出，则切换线程，见 ctxswap.asm 的 262~303 行代码。

因此，从这里可以看出，线程抢占和时限用完都发生在 KiDispatchInterrupt 函数中。在线程抢占的情形下，KiDispatchInterrupt 直接调用 SwapContext 函数执行线程切换；而在时限用完的情形下，它将调用 KiQuantumEnd 函数。KiQuantumEnd 函数的代码位于 base\ntos\ke\dpcsup.c 文件的 220~346 行，它检查当前线程的时限，并设置新一轮的时限，调用 KiComputeNewPriority 函数重新计算优先级，如果当前处理器尚未有备用线程，则调用 KiSelectReadyThread 函数选出新的线程进入备用状态。接下来，调用 KiSwapContext 函数实施线程切换（它调用 SwapContext 函数完成实际的线程切换工作）。注意，如果没有找到备用线程（NextThread），则 KiQuantumEnd 不实施线程切换，从而当前线程直接进入新一轮时限。

简单总结一下，线程切换的各种情况如图 3.9 所示，其中 3.9(a)代表了一个线程主动放弃执行权的情形，而 3.9(b)表达了一个线程被抢占或时限用完的情形。



(a) 线程自愿放弃执行权　　(b) 被迫放弃执行权

图 3.9　线程切换的各种情形

另有一个函数，KiExitDispatcher，值得在这里专门提出来。它也可能会导致线程切换，上面介绍的线程调度情形并没有涵盖 KiExitDispatcher 的用途。当一个线程在执行过程中进入到调度器的代码逻辑中，它可能会调用 KiExitDispatcher 函数温和地退出调度器的代码逻辑，回到线程自身的代码逻辑中。KiExitDispatcher 函数的功能比较简单，它恢复当前处理器在进入调度器逻辑以前的 IRQL。如何定义一个线程进入了调度器的代码逻辑呢？可以认为从它提升 IRQL 到 DISPATCH_LEVEL 开始。所以，KiExitDispatcher 恢复原来的 IRQL，表示退出了调度器。

KiExitDispatcher 的代码位于 base\ntos\ke\waitsup.c 文件的 23~112 行，它最核心的工作是恢复原来的 IRQL，所以，它唯一的参数是老的 IRQL。除此以外，它还检查当前处理器的延迟就绪链表是否非空，若是，则调用 KiProcessDeferredReadyList 函数，为这些尚未确定处理器的延迟就绪线程找到适当的处理器，从而变成就绪或备用状态。然后，若原来的 IRQL 低于 DISPATCH_LEVEL，并且当前处理器的下一个线程非空，则切换至此线程，让它运行。这使得当一个线程抢占其他线程的时候，并不一定要等到下次时钟中断到来时才把处理器抢过来，而是调度器一旦获得控制权就有机会抢占处理器。

最后，我们看一下线程切换是如何进行的。在上面的描述中，我们看到了 KiSwapThread、KiSwapContext 和 SwapContext 是在线程需要切换时被调用的函数。实际上，KiSwapThread 调用了 KiSwapContext，而 KiSwapContext 函数又包装了 SwapContext 函数，真正执行线程切换的是汇编函数 SwapContext，位于 base\ntos\ke\i386\ctxswap.asm

文件的 374~796 行。不难理解，SwapContext 一定是处理器相关的，它需要把处理器中所有与线程相关的状态（寄存器）都从老的线程环境变换到新的线程环境中，通常而言，指令指针、栈指针（包括用户栈和内核栈），以及进程地址空间（即页表目录指针）是必不可少的状态信息。此外，Windows 内核还需要维护一些全局的状态信息（位于当前处理器 KPCR 和 KPRCB 对象中），包括一些性能计数器等。SwapContext 函数的流程大致如下：

- 等待将要被切换过去的新线程的切换标志（KTHREAD 的 SwapBusy 域）被清除。
- 当前处理器上的线程切换计数器增 1。
- 原来线程的异常链表头保存到栈中。
- 根据需要保存协处理器的状态。
- 保存原来线程的栈指针，保存到 KTHREAD 的 KernelStack 域中。
- 检查新线程的协处理器状态，判断 CR0 寄存器与原来线程的 CR0 寄存器是否匹配，如果不是，则重新加载新线程的 CR0 寄存器。
- 设置新线程的栈指针，即新线程的 KTHREAD 对象的 KernelStack 域。
- 判断新线程是否与原来的线程同属于一个进程。如果不是，则先维护一下进程对象的 ActiveProcessors 域，然后判断 LDT 是否匹配，如果不匹配，则加载 LDT，同时更新 IDT 的 INT 21 项。接下来，让 CR3 寄存器指向新的目录表，这样把地址空间切换到新进程中。
- 清除原来线程的切换标志（KTHREAD 的 SwapBusy 域）。
- 设置当前处理器 KPRCB 中的 TEB，指向新线程的 TEB，并且设置 GDT（全局描述符表）中的 TEB 项也指向新线程的 TEB。
- 调整初始的内核栈地址，并设置到 TSS 的 Esp0 域。
- 设置新进程的 IOPM 表的位置。
- 新线程的环境切换次数增 1（即 KTHREAD 的 ContextSwitches 域）。
- 恢复线程的异常链表头。
- 判断当前是否在一个 DPC 中进行线程切换，如果是，则调用 BugCheck，蓝屏，在 Windows 中这是不允许的。
- 判断在新线程中是否有内核模式的 APC 正在等待处理，如果有，则调用 HAL 函数 HalRequestSoftwareInterrupt 以请求一个 APC_LEVEL 的软件中断，然后返回。如果没有 APC 在等待处理，则直接返回。

SwapContext 函数相对简捷高效，它涉及处理器的一些硬件特性，要想完全理解 SwapContext 所做的各种细节工作，请参考 Intel x86 处理器的硬件手册。

3.6 进程和线程运行状态监视工具

通过本章前面的介绍，我们已经理解了 Windows 中进程和线程的管理机制、进程和线程的状态信息，以及线程是如何被调度的。这一节将介绍一个实用小工具 ProcMon，它可以实时地监视当前系统中哪些进程和线程得到了处理器的执行权。我们首先看一下 ProcMon 工具的使用情况，然后介绍它的实现原理。

3.6.1 ProcMon 使用示例

ProcMon 是一个可直接运行的应用程序，其主程序为 ProcMon.exe，在启动时需要加载一个驱动程序 KMonDrv.sys。图 3.10 显示了在 Windows Server 2003 系统中正在监视模式下运行的 ProcMon。

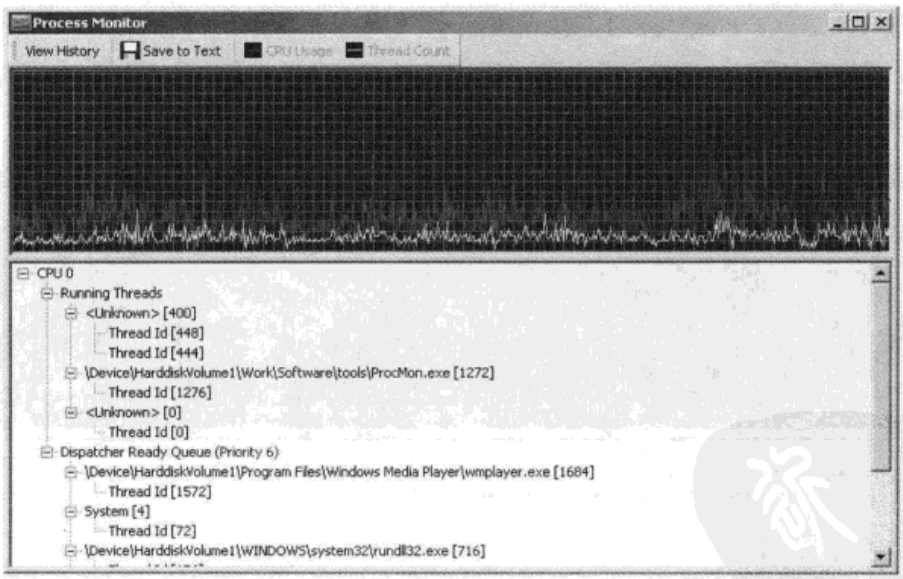


图 3.10 运行中的 ProcMon 工具

ProcMon 界面的上半部分显示了两条曲线，分别代表了在特定时间点上的 CPU 使用率（红色线）和可运行线程的数量（黄色线）。这里的可运行线程是指在一段时间间隔内被真正运行或者还在就绪链表或延迟就绪链表中排队的线程，我们可以认为这是指满足运行条件的线程。基本上，可运行的线程越多，则 CPU 使用率相对越高。

ProcMon 是一个监视程序，它通过驱动程序 KMonDrv.sys 在内核中收集信息，它每隔一定间隔（在程序中设定为 1 s）获取这些信息，并将收集到的线程信息解析成可显示的字符串，刷新到 ProcMon 界面下方的树状视图中。

在 ProcMon 的树状视图中，每个处理器对应一个根目录节点，在该节点的下面是两个子节点："Running Threads"和"Dispatcher Ready Queue"。前者是指在最近一段时间间隔内被监视到确实成为该处理器上运行线程的那些线程，即状态为"运行"的线程；后者是指在这一间隔中被插入到该处理器的就绪链表或延迟就绪链表中，但尚未被运行的线程，即状态为"就绪"、"延迟就绪"或"备用"的线程。在树状视图中，这些线程被聚集到进程下面，进程用它的映像名称来表示，进程 ID 和线程 ID 也跟随显示在每个名称的后面。因此，从图中可以清楚地看出，在一段时间间隔内哪些进程的哪些线程被运行了，而哪些进程的哪些线程被排队了。由于 ProcMon 进程定时地向内核取数据并刷新到树状视图中，所以，几乎每个间隔它都会出现在"Running Threads"子树中。

除了在运行过程中动态地刷新每一个间隔采样点的可运行线程集合以外，ProcMon 也允许用户选择特定的时间间隔点，并在树状视图中检查该时间间隔中的可运行线程集合。图 3.11 显示了 ProcMon 的"View History"模式（见左上角工具栏的第一个按钮）。界面上方格子坐标的中间有一条白线，代表用户选择了此时间点的采样数据；下方的树状视图列出了对应时间点处的可运行线程集合，显示格式与监视模式相同。

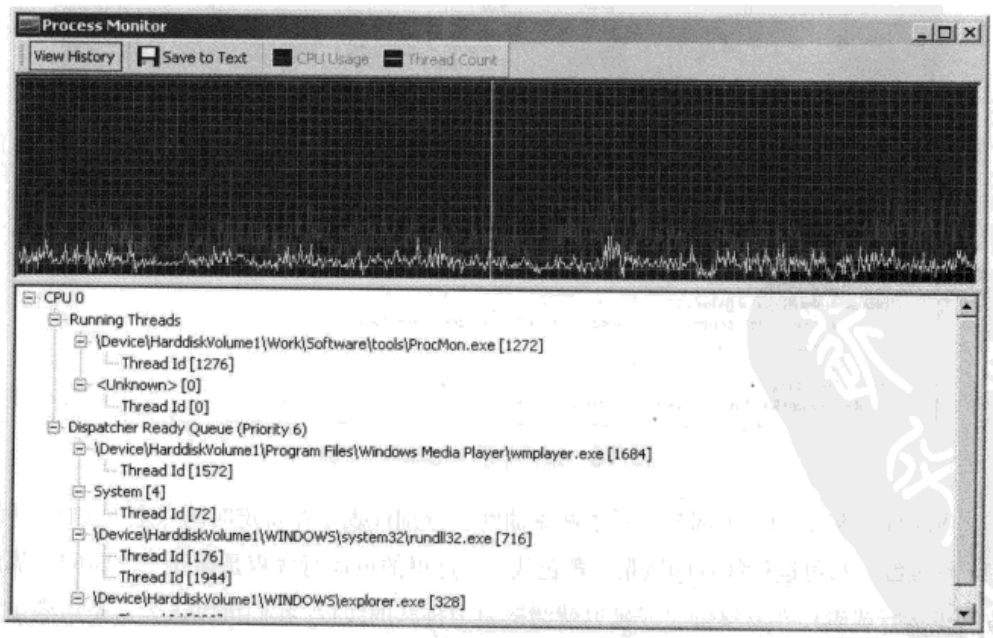


图 3.11 用户可以选择特定的间隔点来查看可运行线程

除了在界面中查看可运行线程集合以外，ProcMon 也可以将收集到的进程和线程数据导出到一个文本文件中，从而允许用户以文本方式查看整个监视时间范围内的线程运行情况。

目前 ProcMon 工具支持 Windows XP/Server 2003/Vista/Server2008/7，可以在多处理器或多核的系统上运行，它能够显示每一个处理器的可运行线程。

3.6.2 ProcMon 实现原理

ProcMon 工具的功能虽然很简单，但为了实现它的监视能力，需要在底层谨慎地操纵内核中有关线程调度的数据结构。用户模式的 ProcMon.exe 是一个典型的 MFC 程序，它负责加载驱动程序 KMonDrv.sys，然后通知驱动程序启动线程监视功能。ProcMon 会注册一个间隔为 1 s 的定时器。在处理定时器消息 WM_TIMER 的函数中，ProcMon 向驱动程序请求最近一个时间间隔内收集到的数据。然后，解析这些数据，根据进程 ID 找到进程映像文件的名称，再按照图 3.10 所示的树状结构组织起来，并刷新到树状视图中。

Windows 内核并没有提供运行线程变化通知，所以，KMonDrv 驱动程序无法以最有效的方式获得可运行线程的集合。目前 KMonDrv 驱动程序的做法是，定期地检查每个处理器当前正在运行哪个线程，它的下一个线程、就绪链表和延迟就绪链表中都有哪些线程。KMonDrv 的核心逻辑包括三部分：初始化、定时收集数据，以及响应 ProcMon 的数据请求，分别如下所述：

在初始化阶段：获得每个处理器的 KPRCB 数据结构，并且创建一个查询定时器。为了确保能获得尽可能多的运行线程信息，此定时器的间隔必须足够小，考虑到对性能的影响，此间隔值也不宜过小。对于服务器系统，比如 Windows Server 2003/Server 2008，实际的间隔取值为 50 ms；而对于客户系统，此值为 30 ms。这样的取值可以确保一个用完时限值的线程（即触发了 KiQuantumEnd 的线程）总是被 KMonDrv 监视到。在初始化阶段还会分配一个合适大小的缓冲区以便将来存放收集到的线程数据，注意，此缓冲区必须从系统的非换页内存池中分配。

在定时器的处理例程中收集数据：此例程运行在 DISPATCH_LEVEL 上，由于需要访问每个处理器的 KPRCB 对象中的数据，所以，它必须获得 KPRCB 的自旋保护锁（KPRCB 结构的 PrcbLock 成员），然后分别访问 CurrentThread、NextThread、DispatcherReadyListHead 和 DeferredReadyListHead 成员，并进而获得每个线程的基本信息。每个线程依据其 CLIENT_ID 值在缓冲区中占据一项。凡是已经运行过的线程，便不再记录它们还出现在就绪链表或延迟就绪链表中的情况。因此，在 ProcMon 的树状视图中，“Running Threads”子树下的线程一定不会再出现在“Dispatcher Ready Queue”子树中，尽管很多线程都是经

过排队才有机会被运行的。如果缓冲区已满，则新出现线程的信息无法被加入到缓冲区中。

响应 ProcMon 的数据请求：ProcMon 程序与 KMonDrv 驱动程序通过 Windows 的标准 I/O 接口进行通信。KMonDrv 响应一个自定义编码的设备控制 I/O 请求，从而将缓冲区中的线程数据传送给 ProcMon。一旦线程数据已被取走，则 KMonDrv 清空缓冲区，以便继续记录下一个定时器周期内的线程运行情况。注意，由于定时器例程和这里的 I/O 处理例程都要访问此缓冲区，因此，对缓冲区进行同步保护是必需的。

ProcMon 的数据收集机制可以不依赖于内核模块的符号信息，但是，它需要根据内核的版本信息来确定 KPRCB 和 KTHREAD 数据结构中一些成员的位置。目前 ProcMon 支持 Windows XP/Server 2003/Vista/Server 2008/7 系统，它可以检测出这些系统的内核的版本，包括它们的重要 Service Pack 的版本。因此，对于目前的主流 Windows 系统，以及未经修改的 WRK 内核，ProcMon 都可以正常工作。

然而，ProcMon 仍然不是一个完善的工具。一个严重的问题是，它的定时查询（polling）机制无法保证总是能够获得一个定时器周期内所有被执行或排队过的线程。譬如，一个高优先级线程抢占了当前正在运行的低优先级线程，并且在很短的时间内放弃了执行权，这样，KMonDrv 的定时器例程将没有机会看到这一高优先级线程曾经被执行过。这个问题可能有两种解决方案。第一，利用内核日志记录器（一个 ETW 提供者）提供的日志数据，也可以分析出一个系统中的线程运行情况。内核日志记录器的 CSwitch 事件记录了每个处理器上发生的线程切换信息，ReadyThread 事件记录了一个线程进入延迟就绪状态的信息。利用这两个事件，我们几乎可以构造出类似于 ProcMon 中的视图。关于内核日志记录器的事件的用法，请参考本书 9.3 节的介绍。第二，利用 WRK 的内核代码，或者采用内核钩子的做法，将线程运行信息毫无遗漏地记录到一个缓冲区中，供上层应用程序或者驱动程序提取并显示。有兴趣的读者不妨将第二种做法当做一个 WRK 实验来练习。

3.7 本章总结

本章讨论了进程和线程的原理，以及 Windows 中进程和线程的对象表示形式，还介绍了 Windows 中进程和线程是如何被创建和终止的，以及系统的初始进程和初始线程是如何建立起来的。本章后面部分重点讨论了 Windows 的优先级调度机制。

进程和线程是现代操作系统中的重要概念，虽然本章始终围绕这两个概念来展开讨论，但它们并非独立于系统的其他组件而存在，所以，在讨论过程中难免引用到许多与系统同步和内存管理相关的知识，毕竟进程定义了一个内存地址空间，而线程总是在特定的地址空间中运行。接下来的两章我们将分别讨论 Windows 中的内存管理和同步机制。

4

Windows 内存管理

在现代计算机体系结构中，处理器所做的绝大多数工作要依赖于存储器的配合，虽然处理器本身也有存储数据的能力，可以利用自身的存储能力来快速地完成一些计算任务，但是，对于稍微复杂一点的算法，则必须要借助于处理器之外的存储能力。随着计算机处理器的计算能力不断提高，它们能够处理的数据量也在飞速地增长。因此，存储管理是操作系统的一项重要任务。在计算机存储体系中，内存直接受处理器控制和管理，即计算机的指令可以直接操纵内存单元中的数据，所以，内存管理是操作系统中除了进程和线程管理以外最为重要的一部分。本章我们将讨论 Windows 中的内存管理。

操作系统往往会结合所支持的硬件体系结构来选择相应的内存管理机制。由于内存管理的优劣会直接影响到系统本身的性能，所以，内存管理组件的框架结构通常在很大程度上是由处理器的硬件特性来决定的。譬如，Windows 支持的主流硬件体系结构是 Intel 的处理器，因而它的内存管理不可避免地受到 Intel 处理器硬件特性的影响。而另一方面，在多进程环境中，为了支持每个进程有独立的地址空间，以及进程地址空间和系统地址空间之间的隔离保护，Windows 在处理器寻址机制的基础上，又应用了大量的内存管理技术来满足各种需求。当多个进程的内存需求总和超过实际可用的物理内存时，Windows 还必须有效地平衡这些进程的内存需求，并借助于外部存储能力来达到它们的要求。

在这一章中，我们将首先概述一下现代计算机上主要的内存管理方案和算法，然后总体介绍 Windows 的内存管理机制。接着分别讲述 Windows 内存管理的 5 个主要方面：系统内存管理、进程内存管理、页面交换机制、物理内存管理和工作集管理。最后介绍一个内存监视工具 MemMon。

4.1 内存管理概述

在现代计算机系统中，内存（memory）是指处理器可以直接访问，但位于处理器之外的存储器。在硬件上，处理器通过一组地址线连接到这些存储器上，这组地址线构成了内存总线。而在软件上，处理器的许多指令允许用内存单元的地址作为指令的操作数，从而直接操纵这些内存单元。这里提到了内存地址，在 Intel x86 体系结构中，内存地址有三种类型：

- 物理地址。即内存存储器的索引，处理器操纵内存芯片时，通过地址线管脚加上电信号来读或写相应的内存单元。在 Intel x86 体系结构上，物理地址是一个 32 位或 36 位的无符号整数。
- 虚拟地址（有时候称为线性地址）。在 32 位系统上，虚拟地址空间可以达到 4 GB 大小，也就是说，整个空间可以有 2^{32}=4 294 967 296 个字节单元。Intel x86 芯片内有专门的电路负责把一个虚拟地址转译成物理地址。
- 逻辑地址。逻辑地址包含两部分：段（segment）和偏移（offset）。段部分指定了在整个地址空间中的一个基地址以及段空间的大小，当然还有段的一些其他属性。与寻址相关的是段的基址和大小。偏移部分指定了一个逻辑地址相对于段基址的偏移量。此偏移量不能超过段的边界。因此，逻辑地址的实际地址是段基址加上偏移量。Intel x86 芯片也有专门的电路把逻辑地址转译成一个虚拟地址或物理地址。

把一个地址告诉处理器，让它访问相对应的物理内存单元，这一过程是操作系统和处理器相互协作来完成的。处理器最终需要的是一个物理地址，所以，它必须把软件指令中的地址转译成物理地址，在转译过程中可能会涉及一些数据结构，甚至涉及 I/O 操作。在这一章中，我们将会看到 Windows 如何利用 Intel x86 的寻址特性来有效地实施内存管理。

另外，从操作系统的角度来看，一方面，它需要有效地管理所有的物理内存，使得当一个进程需要内存时，能够分配足够的内存单元给这一进程；另一方面，正如上一章所讲，进程代表一个相对独立的任务，它有一个逻辑上独立的地址空间。不同进程的地址空间应该是相互隔离的。也就是说，在一个进程中用地址 A 来访问的内存单元，与在另一个进程中用地址 A 访问的是不同的物理内存。这样做可以避免一个进程有意或无意地侵入到另一个进程的地址空间中。这种性质称为地址空间的隔离性，它使得每个进程有自己的私有地址空间。

虽然现代计算机配备的物理内存越来越多，比如现在的 PC 通常配有 1 GB~16 GB 内存，但是，当系统中进程数量增加以后，这些进程所需要的内存数量往往超过了机器的总物理内存，在这种情况下，操作系统必须合理地安排内存的使用，使得内存紧缺时，既不

会波及系统本身的稳定性，同时也不会严重影响系统的性能。在现代操作系统中，当发生这种情况时，一般的做法是把不紧急的进程中的数据或代码先存放到外存（通常是硬盘）中，从而把它们占用的物理内存腾出来给紧急的进程使用，或者交给系统使用。以后，当内存紧缺的状况缓解时，系统再把外存中的进程数据或代码装回到已经空闲下来的内存单元中，从而使这些进程有机会继续运行。这两个过程称为内存换出和换入。几乎所有的多进程操作系统都支持这种内存管理。

接下来两小节将分别介绍页式内存管理和段式内存管理，这是两种主流的内存管理方式，并且 Intel x86 处理器都支持。从寻址角度而言，页式管理的核心是如何把虚拟地址转译成物理地址；而段式管理的核心则是如何把“段+偏移”形式的逻辑地址转译成物理地址。这里，转译的机制是由芯片硬件提供的，但是转译中用到的数据结构则可能是由操作系统来管理的。此后，我们将讨论一些常用的内存管理算法，然后介绍 Windows 内存管理的总体结构，以及 Windows 内存管理器包含的一些组件。

4.1.1 页式内存管理

在如今的计算机体系结构中，内存基本单元总是字节，即 8 个二进制位。每个地址指向一个字节，地址值加 1 以后指向下一个字节。物理内存的编址，即前文讲到的物理地址，是固定的，所以，若直接让进程使用物理地址来访问内存将使得进程的动态分配难以有效实施，因为内存单元与进程将通过物理地址紧密地联系在一起了，从而内存的回收和再分配将受限于特定的进程和物理地址。为了打破这种关联关系，简单的思路是，让进程使用虚拟地址，而虚拟地址和物理地址之间通过一个映射表来完成转译。本节将要介绍的 Intel x86 的页式内存管理正是这样一种方案。

在页式内存管理中，虚拟地址空间是按页（page）来管理的，对应于物理内存也按页来管理，物理内存中的页面有时候称为页帧（page frame），其大小与虚拟空间中的页面相同。因此，映射关系是在内存页面的基础上进行的。在虚拟地址空间中连续的页面对应于在物理内存中的页面可以不必连续，并且，通过小心地维护好虚拟地址空间的页面与物理内存页面之间的映射关系，物理页面可以被动态地分配给特定的虚拟页面，从而只有当真正有必要的时候才把物理页面分配给虚拟页面，毕竟物理页面相对来说是稀缺资源。如图 4.1 所示，物理地址空间（其地址范围取决于系统中物理内存的数量，图中假设为 1 GB 物理内存）中有一部分页面被映射到了左侧的虚拟地址空间（在 32 位平台上，其地址范围为 0x0~0xffffffff）。

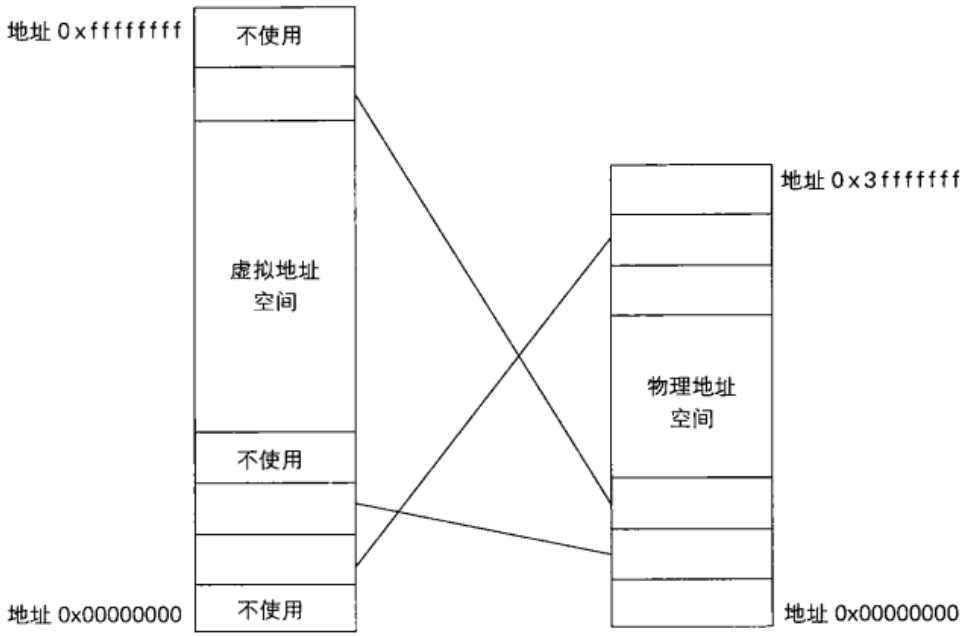


图 4.1 虚拟页面和物理页面之间的映射

注意，在一个系统中，物理地址空间只有一个，但虚拟地址空间可以有多个。每个虚拟地址空间都必须有一个映射关系，而且，虚拟地址空间中有相当一部分页面并没有对应的物理页面（在图 4.1 中标记为“不使用”的页面）。实际上，每个虚拟地址空间往往只能映射到很少一部分物理页面。反过来，每个物理页面往往只被映射到一个虚拟地址空间中。如果有一个物理页面被映射至两个或两个以上的虚拟地址空间，那么，这些地址空间将共享此页面，若在一个虚拟地址空间中改写了此页面中的数据，则在其他的虚拟地址空间中将可以看到这样的变化。

有了页面划分的机制以后，我们可以想象，每个虚拟地址 32 位信息中，其中一部分位信息指定了一个物理页面，其余的位信息则指定了页内的偏移量。也就是说，虚拟地址分成了两部分：页索引+页内偏移，其结构如图 4.2 所示。页索引是指该虚拟地址在映射关系中的索引编号，页内偏移则指定了该地址在页面内部的具体位置。

图 4.2 按页划分后的虚拟地址结构

下面我们以 Intel x86 为例来介绍从虚拟内存页面到物理内存页面的映射。首先，寻址系统必须确定页面的大小，标准的大小为 4 KB，即 2^{12} 字节，所以，32 位地址值的最后

12 位是页内偏移，而前 20 位则是页索引部分，用于找到一个实际的物理页面。因此，在这样的系统上，页面映射表是一个 2^{20}=1 048 576（即 1 M）大小的表。如果直接用一个简单的线性表来表达这一映射表的话，需要 4 MB 内存，因为表中每一项都要表达一个物理页面地址，即使考虑到页面起始地址的后面 12 位总是 0，那也需要 2.5 MB；如果再考虑到物理内存最高几位可能总是 0，则还可以有一定程度的压缩，这取决于当前系统中物理内存的数量，譬如 1 GB 内存的话，则物理内存地址的最高 2 位总是 0。这里我们忽略这些要素，假设映射表中的每一项都指定了一个 32 位物理内存地址。

另一方面，在此 1 M 大小的映射表中，有相当数量的表项其实并没有用，所以，为这部分表项分配存储空间是浪费。Intel x86 采用了分级页表的方式来管理这一映射关系。32 位虚拟地址中的页索引部分又被分成页目录索引（10 位）和页表索引（10 位）两部分，所以，一个 32 位虚拟地址的实际构成如图 4.3 中最上边的虚拟地址结构所示。

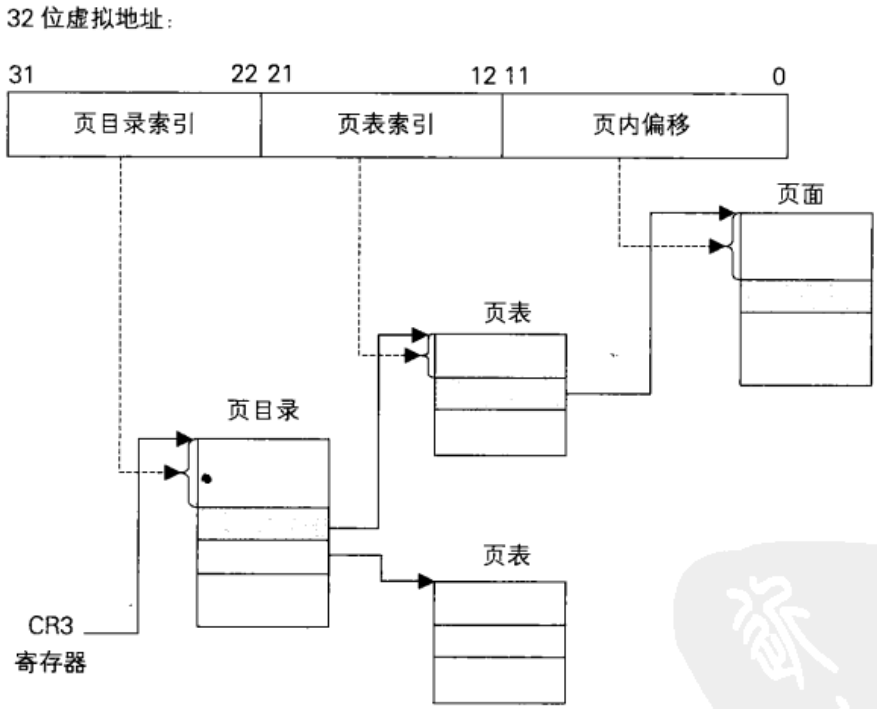


图 4.3 Intel x86 的 32 位虚拟地址的解析过程

基于这样的虚拟地址构成，每个虚拟地址空间对应有一个页目录，其中包含 2^{10}=1 024 个目录项（PDE，Page Directory Entry）；每一个目录项指向一张包含 1 024 项的页表。所以，Intel x86 处理器在解析一个虚拟地址时，首先根据最高 10 位在页目录中定位到一个页目录项，它指向一个页表。然后根据接下来的 10 位，在页表中定位到一个页表项（PTE，Page Table Entry），此页表项指定了目标页面的物理地址。最后在此物理地址的基础上加

上页内偏移，即得到最终的物理地址。在图 4.3 所示的地址解析过程中，实线箭头代表页目录、页表和页面之间的指示关系，而虚线箭头则代表了一个虚拟地址各个组成部分在页目录、页表或页面内部的索引关系（或偏移）。

在以上二级页表结构中，CR3 寄存器包含了页目录的物理地址，页目录的大小是 4 096 个字节，每个目录项为 4 个字节；同样，每个页表的大小也是 4 096 个字节，其中每个页表项为 4 个字节。目录项和页表项均指向一个 32 位地址，但只有前 20 位真正指向一个物理地址，后 12 位用于各种标志信息，比如页面是否已被访问过、是否允许缓存等。在本章后面介绍 Windows 虚拟内存管理时，我们再进一步讨论 PDE 和 PTE 的结构。

对于一个满的虚拟地址空间，为维护映射关系共需要 1 个页目录和 1 024 个页表，所占空间为 4 096+1 024 × 4 096 个字节，即 4 KB+4 MB 大小。这比简单的线性页表多了 4 KB 开销，但它带来的好处是，当虚拟地址空间中实际使用的内存的比例较小时，很多页表不必在内存中构建出来，从而可以极大地节省这些页表的开销。不过，二级页表结构也要付出性能的代价，因为在解析一个虚拟地址时，需要两次查表操作。

为了避免在多级页表解析过程中多次查表而导致性能下降的问题，Intel x86 处理器缓存了地址转译信息，即从虚拟地址到物理地址的映射关系。这样，当处理器重复访问同一个地址时无须再进行转译。此缓存是一种位于处理器内部的关联存储单元阵列，也被称为地址转译快查缓冲区（TLB，Translation Look-aside Buffer）。TLB 中包含了最近使用过的页面的内存映射信息，处理器提供了专门的电路来并发地读取并比较 TLB 中的页面映射项。因此，对于频繁使用的虚拟地址，它们很可能在 TLB 中有对应的映射项，因而处理器可以绝对快速地将虚拟地址转译成物理地址；反之，如果一个虚拟地址没有出现在 TLB 中，那么处理器必须采用以上介绍的两次查表过程（意味着要两次访问内存）才能完成地址转译。在这种情况下，这一次内存访问会慢一些，但是，经过这次访问以后，此虚拟页面与对应物理页面之间的映射关系将被记录到 TLB 中，所以，下次再访问此虚拟页面时，处理器就可以从 TLB 中实现快速转译，除非此映射项已经被 TLB 移除了。研究表明，由于计算机程序的内存访问有一定的局部性，因此，即使处理器只维护一个相对较小的 TLB，程序的运行也能获得较显著的性能提升[TLB-MISS]。

在 Intel x86 处理器中，TLB 是完全由硬件来维护的，换句话说，软件无法操纵 TLB 以便加入、保留或移除其中的映射项。（但其他有些处理器允许软件通过异常机制来操纵 TLB。）然而，有一种情况可以使 TLB 中的映射项失效，那就是当处理器切换 CR3 寄存器的时候，原因很简单，一旦 CR3 寄存器切换了，就意味着从一个虚拟地址空间切换到了另一个虚拟地址空间，因此原来那些项没有理由再保留。但有一个例外，如果映射项的 PTE

的全局标志位（32 位 PTE 的低 12 位都是标志位，其中第 8 位指明了这是一个全局项还是局部项）已置上，则在 CR3 寄存器切换虚拟地址空间的过程中，此映射项仍然留在 TLB 中。此外，在 Intel x86 Pentium Pro 以后的处理器中，通过 invlpg 指令可以使单个 TLB 项失效。

以上介绍了在 Intel x86 处理器中，一个 32 位虚拟地址如何被转译为一个物理地址。另外，Intel x86 Pentium Pro 处理器还引入了一种称为物理地址扩展（PAE，Physical Address Extension）的内存映射模式，它支持 36 位物理地址，但虚拟地址仍然是 32 位。因此，在 PAE 模式下，系统支持 64 GB 物理内存，其地址映射方式和页表结构也有所不同。虚拟地址的转译采用了三级页表机制，如图 4.4 所示。在页目录之前增加了一个页目录指针索引，而页目录和页表中的每一项都为 64 位，所以，4 KB 大小的页目录和页表只能存放 512 项，正好对应于虚拟地址中的 9 位索引值。页目录指针表包含 4 项，分别指向 4 个页目录之一。由于页目录和页表中的每一项都变成了 64 位，因此它们可以描述更长的物理地址。若简单地增加 4 位物理地址，即从原来的 20 位（注意，还有 12 个标志位）扩展到 24 位，则处理器将允许系统使用 64 GB 物理内存。Windows 用 26 位来表达物理地址，因此可以支持 2^{26+12}=256 GB 物理内存。

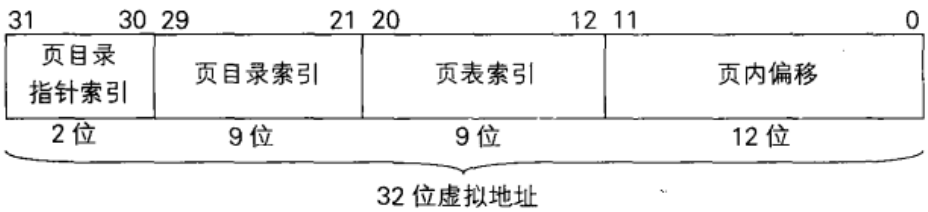


图 4.4 Intel x86 PAE 模式下的虚拟地址组成结构

简而言之，PAE 模式并没有增加虚拟地址空间的大小，但允许系统支持更多的物理内存。最后顺便提一下，在 Intel x64 处理器中，支持更多级页表机制。例如，64 位 Windows 使用四级页表，将一个 48 位虚拟地址转译成一个 40 位物理地址（从而允许物理内存达 1 TB 大小）。在 WRK 的代码中，我们可以看到 Windows 对于 Intel x86 PAE 模式和 64 位处理器的支持，但是，本书只讨论 Windows 对于 Intel x86 的 32 位物理内存的支持。

4.1.2 段式内存管理

除了用虚拟地址来实现有效和灵活的内存管理以外，在计算机发展史上，另一种重要的内存管理方式是将物理内存划分成若干个段（segment），处理器在访问一个内存单元时，通过“段基址+偏移”的方式计算出实际的物理地址。每个段都可以有自己的访问属性，包括读写属性、特权级别等。例如，在 Intel x86 处理器中，有专门的段寄存器，允许每条指

令在访问内存时指定在哪个段上进行。段的概念在 Intel 8086/8088 实模式中已经使用了，但当时段的用途是扩展地址范围，将系统的内存从 64 KB 扩展到 1 MB。处理器中的寄存器（包括段寄存器）都是 16 位的，但地址线有 20 根。为了访问整个地址范围，处理器的做法是，将段寄存器中的值左移 4 位再加上一个 16 位地址值，就形成了目标地址。实际上，这么做可以达到的最大地址值是(64 K−1) × 16+(64 K−1)=1 114 095。到了 80286 处理器（24 位地址），在保护模式下，段变成了一个索引，指向段描述符表中的某一个段描述符，而段描述符才真正指定了段的基地址、段长度以及一些保护属性。而到了 80386 以后，处理器中的寄存器才真正转移到了 32 位。本节接下来将基于 32 位的段寻址模式来讨论段式内存管理。

在 Intel x86 中，逻辑地址的段部分称为段选择符（segment selector），指定了段的索引以及要访问的特权级别。段寄存器 cs、ss、ds、es、fs 和 gs 专门用于指定一个地址的段选择符。虽然只有这六个段寄存器，但是软件可以灵活地使用它们来完成各种功能。其中有三个段寄存器有特殊的用途：

- cs：代码段寄存器，指向一个包含指令的段，即代码段。
- ss：栈段寄存器，指向一个包含当前调用栈的段，即栈段。
- ds：数据段寄存器，指向一个包含全局和静态数据的段，即数据段。

虽然地址寄存器和数据寄存器都是 32 位，但是段选择符只有 16 位，其格式如图 4.5 所示。

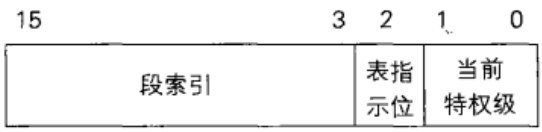


图 4.5 段选择符格式

在段选择符中，段索引指定了一个段在段描述符表中的编号，它有 13 位，这也说明了一个段描述符表只包含 2^{13}=8 192 个段；表指示位说明了此段位于全局描述符表（GDT，Global Descriptor Table）还是局部描述符表（LDT，Local Descriptor Table）中。当前特权级（CPL，Current Privilege Level），也称为请求者特权级，是一个两位的值（0~3），代表了请求者的当前特权级别。特权级是 CPU 的运行模式，0 表示最高特权级，3 表示最低特权级。Windows 和 Linux 只使用这两种特权级，分别称为内核模式（kernel-mode）和用户模式（user-mode）。

在介绍 GDT 和 LDT 两张表以前，我们先来看一看段描述符的内容。每个段描述符用来定义一个段，其中包括段的起始地址、有效范围和一些属性。图 4.6 给出了段描述符的结构。段描述符指定了 32 位基地址，以及 20 位段长度（即段内最大偏移）。当 G 位为 0

时，此长度单位为字节；当 G 位为 1 时，此长度单位为 4 096 字节。所以，段长度可达 $2^{20} \times 4\,096$=4 GB，即整个 32 位线性地址空间。描述符特权级（DPL，Descriptor Privilege Level）是允许访问此段的最低特权级，比如，DPL 为 0 的段只有当 CPL=0 时才可以访问，而 DPL 为 3 的段，可由任何 CPL 的代码访问。类型域（共 4 位）指定了段的类型，包括代码段、数据段、TSS 段和 LDT 段。

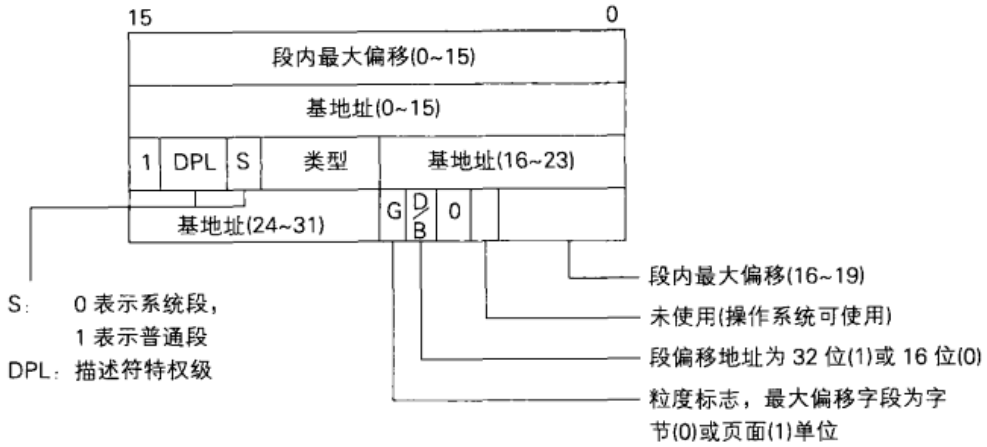


图 4.6 段描述符格式

接下来讨论 GDT 和 LDT。顾名思义，全局描述符表 GDT 是系统全局范围内有效的一张表，它包含最多 8 192 个段描述符，所以，一张完全的 GDT 表需要 8 192 × 8=64 KB 内存空间。CPU 有一个寄存器 gdtr 包含了 GDT 的地址。除了 GDT 外，处理器另有一个 LDT，它也同样最多包含 8 192 个段描述符，对应于 LDT 的寄存器为 ldtr，它包含了 LDT 的地址。现在我们可以理解 Intel x86 转译一个逻辑地址的过程，如图 4.7 所示。

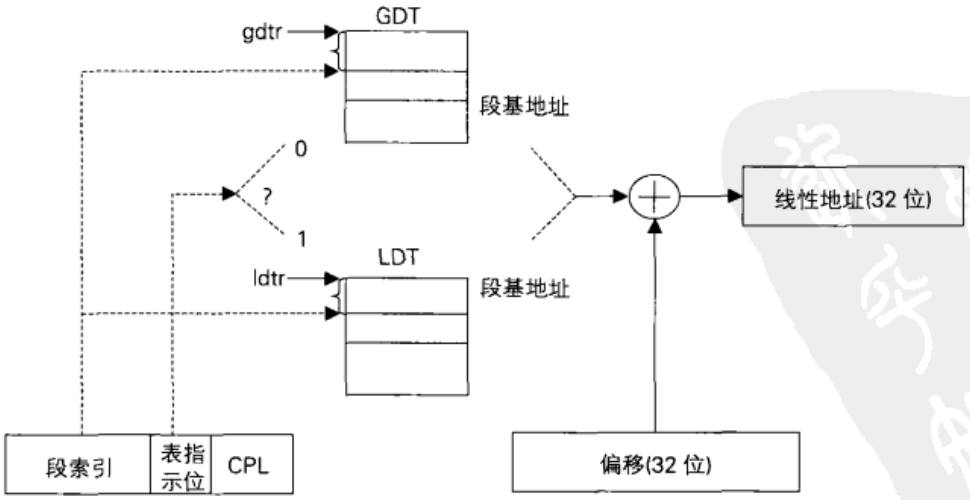


图 4.7 Intel x86“段+偏移”形式的逻辑地址的解析过程

处理器在解析一个“段+偏移”的逻辑地址时，首先根据段寄存器中的表指示位确定应该使用 GDT（若表指示位为 0）还是 LDT（表指示位为 1），然后从 gdtr 或 ldtr 中得到描述符表的地址，再加上段索引部分乘以 8，即得到段描述符的地址，然后根据段描述符的格式，拼出 32 位段基地址，最后加上 CPU 指令中的偏移值，得到最终的线性地址。在实际执行指令过程中，每个段寄存器内部都有一个 8 字节的缓存（或称为内部寄存器），存放了对应于段寄存器的段描述符，如果段寄存器没有改变，则以上的地址计算过程可以省略查描述符表的步骤，从而直接在处理器内部计算出线性地址。

现在我们可以想象一下，如何利用以上介绍的段式内存访问机制来实现操作系统中的多进程地址空间。显然，一种自然的设计思想是，系统全局共享的空间可以通过 GDT 来安排和访问，比如，操作系统本身的代码和数据是系统全局可见的，各个进程的地址空间中不可避免地要包含这部分内存。此外，各个进程私有的数据和代码存放在 LDT 中，因而进程切换时，只需改变 LDT 表，即可实现进程私有地址的切换。在现代的应用程序中，每个进程往往包含多个二进制模块，以及一些动态数据区；各个二进制模块中既有代码，也有数据（比如全局变量和静态变量等）。所以，如果用段机制来管理内存的话，每个模块都需要一个段，动态数据区往往需要一个或多个段（比如全局堆和局部堆，以及栈等）。对于绝大多数应用程序，8 192 个段（指 LDT 中的段）足够使用了。对于系统全局空间，只要小心地安排好段的使用，8 192 个段（指 GDT 中的段）也能满足正常的内存分配。利用这种方法，既可以做到进程之间的空间隔离性，也可以很方便地在进程之间共享数据。

最后需要说明的是，段式内存管理和页式内存管理并不是对立的，它们可以组合起来在同一个系统中使用。事实上，Intel x86 处理器的内存管理单元（MMU，Memory Management Unit）结合了这两种寻址方法。例如，图 4.8 显示了一个逻辑地址被解析成虚拟地址，再进一步被解析成物理地址的全过程。操作系统可以有选择地使用段式内存管理单元或页式内存管理单元来管理进程地址空间和系统物理内存。Windows 和 Linux 都选择了页式内存管理作为主要的内存管理手段，但同时也不可避免地涉及了段机制。我们在 Windows 和 Linux 的内核代码中都可以看到有关 GDT 和段操作的代码。

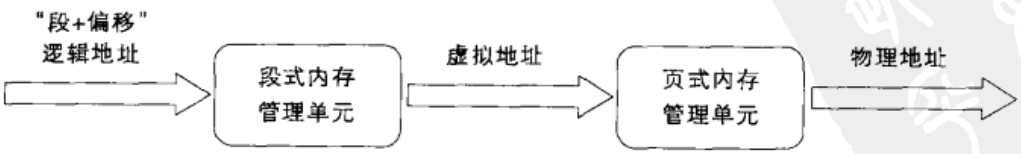


图 4.8 Intel x86 逻辑地址的完整解析过程

4.1.3 内存管理算法介绍

内存是计算机系统中除了处理器以外最为重要的资源，任何一个程序的运行都离不开

内存资源的有效使用。前面两小节介绍了硬件支持的内存管理机制，尤其是如何将虚拟地址或者逻辑地址转译成物理内存地址。这一节我们将首先讨论在一个地址空间内部如何有效地进行动态内存管理，然后介绍常用的页面替换算法，以及在进程内存管理中常常用到的工作集概念和相应的算法。

假设操作系统或者一个进程已经获得了一块连续地址的内存，系统或进程在执行过程中需要利用这块内存来满足各种内存请求。由于内存请求存在动态性，即每次请求的内存大小可能不相同，甚至差异很大，并且这些小内存块的生命周期也不尽相同，所以，系统需要提供合适的算法来尽可能地满足这些动态的内存请求。在现代计算机系统中，堆（heap）正是这样一个提供动态内存分配能力的内存抽象。操作系统使用堆来满足各种动态内存请求，应用程序通过堆获得内存。我们常用的 C/C++基本运行库提供了堆内存管理的能力，所以，C/C++程序中的 malloc/free 和 new/delete 可以直接在堆的接口上工作。

就本质而言，内存管理算法可以分为两大类：位图标记法和空闲链表法。**位图标记法**的思路很简单：假设总的待分配内存的大小为 N 字节，管理内存的粒度为 M 字节，并且 $N=M\times K$，也就是说，内存管理的基本单元为 M 字节，现在共有 K 个单元需要动态管理。为了记录这 K 个内存单元的使用情况，位图标记法将使用一个共有 K 位的位图（bitmap），其中每一位的值（0 或者 1）用来说明这一位所对应的内存单元是否空闲。由于位图精确地记录了每一个内存单元的空闲或已被使用的情况，所以，当内存管理器接收到一个新的内存申请时，只须扫描位图，就能确定是否有合适的空闲内存可以满足此请求。其做法是，根据所请求内存的大小，确定需要多少个连续的内存单元来满足此请求，然后在位图中扫描是否存在连续这么多 0 位，如找到了，则将它们所对应的内存分配给客户，并且将这些位置成 1。当释放内存时，要求客户指定待释放内存的起始地址和大小，这样内存管理器就可以计算出此次内存释放对应于位图中哪些连续位，并且将它们置成 0。

位图标记法的实现比较简单，但是需要额外的内存开销，通常为 $\frac{N}{M\times 8}$，所以，只需适当地选取 M，就可以控制这部分额外开销。当然，用户请求的内存大小不一定是 M 字节的倍数，因而在分配内存时有一定的浪费。平均而言，每次用户请求将导致 $M/2$ 字节的浪费。另外，内存管理器在分配内存时需要扫描连续多个 0 位，此操作并不高效（复杂度为 $O(K)$），这是该算法的一个缺点。

另外一类动态内存管理方法使用链表来描述已分配的和空闲的内存块，称为**空闲链表法**。在初始时，整个内存块被当做一个大的空闲块加入到空闲链表中。以后，当内存管理器接收到一个内存分配请求时，将从空闲链表中找到一个合适的、能提供足够内存的空闲块，并从该空闲块中分离出足够多的内存，交给客户，剩下的内存（如果还有的话）仍然

是一个空闲块，而已分配的内存则加入到已分配内存链表中。当释放一块内存时，内存管理器将已分配的内存块从已分配链表转移到空闲链表中，如果有可能，与相邻的内存块合并以便构成更大的空闲块，从而尽可能地满足客户的大内存请求。在这一类方法中，当内存管理器接收到内存请求时，将按照以下不同的策略来查找适当的空闲内存块：

(a) 最先匹配法（first-fit），从空闲链表中找到第一个满足客户请求的空闲块。

(b) 最佳匹配法（best-fit），从空闲链表中找到最接近于客户请求大小的空闲块。

(c) 最差匹配法（worst-fit），从空闲链表中找到最大的空闲块。

(d) 下一个匹配法（next-fit），从空闲链表的当前位置往后扫描，找到第一个满足客户请求的空闲块。

除了以上介绍的位图标记法和空闲链表法以外，还有两种内存管理算法也值得介绍一下：slab 算法和伙伴系统（buddy system）。**Slab 算法**实际上是以上介绍的位图标记法和空闲链表法的结合，它针对某个阈值以下的内存块使用位图法，按照 2 的幂次，每一阶有一块内存和对应的位图；在此阈值以上，slab 算法使用链表来管理内存。Linux 和 Solaris 系统的内核使用了 slab 算法。

Slab 算法对于小内存的分配非常快速高效，但也有空间浪费，当所请求的内存大小介于 2 的两个连续幂次之间时，所分配的内存块（大小为 2 的幂次）就存在部分空间浪费。浪费的部分称为内碎片（internal fragmentation），因为它位于已分配的内存块内部。对应地，如果碎片位于已分配的两个内存块之间，则称为外碎片（external fragmentation）。例如，前面介绍的空闲链表法在经过一段时间的动态内存分配和释放以后，往往会造成很多外碎片，因此，即使实际的空闲内存还有很多，但由于外碎片的原因，对于较大内存的申请也往往无法满足。

解决外碎片问题的一个算法是**伙伴系统**（buddy system）。下面以二进制伙伴系统为例来说明它的主要思想。首先，假设待分配的整块内存的大小为 2 的幂次，比如说 2^m 字节，每个字节相对于基地址的偏移量为 0，1，2，…，2^m-1；另有一个数组 avail[m]，其中每个元素 avail[i]记录了大小为 2^{i+1} 的内存块的空闲链表头。内存的分配按照 2 的幂次进行，也就是说，分配给客户的内存块总是 2 的幂次方，不管客户是否真正需要这么多内存。对于任一大小为 2^i 的内存块，假设其相对于基地址的偏移量为 p，则该内存块的伙伴被定义为，p 的第 i+1 位取补码加上基地址，这样得到的地址所指向的同样大小的内存块。图 4.9 演示了伙伴内存块和非伙伴内存块。

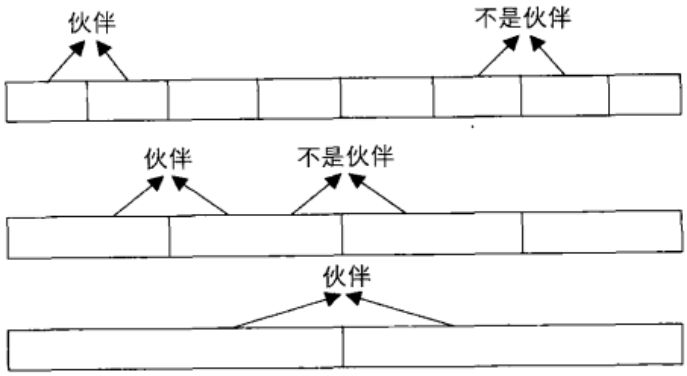


图 4.9 伙伴系统中的伙伴内存块

伙伴系统在初始时，整个待分配内存块都是空闲的，所以，avail[m-1]链表指向此内存块，该数组中其他所有的链表都是空的，当客户申请一块大小为 k 的内存块时，伙伴系统在所有≥⌈log$k$⌉的 avail[]链表中查找空闲块，从第一个找到的非空链表中找出一个内存块，经过分裂变成合适大小，然后返回给客户。因此，伙伴系统的内存分配过程实际上是大内存块分裂成小内存块的过程，分裂得到的小内存块一部分给客户，剩下的挂到适当的空闲链表中，以备下次继续分配给客户。

在内存回收过程中，如果待回收的内存块与链表中已有的一块内存互为伙伴，则它们可以合并成更大的内存块，从而转移到大内存块对应的空闲链表中。因此，内存块的合并在互为伙伴的内存块之间进行，并且从小到大，一直到不能合并为止。

伙伴系统的内存分配和回收的执行效率比较高（O(log$n$)），但也有问题：第一，空间利用率的问题，由于它总是按照 2 的幂次来分配内存块，所以，如果客户总是按照略大于 2 的幂次来申请内存，则空间浪费的现象将较为严重；第二，外碎片问题仍然存在，例如，两个相邻的非伙伴内存块即使能满足客户的内存要求，伙伴系统也不会把它们连起来分配给客户。有一些改进的伙伴系统能缓解这些问题，但效率可能不如二进制伙伴系统这么好。关于伙伴系统的更详细信息，读者可以参考相关文献[TAOCP-1]。

接下来我们讨论在页式内存管理系统中，当物理内存紧缺时，该从哪些进程中选择哪些页面，把它们的内容写到磁盘上，从而腾出这些页面所对应的物理页面，以便用于后续的内存需要。由于在一个实际的多进程、多任务系统中，所有进程使用的页面总数可能会超过系统中的页面数量，因此，当一个进程向系统请求更多的物理页面时，系统必须有一套算法或策略来保证适时地满足该进程的需要。对于操作系统而言，这实际上也是在页面粒度上的物理内存管理，其中的算法往往称为**页面替换算法**。通常以下一些算法是值得考虑的[MOS]：

最优页面替换算法（The Optimal Page Replacement Algorithm）。这是一个理论上

最优的算法，它要求能够预测每个页面下次使用的时间，从而确定该页面还需要等待多长时间才会真正使用，所以，在选择该换出哪些页面时，优先考虑那些等待时间最长的页面。此算法的基本原理是，越是频繁使用的页面，越应该留在物理内存中；相对而言，如果要换出换入的话，应尽可能选择那些不会被频繁使用的页面，或者在某段时间内不会被频繁使用的页面。最终达到的效果是减少页面换入换出的次数。但是，这一算法的问题是，预测一个页面下次被使用的时间在实际系统中往往是不可行的，除非是在做事后分析。因此，这种算法可以当做一个基准来对已有的算法做性能评价，具体做法是，在一个系统中记录下每次页面被访问的历史痕迹数据，有了这些数据以后，就可以在模拟环境中，使用最优页面替换算法，得到最少的换页次数，作为被评估算法的一个理论最优参照。

最近未使用（NRU）页面替换算法（The Not Recently Used Page Replacement Algorithm）。这一算法的思路是，当系统需要物理页面时，检查所有的页面，优先替换那些最近（所谓最近，是一个相对时间，比如，最近的几个时钟滴答）一直没有被访问或修改的页面。为了建立起最近是否被访问或修改的参考依据，通常每个页面需要记录下它被访问的情况，从效率来考虑，这往往需要硬件的支持。有两个标志位具有特别的意义：访问位R和修改位M。当一个页面被初次使用时，它的访问位R被置上，若页面被修改，则修改位M被置上。在进程的运行过程中，R位被定期清除，这样系统就能区分最近未被访问过的页面和最近被访问过的页面。当需要替换页面时，首先找那些未被访问过的页面；如果还不够，则继续找已被访问但未被修改过的页面；如果还需要更多的页面，则只好找那些已被访问且已被修改过的页面。这一算法相对简单并且易实现，尤其在硬件的支持下可以有较好的性能，虽然它不是最优的，但在实用中是有效的。

先进先出页面替换算法（The First-In First-Out Page Replacement Algorithm）。顾名思义，这种算法的思路是把所有已在内存中的页面组织成一个队列（也可以是一个链表），每次当有页面换入到内存中的时候，就添加到队列的末尾；当需要页面换出时，直接从队列中移除页面。这一算法听起来很有道理，把留在内存中时间最长的页面换出内存，但实际上，经常要访问的页面也不得不在队列中流动，从而会造成不必要的换出和换入开销。由于这样的原因，在实践中，这一算法很少被单独使用。

第二次机会页面替换算法（The Second Chance Page Replacement Algorithm）。这是对先进先出页面替换算法的改进，很自然，对于最老的页面，即队列头的页面，如果它的访问位R为0，则说明这个页面不仅老，而且很久没用了，理应换出去；如果R位非0，则说明该页面最近被访问过，因此再给它一次机会，做法是，先把R位清零，然后把它移到队列尾，就好像它是一个新换入的页面一样。然后系统再进一步检查队列头的页

面。如果队列中的页面最近都被访问过，那么，它们将被依次检查一遍并清除其访问位，然后在下次再被检查到的时候被依次换出内存。

第二次机会页面替换算法在具体实现的时候，有一种优化方法，它可以避免在队列中频繁地移动页面，而是把页面组织成环形链表，然后用一个指针指向时间上最早加入的页面。这样做的好处是，如果有一个页面被检查了访问位以后，并不从链表中清除，则只须清除其访问位，并移动指针指向下一个页面即可，无须将页面从链表头移到链表尾。由于这种做法就像一个时钟的指针在钟面上移动一样，因此该算法有时候也被称为**时钟页面替换算法**（The Clock Page Replacement Algorithm）。其实质跟第二次机会页面替换算法完全相同，只是实现上不同而已。

最近最久未使用（LRU）页面替换算法（The Least Recently Used Page Replacement Algorithm）。这一算法的思路是，当系统在选择换出一个页面的时候，优先考虑最近最久未使用的那个页面。此算法实际上是对最优页面替换算法的一个估计，其依据是页面访问的局部性原理。既然无法直接测量页面将来被访问的时间，不妨用最近一段时间内页面被访问的频率来猜测它将来被访问的情况，从而协助作出页面替换决策。若在最近一段时间内，某些页面被频繁地访问，则在将来的一段时间内，它们还可能会被频繁地访问。反之，若某些页面长时间未被访问，则在将来，它们极有可能仍然长时间不会被访问，所以，在选择页面的时候优先考虑这些页面。

最近最久未使用（LRU）页面替换算法与前面介绍的最近未使用（NRU）算法并不一样。NRU 算法基于页面的访问位和修改位来作出决定，而 LRU 算法基于页面最近被访问的时间长短来作出选择。在实现 LRU 算法时，要求能够定位到最久没有使用过的页面，这可以通过维护一个页面链表来实现，但每次访问一个页面都要把这个页面移到链表首，表示它刚刚被访问过。因而算法的维护成本较高，难以硬件实现。

最不经常使用（NFU）页面替换算法（The Not Frequently Used Page Replacement Algorithm）。此算法的思路与 LRU 一致，它主要是提供了一种软件实现。其做法是，为每个页面维护一个计数器，初值为 0。每次时钟中断时，系统对所有页面，把它们的访问位（0 或者 1）加到计数器上，这样，经常被访问的页面就有机会增加其计数器，而不常访问的页面其计数器相对较少得到增加。但这个算法的问题在于，计数器只增不减，这意味着页面的历史会长久地影响页面替换算法的决策。

对 NFU 算法的一个改进算法称为**页面老化算法**（page aging algorithm）。它对 NFU 做了修改，使其更好地模拟 LRU 算法。其做法是，在时钟中断修改页面计数器的时候，

并不是简单地递增计数器，而是先把计数器的值右移一位，然后把访问位 R 加到计数器的最左边位上，而不是最右边位。经过这样修改以后，如果一个页面经过了一段频繁访问的时间过后，它慢慢地不再被访问了，则因频繁访问而对计数器的影响在经过几次右移位以后，逐渐消除了，取而代之的是最近该页面被访问的情况。

老化算法只用有限个位来模拟页面最近被访问的情况，它提供的计数器并非精确的时间计数值，而只是一个相对的最近被访问的参照，但是其优点在于，它能够逐渐地抹去历史的影响，而让最近一段时间的被访问情况参与到决策中。在实践中，这些相对久远的历史对于决策的重要性并不高，所以，老化算法比较具有实际意义。

以上介绍了操作系统在替换页面时的一些常用算法和选择依据。现在我们来看一看，系统在进程层次上是如何管理和控制物理内存资源的。这对于多进程系统有重要的意义，因为尽管每个进程都有非常大的虚拟地址空间（比如在 32 位 Windows 上有 2 GB 或 3 GB 私有的虚拟空间），但它们能得到的物理内存往往只是相对较少的一部分，进程之间实际上是在争抢有限的物理内存资源。所以，操作系统必须小心地平衡每个进程的需求和分配给它的内存。为了衡量进程得到的物理内存资源，这里首先介绍进程工作集的概念。

当一个进程被创建并开始运行时，最初所有的页面都还在磁盘上（包括进程的可执行文件），随着控制流不断前进，全局数据和栈的地址范围被不停地访问到，并且动态内存的需求也开始出现，该进程逐渐获得越来越多的物理内存页面。对内存页面的请求和满足通常是以中断或异常的方式来完成的。随着进程占有的物理内存越来越多，它的运行趋向平滑，因为对于物理内存的需求开始减少。操作系统根据需要而分配物理页面的做法称为按需换页（demand paging）。

结合前面介绍的页面替换策略，我们可以理解：操作系统在内存紧缺时将根据一定的算法和规则，向进程要回物理页面（也就是说，选择哪些页面被替换）；而进程则在必要的时候向系统请求更多的物理页面。操作系统就在这两者之间管理着有限的物理内存资源。那么，对于一个进程而言，一方面，当它占用物理内存太多时，自然地，有些页面会被操作系统收回去；另一方面，当它的执行逻辑需要更多物理内存时，操作系统可以把当前空闲的或者收回来的（既可能从其他进程收回来，也可能从它自身所占的内存中收回来）物理页面分配给它。在任一给定时刻，进程所占的物理内存是确定的；从一个过程来看，它所占的内存数量呈现出动态变化的特性。工作集模型正是刻画一个进程的内存使用情况的模型。这里，**工作集**（**working set**）指一个进程当前正在使用的物理页面的集合。

应用程序在工作时，对于内存的访问通常呈现出一定的局部性，也就是说，在一段时

间内，程序对内存的访问往往集中在一定的范围内。这也意味着，进程的工作集的变化相对而言是缓慢的。进程工作集变化越缓慢，则单位时间内页面换入换出发生的次数越少，这当然有利于进程的运行，它的性能自然越好；另一方面，进程的工作集也是由操作系统来管理和控制的，工作集越大，则它的变化自然越缓慢。因此，工作集管理也是操作系统内存管理的一个重要方面。

在工作集理论模型中，进程的工作集可以用一个二元函数 $w(t, \delta)$来表示，其中 t 代表时间点，δ代表一段时间间隔，也称为工作集窗口。$w(t, \delta)$表示 t-δ与 t 两个时间点之间进程所访问到的页面集合，显然，随着δ的增大，$w(t, \delta)$只可能增加而不会减小，即 $w(t, \delta)$是δ的单调非递减函数。但由于程序的内存访问的局部性原则，$w(t, \delta)$的递增在δ较小时很快，然后就会稳定下来，其曲线大致如图 4.10 所示。当δ大到一定程度，$w(t, \delta)$可能又会有一段快速增长，然后稳定下来。当δ较大时，w 曲线取决于程序的执行逻辑。

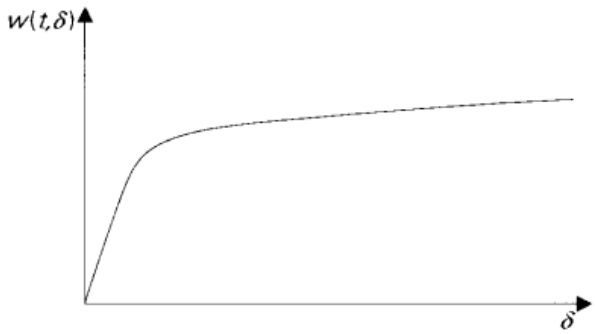


图 4.10 工作集理论模型

工作集理论模型可以用来指导对进程页面的有效管理，例如，在进程初始执行期间，进程的工作集很快增长，但到一定时候，工作集就会稳定下来。因此，一种有效的优化手段是，记录下工作集稳定下来以后某一时刻的工作集内容（即进程中的哪些页面被访问了），下次该进程启动时，直接为这些页面赋予物理内存，并且从磁盘加载相应的内容，这样可以避免以按需换页的方式逐渐地从磁盘读取文件内容，从而大大加快进程的启动速度。Windows 使用了这种优化手段，称为逻辑预取器（Logical Prefetcher）。

那么，如何维护进程工作集信息呢？一种简便的做法是，记录每个页面最近被访问的时间，这样，根据预设的δ值，一旦当前时间超过了该页面最近被访问的时间再加上δ值，则从工作集中删除此页面。根据工作集理论模型，预设的δ值（当然不能太小）对于工作集并没有很大的影响。而且，工作集的这一维护机制可以与页面老化算法有机地结合起来，在实践中这不难做到，例如直接根据页面的老化程度来决定是否从工作集中移除一个页面。

有了进程工作集的信息以后，我们可以用这些信息来改进页面替换算法。例如，在前面介绍的时钟页面替换算法中，如果指针所指的页面的访问位为 0，则意味着该页面可以被替换，但现在有了工作集信息以后，需进一步检查此页面是否属于当前进程的工作集，如果是，则不被替换，算法继续往前查找其他的页面。这一改进算法称为 WSClock[WSCLOCK]。

4.1.4 Windows 内存管理概述

以上介绍了现代计算机体系结构支持的两种虚拟内存管理机制以及操作系统在管理内存时涉及的算法和概念，这一节我们来看一看 Windows 内核中的内存管理器如何有效地管理系统的物理内存以及每个进程中的虚拟内存。

首先，Windows 采用了页式内存管理方案，在 Intel x86 处理器上，Windows 不使用段来管理虚拟内存，但是，Intel x86 处理器在访问内存时必须要通过段描述符，这意味着 Windows 将所有的段描述符都构造成了从基地址 0 开始，且段的大小设置为 0x80000000、0xc0000000 或 0xffffffff，具体取决于段的用途和系统设置。所以，Windows 系统中的代码，包括操作系统本身的代码和应用程序代码，所面对的地址空间都是线性地址空间。这种做法相当于屏蔽了处理器中的逻辑地址概念，段只被用于访问控制和内存保护。

正如我们在第 2 章中所讲，Windows 使用了两种特权级别：0 和 3，其中特权级 0 称为内核模式，特权级 3 称为用户模式。当处理器执行内核模式代码时，它们处于系统地址空间，位于 0x80000000~0xffffffff，所有的进程共享此空间；当处理器执行用户模式代码时，它们处于进程地址空间，位于 0x00000000~0x7fffffff，这部分空间是进程私有的。用户模式代码只能访问进程自身的数据，而内核模式代码不仅可以访问当前进程的数据，也可以访问系统地址空间中的数据。所有的进程，一旦进入到内核模式，则共享同样的系统地址空间。

其次，在 Windows 的每个地址空间中，虚拟地址的分配和回收都必须按照严格的规则进行。Windows 规定，应用程序在使用内存以前必须先申请，所以，操作系统内部可以根据应用程序的申请和释放操作来维护好整个虚拟地址空间的内存分配情况。而且，Windows 也采用了按需分配的策略，也就是说，只有当一段虚拟内存真正被使用的时候，系统才会为它分配页表和物理页面。每个进程的虚拟地址空间的分配情况通过一组虚拟地址描述符（VAD，Virtual Address Descriptor）记录下来，这些描述符构成了一棵平衡二叉树，以便于快速地定位到一个指定虚拟地址的描述符上。

在 Windows 内存管理器中，除了虚拟地址空间的管理，另一重要职责是管理物理页面，以及实现进程页面的换入和换出过程。在 Windows 中，页帧编号数据库（Page Frame

Number Database，简称 PFN 数据库）描述了物理内存各个页面的状态。PFN 数据库实际上是一个结构数组，每个页面对应有一个 PFN 项，记录了该页面的使用情况，包括它的状态、对应页表项的地址等信息。此外，操作系统还维护了一组链表，分别将相同类型的页面链接起来，比如，所有空闲的页面都插入到一个空闲链表中，所以，操作系统可以快速地从该链表中获得一个空闲的物理页面。页面也会随着其状态的变化，以及被进程的使用情况，而在不同的链表中流动，或者不属于任何一个链表。Windows 的页面替换算法也正是在这些数据结构的基础上实现的。

虽然 Intel x86 定义了虚拟地址与物理地址之间的转换方式，但是页目录和页表需要操作系统来维护，Windows 定义了 PDE 和 PTE，并且小心地维护好这些数据结构，以便处理器能够正确地转译虚拟地址。我们在上一章的 KPROCESS 数据结构中看到的 DirectoryTableBase 域正是指向进程页目录的指针，所以，操作系统在创建每个进程时，都需要为这个进程建立一套页目录和页表数据结构，从而建立起该进程的地址空间。在 4.3.1 节，我们将会看到一个地址空间是如何建立起来的。

地址空间建立起来以后，并不等于所有的页面都分配好了，Windows 采用的是按需换页的策略，当进程或系统使用某个尚未得到物理页面的虚拟地址时，处理器会触发页面错误（page fault）异常，所以，操作系统可以在页面错误的异常处理例程中为其分配页面，并设置好页表项和页面之间的逻辑关系。这一异常的发生对于使用此虚拟地址的代码而言是不可见的，一旦异常处理例程完成，则原来的代码指令继续执行，就好像此换页过程什么也没有发生一样。

什么时候页面被换出呢？当系统认为内存紧缺，或者一个进程由于工作集的限制而不允许拥有更多的物理页面时，Windows 会替换工作集中的页面。这项工作是由一个称为工作集管理器（working set manager）的组件来完成的，它运行在一个称为平衡集管理器（balance set manager）的系统线程中。工作集缩减的过程称为修剪（trim）。Windows 实现了全面的工作集模型，包括控制进程的工作集，以及与页面替换算法结合起来管理物理内存的全局分配。

至此，我们已经理解了 Windows 操作系统中内存管理的基本概况，下面列出在 Windows 内存管理中一些比较重要的组件：

- 执行体层提供了一组内存管理服务，用于分配、释放和管理虚拟地址。执行体也包括一个堆管理器（heap manager），提供动态小内存块的管理。

- 页面错误异常处理器（或称为页面错误例程）。负责分配物理页面，或者把磁盘上的数据读入到页面中。
- 一组系统线程，负责维护操作系统的内存，其中包括：
  - 平衡集管理器，它包括工作集管理器，每 1 s 被调用一次，工作集管理器负责实施一些全局性的内存管理策略，比如工作集修剪。
  - 进程/栈交换器。当系统需要执行换入或换出操作时，通知此线程完成这些任务。第 3 章介绍线程状态转移时，我们曾经看到，如果一个进程的所有线程都处于等待状态，则它有可能被换出内存，即线程进入转移状态，以后该进程满足特定条件时再换回内存。此换入换出过程正是由进程/栈交换器来完成的。
  - 修改页面写出器。它负责将脏页面写回到映射文件或页面文件中。在 WRK 中，写到映射文件和写到页面文件是分别由两个线程来完成的。
  - 零页面线程。它运行的优先级为 0，负责将空闲链表上的页面清零（即内存零化），以便当系统需要零页面时可以满足其要求。

系统地址空间包含了全局的系统代码和数据结构，并且对于所有的进程都是可见的，操作系统在初始化过程中首先建立起系统地址空间。因此，我们接下来先从系统地址空间的内存管理开始介绍。为简化起见，在本章的讲解中，我们不考虑 3 GB 进程地址空间选项，以及对 PAE 和大页面（页面大小为 4 MB）的支持。

4.2 Windows 系统内存管理

在 Intel x86 处理器的 Windows 系统中，0x80000000~0xffffffff 是所有进程共享的系统地址空间。在这段地址空间中，其布局结构是在内核初始化阶段完成的。本节将首先介绍系统地址空间的初始化过程，然后描述系统地址空间中的动态内存管理算法。

4.2.1 系统地址空间初始化

在第 2 章中，我们已经介绍过 Windows 的引导过程，在内核获得控制以前，Windows 的加载程序（即 ntldr）已经打开了 Intel x86 处理器的分页机制，并且预先建立了足够的页表以便 16 MB 以下的低地址可以通过页表来访问其物理内存，也就是说，16 MB 以下的虚拟地址将直接映射到相同地址的物理内存上。因此，16 MB 地址以下的代码仍然以透明的方式在保护模式和分页机制下运行。而且，ntldr 在加载内核模块（ntoskrnl.exe）时将

把它映射到特定的虚拟地址上，然后再把控制权交给其主函数 KiSystemStartup。

GDT 的设置是在 ntldr 中完成的，虽然 WRK 中没有这部分代码，但是，通过在调试器中跟踪 WRK 的启动代码，我们可以看到，在 KiSystemStartup 函数获得控制时，段寄存器 CS、DS、ES、SS 和 FS 的值分别如下：

CS：0x8 ——对应二进制表示 1000

SS：0x10 ——对应二进制表示 1 0000

DS、ES：0x23 ——对应二进制表示 10 0011

FS：0x30 ——对应二进制表示 11 0000

而寄存器 gdtr 的值为 0x8003f000。参照图 4.5 关于段选择符格式的描述，我们可以知道，CS 指向 GDT 中索引为 1 的段，SS 指向 GDT 中索引为 2 的段，DS 和 ES 指向 GDT 中索引为 4 的段，FS 指向 GDT 中索引为 6 的段。根据 gdtr 的值，我们检查这些段的段描述符，如表 4.1 所示。CS、DS、ES 和 SS 段指向整个地址空间，从地址 0 一直到 32 位最大地址（差最后一个页面）。FS 指向一个特殊的页面，后面我们还会讲到，此页面包含了当前处理器的控制区（KPCR）信息。正因为如此，我们在系统代码中常常可以看到通过 FS 来获得当前处理器的全局信息，比如当前线程（参考 3.4.2 节）。

表 4.1 Windows 系统中用到的段描述符设置

段	描述符地址	描述符内容（4 个 16 位整数）	段基地址	段最大偏移
CS	0x8003f008	ffff 0000 9b00 00cf	0x00000000	0xfffff000
SS	0x8003f010	ffff 0000 9300 00cf	0x00000000	0xfffff000
DS、ES	0x8003f020	ffff 0000 f300 00cf	0x00000000	0xfffff000
FS	0x8003f030	0001 f000 93df ffc0	0xffdff000	0x00001000

由于 CS、DS、ES 和 SS 段的这种设置方式，相当于段机制被屏蔽了，“段+偏移”形式的逻辑地址直接被映射成线性地址，这种做法也称为地址空间的平面化。因此，在 Windows 中，所有的内存访问都是线性地址空间中的内存地址。段的设置无须特别考虑。

现在回到 KiSystemStartup 函数，它调用 KiInitializeKernel 函数进行内核初始化。KiInitializeKernel 函数在 P0 处理器即引导处理器上，执行系统全局范围的内核初始化。KiInitializeKernel 函数调用 ExpInitializeExecutive，对执行体进行初始化。内存管理器是在执行体中初始化的。

在 ExpInitializeExecutive 函数（base\ntos\init\initos.c 文件的 241~901 行）中，除了执

行体自身被初始化以外，执行体的各个子组件，包括内存管理器，也被初始化。而且，每个子组件都被调用两次初始化，分别对应于阶段 0 初始化和阶段 1 初始化。关于内核的两阶段初始化的完整介绍，请参考 2.6.2 节。图 4.11 显示了与内存管理有关的初始化过程。

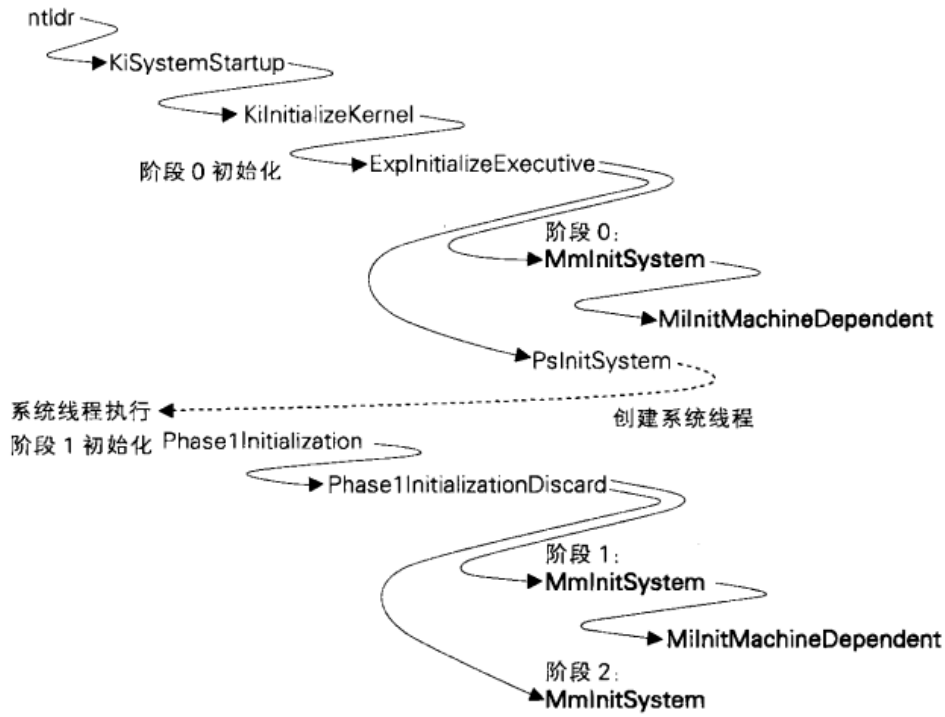


图 4.11 涉及内存管理的系统初始化过程

Windows 加载程序 ntldr 只提供了必要的内存环境，系统空间的主要初始化工作是在 MmInitSystem 函数中完成的，因此，接下来我们看一看 MmInitSystem 函数的代码（base\ntos\mm\mminit.c 文件的 336~2 397 行）。MmInitSystem 函数包括三部分代码逻辑，分别对应于阶段 0、阶段 1 和阶段 2 的初始化，如图 4.11 所示。这里阶段 1 与阶段 2 初始化都是在 Phase1InitializationDiscard 函数中被调用的。我们可以把它看做三个函数的简单组合。

MmInitSystem 函数在阶段 0 所做的初始化工作（434~2 208 行代码）主要是完成数据结构的初始化以及一些全局变量的设置。在 466~468 行，我们看到三个全局变量 MmHighestUserAddress、MmUserProbeAddress 和 MmSystemRangeStart 的设置如下：

```
MmHighestUserAddress = (PVOID)(KSEG0_BASE - 0x10000 - 1);
MmUserProbeAddress = KSEG0_BASE - 0x10000;
MmSystemRangeStart = (PVOID)KSEG0_BASE;
```

这里 KSEG0_BASE 为 0x80000000（见 base\ntos\inc\i386.h 的 1 980 行），所以，用

户地址空间（也称为进程地址空间）最高地址为 0x7ffeffff，而系统地址空间从 0x80000000 开始。接下来我们先看一下宏 MiGetPteAddress：

```
#define MiGetPteAddress(va) ((PMMPTE)(((((ULONG)(va)) >> 12) << 2) + PTE_BASE))
```

这里 PTE_BASE 的值为 0xc0000000（见 base\ntos\inc\i386.h 的 1 972 行），因此，MiGetPteAddress 的含义是，给定一个虚拟地址，计算出其对应的 PTE 的地址，即虚拟地址所在页面的页表项的地址。从该定义也可以看出，所有的页表项都按顺序存放在以 0xc0000000 起始的内存处。再看宏 MiGetPdeAddress：

```
#define MiGetPdeAddress(va)  ((PMMPTE)(((((ULONG)(va)) >> 22) << 2) + PDE_BASE))
```

这里 PDE_BASE 的值为 0xc0300000 或 0xc0600000，取决于 PAE（物理地址扩展）是否打开（见 base\ntos\inc\i386.h 的 1 964 或 1 968 行）。我们不考虑 PAE 的情形，所以，PDE_BASE 的值为 0xc0300000，即页目录项位于 0xc0300000 处。

说明一点，ntldr 在将控制权交给内核以前，已经将内核、HAL 和一些被标记为“引导-启动”的驱动程序映射到了 0x80000000 偏上的位置处（在 Windows Server 2003 SP1 系统中，此位置为 0x80800000）。其中引导-启动的驱动程序稍后将被重定位到高端内存（系统 PTE）区域。

回到 MmInitSystem 函数（见 mminit.c 的 499 行）中，接下来设定系统视图（system view）大小为 16 MB，会话空间（session）的大小为 48 MB（见 base\ntos\mm\mi.h 的 8 261~8 317 行）。会话空间直接位于 0xc0000000 之下的 48 MB，即 0xbd000000~0xbfffffff。从 mminit.c 的 582 行一直到 754 行，计算会话空间和系统视图的内存位置，计算的结果以及对应的一些全局变量如图 4.12 所示。在图中，会话映像文件区包括 win32k.sys、视频驱动程序以及一些打印驱动程序的映像文件；会话内存池是指属于会话空间的换页内存池。

接下来 MmInitSystem 函数初始化系统缓存的位置，MmSystemCacheStart 是个已初始化的全局变量，为 0xc1000000。系统缓存的结束位置是 0xe1000000，宏 MM_SYSTEM_CACHE_END 包含此值。然后是换页内存池的变量设置，MmPagedPoolStart 的初值为 0xe1000000。换页内存池的大小为 MmSizeOfPagedPoolInBytes，默认 32 MB，后面还要对它进行调整。

然后计算系统 PTE 的数量（见 mminit.c 的 1 173~1 287 行）。MmInitSystem 根据系统可用页面的数量，对 MmNumberOfSystemPtes 全局变量进行赋值，最少 7 000 个页面，最多 50 000 个页面。如果要检验驱动程序的话，还需要额外的系统 PTE（见 mminit.c 的 1 282~1 287 行）。

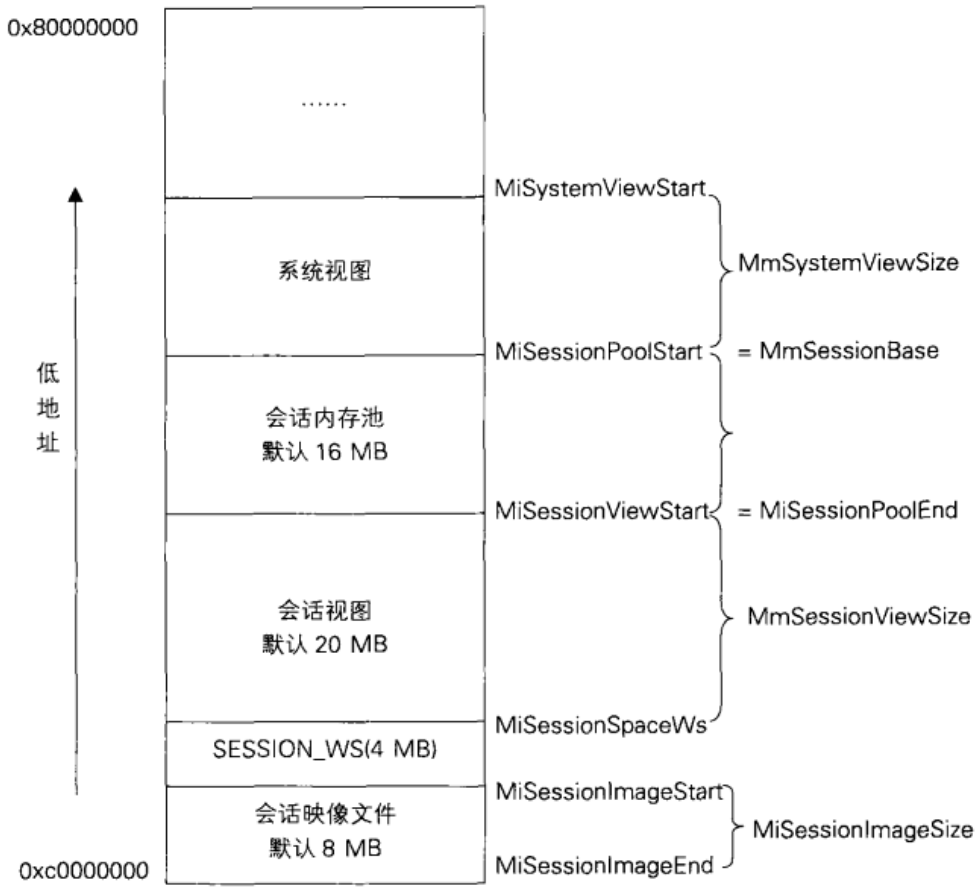


图 4.12 会话空间和系统视图的内存布局结构

接着初始化与堆内存管理有关的全局变量。然后有一个重要的函数调用：MiInitMachineDependent。此函数的主要功能是，真正让 Windows 的虚拟内存运转起来，前面介绍的 MmInitSystem 函数所做的工作只是在划分虚拟地址空间，并没有真正建立页目录项和页表。而 MiInitMachineDependent 真正建立页目录，以及建立页表来映射内核各个区域。特别是，非换页内存池也是在此函数中初始化的。关于该函数，我们稍后详细介绍。

然后 MmInitSystem 函数初始化跟物理内存相关的全局变量。首先调用 MmInitializeMemoryLimits 获得有关物理内存的基本信息。为了方便地管理物理内存，系统使用一个足够大的位图来记录物理页面的有效性。在完成了内存基本初始化以后，MmInitSystem 调用 MiReloadBootLoadedDrivers 函数，将 ntldr 加载的引导-启动驱动程序重定位到系统 PTE 区域，因而它们也能得到页面机制的保护，并有可能被换页。

接下来根据可用物理页面的数量来确定系统内存的规模，基本的规则是，小于 19 MB，则认为是小系统。对于工作站而言，在 19~31 MB 之间，则认为是中等规模系统；大于等于

32 MB，则认为是大系统。对于服务器而言，在 19~63 MB 之间，则认为是中等规模系统；大于等于 64 MB，则认为是大系统。另外，MmInitSystem 也设置一组全局变量（见 mminit.c 的 1 554~1 676 行），包括 MmMaximumDeadKernelStacks、MmModifiedPageMaximum、MmSystemCacheWsMinimum、MmDataClusterSize、MmCodeClusterSize、MmReadClusterSize、MmInPageSupportMinimum、MmFreedExpansionPoolMaximum 等。这些全局变量用于控制系统空间中一些特殊用途的物理内存页面的数量或者上下限。然后，调整可用物理内存页面的数量，即全局变量 MmResidentAvailablePages，从当前可用页面数量中减掉 32 个保留页面（1 717 行），再减掉非换页扩展所需要的页面数（1 725 行，系统启动时物理页面数的 1/6，最多不超过 256 MB），还要减掉系统缓存所占用的最少物理页面数（1 731 行）。

然后 MmInitSystem 函数建立起系统缓存结构（1 761~2 037 行），这其中的大部分代码都在处理多级页表（三级或四级），为简化起见，我们只考虑 Intel x86 的二级页表机制。在 WRK 中，系统缓存结构的地址是 0xc0c00000（全局变量 MmSystemCacheWorkingSetList），而系统缓存的地址是 0xc1000000（全局变量 MmSystemCacheStart），两者之间正好差 4 MB，所以，这两个地址的 PDE 正好是相邻的（见 1 843 行的 ASSERT）。系统缓存的最大可能地址是 0xe1000000，但实际上其真正的结束地址可能会小一些，因而系统缓存的尺寸不会有 0xe1000000~0xc1000000 那么大。读者可以在 MmInitSystem 函数中跟踪 MaximumSystemCacheSize 的变化情况。计算得到 MmSystemCacheEnd 和 MmSizeOfSystemCacheInPages 以后，就可以为这部分空间分配页表了（注意，这里仅仅分配页表而非页面）。最后调用 MiInitializeSystemCache 初始化系统缓存，由它负责初始化系统缓存工作集，并建立起相应的管理数据结构。关于系统缓存空间的内存管理，请参考 7.2.1 节。

至此，MmInitSystem 函数已经基本上定义好系统空间了，并且完成了初步的初始化工作。现在设置全局变量 MmTotalCommitLimit 为一个真正有意义的值，它代表了最多可以提交的内存数量，即最多可以兑现多少物理内存。另外，全局变量 MmTotalCommitLimitMaximum 等于 MmTotalCommitLimit。工作集的高限 MmMaximumWorkingSetSize 为总可用页面数减去 512，当然它不可能超过 2 GB。

然后 MmInitSystem 调用 MiBuildPagedPool 函数，建立起换页内存池。在建立了换页内存池以后，就可以初始化已加载的模块表了（因为需要申请换页内存）。把这些模块映射到系统空间中。这是通过调用 MiInitializeLoadedModuleList 函数（2 099 行）来完成的。

最后，若可用物理内存仍然超过 127 MB，则增加更多的系统 PTE，让非换页区域后

移（2 139~2 176 行）。然后拷贝一份页目录，以便将来大页面系统 PTE 映射被删除时（发生在 MiUnmapLargePages 函数中），还可以恢复原始的页目录项。

在 MmInitSystem 函数的阶段 0 初始化部分，有相当一部分工作是在 MiInitMachineDependent 函数中完成的，包括非换页内存池的初始化工作和 PFN 数据库的初始化。为了理解物理内存的初始分配情况，有必要看一下这个函数，其代码位于 base\ntos\mm\i386\init386.c 的 762~3 568 行。由于此函数的代码较多，这里只介绍与系统空间初始化相关的代码逻辑。并且，为简便起见，在下面解释代码的过程中，将不考虑与 MmVirtualBias 变量非零（即/3GB 引导选项）、64 位版本、PAE 支持和非对称内存模型相关的代码逻辑。

在 854~973 行代码，MiInitMachineDependent 处理大页面支持、变量初始化，以及生成页目录项和页表项的模板。然后设置当前进程的页目录，代码如下：

```
PointerPte = MiGetPdeAddress (PDE_BASE);
PdePageNumber = MI_GET_PAGE_FRAME_FROM_PTE (PointerPte);

CurrentProcess = PsGetCurrentProcess ();

DirBase = MI_GET_PAGE_FRAME_FROM_PTE (PointerPte) << PAGE_SHIFT;

CurrentProcess->Pcb.DirectoryTableBase[0] = DirBase;
KeSweepDcache (FALSE);
```

这里我们不考虑 PAE 情形，KeSweepDcache 是个空语句，什么也不做。宏 MI_GET_PAGE_FRAME_FROM_PTE 直接从 PTE 中提取出它的页帧编号，即物理地址。因此，进程对象的 KPROCESS 结构的 DirectoryTableBase[0]记录了该进程的页目录物理内存地址。接下来 1 020~1 023 行代码把页目录表中对应于 0~2 GB 之间的页目录项写成零，意味着当前进程（即空闲进程）不使用这部分空间。

然后 MiInitMachineDependent 根据 ntldr 传递进来的关于物理内存的描述符链表 LOADER_PARAMETER_BLOCK::MemoryDescriptorListHead 的信息，求出物理内存页面的数量，以及空闲物理内存的最低地址（见 1 030~1 083 行）。这样得到全局变量 MmNumberOfPhysicalPages 和 MmLowestPhysicalPage 的值。接着，设置全局变量 MmDynamicPfn 和 MmHighestPossiblePhysicalPage，即最高可用的物理地址页编号和最高可能的物理页面。然后调整全局变量 MmSizeOfNonPagedPoolInBytes 和 MmMaximumNonPagedPoolInBytes，即非换页内存池的大小和最大尺寸（1 281~1 489 行）。

接下来从 1 499 行代码开始，首先获得用于物理内存管理的辅助颜色值及其掩码，即全局变量 MmSecondaryColors 和 MmSecondaryColorMask。然后计算 PFN 本身需要的非换页内存开销 MxPfnAllocation。由于非换页内存池是从高地址向低地址方向计算的，而非换页池

的结束地址 MmNonPagedPoolStart 等于 0xffbe0000，所以，其起始地址 MmNonPagedPoolStart 为结束地址减去最大可能的地址 MmMaximumNonPagedPoolInBytes，再加上当前大小，即 MmSizeOfNonPagedPoolInBytes（见代码 1 589 行），然后对 MmNonPagedPoolStart 做对齐调整（见 1 628 行）。

在 Windows 系统空间的内存布局中，系统 PTE 区域位于非换页内存池的紧前面，全局变量 MmNonPagedSystemStart 记录了系统 PTE 区域的开始地址，由于系统 PTE 区域和非换页内存池两者都是不可以被换出到外存中的，所以这两部分的页表都需要在初始化的时候建立好。首先计算 MmNonPagedSystemStart 的位置（见 1 730 行），此地址不得低于 MM_LOWEST_NONPAGED_SYSTEM_START，即 0xeb000000（见 1 734 行）。另一方面，如果换页内存池的结束位置越过了系统 PTE 的起始位置，则需要对换页内存池的大小作调整（见 1 744~1 800 行代码）。然后，检查内核映像和 HAL 映像是否可以用大页面映射，如果满足条件，则对它们使用大页面映射（见代码 1 808~2 033 行），记录在 MiLargeVaRanges 全局数组中。

接下来开始分配物理内存，考虑 PFN 数据库和非换页内存池所需要的页面（见 2 048 行代码中的 PagesNeeded 变量）。尽可能地使用大页面映射，如有必要，减小非换页内存池的大小以便能对齐到大页面（4 MB）边界。PFN 数据库的位置（全局变量 MmPfnDatabase）处于已加载的映像区（包括内核、HAL 等）之上，并且对齐到大页面边界。如果从 PFN 数据库开始位置一直到系统视图之间的虚拟地址空间不足以容纳所需页面数量，则相应地减小非换页内存池（见代码 2 080~2 101 行）。2 133~2 225 行代码真正从物理内存描述符结构 MxFreeDescriptor 中，扣除 PFN 数据库和非换页内存池所需要的页面，记录在局部变量 FirstPfnDatabasePage 和 PagesNeeded 中，并且这部分空间对齐到大页面边界，加入到大页面映射数组 MiLargeVaRanges 中。记录非换页内存池起始地址的全局变量 MmNonPagedPoolStart 也随之更新为紧跟在 PFN 数据库之后（见代码 2 235 行）。2 284~2 307 行代码是当 PFN 数据库和非换页内存池不使用大页面映射时非换页内存池的分配逻辑。

接下来 2 320~2 337 行代码真正分配系统 PTE 和扩展的非换页内存池地址范围的页表。代码如下：

```
StartPde = MiGetPdeAddress (MmNonPagedSystemStart);
EndPde = MiGetPdeAddress ((PVOID)((PCHAR)MmNonPagedPoolEnd - 1));

while (StartPde <= EndPde) {

   ASSERT (StartPde->u.Hard.Valid == 0);

   //
   // Map in a page table page, using the
```

```
    // slush descriptor if one exists.
    //

    TempPde.u.Hard.PageFrameNumber = MxGetNextPage (1, TRUE);
    *StartPde = TempPde;
    PointerPte = MiGetVirtualAddressMappedByPte (StartPde);
    RtlZeroMemory (PointerPte, PAGE_SIZE);
    StartPde += 1;
}
```

在这段代码中，首先确定从 MmNonPagedSystemStart 开始到 MmNonPagedPoolEnd – 1 的地址范围所对应的起始和结束 PDE：StartPde 和 EndPde，然后分别填充其中的每一个页目录项。MxGetNextPage 分配一个物理页，返回其页帧号，TempPde 中的 PageFrameNumber 填好以后，即被赋给 StartPde 所对应的 PDE。MiGetVirtualAddressMappedByPte 函数（实际上是一个 C 语言宏）的功能是，根据一个 PTE 项的地址（这里把 StartPde 当做一个 PTE 来看待），计算出对应页表的虚拟地址，也就是说，获得刚刚分配的页表的虚拟地址，从而可以初始化此页表（填充零）。MiGetVirtualAddressMappedByPte 的逻辑很简单，左移 10 位，但是，这里实际上用到了一个设计技巧，下面的插入文本解释了这一机制。

Windows 的页目录自映射方案

由于 Windows 的分页机制已经起作用，处理器的内存访问指令只认虚拟地址，不认物理地址，所以，为了访问刚刚分配的页表，即例子代码中 MxGetNextPage 返回的那个页面，必须要经过 PDE 和 PTE 的查询，但是，我们现在只有物理页帧编号（PFN），尚不知道哪个虚拟地址可以访问此页面。Intel x86 的内存访问要经过两次查表，参见图 4.3。并且，第二次查表的结果恰好指向目标页面。现在我们来看 Windows 的页目录和页表结构如何使这一逆过程也非常高效和便捷。

Windows 的页目录地址(即 PDE 的基地址，在代码中为宏 PDE_BASE)是 0xc0300000，PTE 的起始地址是 0xc0000000（在代码中为宏 PTE_BASE），并且 Intel x86 上的 PDE 和 PTE 的结构可以认为是相同的（见 4.4.1 节的描述）。为了映射一个页表并将其中的内容清零，首先需要在页目录中增加一个 PDE，即赋值语句“*StartPde = TempPde;”所做的事情，其次，要能够以虚拟地址方式来访问此页表。实际上，对于页目录中的 PDE 本身的访问也需要经过两次查表，即赋值语句中的“*StartPde”其实已经在访问页目录页面的内容了。

PDE 的查表是根据 CR3 寄存器和虚拟地址高 10 位来确定的，因此，一旦 CR3 寄存器已经置值，则第一次查表便已确定。关键是第二次查表。Windows 做了一个巧妙的设计，让一个页表的地址范围（4 MB 大小）具有某种结构，使得对页表页面的访问可直接利用上

一级对它引用的虚拟地址来导出，即根据 PDE 或 PTE 的虚拟地址来导出它所指的物理页面的虚拟地址（MiGetVirtualAddressMappedByPte 或 MiGetVirtualAddressMappedByPde）。

方案是这样的：假设页表的 PDE 的虚拟地址已知（即例子代码中的 StartPde），为了构造页表页面的虚拟地址，我们这样设计两次查表：第一次，查找页目录本身的页面；第二次，查找刚刚增加的那个 PDE（实际上，这次要把 PDE 当做 PTE 来看待）。基于这一思路，我们得到这样的虚拟地址：最高 10 位与页目录虚拟地址的高 10 位相同，任何一个 PDE 都满足此条件，接下来的 10 位应该是 StartPde 与页目录基地址之间的差除以 4，即两者之间的 PDE 数量。由于页目录地址 0xc0300000 最高 10 位与紧接着的 10 位都是 1100000000b，所以，StartPde 左移 10 位恰好满足刚才提到的两个条件。这正是 MiGetVirtualAddressMappedByPte 如此简单的原因，并且 PDE_BASE 取值 0xc0300000 也使得任何一个 PDE 左移 10 位以后其对应的 PDE 仍然不变。图 4.13 显示了页目录与页表之间的关系，注意，页目录中的 1100000000b 项（十进制 768）指向页目录本身。图中分别演示了 StartPde 作为一个 PDE 以及地址 p 作为一个页表页面的地址转译过程。读者可以自行验证，0xc0300c00 中的页帧编号与 CR3 寄存器中的页帧编号是相同的。实际上，从这里我们也可以看出，一旦确定了 0xc0300c00 作为页目录地址以后，整个虚拟地址空间的所有页表自然位于 0xc0000000~0xc0400000 之间。

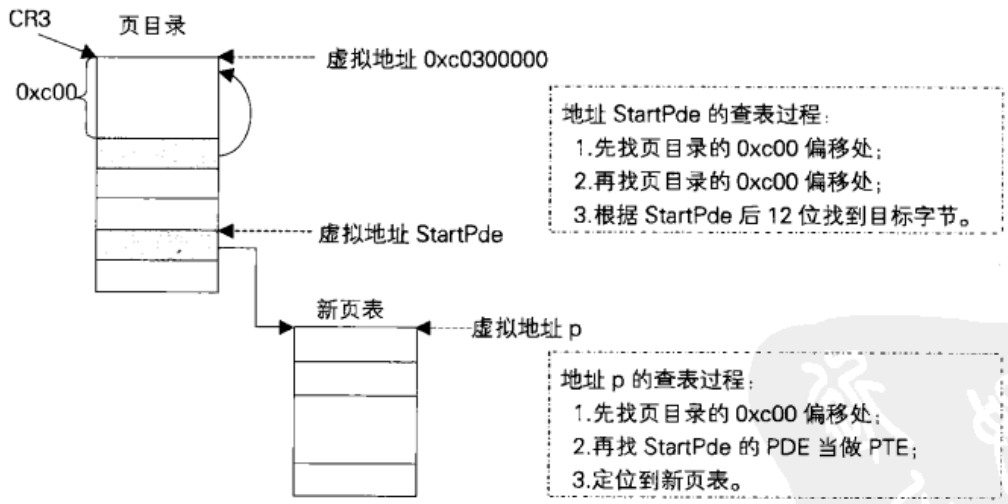


图 4.13 Windows 中页表虚拟地址定位示意图

这种方案的关键之处在于，页目录的 0xc00 项指向它自己，并且 PDE 和 PTE 的格式是兼容的，所以，页目录中的每一个 PDE 在用于访问页表页面时也被当做 PTE 来映射。因此，这种方案被称为页目录自映射方案。

进一步可以推广，在多级页表结构中，从任何一级 PTE 都可以利用同样的左移位方法获得其所指的物理页面的虚拟地址。例如，在图 4.13 中，对于页表中的任何一个 PTE，它所指的物理页面的虚拟地址都同样可以用 MiGetVirtualAddressMappedByPte 来实现。请读者自行验证。此外，MiGetVirtualAddressMappedByPde 的实现更加直截了当，将 PDE 左移 20 位代表了它所映射的虚拟地址起始位置，大小为 4 MB。

接下来的代码 2 341~2 484 行分配 PFN 数据库和非换页内存池，一直到系统视图开始处这段地址范围的页表，如果在非换页内存池与系统视图之间有空隙（至少容得下一个 PDE，即 4 MB 大小），则这段空隙用做额外的系统缓存和系统 PTE（参见 7.2.1 节）。此外，如果 1 GB~2 GB 范围也被用做系统空间，则为指定的范围，从 MiUseMaximumSystemSpace 到 MiUseMaximumSystemSpaceEnd，分配相应的页表。注意，这两个全局变量在 MmInitSystem 函数中已经根据系统的引导参数而赋值了。

Windows 在系统地址空间高端处保留了一块地址范围用做系统崩溃时的信息转储区域，其起始地址为 0xffbe0000，2 497~2 509 行代码为这块内存（不超过 4 MB）分配了一个页表。

接下来从 2 515 一直到 2 564 行代码，为非换页内存池分配页面，这些页面前面已经预留了，并且页帧编号是连续的。然后在 2 579 行调用 MiInitializeNonPagedPool 函数建立起非换页内存池的管理结构，在 2 581 行调用 MiInitializeNonPagedPoolThresholds 函数设定非换页内存池的高、低阈值。

然后分配 PFN 数据库所需要的物理页面。在使用大页面的情况下，这些页面前面已经预留了，从 FirstPfnDatabasePage 开始，总共 MxPfnAllocation 个页面，见代码 2 590~2 624 行。在不使用大页面的情况下，所需页面尚未预留，需要保证为每一个物理页面对应一个 MMPFN 结构（其定义见 base\ntos\mm\mi.h 中，共 24 个字节长）。PFN 数据库实际上是一个位于内核映像之后的 MMPFN 数组，所以，这部分代码为此数组分配物理页面，见代码 2 628~2 728 行。

接下来 2 752~2 783 行代码，为按颜色的空闲页面链表数组 MmFreePagesByColor 本身（并非链表）分配物理页面并初始化其内部链表结构。这里 MmFreePagesByColor 数组用于管理物理页面，参见 4.5.3 节。

2 790~2 900 行代码为所有已设置为有效（valid）的页面，更新一下对应的 PFN 数据库元素的状态。这也算是 PFN 数据库的初始化，这样，物理页面的分配便跟 PFN 数据库中的状态对应起来了。

接下来检查最低的物理内存页面，看它是否为 0，如果为 0 并且当前尚未被使用，则将它标记为已经在使用。这样做是为了查找“物理页面被指定为零”的软件错误（bug），见代码 2 908~2 928 行。

从 2 964 行开始，根据系统加载程序传递过来的物理内存描述符链表，检查所有的内存描述符，并把可用的内存都加入到 PFN 数据库的空闲链表中。由于加载程序传递过来的链表是从低地址到高地址排序的，而这里的循环检查按相反的顺序，因而高地址的可用物理内存先插入到空闲链表中，低地址的内存后插入到链表中。这种做法的考虑是，系统刚开始时分配的内存不太可能被释放，因而低地址的物理内存（比如最低的 16 MB）可以为某些特殊的设备或驱动程序使用。

接下来扫描 PFN 数据库所用到的 PTE，并把它们对应的页面标记为已经在使用，即在数据库中将引用计数加 1。代码见 3 105~3 239 行。

然后调用 InitializePool 函数初始化非换页内存池，见代码 3 263 行。至此，非换页内存池已经建立起来，可供使用了。

接下来初始化系统 PTE 池，这些 PTE 主要用于映射 I/O 空间、驱动程序的映像，以及内核栈。系统 PTE 池的范围从高端的非换页区域（由全局变量 MmNonPagedSystemStart 指定），一直到高端的非换页内存池（0xffbe0000 前面的那一段非换页内存池）。这段地址范围为系统 PTE 区域。在代码 3 295 行调用 MiInitializeSystemPtes 函数对系统 PTE 池进行初始化。

如果前面在分配虚拟地址空间时已经分配了额外的系统 PTE 区域，则将额外的区域加入到系统 PTE 池中。见代码 3 302~3 346 行。

最后，初始化当前进程的内存管理结构，并建立起工作集链表，这涉及创建一个 PDE 来映射超空间（位于 0xc0400000），并初始化该 PDE 所指的页表（见代码 3 359~3 379 行）。超空间建立起来以后，初始化相应的变量，包括超空间中管理 PTE 的全局变量，以及进程的工作集链表（见代码 3 385~3 390 行）。然后为零页面线程保留用于零化操作的 PTE（见代码 3 396 行）。接着为进程创建 VAD 位图（见代码 3 412~3 430 行），关于进程地址空间的 VAD 位图，请参考 4.3.2 节。之后，设置页目录的 PFN 属性（见代码 3 475~3 480 行），以及为工作集链表分配一个页面（见代码 3 500~3 514 行），然后，设置工作集的属性（见代码 3 524~3 528 行）。最后，把管理内存所需要的物理页面标记为已在使用（见代码 3 534~3 565 行）。

至此，MiInitMachineDependent 函数完成其初始化工作，并返回。

经过了阶段 0 初始化以后，根据 MmInitSystem 和 MiInitMachineDependent 两个函数所做的事情，系统地址空间的初始化已经完成。其布局结构已完全建立起来，而且，非换页内存区域已经分配好页表和页面，完整的内存结构如图 4.14 所示。有关的全局变量如表 4.2 所列，表中也给出了在 Windows Server 2003 SP1 一个典型系统配置（512 MB 内存）下这些变量的值。

图 4.14 系统地址空间的内存布局结构

表 4.2 Windows 系统中用于系统空间管理的一些全局变量

全局变量名	典型取值	全局变量名	典型取值
MmHighestUserAddress	0x7ffeffff	MiSessionImageStart	0xbf800000
MmUserProbeAddress	0x7fff0000	MiSessionImageEnd	0xc0000000
MmSystemRangeStart	0x80000000	MmSystemPteBase	0xc0000000
MmPfnDatabase	0x81000000	MmWorkingSetList	0xc0502000
MmNonPagedPoolStart	0x81301000	MmHyperSpaceEnd	0xc0bfffff
MmNonPagedPoolEnd0	0x82000000	MmSystemCacheWorkingSetList	0xc0c00000
MiSystemCacheStartExtra	0x82000000	MmSystemCacheStart	0xc1000000
MiMaximumSystemCacheSizeExtra	0x2f800	MmPagedPoolStart	0xe1000000
MiSystemViewStart	0xbb000000	MmPagedPoolEnd	0xf0bfffff
MmSessionBase	0xbc000000	MmNonPagedSystemStart	0xf0c00000
MiSessionPoolStart	0xbc000000	MmNonPagedPoolExpansionStart	0xf8ba0000
MiSessionViewStart	0xbc400000	MmNonPagedPoolEnd	0xffbe0000
MiSessionSpaceWs	0xbf400000	MmNumberOfPhysicalPages	0x0001ff7a

现在我们来看 Windows 系统的阶段 1 初始化过程，其中内存管理的初始化也是在 MmInitSystem 函数中完成的，见 base\ntos\mm\mminit.c 文件中的 2 211~2 389 行代码。MmInitSystem 再次调用 MiInitMachineDependent 函数，不过，这次 MiInitMachineDependent 函数只做了很简单的一点事情（见 base\ntos\mm\i386\init386.c 文件中的 863~870 行代码），即，如果处理器支持大页面，并且在前面初始化过程中有记录下来要转换成大页面的内存区域（在全局数组 MiLargeVaRanges 中），则把这些区域转换成大页面映射，其中可能包括的区域为内核本身的映像区（包括内核和 HAL）、PFN 数据库，以及非换页内存池。

然后，MmInitSystem 调用 MiMapBBTMemory 函数，此函数的用意是为 BBT（Basic Block Tools，一种代码优化技术）预留一块内存缓冲区。由于在默认配置下，全局变量 BBTPagesToReserve 为 0，所以，此函数实际上什么也不做，直接返回。接下来 MmInitSystem 调用 MiSectionInitialization 函数创建内存区对象类型，创建段解引用系统线程（dereferencing segment thread），以及创建内存区对象“\\Device\\PhysicalMemory”，再插入到当前进程（System 进程）句柄表中。

接下来 MmInitSystem 从换页内存池申请一个 PTE 对象，由全局变量 MmSharedUserDataPte 指向此 PTE 对象。该 PTE 记录了用户共享数据页的 PTE，对应的页面在内核和执行体初始化以前已分配好，位于 0xffdf0000 处。进一步，把该 PTE 赋到

进程地址空间的 0x7ffe0000 地址所对应的 PTE 中，这样就实现了在 0xffdf0000 和 0x7ffe0000 这两个地址映射到同一个物理页面上。因而，进程地址空间和系统地址空间共享同一个页面，从而用户模式代码可以访问系统中的一些状态数据（参考 KUSER_SHARED_DATA 数据结构）。参见代码 2 236~2 289 行。

然后 MmInitSystem 调用 MiSessionWideInitializeAddresses、MiInitializeSessionWsSupport 和 MiInitializeSessionIds 函数初始化会话空间。接着，MmInitSystem 创建一个系统线程，称为修改页面写出器（modified page writer），见代码 2 299~2 310 行。然后，初始化系统内存事件，包括高内存和低内存事件，以及换页内存池和非换页内存池的高低事件。这是由 MmInitSystem 调用 MiInitializeMemoryEvents 函数来完成的，见 2 317 行代码。

接下来 MmInitSystem 启动平衡集管理器，这是两个系统线程，分别为 KeBalanceSetManager 和 KeSwapProcessOrStack 函数。前者定期检查并管理进程的工作集；后者控制进程和内核栈的换入和换出。本章 4.5 和 4.6 节将进一步介绍这两个线程。

然后 MmInitSystem 调用 MiStartZeroPageWorkers 函数，启动零化页面的系统辅助线程（它们仅负责在初始化阶段零化页面）。接下来是一个循环，对所有已加载的模块，调用 MiWriteProtectSystemImage 函数以设置保护属性。见代码 2 362~2 382 行。

至此，阶段 1 初始化完成，全局变量 MiFullyInitialized 置成 1。

在阶段 2 初始化过程中，MmInitSystem 仅仅调用 MiEnablePagingTheExecutive 函数（其代码位于 base\ntos\mm\mminit.c 中 4 247~4 664 行）。此函数使当前已加载的模块中的可换页代码区变成可换页的，它定位到每个模块的 PAGE 内存区，然后调用 MiEnablePagingOfDriverAtInit 函数（其代码在 base\ntos\mm\mminit.c 中 4 667~4 800 行）以改变相应 PTE 中的标记。原来的 PTE 是由系统加载程序 ntldr 设置的，当时内存管理器尚未建立起来，而现在系统空间已经完成初始化，所以，作为内存管理器初始化过程的最后一步，MmInitSystem 对系统空间中可以换页的代码区，修改其 PTE 使之能在内存紧缺的情况下腾出物理内存页面。

到这里，内存管理器的初始化工作全部完成，而且系统空间已经完全建立起来。此后执行的系统代码可以充分利用系统提供的内存管理功能，尤其是非换页内存池和换页内存池都已经可以正常使用了。

从以上的代码分析可以看出，Windows 的系统地址空间 0x80000000~0xffffffff 是经过精心安排的，一组全局变量规定了各个区域的范围。阶段 0 初始化的主要职责是划分系统

地址空间，它充分考虑到了物理内存的数量和引导选项指定的要求，使得最终得到的各部分区域的范围相对比较合理；阶段 0 也负责建立起页目录和页表结构，并且完成非换页区域的物理页面分配。阶段 1 初始化主要集中在一些内存管理任务的初始化，包括创建修改页面写出器和平衡集管理器，以及页面零化任务等。阶段 2 初始化只是简单地使系统模块中的换页代码区可被换页而已。

4.2.2 系统地址空间内存管理

在这一节中，我们将讨论系统地址空间中的内存是如何动态管理的。在系统地址空间中，有些部分是供一些特殊模块使用的，比如会话空间是由会话管理器和 Windows 子系统使用的；而换页内存池和非换页内存池则是提供给系统内核模块和设备驱动程序使用的。在换页内存池中分配的内存有可能在物理内存紧缺的情况下被换出到外存中；而非换页内存池中分配的内存总是处于物理内存中。为了实现这两种内存池，Windows 使用了两层内存管理。下层是基于页面的内存管理，仅限于执行体内部使用；上层建立在下层的内存管理功能基础之上，对外提供各种粒度的内存服务。两层的结构如图 4.15 所示（这里不考虑会话空间的内存管理）。为描述方便起见，下层的内存池称为系统内存池，上层的内存池称为执行体内存池。本节我们将首先讨论系统非换页内存池和换页内存池的管理算法和实现方法，然后介绍执行体内存池的算法和实现原理。下一节再介绍系统 PTE 区域的管理算法和实现。

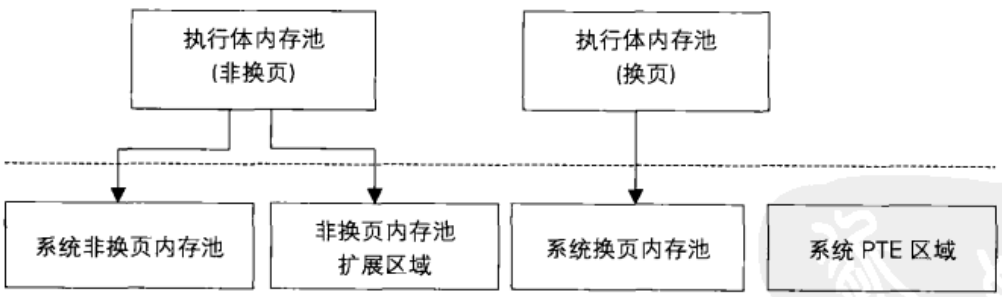


图 4.15 Windows 内核中的动态内存管理结构

（一） 系统非换页内存池的管理算法

在上一节解读 MiInitMachineDependent 函数时，我们曾经提到，该函数调用 MiInitializeNonPagedPool 和 MiInitializeNonPagedPoolThresholds 函数来初始化非换页内存池的管理信息。MiInitializeNonPagedPool 函数位于 base\ntos\mm\allocpag.c 文件中，见代码 3 399~3 599 行。它的主要任务是初始化用于管理内存池的一些全局变量，尤其是用于

存放空闲页面链表头的 MmNonPagedPoolFreeListHead 数组。空闲页面链表中的每一项都是一个 MMFREE_POOL_ENTRY 结构，其定义如下（见 base\ntos\mm\mi.h 头文件）：

```
typedef struct _MMFREE_POOL_ENTRY {
   LIST_ENTRY List;        // maintained free&chk, 1st entry only
   PFN_NUMBER Size;        // maintained free&chk, 1st entry only
   ULONG Signature;        // maintained chk only, all entries
   struct _MMFREE_POOL_ENTRY *Owner; // maintained free&chk, all entries
} MMFREE_POOL_ENTRY, *PMMFREE_POOL_ENTRY;
```

在初始化状态下，整个非换页池空间，即从 MmNonPagedPoolStart 开始，一共 MmSizeOfNonPagedPoolInBytes 大小的内存，所有的页面均加入第一个页面所在的 MMFREE_POOL_ENTRY 项。MmNonPagedPoolFreeListHead 数组在默认情况下包括 4 项：第一个数组项包含所有单个空闲页面的 MMFREE_POOL_ENTRY 项，第二个数组项包含所有 2 个空闲页面的 MMFREE_POOL_ENTRY 项，第三个数组项包含所有 3 个空闲页面的 MMFREE_POOL_ENTRY 项，第四个数组项包含所有大于等于 4 个空闲页面的 MMFREE_POOL_ENTRY 项。在初始情况下，只有一个 MMFREE_POOL_ENTRY 项加入到第四个数组项链表中。

然后，MiInitializeNonPagedPool 函数确定非换页内存池的起始和结束物理页面帧 MiStartOfInitialPoolFrame 和 MiEndOfInitialPoolFrame，最后，为非换页内存池扩展区建立起系统 PTE。

一旦非换页内存池的结构已建立，接下来系统代码可以通过 MiAllocatePoolPages 和 MiFreePoolPages 函数来申请和归还页面。这两个函数也位于 base\ntos\mm\allocpag.c 文件中，见代码 1 188~3 317 行。其中非换页内存的页面申请逻辑位于代码行 1 262~1 828，页面回收的代码逻辑位于 2 507~3 093 行。这两个函数的其余部分基本上在处理换页内存池的页面申请和回收，其中也包括一部分代码在处理会话空间中的内存池。

非换页内存池有两部分：地址范围 MmNonPagedPoolStart 与 MmNonPagedPoolEnd0 之间的部分是基本内存池，其中所有的页面都已经分配了物理页面；地址范围 MmNonPagedPoolExpansionStart 与 MmNonPagedPoolEnd 之间的部分是扩展区，只有当基本区无法满足内存申请需求时才会分配真正的物理页面。由于非换页内存池的空闲页面已经被映射到物理页面，所以，Windows 充分利用这些页面自身的内存来构建起一组空闲页面链表，换句话说，每个页面都是一个 MMFREE_POOL_ENTRY 结构，如图 4.16 所示。MiInitializeNonPagedPool 函数已经把数组 MmNonPagedPoolFreeListHead 初始化成只包含一个完整的空闲内存块，该内存块包括所有的非换页页面。

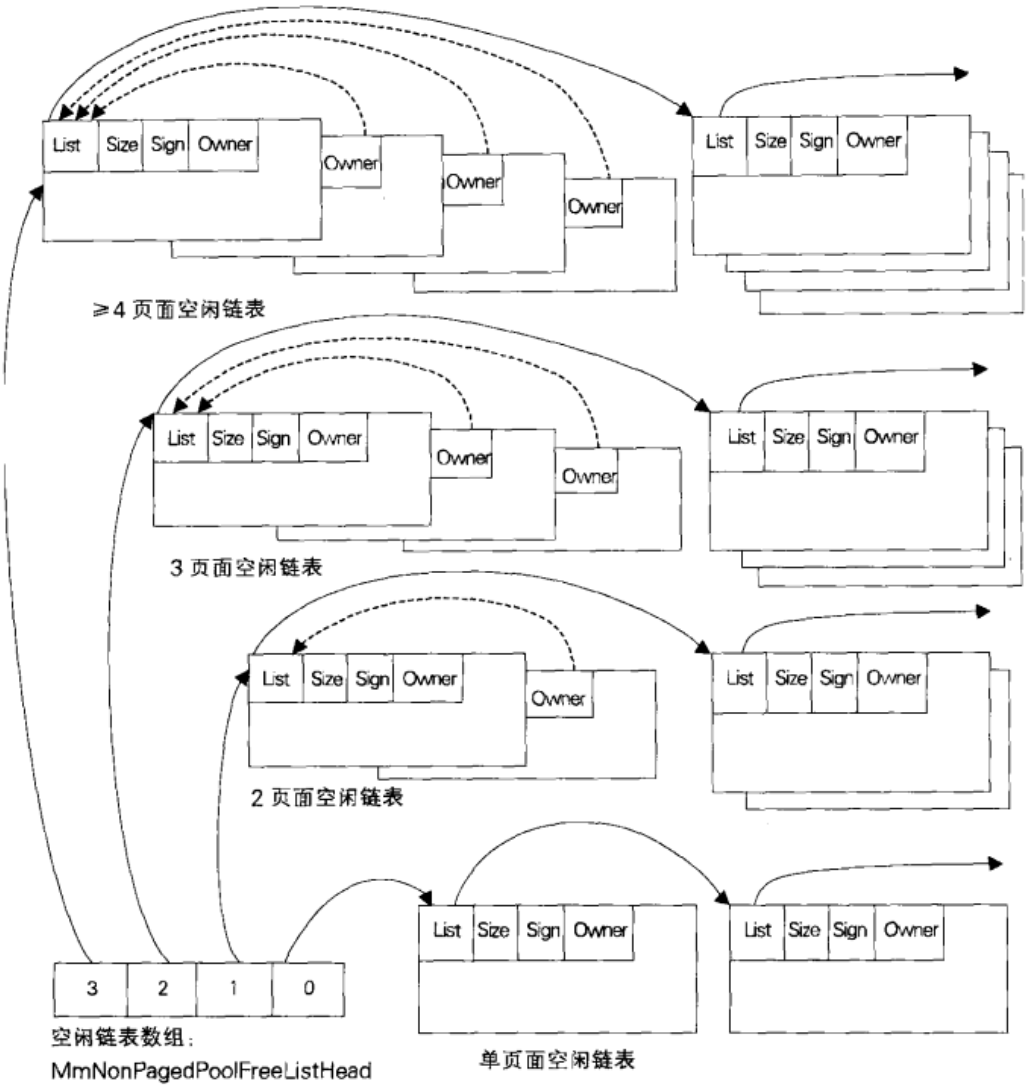


图 4.16 非换页内存池的管理结构

如前所述，在非换页内存池的结构中，每个空闲链表中的每个节点包含 1、2、3、4 或 4 个以上的页面，在同一个节点上的页面其虚拟地址空间是连续的。第一个页面的 List 域构成了链表结构，Size 域指明了这个节点所包含的页面数量，Owner 域指向自己；后续页面的 List 和 Size 域没有使用，但 Owner 域很重要，它指向第一个页面。

当 MiAllocatePoolPages 函数从非换页内存池中分配页面时，它根据指定的大小值，换算成对应的页面数，即局部变量 SizeInPages，然后从 MmNonPagedPoolFreeListHead 数组中选择适当的空闲链表开始搜索，直至找到一个节点拥有这么多的页面，将该节点从链表中移除出来。如果该节点除了满足客户所要求的页面数 SizeInPages 以外，还有剩余的

页面，那么，该节点后端部分的 SizeInPages 个页面将作为结果返回给客户，前端部分剩余的那些页面被当做一个新的空闲链表节点加入到适当的空闲链表中。这些页面的首页面中的 Size 域需要调整，后续的页面皆指向首页面，无须任何改变。

对于已经确定的 SizeInPages 个页面，MiAllocatePoolPages 函数分别找到其起始页面和结束页面在 PFN 数据库中的记录，并设置它们的 StartOfAllocation 和 EndOfAllocation 位，因此，一次有效的页面分配也会在 PFN 数据库中留下痕迹。关于 PFN 数据库的定义和管理，请参考 4.5.1 节的介绍。

如果在空闲链表数组中搜索不到满足条件的空闲节点，则 MiAllocatePoolPages 函数试图扩展非换页内存池，并满足所要求的页面数。

非换页内存池中的页面回收是通过 MiFreePoolPages 函数来完成的，对于一个给定的起始地址 StartingAddress，找到它的 PTE、并提取出页帧编号，从而定位到 PFN 数据库中起始页面的 PFN 项。凡是通过 MiAllocatePoolPages 函数分配得到的连续页面，其首页面的 PFN 项的 StartOfAllocation 位必定是 1，沿着此 PFN 项进行搜索，可以找到结束页面，其 PFN 项的 EndOfAllocation 位为 1。为了维护非换页内存池中空闲链表的结构，被回收的页面如果有可能的话，还需要跟它们后续的空闲页面或者它们前面的空闲页面合并成一个更大的空闲内存块，并且维护好首页面或中间页面的 MMFREE_POOL_ENTRY 结构中的域。如果 StartingAddress 指示了一个非换页扩展区中的地址，则可能有必要调用 MiFreeNonPagedPool 函数来收回它的页面。

在 MiAllocatePoolPages 和 MiFreePoolPages 函数的代码中，当申请或释放单个页面时，我们可以看到 MiNonPagedPoolSListHead 链表的用法。这基本上算是一种优化，在收回单个页面的时候，如果 MiFreePoolPages 函数发现，MiNonPagedPoolSListHead 链表中的页面数（即 Depth 域）小于 MiNonPagedPoolSListMaximum（全局变量，被初始赋值为 4），则将此页面加入到 MiNonPagedPoolSListHead 链表中，参见 base\ntos\mm\allocpag.c 的 2 575~2 582 行代码。因此，在 MiAllocatePoolPages 的开始处（见 allocpag.c 文件的 1 264~1 286 行代码），如果客户要申请单个页面，并且 MiNonPagedPoolSListHead 链表不为空，则直接从链表头提取一个页面返回给客户。所以，MiNonPagedPoolSListHead 链表相当于非换页内存池的一个单页面缓存区。频繁地申请和回收单页面，只相当于在单链表中的插入和移除动作而已，从而避免了在 MmNonPagedPoolFreeListHead 数组的单页面链表中相对昂贵的分配和回收操作。

（二）系统换页内存池的管理算法

接下来看系统换页内存池的管理算法。显然，非换页内存池的空闲页面链表的做法不

适合于换页内存池，因为换页内存池中的空闲页面并不保证有对应的物理页面，而且，仅仅出于管理的原因而为换页内存池中的空闲页面分配物理页面，并不符合换页内存池的设计意图。换页内存池有两个，一个是系统全局范围的，另一个是会话空间中的。它们在系统地址空间中的起始地址分别对应于全局变量 MmPagedPoolStart 和 MiSessionPoolStart，参见上一节中的介绍。换页内存池有一个数据结构来描述其页面分配状态，定义如下：

```
typedef struct _MM_PAGED_POOL_INFO {
   PRTL_BITMAP PagedPoolAllocationMap;
   PRTL_BITMAP EndOfPagedPoolBitmap;
   PMMPTE FirstPteForPagedPool;
   PMMPTE LastPteForPagedPool;
   PMMPTE NextPdeForPagedPoolExpansion;
   ULONG PagedPoolHint;
   SIZE_T PagedPoolCommit;
   SIZE_T AllocatedPagedPool;
} MM_PAGED_POOL_INFO, *PMM_PAGED_POOL_INFO;
```

其中，系统全局换页内存池由全局变量 MmPagedPoolInfo 来定义；会话换页内存池则通过会话空间中的成员变量来定义，即全局变量 MmSessionSpace 中的 PagedPoolInfo 成员。

换页内存池中的页面是通过位图来管理的，一个页面的分配与否，是由该页面所对应的位来表达的。在 MM_PAGED_POOL_INFO 结构中，PagedPoolAllocationMap 域是基本的位图，用于指明每一个页面的分配状态；EndOfPagedPoolBitmap 域也是一个位图，标明了每个页面是否为一次内存申请的最后一个页面。FirstPteForPagedPool 和 LastPteForPagedPool 域规定了内存池的地址范围。NextPdeForPagedPoolExpansion 域定义了换页内存池的下次扩展位置。PagedPoolCommit 域记录了换页内存池中有多少个页面已经分到了内存（已提交，未必对应有物理页面），而 AllocatedPagedPool 域记录了换页内存池已分配了多少页面。PagedPoolHint 域指示了在分配页面时的起始搜索位置。

上一节曾经提到，MmInitSystem 调用 MiBuildPagedPool 函数来建立起系统的换页内存池。MiBuildPagedPool 函数（见 base\ntos\mm\mminit.c 中 3 300~3 786 行代码）所做的事情，本质上来讲是填充全局变量 MmPagedPoolInfo 中的各个成员，完成系统换页内存池的初始化。

全局变量 MmSizeOfPagedPoolInBytes 和 MmSizeOfPagedPoolInPages 记录了系统换页内存池的大小，最大不超过系统地址空间中划给换页内存池的范围，即 MmNonPagedSystemStart-MmPagedPoolStart，参见图 4.14。MmPagedPoolStart 和 MmPagedPoolEnd 分别记录了换页池的起始地址和结束地址。因此，MmPagedPoolInfo 结构中的 FirstPteForPagedPool 和 LastPteForPagedPool 域可根据 MmPagedPoolStart 和 MmPagedPoolEnd 的值直接导出，见 mminit.c 文件的 3 592~3 594 行代码。MiBuildPagedPool

函数分配好换页内存池地址范围的第一个页表，即起始地址的页目录项（PDE）所指的页表，MmPagedPoolInfo 的 NextPdeForPagedPoolExpansion 域值为接下来的页表地址，即换页内存池地址范围的第二个页表的虚拟地址。然后调用 MiCreateBitMap 函数创建一个位图，位图的大小等于页面的数量，位图本身的内存是从非换页内存池中分配的。由于第一个页表中的 PTE 是有效的，所以位图中最前面的 1 024 位（一个页表所能容纳 PTE 的数量）清零，其余的位被置位，这表明前 1 024 个页面是空闲的。这样，MmPagedPoolInfo 的 PagedPoolAllocationMap 域已经初始化；结束页面位图，即 EndOfPagedPoolBitmap 成员，也依次创建，所有的位全清零，因为尚未任何有效的页面分配。

然后 MiBuildPagedPool 调用 InitializePool 函数来初始化执行体换页内存池，参见本节后文关于执行体内存池的介绍。最后，MiBuildPagedPool 函数设置预警阈值 MiLowPagedPoolThreshold 和 MiHighPagedPoolThreshold，以便在适当时候通知系统换页内存池的使用情况。

在 MiBuildPagedPool 函数的代码中，我们可以看到，如果系统禁止执行体内存池换页，即全局变量 MmDisablePagingExecutive 中的 MM_PAGED_POOL_LOCKED_DOWN 标志已置上，则换页内存池实际上变成了非换页内存池，在这种情况下，内存池所需要的页面都必须在该函数中分配好。

介绍了 MiBuildPagedPool 函数的逻辑以后，我们来看换页内存池的页面分配和回收。如同非换页内存池一样，这两项基本任务也是由 MiAllocatePoolPages 和 MiFreePoolPages 函数来完成的；但是，在这两个函数中，针对换页池和非换页池的实现代码基本上是完全分开的。MiAllocatePoolPages 函数的 PoolType 参数指定了在哪个内存池中申请页面；而 MiFreePoolPages 函数则直接根据待回收页面的起始地址来确定内存池的类型。

我们先来解释 MiAllocatePoolPages 函数中的页面分配过程，其代码逻辑从 base\ntos\mm\allocpag.c 文件的 1 834 行开始。首先根据 PoolType 参数找到内存池 MM_PAGED_POOL_INFO 结构，置于局部变量 PagedPoolInfo 中。只有两种可能，要么是系统换页内存池，即全局变量 MmPagedPoolInfo，要么是会话空间的换页内存池，即 MmSessionSpace->PagedPoolInfo。由于换页内存池采用位图来管理页面的分配与否，所以，MiAllocatePoolPages 函数只是简单地调用辅助函数 RtlFindClearBitsAndSet，在 PagedPoolInfo->PagedPoolAllocationMap 位图中搜索指定数量的连续零位，即换页内存池中对应的连续空闲页面。如果未能找到这么多连续零位，则设法扩展换页内存池，即，使内存池的 NextPdeForPagedPoolExpansion 成员往后移，当然不能越过换页内存池结束地址的 PDE。所谓扩展一个换页内存池，是指为它后面的内存区分配页表，并非分配真正的

物理页面。换页内存池经过扩展以后，再调用 RtlFindClearBitsAndSet 函数搜索满足要求的连续零位。

非换页内存池的页面分配，其起始和结束页面直接被标记在 PFN 数据库的对应 PFN 项中，对于换页内存池，这种做法行不通，因为换页内存池中的页面可能会被换出到外存中，所以根本就不存在对应的 PFN 项。因此，为了记录一次页面分配的起始和结束位置，MM_PAGED_POOL_INFO 包含一个并行的位图，专门记录每次页面分配的结束页面。结合这样两个位图，当回收页面时，只需给出起始地址，就可以验证该地址的正确性，如果此地址前面的那个页面尚未被分配，则此地址可以认为是一次页面分配的起始处；如果前面的那个页面已被分配，而且结束页面位图并未表明它是一次页面分配的结束页面，则参数 StartingAddress 中给出的起始地址是错误的，碰到这种情况，系统会崩溃（bugcheck）。

接下来，除了在结束页面位图中设置好相应的位（见代码 2 269 行）以外，MiAllocatePoolPages 函数还要更新换页内存池结构 PagedPoolInfo 中的已分配页面数量，对应于 MM_PAGED_POOL_INFO 结构的 AllocatedPagedPool 域。如果内存池的空闲页面数量小于特定阈值（全局变量 MiLowPagedPoolThreshold），则发出预警信号（系统全局事件 MiLowPagedPoolEvent）。然后检查这些页面是否需要刷新它们的快表（TLB）项，若是，则通过 KeFlushSingleTb、KeFlushMultipleTb 或 KeFlushEntireTb 函数刷新其快表项。最后，正确地填好已分配页面的 PTE，返回这组连续页面的起始地址。

系统换页内存池的页面回收过程非常直截了当，在 MiFreePoolPages 函数中，如果传进来的起始地址 StartingAddress 介于 MmPagedPoolStart 和 MmPagedPoolEnd 之间，则说明这是系统换页内存池中的页面。页面回收逻辑从 allocpag.c 的 3 100 行开始，首先利用 PagedPoolInfo 中的两个位图来验证起始地址的有效性。通过验证以后，MiFreePoolPages 修改这两个位图，使得这一段内存保持未分配的状态，同时也维护好 PagedPoolInfo 中的 AllocatedPagedPool 和 PagedPoolCommit 域。

关于 MiFreePoolPages 函数回收系统换页内存池中的页面，有几点需要说明：

(1) 如果换页内存池被配置成不可换页的（non-pageable），则只需维护位图和相应的全局变量与内存池变量即可，见代码 3 182~3 230 行。

(2) 如果是单个页面被释放，则有可能加入到一个单链表中，见代码 3 170~3 175 行。此单链表是由全局变量 MiPagedPoolSListHead 来记录的。链表中最多存放 8 个单页面。因此，如果客户下次申请单个页面，则 MiAllocatePoolPages 函数直接从该链表中提取页面即可，参见代码 1 862~1 870 行。

(3) 系统换页内存池中的内存也遵从换页空间（即系统页面文件中的空间）的管理，

因为一旦内存紧缺，则这些页面需要被换到外存中，从而占用外存空间。这是换页内存池与非换页内存池的一个关键区别。我们可以看到，在 MiAllocatePoolPages 函数中调用了 MiChargeCommitmentCantExpand 函数；而在 MiFreePoolPages 函数中，则调用了 MiReturnCommitment（实际上，这是一个 C 语言宏）。而且，在 MiFreePoolPages 函数中，它还调用了 MiDeleteSystemPageableVm 函数，以删除可换页的系统地址范围。

关于换页内存池使用的两个位图，这里有必要说明一下位的含义。对于 PagedPoolAllocationMap 中的位，置位表明所对应的页面是不可用的，零位表明是可用的。系统换页内存池初始时，第一个 PDE 所指的页表已分配，所以该页表中的 PTE 所对应的页面（1 024 个）已经可以使用了。因而，在位图中，前 1 024 位为零位，其余皆为置位状态。随着换页内存池被扩展，即 NextPdeForPagedPoolExpansion 不断后移，零位出现的位置也开始后移。但是，NextPdeForPagedPoolExpansion 后面对应的位一定是置位的。因此，对于 PagedPoolAllocationMap 中的位，零位表示空闲页面，处于待分配状态；置位表示可能已被分配，或者尚未扩展到该页面。若不考虑 PDE 扩展的因素，概括起来可以这样说，MiAllocatePoolPages 做的事情是把连续的零位变成置位；而 MiFreePoolPages 函数做的事情是把连续的置位变成零位。EndOfPagedPoolBitmap 位图初始时，所有的位皆为零位。每次分配页面时，最后一个分配的页面所对应的位被置位；MiFreePoolPages 函数依据此位图来判断传进来的起始地址的合法性，并且把从该地址开始的已分配连续页面中的最后一个页面所对应的位清零。

（三） 执行体内存池的管理算法

上面介绍的换页内存池和非换页内存池是 Windows 操作系统提供的基本动态内存管理手段，它以页面为基本粒度来管理系统中划定的地址范围。显然，页面粒度对于一般的代码逻辑而言太大了，比如，第 2 章介绍的对象管理器往往在更小的粒度上工作。所以，为了适应各种内核组件对于内存管理的需要，Windows 内核在系统的非换页内存池和换页内存池的基础上，实现了灵活的、可适应各种大小内存需求的内存池，这便是执行体内存池，其实现代码位于 base\ntos\ex\pool.c 文件中。

执行体内存池对象是由数据结构 POOL_DESCRIPTOR 来描述的，以下是它的定义，位于 base\ntos\inc\pool.h 文件中。

```
typedef struct _POOL_DESCRIPTOR {
    POOL_TYPE PoolType;
    ULONG PoolIndex;
    ULONG RunningAllocs;
    ULONG RunningDeAllocs;
    ULONG TotalPages;
```

```
    ULONG TotalBigPages;
    ULONG Threshold;
    PVOID LockAddress;
    PVOID PendingFrees;
    LONG PendingFreeDepth;
    SIZE_T TotalBytes;
    SIZE_T Spare0;
    LIST_ENTRY ListHeads[POOL_LIST_HEADS];
} POOL_DESCRIPTOR, *PPOOL_DESCRIPTOR;
```

此外，有一组全局变量与执行体内存池有关，其定义和声明分别位于 pool.c 和 pool.h 中，本章我们仅着眼于内存管理，而忽略与同步有关的逻辑，所以下面列出的全局变量中不包含与同步有关的对象：

```
#if defined(NT_UP)
#define NUMBER_OF_PAGED_POOLS 2
#else
#define NUMBER_OF_PAGED_POOLS 4
#endif

ULONG ExpNumberOfPagedPools = NUMBER_OF_PAGED_POOLS;

ULONG ExpNumberOfNonPagedPools = 1;

POOL_DESCRIPTOR NonPagedPoolDescriptor;

#define EXP_MAXIMUM_POOL_NODES 16

PPOOL_DESCRIPTOR ExpNonPagedPoolDescriptor[EXP_MAXIMUM_POOL_NODES];

#define NUMBER_OF_POOLS 2
PPOOL_DESCRIPTOR PoolVector[NUMBER_OF_POOLS];

PPOOL_DESCRIPTOR ExpPagedPoolDescriptor[EXP_MAXIMUM_POOL_NODES + 1];

volatile ULONG ExpPoolIndex = 1;
```

根据以上定义的全局变量以及它们的赋值情况，我们可以知道，ExpNumberOfNonPagedPools 代表了非换页内存池的数量，只有一个；而换页内存池的数量，即变量 ExpNumberOfPagedPools，在单处理器系统上是三个，在多处理器系统上有五个（注意，数组的元素个数是 EXP_MAXIMUM_POOL_NODES+1）。PoolVector 是一个包含两个数组的元素，第一个元素 PoolVector[0]指向非换页内存池，即 NonPagedPoolDescriptor 的地址；第二个元素 PoolVector[1]指向换页内存池，即指向第一个换页内存池。数组 ExpNonPagedPoolDescriptor 仅用于有多个非换页内存池的情形；数组 ExpPagedPoolDescriptor 中的每个元素分别指向一个换页内存池。

执行体内存池对象的初始化也是按换页内存池和非换页内存池分开进行的。MiInitMachineDependent 函数在执行了系统非换页内存池的初始化（即调用

MiInitializeNonPagedPool 函数）并完成了 PFN 数据库的初始化以后，还调用了 InitializePool(NonPagedPool, 0)，这正是初始化执行体非换页内存池对象。另外，在 MiBuildPagedPool 函数中，当系统换页内存池的初始化基本完成以后，该函数直接调用 InitializePool(PagedPool, 0L)，以完成执行体换页内存池的初始化。

InitializePool 函数由两个相对独立的部分构成，前面部分执行非换页内存池的初始化，后面部分执行换页内存池的初始化。它所做的事情是，初始化以上这些全局变量，并且为每个执行体内存池对象调用 ExInitializePoolDescriptor 函数，填充 POOL_DESCRIPTOR 结构中的域。从 InitializePool 函数中可以看到，执行体的非换页内存池的 POOL_DESCRIPTOR 对象是全局变量 NonPagedPoolDescriptor，而执行体的换页内存池的 POOL_DESCRIPTOR 对象则是从执行体的非换页内存池中分配的。这是一个合理的内存分配顺序。当 InitializePool 函数被调用来初始化执行体换页内存池时，执行体的非换页内存池已经被初始化，因而 InitializePool 可以向它申请内存了。

InitializePool 和 ExInitializePoolDescriptor 函数的细节可以参看 base\ntos\ex\pool.c 文件中的 893~1 352、659~756 行，这里不再进一步解释。在 POOL_DESCRIPTOR 结构中，PoolType 域为枚举类型 POOL_TYPE 的 NonPagedPool 或 PagedPool，分别代表执行体非换页内存池和换页内存池。尽管在 POOL_TYPE 中还有其他的内存池类型，这里我们只考虑 NonPagedPool 和 PagedPool 两种类型。PoolIndex 域是指当前对象在该类型的内存池数组中的索引，对于换页内存池，PoolIndex 是它在 ExpPagedPoolDescriptor 数组中的下标值。ListHeads 数组是一个关键数据成员，执行体内存池对于小内存的分配和回收，是通过一组空闲内存块链表来实现的。接下来我们通过解释 ExAllocatePoolWithTag/ExFreePoolWithTag 函数来介绍执行体内存池的管理算法。

在 Windows NT 的早期，执行体内存池的管理是通过伙伴系统算法（参考 4.1.3 节）来实现的。在 WRK 的执行体内存池实现代码中，我们可以看到，它采用的并非伙伴系统算法，而是用一组称为快查表（lookaside list）的空闲内存块链表分别记录了各种大小的待分配内存。当一个内存池需要更多内存时，它可以向对应的系统内存池申请更多的页面。而当内存释放时，则让待释放的内存与相邻的内存块尽可能地合并，以形成更大的空闲内存块，如果构成一个完整的页面，则将页面交还给系统内存池。对于需要一个页面或以上的内存申请，执行体内存池直接向系统内存池申请足够的页面并交给客户；对于小于一个页面大小的内存申请，执行体内存池使用快查表来管理小粒度的内存块。

现在我们来看 ExAllocatePoolWithTag 函数，其代码位于 base\ntos\ex\pool.c 的 1 738~2 677 行。ExAllocatePoolWithTag 函数带三个参数：PoolType 参数指明了要从哪种内存池

中分配内存，NumberOfBytes 表示要申请多少字节的内存，Tag 是一个 32 位整数。根据 PoolType 以及前面列出的 PoolVector 全局变量，ExAllocatePoolWithTag 可以确定目标内存池对象，即局部变量 PoolDesc 的初始值，见代码 1 885~1 890 行。

POOL_DESCRIPTOR 对象中维护了一组快查表，即它的 ListHeads 成员。这些快查表包含了 8 字节倍数大小的空闲内存块链表。通过执行体内存池分配的内存块有固定大小的头，即它们的管理开销，下面是一些相关的定义：

```
typedef struct _POOL_HEADER {
   union {
      struct {
         USHORT PreviousSize : 9;
         USHORT PoolIndex : 7;
         USHORT BlockSize : 9;
         USHORT PoolType : 7;
      };
      ULONG Ulong1;
   };

   union {
      ULONG PoolTag;
      struct {
         USHORT AllocatorBackTraceIndex;
         USHORT PoolTagHash;
      };
   };
} POOL_HEADER, *PPOOL_HEADER;

#define POOL_PAGE_SIZE  0x1000
#define POOL_BLOCK_SHIFT 3

#define POOL_OVERHEAD ((LONG)sizeof(POOL_HEADER))

#define POOL_FREE_BLOCK_OVERHEAD  (POOL_OVERHEAD + sizeof (LIST_ENTRY))

typedef struct _POOL_BLOCK {
   UCHAR Fill[1 << POOL_BLOCK_SHIFT];
} POOL_BLOCK, *PPOOL_BLOCK;

#define POOL_SMALLEST_BLOCK (sizeof(POOL_BLOCK))

#define POOL_BUDDY_MAX  \
  (POOL_PAGE_SIZE - (POOL_OVERHEAD + POOL_SMALLEST_BLOCK ))

#define POOL_LIST_HEADS (POOL_PAGE_SIZE / (1 << POOL_BLOCK_SHIFT))
```

因此，每一次内存分配还必须把 POOL_HEADER 部分也算上。执行体内存池的基本管理算法是这样的：如果一次内存分配的大小超过了 POOL_BUDDY_MAX，则执行体内存池直接调用底层的系统内存池来申请足够的页面，并直接交给客户，在这种情况下，客户拿到的内存地址是页面对齐的，也就是说，内存块地址的最后 12 位为 0；否则，

ExAllocatePoolWithTag 函数检查内存池内部的快查表，以期找到合适大小的空闲内存块，若未能找到满足条件的内存块，则向系统内存池申请一个新的页面，把页面的一部分返回给客户，剩下的加入到适当大小的快查表中。

最小的内存块是 8 个字节，POOL_HEADER 的大小也是 8 个字节，所以，POOL_BUDDY_MAX 的值是 4 096-16=4 080，而 POOL_LIST_HEADS 的值是 512。POOL_DESCRIPTOR 中的快查表包含 512 项（实际使用 510 项），分别对应于 8、16、24……4 072 和 4 080 大小的空闲内存块。若客户申请大于 4 080 而小于 4 096 的内存块，则直接使用整个页面，因为剩下的空间不足以容纳最小内存块（8 字节）加上管理开销（8 字节）。图 4.17 显示了执行体内存池的管理结构。

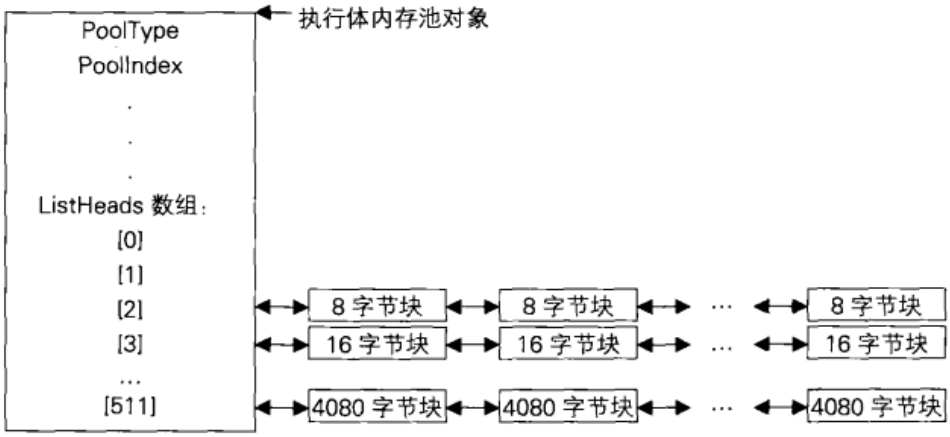


图 4.17 执行体内存池的管理结构

POOL_HEADER 中的第一个 C 联合成员中的各个域的含义如下：PreviousSize 记录了当前内存块前面的那个内存块的大小，依据此域可以找到同一个页面中前面的内存块；BlockSize 记录了当前内存块的大小，依据此域可以找到同一个页面中后面的内存块；PoolIndex 说明了当前内存块属于哪个执行体内存池；PoolType 成员记录了当前内存块所在的内存池的类型，对于空闲内存块，PoolType 成员为 0。这里 PreviousSize 和 BlockSize 都是指实际大小除以 8 之后得到的值，并非原值本身。POOL_HEADER 的 PoolTag 域记录了当前被分配内存块的一个标记。图 4.18 显示了一个受执行体内存池管理的页面的内部结构。

我们回到 ExAllocatePoolWithTag 函数的代码上。如果 NumberOfBytes 参数大于 POOL_BUDDY_MAX，则直接调用系统内存池的内存分配函数，即 MiAllocatePoolPages 函数，来完成内存分配，参见 pool.c 文件的 1 907~2 014 行代码。否则需要使用内存池对象中的快查表。首先根据 NumberOfBytes 参数计算得到快查表在 ListHeads 数组中的索引，即 ListNumber 局部变量，见代码 2 042 行。

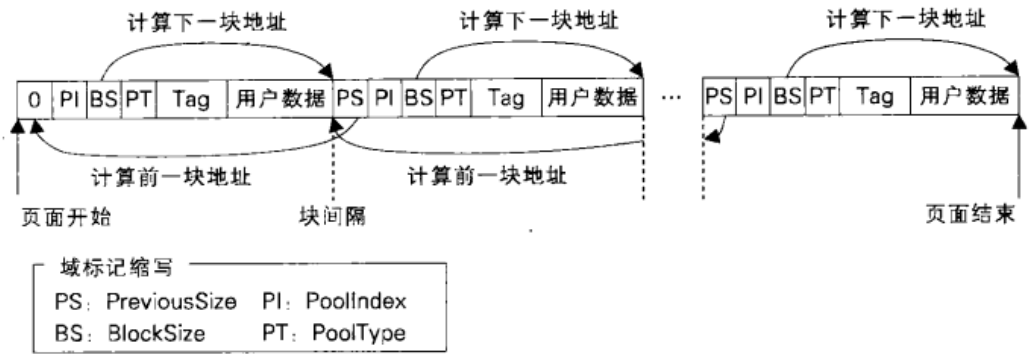


图 4.18 在执行体内存池中的页面的内部结构

接下来确定要在哪个内存池对象中分配内存，即确定 POOL_DESCRIPTOR 对象，见 pool.c 文件的 2 046~2 353 行代码。对于换页池，在 1 至 ExpNumberOfPagedPools 之间找到一个尚未被锁住的换页池，从这里也可以看出，Windows 使用多个换页内存池可以增加并发性，允许多个线程甚至多个处理器同时使用不同的换页内存池。对于非换页池，只有一个，则直接使用全局的非换页内存池，此种情形下，局部变量 PoolDesc 在 1 886 行已经被赋以正确的值。

确定了 POOL_DESCRIPTOR 对象 PoolDesc 和快查表数组的下标 ListNumber 以后，接下来就可以搜索满足条件的空闲内存块。2 368~2 523 行之间的 do 循环正是完成这一任务。从 ListNumber 指定的快查表开始，顺次检查链表头：只要当前链表不空，就能找到满足条件的内存块；如果当前链表为空，则继续找大一号的链表。若找到的内存块大于所要求的内存大小，则把它分成两块，一块作为结果返回给客户，另一块加到适当大小的快查表中。POOL_HEADER 结构的 PreviousSize 如果为 0，则说明它是页面的开始处，在这种情况下，取前面部分作为结果内存块（记录在 Entry 变量中），后面部分稍后要归还到快查表中（记录在 SplitEntry 变量中）。如果 POOL_HEADER 结构的 PreviousSize 不为 0，则说明这不是页面的起始内存块，在这种情况下取后面部分作为结果内存块返回给客户（记录在 Entry 变量中），前面部分稍后要归还到快查表中（记录在 SplitEntry 变量中）。经过如此切分以后，如果 SplitEntry 内存块的大小足够容纳一个 POOL_HEADER 加上双链表的两个指针域（实际上应理解为 POOL_SMALLEST_BLOCK），则将 SplitEntry 插入到适当的快查表中。在 do 循环体中，不管是否要切分一个内存块，Entry 总是指向要返回的内存块，设置其中的 PoolTag，并维护好内存池 POOL_DESCRIPTOR 中的相关成员，擦除内存块中用于快查表的链表指针信息，最后返回内存块中的有效数据区地址。

如果在 do 循环完成以后，仍然未能找到有效的空闲内存块，则说明内存池中没有大于等于所要求大小的内存块，在这种情况下，有必要向系统内存池申请一个新的页面，并

且把新页面的前面部分作为结果内存块返回给客户，后面部分构造一个新的空闲内存块插入到适当的快查表中。申请新页面是通过调用MiAllocatePoolPages函数来完成的，见pool.c文件的 2 532 行代码；在新页面中构造内存块结构并切分此内存块的代码见 2 587~2 602 行；最后，把切分后剩下的空闲内存块加入到快查表中并返回结果内存块的代码见 2 628~2 676 行，此部分代码的逻辑与 do 循环中的代码类似。

现在转到 ExFreePoolWithTag 函数，该函数位于 base\ntos\ex\pool.c 文件的 4 296~5 024 行。它的第一个参数 P 指定了要释放的内存块地址，第二个参数 TagToFree 指定了被释放的内存块的标记。代码行 4 363~4 503 是一些检查和日志记录逻辑。代码行 4 510~4 600 针对那些直接通过系统内存池申请到的内存块的释放，即在 ExAllocatePoolWithTag 函数中针对大于 POOL_BUDDY_MAX 的内存请求而直接调用 MiAllocatePoolPages 函数得到的内存块。对应地，内存的释放通过 MiFreePoolPages 函数来完成。

接下来，ExFreePoolWithTag 函数根据参数 P，定位到相应的 POOL_HEADER，置于局部变量 Entry 中，并检查其中的成员信息，进一步定位到它后面的内存块的 POOL_HEADER，检查后面内存块的 PreviousSize 是否与当前内存块的 BlockSize 相等，这算是对地址 P 的有效性检查。所以，如果 ExFreePoolWithTag 函数接到一个不正确的地址 P，除非真的是特意构造的，否则它可以有很大的概率能检测到系统中内存分配与释放的不一致性。经过有效性检查以后，ExFreePoolWithTag 函数根据内存块中的 PoolType 和 PoolIndex 信息，找到内存池描述符对象，放在局部变量 PoolDesc 中。

然后，ExFreePoolWithTag 函数进行系统处理，其核心的思想与我们在系统换页内存池和非换页内存池中看到的相仿。对于小内存块的释放，每个处理器有两个缓存链表，位于 KPRCB 结构的 PPPagedLookasideList 和 PPNPagedLookasideList 成员中，分别对应于换页内存池和非换页内存池的缓存链表。阈值 POOL_SMALL_LISTS 指明了只有小于等于 32×8 大小的内存块才使用缓存链表。因此，在 ExAllocatePoolWithTag 函数中，当申请一个小内存块时，如果在当前处理器的缓存链表中能找到对应大小的内存块，则直接从缓存链表中摘除一个内存块，快速地返回给客户。对应地，在 ExFreePoolWithTag 函数中，如果当前处理器的缓存链表尚未达到预定的深度，则将待释放的内存块插入到相应的缓存链表中。Pool.c 文件中的 4 725~4 844 代码行正是在做这样的工作。ExAllocatePoolWithTag 函数中也有相应的逻辑，这里不再讨论。

接下来，从 4 880 行代码开始，ExFreePoolWithTag 函数考虑将当前内存块与后面的空闲内存块或者前面的空闲内存块合并在一起，构成更大的空闲内存块，从而减少内存池中的小碎片。4 900~4 929 行代码完成与后面的空闲内存块的合并，4 935~4 965 行代码完

成与前面的空闲内存块的合并。当然，合并的前提条件是，当前待释放的内存块前面或后面的内存块恰好也是空闲的，这可以依据内存块的 POOL_HEADER 中的 PoolType 的值是否为 0 来确定。如果合并以后的内存块是一个完整的页面，则将空闲页面归还给系统内存池，这也是通过调用 MiFreePoolPages 函数来完成的，见代码 4 972~4 982 行；否则，将新的内存块（可能已经合并了其相邻的内存块）插入到适当的快查表中，见代码 4 989~5 022 行。至此，ExFreePoolWithTag 函数完成了参数 P 所指内存块的回收工作。

在 base\ntos\ex\pool.c 文件中，我们还可以看到其他一些分配和释放内存的函数，就本质上讲，执行体内存池是通过 ExAllocatePoolWithTag 和 ExFreePoolWithTag 来工作的，而且它们也是在其他内核模块，甚至设备驱动程序中常常被调用的内存管理函数。ExFreePoolWithTag 函数并没有检查内存块的标记，即 TagToFree 没有实际的用途。内存管理是与系统模块的稳定性息息相关的事项，Windows 除了提供高效的、低碎片的内存管理功能以外，还提供了其他一些用于检测设备驱动程序内存错误的设施，其中包括特殊内存池和内存池跟踪机制。我们在 ExAllocatePoolWithTag 和 ExFreePoolWithTag 函数中可以看到这两种机制的身影。当全局的池标记变量 ExpPoolFlags 中的特殊内存池标记（EX_SPECIAL_POOL_ENABLED）打开时，内存分配可能会转到 MmAllocateSpecialPool 函数中。

特殊内存池机制对于每次内存分配，都需要额外的内存，基本的原理是，用无效页面将内存块分隔开，因此，一旦代码引用到内存块的前面或者后面，将会导致一个内核模式的访问违例，从而使系统崩溃，允许捕捉到有错误的设备驱动程序。另外，特殊内存池机制也允许在驱动程序分配或者释放内存时执行一些额外的检查。

内存池跟踪机制是指根据内存申请和释放时内存块的 Tag，来记录相同 Tag 的内存使用情况。我们在 ExAllocatePoolWithTag 函数中，看到它调用了 ExpInsertPoolTrackerInline 函数，而在 ExFreePoolWithTag 函数中，它调用了 ExpRemovePoolTrackerInline 函数。Windows 的内存管理器维护了一张 Tag 表，其中记录了每种 Tag 的内存使用情况。因此，如果系统的内存池跟踪功能打开的话，当一个驱动程序卸载时，内存管理器可以检查它申请的内存是否已经全部释放，如果还没有全部释放，它可以让系统崩溃，并指明该驱动程序有内存错误。ExpInsertPoolTrackerInline 和 ExpRemovePoolTrackerInline 的代码也在 base\ntos\ex\pool.c 文件中，请读者自行查看，这里不再解释。

4.2.3 系统 PTE 区域的管理

上一节介绍了系统内存池和执行体内存池的内存管理机制，在 Windows 的整个地址空间中，除了会话空间部分以外，还有一部分内存区也是以动态内存的方式来管理的，这就

是系统 PTE 区域，其起始地址由全局变量 MmNonPagedSystemStart 指定，而结束地址正好在 MmNonPagedPoolExpansionStart 之前。另外，在系统视图和非换页内存池之间若有空隙，则这段地址范围的全部或一部分（1/6）也用作系统 PTE 区域，称为系统 PTE 额外区，如图 4.14 中所示；而前者以 MmNonPagedSystemStart 开始的区域称为系统 PTE 基本区。Windows 将这两部分区域链接在一起进行管理，因此，在本节下文中，我们不区分系统 PTE 区域和额外区，而统称为系统 PTE 区域。

系统 PTE 区域是以 PTE 的形式来管理的，当内核代码需要一段虚拟地址来映射物理页面时，它可以使用系统 PTE 区域中的地址范围；当内核代码需要额外的非换页区域时，它也可以使用系统 PTE 区域来映射物理内存。有一点请注意，这部分虚拟地址范围称为系统 PTE 区域，并非表明这段地址范围存放的是 PTE，而表示这段地址范围是以 PTE 的形式来管理的，即把 PTE 当做资源来管理。这一节我们来讨论系统 PTE 区域的管理算法和实现。另外，非换页内存池的扩展区域也是通过 PTE 来管理的，所以，我们在代码中会看到有两部分区域，这是通过 C 枚举类型 SystemPtePoolType 来区分的。在本节中，我们只关心 SystemPteSpace 类型的 PTE 区域。

系统 PTE 区域中页面的 PTE 也同样位于系统地址空间的页表区域，即从 0xc0000000 开始的区域，只不过，这些 PTE 未必按照硬件所定义的 PTE 格式来解释。对于空闲页面的 PTE，它们并没有对应的物理页面，其 PTE 被按照以下的数据结构来解释：

```
typedef struct _MMPTE_LIST {
   ULONG Valid : 1;
   ULONG OneEntry : 1;
   ULONG filler0 : 8;
   ULONG Prototype : 1;          // MUST BE ZERO
   ULONG filler1 : 1;
   ULONG NextEntry : 20;
} MMPTE_LIST;
```

从 MMPTE_LIST 结构的定义可以看出，硬件 PTE 中的 PFN 域（见 4.4.1 节）被定义成了 NextEntry，因此，可以想见，PTE 是按照链表的方式来管理的。因为 PTE 对应的页面地址一定是页面对齐的，即最后 12 位总是 0，所以，这里只用 20 位就能表达系统 PTE 区域中的一个页面地址。在接下来的描述中，我们把通过 PTE 建立起来的链表称为 PTE 链表。

在系统 PTE 区域的内存管理中，除了 PTE 链表以外，还有 PTE 簇的概念，即连续地址的页面构成一个簇。PTE 链表只把簇头的 PTE 连接起来，簇的大小是这样来规定的：如果一个簇只有一个页面，则它的 OneEntry 位域为 1；如果有多个页面，则第二个页面 PTE 中的 NextEntry 项包含了页面数量，也即 PTE 的数量。图 4.19 显示了 PTE 链表和 PTE 簇。其中浅灰色 PTE 表示不在空闲链表中，说明已经被保留了。每个 PTE 的第 1 位为 OneEntry 位，高 20 位为 NextEntry。图中显示的 PTE 链表的前三个节点分别包含 2 个、1

个和 5 个 PTE，所以，第二个 PTE 簇的 OneEntry 位被置 1。最后一个 PTE 簇包含 3 个 PTE，它的 NextEntry 为-1，即 0xfffff。

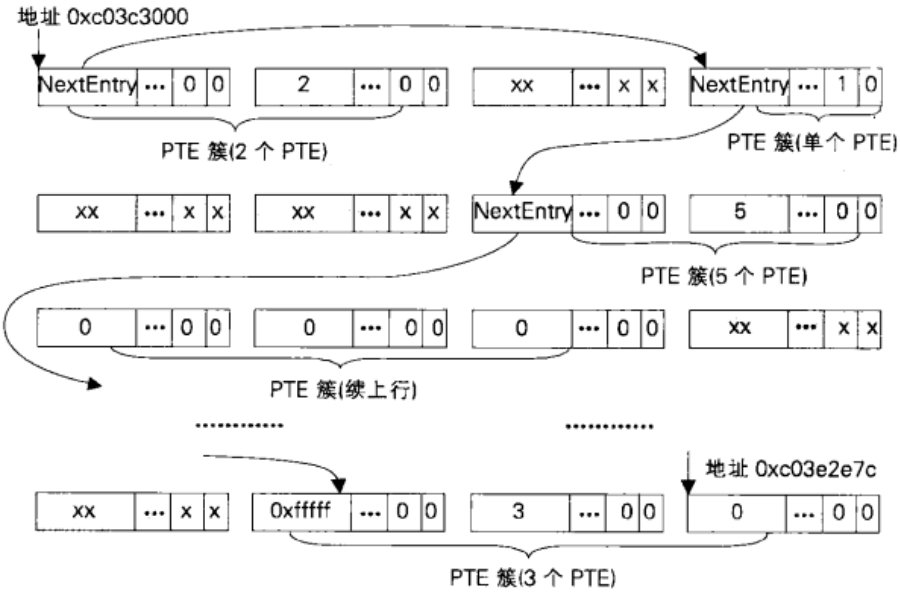


图 4.19 PTE 链表和 PTE 簇示意图

接下来看系统 PTE 区域的 PTE 管理算法。以下是一组全局变量定义：

```
#define MM_SYS_PTE_TABLES_MAX 5
#define MM_PTE_TABLE_LIMIT 16
ULONG MmSysPteIndex[MM_SYS_PTE_TABLES_MAX] = {1,2,4,8,MM_PTE_TABLE_LIMIT};
UCHAR MmSysPteTables[MM_PTE_TABLE_LIMIT+1] = {0,0,1,2,2,3,3,3,3,4,4,4,4,4,4,4,4};

ULONG MmSysPteMinimumFree [MM_SYS_PTE_TABLES_MAX] = {100,50,30,20,20};

PVOID MiSystemPteNBHead[MM_SYS_PTE_TABLES_MAX];
ULONG MmSysPteListBySizeCount [MM_SYS_PTE_TABLES_MAX];

ULONG MmTotalFreeSystemPtes[MaximumPtePoolTypes];
PMMPTE MmSystemPtesStart[MaximumPtePoolTypes];
PMMPTE MmSystemPtesEnd[MaximumPtePoolTypes];
MMPTE MmFirstFreeSystemPte[MaximumPtePoolTypes];
```

数组 MmSysPteIndex 定义了 5 种大小的连续页面内存块：1、2、4、8 或 16，数组 MmSysPteTables 则是数组 MmSysPteIndex 的索引（即下标值）。当需要从系统 PTE 区域中分配给定数量的页面时，比如 NumberOfPtes（≤16），使用 NumberOfPtes 作为 MmSysPteTables 的下标就能直接得到最小的且大于等于 NumberOfPtes 的 MmSysPteIndex 元素的下标。因此，MmSysPteTables 提供了快速查表就能得到这 5 种大小之中最合适的下标值的能力。

数组 MiSystemPteNBHead 是一个重要的数据结构，它为每一种大小的连续页面内存

块定义了一个队列。当客户需要申请一个内存块，即一段连续页面的地址范围时，内存管理器直接从对应的队列中取一块交给客户，若队列中内存块的数量减少了，降到了 MmSysPteMinimumFree 数组定义的阈值以下时，系统再往队列中加一些相同大小的内存块。数组 MmSysPteListBySizeCount 记录了每种大小的内存块的数量。数组 MmTotalFreeSystemPtes、MmSystemPtesStart、MmSystemPtesEnd 和 MmFirstFreeSystemPte 分别记录了系统 PTE 区域（和非换页内存池扩展区域）中空闲 PTE 的数量、起始 PTE、结束 PTE 和第一个空闲 PTE。MmFirstFreeSystemPte 的元素相当于是一个链表头，指向第一个空闲簇，系统 PTE 区域中所有的空闲簇直接在 PTE 中连接起来，形成一个 PTE 链表，如前所述。

当客户请求释放内存块时，如果对应大小的队列中尚有空间可供插入，则直接插入到队列中；如果已经达到设定的最大限制或插入队列失败，则归还到 PTE 维护的链表中，必要时可以合并成更大的簇。

下面我们讨论实现系统 PTE 区域内存管理算法的代码，位于 base\ntos\mm\sysptes.c 文件中。主要的函数有三个：MiInitializeSystemPtes、MiReserveSystemPtes 和 MiRelease-SystemPtes。

在系统的阶段 0 初始化过程中，当 MiInitMachineDependent 函数完成了系统非换页内存池的初始化以后，它调用 MiInitializeSystemPtes 函数来执行系统 PTE 区域的初始化，见 base\ntos\mm\i386\init386.c 文件的 3 295 行。MiInitializeSystemPtes 函数的代码位于 sysptes.c 的 2 247~2 510 行。它首先设定系统 PTE 区域的起始 PTE 和结束 PTE 的地址，即 MmSystemPtesStart 和 MmSystemPtesEnd 中对应于 SystemPteSpace 数组项的元素，然后将这段 PTE 全部清零。

在 MiInitializeSystemPtes 函数中，它也初始化一条单链表，其中每个节点是一块内存，称为 chunk。MmSysPteIndex 数组实际上定义了 5 种 chunk 的大小：1，2，4，8 和 16。在 MiInitializeSystemPtes 函数中有一个局部数组 Lists，定义了每种 chunk 的数量，在 Intel x86 平台上分别为：400，200，60，50 和 40。这样累加起来，共涉及 PTE 数量为 2 080 个，chunk 总数为 750 个。这些 chunk 节点本身是从系统非换页内存池中分配的，它们加入到由全局变量 MiSystemPteSListHead 定义的单链表中。然后利用此单链表来初始化队列数组 MiSystemPteNBHead，因此，由 MiSystemPteSListHead 定义的单链表对于外部不再可见，以后由 MiSystemPteNBHead 数组中的队列自动取用单链表中的节点。关于这部分代码逻辑，请参考 sysptes.c 文件的 2 359~2 427 行。

接下来通过以下一段代码完成对这些队列的初始化，它首先保留一个大的内存块，然后逐个释放各种小尺寸的内存块。这段代码的思路是，利用 MiReserveSystemPtes 申请一

个足够大的空闲 PTE 簇（包含 TotalPtes 个页面），然后按小块来释放这些页面，MiReleaseSystemPtes 函数自动将这些小内存块加入到各种大小的队列中。

```
PointerPte = MiReserveSystemPtes (TotalPtes, SystemPteSpace);

if (PointerPte == NULL) {
    MiIssueNoPtesBugcheck (TotalPtes, SystemPteSpace);
}

i = MM_SYS_PTE_TABLES_MAX;
do {
    i -= 1;
    do {
        Lists[i] -= 1;
        MiReleaseSystemPtes (PointerPte,
                             MmSysPteIndex[i],
                             SystemPteSpace);
        PointerPte += MmSysPteIndex[i];
    } while (Lists[i] != 0);
} while (i != 0);
```

最后，MiInitializeSystemPtes 函数完成初始化工作。

现在我们来看 MiReserveSystemPtes 是如何分配非换页页面的。它只有两个参数：NumberOfPtes 指页面的数量；SystemPtePoolType 只有两种可能：SystemPteSpace 或 NonPagedPoolExpansion，我们只考虑 SystemPteSpace 的情形。首先根据 NumberOfPtes 参数的值确定应该使用哪种大小的队列，如果能从队列中提取出一个节点，则直接把队列节点中的内存块（即起始 PTE）解出来返回给客户。而且，如果此种大小的内存块数量小于预定的最小数，即 MmSysPteListBySizeCount 数组的对应元素小于 MmSysPteMinimumFree 数组相应的元素，则调用 MiFeedSysPtePool 函数，以获得更多的此种大小的内存块。MiFeedSysPtePool 函数的逻辑很简单，其代码位于 base\ntos\mm\sysptes.c 文件的 475~538 行，它循环 10 次，每次调用 MiReserveAlignedSystemPtes 函数获得指定大小的 PTE 内存块，然后调用 MiReleaseSystemPtes 函数自动插入到相应的队列中。其结果相当于往特定的队列中插入 10 个指定大小的 PTE 内存块。

回到 MiReserveSystemPtes 函数中，如果参数 NumberOfPtes 指定的页面数量超过了 16，或者从队列中无法获得指定大小的内存块，则调用 MiReserveAlignedSystemPtes 函数申请新的内存块，并返回。

所以，我们现在来看 MiReserveAlignedSystemPtes 函数的逻辑。该函数比 MiReserveSystemPtes 多一个对齐参数。MiReserveSystemPtes 调用此函数时，设定对齐参数为 0，因此我们这里不考虑对齐的情形。MiReserveAlignedSystemPtes 首先从第一个空闲 PTE 开始，即 MmFirstFreeSystemPte 数组的元素，在一个 while 循环中找到合适大小的

空闲PTE簇，见代码672~781行。这是一个顺序扫描过程，如果找到第一个满足条件的簇恰好是所要求的大小，则执行ExactFit分支，把该簇从空闲簇链表中摘除下来；否则，找到的簇大于所要求的页面数，则把原来的簇缩小，后面的部分作为结果页面。如果对齐参数超过一个页面，则扫描逻辑更加复杂一些，但本质上仍然是找到满足要求的PTE簇。

最后，刷新这些页面PTE的硬件快表（TLB）项。然后返回。

理解了申请PTE的过程以后，我们现在来看释放PTE的过程，见MiReleaseSystemPtes函数，其代码位于sysptes.c文件的1 822~2 129行。该函数的参数指定了要释放的PTE起始位置和个数，以及PTE区域的类型。同样地，我们只考虑SystemPteSpace的情形，即系统PTE区域中的PTE释放过程。在调用此函数以前，指定范围的PTE必须是无效的，也就是说，对应的虚拟地址没有被映射到物理页面。所以，MiReleaseSystemPtes函数首先把这些PTE全部清零。如果待释放的PTE数量小于等于16，则考虑把这段PTE范围插入到MiSystemPteNBHead数组所指的队列中，而不是直接回收到空闲PTE链表中。参见代码1 901~1 974行。

如果插入队列不成功，或者PTE数量超过了16，则需要归还到PTE链表中。首先从第一个空闲PTE开始，即MmFirstFreeSystemPte数组的元素，在一个while循环中找到第一个其NextEntry域超过当前起始PTE的空闲簇，见代码2 004~2 128行。这是一个顺序扫描过程，因为PTE链表是按照地址先后顺序排列的。找到这样的空闲簇以后，首先考虑是否能跟它合并成一个更大的空闲簇，如果此空闲簇的紧后面恰好是要释放的PTE范围，则将两者合并起来，见代码2 030~2 048行；如果不能合并，则插入一个新的空闲簇在它后面，见代码2 060~2 080行。接着，如果要释放的PTE范围恰好紧贴下一个空闲簇，则将两者合并，见代码2 088~2 111行。然后函数结束。

最后简单小结一下，系统PTE区域是指把PTE当做资源来管理，这些PTE按照顺序构成了一个单链表。当内核代码通过MiReserveSystemPtes申请得到一段连续的PTE时，它必须自己映射物理页面。MiReserveSystemPtes函数仅仅返回了这段PTE的起始地址，它并不负责填充PTE中的内容。对称地，当内核代码调用MiReleaseSystemPtes函数来释放PTE范围时，它必须先解除这些PTE的页面映射。这两个函数既可以用于对系统PTE区域的内存管理，也用于非换页内存池扩展区域的内存管理，关键的区别在于：系统PTE区域的管理算法采用了5个队列作为缓存，以缓解频繁的保留和释放操作所带来的开销；而非换页内存池扩展区域未引入队列缓存，所以它的保留和释放总是直接扫描PTE链表。在Windows系统中，系统PTE区域主要用于I/O、内核栈等动态映射页面的情形，而非换页内存池扩展区域仅用于非换页内存池的扩展。

4.3 进程内存管理

在现代操作系统中，一个重要的特征是每个进程有其独立的地址空间，因而本质上，每个进程的数据是隔离的，不同进程之间不会干扰彼此的数据。本章前面曾经提到，在 Intel x86 体系结构下，每个进程的虚拟地址空间是 4 GB，其中 0x0~0x7fffffff 是进程地址空间，0x80000000~0xffffffff 是系统地址空间。系统地址空间中的数据仅在内核模式下才可以访问。上一节介绍了系统地址空间的布局结构和内存管理。这一节我们来介绍进程地址空间的内存管理，首先讨论进程地址空间是如何建立并初始化的，然后介绍地址空间的切换以及虚拟地址空间中的内存管理，最后介绍进程地址空间中一种特殊的内存资源管理对象——内存区对象。

4.3.1 地址空间的创建和初始化

第 3 章在介绍进程创建过程时曾经提到，当 PspCreateProcess 创建一个进程时，如果指定的父进程不为 NULL，则需要创建一个新的地址空间，参见 3.4.3 节中关于 PspCreateProcess 函数流程的描述。进程地址空间的创建过程是由 MmCreateProcessAddressSpace 函数来完成的，而且，地址空间被创建起来以后，PspCreateProcess 还调用 MmInitializeProcessAddressSpace 函数来初始化该地址空间。

PspCreateProcess 函数是在父进程的环境中被调用的，它首先创建一个进程对象（EPROCESS），并初始化 EPROCESS 对象中的部分成员，然后调用 MmCreateProcessAddressSpace 函数创建地址空间，下面是该函数的原型：

```
BOOLEAN
MmCreateProcessAddressSpace (
    IN ULONG MinimumWorkingSetSize,
    IN PEPROCESS NewProcess,
    OUT PULONG_PTR DirectoryTableBase
    );
```

参数 MinimumWorkingSetSize 和 NewProcess 分别指定了待创建进程的最小工作集的大小和它的进程对象，DirectoryTableBase 是输出参数，指向进程地址空间的页目录地址。

WRK 包含三份 MmCreateProcessAddressSpace 函数的代码，分别位于 base\ntos\mm\i386\procx86.c、base\ntos\mm\procsup.c 和 base\ntos\mm\i386\procpae.c 文件中。这三份代码分别对应于 Intel x86、64 位 Windows 以及 PAE 模式 Intel x86 的实现版本。三份实现有些微差别，但基本逻辑相仿，这里仅介绍 Intel x86 版本的 MmCreateProcessAddressSpace 函数，其代码位于 base\ntos\mm\i386\procx86.c 文件的 28~362 行。MmCreateProcessAddressSpace 函数的流程如下：

(1) 首先调用 MiChargeCommitment 函数确保系统有足够的页面文件空间。在 Intel x86 系统上，需要 4 个页面。

(2) 检查系统当前可用的物理页面是否达到 MinimumWorkingSetSize 参数指定的要求。如果不够，则返回失败。

(3) 申请一个物理页面作为页目录页面。该页面已清零，其页帧编号置于局部变量 PageDirectoryIndex 中。

(4) 再申请一个物理页面作为超空间页表页面。同样地，该页面已清零，其页帧编号置于局部变量 HyperSpaceIndex 中。

(5) 再申请一个物理页面作为 VAD 位图。关于 VAD（虚拟地址描述符），请参考 4.3.3 节的介绍。同样地，该页面已清零，其页帧编号置于局部变量 VadBitMapPage 中。

(6) 再申请一个物理页面作为工作集链表。同样地，该页面已清零。

(7) 接下来初始化新进程 EPROCESS 对象的 Vm.MinimumWorkingSetSize、WorkingSetPage 和 DirectoryTableBase 域。

(8) 初始化超空间页表页面。在系统 PTE 区域中保留一个 PTE，用于映射超空间页表页面，以便访问其中的内容。在超空间页表页面中，填写好对应于 VAD 位图和工作集链表的页表项。完成以后，释放此 PTE。见代码 229~262 行。这里我们看到了系统 PTE 区域中 PTE 资源的一个用法。

(9) 在 PFN 数据库中，设置页目录页面的 PTE 地址，为 0xc0300000。

(10) 把新进程加入到系统内部维护的进程链表 MmProcessList 中，EPROCESS 对象的 MmProcessLinks 域是该进程在这一链表中的节点。凡是有自己独立地址空间的进程均在此链表中。

(11) 接下来初始化页目录页面。首先在系统 PTE 区域中保留一个 PTE，用于映射页目录页面，以便访问其中的内容。在页目录页面中，把系统空间（从 0x80000000 开始的地址范围）的 PDE 复制到页目录页面中（实际上恰好后半个页面）。然后映射超空间（0xc0400000~0xc0c00000）中第一个 PDE 项，余下的清零。接下来，在页目录中，指定 0xc0000000 的页目录项指向页目录页面自身，这一步是为了在新进程中也具有图 4.13 所示的结构。然后，把这 4 个页面的开销记录到 MmProcessCommit 全局变量中。完成以后，释放此 PTE。见代码 294~353 行。

(12) 最后，调用 MiSessionAddProcess 函数把新进程加入到当前进程（即新进程的父进

程）所在的会话空间（即全局变量 MmSessionSpace）中。MiSessionAddProcess 函数所做的工作是：更新会话空间中的进程计数器，并且把新进程加入到会话空间维护的进程链表中。关于 MiSessionAddProcess 函数的代码，请参考 base\ntos\mm\session.c 文件的 198~264 行。

我们可以看到，在新进程地址空间中，系统空间部分的页目录项已经被拷贝到页目录中。由于这些页目录项指向相应页表的页帧编号，所以，在新地址空间中，系统空间部分已经映射好，包括超空间的页目录项；除此以外，MmCreateProcessAddressSpace 函数也已经初始化新进程的超空间，包括映射好它的 VAD 位图页面和工作集链表页面。该函数返回以后，在新进程地址空间中，用户空间部分，即 0x0~0x7fffffff，仍然是一片空白。这部分的初始化工作是由 MmInitializeProcessAddressSpace 函数来完成的，所以，接下来我们介绍此函数，其原型如下：

```
NTSTATUS
MmInitializeProcessAddressSpace (
    IN PEPROCESS ProcessToInitialize,
    IN PEPROCESS ProcessToClone OPTIONAL,
    IN PVOID SectionToMap OPTIONAL,
    IN OUT PULONG CreateFlags,
    OUT POBJECT_NAME_INFORMATION *AuditName OPTIONAL
    );
```

MmInitializeProcessAddressSpace 函数也是在父进程的环境中被调用的，其代码位于 base\ntos\mm\procsup.c 文件的 602~1 466 行。参数 ProcessToInitialize 代表要初始化的目标进程；ProcessToClone 是一个可选参数，表示新进程的地址空间可以从该进程拷贝获得；SectionToMap 也是一个可选参数，它提供了一个内存区对象，表示在新进程地址空间中映射此对象；CreateFlags 表示各种与进程创建相关的标志，在创建进程时，与内存管理有关的标志是 PROCESS_CREATE_FLAGS_ALL_LARGE_PAGE_FLAGS。CreateFlags 参数是一个输入输出参数，当指定了此标志时，其输出值表示是否成功地以大页面的方式来映射进程的映像文件。最后一个参数 AuditName 是一个输出参数，也是可选的，它提供了一个对象名称信息指针。注意，ProcessToClone 和 SectionToMap 参数至多只能有一个是有效的。

在 WRK 中，MmInitializeProcessAddressSpace 函数在以下四种情形下被调用：

(1) 创建一个新的用户进程，此时 SectionToMap 参数有效，ProcessToClone 为 NULL。见 base\ntos\ps\create.c 文件的 1 366 行代码，位于 PspCreateProcess 函数中。

(2) 创建一个新的用户进程，但没有内存区对象，类似于 fork()这样的操作，所以根据父进程来初始化新进程。见 base\ntos\ps\create.c 文件的 1 394 行代码，也位于 PspCreateProcess 函数中。

MiAllocateVad 函数创建一个 VAD，来保留共享的用户页面（其地址为 0x7ffe0000，它与系统空间中的一个页面映射到同一个物理页面上，从而实现共享）。若当前系统不是以 /3GB 方式启动的，则用户代码能够申请的最高 VAD 总是低于此共享页面，所以用户代码不能删除它，因而这里就不需要创建 VAD 来保留该页面。

(9) 1 073~1 142 行代码仅对多级页表的硬件结构起作用，这里忽略。

(10) 接下来根据系统注册表设置以及当前系统环境来确定是否在新进程中由高向低分配虚拟地址空间，见 1 157~1 161 行代码。

(11) 接下来 1 229~1 362 行代码专门处理创建新进程的情形。在这种情况下，参数 SectionToMap 指向一个内存区对象。首先，利用内存区对象中的文件名信息，截取其最后部分，构成一个 ImageFileName，赋到新进程 EPROCESS 对象的 ImageFileName 域中。然后调用 MmMapViewOfSection 函数，在新进程中映射同样的内存区对象。关于 MmMapViewOfSection 函数，请参考 4.3.4 节。我们现在可以知道的一点是，经过此调用以后，在新进程中也同样映射了 SectionToMap 内存区对象的可执行映像文件。显然，这是必要的步骤，因为新进程的执行将从指定的可执行文件开始。接下来调用 PsMapSystemDll 函数，将系统 DLL 即 ntdll.dll 映射到新进程地址空间中。PsMapSystemDll 函数同样通过调用 MmMapViewOfSection 函数来完成其功能，见 base\ntos\ps\psinit.c 文件 759~837 行代码。这里的系统 DLL 是一个已经在系统初始化阶段建立起来的内存区对象，供每个进程创建时直接映射。最后，MmInitializeProcessAddressSpace 调用 MiAllowWorkingSetExpansion 函数，把新进程的虚拟地址空间插入到工作集管理链表中，然后，它调用 KeDetachProcess 函数并返回。

(12) 1 364~1 446 行代码处理进程复制的情形。不考虑 64 位系统的情形，这部分代码非常简洁。它首先调用 KeDetachProcess 函数以便回到父进程环境中，然后调用 MiCloneProcessAddressSpace 函数复制地址空间，再调用 MiAllowWorkingSetExpansion 函数，把新进程的虚拟地址空间插入到工作集管理链表中。然而，MiCloneProcessAddressSpace 函数较为复杂，其代码位于文件 base\ntos\mm\forksup.c 的 71~2 461 行。它的主要逻辑是，检查当前进程的每一个 VAD，根据 VAD 中描述的页面属性，检查 PTE，并拷贝到新进程中。对于父进程中的私有页面，它们将成为新旧进程之间共享且符合写时复制特性的页面。关于写时复制特性，请参考 4.4.4 节。

(13) 如果是系统进程，则什么也不做，直接调用 KeDetachProcess。

(14) 最后，如果新进程是一个复制的进程或系统进程，则关闭大页面创建标志（PROCESS_CREATE_FLAGS_LARGE_PAGES）。然后退出。

经过以上两个步骤的初始化以后，一个进程的地址空间已经初步建立起来，可执行文件的映像和 ntdll.dll 文件已被映射到恰当的虚拟地址范围中。并且，新进程的工作集也已经初始化，可以支持页面换入换出了。

4.3.2 地址空间切换

了解了进程地址空间的创建和初始化过程以后，我们来看一看当系统在不同进程之间切换时，内存地址空间是如何转移的。每个进程都有独立的页目录，页目录的前半部分对应于虚拟地址空间的 0x0~0x7fffffff 范围，这是每个进程私有的，所以，从老的进程切换到新进程以后，这部分地址空间将完全转移到新进程的地址空间。

然而，0x80000000~0xffffffff 之间的地址范围是所有进程共享的，但每个进程的页目录页面是独立的。我们在 MmCreateProcessAddressSpace 函数中已经看到，新进程页目录拷贝了此地址范围的 PDE 项，因此切换到新进程以后，新地址空间中的 PDE 项将指向全局的页表。换句话说，无论在哪个进程环境下，系统地址空间中的页表都是共享的，因为页表页面是由 PDE 中的 PFN 域来指定的，而这些 PDE 是在父子进程之间拷贝得来的。唯独有两个地方例外，下面分别介绍。

参考图 4.14 所示的系统地址空间结构，系统空间中使用的绝大多数页表在系统初始化的时候已经分配并映射好，唯独会话空间、系统视图和换页内存池部分的地址范围未经初始化。其中会话空间和系统视图部分由会话管理器和 Windows 子系统来初始化和管理，这里不作讨论。换页内存池部分有可能在不同进程的页目录中有不同的视图。当系统内核代码在某个进程环境中申请了换页内存池的内存，从而导致新的页表被映射到页目录中时，只有这个进程的页目录中的 PDE 指向新的页表页面；若系统内核代码在另一个进程环境中访问这一 PDE 所映射的虚拟地址范围，则内存访问指令将触发页面错误，在页面错误处理例程中，这一项将被填充上。系统变量 MmSystemPagePtes 记录了换页内存池的页表映射。因此，Windows 利用“延迟计算（lazy evaluation）”的做法来保持换页内存池页表映射的一致性，会话空间中的内存管理也采用了类似的做法。从逻辑角度来看，换页内存池对于每个进程也是共享和统一的，只是一致性维护滞后而已。

从上一节描述的进程地址空间创建过程我们可以看出，在每个进程的系统地址空间中，超空间部分，包括 VAD 位图和进程工作集链表，也是每个进程单独拥有的。在进程的页目录中，此地址范围的 PDE 指向私有的页表页面，所以，在进程切换前后，新老进程的工作集链表和 VAD 位图的地址所对应的 PDE 指向各自的超空间页表页面。在 WRK 的代码中，当我们看到全局变量 MmWorkingSetList 时，应意识到这是当前进程的工作集

链表，而并非像其他的全局变量那样，是一个指向全系统范围共享内存的指针。

因此，Windows 在进程切换时，只需直接切换页目录页面，即 CR3 寄存器，而无须对页目录中的 PDE 作任何调整或一致性维护，即可实现从一个进程的地址空间转移到另一个进程的地址空间，并且系统空间中除了超空间和会话空间部分以外，在新老进程中保持一致。

4.3.3 进程地址空间的内存管理

进程地址空间的范围是 0x0~0x7fffffff，它是进程整个虚拟地址空间的一部分。用户模式的代码只能访问这一段地址范围。对于进程地址空间，用户程序必须经过“保留（reserve）”和“提交（commit）”两个阶段才能使用一段地址范围。“保留一段地址范围”的用意是，将这段地址范围保留起来，但并不真正使用，由于这段地址范围不占用任何物理内存或其他外存空间，所以并不形成实质的开销。这对于有些需要连续地址空间的程序有意义，它们可以在初始时保留一段大地址范围，以后需要的时候陆续使用。提交地址范围是指这段地址终究要消耗物理内存，由于 Windows 支持物理内存与页面文件之间的交换，因此可提交的内存数量是，可用物理内存总量去除系统使用的物理内存数量后，再加上页面文件的大小。

在已提交的地址范围中，当页面被访问时，一定要先映射到物理内存页面上。这些页面要么是一个进程私有的、不共享的，要么被映射到一个内存区的视图上。后者可以被多个进程共享，前面 4.3.1 节我们已经看到过内存区映射的用法了。下一节我们还会进一步讨论 Windows 中内存区对象是如何实现的。

在 Windows API 中，应用程序通过 VirtualAlloc 或 VirtualAllocEx 来保留或提交地址范围；以后通过 VirtualFree 或 VirtualFreeEx 函数来解除已提交的地址范围，或者完全释放此地址范围。解除提交是指回到保留状态。因此，对于进程地址空间中的任何一个页面地址范围，它一定处于三种状态之一：空闲的、保留的，或已提交。

对于已提交的页面，系统会根据总的内存使用情况来调度它们。当物理内存紧张时，系统会选择一些页面，将它们换出到页面文件中，待下次使用的时候，再将它们换回来。通常情况下，应用程序并不需要干预系统的页面调度机制。在一些特殊情况下，应用程序也可以通过 VirtualLock 函数来锁住已提交的页面，使得它们总是留在物理内存中；以后再调用 VirtualUnlock 函数来解除这种锁定。关于系统对已提交页面的交换和调度，我们将在本章后面介绍页面交换和工作集管理时再进一步解释。接下来我们讨论进程地址空间的管理，即地址空间中保留区域的管理。

Windows 的进程地址空间是通过 VAD（虚拟地址描述符，Virtual Address Descriptor）来管理的。VAD 对象描述了一段连续的地址范围。在整个地址空间中，保留的或提交的地址范围有可能是不连续的，所以，Windows 使用一棵平衡二叉搜索树（称为 AVL 树）来管理 VAD 对象。如 4.3.1 节所述，VAD 树的初始化是在 MmInitializeProcessAddressSpace 函数中进行的。进程 EPROCESS 对象中的 VadRoot 域指向此树的根。以下是 VadRoot 的类型 MM_AVL_TABLE 以及 AVL 树中各节点的类型 MMADDRESS_NODE 的定义：

```
typedef struct _MMADDRESS_NODE {
   union {
      LONG_PTR Balance : 2;
      struct _MMADDRESS_NODE *Parent;
   } u1;
   struct _MMADDRESS_NODE *LeftChild;
   struct _MMADDRESS_NODE *RightChild;
   ULONG_PTR StartingVpn;
   ULONG_PTR EndingVpn;
} MMADDRESS_NODE, *PMMADDRESS_NODE;

typedef struct _MM_AVL_TABLE {
   MMADDRESS_NODE  BalancedRoot;
   ULONG_PTR DepthOfTree: 5;
   ULONG_PTR Unused: 3;
   ULONG_PTR NumberGenericTableElements: 24;

   PVOID NodeHint;
   PVOID NodeFreeHint;
} MM_AVL_TABLE, *PMM_AVL_TABLE;
```

当使用 AVL 树来管理虚拟地址空间中的 VAD 时，树中节点的真正类型是 MMVAD，其定义如下：

```
typedef struct _MMVAD {
   union {
      LONG_PTR Balance : 2;
      struct _MMVAD *Parent;
   } u1;
   struct _MMVAD *LeftChild;
   struct _MMVAD *RightChild;
   ULONG_PTR StartingVpn;
   ULONG_PTR EndingVpn;

   union {
      ULONG_PTR LongFlags;
      MMVAD_FLAGS VadFlags;
   } u;
   PCONTROL_AREA ControlArea;
   PMMPTE FirstPrototypePte;
   PMMPTE LastContiguousPte;
   union {
      ULONG LongFlags2;
      MMVAD_FLAGS2 VadFlags2;
   } u2;
} MMVAD, *PMMVAD;
```

需要说明的一点是，由于 VadRoot 的 BalancedRoot 域是一个内嵌结构体，它无法从 MMADDRESS_NODE 适应到 MMVAD，所以，虚拟地址空间 VAD 树的真正根节点是 VadRoot.BalancedRoot.RightChild ， VadRoot.BalancedRoot.LeftChild 并未使用。VadRoot.BalancedRoot.RightChild 的静态类型是 MMADDRESS_NODE，但其实际的对象类型是 MMVAD。从这两种类型的定义可以看出，MMVAD 是 MMADDRESS_NODE 的一个扩展，它增加了几个数据成员。在 Windows 内存管理系统中，除了 VAD 的管理使用了 AVL 树以外，进程的 PhysicalVadRoot 也是一棵 AVL 树，其中的节点类型为 MI_PHYSICAL_VIEW。所以，这里 AVL 树的做法正是 C++多态性的 C 语言实现。

初始时，一个进程的 VAD 树是空的，随着运行的需要，不断地有地址范围被保留或提交，从而逐渐形成了 VAD 树，图 4.20 显示了一个实际正在运行中的进程的 VAD 树，其中每个节点都显示了它的地址范围，这里给出的数是十六进制形式的虚拟页面编号（即虚拟地址的高 20 位），右上箭头表示左子节点，右下箭头表示右子节点。在查找地址或者插入一个新节点时，可以比较待查找的地址或地址范围是否落在某个地址范围中，从而知道该地址或地址范围是否已经被保留或提交。

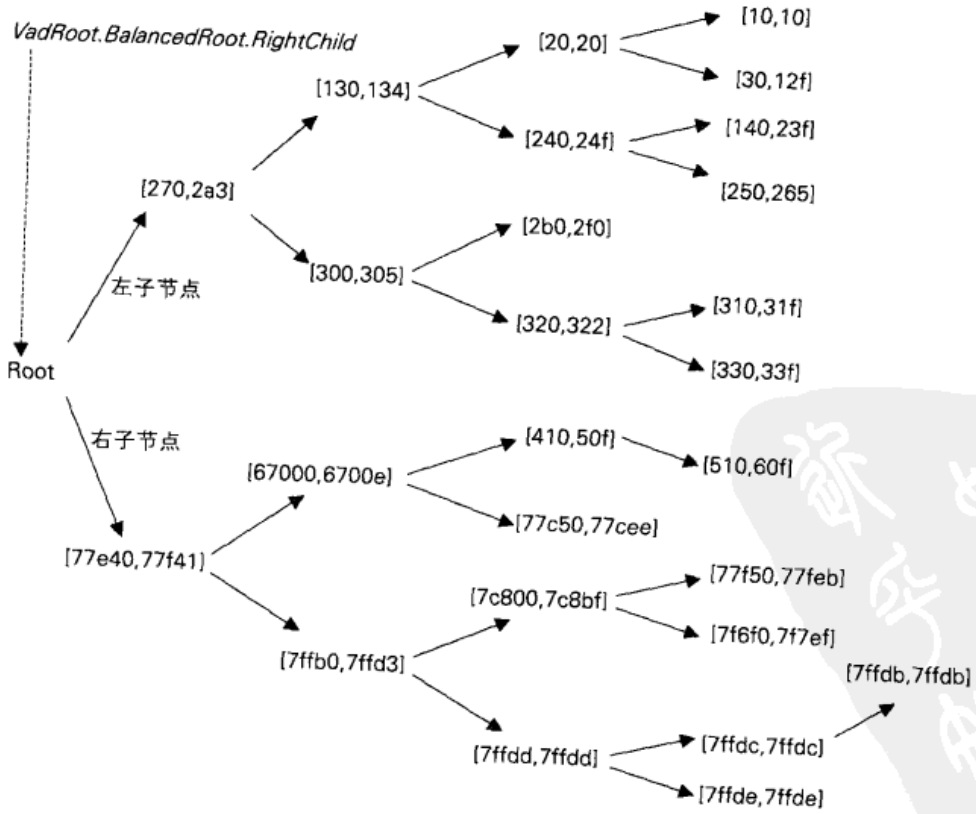


图 4.20 一个实际进程的 VAD 树

在各个节点的 MMVAD 结构中，Balance、Parent、LeftChild 和 RightChild 域用于维护 AVL 树本身，StartingVpn 和 EndingVpn 域规定了一个节点的地址范围。AVL 树的实现代码位于 base\ntos\mm\addrsup.c 文件中，其中特别值得一提的是以下三个函数：

```
TABLE_SEARCH_RESULT
MiFindNodeOrParent (
    IN PMM_AVL_TABLE Table,
    IN ULONG_PTR StartingVpn,
    OUT PMMADDRESS_NODE *NodeOrParent
    );

VOID
FASTCALL
MiInsertNode (
    IN PMMADDRESS_NODE NodeToInsert,
    IN PMM_AVL_TABLE Table
    );

VOID
FASTCALL
MiRemoveNode (
    IN PMMADDRESS_NODE NodeToDelete,
    IN PMM_AVL_TABLE Table
    );
```

其中，MiFindNodeOrParent 函数的功能是，对于一个给定的虚拟地址，即参数 StartingVpn，要么找到一个包含此虚拟地址的 AVL 节点，要么找到一个可当做该地址父节点的 AVL 节点。其返回值有四种可能：TableEmptyTree、TableFoundNode、TableInsertAsLeft 或 TableInsertAsRight。该函数的实现代码位于 addrsup.c 文件的 212~339 行。MiInsertNode 和 MiRemoveNode 函数实现了标准的 AVL 树的节点插入和删除算法。MiInsertNode 利用 MiFindNodeOrParent 函数找到插入点，并插入新节点，然后依据 AVL 树的插入节点调整规则，使 AVL 树保持平衡。MiRemoveNode 函数删除指定的节点，并根据需要调整节点，使 AVL 树仍然保持平衡。关于 AVL 树的算法，请参考有关的算法论著[TAOCP-2][ALG]。Windows 内核中的 AVL 树实现对应于 Donald Knuth 的书中的算法描述，代码非常简洁，这里不再赘述。

MMVAD 结构的 VadFlags、VadFlags2、ControlArea、FirstPrototypePte 和 LastContiguousPte 域描述了一段虚拟地址范围的必要信息：VadFlags 和 VadFlags2 域是两个 32 位整数，它们构成了一些标志域，包括该地址范围中数据的保护属性、子进程是否可以继承、VAD 类型、是否为私有内存、是否写时复制，等等；ControlArea 域的类型为 CONTROL_AREA 结构，它指向与这段地址范围相关联的内存区对象的控制区，在下一节介绍内存区对象时，我们将会看到 MMVAD 与内存区对象之间的关系；FirstPrototypePte 域指向一个原型 PTE，当内存区对象被映射到一个进程中时，FirstPrototypePte 域指向所

映射的视图在该内存区对象的原型 PTE 阵列中的第一个 PTE；LastContiguousPte 域指向与所映射的视图相对应的最后一个原型 PTE。关于原型 PTE，请参考 4.4 节。当内存区对象被映射到一个进程中时，内存管理器创建一个 VAD 对象，并初始化这两个 PTE 指针域。对于一个 VAD 的地址范围中的页面，当第一次被访问时，内存管理器可能需要为该页面构造一个有效的 PTE，它正是利用 VAD 中的信息来做到这一点。如果用户访问的地址不落在 VAD 树的任何一个节点上，或者落在一个已保留但尚未提交的地址范围内，那么，内存管理器能确切地知道客户线程的访问是非法的，从而产生一个访问违例。

MiInsertVad 和 MiRemoveVad 是两个内联 C 函数，见 base\ntos\mm\mi.h 中的代码，它们分别调用了 MiInsertNode 和 MiRemoveNode 来实现在进程的 VAD 树中插入和删除节点的功能。通过 VAD 树可以快速地查找到一个地址是否已被保留或提交，但如果想找到一段连续的空闲地址空间，则通过 VAD 树来搜索并非最佳选择。为此，Windows 另有一个专门的页面来记录进程地址空间中的 VAD 覆盖情况。在 4.3.1 节介绍地址空间创建过程时，我们看到过有一个 VAD 位图页面，它位于当前进程的超空间区域，确切的位置由宏 VAD_BITMAP_SPACE 指定。

VAD 位图中的每一位代表了 64 KB 内存是否出现在 VAD 树中。因此，要找到连续的空闲地址范围，只需在 VAD 位图中搜索连续的空闲标志位即可。搜索位图的效率比较高，在介绍系统地址空间换页内存池的管理算法时我们已经看到过这种用法。除了 MmCreateProcessAddressSpace 函数以外，涉及 VAD 位图的函数主要有三个：MiInsertVadCharges、MiReturnPageTablePageCommitment 和 MiFindEmptyAddressRange。MiInsertVadCharges 函数往往伴随在 MiInsertVad 函数调用之后，负责记录各种内存开销，包括 VAD 对象本身和 VAD 中指定地址范围所需要的页表等开销。而且它也在 VAD 位图中记录相应的位，其代码片段如下（见 base\ntos\mm\vadtree.c 文件）：

```
StartBit = (ULONG)(((ULONG_PTR)MI_64K_ALIGN (MI_VPN_TO_VA (Vad->StartingVpn))) / X64K);
EndBit = (ULONG) (((ULONG_PTR) MI_64K_ALIGN (MI_VPN_TO_VA (Vad->EndingVpn))) / X64K);

VadBitMap.SizeOfBitMap = MiLastVadBit + 1;
VadBitMap.Buffer = VAD_BITMAP_SPACE;

RtlSetBits (&VadBitMap, StartBit, EndBit - StartBit + 1);
```

在 MiReturnPageTablePageCommitment 函数中，除了归还页表页面以外，也会把指定地址范围所对应的 VAD 位图中的位清除，相应的代码片段如下（见 base\ntos\mm\mmquota.c 文件）：

```
StartBit = (ULONG) (((ULONG_PTR) MI_64K_ALIGN (StartingAddress)) / X64K);
EndBit = (ULONG) (((ULONG_PTR) MI_64K_ALIGN (EndingAddress)) / X64K);
```

```
if (StartBit <= EndBit) {

    VadBitMap.SizeOfBitMap = MiLastVadBit + 1;
    VadBitMap.Buffer = VAD_BITMAP_SPACE;

    RtlClearBits (&VadBitMap, StartBit, EndBit - StartBit + 1);
    ......
}
```

MiFindEmptyAddressRange 函数本质上是扫描位图的过程。如果要找的地址范围正好是 64 KB 的倍数，则首先尝试在进程的 VAD 位图中找到连续的位。若不成功，则退回到 VAD 树，利用 MiFindEmptyAddressRangeInTree 函数在 VAD 树中查找相邻两个 VAD 之间有足够的地址间隔。MiFindEmptyAddressRange 函数的代码请参见 base\ntos\mm\vadtree.c 文件。

在介绍了 Windows 进程地址空间的底层管理数据结构以后，我们再来看一下上层组件是如何使用这些数据结构和功能的。Windows 内核向应用层暴露了一组与内存有关的系统服务，其中最主要的是两个函数：NtAllocateVirtualMemory 和 NtFreeVirtualMemory。通常应用程序调用的 VirtualAlloc、VirtualAllocEx、VirtualFree 和 VirtualFreeEx 函数正是通过这两个函数来实现的。NtAllocateVirtualMemory 和 NtFreeVirtualMemory 函数用到了刚刚介绍的 VAD 管理结构，接下来对这两个函数作一简单介绍。

NtAllocateVirtualMemory 函数位于 base\ntos\mm\allocvm.c 文件的 53~2 033 行。由于该函数要处理多种情形，包括保留内存、提交内存以及其他一些针对应用层需要的功能，而且还要解析所提交内存的各种保护属性，并且需要考虑物理地址扩展（PAE）以及 32 位和 64 位兼容性等因素，所以，其代码涉及的情形较为复杂。它首先根据进程的句柄找到进程的 EPROCESS 对象，然后构造一个 VAD 对象，利用参数中的信息来填充 VAD 对象中的有关域，再把 VAD 对象插入到 VAD 树中，并调用 MiInsertVadCharges 函数记录内存的使用开销，最后更新进程中的相关域。如果是提交内存的情形，则还需要计算各种管理开销，以及更新页目录项和页表项属性。

NtFreeVirtualMemory 函数位于 base\ntos\mm\freevm.c 文件的 41~861 行。它首先找到进程的 EPROCESS 对象，然后根据参数中给出的地址，找到相应的 VAD 节点。如果要释放的地址范围超出了 VAD 节点的范围，则返回失败。也就是说，NtFreeVirtualMemory 函数仅用于一个 VAD 节点地址范围内的内存释放。然后，NtFreeVirtualMemory 函数检查 VAD 节点中的信息以确保此释放操作是合适的，并且删除此 VAD 节点或者将 VAD 节点的地址范围缩小，甚至有可能还要切分出一个新的 VAD 节点来。最后，它调用 MiDeleteVirtualAddresses 清除这段地址范围的 PTE 以及页面，并且调用 MiReturnPageTablePageCommitment 函数归还页表页面以及恢复 VAD 位图中相应的位。

4.3.4 内存区对象

在 VAD 节点中，有一种重要的类型是内存区对象（section object）。内存区对象是 Windows 平台上两个或多个进程之间共享内存的一种常用方法，它可以被映射到系统的页面文件或者其他的文件中。实际上，内存区对象代表了一种物理存储资源，它可能在物理内存中，也可能在系统页面文件中，或者在其他的文件中。Windows 进程管理器通过内存区对象把一个可执行文件的映像映射和加载到内存中，然后创建新进程的地址空间，所以，在创建进程地址空间的函数中，我们已经看到过内存区对象的用法了。有些程序利用内存区对象将一个文件映射到进程地址空间中，从而直接以操作内存的方式来访问文件，这既简化了对文件的读写操作，也提高了性能。Windows 缓存管理器利用内存区对象来访问缓存文件中的数据。

内存区对象也是 Windows 的一种内核对象，由对象管理器来分配和释放。当然，对象管理器只管理对象头部分，而对象体是由内存管理器来管理的。以下是内存区对象体的定义（见 base\ntos\mm\mi.h 文件）：

```
typedef struct _SECTION {
    MMADDRESS_NODE Address;
    PSEGMENT Segment;
    LARGE_INTEGER SizeOfSection;
    union {
        ULONG LongFlags;
        MMSECTION_FLAGS Flags;
    } u;
    MM_PROTECTION_MASK InitialPageProtection;
} SECTION, *PSECTION;
```

SECTION 结构中的成员非常易于理解：Address 域是内存区的 VAD 节点，用于把所有特定类型（SEC_BASED）的内存区对象组织成一棵平衡树，参见本节后文的描述；Segment 域指向一个段对象，本节后文将会介绍段对象；SizeOfSection 域是内存区的大小；Flags 域包含了内存区的一组标志；InitialPageProtection 域指定了内存区中页面的保护属性。

内存区的创建是由 MmCreateSection 函数来完成的。该函数的代码位于 base\ntos\mm\creasect.c 文件的 402~1 611 行，其原型如下：

```
NTSTATUS
MmCreateSection (
    __deref_out PVOID *SectionObject,
    __in ACCESS_MASK DesiredAccess,
    __in_opt POBJECT_ATTRIBUTES ObjectAttributes,
    __in PLARGE_INTEGER InputMaximumSize,
    __in WIN32_PROTECTION_MASK SectionPageProtection,
```

```
__in ULONG AllocationAttributes,
__in_opt HANDLE FileHandle,
__in_opt PFILE_OBJECT FileObject
);
```

内存区对象有两种，一种建立在页面文件的基础之上，称为页面文件支撑的内存区（page-file-backed section），另一种被映射到其他文件中，称为文件支撑的内存区（file-backed section），也称为文件映射对象（file mapping object）。在 MmCreateSection 的参数中，FileHandle 参数若非空，则说明待创建的内存区对象是后一种，此时 InputMaximumSize 参数指定了内存区最大可被扩展或映射的大小。FileObject 参数仅用于缓存管理器。如果 FileHandle 参数为空，则说明这是要创建一个由页面文件来支撑的内存区，即第一种对象，此时，InputMaximumSize 参数指定了内存区的大小。ObjectAttributes 参数指定了对象管理器所需要的信息，包括对象名、安全描述符等。如果内存区对象具有名称，则任何一个有权限访问此内存区的进程都可以通过名称来引用此内存区对象，从而实现共享访问；反之，未命名的内存区对象只能通过父子进程的继承关系来实现共享，甚至不能共享。在不能共享的情况下，内存区的页面都是私有页面。DesiredAccess 参数指定了访问方式：EXECUTE、READ 和 WRITE。SectionPageProtection 参数指定了内存区中各个页面的保护标志。AllocationAttributes 参数指示了内存区对象中内存段的分配方式：SEC_BASED 要求在每个进程地址空间中相同的虚拟地址上分配；SEC_RESERVE 要求内存区所有的页面被保留；SEC_COMMIT 要求内存区所有的页面被提交；SEC_IMAGE 表明 FileHandle 参数指定了一个可执行映像文件；SEC_FILE 表明 FileHandle 参数指定了一个映射文件；SEC_LARGE_PAGES 表明该内存区使用大页面，并且立即分配物理内存。

MmCreateSection 函数的代码逻辑大致如下：如果 FileHandle 或者 FileObject 参数非空，则利用文件对象中的信息来填充一个 CONTROL_AREA 对象，包括其中的 SEGMENT 对象。如果文件对象还没有 SEGMENT 对象，则调用 MiCreateImageFileMap 或 MiCreateDataFileMap 创建一个新的 SEGMENT 对象。如果参数中没有指定文件句柄或文件对象，则调用 MiCreatePagingFileMap 函数创建一个 SEGMENT 对象。有了 SEGMENT 对象以后，MmCreateSection 调用对象管理器的 ObCreateObject 函数创建一个 SECTION 对象，并填充其中的信息，最后返回该 SECTION 对象。

不难想象，内存区对象与文件对象之间有特殊的关联关系，并且，还有其他一些对象，如 VAD 对象、CONTROL_AREA 对象和 SEGMENT 对象，也参与到它们之间的关联关系中。图 4.21 显示了这些对象之间的结构关系。

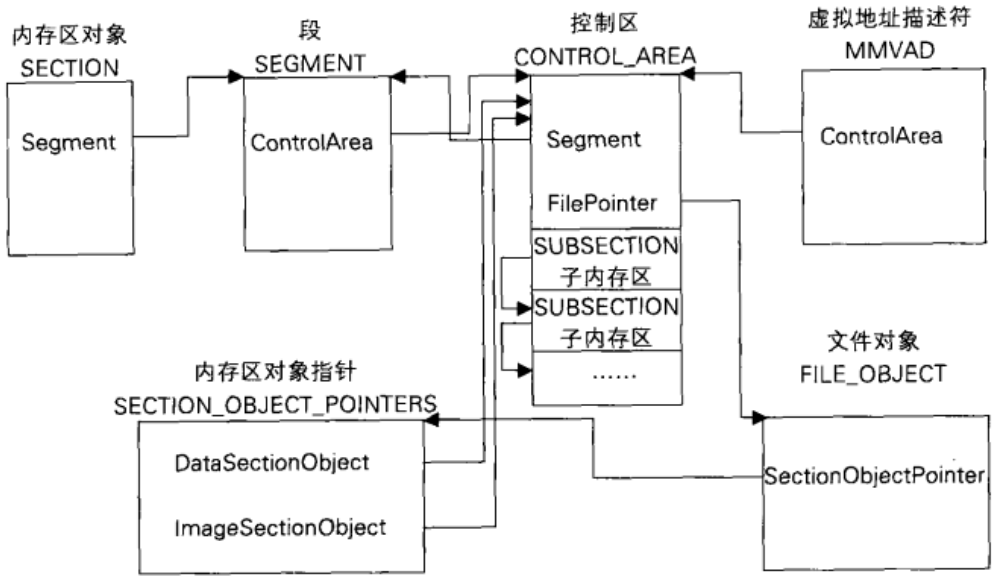


图 4.21 内存区对象与其他对象的关联结构

内存区对象 SECTION 结构中的 Segment 成员指向一个段（SEGMENT）对象。段对象是真正描述内存区数据的对象。在 MmCreateSection 函数中，我们看到有三个函数可以创建段对象，分别是 MiCreatePagingFileMap、MiCreateImageFileMap 和 MiCreateDataFileMap。这三个函数分别对应于三种内存区：由页面文件支撑的内存区，可执行映像文件内存区，以及数据文件映射内存区。后两者有对应的文件对象，可以在多个进程之间共享。若多个内存区对象映射了同一个文件，则它们也引用同一个段对象，这可以从 MmCreateSection 函数的逻辑看出来。

段对象本身是在换页内存池中分配的，它包含了段内页面数量以及字节数、用于描述此段内页面映射的原型 PTE 阵列等信息，并且有一个指针指向一个控制区对象。控制区对象（CONTROL_AREA 结构）是在非换页内存池中分配的，它有一个指针指向段对象，所以，段对象和控制区对象是互指的。控制区对象的职责是维护 I/O 操作所必需的信息，它包含有一个文件对象指针成员。计数成员 NumberOfMappedViews 反映了与之关联的内存区对象被映射了多少次。与控制区对象同在一块内存中的是一个或多个子内存区对象（SUBSECTION），它们记录了从原型 PTE 到磁盘上逻辑扇区之间的关联信息，子内存区对象构成了一条链，并且有一个指针指回到控制区对象（图 4.21 中没有画出）。

文件对象是 Windows I/O 和存储体系中的一个核心对象，也是内存管理子系统与 I/O 子系统打交道的重要途径，而内存区对象恰好集成了这两个子系统中最重要的能力。在文件对象中，有一个指向 SECTION_OBJECT_POINTERS 数据结构的指针成员

SectionObjectPointer，而 SECTION_OBJECT_POINTERS 数据结构包含两个指针成员：DataSectionObject 和 ImageSectionObject，它们又分别指回到了控制区对象。所以，我们可以看到，控制区对象与文件对象实际上也有一个互指关系。DataSectionObject 和 ImageSectionObject 指针分别用于数据文件和可执行映像文件的映射。

在上节介绍虚拟内存描述符时，我们曾经看到 MMVAD 有一个指向控制区的指针成员 ControlArea，但是，在 NtAllocateVirtualMemory 中申请的 MMVAD 是 MMVAD_SHORT 类型，它实际上并不包含 ControlArea 成员，其 VadFlags.PrivateMemory 标志指明了这一点。然而，凡是跟内存区对象关联的 MMVAD 对象都有 ControlArea 指针域，以便将两者关联起来，在本节后面介绍内存区视图映射时我们会看到这种关联关系。在 SECTION 数据结构中有一个 MMADDRESS_NODE 类型的数据成员 Address，它与控制区所关联的 VAD 节点并没有关系，相反，它参与到另一棵由全局变量 MmSectionBasedRoot 定义的 AVL 树中，该 AVL 树把所有 SEC_BASED 类型的内存区对象组织起来。由于这些内存区对象可供多个进程共享，又要求每个进程在映射内存区对象时必须映射到相同的虚拟地址上，所以，这样的全局设计也是合理的。

以上这些数据结构构成了内存区对象工作的整体框架，在 WRK 中可以看到它们的定义和用法。正如本节开头所提到的，内存区对象代表了物理存储资源，对于不同类型的内存区对象，Windows 使用不同的方法来创建和映射内存区对象，以供应用程序使用。下面我们针对不同类型的内存区对象来分析 Windows 内核中的实现方法。

最简单的情形是由页面文件支撑的内存区对象。在 MmCreateSection 函数中，若 FileHandle 和 FileObject 参数均为空，则会调用 MiCreatePagingFileMap 函数来创建一个段对象。然后利用此段对象中的信息来填充 SECTION 对象的域。MiCreatePagingFileMap 函数位于 base\ntos\mm\creasect.c 文件的 4 670~4 978 行，其函数原型如下：

```
NTSTATUS
MiCreatePagingFileMap (
   OUT PSEGMENT *Segment,
   IN PUINT64 MaximumSize,
   IN MM_PROTECTION_MASK ProtectionMask,
   IN ULONG AllocationAttributes
   );
```

该函数的参数的含义比较明显：Segment 是输出参数，一旦创建段对象成功，则该参数包含了段对象的地址；MaximumSize 是待创建段对象的大小；ProtectionMask 指定了初始的页面保护标志；AllocationAttributes 参数的含义与 MmCreateSection 的 AllocationAttributes 参数相同，但在此函数环境中，只有 SEC_BASED、SEC_RESERVE、SEC_COMMIT 和 SEC_LARGE_PAGES 才有意义。

MiCreatePagingFileMap 函数首先确定段对象所需要的换页内存池中的内存大小，包括 SEGMENT 对象本身的大小和原型 PTE 阵列所占用的空间。然后调用 ExAllocatePoolWithTag 函数申请一块换页内存池中的内存。MiCreatePagingFileMap 函数又申请一个控制区 CONTROL_AREA 对象，也包括一个子内存区 SUBSECTION 对象，这是在非换页内存池中申请的。然后初始化控制区对象和子内存区中的成员。接下来初始化段对象中的成员，尤其是原型 PTE。如果是 SEC_LARGE_PAGES 类型的内存区，则直接映射好大页面物理内存；如果是 SEC_COMMIT 类型的内存区，则设置好原型 PTE 中的保护掩码（即 ProtectionMask 参数）；对于其他类型的内存区，所有的原型 PTE 初始化为零。

因此，页面文件支撑的内存区对象相对简单，不涉及文件对象。MmCreateSection 函数只是创建了一个内存区对象，以及相关联的段对象和控制区对象。应用程序获得了内存区对象的句柄，但还不能访问内存区对象中的数据。即使已经设定了提交标志，也还不能访问。为了使用内存区对象中的数据，应用程序必须映射一个视图。Windows 内核的系统服务例程是 NtMapViewOfSection 函数，对应于内存管理器中的函数是 MmMapViewOfSection。NtMapViewOfSection 和 MmMapViewOfSection 函数的代码都在 base\ntos\mm\mapview.c 文件中。MmMapViewOfSection 函数根据内存区对象的段对象所指的控制区对象中的标志信息来确定内存区的类型，如果 PhysicalMemory 位为 1，则调用 MiMapViewOfPhysicalSection 函数来映射内存区；如果 Image 位为 1，则调用 MiMapViewOfImageSection 函数来映射内存区；否则，说明这是一个数据区，则调用 MiMapViewOfDataSection 函数来映射内存区。对于页面文件支撑的内存区，或数据文件内存区，都是通过 MiMapViewOfDataSection 函数来映射的。因此，我们现在来看 MiMapViewOfDataSection 函数的逻辑。

MiMapViewOfDataSection 函数的代码见 base\ntos\mm\mapview.c 文件的 3 503~4 157 行，其原型如下：

```
NTSTATUS
MiMapViewOfDataSection (
    IN PCONTROL_AREA ControlArea,
    IN PEPROCESS Process,
    IN PVOID *CapturedBase,
    IN PLARGE_INTEGER SectionOffset,
    IN PSIZE_T CapturedViewSize,
    IN PSECTION Section,
    IN SECTION_INHERIT InheritDisposition,
    IN ULONG ProtectionMask,
    IN SIZE_T CommitSize,
    IN ULONG_PTR ZeroBits,
    IN ULONG AllocationType
    );
```

MiMapViewOfDataSection 函数的任务是把内存区对象中描述的存储资源映射到指定进程的地址空间中。其中 ControlArea 参数代表了内存区对象中段对象所指的控制区对象，其他的参数基本上来自于 MmMapViewOfSection，这里不一一介绍。

MiMapViewOfDataSection 函数首先确定要映射的内存区对象在目标进程中的地址范围，它调用 MiFindEmptyAddressRangeDown 或 MiFindEmptyAddressRange 函数，在进程的虚拟地址空间中找到一段能包含指定大小的地址范围。然后申请一个 VAD 对象，并初始化此 VAD 对象的数据成员，让它的 ControlArea 指向内存区的控制区对象。然后记录各种系统开销，并且把 VAD 节点插入到目标进程的 VAD 树中。最后，把所有要提交部分的地址范围的 PTE 设置为段对象的 SegmentPteTemplate 值。对于 MiCreatePagingFileMap 函数创建的页面文件支撑的内存区对象，此原型 PTE 中仅仅设置了保护属性。

经过 MiMapViewOfDataSection 函数映射了虚拟地址范围以后，应用程序可以直接访问此地址范围中的数据。内存管理器会通过页面交换机制来灵活地为应用程序提供所需要的页面和数据。关于页面交换机制，请参考 4.4 节的内容。

接下来看一看可执行映像文件的内存区对象。对于这一类内存区对象，在 MmCreateSection 函数中，其 AllocationAttributes 参数包含 SEC_IMAGE 标志，并且 FileHandle 参数指向可执行文件的句柄。如果文件对象还没有控制区对象和段对象，那么，MmCreateSection 调用 MiInsertImageSectionObject 函数把一个新建的控制区对象插入到映像文件对象中，并调用 MiCreateImageFileMap 函数创建一个新的段对象。MiInsertImageSectionObject 函数位于 base\ntos\mm\creasect.c 文件的 5 417~5 528 行，它把参数中指定的控制区对象设置到文件对象的内存区对象指针（SectionObjectPointer 成员）的 ImageSectionObject 成员中。注意，MmCreateSection 函数中创建的控制区对象并非真正用于映射文件对象的控制区对象，而仅仅是个占位的控制区对象；真正能理解映像文件因而可用于映射内存区对象的控制区是由 MiCreateImageFileMap 函数创建的。当段对象被创建以后，MmCreateSection 将段对象所指的控制区对象设置到文件对象的内存区对象指针的 ImageSectionObject 成员中，而原来由 MmCreateSection 创建的控制区对象则被释放掉。因此，内存区对象、文件对象、段对象和控制区对象之间的关系仍然是一致的。

MiCreateImageFileMap 函数位于 base\ntos\mm\creasect.c 文件的 1 677~3 771 行，虽然代码量较大，但是它所做的事情很容易理解。首先从映像文件读取头信息，Windows 支持 PE 格式（参考 2.2.2 节中的介绍）的映像文件，所以，MiCreateImageFileMap 函

数按照 PE 格式来解析映像文件头的信息。然后，在非换页内存池中创建一个控制区对象，在换页内存池中创建一个段对象。控制区对象附带多个子内存区对象，子内存区对象的数量与映像文件中区域（SECTION）的数量相等；而段对象附带的原型 PTE 阵列能描述整个映像文件大小。然后，MiCreateImageFileMap 利用文件头中的信息填充控制区和段对象中各个成员。通过阅读这部分代码，我们可以了解控制区和段对象中各个成员在映像文件的情形下是如何被初始化的。值得一提的是，段对象中的子内存区，正好对应于映像文件中的各个区域，它们有自己的保护属性、原型 PTE、在磁盘上的位置等信息。

同样地，对于映像文件的内存区对象，也必须经过 MmMapViewOfSection 函数映射到一个进程地址空间中以后，该进程才可以访问映像文件中的内容。在这种情况下，MmMapViewOfSection 调用 MiMapViewOfImageSection 函数来映射内存区。因此，我们来看一下 MiMapViewOfImageSection 函数，位于 base\ntos\mm\mapview.c 文件的 1 903~2 745 行，其原型如下：

```
NTSTATUS
MiMapViewOfImageSection (
    IN PCONTROL_AREA ControlArea,
    IN PEPROCESS Process,
    IN PVOID *CapturedBase,
    IN PLARGE_INTEGER SectionOffset,
    IN PSIZE_T CapturedViewSize,
    IN PSECTION Section,
    IN SECTION_INHERIT InheritDisposition,
    IN ULONG_PTR ZeroBits,
    IN ULONG AllocationType,
    IN SIZE_T ImageCommitment
    );
```

MiMapViewOfImageSection 函数的任务是把映像文件中的各个区域映射到指定进程的虚拟地址空间中。其中 ControlArea 参数代表了内存区对象中段对象所指的控制区对象，Section 代表了内存区对象。绝大多数参数来自于 MmMapViewOfSection 函数。

MiMapViewOfImageSection 函数首先确定目标虚拟地址，有三种可能：由参数 CapturedBase 指定；若 CapturedBase 参数为空，则由内存区对象的段对象的 BasedAddress 成员决定（在 MiCreateImageFileMap 函数中该成员已被初始化为映像文件的基地址）；若此 BasedAddress 开始的地址范围在进程的 VAD 树中查到已被占用，则调用 MiFindEmptyAddressRangeDown 或 MiFindEmptyAddressRange 找到一个大小足够的空闲地址范围。然后创建一个 VAD 节点对象，并填充其中的值，包括让它的 ControlArea 成员指向参数 ControlArea 所指的控制区对象。因此，图 4.21 中的控制区对象和 VAD 对象

的关系建立起来。然后分别调用 MiInsertVadCharges 和 MiInsertVad 函数把 VAD 对象插入到进程的 VAD 树中，并记录相应的开销。最后，如果采用了大页面映射，则还需要回调到小页面视图方式，以便将映像文件的数据拷贝到大页面视图中；如果映像文件在软盘上，则将页面转移到页面文件中。

我们再来看一下基于数据文件的内存区对象。对于这种类型的内存区对象，MmCreateSection 调用 MiCreateDataFileMap 函数来创建段对象。MmCreateSection 在调用该函数以前先创建了一个控制区对象，存放在文件对象的内存区对象指针（SectionObjectPointer）的 DataSectionObject 成员中。MiCreateDataFileMap 函数位于 base\ntos\mm\creasect.c 文件的 4 198~4 590 行，其代码比 MiCreateImageFileMap 函数要相对简单很多，毕竟它不需要解析文件内部的格式信息。它首先确定内存区的大小，如果文件的大小比参数中指定的最大尺寸要小，则要求内存区的页面必须是可读写的，并且文件大小可以扩充到指定的最大尺寸。

然后 MiCreateDataFileMap 函数在换页内存池中申请一个段对象（MAPPED_FILE_SEGMENT 类型），并填充其中的成员。注意，这里的段对象并不直接包含原型 PTE 阵列（以后，当相应的内存区对象被映射时，才会构造必要的原型 PTE）。另外，控制区对象的子内存区也被从一个扩展到多个。MiCreateDataFileMap 按照全局变量 MmAllocationFragment 指定的大小把原文件打散成一组 chunk。这里 MmAllocationFragment 的值默认设置成 64 KB、32 KB 或 16 KB，取决于系统中物理内存的数量，这是在 MmInitSystem 中设定的。每个 chunk 对应一个子内存区。相比于映像文件的做法，这是一种优化，MiCreateDataFileMap 人为地把描述一个映射文件的子内存区打散成多个子内存区，并且不立即分配原型 PTE，而只是当映射视图时才向换页内存池申请所映射部分的页面的原型 PTE。这样做，一方面可以避免当创建大文件的内存区对象时过度地消耗换页内存池的内存空间，另一方面，对于特大的文件，即使换页内存池的所有可用空间容纳不下该文件所有页面的原型 PTE，MiCreateDataFileMap 也仍然可以为它创建一个内存区对象，以后只要每次映射此文件的视图时，视图的大小不超过换页内存池所能承受的原型 PTE 数量即可。

最后，MiCreateDataFileMap 函数初始化控制区对象和段对象，填充好控制区对象的子内存区链表。然后，函数返回。

数据文件内存区对象也是通过 MiMapViewOfDataSection 函数来映射的。如前所述，MiMapViewOfDataSection 函数的任务是把内存区对象中描述的段映射到进程的虚拟地址

空间中，如果虚拟地址范围尚未确定，则尝试在进程的 VAD 树中找到合适的地址范围，然后利用控制区中的信息建立内存区映射，其逻辑过程与页面支撑的内存区对象的映射过程相同，这里不再赘述。

除了以上介绍的这三种内存区对象以外，还有一个名为“\\Device\\PhysicalMemory”的特殊内存区对象，它是在系统初始化阶段由 MiSectionInitialization 函数（位于 base\ntos\mm\sectsup.c 文件中）直接创建，并插入到对象管理器的名字空间中的（注意，这里并没有调用 MmCreateSection 函数）。此内存区对象可用于直接把物理内存映射到进程地址空间中。其控制区对象的 PhysicalMemory 标志为 1，MmMapViewOfSection 为它调用 MiMapViewOfPhysicalSection 函数来映射虚拟地址范围。

同样地，在 MiMapViewOfPhysicalSection 函数中，待映射的范围是由 SectionOffset 和 CapturedViewSize 两个参数决定的。MiMapViewOfPhysicalSection 函数确定了进程的虚拟地址范围以后，其映射过程并不复杂，它只是把虚拟地址范围中的页目录项和页表项正确地设置好，并维护好进程内部的数据结构。特别值得一提的是，进程 EPROCESS 除了维护进程地址空间的 VAD 树以外，还维护了物理地址空间中通过这种方式映射的物理内存视图而构成的另一棵 VAD 树，其树根是 EPROCESS 对象中的 PhysicalVadRoot 成员。此树并不总存在，只有当它第一次被引用的时候才创建出来，并插入物理内存视图节点。应用程序通过 NtAllocateVirtualMemory 系统服务也可以映射物理内存。

最后，当应用程序不再使用内存区对象的映射视图时，它们通过 NtUnmapViewOfSection 系统服务来解除映射。NtUnmapViewOfSection 调用 MiUnmapViewOfSection 函数来解除内存区对象的映射，这两个函数的代码均位于 base\ntos\mm\umapview.c 文件中。与建立映射视图相比，解除映射要容易得多。MiUnmapViewOfSection 函数根据给定的基地址，找到相应的 VAD 节点，并从 VAD 树中移除此节点。然后调用 MiRemoveMappedView 函数来真正解除映射。MiRemoveMappedView 函数针对各种情形归还系统资源，甚至删除控制区对象和段对象，并根据需要调用 MiDeleteVirtualAddresses 函数以删除指定的虚拟地址范围，包括其占用的 PTE 资源。

综上所述，本节中解释的内存区对象的类型和代码结构如图 4.22 所示，其中实线箭头表示调用关系。在创建内存区对象和映射视图这两个阶段分别有一组函数来处理其关键逻辑；解除映射是由 NtUnmapViewOfSection 函数来完成的；而内存区对象的生命周期则是由对象管理器通过其中的引用计数来控制的。

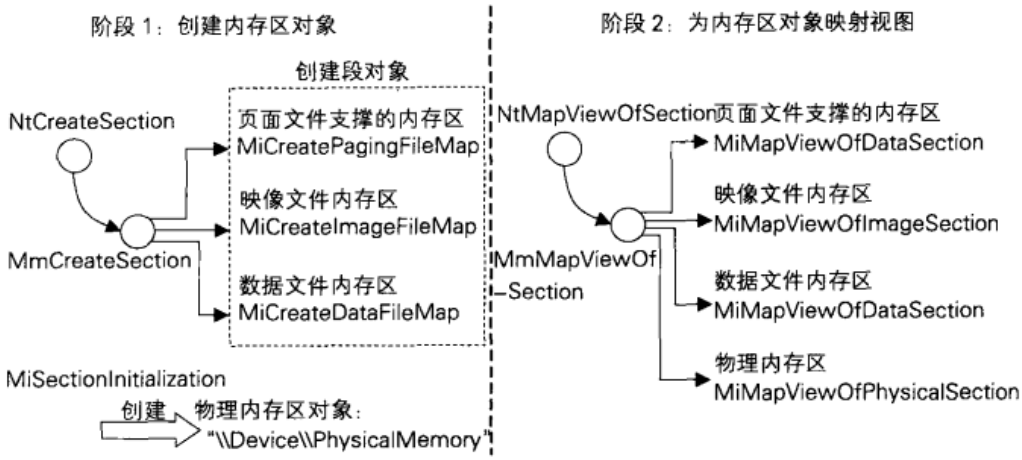


图 4.22 内存区对象的主要代码结构

4.4 内存页面交换

当 Intel x86 处理器在执行一个执行流过程中需要翻译一个虚拟地址引用时，如果该地址的页表项（PTE）中的有效位为 0，则处理器会引发一个异常，也称为页面错误（page fault）。Windows 内核的陷阱处理器（trap handler）会把这一异常交给内存管理器的页面错误处理例程（page fault handler），由它来解决页面无效的问题。一般来说，只要这个页面错误是合理的（比如，页面已被换出到外存），则页面错误处理例程将试图分配一个物理页面，并设置好 PTE，从而消除虚拟地址引用错误，使得该地址引用能被正确地翻译成恰当的物理页面。由于异常是硬件触发的，而页面错误处理例程是操作系统提供的组件，所以这一页面分配并重映射的过程对于原始的指令流是完全透明的，最多只是时间上稍有停顿。

如果一个 PTE 已经正确地指向一个物理页面，则页面错误不会发生，虚拟地址翻译可以快速地进行。然而，物理内存毕竟是有限的资源，随着越来越多的进程使用越来越多的页面，最终系统的可用物理内存可能会消耗光。为了避免因内存消耗光而导致应用程序或系统的正常功能无法执行，现代操作系统提供了页面交换的能力，即把有些正在使用的页面中的内容存放到磁盘上，从而腾出这些物理页面以供他用。以后，当这些被换出到磁盘上的页面再次被引用时，页面错误处理例程再把它们换回到内存中。4.1.3 节中提到的一些页面替换算法正是针对如何有效地替换页面而给出的一系列方案。

Windows 系统支持一个或多个页面文件（每个磁盘分区最多一个页面文件），用来存放被交换页面的内容。页面文件可被看做是物理内存的一种延伸。对于页面文件中的内容而言，

页面文件与物理内存是同等的，只不过受页面调度的影响而在不同时刻位于不同的物理位置。上一节介绍的页面文件支撑的内存区对象，正是在这一概念下形成的一种内存区。

本节将介绍在 Intel x86 体系结构下 Windows 的页面交换机制。首先讨论 Intel x86 硬件 PTE 机制，然后介绍 Windows 中引入的无效 PTE 和原型 PTE，有了这些基础以后，接下来讨论页面错误处理例程的实现流程，最后介绍 Windows 中写时复制的实现机理。

4.4.1 Intel x86 中的 PTE

4.1.1 节介绍了 Intel x86 体系结构中 32 位虚拟地址是如何被翻译成物理地址的，参见图 4.3。现在我们来看一看 PTE 中各个位的确切定义，如图 4.23 所示。PDE（页目录项）与 PTE（页表项）使用相同的格式，唯一的区别在于，第 7 位即 L 位仅对 PDE 有效。PDE 中 L 位为 1 说明这是一个大页面 PDE，即该 PDE 的虚拟地址范围使用大页面映射。

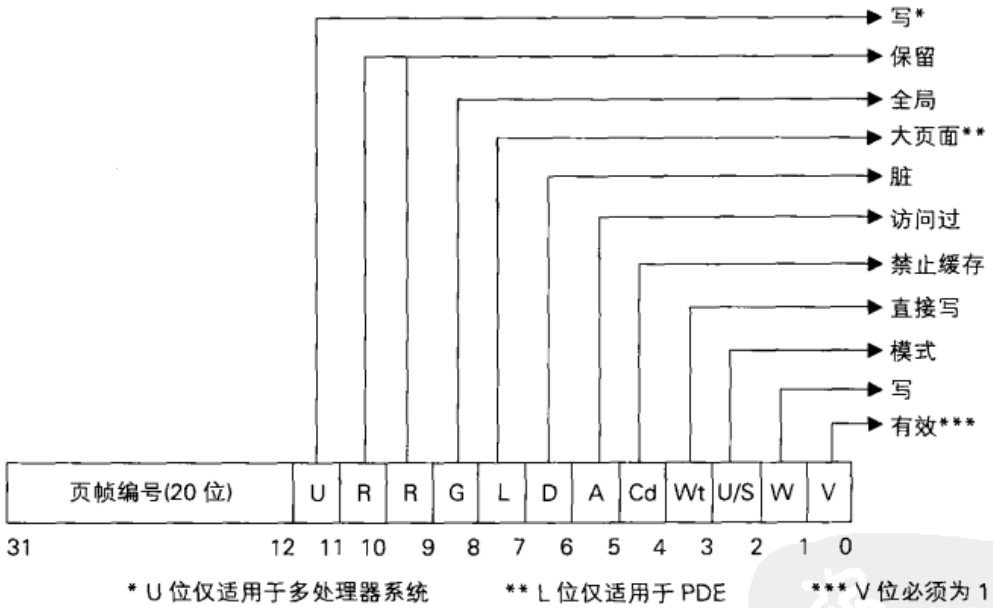


图 4.23 Intel x86 的硬件 PTE

当处理器在翻译一个虚拟地址时，如果在页表页面中查到的 32 位 PTE，其 V 位为 1，则说明这是一个有效的 PTE，于是按照图 4.23 中所示的那样来解释，这里页帧编号指向对应的物理页面的地址。因为物理页面地址是页面对齐的，所以 20 位足以描述 32 位物理地址空间中的页面。在 Intel x86 处理器中，PDE 也按照同样的格式来解释。如果 V 位为 0，则处理器引发页面错误，然后由系统软件来修正。通常页面错误处理例程会分配一个物理页面，然后根据需要把页面内容从磁盘读入物理页面，或者直接将页面清零。最后，设置好 PTE 并返回，让处理器按修正后的 PTE 访问内存。PTE 中各个状态位的解释如表 4.3 所示。

表 4.3 PTE 状态位的含义

位	名称	说明
V	有效	指明此虚拟地址是否映射到一个物理页面
W	写	在单处理器上，指明可读写或只读；在多处理器上，指明是否可写
U/S	模式	（User/Supervisor）内核模式代码才可访问（0）或用户模式代码可访问（1）
Wt	直接写	直接写内存（write-through）或写回缓存（write-back）策略
Cd	禁止缓存	该页面禁止缓存
A	访问过	页面已经被访问过（读或写）
D	脏	页面已经被写过
L	大页面	这是一个大页面 PDE，该位对于 PTE 无用
G	全局	此 PTE 适用于所有的进程
U	写	由软件控制的位，此位仅适用于多处理器系统，指明了可读写或只读

当一个页面第一次被访问时，不管是读还是写操作，A 位都会被置位，如果是写操作，则 D 位也会被置位。D 位和 A 位由硬件自动置位，但需要由软件来清除。W 位提供了一种访问控制能力，若它被置位，则说明该页面可写，否则是只读的。试图往一个 W 位为 0 的页面执行写操作，会引发一个页面错误，系统的页面错误处理例程可以决定该操作是否合法。这一能力可以用来实现像“写时复制（copy-on-write）”这样的特性。另外，U/S 位从另一个角度提供了访问控制能力，如果它被置位，则说明只有内核模式代码才可以访问，否则用户模式代码也可以访问。U 位是由软件实现的一个写位，用于多处理器系统，指明一个页面已经被另一个处理器写过了。

4.1.1 节曾经提到，地址转译快查缓冲区（TLB，Translation Look-aside Buffer）包含了最近使用过的虚拟页面与物理页面之间的映射关系，以便于快速转译虚拟地址，从而避免在地址转译时需要多次内存访问。PTE 中的 G 位若被置位，则说明当系统切换进程时该 PTE 仍然有效，而不必在 TLB 中被置为无效；相反，若 G 位为 0，则说明在新进程中此 PTE 项不再有效，因而在 TLB 中它被置为无效。

以上显示的硬件 PTE 完全由 Intel x86 处理器来定义，WRK 中也给出了对应的 C 语言定义（见 base\ntos\mm\i386\mi386.h 文件），如下所示。其中上文提到的模式位，即 U/S 位，在代码中称为 Owner（所有者）。

```
typedef struct _MMPTE_HARDWARE {
   ULONG Valid : 1;
#if defined(NT_UP)
   ULONG Write : 1;      // UP version
#else
```

```
    ULONG Writable : 1;    // changed for MP version
#endif
    ULONG Owner : 1;
    ULONG WriteThrough : 1;
    ULONG CacheDisable : 1;
    ULONG Accessed : 1;
    ULONG Dirty : 1;
    ULONG LargePage : 1;
    ULONG Global : 1;
    ULONG CopyOnWrite : 1; // software field
    ULONG Prototype : 1;   // software field
#if defined(NT_UP)
    ULONG reserved : 1;    // software field
#else
    ULONG Write : 1;       // software field - MP change
#endif
    ULONG PageFrameNumber : 20;
} MMPTE_HARDWARE, *PMMPTE_HARDWARE;
```

4.4.2 软件 PTE：无效 PTE 和原型 PTE

上一小节我们看到了一个有效的 PTE 的定义，现在我们来看当 PTE 的 V 位为 0 时的情形。此时，处理器引发页面错误异常，该异常的陷阱处理器将把控制权交给页面错误处理例程，所以，PTE 其他位的含义由操作系统来解释。正是由于有了这一层灵活性，Windows 才可以在这样的 PTE 中承载一些信息，以便实现页面管理功能。这种 PTE 称为无效 PTE，这一节我们来讨论无效 PTE 以及另一种相关联的 PTE——原型 PTE。

在 Windows 中，无效 PTE 有四种情形，如图 4.24 所示。这些情形说明了该 PTE 所对应页面的去向或状态，可以指示页面错误处理例程如何正确地处理和解决该错误。在第一种情形中，如图 4.24(a)所示，第 10 和 11 位为 0，第 1~4 位指示了在哪个页面文件中，第 12~31 位共 20 位表示了该页面在页面文件中的偏移量，以页面大小（4 096 字节）为单位。第二种情形，如图 4.24(b)所示，是第一种情形的特例，若 1~4 位和 12~31 位全为 0，则表明这是一个待分配的页面，它需要一个填满零的页面。在第三种情形中，如图 4.24(c)所示，第 10 位为 1，说明这个页面尚在内存中，但正在换出过程中，位于系统的某个物理页面链表中。此时第 12~31 位指示了物理页帧编号，根据 PFN 数据库可以定位到该页面的实际状态。本章后面 4.5.1 节将进一步介绍 PFN 数据库。最后一种情形，如图 4.24(d)所示，是一个全零的 PTE，此时该页面的状态无法确定，甚至该 PTE 对应的虚拟地址是否已被提交也不能确定，所以页面错误处理例程应检查 VAD 树中对应的节点，以确定该页面的状态。第二种情形与第四种情形不会混淆，因为对于一个正常的页面，其保护位不会全零。

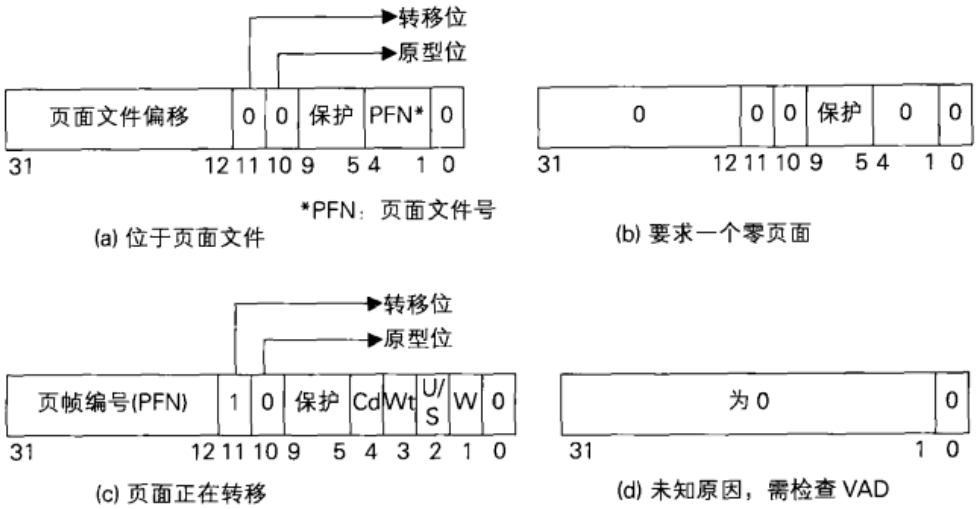


图 4.24 无效 PTE 的四种情形

无效 PTE 中承载的信息很重要，因为这是页面错误处理例程的切入点，下一节介绍页面错误处理过程时我们将会看到这些无效 PTE 是如何被解释和处理的。以下是 WRK 中有关无效 PTE 的两个数据结构定义（见 base\ntos\mm\i386\mi386.h 文件）：

```
typedef struct _MMPTE_SOFTWARE {
   ULONG Valid : 1;
   ULONG PageFileLow : 4;
   ULONG Protection : 5;
   ULONG Prototype : 1;
   ULONG Transition : 1;
   ULONG PageFileHigh : 20;
} MMPTE_SOFTWARE;

typedef struct _MMPTE_TRANSITION {
   ULONG Valid : 1;
   ULONG Write : 1;
   ULONG Owner : 1;
   ULONG WriteThrough : 1;
   ULONG CacheDisable : 1;
   ULONG Protection : 5;
   ULONG Prototype : 1;
   ULONG Transition : 1;
   ULONG PageFrameNumber : 20;
} MMPTE_TRANSITION;
```

以上介绍的无效 PTE 是真正位于页目录和页表页面中的 32 位数据项，处理器检测到它们的无效位，从而通过页面错误异常交给内存管理器来处理。在 Windows 中，还有一种称为原型 PTE 的数据结构，4.3.4 节在介绍内存区对象时曾经多次提到原型 PTE。内存区的段对象中可能包含了其页面的原型 PTE 阵列，当进程第一次引用一个已被映射的内存区对象的视图中的页面时，内存管理器利用原型 PTE 中的信息来填充页表中实际的 PTE。段对象中的原型 PTE 与段对象一起创建，或者在映射内存区对象时根据需要而创建。

原型 PTE 并非一成不变，它们也会随着内存区对象中页面的变化而处于不同的状态。原型 PTE 可以描述 6 种状态的页面：

- 有效。对应的页面位于物理内存中。对于内存区对象中的页面，若有一个进程访问了该页面的数据，则该页面将被移到内存中，于是原型 PTE 就变成了一个有效的 PTE，其格式如 4.4.1 节所述。
- 位于页面文件中。对应的页面位于页面文件中，其格式如图 4.24(a)所示。
- 位于映射文件中。对应的页面位于映射文件中，其格式也如图 4.24(a)所示，但是第 12~31 位并非指页面文件中的偏移量，而是指映射文件中的偏移量。
- 要求零页面。页面尚未分配，待下次访问时请求一个零页面。其格式如图 4.24(b)所示。
- 转移。页面在物理内存中，但已被转移到某个物理页面链表中，可通过查询 PFN 数据库获知确切状况。其格式如图 4.24(c)所示。
- 页面已修改但不写出（modified-no-write）。页面在物理内存中，已经被修改过了，但修改页面写出器不会把它写回到磁盘上。其格式也如图 4.24(c)所示。这种页面的用法我们将在 4.5 节介绍物理内存管理时再进一步介绍。

原型 PTE 只是在内存区的段对象中记录了页面的状态，它们并不参与处理器的地址转译，我们可以把原型 PTE 理解为一个页面在其生命周期中全程有效的状态信息，而系统为处理器地址转译而准备的 PTE 只不过是此生命周期中一部分时间有效的状态信息。当内存区对象中的一个页面从有效变成无效时，它的硬件 PTE 将直接指向原型 PTE，其格式如图 4.25 所示。

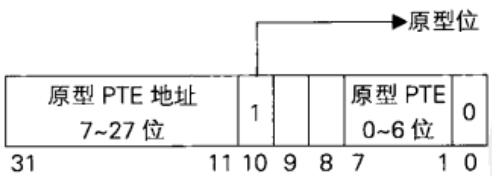


图 4.25 指向原型 PTE 的无效 PTE

这里用 28 位来描述一个原型 PTE 的地址，因为原型 PTE 是 4 字节对齐的，也就是说原型 PTE 的地址的最后两位一定是 0，所以，这里 28 位相当于描述了 30 位的空间。另外，原型 PTE 总是在系统换页内存池分配的，所以，无效 PTE 中指定的原型 PTE 是指它相对于系统换页内存池的偏移。下面的 C 宏 MiPteToProto 说明了这一转换：

```
#define MiPteToProto(lpte) (PMMPTE)((PMMPTE)(((((lpte)->u.Long) >> 11) << 9) + \
            (((((lpte)->u.Long)) << 24) >> 23) + \
            MmProtopte_Base))
```

WRK 中定义的原型 PTE 如下所示（也位于 mi386.h 文件中）：

```
typedef struct _MMPTE_PROTOTYPE {
   ULONG Valid : 1;
   ULONG ProtoAddressLow : 7;
   ULONG ReadOnly : 1;  // if set allow read only access.
   ULONG WhichPool : 1;
   ULONG Prototype : 1;
   ULONG ProtoAddressHigh : 21;
} MMPTE_PROTOTYPE;
```

以上介绍了各种类型的 PTE，无论是硬件 PTE 还是软件 PTE，均是 32 位长，Windows 把所有这些类型的 PTE 组织在了一个 C 语言联合（union）中，因而在代码中可以方便地取用任何一种类型的 PTE，MMPTE 的定义如下（同样位于 mi386.h 文件中）：

```
typedef struct _MMPTE {
   union  {
      ULONG Long;
      HARDWARE_PTE Flush;
      MMPTE_HARDWARE Hard;
      MMPTE_PROTOTYPE Proto;
      MMPTE_SOFTWARE Soft;
      MMPTE_TRANSITION Trans;
      MMPTE_SUBSECTION Subsect;
      MMPTE_LIST List;
      } u;
} MMPTE;

typedef MMPTE *PMMPTE;
```

这里 HARDWARE_PTE 和 MMPTE_SUBSECTION 两种类型我们忽略不考虑。而 MMPTE_LIST 的用意是让 PTE 构成一个链表，在 4.2.3 节介绍系统 PTE 区域的管理时，我们曾经看到过它的用法。这里不再进一步介绍。

在这一节最后，我们通过一个例子来看一看原型 PTE 在内存区对象中的用法。如图 4.26 所示，假设两个进程 A 和 B 共享一个内存区对象，它们分别拥有该内存区的一个视图，其虚拟地址可能相同，也可能不相同。内存区中的页面 P1 已经被两个进程访问过了，所以，该页面对应的原型 PTE 是一个有效的 PTE，指向物理内存中的页面；P1 在两个进程中的 PTE 也是有效的，指向物理页面。页面 P2 位于页面文件中，所以，它对应的原型 PTE 是无效的，但指向页面文件中的页面。两个进程中对应于页面 P2 的 PTE 也是无效的，但它们指向该页面的原型 PTE。图 4.26 显示了这些 PTE 之间的关系。这时候，如果有一个进程，不妨假设进程 A，访问了页面 P2，那么，此时内存管理器会为 P2 分配一个物理页面，并且让页面 P2 在进程 A 中的 PTE 指向该物理页面，而且 P2 在段对象中的原型 PTE 也指向该物理页面。以后，当进程 B 要访问页面 P2 时，页面错误仍然发生，但是页面错误处理例程会发现这是一个指向原型 PTE 的无效 PTE，它检查原型 PTE，知道页面 P2 已

经在物理内存中了，于是让 P2 在进程 B 中的 PTE 也指向此物理页面，从而两个进程访问的是同一个页面 P2。由此可以看出，原型 PTE 在内存页面的跨进程共享中起了重要的作用。另外，共享的物理页面在 PFN 数据库中有一个共享计数，只有当共享计数为 0 时，它才有可能被换出到外存中。

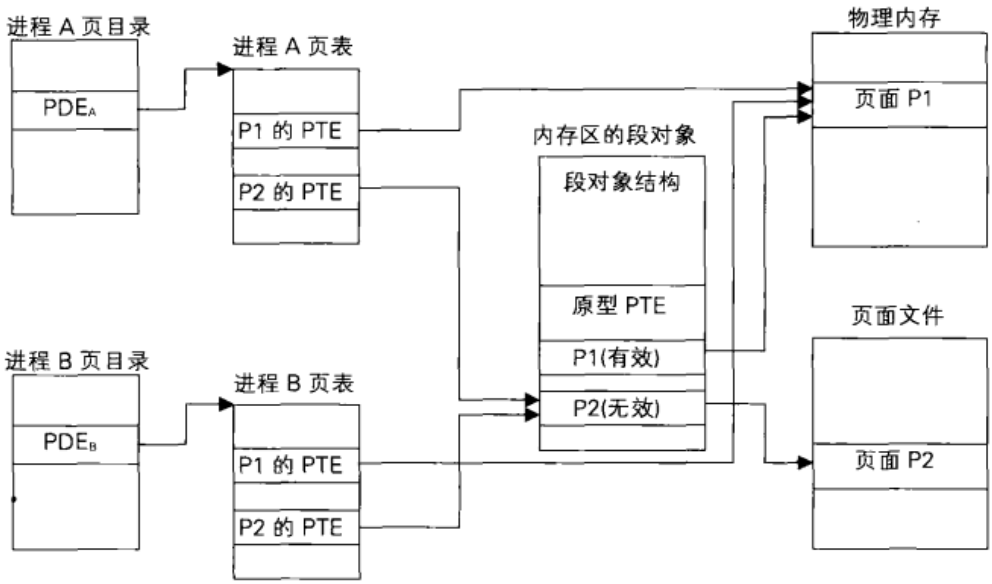


图 4.26 共享内存区中原型 PTE 的用法

4.4.3 页面错误处理

前面两小节讨论了 PTE 的类型，以及每一种 PTE 的格式及其含义。Intel x86 处理器在转译一个地址时，一旦看到无效的 PDE 或 PTE，就会触发一个页面错误，即 14 号系统异常，并将控制权交给系统设定的相应陷阱处理器。页面错误异常的陷阱处理器是 _KiTrap0E 函数，这是位于 base\ntos\ke\i386\trap.asm 文件中的一个汇编语言函数。它首先构造一个陷阱帧，然后把控制权交给内存管理器的 MmAccessFault 函数。如果 MmAccessFault 函数能解决此页面错误，则陷阱处理器正常返回；若不能解决此页面错误，譬如访问违例（access violation，例如写一个只读页面），则_KiTrap0E 函数判断此页面错误是否发生在一些特殊的点上，并进行相应的处理，否则交给 CommonDispatchException 或 CommonDispatchException2Args 函数，由 Windows 统一的异常分发设施来处理。本节我们只关心页面错误处理例程，即 MmAccessFault 函数，如何解决一个页面错误。关于 Windows 异常分发的全面讨论，请参考 5.2.7 节。

在解读 MmAccessFault 函数以前，我们先来看一看哪些情形会导致页面错误，以及应

该如何来解决页面错误。通过上一节的介绍，我们知道当处理器转译虚拟地址时，如果 PDE 或 PTE 的 V 位为 0，则页面错误就会发生。实际上，还有其他一些情形也会招致页面错误。下面列出了可能发生的页面错误情形。

- 无效 PTE。无效 PTE 又分以下一些情形：
  - 页面位于页面文件或映射文件中。解决的办法是分配一个物理页面并从磁盘上读入内容。这种页面错误也称为“硬（hard）”页面错误，因为页面数据确实要从磁盘上读回来。
  - 访问一个尚在内存中但正在转移过程中的页面。将该页面转移回到进程工作集或系统工作集。这种页面错误也称为“软（soft）”页面错误，因为它们只是由于管理的原因而产生的，数据本身仍然在内存中。
  - 访问一个尚未提交的页面。这是访问违例错误。
  - 访问一个要求零的页面。申请一个填满零的页面，并加入到当前进程的工作集。
- 在用户模式下访问一个只有在内核模式下才允许访问的页面。返回访问违例错误。
- 写一个只读页面。返回访问违例错误。
- 写一个守护页面。返回守护页面违例错误。
- 执行的代码位于“不可执行”的页面中。返回访问违例错误。
- 写一个标记为“写时复制”的页面。复制一份属于当前进程私有的页面拷贝，并替换原来的页面。

从上面的列表可以看出，页面错误实际上分为两类：地址转译不成功；PTE 中的访问控制位检查没通过，包括模式位和读写位的检查。

MmAccessFault 函数需要判断所有这些情形，并在可能的情况下解决页面错误，然后返回处理的结果，其结果为以下四种情形之一：

- 页面错误解决成功：STATUS_SUCCESS，或者其他一些指示某些特性的返回值，包括 STATUS_PAGE_FAULT_PAGING_FILE、STATUS_PAGE_FAULT_TRANSITION、STATUS_PAGE_FAULT_DEMAND_ZERO 和 STATUS_PAGE_FAULT_COPY_ON_WRITE 等。
- 访问违例：STATUS_ACCESS_VIOLATION。
- 守护页面违例：STATUS_PAGE_FAULT_GUARD_PAGE。

- 页面换入失败：STATUS_IN_PAGE_ERROR。

下面我们来解读 MmAccessFault 函数的代码。该函数位于 base\ntos\mm\mmfault.c 文件的 50~2 081 行，其原型如下：

```
NTSTATUS
MmAccessFault (
   IN ULONG_PTR FaultStatus,
   IN PVOID VirtualAddress,
   IN KPROCESSOR_MODE PreviousMode,
   IN PVOID TrapInformation
   );
```

参数 FaultStatus 提供了发生错误时的状态信息位；VirtualAddress 包含了发生页面错误的虚拟地址；PreviousMode 参数指明了此页面错误是发生在内核模式下还是用户模式下；TrapInformation 参数是页面错误陷阱帧的地址。

MmAccessFault 函数首先处理发生在中断请求级别 APC_LEVEL 之上的页面错误（173~303 行代码）。按照 Windows 内核的设计，运行在此级别之上的代码不允许换页，否则就是一个严重错误。关于中断请求级别（IRQL），请参考 5.2.2 节。MmAccessFault 检查虚拟地址的 PDE 和 PTE，只要有一个无效，便是严重错误。然后进一步检查以下条件：执行权限是否满足（在 WRK 中针对 Intel x86 处理器实际上不检查执行权限）、是否写时复制的页面错误，以及是否在写一个没有写权限的页面。如果通过了这些检查，则没有发现错误，返回成功。

接下来 MmAccessFault 函数处理虚拟地址位于系统地址空间的页面错误（308~870 行代码）。首先，用户模式代码当然是不允许访问的。然后，检查 PDE 是否有效。若 PDE 无效，则调用 MiCheckPdeForPagedPool 函数（位于 base\ntos\mm\pagfault.c 文件），以处理当进程地址空间没有及时更新 PDE 时产生的 PDE 不一致问题，这仅仅发生在系统地址空间的换页内存区，包括系统换页内存池和会话空间。由于不同进程地址空间的页目录是独立的，所以，换页内存区的 PDE 有可能不一致，Windows 使用这种在页面错误处理例程中更新 PDE 的方法来保持换页内存区页表映射的一致性。

检查了 PDE 以后，再检查 PTE。如果 PTE 有效，则调用 MiCheckSystemPteProtection 函数检查虚拟地址是否落在系统全局设定的 PTE 映射保护区，若是，则直接由该函数处理此错误。然后 MmAccessFault 函数检查是否存在写访问和执行访问违例。若页面错误发生在系统地址空间的页表区域或超空间区域，则先调用 MiCheckPdeForPagedPool 函数以处理不同进程空间 PTE 页表不一致的问题，若该函数不能处理，则当做用户地址空间的页面错误来处理。

在 614~732 行代码，PTE 已经是有效的，于是只需检查地址访问的合法性即可。若这是一个不合法的写操作，则直接出错（KeBugCheckEx），除非这是在写会话空间中的映像文件，在这种情况下转换为页面的写时复制。然后检查执行权限。最后，设置 PTE 的脏位。如果这是会话空间的写时复制操作，则调用 MiCopyOnWrite 函数来处理。关于此函数，这里不再进一步解释，读者可以参考 base\ntos\mm\wrtfault.c 中的代码以及 4.4.4 节关于写时复制实现机理的介绍。

在 734~780 行代码，PTE 是无效的，但是它的原型位已置上，即这是一个如图 4.25 所示的指向原型 PTE 的无效 PTE。首先确保被访问的虚拟地址不是在非换页内存区域，若是则直接出错。然后通过前面介绍的 MiPteToProto 宏得到原型 PTE 的指针。

在 781~798 行代码，PTE 是无效的，原型位为 0，转移位为 0，说明在页面文件中，但若保护位也为 0，则出错。799~810 行代码处理守护页面出错的情形。812~832 行代码处理一个正在转移的页面，若写权限不满足则出错。

经过上述筛查以后，真正的页面错误处理由 MiDispatchFault 函数来完成，本节后面将解释此函数的工作流程。

处理用户地址空间或页表页面中的页面错误是从 878 行代码开始的。首先看是否需要延迟一下再处理。926~1 103 行代码是针对 3 级或以上页表的处理器而做的处理，我们只考虑 Intel x86 两级页表的情形。1 115~1 315 行代码处理 PDE 无效的错误。如果 PDE 无效，则对于零 PDE 的情形，看是否为守护页面，否则将它变成一个要求零的 PDE，即图 4.24(b)所示的情形。然后调用 MiDispatchFault 函数来解决 PDE 无效的问题。如果此函数返回以后，PDE 仍然无效，则返回错误。否则，设置 PDE 的脏位。

接下来看 PTE，如果 PTE 是有效的，则对于写操作，看它是否为“写时复制”页面，若是，则调用 MiCopyOnWrite 函数来处理；否则为访问违例。1 322~1 544 行代码有相当一部分在处理地址扩展和执行权限的检查，这里跳过这些代码。

然后检查 PTE 是否为“要求零页面”，当然要求可写，这对应于图 4.24(b)所示的情形。若是，则调用 MiResolveDemandZeroFault 函数来满足要求。此函数会根据需要申请一个零页面，以满足当前的页面错误。

接下来检查 PTE 是否为“零 PTE”，对应于图 4.24(d)所示的情形，这部分处理代码位于 1 569~1 896 行。这段代码首先调用 MiCheckVirtualAddress 函数，检查发生错误的虚拟地址的 VAD。如果 MiCheckVirtualAddress 函数能找到包含此虚拟地址的 VAD，则利用 VAD 的信息来构造一个 PTE，并返回此 PTE；若不能找到 VAD，则返回 NULL。然后，

MmAcessFault 根据 MiCheckVirtualAddress 函数返回的结果，逐一处理。如该虚拟地址的保护属性为不可访问（即 MM_NOACCESS），则检查是否是换页内存池需要扩展而导致页表页面需要更新。再检查该地址是否在一个守护页面中，若是，则调用 MiCheckForUserStackOverflow 函数。该函数检查是否该地址在线程的栈范围内，若是则扩展栈的范围，再创建一个新的守护页面，若创建失败则返回栈溢出错误。经过上述检查以后，如果 MiCheckVirtualAddress 函数返回的 PTE 指针是 NULL，则将它解析为一个“要求零页面”的 PTE，并为它分配一个页面，然后返回。排除了以上情形，则应该是一个原型 PTE，留待后面继续处理。

如果原来的 PTE 既非全零的 PTE 也非有效的 PTE，则考虑是一个原型 PTE，见代码 1 903~1 946 行。如有必要，同样调用 MiCheckVirtualAddress 函数。否则可以通过 C 宏 MiPteToProto 直接得到原型 PTE。代码 1 949~1 988 行对未确定页面保护属性的情形进一步检查。

最后，经过以上层层检查，MmAccessFault 函数调用 MiDispatchFault 来解决页面错误，本节接下来将介绍该函数的处理过程。MmAccessFault 函数的最后部分是几个不同的出口。这里不再解释。

MmAccessFault 函数体虽然庞大，但其主要工作是做各种检查，对于需要解决的页面错误，绝大多数是调用 MiDispatchFault 函数来完成的。但也有一些情形调用了 MiCopyOnWrite 或 MiResolveDemandZeroFault 函数来解决页面错误。下面我们来看 MiDispatchFault 函数，它需要更多的参数信息，以下是它的原型：

```
NTSTATUS
MiDispatchFault (
    IN ULONG_PTR FaultStatus,
    IN PVOID VirtualAddress,
    IN PMMPTE PointerPte,
    IN PMMPTE PointerProtoPte,
    IN LOGICAL RecheckAccess,
    IN PEPROCESS Process,
    IN PVOID TrapInformation,
    IN PMMVAD Vad
    );
```

MiDisptachFault 的 FaultStatus、VirtualAddress 和 TrapInformation 三个参数与 MmAccessFault 函数相同，并且 MmAccessFault 直接将这些参数传了过来。这里 PointerPte 指向出错地址的 PTE；PointerProtoPte 参数指向对应的原型 PTE，若没有原型 PTE 则该参数为空；RecheckAccess 参数指示是否需要重新检查原型 PTE 中的访问权限，这只有在进程 fork 的情形下才使用；Process 参数指向当前进程；Vad 参数指向虚拟地址所在的 VAD

节点，它纯粹是一个可选参数，即使内存区的页面错误，它也可以是 NULL。

MiDispatchFault 函数的代码位于 base\ntos\mm\pagfault.c 文件的 99~1 606 行。如果参数中指定了原型 PTE 的指针 PointerProtoPte，则调用 MiResolveProtoPteFault 函数来解决页面错误；否则，若是一个正在转移的 PTE，则调用 MiResolveTransitionFault 函数来解决页面错误，见代码 1 060~1 076 行；若是一个要求零页面的无效 PTE，则调用 MiResolveDemandZeroFault 函数来解决页面错误，见代码 1 077~1 087 行；对于驻留在页面文件中的无效 PTE，则调用 MiResolvePageFileFault 函数来解决页面错误，见代码 1 088~1 115 行。

MiDispatchFault 函数对于原型 PTE 的处理相对比较复杂，见代码 236~1 050 行。它首先考虑将出错虚拟地址开始的连续多个页面一起来处理，通过检查它们的原型 PTE，可以将这些页面聚集起来处理。如果原型 PTE 表明了页面尚在内存中，比如原型 PTE 是有效的，或者页面处于转移状态，则可以首先将原型 PTE 变成有效的，然后设置实际的 PTE，因而解决了页面错误。所以，对于这种情形，无须再调用 MiResolveProtoPteFault 函数。这部分处理代码见 549~816 行。

在继续介绍 MiDispatchFault 函数的实现代码以前，我们先来看一下上面提到的四个辅助函数所做的事情，这些函数的代码也位于 base\ntos\mm\pagfault.c 文件中，这里仅对它们的功能作一简单介绍，而不再一一解释其代码：

- MiResolveDemandZeroFault 函数，见 1 609~1 915 行代码。该函数完成的任务是，向系统要一个内存页面（通过 MiRemoveZeroPage 或 MiRemoveAnyPage 函数），然后设置好 PFN 数据库中对应于该页面的项，以及出错虚拟地址的硬件 PTE。
- MiResolveTransitionFault 函数，见 1 918~2 502 行代码。该函数完成的主要任务是，把正在转移的页面从它所在的链表中移除，并重新设置 PTE，使它变成一个有效的 PTE。链表移除是通过 MiUnlinkPageFromList 函数来完成的。该函数复杂的地方在于，当一个线程正在访问一个转移的页面时，另一个线程可能正在读这个页面，这在多线程环境下是有可能发生的。因此，MiResolveTransitionFault 函数需要处理这种可能的、冲突的页面错误。
- MiResolvePageFileFault 函数，见 2 505~2 984 行代码。虽然一次页面错误只发生在一个页面上，但是 MiResolvePageFileFault 函数尽可能地聚集多个相邻的页面，以便将它们一起从页面文件中读进来，从而提高应用程序的执行效率。然后，将出错页面的 PTE 设置为正在转移（u.Trans.Transition 标志置为 1），并且读操作正在进行（即 PFN 数据库中的 u3.e1.ReadInProgress 标志置为 1）。对于多个页面聚集的情形，MiResolvePageFileFault

函数申请了一组物理页面，用一个 MDL 来描述（4.5.3 节将介绍 MDL），同样地，它将这些聚集的页面设置为正在转移，并且读操作正在进行。该函数并不发起 I/O，而只是填充好这些信息，放在输出参数 ReadBlock 中。

- MiResolveProtoPteFault 函数，见 2 986~3 392 行代码。该函数根据原型 PTE 的状态做相应的处理，它可能已经有效了，也可能是一个“要求零页面”的 PTE 了，或者是一个映射文件 PTE，或者在转移状态，或者在页面文件中。所以，它会根据情况调用以上三个函数之一，或者调用 MiResolveMappedFileFault 函数。MiResolveMappedFileFault 函数的逻辑与 MiResolvePageFileFault 相仿。最后，它调用 MiCompleteProtoPteFault 函数完成原型 PTE 错误。MiCompleteProtoPteFault 函数的功能是把已经有效的原型 PTE 影响到真正的 PTE 中。

现在我们回到 MiDispatchFault 函数，前面提到了它调用以上这些辅助函数来处理页面错误。之后，从 1 122 行代码开始，它又根据情况做后处理。如果辅助函数返回成功了，那么 MiDispatchFault 函数的后处理当然比较简单，见 1 126~1 159 行代码。如果辅助函数表明需要发起一个换页 I/O，则根据返回参数中的指示，调用 IoPageRead 发出 I/O 请求，然后等待 I/O 操作，待 I/O 完成后，它还需要清除 PFN 数据库中的读操作状态信息。若 I/O 出错，则需要把置为转移状态的页面放到空闲链表中。若 I/O 成功，那么，由于页面仍在转移状态，它需要重新填充 PTE 和 PFN 数据库中对应的项。如果是原型 PTE 的情形，则调用 MiCompleteProtoPteFault 函数，让硬件 PTE 变成有效。最后，释放资源以及维护锁的一致性，函数返回。

通过以上对于 MmAccessFault、MiDispatchFault 以及几个辅助函数的解释，我们已经了解了 Windows 的页面错误处理过程，更加清楚图 4.24 和 4.25 所示的软件 PTE 在整个页面交换机制中所起的作用。尤其是原型 PTE，虽然增加了页面交换机制的复杂性，但是它实现了进程间的内存共享，并且将页面交换机制与进程间内存共享机制有机地融合在一起。

4.4.4 Windows 的写时复制

写时复制（copy-on-write）是现代操作系统的一个重要特性，几乎所有的现代操作系统都支持这一特性，Windows 也不例外。一种典型的情形是，当 UNIX 的 fork 例程生成一个子进程时，原则上父进程和子进程各有自己独立的地址空间，但如果系统真的为子进程分配完全独立的页面，并且把从父进程继承的数据和代码拷贝到子进程的页面中，这不仅费时，也消耗了大量不必要的内存空间。通常的做法是，两个进程虽然有自己独立的页表，但页表项指向相同的物理页面，所有的数据页面被映射成“只读”页面。如果父进程

或子进程以后读这些数据，但并不修改它们，则一切都没有问题。但是，如果有哪个进程要写数据，哪怕只是修改一个页面中的一个字节，则系统就会捕捉到访问违例。在这种情况下，内存管理器会复制一个页面，让两个进程有自己私有的页面，并且都是可读写的，这样以后再访问这个页面就不会导致访问违例了。这种机制称为“写时复制”，它把页面复制的动作延迟到了真正需要为两个进程分配各自私有页面的时候，因此，那些实际上并不会被写入数据的页面不会再额外占用内存资源。写时复制的优势是，既避免了昂贵的、不必要的数据拷贝操作，也减缓了对内存的实际需求。

在上一节介绍页面错误处理时，我们实际上已经看到过写时复制的支持代码了。接下来我们进一步讨论 Windows 中关于写时复制的实现机理。首先，我们从内存区对象的创建开始。NtCreateSection 和 MmCreateSection 函数有一个参数 SectionPageProtection，它指定了内存区对象中页面的保护属性，可以是 PAGE_READ、PAGE_READWRITE、PAGE_EXECUTE 或 PAGE_WRITECOPY（也可以指定 PAGE_NOCACHE 或 PAGE_WRITECOMBINE）。这里 PAGE_READWRITE 的含义是，不管将来哪个进程读或者写这个页面，都始终只有一个页面，也就是说，完全共享这个页面；而 PAGE_WRITECOPY 的含义则是，如果有一个进程要写这个页面，则生成其自己的私有拷贝，并且私有拷贝的保护属性为 PAGE_READWRITE。显然，保护属性为 PAGE_WRITECOPY 的页面即上面提到的写时复制页面。

参考图 4.22，MmCreateSection 调用 MiCreateImageFileMap、MiCreateDataFileMap 或 MiCreatePagingFileMap 函数来创建内存区对象。内存区的页面的原型 PTE 中的 Protection 域决定了该内存区页面是否写时复制。检查这三个函数的代码可以发现：通过 MiCreateImageFileMap 创建的映像文件内存区，默认情况下其页面都是写时复制的，除非在映像文件中为它的区域（在映像文件中称为 SECTION，对应于控制区对象中的子内存区 SUBSECTION 对象）指定了与写时复制不相容的保护属性；通过 MiCreateDataFileMap 创建的内存区对象，其页面的保护属性为 MM_EXECUTE_READWRITE，即可读写并且可执行，不支持写时复制；而通过 MiCreatePagingFileMap 创建的内存区对象，其保护属性是由 MmCreateSection 函数的参数 SectionPageProtection 通过 MiMakeProtectionMask 函数变化得来，其中 PAGE_WRITECOPY 变成 MM_WRITECOPY。MiMakeProtectionMask 函数的代码位于 base\ntos\mm\mmsup.c 文件中，基本上是一个简单的查表操作。

当一个进程使用内存区对象时，它必须映射一个视图，视图本身并不影响内存区对象的保护属性。然而，在视图所对应的 VAD 节点中包含了这一保护属性，代表该进程对于此内存区对象的操作类型。MMVAD 数据结构的 u.VadFlags.Protection 成员即为此目的。映像文件的视图，此值为 MM_EXECUTE_WRITECOPY，支持写时复制。而其他类型的内存区

视图，此值由 MmMapViewOfSection 的参数来决定（也要通过 MiMakeProtectionMask 变换）。

真正实现写时复制能力的代码是在页面错误例程中。在 Windows 为硬件 PTE 定义的数据结构 MMPTE_HARDWARE 中，第 9 位是一个保留位（参见图 4.23），但 Windows 将它解释成了 CopyOnWrite 位。接下来我们从两个阶段来讨论写时复制的实现。

首先，我们考虑一个写时复制的页面是如何被第一次分配的。假设一个进程映射了一个视图，比如说映像文件视图。当该程序第一次访问一个页面时，其硬件 PTE 是无效的，页面错误处理例程有必要根据它的原型 PTE 来生成实际 PTE，正如上一节最后部分所介绍的那样，这一过程是在 MiCompleteProtoPteFault 函数中完成的。MiCompleteProtoPteFault 函数调用宏 MI_MAKE_VALID_USER_PTE 或 MI_MAKE_VALID_PTE 来填充 PTE 中的位，与保护属性有关的位填充是通过常量数组 MmProtectToPteMask 来做到的。在 MmProtectToPteMask 的定义中（见 base\ntos\mm\miglobal.c），写时复制保护属性被解释成 0x200，即 PTE 中的第 9 位（也即 CopyOnWrite 软件位）置位，而写位（即第 1 位）并没有置上，所以，如果进程要写这个页面的话，仍然会触发页面错误。这就进入下一个阶段。

在第二阶段，由于一个进程对一个支持写时复制的页面执行了写操作，因而触发一个页面错误。MmAccessFault 在检测到这一条件后，调用 MiCopyOnWrite 函数来处理，见 base\ntos\mm\mmfault.c 文件的 701~721、1 391~1 397 行代码。MiCopyOnWrite 函数是真正将一个写时复制页面分离出私有页面的函数。

MiCopyOnWrite 函数位于 base\ntos\mm\wrtfault.c 文件的 22~376 行，它实现了写时复制的复制过程。它首先根据参数中指定的出错地址定位到 PFN 数据库中对应的项，验证该项应该是一个原型 PTE。然后通过 MiRemoveAnyPage 函数申请一个物理页面，调用 MiInitializeCopyOnWritePfn 初始化其 PFN 数据库中对应的项。接着，从系统的 PTE 区域中申请一个空闲 PTE，以便完成内存页面的拷贝，将原来只读页面的内容拷贝到新申请的内存页面中。最后，填好出错地址的 PTE 项，并将旧页面在 PFN 数据库中的计数减 1，表明针对此页面，当前进程拥有私有拷贝，不再参与写时复制式的页面共享了。

4.5 物理内存管理

在系统地址空间，有一个区域称为 PFN 数据库，用于管理系统的物理内存。在前两节介绍进程内存管理和页面错误处理时，我们在代码中经常会看到 PFN 数据库的使用，也看到了利用 MiRemoveAnyPage 或 MiRemoveZeroPage 函数来申请物理内存页面的用法。本节我们来讨论 Windows 中物理内存的管理。

4.5.1 PFN 数据库

PFN 数据库是一个阵列，或者说是一个数组，每一项都描述了一个物理页面的状态。在 4.2.1 节介绍系统地址空间初始化时，我们已经看到了 PFN 数据库的初始化。现在来看一下 PFN 数据库的定义：

```
typedef struct _MMPFN {
    union {
        PFN_NUMBER Flink;
        WSLE_NUMBER WsIndex;
        PKEVENT Event;
        NTSTATUS ReadStatus;
        SINGLE_LIST_ENTRY NextStackPfn;
    } u1;
    PMMPTE PteAddress;
    union {
        PFN_NUMBER Blink;
        ULONG_PTR ShareCount;
    } u2;
    union {
        struct {
            USHORT ReferenceCount;
            MMPFNENTRY e1;
        };
        struct {
            USHORT ReferenceCount;
            USHORT ShortFlags;
        } e2;
    } u3;
    union {
        MMPTE OriginalPte;
        LONG AweReferenceCount;
    };
    union {
        ULONG_PTR EntireFrame;
        struct {
            ULONG_PTR PteFrame: 25;
            ULONG_PTR InPageError : 1;
            ULONG_PTR VerifierAllocation : 1;
            ULONG_PTR AweAllocation : 1;
            ULONG_PTR Priority : MI_PFN_PRIORITY_BITS;
            ULONG_PTR MustBeCached : 1;
        };
    } u4;
} MMPFN, *PMMPFN;

extern PMMPFN MmPfnDatabase;

#define MI_PFN_ELEMENT(index) (&MmPfnDatabase[index])
```

可以看出，MmPfnDatabase 数组是以页帧编号为索引的，因此，要查看一个物理页面的状态，只需直接以页帧编号为索引，就可以获取到该页面的 PFN 项，这正是宏

MI_PFN_ELEMENT 所做的事情。因此，从一个有效的 PTE，可以很方便地找到其页面的 PFN 数据库项。另一方面，在 PFN 数据库项 MMPFN 的定义中，它包含 6 个成员，每个成员均是 32 位，所以，PFN 数据库项的长度是 24 字节。这些 C 联合类型也表明了当物理页面处于不同的状态时，PFN 数据库项可以有不同的解释方法。

页面的状态与工作集的概念紧密相关，在 Windows 中，除了每个进程都有一个明确的工作集以外，系统也有一个工作集，每个会话也都有其工作集。另有一些内存不属于任何一个工作集，比如非换页内存池。关于工作集的详细讨论将在 4.6 节中进行，在这一节中我们只需知道每个进程和系统都有一个工作集，而页面可能属于某个工作集，也可能不属于任何一个工作集。一个物理页面的可能状态如下：

- 活动状态（active）。也称为有效状态（valid）。页面处于活动状态是指正在被某个进程使用，或者被用于系统空间（非换页内存池，或者在系统工作集）。对应有一个有效的 PTE 指向该页面。
- 备用状态（standby）。这种页面原来属于某个进程或系统工作集，但现在已经从工作集中移除。这种页面包含的数据对于原来的工作集仍然是有效的，也就是说，在原来工作集中，页面的内容尚未被修改。原来工作集中的 PTE 仍然指向该页面，但是已被标记成正在转移的无效 PTE。这种页面处于被回收状态，既可以被系统回收以作他用，也可以被原来的工作集回收而继续留用。
- 已修改状态（modified）。类似于备用状态，已经从原来的工作集中移除，但是，页面包含的内容已经被修改过。原来工作集中的 PTE 仍然指向物理页面，但已被标记成正在转移的无效 PTE。如果系统要把这种页面回收作他用，则必须将其中的内容写到磁盘上。
- 已修改但不写出（modified no-write）。类似于上一种状态，但区别在于，内存管理器不会将它的内容写到磁盘上。
- 转移状态。说明一个页面正处于 I/O 操作进行中，当两个线程并发地在同一个页面上引发页面错误时，页面错误处理例程通过这一状态可以判断出冲突的页面错误的情形，从而正确地处理。在 4.4.3 节中我们看到过这种页面错误冲突的处理过程。值得一提的是，这里的转移状态是针对一个物理页面中的内容，而无效 PTE 的转移状态则是针对它所指的页面已被转移到备用链表或已修改链表中。两者含义大不相同。
- 空闲状态。页面是空闲的，不属于任何一个工作集，它们包含了不确定的数据，内存管理器在重新使用这些页面以前，应根据需要（尤其是出于安全考虑）清除脏数据。

- 零化状态。页面是空闲的，不属于任何一个工作集，其中的内容已经被全部清零。
- 坏状态。页面产生硬件错误。系统不再使用这样的页面。

除了坏状态的页面以外，其他七种状态的页面都是由内存管理器来管理和调度的。在 MMPFN 数据结构中，虽然每一种页面都是固定长度（24 字节）的数据结构，但是它们使用的 C 联合成员并不完全相同。图 4.27 把 MMPFN 的成员针对每种状态拆分了出来，这样我们可以更加清楚 MMPFN 的用法。图中斜体部分是对于不同状态页面的 PFN 项有所区别。

```
union {
    WSLE_NUMBER WsIndex;
} u1;
PMMPTE PteAddress;
union {
    ULONG_PTR ShareCount;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
  };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
  };
} u4;
```

(a) 活动状态页面的 PFN 项

```
union {
    PFN_NUMBER Flink;
} u1;
PMMPTE PteAddress;
union {
    PFN_NUMBER Blink;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
  };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
  };
} u4;
```

(b) 备用状态或已修改状态的 PFN 项

```
union {
    PFN_NUMBER Flink;
} u1;
PMMPTE PteAddress;
union {
    PFN_NUMBER Blink;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
  };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
  };
} u4;
```

(c) 零化的或空闲页面的 PFN 项

```
union {
  PKEVENT Event;
    NTSTATUS ReadStatus;
} u1;
PMMPTE PteAddress;
union {
    ULONG_PTR ShareCount;
} u2;
union {
    struct {
        USHORT ReferenceCount;
        MMPFNENTRY e1;
  };
} u3;
MMPTE OriginalPte;
union {
    struct {
        ULONG_PTR PteFrame: 25;
        ULONG_PTR InPageError : 1;
        ULONG_PTR
VerifierAllocation : 1;
        ULONG_PTR AweAllocation : 1;
        ULONG_PTR Priority : 3;
        ULONG_PTR MustBeCached : 1;
  };
} u4;
```

(d) 转移状态的 PFN 项

图 4.27 处于不同状态的页面的 PFN 数据库项的定义

对于活动状态的 PFN 项，这样的页面属于一个工作集，所以，WsIndex 包含了该页面在所属工作集链表中的索引。如果这个页面只被一个进程映射，也就是说它是一个私有页面，则 WsIndex 等于该页面在进程工作集链表中的索引。如果该页面被多个进程共享，则该域只保证第一个使该页面变成有效的进程是正确的，通过此域，可以快速地在工作集中定位到它的工作集链表项。进程的工作集链表中包含一个数组，该数组描述了此进程中的每一个页面，参见 4.6.1 节关于进程工作集链表结构的介绍。PteAddress 域指向此页面的 PTE 的虚拟地址。ShareCount 域指明了指向该页面的 PTE 的数量，而对于页表页面，该域表示页表中有效 PTE 和转移 PTE 的数量。只要 ShareCount 的值大于 0，它就不能从内存中移除。而 ReferenceCount 域（在 WRK 代码中通过 u3 的 e2 结构成员来引用）代表了这个页面必须要保留在内存中的引用计数，包括该页面被加入工作集或者由于 I/O 的需要而被锁定的次数。第一次加入工作集（共享计数也必然大于 0），其引用计数增 1；I/O 每锁定一次，引用计数增 1。ShareCount 减为 0，引用计数减 1；I/O 解锁一次，引用计数减 1。当引用计数为 0（共享计数也必然是 0）时，该页面被转移到空闲链表、备用链表或修改链表中。u3 的 e1 成员是一个重要的域，其中包含了各种标志位，MMPFNENTRY 的定义如下：

```
typedef struct _MMPFNENTRY {
   USHORT Modified : 1;
   USHORT ReadInProgress : 1;
   USHORT WriteInProgress : 1;
   USHORT PrototypePte: 1;
   USHORT PageColor : 4;
   USHORT PageLocation : 3;
   USHORT RemovalRequested : 1;
   USHORT CacheAttribute : 2;
   USHORT Rom : 1;
   USHORT ParityError : 1;
} MMPFNENTRY;
```

以上的标志信息是所有状态的 PFN 项共享的，反映了所描述的物理页面的一些状态。其中 Modified 标志指页面已被修改，如果该页面要从内存中移除，则要求先保存其内容；PrototypePte 标志表示该 PFN 项引用的 PTE 是一个原型 PTE；ReadInProgress 和 WriteInProgress 位分别代表读操作和写操作正在进行，它们也被用做非换页内存池中页面的 StartOfAllocation 和 EndOfAllocation 位，参见 4.2.2 节。PageLocation 域代表了当前页面状态的类型，也指明了该页面在系统定义的哪个链表中，稍后有进一步解释。关于 PageColor 域，4.5.3 节在介绍物理页面管理和分配时将会用到它。

MMPFN 的 OriginalPte 成员包含了指向此页面的 PTE 的原始内容，可能是一个原型 PTE。它的用意是，当将一个物理页面分配给一个 PTE 时，它记录了原来的 PTE；以后当该物理页面不再为它所用时，可以恢复其原来的 PTE。最后一个成员 u4 指向该页面的 PTE 所在的页表页面的物理页帧编号，以及一些标志位。

对于备用状态或已修改状态的物理页面，显然不再有有效的 PTE 指向它，也不再属于任何一个工作集，但它们会加入到系统的备用链表或修改链表中，所以，MMPFN 的第一个和第三个 4 字节成员直接变成它在链表中的前后指针。已修改但不写出状态的页面也使用同样的数据结构定义。

对于零化或空闲页面的 PFN 项，其第一项构成了它们在零化链表或空闲链表上的指针，第三个 4 字节未使用。

对于正在转移状态的 PFN 项，其第一项要么指向一个同步事件，要么是一个 I/O 错误码。如果 I/O 正在进行中，则该事件对象将会被通知到；如果在页面换入过程中产生错误，则 ReadStatus 包含了代表此 I/O 错误的状态码。正如前面所讲，转移状态的 PFN 项的用途是识别和消除冲突的页面错误。

在介绍了物理页面的状态以及 MMPFN 数据结构中有关的域以后，我们再来看一下物理页面的链表组织结构。首先看以下 C 枚举类型 MMLISTS 以及 PFN 链表类型 MMPFNLIST 的定义（见 base\ntos\inc\mm.h 文件）：

```
typedef enum _MMLISTS {
    ZeroedPageList,
    FreePageList,
    StandbyPageList,  //this list and before make up available pages.
    ModifiedPageList,
    ModifiedNoWritePageList,
    BadPageList,
    ActiveAndValid,
    TransitionPage
} MMLISTS;

typedef struct _MMPFNLIST {
    PFN_NUMBER Total;
    MMLISTS ListName;
    PFN_NUMBER Flink;
    PFN_NUMBER Blink;
} MMPFNLIST;

typedef MMPFNLIST *PMMPFNLIST;
```

MMPFN 数据结构的 u3.e1.PageLocation 域（共 3 位）保存的是以上某一个枚举值，说明这个页面处于什么状态，在哪个链表中。相应地，系统定义了 6 个链表，分别有 6 个全局变量代表这些链表的链表头。全局数组 MmPageLocationList 将这些链表按照 MMLISTS 枚举值的顺序组织在一起，其定义如下（见 base\ntos\mm\miglobal.c 文件）：

```
PMMPFNLIST MmPageLocationList[NUMBER_OF_PAGE_LISTS] = {
                                &MmZeroedPageListHead,
                                &MmFreePageListHead,
                                &MmStandbyPageListHead,
```

```
                    &MmModifiedPageListHead,
                    &MmModifiedNoWritePageListHead,
                    &MmBadPageListHead,
                    NULL,
                    NULL };
```

因此，在 Windows 内核代码中，根据 PFN 可以很方便地找到 MMPFN 项，再根据 PageLocation 域的值就可以知道该页面当前的状态，以及位于哪个链表中。

最后，图 4.28 显示了 PFN 数据库和这组链表形成的物理页面管理结构。活动页面和转移状态的页面没有形成链表，零化页面和空闲页面仅构成单链表。理解了 Windows 中物理页面的组织结构以后，在下一小节中，我们将看到，一个物理页面的状态是如何转移的，它又是如何在这些链表之间移动的。我们也会回顾页面错误处理过程，以便更加清楚如何使用这些链表来解决页面错误。

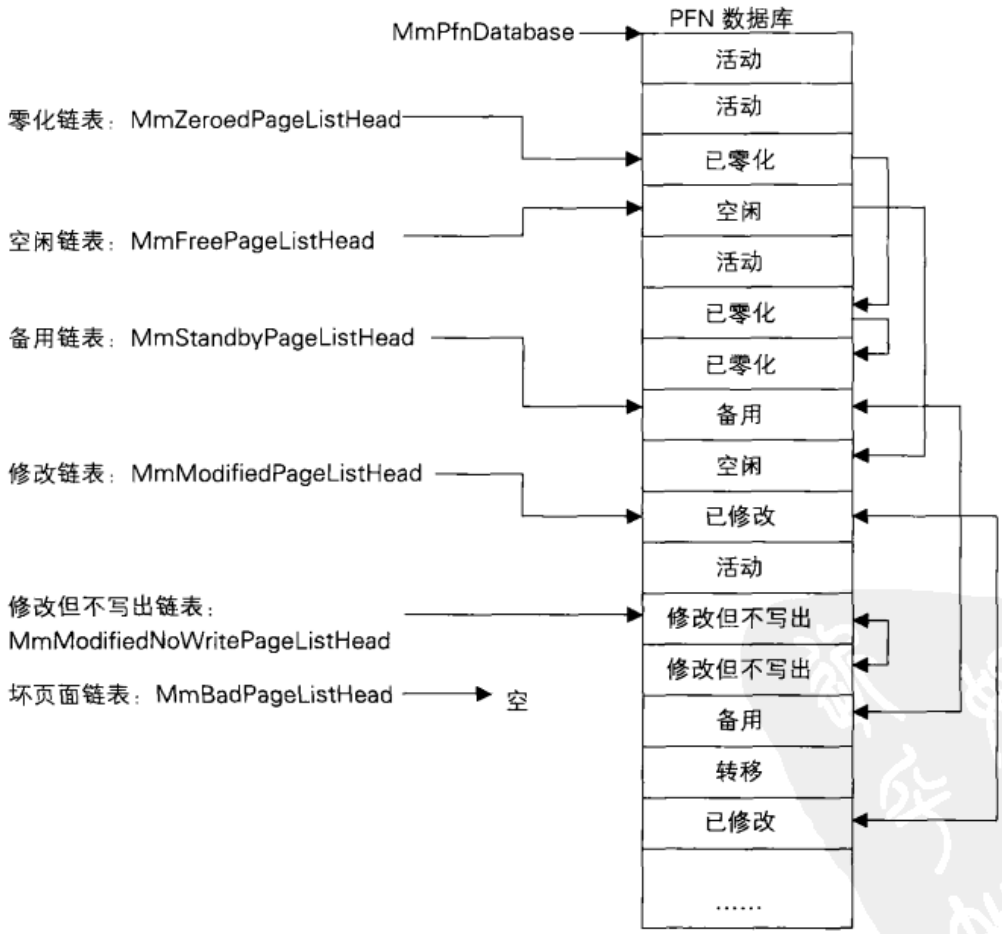


图 4.28 PFN 数据库和物理页面链表

4.5.2 物理页面的状态变化

上一小节介绍了物理页面的 8 种状态以及 6 个页面链表将其中 6 种状态的页面串联起来。内存管理器根据系统内存的数量以及各个进程对于内存的需求，动态地调度这些物理页面的使用。例如，当一个进程需要内存时，内存管理器可以从零化链表或空闲链表中找到一个或多个页面来满足进程的需求；当进程退出时，内存管理器回收该进程的页面；当物理内存紧缺时，内存管理器按照特定的策略将有些进程中的页面换出到外存中，从而腾出物理内存以作他用。在内存管理器的动态调度过程中，每个页面都有可能经历各种状态变化。这一节我们来讨论物理页面的状态变化。

为简便起见，我们不考虑已修改但不写出（modified no-write）页面链表和坏页面链表，这两个链表的特点将在下一节讨论。图 4.29 显示了 Windows 中的物理页面状态转移图。

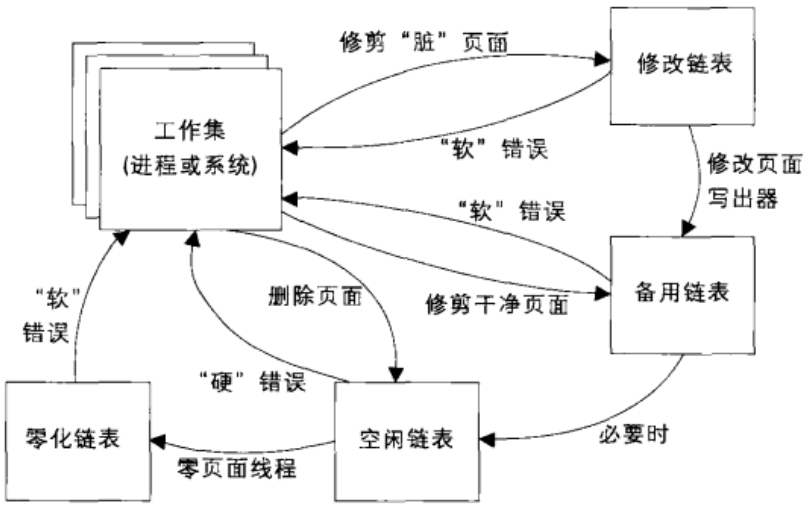


图 4.29 物理页面的状态转移图

在页面错误处理过程中，当进程需要一个页面时，它有可能从图中所示的 4 个链表中获得物理内存页面，从而解决页面错误。所以，这 4 个链表中的页面都有可能进入到一个工作集中。

- 如果页面错误是由一个正在转移状态的无效 PTE 引起的，则根据转移状态的无效 PTE 中的 PFN 定位到物理页面的 PFN 数据库项，然后调用 MiUnlinkPageFromList 函数将该页面从备用链表或修改链表中移除，并置为有效 PTE。在这种情况下，页面可能位于修改链表或备用链表中，由于 PFN 数据库项本身包含前后指针，所以，在调用 MiUnlinkPageFromList 函数时并不需要传递链表头，只需 PFN 项即可。4.4.3 节中介绍的 MiResolveTransitionFault 函数解决了这种类型的页面错误。

- 如果页面错误是由一个位于页面文件中的 PTE 而引起的，则页面错误处理例程会通过 MiRemoveAnyPage 函数获得一个物理页面来满足此页面错误。MiRemoveAnyPage 函数的大致逻辑是，首先检查空闲链表，只要它不为空，就可以获得空闲页面。然后检查零化链表，如果它为空，再进一步检查备用链表。MiRemoveAnyPage 函数的代码位于 base\ntos\mm\pfnlist.c 文件的 2 353~2 592 行，它的实际逻辑还要更复杂一些，参见 4.5.3 节的介绍。4.4.3 节中介绍的 MiResolvePageFileFault 函数解决了这种类型的页面错误。
- 如果页面错误需要一个零页面来满足，则调用 MiRemoveZeroPage 函数获得一个零化的页面，或者调用 MiRemoveAnyPage 函数再显式地赋零。MiRemoveZeroPage 函数检查零化链表，如果为空的话，再检查空闲链表和备用链表，并显式地赋零。其代码位于 base\ntos\mm\pfnlist.c 文件的 2 047~2 351 行，实际逻辑更复杂一些。4.4.3 节中介绍的 MiResolveDemandZeroFault 函数解决了这种类型的页面错误。

从空闲链表到零化链表的转移是由一个称为零页面线程（zero page thread，System 进程中的第一个线程）的系统线程来完成的。系统在阶段 1 初始化完成以后，调用 MmZeroPageThread 函数，因此，阶段 1 初始化线程蜕变成零页面线程，参见 2.6.2 节关于内核初始化过程的介绍。MmZeroPageThread 函数的代码位于 base\ntos\mm\zeropage.c 文件的 43~431 行，它永不退出。

MmZeroPageThread 函数的主体是一个无限循环，它等待两个事件之一：MmZeroingPageEvent 和 PoSystemIdleTimer。在函数 MiInsertPageInFreeList（往空闲链表插入页面）中，一旦空闲链表中超过 MmMinimumFreePagesToZero（等于 8）个页面，则该函数通过 MmZeroingPageEvent 事件通知零页面线程，可以执行零化操作了。由于零页面线程的优先级为 0，所以，只有当处理器空闲的时候，它才有机会被执行。接到零化事件通知以后，MmZeroPageThread 首先从空闲链表中取得一定数量的页面，并临时将它们置上坏页面类别，然后显式赋零，再逐个调用 MiInsertPageInList 函数将其插入到零化链表 MmZeroedPageListHead 中。在插入此链表后，这些页面又变成已零化的页面类别。如此循环往复，直至当前空闲链表上的页面全部零化，再回到外层循环的等待函数处。

从备用链表到空闲链表的转移是按需进行的。内存管理器有一个工作集管理器，它会定时地按照特定的准则将工作集中的某些页面移到备用链表中，但备用链表中的页面并不会被主动地转移到空闲链表中，除非这些页面已确定没有用处了。发生这种转移的一种情形是，当进程删除一段内存（发生在 MiDeleteVirtualAddresses 函数中）时，处于备用状态的页面被加入到空闲链表中。这也包括当进程、线程结束时清除内存所做的操作。

MiDeleteVirtualAddresses 调用 MiDeletePte 函数来完成页面状态的转移，见 base\ntos\mm\deleteva.c 文件的 1 171~1 207 行代码。另一方面，当 MiRemoveAnyPage 或 MiRemoveZeroPage 需要找到一个可用的物理页面时，如果在零化链表或者空闲链表中未能找到页面，则直接从备用链表中寻找页面，并根据需要零化或不零化。在这种情况下，备用页面以空闲页面或者零化页面的形式又立即回到进程工作集中，但并非恢复原来的有效页面，而是被映射到新的虚拟地址上，甚至新的进程中。

当进程的一段地址被删除时，譬如进程调用 NtFreeVirtualMemory 函数时，或者当进程或线程结束时，MiDeleteVirtualAddresses 函数被调用到，它除了把备用页面转移到空闲链表中以外，还把不再共享的有效页面转移到空闲链表中。因此，我们可以看到，在图 4.29 中，有些页面直接从进程工作集转移到了空闲链表中。此过程是这样发生的：MiDeleteVirtualAddresses 对满足条件的页面逐个调用 MiDeletePte 函数，而 MiDeletePte 又调用 MiDecrementShareCount 函数。在 MiDecrementShareCount 函数中，如果共享计数减 1 后为 0，再进一步判断引用计数是否为 1，若是，则说明可以立即删除，故调用 MiInsertPageInFreeList 函数将页面转移到空闲链表中；否则，只需引用计数减 1 即可。MiDecrementShareCount 函数的代码位于 base\ntos\mm\pfndec.c 文件的 25~198 行，从该函数的处理过程可以看出共享计数和引用计数之间的关系，如上一小节所述。

当系统运行到可用内存数量很少时，内存管理器会修剪进程或系统的工作集，这项工作是由一个称为工作集管理器（MmWorkingSetManager 函数）的组件来完成的。工作集管理器定期运行（每 1 s 醒来一次），并根据一定的规则来选择哪些工作集可以被修剪，修剪哪些页面、多少页面。脏的页面被移到修改链表中，干净的页面被移到备用链表中。MmWorkingSetManager 调用 MiProcessWorkingSets 函数来处理修剪事宜，而 MiProcessWorkingSets 又进一步调用 MiTrimWorkingSet 函数来处理单个工作集的修剪。MiTrimWorkingSet 真正实施工作集修剪，它调用 MiFreeWsleList 函数剪掉满足条件的页面，并调用 MiRemoveWorkingSetPages 函数进一步去掉不必要的工作集项。关于工作集管理器的详细工作过程，请参考 4.6.1 节。

随着工作集中的页面被加入到修改链表中，修改链表可能会变得很大，而零化链表或备用链表则越来越小，到一定时候，内存管理器会把修改链表中的页面数据写到磁盘上，从而把页面转移到备用链表中，这一任务是由一个称为修改页面写出器（modified page writer）的组件（MiModifiedPageWriter 函数，位于 base\ntos\mm\modwrite.c 文件）和一个称为映射页面写出器（mapped page writer）的组件（MiMappedPageWriter 函数，在同一文件中）来完成的，关于这两个写出器的工作过程，请参考 4.5.4 节。

4.5.3 物理页面链表的管理和操作

我们已经看到过，当内存管理器需要空闲页面或者零化页面时，它可以通过 MiRemoveAnyPage 或 MiRemoveZeroPage 函数来获得。这两个函数依次检查空闲链表、零化链表和备用链表，并根据需要对页面进行零化。检查它们的代码，我们可以发现，这两个函数都带有一个称为 color（颜色）的参数，而且除了在 4.5.1 节中我们看到的链表 MmZeroedPageListHead 和 MmFreePageListHead 以外，还有一个链表数组 MmFreePagesByColor。MMPFN 数据结构的 u3.e1 成员有一个 4 位的 PageColor 域，代表了一个 PFN 项的颜色。此外，我们还看到，真正的备用链表是一个链表数组 MmStandbyPageListByPriority，而非 4.5.1 节中列出的 MmStandbyPageListHead。Windows 实现的备用页面是有优先级的，MMPFN 数据结构中有一个 3 位的域 Priority 指明了一个备用页面的优先级，默认的 Priority 为 3。

这里，PFN 项的颜色表示一个页面在处理器的缓存中的位置。Windows 试图让页面尽可能遍布在缓存中，从而更加有效地发挥缓存的作用。这种做法对于 MIPS 体系结构和 NUMA 结构有显著的意义。在 NUMA 结构中，颜色的高位部分代表了节点编号，因此，当特定颜色的页面无法满足时，内存管理器优先考虑同一节点上其他颜色的页面，其次考虑不同节点上的页面。我们可以将颜色抽象地理解成缓存的索引号。

引入了颜色的概念以后，在空闲链表和零化链表中插入和删除页面稍微复杂了一些。我们首先来看带颜色的页面链表的定义，如下：

```
typedef struct _MMCOLOR_TABLES {
    PFN_NUMBER Flink;
    PVOID Blink;
    PFN_NUMBER Count;
} MMCOLOR_TABLES, *PMMCOLOR_TABLES;

extern PMMCOLOR_TABLES MmFreePagesByColor[2];
```

颜色链表的初始化是在 MiInitMachineDependent 函数中进行的，有两个全局变量与颜色链表有关：MmSecondaryColors 和 MmSecondaryColorMask。默认情况下，MmSecondaryColors 为 64，MmSecondaryColorMask 为 63，这意味着系统定义了 64 种颜色。MmFreePagesByColor 数组有两项，第一项对应于零化页面，第二项对应于空闲页面。每一项又都指向一个有 64 个元素的数组，分别对应于每种颜色的表头。譬如说，MmFreePagesByColor[0][0]代表颜色为 0 的零化页面链表头，MmFreePagesByColor[0][1]代表颜色为 1 的零化页面链表头……MmFreePagesByColor[1][0]代表颜色为 0 的空闲页面链表头，依此类推。这些链表也是双链表，页面 PFN 项的 u4.PteFrame 有 25 位，足够用

做一个链表指针；另一个指针是OriginalPte。所以，一个页面加入到这样的一个链表中，并不妨碍该页面在MmFreePageListHead双链表中的前后指针。下面来看零化和空闲页面的插入和移除操作，代码位于pfnlist.c文件中：

- 空闲页面的插入。在MiInsertPageInFreeList函数中，首先将页面插入到MmFreePageListHead链表的尾部，然后根据页面的颜色（跟MmSecondaryColorMask进行与操作，也就是说，取低6位）Color，找到相应的颜色链表MmFreePagesByColor[FreePageList][Color]。再把页面插入到颜色链表的尾部。然后返回。
- 零化页面的插入。在MiInsertPageInList函数中，首先将页面插入到MmZeroedPageListHead链表的前端，见base\ntos\mm\pfnlist.c文件的558~571行。然后，将页面插入到MmFreePagesByColor[ZeroedPageList][Color]链表中，同样地，这里的Color等于页面的颜色，见代码658~677行。另外，在MiInsertZeroListAtBack函数中，页面被插入到这两个链表的尾部，见同一文件中的809~828、888~907行代码。
- 从链表头移除一个零化或空闲页面。在MiRemovePageFromList函数中，首先从指定的链表头部移除一个页面，然后更新颜色链表，见pfnlist.c文件的1 231~1 242行。在MiRemovePageByColor函数中，根据指定的颜色和页面PFN，从链表头移除页面，并更新相应的颜色链表，该函数的代码位于pfnlist.c文件的2 595~2 749行。
- 从链表中间移除一个零化或空闲页面。在MiUnlinkFreeOrZeroedPage函数中，根据页面PFN项的u3.e1.PageLocation成员定位到链表，并利用它的前后指针关系，将自己从链表中摘除出来；同样地，根据自己的颜色定位到颜色链表，并从双链表中摘除出来。MiUnlinkFreeOrZeroedPage函数的代码位于pfnlist.c文件的1 415~1 560行。

接下来我们看一看备用链表的用法。正如前面所讲，MMPFN用3位描述一个页面的备用优先级（0~7），相应地，有8个备用链表分别用来存放各个优先级的页面。全局数组MmStandbyPageListByPriority正是这一用途。注意，在WRK中，备用链表MmStandbyPageListHead虽然有定义，但其实并没有用到，真正的备用链表是MmStandbyPageListByPriority数组。MmStandbyPageListByPriority数组的初始化位于mminit.c文件的MmInitSystem函数中。下面是备用页面的插入和移除操作：

- 在链表前端插入备用页面。在MiInsertStandbyListAtFront函数中，根据页面PFN项的u4.Priority定位到链表头，并将自己插入到此双链表中。其代码并不复杂，见pfnlist.c文件的918~1 071行。另外，在MiInsertPageInList函数中，即使参数中指定了MmStandbyPageListHead作为链表头，真正的目标链表头也要转变成MmStandbyPage-

ListByPriority 数组中对应优先级的链表，见 471~474 行代码。插入双链表的方法并没有变化。

- 从链表中移除一个备用页面。在 MiRemovePageFromList 函数中，如果要移除备用页面，那么，客户代码将不能指定 MmStandbyPageListHead 作为备用链表，而必须是 MmStandbyPageListByPriority 数组的一个成员。从双链表中移除一个节点并没有特殊之处，但是，对于一个备用页面，它对应的 PTE 处于转移状态，所以，这样的页面移除以后，须恢复对应 PTE 的内容，从而它不再处于转移状态。这是通过 MiRestoreTransitionPte 函数来完成的。
- 从链表中间移除一个备用页面。在 MiUnlinkPageFromList 函数中，如果根据页面 PFN 项得到的链表是 MmStandbyPageListHead，则将链表转换为 MmStandbyPageList-ByPriority 数组中对应的链表，移除操作本身并无特殊之处。

修改链表分成了两部分：若页面 PFN 项的 OriginalPte.u.Soft.Prototype 位域为 0，则该页面被置于 MmModifiedPageListByColor[0]链表中，这些页面的外存位置是页面文件，因此希望通过颜色来聚集同色页面，但 WRK 的实现并没有分多种颜色；其余的页面被放在全局链表 MmModifiedPageListHead 中，这些页面的外存位置是映射文件。值得注意的一点是，MmModifiedPageListHead 中的计数值也包括位于页面文件的那些页面。修改页面的插入通过 MiInsertPageInList 函数来实现，移除通过 MiUnlinkPageFromList 函数来实现。

修改但不写出页面的链表是一个单独处理的链表，该链表上的页面也被修改了，但是内存管理器的修改页面写出器（modified page writer，下一小节介绍）并不会把这些页面写到外存中。通过 MiInsertPageInList 函数可在该链表的尾部插入一个页面；通过 MiInsertFrontModifiedNoWrite 函数可在该链表的前端插入一个页面；页面移除则通过 MiUnlinkPageFromList 函数来完成。因为内存管理器并不自动调度该链表中的页面，所以，客户必须显式地移除该链表中的页面，然后它们才有机会被刷新到外存中。这一显式转换是由 MmEnableModifiedWriteOfSection 函数（位于 base\ntos\mm\modwrite.c 文件的 4 561~4 646 行）来完成的，它的做法是，先调用 MiUnlinkPageFromList 函数把页面的 PFN 项从修改但不写出页面链表中移除，然后调用 MiInsertPageInList 函数将它插入到修改链表中，变成普通的修改页面，从而允许修改页面写出器将页面内容写到外存中。注意，内存区对象的控制区对象的 u.Flags.NoModifiedWriting 标志位用于指明这一特性。

修改但不写出页面的一个典型用途是，NTFS 文件系统利用这种页面来映射文件系统元数据（metadata），所以，这些元数据不会被自动刷新到磁盘上。只有当记录元数据修改情况的日志信息被写到磁盘上以后，这些元数据才被显式地写到磁盘上。这样，NTFS

可以保证对元数据的任何修改总是会被记录下来，从而一旦发生意外事故，元数据可以恢复。但是，一个副作用是，这些页面将占据系统物理内存，所以，客户有责任确保它们不会被累积太多，并且应显式地通过 MmEnableModifiedWriteOfSection 函数将它们移除。

坏页面链表是一个相对简单的链表，它并不参与系统的页面调度。通过 MiInsertPageInList 函数可以将一个坏页面插入 MmBadPageListHead 链表。如果零化、空闲或备用页面的 PFN 项的 u3.e1.RemovalRequested 位域被置上，则该页面将被插入到坏页面链表中，无论它是什么类型的页面。理论上，坏页面插入以后就不再移除。

有了以上这些关于页面链表的基础，我们再回顾一下前面多次用到（尤其在页面错误处理例程中）的两个物理页面申请函数：

- MiRemoveZeroPage，其代码位于 base\ntos\mm\pfnlist.c 文件的 2 047~2 351 行。它根据指定的颜色，定位到零化颜色链表，若链表不为空，则调用 MiRemovePageByColor 函数，移除该颜色的链表头页面，然后返回。若指定颜色的零页面找不到，则检查零化链表，只要不为空，就移除一个其他颜色的页面，并返回。接下来按照同样的逻辑检查空闲链表，先找指定颜色的链表中的页面，再找其他颜色的空闲页面。若能找到，则跳转到零化逻辑，经过零化之后再返回。如果空闲链表也为空，则检查备用链表，从优先级 0 开始一直找到最高优先级，并调用 MiRemovePageFromList 函数移除一个页面，然后零化该页面。页面零化是通过 MiZeroPhysicalPage 函数来完成的。
- MiRemoveAnyPage，其代码位于 pfnlist.c 文件的 2 353~2 592 行。其逻辑与 MiRemoveZeroPage 相仿，只是查找链表的顺序不同，并且不需要零化。它根据指定的颜色首先在该颜色的空闲链表 MmFreePagesByColor[FreePageList][Color]中找页面，若没有，则继续在该颜色的零化链表 MmFreePagesByColor[ZeroedPageList][Color]中找页面；若还是没有，则在总的空闲链表 MmFreePageListHead 中寻找任意颜色的页面；若仍然没有，则在总的零化链表 MmZeroedPageListHead 中寻找任意颜色的零化页面。最后检查的链表是备用链表，从优先级 0 开始直至最高优先级。在任何一个分支上，一旦找到页面，无须零化，直接返回即可。

以上介绍了物理页面链表的管理和一些典型的操作。对于零化和空闲链表中的页面，颜色只是一个指示，其目的是让处理器的硬件缓存工作得更好。颜色的引入仅仅是一种优化，而非一种保证。在 Intel x86 系统上，颜色可以是小于 MmSecondaryColors 的任意值。在每个处理器的 KPRCB（处理器控制块）结构中都有一个 PageColor 成员，是处理器范围的全局颜色种子，每次通过 MmGetNodeColor 函数获得一个颜色值以后，此成员值加 1。其他对象（比如会话空间对象）也可以维护自己的颜色值。

最后，我们简单介绍另一种描述物理内存的数据结构。当内核或设备驱动程序需要多个物理内存页面时，Windows 内存管理器提供了一个称为 MDL（Memory Descriptor List，内存描述符链表）的数据结构来描述这些页面。MDL 的定义如下：

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

MDL 描述的内存是一组物理页面，其 PFN 紧跟在 MDL 对象之后，所以，通过以下的代码可以访问到包含这些页面 PFN 的数组：

```
    Pages = (PPFN_NUMBER) (Mdl + 1);
```

我们知道，Windows 内核和应用程序使用虚拟地址来访问内存，但有些设备驱动程序，出于数据传输性能方面的考虑，或者为了某些硬件特性（比如 DMA），有必要在物理内存地址上工作。在这种情况下，使用 MDL 是一条便捷的途径。以下是三个接口函数：

```
PMDL
MmAllocatePagesForMdl (
    __in PHYSICAL_ADDRESS LowAddress,
    __in PHYSICAL_ADDRESS HighAddress,
    __in PHYSICAL_ADDRESS SkipBytes,
    __in SIZE_T TotalBytes
    );

PMDL
MmAllocatePagesForMdlEx (
    __in PHYSICAL_ADDRESS LowAddress,
    __in PHYSICAL_ADDRESS HighAddress,
    __in PHYSICAL_ADDRESS SkipBytes,
    __in SIZE_T TotalBytes,
    __in MEMORY_CACHING_TYPE CacheType,
    __in ULONG Flags
    );

VOID
MmFreePagesFromMdl (
    __in PMDL MemoryDescriptorList
    );
```

MmAllocatePagesForMdl 和 MmAllocatePagesForMdlEx 函数用于申请物理内存，并返回一个 MDL 对象；MmFreePagesFromMdl 函数用于释放 MDL 对象中的物理内存。检查 MmAllocatePagesForMdl 和 MmAllocatePagesForMdlEx 函数的代码（位于 base\ntos\mm\

iosup.c 文件中）可以发现，它们实际上是通过 MiAllocatePagesForMdl 函数来完成其内存申请功能的。MiAllocatePagesForMdl 函数的参数与 MmAllocatePagesForMdlEx 完全相同，这里不再列出。

为了在指定的地址范围内获得足够的内存，MiAllocatePagesForMdl 函数搜索零化链表、空闲链表和备用链表，以尽可能地满足客户的内存需求。但该函数返回时，实际申请到的内存页面数有可能少于客户要求的数量，所以客户必须检查结果 MDL 对象中的成员值。另外，MiAllocatePagesForMdl 函数并不将这些物理内存映射到虚拟地址空间，客户如果需要虚拟地址，则必须自己负责映射。MmFreePagesFromMdl 函数负责把物理内存归还给 PFN 数据库，让它们继续为系统所用。有兴趣的读者可以阅读 MiAllocatePagesForMdl 和 MmFreePagesFromMdl 函数的代码（位于 base\ntos\mm\iosup.c 文件中）。

4.5.4 修改页面写出器

当系统的内存消耗很多时，零化链表、空闲链表和备用链表上的页面数量就会减少，当少到一定程度时，系统中将有两个线程开始工作，把已经修改的页面写出到外存中，其中一个负责往页面文件中写页面，另一个负责往映射文件中写页面。一旦页面的内容被写到外存，这些页面就将被转移到备用链表中，供系统或其他进程使用。这两个线程被称为修改页面写出器（modified page writer）。这一节我们来讨论它的实现。

修改页面写出器是在 MiModifiedPageWriter 函数中实现的。当内存管理器初始化时，MmInitSystem 函数创建一个系统线程，其启动函数为 MiModifiedPageWriter，见 base\ntos\mm\mminit.c 文件的 2 301 行代码。MiModifiedPageWriter 函数在执行起来以后，创建了第二个线程，启动函数为 MiMappedPageWriter，见 base\ntos\mm\modwrite.c 文件的 2 947 行。这两个线程的优先级均为 17，属于实时优先级，因此，相对于普通的线程而言，它们有更高的优先执行权。

我们首先来看针对页面文件的修改页面写出器 MiModifiedPageWriter 函数，其代码位于 base\ntos\mm\modwrite.c 文件的 2 846~2 984 行。它的流程是，先执行初始化工作，为以后的写文件操作预留好物理内存，并启动映射页面写出器线程，然后调用 MiModifiedPageWriterWorker 函数执行其主体逻辑。MiModifiedPageWriterWorker 函数在一个无限循环中等待两个事件（除非在停机时返回）：

- MmModifiedPageWriterEvent 事件。该事件的触发条件是：已修改页面的数量超过了一个预设的变量 MmModifiedPageMaximum（对于物理内存超过 33 MB 的机器，此

阈值为 800）；或者可用页面数量小于 256，并且修改链表中的页面数达到了修改页面写操作的聚集数，即 MmModifiedWriteClusterSize 变量中的值（等于 16）。请参考 base\ntos\mm\pfnlist.c 文件的 720~727 行，当在修改链表中插入页面时，这一条件有可能会满足。另外，在工作集管理器中，当内存紧张时，若已修改页面数达到了 MmModifiedWriteClusterSize，该事件也会被触发。

- MiMappedPagesTooOldEvent 事件。为了防止位于映射文件中的已修改页面未能被及时地写到外存中，该事件每隔 300 s 被触发一次（在一个定时器函数 MiModifiedPageWriterTimerDispatch 中）。此事件的用意是，防止对映射文件中的数据修改之后由于系统崩溃或者电源故障等意外造成大量数据未能及时刷新到外存中。

MiModifiedPageWriterWorker 函数等到了事件信号以后，判断应该处理页面文件的写操作还是映射文件的页面写操作。如果是页面文件的写操作，则调用 MiGatherPagefilePages 函数来完成实际的页面文件 I/O；如果是映射文件的页面写操作，则调用 MiGatherMappedPages 函数来完成映射文件 I/O。这是一个循环过程，直至页面都写出去，或修改页面降低到 MmModifiedWriteClusterSize 以下。完成这一轮循环以后，MiModifiedPageWriterWorker 继续等待下一次事件信号。

MiGatherPagefilePages 函数的代码见 modwrite.c 文件的 3 835~4 333 行，其处理流程大致如下：首先在页面文件的位图（标记了页面文件中每个页面的状态，参考页面文件数据结构 MMPAGING_FILE 的定义，本书不讨论页面文件结构）中扫描，以期找到能聚集在一起写出去的连续空闲页面，然后，从修改链表（确切地说，应该是 MmModifiedPageListByColor[0]）中逐个检查已修改页面，将它们的 PTE 设置成位于页面文件中，并设置 PFN 项的写标志位 u3.e1.WriteInProgress。完成这些设置以后，再调用 IoAsynchronousPageWrite 函数，发出写操作，这是一个异步调用，它指定的完成函数是 MiWriteComplete。

在 MiWriteComplete 函数中，这些页面的 PFN 项的写标志位被清除，并且物理页面被转移到备用链表中，这是间接地调用了 MiDecrementReferenceCount 函数而完成的。MiWriteComplete 函数的代码位于 modwrite.c 文件的 2 259~2 750 行。

MiGatherMappedPages 函数检查 MmModifiedPageListHead 链表中的页面，并检查与之相邻的原型 PTE，以期尽可能多地聚集多个针对同一个映射文件的已修改页面。一旦建立起这样的页面聚集群，将它插入到链表 MmMappedPageWriterList 中，并设置事件 MmMappedPageWriterEvent 有信号，从而通知映射页面写出器线程来完成实际的写出动作。MiGatherMappedPages 函数的代码见 modwrite.c 文件的 3 400~3 833 行。

了解了修改页面写出器的工作流程以后，接下来我们转到映射页面写出器线程。其主函数 MiMappedPageWriter 位于 modwrite.c 文件的 4 335~4 487 行，它的主循环在等待 MmMappedPageWriterEvent 事件。我们刚刚在 MiGatherMappedPages 函数中已经看到该事件的触发情形了。因为页面已经聚集好了，所以，它只需从 MmMappedPageWriterList 链表中取出一项，然后调用 IoAsynchronousPageWrite 函数发出 I/O 请求。完成函数同样是 MiWriteComplete，其处理流程如前所述。

从修改页面写出器的这两个线程的分工可以看出，MiModifiedPageWriterWorker 函数以及辅助函数 MiGatherPagefilePages 和 MiGatherMappedPages 承担了几乎所有的处理工作，而 MiMappedPageWriter 只是承担了发 I/O 请求的动作而已，当然映射页面的写完成过程也是在这个线程中进行的。之所以将映射页面的写过程移到一个单独的线程中，是为了防止死锁。请考虑这样的情形：假设映射页面的写操作引起页面错误，而在解决页面错误过程中又需要一个空闲页面，但恰好这时候没有空闲页面。如果映射页面的写过程也在修改页面写出器的线程中执行，则死锁就会发生；而把映射页面的写过程分离出来，它就不会阻塞页面文件的写出操作了。

4.5.5 进程/栈交换器

内存管理器在初始化时，还创建了另外一个称为进程/栈交换器（process/stack swapper）的线程。该线程的主函数是 KeSwapProcessOrStack，其优先级为 23，它的主要任务是换出内核栈、换出进程、换入进程和换入内核栈。该线程每 4 s（对于小内存系统，即系统物理内存不超过 19 MB）或 8 s（若物理内存超过 19 MB）被唤醒一次，然后判断是否有符合条件的内核栈或进程需要换出或换入。下面我们来介绍进程/栈交换器的工作过程。

KeSwapProcessOrStack 函数的代码位于 base\ntos\ke\balmgr.c 文件的 328~435 行。在循环中等待的事件 KiSwapEvent 在以下情形下被触发：

- 内存管理器的平衡集管理器每隔 4 个或 8 个检查周期（1 s），就会设置 KiSwapEvent 事件的信号，见 KeBalanceSetManager 函数的代码。关于平衡集管理器，将在 4.6.2 节进一步介绍。
- 有关线程和进程的一些操作，比如终止线程、线程脱离（detach）一个进程、就绪一个线程等情形，若导致有必要换出或换入进程或内核栈，也会立即触发 KiSwapEvent 事件。

KeSwapProcessOrStack 函数分别调用 4 个辅助函数来完成内核栈换出、进程换出、进程换入和内核栈换入这 4 个子任务。这 4 个辅助函数是：KiOutSwapKernelStacks、KiOutSwapProcesses、KiInSwapProcesses 和 KiInSwapKernelStacks，其代码均在 base\ntos\mm\balmgr.c 文件中。

KeSwapProcessOrStack 首先执行内核栈换出任务。KiOutSwapKernelStacks 函数每次被调用，只检查一个处理器上正在等待的线程。这是简单的轮循，变量 KiLastProcessor 指向下次要检查的处理器的编号。处理器控制区 KPRCB 对象中维护了当前正在等待的线程链表，这些等待线程差不多被按照时间先后顺序插入到链表中。KiOutSwapKernelStacks 函数从链表中提取出至多 5 个满足条件（等待时间已超过 15 s，若小内存系统则为 3 s）的线程，然后从等待链表中移除。如果线程所属进程的所有线程都被换出（即进程对象的 StackCount 为 0），则该进程也被放到进程换出链表 KiProcessOutSwapListHead 中。

选出了要换出内核栈的线程以后，对各个线程调用 MmOutPageKernelStack 函数，而 MmOutPageKernelStack 又调用 MiOutPageSingleKernelStack 函数，将内核栈的页面变成转移状态，后者又调用 MiDecrementShareCount 函数（进而调用 MiDecrementReferenceCount 函数），将页面的 PFN 项插入到备用链表或修改链表中。MmOutPageKernelStack 和 MiOutPageSingleKernelStack 函数的代码位于 base\ntos\mm\procsup.c 文件中。

其次，KeSwapProcessOrStack 调用 KiOutSwapProcesses 函数执行进程换出任务。KiOutSwapProcesses 接受 KiProcessOutSwapListHead 链表作为参数，它依次检查该链表中的每一个进程。这些进程的内核栈都已换出，只要该进程的就绪线程链表 ReadyListHead 成员为空，就将进程的状态置于 ProcessOutSwap，并调用 MmOutSwapProcess 函数执行换出动作。该函数返回后，若进程的就绪线程链表仍然为空，则置进程的状态于 ProcessOutOfMemory，表明该进程已不在内存中。

只有当一个进程的工作集已经降低到最小（在 Intel x86 系统上为 4 个页面）的时候，该进程才会在 MmOutSwapProcess 函数中被换出，否则，MmOutSwapProcess 直接返回，以后工作集管理器会修剪该进程的工作集。当最终修剪到最小工作集尺寸时，该进程会再次被触发换出操作。

MmOutSwapProcess 函数依次换出 4 个页面：工作集链表页面、VAD 位图页面、超空间页表页面和页目录页面。它定位到这些页面的 PTE，将 PTE 变成转移状态，然后针对它们的 PFN 项调用 MiDecrementShareCount，以便将页面转移到备用或修改链表中。因此，我们可以看到，所谓进程换出，实际上只换出这 4 个基本页面（并且从代码中可以看

出，页目录页面的 PTE 虽然被设置成转移状态，但它的 PFN 项仍然处于 ActiveAndValid 状态，并没有被插入到备用或修改链表中），进程的其他页面是通过工作集修剪或内核栈换出的途径来完成的。MmOutSwapProcess 函数的代码位于 base\ntos\mm\procsup.c 文件中。

接下来我们看一下执行进程换入任务的 KiInSwapProcesses 函数。要换入的进程构成了一个链表 KiProcessInSwapListHead。这很容易理解，对于已被换出的进程，只要它有一个线程就绪，就会被加入到该链表中。KiInSwapProcesses 依次检查该链表中的每一个进程，并调用 MmInSwapProcess 函数执行实际的换入操作。

MmInSwapProcess 是 MmOutSwapProcess 函数的逆过程，它依次恢复 4 个页面：目标进程的页目录页面、超空间页表页面、VAD 位图页面以及工作集链表页面，这些页面的恢复是通过调用 MiMakeOutswappedPageResident 函数来完成的。MmInSwapProcess 和 MiMakeOutswappedPageResident 函数的代码均位于 base\ntos\mm\procsup.c 文件中。

最后，我们看一下执行内核栈换入任务的 KiInSwapKernelStacks 函数。要换入内核栈的线程构成了一个链表 KiStackInSwapListHead。不难理解，当一个线程变成就绪时，如果线程 KTHREAD 对象的 KernelStackResident 成员表明这个线程的内核栈不在内存中，则该线程就会加入到 KiStackInSwapListHead 链表中，并且设置 KiSwapEvent 事件的信号，参见 KiReadyThread 的代码（位于 base\ntos\ke\thredsup.c 文件中）。KiInSwapKernelStacks 依次处理 KiStackInSwapListHead 链表中的每一个线程，调用 MmInPageKernelStack 函数执行实际的换入操作。

MmInPageKernelStack 又调用 MiInPageSingleKernelStack 函数来换入每一个独立的内核栈段（线程的内核栈有可能是一条链，链上每个节点都是一段连续的栈空间）。MiInPage-SingleKernelStack 函数针对栈空间中的每个页面调用 MiMakeOutswappedPageResident 函数，使原本已换出的页面变成有效页面。MmInPageKernelStack 和 MiInPageSingleKernelStack 函数的代码也位于 base\ntos\mm\procsup.c 文件中。

至此，我们已经看到了 Windows 的进程/栈交换器是如何工作的。以上介绍的 4 个子任务的顺序也正好是一个进程从“它的所有线程进入等待”，一直到“这些线程逐渐回归执行状态”过程的逻辑顺序。而且，我们也可以看出，进程的换出和换入是一个相对比较极端的行为，并且也只能腾出很有限的物理内存，它只有等到进程工作集中的其他页面（并非以上提到的 4 个页面）都被修剪之后才会发生。因此，在实际系统中，它们很少发生，一旦发生，则常常意味着系统的内存不足以承担起当前的工作负载，而且系统的性能通常会急剧退化。交换器只是保证这些进程能运行，解决不了性能瓶颈问题。有效的解决方案

要么是增加物理内存，要么是减轻工作负载。相反地，进程工作集的修剪过程往往能给系统带来更多的可用内存。在 4.6.1 节中我们将讨论 Windows 是如何管理进程工作集的。

4.5.6 低内存通知和高内存通知

当系统的可用内存变低或者变充裕时，让进程有机会知道这些事件在某些情况下是有意义的，比如，应用程序可以确定它能够处理多大量的数据。Windows 提供了低内存和高内存事件，允许应用程序接收这些通知。

Windows 的内存管理器初始化函数 MmInitSystem 调用 MiInitializeMemoryEvents 函数来完成这些内存事件的初始化。具体做法可参考 MiInitializeMemoryEvents 函数的代码，见 base\ntos\mm\mminit.c 文件的 4 881~5 024 行。低内存事件的触发条件是，可用内存低于一个阈值，即全局变量 MmLowMemoryThreshold。MmLowMemoryThreshold 的值是由系统物理内存的数量来确定的，对于超过 1 GB 内存的系统，第一个 1 GB 为 32 MB，余下每 GB 增加 8 MB，譬如，对于 4 GB 物理内存的系统，MmLowMemoryThreshold 的值为 32 MB+3 × 8 MB=56 MB；对于 128 MB~1 GB 内存的系统，第一个 128 MB 为 3.1 MB（即 800 个页面），余下每 128 MB 增加 4 MB；对于少于 128 MB 内存的系统，MmLowMemoryThreshold 的值等于 MmPlentyFreePages，当内存小于 63 MB 时为 400 个页面，否则为 800 个页面。MmLowMemoryThreshold 的具体算法见 mminit.c 文件的 4 915~4 927 行。高内存事件的阈值 MmHighMemoryThreshold，除非应用系统在注册表中指定，否则为低内存事件阈值 MmLowMemoryThreshold 的 3 倍。

低内存事件和高内存事件的系统变量分别是 MiLowMemoryEvent 和 MiHighMemoryEvent。当在空闲链表、零化链表或备用链表中插入一个页面时，若系统可用页面数 MmAvailable-Pages 等于低内存阈值 MmLowMemoryThreshold，则低内存事件 MiLowMemoryEvent 被清除信号；若 MmAvailablePages 等于高内存阈值 MmHighMemoryThreshold，则高内存事件 MiHighMemoryEvent 被置上信号。相反地，当从空闲链表、零化链表或备用链表中移除一个页面时，若 MmAvailablePages 等于 MmLowMemoryThreshold，则低内存事件 MiLowMemoryEvent 被置上信号；若 MmAvailablePages 等于 MmHighMemoryThreshold，则高内存事件 MiHighMemoryEvent 被清除信号。

应用程序可以通过系统名字空间中的名称“\KernelObjects\LowMemoryCondition”和“\KernelObjects\HighMemoryCondition”来监听这两个事件，从而在内存变低或变充裕时获得通知。除了这两个全局的低内存或高内存通知事件以外，在 MiInitializeMemoryEvents 函数中还看到过针对换页内存池和非换页内存池的低内存和高内存通知事件，这 4 个事件

的工作原理与前面介绍的 MiLowMemoryEvent 和 MiHighMemoryEvent 事件类似，读者可以自行从 WRK 代码中检查它们的阈值设置以及通知条件。

4.6 工作集管理

4.1.3 节提到过，工作集是指一个进程当前正在使用的物理页面的集合。在 Windows 中，除了进程工作集以外，另有两个工作集概念：系统工作集指 System 进程（由全局变量 PsInitialSystemProcess 表示）的工作集，其中包含系统模块的映像区（比如 ntoskrnl.exe 和设备驱动程序）、换页内存池和系统缓存（cache）；会话工作集指一个会话所属的代码和数据区，包括 Windows 子系统用到的与会话有关的数据结构、会话换页内存池、会话的映射视图，以及会话空间中的设备驱动程序。这一节我们讨论 Windows 对进程工作集的管理。

4.6.1 Windows 工作集管理器

在介绍 Windows 工作集管理器以前，我们先看一看进程是如何维护其工作集信息的。每个进程都有一个专门的页面用来存放它的工作集链表，这是在创建进程地址空间时已经映射好的一个页面，其地址位于全局变量 MmWorkingSetList 中，所以，在任何一个进程中，通过变量 MmWorkingSetList 就可以访问到它的工作集链表。MmWorkingSetList 的类型为 MMWSL，其定义如下：

```
typedef ULONG WSLE_NUMBER, *PWSLE_NUMBER;

typedef struct _MMWSL {
    WSLE_NUMBER FirstFree;
    WSLE_NUMBER FirstDynamic;
    WSLE_NUMBER LastEntry;
    WSLE_NUMBER NextSlot;             // The next slot to trim
    PMMWSLE Wsle;
    WSLE_NUMBER LastInitializedWsle;
    WSLE_NUMBER NonDirectCount;
    PMMWSLE_HASH HashTable;
    ULONG HashTableSize;
    ULONG NumberOfCommittedPageTables;
    PVOID HashTableStart;
    PVOID HighestPermittedHashAddress;
    ULONG NumberOfImageWaiters;
    ULONG VadBitMapHint;
    USHORT UsedPageTableEntries[MM_USER_PAGE_TABLE_PAGES];
    ULONG CommittedPageTables[MM_USER_PAGE_TABLE_PAGES/
        (sizeof(ULONG)*8)];
} MMWSL, *PMMWSL;
```

其中 Wsle 成员是一个指针，实际上指向一个数组，其成员类型为 MMWSLE，定义如下：

```
typedef struct _MMWSLENTRY {
    ULONG_PTR Valid : 1;
```

```
    ULONG_PTR LockedInWs : 1;
    ULONG_PTR LockedInMemory : 1;
    ULONG_PTR Protection : 5;
    ULONG_PTR Hashed : 1;
    ULONG_PTR Direct : 1;
    ULONG_PTR Age : 2;
    ULONG_PTR VirtualPageNumber : 20;
} MMWSLENTRY;

typedef struct _MMWSLE {
    union {
        PVOID VirtualAddress;
        ULONG_PTR Long;
        MMWSLENTRY e1;
    } u1;
} MMWSLE;
```

每个 MMWSLE 都描述了一个有效页面，从它的定义可以看出，其中低 12 位为 MMWSLENTRY 中定义的标志位，高 20 位表示页面的虚拟地址。因此，我们现在可以知道，在 MMPFN 中定义的 WsIndex 成员（见图 4.27(a)）可以用来立即定位到一个 MMWSLE 项。

Windows 除了通过 MmWorkingSetList 来跟踪一个进程所使用的物理页面以外，它还为每个进程定义了工作集的最小值和最大值。工作集的最小值是在进程创建之初设定的，MmCreateProcessAddressSpace 函数的第一个参数即工作集最小值，如 4.3.1 节所述。在 WRK 中，默认的最小值为 50（对于超过 32 MB 内存的系统），最大值为 345。进程的工作集最大值和最小值只是在页面调度时的一个指导，它们可以被突破，比如当进程使用很多页面并且系统也确实有足够的可用页面时，进程工作集可以超越所设定的最大值。但是进程工作集不能超过系统设定的一个全局最大值，即变量 MmMaximumWorkingSetSize，它是在系统初始化时设定的，该值等于可用物理内存减掉 512 个页面，并且不得超过 2 GB~84 KB 大小，见 base\ntos\mm\mminit.c 文件的 2 076~2 080 行。另外，当系统内存紧缺，而且一个进程并不十分需要页面时，它的工作集可能会降低到最小值以下。

在 4.5 节介绍物理内存管理时，我们提到，内存管理器包含一个工作集管理器，它会定期检查各个进程的工作集，必要时修剪占用内存较多的进程的工作集，以便更加有效地使用物理内存。在进程 EPROCESS 数据结构中有一个 Vm 成员，其类型为 MMSUPPORT，它包含了工作集管理器在判断是否需要修剪一个进程时所需的各种信息，其中包括为该进程设定的工作集最小值和最大值。MMSUPPORT 的定义如下：

```
typedef struct _MMSUPPORT {
    LIST_ENTRY WorkingSetExpansionLinks;
    LARGE_INTEGER LastTrimTime;

    MMSUPPORT_FLAGS Flags;
    ULONG PageFaultCount;
    WSLE_NUMBER PeakWorkingSetSize;
```

```
    WSLE_NUMBER GrowthSinceLastEstimate;

    WSLE_NUMBER MinimumWorkingSetSize;
    WSLE_NUMBER MaximumWorkingSetSize;
    struct _MMWSL *VmWorkingSetList;
    WSLE_NUMBER Claim;

    WSLE_NUMBER NextEstimationSlot;
    WSLE_NUMBER NextAgingSlot;
    WSLE_NUMBER EstimatedAvailable;
    WSLE_NUMBER WorkingSetSize;

    EX_PUSH_LOCK WorkingSetMutex;

} MMSUPPORT, *PMMSUPPORT;
```

工作集管理器的主函数是 MmWorkingSetManager，它运行在内存管理器的平衡集管理器线程（KeBalanceSetManager 函数）中。MmWorkingSetManager 函数每隔 1 s 被触发一次，或者当可用内存降低到足够少的时候也会被触发。工作集管理器根据一定的准则来修剪进程的工作集。MmWorkingSetManager 调用 MiComputeSystemTrimCriteria 函数来确定是否需要启动修剪工作集或刷新工作集年龄的工作，以及如何安排被修剪的进程。MmWorkingSetManager 的代码位于 base\ntos\mm\wsmanage.c 文件的 594~677 行；MiComputeSystemTrimCriteria 的代码位于 base\ntos\mm\wsmanage.c 文件的 679~863 行。然后，MmWorkingSetManager 调用 MiProcessWorkingSets 函数执行实际的修剪或年龄刷新工作。MiProcessWorkingSets 的代码位于 base\ntos\mm\wsmanage.c 文件的 968~1 262 行。

如果一个进程允许被修剪，则它的 EPROCESS 的 Vm.WorkingSetExpansionLinks 成员将被加入到一个以全局变量 MmWorkingSetExpansionHead 为链表头的链表中。也就是说，以 EPROCESS 的 Vm.WorkingSetExpansionLinks 为节点构成了一个链表，链表头为 MmWorkingSetExpansionHead。

在 MiComputeSystemTrimCriteria 函数中，如果当前可用内存小于一定的数量（全局变量 MmPlentyFreePages，被初始化为 400，若系统内存超过 63 MB，则调整为 800，通过 MmIsMemoryAvailable 函数可以动态调整此阈值），或者最近被重新使用的物理页面超过了当前可用页面的 1/4，则调用 MiRearrangeWorkingSetExpansionList 函数来安排如何有效地修剪工作集，并返回 MI_TRIM_ALL_WORKING_SETS；否则，若当前可用内存足够使用，但小于 MM_ENORMOUS_LIMIT（等于 20 000），则返回 MI_AGE_ALL_WORKING_SETS。MiRearrangeWorkingSetExpansionList 函数把允许修剪的工作集，即 MmWorkingSetExpansionHead 链表中的工作集，分成不同的类别，然后按照修剪优先级重新排序。基本的原则是，进程 EPROCESS 的 Vm->Claim 越大，越放在修剪链表的前部；进程的空闲时间越长，越放在链表的前部；前台进程放在最后。Vm->Claim 是工作

集管理器对于进程工作集的一个评估值，它对进程工作集中的页面进行采样，统计它们的老化情况（即年龄，MMWSLE 的 u1.e1.Age 域），评估结果大致上反映了当系统内存紧张时，一个进程可以为系统提供多少页面的内存。关于 Vm->Claim 的评估算法，请参考 MiAgeWorkingSet 的代码，位于 base\ntos\mm\wsmanage.c 文件的 1 542~1 841 行。MiRearrangeWorkingSetExpansionList 函数的代码位于 base\ntos\mm\wsmanage.c 文件的 1 940~2 165 行。

MiProcessWorkingSets 是真正实施修剪工作或刷新工作集年龄的函数。它的核心逻辑是，针对 MmWorkingSetExpansionHead 链表中的每一个工作集，附载（attach）到该进程中，并根据参数中的指示（即 MiComputeSystemTrimCriteria 函数的返回值）进行以下处理：

- 若要求修剪所有工作集，即 MI_TRIM_ALL_WORKING_SETS，则调用 MiDetermineTrimAmount 确定是否需要修剪该进程工作集，以及修剪多老的页面，是否需要重新计算页面年龄，并返回可修剪页面的数量。然后，若确定需要对该进程进行修剪，则调用 MiTrimWorkingSet 函数执行修剪工作，在参数中指定：期望从工作集中修剪多少页面，工作集信息，以及多老的页面可被修剪。修剪完成以后，调用 MiAgeWorkingSet 函数更新工作集中页面的年龄值，并估计该进程工作集的 Vm->Claim 值。
- 若只是要求更新工作集中页面的年龄值，即 MI_AGE_ALL_WORKING_SETS，则简单地调用 MiAgeWorkingSet 函数更新页面年龄值，并估计该进程工作集的 Vm->Claim 值。

判断工作集是否要修剪以及如何修剪是一个动态过程，不仅每个工作集的状态在发生变化，系统也会随着可用内存的变化而调整。此过程中涉及较多参数，这里不一一解释，有兴趣的读者可以参考 wsmanage.c 文件中的代码。

工作集中页面的年龄计算

对于工作集中的每一个页面，其 MMWSLE 结构中的 u1.e1.Age 域记录了这个页面的年龄。由于该域只有两位，所以只能表达 0、1、2、3，其计算方法如下：

(1) 判断页面 PTE 的“访问过”位（见图 4.23）是否已置位，如果置位了，说明这个页面自上次检查以后已被访问过，于是，年龄值为 0，并清除该位。

(2) 如果“访问过”位未被置上，则说明自上次检查以来该页面未被访问过，于是，年龄值加 1，最大不超过 3。

这种年龄计算方法充分利用了 Intel x86 硬件提供的 PTE“访问过”位，当处理器一

旦访问了一个页面，则 PTE 中的这一位就会被置位，但处理器不会自动清除该位，这需要由软件来完成。工作集管理器在每个周期中只扫描一部分页面，以期望在 16 s 内完成对整个工作集的一遍扫描，但实际上，受到各种动态因素的变化影响，每次计算都会有一些出入和调整。然而，这种计算方法大致上反映了页面的老化程度，最近访问过的页面总是有比较小的年龄值，久未访问的页面的年龄值总是会达到最大值 3。MiAgeWorkingSet 函数的 1 665~1 674 行代码实现了上面的年龄算法。

在 MiProcessWorkingSets 的循环中，它有可能遍历 MmWorkingSetExpansionHead 链表中的工作集不止一遍；当遍历该链表回头时，它调用 MiCheckSystemTrimEndCriteria 函数以确定是否终止循环。如果经过上一轮修剪以后，可用内存达到了预期的目标，则允许结束修剪循环，否则继续下一轮循环。MiDetermineTrimAmount 函数在确定从一个进程的工作集中应该修剪多少页面以及设定年龄界限时，既考虑了进程的 Vm->Claim 值（或 Vm->EstimatedAvailable），也考虑了当前已进展到第几轮循环。MiDetermineTrimAmount 函数的实现较为简单，参见 wsmanage.c 文件的 1 265~1 427 行。

MiTrimWorkingSet 函数的实现非常直截了当，它从进程的工作集链表中找出一批符合条件的（年龄值大于等于参数设定的修剪年龄）页面，并调用 MiFreeWsleList 函数，由该函数将满足条件的页面（譬如该进程的私有页面）变成转移状态，并转移到备用链表或修改链表中，然后调用 MiRemoveWsle 函数将这些页面从工作集链表中移除。MiTrimWorkingSet 函数如此循环往复，直到指定数量的页面被修剪掉，或者扫描了工作集链表的整个动态区域。由于进程的工作集链表是数组形式的链表（MMWSL 中的 Wsle 成员），MiFreeWsleList 函数只是将被移除的页面在该数组中的项做了无效标记，所以，MiTrimWorkingSet 函数在完成了上述修剪工作以后，如果发现在工作集链表中存在较多无效项目，则调用 MiRemoveWorkingSetPages 压缩工作集链表，清除掉无效的数据项。

最后我们看一下，进程的工作集链表是如何组织的，以及如何根据一个页面的虚拟地址或 PTE 找到它在工作集链表中的项。在 MMWSL 数据结构中，Wsle 成员指向 MMWSL 对象的尾部，所以，工作集链表中的 MMWSLE 数组紧跟在 MMWSL 对象的后面。MMWSL 中的 FirstDynamic 成员指向第一个可被修剪的页面，所以，我们在 MiTrimWorkingSet 函数中看到，工作集修剪总是从 FirstDynamic 项开始，而在 FirstDynamic 之前的表项都被锁定在内存中，它们的 LockedInWs 或 LockedInMemory 标志已置上；FirstFree 成员指向一个空闲链表的头，由于工作集链表中的项可能会随着页面从工作集中移除而空出来，这些项在内部构成了一个单链表，FirstFree 指向此链表

的头，因而在工作集中加入一个页面可以很容易地找到一个空闲数组项；LastEntry 成员指向最后一个使用的项，因此 MiTrimWorkingSet 在修剪工作集时绝对不会超过 LastEntry 项。图 4.30 显示了工作集链表 MMWSL 对象的内部结构。

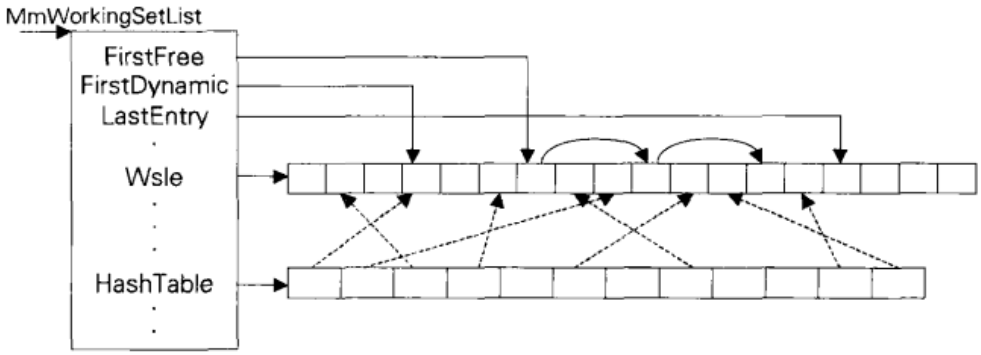


图 4.30 工作集链表 MMWSL 对象的内部结构

MMWSL 还使用了一个散列表，即它的 HashTable 成员，以加速 WSLE 的查找过程。散列表的位置在 Wsle 数组的后面，假设中间预留了最大可能的工作集链表（由 MiMaximumWorkingSet 定义，在 WRK 中为 2 GB~64 KB）。因此，Wsle 和 Hashtable 位于进程地址空间的固定位置处，它们的空间是直接以页面映射而并非通过内存池来管理的，而且它们所占据的页面本身也算在工作集之内。

以下是散列表成员项 MMWSLE_HASH 的定义：

```
typedef struct _MMWSLE_HASH {
    PVOID Key;
    WSLE_NUMBER Index;
} MMWSLE_HASH, *PMMWSLE_HASH;
```

每一项都指定了一个从页面虚拟地址到工作集链表数组索引的关联。工作集链表中的散列表使用简单的模运算作为从页面虚拟地址到桶的映射，并使用线性开放定址法（Open Addressing with Linear Probing）[TAOCP-3]找到空闲的散列桶。从一个虚拟地址到关键字的映射如下所示：

```
#define MI_WSLE_HASH(Address, Wsl) \
   ((WSLE_NUMBER)(((ULONG_PTR)PAGE_ALIGN(Address) >> (PAGE_SHIFT - 2)) % \
     ((Wsl)->HashTableSize - 1)))
```

利用线性开放定址法来插入散列项的做法是，首先根据以上公式计算出关键字 Key，然后在散列表数组中，从 Key 这一项开始逐一往后寻找，直到找到一个空闲项，然后在这一项中插入该页面所对应的散列项。如果往后寻找过程中到达散列表尾部，则绕回头从表头开始继续寻找；如果又回到起始处，则说明散列表已满，无法再插入。MiGrowWsleHash

函数（位于 base\ntos\mm\wslist.c 文件中）提供了散列表扩展的功能，注意，散列表扩展后必须重新插入所有的项。MiFillWsleHash 函数（位于 base\ntos\mm\wstree.c 文件中）实现了在一个空表中插入工作集链表中页面信息的过程。在散列表中查找一个地址的做法比较直截了当，根据地址计算得到 Key，然后在散列表中从 Key 索引项开始，逐一查找包含 Key 的 MMWSLE_HASH 项。如果绕回头还不能找到，则说明散列表不包含这一项。

现在我们再来看一下 MMWSLE 中几个标志位的含义：u1.e1.Valid 说明这一项是有效还是无效；u1.e1.Direct 说明这是一个直接映射项，即页面的 PFN 项中的 WsIndex 直接指向此项的索引。与 u1.e1.Direct 位相关联的是，MMWSL 中的 NonDirectCount 计数值表示工作集链表中无直接映射的项的数量。另外，u1.e1.Hashed 表示这一项是否已被插入到散列表中。

了解了工作集链表和散列表的组织结构以后，我们再来看一下如何根据一个页面的虚拟地址定位到工作集链表中的描述项，这是 MiLocateWsle 函数（位于 base\ntos\mm\wstree.c 文件中）的任务。它首先根据参数中提示的索引项，判断是否该索引项指向此虚拟地址。如果是，则成功找到工作集链表项，并返回。否则，利用参数中指定的虚拟地址在散列表中查找，若找到，则返回散列表项中的 Index。如果在散列表中找不到，则在工作集链表中进行线性搜索，直到找到为止，若还是不能找到，则失败返回（在 WRK 中是一个 KeBugCheckEx）。

因此，从这一搜索过程可以看出，PFN 数据库项中的 WsIndex 成员是一个指向工作集链表的快速索引，但是，若一个页面被多个进程共享，那么，WsIndex 成员仅仅对第一个使该页面有效的进程的工作集有效。散列表的目的是避免线性搜索，它建立起虚拟地址和工作集链表项之间的映射关系。

4.6.2 平衡集管理器

平衡集管理器（balance set manager）是在系统初始化时创建的，它是一个系统线程，其优先级为 16，最低的实时优先级，因此比普通的非实时线程的优先级高。平衡集管理器的实际用途是维持系统内存资源的平衡，当多个进程发生资源竞争时，平衡集管理器负责从系统全局来调度可用的物理内存以及调整每个进程的工作集。

平衡集是指所有具备资源分配资格的进程的集合，当然也包括 System 进程。当系统内存紧缺时，它一方面把拥有较多内存的进程从它们的工作集中换出一些页面，另一方面把不满足运行条件的进程排除在平衡集以外。这些被排除在平衡集以外的进程，只有当其

中的线程满足运行条件并且系统有了足够空闲内存时，才会被重新加入到平衡集中。这是平衡集的概念基础。

在 Windows 的内存管理器中，平衡集管理器的功能主要体现在工作集管理器所实现的修剪工作集的能力，以及 4.5.5 节所介绍的进程/栈交换器上。从实现角度来看，平衡集管理器基本上只是工作集管理器的一个容器而已，它为工作集管理器提供了一个执行环境。另外，它也每隔 4 s 或 8 s 启动进程/栈交换器。

平衡集管理器的主函数是 KeBalanceSetManager，其代码位于 base\ntos\ke\balmgr.c 文件的 142~326 行，它除了调用 MmWorkingSetManager 和向进程/栈交换器线程发送事件以外，还每秒钟调用 ExAdjustLookasideDepth 和 KiAdjustIrpCredits 函数。关于这两个函数，这里不再介绍。

4.7 内存监视工具 MemMon

MemMon 是一个内存监视工具，它能够实时地显示当前系统中内存的使用情况，包括系统地址空间的布局、各个物理内存链表的页面数量，以及每个进程的虚拟地址空间布局和工作集信息等。

4.7.1 MemMon 使用介绍

图 4.31 显示了 MemMon 程序在一个具有 512 MB 内存的 Windows Server 2003 SP1 系统上运行一段时间后的屏幕截图。左上角的两条曲线分别显示了当前系统中物理内存和页面文件中页面使用量的百分比。左下角有 4 个状态显示区：物理内存区域显示了当前系统中物理内存总量、可用物理内存数量，以及系统缓存数量；页面文件区域显示了当前系统中可提交的内存总量，以及还有多少可以提交；内核内存区域显示了换页内存池和非换页内存池的内存数量，以及两者之和；物理内存链表区域显示了 4.5.3 节中提到的内存链表。图中右侧的图表显示了系统地址空间的内存布局结构。

图 4.32 显示了有关进程的内存状态信息。界面的左上角显示了当前系统中的进程树。左下角的虚拟内存区域显示了所选进程的页面错误次数、虚拟内存大小、提交页面数等信息；工作集区域显示了所选进程的当前工作集大小，以及工作集最小值、最大值和尖峰值。正如 4.6.1 节所讲，工作集最大值和最小值只是一个指导，实际的工作集大小可以突破这些限制。图中右侧的图表显示了所选进程的进程地址空间布局结构，其中列出了该进程中的

模块、堆和栈的信息。当用户在进程树中选择不同的进程时，界面中其他的区域也会相应地变化。即使用户不进行任何操作，MemMon 也会每隔 1 s 刷新当前进程的内存状态信息。

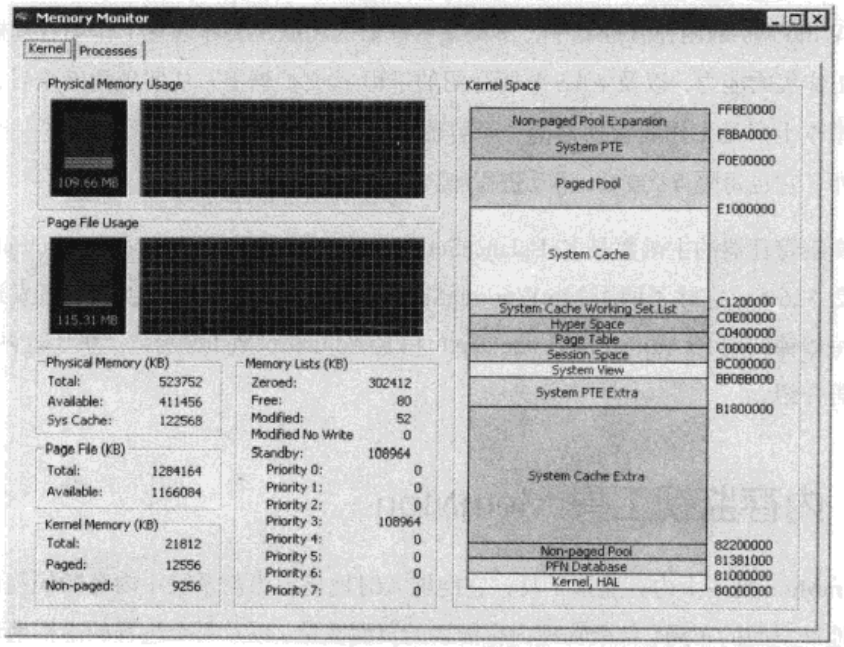


图 4.31 显示系统内存信息的 MemMon 界面

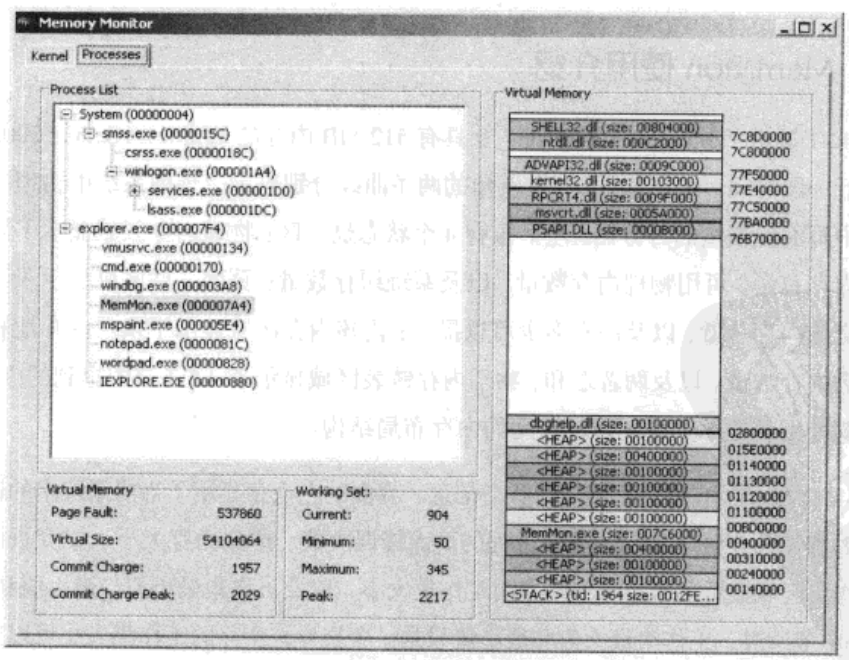


图 4.32 显示进程内存信息的 MemMon 界面

4.7.2 MemMon 实现原理

MemMon 的实现并不复杂，无论是系统内存界面还是进程内存界面，都会每隔 1 s 刷新一次。实现 MemMon 的关键在于，界面上的状态信息是如何获得的。下面简单介绍一下 MemMon 显示的各种信息的来源。

在系统内存界面部分，MemMon 从两方面收集有关的信息：

(1) 通过 Windows 的 API 函数 GetPerformanceInfo 和 GlobalMemoryStatusEx 可以获得当前系统中有关物理内存、页面文件和内核内存的信息，甚至还可以获得当前系统中的句柄数、进程数和线程数等信息。这两个 API 函数分别位于 psapi.dll 和 kernel32.dll 中，在 Windows XP 及以后的系统中都可以使用。

(2) 通过一个设备驱动程序，利用内核模块的符号信息，直接读取内核变量的值，包括物理内存链表变量和系统地址空间中各个区域的位置信息（如图 4.14 所示）。

在进程内存界面部分，MemMon 也从两方面收集相关的信息：

(1) 通过 Windows 的 API 函数 Module32First 和 Module32Next 可以获得有关一个进程的模块的信息；通过 Heap32First 和 Heap32Next 可以获得有关堆的信息；通过 Thread32First 和 Thread32Next 可以获得有关线程的信息。注意，在调用这组函数以前，必须首先调用 CreateToolhelp32Snapshot 函数。这些函数位于 kernel32.dll 中，在 Windows 2000 及以后的系统中都可以使用。

(2) 通过设备驱动程序，获得进程 EPROCESS 对象中的有关信息，包括指定进程的虚拟内存和工作集信息。为了获得一个进程中的线程的栈信息，MemMon 也需要通过驱动程序来访问线程的 KTHREAD 结构。

因此，为了能够显示系统或进程的内存信息，MemMon 必须加载一个设备驱动程序，而且，还要有符号信息才能获得全局变量的值以及解析内核中的数据结构。如果内核的符号信息不可用，那么，MemMon 只能显示一部分内存状态信息。

4.8 本章总结

在本章中，我们看到了 Windows 内存管理器的多个方面：如何在系统空间中有效地管理可换页的内存和不可换页的内存；如何为进程建立起独享的地址空间，并实现共享内存和私有页面的共存；系统如何解决页面错误；如何管理物理内存；以及进程的工作集管

理。这些内容构成了 Windows 内存管理器的基础，但并非全部。

内存管理器一方面有效地管理有限的资源并降低管理开销，另一方面还要保持快速高效。Windows 是一个通用的操作系统，它需要兼顾桌面操作系统和服务器操作系统两方面的需求，所以，我们在查看 WRK 中的代码时会看到大量的参数设置，通常是一些全局变量。而且，它也要兼顾到多级页表管理以及多处理器系统的差异性和复杂性，所幸的是，Windows 内存管理器的框架结构有足够的灵活性，能很好地适应这些不同的需求和体系结构的差异性。

除了本章中介绍的这些机制以外，Windows 内存管理器还内置了各种优化设施，以及对于传统 UNIX 系统的内核支持。比如，逻辑预取器（Logical Prefetcher）就是一种典型的性能优化，它针对系统引导或应用程序启动时按需换入页面性能较差的问题，记录下哪些页面需要在进程启动时直接读入进来，而不是等到发生页面错误时再读取这些页面。另外，Windows 内存管理器也提供了 UNIX 系统中常用的 fork 原语，尽管 Windows 子系统并没有提供这样的编程接口。本章仅介绍了写时复制特性，而没有讲到对进程 fork 原语的支持。

另外有一些话题跟内存管理紧密相关，但为了清晰起见，本章基本上忽略了对这些内容的讨论。它们包括：缓存管理，这是与内存管理紧密相关的一个特性，在第 7 章中将有关于缓存管理器的进一步讨论；文件 I/O 和页面文件的管理，页面的换入换出都离不开文件 I/O，在第 6 章中将有关于 Windows I/O 模型的介绍，在第 7 章中还有关于文件系统的介绍；会话空间的内存管理和用法，本章仅仅讨论了会话空间的基本布局结构。此外，在解读内存管理函数时，我们看到了很多同步对象和锁的用法，下一章我们将介绍 Windows 中的并发和同步支持。

5

Windows 并发和同步

在一个支持多线程、多进程的操作系统中，并发和同步是两个重要的概念。并发是指多个控制流同时在执行，既可能是真正的并行执行（例如在多处理器或多核的硬件环境中），也可能是分时方式的并发执行；而同步则是保证在并发执行的环境中各个控制流可以有序地执行，包括对于资源的共享或互斥访问，以及代码功能的逻辑顺序。

在 Windows 内核中，有多种机制可用于保证操作系统和应用程序有序地执行。系统中的有些数据结构必须保证严格的一致性，比如我们在前两章中看到的一些全局链表；有些功能只有在特定条件下才可以发生，比如，系统在执行某些中断任务时不能引发页面错误。Windows 既利用了处理器提供的并发和同步能力，又设计了一些逻辑机制来保证系统和进程协调前进。

在这一章中，我们首先介绍现代操作系统中关于并发和同步的基础知识，并总体介绍 Windows 中涉及同步的各种机制。然后分别从中断和异常、不依赖于线程调度的同步机制，以及基于线程调度的同步机制等角度来讨论 Windows 中有关这些机制的实现。最后介绍一个可观察和学习线程间同步关系的工具 DPerfLite。

5.1 进程和线程的同步基础

在像 Windows 这样的多线程环境中，如果各个线程始终都具备执行条件，那么，操作系统在调度这些线程时，将根据优先级顺序将处理器资源分派给它们，因而它们总可以在适当时候完成其任务。这些线程相互之间唯一的制约因素是，高优先级的线程会阻止低优先级的线程获得处理器资源。在这种情况下，线程的执行与否完全取决于它的优先级。

然而，在实际的系统环境中，无论是在内核模式下，还是在应用程序环境中，多线程的情形都要复杂得多，每个线程都或多或少地依赖于其他的线程，或者依赖于系统的行为或状态。譬如，一个线程只有在满足特定条件的情形下才可以继续执行，这一条件必须要通过操作系统提供的同步原语来表达；线程可能要读写外部设备，这时它必须利用系统提供的设备通信能力来跟设备打交道。这一节我们将讨论一些经典的同步算法以及同步问题。

5.1.1 并发性基础

多个进程或线程并发执行，其复杂性在于，一旦它们相互之间有依赖关系，则它们的交互过程或计算步骤的顺序将可能存在不确定性，从而使各个进程或线程的正确性难以保证和分析。通常，操作系统会提供一些同步原语供这些进程或线程使用，从而限制它们的计算必须按照特定的约束来进行。另一方面，程序设计语言也可能会提供一些语法要素来保证数据结构的一致性或代码片段的不可重入特性。

我们首先来看一个由于并发性而导致问题的简单程序。图 5.1 显示了一个计算线程和一个控制线程。计算线程根据全局变量 g_nCount 的值来循环执行计算任务，当 g_nCount 达到 100 时，循环结束，从而计算线程退出。控制线程在创建了计算线程以后，在它终止以前，可以通过将 g_nCount 赋值为 100 的方式通知计算线程不必再执行计算任务了。因此，计算线程和控制线程通过一个全局变量进行通信。

```
extern int g_nCount;

int ComputeThreadProc(void *pParam)
{
    ......
    g_nCount = 0;

    while(g_nCount++ < 100)
    {
        // 执行计算任务
        ......
    }
    return 0;
}
```

```
mov eax, [g_nCount]
mov ecx, [g_nCount]
add ecx,1
mov [g_nCount], ecx
cmp eax,64h
```

(a) 计算线程

```
extern int g_nCount;

int ControlThreadProc(void *pParam)
{
    ......
    CreateThread(ComputeThreadProc); // 创建线程
    ......
    // 通知计算线程结束

Terminated:
    ......
    g_nCount = 100;
    return 0;
}
```

```
mov [g_nCount], 64h
```

(b) 控制线程

图 5.1 两个线程通过一个全局变量进行通信

这两段代码逻辑上看起来没有错，而且运行的时候也许符合预期的效果，但若仔细检查生成的汇编代码，可以发现，如果线程切换发生在图 5.1(a)展示的 5 条汇编指令的第 2 条或第 3 条之后，那么，即使控制线程已经将全局变量 g_nCount 赋值为 100，计算线程

中接下来的 mov 指令也会覆盖掉刚刚赋的值，从而使控制线程中的“g_nCount=100;”这条语句完全不起作用。更有甚者，如果编译优化打开了的话，计算线程的循环过程可能直接通过一个寄存器来控制，而根本不访问 g_nCount 的内存单元。为避免这种优化，可以将 g_nCount 变量声明为 volatile。

从这个例子中我们得到的启示是，两个或多个线程在用全局变量进行通信时，如果对全局变量的操作不是原子的，则可能存在潜在的副作用。为确保不会有这种不确定的副作用，必须要对关于全局变量的操作进行保护，要么保证所有的操作都是原子的，要么确保对变量的访问总是互斥的。这正是在多线程环境中有必要对关于共享数据的访问进行同步的一个重要原因。

在初步了解了多线程并发执行可能引起的问题以后，我们来看一看这种并发性的来源。在现代操作系统环境中，进程或线程的并发性有以下一些可能的来源：

- 多处理器环境。各个处理器有独立的指令流、寄存器组和片上缓存，这些处理器通过共享的总线访问内存单元。系统的内存结构对于各个处理器可能对称，也可能不对称。
- 多核环境，即一个处理器中的多个核。各个核有独立的指令流和寄存器组，这些核可能既有独立的缓存，又有共享的缓存。所有的核通过共享的总线访问内存单元。
- 超线程环境。在一个处理器中可以运行多个指令流，但并不是所有的指令都可以并行执行。超线程处理器虽然有多个寄存器组，但只有一套缓存和计算单元（包括整数和浮点数），所以，这些指令流并不能完全地并行执行。
- 处理器外部中断。外部设备通过 APIC（高级可编程中断控制器）与 Intel x86 处理器通信。当外部设备要跟处理器通信时，会通过一个连接至 I/O APIC 的引脚中断处理器，并告诉它一个中断号。只要处理器没有屏蔽中断，它的指令流就会转到一个预先设定的地址处，进行中断处理。因此，处理器上正在执行的指令流可能会被外部设备中断。比较典型的外部中断是 I/O 设备的中断和时钟定时器中断。例如，在第 3 章我们曾经提到，操作系统利用时钟中断来调度线程。另外，在 I/O 中断的处理例程中，可能需要访问与 I/O 相关的数据结构。
- 内部中断。即使没有外部中断，一个正在执行的指令流也有可能被打断，比如一条指令引发了一个异常，从而使处理器转到了一个异常处理例程中；或者，指令流本身引发了一个软件中断，从而进入到了一个中断处理例程中。
- 线程放弃执行权。线程有可能在执行过程中主动放弃执行权，从而使其他的线程获得处理器资源。当线程执行 I/O 操作时，这是一种很常见的情形。

从以上列举的并发性来源可以看出，一个线程在其执行过程中会经历各种不确定因素，如果它操作的内存单元也有可能被其他的线程访问，则这些不确定的并发访问有可能带来灾难性后果，因此，对共享数据的保护是多线程程序设计中不可缺少的一个环节。而操作系统的任务是，提供多种同步原语，供系统模块或应用程序用来保护它们的数据。

正如我们在本节前面的例子中所看到的那样，为了保护对一个内存单元（比如全局变量）的操作，我们必须使该操作成为一个原子操作，也就是说，这个操作不可被打断，其中间状态不会被其他线程看到。显然，像“g_nCount++”这样的操作（被编译成三条指令）不满足这一条件。实际上，由处理器来提供一些基本的原子运算在这种情况下非常有意义。

在 Intel x86 指令体系中，有些运算指令加上 lock 前缀就可以保证其原子性。以下是使用 lock 前缀的两个条件[INTEL-253666]：

1. 指令的目标操作数必须是一个内存操作数；
2. 仅适用于 ADD、ADC、AND、BTC、BTR、BTS、CMPXCHG、CMPXCH8B、DEC、INC、NEG、NOT、OR、SBB、SUB、XOR、XADD 和 XCHG 指令。这些指令的含义见表 5.1。

表 5.1 可以应用 lock 前缀的 Intel x86 指令

指令	含义	指令	含义
ADD	带进位的加法	INC	加 1
ADC	加法	NEG	二进制求补数
AND	逻辑与	NOT	按位求反
BTC	指定的位保存到 CF 中，再取反	OR	逻辑或
BTR	指定的位保存到 CF 中，再清零	SBB	带借位的整数减法
BTS	指定的位保存到 CF 中，再置位	SUB	减法
CMPXCHG	比较并交换	XOR	逻辑异或
CMPXCH8B	8 字节的比较并交换指令	XADD	交换两个操作数，并做加法
DEC	减 1	XCHG	交换两个操作数

在本章后面我们将会看到，Windows 充分利用了这些原子指令来实现一些基本的数据保护。譬如，在 WRK 中，InterlockedIncrement 实现了原子的加一操作。下面的代码可以在多处理器系统上工作。

```
LONG
FASTCALL
InterlockedIncrement (
```

```
    IN OUT LONG volatile *Addend
    )
{
    __asm {
        mov     eax, 1
        mov     ecx, Addend
        lock xadd   [ecx], eax
        inc     eax
    }
}
```

带 lock 前缀的 xadd 指令实现了在内存单元 Addend 中加一的原子操作，如果另一个线程往 Addend 内存单元中赋值 100，则此赋值要么在加一前完成，要么在加一后完成，而不会被覆盖掉，从而避免了本节前面例子中描述的潜在问题。

5.1.2 进程或线程之间的通信

在现代的操作系统和应用程序中，进程或线程之间的通信往往是不可避免的，有些是因为功能本身的需要，有些则是因为资源有限而不得不共享。譬如，两个线程分别负责数据处理的第 1 阶段和第 2 阶段，负责第 1 阶段的线程在完成处理后把数据放在一个共享缓存区中，然后通知负责第 2 阶段的线程，以作进一步处理。在数据流模型中，这是一种典型的通信方式。两个线程可以以互斥的方式访问共享缓冲区，但需要一种机制来保证两者的处理顺序。在有些情况下，线程可能会争抢一些有限的资源，比如打印机或显示器是有限的资源，线程的打印或显示请求，就可能要经过系统的一个全局队列才最终得到响应。因此，线程之间也会因为这些有限的共享资源而相互制约。

如果两个或多个线程在没有任何保护的情况下，直接操纵共享数据，那么，这些线程的运行结果可能会受到它们被调度的时间顺序的影响。譬如在上一小节的例子中，控制线程有可能仅仅由于线程调度的原因而无法控制计算线程。程序的运行结果取决于进程或线程被调度的顺序，这称为竞争条件（race condition）。竞争条件是程序的逻辑错误（bug），但由于它们的时序不确定，因而往往难以发现和调试。试想，一个程序在大多数情况下运行正常，只是偶尔会产生意外的错误，让软件开发人员排除这样的错误，该是一件多么有挑战性的事情！

使不确定的执行逻辑变得确定和可靠，这是各种同步机制所要解决的基本问题。在操作系统的发展史上，有多种机制被提了出来，它们试图从各个角度来解决这一问题。接下来介绍一组常用的同步原语，几乎每一个支持并发程序的环境都会提供这些同步原语中的部分或全部，Windows 当然也不例外。

(一) 互斥或互斥体（mutex）

互斥（mutex，指 mutual exclusion 的缩写）是一个在并发性程序设计中常用的概念，代表了对一个共享资源的独占式访问，任何时候只有一方可以访问。互斥体则是指一种能提供互斥访问特性的同步对象。Windows 提供了互斥体对象，用于实现跨进程的互斥访问。互斥体并非实现互斥访问的唯一方法，其他同步机制也可能可以实现对共享资源的互斥访问。

互斥体这个同步对象具有两种状态：有信号和无信号。当处于有信号状态时，说明互斥体是可用的，任何一个线程都可以获得此互斥体；当处于无信号状态时，互斥体已被某个线程持有，任何其他的线程都将得不到该互斥体对象，直到持有该互斥体对象的线程释放它的所有权。因此，互斥体对象是有所有权的，正持有该互斥体对象的线程就是它的所有者。

为了实现互斥，可以采用硬件的方案，也可以采用软件的方案。硬件的方案可以建立在处理器提供的原子操作的基础上，比如上一小节我们看到的一些带 lock 的指令。如果是在单处理器上，也可以通过关闭中断来实现。纯软件的方案可以通过“忙等待（busy-wait）”来实现，经典的算法如 Peterson 算法[PETERSON][MOS]。由于“忙等待”不适合于可能长时间的竞争，因此，一般情况下不适合用“忙等待”来实现互斥访问。基于硬件提供的互锁能力，是一种相对比较简捷的实现方式。

(二) 信号量（semaphore）

信号量是 E. W. Dijkstra 于 1965 年提出来的一个同步概念[DIJKSTRA]，它使用一个计数器来控制程序对一个共享资源的访问。信号量有两个操作：Down 和 Up，在 Dijkstra 的论文中使用 P 和 V 表示。Down 操作的逻辑是：检查信号量的计数器是否大于 0，若是，则将其减 1 并成功返回；若计数器为 0，则阻塞直至计数器大于 0。Up 操作的逻辑是：递增计数器的值，如果有其他的线程正阻塞在该信号量上，则选择其中一个（按先后顺序或随机选择）允许它完成 Down 操作。

信号量对象有必要维护一个队列，来记录那些正阻塞在该信号量上的线程。只要该队列非空，则计数器的值总是 0，因为每次 Up 操作的结果是解除一个正在等待的线程，而计数器仍然回到 0。信号量的 Down 和 Up 操作都是原子操作，也就是说，Down 操作中的计数器判断零和减一动作是整体完成的，Up 操作中的计数器递增以及解除等待线程而计数器又回到 0 也是整体完成的。两个操作相互之间不会干扰。

信号量的初始值和 Down、Up 操作的顺序决定了同一时刻允许多少个线程或进程访问某一共享资源。当初值为 1 时，信号量的语义与前面介绍的互斥体相同，此时只有一个

线程可以获得信号量的允许。当初值大于 1 时，信号量可以用来控制对一个资源的并发访问数量，一种用法是在生产者-消费者（Producer-Consumer）系统中，用于协调两者对于资源（比如缓冲区）的竞争。

(三) 锁（lock）

锁是一个比较易于理解的概念，在系统模块和应用程序中经常会用到。锁的用途是实现互斥访问。锁对象实质上是一个二值对象，比如，0 值表示锁是空闲的，任何线程都可以获得，一旦获得，则其值赋为 1，因而任何其他线程将无法获得该锁。如果一个线程想要获得锁，但当前另一个线程已经获得了该锁，则这个线程必须被阻塞，直至使用该锁的线程释放该锁为止。

锁的基本操作是获取锁（Lock）和释放锁（Unlock）。当释放锁的时候，如果有其他的线程正在等待获得该锁，则允许其中之一获得。当很多线程都使用同一把锁的时候，可能会出现锁竞争（lock contention）的现象。在设计一个系统时，锁竞争是应该要避免的，因为一旦出现锁竞争，很多线程被阻塞，系统性能就会下降。

Lock 和 Unlock 中的测试和赋值也必须以原子操作来完成，否则就会存在竞争条件。另外，在实际使用中，依据锁的使用范围，又可分为全局锁（global lock）和局部锁（local lock）。全局锁往往用于控制一个较大范围的数据对象，比如一个全局变量；而局部锁则用于管理一个相对小范围的数据的访问。相对而言，全局锁更容易引起锁竞争的问题，因此，在实际系统中，对全局锁的使用要非常谨慎。

(四) 临界区（critical section）

上述几种同步原语侧重于对数据的原子性保护或管理，临界区则是从程序代码的角度来控制线程的并发性。临界区是指程序代码中的一部分区域，在任何一个时刻，至多只有一个线程可以在这部分区域中执行。因此，如果一个数据对象仅仅在临界区中被访问，那么对该数据对象的任何访问都是安全的。

临界区有两个基本操作：进入临界区（EnterCriticalSection）和离开临界区（LeaveCriticalSection）。若一个线程已经进入了临界区，则其他线程要想进入该临界区，将只能在进入点上等待，直到前者离开。临界区可以通过一个互斥标志或者锁变量来实现。如同锁操作一样，对互斥标志的测试和赋值必须通过原子操作来完成。

临界区不必是一个连续的代码区域，它可以散布在多个函数中。在任何一个函数中，要进入临界区，必须显式地调用 EnterCriticalSection，在离开临界区时则显式地调用

LeaveCriticalSection。如果从代码区域的角度来看，EnterCriticalSection 和 LeaveCriticalSection 就好像括号一样把临界区的各个部分括了起来。任何一个线程从临界区外面进入“括号”时，必须排队依次进入，只有在前面进入的线程离开“括号”后，其中一个正在排队的线程才可以进入。

临界区的使用非常灵活，其范围可大可小，在使用时一定要恰当，它可以跟要保护的数据关联，也可以不跟数据关联。如果临界区的“括号”范围太大，则程序的并发性将会变差。缩小临界区的范围，意味着用“括号”将代码切得很碎，从而招致代码更为复杂。因此，在实际使用时需要在程序的并发性和复杂性之间进行平衡。

(五) 自旋锁（spinlock）和忙等待

忙等待是一种软件技术，指重复地检查一个条件，直至这个条件满足为止。例如，一个用户程序可以用忙等待的方式来检查是否有用户输入（键盘或鼠标）或者某个外部设备的端口上是否有数据。通常而言，忙等待是一种应该尽量避免的软件设计（anti-pattern），因为忙等待意味着处理器在不停地空耗。但是，在某些特殊的情形下，忙等待可以有很好的正面用途。一个典型的例子是，在多处理器环境下实现自旋锁（spinlock）。

自旋锁是一种特殊的锁，也有两个操作：Lock 和 Unlock。自旋锁的名称较为形象，一个线程为了获得自旋锁，会占住处理器不放，并且不停地旋转，即空耗处理器资源，直到该锁空闲下来，从而成功地获得它。而对于前面提到的几种同步原语，当竞争发生时，线程会把处理器执行权让出来，直至所等待的对象或资源可用时，被系统或其他线程唤醒。自旋锁则完全不同，等待自旋锁的线程既不用把执行权让出来，也不用其他线程告诉它锁的状态。

自旋锁在单处理器系统上没有任何意义，它仅适用于多处理器系统。而且，从自旋锁的等待时间和效率而言，任何一个线程持有自旋锁的时间都不能长，在现代计算机系统中，通常不应该超过毫秒量级（比如不超过几毫秒）；如果时间长了，则应该考虑把处理器的执行权让出来。但是，在有些特殊情况下，即使把执行权让出来，处理器的计算资源也无法得以利用。例如，处理器被空出来以后，仍然需要访问全局的就绪线程链表才能确定该把执行权交给哪个线程，而如果就绪线程链表也被其他处理器锁上的话，则必须等到其他处理器释放了就绪线程链表以后才能选择线程。因此，自旋锁的设计要充分考虑到线程调度的因素。从自旋锁的逻辑来考虑，它锁的对象一定是特定于处理器的，换句话说，当一个处理器上的线程在等待自旋锁时，一定存在另一个处理器正在使用该锁保护的对象，并且可以期望该处理器很快会释放该锁。

在处理器数量并不很多的情况下，用自旋锁来保护一些关键的内核数据结构，这是非常有效的。在 WRK 中，我们可以看到自旋锁的实现和用法。

(六) 消息

消息传递机制是两个线程、两个进程，甚至两个系统之间的通信原语。在消息机制中，有两个原语操作：Send（发送）和 Receive（接收）。这两个操作可以以同步方式工作，即直至完成操作后才返回；也可以以异步方式工作，即把任务交给其他的线程来完成，它们自己则立即返回。

用于同步的消息机制并不能独立地工作，它一定还需要其他同步原语的支持。比如当一个线程发送消息时，如果另一个线程正在接收操作上等待，则它可以把数据放到某个共享缓冲区中，然后唤醒接收线程。此外，消息机制还必须确定如何标识线程或进程，这是一个实际的设计问题，对于应用层模块，一种简单的做法是使用线程或进程的标识符（ID）。

从以上这些对同步原语的介绍我们可以看出，互斥访问是各种同步原语的基础，几乎每一种同步原语都依赖于在一个原子操作中完成对一个内存单元的读（或测试）和写的两步骤操作。一旦有了这样的原子操作基础，则实现这些同步原语并不复杂。另一方面，这些同步原语并不是独立的，甚至可以相互实现，比如互斥体可以用信号量来实现，临界区可以用互斥体来实现。现代的操作系统往往提供多种同步原语，供应用程序使用；而应用程序则根据自己的需要，选择恰当的同步原语。除了以上这些同步原语，Windows 还提供了其他一些同步机制，5.3 节和 5.4 节将介绍 Windows 内核中实现的各种同步机制。

5.1.3 经典的同步问题

上一小节中介绍的同步原语可以有效地防止竞争条件的发生。然而，虽然它们可以实现对于共享数据的互斥访问，却也引入了另外一些新的问题。这一小节将讨论四个经典的同步问题，通过对于这些问题的解释我们可以进一步理解同步的复杂性。

(一) 死锁（deadlock）

死锁是一种稍不留意就会发生的问题，尤其当线程需要多个资源才能前进的时候。死锁的最简单情形是：线程 A 已经获得了对于资源 a 的访问权，现在要申请访问资源 b；线程 B 已经获得了对于资源 b 的访问权，现在要申请访问资源 a。这两个线程都已经获得了一个资源，但还想要对方的那个资源，结果谁也满足不了，所以，两个线程都不能前进。根据此例子进行扩展可以得到死锁的一个说明性定义：

如果一组线程中的每个线程都在等待只能由其他线程才能满足的条件，那么这组线程是死锁的，所有的线程都将继续等待，无法前进。

若用线程和所等待资源之间的等待和占有关系来形成一个有向图的话，则这组线程和相关资源所构成的图中，一定存在一个环。我们可以这样来理解：从一个线程出发，它在等待某个资源，这个资源已经被另一个线程占有了；而那个线程也在等待，它在等待另外一个资源，该资源已经被其他线程占有了。如此检查下去，就形成了一条链，因为只有有限个线程，所以，要么在某个资源处，链断了，意味着这个资源是可以被满足的，从而链上的所有线程以相反的顺序都可以满足，所以这些线程没有死锁；要么回到一个已经检查过的线程，因而形成了环，即死锁。在前一种情况下，上述定义中的条件并不满足，因为最后的那个线程并非依赖于其他的线程，而是直接可被满足。

死锁是一个已经被大量研究过的问题，早在 1971 年就有学者 Coffman 等总结了死锁的四个必要条件[COFFMAN]：

1. 对资源的访问是互斥的。每个资源要么已被某个线程占有，要么还可以得到。
2. 线程已占有了一个或多个资源，同时还在等待其他的资源。
3. 资源一旦被分配给一个线程，则只能被这个线程释放，其他线程无法抢占资源。
4. 一组正在等待的线程形成了环，环中的每个线程都在等待下一个线程占有的资源。

在实际系统中，死锁的检测、预防比较复杂。即使最近几年也仍然有学者在研究各种应对死锁的方法。一种较为理想的思路是，设法打破上面列举的四个条件，从而使死锁无法形成。比如说，尽可能设计成让每个线程只依赖于一个资源或条件，从而打破上面列举的第二个条件；或者让每个线程的等待有超时值，一旦超时，则释放已占有的资源，过一段时间以后再重试。Dijkstra 提出了一种可以避免死锁的调度算法，称为银行家算法，基本的思路是系统在分配资源的时候，衡量这一资源分配是否安全，好比银行在批复贷款时考虑一笔贷款批了以后是否资金安全一样。关于该算法的细节，可参考有关资料[MOS]。

(二) 饥饿（starvation）

与死锁不同，饥饿是指一个进程或线程满足执行的条件，但一直得不到执行，甚至永远得不到执行（“饿死”）。饥饿通常是由资源分配策略引起的，比如由于策略而导致不公平，有些资源请求在特定的情形下永远得不到满足。例如，在一个数据库系统中，如果按照数据请求的类型来处理，则某些类型的请求可能在繁忙的时候总是得不到处理。而如果按照先进来先处理的策略，则任何一个请求都会得到处理。

严格来讲，饥饿并不是一个同步问题，而应该是一个调度问题。但从表现上来看，却是一个线程虽然满足继续执行的条件，但由于策略的原因总是得不到执行，而不得不保持在等待资源的状态。

(三) 优先级反转（priority inversion）

优先级反转是指这样的问题：在线程调度时，一个低优先级的线程占有一个共享资源，从而导致高优先级的线程虽然其他运行条件都满足，但因为得不到该资源而无法运行。在采用抢占式调度算法的操作系统中，当低优先级的线程占有了资源时，它有可能被中等优先级的其他线程抢占，从而导致高优先级的线程在更长时间内无法运行，实际的效果就是，相当于把高优先级的线程降到低优先级了。

对于基于优先级来调度的操作系统，优先级反转是一个很实际的问题。当两个或多个不同优先级的线程共享一个资源时，就可能发生优先级反转问题。此问题对于实时线程或者实时操作系统的影响更加严重，因为这相当于把一个实时线程降低优先级从而变成了一个非实时线程。

优先级反转问题的一个著名实例是1997年7月发生在火星拓荒者号（Mars Pathfinder）上的实时操作系统中的一个问题。拓荒者号的登录车和导航台之间通过1553总线连接。在它的控制系统中，与1553总线有关的软件有两个任务：一个任务负责从1553总线上采集数据，任务名称为bc_dist，其优先级为系统第3高；另一个任务负责控制总线上数据传输任务的设置，任务名称为bc_sched，其优先级为第1高。总线上的中断频率设置为8 Hz，即每125 ms传输和处理数据一次，称为一个事务。其他的任务，比如一些科学计算和处理数据的任务，优先级更低。一个用于气象科学的设备称为ASI/MET，其对应的ASI/MET任务的优先级低于bc_dist。

事务过程是这样的：总线上数据到达；bc_dist任务被唤醒，并分发数据给其他的计算任务，包括ASI/MET任务；bc_sched任务被唤醒，以设置下一个数据传输事务。如果bc_sched任务被唤醒以后，发现bc_dist尚未完成它的工作，则认为这是系统错误，因而重新启动系统。启动过程会重置所有的软件和硬件，但已经采集的数据不会丢失，除非掉电。在事务处理过程中，有一个优先级反转问题发生在ASI/MET任务和bc_dist任务上。ASI/MET任务占有一个互斥信号量，在释放该信号量以前会被其他一些中等优先级的任务抢占，而bc_dist任务会试图发送最新的ASI/MET数据给它，因而也要等待这个信号量。由于这些中等优先级任务的抢占，使得ASI/MET任务不能释放信号量，因而导致bc_dist任务不能及时完成。结果，当bc_sched任务醒来时发现bc_dist尚未完成，于是重新启动系统。系统重启事件频繁发生，影响到了拓荒者号的正常工作。问题

刚发生时，曾一度被媒体称为“软件小故障”和“计算机正在尝试每次做更多的事情”。幸运的是，这个问题后来被发现，并且被修正了。关于火星拓荒者号上的优先级反转实例的解释，可参考有关材料[MARS-PATHFINDER]。

优先级反转问题通常有两种解决方案。第一种方案是优先级继承（priority inheritance），低优先级线程继承一个正在等待某个共享资源的高优先级线程的优先级。这样中等优先级的线程就不会抢占低优先级线程了。低优先级线程的优先级变化发生在高优先级线程进入等待的时候，而当低优先级线程释放共享资源时，它会恢复原来的低优先级。第二种方案称为优先级限高（priority ceiling），每个资源都关联一个优先级，该值等于它的所有用户线程中最高的优先级加一。当一个线程访问该资源时，其优先级提升为资源的优先级，当用完了该资源时，再回到自己正常的优先级上。

(四) 哲学家进餐问题（dining philosophers problem）

哲学家进餐问题是计算机科学中一个很有说明性的多进程或多线程同步问题，是Dijkstra于1965年提出来的。问题的描述可以改编成这样：5个哲学家坐在一张圆桌旁，桌上有5根筷子，每根筷子都放在两个哲学家中间，每个哲学家面前都有一碗饭，如图5.2所示。哲学家要么在吃饭，要么在思考，而且，哲学家必须拿到左右手边的两根筷子才能吃饭。哲学家相互之间不讲话，没有通信。现在要为每个哲学家写一段程序来模拟其行为，要求避免死锁。

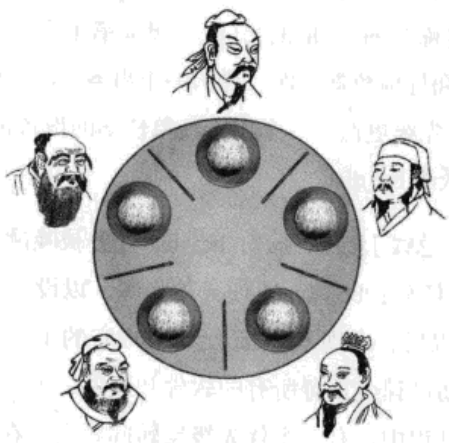


图5.2 哲学家进餐问题图示

如果5个哲学家同时开始启动，他们的行为按以下步骤进行：先拿左边的筷子；再拿右边的筷子；吃饭；吃完了放回左边的筷子；再放回右边的筷子；进入思考。思考一段时间后哲学家饿了，又进入下一轮吃饭的过程。这里存在的死锁是，当哲学家完成了第一步后，每个人都拿到了左边的筷子，下一步就死锁了。

避免死锁的一种做法是，检查要获取的资源是否可用，若不可用，则放弃已经占有的资源，等一段时间后重试。假设哲学家拿到了左边的筷子，然后检查右边筷子是否可用，发现不可用，则放下左边的筷子，等过一段时间后重试。若所有的哲学家同时准备进餐，而且每个人重试的等待间隔相同，那么，虽然死锁避免了，但每个哲学家依然吃不到饭，这正是饥饿的情形（在这里，每个哲学家越来越饿了）。尽管每个哲学家都在努力地做事情，而且也照顾到了其他哲学家的资源需求，但他们的吃饭问题始终不能前进一步。这种情形被称为活锁（livelock）。

哲学家进餐问题是一个很好的用于说明死锁、饥饿和活锁的例子，它蕴含了在计算机系统中以互斥方式访问共享资源时的一些共性问题。因此，它也被用于验证各种同步原语在解决实际问题时的能力和缺陷。

哲学家进餐问题的解决方案有多种。一种方案是引入一个仲裁者（或侍者），由仲裁者来分配筷子，每个哲学家在拿筷子前向侍者请求许可，放下筷子后通知侍者。这样，侍者可以控制筷子的分配和使用。第二种方案是，为每个资源，即筷子，分配一个偏序（partial order）；并且规定，资源的请求按序进行，而释放按逆序进行；序不相关的资源不能被同一方使用。在哲学家进餐问题中，筷子被编号为 1、2、3、4、5，哲学家先拿其左右两边编号较小的筷子，然后再拿编号较大的筷子。因此，若 5 个哲学家同时拿筷子，则只有 4 个哲学家会拿起他们的第一根筷子，而剩下编号最大的筷子，即 5 号筷子，仍然留在桌子上，有一个哲学家在第一轮抢夺中拿不到筷子。并且，5 号筷子只能被一个哲学家拿到。此哲学家吃完饭，先放下编号大的筷子，再放下编号小的筷子。然后，其他的哲学家可以拿起筷子进食了。

资源偏序方案可以避免死锁，它要求每个线程确切地知道要获取哪些锁，以及按什么样的顺序获取这些锁。如果线程的用锁情况是动态变化的，则这一方案工作起来可能会非常低效。譬如，一个线程已经获得了一个高编号的锁和一个低编号的锁，现在又要求获得一个中间编号的锁才能继续进行。那么，它必须先释放高编号的锁，等到获取到中间编号的锁以后，再获取高编号的锁。然而，在用锁情况确定的情形下，这是一种相对比较实用的方案。

1984 年，Chandy 和 Misra 又提出了一种方案，该方案可使任意多个线程竞争任意多个资源。它也是完全分布式的，一旦初始化，就不再需要任何中心控制。针对哲学家进餐问题，其方案如下：

1. 每根筷子都被赋予一个状态，脏或干净。初始时，所有的筷子都被赋给编号较低的哲学家，其状态为脏。

2. 当哲学家想要吃饭时，如果他尚未占有一根筷子，则向邻居发送一个请求消息。
3. 如果哲学家已经占有了筷子，并且收到了一个消息，那么，若筷子是干净的，则他继续占有筷子；如果筷子是脏的，则放弃筷子。如果他要把筷子交出去，则先将筷子置成干净的，再交出去（礼貌起见，洗干净再交出去）。
4. 当哲学家吃完饭以后，两根筷子都变成脏的。如果已经有其他哲学家发过请求，则置成干净的，再交给对方。

Chandy/Misra 方案能适应较大规模的问题，并且也有很好的并发性。有兴趣的读者，可以参考他们的研究论文[CHANDY-MISRA]。

从以上对于一些经典同步问题的介绍可以看出，操作系统虽然提供了各种互斥和同步的机制，但是对于避免这些复杂的同步问题能力有限，不过，它可以在两个方面使这些问题得以缓解：保证系统本身的任务避免诸如死锁之类的问题；提供更多的信息和工具，允许在特定环境下测试、检测，甚至预防这些问题的发生。在 Windows 中，这两方面的努力都存在[MSDN-DEADLOCK]。

5.2 Windows 中断与异常

中断（interrupt）是现代计算机系统中的处理器与外部设备打交道的重要途径，而异常（exception）则是处理器在正常指令流执行过程中产生的一些特殊事件，需要紧急处理。中断和异常都打断了一个正常执行的指令流，但是它们有本质的区别：中断的产生与当前正在执行的指令流并无实质性的关联，所以，中断与当前正在执行的线程和进程无关；而异常则是当前指令流中某些特殊指令产生的结果，比如上一章提到的页面错误就属于异常。中断是异步的，而异常是同步的。

在上一节中也提到了，中断是一个重要的并发性来源，它可以打断一个正在执行的线程，让处理器切换到其他的线程。中断分硬件中断和软件中断。硬件中断往往是由 I/O 设备引发的，当它们需要处理器完成一些事情时，就会以中断的方式告诉处理器。这种方式使得处理器和设备可以并行地工作，仅当需要的时候才打断处理器，从而可以最大可能地发挥处理器的能力。软件也可以产生中断，比如内核代码可以触发一个软件中断来启动线程调度，相当于打断了当前线程的执行。处理器可以禁止中断，这时指令流不会被打断，但一般情况下系统不会这样做，因为这样设备就无法以中断方式跟处理器通信了。通常系统只有在关键的地方才会禁止中断，比如，在单处理器系统上，关闭中断可以实现原子操作。有时候内核在处理一个中断的时候也会禁止中断。

异常是指程序执行过程中发生的特殊事件，它可以是硬件产生的，比如除零错误和页面错误，也可以是软件产生的，比如程序中的断点事件。Windows 内核实现了一套异常分发机制，允许一个异常被按照相应的顺序妥善处理。

这一节我们将详细地介绍 Windows 的中断和异常处理机制。首先来看一下 Intel x86 处理器支持的中断控制器和中断描述符表。

5.2.1 硬件中断的发生和处理

在 Intel x86 体系结构中，外部硬件中断是通过处理器上的中断管脚或者一个称为本地 APIC（local APIC）的内置模块来发生的。早期的 Intel 处理器芯片支持 i8259A 可编程中断控制器（PIC）。PIC 只能在单处理器系统上工作，它可支持 15 条中断线，也就是说，最多有 15 个硬件中断源。现在绝大多数 Intel x86 系统使用了 APIC（高级可编程中断控制器）中断结构，在 Intel x86 处理器的芯片中内置了一个本地 APIC，由它负责接收来自中断管脚、内部中断源和外部 I/O APIC 的中断。而在多处理器系统中，本地 APIC 也负责发送和接收处理器间的中断（IPI，interprocessor interrupt）消息。因此，系统总线上的逻辑处理器之间通过各个处理器的本地 APIC 发送和接收 IPI 消息进行通信。

在 APIC 中断结构中，外部设备或系统硬件通过 I/O APIC 将中断源连接到处理器中。I/O APIC 负责接收外部中断，并且将它们传递给本地 APIC。I/O APIC 有一种“虚拟连线模式（virtual wire mode）”，允许以标准的 8259A 风格的中断控制器进行通信，因此，这相当于可使 Intel x86 处理器接收直接来自 8259A 中断控制器的中断。由于 Intel x86 芯片内置了本地 APIC，而且通过 I/O APIC 来接收外部中断，可以支持超过 15 个中断源（单个 I/O APIC 可支持 24 个），因此，即使单处理器系统也使用 APIC 中断结构。

图 5.3 显示了 Intel x86 APIC 的中断结构。其中本地 APIC 可以接收的中断源包括：处理器管脚（LINT0 和 LINT1）、本地 APIC 定时器（timer）、性能监视计数器中断、热传感器中断、APIC 内部错误中断，这些中断源称为本地中断源，另外还有通过 I/O APIC 转送过来的中断源：外部连接的 I/O 设备的中断消息和 IPI 消息。I/O APIC 也可以有多个，最多可达 8 个，每个 I/O APIC 有 24 根中断管脚，因此对于多 I/O APIC 的系统来说，中断数量可以达到上百个之多。现在大多数机器使用一个 I/O APIC。I/O APIC 的功能是把硬件设备上的中断转送给本地 APIC，它的 24 根管脚本身并没有优先级区别，但通过编程可以重新设定每根管脚的优先级以及中断编号。

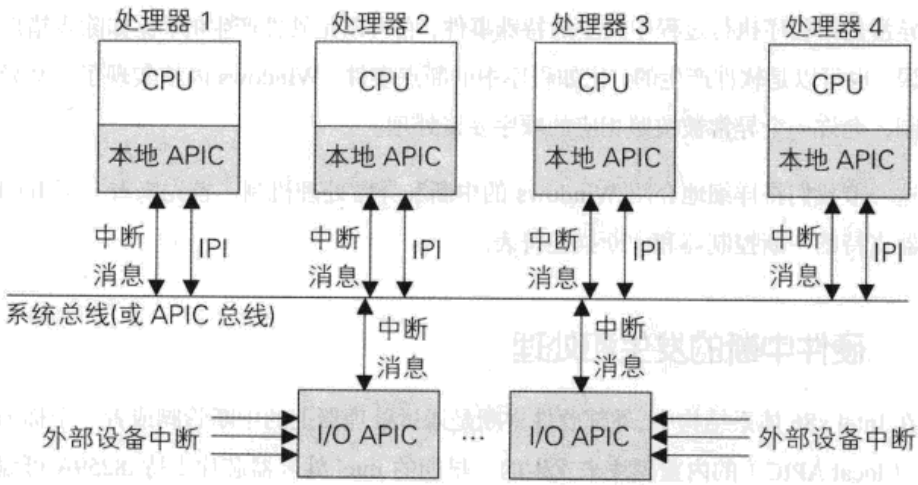


图 5.3 Intel x86 APIC 的中断结构

APIC 的中断编程是在系统初始化时候完成的，Intel x86 定义了 256 个中断向量编号（interrupt vector number），也称为中断向量，从 0 至 255。本地 APIC 和 I/O APIC 支持 240 个中断向量，范围为 16~255，0~15 是非法中断向量。如果在本地 APIC 中，发送或者接收 0~15 之间的中断向量，则它的错误状态寄存器会指明非法中断向量的错误。Intel 预留了 16~31 作为预定义的中断和异常，以及一些保留编号；但本地 APIC 并不会将这段范围内的中断向量当做非法向量来对待。有关本地 APIC 和 I/O APIC 的设置以及协作原理的详细信息，请参考 Intel 的手册[INTEL-253668]。

对于一个处理器，它一旦被中断，则某个预先设定的中断服务例程便被执行。系统软件要做的事情是，提供这些例程，并将它们设定到处理器的硬件中断向量表中。在 Intel x86 体系结构中，处理器的中断向量表也称为中断描述符表（IDT，Interrupt Descriptor Table）。IDT 将每个中断或异常与一个处理该中断或异常的服务例程关联了起来。与 GDT 和 LDT 类似，IDT 中的每一项是一个 8 字节的描述符。IDT 中最多可有 256 项，可以少于 256 项，依据实际使用的中断向量数量而定，其第一项也可能包含有效的描述符。IDT 可以位于线性地址空间的任何地方，但基地址应该 8 字节对齐，最大限制值是 8 的倍数减 1。处理器内部的 IDTR 寄存器记录了 IDT 的位置和它的最大限制。LIDT 和 SIDT 指令分别用于加载和存储 IDTR 寄存器。LITD 只有在特权级 0 下才可以执行。图 5.4 显示了 IDT 和 IDTR 之间的关系。

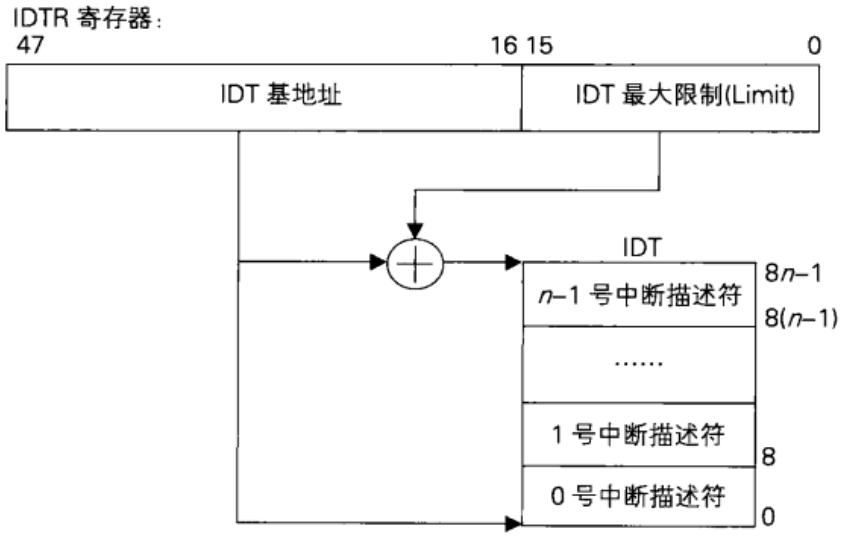
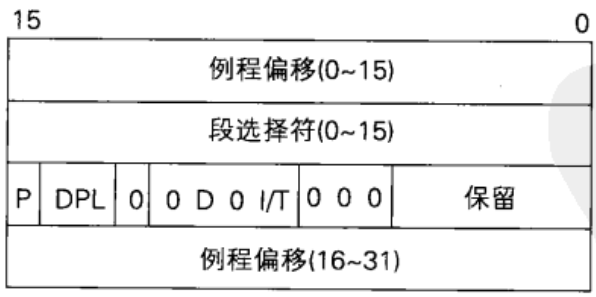


图 5.4 IDT 和 IDTR 之间的关系

IDT 中的每个描述符，除了指定一个例程地址（包括段选择符和 32 位偏移地址）以外，还包含有一些其他的信息，其 8 字节结构如图 5.5 所示。其中描述符特权级（DPL）用于软件中断的情形，可避免低特权级的代码通过软件中断的形式执行关键的异常或中断处理代码。I/T 标志指明了这一描述符是中断门描述符还是陷阱门描述符。中断门和陷阱门的区别在于：当通过一个中断门来访问中断例程或异常例程时，处理器会清除 EFLAGS 寄存器中的 IF 标志（它控制是否屏蔽中断），因而可防止其他的中断妨碍当前的中断处理，随后的 IRET 指令会恢复 IF 标志；而通过一个陷阱门来访问时，处理器的 IF 标志不受影响。IDT 中的描述符还有一种类型，称为任务门描述符，它描述了要切换的任务，指向 GDT 中的一个 TSS 描述符，关于任务门以及 TSS，本书不作介绍。



P: 段存在标志
DPL: 描述符特权级
D: 中断门大小。1 为 32 位，0 为 16 位
I/T: 中断门或陷阱门。1 为陷阱门，0 为中断门

图 5.5 IDT 的中断描述符格式

当一个中断或异常发生时，处理器利用中断向量号作为索引，定位到 IDT 中的描述符，如果此描述符指定了一个中断门或陷阱门，则处理器调用该描述符中指定的中断或异常例程。如果处理例程是在一个更高特权级的代码中执行的，则需要进行栈切换。切换到新栈后，需要把原来被中断的例程的栈信息压到新栈中。处理器在把控制权交给中断或异常处理例程以前，将 EFLAGS、CS 和 EIP 寄存器的状态压到栈中。如果这是一个异常并且有错误码需要保存的话，则把错误码也压到栈中。中断或异常处理例程在完成了它自己的工作以后，必须使用 IRET 指令回到原来的指令流。IRET 指令与 RET 类似，它额外做的事情是，恢复 EFLAGS 寄存器；如果在进入中断或异常处理例程前有栈切换的话，还需要切换回到原来被中断的例程的执行环境。5.2.7 节将介绍当异常（或中断）发生时处理器使用的栈是如何切换的。

前面提到，中断和异常有一个重要的区别，即中断是异步的，而异常是同步的。但处理器使用同一套中断机制来处理中断或异常，那么中断和异常又是如何区分的呢？Intel x86 处理器将 0~31 之间的中断向量保留了，这也正是本地 APIC 应该为外部设备使用 32 及以上的中断向量的原因。在小于 32 的中断向量中，只有 2 号是不可屏蔽中断，其他的都定义为异常或 Intel 保留。而且异常也分为三种类型：错误（fault）、陷阱（trap）和中止（abort）。处理器对待这三种类型的异常的做法是不相同的：对于错误类型的异常，处理器期待这类异常条件可被修正，它会返回到原来出错的指令处，典型的例子是页面错误；对于陷阱类型的异常，处理器期待原来的指令流仍然是连续可执行的，这只是一个陷阱而已，所以，它会返回到发生异常的指令后面的那条指令，典型的例子是调试断点；对于中止类型的异常，处理器并不期待程序或系统还能够从失败中恢复，这通常用于报告严重的错误，比如硬件错误或者系统数据结构中的不一致性。表 5.2 列出了 Intel x86 的中断和异常安排。

表 5.2 Intel x86 的中断和异常一览表

向量号	名称	类型	错误码	描述（或来源）
0	除法错	错误	无	DIV 和 IDIV 指令产生的错误
1	调试	错误/陷阱	无	调试异常
2	NMI 中断	中断	无	不可屏蔽外部中断
3	断点	陷阱	无	INT 3 指令
4	溢出	陷阱	无	INTO 指令
5	BOUND 越界	错误	无	BOUND 指令
6	无效操作码	错误	无	UD2 指令或保留的操作码（opcode）
7	协处理器不可用	错误	无	浮点或 WAIT/FWAIT 指令
8	双重错误	中止	是（零）	可产生异常、NMI 或 INTR 的任何指令

续表

向量号	名称	类型	错误码	描述（或来源）
9	协处理器段溢出	错误	无	浮点指令
10	无效 TSS	错误	是	任务切换或 TSS 访问
11	段不存在	错误	是	装载段寄存器或访问系统段
12	栈段错误	错误	是	栈操作和 SS 寄存器装载
13	一般保护错	错误	是	任何内存引用或其他保护检查
14	页面错误	错误	是	任何内存引用
15	Intel 保留			
16	x87 浮点	错误	无	x87 FPU 浮点或 WAIT/FWAIT 指令
17	对齐检查	错误	是（零）	对内存中数据的引用
18	机器检查	中止	无	错误码和异常来源与特定模型相关
19	SIMD 浮点异常	错误	无	SSE/SSE2/SSE3 浮点指令
20~31	Intel 保留			
32~255	用户定义的中断	中断		外部中断或 INT *n* 指令（*n*≥32）

在软件中通过“INT *n*”指令也可以产生中断，比如“INT 0x2e”指令正是 Windows 操作系统在早期的 Intel x86 处理器芯片上用户模式代码调用系统服务的途径。在 Pentium II 以后的处理器上，Windows 使用专门的 sysenter 和 sysexit 作为进入和退出系统服务的快速指令。通过“INT *n*”指令产生的中断不会被 EFLAGS 寄存器中的 IF 标志屏蔽掉。另外，INTO、INT 3 和 BOUND 可以产生相应的异常，例如，“INT 3”可产生一个断点异常。尽管“INT *n*”可以以软件方式来模拟异常，但有一些限制。“INT *n*”在产生异常时，即使是表 5.2 中规定的小于 32 并且有错误码的异常，处理器也不会把错误码压到栈中，因此，如果异常处理例程试图从栈中弹出一个错误码，则会导致后续的 EIP 错位，从而跳转到错误的地方。

了解了 Intel x86 体系结构的中断机制以后，现在我们来看一下 Windows 如何在系统层充分运用处理器的中断能力。首先，本地 APIC 的初始化工作是在 Windows 体系结构的 HAL 模块中完成的，所以，在 WRK 中我们看不到这部分代码。然而，IDT 的许多初始化工作是在 Windows 内核中完成的，其中，很多中断处理例程和异常处理例程是在 WRK 内核模块中定义的。例如，上一章讨论的页面错误异常的处理例程是 KiTrap0E 函数，其代码位于 base\ntos\ke\i386\trap.asm 文件中，在该文件中我们还可以看到许多其他的异常处理例程。

位于 base\ntos\ke\i386\newsysbg.asm 文件中的 KiSystemStartup 函数，包含了 IDT 的初始化工作。我们可以看到，KiSystemStartup 函数将内核中定义的 IDT，即 base\ntos\ke\i386\trap.asm 文件中的_IDT（见 370~420 行），拷贝到了处理器当前已被设置的 idtr 寄存器所

指的位置处，见 newsysbg.asm 文件的 392~407 行。_IDT 包含了从 0 到 2F 之间的描述符项。但是，有一些项是 KiSystemStartup 特殊处理的，包括 2 号不可屏蔽中断和 8 号双重错误异常。另外，KiSystemStartup 也调用 HalInitializeProcessor 函数，对于处理器间中断的向量进行设置。

虽然我们看不到 ntldr 和 HAL 的代码，但是，从 KiSystemStartup 函数中有关 IDT 的处理可以看出，ntldr 在将控制权交给内核模块 ntoskrnl.exe 以前，已经设置了一个满的 IDT（即有 256 项），KiSystemStartup 只是替换其中真正需要使用的 IDT 项。通过检查 base\ntos\ke\i386\trap.asm 文件中定义的_IDT 表，以及相应的例程，我们可以看到 Windows 是如何处理表 5.2 中列出的中断或异常的。为阅读方便起见，以下列出_IDT 的定义，其中例程名称和对应的中断向量号以粗体显示：

```
_IDT            label byte

IDTEntry        _KiTrap00, D_INT032          ; 0: Divide Error
IDTEntry        _KiTrap01, D_INT032          ; 1: DEBUG TRAP
IDTEntry        _KiTrap02, D_INT032          ; 2: NMI/NPX Error
IDTEntry        _KiTrap03, D_INT332          ; 3: Breakpoint
IDTEntry        _KiTrap04, D_INT332          ; 4: INTO
IDTEntry        _KiTrap05, D_INT032          ; 5: BOUND/Print Screen
IDTEntry        _KiTrap06, D_INT032          ; 6: Invalid Opcode
IDTEntry        _KiTrap07, D_INT032          ; 7: NPX Not Available
IDTEntry        _KiTrap08, D_INT032          ; 8: Double Exception
IDTEntry        _KiTrap09, D_INT032          ; 9: NPX Segment Overrun
IDTEntry        _KiTrap0A, D_INT032          ; A: Invalid TSS
IDTEntry        _KiTrap0B, D_INT032          ; B: Segment Not Present
IDTEntry        _KiTrap0C, D_INT032          ; C: Stack Fault
IDTEntry        _KiTrap0D, D_INT032          ; D: General Protection
IDTEntry        _KiTrap0E, D_INT032          ; E: Page Fault
IDTEntry        _KiTrap0F, D_INT032          ; F: Intel Reserved

IDTEntry        _KiTrap10, D_INT032          ;10: 486 coprocessor error
IDTEntry        _KiTrap11, D_INT032          ;11: 486 alignment
IDTEntry        _KiTrap0F, D_INT032          ;12: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;13: XMMI unmasked numeric exception
IDTEntry        _KiTrap0F, D_INT032          ;14: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;15: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;16: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;17: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;18: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;19: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;1A: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;1B: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;1C: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;1D: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;1E: Intel Reserved
IDTEntry        _KiTrap0F, D_INT032          ;1F: Reserved for APIC

        rept 2AH - (($ - _IDT)/8)
IDTEntry        0, 0                         ;invalid IDT entry
        endm
IDTEntry        _KiGetTickCount,  D_INT332   ;2A: KiGetTickCount service
```

```
IDTEntry        _KiCallbackReturn, D_INT332 ;2B: KiCallbackReturn
IDTEntry        _KiRaiseAssertion, D_INT332 ;2C: KiRaiseAssertion service
IDTEntry        _KiDebugService,   D_INT332 ;2D: debugger calls
IDTEntry        _KiSystemService,  D_INT332 ;2E: system service calls
IDTEntry        _KiTrap0F,         D_INT032 ;2F: Reserved for APIC
```

IDT 的初始化完成以后，一旦有中断或异常发生，控制权就会进入到操作系统设置的例程中。在 Windows 系统中，内核对中断和异常分别定义了不同的系统结构，以便允许内核代码或用户代码扩展中断或异常的处理过程。用于处理中断的机制是一种被称为中断对象的内核对象，它允许设备驱动程序为它的设备注册一个 ISR（中断服务例程）。内核在接收到一个中断时，可以将该中断分发到那些已经为该中断向量注册了中断对象的设备驱动程序中，5.2.3 节将介绍中断对象。类似地，Windows 内核也提供了一套灵活的、可扩展的机制来分发异常，允许内核调试器或进程调试器、内核或应用程序本身，以及环境子系统有机会处理异常，5.2.7 节将介绍异常分发过程。

5.2.2 中断请求级别（IRQL）

尽管 APIC 中断控制器已经提供了中断优先级支持，不过，Windows 自己还是定义了一套优先级方案，称为中断请求级别（IRQL，Interrupt Request Level）。在 Intel x86 系统中，Windows 使用 0~31 来表示优先级，数值越大，优先级越高。软件中断或非中断代码的 IRQL 是在内核中管理的，而硬件中断则在 HAL 中被映射到对应的 IRQL。在 Windows 代码中常常可以看到的一些 IRQL 定义如下所示：

```
#define PASSIVE_LEVEL 0              // Passive release level
#define LOW_LEVEL 0                  // Lowest interrupt level
#define APC_LEVEL 1                  // APC interrupt level
#define DISPATCH_LEVEL 2             // Dispatcher level

#define PROFILE_LEVEL 27             // timer used for profiling.
#define CLOCK1_LEVEL 28              // Interval clock 1 level - Not used on x86
#define CLOCK2_LEVEL 28              // Interval clock 2 level
#define IPI_LEVEL 29                 // Interprocessor interrupt level
#define POWER_LEVEL 30               // Power failure level
#define HIGH_LEVEL 31                // Highest interrupt level

#if defined(NT_UP)
#define SYNCH_LEVEL DISPATCH_LEVEL   // 针对单处理器系统的同步级别
#else
#define SYNCH_LEVEL (IPI_LEVEL-2)    // 针对多处理器系统的同步级别
#endif
```

图 5.6 列出了 Intel x86 系统上 IRQL 的分配方案。PASSIVE_LEVEL（被动级别）代表了最低的 IRQL，运行在 PASSIVE_LEVEL 的线程可以被任何高 IRQL 的事情打断，所有的用户模式代码都运行在此 IRQL 上。APC_LEVEL（APC 级别）仅仅比 PASSIVE_LEVEL 高，这也正是在一个线程中插入一个 APC 可以打断该线程（如果它正

在 PASSIVE_LEVEL 上）运行的原因。DISPATCH_LEVEL 是一个重要的区分点，它代表了线程调度器正在运行。一个处理器运行在此 IRQL 上，说明它正在分配处理器的计算资源，有可能正在做两件事情之一：正在进行线程调度，比如选择新的线程；正在处理一个硬件中断的后半部分（不那么紧急的部分），在 Windows 中，这被称为 DPC（Deferred Procedure Call），DPC 与线程无关。运行在 DISPATCH_LEVEL 上的线程，不会被其他的线程抢占，只有可能被更高级别的中断抢占。3~26 之间的 IRQL 被分配给设备，称为设备 IRQL（或 DIRQL），HAL 规定它们的分配方案。例如，在 Intel x86 多处理器系统中，HAL 会循环地将中断向量号映射到这段 IRQL 范围中。从 IRQL 的角度而言，DIRQL 范围中的 IRQL 并无优先级区别，不同的设备中断只是被映射到相同或不同的 IRQL 而已。理想的分配方案是，按照硬件中断的优先级来分配 DIRQL，但由于多处理器系统使用的 APIC 中断控制器支持 200 多个中断向量，因此，仅仅使用 3~26 这一范围难以精确地表达各个硬件中断之间的优先级和屏蔽关系。

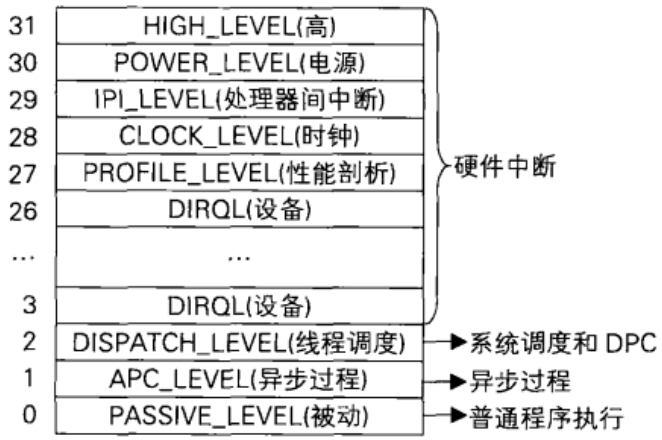


图 5.6 Intel x86 系统的 IRQL 分配方案

PROFILE_LEVEL 是一个专用于性能剖析定时器的 IRQL。当内核的性能剖析功能（利用 Microsoft 提供的 Kernrate 工具）打开时，系统时钟就会周期性地对当前处理器的代码地址（即被中断处的代码的地址）进行采样，从而建立起一张地址采样表，用来分析各个内核模块被执行的情况。在执行内核性能剖析功能时，处理器的 IRQL 将提升到 PROFILE_LEVEL。CLOCK_LEVEL 是系统时钟中断使用的 IRQL，内核利用此 IRQL 来更新系统时间以及触发定时器。IPI_LEVEL 是处理器之间通信的中断级别，它优先于时钟中断。POWER_LEVEL 被定义为电源失败，但并未在 Windows 中真正使用。HIGH_LEVEL 用于一些最高级别的任务，它禁止了所有其他级别的中断；当系统发生不可恢复错误而进入错误检查（BugCheck）状态时，会将 IRQL 提升至 HIGH_LEVEL，以屏蔽所有其他的中断。

Windows 总是运行在一个高度并发的环境中，任何时候，每个处理器都运行在某一IRQL 上。当中断发生时，处理器把被中断的线程的状态保存起来，然后调用 IDT 中设定的中断例程。Windows 提供了一套标准的中断处理机制，允许设备驱动程序通过注册中断对象的方式，将中断服务例程添加到系统中，从而无须操纵处理器的 IDT。下一小节将介绍中断对象的实现方式。

每个处理器的 IRQL 决定了它如何处理中断，以及允许接收哪些中断。另外，IRQL 也被用于实现对一些内核模式数据结构的同步访问。例如，与线程调度相关的数据结构只有在 DISPATCH_LEVEL 上才可以访问。

一个处于内核模式中的线程根据它所要做的事情来提升或者降低当前处理器的 IRQL。尤其是，当线程要访问全局的数据结构或执行关键的任务时，必须考虑是否提升 IRQL，在完成以后，再降低 IRQL。提升 IRQL 可以屏蔽低 IRQL 的线程或中断。但是，作为内核代码，它总是应该尽可能地保持当前处理器的 IRQL 在 PASSIVE_LEVEL 上，这样用户代码才有机会得以运行。改变 IRQL 是通过 KeRaiseIrql 和 KeLowerIrql 函数来完成的，在 Intel x86 体系结构下，这两个函数被定义为 KfRaiseIrql 和 KfLowerIrql，而后者是在 hal.dll 中实现的。处理器的控制数据区 KPCR 结构中有一个 Irql 域记录了该处理器当前的 IRQL。在内核模式代码中调用 KeGetCurrentIrql 可获得当前处理器的 IRQL。

虽然 IRQL 是一个由操作系统定义的逻辑中断级别，但是它涉及与硬件中断控制器的对应，以及需要控制硬件中断优先级，所以，IRQL 的改变不可避免地要与硬件打交道，这是 KfRaiseIrql 和 KfLowerIrql 函数位于 HAL 中的原因。有兴趣的读者可以在调试器中检查这两个函数的实现，代码并不复杂，但可以看到它们对于硬件中断和软件中断的 IRQL 变化是区别对待的。

DISPATCH_LEVEL 是最高的软件中断 IRQL，它低于所有硬件中断的 IRQL。除了系统调度器以外的其他内核代码，一旦运行在这个级别或更高的 IRQL 上，则不得切换到其他线程（比如，等待一个同步对象），因为线程切换是通过系统调度器来完成的，而系统调度器运行在 DISPATCH_LEVEL 上。一个直接的影响是，在中断处理例程中，不能做任何与用户代码或内核低 IRQL 代码进行同步的操作，换句话说，不能做任何等待动作。另一个间接的影响是，在 DISPATCH_LEVEL 或更高的 IRQL 上，不能访问换页内存区，因为一旦发生换页动作，就需要执行 I/O 操作，从磁盘上读入页面，期间至少有一个等待动作。因此，在中断处理例程中，只能访问非换页内存。在 Intel x86 系统上，DPC 也运行在 DISPATCH_LEVEL 上，所以，这些规则均适用于 DPC 代码。在设备驱动程序中，这也是常见的错误原因。

5.2.3 中断对象

中断对象是 Windows 提供的一种中断扩展机制，它允许设备驱动程序将一个中断服务例程（ISR，Interrupt Service Routine）与某一特定的中断向量关联起来。通过中断对象机制，设备驱动程序无须操纵 IDT 就可以处理设备中断。以下是 WRK 中中断对象 KINTERRUPT 的定义：

```
typedef struct _KINTERRUPT {
    CSHORT Type;
    CSHORT Size;
    LIST_ENTRY InterruptListEntry;
    PKSERVICE_ROUTINE ServiceRoutine;
    PVOID ServiceContext;
    KSPIN_LOCK SpinLock;
    ULONG TickCount;
    PKSPIN_LOCK ActualLock;
    PKINTERRUPT_ROUTINE DispatchAddress;
    ULONG Vector;
    KIRQL Irql;
    KIRQL SynchronizeIrql;
    BOOLEAN FloatingSave;
    BOOLEAN Connected;
    CCHAR Number;
    BOOLEAN ShareVector;
    KINTERRUPT_MODE Mode;
    ULONG ServiceCount;
    ULONG DispatchCount;
    ULONG DispatchCode[DISPATCH_LENGTH];
} KINTERRUPT;
```

在一个中断对象中，InterruptListEntry 域使得一组中断对象相互链接起来，与同一个中断向量关联的中断对象构成了一个双链表，因此，当中断发生时，这些中断对象都能被执行。SeviceRoutine 域是中断对象的处理例程，即中断服务例程。Vector 域指明了一个中断对象的向量编号。Irql 域表示该中断处理例程执行时的 IRQL。DispatchCode 域包含了基本的中断处理代码，它是在中断对象初始化时已填好的少量汇编指令。当中断发生时，这些代码首先被执行。Mode 域是一个 C 枚举类型成员，代表了该中断对象的模式，有两种可能：LevelSensitive 或 Latched，LevelSensitive 模式的中断是指，当中断请求信号出现（asserted）时，中断就会发生，因此，中断对象的例程必须消除中断的原因；而 Latched 模式的中断是指，仅当中断请求信号从无信号（deasserted）到有信号（asserted）发生变化时，中断才会发生。

每个中断对象都只能被连接到一个 IDT 项，如果一个服务例程需要被连接到多个处理器的同一个中断上，那么，必须建立多个中断对象。多个中断对象可以被连接到同一个 IDT 项，然而，它们必须使用同样的中断模式，即 LevelSensitive 或 Latched。

为了使用一个中断对象，首先必须调用 KeInitializeInterrupt 函数对它进行初始化。KeInitializeInterrupt 函数的代码位于 base\ntos\ke\i386\intobj.c 文件的 82~226 行。除将参数中的值赋给中断对象中相应的成员以外，KeInitializeInterrupt 函数还将 KiInterruptTemplate 数组中的指令拷贝到中断对象的 DispatchCode 成员中，并且修正其中的一条指令，让它使用当前中断对象。关于 KiInterruptTemplate 数组所指的指令流，可参考 base\ntos\ke\i386\intsup.asm 文件中的_KiShutUpAssembler 函数，其中的“mov edi, 0”指令中的操作数已被修正为中断对象的地址，所以，DispatchCode 中的代码直接在当前中断对象上进行操作。指令“jmp _KeSynchronizeExecution”中的目标地址在接下来的连接步骤中被修正到特定的中断分发例程。

中断对象在经过初始化以后，便可以调用 KeConnectInterrupt 函数，将它挂到中断对象中指定的中断向量的 IDT 项中。KeConnectInterrupt 函数的代码位于 base\ntos\ke\i386\intobj.c 文件的 228~401 行。如果中断对象中指定的中断向量尚未连接任何一个中断对象，则将该中断对象作为链表头，并调用 KiConnectVectorAndInterruptObject 函数将中断对象中的 DispatchCode 代码块连接到此中断向量的 IDT 项上。否则，判断该中断对象是否与该中断向量已有的设置兼容（包括是否允许共享、模式是否相同），若兼容，则将该中断对象插入到与该中断向量相关联的链表尾部，并保证使用链式的中断分发函数，从而使链表中所有的中断对象都有机会被执行。

KeConnectInterrupt 用到了两个辅助函数 KiConnectVectorAndInterruptObject 和 KiGetVectorInfo，并通过数据结构 DISPATCH_INFO 来传递中断设置。其中 KiConnectVectorAndInterruptObject 函数负责把中断对象的 DispatchCode 代码中的“jmp_KeSynchronizeExecution”指令的目标地址修正为与该中断向量相匹配的中断分发函数。系统提供的中断分发函数为：KiInterruptDispatch、KiFloatingDispatch 和 KiChainedDispatch，或者 KiInterruptDispatch2ndLvl、KiInterruptDispatch2ndLvl 和 KiChainedDispatch2ndLvl。其中后三个分发函数针对二级分发，由 HAL 决定。链式中断分发函数，即 KiChainedDispatch 和 KiChainedDispatch2ndLvl 的流程是：针对链表中的每个中断对象，提升 IRQL，获得服务例程自旋锁（即中断对象的 ActualLock 域），调用中断对象的服务例程，然后释放自旋锁，恢复 IRQL。对于 LevelSensitive 模式的中断，一旦链表上有一个中断对象的中断服务例程已经处理了一个中断，则后面的中断对象不再有机会处理该中断；而对于 Latched 模式的中断，中断分发函数会遍历整个链表，因而链表上所有中断对象都可以处理该中断。关于这些中断分发函数的实现，请参考 base\ntos\ke\i386\intsup.asm 中的汇编代码。

KiConnectVectorAndInterruptObject 函数通过 C 宏 KiSetHandlerAddressToIDT 将中断

对象中的中断分发函数插入到 IDT 中。一旦 KeConnectInterrupt 函数完成这一步并打开中断开关，以后当中断发生时，该中断分发函数就会被调用。

基于上述关于中断机制的介绍，我们可以知道，当一个中断发生时，其处理过程如下：

- 中断对象的 DispatchCode 代码块首先获得控制，它把该代码块所在的中断对象放到 edi 寄存器中，然后跳转到该中断向量对应的中断分发函数。
- 在中断分发函数中，它提升 IRQL，获取自旋锁，然后调用中断对象的服务例程，即 KINTERRUPT 的 ServiceRoutine 成员，待返回以后，释放自旋锁，恢复 IRQL。依据不同的中断分发函数以及中断对象的模式，可能还会处理中断对象链表上的其他中断对象，如上文所介绍。
- 最后，中断分发函数通过 Kei386EoiHelper 辅助函数返回。Kei386EoiHelper 中的 iretd 指令结束整个中断处理。

图 5.7 显示了两个中断对象连接到同一个中断向量情形下的中断控制流程。图中的实线箭头代表了实际的控制流，虚线箭头代表了指针关系。

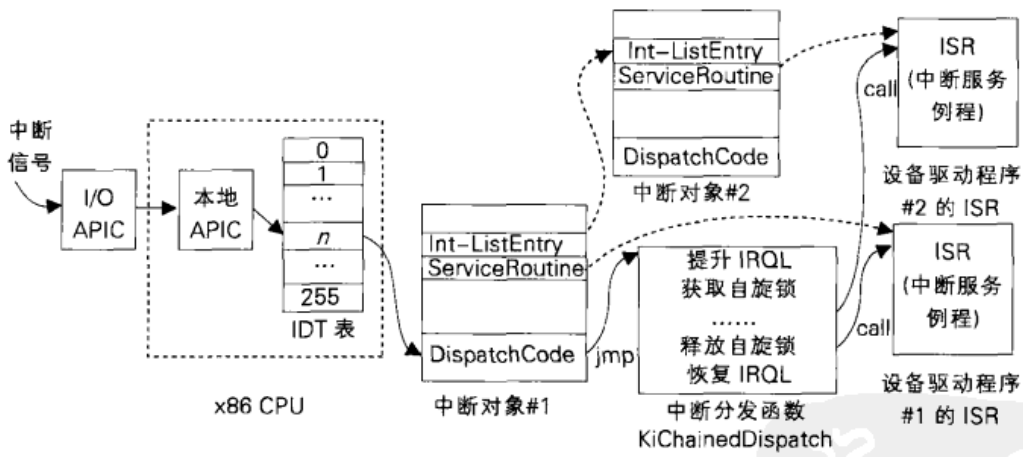


图 5.7 两个中断对象连接到同一个中断向量情形下的控制流程

设备驱动程序并不直接调用 KeInitializeInterrupt 和 KeConnectInterrupt 函数，相反地，它们调用 IoConnectInterrupt 函数来使用中断对象。IoConnectInterrupt 封装了这两个内核函数的使用过程，其代码位于 base\ntos\io\iomgr\iosubs.c 文件的 3 861~4 112 行，这里不再解释。类似地，IoDisconnectInterrupt 函数封装了 KeDisconnectInterrupt 函数的使用。而 KeDisconnectInterrupt 执行一个与 KeConnectInterrupt 相反的过程，它将一个中断对象从相应的中断对象链表中移除，如果该对象是链表中唯一的对象，则断开 IDT 项与中断分发函数

的连接。KeDisconnectInterrupt 函数的代码位于 base\ntos\ke\i386\intobj.c 文件的 403~560 行，这里不再进一步解释。

中断对象是一种强大而方便的系统机制，它允许设备驱动程序通过一种可移植的途径插入它们的中断服务例程，从而支持硬件设备的中断处理。这种能力有重要的意义，它使得硬件设备，包括各种硬件总线、视频设备、网络设备等，共享同样的硬件中断向量。然而，Windows 内核本身并不使用中断对象来处理中断。内核处理的中断包括定时器中断、电源失败、机器错误检查，以及内部使用的一些软件中断。这些中断例程是用汇编语言编写的，往往与具体的系统硬件平台紧密相关。

5.2.4 DPC（延迟过程调用）

中断对象包装了一个在硬件设备 IRQL 上运行的中断服务例程。由于它的 IRQL 高于 DISPATCH_LEVEL，所以，在该例程的执行过程中，不能导致任何线程环境的切换。这意味着它必须在处理器上一直执行，直至例程结束，除非被更高优先级的中断打断。这一节要介绍的 DPC（延迟过程调用，Deferred Procedure Call），它们运行在 DISPATCH_LEVEL 上，低于任何一个硬件中断的 IRQL，所以，DPC 在执行时，不会屏蔽任何硬件中断。但由于它高于或等于任何软件中断的 IRQL，所以，它又优先于任何线程的执行，也屏蔽了线程调度。可以说，DPC 是介于硬件中断和线程调度之间的一个 IRQL，它可以打断当前线程的执行，凌驾于线程调度器之上，但又不屏蔽任何硬件中断。鉴于它的这种特殊能力，DPC 的主要用途是，供设备驱动程序在相对低 IRQL 状态下完成 I/O 任务，以及供内核本身来实现定时器、时限结束处理以及电源失败恢复等任务。

DPC 对象之所以被称为“延迟的”，是因为它往往被用来执行一些相对于当前任务来说不那么紧急的事情，例如，在硬件中断服务例程中，它可以把一些相对不太紧急的事情放到一个 DPC 对象中处理，从而缩短处理器停留在高 IRQL 上的时间。

在 Windows 内核中，DPC 有两种类型：普通的（normal）和线程的（threaded）。普通的 DPC 可以在任何一个线程环境中运行，而线程的 DPC 只能在一个专门的 DPC 线程中运行。在多处理器系统中，每个处理器的 KPRCB 数据结构中都包含两个 DPC 链表，即 KPRCB 的 DpcData 数组成员。DpcData 数组中的两个元素分别代表了普通 DPC 链表和线程 DPC 链表。系统在阶段 1 初始化（Phase1InitializationDiscard 调用 KeInitSystem 函数）时，为每个处理器创建一个专门执行 DPC 的线程，该线程的启动函数为 KiExecuteDpc，其代码位于 base\ntos\ke\dpcsup.c 文件的 47~218 行，它的优先级为最高优先级（HIGH_PRIORITY 或 31）。因此，在 KiExecuteDpc 函数中运行的 DPC 过程，实际上是在线程环境中运行的，

其 IRQL 为 PASSIVE_LEVEL，只不过该线程的优先级最高而已。当插入一个线程 DPC 对象时，一旦当前线程的时限结束，则 KiExecuteDpc 函数中的等待事件就会接到信号，从而触发 DPC 对象被执行（称为交付）。本节接下来的讨论主要针对普通的 DPC 对象。

以下是 WRK 中关于 DPC 对象的定义：

```
typedef struct _KDPC {
    UCHAR Type;
    UCHAR Importance;
    UCHAR Number;
    UCHAR Expedite;
    LIST_ENTRY DpcListEntry;
    PKDEFERRED_ROUTINE DeferredRoutine;
    PVOID DeferredContext;
    PVOID SystemArgument1;
    PVOID SystemArgument2;
    PVOID DpcData;
} KDPC, *PKDPC, *PRKDPC;
```

Type 域说明了 DPC 对象的类型，可能的值为 DpcObject 或 ThreadedDpcObject，这是枚举类型 KOBJECTS 的值，见 base\ntos\inc\ke.h 中关于 KOBJECTS 的定义。Importance 域说明了一个 DPC 对象的重要程度，可以为低（LowImportance）、中（MediumImportance）或高（HighImportance）。当重要程度为高时，DPC 对象被插入到 DPC 链表的头部，否则插入到尾部。Number 域指明了 DPC 对象的目标处理器，即它被插入到哪个处理器的 DPC 链表中，为了将 DPC 对象指定到某一特定的处理器上，应将 Number 域赋为 MAXIMUM_PROCESSORS 加上目标处理器编号，见 KeSetTargetProcessorDpc 函数的做法；否则，DPC 对象被插入到当前处理器的 DPC 链表中。Expedite 域是一个保留成员，在 WRK 中未使用。DpcListEntry 域是 DPC 对象加入到 DPC 链表中的节点对象。DeferredRoutine 域是真正被延迟执行的函数指针。DeferredContext 域是一个指针，它可以指向任意数据结构，在 DPC 对象初始化时指定，当延迟函数被执行时，它也被传递到延迟函数中。SystemArgument1 和 SystemArgument2 域是延迟函数被执行时的两个参数。DpcData 域记录了它被插入到哪个 DPC 链表中。

DPC 对象的用法非常简单，主要涉及三个函数：KeInitializeDpc（实际功能在 KiInitializeDpc 函数中）、KeInsertQueueDpc 和 KeRemoveQueueDpc 函数，它们的代码位于 base\ntos\ke\dpcobj.c 文件中。KeInitializeDpc/KiInitializeDpc 函数初始化一个 DPC 对象，除了 DeferredRoutine 和 DeferredContext 域由参数指定以外，其他的成员均为默认设置，例如，重要程度为中级，Number 为 0。KeInsertQueueDpc 函数将指定的 DPC 对象插入到 DPC 链表中。在插入过程中，IRQL 被提升到最高（HIGH_LEVEL）。KeInsertQueueDpc 首先根据 Number 域的设置，确定目标处理器，并定位到目标 DPC 链

表，获取其自旋锁。如果该 DPC 对象已经在某个链表中了，即它的 DpcData 成员不为 0，则什么也不做。否则，完成插入操作（至链表头或尾），并确定是否要立刻发出一个软件中断。当 DPC 对象的重要程度较高，或目标链表中的 DPC 数量超过了目标处理器设定的最大阈值，或目标处理器的 DPC 请求率小于最低阈值时（参见 dpcobj.c 文件的 403~423 行），KeInsertQueueDpc 函数会请求一个 DISPATCH_LEVEL 的软件中断。如果目标处理器并非当前处理器，则必须要发送 IPI（处理器间中断）请求。最后，释放自旋锁，并恢复 IRQL。

KeRemoveQueueDpc 函数执行一个与 KeInsertQueueDpc 相反的过程，它首先关闭中断，并获取目标处理器的 DPC 链表的自旋锁，然后从链表中移除指定的 DPC 对象，最后释放自旋锁，并打开中断开关。

根据在插入时是否触发软件中断，DPC 对象在以下三种情况下被交付：

- 当处理器的 IRQL 从 DISPATCH_LEVEL 或更高级别降低到 APC_LEVEL 或 PASSIVE_LEVEL 时，内核开始处理该处理器的 DPC 链表中的 DPC 对象，依次调用链表中 DPC 对象的延迟函数，直至链表为空。
- 通过 KeInsertQueueDpc 插入 DPC 对象时，如果依据 DPC 对象的重要程度，以及目标处理器 DPC 链表中的 DPC 数量或请求率，有必要立刻请求一个 DISPATCH_LEVEL 的软件中断，那么，DPC 链表中的 DPC 对象将立即有机会获得处理。
- 在各个处理器的空闲线程中，如果发现有 DPC 对象尚未被执行，则交付这些 DPC 对象。参见 3.4.5 节关于空闲循环的介绍。

在前两种情况下，HAL 调用内核中的 KiDispatchInterrupt 函数（注意，这不同于中断对象分发函数 KiInterruptDispatch），KiDispatchInterrupt 函数进而调用 KiRetireDpcList。在上述第三种情况下，KiIdleLoop 调用 KiRetireDpcList 函数。关于 KiDispatchInterrupt 和 KiIdleLoop 函数的代码，参见 base\ntos\ke\i386\ctxswap.asm 文件。

KiRetireDpcList 函数的代码位于 base\ntos\ke\dpcsup.c 文件的 1 041~1 223 行。其参数是一个指向处理器控制块的指针，所以，它直接从 KPRCB 的 DpcData 成员中得到 DPC 链表，然后在一个 do 循环中处理定时器和 DPC 对象。在循环体中，它首先处理定时器，下一节将介绍定时器的管理。然后在一个内层 do 循环中依次交付 DPC 链表中的 DPC 对象。注意，该函数在访问 DPC 链表时，必须获得 DPC 链表的自旋锁，而在从链表中移除一个 DPC 对象后，立即释放该锁。如此循环，直至 DPC 链表变为空。接下来，KiRetireDpcList

调用 KiProcessDeferredReadyList 以处理当前处理器的延迟就绪链表，使处于延迟就绪状态的线程有机会变成就绪或备用状态。关于 KiProcessDeferredReadyList 函数，请参考 3.5.2 节关于线程状态转移的介绍。

设备驱动程序既可以直接调用 KeInitializeDpc 和 KeInsertQueueDpc 函数来使用 DPC 对象，也可以通过 I/O 管理器的包装函数 IoInitializeDpcRequest 和 IoRequestDpc，实际上，它们并非函数，而是 C 宏，其定义如下（见 base\ntos\inc\io.h）：

```
#define IoInitializeDpcRequest( DeviceObject, DpcRoutine ) (\
   KeInitializeDpc( &(DeviceObject)->Dpc,                \
                    (PKDEFERRED_ROUTINE) (DpcRoutine),    \
                    (DeviceObject) ) )

#define IoRequestDpc( DeviceObject, Irp, Context ) ( \
   KeInsertQueueDpc( &(DeviceObject)->Dpc, (Irp), (Context) ) )
```

图 5.8 显示了一个设备驱动程序在它的中断服务例程中插入一个 DPC 对象，以及 DPC 链表被处理的过程。图中的实线箭头代表了实际的控制流，虚线箭头代表了指针关系。

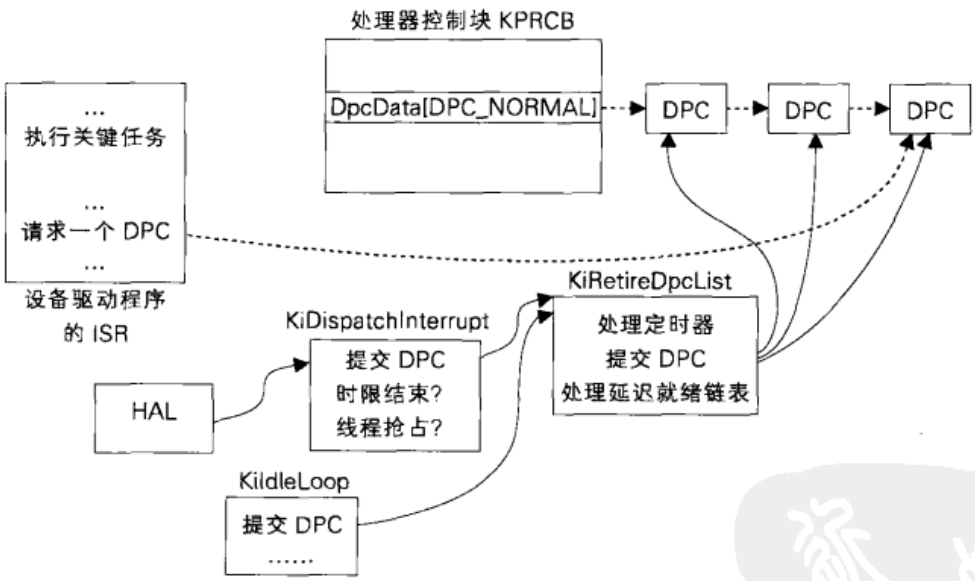


图 5.8 DPC 流程示意图

最后再回顾一下 KiDispatchInterrupt 函数。它除了调用 KiRetireDpcList 函数来交付 DPC 对象，也判断当前处理器的时限结束标志，如果该标志已被置为非 0，则调用 KiQuantumEnd 函数，以确定当前线程的优先级是否要调整，以及是否需要重新调度；如果时限尚未结束，则进一步判断是否有线程抢占当前线程，若是，则直接调用 SwapContext 函数执行线程切换。关于 KiDispatchInterrupt 函数在线程调度中的功能，请参考 3.5.4 节。

5.2.5 时钟中断和定时器管理

定时器（timer）是一种被广泛使用的软件机制，应用软件可以用定时器来实现动画和其他各种周期性任务等，设备驱动程序利用定时器来控制超时，操作系统利用定时器来执行线程调度、时间计量等。Windows 提供了一种定时器内核对象，本章后面 5.4.2 节将会介绍内核提供的各种同步对象，其中就包括定时器对象。这一节我们先讨论 Windows 如何管理定时器和实现定时任务。

Windows 中最底层的定时器机制是通过时钟中断加上 DPC 对象来实现的。我们首先从时钟中断入手来检查最原始的时间控制点。时钟中断是由本地 APIC 来控制的，HAL 模块负责设置时钟中断向量、中断发生的频率，具体的时钟间隔与 HAL 模块有关，通常为 10~15 ms。正如图 5.6 所示，在 Intel x86 处理器的 Windows 系统中，时钟中断的 IRQL 为 28。时钟中断的服务例程由 HAL 提供，它调用了内核模块的 KeUpdateSystemTime 函数。

KeUpdateSystemTime 的代码位于 base\ntos\ke\i386\clockint.asm 文件中。它首先更新中断时间和系统时间（位于用户共享数据区——一个用户模式和内核模式代码均可访问的内存页面），以及滴答计数（全局变量_KeTickCount）。然后，KeUpdateSystemTime 判断定时器数组中是否有定时器到期，如果有，则设置 KPRCB 的 TimerRequest 标志，并通过调用 HalRequestSoftwareInterrupt 函数请求 DISPATCH_LEVEL 软件中断。最后，KeUpdateSystemTime 调用 KeUpdateRunTime 函数，更新线程和进程的运行时间。KeUpdateRunTime 函数的定义紧接在 KeUpdateSystemTime 之后，它首先更新内核模式的总时间、中断所花的时间（如果当前在中断例程中），然后更新当前线程和进程的内核模式时间或用户模式时间。接着，KeUpdateRunTime 函数刷新 DPC 请求率，如果当前 DPC 链表不为空，则有必要的话调用 HalRequestSoftwareInterrupt 函数请求 DISPATCH_LEVEL 软件中断。最后，KeUpdateRunTime 扣除当前线程的时限，扣除的单位为 3（CLOCK_QUANTUM_DECREMENT），参考 3.5.3 节中关于线程时限管理的介绍。如果扣除以后线程的时限仍大于 0，则函数返回，否则设置 KPRCB 的 QuantumEnd 标志，并请求一个 DISPATCH_LEVEL 的软件中断。

值得一提的是，在多处理器系统中，第一个处理器（即 P0）上的时钟中断例程调用 KeUpdateSystemTime 函数，而其他处理器上的时钟中断例程调用 KeUpdateRunTime 函数。从上述两个函数的功能不难理解，系统全局的时间信息由第一个处理器来维护，而其他的处理器只负责处理该处理器上的 DPC 以及与线程调度有关的事项。

在 KeUpdateSystemTime 和 KeUpdateRunTime 函数中，有三件事情可以导致立即请求一个 DISPATCH_LEVEL 的软件中断：有 DPC 需要处理、定时器到期，以及当前线

程的时限已结束。根据上一节的介绍，DISPATCH_LEVEL 软件中断的处理函数 KiDispatchInterrupt，以及它所调用的函数 KiRetireDpcList，正好可以完成这些事情。关于 DPC 处理和时限结束处理，我们不再讨论。现在我们来看 Windows 中定时器的管理。

系统中所有的定时器构成了一个链表数组，全局数组 KiTimerTableListHead 包含了 512 个定时器链表，每个链表都有一个时间值，链表中的定时器为 KTIMER 对象。相关的 C 定义如下：

```
#define TIMER_TABLE_SIZE 512

typedef struct _KTIMER_TABLE_ENTRY {
    LIST_ENTRY Entry;
    ULARGE_INTEGER Time;
} KTIMER_TABLE_ENTRY, *PKTIMER_TABLE_ENTRY;

DECLSPEC_CACHEALIGN KTIMER_TABLE_ENTRY KiTimerTableListHead[TIMER_TABLE_SIZE];

typedef struct _KTIMER {
    DISPATCHER_HEADER Header;
    ULARGE_INTEGER DueTime;
    LIST_ENTRY TimerListEntry;
    struct _KDPC *Dpc;
    LONG Period;
} KTIMER, *PKTIMER, *PRKTIMER;
```

任何一个定时器都有一个指定的到期时间，即 KTIMER 结构中的 DueTime 值。当一个定时器被插入到 KiTimerTableListHead 中时，它被插入到哪个链表是由 DueTime 决定的，即根据 DueTime 的值，计算得到一个下标 Index 值。KiComputeTimerTableIndex 函数完成这一计算，见 base\ntos\ke\i386\timindex.asm 文件。此计算式的基本思路是，将 DueTime 除以每个滴答（tick）的最大间隔时间，即把 DueTime 转换成以滴答为单位，然后模上 TIMER_TABLE_SIZE（在代码中使用了 and 指令，对于 2 的幂次来说，这是等价的）。

因此，每次时钟中断经过一个滴答，它只需检查 KiTimerTableListHead 数组中的一个链表，即符合当前滴答时间值作为到期时间的定时器链表，同时也判断属于下一个滴答的定时器链表。我们在 KeUpdateSystemTime 函数中可以看到，如果检测到当前滴答计数（全局变量 KeTickCount）或下一个滴答计数所对应的链表中的 Time 域已经小于等于当前中断时间值，则设置当前处理器的 KPRCB 中的 TimerRequest 标志，并且设置 KPRCB 的 TimerHand 域指向刚刚到期的定时器链表的数组下标，然后调用 HalRequestSoftwareInterrupt 函数请求 DISPATCH_LEVEL 软件中断。

如果 KiTimerTableListHead 数组中的定时器链表为空，则它的 Time 域的高 32 位部分为 0xffffffff，在实践中，它不可能满足到期条件。现在我们来看定时器被插入链表的做法。KiInsertTimerTable 函数接受一个 KTIMER 对象，以及一个 Hand 值。此 Hand 值必须等于

定时器对象的 DueTime 值所对应的下标，即通过 KiComputeTimerTableIndex 函数计算得到的结果。因此，KiInsertTimerTable 函数定位到待插入的定时器链表，依据定时器的 DueTime 值的先后顺序，插入到链表中合适的位置上。如果恰好插入在链表首，则更新该链表的 Time 值，并根据当前中断时间（通过 C 宏 KiQueryInterruptTime 获得）来判断新插入的定时器是否已经到期。然后返回。KiInsertTimerTable 函数的代码非常直截了当，见 base\ntos\ke\timersup.c 文件的 94~195 行。

综上所述，我们可以知道，如果将定时器的到期时间 DueTime 换算成滴答计数，则 KiTimerTableListHead 数组的每个链表中的定时器对模 TIMER_TABLE_SIZE 同余，并且，这些定时器按照到期时间先后排列，链表对象的 Time 值等于第一个定时器的 DueTime。因此，KeUpdateSystemTime 函数只需判断此 Time 值即可。

现在我们来看定时器的到期处理，前面已经提过，这发生在 KiRetireDpcList 函数中。如果 KPRCB 中的 TimerRequest 标志已被置上，则 KiRetireDpcList 调用 KiTimerExpiration 函数，并且将 KPRCB 中的 TimerHand 传递给它。所以，KiTimerExpiration 函数知道该从 KiTimerTableListHead 数组的哪个定时器链表开始处理。

KiTimerExpiration 函数位于 base\ntos\ke\dpcsup.c 文件的 503~852 行。它首先获得当前的时间信息，以便确定要检查的链表范围。KiTimerExpiration 并不仅仅处理由参数 SystemArgument1 指示的定时器链表，相反，它以该下标为起始位置，根据当前的滴答计数，算出一个结束位置，从而确定一个下标范围。然后通过两重循环遍历 KiTimerTableListHead 数组中这段范围内的定时器链表，对每个链表，找到那些已经到期的定时器对象。对于每个满足到期条件的定时器对象，将它从链表中移除，并且置成有信号状态，然后，判断是否有线程正在等待该定时器，如果有，则解除这些等待线程的等待状态；如果是周期性的定时器（根据 KTIMER 的 Period 域），则计算下一次到期时间，再插入到适当的链表中；如果定时器对象指定了一个非空的 DPC 对象（根据 KTIMER 的 Dpc 域），则要么调用 KeInsertQueueDpc 函数将该 DPC 对象插入到相应的 DPC 链表中（若该 DPC 对象被指定在其他的处理器上执行，或者是一个线程 DPC 对象），要么放到一个临时的 DPC 数组中缓存起来（稍后在当前处理器上执行）。KiTimerExpiration 提供了缓存 DPC 对象的能力。它先缓存一批 DPC 对象，然后调用 KiProcessTimerDpcTable 函数直接交付这些 DPC 对象，再继续缓存下一批。KiProcessTimerDpcTable 函数的代码位于 dpcsup.c 文件的 404~501 行，它只是简单地依次调用数组中的 DPC 对象的例程。

在 KiTimerExpiration 函数的内层循环中，如果在搜索一个链表时到达一个不满足到期条件的定时器，若它变成了新的链表首，则更新该链表的 Time 值。

至此，我们已经了解了 Windows 定时器机制的全貌，如图 5.9 所示。Windows 之所以使用一个二维数组来维护系统中所有的定时器对象，是从效率的角度考虑的，KiComputeTimerTableIndex 根据定时器的 DueTime 计算得到一个下标或索引，相当于做了一个优雅的散列处理。与单个长链表相比，定时器的插入和到期处理都可以在相对较短的链表上进行，而且，在 KeUpdateSystemTime 函数中判断是否有到期定时器也可以非常快捷地实现。

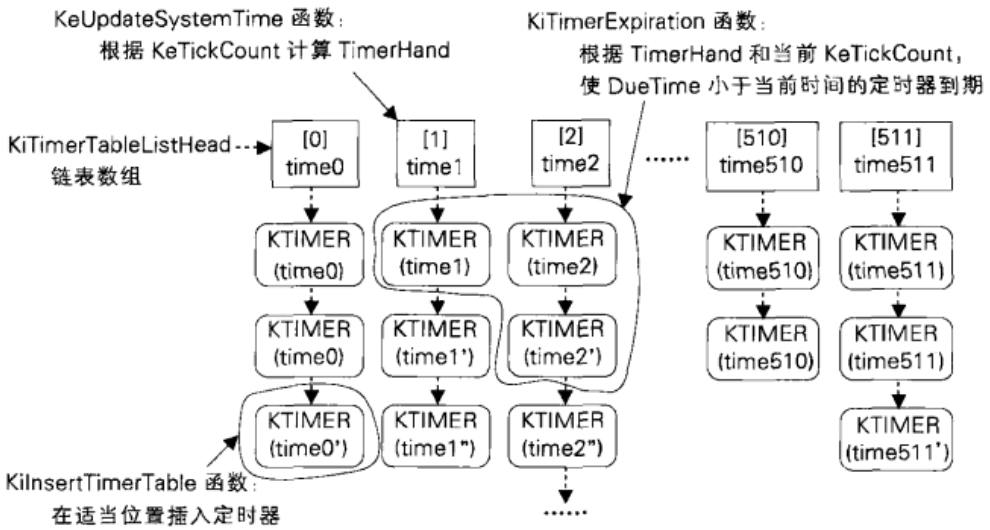


图 5.9 Windows 中的定时器管理结构

从以上的介绍可以看出，定时器对象既具有普通同步对象的特征，也具有 DPC 对象的特征。本节仅仅介绍了定时器对象的到期处理，后面 5.4.2 节还将进一步从同步对象的角度介绍定时器对象。

5.2.6 APC（异步过程调用）

在 Windows 的 IRQL 分配方案中，我们还看到了另一个 IRQL：APC_LEVEL，它位于 PASSIVE_LEVEL 和 DISPATCH_LEVEL 之间，这是专门为另一种称为 APC（异步过程调用，Asynchronous Procedure Call）的软件中断而保留的 IRQL。与 DPC 不同的是，每个 APC 都是在特定的线程环境中运行的，从而也一定在特定的进程环境中运行。DPC 是系统全局的，每个处理器都有 DPC 链表；APC 是针对线程的，每个线程都有自己特有的 APC 链表。同一个线程的 APC 也是被排队执行的。由于 APC 的 IRQL 高于 PASSIVE_LEVEL，所以，它优先于普通的线程代码。当一个线程获得控制时，它的 APC

过程会立刻被执行。这一特性使得 APC 非常适合于实现各种异步通知事件。例如，I/O 的完成通知可以用 APC 来实现。

如同 DPC 一样，APC 也是通过一种内核控制对象来表达的，称为 APC 对象，其定义如下：

```
typedef struct _KAPC {
   UCHAR Type;
   UCHAR SpareByte0;
   UCHAR Size;
   UCHAR SpareByte1;
   ULONG SpareLong0;
   struct _KTHREAD *Thread;
   LIST_ENTRY ApcListEntry;
   PKKERNEL_ROUTINE KernelRoutine;
   PKRUNDOWN_ROUTINE RundownRoutine;
   PKNORMAL_ROUTINE NormalRoutine;
   PVOID NormalContext;
   PVOID SystemArgument1;
   PVOID SystemArgument2;
   CCHAR ApcStateIndex;
   KPROCESSOR_MODE ApcMode;
   BOOLEAN Inserted;
} KAPC, *PKAPC, *PRKAPC;
```

在 KAPC 结构中，Type 域应为 KOBJECTS 枚举类型的 ApcObject；Size 域等于 KAPC 结构的大小；Thread 域指向此 APC 对象所在的线程 KTHREAD 对象；ApcListEntry 域是 APC 对象被加入到线程 APC 链表中的节点对象；KernelRoutine 域是一个函数指针，该函数将在内核模式的 APC_LEVEL 上被执行；RundownRoutine 域也是一个函数指针，当一个线程终止时如果它的 APC 链表中还有 APC 对象，那么，若 RundownRoutine 成员非空，则调用它所指的函数。NormalRoutine 域指向一个在 PASSIVE_LEVEL 上执行的函数。在这三个函数指针成员中，只有 KernelRoutine 是必需的，RundownRoutine 和 NormalRoutine 都是可选的。而且，如果 NormalRoutine 为空的话，则其后的 NormalContext 和 ApcMode 域也将被忽略，本节稍后会解释这三个域的关系。SystemArgument1 和 SystemArgument2 是两个提供给 KernelRoutine 或 NormalRoutine 函数的参数。ApcStateIndex 域说明了 APC 对象的环境状态，它是 KAPC_ENVIRONMENT 枚举类型的成员，一旦 APC 对象被插入到线程的 APC 链表中，则 ApcStateIndex 域指示了它位于线程 KTHREAD 对象的哪个 APC 链表中。最后，布尔类型 Inserted 域指示该 APC 对象是否已被插入到线程的 APC 链表中。

接下来我们重点讨论内核模式 APC。Windows 支持两种内核模式 APC：普通的和特殊的。KAPC 对象中并没有一个标志域来说明它是哪种类型的内核 APC。但从实现上可以这样区分：特殊 APC 是指 NormalRoutine 成员为 NULL 的 APC 对象，而普通 APC 是

指 NormalRoutine 成员不为 NULL，但 ApcMode 成员为 KernelMode 的 APC 对象。特殊 APC 和普通 APC 共用同一个 APC 链表，但是，特殊 APC 位于链表前部，而普通 APC 位于后部。普通 APC 在被交付时，既执行 KernelRoutine 例程，也执行 NormalRoutine 例程（除非在 KernelRoutine 例程执行过程中，将它的一个输出参数 NormalRoutine 清除为 NULL）。而且，特殊 APC 可以抢占普通 APC 的执行，普通 APC 的 NormalRoutine 是在 PASSIVE_LEVEL 上被执行的。如果一个 APC 对象的 NormalRoutine 成员不为 NULL，且 ApcMode 为 UserMode，则这样的 APC 为用户模式 APC。

在介绍 APC 对象的具体用法和实现以前，我们先来讨论 APC 环境的概念。首先，我们回顾一下，在 3.3 节介绍线程的 KTHREAD 数据结构时，看到 KTHREAD 有两个 KAPC_STATE 类型的成员：ApcState 和 SavedApcState，以及一个包含两个元素的指针数组 ApcStatePointer[2]和一个整数索引 ApcStateIndex。当一个线程在它自己所属的进程中运行时，使用的 APC 链表是 ApcState 成员，因此，ApcStatePointer[0]指向 ApcState，并且 ApcStateIndex 等于 0。如果线程被附载（attach）到另一个进程中，那么，尚未交付的 APC 对象被从 ApcState 转移到 SavedApcState 中，并且 ApcStatePointer[0]指向 SavedApcState，ApcStatePointer[1]指向 ApcState，ApcStateIndex 等于 1。所以，附载完成以后，ApcStateIndex 指明了 ApcState 将包含当前进程的信息，包括在新进程环境中要执行的 APC 对象。此后插入的 APC 对象都将进入到 ApcState 链表中。

等到该线程回到它自己的进程（detach）时，KeDetachProcess 首先让属于当前进程的 APC 对象，即 ApcState 中的 APC 对象被交付，然后，将 SavedApcState 中的 APC 转移回到 ApcState 中，并且设置 ApcStatePointer[0]指向 ApcState，ApcStatePointer[1]指向 SavedApcState，ApcStateIndex 等于 0。这样就又恢复了该线程被附载以前的状态。因此，ApcState 总是包含了要在当前进程环境中执行的 APC 对象；ApcStateIndex 总是指向 ApcState 在 ApcStatePointer 数组中的下标值。APC 对象的插入操作总是以 ApcStateIndex 为下标来访问 ApcStatePointer 中的元素。

下面看一下 KAPC_ENVIRONMENT 枚举类型和 KAPC_STATE 结构的定义：

```
typedef enum _KAPC_ENVIRONMENT {
   OriginalApcEnvironment,
   AttachedApcEnvironment,
   CurrentApcEnvironment,
   InsertApcEnvironment
} KAPC_ENVIRONMENT;

typedef struct _KAPC_STATE {
   LIST_ENTRY ApcListHead[MaximumMode]; /*  这里 MaximumMode=2  */
   struct _KPROCESS *Process;
```

```
    BOOLEAN KernelApcInProgress;
    BOOLEAN KernelApcPending;
    BOOLEAN UserApcPending;
} KAPC_STATE, *PKAPC_STATE, *PRKAPC_STATE;
```

因此，引入 APC 环境的概念是为了应对在线程被附载到其他进程以后，以及再回来后的情形。OriginalApcEnvironment 表示原始的进程环境；AttachedApcEnvironment 表示附载以后的新进程环境；CurrentApcEnvironment 表示在 APC 对象初始化时线程所属的进程环境；InsertApcEnvironment 表示在 APC 对象被插入时线程所属的进程环境。而在 APC_STATE 结构中，Process 域指向一个进程对象，代表了这些 APC 所关联的进程。这也解释了在 3.4.2 节提到的 PsGetCurrentProcess 为什么要通过 ApcState 来获得当前进程。ApcListHead 数组有两个元素，分别代表了内核模式 APC 对象链表和用户模式 APC 对象链表。前面提过，两种内核模式 APC（特殊的和普通的）共用同一个链表。三个布尔变量 KernelApcInProgress、KernelApcPending 和 UserApcPending 分别表示：该线程当前正在处理一个普通的内核 APC 对象、有内核模式 APC 对象正在等待交付，以及有用户模式 APC 对象正在等待交付。

理解了 APC 环境的概念以后，我们再来看 APC 对象的用法和实现。在内核代码中，初始化APC对象是通过KeInitializeApc函数来实现的，而插入APC对象则通过KeInsertQueueApc函数来完成。KeInitializeApc 的代码位于 base\ntos\ke\apcobj.c 文件的 29~126 行。其代码非常直截了当，如果参数中指定 APC 环境为 CurrentApcEnvironment，则 APC 对象的 ApcStateIndex 直接等于目标线程的 ApcStateIndex，即使用目标线程当前的进程环境；否则保留 APC 环境参数的值。在 KeInitializeApc 中，APC 对象的线程对象和三个函数指针也被初始化。ApcMode 和 NormalContext 参数仅当 NormalRoutine 参数不为空的时候才有用。

KeInsertQueueApc 函数将一个 APC 对象插入到该 APC 对象中已指定的线程中，其代码位于 apcobj.c 文件的 227~305 行，它将 IRQL 提升至 SYNCH_LEVEL（在多处理器系统中，此 IRQL 的值为 IPI_LEVEL-2，等于 27），并获得线程的 APC 队列锁。如果目标线程的 ApcQueueable 标志为 FALSE，则说明该线程不接受 APC，这发生在线程退出的时候。否则，KeInsertQueueApc 调用 KiInsertQueueApc 函数完成插入 APC 对象的工作。KiInsertQueueApc 的代码位于 base\ntos\ke\apcsup.c 文件的 380~609 行。KiInsertQueueApc 首先根据 APC 环境设置和 ApcStateIndex 成员，确定要插入的目标线程的 APC 链表；然后依据不同的情形，插入到 APC 链表中。用户模式 APC 插入在链表首；普通内核模式 APC 插入在链表尾；对于特殊内核模式 APC，KiInsertQueueApc 从链表尾部开始查找，找到第一个 NormalRoutine 为 NULL 的 APC 对象，然后将新插入的 APC 对象接在它的后面。所以，特殊 APC 和普通 APC 虽然共用同一个链表，但是特殊 APC 总是在普通 APC

对象的前面。然后，KiInsertQueueApc 判断新插入的 APC 对象是否符合其线程的当前进程环境，若是，则进一步处理。如果在当前线程中插入内核模式 APC，并且线程未禁止特殊 APC，则调用 KiRequestSoftwareInterrupt 函数请求一个 APC_LEVEL 的软件中断。如果是其他正在等待的线程，则调用 KiUnwaitThread 函数解除该线程的等待，KiUnwaitThread 会调用 KiReadyThread 函数来唤醒该线程，见 3.5.2 节的介绍。线程醒来后就会交付它的 APC 对象。如果目标线程正在进行门等待，则 KiInsertQueueApc 直接调用 KiInsertDeferredReadyList 函数使它进入延迟就绪状态，从而被调度到就绪和运行状态。如果插入的是用户模式 APC 对象，则只要目标线程是可警醒（alertable）的，就立即调用 KiUnwaitThread 函数使它解除等待。

APC 对象被插入到线程的 APC 链表中以后，一旦 APC_LEVEL 软件中断发生，或者 IRQL 从高降低到 APC_LEVEL 以下，则当前线程的 APC 被交付。这是在 KiDeliverApc 函数中完成的，其代码位于 apcsup.c 文件的 77~378 行。KiDeliverApc 作为软件中断处理例程，只处理当前线程的 APC 链表。只要特殊 APC 没有被禁止，即线程对象的 SpecialApcDisable 标志为 0，就交付内核模式 APC 链表上的 APC 对象。该函数在操纵 APC 链表时，需要将 IRQL 提升到 DISPATCH_LEVEL，并锁住 APC 链表。它在调用每个特殊 APC 对象的 KernelRoutine 例程以前，解锁，并恢复 IRQL。然后交付普通的内核模式 APC 对象，它们也在同一个链表上。对于每一个 NormalRoutine 不为 NULL 的 APC 对象，只要当前线程并非在 APC 处理过程中（即线程对象中 ApcState 成员的 KernelApcInProgress 标志不为 TRUE），并且线程未禁止普通内核模式 APC（即线程对象的 KernelApcDisable 标志为 0），就先调用 APC 对象的 KernelRoutine 例程，把 NormalRoutine 成员指针也传递进去，若 KernelRoutine 返回以后 NormalRoutine 不为 NULL，则继续调用 NormalRoutine 例程。这里，KernelRoutine 是在 APC_LEVEL 上执行的，而 NormalRoutine 是在 PASSIVE_LEVEL 上执行的。并且在这两个例程的执行过程中，线程对象中 ApcState 成员的 KernelApcInProgress 标志被置为 TRUE。因此，在一个普通内核 APC 被交付的过程中，另一个普通内核 APC 不会被交付。

交付完内核 APC 链表中的 APC 对象以后，如果 APC_LEVEL 软件中断发生在用户模式下（即原来的模式是 UserMode），并且该线程确实有用户模式 APC 对象在等待交付，则进入用户模式 APC 对象的交付处理。其过程类似，它在访问 APC 链表时，也要提升 IRQL 至 DISPATCH_LEVEL 并且锁住 APC 链表。类似于普通内核模式 APC 的交付过程，KiDeliverApc 函数通过线程对象中 ApcState 成员的 UserApcPending 标志保证一个用户模式 APC 不会打断另一个用户模式 APC。用户模式 APC 的交付是这样完成的：先调用

APC 对象的 KernelRoutine 例程，将 NormalRoutine 作为参数传递给它，如果它返回以后，NormalRoutine 为 NULL，则调用 KeTestAlertThread 函数，检查该线程是否可以交付另一个用户模式 APC；否则，调用 KiInitializeUserApc 函数，为该用户模式 APC 初始化一个执行环境。

由于用户模式 APC 的 NormalRoutine 是一个在用户模式下运行的函数（位于用户地址空间），而 KiDeliverApc 是在内核模式下运行的，所以，KiDeliverApc 只是调用 KiInitializeUserApc 来设置好用户 APC 例程将来被调用的环境。由于从内核模式到用户模式是通过一个陷阱帧返回的，所以，KiInitializeUserApc 将陷阱帧中的用户模式返回地址（即 Eip 寄存器）设置为 KeUserApcDispatcher 函数的地址，并且将 NormalRoutine 等信息传递到用户栈中恰当的位置上，以便将来 KeUserApcDispatcher 可以访问。这里，KeUserApcDispatcher 是 ntdll.dll 中的函数地址（函数名称为 KiUserApcDispatcher）。KiInitializeUserApc 的代码位于 base\ntos\ke\i386\apcuser.c 文件的 28~188 行。

了解了 APC 对象的交付过程以后，我们来看一看系统在哪些情况下触发 APC 交付：

- 当内核代码离开一个临界区或者守护区（调用 KeLeaveGuardedRegion 或 KeLeaveCriticalRegion）时，通过 KiCheckForKernelApcDelivery 函数直接调用 KiDeliverApc，或者调用 KiRequestSoftwareInterrupt 函数请求一个 APC_LEVEL 的软件中断。这是因为，当线程进入临界区或守护区时，普通内核模式 APC 或特殊内核模式 APC 被禁止了，所以，当离开时，KiCheckForKernelApcDelivery 函数被调用，以便及时地交付内核模式 APC。
- 当一个线程经过一次环境切换而获得控制时，如果有内核模式 APC 需要被交付，则在 KiSwapThread 函数返回以前，调用 KiDeliverApc 函数交付该内核模式 APC。
- 当系统服务或异常处理函数返回到用户模式时，KiDeliverApc 函数被调用以便交付用户模式 APC。
- 在 APC_LEVEL 软件中断发生时，HAL 模块中的软件中断处理函数（HalpDispatchSoftwareInterrupt）调用 KiDeliverApc 来交付内核模式 APC。当内核代码调用 KeLowerIrql 函数降低 IRQL 到 PASSIVE_LEVEL 时，KiDeliverApc 函数也会被调用。

最后我们看一下 APC 对象的用法。由于 APC 对象直接与线程相关，所以它非常适合于向特定的线程发送异步通知。一种典型的用法是，I/O 管理器利用 APC 来发送 I/O 读写操作的完成通知。支持异步文件 I/O 的 Windows API，包括 ReadFileEx 和 WriteFileEx，

使用了用户模式 APC。在内核模式下，除了 KeInitializeApc/KeInsertQueueApc 以外，还有 KeRemoveQueueApc 和 NtQueueApcThread 可以使用。KeRemoveQueueApc 函数将一个已经插入的 APC 对象从链表中移除；NtQueueApcThread 函数接受一个用户模式函数指针，它从非换页内存池申请一个 KAPC 对象，然后插入到指定线程的 APC 链表中，其 APC 对象的 KernelRoutine 是另一个内核函数 PspQueueApcSpecialApc，该函数的功能仅仅是释放 KAPC 对象的内存。NtQueueApcThread 和 PspQueueApcSpecialApc 的代码位于 base\ntos\ps\psctx.c 文件中，读者可以参考该文件，这里不再赘述。

5.2.7 异常分发

与中断不同的是，异常是程序指令流执行过程中的同步过程。这意味着，一旦特定的条件满足，则处理器的控制流会陷入一个异常，待异常得以处理后，此条件消失，原来的指令流继续执行。所以，对于程序的指令流而言，异常就是一个同步过程。异常既可以由处理器硬件产生，也可以由指令流软件产生。与中断相同的是，在 Intel x86 体系结构中，异常也是通过 IDT（中断描述符表）分发的。如表 5.2 所示，0~0x1f 之间的中断向量是 Intel 保留的，除了 2 号中断向量保留给 NMI（不可屏蔽中断）以外，其他已经定义的中断向量都是针对由各种条件引发的异常的。操作系统在 IDT 中填充的表项包含了相应异常的入口处理例程的地址，因此，我们首先从这些入口处理例程开始关于 Windows 中异常的讨论。

从 WRK 中 IDT 的初始定义（见 5.2.1 节最后部分所列的_IDT 定义）可以看到，Windows 的异常处理例程是一些名为_KiTrap??的汇编函数，这里??是十六进制的两位向量号。例如，_KiTrap0E 是页面错误的异常处理例程，因为 0x0E 是页面错误的中断向量号。Windows 提供的异常处理例程，针对各种不同的异常条件，有的直接由系统简单地处理了；有的可能要经过复杂的处理过程，其中涉及内核中多个组件；有的可能会传递给有关的内核组件，甚至交给用户模式的组件来处理。这第三种情形正是 Windows 提供的异常分发机制。而且，由于用户模式代码也有机会参与异常处理过程，这使得系统异常机制可以与编程语言提供的异常机制连接起来。譬如，在 C++程序中，通过 try-catch 语法可以捕捉到一条 DIV 汇编指令产生的异常，原因就在于，处理器陷入异常后，系统给用户程序提供了处理异常的机会。因此，异常本质上是一种系统机制，而不是编程语言的特性。当然，编程语言可以在系统异常机制的基础上进一步扩展，以提供更为灵活的编程设施，不过这是另一个话题，已经超出了本书的范围。

表 5.3 列出了 Intel x86 处理器定义的异常在 Windows 中的处理方式。

表 5.3 Intel x86 异常的处理

向量号	名称	异常处理说明
0	除法错	交由系统分发
1	调试	交由系统分发
2	NMI 中断	系统处理，严重错
3	断点	交由系统分发
4	溢出	交由系统分发
5	BOUND 越界	内核模式异常，则系统退出；用户模式异常，则交由系统分发
6	无效操作码	内核模式异常，则系统退出；用户模式异常，则交由系统分发
7	协处理器不可用	内核模式异常由系统处理，用户模式异常交由系统分发
8	双重错误	严重错误（bugcheck），系统终止
9	协处理器段溢出	仅在 80286 上发生，系统终止
10	无效 TSS	系统处理，终止进程（若用户模式异常）或系统（若内核模式异常）
11	段不存在	内核模式异常，系统终止；用户模式异常，则交由系统分发
12	栈段错误	交由系统分发
13	一般保护错	系统处理段选择符不正确的情形，其他情形交由系统分发
14	页面错误	缺页情形由系统处理；若出错代码的权限不够，则由系统分发
16	x87 浮点	交由系统分发
17	对齐检查	在 Intel x86 上，仅用户模式下可能产生此异常，交由系统分发
18	机器检查	若用户模式异常，则终止进程；若内核模式异常，则终止系统
19	SIMD 浮点异常	交由系统分发

_KiTrap??函数的代码位于 base\ntos\ke\i386\trap.asm 文件中。读者可以参考这些异常处理代码。本节我们要重点讨论的是 Windows 的异常分发机制。首先，我们回顾一下，在 3.3.2 节介绍进程的执行体对象数据结构 EPROCESS 时，曾经看到每个进程都有一个调试端口和一个异常端口（分别是 EPROCESS 的 DebugPort 和 ExceptionPort 成员）。稍后我们将会看到，这两个端口在 Windows 的用户模式异常的分发模型中将会用到。

IDT 中的异常处理例程通过 jmp 指令跳转到异常分发的入口处，根据不同的参数情形，跳转点有所不同（比如 CommonDispatchException0Args），但共同的异常分发入口是汇编过程 CommonDispatchException。这些跳转点以及 CommonDispatchException 过程的汇编代码位于 base\ntos\ke\i386\trap.asm 文件的 2 019~2 090 行。CommonDispatchException 的功能是，在当前栈中分配一条异常记录（exception record），利用异常处理例程指定的参数信息来初始化该异常记录，然后调用 C 函数 KiDispatchException。以下是异常记录 EXCEPTION_RECORD 的定义和 KiDispatchException 的原型：

```
typedef struct _EXCEPTION_RECORD {
   NTSTATUS ExceptionCode;                          // 异常码
```

```
    ULONG ExceptionFlags;                                   // 异常标志
    struct _EXCEPTION_RECORD *ExceptionRecord;              // 相关联的异常记录
    PVOID ExceptionAddress;                                 // 发生异常的指令地址
    ULONG NumberParameters;                                 // 参数数目
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS]; // 参数数组
    } EXCEPTION_RECORD;

VOID
KiDispatchException (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN PKEXCEPTION_FRAME ExceptionFrame,
    IN PKTRAP_FRAME TrapFrame,
    IN KPROCESSOR_MODE PreviousMode,
    IN BOOLEAN FirstChance
    );
```

在讨论 KiDispatchException 函数的逻辑以前，我们先来看异常处理例程_KiTrap??如何向 KiDispatchException 提供参数信息。首先，IDT 的每个异常处理例程都会在入口处，通过 ENTER_TRAP 宏为当前所发生的异常建立起一个陷阱帧。ENTER_TRAP 宏的定义位于 base\ntos\ke\i386\kimacro.inc 文件的 744~831 行，它会将异常发生时的主要寄存器存放到栈中，包括非易失的寄存器（ebp、ebx、esi 和 edi）、段寄存器 fs、当前处理器的异常链表（位于 KPCR 的开头）、发生异常时的处理器模式（内核模式或用户模式）、易失寄存器（eax、ecx 和 edx）、段寄存器（ds、es 和 gs），以及一些调试数据。Intel x86 处理器在异常（或中断）发生时的栈如图 5.10 所示。陷阱帧中的高地址部分是由硬件填充的，包括错误码、eip、cs 和 eflags；如果是用户模式异常，则还包含异常发生前的栈信息；如果是虚拟 86 模式，则另外包含虚拟 86 所用的段寄存器。

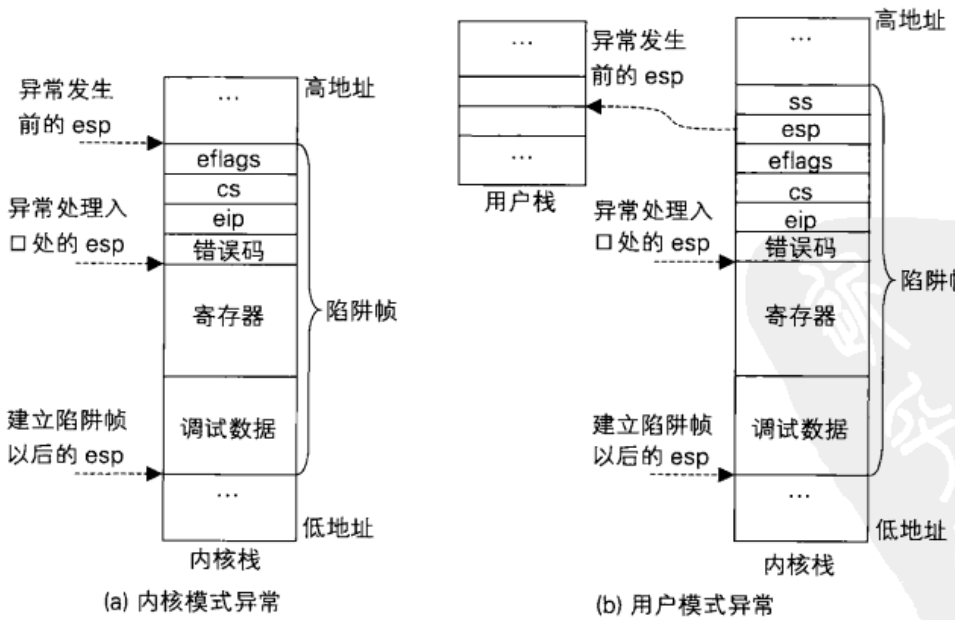


图 5.10 Intel x86 处理器发生异常时的栈

以下是陷阱帧的 C 语言定义，它既适用于异常，也适用于中断：

```
typedef struct _KTRAP_FRAME {
    ULONG   DbgEbp;
    ULONG   DbgEip;
    ULONG   DbgArgMark;
    ULONG   DbgArgPointer;
    ULONG   TempSegCs;
    ULONG   TempEsp;
    ULONG   Dr0;
    ULONG   Dr1;
    ULONG   Dr2;
    ULONG   Dr3;
    ULONG   Dr6;
    ULONG   Dr7;
    ULONG   SegGs;
    ULONG   SegEs;
    ULONG   SegDs;
    ULONG   Edx;
    ULONG   Ecx;
    ULONG   Eax;
    ULONG   PreviousPreviousMode;
    PEXCEPTION_REGISTRATION_RECORD ExceptionList;
    ULONG   SegFs;
    ULONG   Edi;
    ULONG   Esi;
    ULONG   Ebx;
    ULONG   Ebp;
    ULONG   ErrCode;                /* 硬件填充 */
    ULONG   Eip;                    /* 硬件填充 */
    ULONG   SegCs;                  /* 硬件填充 */
    ULONG   EFlags;                 /* 硬件填充 */
    ULONG   HardwareEsp;            /* 硬件填充，仅当有模式切换时才使用 */
    ULONG   HardwareSegSs;          /* 硬件填充，仅当有模式切换时才使用 */
    ULONG   V86Es;                  /* 硬件填充，仅当从 V86 模式切换过来时才使用 */
    ULONG   V86Ds;                  /* 硬件填充，仅当从 V86 模式切换过来时才使用 */
    ULONG   V86Fs;                  /* 硬件填充，仅当从 V86 模式切换过来时才使用 */
    ULONG   V86Gs;                  /* 硬件填充，仅当从 V86 模式切换过来时才使用 */
} KTRAP_FRAME;

typedef KTRAP_FRAME *PKTRAP_FRAME;
typedef KTRAP_FRAME *PKEXCEPTION_FRAME;
```

我们再来看一下异常处理例程的入口代码如何确定发生异常的指令是在内核模式还是用户模式。EFLAGS 寄存器并没有包含处理器模式状态标志。Windows 利用陷阱帧中的 SegCS，即发生异常时的指令所在的段，来确定处理器模式。SegCS 的 0~1 位代表了段选择符的特权级，即异常发生时的当前特权级（CPL），因为 Windows 只用 0 和 3 两种特权级，所以根据 SegCS 的最低位即可确定处理器的模式。关于段选择符和特权级的定义，请参考 4.1.2 节的介绍。

准备好陷阱帧、异常记录和处理器模式这些参数以后，现在回到 KiDispatchException 函数，这是 Windows 异常分发的中心，其代码位于 base\ntos\ke\i386\exceptn.c 文件的

1 033~1 425 行。KiDispatchException 函数首先利用陷阱帧和异常记录参数中的信息，构建一个 ContextFrame 变量，其中包含异常发生时的执行环境信息。注意，ContextFrame 变量的类型为 CONTEXT 结构，该结构也被用于 APC 交付和线程创建过程。提取执行环境信息是通过调用 KeContextFromKframes 函数来完成的。如果是断点异常，则修正 ContextFrame 变量中的 Eip 成员，使它指向断点处的指令；如果是因为执行权限问题而发生的访问违例（异常状态码为 KI_EXCEPTION_ACCESS_VIOLATION，在页面错误 0x0E 异常中的情形），则需要检查是否由于 ATL thunk 而造成了异常。这里的 ATL thunk 是一种程序设计技巧，其做法是，通过一个 thunk 对象（是个数据对象，但其实包含少量指令）将 Windows 窗口过程的窗口句柄以极其高效的方式变换成 C++程序中的 this 指针。

接下来根据发生异常时的处理器模式分别进行处理：

- 对于**内核模式异常**，如果这是第一次机会处理该异常，即 FirstChance 参数为 TRUE，那么，交给内核调试器处理该异常，若内核调试器处理了该异常，则返回，发生异常的指令流继续执行；若不存在内核调试器或内核调试器没处理该异常，则调用 RtlDispatchException 函数，试图将该异常分发到一个基于调用帧的异常处理器（call frame based handler）。只要能找到一个异常处理器能处理此异常，就返回。

  若第一次机会该异常未被处理，则再给内核调试器一次机会。若这次仍未被处理，则进入错误检查（bugcheck）。因此，内核模式的异常必须被处理，一旦未被处理，则系统崩溃。

- 对于**用户模式异常**，处理过程要略微复杂一些。首先判断是否这是第一次机会处理该异常。如果这是第一次机会，若发生异常的进程（即当前进程）的调试端口为空并且有内核调试器，则交给内核调试器；若调试端口不为空，则发送一个消息至调试端口，然后等待应答。如果调试器处理了异常，则返回，继续执行。否则，把异常信息填充到用户栈中，并且设置好用户模式的 Eip 为 KeUserExceptionDispatcher 函数指针（指向 ntdll.dll 中的 KiUserExceptionDispatcher 函数，在进程管理子系统初始化时设置），准备好该函数所需要的参数，然后，KiDispatchException 函数返回。

  因此，当从内核模式的异常处理例程返回以后，ntdll.dll 的 KiUserExceptionDispatcher 函数获得控制，它会试图将该异常分发给一个基于调用帧的异常处理器。如果存在一个基于调用帧的异常处理器处理了该异常，则程序继续执行。如果该异常仍然未被处理，则通过 NtRaiseException 函数再次进入到内核中处理该异常。这一次 KiDispatchException 函数被调用时，FirstChance 参数为 FALSE。

  如果是第二次机会处理一个异常，并且当前进程有一个调试端口，则发送一

个消息至调试端口，然后等待应答。如果进程的调试器处理了该异常，则返回，并继续执行。否则，如果当前进程有一个异常端口，则发送一个消息至异常端口，然后等待应答。如果连接异常端口的环境子系统（Win32 子系统）处理了该异常，则返回，并继续执行。如果该异常仍未被处理，则进程被终止。

在以上介绍的异常分发流程中，有几点值得指出。首先，内核调试器是由 KiDebugRoutine 函数指针来表示的，它也是 Windows 内核的一部分。其次，向调试端口或异常端口发送消息是通过 DbgkForwardException 函数来完成的，DbgkForwardException 的代码位于 base\ntos\dbgk\dbgkport.c 文件中。第三，Windows 内核和用户模式代码都支持基于调用帧的异常处理。在内核中，这是由 RtlDispatchException 函数来完成的，其代码位于 base\ntos\rtl\i386\exdsptch.c 文件的 126~364 行。而在用户模式下，这是由 ntdll.dll 中的 KiUserExceptionDispatcher 函数来完成的。基本的思路是，当一个过程被调用时，在线程的栈帧（stack frame）中包含（或关联）上一个或多个异常处理器，这里的异常处理器是指一块特定的代码，用以处理某种特定类型的异常，因此，RtlDispatchException 或 KiUserExceptionDispatcher 可以沿着栈帧搜索异常处理器。这种异常处理机制也被称为结构化异常处理（SEH，Structured Exception Handling）。自 Windows XP 以后，另一种进程范围的异常处理机制，称为向量化异常处理（VEH，Vectored Exception Handling），也被引入到了 KiUserExceptionDispatcher 的处理流程中。关于 Windows 支持的这两种异常处理机制的细致分析，请参考有关文章[EXCEPT-1][EXCEPT-2]。

异常不仅可以由硬件产生，也可以在程序代码中由软件产生。譬如用户程序利用异常来指示逻辑错误。Windows 提供了一个名为 NtRaiseException 的系统服务，允许用户代码调用 API 函数 RaiseException 来产生软件异常。正如上面关于用户模式异常处理流程的介绍，KiUserExceptionDispatcher 也通过 NtRaiseException 系统服务来传递第二次机会的异常处理。NtRaiseException 调用内核函数 KiRaiseException，而 KiRaiseException 又进一步调用 KiDispatchException 函数来分发异常。关于软件异常的构造和处理过程，读者可以参考 NtRaiseException 和 KiRaiseException 函数的实现，其代码分别位于 base\ntos\ke\i386\trap.asm 和 base\ntos\ke\raisexcp.c 文件中。

在 CommonDispatchException 过程内部，当 KiDispatchException 函数返回以后，它通过一条 jmp 指令跳转到所有异常（和中断）处理例程的总出口处，即 traps.asm 文件的 _KiExceptionExit（或 Kei386EoiHelper）过程。如果这是用户模式异常，并且当前线程有用户模式 APC，那么，在返回用户模式以前，_KiExceptionExit 调用 KiDeliverApc 函数交付这些 APC（参见上一小节关于 APC 的介绍）。这是在汇编宏 DISPATCH_USER_APC 中完成的。然后，恢复必要的寄存器，对于内核模式异常，有必要调整 esp 寄存器的值。最后，通

过 iret 指令回到异常发生处。这些工作在汇编宏 EXIT_ALL 中完成。EXIT_ALL 也被用于系统服务的退出处理。

图 5.11 总结了 Windows 的异常发生和分发流程。在异常分发路径中，如果一个内核模式异常未被处理，则系统最终会崩溃；如果一个用户模式异常未被处理，则引发该异常的线程所在的进程最终会崩溃。与进程关联的调试器有两次机会处理该进程的异常；环境子系统也有机会处理它所管辖的进程中发生的异常。Windows 平台上的错误处理机制 Dr. Watson 和错误报告机制（WER，Windows Error Reporting）通过在一个线程的栈顶插入一个特殊的异常处理器，从而捕捉到未被系统或应用程序处理的异常。

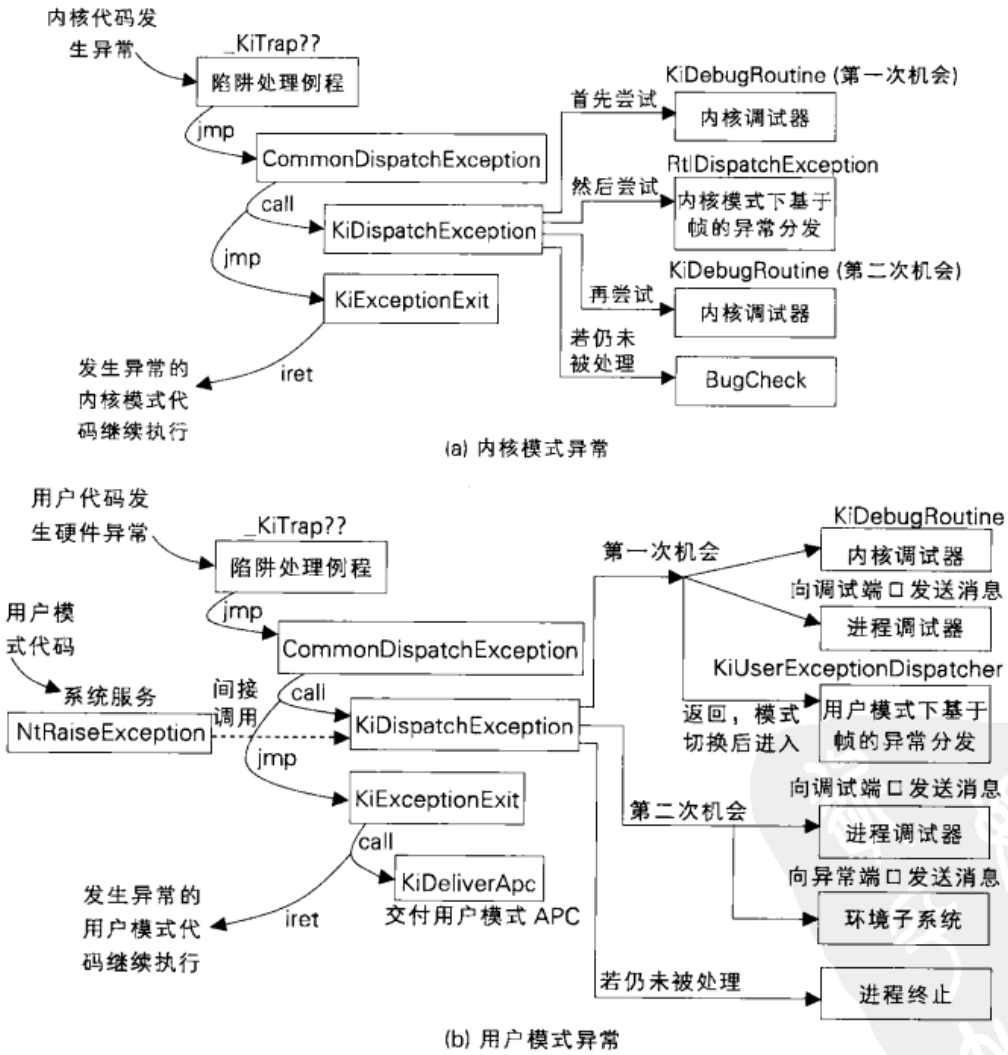


图 5.11 异常分发流程

5.3 不依赖于线程调度的同步机制

我们已经看到了 Windows 的中断和异常处理过程，并了解了各种并发可能性。为了使内核和用户程序能够正确地执行，避免因并发访问而造成数据破坏或发生逻辑错误，Windows 提供了多种机制来保护对共享数据的访问。在本节中，我们将讨论一些不依赖于线程调度的同步机制，包括提升 IRQL、互锁操作、无锁的单链表和自旋锁。这些机制用于 IRQL 较高（大于等于 DISPATCH_LEVEL）的情形，此时 Windows 不适合或不能利用线程切换的能力来同步数据访问。依赖于线程调度的同步机制将在下一节中讲述。

5.3.1 提升 IRQL 实现数据同步

每个处理器都有一个 IRQL 属性，即 KPCR 数据结构的 Irql 域，它表示该处理器当前的中断请求级别。一个基本的规则是，当处理器在某个 IRQL 上运行时，它只能被更高级别的中断打断。这也意味着，它不用担心低 IRQL 的过程会抢占掉当前的执行过程。这一规则对于单处理器系统是很理想的。其做法是，当内核代码要访问一个共享资源时，它只需将 IRQL 提升到任何有可能访问该资源的中断源的最高 IRQL 或以上，就可以屏蔽掉所有可能的访问冲突。一个实际的例子是，当内核需要访问有关线程调度的信息，即每个处理器的就绪队列和就绪摘要（分别对应于 KPRCB 的 DispatcherReadyListHead 和 ReadySummary 成员）时，会将 IRQL 提升至 DISPATCH_LEVEL 或 SYNCH_LEVEL。在 WRK 中，我们可以看到这样的代码（base\ntos\inc\ke.h 文件的 3 069~3 090 行）：

```
#if defined(NT_UP)

#if defined(_X86_)

#define KiLockDispatcherDatabase(OldIrql)          \
   *(OldIrql) = KeRaiseIrqlToDpcLevel()

#define KiTryToLockDispatcherDatabase(OldIrql)     \
   *(OldIrql) = KeRaiseIrqlToDpcLevel(), TRUE

#else

#define KiLockDispatcherDatabase(OldIrql)          \
   *(OldIrql) = KeRaiseIrqlToSynchLevel()

#define KiTryToLockDispatcherDatabase(OldIrql)     \
   *(OldIrql) = KeRaiseIrqlToSynchLevel(), TRUE

#endif

#else   // NT_UP

// 以下是针对多处理器的情形
……
```

在单处理器系统上，仅仅提升 IRQL 就可以确保对资源的独占访问。而在多处理器系统上，这样做还不够，因为其他的处理器可能也位于较高的 IRQL 上，它们可能并发访问资源。因此，在多处理器系统上，还需要使用其他的互斥访问机制来配合，比如后面小节将要介绍的互锁操作或自旋锁。

在 Windows 的 IRQL 分配中，DISPATCH_LEVEL 和 SYNCH_LEVEL 对于资源的同步访问有特殊的意义。首先来看 DISPATCH_LEVEL。运行在该 IRQL 或更高级别的代码，不用担心用户空间代码和 APC 例程会并发访问资源，因为它们的 IRQL 比 DISPATCH_LEVEL 低。另外，所有的 DPC 都工作在 DISPATCH_LEVEL 上，而所有的硬件中断，包括设备中断和 IRQL 为 27~31 之间的中断源，都比 DISPATCH_LEVEL 高。所以，这些硬件中断可以打断线程调度器，或者打断一个正在运行的 DPC。正如 3.5 节所讲，有关线程调度的代码散布在各处，但我们可以认为，所有在 DISPATCH_LEVEL 上工作的代码，除了 DPC 例程以外，合起来构成了 Windows 系统的线程调度器。当内核代码提升 IRQL 至 DISPATCH_LEVEL 时，表示进入调度器，离开调度器是通过 KiExitDispatcher 函数来完成的，它负责将处理器恢复至原来的 IRQL。

SYNCH_LEVEL 是一个针对处理器之间指令流同步的 IRQL。正如 5.2.2 节所介绍，在单处理器系统上，SYNCH_LEVEL 等于 DISPATCH_LEVEL；而在多处理器系统上，它等于 IPI_LEVEL-2。在 SYNCH_LEVEL 上，处理器相互之间可以通过 IPI 中断进行通信，并且时间中断仍然有效，但线程调度和设备中断被屏蔽。例如，当一个处理器需要修改另一个处理器的 KPRCB 结构时，它必须提升 IRQL 至 SYNCH_LEVEL 并获取 KPRCB 的自旋锁，然后才能访问目标处理器的 KPRCB 结构。

在 Windows 的内核代码中，针对 DISPATCH_LEVEL 和 SYNCH_LEVEL，有两个辅助函数专门用于提升 IRQL 至这两个级别：

```
KIRQL
KeRaiseIrqlToDpcLevel (
    VOID
    );

KIRQL
KeRaiseIrqlToSynchLevel (
    VOID
    );
```

最后提一下 HIGH_LEVEL，当一个处理器要屏蔽所有其他的活动时，可以将 IRQL 提升至 HIGH_LEVEL，这通常用于一些关键的任务，比如在系统初始化过程中执行的一些特殊任务，以及当系统休眠时访问整个内存快照。在系统正常运行时，只有一些可快速

完成的任务才适合在 HIGH_LEVEL 上运行。例如，我们可以看到，在 KeInsertQueueDpc 函数中，插入 DPC 对象是在 HIGH_LEVEL 上进行的。

5.3.2 互锁操作

在 5.1.1 节中，我们已经看到了，InterlockedIncrement 函数利用 Intel x86 处理器提供的 lock 指令前缀，可以实现基本的、对整数进行操作的保护。除了 InterlockedIncrement 函数，Windows 内核还实现了一组类似的辅助函数，如表 5.4 所列，它们以原子方式进行整数运算。在每个函数中，当对一个整型内存单元执行运算时，从读取该内存单元，到完成运算并把结果写回内存单元，这整个过程不会被其他的处理器打断，换言之，其他处理器看不到这一操作的中间结果。这些函数直接利用 Intel x86 处理器的原子指令来实现，它们可以在任何 IRQL 上被调用。内核和驱动程序大量使用了这样的函数来保护整数运算。注意，表 5.4 列出的 WRK 函数，有些是 C 内联函数，这意味着读者若在调试器中寻找这些函数的代码，可能无法利用符号信息来查找。

表 5.4 Windows 内核支持的整数原子操作

函数	说明
InterlockedIncrement	对一个 LONG 变量加一，使用 lock xadd 指令
InterlockedDecrement	对一个 LONG 变量减一，使用 lock xadd 指令（加-1）
InterlockedExchange	将一个值交换给一个变量，返回变量原值，使用 xchg 指令，虽然不需要 lock 前缀，但默认为原子操作
InterlockedExchangeAdd	将一个值加到一个变量中，返回变量原值，使用 lock xadd 指令
InterlockedCompareExchange	先做比较，若相等则赋值，使用 lock cmpxchg 指令
InterlockedOr	对一个 LONG 变量做逻辑或运算，使用 lock or 指令
InterlockedAnd	对一个 LONG 变量做逻辑与运算，使用 lock and 指令
InterlockedXor	对一个 LONG 变量做逻辑异或运算，使用 lock xor 指令

注意，根据整数位数的不同，以上列出的这些函数还有一些变体，尤其值得一提的是 64 位的比较交换函数，在 WRK 中为 InterlockedCompareExchange64（这是一个 C 宏）或 ExfInterlockedCompareExchange64，它使用 Intel x86 的 lock cmpxchg8b 指令，直接对一个 64 位内存单元进行原子操作。这些函数或指令可被用于实现无锁（lock-free）的简单数据结构，例如下一小节将要介绍的无锁单链表数据结构。

除了内核中提供的以上这些互锁操作以外，用户空间的系统模块（kernel32.dll）也提

供了类似的函数。因此，用户程序可以直接利用这些函数来保护其基本的数据结构。有兴趣的读者，可以参考有关 Windows SDK 的开发文档，这里不再赘述。

5.3.3 无锁的单链表实现

在内核中使用无锁的数据结构来管理资源，这在多处理器系统上当然是最理想的方案。对于复杂的数据结构，无锁实现方案非常复杂，甚至是不可行的。然而，对于简单的数据结构，譬如其状态不超过 8 字节，那么，使用 Intel x86 处理器本身提供的原子指令，可以相对容易地实现无锁操作。这一节我们讨论 Windows 中无锁单链表的实现。

我们首先来看单链表的数据结构定义，见 public\sdk\inc 目录下的 ntrtl.h 和 ntdef.h 文件：

```
typedef struct _SINGLE_LIST_ENTRY {
   struct _SINGLE_LIST_ENTRY *Next;
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;

#define SLIST_ENTRY SINGLE_LIST_ENTRY
#define PSLIST_ENTRY PSINGLE_LIST_ENTRY

typedef union _SLIST_HEADER {
   ULONGLONG Alignment;
   struct {
      SLIST_ENTRY Next;
      USHORT Depth;
      USHORT Sequence;
   };
} SLIST_HEADER, *PSLIST_HEADER;
```

在 32 位平台上，SLIST_HEADER 的长度为 64 位，如果对于单链表的操作都在链表头上进行，则利用上一小节提到的 InterlockedCompareExchange64 函数，可以用无锁的方式来操纵一个单链表。显然，插入和移除都在头部进行的单链表为 LIFO（后进先出）单链表，它支持 push 和 pop 操作。base\ntos\inc\ex.h 文件中包含了这两个操作的原型声明：

```
NTKERNELAPI
PSLIST_ENTRY
FASTCALL
InterlockedPopEntrySList (
   __inout PSLIST_HEADER ListHead
   );

NTKERNELAPI
PSLIST_ENTRY
FASTCALL
InterlockedPushEntrySList (
   __inout PSLIST_HEADER ListHead,
   __inout PSLIST_ENTRY ListEntry
   );
```

虽然维护链表头可通过原子指令来完成，但是要保证 push 和 pop 操作的正确性，这还不够，因为在原子指令之前，其他处理器可能改变了要操作的链表。因此，为了正确地实现这两个操作，必须小心地设计好原子指令前的准备工作，并且要考虑，若受到其他处理器干扰而导致插入或删除不成功，应该如何进行重试。下面先来看一看 push 操作 InterlockedPushEntrySList 函数的实现，见 base\ntos\rtl\i386\slist.asm 文件的 253~337 行：

```
// 在 base\ntos\rtl\i386\slist.asm 文件中 InterlockedPushEntrySList 函数的实现
cPublicFastCall InterlockedPushEntrySList, 2
cPublicFastCall RtlpInterlockedPushEntrySList, 2
; 由于 FASTCALL，ecx 和 edx 寄存器中分别存放了 ListHead 和 ListEntry 两个参数
;
        push    ebx                     ; 保存 ebx 寄存器
        push    ebp                     ; 保存 ebp 寄存器
        mov     ebp, ecx                ; ListHead 存放到 ebp 中
        mov     ebx, edx                ; ListEntry 存放到 ebx 中
        mov     edx,[ebp] + 4           ; 获得 ListHead->Depth 和 Sequence,放到 edx 中
        mov     eax,[ebp] + 0           ; 获得 ListHead->Next，放到 eax 中
Epsh10: mov     [ebx], eax              ; 设置 ListEntry 的 Next 指针，使它指向链表中当前的 Next
        lea     ecx, [edx+010001H]      ; 增加 ListHead->Depth 和 Sequence，放到 ecx 中

   lock cmpxchg8b qword ptr [ebp]       ; 比较 edx:eax 与 ListHead，若相等，则 ecx:ebx 中
                                        ; 的值复制到 ListHead 中
        jnz     short Epsh10            ; 如果不相等，跳转到 Epsh10 处继续尝试
                                        ; (注意，cmpxchg8b 指令已经将 ListHead 赋到 edx:eax 中了)

        pop     ebp                     ; 恢复 ebp 和 ebx 寄存器
        pop     ebx                     ;

        fstRET  RtlpInterlockedPushEntrySList          ; 返回 eax

fstENDP RtlpInterlockedPushEntrySList
```

上述汇编代码不难理解，只要在 Epsh10 和 cmpxchg8b 指令之间，其他处理器没有操作 ListHead 链表头结构，则 cmpxchg8b 就会成功；如果它不成功，则继续回到 Epsh10 处重试。请注意 Epsh10 点的位置，由于在 cmpxchg8b 指令不成功的情况下，ListHead 处的 8 个字节就又装回到 edx:eax 中了，所以，没有必要将 Epsh10 向前移动到“mov edx,[ebp]+4”指令处。另外，将插入之后的结果 ecx:ebx 赋给 ListHead 是通过 lock cmpxchg8b 指令来完成的，所以，ListHead 的一致性不会被破坏。接下来看一下 pop 操作 InterlockedPopEntrySList 函数的实现，见 slist.asm 文件的 129~247 行：

```
cPublicFastCall InterlockedPopEntrySList, 1
cPublicFastCall RtlpInterlockedPopEntrySList, 1
; 由于 FASTCALL，ecx 寄存器存放了唯一的参数 ListHead

        push    ebx                     ; 保存 ebx 寄存器
        push    ebp                     ; 保存 ebp 寄存器
        mov     ebp, ecx                ; 将 ListHead 存放到 ebp 中

; 以下一条 mov 指令的地址记录下来，若此处发生页面错误，则重试。这是一个重试点
ExpInterlockedPopEntrySListResume:      ;
```

```
Epop10: mov     edx,[ebp] + 4          ; 获得 ListHead->Depth 和 Sequence，放到 edx 中
        mov     eax,[ebp] + 0          ; 获得 ListHead->Next，放到 eax 中

        or      eax, eax               ; 判断链表是否为空
        jz      short Epop20           ; 如果为空，则返回
        lea     ecx, [edx-1]           ; Depth 域减一

; 以下一条 mov 指令的地址记录下来，此处有可能发生页面错误，若发生，则重试
ExpInterlockedPopEntrySListFault:

        mov     ebx, [eax]             ; 获得下一个表项的地址

; 以下指令的地址记录下来，若在此之前直到 ExpInterlockedPopEntrySListResume
; 之间发生任何异常或中断，则回到 ExpInterlockedPopEntrySListResume 处重试
ExpInterlockedPopEntrySListEnd:        ;

  lock cmpxchg8b qword ptr [ebp]       ; 比较 edx:eax 与 ListHead，若相等，则 ecx:ebx 中
                                       ; 的值复制到 ListHead 中
        jnz     short Epop10           ; 若不相等，则继续尝试

Epop20: pop     ebp                    ; 恢复 ebp、ebx 寄存器
        pop     ebx                    ;

        fstRET   RtlpInterlockedPopEntrySList        ; 返回 eax

fstENDP RtlpInterlockedPopEntrySList
```

InterlockedPopEntrySList 函数的基本思路与 InterlockedPushEntrySList 相仿，首先将 ListHead 赋到 edx:eax 中，并判断其 Next 是否为空，若为空，说明链表是空的，直接返回；否则将移除之后的链表头结果放到 ecx:ebx 中，然后通过 cmpxchg8b 指令赋到 ListHead 中。如果在此期间没有其他处理器在操作 ListHead，则 cmpxchg8b 成功，否则，回到 Epop10 处重试。值得注意的是，在 ExpInterlockedPopEntrySListResume 和 ExpInterlockedPopEntrySListEnd 两个地址之间，任何一条指令发生异常或者被中断打断，或者发生 APC 或 DPC 软件中断，Windows 都会将陷阱帧的 eip 恢复到 ExpInterlockedPopEntrySListResume 处，然后重新尝试移除链表头的项。所以，上述代码中的三个指令地址 ExpInterlockedPopEntrySListResume、ExpInterlockedPopEntrySListFault 和 ExpInterlockedPopEntrySListEnd 有特殊的用途，读者可以参考_KiTrap0E 和 KiCheckForSListAddress 函数中的做法，这里不再赘述。

无锁的单链表数据结构对于一些频繁使用的共享资源有重要的意义，它使得多个处理器以极其高效的方式共享一个链表。Windows 的内存管理器和 I/O 管理器是无锁单链表的主要客户，比如，管理系统 PTE 区域的 MiSystemPteSListHead 单链表（见 4.2.3 节）和内存池的缓存链表（见 4.2.2 节）就是典型的例子。最后值得一提的是，从 Windows XP/Server 2003 开始，Windows 的系统模块（ntdll.dll）也提供了同样的一套无锁单链表实现，供用户程序使用。从 KiCheckForSListAddress 函数的实现中，我们可以看到它对用户模式无锁单链表 pop 操作的支持。有关用户模式无锁单链表的用法，请参考 Windows SDK 的文档。

5.3.4 自旋锁

5.1.2 节曾经介绍了自旋锁的概念，它本质上是一种忙等待。一个处理器为了访问一个全局共享资源，不停地检查一个用以保护该资源的锁的状态。一旦其状态变成可用，则立即获取该锁，从而继续后面的执行流程。因此，处理器在获得自旋锁以前，一直在检查锁的状态，而不是选择一个就绪线程来执行，也不交付任何 APC。所以，自旋锁总是在 DISPATCH_LEVEL 或更高的 IRQL 上使用。由于它们妨碍了线程调度和 APC 的交付，因此自旋锁通常仅适合于一些短时间的资源访问。如果一个处理器长时间地抓住一把锁不放，则有可能导致其他处理器空转很长时间（若发生锁竞争）。在预期等待时间很短的情况下，宁可空转一小段时间，而不是“经过环境切换，将控制权交给其他线程，待锁可用以后再切换回来”，这样可以避免两次环境切换，让当前任务继续执行下去。粗略而言，只要空转时间不超过两次环境切换的时间，则使用自旋锁并不会浪费处理器的计算资源。

基于自旋锁的这种特定工作方式，它显然不适合于在单处理器上工作。当有多个处理器在等待同一个自旋锁时，一旦此自旋锁变成可用的，则到底哪个处理器会抢到自旋锁，是不确定的，这取决于这些处理器的旋转时机。自旋锁（spinlock）之所以得名“自旋转”，是因为处理器一直在自转：先测试锁的状态，然后转个圈（CPU 降降温、歇一歇），再测试锁的状态，再转个圈，如此往复，直至锁的状态变成可用，则立即抓住该锁。如果用一个 32 位内存单元来定义锁的状态，则测试锁状态需要使用内存总线，转圈时不必使用内存总线。问题是，如果测试的结果是锁可用，则必须立刻设置它的状态为“已占用”，测试和设置必须在一条原子指令中完成，否则，在测试和设置之间有可能被另一个处理器抢先获取该锁，这是必须要避免的。Intel x86 提供了 bts 指令，可用于对一个内存单元进行测试和设置，加上 lock 前缀即可保证其原子性。Windows 内核实现了两个自旋锁函数 KiAccquireSpinLock 和 KiReleaseSpinLock，分别对应于“获取”和“释放”自旋锁的操作。这两个函数的代码位于 base\ntos\ke\i386\spinlock.asm 文件中，下面列出经过整理的汇编代码（原代码中的宏已被替换，并且与调试有关的代码已去掉）：

```
;/////////////////以下是 KiAcquireSpinLock 函数的汇编代码/////////////////
;  VOID
;  FASTCALL
;  KiAcquireSpinLock (
;     IN PKSPIN_LOCK SpinLock
;     );
;
cPublicFastCall KiAcquireSpinLock  ,1
; 由于 FASTCALL，ecx 寄存器存放了唯一的参数 SpinLock
asl10:  lock bts dword ptr [ecx], 0          ; 测试和设置自旋锁
        jc      asl20                        ; 已被占用，跳转到 asl20 进行旋转
        fstRET   KiAcquireSpinLock           ; 成功获取，返回
```

```
;旋转
asl20:  test    dword ptr [ecx], 1          ; 自旋锁是否已被释放?
        jz      asl10                       ; 是的，回到 asl10，去抓取该锁
        pause                               ; Intel x86 处理器专门提供的用于自旋锁的指令
        jmp     asl20

fstENDP KiAcquireSpinLock

;/////////////////以下是 KiReleaseSpinLock 函数的汇编代码/////////////////
;  VOID
;  FASTCALL
;  KiReleaseSpinLock (
;     IN PKSPIN_LOCK SpinLock
;     );
;
cPublicFastCall KiReleaseSpinLock  ,1
; 由于 FASTCALL，ecx 寄存器存放了唯一的参数 SpinLock
        lock and byte ptr [ecx], 0          ; 仅仅一条原子指令
        fstRET   KiReleaseSpinLock          ; 返回
fstENDP KiReleaseSpinLock
```

以上汇编代码较为直截了当，这里不再解释。由于自旋锁只有两种状态，所以，紧凑的表达方式只需 1 位足够。但 Windows 使用 32 位整数来表达一个自旋锁，从以上代码我们可以看到这一点。另外值得指出的是，Windows 内核层的自旋锁都关联了一个 IRQL，并且 Windows 内核层提供了一组自旋锁函数供内核代码使用，如表 5.5 所列。关于这些函数的实现，读者可以参考 WRK 中的相关代码。

表 5.5 Windows 内核提供的自旋锁函数

函数	说明
KeInitializeSpinLock	初始化一个自旋锁，将自旋锁赋零
KeAcquireSpinLockAtDpcLevel	在 DISPATCH_LEVEL 上获取一个自旋锁
KeReleaseSpinLockFromDpcLevel	在 DISPATCH_LEVEL 上释放一个自旋锁
KeAcquireSpinLockRaiseToSynch	先将 IRQL 提升至 SYNCH_LEVEL，再获取自旋锁
KeAcquireSpinLock	提升 IRQL 至 DISPATCH_LEVEL，获取自旋锁，返回原 IRQL
KeReleaseSpinLock	释放自旋锁，并恢复 IRQL
KeTestSpinLock	检查自旋锁的状态，若已占用，返回 0，否则返回 1
KeTryToAcquireSpinLockAtDpcLevel	试图获取自旋锁，若成功，返回 1，否则返回 0
KeAcquireSpinLockForDpc	根据情况提升 IRQL 至 DISPATCH_LEVEL，并获取自旋锁
KeReleaseSpinLockForDpc	释放自旋锁，然后根据情况降低 IRQL 至原值
KeTryToAcquireSpinLock	试图获取自旋锁，若成功，则 IRQL 提升至 DISPATCH_LEVEL，并返回 1，否则返回 0

内核和设备驱动程序都可以使用自旋锁来保护共享的数据结构，例如我们在第 3 章中

曾经看到，每个进程都有一个进程自旋锁，每个线程都有一个线程自旋锁。设备驱动程序也可以利用自旋锁来保护其数据，因而任何时刻只能有一个处理器在访问其被保护的数据。在使用自旋锁时，需要注意三点：(1) 任何一段代码，其抓住自旋锁的时间不能过长，否则可能会严重影响多处理器的性能；(2) 若内核代码持有了一个自旋锁，则它既不能引发页面错误，也不能调度线程，否则会导致系统崩溃（蓝屏）；(3) 用户程序不能使用自旋锁，因为它们运行在 PASSIVE_LEVEL 上。

除了内核层提供的自旋锁结构（KSPIN_LOCK）和以上这些函数以外，Windows 的执行体层还实现了额外的自旋锁操作。这些操作并非利用 lock bts 指令来测试和设置自旋锁的状态，相反地，它们利用 cmpxchg 指令来完成一次对状态值的原子操作。在执行体的自旋锁实现中，自旋锁是一个计数器，因而允许多个处理器以共享方式同时获取一个自旋锁，但若存在一个处理器以独占方式获取了该自旋锁，则其他的处理器必须等它释放以后才能获取到；若自旋锁已被多个处理器共享，现在一个处理器想要独占此锁，则它必须等所有共享此锁的处理器释放以后才能获取到。自旋锁的状态是一个 32 位整数，其最高位代表是否被独占，余下的位数表示当前已有多少个处理器共享此锁。表 5.6 列出了这组执行体函数的功能。关于它们的实现代码，请参考 base\ntos\inc\ex.h 文件 799~966 行。

表 5.6 Windows 执行体的自旋锁函数

函数	说明
ExAcquireSpinLockShared	提升 IRQL 至 DISPATCH_LEVEL，在循环中测试该锁是否被独占，只要未被独占，则共享计数加一，并返回原 IRQL
ExReleaseSpinLockShared	通过 InterlockedDecrement 将共享计数减一，并降低 IRQL
ExTryAcquireSpinLockExclusive	若自旋锁已被独占，则返回 FALSE。否则，利用 cmpxchg 指令设置独占标志位，若成功，则等待其他共享该锁的处理器释放它。注意，自旋锁计数中包含了当前处理器
ExAcquireSpinLockExclusive	先以共享方式获得自旋锁，再调用上一个函数尝试以独占方式获取它，然后释放属于自己的共享获取
ExReleaseSpinLockExclusive	清除独占标志位，恢复 IRQL

执行体自旋锁在 Windows 中仅被用于大页面内存池的管理。这里介绍执行体自旋锁的意图是，我们可以看到一种既支持独占方式也支持共享方式的自旋锁，并且其实现相对简洁。接下来我们看一种自旋锁的变种。

本节前面提到过，当多个处理器在抢夺自旋锁时，它们获取自旋锁的顺序是不确定的。Windows 内核实现了一种排队的自旋锁，它允许处理器以先进先出（FIFO）的方式获取

自旋锁。在每个处理器的 KPRCB 结构中都有一个 LockQueue 数组成员，数组中的每一项对应于一个全局排队自旋锁，譬如，第一项对应于调度器锁，第三项对应于 PFN 数据库锁。关于这些锁的编号的定义，参见 KSPIN_LOCK_QUEUE_NUMBER 枚举类型（位于 public\sdk\inc\ntkeapi.h 文件中）。因此，排队自旋锁一定是全局自旋锁，在内核设计时指定。处理器要访问排队自旋锁，只需指定锁编号即可。LockQueue 数组中每一项的类型均为 KSPIN_LOCK_QUEUE，其中包含一个 Next 指针和一个自旋锁指针 Lock。由于自旋锁的地址一定是按字对齐的，所以，Lock 成员的最后两位也被用于标志位：第 0 位表示是否在等待；第 1 位表示当前是否占用该锁，即是否为该锁的所有者。KiInitSpinLocks 函数包含了对所有这些排队自旋锁的初始化，参见 base\ntos\ke\kiinit.c 文件中该函数的代码。

排队自旋锁的工作方式如下。如果自旋锁是空闲的，则其值为 0；如果它已被一个处理器获取，则指向该处理器的 KPRCB 中对应于该锁的 LockQueue 数组项。如果有多个处理器正在等待一个排队自旋锁，则以当前已经获取到该自旋锁的处理器的 LockQueue 数组项为链表头，通过 KSPIN_LOCK_QUEUE 的 Next 成员，构成一个排队单链表，自旋锁本身的值指向最后一个插入进来的处理器的 LockQueue 数组项。

基于这样的数据结构，当一个处理器要获取一个已被其他处理器占用或多个处理器正在竞争的排队自旋锁时，将把自己的 LockQueue 数组项插入到链表末尾，并将自旋锁的状态更新为指向自己这一项。然后，它在自己的 LockQueue 数组项的 Lock 成员的第 0 位（即等待位）上旋转，直至该状态位被清除表明已轮到它了。当一个处理器释放排队自旋锁时，它会清除 Lock 成员的第 1 位（即所有者位），然后，若没有处理器在等待该锁，则该自旋锁的状态变为 0，否则，清除排队链表中下一个处理器的 Lock 成员最低 2 位（所以，该处理器的等待标志位被清除，它可以结束旋转），并将自己的 Next 成员清零。图 5.12 显示了两个处理器 P1 和 P2 正在等待一个已被另一个处理器 P0 获取的排队自旋锁的情形。

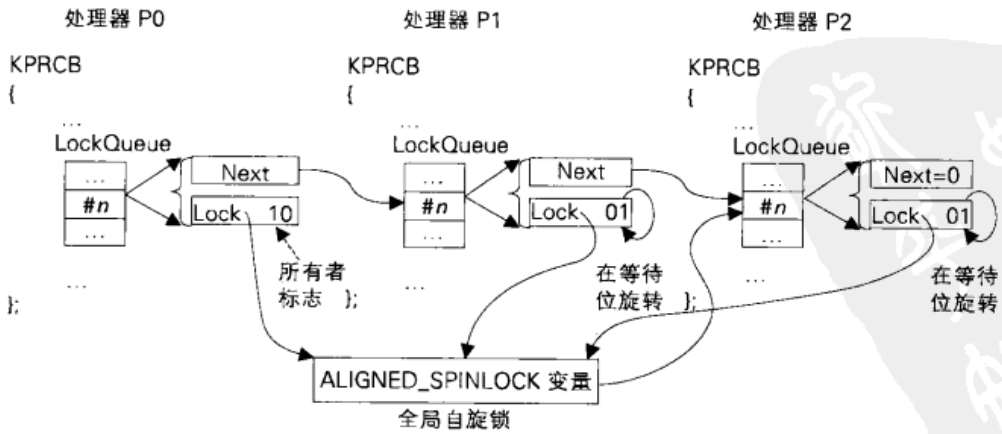


图 5.12 两个处理器在等待一个已被占用的排队自旋锁

排队自旋锁虽然比普通自旋锁复杂，但是，它具有良好的性能，特别是在处理器较多，并且竞争程度较高的情形下，因为每个处理器都只在自己的标志位上旋转，而且，FIFO 获取顺序也意味着多个处理器在竞争锁的时候有了更加一致的行为。在 WRK 中，我们可以看到，每个处理器 KPRCB 的 LockQueue 数组包含 33 项，除了少数几项没有使用以外，这些排队自旋锁项分别被进程和线程管理子系统、内存管理器、I/O 管理器等用于保护一些频繁使用的全局共享数据结构，例如调度器数据库（数组第 0 项）、PFN 数据库（数组第 2 项）、非换页内存池（数组第 6 项）、I/O 数据库（数组第 10 项）等。关于这些锁的定义和分配，参见 KSPIN_LOCK_QUEUE_NUMBER 的定义以及 KiInitSpinLocks 函数中的初始化代码。

Windows 为排队自旋锁提供了一组操作：KeTestForWaitersQueuedSpinLock、KeAcquireQueuedSpinLockAtDpcLevel、KeReleaseQueuedSpinLockFromDpcLevel、KeTryToAcquireQueuedSpinLockAtRaisedIrql、KeAcquireQueuedSpinLock、KeReleaseQueuedSpinLock、KeTryToAcquireQueuedSpinLock、KeAcquireQueuedSpinLockRaiseToSynch 和 KeTryToAcquireQueuedSpinLockRaiseToSynch。如同普通自旋锁一样，这些函数结合了 IRQL 的提升和恢复。这里不再一一解释。

排队自旋锁是由内核使用的，而且它的数量受 KPRCB 结构的 LockQueue 数组大小的影响，无法扩展更多的排队自旋锁。除了这些固有的排队自旋锁以外，Windows 还支持动态分配的排队自旋锁，两个基本的支持函数是 KeAcquireInStackQueuedSpinLock 和 KeReleaseInStackQueuedSpinLock。与普通自旋锁和全局排队自旋锁一样，动态排队自旋锁也有一组结合了 IRQL 提升和恢复的辅助函数。这种排队自旋锁也被称为栈内排队自旋锁（in-stack queued spin lock）。因为没有全局的数据结构支持，所以，KeAcquireInStackQueuedSpinLock 函数需要携带以下参数信息：一个类型为 KSPIN_LOCK 的自旋锁，一个类型为 KSPIN_LOCK_QUEUE 的对象（其中包含 Next 和 Lock 成员），以及一个 IRQL。这后两者被组合成了一个数据结构 KLOCK_QUEUE_HANDLE。栈内排队自旋锁的基本原理与全局排队自旋锁一样，只不过它们的状态位置并非位于 KPRCB 的 LockQueue 数组中而已。

对于栈内排队自旋锁，其 KLOCK_QUEUE_HANDLE 结构可以是一个栈变量，只要获取锁和释放锁是在同一个函数中完成的。栈内排队自旋锁的 KLOCK_QUEUE_HANDLE 结构的生命周期从获取一个锁，即调用 KeAcquireInStackQueuedSpinLock 函数开始，到释放该锁，即调用 KeReleaseInStackQueuedSpinLock 函数以后结束。但锁本身的生命周期可以是全局的，或者随某个对象的生命周期而变化。例如，线程对象有一个 APC 队列锁，即 KTHREAD 的 ApcQueueLock，它在被使用时，是按照栈内排队自旋锁的方式来操作的。

所以，我们在 KiDeliverApc 中可以看到以下的代码：

```
KLOCK_QUEUE_HANDLE LockHandle; // 局部变量声明
……

// 为了访问线程的 APC 队列，首先获取 APC 队列锁
KeAcquireInStackQueuedSpinLock(&Thread->ApcQueueLock, &LockHandle);

// 操作 APC 队列，不用担心其他处理器会访问该线程的 APC 队列
……

// 访问 APC 队列完毕以后，释放 APC 队列锁
KeReleaseInStackQueuedSpinLock(&LockHandle);
// LockHandle 不再使用了
```

关于本节介绍的所有这些自旋锁，最后需要说明的一点是，每一个自旋锁都是为了保护特定的数据而存在。为了让自旋锁机制真正有效地保护目标数据，所有的代码都必须遵守“先获得锁，再访问数据，访问完毕后应立即释放锁”的规则。若有任何一段代码不遵守此规则，则自旋锁的保护将不再有效。试想，在以上代码片段例子中，若内核中有一段代码在不获取 Thread->ApcQueueLock 的情况下，便访问该线程的 APC 队列，则 APC 队列将有可能被破坏，甚至导致严重错误。

5.4 基于线程调度的同步机制

上一节介绍了处理器在 DISPATCH_LEVEL 或更高 IRQL 上使用的一些同步机制，此时处理器不适合或不能切换到其他的线程。然而，当处理器处于 PASSIVE_LEVEL 或 APC_LEVEL 时，它可以将执行权交给其他已经就绪的线程，从而更加有效地利用系统的计算资源。这一节我们讨论线程之间等待方式的同步机制。

5.4.1 线程进入等待

正如 3.5.2 节所介绍的那样，当一个线程的控制流到达一个等待函数时，若等待的条件不满足，则线程调度器会将处理器的执行权交给其他处于备用或就绪状态的线程。以后，当这个线程的等待条件满足时，系统会通过 KiUnwaitThread 与 KiReadyThread 函数使该线程再变成延迟的就绪状态，从而使该线程可以继续执行。在此过程中，线程对象的一个等待块链表，即 KTHREAD 的 WaitBlockList 域，记录了该线程正在等待的那些对象。随着这些对象的状态发生变化，该线程的执行条件将有可能被满足。所以，其他的线程可以通过改变这些对象的状态来控制等待线程的执行与否。

这些被等待的对象可以用来协调线程之间的行为，它们被称为同步对象或者分发器对

象。在 Windows 内核中，凡是头部以 DISPATCH_HEADER 开始的对象都是分发器对象，以下是 DISPATCHER_HEADER 的定义（见 base\ntos\inc\ntosdef.h 文件）：

```
typedef struct _DISPATCHER_HEADER {
   union {
      struct {
         UCHAR Type;
         union {
            UCHAR Absolute;
            UCHAR NpxIrql;
         };

         union {
            UCHAR Size;
            UCHAR Hand;
         };

         union {
            UCHAR Inserted;
            BOOLEAN DebugActive;
         };
      };

      volatile LONG Lock;
   };

   LONG SignalState;
   LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;
```

DISPATCHER_HEADER 的第一个成员，要么是一个锁 Lock，要么按 4 个字节来解释，其中 Type 域代表了该对象的类型；第二个成员 SignalState 代表该分发器对象的信号状态；第三个成员 WaitListHead 是一个链表头，此链表包含了所有正在等待该分发器对象的线程。

现在我们来看线程和分发器对象之间的关系。首先，线程对象的 KTHREAD 结构包含一个 WaitBlockList 成员，它指向一个链表，链表上的每个节点代表了该线程正在等待的一个分发器对象。其次，分发器对象也包含一个链表，即 DISPATCHER_HEADER 的 WaitListHead 成员，链表上的每个节点代表了正在等待该对象的一个线程。这两个链表上的节点都是等待块对象，其数据类型为 KWAIT_BLOCK，定义如下：

```
typedef struct _KWAIT_BLOCK {
   LIST_ENTRY WaitListEntry;
   struct _KTHREAD *Thread;
   PVOID Object;
   struct _KWAIT_BLOCK *NextWaitBlock;
   USHORT WaitKey;
   UCHAR WaitType;
   UCHAR SpareByte;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *PRKWAIT_BLOCK;
```

每个 KWAIT_BLOCK 对象都描述了哪个线程（Thread 域）在等待哪个对象（Object 域）。等待块对象会同时加入到两个链表中：一个是线程对象中的链表，WaitListEntry 域是相应的节点对象；另一个是分发器对象中的链表，NextWaitBlock 域是该链表上的节点对象。WaitKey 域是指当 Thread 等待 Object 成功从而被解除等待时的完成状态值；WaitType 是 WAIT_TYPE 枚举类型：WaitAll 或 WaitAny，说明线程 Thread 要等待所有对象都变成有信号状态，或者只要任意一个对象变成有信号状态即可。

图 5.13 显示了线程、分发器对象和等待块对象三者之间的关系。

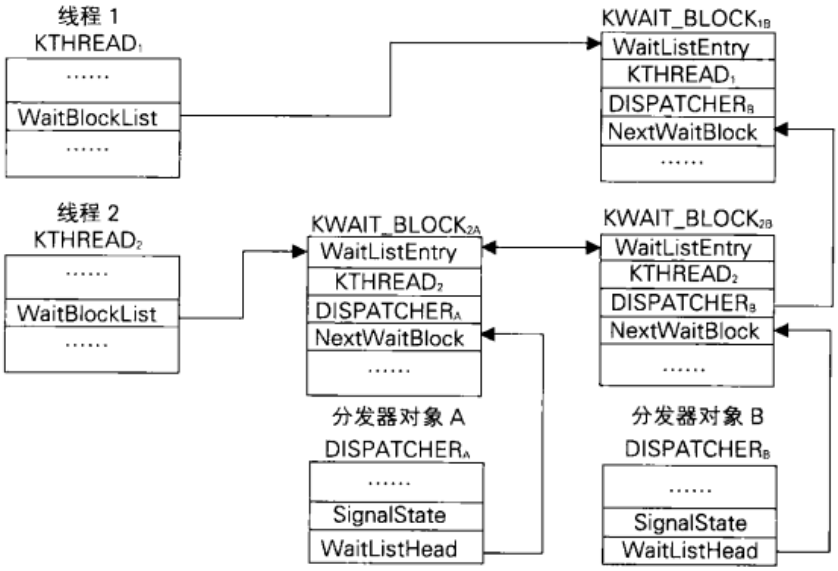


图 5.13 线程和分发器对象之间的等待结构

基于这样的等待结构，我们不难理解，当一个线程通过某个 Wait 函数进入等待状态时，线程对象的 WaitBlockList 指向一个非空链表，链表上的每个等待块对象描述了它所等待的一个分发器对象。只要链表节点上的分发器对象是无信号状态，则该线程始终处于等待状态。在图 5.13 所示的例子中，线程 1 正在等待分发器对象 B，线程 2 正在等待分发器对象 A 和 B。当对象 B 变成有信号状态时，线程 1 的等待条件已满足，于是它变成延迟的就绪状态，以便被调度和执行；而线程 2 的等待条件是否满足，要取决于它的等待类型，KWAIT_BLOCK 中的 WaitType 记录了线程的等待类型，若是 WaitAny，则线程 2 的等待条件满足；若是 WaitAll，则线程 2 继续等待。当对象 A 变成有信号状态时，线程 2 的等待条件满足，于是它变成延迟的就绪状态，以便被调度和执行。另一方面，从对象 B 的角度来看，一旦它变成了有信号状态，则或者同时唤醒两个线程，或者只唤醒其中之一，这取决于对象 B 自身的语义。通常而言，用于同步语义的分发器对象，往往只唤醒链表中的一个等待线程；用于通知语义的分发器对象，则唤醒链表中所有的等待线程。

现在我们来讨论 Windows 中的等待函数如何有效地利用以上这些数据结构来实现等待。3.5.2 节已经介绍了 Windows 内核中的等待函数 KeDelayExecutionThread、KeWaitForSingleObject 和 KeWaitForMultipleObjects 的基本功能，它们的代码位于 base\ntos\ke\wait.c 文件中。

线程对象的 KTHREAD 数据结构预留了 4 个 KWAIT_BLOCK，即 KTHREAD 的 WaitBlock 数组成员。因此，如果一个线程要等待的分发器对象不超过 4 个，则它并不需要申请额外的 KWAIT_BLOCK 对象，直接使用这 4 个预留的等待块对象即可。下面是 KeWaitForSingleObject 函数中构造 KWAIT_BLOCK 链表的代码片段：

```
Thread = KeGetCurrentThread();
Objectx = (PKMUTANT)Object;                // Object 是参数，指定了要等待的对象
OriginalTime = Timeout;                    // Timeout 是指针参数，指定了超时设置
Timer = &Thread->Timer;                    // Timer 指向线程内置的定时器对象，已初始化
WaitBlock = &Thread->WaitBlock[0];         // WaitBlock 和 WaitTimer 是局部指针
WaitTimer = &Thread->WaitBlock[TIMER_WAIT_BLOCK];    // TIMER_WAIT_BLOCK=3

// 以下代码系 C 宏 InitializeWaitSingle 展开所得
Thread->WaitBlockList = WaitBlock;
WaitBlock->Object = Object;
WaitBlock->WaitKey = (CSHORT)(STATUS_SUCCESS);
WaitBlock->WaitType = WaitAny;
Thread->WaitStatus = 0;
if (ARGUMENT_PRESENT(Timeout)) {           // if (Timeout != NULL)
   KiSetDueTime(Timer, *Timeout, &Hand);
   DueTime.QuadPart = Timer->DueTime.QuadPart;
   WaitBlock->NextWaitBlock = WaitTimer;
   WaitTimer->NextWaitBlock = WaitBlock;
   Timer->Header.WaitListHead.Flink = &WaitTimer->WaitListEntry;
   Timer->Header.WaitListHead.Blink = &WaitTimer->WaitListEntry;
} else {
   WaitBlock->NextWaitBlock = WaitBlock;
}
Thread->Alertable = Alertable;
Thread->WaitMode = WaitMode;
Thread->WaitReason = (UCHAR)WaitReason;
Thread->WaitListEntry.Flink = NULL;

// 以下进入一个 do 循环，之所以要有一个 do 循环，是为了在等待过程中可以交付内核 APC，见本节后文解释
......
   InsertTailList(&Objectx->Header.WaitListHead, &WaitBlock-> WaitListEntry);
......
```

在以上代码片段中，KeWaitForSingleObject 使用当前线程的 WaitBlock 数组的第 0 项作为所等待对象的 KWAIT_BLOCK；如果超时值指针 Timeout 不等于 0，则利用 WaitBlock 数组的第 3 项作为一个定时器对象的 KWAIT_BLOCK。在这种情况下，当前线程将等待两个对象：一个是参数中指定的对象，等待类型为 WaitAny；另一个是定时器对象。这两个 KWAIT_BLOCK 对象连接在一起，并且线程对象的 WaitBlockList 指向第一个

KWAIT_BLOCK 对象。定时器对象是在 KeWaitForSingleObject 中临时构造的，它的 DISPATCHER_HEADER 的 WaitListHead 成员指向对应于定时器的 KWAIT_BLOCK 对象。注意，线程的 WaitBlock 数组中的每一项在线程初始化时已经设置好它的 Thread 域，由于线程对象的指针在其生命周期过程中不发生变化，所以，这些预留的 KWAIT_BLOCK 对象无须再设置 Thread 成员。当等待函数进入 do 循环以后，对应于被等待对象的 KWAIT_BLOCK 对象被插入到该对象的 DISPATCHER_HEADER 的 WaitListHead 链表中，如上述代码片段的最后一句所示。

因此，KeWaitForSingleObject 利用 KTHREAD 中预留的 KWAIT_BLOCK 数组来构造出“当前线程等待一个对象，可能还等待一个定时器对象”的结构。然后，根据需要将定时器对象插入到系统的定时器数组链表中。最后，调用 KiSwapThread 函数将处理器的执行权交还给线程调度器。当 KiSwapThread 返回时，意味着此线程又获得了执行权，若不考虑交付内核 APC 的因素，则意味着此等待已满足：或者被等待的对象变成有信号状态，或者超时。

再来看 KeDelayExecutionThread 函数如何使用线程对象的 KWAIT_BLOCK 数组成员。与 KeWaitForSingleObject 相比，它只使用 KTHREAD 的 KWAIT_BLOCK 数组成员的第 3 项，作为一个定时器对象的 KWAIT_BLOCK。这里不再列出其代码，读者可以参考 wait.c 文件的 207~474 行，其处理过程比 KeWaitForSingleObject 简单。

下面看 KeWaitForMultipleObjects 函数如何使用线程对象的 KWAIT_BLOCK 数组成员。显然，由于 KTHREAD 仅仅预留了 4 个 KWAIT_BLOCK，并且数组成员的最后一个保留用于超时处理，所以，只有 3 个 KWAIT_BLOCK 对象可真正被用于调用者指定的分发器对象。KeWaitForMultipleObjects 函数要求，当被等待的对象超过 3 个时，由调用者负责分配一个足够大小的 KWAIT_BLOCK 数组，并且以参数形式传递到函数中，即 KeWaitForMultipleObjects 的 WaitBlockArray 参数。另外，KeWaitForMultipleObjects 函数也限制了最多等待 64 个对象，若超过这个数目，则系统崩溃（调用 KeBugCheck）。

KeWaitForMultipleObjects 函数构造等待块链表的过程与 KeWaitForSingleObject 相仿。同样地，这里不再列出代码，读者可以参考 wait.c 文件的 510~963 行。值得注意的是，KeWaitForMultipleObjects 需要处理一组分发器对象，并且要考虑这些对象的初始信号状态以及参数中指定的等待类型（WaitAny 或 WaitAll）。它需要将 KWAIT_BLOCK 数组中的对象串联起来，构成当前线程对象的 WaitBlockList 链表；还需要将每个 KWAIT_BLOCK 对象都加入到相应分发器对象的 WaitListHead 链表中。关于定时器的处理，与 KeWaitForSingleObject 中的做法完全相同。完成这些工作以后，

KeWaitForMultipleObjects 调用 KiSwapThread，将处理器的执行权交给线程调度器。

理解了进入等待的过程以后，我们再来看一看解除等待的过程。3.5.2 节已经解释过，一些改变对象信号状态的函数，比如 KeSetEvent、KeReleaseSemaphore 和 KeReleaseMutant，它们调用 KiWaitTest、KiWaitTestSynchronizationObject 或 KiWaitTestWithoutSideEffects 函数来测试一个等待是否已经满足。这三个函数都从一个对象变成有信号状态的角度，来检查那些正在等待该对象的线程：是否它们的等待条件已满足，从而可以解除等待，让它们醒过来继续执行。这些函数从分发器对象的 WaitListHead 出发，遍历 KWAIT_BLOCK 链表，逐个进行检查。由于链表上的每个 KWAIT_BLOCK 对象都代表一个线程在等待该对象，所以，它们利用 KWAIT_BLOCK 对象中的信息，获得线程对象（即 Thread 成员）和等待类型（即 WaitType 成员）。然后，调用 KiUnwaitThread 函数，进一步调用 KiReadyThread 函数，使线程又进入延迟的就绪状态。

KiUnwaitThread 函数在调用 KiReadyThread 以前，首先调用 KiUnlinkThread 函数，将一个满足等待条件的线程从它的等待块链表中移除出来，所以，图 5.13 中的线程与分发器对象之间的等待关系会在此函数中被打破。以下是 KiUnlinkThread 函数的代码，位于 base\ntos\ke\ki.h 文件中：

```
FORCEINLINE
VOID
FASTCALL
KiUnlinkThread (
    IN PRKTHREAD Thread,
    IN LONG_PTR WaitStatus
    )
{

    PKQUEUE Queue;
    PKTIMER Timer;
    PRKWAIT_BLOCK WaitBlock;

    // 设置线程的等待完成状态，将线程的等待块从相应对象的等待块链表中移除
    // 同时也将线程从处理器的全局等待链表中移除

    Thread->WaitStatus |= WaitStatus;
    WaitBlock = Thread->WaitBlockList;
    do {
        RemoveEntryList(&WaitBlock->WaitListEntry);
        WaitBlock = WaitBlock->NextWaitBlock;
    } while (WaitBlock != Thread->WaitBlockList);

    if (Thread->WaitListEntry.Flink != NULL) {
        RemoveEntryList(&Thread->WaitListEntry);
    }

    // 如果用于超时控制的线程定时器还在起作用，则取消线程定时器
```

```
    Timer = &Thread->Timer;
    if (Timer->Header.Inserted != FALSE) {
        KiRemoveTreeTimer(Timer);
    }

    // 如果线程正在处理一个队列项，则递增该队列的当前活动线程的计数

    Queue = Thread->Queue;
    if (Queue != NULL) {
        Queue->CurrentCount += 1;
    }

    return;
}
```

KiUnlinkThread 函数将线程的等待块 KWAIT_BLOCK 从相应分发器对象的链表中移除，如果该线程已经启动了一个定时器来控制等待超时的话，则取消该定时器。另外，如果该线程当前正在处理一个队列项，则递增该队列对象的当前活动线程数量。这里队列对象是一种分发器对象，它的典型用法是实现一个线程池，活动线程的数量可以由队列对象来控制。参见下一小节关于队列对象的介绍。

KiUnwaitThread 调用 KiReadyThread 以后，正在等待的线程进入延迟的就绪状态，以后在适当时候被调度，从而再次获得控制权。此时，该线程的控制流回到进入等待时的那个等待函数中，它恢复执行的地点，恰好在 KiSwapThread 调用点返回的地方。KiSwapThread 的返回值正好是上述 KiUnlinkThread 中设定的等待完成状态。如果此返回值不是 STATUS_KERNEL_APC，则说明等待已结束，因而等待函数返回；否则，说明此等待线程被唤醒是因为插入了 APC，因此，解除此等待是为了执行 APC 例程。这正是三个等待函数均使用一个 do 循环来完成等待过程的原因。图 5.14 显示了进入等待和解除等待的典型过程。

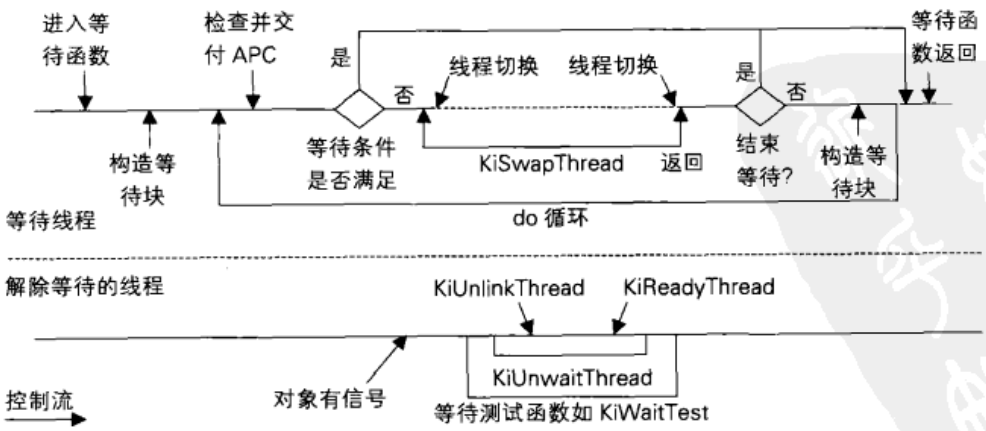


图 5.14 进入等待和解除等待的控制流示意图

对于 KeWaitForMultipleObjects 函数，如果等待类型参数，即 WaitType，是 WaitAll，则情形有一点复杂。假设线程在等待三个分发器对象 A、B 和 C，等待条件是三个对象均为有信号状态。任何一个对象从无信号变成有信号，都必须让线程醒过来，重新判断原始的三个对象是否都为有信号状态，这一判断是在 KeWaitForMultipleObjects 函数中完成的。线程对象的 WaitBlockList 链表总是包含所有它要等待的对象，即使一部分对象已经是有信号状态。每次该线程醒过来时，若 KiSwapThread 的返回值为 STATUS_KERNEL_APC，有可能并不是真正有 APC 需要交付，而是某个被等待的对象变成了有信号状态。在这种情况下，它会重新构造线程对象和分发器对象之间的等待链表结构。一种直观的优化是，让线程的 WaitBlockList 链表只包含那些无信号状态的分发器对象所对应的 KWAIT_BLOCK，这样可以缩短链表长度；并且，在 KiWaitTest 等测试函数中直接检查目标线程的等待条件是否已满足，若不满足，则去掉线程的 WaitBlockList 链表上对应的 KWAIT_BLOCK，而不必唤醒该线程。这样做的好处是可以避免不必要的线程切换。然而，不幸的是，这样的优化是不正确的，因为从线程的 WaitBlockList 链表中移除的 KWAIT_BLOCK，它所对应的分发器对象可能又变成无信号状态了，所以，KeWaitForMultipleObjects 的等待条件仍然要依赖于该对象的状态。例如，初始时，对象 A、B、C 都是无信号状态，经过一段时间后，对象 A 和 B 变成有信号状态，于是线程只需等待 C 即可。但是，又过了一段时间，对象 B 又变成了无信号状态，在优化方案中此时对象 B 已经脱离线程的 WaitBlockList 链表了，因此，当对象 C 变成有信号状态时，线程的等待条件被满足，而实际上对象 B 此时处于无信号状态，等待条件并没有真正被满足。这里的要点是，在构建线程的 WaitBlockList 链表时，无论一个对象的状态如何，都要加入到链表中。

所以，KeWaitForMultipleObjects 函数在每次被唤醒以后，都要重新构造它的等待块链表，并遍历所有指定的对象以判断等待条件是否满足，这样做是正确的。如果真的要去除这些冗余的线程切换操作以作为一种性能优化，那么，正确的做法是，在所有可能导致对象信号状态改变的地方都要测试等待条件满足与否，并且所有被等待的分发器对象，不管有信号状态还是无信号状态，都必须包含在等待线程的 WaitBlockList 链表中。

5.4.2 分发器对象

上一小节讨论了 Windows 的线程等待机制，其核心是通过 KWAIT_BLOCK 对象将线程对象与分发器对象关联起来。在这一等待机制中，分发器对象由 DISPATCHER_HEADER 来描述，其中 Type 域决定了分发器对象的类型；DISPATCHER_HEADER 只是分发器对象的头部，它的对象体（body）紧跟在头部后面。这正符合面向对象语言中的多

态性，也符合软件设计中的机制（或算法）与类型分离的泛型思想。等待机制仅作用于所有分发器对象的共同特征，相当于 C++中的基类，而运行时每个分发器对象都有它自己的特点。这一节我们将介绍 Windows 中各种分发器对象的特殊属性。

表 5.7 列出了 Windows 中所有分发器对象的基本信息。Windows Server 2003 共支持 8 种分发器对象，其中事件对象和定时器对象分别有两种类型：同步和通知。因此，在声明内核对象类型的 KOBJECTS 枚举类型中，共有 10 种内核对象可用于线程等待。

表 5.7 Windows 中的分发器对象一览表

名称	KOBJECTS 中的类型定义	数据结构定义	变成有信号状态的条件
事件	EventNotificationObject 或 EventSynchronizationObject	KEVENT	调用 KeSetEvent 函数设置状态
突变体	MutantObject	KMUTANT	线程释放了突变体对象
信号量	SemaphoreObject	KSEMAPHORE	信号量计数减一
进程	ProcessObject	KPROCESS	最后一个线程终止
线程	ThreadObject	KTHREAD	线程被终止
队列	QueueObject	KQUEUE	在队列中插入一项
门	GateObject	KGATE	调用 KeSignalGateBoostPriority
定时器	TimerNotificationObject 或 TimerSynchronizationObject	KTIMER	设定的时间到期

接下来逐个对象介绍其用法和特性。关于门对象的用法，将在下一小节中介绍。

(1) **事件对象，分为事件通知对象和事件同步对象。**

数据结构 KEVENT 只有一个内嵌的 DISPATCHER_HEADER 对象，它不需要额外的数据特性。与事件对象有关的操作定义在 base\ntos\ke\eventobj.c 文件中，包括：

- KeInitializeEvent，初始化事件对象的类型和信号状态，以及头部的 WaitListHead。
- KeClearEvent，清除事件对象的信号状态，使它变成无信号状态。
- KePulseEvent，设置事件对象为有信号状态，并调用 KiWaitTest 函数以满足那些正在等待该对象的线程，然后设置对象的状态为无信号状态。
- KeResetEvent，设置事件对象为无信号状态，返回原来的状态。
- KeSetEvent，设置事件对象为有信号状态，若为事件通知对象，则调用 KiWaitTestWithoutSideEffects 函数，唤醒所有正在等待该对象的线程；若为事件同步

对象，则调用 KiWaitTestSynchronizationObject 函数，检查它的 KWAIT_BLOCK 链表，唤醒一个正在等待该对象的线程，并且将该对象设置为无信号状态。

- KeSetEventBoostPriority，此操作仅适用于事件同步对象。若它的 KWAIT_BLOCK 链表第一个等待块为 WaitAll，则设置对象为有信号状态，并调用 KiWaitTest-SynchronizationObject 函数；若第一个等待块为 WaitAny，则直接解除所对应线程的等待，不改变事件对象的状态。

事件通知对象和事件同步对象的主要区别在于 KeSetEvent 操作：对于事件通知对象，所有正在等待该对象的线程都将被解除等待，体现了“通知”的含义；而对于事件同步对象，则只唤醒第一个以 WaitAny 方式等待该对象的线程，并重置该对象的信号状态，因而体现了“同步”的含义。

(2) **突变体对象**，这是 Windows 内核中互斥体（mutex）概念的具体实现。

与事件对象不同的是，突变体对象不仅有信号状态，而且，如果它当前无信号，则一定被某个线程所占有，所以，它有所有者的概念。数据结构 KMUTANT 的定义如下：

```
typedef struct _KMUTANT {
   DISPATCHER_HEADER Header;
   LIST_ENTRY MutantListEntry;
   struct _KTHREAD *OwnerThread;
   BOOLEAN Abandoned;
   UCHAR ApcDisable;
} KMUTANT, *PKMUTANT, *PRKMUTANT, KMUTEX, *PKMUTEX, *PRKMUTEX;
```

其中 OwnerThread 域代表了当前正拥有该突变体对象的线程；MutantListEntry 域是一个链表节点。由于突变体对象的所有者概念，所有被一个线程拥有的突变体都将加入到该线程的一个链表中，以 KTHREAD 的 MutantListHead 成员为链表头。因此，当一个线程终止时，它可以把自己拥有的所有突变体释放掉，以便其他的线程有机会获得这些对象（参见 KeRundownThread 函数的代码）。Abandoned 域表明该突变体对象是否已被弃之不用。ApcDisable 域表明是否禁止 APC，会影响到所有者线程的 KernelApcDisable 域。与突变体对象有关的操作定义在 base\ntos\ke\mutntobj.c 文件中，其中针对互斥体（mutex）的操作实际上是突变体（mutant）操作的特例，以下是对这些操作的说明：

- KeInitializeMutant，初始化一个 KMUTANT 对象，并指定当前线程是否为它的所有者。
- KeInitializeMutex，初始化一个 KMUTANT 对象为有信号状态，所以它没有所有者线程。另外，设置 ApcDisable 域为 1，这是不同于 KeInitializeMutant 的地方。
- KeReleaseMutant，释放一个 KMUTANT 对象，递增它的信号计数（即头部的

SignalState 成员）。如果递增后的信号计数为 1，则调用 KiWaitTest 解除一个正在等待该对象的线程。最后返回原来的信号状态。该函数带一个 Abandoned 参数，如果该参数为 TRUE，则强迫释放该对象，并设置信号计数为 1，Abandoned 成员等于 TRUE。注意，只有所有者线程才可以释放一个突变体对象，否则会引发异常。

- KeReleaseMutex，转发给 KeReleaseMutant，其 Abandoned 参数为 FALSE。

此外，在等待函数 KeWaitForSingleObject 和 KeWaitForMultipleObjects 中，当遇到 KMUTANT 对象时，若计数大于 0，或当前线程为所有者线程，则等待该对象可立即满足。在这种情况下，信号计数减 1，若减到 0，则设置当前线程为它的所有者线程，并且将它插入到线程的 MutantListHead 链表中。参见宏 KiWaitSatisfyMutant 和 KiWaitSatisfyAny 的代码，以及它们在这两个等待函数中的用法。

(3) **信号量对象**，Windows 内核中信号量（semaphore）概念的具体实现。

数据结构 KSEMAPHORE 仅在 DISPATCHER_HEADER 后面附加了一个整数 Limit，用于控制最多有多少个线程共享一个资源，即信号量计数器的最大值。与信号量对象有关的操作定义在 base\ntos\ke\semphobj.c 文件中，包括：

- KeInitializeSemaphore，初始化一个 KSEMAPHORE 对象，主要是它的初始计数值（即头部的 SignalState 成员）和最大计数值（即 Limit 域）。
- KeReleaseSemaphore，释放一个 KSEMAPHORE 对象，并指定在它的计数器上增加多少量。如果当前有线程被阻塞在此对象上，则调用 KiWaitTest 函数解除这些线程的等待。这可以从该对象原来的计数值是否为 0 以及它的等待块链表是否为空来判断。

信号量对象的信号状态是一个计数器，但是它没有所有者线程。当一个线程等待一个信号量对象时，只要它的信号计数大于 0，则该等待可以立即满足，并且计数器减 1。请参考宏 KiWaitSatisfyOther 和 KiWaitSatisfyAny 的代码，以及它们在等待函数 KeWaitForSingleObject 和 KeWaitForMultipleObjects 中的做法。

(4) **队列对象**，这是 Windows 内核中用于支持线程池的机制。

数据结构 KQUEUE 用来管理一组线程，它除了 DISPATCHER_HEADER 以外，还维护了额外的数据成员，其定义如下：

```
typedef struct _KQUEUE {
   DISPATCHER_HEADER Header;
   LIST_ENTRY EntryListHead;
```

```
    ULONG CurrentCount;
    ULONG MaximumCount;
    LIST_ENTRY ThreadListHead;
} KQUEUE, *PKQUEUE, *PRKQUEUE;
```

其中 EntryListHead 域是一个链表头，该链表包含了队列对象中待处理的项。MaximumCount 域指定了队列对象中最多可以有多少个活动线程，以便控制通过该队列对象来处理任务的并发程度；CurrentCount 域记录了当前有多少个活动线程；ThreadListHead 域是一个链表头，该链表包含了当前已加入该队列对象的线程，每个节点都为线程 KTHREAD 对象的 QueueListEntry 成员。

有关队列对象的操作位于 base\ntos\ke\queueobj.c 文件中，包括：

- KeInitializeQueue，初始化一个内核队列对象，包括它的链表成员。如果不指定 MaximumCount 的初始值，则默认为处理器的数量。
- KiInsertQueue，这是一个内部辅助函数，其功能是插入一个队列项。如果队列对象的 CurrentCount 小于 MaximumCount，即当前活动线程小于最大允许值，并且，有线程正在等待该队列对象，而且当前线程并非在等待该队列，那么，此等待线程可立即满足，队列对象的 CurrentCount 成员增 1，并调用 KiReadyThread 使之继续执行。如果不满足这些条件，则将参数中指定的项插入到队列对象中的 EntryListHead 链表中，并且信号状态计数器增 1。
- KeInsertQueue，插入一个队列项。其做法是，先提升 IRQL 至 SYNCH_LEVEL，并锁住调度器数据库，然后调用 KiInsertQueue，指定将参数项插入到队列尾部。最后退出时调用 KiExitDispatcher。所以，KeInsertQueue 的功能实际上是由线程调度器来完成的。
- KeInsertHeadQueue，与上一函数相仿，只不过在调用 KiInsertQueue 函数时，指定将参数项插入到队列头部。
- KeRemoveQueue，从队列对象中移除一项，如果队列对象的 EntryListHead 链表为空，则线程进入等待。该函数的逻辑类似于上一小节介绍过的等待函数，这里不多作介绍。该函数退出时也会调用 KiExitDispatcher。
- KeRundownQueue，清除一个队列对象，它会移除掉所有与该队列对象相关联的线程。然后，返回第一个队列项，如果队列项链表为空，则返回 NULL。此函数的功能也是在线程调度器中完成的。

- KiActivateWaiterQueue，这是一个内联辅助函数，其代码位于 base\ntos\ke\ki.h 文件中。此函数的功能反映了队列对象的一个特点：当线程池中一个线程进入等待状态时，其他由于活动线程数量受限制而不得不等待的线程将有机会被执行。上一小节提到的三个等待函数，以及 KeRemoveQueue 和终止线程的函数，都可能调用该函数，因为，一旦队列对象的线程池中正在处理队列项的线程进入等待状态或被终止，则队列对象的活动线程数量将减 1，因而，只要活动线程数量小于最大允许值，并且队列项链表不空，而且有线程正在等待该队列对象，就可以唤醒一个等待线程。

队列对象可以看做是线程调度机制为支持线程池而做的一个扩展，所以，队列对象的主要逻辑都是在线程调度器中完成的。Windows 使用队列对象来实现工作项目（WorkItem）的管理和分发，这里的工作项目是指运行在 System 进程的系统线程中的一些过程，由于它们运行在 PASSIVE_LEVEL 上，所以它们可以像普通的例程一样等待分发器对象或者访问换页内存。DPC 例程或者设备驱动程序可以利用工作项目来完成一些必须在 PASSIVE_LEVEL 上执行的工作。它们通过 ExQueueWorkItem 或 IoQueueWorkItem 把一个工作项目插入到系统中，由专门的系统线程来执行该工作项目中指定的一个过程。

关于内核队列对象的一个典型用途是 I/O 完成端口，见 6.6.5 节关于 I/O 完成端口的介绍。除此以外，在 WRK 中我们还可以看到文件系统运行库（FsRtl）用队列对象来管理线程池，以避免发生栈溢出的情形。当处理一个读页面请求时，若文件系统检测到已接近栈底了，则将该请求交给队列中的辅助线程来处理，从而避免栈溢出的情形。

(5) **进程对象**，第 3 章介绍的内核进程对象本身也是一个分发器对象。

进程对象 KPROCESS 的第一个成员是 DISPATCHER_HEADER 结构，这表明它也是一个分发器对象。在进程对象被创建之初，其状态为无信号；当进程结束时，其状态会变成有信号。所以，如果一个线程通过等待函数来等待一个进程对象，那么，到进程结束时它可以得到通知。

KeSetProcess 函数被用于设置进程对象的信号状态，其功能和做法与事件通知对象相同。KeSetProcess 调用 KiWaitTestWithoutSideEffects 函数，以检查进程对象的等待块链表，并唤醒那些正在等待该进程对象的线程。

在 3.4.4 节介绍进程和线程结束处理时，我们已经看到，当进程的最后一个线程结束时，作为线程退出前的最后步骤之一，PspExitThread 会调用 KeSetProcess，使进程对象变成有信号状态。

(6) **线程对象**，如同进程对象一样，内核线程对象也是一个分发器对象。

线程对象 KTHREAD 的第一个成员也是 DISPATCHER_HEADER 结构。在线程对象被创建之初，其状态为无信号；当线程结束时，其状态变会成有信号。所以，如果一个线程通过等待函数来等待一个线程对象，那么，当被等待的线程结束时，等待线程可以得到通知。

在线程结束处理的最后一个步骤中，PspExitThread 函数会调用 KeTerminateThread 函数，而 KeTerminateThread 又进一步调用 KiWaitTestWithoutSideEffects 以便唤醒所有等待该线程对象的线程，参见 base\ntos\ke\thredobj.c 文件的 2 358~2 361 行。

(7) **定时器对象**，定时器对象既是一个像 DPC 这样的例程的包装对象，也是一个分发器对象，并且分为**定时器通知对象和定时器同步对象**。

在前面 5.2.5 节介绍定时器管理时，我们看到定时器数据结构 KTIMER 的第一个成员是 DISPATCHER_HEADER 结构，因此，KTIMER 对象也可被用于等待函数，换句话说，一个线程可以等待一个定时器对象。有关定时器的操作的实现代码位于 base\ntos\ke\timerobj.c 和 timersup.c 文件中。下面是主要的定时器操作：

- KeInitializeTimer/KeInitializeTimerEx，初始化一个定时器对象。KeInitializeTimer 调用 KeInitializeTimerEx 函数来初始化一个通知类型的定时器。
- KeClearTimer，清除一个定时器对象的信号状态。
- KeCancelTimer，取消一个已经设定的定时器。如果该定时器对象尚未被设定，即 DISPATCHER_HEADER 中的 Inserted 域为 FALSE，则什么也不做，否则将它从系统的定时器链表中移除。
- KeSetTimer/KeSetTimerEx，设定一个定时器。若它已经被插入到了系统的定时器链表中，则首先将它从定时器链表中移除。然后，计算定时器的到期时间，若当前已到期，则调用 KiSignalTimer，以唤醒那些正在等待该定时器的线程，并且以 DPC 方式交付定时器中的到期例程。若当前尚未到期，则设定定时器为无信号状态，并调用 KiInsertOrSignalTimer 函数将它插入到系统的定时器链表中。
- KiSignalTimer，定时器对象变为有信号状态。对于定时器通知对象，调用 KiWaitTestWithoutSideEffects 函数唤醒所有的等待线程；对于定时器同步对象，则调用 KiWaitTestSynchronizationObject，唤醒一个正在等待该对象的线程。对于周期性的定时器，即 KTIMER 的 Period 域不为 0，则再次插入该定时器对象到系统的定时

器链表中。最后，若定时器对象关联了一个 DPC 例程，则调用 KeInsertQueueDpc 函数，将其转化成一个真正的 DPC 对象。

- KiInsertOrSignalTimer，如果定时器对象尚未到期，则将它插入到系统的定时器链表中，否则，调用 KiCompleteTimer 函数，KiCompleteTimer 进一步调用 KiSignalTimer 函数完成定时器对象的到期处理。
- KiTimerExpiration，该函数是系统的定时器到期处理函数，如图 5.9 所示。它遍历特定的定时器链表，对于每个到期的定时器，使之变成有信号状态，并调用 KiWaitTestWithoutSideEffects（针对定时器通知对象）或 KiWaitTestSynchronization-Object 函数（针对定时器同步对象）。

定时器对象是一种常用的分发器对象，内核和设备驱动程序都可以使用定时器对象来实现超时和定时处理。Windows 提供的定时器对象非常灵活：既可以通过到期例程来通知使用者，也允许使用者通过等待函数获得通知；既可以是一次性的，也可以是周期性的；既可以是通知类型，也可以是同步类型。值得指出的一点是，定时器对象的精度受限于系统处理器时钟中断的精度，根据 5.2.5 节的介绍，Windows 仅在时钟中断或 DISPATCH_LEVEL 软件中断发生时，才会处理定时器链表中的到期定时器对象。

最后，我们注意到，每当由于一个分发器对象导致其他的线程被解除等待时，被唤醒的线程都会获得优先级的提升，不同的分发器对象带给等待线程的优先级提升值可能有所不同，事件、信号量和突变体对象的优先级提升值为 1，定时器对象不会提升等待线程的优先级。有关线程的优先级和优先级提升的情形，参见 3.5.1 节的介绍。

由于分发器对象的状态变化会直接影响到系统的线程调度，所以，在以上这些分发器对象的操作函数的代码中，我们常常会看到 IRQL 提升至 DISPATCH_LEVEL 或 SYNCH_LEVEL，并且要锁住调度器数据库，这表明，当一个线程调用这些操作时，它实际上将控制权暂时交给了线程调度器。事实上，系统的线程调度器也正是通过这种方式来工作的，它不仅从代码位置上散布在内核模块各处，而且从运行时间上，也插入在各个线程、中断或异常的执行过程中。绝大多数情况下，进出线程调度器是在同一个函数中完成的，也就是说，如果一个函数进入了调度器，则该函数在返回以前，还会退出调度器。但也有例外，比如 KeSetEvent、KeReleaseMutant、KeReleaseSemaphore 等函数的 Wait 参数就指明了该函数调用以后是否要紧接着调用一个等待函数，若如此，则不退出调度器，而是把这一信息记录在线程对象 KTHREAD 的 WaitNext 和 WaitIrql 成员中，然后在等待函数中再判断线程的状态。

5.4.3 门等待

通过上述两小节的介绍，我们理解了 Windows 的等待机制以及各种分发器对象。显然，这一整套机制足够灵活，可以满足内核和应用程序的同步、异步程序设计的需要。除此以外，Windows 还提供了一套更加轻量的等待机制。在 3.5.2 节介绍线程状态转移时，我们看到，一个线程可以从运行状态进入门等待状态，然后从门等待状态回到延迟的就绪状态。这一状态转移过程正是本节将要介绍的门等待机制。

可以说，门等待是对 Windows 标准等待机制的一个简化，它避免了一个线程从进入等待函数到离开等待函数过程中的许多步骤。唤醒一个门等待的线程几乎是以最快捷的方式来进行的，它直接调用 KiDeferredReadyThread 函数将该线程分发到某个处理器上准备执行。

门等待中的分发器对象是 KGATE，它只有一个 DISPATCHER_HEADER 结构成员。它支持多个线程同时等待一个 KGATE 对象，每次有信号状态时只唤醒一个线程，所以，它不是通知类型的分发器对象，而是同步类型的分发器对象。另外，等待门对象的线程不会等待多个分发器对象，也没有超时设置，所以，线程进入门等待时的处理也相对简单许多。

下面我们来看门等待涉及的一些操作，其代码位于 base\ntos\ke\gateobj.c 文件中：

- KeInitializeGate，初始化一个门对象，无信号状态，并且等待链表为空。
- KeWaitForGate，一个线程调用该函数进入门等待状态。该函数运行在一个 do 循环中，它首先判断是否有 APC 需要处理，若是，则处理 APC；然后，判断要等待的门对象是否有信号，若有信号，则这一等待可立即满足；否则，构造一个等待块，插入到门对象的等待链表中，线程对象并不使用等待块链表 WaitBlockList 成员，而是直接让 KTHREAD 的 GateObject 成员指向门对象。如果该线程属于一个队列对象的线程池，则调用 KiActivateWaiterQueue 以唤醒线程池中可能的等待者。最后，调用 KiSwapThread，交出控制权。
- KeSignalGateBoostPriority，该函数设置门对象为有信号状态，并试图唤醒第一个正在等待该对象的线程。函数的主体是一个 do 循环。如果门对象原来是有信号状态，则该函数什么也不做，退出。如果门对象当前并没有等待者，则设置对象为有信号状态，然后退出。否则，从门对象的等待块链表中取出第一个等待线程，设置该线程的状态为延迟的就绪状态（DeferredReady），若等待线程属于一个队列对象的线程池，则增加队列对象的活动线程计数。最后，调用 KiDeferredReadyThread 函数，将它分发到一个空闲处理器上立即运行，或者抢占其他处理器上正在运行的低优先级线程，或者插入到该线程优先级所对应的就绪线程链表中。

因此，处于门等待状态的线程一旦它所等待的门对象变成有信号状态，则可以以最快的速度被调度和运行。这使得门等待比普通的等待要高效一些，在内核中用于一些性能敏感的同步任务。例如，后面 5.4.5 节介绍的推锁就使用了门对象和门等待来解决锁竞争的问题。Windows 内核中的守护互斥体（guarded mutex）使用门等待来实现其互斥同步能力，参见守护互斥体数据结构 KGUARDED_MUTEX 的定义，以及 KeInitializeGuardedMutex、KeAcquireGuardedMutex 和 KeReleaseGuardedMutex 等操作。内存管理器使用守护互斥体来保护一些全局操作，例如换页内存池的互斥保护。

执行体层上封装了一个称为快速互斥体（fast mutex）的同步对象，在 Intel x86 平台上，它建立在内核中的事件对象基础上；在非 Intel x86 平台上，它建立在门对象和门等待的基础上。参见快速互斥体数据结构 FAST_MUTEX 的定义，以及 KiAcquireFastMutex、XxAcquireFastMutex 和 XxReleaseFastMutex 等操作，对应的执行体函数是 ExAcquireFastMutex 和 ExReleaseFastMutex。快速互斥体提供了更好的性能，因为，当不存在竞争时，它不必等待事件对象或门对象。快速互斥体和守护互斥体都已经被文档化，设备驱动程序可以方便地使用这两种优化的互斥体对象。

5.4.4 执行体资源（executive resource）

Windows 执行体实现的快速互斥体提供了互斥访问的能力，由于快速互斥体建立在事件对象或门对象的基础上，所以，它的实现相对比较简捷。除此以外，执行体还提供了另外两种同步机制：执行体资源（executive resource）和推锁（push lock），它们允许内核以共享或互斥方式保护一个资源。这一节我们将介绍执行体资源，下一小节介绍推锁。

执行体资源建立在内核的事件对象和信号量对象的基础之上，它利用事件对象来实现互斥控制，利用信号量对象来实现共享访问控制。执行体资源的数据结构为 ERESOURCE，其定义如下：

```
typedef struct _ERESOURCE {
    LIST_ENTRY SystemResourcesList;
    POWNER_ENTRY OwnerTable;
    SHORT ActiveCount;
    USHORT Flag;
    PKSEMAPHORE SharedWaiters;
    PKEVENT ExclusiveWaiters;
    OWNER_ENTRY OwnerThreads[2];
    ULONG ContentionCount;
    USHORT NumberOfSharedWaiters;
    USHORT NumberOfExclusiveWaiters;
    union {
        PVOID Address;
        ULONG_PTR CreatorBackTraceIndex;
```

```
    };

    KSPIN_LOCK SpinLock;
} ERESOURCE, *PERESOURCE;
```

所有的执行体资源对象都加入到一个全局链表中，即全局变量 ExpSystemResourcesList，ERESOURCE 中的 SystemResourcesList 域代表一个执行体资源对象在该链表中的节点；OwnerTable 域指向一个动态分配的数组，其中保存了当前以共享方式获取或等待该执行体资源对象的线程及其相关信息，请参考 OWNER_ENTRY 的定义。ActiveCount 域记录了当前有多少个线程获取了该执行体资源对象；Flag 域指明了执行体资源对象的当前状态；SharedWaiters 和 ExclusiveWaiters 域分别包含了内嵌的信号量和事件对象，其中信号量对象的计数最大限制为无限（实际为 0x7fffffff）；OwnerThreads 数组域中的第 0 项表示以独占方式拥有该执行体资源的线程的 OWNER_ENTRY 项，第 1 项表示以共享方式拥有该对象的线程的 OWNER_ENTRY 项；ContentionCount 域记录了发生竞争的次数；NumberOfSharedWaiters 和 NumberOfExclusiveWaiters 域分别记录了当前有多少个线程在等待以共享或独占方式获取该对象。由 Address 和 CreatorBackTraceIndex 构成的 C 联合域未用于同步管理。最后，SpinLock 域用于保护执行体资源本身的完整性，对执行体资源对象内部状态的操作均在 SpinLock 自旋锁的保护下进行。

执行体资源对象位于非换页内存池，它本质上实现了一个读写锁。当内核代码读访问一个资源时，可以申请以共享方式访问该资源；当需要写访问该资源时，则必须申请独占方式的访问。当前处于独占方式访问的资源，再接到其他线程的任何访问请求，都必须等当前获取该资源的线程结束访问以后，才能满足其他线程的共享或独占方式的访问请求。当前处于共享方式访问的执行体资源，如果先接到了独占访问请求，又接到了共享访问请求，则后来的共享访问请求也必须等待，而且，独占访问请求将优先于后来的共享访问请求而被满足。归纳起来，当发生竞争时，共享请求和独占请求都要等待，而当执行体资源对象被释放时，选择下一步该满足哪个或哪些请求的算法如下：

1. 若独占方式的线程释放一个执行体资源对象，则首先满足共享请求，如果没有共享请求，则满足独占请求；
2. 若共享方式的线程释放一个执行体资源对象，则只考虑是否有独占请求，如果有，则满足之。如果没有独占请求在等待，那么一定不会有共享请求在等待。

执行体资源的基本操作包括 ExInitializeResourceLite、ExAcquireResourceExclusiveLite、ExAcquireResourceSharedLite、ExReleaseResouceLite 和 ExDeleteResouceLite，其代码位于 base\ntos\ex\resource.c 文件中。这些操作的基本工作流程如下：

- ExInitializeResourceLite，初始化指定的执行体资源对象的状态，将它加入到系统全局的执行体资源链表中。
- ExAcquireResourceExclusiveLite，以独占方式获取一个执行体资源对象。如果该对象当前尚未被获取，即 ActiveCount 为 0，则设置对象已被独占获取，并且在 OwnerThreads 成员的第 0 项记录下当前线程。如果 ActiveCount 不为 0，则除非该对象已经被当前线程以独占方式获取，或者函数参数 Wait 指示了无须等待而立即返回，否则，ExAcquireResourceExclusiveLite 必须等待，直至获取执行体资源对象成功为止。等待过程是在 ExpWaitForResource 函数中完成的，它调用 KeWaitForSingleObject 来等待执行体资源中的事件对象，若等待一段时间后超时，则试图提升以独占方式拥有该对象的线程的优先级，或者提升所有以共享方式获取该执行体资源的线程的优先级，然后继续等待。这样做的目的是，让使用该执行体资源对象的线程尽快完成它们的任务，从而释放该对象。
- ExAcquireResourceSharedLite，以共享方式获取一个执行体资源对象。如果 ActiveCount 成员为 0，说明当前没有线程获取了该对象，则在它的 OwnerThreads 成员的第 1 项中记录下当前线程，并递增 ActiveCount 成员，然后返回。如果该对象已被当前线程以独占方式获取，则直接在 OwnerThreads 成员的第 0 项中增加计数后即可返回。如果该对象已被共享方式获取，则首先看是否当前线程已经出现在 OwnerTable 数组中，若是则说明当前线程已经共享获取了该对象，增加计数后返回；否则，若当前无其他线程在等待独占方式获取，则加入 OwnerTable 数组即返回。

  经过以上步骤以后，该执行体资源要么已被其他线程独占获取，要么已被共享获取但当前线程尚未共享该对象，而且正有线程在等待独占方式获取，这两者都是发生竞争的情形。若参数指示不必等待，则返回失败。否则，首先将当前线程填充到 OwnerTable 数组的空闲项中，然后调用 ExpWaitForResource 函数，在执行体资源内嵌的信号量对象上等待。同样地，在 ExpWaitForResource 函数中，它可能会提升当前已获取该执行体资源对象的线程的优先级，以便尽可能地加快它们的进度，从而释放执行体资源对象。
- ExReleaseResouceLite，释放当前线程所获取的执行体资源对象。对应于前述的算法，分两种情形进行处理。第一，如果当前线程是以独占方式获取该资源的，则递减执行体资源对象 OwnerThreads 数组成员的第 0 项的计数器，若计数值仍大于 0，则直接返回。否则，说明当前线程不再拥有该对象，于是，清除 OwnerThreads 成员中的第 0 项，并递减执行体资源对象的 ActiveCount 成员。如果当前有线程在等待以共享方式获取该执行体资源，则调用 KeReleaseSemaphore 函数，以满足所有正在等待的线

程，然后返回。若没有共享方式等待的线程，则判断是否有独占方式等待，如果有，则调用事件对象的 KeSetEventBoostPriority 操作，满足其中一个等待者。

第二，如果 ExReleaseResouceLite 发现当前线程是以共享方式获取该资源的，那么，首先从执行体资源对象的 OwnerThreads 或 OwnerTable 中找到属于当前线程的共享项，然后递减该共享项的计数器，若计数值仍大于 0，则直接返回。否则，递减执行体资源对象的 ActiveCount 成员，若等于 0，则判断是否有独占方式等待，如果有，则调用事件对象的 KeSetEventBoostPriority 操作，满足其中一个等待者。

- ExDeleteResouceLite，删除一个执行体资源对象。在调用此函数以前，必须确保没有线程在等待获取该资源对象。它首先从系统的执行体资源链表中移除对象，然后释放该对象内部占用的系统资源，包括记录共享信息的 OwnerTable 数组成员，以及内嵌的事件对象和信号量对象。

执行体资源封装了事件和信号量对象来实现一个读写锁的语义，它通过提升当前已获取资源的线程的优先级来促使它们尽快完成其功能逻辑。执行体资源被广泛应用于 Windows 内核中。内核中的各个子系统，特别是 I/O 子系统和文件系统，使用执行体资源来保护一些全局的数据结构。此外，执行体资源也被暴露给了设备驱动程序使用，以上介绍的这些函数在 Windows DDK 文档中都有相应的用法说明。

5.4.5 推锁（push lock）

现在我们讨论执行体中另一种支持互斥和共享两种模式的同步机制——推锁。推锁是一个很小的数据结构，在 32 位机器上为一个 32 位整数，在 64 位机器上为一个 64 位整数，并且可以从换页内存区分配。下面是推锁的定义：

```
typedef struct _EX_PUSH_LOCK {
   union {
      struct {
         ULONG_PTR Locked          : 1;
         ULONG_PTR Waiting         : 1;
         ULONG_PTR Waking          : 1;
         ULONG_PTR MultipleShared  : 1;
         ULONG_PTR Shared          : sizeof (ULONG_PTR) * 8 - 4;
      };
      ULONG_PTR Value;
      PVOID Ptr;
   };
} EX_PUSH_LOCK, *PEX_PUSH_LOCK;

// 当发生竞争时，EX_PUSH_LOCK（高 28 位）指向等待块，等待块的类型为 EX_PUSH_LOCK_WAIT_BLOCK
typedef struct DECLSPEC_ALIGN(16) _EX_PUSH_LOCK_WAIT_BLOCK {
```

```
    union {
        KGATE WakeGate;
        KEVENT WakeEvent;
    };
    PEX_PUSH_LOCK_WAIT_BLOCK Next;
    PEX_PUSH_LOCK_WAIT_BLOCK Last;
    PEX_PUSH_LOCK_WAIT_BLOCK Previous;
    LONG ShareCount;
    LONG Flags;
} DECLSPEC_ALIGN(16) EX_PUSH_LOCK_WAIT_BLOCK;
```

我们可以看到，推锁的最低 4 位分别为 L、W、K 和 M 位，如图 5.15 所示。L 位表示推锁已被某个线程锁住：对于独占锁，从一个线程成功获取，到它释放该锁，这期间 L 位被置位；对于共享锁，从第一个线程获取该推锁，导致共享计数变为 1，一直到某个释放操作导致共享计数变为 0，此期间 L 位一直被置位。W 位表示有线程正在等待该推锁，如果 W 位为 0，则高 28 位代表共享计数；否则，代表一个指向等待块的指针。K 位的含义是，一个线程正在唤醒另一个线程。设置了 K 位的线程在执行唤醒操作时，如果它发现该推锁仍然是锁住的，则会清除 K 位并返回；否则，它会遍历推锁的等待块链表，唤醒链表末尾的等待者，即最早进入等待的线程。M 位表示在等待块链表的末尾节点记录了共享推锁的线程数，即有竞争情况下共享推锁的所有者数量。

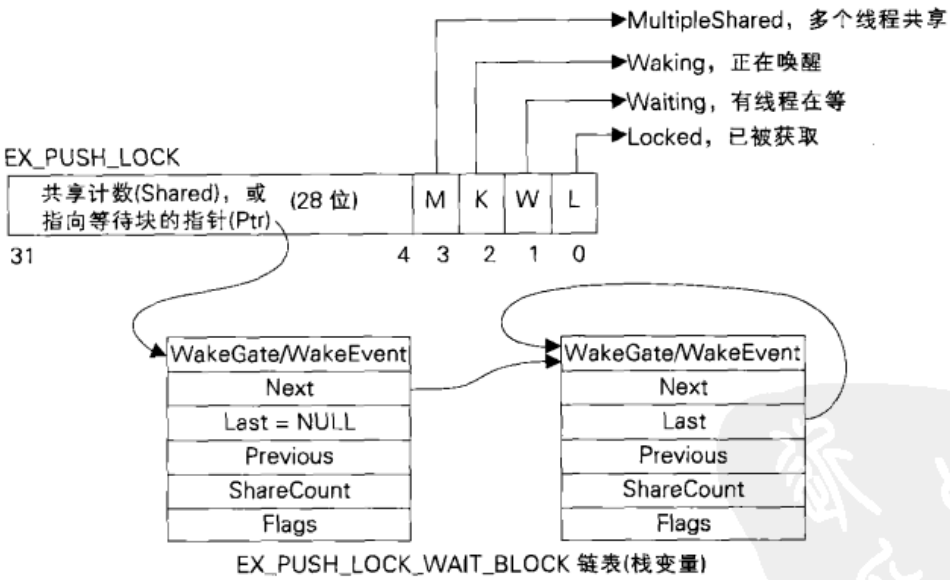


图 5.15 推锁的数据结构示意图

接下来介绍推锁的算法，为描述简便起见，在 4 个标志位中我们仅考虑 L 位和 W 位。一个线程以独占或共享方式获取一个推锁，然后释放独占获取或共享获取的推锁，这样共有 4 个操作，下面是无竞争情况下这 4 个操作的算法：

1. 独占获取。推锁是可用的，L=0，W=0，Shared=0，获取以后，推锁状态为 L=1，W=0，Shared=0。
2. 共享获取。推锁当前是可用的，或已被共享获取但无等待线程，即 L=0，W=0，Shared=0，或者 L=1，W=0，Shared>0，获取以后，推锁状态为 L=1，W=0，Shared=Shared+1。
3. 释放独占推锁。已被独占获取，并且无等待线程，所以，L=1，W=0，Shared=0，释放以后，推锁状态为 L=0，W=0，Shared=0。
4. 释放共享推锁。已被共享获取，并且无等待线程，所以，L=1，W=0，Shared>0，释放以后，推锁状态为 W=0，Shared=Shared-1，若 Shared 为 0 则 L=0，否则 L=1。

由此可看出，在无竞争情况下，W 位总是 0。如果发生竞争，则必定有线程要等待，此时 W=1，并且推锁的高 28 位指向一个等待块链表，下面是有竞争情况下 4 个操作的算法：

1. 独占获取。推锁已经被获取，如果 W=0，则表明这是第一个进入等待的线程，于是构造一个等待块，将推锁的 Shared 值保存到等待块的 ShareCount 中，然后置 W=1，推锁指向该等待块；如果 W=1，则表明已经有线程在等待了，于是构造一个等待块，并插入到等待块链表首。最后线程在等待块中的门对象上等待，直至等待返回。
2. 共享获取。有两种可能：推锁已经被独占获取，但当前无等待线程（即 W=0），则构造一个等待块，等待块中的 ShareCount 为 0，然后置 W=1，推锁指向该等待块；已经有等待线程了，即 W=1，则构造一个等待块，并插入到等待块链表首。最后线程在等待块中的门对象上等待，直至等待返回。
3. 释放独占推锁。定位到等待块链表末尾节点，若最后一个节点对应于独占获取请求，则唤醒该节点对应的线程；否则，唤醒末尾连续的以共享方式获取该推锁的线程。
4. 释放共享推锁。定位到等待块链表末尾节点，递减其 ShareCount，若已减为 0，则唤醒此等待块代表的等待线程，它必定是一个独占方式的获取操作；若仍大于 0，则直接返回。

Windows 执行体在实现推锁算法时使用了另外两位 K 和 M 位，用于同步或协调多个线程访问推锁内部的数据成员，这两位的含义如本节前面所描述。另外，推锁本身可以位于换页内存区域，而与推锁关联的内部动态数据结构，即等待块，是栈变量。当一个线程要获取推锁时，如果推锁当前的状态不能满足此请求，那么，该线程所调用的获取函数会将一个等待块栈变量作为节点加入到等待块链表中，并且初始化等待块中的门对象，然后调用 KeWaitForGate 函数进入等待。当门等待结束时，该等待块已经脱离链表，所以，这里使用栈变量是安全的。门等待的唤醒动作是由其他线程在释放推锁时触发的。

下面介绍推锁的基本操作，包括 ExpInitializePushLocks、ExInitializePushLock、ExfAcquirePushLockExclusive、ExfAcquirePushLockShared、ExfReleasePushLockShared 和 ExfReleasePushLockExclusive，其代码位于 base\ntos\ex\pushlock.c 或 base\ntos\inc\ex.h 文件中。这些操作的基本工作流程如下：

- ExpInitializePushLocks，这是一个针对所有推锁的初始化函数，在执行体初始化时被调用，它仅仅设置一个全局的推锁自旋计数值（ExPushLockSpinCount）。
- ExInitializePushLock，这是针对单个推锁的初始化函数，它仅仅将推锁的 Value 赋为 0，即所有的域都初始化为 0。
- ExfAcquirePushLockExclusive，以独占方式获取一个推锁。函数的主体是一个 while(1) 循环。如果推锁的 L 位为 0，则利用 InterlockedCompareExchangePointer 函数，以原子方式将 1 写入 L 位，若成功，则获取到锁，返回。否则继续循环。

  如果 L 位为 1，则表明发生了竞争，于是初始化一个等待块，即栈变量 WaitBlock：如果 W 位为 1，说明已经有其他线程在等待了，于是将 WaitBlock 插入在第一个等待块的前面，即用推锁中的指针作为 WaitBlock 的 Next；否则表明这是第一个进入等待的线程，于是设置相应的状态标志。然后调用 InterlockedCompareExchangePointer 函数将新的 WaitBlock 插入到推锁中，若成功，则初始化 WaitBlock 中的门对象，并自旋一段很短的时间（由 ExPushLockSpinCount 控制），若经过自旋以后仍未获得推锁，则调用 KeWaitForGate 进入等待。如果调用 InterlockedCompareExchangePointer 函数将新的 WaitBlock 插入到推锁中不成功，则进入下一轮循环，继续尝试。注意，KeWaitForGate 返回以后，并不是立即返回成功，而是回到循环开始处继续，但此时 L 位已被清除。
- ExfAcquirePushLockShared，以共享方式获取一个推锁。如同 ExfAcquirePushLockExclusive 一样，函数的主体也是一个 while(1)循环。如果推锁的 L 位为 0，或者 L 位虽然为 1，但 W 位为 0，且共享计数大于 0，则满足无竞争条件，所以，利用 InterlockedCompare-ExchangePointer 函数，写入 L 位，或者增加共享计数。若写入不成功，则继续循环。

  如果不满足无竞争条件，则初始化栈变量 WaitBlock：如果 W 位为 1，说明已经有其他线程在等待了，于是将 WaitBlock 插入在第一个等待块的前面，即用推锁中的指针作为 WaitBlock 的 Next；否则表明这是第一个进入等待的线程，于是设置相应的状态标志。然后调用 InterlockedCompareExchangePointer 函数将新的 WaitBlock 插入到推锁中，若成功，则初始化 WaitBlock 中的门对象，并自旋一段很短的时间（由 ExPushLockSpinCount 控制），若经过自旋以后仍未获得推锁，则调用 KeWaitForGate

进入等待。如果调用 InterlockedCompareExchangePointer 函数将新的 WaitBlock 插入到推锁中不成功，则进入下一轮循环，继续尝试。

- ExfReleasePushLockShared，释放以共享方式获取到的推锁。如果推锁的 W 位为 0，则表明无线程在等待推锁，于是通过 InterlockedCompareExchangePointer 函数确保推锁的值被修改以后写回到推锁中。

  如果推锁的 W 位被置上，而且 M 位也被置上，则找到等待块链表的最后一项，递减其共享计数，若递减以后该共享计数仍大于 0，则返回。

  接下来是一个 while(1)循环。在循环中，如果推锁的 K 位为 1，则调用 InterlockedCompareExchangePointer 函数清除推锁中的 L 位和 M 位即可；否则，清除 L 位和 M 位，并且置上 K 位，然后调用 ExfWakePushLock 函数，以唤醒等待块链表上末尾节点（如果它要独占获取）或所有节点（否则）的相应线程。ExfWakePushLock 函数也会清除推锁的 K 位。

- ExfReleasePushLockExclusive，释放以独占方式获取到的推锁。该函数也是在一个 while(1)循环中进行。如果推锁的 W 位为 1，但 K 位为 0，则表明有线程在等待推锁，而且没有其他线程在唤醒等待者，于是通过 InterlockedCompareExchangePointer 函数去掉推锁中的 L 位，置上 K 位。然后调用 ExfWakePushLock 函数唤醒等待块链表上末尾节点或所有节点的相应线程。ExfWakePushLock 函数也会清除推锁的 K 位。

  如果推锁的 W 位为 0，或者虽然 W 位为 1，但 K 位也为 1，则可以通过 InterlockedCompareExchangePointer 函数去掉推锁中的 L 位，然后返回即可。

推锁是 Windows 内部使用的读写锁，它利用 InterlockedCompareExchangePointer 函数来更新自己的状态，从而避免多个处理器同时操作推锁时产生状态不一致的情形。另外，它利用门对象和门等待来解决锁竞争的问题。相比较执行体资源使用自旋锁来保护对象的状态、使用事件和信号量对象来实现互斥和共享语义，推锁有更好的性能和伸缩能力。

Windows 内核中有多个组件使用推锁来保护一些数据结构，比如我们在第 3 章中看到的执行体进程对象和线程对象中的进程锁（EPROCESS 的 ProcessLock 成员）和线程锁（ETHREAD 的 ThreadLock 成员），以及进程的工作集锁等。还有，对象管理器也使用推锁来保护对象管理器数据结构。

除了基本的推锁以外，Windows 还提供了一种称为**可感知缓存的推锁**（cache aware push lock，或者称为**缓存友好的推锁**，cache friendly push lock）的同步对象。这种推锁

的基本思想是，为系统中的每个处理器分配一个普通的推锁，当一个线程以共享方式获取一个可感知缓存的推锁时，它只是简单地以共享方式获得对应于当前处理器的那个推锁；如果它欲以独占方式获取这一可感知缓存的推锁，则以独占方式获取所有处理器的推锁。之所以称为可感知缓存的或缓存友好的，是因为，针对每个处理器分配的子推锁，对应于它的一条缓存线（cache line），因此，如果以共享方式获取该锁，则每个处理器只需获得自己缓存线上的子推锁即可。有关可感知缓存的推锁的操作，参见 ExAllocateCacheAwarePushLock、ExAcquireCacheAwarePushLockExclusive 、 ExReleaseCacheAwarePushLockExclusive 、ExAcquireCacheAwarePushLockShared 、 ExReleaseCacheAwarePushLockShared 和 ExFreeCacheAwarePushLock 函数的代码。

5.4.6 死锁

死锁是并发系统中的基本问题，很难有有效的方法可以根绝它的发生。Windows 并没有提供统一的或者系统范围的检测或打破死锁的解决方案，但是，Windows 内核自身在设计和实现时已经小心地考虑过，以避免发生死锁（以及饥饿）的可能。下面举三个典型的例子来说明这样的考虑是必要的。

第一个例子与线程切换有关。在 KiSwapThread 函数中，如果新选择的线程的 SwapBusy 标志为 TRUE，则说明另外一个处理器正在从该线程切换到另一个线程。这里存在死锁的可能：假设处理器 A 从线程 1 切换出去，在 KiSwapThread 函数中，线程 1 的 SwapBusy 被置为 TRUE，现在新选择线程 2；与此同时，如果处理器 B 从线程 2 切换出去，它设置了线程 2 的 SwapBusy 域为 TRUE，KiSwapThread 函数中新选择的线程为线程 1。在这种情况下，两个处理器就形成了死锁，因为 KiSwapThread 函数要调用 SwapContext，而 SwapContext 在实施切换以前要等待新线程的 SwapBusy 为 FALSE，所以，两个处理器都在等待对方将新选出来的线程的 SwapBusy 域清除。Windows 的解决方案是，在 KiSwapThread 函数中，如果新选择的线程的 SwapBusy 域为 TRUE，则 KiSwapThread 设置新的线程为该处理器的 IdleThread，而将原来选出来的线程置为该处理器的备用（standby）线程。由于 IdleThread 是各个处理器私有的，所以，切换到 IdleThread 线程不存在任何碰撞。有兴趣的读者，可参考 base\ntos\ke\thredsup.c 文件的 1 960~1 969 行代码。

第二个例子是内存管理器中的修改页面写出器，正如 4.5.4 节所介绍，之所以使用两个线程分别完成修改页面写出任务和映射页面写出任务，正是为了防止用单个线程来完成这两项任务可能会出现死锁的情形。关于死锁情形的描述，请参考 4.5.4 节。

第三个例子是，执行体资源要求运行在 PASSIVE_LEVEL 的线程在获取执行体资源以

前必须禁止内核 APC。如果不这样做的话，用户模式的代码可能会调用 NtSuspendThread 函数，将线程挂起，而 NtSuspendThread 函数是利用内核 APC 来实现的。因此，若一个线程在获取执行体资源以前不禁止内核 APC，则它有可能在获取到执行体资源后被挂起，在极端情况下可能使整个系统死锁。Windows DDK 文档中关于执行体资源的用法明确地指出了这一要求。

Microsoft 提供了一个驱动程序检验器（Driver Verifier），该工具有一个检测死锁的选项，可以监视驱动程序使用同步资源的情况，并检测出有潜在发生死锁可能的代码逻辑。它监视的资源包括自旋锁、互斥体和快速互斥体。检测算法并无奇特之处，无非是在线程和资源的等待占有关系图中寻找出现环的可能性，由于驱动程序检验器以驱动程序为单位来检测死锁，所以，图的规模相对较小，从而有可能进行实时检测。关于驱动程序检验器的死锁检测功能的进一步信息，请参考 MSDN 或 DDK 中的相关文档[MSDN-DEADLOCK]。

5.5 观察线程同步关系——DPerfLite

DPerfLite 是一个观察线程之间同步关系的工具。它记录了系统中与同步有关的操作，并允许用户在窗口中直观地观察这些操作。DPerfLite 除了显示每一个操作的属性以外，还可以揭示出同步操作之间的关联关系，比如一个线程解除了另一个线程的等待。

5.5.1 DPerfLite 使用示例

图 5.16 显示了 DPerfLite 的一个典型使用场景。该屏幕截图表明，OUTLOOK 进程中的一个线程 0xadc 在等待，而 csrss 进程中的线程 0x2b0 唤醒了线程 0xadc。从线程 0xadc 的栈可以看出，这是一个 KeWaitForMultipleObjects 调用，IRQL 为 0，它只等待一个事件对象，该事件对象的地址为 0xa5200ce0。

DPerfLite 也可以跟踪和显示异步过程调用（APC）的插入点和交付点。图 5.17 显示了一个例子。System 进程的 0x888 线程向 svchost 进程的 0x8ac 线程插入了一个 APC 例程，此时 0x8ac 线程正在等待，但是它仍然可以交付 APC 例程。而且，我们可以看到，在 APC 例程执行过程中，有一个释放操作。通过 DPerfLite 可以进一步看到，此释放操作将 0x8ac 线程自身的等待解除了。在 Windows 内核中，这是一种常见的模式，当系统线程完成一个操作时，并不是直接唤醒正在等待的线程，而是会在目标线程中插入一个 APC 例程，由 APC 例程解除其等待。这么做的好处是，在 APC 例程中，它可以直接访问目标线程所在进程的地址空间。譬如，在 I/O 完成通知例程中，它可以将数据从系统地址空间传输到进程地址空间中。

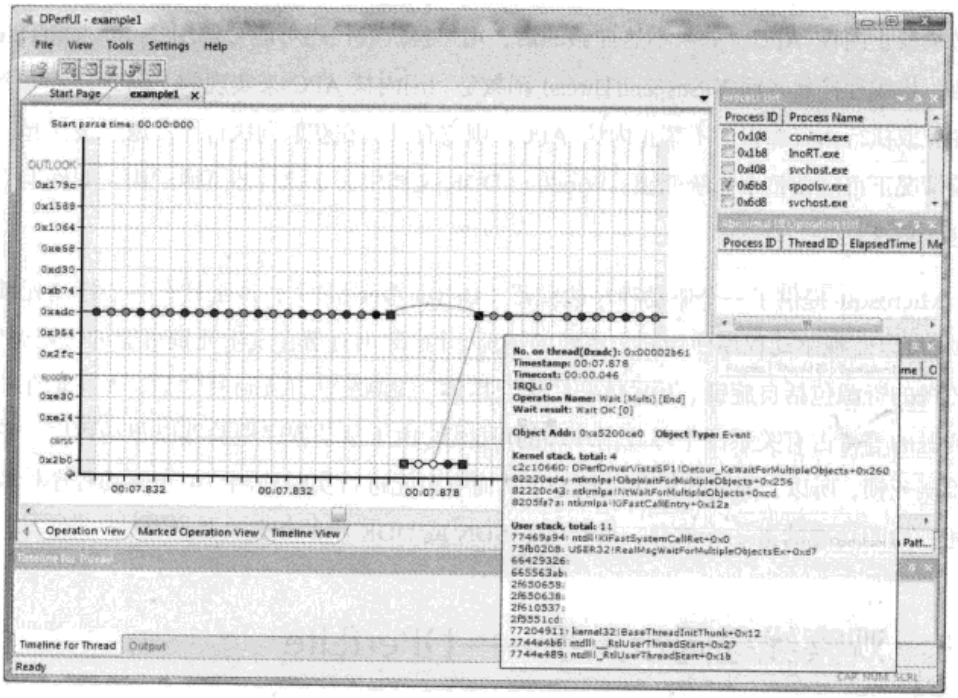


图 5.16 DPerfLite 的一个典型使用场景

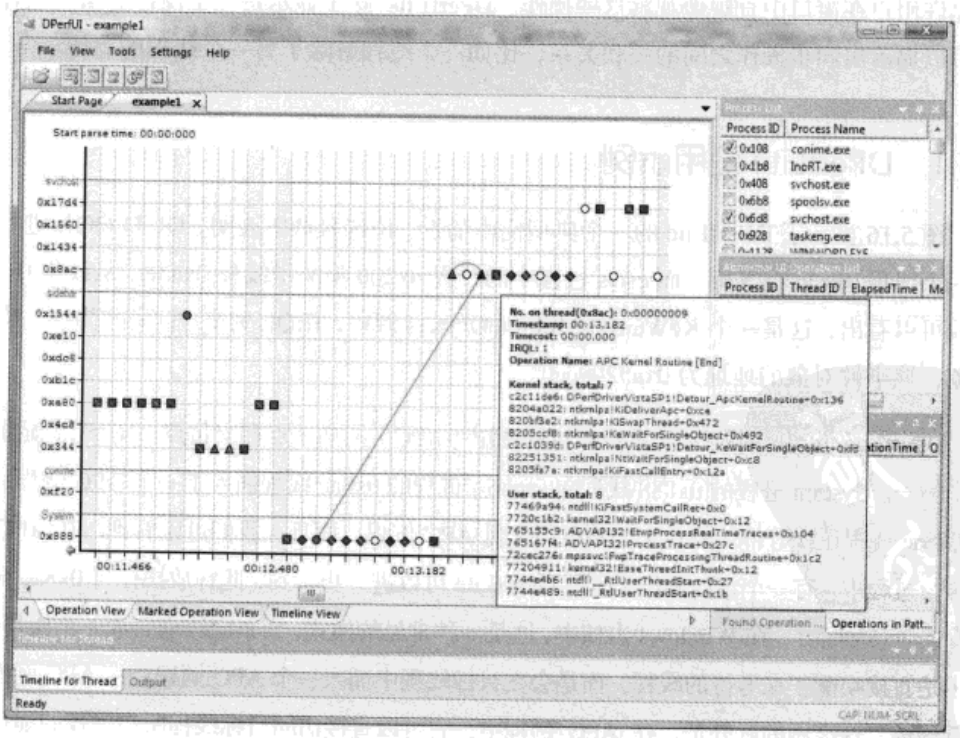


图 5.17 DPerfLite 显示 APC 的插入和交付关系

DPerfLite 显示线程之间的同步关系并不是实时进行的，它需要先录制一个痕迹文件（trace file）。使用 DPerfLite 的录制命令可以开始记录痕迹数据，当用户按下停止按钮时，DPerfLite 会生成一个痕迹文件，其中包含了这一段时间内系统中发生的各种同步操作。有了痕迹文件以后，就可以打开该文件来观察线程之间的同步关系了。

5.5.2 DPerfLite 实现原理

DPerfLite 是笔者所在研究组的一个研究项目（名为 DPerf）的简化原型工具。DPerf 项目的目标是利用线程之间的时序依赖性来诊断系统和应用程序的性能问题。

DPerfLite 截取了系统内核中的各种同步操作，包括 5.4.2 节中介绍的各种分发器对象的有关操作。而且，DPerfLite 也记录了 APC 和工作项目（WorkItem）的插入和交付操作，所以，发生在异步过程中的事件可以被进一步追溯到插入该异步过程的线程上。DPerfLite 允许用户指定在录制系统事件时是否将事件发生点的调用栈信息包括进来。在同等事件数量的情况下，包含调用栈信息的痕迹文件会大很多，但是，调用栈信息被解析成有意义的符号名以后，可以提供更为有用的线索，让使用者知道一个事件是如何发生的。

系统事件的采集是通过一个驱动程序来完成的，这些事件首先被记录在一个内核缓冲区中。考虑到存储空间的容量问题，DPerfLite 使用一个循环缓冲区来记录系统事件。如果事件数量太多而导致缓冲区满了的话，DPerfLite 的事件记录器将不再记录新的事件，而直接将它们丢弃。DPerfLite 的用户模式程序每隔一定时间（默认设置为 500 ms），将内核缓冲区中的事件复制到用户模式的一个缓冲区中。由于有些被记录的事件可能发生在 IRQL 大于等于 DISPATCH_LEVEL 的代码中，所以，内核循环缓冲区是从非换页内存池中申请的，默认大小为每个处理器 5 MB。而用户模式程序中的缓冲区可以是任意大小，取决于系统中磁盘和内存的配置。

DPerfLite 使用 Microsoft 提供的 dbghelp.dll 来解析符号名，它使用 Microsoft 公开的符号服务器来获得系统模块的符号信息，所以，DPerfLite 除了可以在 WRK 这样的自编内核模块的系统中截取系统事件和解析调用栈信息以外，也可以在商业版本的 Windows 操作系统上运行，包括 Windows XP/Vista/7 和 Windows Server 2003/2008。关于如何设置和使用 Microsoft 提供的符号信息，请参考附录 A。

在 DPerfLite 中，同步事件之间的关联关系是根据分发器对象的地址建立起来的，譬如，KeSetEvent 使一个事件对象变成有信号状态，紧接着一个正在等待的线程被唤醒，如果后者正在等待的分发器对象正是 KeSetEvent 操作中的事件对象，那么这两个操作被关联在一起。

其他类型的分发器对象皆同此理。异步例程则根据 APC 对象或工作项目（WorkItem）的地址来关联。由于所有进程共享同样的系统地址空间，所以这种做法对于内核中的对象地址或函数地址是有效的。

5.6 本章总结

在本章中，我们看到了 Windows 中处理并发和同步的核心机制。中断和异常既是硬件提供的并发机制，也是操作系统必须谨慎处理并善加利用的两个特性。Windows 提供了一套通用的中断分发机制，允许设备驱动程序在不操纵处理器的中断描述符表（IDT）的情况下注册中断服务例程（ISR）。Windows 还实现了一套系统范围的异常处理机制，允许用户程序中的异常得以妥善处理，并使内核、应用程序自身、调试器以及 Windows 子系统都有机会处理一个异常。

Windows 实现了多种同步机制，有些机制不依赖于线程调度，比如用于多处理器环境下的互锁操作、自旋锁，以及 Windows 制定的 IRQL 管理机制。低 IRQL 的中断不可抢占高 IRQL 的执行过程。APC 和 DPC 是两种特殊 IRQL 的可执行例程，它们可被用于实现一些特殊的任务。除了普通的自旋锁支持，Windows 还巧妙地实现了一种排队的自旋锁，允许自旋锁按照 FIFO 顺序进行排队。此外，Windows 还在线程调度的基础上，提供了线程间的同步机制，并实现了操作系统领域中常见的同步和互斥原语。本章最后还介绍了一个可用于观察和学习线程间同步关系的工具 DPerfLite。

并发和同步往往是操作系统学习过程中的难点所在，本章力求将 Windows 中实现的同步机制以相对独立的方式展示出来，但实际上，这些机制往往是综合使用的。除了内核自身的代码广泛使用这些机制以外，设备驱动程序更是依赖于这些系统机制以使其代码逻辑能够平滑地融入到整个操作系统中。下一章我们将讨论设备驱动程序所遵守的 I/O 模型。

6

Windows I/O 系统

在现代操作系统中，I/O（Input/Output，输入/输出）是计算机完成各种功能的一个重要方面。处理器负责执行各种计算任务，并且通过内存总线操纵整个内存空间，但作为一个现实可用的计算机系统，仅仅具备这样的计算和内存访问能力还不够，还要有各种外部设备的参与才可使计算机真正“有用”。在一台典型的个人计算机中，常见的外部设备或外部接口包括：键盘、鼠标、硬盘、光盘驱动器、显示器、打印输出、声音输出、网络适配器接口，以及摄像头等。尽管这些设备的功能和用途各式各样，但是处理器利用了标准的接口技术，通过设备的控制器与它们打交道，因此，从硬件层面上，处理器支持的接口技术是计算机系统能够协作运行的基础。

从软件层面上，操作系统必须提供相应的软件来操纵这些设备的控制器，并且定义对应的软件接口或系统服务，使得应用程序可以方便地操纵或使用外部设备。而且，在一个多任务、多进程操作系统中，这些外部设备是共享的，所以，操作系统必须协调对外部设备的访问。典型的做法是，应用程序不直接访问外部设备，而是通过操作系统来完成对设备的访问。Windows 作为一个通用操作系统，提供了一套可扩展的 I/O 处理框架，从而允许第三方硬件厂商编写专门的系统模块来控制其硬件设备。本章将这一 I/O 处理框架称为 Windows I/O 模型，它是对计算机外部设备工作原理的一个抽象，从而使得各种硬件设备都能够纳入到这一模型中。

在 Windows I/O 模型中，用于控制外部设备的软件模块称为设备驱动程序（Device Driver），它们运行在特权模式下，即内核模式下，所以它们的代码与内核代码具有同等的执行权限。因此，设备驱动程序必须遵循内核代码所必须遵守的一切规则，比如 IRQL

要求、内存管理规范等。另外，此 I/O 模型也是 Windows 操作系统的软件扩展模型，这意味着，一个设备驱动程序可以与硬件设备毫无关联，而仅仅是扩展内核的功能，或者由于其他目的而需要在内核中执行代码。Windows 的许多系统组件也正是以设备驱动程序的形式而提供的。因此，设备驱动程序并非真的要“驱动”一个硬件设备。

在这一章中，我们首先介绍现代计算机系统中 I/O 的工作方式，并总体介绍 Windows 内核中的 I/O 系统结构。然后分别讨论 Windows I/O 系统的三大部件：I/O 管理器、即插即用管理器与电源管理器。接下来讨论 Windows 的 I/O 模型，介绍设备驱动程序的基本结构以及 I/O 处理过程。最后，介绍一个可监视系统中 I/O 请求完成情况的工具 IRPMon。

6.1 I/O 概述

对于计算机的处理器来说，I/O 硬件设备其实只是一些符合某些接口规范的控制器而已，处理器并不直接与设备打交道，而是向设备控制器发号施令，或者接收设备控制器的通知或命令。这一节我们首先讨论现代计算机系统的 I/O 技术，以及操作系统中涉及 I/O 的软件技术，然后介绍 Windows 的 I/O 系统结构。

6.1.1 现代计算机系统的 I/O

按照传统的分类方法，I/O 设备可以分为块设备和字符设备。像磁盘、光盘是典型的块设备，寻址和读写操作都是以块为单位进行的。另一类是字符设备，它们接收或发送的数据是字节流，而不是数据块。典型的字符设备有键盘和鼠标。这种分类方法以设备数据的组织单元为基础，它显然适用于单纯以传输数据作为交互方式的设备。然而，有些外部设备，比如时钟发生器、电源控制器等，显然既不能归为字符设备，更不会是块设备。

尽管这种分类方法并不能恰当地涵盖所有可能的设备，但是，它至少将块设备作为一种类别提了出来。现代操作系统往往针对块设备使用专门的软件结构，比如设计文件系统时只须考虑抽象的块设备（诸如块的大小、总数等可以由特定的配置参数来控制），而实际的块设备读写过程则由更低层的驱动程序来处理。

处理器往往只跟设备的控制器打交道，而不直接操纵设备。处理器与控制器之间通过总线进行通信，而控制器与设备之间的通信则往往通过专门的接口进行，如图 6.1 所示。这些控制器的功能可能非常简单，比如像键盘控制器，它只处理键盘扫描码以及与系统进行简单的通信；有些控制器的功能可能非常复杂，比如现代的硬盘控制器，除了最基本的

控制磁头读写数据的能力以外，还提供了校验数据的功能，以及不小的缓存空间。上一章中介绍的中断控制器也是一种控制器，其特殊之处在于，通过给处理器的一个管脚加上电信号，从而打断处理器当前的运行。许多设备利用中断控制器来通知处理器某些事情的发生，比如设备驱动程序请求的数据已经准备就绪。在上一章中我们已经看到了 Windows 如何利用 Intel x86 处理器的中断机制来实现统一的中断处理。

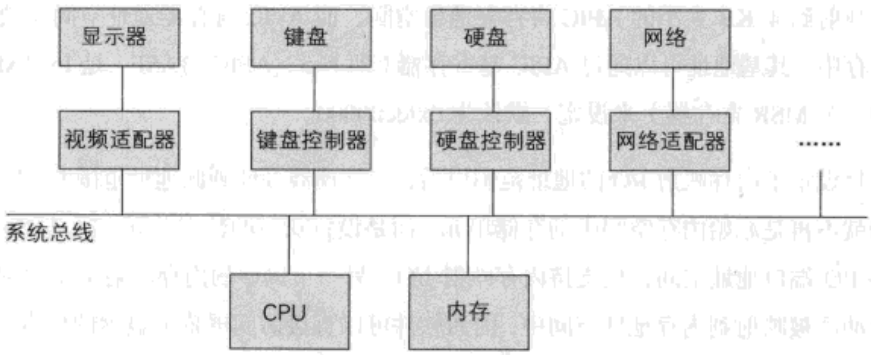


图 6.1 计算机 I/O 硬件模型

设备控制器中往往有一些状态寄存器或者命令寄存器，因而处理器可以通过操纵这些寄存器，从而控制相应的设备，比如告诉控制器发送数据、接收数据、打开或关闭某些特定的功能。所以，操作系统的任务是根据设备的工作模式，读或写这些寄存器，从而达到控制外部设备的目的。

控制器中除了状态寄存器或者命令寄存器以外，可能还包含了可供处理器读写的数据缓冲区，例如块设备需要一个双方都可以访问的缓冲区用于传输数据；视频适配器往往有一段视频内存地址范围，允许处理器操纵显示器的屏幕输出结果。

我们来看一下 Intel x86 处理器提供的设备接口功能。首先，Intel x86 处理器除了内存地址空间以外，另外定义了一个 I/O 端口（port）空间，指令 in 和 out 用于操纵端口空间中的数据单元。例如，在下面的汇编指令中：

```
out 21h, al
```

0x21 是 8259A 中断控制器的中断屏蔽寄存器（IMR，Interrupt Mask Register），该指令将 Intel x86 处理器的 al 寄存器中的值写到中断控制器的控制寄存器中，从而达到屏蔽某些中断信号的目的。类似地，在下面的汇编指令中：

```
in al, 20h
```

0x20 是 8259A 中断控制器的正在服务寄存器（ISR，In Service Register），该指令将此寄

存器中的状态值传递到处理器的 al 寄存器中。在 Intel x86 处理器中，in 和 out 指令的 I/O 端口操作数可以是 8 位或 16 位（使用 dx 寄存器），而 I/O 端口的值可以是 8 位、16 位或 32 位。

设备控制器中的寄存器除了通过 I/O 端口来访问以外，也可以被映射到系统内存空间。这种访问方案称为内存映射 I/O（memory-mapped I/O）。例如，APIC 中断控制器中的寄存器被映射到 4 KB 大小的 APIC 寄存器地址空间，而 APIC 寄存器地址空间又被映射到系统内存中，其基地址可以通过 APIC 基寄存器（即 IA32_APIC_BASE，是 Intel x86 处理器中的一个 MSR 寄存器）来设定，默认为 0xfee00000。

一旦设定了内存映射 I/O 的地址范围以后，当处理器寻址到此地址范围时，它实际上访问的就不再是原始内存空间中的存储单元，而是设备映射的数据单元。Intel x86 处理器既支持 I/O 端口地址空间，也支持内存映射 I/O。另一个典型的内存映射 I/O 的例子是，视频缓冲区被映射到内存地址空间中，因而软件可以直接访问屏幕上显示的像素。在早期的 IBM PC 中，从 640 KB 至 1 MB 之间的地址专门保留给设备的数据缓冲区，其中包括单色和彩色 VGA 的显示缓冲区。

因此，处理器可以通过 in/out 指令或者直接读写内存的方式与各种控制器打交道，那么，设备在执行任务时，如何将任务的状态反馈给处理器呢？一种做法是，设备控制器提供一个状态寄存器，因而处理器可以定期或者用忙等待（busy-wait）的方式检查此寄存器。然而，除非要等待的时间预计不是很长，否则忙等待的做法应该避免。另外一种做法是，设备用中断的方式来通知处理器。当处理器接收到中断信号时，它可以根据中断的设置或者设备寄存器的状态，来作相应的处理。

对于少量的数据交换，或者像 VGA 显示缓冲区这样固定地址范围的数据映射，使用上述做法是合适的。但是，对于像磁盘这样需要大量动态数据交换的设备，如果处理器使用 I/O 寄存器逐个内存单元进行访问，或者通过固定地址范围的数据映射，则不仅存取效率不高，而且处理器操作也极为不便。对于这种情形，现代计算机通常采用一种称为直接内存访问（DMA，Direct Memory Access）的方案。DMA 需要硬件的支持，它也是一种控制器。既可以整个系统共享一个 DMA 控制器，也可以让一个设备，比如磁盘，有单独的 DMA 控制器，这取决于硬件的设计。DMA 控制器通过系统总线来传输数据，但是它在传输数据时不占用处理器的指令周期，所以，使用 DMA 可以将处理器解放出来。当数据传输完成时，DMA 控制器以中断的方式通知处理器。图 6.2 显示了通过 DMA 控制器来读写硬盘数据的硬件结构。

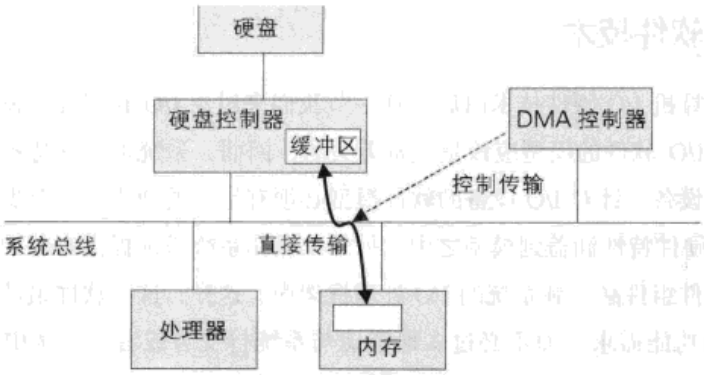


图 6.2 DMA 数据传输示意图

处理器通过 DMA 控制器中的寄存器对数据传输进行设置，指定所读写数据在内存中的地址、数量，以及传输方向。如果不使用 DMA 控制器，那么处理器必须自己逐个字节或逐个字地在硬盘控制器和内存之间传输数据，数据的传输通过总线来完成。而使用了 DMA 控制器以后，DMA 控制器代替处理器来做同样的事情，它逐个字节或逐个字地向硬盘控制器发送命令，同样地通过系统总线来完成数据传输。当数据传输完成以后，它中断处理器，因此，当处理器接收到中断时，就知道数据已经在内存中指定的地方，或者已经写到硬盘控制器中了。

DMA 控制器接到了处理器的读取硬盘数据的命令时，会首先告诉硬盘控制器，将要读取的数据先放到硬盘控制器的缓冲区中。从硬盘到硬盘控制器之间的数据传输是通过专用接口完成的，不需要使用系统总线，此通信完全不受硬件系统其他模块的影响。当数据到达硬盘控制器的缓冲区中，并且通过了校验和验证以后，DMA 传输就可以进行了，接下来的数据传输需要用到系统总线。

当 DMA 控制器通过总线向硬盘控制器发出读请求时，磁盘控制器既不知道也不关心此请求来自处理器还是 DMA 控制器，由于要写的内存地址已经在总线的地址线上了，所以，磁盘控制器从内存缓冲区中取出当前要传输的字，将它送到总线上。这样就完成了一个字的传输。DMA 控制器用这种方法，将处理器指定要读取的所有字逐个地传输到内存中。然后向处理器请求中断。向硬盘写数据的过程类似，只是方向相反，这里不再介绍。

简单小结一下，在现代计算机体系结构中，处理器通过 I/O 端口或者内存映射 I/O 的方式与硬件设备的控制器打交道，而非直接操纵硬件设备。Intel x86 处理器提供了 in/out 指令来访问 I/O 端口空间，并且也支持内存映射 I/O。为了在内存与设备之间传输大量数据，现代计算机往往采用 DMA 结构，以便将处理器从数据传输任务中解脱出来。当 DMA 数据传输完成时，DMA 控制器以中断的方式通知处理器。

6.1.2 I/O 软件技术

了解了计算机 I/O 硬件技术以后，这一节我们来讨论 I/O 软件设计技术。从操作系统角度而言，I/O 软件的模型应该是设备无关的，因而，系统可以支持各种设备，甚至是未来出现的设备。针对 I/O 设备的软件模型必须有足够的通用性，至少能够将上节介绍的各种 I/O 硬件特性涵盖到模型之中。另外，操作系统必须提供有效的管理手段，从而让设备的软件组件融入到系统的 I/O 处理框架中，这样，这些软件组件可以专注于针对特定设备的功能需求，而不必过多地考虑与系统打交道或者与系统中其他模块的协作。例如，针对磁盘读写的软件组件无须考虑磁盘上的文件系统。

设备的软件组件通常称为**设备驱动程序**（device driver），或简称驱动程序。设备驱动程序或者由设备的硬件厂商提供，或者由操作系统厂商提供。由于设备驱动程序直接与硬件打交道，它们需要一定的特权才能操纵硬件设备，所以，驱动程序通常运行在处理器的特权模式下，其代码与操作系统的内核代码享有同样的特权。这种设计的一个潜在后果是，驱动程序中的错误可能会导致整个操作系统崩溃。因此，驱动程序的代码往往需要经过精心的设计、检查和调试，其代码规模应尽可能地小。从代码功能划分的角度来考虑，应尽可能把代码逻辑移到应用程序中，而让驱动程序只做最必要的事情。

与设备关联的驱动程序，其最基本的功能是，将操作系统的功能请求，解释成针对该设备的请求，然后操纵设备以完成该请求，最后将结果以指定的方式返回给调用者。在此过程中，有些事项往往是驱动程序必须要考虑并解决的。下面逐一讨论。

首先，**与设备的通信是同步或异步方式**。从软件控制流的角度，同步方式的通信容易实现，而且错误处理和恢复也相对容易。若一个 I/O 请求是异步方式，则处理器在向硬件发送请求以后，其指令流返回到上一级调用代码，此 I/O 请求被搁置起来，处理器的指令流转而执行其他不依赖于此 I/O 请求结果的任务。当设备或设备控制器完成了该请求（比如 DMA 数据传输）以后，它中断处理器的当前指令流，报告该请求的完成情况。处理器必须在中断处理过程中将原先被搁置的 I/O 请求接手过来，继续该请求后续的处理。

异步方式的通信虽然复杂，但是可以提高处理器的利用率，因而在现代计算机系统中被广泛地采用。因此，设备驱动程序除了要操纵设备控制器，还必须引入中断处理代码，以完成异步通信的后半部分任务。在上一章我们看到，Windows 提供了一套中断分发机制，允许设备驱动程序在不操纵处理器的 IDT 的情况下，向系统中增加中断服务例程。而且，在中断处理过程中，一些不那么紧急的任务可以放到 DPC 中完成，从而缩短处理器停留在高 IRQL 上的时间。

在设计 I/O 驱动程序时另一个往往要考虑的问题是**缓冲区管理**。不同设备对于数据传输

的效率有不同的要求，譬如，实时性要求高的软件或设备应尽可能减少数据拷贝的次数；驱动程序为了提高设备的 I/O 吞吐量，可能会将数据缓存起来，而不是立即发送。有些设备要求直接使用物理内存地址；有些设备驱动程序直接使用应用程序提供的缓冲区以减少数据拷贝次数。驱动程序的缓冲区管理相对比较复杂，尤其采用了异步通信方式以后，缓冲区作为一种有限的资源，更需要谨慎地设计和管理，以避免资源泄漏或不恰当地重复使用。

另一个值得提及的问题是**共享和独占设备**。在现代操作系统中，设备作为一种公共资源，通常是多个程序甚至多个用户所共享的。譬如，机器上的硬盘往往是系统中所有用户共享的，任何活动的应用程序，只要有权限许可，都可以访问它们。对于磁盘设备驱动程序而言，每个程序的硬盘 I/O 请求都是独立的，这些 I/O 请求可以并存。然而，对于有些设备，比如打印机，一个应用程序必须以独占方式使用该设备。并发地响应两个应用程序的打印请求，将会导致无效的打印结果。共享和独占方式的设备访问都有可能会引发上一章中提到的一些并发性问题，比如死锁。为避免这些问题，需要操作系统和设备驱动程序协同工作以打破这些问题的基本形成条件。

除了传输数据，设备驱动程序另一项重要的职责是**设备的状态管理**，包括设备初始化、即插即用、电源管理以及错误处理等。其中即插即用和电源管理是操作系统设备管理的一个重要任务，这些特性对于现代通用型计算机系统具有重要的意义。设备驱动程序必须在操作系统提供的管理框架下，达到设备状态与系统状态一致。错误处理应尽可能在下层软件（比如设备驱动程序）中完成，从而避免错误扩散到上层应用软件，这样也可以简化上层软件的设计。

接下来我们看操作系统如何为驱动程序提供一个环境。首先，设备驱动程序中的各种实体，包括设备、驱动程序本身，需要有一套标识方法。常见的做法是定义一个名字空间（name space），因而操作系统或者应用软件可以方便地命名与设备有关的各种实体。Windows 和 UNIX 都使用了字符串形式的名字空间。

设备驱动程序与操作系统之间应该是互补关系。设备驱动程序为操作系统提供了必要的访问硬件设备的能力，同时也依赖于操作系统提供的软件功能，比如内存管理、错误处理、同步互斥机制等。图 6.3 显示了两者之间的关系。对于多种设备通用的功能，操作系统尽可能在统一的系统模块中实现，比如同一总线上的设备共享总线协议，相同类型的设备共享许多基本特性。这种设计的好处是，设备驱动程序中的代码量可以尽可能地少，从而减小给操作系统带来不稳定因素的可能性。而且，这些可被多个设备共享的功能可以由操作系统厂商或其他专门的厂商来提供，他们可以比单个设备厂商投入更多的资源来实现并测试这些系统模块。

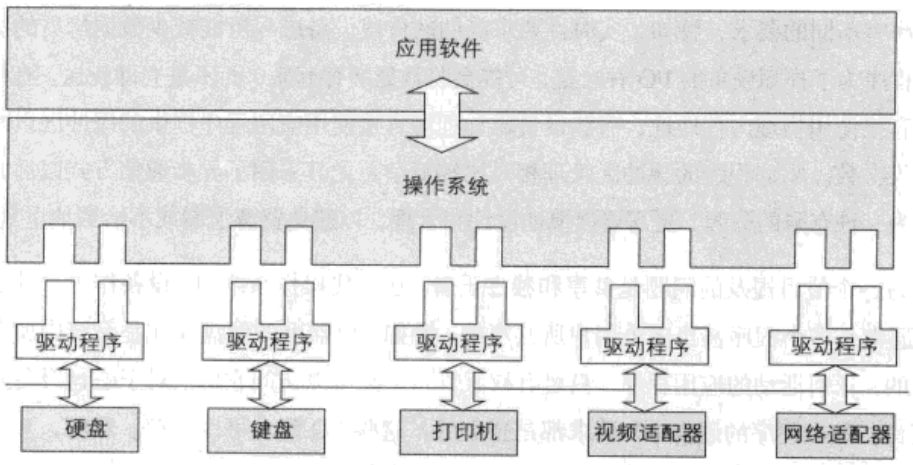


图 6.3 设备驱动程序与操作系统之间的互补关系

在现代计算机系统中，诸如即插即用、电源管理等特性越来越重要，操作系统的职责是提供一个符合即插即用和电源管理工业标准的软件框架。为了支持计算机系统的即插即用特性，操作系统自引导时开始，就将自动检测设备是否存在，并协调和分配相关的资源（比如中断、I/O 端口地址）；以后在系统运行过程中，继续管理这些设备的状态，包括动态地挂载和卸载设备，甚至升级驱动程序。同样地，电源管理也是一项需要软件和硬件紧密配合才能行之有效的特性，当操作系统接收或检测到电源状态变化时，它负责通知设备驱动程序，由驱动程序来确定设备的电源状态。简单说来，操作系统负责发送电源命令，而设备驱动程序负责实施电源命令，并向操作系统反馈设备的电源状态。

最后看一下用户程序中的 I/O 操作。通常，用户程序调用操作系统提供的系统服务来完成 I/O 功能。譬如，C 语言程序常常通过调用 C 运行库的 printf/scanf 函数来完成基本的 I/O 任务，这些函数可以把用户程序中的 I/O 请求转变成针对特定设备的 I/O 命令。虽然用户程序不能直接操纵设备，但是，它们通过操作系统提供的服务，可以共享或独占使用 I/O 设备。在实践中，操作系统的全局名字空间往往使得用户程序指定特定的设备非常方便。此外，让用户程序通过系统层的服务来访问设备，还有另一方面的好处，即操作系统可以提供虚拟的 I/O 设备或者用软件来模拟硬件设备。例如，操作系统可以利用网络功能来模拟大容量存储设备，从而使应用程序无须额外的努力便可以访问远程存储设备。

6.1.3 Windows I/O 系统结构

图 6.4 显示了 Windows 操作系统中的 I/O 系统结构。Windows 的 I/O 系统由 5 个部分组成：I/O 管理器、即插即用管理器、电源管理器、WMI 例程，以及设备驱动程序。其中

I/O 管理器是整个 I/O 系统的核心，它定义了一个非常通用的框架，允许各种功能的设备驱动程序容纳于其中。WRK 包含有 I/O 管理器的绝大部分代码，位于 base\ntos\io\iomgr 目录下。

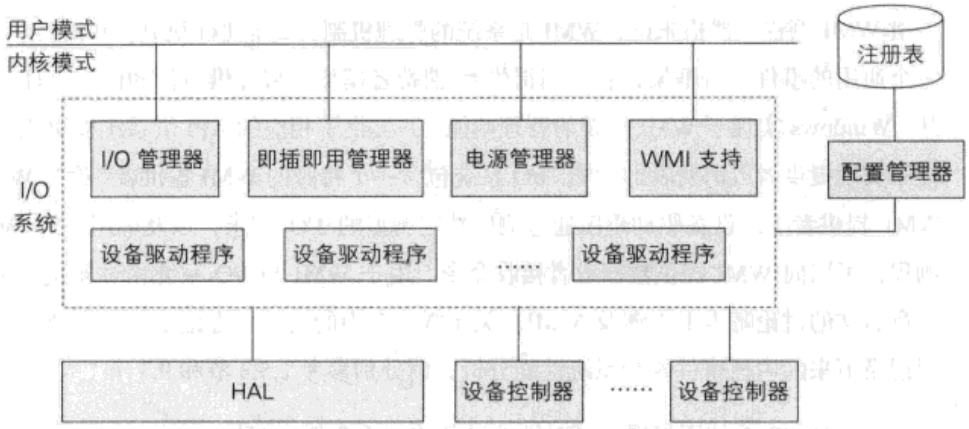


图 6.4 Windows I/O 系统结构图

I/O 管理器除了支持与设备相关的驱动程序以外，它也允许与设备无关的驱动程序加入到内核中。这一类驱动程序并不操纵任何硬件设备，它们进入到内核中以后，将变成内核的一部分，一旦经过 I/O 管理器的初始化，便与内核融为一体。因此，设备驱动程序也是内核扩展的一种形式。在本章后面我们将会看到这一类驱动程序是如何工作的。

Windows 设备驱动程序可以直接访问硬件，或者通过硬件抽象层（HAL）访问硬件。正如第 2 章所介绍，HAL 实际上将不同平台之间的差异隐藏了起来，向操作系统提供一组例程，允许操作系统的内核在多种体系结构下工作。这也同样适用于设备驱动程序，因此，设备驱动程序可以调用 HAL 中的例程。但对于 HAL 未涵盖的硬件特性，设备驱动程序可以直接操纵设备控制器，以完成其功能。

自 Windows 2000（也包括 Windows 98/Me）开始，Microsoft 定义了设备驱动程序的模型，称为 WDM（Windows Driver Model）。WDM 驱动程序除了遵循 I/O 管理器规定的驱动程序框架，还增加了对 Windows 即插即用（Plug and Play，简称 PnP）、电源管理以及 WMI（Windows Management Instrumentation，Windows 管理规范）的支持：

- Windows 定义了即插即用设备的管理框架，即插即用管理器与一种称为总线驱动程序（bus driver）的驱动程序一起协作，以便在检测到一个设备加入或移除时，加载或卸载该设备的驱动程序，而且，设备的硬件资源，比如 I/O 端口地址、中断向量等，也可以实现自动择优分配。

- 电源管理器也是 Windows I/O 系统的一部分，负责指示设备驱动程序执行电源状态的变化。Windows 的电源管理建立在 ACPI(Advanced Configuration and Power Interface，高级配置和电源接口) 规范的基础上。
- 一组 WMI 例程。严格来讲，WMI 是系统的管理机制，而非 I/O 机制。WMI 定义了一个通用的事件报告框架，它采用提供者-消费者模型，将提供事件和使用事件分离开。Windows 实现了 WMI 框架的管理功能，并提供了相应的 API 供系统模块或应用程序实现提供者或消费者的功能。I/O 系统包含一个特殊的 WMI 提供者（称为 WDM WMI 提供者）。设备驱动程序通过响应特定类型的 I/O 请求，以及调用一组 WMI 例程，可以向 WMI 提供数据或者接收命令。由于 WMI 与 I/O 系统相对独立，因此本章后文的讨论基本上不涉及 WMI。关于 WMI 中的内核日志记录器，以及如何利用记录下来的内核事件进行系统性能分析，请分别参考 2.5.3 节和 9.3 节。

Windows 的 I/O 系统不仅提供了对硬件设备的灵活控制，而且也允许多个驱动程序协同完成 I/O 任务，这是 Windows 的层次驱动程序模型，或者称为分层的驱动程序模型。当 I/O 管理器接到一个 I/O 请求时，它可以将该请求传递给一个设备栈。位于栈顶的驱动程序首先处理 I/O 请求，然后依次将其向下传递，直到该 I/O 请求被完成。通常，与硬件关联的驱动程序位于设备栈的底端，它们直接驱动硬件设备来完成 I/O 任务。这种分层模型带来了极大的灵活性，每个驱动程序只需关注自己的职责，而不必包揽 I/O 请求的完整处理过程。

Windows I/O 系统非常依赖注册表，例如，设备与驱动程序的很多配置信息都保存在注册表中，包括与硬件有关的一些描述信息，以及驱动程序初始化所需要的信息等。注册表是由 Windows 内核中的配置管理器来实现的。在本章中，我们只是简单地认为注册表是一个全局的信息仓库。关于注册表和配置管理器的详细描述，请参考 2.5.2 节。

6.2 I/O 管理器

I/O 管理器是 Windows I/O 系统的核心组件，它负责建立起基本的 I/O 框架结构，允许各种类型的驱动程序纳入到这一框架结构中。在 I/O 系统中，有三种基本内核对象：**驱动程序对象**代表了一个设备驱动程序被加载到系统中以后的内部表示；**设备对象**代表了系统中的一个设备，它既可以是物理设备，也可以是逻辑设备；**文件对象**代表了一个设备对象被打开后的实例。I/O 管理器负责管理和协调这三种对象，并建立起它们之间的关系。这一节我们将讨论这些对象的创建和管理，首先讲述驱动程序的加载和初始化过程。

6.2.1 驱动程序初始化

正如 2.6.2 节所介绍，I/O 系统的初始化是在内核的阶段 1 初始化过程中完成的。阶段 1 初始化的主函数是 Phase1InitializationDiscard，它调用 IoInitSystem 函数来初始化 I/O 系统，而且 IoInitSystem 函数的执行过程占据系统引导进度条的 25%~75%，请参考 base\ntos\init\initos.c 文件中 Phase1InitializationDiscard 函数的代码。

IoInitSystem 函数的代码位于 base\ntos\io\iomgr\ioinit.c 文件的 140~860 行，它首先初始化 I/O 系统中用到的各种全局数据结构，然后执行以下初始化工作：

- 调用 IopCreateObjectTypes 函数，创建 7 种类型对象：Adapter、DeviceHandler、Controller、Device、Driver、IoCompletion 和 File，分别存放在相应的全局变量中，这些全局类型变量如表 2.4 所列。
- 调用 IopCreateRootDirectories 函数，在对象管理器的根目录下创建 3 个目录对象：\Driver、\FileSystem 和\FileSystem\Filters。
- 调用 IopInitializePlugPlayServices 函数，执行即插即用管理器的阶段 0 初始化。
- 调用 PoInitDriverServices 函数，执行电源管理器的阶段 0 初始化。
- 调用 HalInitPnpDriver，执行 HAL 的即插即用总线驱动程序初始化。
- 调用 WMIInitialize，执行 WMI 的阶段 0 初始化。
- 再次调用 IopInitializePlugPlayServices，执行即插即用管理器的阶段 1 初始化。
- 调用 IopInitializeBootDrivers，初始化“引导-启动”类型的驱动程序。
- 调用 PpLastGoodDoBootProcessing，执行“最近一次的正确配置（LKG）”的处理。
- 调用 PsLocateSystemDll，初始化系统 DLL（ntdll.dll），并映射到 System 进程中。
- 调用 IopInitializeSystemDrivers，加载“系统-启动”类型的驱动程序，并对它们进行初始化。
- 调用 IopCallDriverReinitializationRoutines，处理需要重新初始化的驱动程序。
- 调用 IopReassignSystemRoot，将系统根目录（\SystemRoot）替换成 NT 路径名，这是一个符号链接（symbolic link）名称。
- 调用 IopProtectSystemPartition，保护系统分区。

- 调用 IoAssignDriveLetters，为磁盘分区和 CD-ROM 驱动器分配 DOS 驱动器字母。
- 再次调用 WMIInitialize，执行 WMI 的阶段 1 初始化。
- 再次调用 PoInitDriverServices，执行电源管理器的阶段 1 初始化。

WRK 并没有包含以上所有被调用的函数，但是，IoInitSystem 函数所做的初始化事项却是简单明了。在以上步骤中，“引导-启动”的驱动程序和“系统-启动”的驱动程序分别被加载和初始化。下面我们讨论驱动程序的启动类型以及它们的初始化过程。

Windows 中的每一个驱动程序（和 Windows 服务）在安装时都被指定了一个启动类型，存放在注册表中。启动类型是一个整数值，其定义如下：

```
#define SERVICE_BOOT_START          0x00000000
#define SERVICE_SYSTEM_START        0x00000001
#define SERVICE_AUTO_START          0x00000002
#define SERVICE_DEMAND_START        0x00000003
#define SERVICE_DISABLED            0x00000004
```

除了禁止启动的驱动程序，系统中安装的驱动程序有 4 种可能的启动方式：“引导-启动”、“系统-启动”、“自动-启动”和“按需-启动”，其值分别对应于 0~3，如上述宏所定义。这些注册表值定义在 HKLM\System\CurrentControlSet\Services\<*DriverName*>键的 Start 值中，这里 *DriverName* 代表了驱动程序的名称。

下面逐一介绍这 4 种启动类型的驱动程序的加载和初始化过程。首先，“引导-启动”类型的驱动程序是由系统加载器，即 ntldr 程序，加载到系统空间中的，因此，内核无须加载它们，直接初始化即可。如上面所介绍，这些驱动程序的初始化过程是在 IopInitializeBootDrivers 函数中完成的。IopInitializeBootDrivers 为每个驱动程序调用 IopInitializeBuiltinDriver 函数以完成实际的初始化工作。IopInitializeBuiltinDriver 函数的代码位于 ioinit.c 文件的 2 430~2 740 行，其原型如下：

```
NTSTATUS
IopInitializeBuiltinDriver(
    IN PUNICODE_STRING DriverName,
    IN PUNICODE_STRING RegistryPath,
    IN PDRIVER_INITIALIZE DriverInitializeRoutine,
    IN PKLDR_DATA_TABLE_ENTRY DriverEntry,
    IN BOOLEAN IsFilter,
    OUT PDRIVER_OBJECT *Result
    );
```

这里 DriverName 参数指定了要初始化的驱动程序的名称，RegistryPath 参数指定了该驱动程序对应的注册表键的路径，DriverInitializeRoutine 参数指定了该驱动程序的初始化例程，DriverEntry 参数指定了该驱动程序在加载参数块中的相应表项（这是由 ntldr 准备

好并传递给内核 ntoskrnl.exe 模块的），IsFilter 参数指定了是否为过滤驱动程序（参见 6.5.1 节的介绍），Result 参数是一个返回参数，若初始化成功，则指向一个驱动程序对象。

IopInitializeBuiltinDriver 函数的基本流程如下：根据参数中指定的驱动程序名称，创建一个驱动程序对象，其类型是 IoInitSystem 函数已经注册的 IoDriverObjectType；然后初始化该驱动程序对象，设置它的 DriverInit 域为参数中指定的初始化例程；将驱动程序对象插入到当前进程（即 System 进程）的句柄表中；搜索系统的已加载模块列表（全局变量 PsLoadedModuleList），找到该驱动程序所在的模块项，并初始化驱动程序对象中与映像模块有关的域；将驱动程序的名称复制到驱动程序对象的名称缓冲区中，该缓冲区是从换页内存池中分配的；记录下驱动程序注册表键的名称；调用驱动程序的初始化函数，即 DriverInitializeRoutine 参数中指定的例程；最后，调用 IopReadyDeviceObjects 函数，将该驱动程序创建的设备对象设置成已被初始化，从而可被其他驱动程序或客户访问。

值得一提的是，所加载的驱动程序的初始化例程实际上是驱动程序二进制映像文件中的入口函数（通常该函数的名称为 DriverEntry）。I/O 管理器在调用该例程时，会将驱动程序对象和注册表路径传递进去，因此，初始化例程可以访问驱动程序对象中的信息，尤其是其中的 HardwareDatabase 域，它包含了当前系统中关于硬件的描述。驱动程序在得到了它的注册表路径以后，可以利用此路径来保存有关的信息。如果驱动程序要保存该路径，则必须自己复制一份路径字符串，因为 IopInitializeBuiltinDriver 函数传递给它的字符串是临时缓冲区。“引导-启动”的驱动程序不是过滤驱动程序，所以 IsFilter 参数为 FALSE。

接下来我们来看“系统-启动”类型的驱动程序的初始化，这是在 IopInitializeSystemDrivers 函数中完成的。该函数通过 CmGetSystemDriverList 函数（其代码位于 base\ntos\config\cmsysini.c 文件中），获得在注册表中“Start”值被指定为 SERVICE_SYSTEM_START 的驱动程序的列表，然后对尚未加载到系统中的驱动程序逐个调用 IopLoadDriver 函数，将其加载到系统地址空间，并执行初始化。IopLoadDriver 函数的原型如下：

```
NTSTATUS
IopLoadDriver(
    IN  HANDLE     KeyHandle,
    IN  BOOLEAN    CheckForSafeBoot,
    IN  BOOLEAN    IsFilter,
    OUT NTSTATUS   *DriverEntryStatus
    );
```

IopLoadDriver 函数的代码位于 base\ntos\io\iomgr\internal.c 文件的 3 688~4 380 行。其中 KeyHandle 参数指向一个驱动程序的注册表键；CheckForSafeBoot 参数指示是否要检查

安全模式，如果该参数为 TRUE，那么，当系统以安全模式引导时，只有该驱动程序属于安全模式允许的驱动程序列表，它才被加载。IsFilter 参数指定了是否为过滤驱动程序；DriverEntryStatus 是返回参数。

IopLoadDriver 是一个通用函数，它不仅用于引导时加载“系统-启动”类型的驱动程序，而且也被用于系统正常运行过程中动态地加载驱动程序。本节后面我们还会看到该函数被调用的情形。当 IopInitializeSystemDrivers 调用该函数时，CheckForSafeBoot 参数为 TRUE，而 IsFilter 参数为 FALSE。

IopLoadDriver 函数的流程大体如下：首先根据参数中指定的注册表键，构造出驱动程序的全路径名；然后，调用 MmLoadSystemImage 函数，将驱动程序加载到系统地址空间。后面的步骤与 IopInitializeBuiltinDriver 相仿：创建一个驱动程序对象，初始化该对象的域，尤其是，它的 DriverInit 成员指向驱动程序的入口函数；然后将该对象插入到当前进程的句柄表中。完成了对象成员初始化以后，再调用驱动程序的初始化函数，即 DriverInit 成员，并将驱动程序对象和注册表路径传递进去。

然而，IopLoadDriver 函数实际的代码比以上这些步骤要复杂一些。它需要检查安全模式引导选项，还需要检查此驱动程序是否已经在系统的已加载模块列表中。它还需要考虑该驱动程序已经被初始化了的情形，所以要试着打开该驱动程序。另外，它也会考虑，如果该驱动程序并非是老式的驱动程序（Legacy Driver）但是又没有增加任何设备对象，则认为该驱动程序的初始化不成功，于是调用它的 Unload 函数将其立即卸载。

有一点值得一提，IopLoadDriver 和 IopInitializeBuiltinDriver 函数都必须在 System 进程中运行，所以，新创建的驱动程序对象将被加入到该进程的句柄表中。这对于“引导-启动”和“系统-启动”类型的驱动程序不成问题，因为它们的上游调用者 IopInitializeBootDrivers 和 IopInitializeSystemDrivers 都是在 IoInitSystem 函数中被调用的。然而，对于“自动-启动”类型的驱动程序，情形就不同了。

“自动-启动”类型的驱动程序，即注册表中“Start”值为 SERVICE_AUTO_START（2）的驱动程序，是在系统引导过程的后期，由 SCM（服务控制管理器，Service Control Manager，services.exe 进程）加载的，参见 2.6.3 节。在内核中，这是通过 NtLoadDriver 函数来完成的。NtLoadDriver 是一个系统服务，其原型如下：

```
NTSTATUS
NtLoadDriver (
   __in PUNICODE_STRING DriverServiceName
   );
```

参数 DriverServiceName 指定了待加载的驱动程序在注册表中的名称。NtLoadDriver

的功能并不复杂，其代码位于 base\ntos\io\iomgr\loadunld.c 文件的 26~172 行。如果用户模式代码调用 NtLoadDriver，那么，它需要检查当前进程是否有加载驱动程序的特权。然后，若当前进程是 System 进程，则直接调用 IopLoadUnloadDriver 函数完成驱动程序初始化工作；否则，将 IopLoadUnloadDriver 放到一个工作项目（WorkItem）中，由 System 进程中的系统线程来调用 IopLoadUnloadDriver 函数。

现在来看 IopLoadUnloadDriver 函数，它位于 base\ntos\io\iomgr\internal.c 文件的 4 583~4 681 行。该函数既可以用于加载驱动程序，也可以用于卸载驱动程序，取决于参数（LOAD_PACKET 数据结构）中指定的 DriverObject 是否已被赋了非零值。它直接调用 IopLoadDriver 函数来完成驱动程序的加载和初始化，这里不再赘述。

最后是“按需-启动”类型的驱动程序，这是在系统运行过程中，有些用户程序或者系统模块根据需要而加载的驱动程序。譬如前面三章中介绍的工具软件在第一次运行时，需要加载一个驱动程序，以便完成某些必需在内核中完成的工作。这些工具软件通过 SCM 来加载和初始化驱动程序。在这种情况下，驱动程序的加载经由 NtLoadDriver 系统服务来完成。此外，内核中的模块也可以根据需要加载并初始化“按需-启动”类型的驱动程序，譬如，在前面三类驱动程序的初始化过程中就存在这样的需求。在这种情况下，内核代码要么直接调用 IopLoadDriver 函数来完成驱动程序的加载，要么通过一个工作项目由系统辅助线程来完成加载工作。

通过上面的介绍，我们看到，这 4 类驱动程序都有一个映像文件，而且，驱动程序的初始化例程直接对应于映像文件的入口函数。实际上，Windows 也允许创建无映像文件的驱动程序对象。譬如，Windows 创建的第一个驱动程序对象“\Driver\PnpManager”正是这样一个驱动程序。这一类驱动程序是通过 IoCreateDriver 函数来创建的，该函数位于 base\ntos\io\iomgr\iosubs.c 文件中，以下是其函数原型：

```
NTSTATUS
IoCreateDriver(
   IN PUNICODE_STRING DriverName   OPTIONAL,
   IN PDRIVER_INITIALIZE InitializationFunction
   );
```

该函数非常直截了当，可选参数 DriverName 指定了驱动程序的名称，InitializationFunction 参数指向一个初始化函数。IoCreateDriver 函数首先构造出驱动程序的名称，如果 DriverName 没指定名称，则 IoCreateDriver 函数利用当前时间信息生成一个驱动程序名称。然后，调用 ObCreateObject 函数创建一个驱动程序对象，并初始化该对象，再调用 ObInsertObject 函数，将该对象插入到当前进程的句柄表中。最后，调用参数 InitializationFunction 指定的初始化函数，并将驱动程序对象传递给它。

至此，我们已经了解了驱动程序的加载和初始化过程。一旦一个驱动程序被加载并且其初始化例程被执行，则该驱动程序就已经融入内核：它的代码已进入到系统地址空间，成为内核模式下可执行代码的一部分；它已经获得了初始的执行权，以后如何再次获得执行权，取决于初始例程中的代码逻辑。本章 6.5.3 节将介绍驱动程序的代码结构，我们将可以看到驱动程序中的例程如何获得控制。

在对象管理器的名字空间中，驱动程序对象位于\FileSystem 和\Driver 目录下，表 6.1 显示了在 Virtual PC 环境中一个典型的 Windows Server 2003 SP1 系统的“\Driver”目录中的驱动程序对象，表 6.2 显示了同一系统的“\FileSystem”目录中的驱动程序对象。通过这两张表，我们可以看到一个典型的 Windows 系统包含哪些驱动程序。不难看出，Windows 中涉及 I/O 的绝大多数功能都是通过驱动程序来完成的。

表 6.1 Virtual PC 环境下 Windows Server 2003 SP1 系统的“\Driver”目录中的驱动程序

驱动程序	说明	驱动程序	说明
1-driver-vmsrvc	虚拟机附加服务驱动程序	NdisTapi	NDIS TAPI 驱动程序
ACPI	ACPI 驱动程序	Ndisuio	NDIS 用户模式 I/O 驱动程序
ACPI_HAL	HAL 中有关 ACPI 的虚拟驱动	NdisWan	NDIS WAN 驱动程序
AFD	WinSock 辅助功能驱动程序	NDProxy	NDIS 代理
Atapi	标准的 IDE/ESDI 硬盘控制器	NetBT	基于 TCP/IP 的 NetBIOS 驱动程序
audstub	声音存根驱动程序	Null	NULL 驱动程序
Beep	BEEP 驱动程序	Parport	并口驱动程序
Cdrom	CD-ROM 驱动程序	PartMgr	磁盘分区管理器
crcdisk	磁盘块检验驱动程序	Parvdm	VDM 并口驱动程序
DC21x4	DC21x4 网络适配器驱动程序	PCI	PCI 总线驱动程序
Disk	即插即用磁盘驱动程序	PnpManager	即插即用管理器
Dmio	NT 磁盘管理器 I/O 驱动程序	PptpMiniport	对等网络隧道协议
dmload	NT 磁盘管理器启动驱动程序	Ptilink	直接并行链接驱动程序（Direct Parallel Link Driver）
Fdc	软盘控制器驱动程序	RasAcd	RAS 自动连接驱动程序
Fips	FIPS 驱动程序	Rasl2tp	RAS L2TP 小端口驱动程序
Flpydisk	软盘驱动程序	RasPppoe	RAS PPPoE 小端口驱动程序
Ftdisk	简单卷管理驱动程序	Raspti	PTI 直接并行小端口驱动程序
gameenum	游戏端口列举器（Enumerator）	RDPCDD	RDP 小端口驱动程序

续表

驱动程序	说明	驱动程序	说明
Gpc	通用包分类器	rdpdr	RDP 设备重定向器驱动程序
i8042prt	i8042 端口驱动程序	serenum	串口列举器驱动程序
IntelIde	Intel PCI IDE 驱动程序	Serial	串口驱动程序
IPSec	IPSec 驱动程序	swenum	即插即用软件设备列举器
isapnp	PNP ISA 总线驱动程序	Tcpip	TCP/IP 协议驱动程序
Kbdclass	键盘类驱动程序	TermDD	终端服务驱动程序
KSecDD	内核安全支持提供者接口	Update	Microsoft 更新驱动程序
mnmdd	帧缓冲区模拟器	VgaSave	VGA 视频驱动程序
Mouclass	鼠标类驱动程序	VolSnap	卷影像拷贝驱动程序
MountMgr	挂载点管理器	vpc-s3	虚拟机 S3 小端口驱动程序
mssmbios	系统管理 BIOS 驱动程序	Wanarp	远程访问路由 ARP 驱动程序
msvmmouf	虚拟机附加的鼠标过滤驱动	Win32k	Win32 子系统驱动程序
NDIS	NDIS 系统驱动程序	WMIxWDM	WMI 驱动程序

表 6.2 Virtual PC 环境下 Windows Server 2003 SP1 系统的“\FileSystem”目录中的驱动程序

驱动程序	说明	驱动程序	说明
Cdfs	CD-ROM 文件系统驱动程序	Mup	多 UNC 提供者驱动程序
DfsDriver	分布式文件系统过滤驱动程序	NetBIOS	NetBIOS 接口驱动程序
FltMgr	文件系统过滤管理器	Npfs	命名管道（named pipe）驱动程序
Fs_Rec	文件系统识别器驱动程序	Ntfs	NTFS 驱动程序
MRxSmb	NT SMB 驱动程序	RAW	内置的 RAW 文件系统驱动程序
MrxVPC	虚拟机文件夹共享驱动程序	Rdbss	重定向的驱动器缓冲子系统驱动程序
Msfs	邮件槽（mailslot）驱动程序	Srv	服务器驱动程序

6.2.2 驱动程序对象和设备对象

当 I/O 管理器加载一个设备驱动程序时，它会创建一个驱动程序对象，该对象在对象管理器目录中的路径为：\Driver\<*DriverName*>或\FileSystem\<*DriverName*>。如果是文件系统类型的驱动程序，则该对象被放置在“\FileSystem”目录下，否则放在“\Driver”目录下。因此，驱动程序可粗略地分为文件系统驱动程序和非文件系统驱动程序，后面 6.5.1 节将全面地讨论驱动程序的分类。

与驱动程序相关的另一种对象是设备对象。每个设备对象代表了系统中的一个设备，包括逻辑设备和物理设备。正常情况下，有两种途径可创建设备对象：即插即用管理器在检测到设备时，通过调用驱动程序的 AddDevice 例程来创建设备对象；或者，非即插即用驱动程序在它们的初始化例程中创建设备对象。每个设备对象都必定有一个为它负责的驱动程序。对设备对象的各种操作实际上是由为它负责的驱动程序中的例程来完成的。一个驱动程序可以支持多个设备，所以，驱动程序对象中有一个链表记录了它所负责的所有设备对象。

在上一小节中我们已经看过了驱动程序对象的创建过程，现在来看设备对象的创建。在 Windows 中，这是通过 I/O 管理器的 IoCreateDevice 函数来完成的，该函数原型如下：

```
NTSTATUS
IoCreateDevice(
   IN PDRIVER_OBJECT DriverObject,
   IN ULONG DeviceExtensionSize,
   IN PUNICODE_STRING DeviceName OPTIONAL,
   IN DEVICE_TYPE DeviceType,
   IN ULONG DeviceCharacteristics,
   IN BOOLEAN Exclusive,
   OUT PDEVICE_OBJECT *DeviceObject
   );
```

DriverObject 参数指向负责该设备的驱动程序对象；DeviceExtensionSize 参数定义了待创建设备对象的扩展部分的大小，此扩展部分是由驱动程序来指定和使用的；DeviceName 可选参数指定了设备的名称。DeviceType 参数定义了设备的类型，DEVICE_TYPE 是无符号整数类型，Microsoft 已经预定义了所有常用的设备，它们的整数值小于 32 767，若创建者需要使用自定义的设备类型，可以使用大于 32 767 的值。关于这些预定义值，参见 public\sdk\inc\devioctl.h 文件中的常量定义。DeviceCharacteristics 参数指定了设备的特征；Exclusive 参数指定了在创建设备对象时是否使用互斥标志；DeviceObject 参数是一个输出参数，用于存放所创建的设备对象。

IoCreateDevice 函数的代码位于 base\ntos\io\iomgr\iosubs.c 文件的 4 275~4 713 行。它首先根据参数中指定的要求，构造出设备的名称，并且创建一个安全描述符，用于对该设备的访问控制。然后，调用 ObCreateObject 函数创建一个 IoDeviceObjectType 类型的内核对象。如果由于自动产生的名称发生冲突而导致 ObCreateObject 调用不成功，则重试此过程。然后，IoCreateDevice 函数初始化新建的设备对象中的成员，并调用 ObInsertObject 函数，将设备对象插入到进程的句柄表中。最后，设定该设备对象中的驱动程序对象，并将设备对象插入到驱动程序对象的设备链表中，因而将设备对象与驱动程序对象关联起来。

下面来看一下驱动程序对象和设备对象的定义。以下是驱动程序对象的数据结构：

```
typedef struct _DRIVER_OBJECT {
   CSHORT Type;
```

```
    CSHORT Size;

    PDEVICE_OBJECT DeviceObject;            // 指向设备对象，所有的设备对象构成一个链表
    ULONG Flags;                            // 驱动程序标志

    PVOID DriverStart;                      // 驱动程序映像起始地址
    ULONG DriverSize;                       // 驱动程序映像大小
    PVOID DriverSection;                    // 指向驱动程序映像的内存区对象
    PDRIVER_EXTENSION DriverExtension;      // 指向驱动程序对象的扩展部分

    UNICODE_STRING DriverName;              // 驱动程序名称

    PUNICODE_STRING HardwareDatabase;       // 指向注册表中包含硬件信息的路径

    PFAST_IO_DISPATCH FastIoDispatch;       // 指向快速 I/O 的分发结构

    PDRIVER_INITIALIZE DriverInit;          // 驱动程序的初始化例程
    PDRIVER_STARTIO DriverStartIo;          // 驱动程序的启动 I/O 例程
    PDRIVER_UNLOAD DriverUnload;            // 驱动程序的卸载例程
    PDRIVER_DISPATCH MajorFunction[IRP_MJ_MAXIMUM_FUNCTION + 1];
} DRIVER_OBJECT;

// 以下是扩展部分的数据结构 DRIVER_EXTENSION 的定义
typedef struct _DRIVER_EXTENSION {
    struct _DRIVER_OBJECT *DriverObject;    // 指向驱动程序

    // AddDevice 是一个函数指针，当即插即用管理器检测到一个新的设备由该驱动程序负责时，
    // 调用该函数，以便通知该驱动程序
    PDRIVER_ADD_DEVICE AddDevice;

    ULONG Count;                            // 记录了重新初始化调用的次数
    UNICODE_STRING ServiceKeyName;          // 驱动程序的服务名称

    PIO_CLIENT_EXTENSION ClientDriverExtension; // 指向驱动程序的客户扩展部分
    PFS_FILTER_CALLBACKS FsFilterCallbacks;     // 用于文件系统过滤驱动程序
} DRIVER_EXTENSION, *PDRIVER_EXTENSION;
```

驱动程序对象中的 MajorFunction 数组包含了一组例程，当 I/O 管理器接收到一个 I/O 请求时，它将根据 I/O 请求中的有关信息，找到设备对象的驱动程序对象，并调用驱动程序中相应的例程来处理该 I/O 请求。通常，设备驱动程序的初始化例程会填充 MajorFunction 数组中的例程。对于初始化例程未填充的数组项，创建驱动程序对象的函数（如 IopLoadDriver 和 IoCreateDriver）会将其填充为 IopInvalidDeviceRequest 函数。

下面是设备对象的数据结构：

```
typedef struct _DEVICE_OBJECT {
    CSHORT Type;
    USHORT Size;
    LONG ReferenceCount;                        // 引用计数值
    struct _DRIVER_OBJECT *DriverObject;        // 指向关联的驱动程序对象
    struct _DEVICE_OBJECT *NextDevice;          // 指向关联同一驱动程序对象的下一个设备对象
    struct _DEVICE_OBJECT *AttachedDevice;      // 附载的设备，与 AttachedTo 域构成双链表关系
    struct _IRP *CurrentIrp;                    // 当前正在处理的 I/O 请求包
    PIO_TIMER Timer;                            // 设备对象的定时器
    ULONG Flags;                                // 设备对象标志，以 DO_作为前缀的一组常量
    ULONG Characteristics;                      // 设备的特征，以 FILE_作为前缀的一组常量
```

```
    PVPB Vpb;                                   // 指向设备的卷参数块（Volumn Parameter Block）
    PVOID DeviceExtension;                      // 指向设备对象的扩展部分
    DEVICE_TYPE DeviceType;                     // 设备类型
    CCHAR StackSize;                            // 设备栈的大小
    union {
        LIST_ENTRY ListEntry;                   // 用于文件系统的设备对象，形成一个链表
        WAIT_CONTEXT_BLOCK Wcb;                 // 等待环境块，用于与控制器对象协作
    } Queue;
    ULONG AlignmentRequirement;                 // 缓冲区的对齐要求，其值等于对齐边界减一
    KDEVICE_QUEUE DeviceQueue;                  // 设备队列，存放针对该设备的 I/O 请求
    KDPC Dpc;

    ULONG ActiveThreadCount;                    // 用于文件系统：使用此设备对象的线程数
    PSECURITY_DESCRIPTOR SecurityDescriptor;  // 设备的安全描述符
    KEVENT DeviceLock;                          // 设备锁

    USHORT SectorSize;                          // 扇区大小
    USHORT Spare1;

    struct _DEVOBJ_EXTENSION  *DeviceObjectExtension;  // 指向扩展部分
    PVOID  Reserved;
} DEVICE_OBJECT;
typedef struct _DEVICE_OBJECT *PDEVICE_OBJECT;
```

设备对象扩展部分的定义如下：

```
typedef struct _DEVOBJ_EXTENSION {
    CSHORT        Type;
    USHORT        Size;

    PDEVICE_OBJECT  DeviceObject;                  // 指向关联的设备对象
    ULONG         PowerFlags;                      // 电源标志
    struct        _DEVICE_OBJECT_POWER_EXTENSION  *Dope; //设备对象的电源扩展部分
    ULONG ExtensionFlags;                          // 设备对象扩展标志

    PVOID         DeviceNode;                      // 设备节点域，由即插即用管理器使用
    PDEVICE_OBJECT  AttachedTo;                    // 当前设备对象被附载到此设备对象

    // 以下三个域用于 IoStart*函数
    LONG          StartIoCount;                    // 已启动但未完成的 I/O 的数量
    LONG          StartIoKey;                      // 下一个启动 I/O 的键
    ULONG         StartIoFlags;                    // 启动 I/O 标志
    PVPB          Vpb;          // 已挂载卷的 VPB，用于文件系统的卷设备对象
} DEVOBJ_EXTENSION, *PDEVOBJ_EXTENSION;
```

设备对象描述了一个特定设备的状态信息，包括它所接收到的 I/O 请求和设备的电源特性等。正如我们在 IoCreateDevice 函数中看到的那样，设备对象的 DriverObject 域指向负责该设备的驱动程序，而它的 NextDevice 域构成了同属一个驱动程序的设备对象单链表。IoCreateDevice 调用 IopInsertRemoveDevice 函数，将设备对象插入到此链表中。另一方面，设备对象的 AttachedDevice 域和扩展部分的 AttachedTo 域构成了一个双链表节点中的前后指针。本章前面提到过，在 Windows 的层次驱动程序模型中，I/O 请求可以被传递给一个设备栈进行处理。设备栈中的设备对象相互链接起来，创建这些设备对象的驱动程序相互协作来处理针对特定设备的 I/O 请求。因而，当 I/O 管理器接收到针对这一设备

的 I/O 请求时，它会依次将该 I/O 请求传递给这些设备对象，交给它们处理。设备对象的 StackSize 域指定了为处理 I/O 请求而最小需要的栈深度，而 AttachedTo 成员将设备栈中的设备对象自栈顶向底层（接近硬件设备的设备对象）链接起来，AttachedDevice 成员则相反地自底向上将这些设备对象链接起来。

图 6.5 显示了在虚拟机环境中 Windows Server 2003 SP1 系统的三个设备对象所构成的设备栈，我们可以看到设备对象中的 AttachedDevice 和 AttachedTo 成员所形成的链表结构。驱动程序 ACPI 创建了最底层的设备对象“\Device\0000003b”，它要求最小的栈深度为 4，这是在 ACPI 驱动程序中指定的。驱动程序 i8042prt 创建了中间的无名设备对象，它的 AddDevice 例程调用 IoAttachDeviceToDeviceStack 函数，完成此无名设备对象的挂载工作。进一步，驱动程序 kbdclass 创建了最上边的设备对象“\Device\KeyboardClass0”，类似地，它的 AddDevice 例程调用 IoAttachDevice 函数，完成 KeyboardClass0 设备对象的挂载工作。IoAttachDeviceToDeviceStack 和 IoAttachDevice 函数的代码位于 base\ntos\io\iomgr\iosubs.c 文件中，真正实现挂载功能的函数是位于同一文件中的 IopAttachDeviceToDeviceStackSafe 函数，其代码非常直截了当，只是将一个设备对象按照前面描述的结构关系挂载到另一个设备对象所在设备栈的顶部。

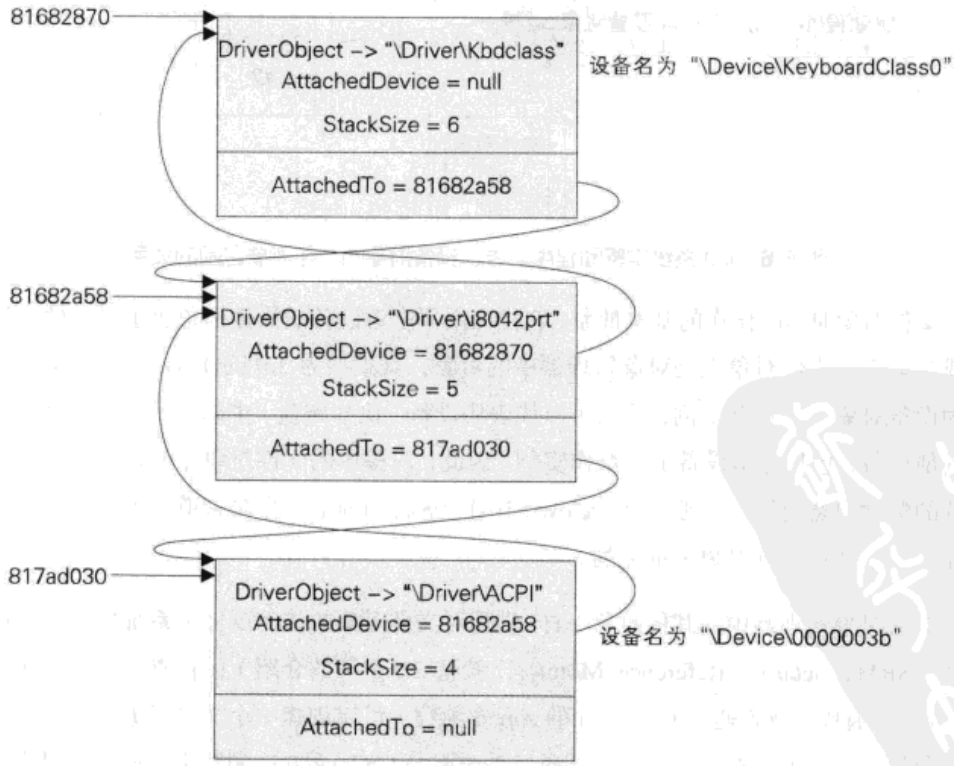


图 6.5 设备栈示例图

6.2.3 文件对象

上一小节介绍的驱动程序对象和设备对象都是内核中的对象。在 Windows 中，所有的 I/O 请求都是针对设备对象的，然而，应用程序并不能直接访问设备对象，尽管它们可以通过符号名称来指定已命名的设备对象。Windows I/O 系统提供给应用程序的 I/O 操作的目标对象是文件对象（File Object）。文件对象代表了设备对象的已打开实例，也就是说，内核或应用程序每打开（open）一个设备对象，就将得到一个文件对象。文件对象也是内核对象，但用户模式代码可以通过句柄来引用文件对象。驱动程序对象、设备对象和文件对象之间的关系如图 6.6 所示。

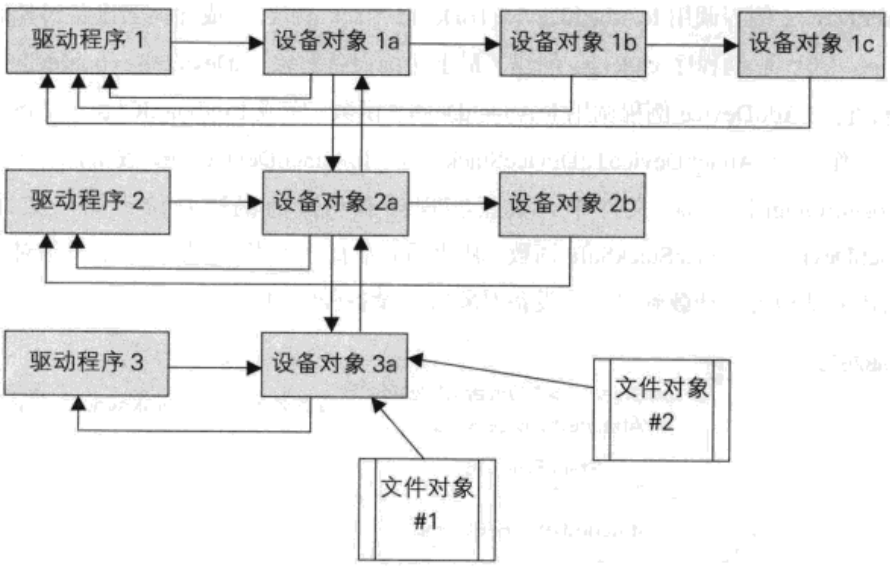


图 6.6 I/O 系统中驱动程序对象、设备对象和文件对象之间的关系

文件对象是 I/O 操作的基本抽象，它将应用程序对设备的操作抽象成了对文件数据的读或写动作。文件对象也是对象管理器中的对象，其类型为 IoFileObjectType。文件对象作为设备对象的已打开实例，不仅可以代表块设备（比如磁盘）中的文件，还可以代表任何其他设备，包括逻辑设备上的操作实例。因此，内核中的文件对象比通常意义上的磁盘文件的概念要宽泛得多。例如，Windows 中用于进程间通信的命名管道对象和邮件槽对象实际上也是通过文件对象来描述的。

如同对象管理器中的其他对象一样，进程对文件对象的访问要经过系统的安全引用监视器（SRM，Security Reference Monitor，参见 2.5.4 节的介绍）的检查。应用程序调用 CreateFile 函数（或者通过 C 运行库的 fopen 函数）可以创建一个文件对象，并得到该文件对象的一个句柄。在内核中，这是通过系统服务 NtCreateFile 函数来完成的。以后，应用程序通过 ReadFile 或 ReadFileEx，以及 WriteFile 或 WriteFileEx 函数来读写此文件对象。

当应用程序要结束对文件对象的操作时，只需调用 CloseHandle 函数即可。文件对象通过引用计数来维护其自身的生命周期。

文件对象代表了设备对象的已打开实例，它并不承担设备对象的数据存储和状态变迁的能力。真正的文件数据位于设备对象而非文件对象中。多个文件对象可以指向同一个设备对象，因而它们共享同样的设备对象。对文件对象的操作有必要进行同步。譬如，如果一个线程要对一个文件进行写操作，那么它在打开该文件时，必须指定它要对文件进行互斥写访问，以防止其他的线程或进程同时执行写操作。

在介绍文件对象的打开和读写操作以前，我们先看一下 Windows 中文件对象的定义，如下所示（见 base\ntos\inc\io.h 文件）：

```
typedef struct _FILE_OBJECT {
    CSHORT Type;
    CSHORT Size;
    PDEVICE_OBJECT DeviceObject;        // 指向文件所在的设备对象
    PVPB Vpb;                           // 指向文件对象所在卷的卷参数块（VPB）
    PVOID FsContext;                    // 指向驱动程序为该文件对象维护的状态信息
    PVOID FsContext2;                   // 指向驱动程序为该文件对象维护的额外状态信息
    PSECTION_OBJECT_POINTERS SectionObjectPointer;  // 文件对象的内存区对象指针
    PVOID PrivateCacheMap;              // 文件对象的私有缓存表
    NTSTATUS FinalStatus;               // 文件对象 I/O 请求的最终状态
    struct _FILE_OBJECT *RelatedFileObject;// 相关的文件对象
    BOOLEAN LockOperation;              // 是否已在文件对象上执行了锁（lock）操作
    BOOLEAN DeletePending;              // 正在执行一个删除与文件对象关联的文件的操作
    BOOLEAN ReadAccess;                 // 以读访问方式打开该文件
    BOOLEAN WriteAccess;                // 以写访问方式打开该文件
    BOOLEAN DeleteAccess;               // 以删除访问方式打开该文件
    BOOLEAN SharedRead;                 // 以读共享访问方式打开该文件
    BOOLEAN SharedWrite;                // 以写共享访问方式打开该文件
    BOOLEAN SharedDelete;               // 以删除共享访问方式打开该文件
    ULONG Flags;                        // 标志，以 FO_作为前缀定义的一组常量，可以组合
    UNICODE_STRING FileName;            // 文件名，仅在 IRP_MJ_CREATE 请求中有效
    LARGE_INTEGER CurrentByteOffset;    // 文件中的当前偏移位置，以字节为单位
    ULONG Waiters;                      // 有多少个线程在等待该文件对象，以进行同步访问
    ULONG Busy;                         // 当前是否有线程在以同步方式访问该文件对象
    PVOID LastLock;                     // 指向上一个应用在该文件对象上的字节范围锁
    KEVENT Lock;                        // 文件对象锁，用于同步访问该文件对象
    KEVENT Event;                       // 文件对象锁，用于 I/O 请求的完成通知
    PIO_COMPLETION_CONTEXT CompletionContext;  // 指向与文件对象关联的完成端口信息
} FILE_OBJECT;
typedef struct _FILE_OBJECT *PFILE_OBJECT;
```

文件对象数据结构已被文档化，并包含在 Windows 的驱动程序开发包中。设备对象是系统范围内共享的真正实体对象，而文件对象只是代表了它的一个已打开实例，因此，FILE_OBJECT 数据结构只需维护一个访问实例所需要的信息即可。在以上定义中，我们可以看到，FILE_OBJECT 包含了指向设备对象的指针、由驱动程序为文件对象维护的状态环境、当前位置信息、访问方式和文件对象标志，以及当多个线程访问同一个文件对象时所需要的各种锁。

文件对象经由 IoCreateFile 函数来创建，三个系统服务函数 NtCreateFile、NtCreateNamedPipeFile 和 NtCreateMailslotFile 都把创建普通文件对象、命名管道或邮件槽对象的工作交给 IoCreateFile 函数。IoCreateFile 进一步调用 IopCreateFile 函数来创建文件对象。IopCreateFile 函数的代码位于 base\ntos\io\iomgr\iosubs.c 文件中，除了要对传递进来的参数进行各种检查和预处理以外，它并不是简单地调用 ObCreateObject 函数来创建一个 IoFileObjectType 类型的对象，再执行文件对象的初始化工作；相反地，IopCreateFile 函数调用对象管理器的 ObOpenObjectByName 函数来创建文件对象。ObOpenObjectByName 的结果是一个指向所创建对象的句柄，只要 ObOpenObjectByName 返回成功，IopCreateFile 函数即成功返回。

在 2.5.1 节介绍对象管理器时，我们曾经解释过 ObOpenObjectByName 利用对象管理器的另一个函数 ObpLookupObjectName 来打开一个对象，并且也介绍了 ObpLookupObjectName 函数的工作流程。ObpLookupObjectName 从指定的根目录或者系统全局根目录开始，调用 ObpLookupDirectoryEntry 函数，一层一层地进入子目录，直至解析完毕，或者碰到实现了 Parse 方法的对象，从而把余下的路径名称交给该对象进一步解析。

因此，我们可以想象，文件对象是在此递进过程中，由最后一层的子目录对象或者负责解析最后一部分名称串的对象的 Parse 方法来创建的。而且，在此过程中，有可能会生成多个文件对象，这些文件对象形成了上下层的依赖关系，由此也可以理解 FILE_OBJECT 结构中的 RelatedFileObject 成员的意义。为便于理解，我们通过一个例子来说明文件对象的创建过程。在这个例子中，待打开的文件对象的名称是“\Device\MyDevice”，打开文件的过程如图 6.7 所示。

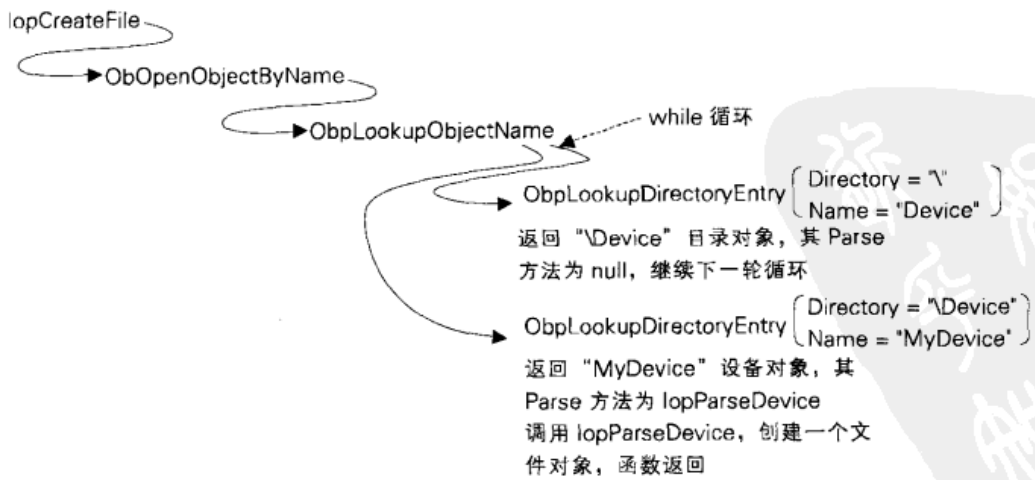


图 6.7 创建文件对象“\Device\MyDevice”的控制流

在 ObpLookupObjectName 函数中，它从对象管理器的根目录，即全局变量 ObpRootDirectoryObject（见 ObInitSystem 函数中的代码）中查找“Device”目录对象。“\Device”目录是系统在阶段 1 初始化过程中调用 CreateSystemRootLink 函数创建的，见 base\ntos\init\initos.c 文件中的代码。目录对象的类型为 ObpDirectoryObjectType，其 Parse 方法，即创建该类型对象时在 OBJECT_TYPE_INITIALIZER 参数中指定的 ParseProcedure 成员为 NULL，所以，在 ObpLookupObjectName 函数中，进入 while 循环的下一次迭代。

ObpLookupObjectName 第二次调用 ObpLookupDirectoryEntry，这一次从“\Device”目录中查找“MyDevice”对象。这次调用得到的是设备对象“MyDevice”，其类型为 IoDeviceObjectType。类型对象 IoDeviceObjectType 是在 I/O 系统初始化时调用 IopCreateObjectTypes 函数创建的，它的 Parse 方法为 IopParseDevice 函数，参见 base\ntos\io\iomgr\ioinit.c 文件中的代码。然后，ObpLookupObjectName 调用设备对象的 Parse 方法，即 IopParseDevice 函数，并将名称“\Device\MyDevice”传递进去。

IopParseDevice 函数调用 ObCreateObject 来创建一个 IoFileObjectType 类型的对象，该文件对象的 RelatedFileObject 成员为 NULL，DeviceObject 成员指向“MyDevice”设备对象。然后，它调用 IoCallDriver 函数，向驱动程序发送一个“IRP_MJ_CREATE”I/O 请求，即通知驱动程序要创建该设备对象的一个文件对象。IopParseDevice 函数的代码位于 base\ntos\io\iomgr\parse.c 文件的 303~1 896 行，这里不再详细解释。

由以上例子可以看出，文件对象的创建是对象管理器与设备驱动程序协作完成的结果。实际上，对象管理器提供的名称目录结构是系统内核名字空间的切入点，该目录中的对象的类型对象所提供的 Parse 例程将此目录结构进一步扩充到设备对象的名字空间中，从而全局目录结构与设备私有的目录结构无缝地结合起来。例如，文件系统驱动程序扩展了内核的目录结构，它们在磁盘卷上构建出延伸的文件名字空间。

文件对象一旦被创建，调用者即获得一个指向该文件对象的句柄，以后，它利用该句柄，通过 I/O 管理器提供的系统服务，比如 NtWriteFile、NtReadFile 等，可读写设备中的数据。关于 I/O 请求的处理过程，参见 6.6 节。

6.2.4 对象生命周期管理

前面两小节介绍了设备对象、驱动程序对象和文件对象的数据结构，以及它们的创建和初始化过程。由于它们都是对象管理器中的对象，所以，这些对象的生命周期都是通过引用计数来管理的。然而，在删除这些对象时，除了要删除它们本身占用的基本存储资源以外，还有些额外的事情要做，下面一一加以说明。

当一个设备对象的引用计数减到 0 时，设备对象类型的删除函数 IopDeleteDevice 被调用，它的代码位于 base\ntos\io\iomgr\objsup.c 文件的 823~873 行。由于已经没有任何客户再引用该设备对象了，所以，IopDeleteDevice 首先调用 IopDestroyDeviceNode 销毁掉相应的设备节点（关于设备节点，参见 6.3.3 节），然后，若有必要，释放卷参数块（VPB）所占用的内存。最后，将对应驱动程序的引用计数也减 1。这是因为，每个设备对象的存在正是驱动程序对象留在内存中的理由之一。所以，只要驱动程序对象还有一个设备对象存在，它就不会被删除。

另一个函数，IoDeleteDevice，也是设备对象的删除函数，但是，它被调用的场合有所不同：当设备驱动程序被卸载时；在驱动程序初始化失败时，要把刚刚创建的设备对象删除掉；或者当对应的设备被移除时。在这些情况下，设备对象的引用计数有可能仍然大于 0，但是它的存在已经没有意义，因此，可调用 IoDeleteDevice 函数来删除设备对象。

在对象管理器中注册的驱动程序对象的删除例程是 IopDeleteDriver。当驱动程序对象的引用计数减到 0 时，对象管理器调用 IopDeleteDriver 函数来清除该驱动程序对象。IopDeleteDriver 函数的代码位于 base\ntos\io\iomgr\objsup.c 文件的 737~821 行，其功能逻辑非常直截了当，只是将驱动程序对象所占用的资源释放掉，并调用 MmUnloadSystemImage 函数卸载驱动程序的映像文件。另一个函数，IoDeleteDriver，只是简单地调用 ObDereferenceObject，将删除驱动程序对象的任务交给对象管理器来完成。

文件对象的删除例程是 IopDeleteFile。这是当文件对象的引用计数减到 0 时被调用的函数，其代码位于 base\ntos\io\iomgr\objsup.c 文件的 434~735 行。它除了要清除文件对象本身所占用的资源以外，还要向与之关联的设备对象发送一个 IRP_MJ_CLOSE 类型的 I/O 请求。然后，若有必要，它会调用 IopDecrementDeviceObjectRef 函数，使关联的设备对象以及文件系统的卷设备对象，引用计数减 1。

6.3 即插即用管理器

即插即用管理器也是 I/O 系统的一部分，它负责在内核中对硬件设备的即插即用提供支持。即插即用管理器为 Windows 支持众多的硬件设备提供了极大的便利，也方便了用户使用各种可插拔的设备。在 Windows 中，即插即用管理器的职责是：自动检测设备的插入和移除，既可能在系统引导时，也可能在系统运行过程中；动态地分配硬件资源，包括中断向量、I/O 端口、I/O 寄存器以及与总线有关的资源，以避免设备之间产生资源冲突；指示 I/O 管理器为设备加载正确的驱动程序，必要时通过一个用户模式的应用程序允许用户指定或搜索驱动程序；向内核及应用程序提供有关设备插入或移除的通知。

即插即用的能力并非操作系统内核单方面的责任，它还涉及硬件设备本身的支持能力和操作系统管理层面的配合。WRK 并没有包含即插即用管理器的代码，但是，作为 I/O 系统中一个重要的组成部分，我们有必要理解即插即用管理器的工作机理，以及它与 I/O 管理器和电源管理器之间的协作过程。

6.3.1 即插即用的基本要求

即插即用框架由 Microsoft 协同 Intel 于 1993 年提出，而后在众多硬件厂商的支持下，逐渐成为各种硬件设备的即插即用工业标准。自 1995 年以后设计和生产的大多数计算机和硬件设备都能够支持即插即用，而 Windows 95 以后的 Windows 操作系统（包括 Windows NT 内核的系统和 Windows 9x 系列）都提供了即插即用的能力。

一个计算机系统的即插即用特性体现在两个方面：第一，当计算机引导时，操作系统可以自动扫描当前系统中所有出现的硬件设备，并且正确地配置这些设备，包括为它们分配资源以及加载适当的驱动程序，当出现资源冲突时，能合理地进行资源仲裁；第二，在系统运行过程中，当设备插入或移除时，操作系统可以检测到硬件设备的变化，并对硬件设备进行配置，增减驱动程序，必要时向系统中正在运行的软件（系统组件或应用程序）发送设备变化通知。硬件设备使用的公共资源主要是中断向量、I/O 端口、I/O 内存，以及其他与总线相关的资源。如果系统中存在不支持即插即用特性的设备，则操作系统必须兼顾这些设备的“硬”需求。

在系统引导阶段，即插即用的支持主要是由计算机的 BIOS（Basic Input Output System）来完成的。系统 BIOS 是即插即用框架结构中的一个重要部件。通常的做法是，BIOS 标识出系统总线（比如 PCI 总线）上的即插即用设备和非即插即用设备，并且获得它们的配置要求，然后判断是否存在资源冲突。如果没有资源冲突，则引导继续进行，否则需要重新分配资源。系统中关于设备的资源配置可以保存在一个称为 ESCD（Extended System Configuration Data）的非易失存储空间中。操作系统利用 BIOS 中的设备资源配置信息，正确地配置设备的驱动程序和相关的软件，从而保证它们能正常地工作。

除了 BIOS 的即插即用支持，系统总线也必须提供即插即用能力，这样才能自动地列举出该总线上附载的设备。总线和设备之间按照约定的规范进行通信。当总线需要列举设备时，支持即插即用的设备能够标识它们自身，并报告它们的资源需求；当系统要进行资源重新分配时，这些设备可以接受新的资源分配。

在操作系统中，即插即用能力主要通过设备的驱动程序来体现。总线的驱动程序和附载在总线上的设备的驱动程序相互协作，完成设备的列举和自动配置工作。例如，在系统

引导之初，操作系统指示总线的驱动程序列举其上的设备，并收集这些设备的配置信息，然后，它利用这些信息，加载并初始化相应的驱动程序。另一方面，当系统中插入或移除一个设备时，总线的驱动程序也要与设备的驱动程序进行通信，以确保设备获得或释放它的 I/O 资源，而且，操作系统可能也需要维护系统状态的一致性，以避免因为设备的动态变化而造成系统不稳定，从而真正做到支持“热插拔”。

6.3.2 Windows 中驱动程序的即插即用支持

在 Windows 中，对硬件设备的操纵是通过设备驱动程序来实现的，而设备的即插即用支持则是由驱动程序对象的 AddDevice 成员和 MajorFunction[IRP_MJ_PNP]分发例程（参见 6.2.2 节关于驱动程序对象 DRIVER_OBJECT 结构的说明）来完成。即插即用管理器通过这两个例程发出各种命令，指示驱动程序控制设备或询问设备的状态。即插即用管理器的命令和硬件设备的状态实际上构成了一个有限状态机，其状态转移如图 6.8 所示。

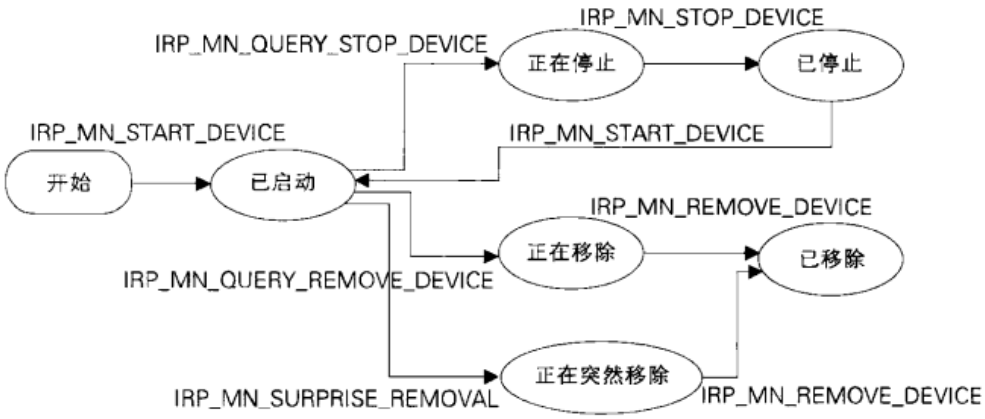


图 6.8 设备的状态转移图

为了指示设备进行状态转移，即插即用管理器向设备发送 I/O 请求，该 I/O 请求的主功能（major function）码为 IRP_MJ_PNP，次功能（minor function）码为图 6.8 中所示的 IRP_MN_<*XXX*>。I/O 管理器将该 I/O 请求分发给驱动程序登记的 MajorFunction[IRP_MJ_PNP]例程，在此例程中，驱动程序根据次功能码即可获知即插即用管理器发送的命令，因而，它可以根据设备当前的状态进行处理。从图中可以看出，当即插即用管理器要停止一个设备时，它首先通过发送 IRP_MN_QUERY_STOP_DEVICE 命令，询问驱动程序是否可以停止该设备，如果驱动程序同意此请求，那么，即插即用管理器再发送一个 IRP_MN_STOP_DEVICE 命令，让驱动程序停止该设备。移除设备的操作也会经历类似的过程，这里不再重复。

关于功能型设备的状态转移过程，有两点值得说明：

1. 除了突然移除或者设备故障的情形以外，设备的停止或移除是一个温和的协商过程，即插即用管理器在接收到用户的停止或移除设备的请求（在 Windows 中，这是通过一个设备管理用户界面程序来做到的）时，会询问驱动程序，让驱动程序有机会完成一些正在等待或尚未完结的 I/O 任务。驱动程序也可以拒绝即插即用管理器的请求，比如，当前还有其他的客户程序正在使用它的设备。

2. 当即插即用管理器需要重新分配一个设备的资源时，它可以按照图中的路径让驱动程序停止该设备，然后，对设备重新配置资源，再向驱动程序发送一个 IRP_MN_START_DEVICE 命令，该命令包含了即插即用管理器的资源分配方案，因而驱动程序可以指示设备在新的 I/O 资源条件下继续工作。

除了图 6.8 中显示的 IRP_MN_*XXX* 命令，即插即用管理器通过 MajorFunction[IRP_MJ_PNP]例程发送给驱动程序的实际命令还有很多，请参考 base\ntos\inc\io.h 文件中的 156~182 行。实际上，从这些命令也可以看出即插即用管理器与设备打交道的协议。值得一提的是，这组命令不仅适用于功能型设备，也适用于总线设备。在一个支持即插即用的设备驱动程序中，MajorFunction[IRP_MJ_PNP]例程往往是一个典型的状态机实现，它根据接收到的 IRP_MN_*XXX* 命令来改变设备的状态。然而，为状态机选择支持哪些命令和哪些状态，则可能随着设备的不同而有所不同。

设备驱动程序的另一个成员 AddDevice 例程是即插即用管理器在接收到总线驱动程序提供的设备描述以后，针对总线上检测到的每个设备而调用的。典型的情形是，当系统引导时，即插即用管理器必须列举（enumerate）出当前系统中所有的硬件设备，为此，它请求总线驱动程序提供有关硬件设备的信息，包括硬件设备的资源要求，然后，即插即用管理器加载这些设备的驱动程序，并调用相应的 AddDevice 例程。下一小节我们讨论此设备列举过程，以及在此过程中建立起来的设备树。

6.3.3 设备列举与设备树

设备列举是在 I/O 系统初始化过程中进行的，I/O 系统创建的第一个设备对象是由驱动程序“\Driver\PnpManager”实现的一个无名设备对象。该设备对象是列举的起始点。HAL 本身也被认为是一个总线设备，在支持 ACPI 的系统中，通过 HAL 设备，可以进一步列举到 ACPI。然后进一步列举到 PCI 总线上的各种设备，包括通过 PCI-ISA 桥连接的设备。如此进行下去，所有具有总线能力的设备都以递归的方式列举它们的设备，直至所有的设备都被列举出来。

在列举过程中，每个节点都是通过一个类型为 DEVICE_NODE 的对象来描述的。随着列举过程的进行，一棵以全局变量 IopRootDeviceNode 为根节点的树被建立起来。根节点代表了驱动程序“\Driver\PnpManager”在列举开始时创建的无名设备对象。每一个总线设备所列举到的设备都挂到该总线设备所在节点的下方，作为它的子节点。显然，DEVICE_NODE 数据结构维护了树的连接信息，以及指向相应设备对象的指针。图 6.9 是在虚拟机环境下一个 ACPI 兼容的 Windows Server 2003 SP1 系统的设备树，它并没有显示出所有的设备节点，这里显示的大多数节点都与驱动程序相对应。因此，对照表 6.1，我们基本上可以理解这些设备节点的含义和用途。从图中我们可以看出，在该系统中，支持即插即用的设备都是通过 PCI 总线和 PCI-ISA 总线列举出来的。另外有两点值得说明：第一，如果在一台物理机器的系统上显示设备树，那么，在 ACPI 节点下面，除了 pci 节点，我们往往还可以看到有关电源、风扇或处理器等设备节点；第二，对于那些不支持即插即用的设备，它们的设备节点直接挂在根节点下。

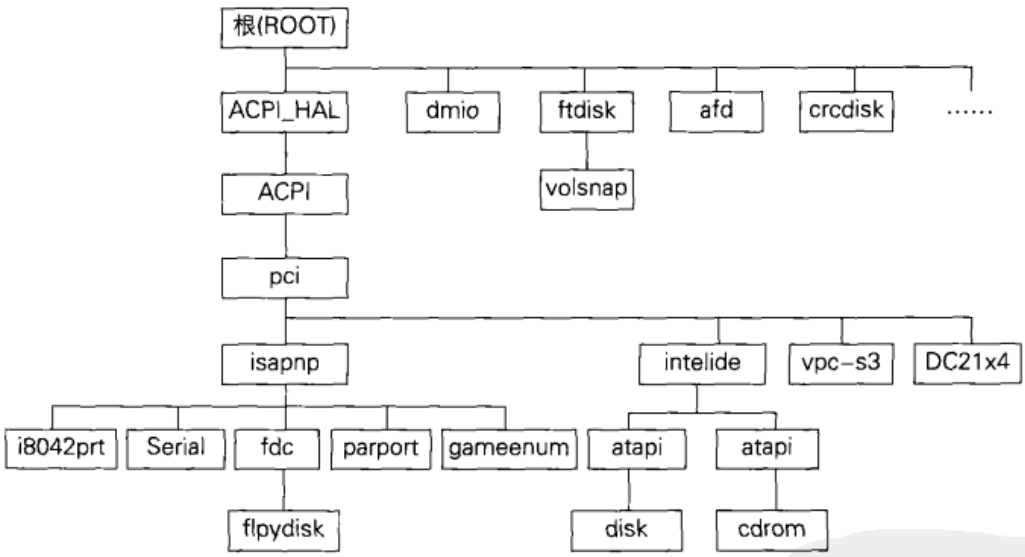


图 6.9 设备树示意图

硬件总线设备有能力列举其从属的设备，这是设备列举的关键所在，此外，注册表中的信息也起到了重要的作用。例如，根节点作为一个虚拟设备节点，利用注册表中的信息来构造它的子节点。这些信息保存在 HKLM\SYSTEM\CurrentControlSet\Enum\Root 键下面的子键中。在列举过程中，根节点驱动程序对每个子键都会创建一个设备对象和设备节点。注册表中的信息是如何来的呢？这是在安装操作系统时由安装程序提供的，或者在安装某些驱动程序时加入进来的。

结合注册表中的信息，即插即用管理器在列举设备的过程中，不仅会建立起一棵完整

的设备树，而且也会把相关的驱动程序加载到系统中并执行初始化。正如 6.2.1 节所介绍，I/O 系统在初始化时，首先加载并初始化“引导-启动”类型的驱动程序，这是由 IopInitializeBootDrivers 函数针对每个这种类型的驱动程序调用 IopInitializeBuiltinDriver 函数来实现的。IopInitializeBootDrivers 同时也针对每个驱动程序的设备执行列举操作。类似地，当 I/O 系统在加载并初始化“系统-启动”的驱动程序时，它也会执行设备列举操作，这发生在 IopInitializeSystemDrivers 函数中。由此可见，I/O 系统在初始化过程中已经建立了设备树。

当有设备变化，比如检测到新的设备等情况发生时，驱动程序可以调用 I/O 系统提供的一些函数重新列举设备或者分配资源，这样的 I/O API 函数包括 IoInvalidate-DeviceRelations、IoSynchronousInvalidateDeviceRelations、IoReportDetectedDevice 和 IoInvalidateDeviceState 等。

设备树中的每个设备节点都有一个名称，称为实例路径（InstancePath），其形式为 *<Enumerator>\<DeviceID>\<InstanceID>*，这里 Enumerator 是总线驱动程序的名称，DeviceID 是设备的唯一标识，而 InstanceID 则唯一标识了一个硬件设备的实例。注册表键 HKLM\SYSTEM\CurrentControlSet\Enum 下面正好以 InstancePath 的形式存放了一个设备节点的信息，这也正是通过 HKLM\SYSTEM\CurrentControlSet\Enum\Root 键可以列举出根节点下面的所有子节点的原因。

在每个设备节点上都必定有一个设备对象，称为 PDO（Physical Device Object），这是在列举过程中由总线驱动程序创建的设备对象。为了管理该设备，即插即用管理器加载一个功能驱动程序，并调用它的 AddDevice 例程，同时将 PDO 传递给该例程。此 AddDevice 例程通常会创建一个功能设备对象（FDO，Functional Device Object），并让该 FDO 附载在 PDO 之上，因此，这两个对象构成了一个基本的设备栈，如 6.2.2 节所述。

对于一个设备而言，PDO 相当于它的物理接口，而 FDO 则相当于它的逻辑接口。应用程序或内核向一个设备发送的 I/O 请求，首先到达 FDO，如果 FDO 的驱动程序能够直接处理，则不必传递给 PDO；否则，I/O 管理器将此请求传递给 PDO。另外值得一提的是，在设备节点的设备栈中，除了 PDO 和 FDO，可能还会包含一个或多个起过滤作用的设备对象，它们是由过滤驱动程序创建的，本章 6.5.5 节将介绍过滤驱动程序的工作机理。

以上介绍了即插即用管理器的设备列举和设备树。即插即用管理器在设备列举过程中需要为设备加载驱动程序，它如何找到对应的驱动程序，以及驱动程序在安装时应提供哪些信息，关于这些问题，本书不再进一步介绍，读者可参考相关资料[WIN-INTERNALS]。

6.4 电源管理器

在现代计算机系统中，电源管理正变得越来越重要，无论是便携式计算机，还是台式机或 PC 服务器，有效的电源管理都既可以起到节能的效果，也可以延长机器或设备的使用周期。电源管理不仅仅是操作系统单方面的职责，也需要硬件提供相应的支持，包括 BIOS 和硬件设备。电源管理的工业标准是 APM（Advanced Power Management，高级电源管理）和 ACPI（Advanced Configuration and Power Interface，高级配置和电源接口）。APM 是一个早期的标准，目前普遍采用和支持的标准是 ACPI。ACPI 标准最早由 Intel、Microsoft 和 Toshiba 发起并制定，自 1998 年以后生产的绝大多数 Intel x86 计算机均支持 ACPI。有关 ACPI 规范，可通过 *http://www.acpi.info/*获得。

6.4.1 电源管理概述

ACPI 涉及计算机系统中电源管理的各个方面，包括系统电源管理、设备电源管理、处理器电源管理、设备和处理器的性能管理、系统配置/即插即用、系统事件机制、电池管理、温度管理、内嵌的控制器，以及系统管理总线控制器（System Management Bus Controller，简称 SMBus 控制器）。这一节我们讨论系统电源状态、设备电源状态和处理器电源状态，以及这些状态之间的转移。

系统电源状态是指整个系统的电源状态，对于用户是可见的。划分系统电源状态的依据是：应用软件是否可以运行？应用软件响应外部事件的延迟如何？电源消耗如何？是否需要经过一次引导过程才能回到工作状态？是否可以安全地拆卸机器？系统电源状态可以被标记为 G*x* 状态，以下是 G*x* 状态的定义：

- G3，机械关闭（Mechanical Off）。机械方式关闭一台机器，必须经过重启以后操作系统才能回到工作状态，除了系统时钟，其他无电源消耗。
- G2/S5，软件关闭（Soft Off）。机器消耗最小的电源，无代码运行。此状态下，必须经过重启以后操作系统才能回到工作状态。拆卸硬件是不安全的。
- G1，睡眠（Sleeping）。机器消耗很小的电源，应用代码不运行，系统看起来像是关闭的，譬如，无屏幕输出。回到工作状态的延迟可长可短。无须经过重启，操作系统就可以工作，因为大部分系统环境已完好保存。拆卸硬件是不安全的。
- G0，工作（Working）。应用代码可以运行，在此系统状态下，外部设备的电源状态可以动态变化。用户可以选择系统的电源方案，以便优化电池的有效使用时间。系统可以实时地响应外部事件。同样地，拆卸硬件是不安全的。

其中 G1 状态又可以进一步细分为 S1~S4 共 4 种不同程度的睡眠状态：

- S1 睡眠状态。唤醒此状态所经历的延迟较短。在此状态下，系统运行环境并未丢失，硬件维护了运行环境中的信息。
- S2 睡眠状态。唤醒此状态所经历的延迟较短。此状态与 S1 类似，但是，系统处理器和缓存（cache）中的信息已丢失，操作系统维护此信息。
- S3 睡眠状态。唤醒此状态所经历的延迟不长。在这种状态下，除了系统内存中存放的信息以外，其他的系统环境信息，包括处理器、缓存等，均已丢失。硬件维护着系统内存，并且负责恢复其他的运行环境信息。
- S4 睡眠状态。唤醒此状态所经历的延迟相对最长，但是它消耗的电源最少。为了尽可能地减少电源消耗，所有的设备均已关闭电源。系统将运行环境保存在非易失的存储体中。

这 4 种睡眠状态，睡眠的程度由浅入深，消耗的电源由多到少，唤醒所经历的延迟由短到长。如果将 G0 看做 S0，G2 和 G3 看做 S5，那么，系统电源状态可以用 S0~S5 来表示，这样完全从系统的睡眠程度来划分其电源状态。对于 S4 和 S5 状态，为了唤醒机器进入工作状态，系统需要经历一次完整的引导过程；然而，两者的区别是，在经过 BIOS 引导时，是否此次引导过程要从一个原先保存下来的内存映像中恢复系统的运行环境。

设备的电源状态与系统电源状态不同，它通常对于用户是不可见的，例如，当系统正在运行时，设备可能已关闭。ACPI 从设备的电源消耗、设备运行环境的维护程度、设备驱动程序如何恢复设备工作状态，以及恢复工作状态所需时间长短这 4 个角度来定义 4 种设备状态 D0~D3：

- D0，完全工作状态。设备可以完全响应各种请求，并且可以维护所有的执行环境。
- D1，设备状态。由每种设备类定义，通常而言，会省一点电，维护一部分执行环境信息。
- D2，设备状态。由每种设备类定义，通常而言，比 D0 和 D1 状态节省更多的电，但维护更少的执行环境信息。处于 D2 状态的总线可能会导致设备丢失一些执行环境信息，例如，降低总线的功率以后，总线上的设备被迫停掉一些功能。
- D3，关闭状态。设备完全停电。进入该状态以后，设备的执行环境信息完全丢失；当重新加电时，操作系统将会重新初始化该设备。处于此状态的设备，其恢复时间最长。

在这 4 种设备状态中，D0 和 D3 对于所有设备都有明确定义，而 D1 和 D2 则由设备类所决定。事实上，许多设备类并未定义这两种中间状态。由于总线和设备之间存在从属关系，总线的电源状态不能弱于其设备的最高状态（假设 D0 最强，D3 最弱）。ACPI 规范定义了音频设备、COM 端口设备、显示器设备、输入设备（包括键盘、鼠标、游戏操纵杆等）、调制解调器、网络设备和存储设备等各类设备的电源状态规范。

ACPI 也为 G0 状态的系统定义了处理器的电源状态，以反映出处理器的电源消耗和热量管理状态。处理器的电源状态有 4 种，分别为 C0~C4，其定义如下：

- C0，处理器正在执行指令。
- C1，处理器不在执行状态，但是它可以以最短的延迟进入执行状态。软件可以忽略此延迟。
- C2，比 C1 状态更加节能，除了处理器不在执行状态以外，对软件没有其他的影响。关于最坏情况下硬件延迟的信息应该由 ACPI 固件提供，因而操作系统可以根据此信息来决定应该将处理器置于 C1 还是 C2 状态。
- C3，比 C1 和 C2 状态更加节能。此状态中的处理器仍然维护缓存中的数据，但不再保证缓存一致性，因而缓存一致性需要操作系统以软件方式来维护。在此状态下，关于最坏情况下硬件延迟的信息应该由 ACPI 固件提供，因而操作系统可以根据此信息来决定应该将处理器置于 C2 还是 C3 状态。

ACPI 规定，C0 和 C1 是处理器必须支持的状态，而 C2 和 C3 是可选的状态。在 Intel x86 处理器上，利用 hlt 指令可进入 C1 状态。在 C1~C3 状态中，一旦有中断发生，则处理器将立即退出该状态，而进入 C0 状态。

在介绍了系统电源状态、设备电源状态和处理器电源状态的规范以后，我们再来看一看这些状态之间的关系，如图 6.10 所示。

若不考虑计算机关机的情形，则计算机总是在工作状态（G0/S0）和睡眠状态（G1）之间切换。在工作状态下，计算机在从事各种计算任务：应用程序在运行，操作系统也在运行，计算机维护了当前运行系统的执行环境。此时，设备和处理器有可能根据系统的工作状态、负荷等的不同而处于不同的状态，但对于用户而言，整个系统正在运行。即使某个设备已经关闭（状态 D3），只要有任务需要它完成，系统就会让它进入工作状态（D0），只是可能会有一段明显的延迟（设备需要重新初始化）。因此，虽然系统处于工作状态，但是由于设备和处理器所处的子状态不同，计算速度、电源的消耗、所产生的热量和噪声也会有所不同。操作系统中的电源管理组件有责任在这些因素上做好平衡。

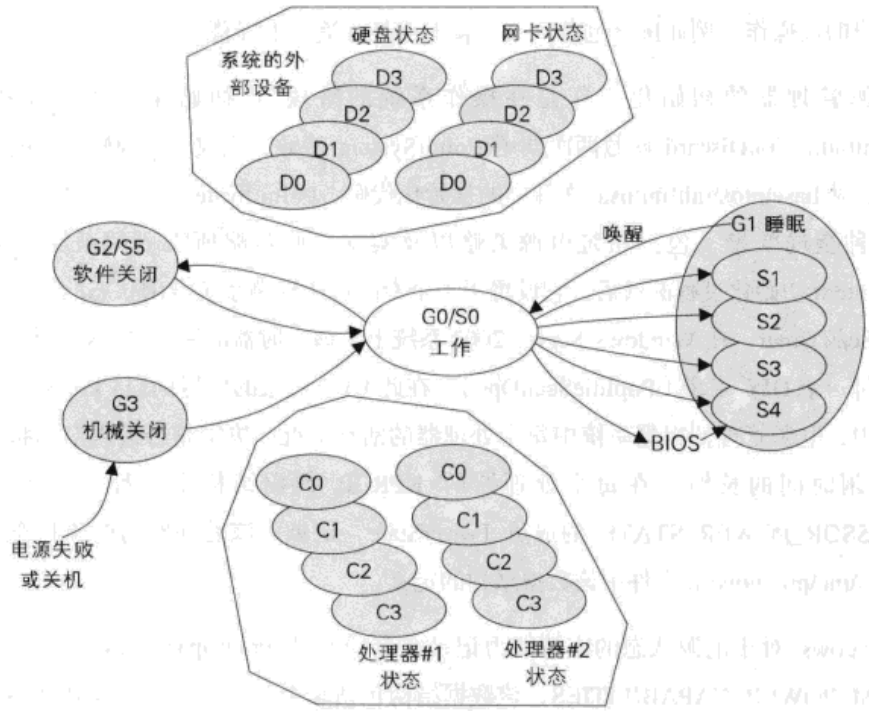


图 6.10 电源状态转移图

另一方面，当计算机进入睡眠状态时，在用户看来，无任何计算行为在进行。不同睡眠状态（S1~S5）的区别在于，以何种方式让系统回到工作状态，以及回到工作状态所需的时间。节能越多，维护执行环境的电源消耗越少，但恢复到工作状态所需的时间越长。在睡眠状态 S4，用户需要手工按电源开关才能恢复系统的运行，系统的执行环境被保存在非易失的存储体中，恢复过程涉及 BIOS 操作。S4 状态几乎不需要电源，睡眠时间可以长达数年，而恢复系统只需要分钟量级的时间。

6.4.2 Windows 中的电源管理

在 Windows 中，电源管理是由 I/O 系统的电源管理器和设备驱动程序配合完成的。电源管理器决定了系统所处的电源状态，以及何时睡眠、休眠（hibernation）或停机，它也通过设备驱动程序指示那些支持多种电源状态的设备，执行适当的电源状态转移。电源管理器根据以下一些条件来作出电源状态转移的决定：

- 系统的活动情况；
- 系统的电池使用情况；
- 用户的软操作，如停机、休眠或睡眠动作；

- 用户的硬操作，例如按下电源开关、合上或打开笔记本屏幕。

电源管理器的初始化工作是在操作系统的阶段 1 初始化过程中完成的，PhaseIInitializationDiscard 函数两次调用 PoInitSystem 函数，完成了电源管理器的两阶段初始化，见 base\ntos\init\initos.c 文件中的初始化代码。PoInitSystem 初始化电源管理器使用的各种全局变量，包括系统电源策略以及实施这些策略所需要的系统对象。在 PoInitSystem 初始化过程的最后，它设置了一个专门用于检测系统空闲状态的定时器对象 PopIdleScanTimer，在 Windows Server 2003 系统上，该定时器的周期是 1 s，当定时器到期时交付一个 DPC 对象（PopIdleScanDpc）。在此 DPC 对象的延迟例程（PopScanIdleList 函数）中，电源管理器根据系统中每个处理器的活动情况，决定是否进入空闲状态，并跟踪空闲时间的长短。在每个处理器的 KPRCB 数据结构中，都有一个类型为 PROCESSOR_POWER_STATE 的成员 PowerState，记录了该处理器的电源状态，参见 base\ntos\inc\procpowr.h 文件中该数据结构的定义。

Windows 对于电源状态的支持能力记录在全局变量 PopCapabilities 中，其类型为 SYSTEM_POWER_CAPABILITIES，该数据结构包括是否支持休眠，以及处理器、磁盘和电池等硬件组件的电源支持信息。应用程序（比如电源管理程序）可通过 NtPowerInformation 系统服务获得或设置 PopCapabilities 中的成员值。另一个相关联的全局变量是 PopPolicy，其类型为 SYSTEM_POWER_POLICY，它记录了系统中有关唤醒、空闲、睡眠、电池的策略方面的设置。关于这两个数据结构，参见 public\sdk\inc\ntpoapi.h 文件中的 C 结构定义。

当即插即用管理器执行设备列举时，在它所接收到的关于设备能力的信息中，其中有关于设备电源的信息。即插即用管理器通过向设备驱动程序发送一个 MajorFunction 为 IRP_MJ_PNP、MinorFunction 为 IRP_MN_QUERY_CAPABILITIES 的 I/O 请求，来获得设备能力信息。设备能力由 DEVICE_CAPABILITIES 结构来描述，其中与电源管理有关的信息如下所列：

```
typedef struct _DEVICE_CAPABILITIES {

    ……
    ULONG DeviceD1:1;                       // 是否支持 D1 状态
    ULONG DeviceD2:1;                       // 是否支持 D2 状态
    ……

    ULONG WakeFromD0:1;                     // 是否可以从 D0 状态唤醒系统
    ULONG WakeFromD1:1;                     // 是否可以从 D1 状态唤醒系统
    ULONG WakeFromD2:1;                     // 是否可以从 D2 状态唤醒系统
    ULONG WakeFromD3:1;                     // 是否可以从 D3 状态唤醒系统
    ULONG WarmEjectSupported:1;             // 是否支持热拔除
```

```
    ……

    DEVICE_POWER_STATE DeviceState[POWER_SYSTEM_MAXIMUM];    // 状态映射表
    SYSTEM_POWER_STATE SystemWake;            // 系统唤醒状态
    DEVICE_POWER_STATE DeviceWake;            // 设备唤醒状态
    ULONG D1Latency;                          // 从 D1 转移到 D0 状态的延迟
    ULONG D2Latency;                          // 从 D2 转移到 D0 状态的延迟
    ULONG D3Latency;                          // 从 D3 转移到 D0 状态的延迟

} DEVICE_CAPABILITIES, *PDEVICE_CAPABILITIES;
```

在设备能力数据结构中，DeviceState 数组描述了从系统电源状态到设备电源状态的映射关系，即针对每一种系统状态，设备的最低可能电源状态。也就是说，当系统睡眠或休眠时，设备可以处于哪一级状态。如果设备不支持 D1 和 D2 状态，那么，一旦系统离开 S0 状态，设备就会完全停止（状态 D3），因而不再消耗电源。然而，有些设备，比如服务器系统上的网卡适配器，支持“唤醒一个正在睡眠的系统”的能力，这样的能力正是在设备列举过程中报告的。例如，设备驱动程序通过下面的代码可以做到这一点：

```
deviceCapabilities->DeviceState[PowerSystemWorking] = PowerDeviceD0;   // S0->D0
deviceCapabilities->DeviceState[PowerSystemSleeping1] = PowerDeviceD1; // S1->D1
deviceCapabilities->DeviceState[PowerSystemSleeping2] = PowerDeviceD3; // S2->D3
deviceCapabilities->DeviceState[PowerSystemSleeping3] = PowerDeviceD3; // S3->D3
deviceCapabilities->DeviceState[PowerSystemHibernate] = PowerDeviceD3; // S4->D3
deviceCapabilities->DeviceState[PowerSystemShutdown] = PowerDeviceD3;  // S5->D3

// 从设备 D1 状态可以唤醒系统
deviceCapabilities->DeviceWake = PowerDeviceD1;

deviceCapabilities->DeviceD1 = TRUE;        // 可以从 D1 状态唤醒系统
deviceCapabilities->DeviceD2 = FALSE;       // 设备没有 D2 状态

// 当设备处于 D0、D1、D2 和 D3 状态时，设备是否可以响应外部信号
deviceCapabilities->WakeFromD0 = FALSE;
deviceCapabilities->WakeFromD1 = TRUE; // 可以
deviceCapabilities->WakeFromD2 = FALSE;
deviceCapabilities->WakeFromD3 = FALSE;
```

DEVICE_CAPABILITIES 中的 DeviceWake 域是指设备至少处于哪个电源状态可以唤醒当前的系统，SystemWake 域是指系统至少处于哪个电源状态时才可以被唤醒。这两者之间的关系是，SystemWake 状态所对应的设备电源状态，即 DeviceState[SystemWake]，数值上不能超过 DeviceWake。也就是说，当通过设备唤醒系统时，此时系统的供电应该足够它醒过来。驱动程序可以不为设备指定 SystemWake，即插即用管理器能够根据 DeviceWake 值和 DeviceState 数组推断出正确的 SystemWake。

电源管理器除了维护系统的电源状态，也通过设备驱动程序控制设备的电源状态。在 Windows 中，电源管理器通过驱动程序获得有关设备的电源状态支持，驱动程序在 MajorFunction[IRP_MJ_POWER]例程中接收电源管理器的电源命令，包括以下 4 个子命令：

```
#define IRP_MN_WAIT_WAKE                    0x00
#define IRP_MN_POWER_SEQUENCE               0x01
#define IRP_MN_SET_POWER                    0x02
#define IRP_MN_QUERY_POWER                  0x03
```

在前面 6.3 节我们曾经提到，一个设备可能有多个驱动程序为它服务，尽管它们都可以响应针对该设备的 IRP_MJ_POWER 请求，但只有一个驱动程序（往往是该设备的功能驱动程序）是设备的电源策略所有者（device power policy owner）。设备的电源管理任务涉及多个方面，包括报告设备的电源能力（如前所述，这发生在即插即用管理器列举设备时）、处理电源 I/O 请求、控制设备的电源状态等。然而，这些电源管理任务并非都由电源策略所有者驱动程序来承担。譬如，通常的做法是，总线驱动程序报告设备的电源能力，而功能驱动程序发出请求让设备睡眠或醒来。

参考图 6.8 中即插即用管理器与设备驱动程序之间的交互关系，当驱动程序接收并处理 IRP_MJ_PNP 请求的 IRP_MN_START_DEVICE 和 IRP_MN_REMOVE_DEVICE 子命令时，它也会为设备加电或断电。在处理 IRP_MN_START_DEVICE 请求时，驱动程序打开设备的电源，并调用 PoSetPowerState 以通知电源管理器设备状态变成 PowerDeviceD0；而在处理 IRP_MN_REMOVE_DEVICE 请求时，驱动程序关闭设备的电源，并调用 PoSetPowerState 以通知电源管理器设备状态变成 PowerDeviceD3。所以，设备初始时的电源启动和移除时的电源关闭是由即插即用管理器与设备驱动程序协作来完成的。

在其他情况下，电源管理器利用 IRP_MJ_POWER 请求，与设备驱动程序进行通信，以实现系统和设备的电源状态转移。IRP_MJ_POWER 请求的 4 个子命令的含义和用法如下所述：

- IRP_MN_WAIT_WAKE，此命令对于那些有能力唤醒一个正在睡眠的系统，或者支持从睡眠状态中醒来的设备有意义。该命令应该由设备的 PDO（物理设备对象）来处理，它允许设备响应外部信号，比如网卡适配器可接收外来的唤醒信号。驱动程序可以在设备的 D0 状态发送此命令，它本身并不改变设备的电源状态。IRP_MN_WAIT_WAKE 的 I/O 请求直到外部信号到达或者发送者取消该请求以后才完成，而且，针对一个设备，同一时刻只能处理一个这样的 I/O 请求，后续的 IRP_MN_WAIT_WAKE 请求应该失败。因此，发送者在该 I/O 请求成功完成以后，可以通过 IRP_MN_SET_POWER 命令将设备的电源状态转移到 D0。
- IRP_MN_POWER_SEQUENCE，驱动程序通过发送该命令的 I/O 请求，以获得一个设备的电源序列值。这里的电源序列值，是指设备分别处于 D1（或更低）、D2（或更低）和 D3 状态的次数。通常总线驱动程序可能会（可选）处理此 I/O 请求，而功

能驱动程序或过滤驱动程序（本章后面介绍）可能会发送这样的 I/O 请求。设备的电源策略所有者驱动程序可以利用电源序列值信息做一些优化。

- IRP_MN_SET_POWER，通知驱动程序改变系统电源状态，或者设置设备的电源状态。电源管理器发送此命令的 I/O 请求给驱动程序，以通知它们系统电源状态的变化；对于那些已经注册了要执行设备空闲检测的驱动程序，它也会发送此命令的 I/O 请求以改变空闲设备的电源状态。设备的电源策略所有者驱动程序发送此命令的 I/O 请求以设置设备的电源状态。

  对于一个改变系统电源状态的 I/O 请求，它在设备栈的总线驱动程序中完成；而在设备的电源策略所有者驱动程序中，由它的完成例程发送一个改变设备电源状态的 IRP_MN_SET_POWER 命令。

  对于一个改变设备电源状态的 I/O 请求，它也在总线驱动程序中完成。总线驱动程序调用 PoSetPowerState 以通知电源管理器该设备的电源状态变化，而且，它也会根据需要，在进入睡眠或休眠前保存必要的环境信息以便将来可以恢复到工作状态。

- IRP_MN_QUERY_POWER，电源管理器或者设备驱动程序询问一个设备是否可以转移到指定的系统电源状态或设备电源状态。通常的用法是，询问是否可以进入睡眠状态。然而，有些情况下，电源管理器在发出 IRP_MN_SET_POWER 命令前并不发送询问请求。比如，当用户按下电源开关或电池快要耗尽时。对于询问系统电源状态转移的请求，总线驱动程序完成此 I/O 请求，电源策略所有者驱动程序应该在它的完成例程中进一步发出针对设备电源状态转移的询问请求。驱动程序可以让一个 IRP_MN_QUERY_POWER 请求失败，但这并不保证电源管理器就不会紧接着发出一个转移到所询问电源状态的 IRP_MN_SET_POWER 请求，所以，驱动程序在接收到 IRP_MN_QUERY_POWER 请求以后，总是应该做好随后接收 IRP_MN_SET_POWER 请求的准备。

在 Windows Server 2003 中，IRP_MN_POWER_SEQUENCE 命令应该通过 PoCallDriver 函数来发送，另 3 个命令应该通过 PoRequestPowerIrp 函数来发送。PoCallDriver 函数要求调用者自己准备一个 I/O 请求数据结构（即 IRP 对象，见 6.6.1 节介绍），而后者无须调用者申请 IRP 对象。

最后，我们从设备栈的角度来总结一下电源管理器发出的系统电源 I/O 请求（IRP_MN_QUERY_POWER 和 IRP_MN_SET_POWER）是如何被处理的。如图 6.11 所示，电源管理器发出的询问或设置系统电源状态的 I/O 请求，沿着设备栈逐层处理。当到

达该设备的电源策略所有者驱动程序时，它设置一个 I/O 完成例程，然后继续传递该 I/O 请求；当到达总线驱动程序时，该 I/O 请求完成，然后通过 I/O 管理器逐层返回。电源策略所有者驱动程序设置的完成例程有机会得以执行，它调用 PoRequestPowerIrp 函数，发送一个针对设备电源状态的 I/O 请求，同样地，该 I/O 请求也被沿着设备栈传递和处理。注意，图中所示的两个设备栈实际上是同一个。由此我们可以看出，系统电源状态的 I/O 请求和设备电源状态的 I/O 请求相互协作以完成系统电源状态的转移。设备驱动程序在接收到这些 I/O 请求时应恰当地完成自己的职责。

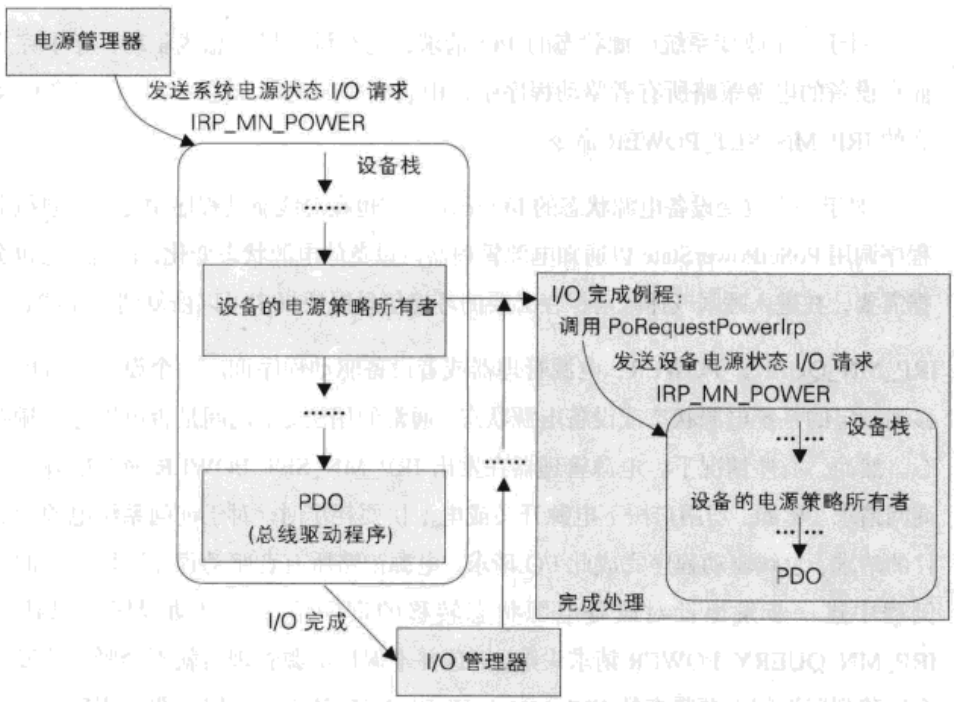


图 6.11 系统电源状态 I/O 请求的处理路径

Windows 为系统和设备的电源管理定义了一个完整的框架，本节只是给出了该框架的一个简略说明。关于电源管理的更详细信息，请参考 Windows DDK 的文档说明[MSDN-PO]。

6.5 设备驱动程序

设备驱动程序是操纵设备的内核模块，I/O 管理器、即插即用管理器和电源管理器都需要与设备的驱动程序打交道。在 6.2.2 节，我们已经看到了内核中设备驱动程序的对象表示，这一节我们首先讨论 Windows 所支持的设备驱动程序的分类，然后通过例子来说明驱动程序的代码结构以及各种过滤驱动程序，最后介绍一下非即插即用驱动程序。

6.5.1 设备驱动程序分类

在 Windows I/O 系统中，设备驱动程序不仅为操作系统提供了支持各种 I/O 设备的能力，也是 Windows 内核本身扩展的基础。Windows 可以动态地加载或卸载设备驱动程序，通过这些驱动程序来调整或扩展内核的功能。Windows I/O 系统规定了设备驱动程序应遵循的接口，这组接口是通用的，可适用于所有的内核模式驱动程序。设备驱动程序依据其用途不同，可以分为以下三类：

- **即插即用驱动程序**，也称为 WDM 驱动程序。它们通常是为了驱动硬件设备而由硬件厂商提供，与 Windows 的 I/O 管理器、即插即用管理器和电源管理器一起工作。Windows 自身携带了大量即插即用驱动程序，用于支持各种常见的存储设备、视频适配器、网络适配器以及输入设备等。

- **内核扩展驱动程序**，也称为非即插即用驱动程序。它们扩展内核的功能，或者提供了访问内核模式代码和数据的一种途径。它们并没有集成到即插即用管理器和电源管理器的管理框架中。早期的 Windows NT 驱动程序（在引入即插即用管理机制以前）都属于这一类型，现在仍然有大量的内核扩展驱动程序。本书前几章展示的工具都通过这种类型的驱动程序来获得内核中的信息。

- **文件系统驱动程序**。它们接收针对文件的 I/O 请求，再进一步将这些请求转变成真正对于存储设备或网络设备的 I/O 请求，从而满足客户的原始请求。在 6.2.1 节我们曾经看到过，文件系统类型的驱动程序被放在对象管理器的\FileSystem 目录下，其他的驱动程序放在\Driver 目录下。表 6.2 中列出的驱动程序都是文件系统驱动程序。

Windows 驱动程序模型（WDM）在 I/O 模型中增加了对即插即用、电源管理和 Windows 管理规范（WMI）的支持。而且，遵从 WDM 的驱动程序在 Windows 平台（包括 Windows 98/Windows 2000 以后的所有版本）上至少是源代码兼容的，甚至也可能是二进制兼容的。WDM 驱动程序又进一步可以划分成以下三类：

- **总线驱动程序**。顾名思义，总线驱动程序管理一个总线设备，它负责检测总线上附载的所有设备，并通知即插即用管理器关于这些设备的情况。总线驱动程序也负责总线的电源管理。

- **功能驱动程序**。功能驱动程序管理具体的设备，在一个设备的设备栈中，功能驱动程序创建的设备对象（即 FDO）相当于操作系统控制该设备的逻辑接口。功能驱动程序是实际管理该设备的功能模块。

- **过滤驱动程序**。在设备栈中，过滤驱动程序位于功能驱动程序之上或之下，它的用途是：监视一个设备的 I/O 请求以及这些请求的处理情况，或者，增加或改变一个设备或另一个驱动程序的行为。例如，病毒扫描工具常常利用过滤驱动程序来监视被读写的文件数据。

在 WDM 中，总线是指可供其他设备附载的设备，其中既有像 PCI 和 SCSI 这样的物理总线设备，也有像 HAL 这样的虚拟总线设备。总线驱动程序负责检测总线上的设备，并且协助即插即用管理器列举这些设备，而且它也控制该总线的物理配置。相反地，功能驱动程序要简单得多，通常只是控制一个设备的硬件而已。

在即插即用管理器列举得到的设备树中，每个设备节点都包含一个设备栈，设备栈中的各个设备对象分别由对应的驱动程序创建和实现。图 6.12 显示了不同类型 WDM 驱动程序在设备栈中的角色，以及它们之间的关系。

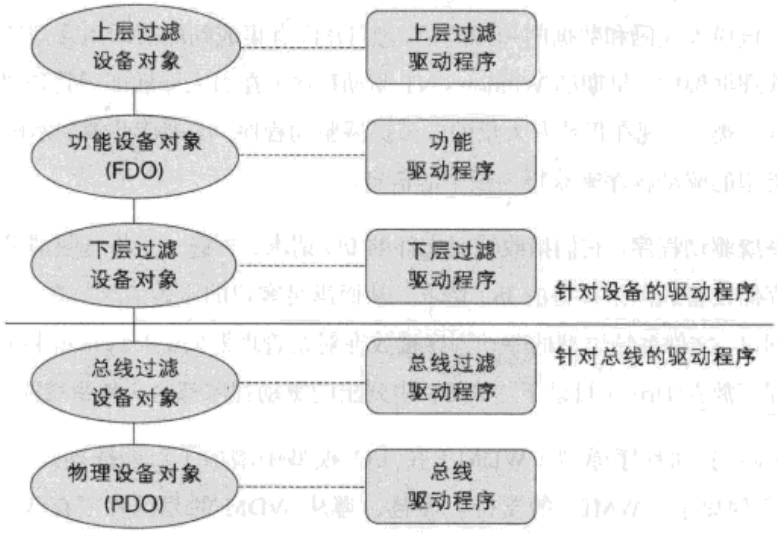


图 6.12 WDM 驱动程序与设备栈

每个设备对象都是由对应的驱动程序创建的，设备的 PDO 是由总线驱动程序创建的，而 FDO 是由功能驱动程序创建的。在 PDO 和 FDO 之间，紧挨着 PDO，可以有零个或多个总线过滤设备对象，它们由相应的总线过滤驱动程序创建。在总线过滤设备对象和 FDO 之间，可以有零个或多个下层过滤设备对象。而在 FDO 之上，则可以有零个或多个上层过滤设备对象。这些下层或上层过滤设备对象分别由相应的下层或上层过滤驱动程序创建。WDM 规定了这样的设备栈结构，既合理地让参与其中的每个驱动程序实现它所负责的那部分功能，也提供了足够的扩展性，允许过滤驱动程序在不同层次上监视或修正一个设备的工作方式。后面 6.5.4 节将介绍过滤驱动程序的加载以及过滤设备对象的创建顺序。

对于一个具体的设备而言，FDO 代表了它对于操作系统的逻辑接口。功能驱动程序往往会创建一个代表相应 PDO 的设备接口（通过 I/O 管理器函数 IoRegisterDeviceInterface），因而应用程序或内核其他部分可以通过此设备接口与该设备打交道。对于一些复杂而又通用的设备，例如磁盘设备和网络适配器等，功能驱动程序又被进一步分成多个独立的驱动程序，由它们联合起来管理该 FDO 的 I/O 请求。从这一层意义上，WDM 驱动程序对硬件的支持可以进一步划分为类驱动程序（class driver）和端口驱动程序（port driver），以及小端口驱动程序（miniport driver）。**类驱动程序**实现了某一种类型的设备的 I/O 处理。对于已经标准化的设备类型，比如磁盘、网络适配器等，提供一个类驱动程序可以为不同厂商生产的设备实现通用的服务。**端口驱动程序**实现了与某一类型 I/O 端口相关的 I/O 处理，它们并不遵从驱动程序的接口要求，而只是一些内核模式的辅助例程。**小端口驱动程序**则实现了驱动某一特定设备而需要的 I/O 服务。类驱动程序和小端口驱动程序的分工可以看成是针对一个或一类设备的“通用”和“特殊”的功能部分。类驱动程序实现的是“通用”或“公共”的 I/O 服务，而小端口驱动程序实现的是针对某一特定设备的“特殊”功能部分。在 7.3.1 节介绍存储栈结构时，我们将会看到这两种驱动程序的例子，这里不再进一步介绍。

6.5.2 例子驱动程序 toaster

Windows DDK 提供了大量的驱动程序例子，几乎遍及 Windows 操作系统对设备支持的方方面面。其中 toaster 是一个适合于学习 WDM 驱动程序开发的范例[TOASTER]。这一节介绍 toaster 例子的组成部分，以及该例中涉及的一些概念和用法。读者可以编译并安装 toaster 例子中的驱动程序，以加强对 Windows I/O 系统的认识。

Toaster 并非单个设备驱动程序，相反地，它在 Windows 系统中建立一条虚拟总线，并提供一个总线驱动程序、一个功能驱动程序、一组过滤驱动程序，以及一些辅助工具。Toaster 例子的代码和数据位于 Windows DDK 例子程序安装目录的 src\general\toaster 子目录下。表 6.3 列出了 toaster 中的驱动程序以及它们的代码所在的子目录。

表 6.3 toaster 例子中的驱动程序

驱动程序	代码目录（相对于 toaster 例子目录）	说明
BusEnum.sys	bus	toaster 总线驱动程序
toaster.sys	func\incomplete1 func\incomplete2 func\featured1 func\featured2	toaster 设备的标准功能驱动程序，例子中包含 4 个版本的 toaster.sys，演示了在功能驱动程序中逐步增加功能的过程。featured1 是一个完整的版本，支持电源管理和 WMI，featured2 进一步支持等待-唤醒和事件追踪

续表

驱动程序	代码目录（相对于 toaster 例子目录）	说明
clsupper.sys clslower.sys devupper.sys devlower.sys bfdupper.sys bfdlower.sys	filter	这是 6 个过滤驱动程序，分别为类上层过滤驱动程序、类下层过滤驱动程序、设备上层过滤驱动程序、设备下层过滤驱动程序、总线 FDO 上层过滤驱动程序和总线 FDO 下层过滤驱动程序。这些驱动程序只是一个框架，它们的代码相同。通过注册表中的信息来指示它们位于设备栈中的位置
Toastmon.sys	toastmon	此驱动程序演示了如何在内核模式下接收和处理 toaster 总线上的即插即用通知

在 Windows 中，设备驱动程序是一个可动态链接的模块（.sys 文件，符合 PE 文件格式[PE-SPEC]）。然而，为了让驱动程序被加载到系统中并且正确地工作，驱动程序的提供商通常还需要编写必要的配置信息，最基本的是一个.inf 文件，有时候是一些辅助的模块，甚至专门的安装软件。Toaster 例子展示的是一条虚拟总线，它提供了一些工具来模拟在总线上插入或拔除设备，同时也提供了相应的信息以便让即插即用管理器能够识别此总线以及总线上的设备。

在 Windows Server 2003 中，总线驱动程序的安装可以通过控制面板中的“添加硬件向导（Add Hardware Wizard）”来完成。在此安装过程中，除了总线驱动程序，.inf 文件也是必需的。此安装向导除了将.sys 和.inf 文件拷贝到相应的系统目录中以外，也会根据.inf 文件中的设置，在注册表中加入相应的键或值，以便即插即用管理器能够列举和管理 toaster 总线上的设备。

由于 toaster 并非物理总线，为了模拟在 toaster 总线上插入（plug）、拔除（unplug）和弹出（eject）设备的动作，toaster 例子提供了一个模拟工具 enum.exe，以命令行的方式插入、拔除或弹出一个指定编号的设备。另两个工具 toast.exe 和 notify.exe 分别演示了如何在应用程序中读写 toaster 总线上的设备，以及如何编写能感知 PnP 的应用程序（PnP-aware application）。Notify.exe 向即插即用管理器注册，当 toaster 总线上有设备插入或拔除时它可以得到通知，因此，如果它所打开的 toaster 设备要被移除，它就会被通知到，从而可以关闭该设备，以便该设备被从内存中移除。Notify.exe 例子程序演示了这一做法。

当即插即用管理器检测到 toaster 总线上有新的设备到达（实际上通过“enum.exe –p”命令来模拟）时，它启动 Windows 标准的“添加硬件向导”来安装该设备。Toaster 例子提供了此安装过程中所需要的文件和代码，包括.inf 文件、一个类安装器 DLL（class installer DLL）、一个协安装器 DLL（co-installer DLL），以及一组用于软件先安装（software-first）的介质（例如 CD）软件包。这里的类安装器 DLL 提供了针对 toaster 设备类的安装信息，

这只有在定义新的设备类时才是必要的。Toaster 的类安装器 DLL 提供了一个自定义的图标和属性对话框，当 Windows 设备管理器显示 toaster 设备时将会使用此图标和属性对话框。协安装器 DLL 执行的是针对特定设备实例的安装任务，toaster 的协安装器 DLL 为每个设备提供了一个唯一的友好名称，因而在 Windows 设备管理器中，各个 toaster 设备有不同的名称。

有关 Windows 设备和设备驱动程序的安装过程，以及类安装器 DLL 和协安装器 DLL 的完整介绍，请参考 Windows DDK 中关于设备安装的详细文档。关于 toaster 例子，读者可以参考例子本身随带的文档和 Microsoft 提供的相应文档（toastersamp.doc）。

6.5.3 驱动程序的代码结构

这一节我们首先来看一个驱动程序的基本代码结构，然后通过 toaster 例子中的总线驱动程序、功能驱动程序和过滤驱动程序，来学习这些驱动程序的设计要点。图 6.13 显示了一个驱动程序应该暴露的各种例程。

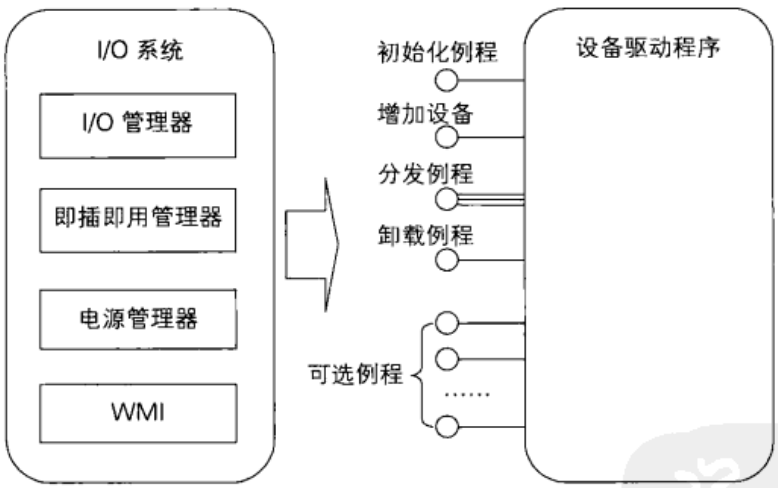


图 6.13 设备驱动程序的结构

虽然设备驱动程序中有明确的对象概念，包括驱动程序对象和设备对象等，但是，绝大多数设备驱动程序是用 C 语言来编写的，包括上一小节介绍的 toaster 例子中的驱动程序。在图 6.13 中，驱动程序与 I/O 系统之间的接口，即图中所示的各种例程，实际上是驱动程序实现的一组 C 风格的函数。驱动程序本身是一个动态链接库，扩展名为.sys，系统中已安装的驱动程序位于 Windows 目录的 system32\drivers 子目录下。驱动程序的主入口函数也被称为初始化例程，这是驱动程序被初始化时首先获得控制权的代码。其他的例程

都是通过此初始化例程来指定的。下面逐一介绍驱动程序暴露的例程。

- 初始化例程，是驱动程序 DRIVER_OBJECT 对象的 DriverInit 域。在驱动程序中，该例程通常被命名为 DriverEntry。此例程带两个参数：一个 DRIVER_OBJECT 对象指针和一个针对该驱动程序的注册表路径。在初始化例程中，驱动程序需要将自己实现的接口例程填充到第一个参数所指定的 DRIVER_OBJECT 对象中，这样相当于将这些接口例程告诉了 I/O 系统。

  正如 6.2.1 节中所介绍的那样，系统在加载驱动程序时，无论是通过 IopInitializeBuiltinDriver 函数，还是 IopLoadDriver 函数，都会调用初始化例程。

- 增加设备例程，是驱动程序 DRIVER_OBJECT 对象的扩展部分 DriverExtension 的 AddDevice 域。当即插即用管理器检测到一个由此驱动程序负责的设备时，即调用此例程来增加一个设备。通常，该例程调用 IoCreateDevice 创建一个 DEVICE_OBJECT 对象，并将它加入到设备栈中。

- 一组分发例程，是驱动程序 DRIVER_OBJECT 对象的 MajorFunction 数组域。通过 I/O 管理器发送的 I/O 请求最终由该数组中的函数来处理。在 Windows Server 2003 中，此数组共包含 27 项，其中有 IRP_MJ_READ、IRP_MJ_WRITE，以及前文看到过的 IRP_MJ_PNP 和 IRP_MJ_POWER 等，具体定义请参考 base\ntos\inc\io.h 文件中的 IRP_MJ_<*XXX*>宏定义。这些数组项与驱动程序能处理的 I/O 请求是对应的，其中有些 I/O 请求还有子命令，参见 io.h 文件中的 IRP_MN_<*XXX*>定义。如果设备驱动程序没有为一个 I/O 请求提供相应的处理例程，那么，I/O 管理器将该 I/O 请求对应的数组项设置为 IopInvalidDeviceRequest 函数，该函数只是简单地设置 I/O 请求包（IRP）中的状态，指明这是一个无效操作，然后完成此 I/O 请求。

- 卸载例程，是驱动程序 DRIVER_OBJECT 对象的 DriverUnload 域。该函数释放当前驱动程序所用到的系统资源，之后 I/O 管理器将驱动程序移除出内存。通常卸载例程与初始化例程相对应，也就是说，在初始化例程中申请的资源，应该在卸载例程中释放。正确的卸载例程可使得驱动程序被动态地卸载而不会造成任何资源泄漏。驱动程序的卸载是在系统线程（System 进程）中由 IopUnloadDriver 函数来完成，或者在一个工作项目（WorkItem）中由 IopLoadUnloadDriver 函数来完成的，参见 WRK 中这两个函数的代码。

- 一些可选的例程。通常驱动程序可能会涉及以下的例程：

  - ReInitialize 例程，这是第二阶段初始化例程。在驱动程序的初始化例程和增加设

备例程被调用，并且其他的驱动程序被加载和初始化以后，ReInitialize 例程将被调用。通常，一个驱动程序在初始化例程 DriverEntry 中调用 IoRegisterDriverReinitialization 函数可以登记一个 ReInitialize 例程。

- StartIo 例程，是驱动程序 DRIVER_OBJECT 对象的 DriverStartIo 域。驱动程序通过一个 StartIo 例程来发起一次数据传输，通常对于最底层的设备驱动程序有意义。这使得驱动程序可以并发地处理多个 I/O 请求。
- 中断服务例程（ISR）。若设备以中断方式通知处理器，则驱动程序必须编写并注册至少一个中断服务例程。Windows 提供了中断对象机制，允许驱动程序在不操纵 IDT（中断描述符表）的情况下，将一个例程与特定的中断向量关联起来，参见 5.2.3 节的介绍。中断服务例程通常只是保存设备的状态，然后插入一个 DPC 对象，以便在相对较低的 IRQL 上完成 I/O 任务。
- DPC 例程。驱动程序在 DPC 对象的延迟例程中完成 I/O 任务，参见 5.2.4 节的介绍。
- SynchCritSection 例程，用于同步访问设备硬件资源或驱动程序数据的例程。驱动程序通过 KeSynchronizeExecution 函数来同步一个 SynchCritSection 例程的执行，从而可以确保该例程与 ISR 不会并发执行。
- AdapterControl 例程。DMA 设备往往有一个 AdapterControl 例程，由它负责发起 DMA 操作。驱动程序调用 IoGetDmaAdapter 函数获得一个适配器对象（adapter object），以后调用适配器对象中的 AllocateAdapterControl 函数来准备 DMA 传输，而 AllocateAdapterControl 会调用参数中指定的 AdapterControl 例程来执行 DMA 操作。
- IoCompletion 例程，即 I/O 完成例程。驱动程序调用 IoSetCompletionRoutine 函数可以设置一个 I/O 请求的完成通知，设备栈中的下层驱动程序一旦调用了 IoCompleteRequest 函数，I/O 管理器就会依次调用上层驱动程序设置的完成例程，最后返回至 I/O 请求的发起者。参考 6.6.4 节介绍的 I/O 完成过程。
- Cancel 例程，即 I/O 取消例程。与完成例程的情形类似，一个驱动程序可以调用 IoSetCancelRoutine 函数，来设置一个 I/O 请求的取消例程。当驱动程序调用 IoCancelIrp 函数时，取消例程将被调用。与完成例程情形不同的是，取消例程并不针对设备栈层次中特定的驱动程序，而是针对一个 I/O 请求，所以，一个 I/O 请求只有一个取消例程。

- 定时器例程（IoTimer）。由于硬件操作的不确定性，设备驱动程序有时候需要周期性地做一些检查工作，比如查看一个 I/O 是否超时、定期刷新驱动程序中的某些变量（比如计数器），或者为某些内部操作进行计时等。驱动程序通过 IoInitializeTimer/IoStartTimer/IoStopTimer 函数来使用 I/O 定时器。I/O 定时器一旦被注册，则每秒钟被调用一次。I/O 定时器例程实际上是 DPC 例程，只不过它关联了一个设备对象，并且 I/O 管理器每秒钟调用它一次。设备对象 DEVICE_OBJECT 结构中的 Timer 成员记录了与之关联的定时器对象。I/O 管理器将所有的 I/O 定时器组织成一个链表，然后在一个 1 s 的系统定时器（全局的定时器变量 IopTimer）中依次触发所有已经被启动的 I/O 定时器。读者可以参考 IoInitializeTimer、IoStartTimer、IoStopTimer 以及 IopTimerDispatch 函数的代码实现。

因此，我们可以看到，驱动程序本质上是以上这些接口例程的集合。这些例程分别参与到与设备相关的 I/O 请求的处理中，有些例程的意图是直接操纵硬件设备，而有些例程是为了参与到 I/O 管理器的管理框架中。下面介绍 toaster 例子中主要驱动程序的基本实现，如表 6.4 所示。

表 6.4 toaster 例子中主要驱动程序的例程说明

驱动程序	支持的例程	说明
BusEnum.sys（总线驱动程序）	初始化例程	DriverEntry 函数，将所支持的其他例程设置到 DriverObject 中
	一组分发例程	一组名为 Bus_<Xxx>的函数，包括 Bus_CreateClose、Bus_PnP、Bus_Power、Bus_IoCtl 和 Bus_SystemControl
	卸载例程	Bus_DriverUnload 函数，清除资源
	增加设备例程	Bus_AddDevice 函数，创建设备对象（FDO）并附载到设备栈中，注册设备接口
toaster.sys（功能驱动程序）	初始化例程	DriverEntry 函数，将所支持的其他例程设置到 DriverObject 中
	一组分发例程	包括 ToasterDispatchPnp、ToasterDispatchPower、ToasterCreate、ToasterSystemControl、ToasterCleanup、ToasterDispatchIO、ToasterReadWrite、ToasterDispatchIoctl、ToasterClose
	卸载例程	ToasterUnload 函数，清除资源
	增加设备例程	ToasterAddDevice 函数，创建设备对象（FDO）并附载到设备栈中，注册设备接口，初始化电源状态

续表

驱动程序	支持的例程	说明
devupper.sys 等（过滤驱动程序）	初始化例程	DriverEntry 函数，将所支持的其他例程设置到 DriverObject 中
	一组分发例程	包括 FilterDispatchPnp、FilterDispatchPower 和 FilterDispatchIo，以及一个默认分发例程 FilterPass
	卸载例程	FilterUnload 函数，不做任何实质性事情
	增加设备例程	FilterAddDevice 函数，创建一个过滤设备对象（FiDO）并附载到设备栈中

在总线驱动程序 BusEnum.sys 中，对 IRP_MJ_PNP 命令的处理是在 Bus_PnP 函数中完成的。总线上插入、拔除和弹出设备的动作是由 IRP_MJ_DEVICE_CONTROL 命令中的三个子命令来模拟的。在接收到插入一个 toaster 设备的 I/O 请求时，驱动程序创建一个设备对象（成为该设备的 PDO），并执行初始化，然后调用 IoInvalidateDeviceRelations 函数通知即插即用管理器此设备插入进来；拔除设备的过程类似，它首先设置设备对象的存在标志（设备对象 DeviceExtension 的 Present 域，由 BusEnum.sys 驱动程序来定义）为 FALSE，然后调用 IoInvalidateDeviceRelations 函数，通知即插即用管理器重新检查总线上的设备节点。弹出设备的处理是通过调用即插即用管理器的 IoRequestDeviceEject 函数来完成的。

总线驱动程序 BusEnum.sys 负责两种类型的设备对象：一种是总线设备的 FDO，另一种是该总线上附载的设备的 PDO。设备对象的 DeviceExtension 成员的 IsFDO 域用于区分这两种情形。在 Bus_PnP 函数中，我们可以看到这两种设备对象的即插即用命令分别由 Bus_FDO_PnP 和 Bus_PDO_PnP 函数来处理。对于电源命令 IRP_MJ_POWER 的处理也类似，Bus_Power 依据不同的设备对象类型，分别调用 Bus_FDO_Power 或 Bus_PDO_Power 函数。关于这两种设备对象的即插即用命令和电源命令的具体实现，请参考 toaster 例子的 bus\pnp.c 和 bus\power.c 文件中的代码。Toaster 例子的说明文档（toaster.htm）解释了电源子命令 IRP_MN_QUERY_POWER 和 IRP_MN_SET_POWER 在几个驱动程序中的传递路径。

Toaster.sys 是 toaster 设备的功能驱动程序，它演示了即插即用命令的处理、电源命令的处理（支持 D0、D1 和 D3 设备电源状态）和 WMI 支持，同时它也支持设备的读写操作，允许应用程序通过 API 函数 DeviceIoControl 来操纵设备。Toaster 例子中的应用程序 toast.exe 可以测试 toaster 设备的读操作和 DeviceIoControl 操作。

Toaster 例子提供了 6 个过滤驱动程序，这些驱动程序只是一个基本框架，它们共用同一份代码，区别仅在于它们的编译配置和.inf 文件。过滤驱动程序把所有的非即插即用

I/O 请求直接传递给下层的驱动程序（命令处理函数为 FilterPass）；而对于绝大多数即插即用命令，它基本上也是忽略不予处理，但不管怎么样，它都会将 I/O 请求传递给下层驱动程序。过滤驱动程序实现了一把自定义的锁（设备对象的 DeviceExtension->RemoveLock 成员），以保护设备对象在 I/O 请求进行过程中不会被卸载（detach）和删除。

Toaster 例子中的驱动程序并没有实现实质性的功能，但是，其中对于即插即用命令和电源命令的实现代码演示了一个驱动程序在通常情形下应如何处理这些命令。结合例子中给出的一些辅助工具，并观察调试输出信息（即代码中的 DbgPrint 输出），这对于理解这些驱动程序与 I/O 管理器、即插即用管理器以及电源管理器的交互非常有帮助。建议读者在调试器或者可观察调试信息的工具（例如 DebugView）中检查 I/O 请求的执行情况。

6.5.4 toaster 设备的设备栈

前面 6.3.3 节介绍了即插即用管理器列举设备的过程，以及由此而建立起来的设备树及设备节点。现在我们来看一看 toaster 例子中的驱动程序如何参与到此过程中。首先，我们看一下 toaster 设备在设备树中的位置，如图 6.14 所示。

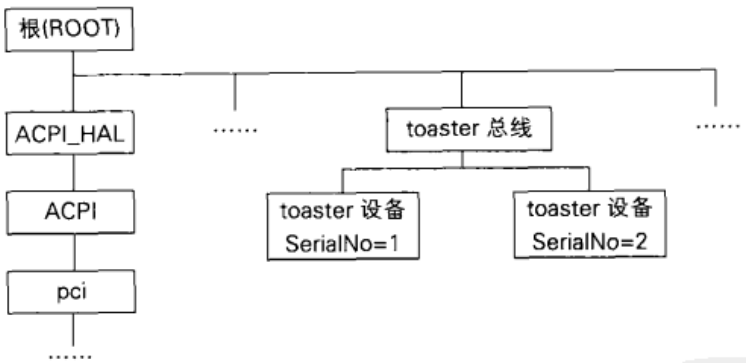


图 6.14 包含 toaster 总线和设备的设备树

Toaster 总线是直接连接在根（ROOT）节点下的设备，当 toaster 总线被安装到系统中以后，除非显式地删除它，否则每次系统引导时，即插即用管理器都会创建 toaster 总线设备，并且将总线驱动程序 BusEnum.sys 加载到系统中。正如本章前面所讲，注册表中记录了 ROOT 下面的设备（位于路径 HKLM\System\CurrentControlSet\Enum\Root）。因此，虽然 BusEnum.sys 驱动程序的启动类型为“按需-启动”，但是它在引导时被加载到系统中。如果我们将 toaster 总线设备从设备树中手动删除或禁止（在 Windows 的设备管理器中，在 toaster 总线设备上选择“卸载”或“禁用”即可），那么，BusEnum.sys 驱动程序将不会再在引导时被加载和启动。

在 toaster 总线设备的节点上，其设备栈的结构为：PDO 是 PnPManager 驱动程序创建的设备对象，FDO 则是 BusEnum.sys 驱动程序创建的设备对象，如图 6.15(a)所示。这两个设备对象是即插即用管理器在列举设备过程中指示 PnPManager 和 BusEnum.sys 驱动程序创建的。其中 FDO 是由 BusEnum.sys 驱动程序的增加设备例程创建的。该例程除了创建 FDO 并将其附载到设备栈中以外，还调用了 IoRegisterDeviceInterface 函数，创建一个设备接口（device interface）并注册到 I/O 系统中。

(a) toaster 总线的设备栈　　(b) toaster 设备的设备栈

图 6.15　toaster 总线和设备的设备栈

设备接口是应用程序与设备之间进行通信的重要途径。每一个设备接口类都有一个 GUID（128 位唯一标识符），toaster 总线的设备接口类的标识符为 GUID_DEVINTERFACE_BUSENUM_TOASTER，而 toaster 设备的标识符为 GUID_DEVINTERFACE_TOASTER。当设备驱动程序调用 IoRegisterDeviceInterface 函数注册设备接口类和该接口类的一个实例时，I/O 管理器提供一个符号链接名来表示该设备实例，此符号链接名永久保存在注册表中。因此，应用程序可以查询设备接口的实例，并获得符号链接名，然后利用此符号链接名访问相应的设备。Toaster 例子中的 enum.exe 和 toast.exe 程序演示了这一用法。

图 6.14 显示了在 toaster 总线下面有两个 toaster 设备节点。Toaster 设备节点的设备栈为：PDO 是 BusEnum.sys 驱动程序创建的设备对象，FDO 则是 toaster.sys 驱动程序创建的设备对象，如图 6.15(b)所示。正如上一小节所述，当模拟程序 enum.exe 指示 toaster 总线插入一个 toaster 设备时，toaster 总线驱动程序创建一个 PDO 对象，然后利用 IoInvalidateDeviceRelations 函数，通知即插即用管理器新设备的到来。即插即用管理器为 toaster 加载驱动程序（若有必要），并调用它的增加设备例程，为新设备创建 FDO。在 toaster.sys 驱动程序的增加设备例程（即 ToasterAddDevice 函数）中，它创建设备对象并执行初始化，然后调用 IoAttachDeviceToDeviceStack 函数，将设备对象加入到设备栈中。最后调用 IoRegisterDeviceInterface 函数为该 toaster 设备注册 toaster 设备类和设备类实例。因此，toast.exe 可以查询 toaster 设备类的实例，并通过 I/O 管理器向它们发送 I/O 请求。

6.5.5 过滤驱动程序的配置和加载

在一个即插即用设备的设备栈中，PDO 和 FDO 是必需的，它们是确保该设备正常工作的基础。然而，正如图 6.12 所示，在设备栈中还可以有多个可选的过滤设备对象，这些过滤设备对象位于 FDO 的上面或者下面，它们由相应的过滤驱动程序创建。Windows I/O 系统支持三种过滤驱动程序：总线过滤驱动程序、上层过滤驱动程序和下层过滤驱动程序。总线过滤驱动程序针对总线上所有的设备；而上下层过滤驱动程序分别针对特定的设备类或特定的设备实例，所以，它们又分为类过滤驱动程序和设备过滤驱动程序。这一小节我们将通过例子来说明不同类型过滤设备对象的创建过程。

总线过滤驱动程序是一种针对一条总线上所有设备的过滤驱动程序，它所创建的设备对象位于这些设备的设备栈的 PDO 之上，但位于 FDO 和其他过滤驱动程序之下。Windows 和 DDK 的文档并没有描述如何创建和配置总线过滤驱动程序。作为一个典型的例子，ACPI 驱动程序除了是一个总线驱动程序以外，也是一个总线过滤驱动程序。图 6.16 显示了一棵系统设备树的一部分，图中的粗体部分代表了总线过滤驱动程序创建的设备对象。PCI 总线上的设备，凡是支持 ACPI 的，都通过 ACPI 过滤驱动程序来执行设备电源策略和处理电源 I/O 请求。那么这些 ACPI 设备对象是如何创建、由谁创建的呢？它们是由 PCI 总线的 PDO，即 ACPI 总线驱动程序创建的设备对象，在处理 IRP_MJ_PNP 命令（IRP_MN_QUERY_DEVICE_RELATIONS 子命令）时创建并附载到 PCI 设备的设备栈中的。

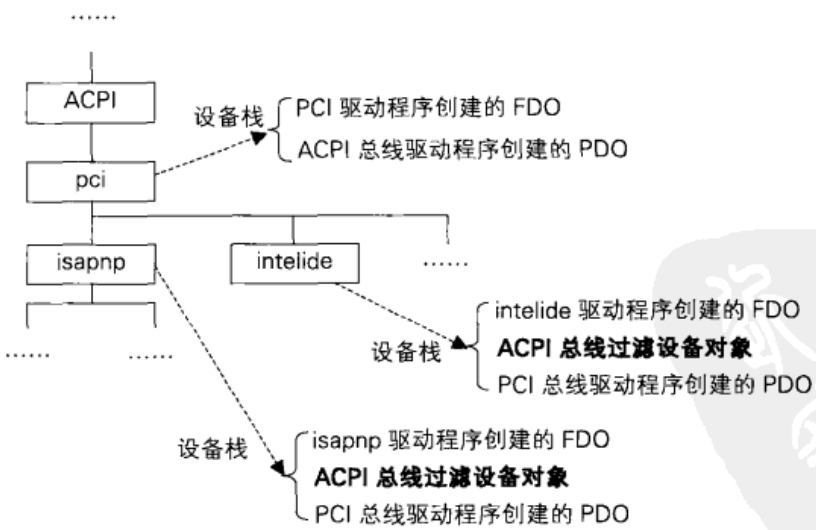


图 6.16 ACPI 总线过滤驱动程序示意图

总线过滤驱动程序改变或增强的是总线驱动程序的行为（即设备的 PDO 的功能），它们总是位于设备的 PDO 之上，而且，多个总线过滤设备对象可以附载到同一个 PDO 上。

如果把 toaster 例子中的 6 个过滤驱动程序（其代码位于 toaster\filter 目录下）编译出来，则可以得到名称不同的过滤驱动程序。若在安装 toaster 总线和 toaster 设备的驱动程序（通过“添加硬件向导”）时，指定该例子中提供的 busf.inf 和 toasterf.inf 文件，则这些过滤驱动程序也会被安装到系统中。具体的安装步骤参见 toaster 例子的 toaster.htm 文件。安装完成以后，我们可以得到图 6.17 所示的设备栈，由此可以看到这 6 个过滤驱动程序在设备栈中的位置。注意，toaster 例子并没有包含总线过滤驱动程序。在这 6 个过滤驱动程序中，BFdoUpr.sys 和 devupper.sys 是上层设备过滤驱动程序，clsupper.sys 是上层类过滤驱动程序；BFdoLwr.sys 和 devlower.sys 是下层设备过滤驱动程序，clslower.sys 是下层类过滤驱动程序。

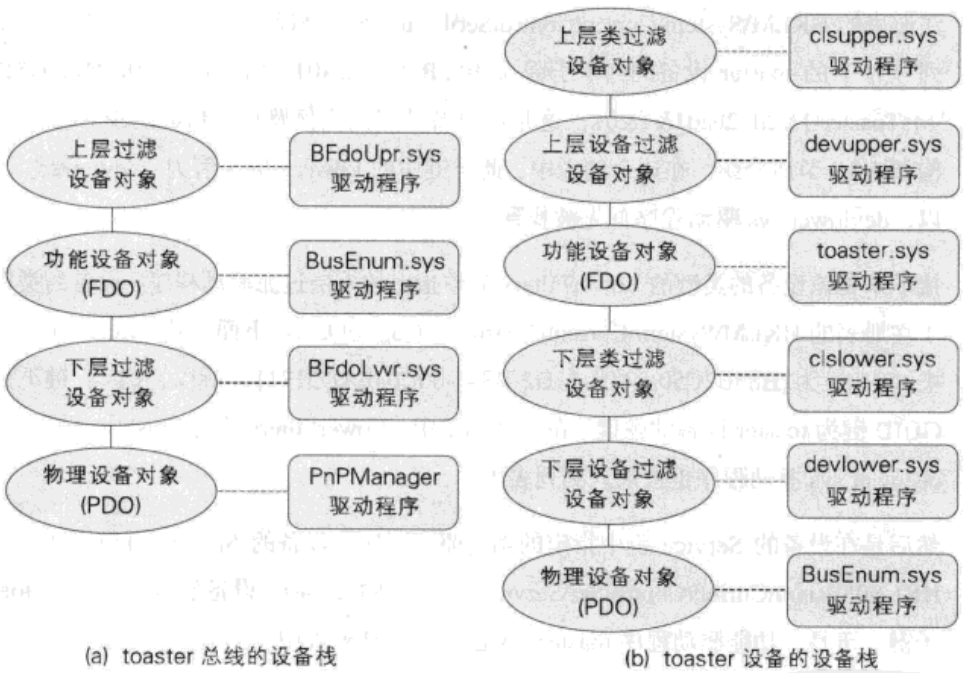


图 6.17 安装了过滤驱动程序后的 toaster 总线和设备的设备栈

在 toaster 例子中，toaster 设备的设备栈中包含了上下层的类和设备过滤设备对象，如图 6.17(b)所示。当即插即用管理器为一个 toaster 设备加载驱动程序时，它根据注册表中的信息来确定这些过滤驱动程序，并且调用驱动程序的增加设备例程，以便创建过滤设备对象并将其附载到设备栈中。在 Windows Server 2003 中，这一过程是在即插即用管理器的 PipCallDriverAddDevice 函数中完成的。

WRK 没有提供 PipCallDriverAddDevice 函数的代码，该函数的职责是，检查一个设备节点的驱动程序是否已经加载，若有必要，则加载它所依赖的驱动程序。在

PipCallDriverAddDevice 函数被调用以前，设备节点中的 PDO（以及总线过滤设备对象，若有的话）已经创建完成，但是 FDO 尚未创建。PipCallDriverAddDevice 函数根据注册表中的信息，构造一个驱动程序列表，包括上下层过滤驱动程序，然后按照顺序逐个调用驱动程序的增加设备例程。在驱动程序的增加设备例程中，标准的做法是，首先调用 IoCreateDevice 函数创建一个设备对象，然后通过 IoAttachDeviceToDeviceStack 函数，将新建的设备对象加入到设备节点中。由于 IoAttachDeviceToDeviceStack 函数总是将设备对象插入到设备栈的顶部，所以，PipCallDriverAddDevice 函数必须按照由下往上的顺序来调用增加设备例程。PipCallDriverAddDevice 构造的驱动程序列表的顺序如下所示：

- 首先是设备列举键的 LowerFilters 值中指定的下层过滤驱动程序。设备的列举键位于注册表的 HKLM\System\CurrentControlSet\Enum 下面该设备所对应的子键，例如，序列号为 1 的 toaster 设备的列举子键为{B85B7C50-6A01-11d2-B841-00C04FAD5171}\MsToaster\1&2d12bed1&1&01，这是该设备节点的实例路径（InstancePath），其结构如 6.3.3 节所介绍。在这个例子中，此子键中的 LowerFilters 值为“devlower”，所以，devlower.sys 驱动程序首先被找到。
- 接下来是该设备的类键的 LowerFilters 值中指定的下层过滤驱动程序。设备的类键位于注册表的 HKLM\System\CurrentControlSet\Control\Class 下面，例如，toaster 设备的类 GUID 为{B85B7C50-6A01-11D2-B841-00C04FAD5171}，所以，Class 键下面该 GUID 键为 toaster 设备的类键。在这个例子中，LowerFilters 值为“clslower”，所以，clslower.sys 驱动程序也被加入到列表中。
- 然后是在设备的 Service 键中指定的功能驱动程序。设备的 Service 键位于注册表的 HKLM\System\CurrentControlSet\Services 下面。例如，toaster 设备的 Service 键为 toaster 子键。于是，功能驱动程序 toaster.sys 也被加入到列表中。
- 再接下来是设备列举键的 UpperFilters 值中指定的上层过滤驱动程序。类似地，在{B85B7C50-6A01-11d2-B841-00C04FAD5171}\MsToaster\1&2d12bed1&1&01 子键中，UpperFilters 值为“devupper”，所以，devupper.sys 被加入到列表中。
- 最后是设备类键的 UpperFilters 值中指定的上层过滤驱动程序。类似地，在 toaster 设备类 GUID 键的 UpperFilters 值为“clsupper”，所以，clsupper.sys 被加入到列表中。

经过以上搜索过程，PipCallDriverAddDevice 函数获得了设备节点的所有驱动程序，从而可以构造出该设备的设备栈。图 6.17(b)正是以上完整搜索步骤的一个例子。而图 6.17(a)中的 BFdoLwr.sys 和 BFdoUpr.sys 驱动程序则是在 toaster 总线设备的列举键中的

LowerFilters 和 UpperFilters 值中指定的，该列举键位于 HKLM\System\CurrentControlSet\Enum\Root\System\<*nnnn*>，这里 *nnnn* 是一个 4 位数字编号，从 0000 开始，toaster 总线的具体编号取决于安装其总线驱动程序时已有系统设备（System Device）的编号情况。

以上描述了过滤驱动程序的搜索顺序，由于上面提到的 UpperFilters 和 LowerFilters 注册表值是多字符串类型（REG_MULTI_SZ），因此每个值都可以指定多个过滤驱动程序。从设计角度，这些值中指定的过滤驱动程序不应该依赖于它们在这些值中的顺序。譬如，一个下层类过滤驱动程序不应该要求它必须出现在 LowerFilters 值的某个特定位置上，但是，有一点是可以保证的，即它创建的设备对象一定位于 FDO 和上层过滤驱动程序的下面。这些确定的和不确定的顺序关系会影响到 I/O 请求在设备栈中的处理顺序。

即插即用管理器从 LowerFilters 和 UpperFilters 注册表值中获得的仅仅是过滤驱动程序的服务名，而过滤驱动程序真正的信息位于 HKLM\System\CurrentControlSet\Services 下面。所有这些驱动程序，包括功能驱动程序和过滤驱动程序，如果尚未被加载到系统中，则通过 IopLoadDriver 函数被加载进来。

那么，有关过滤驱动程序的注册表信息如何建立起来呢？这是由过滤驱动程序的相应安装程序来完成，或者随功能驱动程序一起安装的。在安装设备（譬如通过系统的“添加硬件向导”）或软件时，通常有一个.inf 文件描述了驱动程序的相关信息，包括设备的类 GUID、厂商信息、注册表路径，以及每个驱动程序单独的注册信息等。在 toaster 例子中，我们可以看到多个这样的.inf 文件，包括本节前文提到的 busf.inf 和 toasterf.inf，通过这两个文件中描述的信息可以建立起图 6.17 中所示的包括过滤设备对象的设备栈。

关于 toaster 例子的安装过程以及.inf 文件的内容，请参考 6.5.2 节的介绍以及该例子随带的一些说明文件，也可以从 Internet 上查看 Windows DDK 中关于 toaster 例子的联机文档[TOASTER][TOASTER-DOC]。

6.5.6 非即插即用驱动程序

除了与设备关联的即插即用驱动程序，Windows 还支持非即插即用驱动程序和文件系统驱动程序。这一节我们讨论非即插即用驱动程序，这种驱动程序也称为内核扩展驱动程序，在 Windows 驱动程序模型（WDM）出现以前的驱动程序均属于这一类，它们在 Windows 2000/XP/Server 2003 及后续版本中仍然可以使用。但由于这一类驱动程序不支持 WDM，所以，即插即用管理器和电源管理器无法管理它们，以及它们所创建的设备对象。在有些情况下，非即插即用驱动程序仍然是一个合适的选择，例如，本书前几章介绍的所有工具（以及后面 6.7 节将要介绍的 IRPMon 工具）都使用了非即插即用驱动程序。

非即插即用驱动程序最主要的意义在于，让用户程序与内核之间多了一种通信方式，并且用户程序可以指示驱动程序做一些只有在内核中才能完成的任务，例如，访问系统地址空间中的数据，提供一种新的跨进程通信机制，或者访问内核提供的功能。当然，非即插即用驱动程序也可以驱动设备，直接与硬件打交道。不具备即插即用能力的硬件设备仍然可以通过这种驱动程序来工作。

这一类驱动程序的一种典型用法是，当软件需要内核中的功能时，应用程序通过服务控制管理器（SCM）来安装和加载驱动程序，然后通过 I/O 管理器与驱动程序进行通信；当应用程序退出时，如不再需要该驱动程序则通过 SCM 动态地卸载。下面我们通过 Windows DDK 中提供的一个例子程序 IOCTL 来说明这一过程。

IOCTL 例子包括两个程序：ioctlapp.exe 和 sioctl.sys。其中 ioctlapp.exe 是一个命令行应用程序，该程序在启动时安装和加载驱动程序，并通过 DeviceIoControl API 函数向驱动程序发送 4 个不同类型的 I/O 请求，分别为 METHOD_IN_DIRECT、METHOD_OUT_DIRECT、METHOD_NEITHER 和 METHOD_BUFFERED，关于这些 I/O 请求类型，参见 6.6.3 节的说明。Ioctlapp 程序在发送了这 4 个请求以后，卸载驱动程序并退出。

Sioctl.sys 是一个非即插即用驱动程序，它提供了初始化例程、卸载例程，以及 3 个分发例程，分别处理 IRP_MJ_CREATE、IRP_MJ_CLOSE 和 IRP_MJ_DEVICE_CONTROL 命令。它的初始化例程除了注册其他的例程以外，也调用 IoCreateDevice 函数创建一个设备对象，该设备对象的名称为“\Device\SIOCTL”，而且它还调用 IoCreateSymbolicLink 函数来创建一个 DOS 设备名称“\DosDevices\IoctlTest”。因此，ioctlapp 程序可以通过 CreateFile 函数来打开一个指向此设备对象的文件对象，从而请求该设备对象的服务，相当于调用到该驱动程序的例程中。

现在我们来看 ioctlapp 程序如何操纵 sioctl.sys 驱动程序。为了使用 SCM 的功能，ioctlapp 首先调用 API 函数 OpenSCManager，以便连接到 SCM，并打开系统服务数据库。然后通过 SCM 中的以下服务来操纵 sioctl.sys：

- CreateService，创建一个 Windows 服务，将它安装到 SCM 数据库中。CreateService 会在 HKLM\System\CurrentControlSet\Services 下面创建一个子键，子键的名称等同于新建服务的名称，由参数来指定。CreateService 的参数较多，因为它必须指定新建服务的服务类型、启动类型、驱动程序路径等各种信息。这些信息都将保存在 SCM 数据库中。
- OpenService，打开一个已安装的服务。在参数中指定服务名称和访问方式。
- StartService，启动一个已通过 CreateService 或 OpenService 打开的服务。

- ControlService，指示一个服务暂停（pause）、继续（continue）、停止（stop）或者向 SCM 报告其状态。
- DeleteService，将指定的服务从 SCM 数据库中删除。

应用程序通过句柄（其类型为 SC_HANDLE）来管理所用到的服务，以及它与 SCM 之间的连接。当应用程序完成了 SCM 连接或者一个服务时，它调用 CloseServiceHandle 来结束相应的句柄。关于以上这些 SCM API 函数的用法，请参考 ioctlapp 应用程序的代码或者 Windows SDK 中的相关文档。

从 sioctl.sys 驱动程序的角度来看，当 SCM 的 StartService 服务被调用时，该驱动程序被加载到系统中。驱动程序的加载过程是通过 NtLoadDriver 函数，进而由 IopLoadDriver 函数来完成的。驱动程序的初始化例程（即 DriverEntry 函数）是在 IopLoadDriver 函数中被调用的，参见 6.2.1 节的介绍，这里不再赘述。驱动程序的卸载例程（SioctlUnloadDriver）是在 IopLoadUnloadDriver 函数中被调用的，这发生在 ioctlapp 应用程序调用 ControlService 来停止服务之时。

通过这种方式创建的设备对象并没有参与到系统的设备树和设备节点中。然而，如果在系统引导时由即插即用管理器来加载一个非即插即用的驱动程序，那么，该驱动程序也会参与到设备树中。如果一个非即插即用的驱动程序被安装到系统中，并且它的启动类型设置为“引导-启动”或“系统-启动”，那么，在系统引导时，即插即用管理器会按照即插即用驱动程序来加载此驱动程序，并且为它建立一个设备节点插入到设备树中。这些设备节点直接附载在根节点的下面，其 PDO 为驱动程序“\Driver\PnpManager”创建的设备对象。由于被加载的驱动程序通常并未提供增加设备例程，因此这些设备节点中只有总线驱动程序创建的 PDO，而并不存在 FDO。在这种情况下，即插即用管理器加载并初始化驱动程序，但驱动程序并不会参与到设备节点中。这些驱动程序可以创建一个命名的设备对象，从而使应用程序可以通过名称来指定 I/O 请求的目标。

非即插即用驱动程序也可以通过“添加硬件向导”来安装，在这种情况下.inf 文件是必需的。即插即用管理器会创建一个设备节点，并创建 PDO，其加载过程类似于引导时加载非即插即用驱动程序的情形，这里不再赘述。

最后简单小结一下以上关于 Windows 设备驱动程序的讨论。本质上讲，Windows 的设备驱动程序是一个动态链接库，它的入口点是初始化例程，I/O 系统所要求的其他例程都是在初始化例程中指定的。在本节中，我们通过 toaster 例子讨论了硬件设备的设备栈是如何构建起来的，以及过滤驱动程序如何参与其中。然而，本节并没有介绍驱动程序应

该如何响应各种 I/O 请求。关于如何编写驱动程序的例程来响应 I/O 请求，请读者参考 Windows DDK 的文档说明，或者 Walter Oney 的著作[WALTER]。另外，Windows DDK 提供的丰富的例子展示了各种类型的硬件设备的驱动程序应如何实现。

6.6 I/O 处理

在本章前面的介绍中，我们总是认为一个 I/O 请求会被正确地传递到恰当的驱动程序中，由恰当的例程来处理。我们已经明白了驱动程序如何被加载到系统中，哪些例程可能会暴露出来，以及设备对象与驱动程序对象之间的关系。在这一节中，我们将讨论一个 I/O 请求是如何被表达和处理的，它在设备栈中又是如何被传递和完成的，以及一种特殊的 I/O 完成机制——I/O 完成端口。

6.6.1 I/O 请求包（IRP）

IRP 是 Windows 系统中用于表达一个 I/O 请求的核心数据结构，当内核模式代码要发起一个 I/O 请求时，它会构造一个 IRP，用于在处理该 I/O 请求的过程中代表该请求。下面是 IRP 数据结构的定义（位于文件 base\ntos\inc\io.h 中）：

```
typedef struct _IRP {
    CSHORT Type;                        // IRP 类型，等于 IO_TYPE_IRP 宏
    USHORT Size;                        // IRP 大小

    PMDL MdlAddress;                    // 该 I/O 请求的用户缓冲区的 MDL，仅用于“直接 I/O”类型
    ULONG Flags;                        // 用于记录各种标志
    union {
        struct _IRP *MasterIrp;         // 若这是一个关联 IRP，则指向主 IRP
        LONG IrpCount;                  // 若这是一个主 IRP，则必须先完成多少个关联 IRP
        PVOID SystemBuffer;             // 该操作被缓冲起来，指向系统地址空间缓冲区的地址
    } AssociatedIrp;
    LIST_ENTRY ThreadListEntry;         // 链表项，可以加入到线程的未完成 I/O 请求链表中
    IO_STATUS_BLOCK IoStatus;               // I/O 操作的状态
    KPROCESSOR_MODE RequestorMode;          // 内核模式 I/O 请求或用户模式 I/O 请求
    BOOLEAN PendingReturned;                // 未完成返回
    CHAR StackCount;                        // 栈单元计数
    CHAR CurrentLocation;                   // 当前栈单元位置
    BOOLEAN Cancel;                         // 该 I/O 请求是否已被取消
    KIRQL CancelIrql;                       // 取消自旋锁在哪级 IRQL 上被获取
    CCHAR ApcEnvironment;                   // 用于当该 IRP 被初始化时保存 APC 环境
    UCHAR AllocationFlags;                  // 该 IRP 内存的分配控制标志
    PIO_STATUS_BLOCK UserIosb;              // 用户的 I/O 状态块
    PKEVENT UserEvent;                      // 用户事件对象
    union {
        struct {
            PIO_APC_ROUTINE UserApcRoutine;     // 当 I/O 请求完成时执行的 APC 例程
            PVOID UserApcContext;               // 传递给 UserApcRoutine 的环境参数
        } AsynchronousParameters;
```

```
        LARGE_INTEGER AllocationSize;          // 分配块的大小
    } Overlay;

    PDRIVER_CANCEL CancelRoutine;        // 若是可取消的 I/O 请求，该域包含了取消时调用的例程
    PVOID UserBuffer;                    // 调用者（即发起者）提供的输出缓冲区的地址

    // 以下 Tail 联合成员用于当 I/O 管理器处理该 I/O 请求时存放各种工作信息
    union {
        struct {
            union {
                KDEVICE_QUEUE_ENTRY DeviceQueueEntry; // 设备队列项
                struct {
                    PVOID DriverContext[4];     // 由驱动程序解释和使用
                } ;
            } ;
            PETHREAD Thread;                    // 指向发起者线程的 ETHREAD
            PCHAR AuxiliaryBuffer;              // 辅助缓冲区
            struct {
                LIST_ENTRY ListEntry;           // 存放到完成队列中的链表项
                union {
                    struct _IO_STACK_LOCATION *CurrentStackLocation;
                                                // 指向当前栈单元，驱动程序不可直接访问
                    ULONG PacketType;           // Minipacket 的类型
                };
            };
            PFILE_OBJECT OriginalFileObject;  // 指向原始的文件对象
        } Overlay;
        KAPC Apc;                       // 特殊内核模式 APC 或发起者的 APC
        PVOID CompletionKey;            // 完成键，用于标识在不同文件句柄上的 I/O 请求
    } Tail;
} IRP, *PIRP;
```

IRP 对象从一个 I/O 请求被发起时开始存在，一直到该 I/O 请求被完成或者取消为止，在此过程中，会有多方操纵此 IRP 对象，包括 I/O 管理器、即插即用管理器、电源管理器以及一个或多个驱动程序等。Windows I/O 系统本质上支持异步 I/O 请求，所以，IRP 对象必须携带足够多的环境信息，以便能够描述一个 I/O 请求的所有状态。关于每个数据成员的含义，请参考上述代码中的注释部分，下面列出 IRP 对象的设计特点：

- 所有的 I/O 请求都被抽象成针对文件对象的操作，OriginalFileObject 成员记录了该文件对象，代表一个 I/O 请求的目标。
- 栈单元。由于可能有多个驱动程序依次参与到一个 I/O 请求的处理过程中，所以，一个 IRP 对象除了以上列出的 IRP 数据结构，其后还有一个称为栈单元（Stack Location）的数组。每个参与处理的驱动程序都可以有它自己的栈单元。
- 支持取消。线程可以指定一个取消例程，以便当 I/O 请求被取消时可以执行相关的动作。
- I/O 请求的完成。这是一个显式的通知，当一个驱动程序的分发例程认为它已经完成了该 I/O 请求时，它会显式地告诉 I/O 管理器，此 I/O 请求已完成。参与处理的驱动

程序可以为 I/O 请求注册一个完成例程，从而当它的下级驱动程序完成了该 I/O 请求时，此完成例程会被调用。

在 Windows 中，I/O 请求的目标是一个设备对象，但 IRP 并不直接指向该设备对象。正如 6.2.3 节所介绍，文件对象代表了设备对象的已打开实例。IRP 对象记录了 I/O 请求的目标是一个文件对象，即它的 OriginalFileObject 域。I/O 管理器通过 IoGetRelatedDeviceObject 函数可以定位到与该文件对象关联的设备对象，而且，如果目标设备对象位于一个设备栈中，那么该函数得到的是栈顶处的设备对象。关于如何从一个文件对象获得相关的设备对象，请参考 base\ntos\io\iomgr\iosubs.c 文件中 IoGetRelatedDeviceObject 和 IoGetAttachedDevice 函数的代码；关于如何创建或打开一个文件对象的过程，请参考 6.2.3 节的介绍。

一旦定位到设备对象，接下来就可以进一步定位到驱动程序对象，因而，I/O 管理器可以将 I/O 请求传递给驱动程序中的例程，交由驱动程序来处理。然而，从 IRP 数据结构的定义中，我们没有看到 I/O 请求与驱动程序分发例程之间的关系。换句话说，从 IRP 的定义，看不出一个 I/O 请求对应于驱动程序中哪个分发例程。实际上，IRP 数据结构仅仅是一个 I/O 请求的固定描述部分，另一部分是一个或者多个栈单元。每个栈单元针对单个驱动程序，I/O 管理器在处理一个 I/O 请求时，在调用驱动程序的分发例程以前，必须准备好一个栈单元空间，IRP 结构中的 CurrentLocation 指明了供驱动程序使用的栈单元位置。栈单元数组依附在 IRP 数据结构的尾部。内核模式代码在发起一个 I/O 请求时，根据目标设备对象的 StackSize 域，可以知道最多有多少个驱动程序需要参与到该 I/O 请求的处理过程中，所以，它在分配 IRP 对象时，能够清楚地指示应该分配多少个栈单元。

以下是栈单元 IO_STACK_LOCATION 数据结构的定义（位于文件 base\ntos\inc\io.h 中）：

```
typedef struct _IO_STACK_LOCATION {
    UCHAR MajorFunction;                    // 在该设备对象上的 I/O 请求主功能码
    UCHAR MinorFunction;                    // 在该设备对象上的 I/O 请求次功能码
    UCHAR Flags;
    UCHAR Control;

    union {
        struct { … } Create;                // NtCreateFile 的参数
        struct { … } CreatePipe;            // NtCreateNamedPipeFile 的参数
        struct { … } CreateMailslot;        // NtCreateMailslotFile 的参数
        struct { … } Read;                  // NtReadFile 的参数
        struct { … } Write;                 // NtWriteFile 的参数
        struct { … } QueryDirectory;        // NtQueryDirectoryFile 的参数
        struct { … } NotifyDirectory;       // NtNotifyChangeDirectoryFile 的参数
        struct { … } QueryFile;             // NtQueryInformationFile 的参数
        struct { … } SetFile;               // NtSetInformationFile 的参数
```

```
        struct { … } QueryEa;                   // NtQueryEaFile 的参数
        struct { … } SetEa;                     // NtSetEaFile 的参数
        struct { … } QueryVolume;               // NtQueryVolumeInformationFile 的参数
        struct { … } SetVolume;                 // NtSetVolumeInformationFile 的参数
        struct { … } FileSystemControl;         // NtFsControlFile 的参数
        struct { … } LockControl;               // NtLockFile/NtUnlockFile 的参数
                                                // NtFlushBuffersFile 的参数（无参数）
                                                // NtCancelIoFile 的参数（无参数）
        struct { … } DeviceIoControl;           // NtDeviceIoControlFile 的参数
        struct { … } QuerySecurity;             // NtQuerySecurityObject 的参数
        struct { … } SetSecurity;               // NtSetSecurityObject 的参数

        struct { … } MountVolume;               // MountVolume 的参数
        struct { … } VerifyVolume;              // VerifyVolume 的参数
        struct { … } Scsi;                      // SCSI 内部设备控制的参数

        struct { … } QueryQuota;                // NtQueryQuotaInformationFile 的参数
        struct { … } SetQuota;                  // NtSetQuotaInformationFile 的参数
        struct { … } QueryDeviceRelations;      // IRP_MN_QUERY_DEVICE_RELATIONS 的参数
        struct { … } QueryInterface;            // IRP_MN_QUERY_INTERFACE 的参数
        struct { … } DeviceCapabilities;        // IRP_MN_QUERY_CAPABILITIES 的参数
        struct { … } FilterResourceRequirements;
                                        // IRP_MN_FILTER_RESOURCE_REQUIREMENTS 的参数
        struct { … } ReadWriteConfig;
                                        // IRP_MN_READ_CONFIG 和 IRP_MN_WRITE_CONFIG 的参数
        struct { … } SetLock;                   // IRP_MN_SET_LOCK 的参数
        struct { … } QueryId;                   // IRP_MN_QUERY_ID 的参数
        struct { … } QueryDeviceText;           // IRP_MN_QUERY_DEVICE_TEXT 的参数
        struct { … } UsageNotification;         // IRP_MN_DEVICE_USAGE_NOTIFICATION 的参数
        struct { … } WaitWake;                  // IRP_MN_WAIT_WAKE 的参数
        struct { … } PowerSequence;             // IRP_MN_POWER_SEQUENCE 的参数
        struct { … } Power;             // IRP_MN_SET_POWER 和 IRP_MN_QUERY_POWER 的参数
        struct { … } StartDevice;               // StartDevice 的参数
                                                // Cleanup 的参数（无参数）
        struct { … } WMI;                       // WMI IRP 的参数
        struct {
            PVOID Argument1;
            PVOID Argument2;
            PVOID Argument3;
            PVOID Argument4;
        } Others;                               // 其他，由驱动程序自己定义和解释
    } Parameters;

    PDEVICE_OBJECT DeviceObject;
    PFILE_OBJECT FileObject;
    PIO_COMPLETION_ROUTINE CompletionRoutine;
    PVOID Context;
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

IO_STACK_LOCATION 的定义虽然很长，但它的结构并不复杂，其 Parameters 成员囊括了发起 I/O 请求的各种可能情形下的原始参数信息。驱动程序可以参考这些信息来处理 I/O 请求。其中 MajorFunction 域指示驱动程序应该由哪个例程来处理该 I/O 请求，有些主功能码还有相应的次功能码，即 MinorFunction 域。Flags 域包含了与 I/O 请求的类型（即

MajorFunction）相关的标志信息，参见 base\ntos\inc\io.h 文件中以“SL_”打头的一些 C 宏定义。Control 域包含了栈单元的控制标志，该域对于驱动程序来说是只读的，可能的值包括 SL_PENDING_RETURNED、SL_ERROR_RETURNED、SL_INVOKE_ON_CANCEL、SL_INVOKE_ON_SUCCESS 和 SL_INVOKE_ON_ERROR。DeviceObject 域指向与该栈单元相对应的设备对象，FileObject 域指向与 DeviceObject 相关联的文件对象，是它的一个已打开实例。CompletionRoutine 域是该栈单元对应的驱动程序通过 IoSetCompletionRoutine 设置的完成例程，Context 域是由驱动程序指定的、传递给 CompletionRoutine 完成例程的参数。

图 6.18 显示了 IRP 与文件对象、设备对象和驱动程序之间的关系。I/O 请求的发起者负责创建 IRP 对象，包括紧随其后的栈单元数组，然后初始化第一个栈单元。由于栈单元按照栈（stack）的方式来工作，所以，第一个栈单元即数组的最后一个元素。栈单元中的文件对象指向它的目标设备对象，设备对象包含了指向驱动程序对象的指针，而驱动程序对象则包含了一组分发例程，因此，基于这样的数据结构关系，I/O 管理器可以将 I/O 请求转发给相应驱动程序的分发例程。

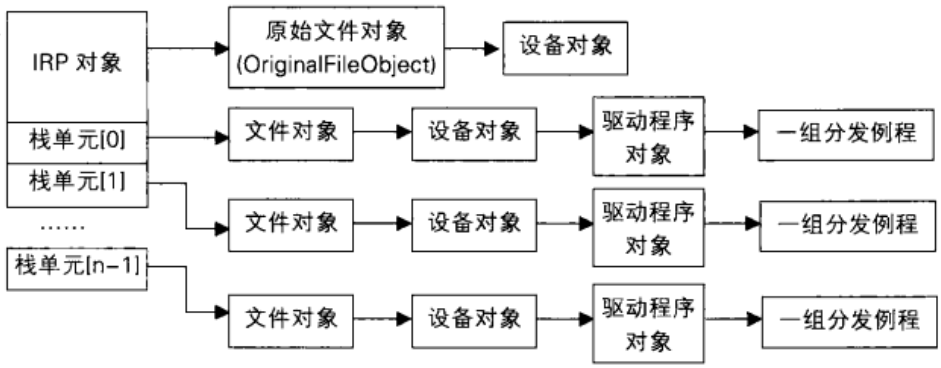


图 6.18 IRP 与其他数据结构的关系

不难理解，一个 IRP 对象的栈单元所指的这些设备对象正是本章前面讲述的设备栈。I/O 请求的发起者在构造 IRP 对象时，根据 OriginalFileObject 所指设备对象的栈深度，即 DEVICE_OBJECT 数据结构的 StackSize 成员，来决定应该为 IRP 对象分配多少个栈单元。值得一提的是，IRP 对象总是从非换页内存池中分配的，所以，即使处理 I/O 请求的驱动程序代码运行在 DISPATCH_LEVEL 或更高的 IRQL 上，也可以安全地访问 IRP 对象中的信息。

在 Windows 中，IRP 对象总是通过 IoAllocateIrp 函数来分配的，该函数的代码位于 base\ntos\io\iomgr\iosubs.c 文件中，它仅仅将函数调用转发给 pIoAllocateIrp 函数

指针。pIoAllocateIrp 是一个全局变量，在 I/O 系统的初始化过程中被设置为指向 IopAllocateIrpPrivate 函数，所以，IRP 对象的分配工作实际上是在 IopAllocateIrpPrivate 函数中完成的。类似地，IRP 对象的释放工作是在 IopFreeIrp 函数中完成的，但 I/O 管理器提供的接口函数为 IoFreeIrp。

I/O 管理器维护了两个快查表 IopLargeIrpLookasideList 和 IopSmallIrpLookasideList。前者链表中的 IRP 对象包含 IopLargeIrpStackLocations 个栈单元，IopLargeIrpStackLocations 的默认值为 8；后者链表中的 IRP 对象包含一个栈单元。这两个快查表也被记录在处理器 PRCB 结构的快查表数组成员 PPLookasideList 中。快查表中的 IRP 对象是系统资源，而并非属于某个进程。每个处理器的 PRCB 结构的 LookasideIrpFloat 成员记录了当前允许从快查表中提供给进程的 IRP 对象的数量，LookasideIrpFloat 成员的初始值是在 IoInitSystem 函数中设置的，其大小取决于当前系统内存以及处理器的数量，参见 IoInitSystem 函数中的代码。另外，平衡集管理器会定期平衡各个处理器剩余的 IRP 对象配额，参见 base\ntos\ke\balmgr.c 文件中的 KiAdjustIrpCredits 函数，该函数仅在多处理器的情况下被 KeBalanceSetManager 函数调用。

IopAllocateIrpPrivate 函数的代码位于 iosubs.c 文件的 744~909 行，在代码中我们可以看到以上两个快查表的用法。该函数的逻辑较为简单，首先根据参数中指定的栈单元个数确定应该检查哪个快查表，如果能够从快查表中获得已有的 IRP 对象，则直接从快查表的单链表中摘除（pop）一项作为结果 IRP 对象；否则，调用 ExAllocatePoolWithQuotaTag 或 ExAllocatePoolWithTag 函数从非换页内存池中申请一个新的 IRP 对象。最后，IopAllocateIrpPrivate 调用 IopInitializeIrp 函数（实际是一个 C 宏）对 IRP 对象执行基本的初始化，并设置它的分配标志，即 AllocationFlags 域。

IopFreeIrp 函数的代码位于 iosubs.c 文件的 7 143~7 257 行。它首先根据栈单元的个数确定应该检查哪个快查表，如果快查表的深度（即 Depth 域）还允许它继续往快查表的单链表中插入 IRP 对象，则通过 InterlockedPushEntrySList 函数将它插入到快查表中；否则释放 IRP 对象的内存。这里要注意的一点是，在分配 IRP 对象时，IopAllocateIrpPrivate 利用 PRCB 的 LookasideIrpFloat 成员来控制从快查表转移到进程内存的 IRP 对象的数量；而在释放 IRP 对象时，IopFreeIrp 函数除了维护 PRCB 的 LookasideIrpFloat 成员以外，还根据快查表的 Depth 成员，来控制是将待释放的 IRP 对象加入到快查表的单链表中，还是直接返还给系统的非换页内存池。

理解了 IRP 对象的内存管理原理以后，我们再来看一看 IRP 栈单元的用法。首先，栈单元数组紧附在 IRP 数据结构的后面，所以，在调用 IoAllocateIrp 申请 IRP 对象时，必

须指定栈单元的数量。IRP 对象在初始构造时，其当前栈单元的位置是栈单元数组的末尾（即第一个栈单元位置），参考 C 宏 IopInitializeIrp 的代码。I/O 管理器在将 IRP 对象传递给第一个驱动程序以前，会将 IRP 对象的 Tail.Overlay.CurrentStackLocation 域赋值为指向栈单元数组的最后一个栈单元项。请参考 base\ntos\io\iomgr\iomgr.h 文件中 IopfCallDriver 函数的代码。如果一个驱动程序要将一个 I/O 请求传递给它下层的驱动程序来处理，它自己不做进一步处理，则下面的代码可以做到这一点：

```
PDEVICE_EXTENSION          deviceExtension;
NTSTATUS    status;

deviceExtension = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;
IoSkipCurrentIrpStackLocation (Irp);
status = IoCallDriver (deviceExtension->NextLowerDriver, Irp);

return status;
```

这里 IoSkipCurrentIrpStackLocation 宏的定义如下：

```
#define IoSkipCurrentIrpStackLocation( Irp ) { \
   (Irp)->CurrentLocation++; \
   (Irp)->Tail.Overlay.CurrentStackLocation++; }
```

因此，IoSkipCurrentIrpStackLocation 宏相当于将当前栈位置又倒回去一格，而到了 IoCallDriver 函数中，它进一步调用 IopfCallDriver 函数，该函数在调用下层驱动程序以前，又会将 Tail.Overlay.CurrentStackLocation 域向前移动一格栈单元（即数组下标减小）。所以，下层驱动程序仍然使用同一格栈单元。

相反地，如果一个驱动程序在处理一个 I/O 请求时设置了完成例程，并且还需要下层驱动程序做进一步处理，则不能像前面所列代码那样使用 IoSkipCurrentIrpStackLocation 宏，而应该在调用 IoCallDriver 函数以前使用 IoCopyCurrentIrpStackLocationToNext 宏，该宏负责将当前栈单元中完成例程之前的所有内容复制到下一个栈单元（即栈单元数组中下标减一的元素）中。因此，在下层驱动程序的分发例程中，它可以获得栈单元中除了完成例程以外的 I/O 请求信息。

以上我们看到了 I/O 管理器利用栈单元数组为每个参与处理 I/O 请求的驱动程序提供了一个执行环境，驱动程序通过调整栈单元的位置来保留或忽略它自己的栈单元。一个 I/O 请求从设备栈的顶部向下流动，从驱动程序的角度来看，则是通过调用 I/O 管理器的 IoCallDriver 函数逐层向下。当一个驱动程序完成了该 I/O 请求时，它通过调用 I/O 管理器的 IoRequestComplete 函数向上流动，凡是在栈单元中注册了完成例程的驱动程序都有机会按照相反的顺序接收到完成通知。关于 I/O 请求在设备栈中的流动，参见 6.6.4 节。

6.6.2 针对独立设备对象的 I/O 处理

I/O 请求可能来自用户模式程序，也可能来自内核中的组件。IO_STACK_LOCATION 结构的 Parameters 成员的定义其实已经指示了 I/O 请求的很多来源。内核代码在初始化 IRP 对象时，往往也会提供与 I/O 请求来源相关的信息，以方便 I/O 管理器或驱动程序处理该请求。I/O 管理器接收 I/O 请求的统一接口函数是 IoCallDriver，但内核接收 I/O 请求的系统服务有很多，图 6.19 显示了应用程序通过系统 DLL 发送 I/O 请求的控制流路径。

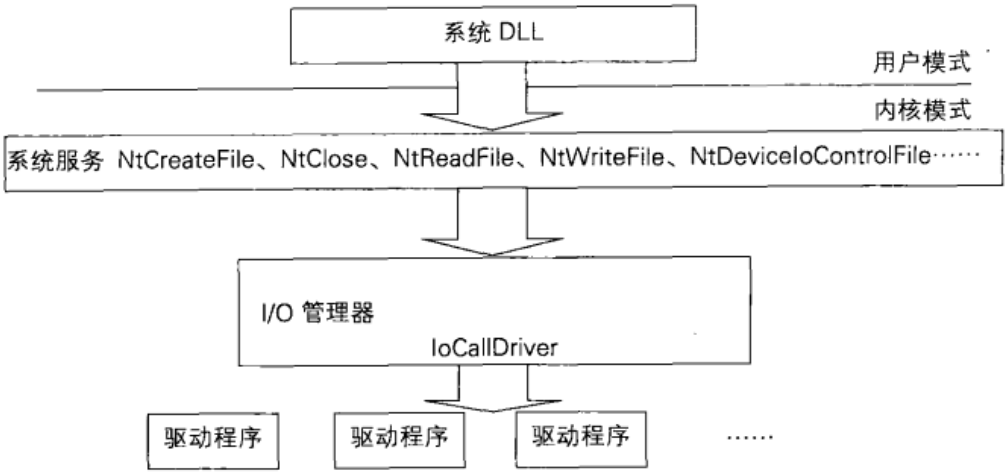


图 6.19 来自应用程序的 I/O 请求的控制流路径

下面我们以 IOCTL 例子中的代码来说明用户程序发起的 I/O 请求如何被交给驱动程序进行处理。我们首先看一下用户程序的操作，分三个步骤进行。首先，用户程序通过 CreateFile 函数创建一个指向“\\.\IoctlTest”设备对象的文件对象，该设备对象是在驱动程序被加载并初始化时创建的，并且已被注册在对象管理器中。在这个步骤中，对应的系统服务是 NtCreateFile 函数。然后，用户程序通过调用 DeviceIoControl，向设备对象发出 I/O 请求，对应的系统服务是 NtDeviceIoControlFile 函数。最后，当用户程序不再使用该设备对象时，调用 CloseHandle 函数关闭文件对象，这里对应的系统服务是 NtClose 函数。

NtCreateFile 函数直接将调用转发给 IoCreateFile 函数，IoCreateFile 函数除了支持文件对象的创建，也支持命名管道和邮件槽对象的创建。正如 6.2.3 节所述，IoCreateFile 调用 IopCreateFile 函数，而 IopCreateFile 进一步调用 ObOpenObjectByName 函数来创建文件对象。如图 6.7 所示，对象管理器通过逐层解析文件对象名称，最终在 IopParseDevice 函数中为指定的设备对象创建一个文件对象。IopParseDevice 除了构造一个文件对象，还调用了 IoCallDriver 函数，向驱动程序发送一个“IRP_MJ_CREATE”请求，以通知该驱动程序要创建该设备对象的一个文件对象。因此，驱动程序会接收到此 IRP_MJ_CREATE

请求。为了使文件对象创建成功，驱动程序必须返回成功，譬如，sioctl.sys 驱动程序中的处理函数 SioctlCreateClose 返回 STATUS_SUCCESS。如果此 I/O 请求不成功，则无法成功地创建文件对象。

关闭一个文件对象的过程要简单得多，NtClose 系统服务函数只是简单地调用 ObpCloseHandle 函数，通过对象管理器的标准机制来关闭一个文件对象。ObpCloseHandle 调用 ObpCloseHandleTableEntry 函数来关闭一个句柄项所引用的对象。如果类型对象中指定了 OkayToCloseProcedure 例程，则 ObpCloseHandleTableEntry 首先询问该例程，是否可以关闭此对象的句柄，若回答可以，则继续下面的动作，否则返回 STATUS_HANDLE_NOT_CLOSABLE。文件对象的类型对象并没有指定此例程，所以，ObpCloseHandleTableEntry 忽略这一步骤。然后，ObpCloseHandleTableEntry 调用 ObpDecrementHandleCount 以递减对象的句柄计数，在 ObpDecrementHandleCount 函数中，它调用类型对象的 CloseProcedure 例程。由于文件对象类型对象的 CloseProcedure 例程被指定为 IopCloseFile，所以，在关闭文件对象时，IopCloseFile 函数被调用。ObpCloseHandleTableEntry 最后调用 ObDereferenceObject 函数（实际上是 ObfDereferenceObject 函数），以管理文件对象的生命周期。当文件对象的引用计数为 0 时，ObfDereferenceObject 调用 ObpRemoveObjectRoutine 函数，而 ObpRemoveObjectRoutine 又进一步调用类型对象的 DeleteProcedure 例程。在文件对象的类型对象中，此 DeleteProcedure 例程被指定为 IopDeleteFile 函数。所以，当文件对象被关闭时，若引用计数为 0，则 IopDeleteFile 函数也会被调用。

关于以上提到的对象管理器函数，包括 ObpCloseHandle、ObpCloseHandleTableEntry、ObpDecrementHandleCount、ObfDereferenceObject 和 ObpRemoveObjectRoutine，请参考 WRK 中相应的代码，这里不再赘述。我们现在知道，文件对象类型对象的 CloseProcedure 例程一定被调用到，而 DeleteProcedure 例程则在引用计数被减到 0 的情况下也会被调用。

IopCloseFile 函数向文件对象的设备对象发送一个 IRP_MJ_LOCK_CONTROL 请求，次功能码为 IRP_MN_UNLOCK_ALL，指示驱动程序释放所有的锁。IopDeleteFile 函数向文件对象的设备对象发送一个 IRP_MJ_CLOSE 请求，告诉驱动程序关闭此文件对象。只要驱动程序的初始化例程注册了这两个分发例程，它们就会在文件对象的关闭过程中被调用。Sioctl.sys 驱动程序没有注册 IRP_MJ_LOCK_CONTROL 的分发函数，所以 I/O 管理器默认将它设置为 IopInvalidDeviceRequest 函数；sioctl.sys 驱动程序注册的 IRP_MJ_CLOSE 例程是 SioctlCreateClose 函数，它只是简单地请求完成并返回 STATUS_SUCCESS。

以上介绍了应用程序在创建和关闭文件对象过程中，通过 Windows 的系统服务与 I/O 管理器以及设备驱动程序打交道的过程。通过进一步检查 IopParseDevice、IopCloseFile

和 IopDeleteFile 函数的代码，我们可以了解到，这些函数调用 IoAllocateIrp（或 IopAllocateIrpMustSucceed）来构造 IRP 对象，并初始化 IRP 对象的属性，包括构造第一个栈单元，然后调用 IoCallDriver，将 I/O 请求发送给驱动程序。在 Windows 中，IoCallDriver 只是简单地将函数调用转发给 IofCallDriver，进而转发给 IopfCallDriver，为便于理解，以下列出 IopfCallDriver 函数的代码：

```
NTSTATUS
FORCEINLINE
IopfCallDriver(
    IN PDEVICE_OBJECT DeviceObject,
    IN OUT PIRP Irp
    )
{
    PIO_STACK_LOCATION irpSp;
    PDRIVER_OBJECT driverObject;
    NTSTATUS status;

    // IRP 对象的类型必须是 IO_TYPE_IRP
    ASSERT( Irp->Type == IO_TYPE_IRP );

    // 更新 IRP 的栈单元指针，使其指向下一个位置，所以，在调用 IoCallDriver 以前，必须
    // 小心地设置好栈单元指针的位置，使其指向将使用的栈单元位置加 1
    Irp->CurrentLocation--;

    if (Irp->CurrentLocation <= 0) {
        KiBugCheck3( NO_MORE_IRP_STACK_LOCATIONS, (ULONG_PTR) Irp, 0, 0 );
    }

    irpSp = IoGetNextIrpStackLocation( Irp );
    Irp->Tail.Overlay.CurrentStackLocation = irpSp;

    // 在栈单元中记录下设备对象的指针，以便在完成例程中可以访问设备对象
    irpSp->DeviceObject = DeviceObject;

    // 调用驱动程序中对应的分发例程
    driverObject = DeviceObject->DriverObject;
    status = driverObject->MajorFunction[irpSp->MajorFunction] ( DeviceObject,Irp );
    return status;          //返回分发函数的处理结果
}
```

应用程序与 I/O 设备打交道最常用的系统服务是 NtReadFile、NtWriteFile 和 NtDeviceIoControlFile，与之对应的 I/O 请求的主功能码分别是 IRP_MJ_READ、IRP_MJ_WRITE 和 IRP_MJ_DEVICE_CONTROL（或 IRP_MJ_FILE_SYSTEM_CONTROL）。应用程序通过 IRP_MJ_DEVICE_CONTROL 主功能码与驱动程序进行通信是一种常见的做法，它允许驱动程序自行定义次功能码（使用 CTL_CODE 宏）。IOCTL 例子演示了这一用法，它定义的次功能码分别对应于 I/O 管理器使用的 4 种不同的 IRP 缓冲区管理方法。Ioctlapp.exe 在调用 API 函数 DeviceIoControl 时指定了次功能码，因而 sioctl.sys 在接收到 IRP_MJ_DEVICE_CONTROL 请求时，知道如何使用缓冲区。关于 IRP 缓冲区的管理，参见本章后面 6.6.3 节的介绍。

接下来我们介绍 NtDeviceIoControlFile 的处理流程。NtDeviceIoControlFile 调用一个更为通用的函数 IopXxxControlFile，IopXxxControlFile 既处理来自 NtDeviceIoControlFile 的 IRP_MJ_DEVICE_CONTROL 请求，也处理来自 NtFsControlFile 的 IRP_MJ_FILE_SYSTEM_CONTROL 请求。IopXxxControlFile 函数的原型如下：

```
NTSTATUS
IopXxxControlFile(
    IN HANDLE FileHandle,                          // 此 I/O 请求的目标文件对象的句柄
    IN HANDLE Event OPTIONAL,                      // 可选参数，当 I/O 请求完成时，此事件对象有信号
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,        // 可选 APC 例程，当 I/O 请求完成时，它被执行
    IN PVOID ApcContext OPTIONAL,                  // 传递给 APC 例程的环境参数
    OUT PIO_STATUS_BLOCK IoStatusBlock,            // 调用者提供的状态块地址
    IN ULONG IoControlCode,                        // 次功能码，确定要执行哪个子命令
    IN PVOID InputBuffer OPTIONAL,                 // 可选参数，提供给驱动程序的输入缓冲区
    IN ULONG InputBufferLength,                    // 输入缓冲区的长度，以字节为单位
    OUT PVOID OutputBuffer OPTIONAL,               // 可选参数，输出缓冲区，以接收驱动程序的处理结果
    IN ULONG OutputBufferLength,                   // 输出缓冲区的长度
    IN BOOLEAN DeviceIoControl                     // 是否为 IRP_MJ_DEVICE_CONTROL 请求
    );
```

IopXxxControlFile 函数的代码位于 base\ntos\io\iomgr\internal.c 文件的 8 482~9 222 行。它首先检查参数的有效性，然后利用 ObReferenceObjectByHandle 函数将文件句柄转换成对文件对象的引用。接下来检查调用者是否有权限访问此子功能。然后解出通知事件对象的引用。如果这是一个同步 I/O 请求，则必须获取文件对象锁之后才能进行。

做好以上这些准备工作以后，IopXxxControlFile 获得与文件对象关联的设备对象。对于 IRP_MJ_DEVICE_CONTROL 请求，它进一步判断设备对象的驱动程序是否支持快速 I/O，这里快速 I/O 是指不需要构造 IRP 对象就可以直接调用驱动程序对象的 FastIoDispatch 成员中的例程，参见下一小节的介绍。若驱动程序支持快速 I/O，则 IopXxxControlFile 调用 FastIoDispatch 成员中的 FastIoDeviceControl 例程，然后执行完成处理，包括：设置状态块，根据需要通知参数中指定的 Event 事件对象，若这是同步 I/O 则释放文件对象锁，若文件对象支持 I/O 完成端口则通知完成端口的队列，最后解除文件对象的引用并返回。

若驱动程序不支持快速 I/O，则 IopXxxControlFile 必须构造一个 IRP 对象，并初始化该 IRP 对象和第一个栈单元，再根据次功能码的最后两位所指定的缓冲区管理方法，为该 I/O 请求准备好缓冲区，最后，调用 IopSynchronousServiceTail 函数以发送 I/O 请求。

IopSynchronousServiceTail 函数的代码位于 internal.c 文件的 7 417~7 574 行。它首先将异步 I/O 请求插入到当前线程的 IRP 链表中。然后，更新当前进程的 I/O 操作计数器。接下来调用 IoCallDriver 函数，将控制权交给驱动程序中的分发例程。然后 IopSynchronousServiceTail 根据参数 DeferredIoCompletion 中指定的布尔值，确定是否为延迟 I/O 完成的情形。对于 IRP_MJ_DEVICE_CONTROL 请求，DeferredIoCompletion 为

FALSE，所以忽略这一步。最后，若这是同步 I/O 请求，并且驱动程序返回 STATUS_PENDING，那么，IopSynchronousServiceTail 函数调用 KeWaitForSingleObject，在文件对象的事件对象（即 Event 域）上等待，直至该事件对象有信号，或者此 I/O 请求被取消。然后，函数返回，NtDeviceIoControlFile 系统服务也随之完成。

以上我们分析了系统服务 NtDeviceIoControlFile 的处理过程。表 6.5 列出了以类似方式来实现 I/O 功能的系统服务。这些系统服务的实现模式是，首先检查参数的有效性，有可能的话使用快速 I/O；然后，构造一个 IRP 对象，填充 IRP 对象中的状态信息，包括第一个栈单元；最后调用 IopSynchronousServiceTail 函数完成 I/O 操作。WRK 包含了所有这些函数的代码，有兴趣的读者可以参考这些函数的实现。

表 6.5 调用 IopSynchronousServiceTail 来实现 I/O 功能的系统服务

NtDeviceIoControlFile	NtFlushBuffersFile	NtFsControlFile
NtLockFile	NtNotifyChangeDirectoryFile	NtQueryDirectoryFile
NtQueryEaFile	NtQueryQuotaInformationFile	NtQueryVolumeInformationFile
NtReadFile	NtReadFileScatter	NtSetEaFile
NtSetQuotaInformationFile	NtSetVolumeInformationFile	NtUnlockFile
NtWriteFile	NtWriteFileGather	

除了这些系统服务需要发起 I/O 请求，内核中的各个模块以及驱动程序本身也会调用 IoCallDriver 来发出 I/O 请求。即使在 I/O 系统初始化过程中，它也会通过 IoCallDriver 与已经加载的驱动程序打交道，譬如，在即插即用管理器列举设备时，它与总线驱动程序和设备驱动程序进行紧密的通信才能建立起设备树。像文件系统服务以及其他依赖于设备驱动程序的服务，都通过 IoCallDriver 来发起 I/O 请求。由于 IoCallDriver 函数带一个设备对象和一个 IRP 对象作为参数，因此，任何内核模块若需要发起 I/O 请求，则必须准备好设备对象和 IRP 对象。上一小节曾经提到，IRP 对象可通过 IoAllocateIrp 函数（实际上是 IopAllocateIrpPrivate）来分配。此外，I/O 系统还提供了三个包装函数用来构造 IRP 对象：IoBuildAsynchronousFsdRequest、IoBuildSynchronousFsdRequest 和 IoBuildDeviceIoControlRequest，它们可用于构造某些具有特定用途的 IRP 对象。这些函数的代码位于 base\ntos\io\iomgr\iosubs.c 文件中，Windows DDK 包含了关于这些函数的说明文档。

6.6.3 处理 I/O 请求过程中的事项

上一小节介绍了一个 I/O 请求的基本处理流程：Windows 的系统服务例程接收到来自应用程序的 I/O 请求，然后，它们或者通过快速 I/O 通道处理此 I/O 请求，或者通过 I/O 管理器的 IoCallDriver 函数将它交给驱动程序进行处理。这一小节讨论在此处理过程中的

一些特别事项，包括快速 I/O 处理、缓冲区管理、异步 I/O、I/O 请求的完成过程，以及 I/O 请求的取消。

(1) 快速 I/O。

正如我们在 IopXxxControlFile 函数中所看到的那样，如果一个驱动程序对象的 FastIoDispatch 成员指向一个有效的分发结构，并且其 FastIoDeviceControl 成员为非空例程，那么，IopXxxControlFile 无须构造 IRP 对象和调用 IoCallDriver，而是直接调用 FastIoDeviceControl 例程来完成此 NtDeviceIoControlFile 系统服务。下面是 FastIoDispatch 成员的数据类型 FAST_IO_DISPATCH 的定义：

```
typedef struct _FAST_IO_DISPATCH {
    ULONG SizeOfFastIoDispatch;
    PFAST_IO_CHECK_IF_POSSIBLE FastIoCheckIfPossible;
    PFAST_IO_READ FastIoRead;
    PFAST_IO_WRITE FastIoWrite;
    PFAST_IO_QUERY_BASIC_INFO FastIoQueryBasicInfo;
    PFAST_IO_QUERY_STANDARD_INFO FastIoQueryStandardInfo;
    PFAST_IO_LOCK FastIoLock;
    PFAST_IO_UNLOCK_SINGLE FastIoUnlockSingle;
    PFAST_IO_UNLOCK_ALL FastIoUnlockAll;
    PFAST_IO_UNLOCK_ALL_BY_KEY FastIoUnlockAllByKey;
    PFAST_IO_DEVICE_CONTROL FastIoDeviceControl;
    PFAST_IO_ACQUIRE_FILE AcquireFileForNtCreateSection;
    PFAST_IO_RELEASE_FILE ReleaseFileForNtCreateSection;
    PFAST_IO_DETACH_DEVICE FastIoDetachDevice;
    PFAST_IO_QUERY_NETWORK_OPEN_INFO FastIoQueryNetworkOpenInfo;
    PFAST_IO_ACQUIRE_FOR_MOD_WRITE AcquireForModWrite;
    PFAST_IO_MDL_READ MdlRead;
    PFAST_IO_MDL_READ_COMPLETE MdlReadComplete;
    PFAST_IO_PREPARE_MDL_WRITE PrepareMdlWrite;
    PFAST_IO_MDL_WRITE_COMPLETE MdlWriteComplete;
    PFAST_IO_READ_COMPRESSED FastIoReadCompressed;
    PFAST_IO_WRITE_COMPRESSED FastIoWriteCompressed;
    PFAST_IO_MDL_READ_COMPLETE_COMPRESSED MdlReadCompleteCompressed;
    PFAST_IO_MDL_WRITE_COMPLETE_COMPRESSED MdlWriteCompleteCompressed;
    PFAST_IO_QUERY_OPEN FastIoQueryOpen;
    PFAST_IO_RELEASE_FOR_MOD_WRITE ReleaseForModWrite;
    PFAST_IO_ACQUIRE_FOR_CCFLUSH AcquireForCcFlush;
    PFAST_IO_RELEASE_FOR_CCFLUSH ReleaseForCcFlush;
} FAST_IO_DISPATCH, *PFAST_IO_DISPATCH;
```

通常只有文件系统或网络传输驱动程序使用快速 I/O 来提供快速的数据访问。对于同步的读或写操作，如果要访问的目标文件已被缓存，那么，NtReadFile 和 NtWriteFile 使用快速 I/O 来完成该操作，此时文件系统驱动程序通过 Windows 系统的缓存管理器（cache manager）来访问文件数据。若文件系统驱动程序的快速 I/O 例程认为无法通过缓存管理器来完成 I/O 操作，则 NtReadFile 和 NtWriteFile 仍然使用 IRP 对象的方式，通过常规的分发例程来完成 I/O 操作。读者可以参考 WRK 中 NtReadFile 和 NtWriteFile 函数的代码。

如同常规的分发例程一样，驱动程序的快速 I/O 例程也是在初始化例程中指定的，FAST_IO_DISPATCH 结构中的每个例程都有特定的用途。这些快速 I/O 例程的功能代表

了 I/O 操作的某些特定方面，它们的函数原型也不尽相同。这里不详细解释每一个快速 I/O 例程，有兴趣的读者可以在 WRK 中找到大多数例程的调用点。如果读者想知道如何实现这些快速 I/O 例程，由于 WRK 中并未包含文件系统驱动程序的代码，建议读者参考 Windows DDK 随带的文件系统驱动程序例子（位于 src\filesys 目录下），例如 cdfs 和 fastfat 文件系统驱动程序。

(2) 缓冲区管理。

当应用程序或者内核模块通过 I/O 系统提供的服务与驱动程序进行通信时，I/O 请求的发起者与驱动程序必须使用一致的缓冲区管理方法，才能正确地传递数据。在 IOCTL 例子中，我们可以看到 I/O 管理器支持的 4 种缓冲区管理方法：

- **缓冲的 I/O**（宏 METHOD_BUFFERED）。I/O 请求的发起者从非换页内存池中申请一块内存，可容纳输入缓冲区或者输出缓冲区。对于调用者提供的输入缓冲区，I/O 请求的发起者在调用 IoCallDriver 函数以前，将输入缓冲区中的数据拷贝到这一新申请的内存中，参考 IopXxxControlFile 函数中的做法；对于调用者提供的输出缓冲区，I/O 管理器在完成了 I/O 请求以后，将新内存块中的数据拷贝到输出缓冲区中，然后释放所申请的内存，这一拷贝和释放内存块的工作是在 IopCompleteRequest 函数中完成的，参见 internals.c 文件中该函数的代码。
- **直接输入 I/O**（METHOD_IN_DIRECT）。对于从设备读取数据的操作（例如 NtReadFile），I/O 请求的发起者调用 IoAllocateMdl 函数申请一个 MDL（Memory Descriptor List，内存描述符链表，参见 4.5.3 节的介绍），将调用者指定的缓冲区的物理页面构成一个 MDL，以便于设备驱动程序使用 DMA 方式来传输数据。IRP 对象中的 MdlAddress 域记录了一个 I/O 请求所使用的 MDL。
- **直接输出 I/O**（METHOD_OUT_DIRECT）。类似地，对于往设备写数据的操作（例如 NtWriteFile），I/O 请求的发起者也调用 IoAllocateMdl 函数申请一个 MDL，将参数中指定的输出缓冲区的物理页面构成一个 MDL，以便于设备驱动程序使用 DMA 方式来传输数据。
- **两者皆非 I/O**（METHOD_NEITHER）。I/O 管理器或者 I/O 请求的发起者不负责缓冲区管理工作，而由驱动程序自行决定应该如何使用缓冲区。譬如，在 NtDeviceIoControlFile 系统服务执行过程中，IopXxxControlFile 函数对于这种类型的缓冲区管理方法，只是简单地将用户的输入缓冲区指针存放在 IRP 对象的 Parameters.DeviceIoControl.Type3InputBuffer 参数中，将输出缓冲区的指针存放在 IRP 对象的 UserBuffer 域中，而对缓冲区本身不做任何处理。

I/O 请求的发起者在确定了缓冲区管理方法以后，把输入或/和输出缓冲区的地址保存在 IRP 对象中。设备驱动程序可以通过设备对象的 Flags 域指示 I/O 管理器它支持哪些缓冲区管理方法，而且，它在接收到读或写 I/O 请求时，根据 IRP 对象的 Flags 域以及缓冲区指针值来确定使用哪种缓冲区管理方法。NtDeviceIoControlFile 系统服务允许调用者同时支持输入和输出缓冲区，调用者通过它的子功能码参数 IoControlCode 的最低 2 位来指示该系统服务和驱动程序如何处理输入和输出缓冲区。如果调用者指定了直接 I/O 方法（METHOD_IN_DIRECT 或 METHOD_OUT_DIRECT），那么，NtDeviceIoControlFile 为输入缓冲区使用“缓冲的 I/O”方法，即通过 IRP 对象的 AssociatedIrp.SystemBuffer 成员来传递；而对输出缓冲区使用 MDL，即通过 IRP 对象的 MdlAddress 来传递。

在 IOCTL 例子中，当驱动程序接收到直接输入 I/O 或直接输出 I/O（子功能码为 IOCTL_SIOCTL_METHOD_IN_DIRECT 或 IOCTL_SIOCTL_METHOD_OUT_DIRECT）请求时，它利用 Irp->AssociatedIrp.SystemBuffer 可获得调用者的输入缓冲区，利用 Irp->MdlAddress 可得到输出缓冲区的 MDL；若要访问输出缓冲区，则通过 MmGetSystemAddressForMdlSafe 函数可获得输出缓冲区页面被映射在系统地址空间中的虚拟地址。此外，IOCTL 例子中处理 METHOD_NEITHER 情形的代码是一种典型的处理方式，以下两点值得特别注意：

- 利用 ProbeForRead 或 ProbeForWrite 来检查缓冲区是否有效，以及在当前执行环境中是否可以访问。
- 如果驱动程序需要在其他的执行环境中访问这些缓冲区，比如在 DPC 或 ISR（中断服务例程）中，那么，驱动程序必须首先通过 IoAllocateMdl 申请一个 MDL，并利用 MmProbeAndLockPages 将缓冲区中的页面锁定在内存中。然后，或者直接将 MDL 传递给这些执行环境的模块，或者经过 MmGetSystemAddressForMdlSafe 调用以后传递一个系统地址空间的虚拟地址。最后，当 I/O 请求完成以后，驱动程序应调用 MmUnlockPages 和 IoFreeMdl 以解除页面锁定并释放 MDL。

(3) 同步和异步 I/O。

从 I/O 请求的发起者角度来看，一个 I/O 请求既可能是同步的，也可能是异步的。对于同步 I/O，发起者只有等到 I/O 请求完成以后，才会重获控制权；而对于异步 I/O，I/O 管理器可以在 I/O 请求尚未完成的情况下将控制权交还给发起者。虽然应用程序看到的大多数 I/O 请求是同步的，但是，Windows 的 I/O 模型本质上是异步的。异步 I/O 可以提升一个应用程序的处理能力和响应能力，因为在 I/O 请求进行时，应用程序的线程可以不用等待 I/O 完成，转而继续执行其他的任务。

像 NtDeviceIoControlFile、NtReadFile 和 NtWriteFile 这样的系统服务（如表 6.5 所列），最简单的形式是以同步方式执行 I/O 操作，当控制权从这些系统服务返回到调用者时，I/O 操作已经完成了。然而，这些服务也支持以异步方式执行 I/O 操作。为了使用同步 I/O，当应用程序在创建文件对象时，NtCreateFile 系统服务接收到的 CreateOptions 参数含有 FILE_SYNCHRONOUS_IO_NONALERT（或 FILE_SYNCHRONOUS_IO_ALERT）项，对应于应用程序在调用 API 函数 CreateFile 时，不能在 dwFlagsAndAttributes 参数中包含 FILE_FLAG_OVERLAPPED 标志。参考 6.2.3 节关于文件对象创建过程的介绍，其中 FILE_SYNCHRONOUS_IO_NONALERT 标志项被从 IopCreateFile 函数中经过 ObOpenObjectByName 和 ObpLookupObjectName，最终传递到 IopParseDevice 函数中，并演变成为文件对象的 Flags 域中的 FO_SYNCHRONOUS_IO 项。参见 IopParseDevice 函数的代码。简而言之，若应用程序在 CreateFile 调用中不包含 FILE_FLAG_OVERLAPPED 标志，则创建得到的文件对象的 Flags 域中含有 FO_SYNCHRONOUS_IO 项，说明这是一个同步的文件对象；反之，若 CreateFile 调用中指定了 FILE_FLAG_OVERLAPPED 标志，则文件对象的 Flags 域不包含 FO_SYNCHRONOUS_IO 项，说明这是一个异步操作的文件对象。

NtReadFile 和 NtWriteFile 等系统服务在得到了文件对象以后，根据其 Flags 域中是否含有 FO_SYNCHRONOUS_IO 项来确定这是一个同步或异步 I/O。它们在调用 IopSynchronousServiceTail 函数时，传递给它一个布尔参数以指明这是同步 I/O 还是异步 I/O。在 IopSynchronousServiceTail 函数中，当 IoCallDriver 函数返回以后，若这是一个同步 I/O，但 IoCallDriver 的返回值为 STATUS_PENDING，则说明此 I/O 尚未真正完成。在这种情况下，IopSynchronousServiceTail 在文件对象的事件对象（即 Event 域）上等待，当驱动程序真正完成了该 I/O 时，完成例程应该将该事件对象置成有信号状态，从而等待可以结束。这是一种典型的将驱动程序的异步 I/O 处理转变成同步 I/O 的做法。另外，对一个文件对象的同步 I/O 操作，必须要获得文件对象的锁（通过 IopAcquireFileObjectLock 函数），当 I/O 操作完成以后，再释放该锁（通过 IopReleaseFileObjectLock 函数）。这样可以确保一个同步方式的文件对象上的 I/O 操作都是顺序进行的。

在 IopSynchronousServiceTail 中，当 IoCallDriver 函数返回以后，若这是一个异步 I/O，那么 IopSynchronousServiceTail 函数返回，从而相应的系统服务也返回。以后，当 I/O 完成时，系统服务的调用者指定的 APC 例程将会被执行，从而应用程序得到完成通知。

在设备驱动程序中，当接收到一个 I/O 请求时，总是应该尽快地完成自己的职责，然后将控制返回给 I/O 管理器；若需要发出一个 I/O 请求，则发出请求后应立即返回 STATUS_PENDING。至于发出的 I/O 请求何时完成，以及如何指定完成处理，本节后面将会介绍。

(4) I/O 请求的完成和未完成的 IRP。

由于 I/O 处理的异步本质，IoCallDriver 函数返回并不意味着一个 I/O 请求已经完成了，相反地，Windows 的 I/O 系统有一套单独的 I/O 请求完成机制。例如，对于采用中断通知方式的设备，当设备驱动程序发出了硬件 I/O 请求以后，接收 I/O 请求的分发函数便返回了。现在我们来看设备驱动程序的完成处理。

当硬件设备完成了 I/O 时，它以中断方式通知处理器，此 I/O 任务已经完成，于是驱动程序中注册的 ISR（中断服务例程）获得处理器执行权。在 ISR 中，典型的做法是，捕获或设置硬件设备的状态，然后将真正的数据处理交给一个 DPC 例程来完成，所以，它会插入一个 DPC 对象到处理器的 DPC 链表中，然后退出。当该 DPC 例程被执行时，驱动程序再进一步处理并最终完成 I/O 请求，也有可能启动下一个正在等待的 I/O 请求。驱动程序调用 IoCompleteRequest 函数来通知 I/O 管理器一个 I/O 请求已经完成，因此 I/O 管理器可以根据该 I/O 请求的特性，包括同步或异步性质、缓冲区管理方法等，以恰当的方式通知发起者线程，必要时将数据复制到发起者指定的缓冲区中。在最后一个步骤中，I/O 管理器可能要在发起者线程中插入一个 APC，以便能够访问发起者线程所在的进程地址空间。图 6.20 显示了这样的 I/O 请求的完成路径。

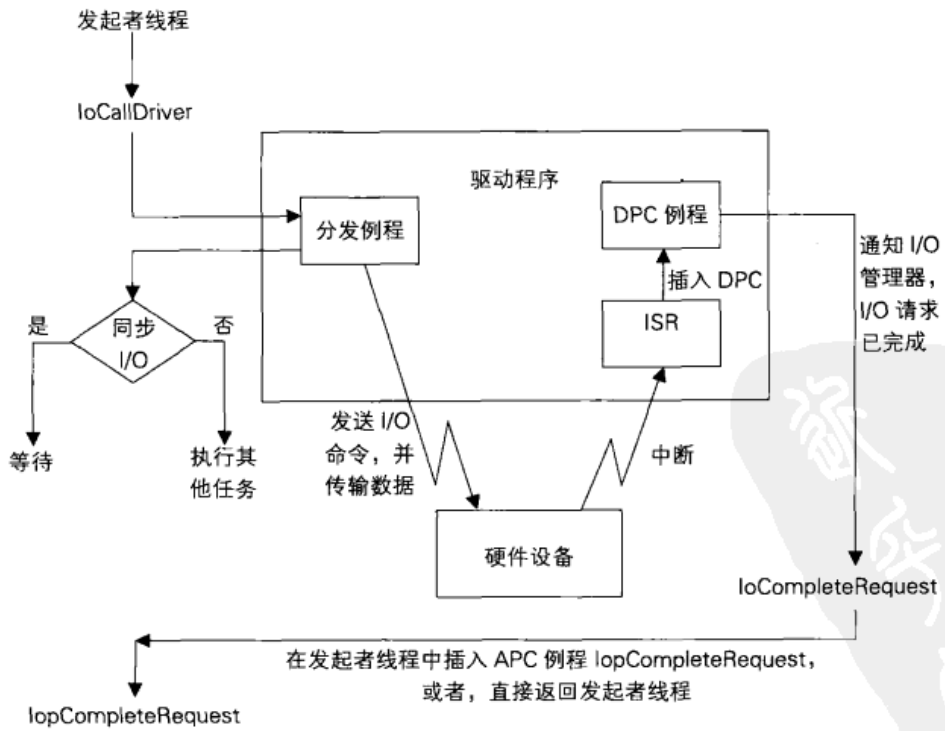


图 6.20 以硬件中断方式通知 I/O 请求已完成的执行路径

为了提高设备的吞吐量，设备驱动程序可以把接收到的 I/O 请求存放在一个队列中，因此，当在 DPC 中完成了一个 I/O 请求时，它转而指示设备处理队列中的下一个 I/O 请求。这种将 ISR 与 DPC 分开的做法，其好处是，处理器在设备 IRQL 上的时间可以尽可能地短，由于 DPC 的 IRQL 低于设备 IRQL，因而其他工作在低 IRQL 上的设备有更多的机会中断处理器，及时地完成它们的 I/O 任务。

每个 I/O 请求分别由单独的 IRP 对象来表达，这是确保一个 I/O 请求能以完全异步方式来进行处理的基础。对于一个驱动程序，ISR 与 DPC 例程，或者其他的分发例程有可能被并发执行，而在多处理器系统中，甚至有可能被并行执行，因此，一旦这些例程需要访问共享数据，则它们必须对共享数据进行同步。驱动程序可以采用提升 IRQL 或者自旋锁的做法来保护共享数据。

Windows 内核提供了一个专门用于中断对象的同步辅助例程 KeSynchronizeExecution，当驱动程序需要访问其 ISR 中也要访问的数据时，它应该调用 KeSynchronizeExecution 函数来访问此数据。KeSynchronizeExecution 的实现并不复杂，其代码位于 base\ntos\ke\i386\intsup.asm 文件中。它首先将 IRQL 提升到中断对象 KINTERRUPT 结构的 SynchronizeIrql 域指定的中断级别上，若是多处理器内核，则还需要获得中断对象的 ActualLock 域指定的自旋锁；然后，调用 SynchronizeRoutine 参数指定的例程，待该例程返回后，再释放自旋锁和恢复 IRQL。若驱动程序不使用 KeSynchronizeExecution 函数，也可以使用 KeAcquireInterruptSpinLock/KeReleaseInterruptSpinLock 函数，实现与 ISR 的数据同步。这两个函数中的同步原理与 KeSynchronizeExecution 相同，这里不再赘述。

现在来看 I/O 请求的完成路径。与 IoCallDriver 的情形类似，真正完成 IoCompleteRequest 函数的功能是在一个名为 IopfCompleteRequest 的函数中。若驱动程序中的 I/O 请求分发例程调用 IoCompleteRequest 函数，则表示这一 I/O 请求已经完成。通常，DPC 例程会调用 IoCompleteRequest 函数来完成一个 I/O 请求。譬如，当应用程序通过 NtReadFile 从设备读取数据时，一旦数据传输完毕，ISR 就会插入一个 DPC 对象，该 DPC 对象的延迟例程会调用 IoCompleteRequest 函数来通知 I/O 管理器此 I/O 已完成，不管应用程序的线程是否在等待此 I/O，都会被以某种方式通知到。下面是 IopfCompleteRequest 函数的原型：

```
VOID
FASTCALL
IopfCompleteRequest(
    IN PIRP Irp,
    IN CCHAR PriorityBoost
    );
```

这里 Irp 参数代表了待完成的 I/O 请求，而 PriorityBoost 参数则是要通知的目标线程在

获得调度时的优先级提升值，实际上是向目标线程插入 APC 例程时所指定的优先级提升值。IopfCompleteRequest 函数的代码位于 base\ntos\io\iomgr\iosubs.c 文件中，其功能如下：

- 首先依次检查 IRP 对象的栈单元，并调用栈单元中已设定的完成例程。对于设备栈中的 IRP 对象，每个参与 I/O 处理的驱动程序在接收到 I/O 请求时，都可以通过 IoSetCompletionRoutine 函数来设置完成例程，以便当 I/O 请求被完成时，有机会做一些资源释放或者状态恢复等工作。
- 检查当前 IRP 是否为一个关联的 IRP 对象。主 IRP 与关联 IRP 是一种“主-从（master-slave）”关系，譬如，文件系统在接收到一个读文件请求时，由于此读操作涉及磁盘上不同位置处的数据，因此，它可以创建多个关联 IRP，每个 IRP 从不同的扇区读取一部分数据。如果当前 IRP 是一个关联 IRP（在 Flags 中有 IRP_ASSOCIATED_IRP 标记），那么，它的完成将导致主 IRP 的计数（即它的 AssociatedIrp.IrpCount 域）减 1，若此计数已减到 0，则意味着主 IRP 也已经完成，于是再调用 IoCompleteRequest 函数，让主 IRP 也完成。
- 如果这是一个换页 I/O 操作，或者一个关闭（Close）I/O，则只是简单地设置 I/O 状态已完成，并使该 IRP 的事件对象变成有信号状态。对于换页 I/O 请求，在这里回收 IRP 对象的内存资源。
- 如果此 IRP 使用了“直接 I/O”缓冲区管理方法（IRP 对象的 MdlAddress 域非空），则调用 MmUnlockPages 解除物理页面的锁定。
- 确定此 I/O 请求是否是一个要在发起者线程环境中完成的同步 I/O，若是，则 IopfCompleteRequest 函数立即返回，而由该 I/O 请求的发起者来完成它。在这种情况下，当前线程必定是发起者线程，因而可以避免使用 APC 机制来通知完成，从而消除了由 APC 机制带来的额外开销（包括软中断、处理器的线程切换等）。注意，发起者线程需要调用 IopCompleteRequest 函数，以进一步完成此 I/O 请求，读者可以参考 IopSynchronousServiceTail 函数中 IoCallDriver 返回后的进一步处理。
- 最后，在发起者线程中插入一个 APC 对象以便通知该线程此 I/O 已完成，其中 APC 例程为 IopCompleteRequest。或者，若此 IRP 对象的发起者线程已不存在，则调用 IopDropIrp 函数释放所有被该 IRP 占有的资源。

我们看到，IRP 请求的最后完成步骤是，APC 例程 IopCompleteRequest 在发起者线程中被执行。有些跟发起者线程相关的任务必须在这一步骤中完成。IopCompleteRequest 函数的代码位于 base\ntos\io\iomgr\internal.c 文件中，其功能如下：

- 从参数中获得当前 IRP 对象以及文件对象和线程信息。
- 若有必要，将数据拷贝到用户缓冲区中，并释放系统缓冲区的内存。
- 若有必要，释放由 MdlAddress 指示的 MDL。
- 判断该 I/O 请求是否在处理过程中出错，如果发生错误，则执行必要的错误处理，包括解除正在等待的线程、解除对文件对象和用户事件对象的引用、将 IRP 从线程的未完成 I/O 请求链表中移除、释放 IRP 对象等。若处理过程未出错，那么：
  - 将 I/O 状态信息从 IRP 对象拷贝到发起者指定的 I/O 状态块中。
  - 若发起者在 IRP 对象中指定了用户事件对象，则使其变成有信号状态；否则设置文件对象的事件对象（即 Event 域）为有信号状态。
  - 更新进程的 I/O 操作计数，并且将 IRP 对象从线程的未完成 I/O 请求链表中移除。
  - 若 I/O 请求的发起者要求通过一个 APC 来通知它，则将 APC 插入发起者线程；若文件对象关联了一个 I/O 完成端口，则将 IRP 对象插入到 I/O 完成端口中；否则，简单地释放 IRP 对象。关于 I/O 完成端口，参见 6.6.5 节的介绍。
  - 除非是一个创建文件对象的请求，否则解除对文件对象的引用。

从以上的 I/O 请求的完成处理我们可以看出，I/O 管理器依赖于 APC 机制将设备的行为与发起者线程联系起来，设备驱动程序除了遵从约定的缓冲区管理方法以外，并不需要特别通知请求线程 I/O 管理器负责相关的跨线程通信事宜。而对于异步 I/O 请求，发起者线程可以指定自己的 APC 例程来接收完成通知。应用程序可以指定用户模式 APC 来接收完成通知，这正是 NtReadFile 和 NtWriteFile 所提供的异步数据传输功能，对应的 Windows API 函数为 ReadFileEx 和 WriteFileEx。

图 6.20 显示的 I/O 请求针对硬件设备的驱动程序，而对于内核扩展类型的驱动程序，通常 I/O 请求可以直接在分发例程中完成。在这种情况下，分发例程在完成了应用程序指定的 I/O 任务或只有在内核中才能完成的任务以后，调用 IoCompleteRequest 函数以完成此次请求。APC 例程 IopCompleteRequest 同样也会被执行，只不过这一切都发生在同一个线程中。

每一个线程对象都记录了它所发出的未完成 I/O 请求，即 ETHREAD 结构的 IrpList 域。当一个线程发起一个 I/O 请求时，在某些情形下该 I/O 请求将被插入到线程的未完成 I/O 请求链表中。在 Windows 中，这一插入链表的动作是通过 IopQueueThreadIrp 函数来完成的。在 I/O 请求的完成处理过程中，IopCompleteRequest 函数调用 IopDequeueThreadIrp

（实际上是一个 C 宏）将 IRP 对象从链表中移除。注意，插入链表的动作是在 IoCallDriver 被调用前完成的，譬如，IopSynchronousServiceTail 函数在调用 IoCallDriver 以前，将异步方式的 I/O 请求插入到线程的未完成 I/O 请求链表中。IRP 对象在被构造时，IopAllocateIrpPrivate 函数总是将它的 ThreadListEntry 域初始化为指向其自身（见 IopInitializeIrp 宏定义），所以，即使对于未加入链表的 IRP 对象，IopCompleteRequest 函数调用 IopDequeueThreadIrp 也不会引起任何问题（除了多执行几条指令）。

线程记录了未完成的 I/O 请求，一个重要的用途是，当线程被终止时，如果它的未完成 I/O 请求链表非空，则 I/O 管理器可以找到这些 IRP 对象并取消它们。参见 PspExitThread 函数的代码，它调用 IoCancelThreadIo 函数取消所有尚未完成的 I/O 请求。如果一个 I/O 请求在一段时间（5 分钟）内未能取消，则 IoCancelThreadIo 调用 IopDisassociateThreadIrp 函数强行使它从线程的未完成 I/O 请求链表中脱离。

(5) 取消 I/O 请求。

一个 I/O 请求在处理过程中可能由于某些原因要取消，譬如像上一段所描述的，发起者线程要终止了，或者应用程序要求取消一个 I/O。IRP 数据结构的 Cancel 和 CancelRoutine 域记录了一个 I/O 请求是否已被取消，以及被取消时执行的例程。驱动程序可通过 IoSetCancelRoutine 函数设置一个 I/O 请求的取消例程。注意，每个 I/O 请求只能注册一个取消例程，后注册的例程会覆盖前面注册的取消例程。

取消一个未完成的 I/O 请求是通过 IoCancelIrp 函数来实现的，该函数设置 IRP 对象的 Cancel 标志，并调用取消例程。Windows 使用一个专门为 I/O 取消而提供的全局自旋锁来同步 I/O 取消操作，因此，当驱动程序对一个正在处理中的 IRP 对象调用 IoSetCancelRoutine 函数时，必须使用 IoAcquireCancelSpinLock/IoReleaseCancelSpinLock 函数进行同步保护。IoCancelIrp 函数在执行取消处理时也使用此全局自旋锁进行同步保护，IRP 对象的取消例程有责任释放此自旋锁。若 IRP 对象没有已注册的取消例程，则 IoCancelIrp 在返回以前释放此自旋锁。关于 IoCancelIrp 使用此自旋锁的逻辑，请参考 IoCancelIrp 函数的代码（位于 iosubs.c 文件的 2 277~2 368 行）。

在取消例程中，除了要及时地调用 IoReleaseCancelSpinLock 以尽快释放全局的取消自旋锁，还要设置 I/O 状态块中的 Status 成员为 STATUS_CANCELLED、Information 成员为 0，并调用 IoCompleteRequest 以完成此 I/O 请求。因此，取消 I/O 请求是加速一个 I/O 请求的完成，它改变了一个 I/O 请求的处理逻辑，但仍然执行其完成处理过程。即使驱动程序未能在取消例程中立即完成此 I/O 请求，也必须保证在较短时间内通过某种方式完成

它。从设计角度而言，预期有可能花较长时间才能完成的 I/O 请求应该注册一个取消例程，以支持取消该 I/O 请求。

6.6.4 针对设备栈的 I/O 处理

前面 6.6.2 节讨论了针对独立设备对象的 I/O 请求的处理路径，驱动程序的分发例程接受 IoCallDriver 的调用，在完成必要的工作以后，调用 IoCompleteRequest 以结束该 I/O 请求的处理，然后返回。对于异步 I/O 的情形要复杂一些，分发例程先返回 STATUS_PENDING，待 I/O 真正完成以后驱动程序再调用 IoCompleteRequest 函数。

针对设备栈的 I/O 请求的处理过程本质上也遵循同样的处理流程，但由于涉及多个驱动程序处理同一个 I/O 请求，因此，一旦 I/O 管理器将控制权交给驱动程序，此 IRP 对象就将在多个驱动程序之间流转，每个参与处理的驱动程序只负责一部分功能，然后便将其递交给其他驱动程序。当一个驱动程序认为该 I/O 请求已经完成时，它调用 IoCompleteRequest 函数以完成该 I/O 请求。

一个典型的例子是针对文件系统的读写操作，文件系统驱动程序接收到用户读写某个文件的请求时，它根据该文件在文件系统中的逻辑位置，计算出要读写的数据在磁盘上的物理位置，从而将这一针对文件的读写请求转变成针对磁盘的读写请求，并发送给磁盘驱动程序。代表这一类 I/O 请求的 IRP 对象包含多个栈单元，其中栈单元的数量至少与设备栈中设备对象的数量一样多。因此，当 IRP 对象从设备栈顶部向下流动时，每个设备对象的驱动程序都可以独立地使用一个栈单元，除非它在转送 I/O 请求以前明确地指明自己不再使用当前栈单元。

为了完成一个 I/O 请求，驱动程序可能不得不向设备栈中下面的设备对象发送多个 I/O 请求，在这种情况下，它必须构造多个 IRP 对象，并且将它们与原来的 IRP 对象关联起来，这正是 IRP 对象包含 AssociatedIrp 子结构（实际上是一个 C 联合）的原因。举例来说，若文件系统驱动程序接收到的文件读写操作涉及多个不连续的扇区，则它将根据这些被读写数据的位置分布，构造多个磁盘读写请求分发给磁盘驱动程序。当所有这些磁盘读写请求都完成时，文件系统的原始读写请求才真正完成，此时文件系统驱动程序可通过 IoCompleteRequest 通知 I/O 管理器，该文件读写请求已经完成。

因此，不难理解，I/O 请求的传递是沿着设备栈由顶向底进行的，而 I/O 请求的完成过程则是由底向顶进行的。结合 IRP 对象中栈单元的使用规则，I/O 请求的传递和完成分别如图 6.21(a)和(b)所示。这里假设设备栈是由三个设备对象构成的，分别对应了三个驱动程序。I/O 请求的发起者在申请 IRP 对象时指明了 3 个栈单元。图中设备对象之间的上

下箭头并非代表函数调用关系，而是 IRP 对象在这些设备对象的驱动程序分发例程或完成例程之间的流转方向。

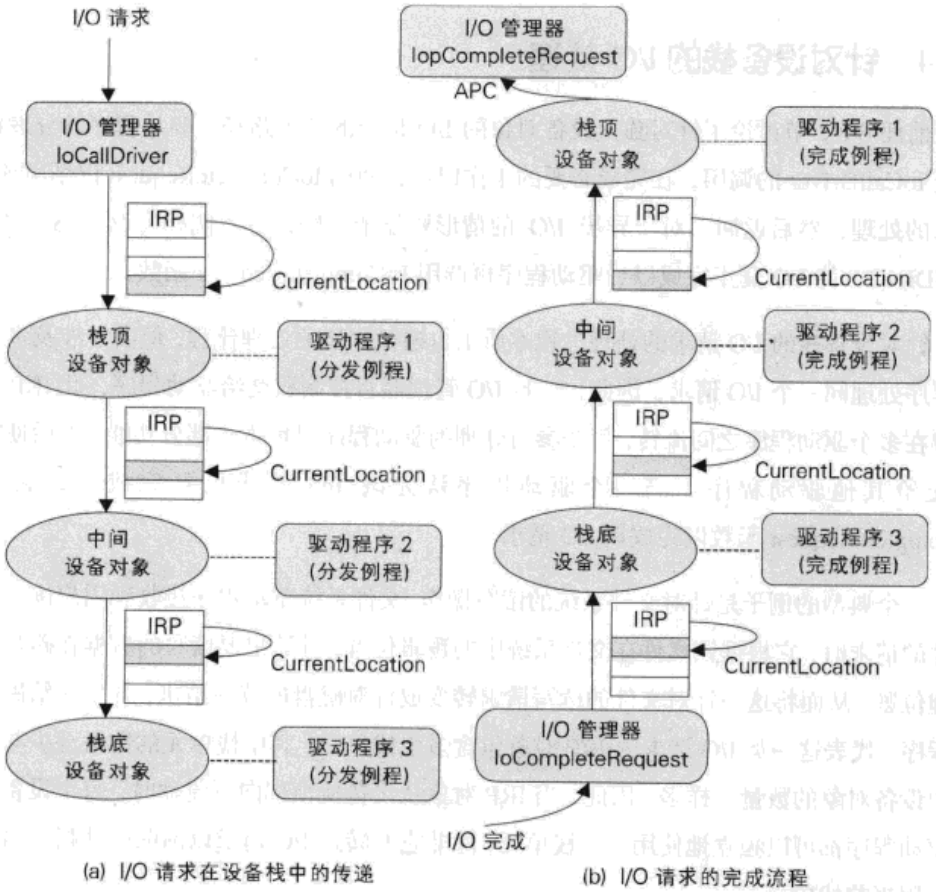


图 6.21 针对设备栈（由 3 个设备对象构成）的 I/O 请求的传递和完成

我们首先来看 I/O 请求从 I/O 管理器自顶向下的流动。接收 I/O 请求的 IoCallDriver 函数，实际上是 IopfCallDriver 函数，其代码如 6.6.2 节所列，第一个参数指定了栈顶的设备对象，第二个参数指定了 IRP 对象。当栈顶设备对象的驱动程序中的分发例程接收到来自 IoCallDriver 的 I/O 请求时，它完成属于自己职责范围内的工作，然后指示设备栈中的下级驱动程序继续处理此 I/O 请求，通常移交此 I/O 请求的代码如下：

```
IoCopyCurrentIrpStackLocationToNext(Irp);

IoSetCompletionRoutine(Irp,
                       CompletionRoutine,          // 当前驱动程序指定的完成例程
                       ContextInfo,                // 传递给完成例程的环境信息
                       InvokeOnSuccess,            // 布尔值，是否成功完成时调用
                       InvokeOnError,              // 布尔值，是否发生错误时调用
                       InvokeOnCancel              // 布尔值，是否被取消时调用
                       );
```

```
status = IoCallDriver(NextLowerDevice, Irp);
return status;
```

这里的 IoCopyCurrentIrpStackLocationToNext 调用只是简单地将当前驱动程序所使用的栈单元复制到下一个驱动程序将要使用的栈单元中，并且将下一栈单元的 Control 域初始化为 0。然后分发例程调用 IoSetCompletionRoutine 函数注册一个完成例程，以便当该 I/O 请求完成时，它可以做进一步的完成处理。最后，它调用 IoCallDriver 将控制权交给下一级设备对象的驱动程序的分发例程。

由此我们可以看到，I/O 请求在驱动程序之间的流转完全依赖于 IoCallDriver 函数来进行。而 IoCallDriver 函数实际上只是一个相对比较简单的同步转发而已。这样，在 I/O 请求经过每一级设备对象时，该设备对象的驱动程序总是可以选择以何种方式来处理此 I/O 请求：它可以利用 IoSetCompletionRoutine 函数注册一个完成例程，从而将它这一级上的完成处理单独放到一个例程中；也可以不注册完成例程，而是在完成自己的职责以后，转交给下一级设备对象的驱动程序，此时，它需要使用如下类似的代码：

```
IoSkipCurrentIrpStackLocation (Irp);
status = IoCallDriver (NextLowerDevice, Irp);
return status;
```

这里 IoSkipCurrentIrpStackLocation 只是简单地将 IRP 对象中的 CurrentLocation 域加 1（如 6.6.1 节所述），因此，下一级设备对象的驱动程序将仍然使用当前的栈单元（IopfCallDriver 将对 CurrentLocation 域做减 1），由于这一级驱动程序未注册完成例程，所以，此栈单元可以被安全地复用。

设备驱动程序在完成了自己职责范围内的 I/O 处理以后，可以直接调用 IoCompleteRequest 来完成此 I/O，这样便启动了图 6.21(b)所示的完成流程。注意，完成流程可以在设备栈的任何一级上启动，而不限于栈底设备对象的驱动程序。

正如 6.6.3 节关于 IopfCompleteRequest 执行流程的介绍，IopfCompleteRequest 首先根据 IoSetCompletionRoutine 设定的条件依次调用每一个已注册的完成例程，参见 base\ntos\io\iomgr\iosubs.c 文件的 3 489~3 569 行代码。每个已注册的完成例程仍然看到当初调用 IoSetCompletionRoutine 函数来注册该完成例程时的当前栈单元，并获得当初设定的环境信息（即 ContextInfo 参数）。在设备栈中，完成例程的调用顺序是由底向上，对应于栈单元的索引则是由小变大。从这一段代码可以看出，一个例外的情形是，一旦有一个完成例程返回 STATUS_MORE_PROCESSING_REQUIRED 值，则接下来的完成流程被打断，I/O 管理器不再执行后续的完成处理，在这种情况下，通常 I/O 请求的发起者会负责后续的处理，譬如释放 IRP 对象等。

对于每个设备对象和驱动程序而言，分发例程接收一个 I/O 请求，完成例程则是同一 I/O 请求的结束处理，两者使用同样的栈单元。可以说，栈单元是一个驱动程序在处理 I/O 请求时专用的执行环境。自然地，如果一个驱动程序的分发例程没有注册完成例程，则无须保留其栈单元，这正是前面介绍的 IoSkipCurrentIrpStackLocation 宏的用法。

针对设备栈的 I/O 请求的完成处理，除了可能要调用多个完成例程以外，其余流程与针对独立设备对象的完成处理相同，IopfCompleteRequest 的最后一步也是通过 APC 机制，在发起者线程中执行 IopCompleteRequest 例程，或者直接返回，由发起者线程调用 IopCompleteRequest 函数。IopCompleteRequest 负责最后的完成工作，包括拷贝缓冲区、释放资源、更新线程或进程的 I/O 状态、设置事件对象的信号状态或插入用户 APC，以及设置 I/O 状态块中的信息，参见 6.6.3 节的介绍。

以上介绍了一个 IRP 对象在设备栈中的传递过程，包括分发和完成的处理。正如本节前面所举的例子，一个驱动程序可能需要向低层设备对象发出多个额外的 I/O 请求才能完成它正在处理的 I/O 请求。在这种情况下，它可以创建一组关联 IRP，并设置好它们与原 IRP 对象的关联关系，然后逐个调用 IoCallDriver，待所有这些关联 I/O 请求都完成以后，原 I/O 请求也随之完成。图 6.22 显示了一种可能的关联 IRP 用法。

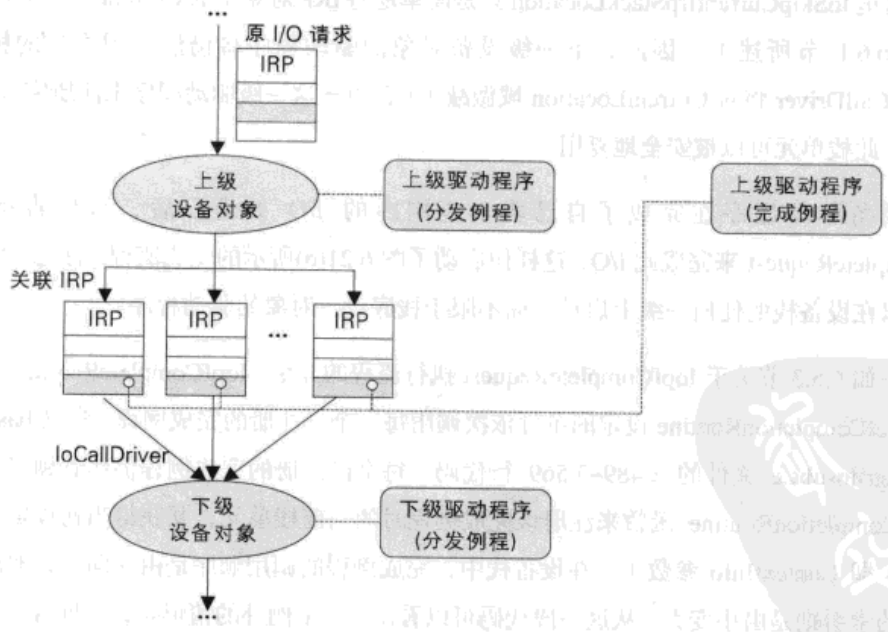


图 6.22 关联 IRP 对象的一种典型用法

在图 6.22 中，上级驱动程序的分发例程在发出新的子 I/O 请求以前，会将这些关联 IRP 对象中下一个栈单元的完成例程设置为当前驱动程序中的一个函数，并以某种方式

（例如 AssociatedIrp.MasterIrp 成员）在关联 IRP 对象中保存一个指向原 IRP 对象的地址，因此，当关联 IRP 完成时，上级驱动程序的完成例程将获得控制，它可以检查原 IRP 对象的状态（譬如，未完成的关联 IRP 的计数为 0，或者，存放未完成关联 IRP 的链表为空），以确定是否其所有关联 IRP 对象都已经完成。如果所有的关联 IRP 均已完成，则原 I/O 请求也可以完成，因而它调用 IoCompleteRequest 以触发完成处理。通常关联 IRP 对象的完成例程返回 STATUS_MORE_PROCESSING_REQUIRED，以缩短完成处理流程和避免不必要的 APC 例程。Windows DDK 中有一些例子演示了这种关联 IRP 的用法，例如 usb\isousb 和 storage\class\tape 驱动程序。

6.6.5 I/O 完成端口

在 Windows I/O 模型中，I/O 请求的完成处理是以 APC 的方式回到发起者线程，或者直接在发起者线程中返回。所以，I/O 请求的完成处理的最后步骤总是在发起者线程中进行。对于像 Web 服务器、数据库这样的服务器应用程序来说，安排太多或太少的线程来接收并处理客户请求，都可能导致性能问题。如果线程数量太少，则在多核或多处理器系统中，有些处理器或核可能无法发挥作用，而且当客户请求密集到来时，有些请求可能要等待较长时间才有机会被处理；如果线程数量太多，对于每一个客户请求都安排一个单独的线程来处理，则有可能造成大量的线程争抢有限的处理器资源，从而导致线程频繁轮换（thread-thrashing）的问题。

服务器应用程序的目标是尽可能避免过多的线程环境切换，它的线程应尽可能快速地完成客户的请求。理想的情形应该是，服务器处理客户请求的线程数量接近于处理器的数量，这样，每个处理器都能发挥作用，同时又不会造成线程频繁轮换的问题。Windows 提供了一种称为 I/O 完成端口（I/O Completion Port）的机制，能够让 I/O 的完成处理交由一个专门的线程池来完成，而线程池的线程数量是一个可配置的参数。这种做法将 I/O 请求的发起动作与完成处理分离到了不同的线程中，通过调节 I/O 完成端口的并行度（即线程池的线程数）使得系统处理客户请求的吞吐量最大化。

I/O 完成端口是一种内核对象，其类型对象为全局变量 IoCompletionObjectType，当 I/O 系统初始化时创建此类型对象，参见 IopCreateObjectTypes 函数的代码。I/O 完成端口实际上是 5.4.2 节介绍的内核队列对象的一个应用，其数据结构等同于 KQUEUE。下面我们来看一下 I/O 完成端口的用法及其实现机理。

首先，应用程序要创建一个 I/O 完成端口对象，这可以通过调用 API 函数 CreateIoCompletionPort 来完成，对应的系统服务为 NtCreateIoCompletion 函数。

NtCreateIoCompletion 的代码位于 base\ntos\io\iomgr\complete.c 文件的 44~169 行，其中参数 Count 指定了 I/O 完成端口的并发性值，即该完成端口允许多少个线程并发运行。NtCreateIoCompletion 的代码非常简短，它创建一个 KQUEUE 对象，类型为 IoCompletionObjectType，然后调用 KeInitializeQueue 初始化该 KQUEUE 对象，设置它的最多活动线程数，以及初始化其内部各个链表。最后将该完成端口对象插入到进程的句柄表中。

仅仅创建一个 I/O 完成端口对象当然还不够，应用程序还要指定将哪些文件的完成通知转发到此完成端口对象，这一步骤也要通过调用 API 函数 CreateIoCompletionPort 来完成，但这次并非创建完成端口对象，而是将一个文件对象关联到一个已创建的完成端口上。应用程序在调用 CreateIoCompletionPort 函数时为该文件指定一个完成键，即 CompletionKey 参数。这一次被调用的系统服务是 NtSetInformationFile，其中文件信息类别为 FileCompletionInformation，参见 base\ntos\io\iomgr\qsinfo.c 文件的 1 539~1 616 行代码，这一段代码是真正建立文件对象与完成端口对象关联的过程。我们可以看到，NtSetInformationFile 申请了一个 IO_COMPLETION_CONTEXT 对象，它只包含 Port 和 Key 成员，其中 Port 被初始化为参数中指定的端口对象，Key 被初始化为参数 FileInformation 中指定的 Key。凡是加入完成端口的文件对象都必须以异步方式创建，即应用程序在创建文件时必须指定了 FILE_FLAG_OVERLAPPED 标志，否则，NtSetInformationFile 返回 STATUS_INVALID_PARAMETER。一旦 NtSetInformationFile 成功地为文件对象设置了完成端口，则该文件对象的 CompletionContext 域将指向这一新创建的 IO_COMPLETION_CONTEXT 对象。

建立了这种关联以后，我们来看一下，当文件对象上的一个异步 I/O 请求完成时，在完成过程最后的步骤 IopCompleteRequest 函数中，若该文件对象已经关联了一个完成端口，即 CompletionContext->Port 非空，那么，IopCompleteRequest 调用 KeInsertQueue 函数，将 IRP 对象的 Tail 部分的 I/O 完成包插入到完成端口队列中。这样就把该 I/O 请求的后续完成处理交给完成端口对象了。

另一方面，应用程序中负责 I/O 完成处理的线程调用 API 函数 GetQueuedCompletionStatus，试图从 I/O 完成端口的队列中获得一个 I/O 完成包。这一调用实际上相当于将当前线程加入到完成端口的线程池中，对应的系统服务是 NtRemoveIoCompletion 函数，其代码位于 base\ntos\io\iomgr\complete.c 文件的 478~661 行。NtRemoveIoCompletion 首先对参数进行检查，然后调用 KeRemoveQueue 函数，试图从完成端口的队列中移除一个 I/O 完成包，若成功获取到，则释放 IRP 对象，并通过 NtRemoveIoCompletion 的输出参数返回必要的 I/O 完成状态信息。

总结以上步骤，NtCreateIoCompletion 函数调用 KeInitializeQueue 初始化一个 KQUEUE 对象；NtSetInformationFile 函数将完成端口对象的信息设置到文件对象的 CompletionContext 域中；IopCompleteRequest 函数调用 KeInsertQueue 函数，将 I/O 请求的完成处理转交给完成端口对象；最后，NtRemoveIoCompletion 函数调用 KeRemoveQueue 函数，从完成端口的队列中获得一个 I/O 请求的完成状态信息。图 6.23 显示了完成端口的工作过程，其中 Nt<*Xxx*>为相应的系统服务，而 Ke<*Xxx*>为相应的队列对象操作。

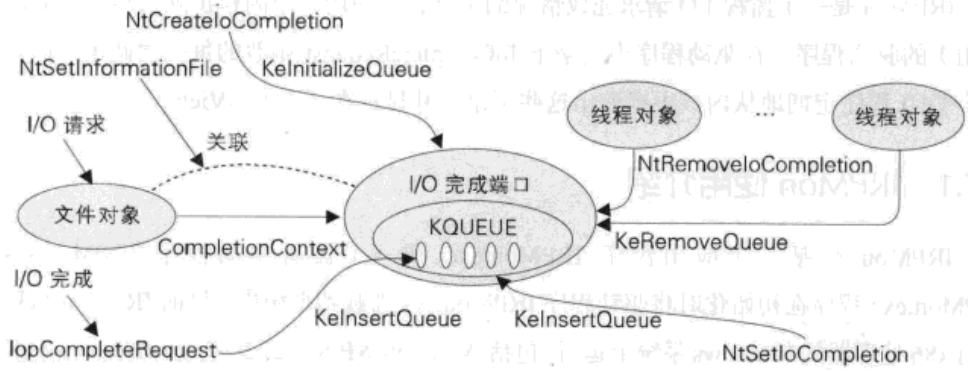


图 6.23 I/O 完成端口的工作原理

Windows 提供的 KQUEUE 队列对象是一种内核机制，它跟踪一个队列对象中的线程是否在执行，以确保当前系统中该队列对象的活动线程数不超过预设的最大值。当一个线程获得了一个 I/O 完成包时，它便进入活动状态，完成端口的活动线程数加 1，若该线程在执行过程中又进入等待（比如执行了 KeWaitForSingleObject、KeWaitForMultipleObjects 或 KeDelayExecution 函数），则完成端口的活动线程数减 1（通过 KiActivateWaiterQueue 函数）。此时，若完成端口的队列中有 I/O 完成包，并且有其他线程正在等待处理该完成端口对象中的 I/O 完成包，而完成端口的活动线程数未达到预设的最大值，则线程调度器将从完成端口的队列对象的等待块链表中取出一个等待块，让对应的等待线程处理队列中的 I/O 完成包，并且，线程调度器在唤醒一个线程的同时，也将使完成端口的活动线程数加 1。假设此时完成端口的活动线程数已经达到了最大值，而原先进入等待的线程现在满足执行条件了，则该线程被唤醒，并且活动线程数加 1（由 KiUnlinkThread 函数来完成，它被 KiUnwaitThread 函数调用）。在这种情况下，完成端口的活动线程数就超出了预设的最大值。然而，这种超出最大值的现象只是暂时的，一旦有一个活动线程结束了对它所获得的 I/O 完成包的处理，则活动线程数又会回到设定的范围内。注意，当活动线程数达到了预设的最大值时，新进来的 I/O 完成包只能进入队列，而不会被立即处理。

除了以上提到的系统服务以外，另一个系统服务 NtSetIoCompletion 可以主动地构造一个 I/O 完成包，并通过 KeInsertQueue 函数插入到完成端口对象的队列中，如图 6.23 右

下角部分所示。应用程序可通过 API 函数 PostQueuedCompletionStatus 来使用该系统服务。NtSetIoCompletion 只是简单地调用 IoSetIoCompletion 函数，IoSetIoCompletion 的代码位于 complete.c 文件的 663~787 行。

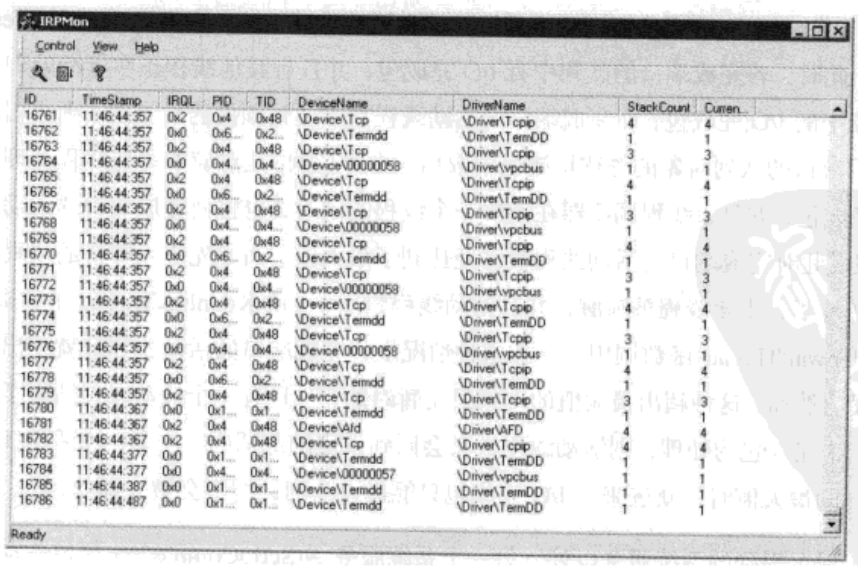
6.7 I/O 请求监视工具 IRPMon

IRPMon 是一个监视 I/O 请求完成情况的工具，它使用一个内核扩展类型（即非即插即用）的驱动程序，在驱动程序中记录下 IoCompleteRequest 函数的每一次调用。同时，一个 GUI 程序定期地从内核中提取出这些记录，并显示在一个 ListView 中。

6.7.1 IRPMon 使用介绍

IRPMon 包括一个应用程序 IRPMon.exe 和一个设备驱动程序 IRPMon.sys。IRPMon.exe 程序在初始化时将驱动程序 IRPMon.sys 加载到系统中。目前 IRPMon 可以在 Intel x86 处理器的 Windows 系统上运行，包括 Windows XP/Server 2003/Vista/Server2008/7。

IRPMon.exe 是一个单文档用户界面程序，它通过驱动程序 IRPMon.sys 实时地捕捉当前系统中发生的 I/O 请求，并以 ListView 用户界面形式显示在主窗口中。图 6.24 显示了正在运行中的 IRPMon 工具。视图中的每一行代表一个 I/O 请求，其中包括该请求完成时的时间戳、IRQL、进程 ID 和线程 ID，以及启动完成处理的驱动程序和对应的设备对象名称，另外也包括栈单元的数量和当前栈单元位置。



ID	TimeStamp	IRQL	PID	TID	DeviceName	DriveName	StackCount	Curren...
16761	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	4	4
16762	11:46:44:357	0x0	0x6...	0x2...	\Device\Termdd	\Driver\TermDD	1	1
16763	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	3	3
16764	11:46:44:357	0x0	0x4...	0x4...	\Device\00000058	\Driver\vpcbus	1	1
16765	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	4	4
16766	11:46:44:357	0x0	0x6...	0x2...	\Device\Termdd	\Driver\TermDD	1	1
16767	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	3	3
16768	11:46:44:357	0x0	0x4...	0x4...	\Device\00000058	\Driver\vpcbus	1	1
16769	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	4	4
16770	11:46:44:357	0x0	0x6...	0x2...	\Device\Termdd	\Driver\TermDD	1	1
16771	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	3	3
16772	11:46:44:357	0x0	0x4...	0x4...	\Device\00000058	\Driver\vpcbus	1	1
16773	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	4	4
16774	11:46:44:357	0x0	0x6...	0x2...	\Device\Termdd	\Driver\TermDD	1	1
16775	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	3	3
16776	11:46:44:357	0x0	0x4...	0x4...	\Device\00000058	\Driver\vpcbus	1	1
16777	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	4	4
16778	11:46:44:357	0x0	0x6...	0x2...	\Device\Termdd	\Driver\TermDD	1	1
16779	11:46:44:357	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	3	3
16780	11:46:44:367	0x0	0x1...	0x1...	\Device\Termdd	\Driver\TermDD	1	1
16781	11:46:44:367	0x2	0x4	0x48	\Device\Afd	\Driver\AFD	4	4
16782	11:46:44:367	0x2	0x4	0x48	\Device\Tcp	\Driver\Tcpip	4	4
16783	11:46:44:377	0x0	0x1...	0x1...	\Device\Termdd	\Driver\TermDD	1	1
16784	11:46:44:377	0x0	0x4...	0x4...	\Device\00000057	\Driver\vpcbus	1	1
16785	11:46:44:387	0x0	0x1...	0x1...	\Device\Termdd	\Driver\TermDD	1	1
16786	11:46:44:487	0x0	0x1...	0x1...	\Device\Termdd	\Driver\TermDD	1	1

图 6.24 正在运行中的 IRPMon 工具

IRPMon.sys 是非即插即用驱动程序，IRPMon.exe 直接通过 SCM（服务控制管理器）来安装、加载和启动它。当程序退出时，它停止并卸载该驱动程序。因此，IRPMon.sys 是仅为 IRPMon.exe 程序所做的内核扩展，这是一种通过驱动程序来获取内核信息的典型用法。

6.7.2 IRPMon 实现原理

IRPMon.exe 是一个普通的应用程序，它每隔 100 ms 向驱动程序要一次数据，然后解析数据并插入到界面视图中。IRPMon 的视图可以实时滚动，以便用户总是能看到最近完成的 I/O 请求，也可以不实时滚动，保持窗口中显示的条目不变。用户也可以清除视图中的内容。所有这些功能，都是通过 Windows 提供的标准窗口 ListView 来实现的，请参考 Windows SDK 中关于 ListView 窗口用法的文档。

IRPMon 的核心功能在于驱动程序 IRPMon.sys 中，它的初始化例程注册了 4 个分发例程和一个卸载例程，4 个分发例程的主功能码分别是 IRP_MJ_CREATE、IRP_MJ_CLOSE、IRP_MJ_READ 和 IRP_MJ_DEVICE_CONTROL，其中 IRP_MJ_READ 分发例程用于向应用程序传递数据，IRP_MJ_DEVICE_CONTROL 分发例程用于获取一些额外的信息，比如设备对象的名称等。

为了记录下每一个 I/O 请求，我们选择截取在 I/O 完成时驱动程序要调用的函数，IofCompleteRequest 是一个导出函数，因此，我们无须符号表信息就可以在驱动程序中得到该函数的地址。这样，我们就可以记录下每次当 IofCompleteRequest（实际上即 IoCompleteRequest 函数）被调用时的各种信息。

记录下来的 I/O 请求信息被保存在一个环形缓冲区中，当应用程序通过 API 函数 ReadFile 来提取数据时，驱动程序将环形缓冲区中的 I/O 请求信息拷贝到用户提供的缓冲区中，然后清空环形缓冲区。如果环形缓冲区满了，则驱动程序擦除以前的记录，以使缓冲区中总是存放最新的 I/O 请求信息。

IRPMon 的做法并非最理想的，它对系统的性能有一定的影响。Windows 本身提供的事件追踪（ETW，Event Tracing for Windows）机制是更好的获取内核事件的途径，自 Windows Vista/Server 2008 以后，内核中大量的事件已经被暴露出来，包括像系统调度（线程切换）和各种 I/O 事件等（参见 9.3 节的介绍）。然而，在性能并非关键的情况下，通过 IRPMon 工具，我们可以观察到 Windows 系统是如何响应各种 I/O 请求的，或者说，Windows 系统中的 I/O 是如何被处理和完成的。

6.8 本章总结

在本章中，我们讨论了 Windows I/O 系统的三个重要部件：I/O 管理器、即插即用管理器和电源管理器。I/O 管理器建立了 Windows 的 I/O 模型，它定义了一套可扩展的驱动程序框架，允许第三方编写的驱动程序很方便地插入到系统中来；即插即用管理器是 Windows 管理硬件设备的核心模块，它利用总线驱动程序来自动地列举总线上的设备，并根据设备的信息来加载和启动驱动程序；电源的管理对于现代计算机系统越来越重要，Windows 的电源管理器遵从 ACPI 规范，它一方面确定系统所处的电源状态，另一方面通过设备驱动程序的 IRP_MJ_POWER 分发例程来执行电源状态转移。

本章还利用 Windows DDK 中提供的 toaster 例子，分析了 Windows 驱动程序的代码结构，以及各种类型驱动程序的特点，并讨论了层次驱动程序的组织结构。过滤驱动程序是一种特殊类型的驱动程序，它可以改变或者监视 I/O 请求的处理过程，本章介绍了过滤驱动程序的配置和加载过程。在 Windows 系统中，非即插即用驱动程序仍然被广泛使用，本章以 Windows DDK 中的 IOCTL 为例介绍了这一类驱动程序的加载和用法。

Windows 完全支持异步 I/O 处理，这使得 I/O 请求的处理过程相对比较复杂，尤其当一个 I/O 请求在多个驱动程序之间传递时。本章详细地探讨了 I/O 处理过程中涉及的各种事项，包括缓冲区管理和 I/O 完成处理。除此以外，本章还介绍了一种特殊的 I/O 完成机制——I/O 完成端口，这种机制对于提高服务器应用程序的吞吐量有重要的意义。

对于建立在 Windows I/O 模型之上的两个重要的子系统：文件系统和网络，本章并没有介绍。WRK 没有直接包含这两部分代码，但是，在 WRK 中我们还是可以看到很多有关文件系统的代码痕迹，毕竟文件系统对于操作系统的运行有着举足轻重的作用。有关这两部分内容，在本书第 7 章和第 9 章中将分别作进一步介绍。

7

Windows 存储管理

本书第 4 章讨论了 Windows 的内存管理，我们已经知道，处理器要操纵一份数据，必须将其加载到物理内存中，而无论该数据来自哪里。在现代计算机体系结构中，内存是易失的，即一旦系统失败或宕机，则内存中的数据将不复存在；若操作系统想要再次访问这些数据，则必须将它们重新加载到内存中。计算机用到的数据可能位于硬盘上，也可能位于其他类型的存储介质中。当数据被加载到内存中以后，可能又会在特定情形下被交换到硬盘交换空间中。所以，从数据的角度来看，它们在系统中有不同的形态，受不同系统模块的管理；而从操作系统的角度来看，存储管理是一个更为广泛的概念，它涵盖了对内存、硬盘和其他存储介质的管理。

本章将讨论 Windows 中的存储管理体系，除了第 4 章已经详细介绍过的内存管理，本章将重点讨论 Windows 的缓存管理、卷的管理，以及文件系统。

7.1 存储管理概述

从应用程序的角度来看，通常会看到两种存储介质：系统主内存和磁盘空间，有时候也会透明地将光盘或移动存储介质等同于磁盘空间来访问，这些非内存存储介质被统称为外存。这种简单的二级划分方法隐藏了系统中许多复杂的部件和机制，但对于应用程序的开发人员来说这又非常合理，因为他们总是可以假定数据要么在内存中（通过地址或指针来引用），要么可以通过像 ReadFile 或 WriteFile 这样的 API 函数将数据读入内存或写到外存中（将外存抽象成文件）。为了提供这样的抽象存储模型，操作系统既要针对处理器体系结构的特点小心地设计内存管理机制，又要在 I/O 模型基础上提供访问磁盘或其他外存的能力。下面我们首先来看一看 Intel x86 处理器支持的存储体系，然后介绍 Windows 的存储管理结构。

7.1.1 硬件存储体系（memory hierarchy）

到现在为止，我们只是简单地认为 Intel x86 处理器在执行一条指令时，指令本身位于内存中，指令用到的数据要么在内存中，要么在处理器内部的寄存器中。虚拟地址和物理地址是两种寻址方式，若一条指令通过虚拟地址来引用一个内存单元，则处理器在执行该指令时还要涉及页表查找动作。这是我们在第 4 章中看到的 Intel x86 处理器操纵内存单元的方式。实际上，Intel x86 处理器操纵内存单元的方式还要复杂一些，在处理器与内存单元之间往往还存在两级缓存。图 7.1 显示了 Intel x86 处理器支持的存储体系结构。

图 7.1 Intel x86 处理器支持的存储体系结构

在图 7.1 中，大体而言，从上向下，存储体的访问速度越来越慢，这里列出的速度是指访问单个存储单元所需要的时间；单位存储体的成本则越来越便宜；在实际系统中的容量也越来越大。图中列出的速度和容量值反映了在本书写作时主流 PC 系统上这些存储体的大致范围。下面逐一介绍每一层存储体的特点：

- 寄存器，这是处理器内部定义的存储体，它们除了存储功能，往往还兼有其他的能力，比如参与运算、地址解析、指示处理器的状态，等等。寄存器是由处理器内部专门的触发器电路实现的，处理器往往只有少量的寄存器，其存储能力不超过几百字节，而且，一旦系统掉电，则寄存器的内容随之丢失。

- L1 缓存，这是靠近处理器的缓存，它存放的是处理器最近访问过的内存数据。缓存通常是由 SRAM（静态随机访问存储器）来实现，而系统的主内存则是由 DRAM（动态随机访问存储器）来实现的。从技术上讲，SRAM 比 DRAM 要快，但是它的集成度低，需要更多的晶体管。SRAM 只要有电，就可以持续保持其状态（0 或 1），所以不需要刷新；DRAM 不同，它需要定时（<100 ms）刷新，即读出来再写回去，才能保持其状态。

L1 缓存的工作原理是，当处理器的指令访问内存时，它使用虚拟地址（VA）来引用内存单元，首先检查 L1 缓存中是否有该内存单元，若有则无须再访问物理内存，而是直接读或写 L1 缓存中的单元。如果 L1 缓存中没有该内存单元，那么，首先计算出对应的物理地址（PA），然后检查 L2 缓存中是否有该内存单元。若 L2 缓存中有，则将该单元复制到 L1 缓存中，否则需要从物理内存中获得该内存单元。

L1 缓存在获取数据时，通常会获取到比当前指令所要求的内存单元更多的数据，比如连续的 32 或 64 字节，这取决于缓存的结构和设计。缓存中的每个数据单元有时候也称为 cache line（缓存行）。

- L2 缓存，如同 L1 缓存一样，它也是由 SRAM 来实现的，只不过更加靠近系统的主内存而不是处理器。与 L1 缓存不同的是，L2 缓存是用物理地址来寻址的。当 L1 缓存需要物理内存单元时，它会检查 L2 缓存，并使用其中的存储单元，若有必要，则从物理内存复制到 L2 缓存中。在实践中，增加 L2 缓存对于许多软件会有显著的性能提升，因此，处理器厂商在平衡成本开销的同时，应尽可能地加大 L2 缓存。
- 主内存，或简称内存。在物理上，内存是由 DRAM 来实现的，它比 L1 和 L2 缓存慢，但容量要大得多。一个系统中当前正在使用的数据，包括系统本身的数据，以及各个应用程序用到的数据，都存放在内存中。内存是易失的，即，一旦系统掉电，其内容将随之丢失。本书第 4 章讨论的内存管理，所指的正是这部分存储体。
- 本地一级存储设备和远程存储设备。在现代计算机系统中，像硬盘这样的本地永久存储设备仍然是不可或缺的，操作系统所需要的各个模块和数据都存放在硬盘上，应用程序所需要的可执行文件和数据通常也存放在硬盘上。本地一级存储设备还包括像光盘、USB 盘这样的可移动存储介质，在实践中，软盘已经很少使用了。另外，系统或应用程序要访问的数据也可能来自远程系统，在这种情况下，操作系统通过网络适配器，直接将数据复制到内存中，以供处理器使用。

  这一层的存储设备是永久存储体，系统掉电以后其数据不会丢失。操作系统或应用程序若要访问这些存储体上的代码或数据，必须通过文件系统和设备驱动程序，将代码或数据载入到内存中，然后处理器才可以访问。
- 本地二级存储和远程二级存储。为了保障数据安全或提供更大空间的存储能力，有些系统还配备了二级永久存储设备，比如用磁带设备来备份一个系统中的数据，或者通过索引库来访问一个包含大量光碟的影像资料库。二级存储也可能是远程的，比如将数据导入到一个分布式存储系统中，或者连接一个网络文件系统。这一层不是必需的，例如，在个人用户的系统中，这一层往往较少用到。

在以上的存储体系中，下层存储体包含了上层存储体的内容。系统或应用程序在访问数据时，若能够在上层存储体中获得，则无须再往下读取数据，否则需要逐层向下寻找数据。所以，这整个存储体系构成了一个多层次的缓存结构。为了使系统的性能表现得更好，每一层应该包含下层最频繁使用的数据，或者最近需要使用的数据，从而尽可能地在上层满足系统或应用程序的需要。在计算机系统中，局部性原理（principle of locality）往往是成立的，即系统或应用程序倾向于引用临近的数据单元，这是空间局部性；或者，刚刚被引用过的数据有可能很快被再次引用，这是时间局部性。上层存储体从下层存储体读取数据时，一次读取的数据比实际需要的数据要多，比如 L1/L2 缓存的缓存线为 32 ~ 256 字节，而系统从磁盘读取数据到内存中时，一次至少要读几 KB。这种做法正是因为希望多读取的数据能很快被使用，即空间局部性。

L1、L2 缓存位于处理器内部，它们的工作方式对于处理器中的指令是透明的，所以，操作系统并不需要显式地管理 L1 和 L2 缓存的内容，处理器会自动维护这两级缓存。而且，在多核处理器中，一种可能的设计是，L1 缓存是每个核独立的，而 L2 缓存则是核间共享的，处理器会小心地维护好这些缓存的一致性。

本地一级、二级存储体和远程存储体都是通过设备来支持的，所以，这些存储体的管理需要由设备驱动程序来完成，操作系统往往只是提供一个框架，譬如，提供一个抽象的文件系统结构或者一个名字空间，然后由驱动程序来完成实际的存储体管理。对于像分布式文件系统这样的存储机制，更是超越了单个驱动程序的职责范围，它们需要多个驱动程序，甚至多个系统协同工作，才能有效地完成存储管理任务。

7.1.2 Windows 的存储管理结构

在 Windows 的存储管理体系中，除了内存管理以外，文件系统无疑是最为重要的。而且，Windows 的所有设备也都以文件对象的形式被应用程序访问。在上一章中，我们已经看到了设备对象、驱动程序对象和文件对象之间的关系，以及应用程序利用文件对象来访问设备对象的过程，包括 I/O 请求的传递路径和完成处理。但是，我们尚未讨论文件系统，以及对文件系统中的文件对象的读写操作。对照图 7.1 所示的硬件存储体系，我们不难理解，在 Windows 中，主内存以上的存储体是由处理器本身决定的，操作系统并没有显式地参与对这些存储体的管理工作，而主内存的管理是由操作系统来实施的，我们在第 4 章中已经详细地讨论了 Windows 的内存管理机制。主内存以下的层次是针对外部设备的，因此，这些存储体的管理是通过设备驱动程序来完成的，系统和应用程序通过 I/O 管理器与这些设备的驱动程序打交道，来完成数据存储任务。

图 7.2 显示了在 Windows 中从磁盘读写数据的路径。Windows 为文件操作提供了缓存机制。如果要读取或写入的数据已经被映射到缓存中了，则读写文件的操作转入到一个被称为缓存管理器的组件，这里的缓存概念与上一小节介绍的处理器 L1/L2 缓存并不相同，它是指系统虚拟地址空间中一段被用于映射文件视图的地址范围。如果要读写的文件数据尚未被加载到缓存空间中，则内存管理器通过文件系统驱动程序（FSD，File System Driver），从磁盘中加载该文件的这部分数据。

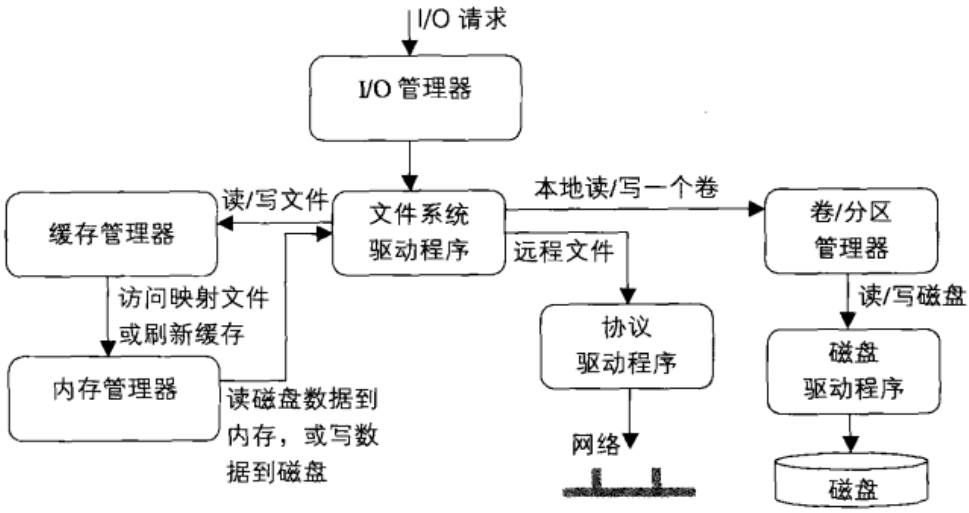


图 7.2 Windows 的存储管理结构

缓存管理器是 Windows 执行体层的组件，它为操作系统各种类型的文件系统提供了统一的缓存服务。为了访问文件系统中的文件，Windows 提供的一种常见做法是，首先将文件映射到系统地址空间中，然后访问此虚拟内存。若待访问的内存范围尚未包含文件数据，或者与磁盘上的文件数据不一致，则内存管理器通过文件系统驱动程序读写磁盘上的文件。由此可见，缓存空间代表了系统访问磁盘文件（或网络文件）的内存范围，而缓存管理器负责将内存读写操作与背后的磁盘操作联系起来，在有些情况下还可以实施优化，比如提前预读文件数据或者将磁盘写操作延迟到后台进行。本章 7.2 节将介绍 Windows 的缓存管理。

文件系统驱动程序虽然在模块结构上也是一个标准的内核模式驱动程序，但是，它们需要遵守特定的规范以便能够与 Windows 的存储管理体系有机地集成在一起，例如，它们必须向 I/O 管理器注册一个文件系统（通过 IoRegisterFileSystem 函数），这样 I/O 管理器在挂载（mount）一个卷的时候就会询问这些已注册的文件系统设备对象，看是否能识别该卷。分区管理器负责管理磁盘上的分区，并且将这些分区通知即插即用管理器；卷管理器则在分区管理器的帮助下，了解到当前系统中的分区情况，并且为每个卷创建一个卷

设备对象。当系统或应用程序第一次访问一个卷时，Windows 执行卷的挂载动作，从而将卷设备对象与文件系统驱动程序联系起来。对磁盘设备的访问最终通过磁盘驱动程序来完成。本章 7.3 节将介绍 Windows 的卷管理，7.4 节介绍 Windows 中的文件系统。

Windows 也支持远程文件系统，这需要由客户机的远程 FSD 与服务器的远程 FSD 通过网络协议协同工作。本章不讨论远程文件系统。

7.2 Windows 缓存管理

Windows 的缓存管理器介于内存管理器与文件系统驱动程序之间，它包含一组以“Cc”打头的内核模式函数、全局变量，以及一些系统线程。缓存管理器使用了系统地址空间中预留的一段地址范围，并且控制哪些文件的哪些部分可以使用这些地址和相应的物理内存，以达到在大量文件请求和系统有限内存资源之间的一种平衡。缓存管理器并不特定于具体的文件系统类型，原则上它可支持所有的文件系统类型，包括本地的和远程的文件系统。另外，缓存管理器也支持可恢复的文件系统，例如 NTFS，从而在系统失败以后可以恢复数据。WRK 包含了缓存管理器的完整代码，位于 base\ntos\cache 目录下，读者可以参考 WRK 中缓存管理器的实现代码来理解其工作机理。

7.2.1 Windows 缓存空间的内存管理

在 4.2.1 节介绍系统地址空间布局时，我们已经看到，系统中有一段地址范围专门用于系统缓存空间：变量 MmSystemCacheWorkingSetList 定义了系统缓存工作集的地址，其值被初始化为 0xc0c00000；变量 MmSystemCacheStart 定义了系统缓存的起始地址，其值被初始化为 0xc1000000；系统缓存可一直扩展到系统的换页内存池，即 MmPagedPoolStart-1，而 MmPagedPoolStart 的默认值为 0xe1000000；系统缓存的结束地址存放在变量 MmSystemCacheEnd 中，按默认配置，该值为 0xe1000000-1=0xe0ffffff。因此，在 Intel x86 系统中，系统缓存的默认大小为 512 MB。

除了这部分地址范围，如果系统注册表指示了使用大系统缓存（MmLargeSystemCache 变量，注册表路径为 HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\LargeSystemCache），那么，系统会将基本的非换页内存池之后一直到系统视图（System View）之间的虚拟地址空间也用做系统缓存。由于非换页内存池的基本区域为 MmNonPagedPoolStart 与 MmNonPagedPoolEnd0 定义的地址范围，参见 4.2.2 节的介绍，所以，这第二部分系统缓存空间从 MmNonPagedPoolEnd0 开始，一直至 MiSystemViewStart-1。然而，系统缓存并没有用完这整个地址范围，而只使用了其中的

5/6。这部分缓存空间称为系统缓存额外区域，分别由变量 MiSystemCacheStartExtra 和 MiSystemCacheEndExtra 指定起始和结束地址，此两变量值均以 4 M 为边界对齐。关于系统缓存额外区域的初始化过程，请参考 base\ntos\mm\i386\init386.c 文件的 2 349~2 445 行。系统缓存额外区域的大小受非换页内存池的位置和会话空间的结构影响，例如，在一台 512 MB 的 Windows Server 2003 SP1 默认配置的机器上，MiSystemCacheStartExtra 为 0x82000000，MiSystemCacheEndExtra 为 0xb17fffff，因而大小为 760 MB。图 7.3 显示了在系统地址空间的布局结构中，系统缓存基本区域和额外区域的位置，图中右侧的地址值显示了系统缓存区域在此示例系统中的确切位置。

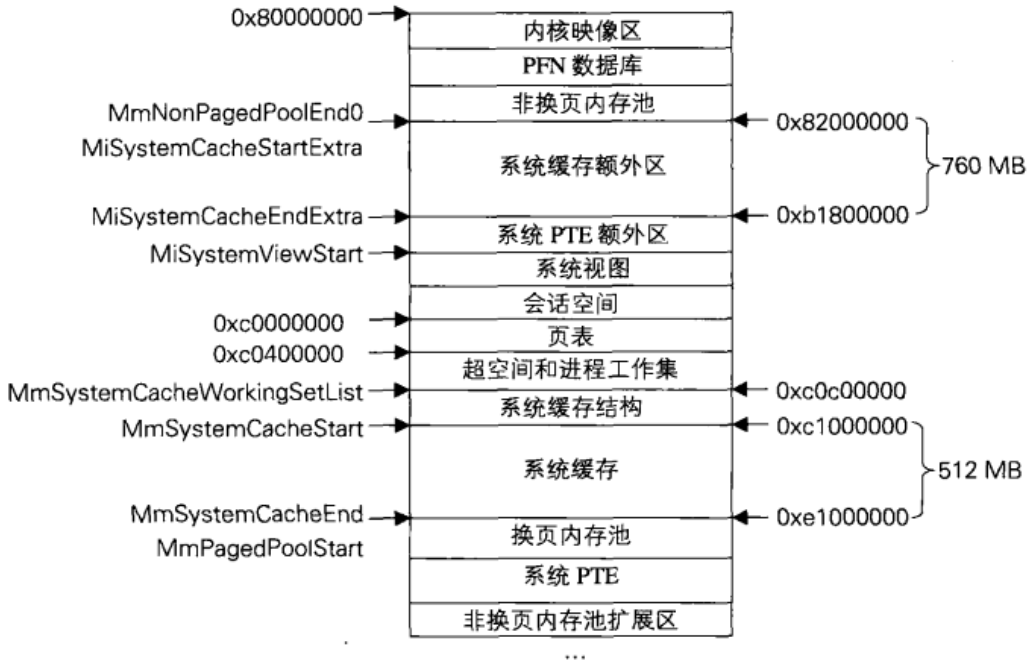


图 7.3 系统缓存在系统地址空间中的位置

系统缓存的基本区域和额外区域合起来的大小存放在变量 MmSizeOfSystemCacheInPages 中。在 Windows 的客户系统中，注册表变量 LargeSystemCache 的默认值为 0，所以，系统缓存没有额外区域，此时 MmSizeOfSystemCacheInPages 变量仅包含基本区域的大小；而在 Windows 的服务器系统中，LargeSystemCache 的默认值为 1，所以，系统缓存有较大的额外区域。这也说明，在服务器系统中，操作系统会将尽可能多的系统地址空间用于缓存区域，因而可以同时映射大量的文件，避免频繁的解除映射和重新映射的操作，这对于执行大量文件操作的服务器系统，比如文件服务器、Web 服务器和数据库服务器等，有重要的意义。

现在我们来看系统缓存区域的内存管理。系统缓存区域被等分成 256 KB 大小的槽，

这些槽称为视图（view）。当系统地址空间初始化时，系统缓存区域的页表已经被分配好，这些视图的第一个 PTE 链接起来，构成一个视图空闲单链表，链表的头是全局变量 MmFirstFreeSystemCache。关于链表的初始化过程，请参考 MiInitializeSystemCache 函数，链表的结构类似于 4.2.3 节介绍的系统 PTE 区域的内存管理方法。只不过，在系统缓存区域，只有每个视图的第一个 PTE 而并非所有的 PTE 加入到链表中。这样，当缓存管理器需要一个视图时，从链表中摘除一个即可；而当回收一个视图时，将它加入链表即可。仅仅依靠 PTE 即可完成视图的管理，无须额外内存开销。

当一个文件被访问时，该文件中包含所访问数据的部分被映射到系统缓存区域的槽中。文件也被分成 256 KB 大小的槽，因此，如果一个读写操作涉及相邻的两个槽，即使要读写的数据长度小于 256 KB，缓存管理器也需要映射两个视图。图 7.4 显示了系统缓存中的视图和文件中的视图之间的关系。一个文件并非在打开的时候就被映射所有的视图，而是当其中的数据被第一次访问时才被映射。在文件的读或写操作的过程中，被映射的视图称为活动的，当操作完成以后，这些视图变成不活动的，但是，它们的映射关系并不立即被解除。图 7.4 中的实线代表了活动的映射关系，说明针对该文件这部分数据有一个读或写操作正在进行中；虚线代表了不活动的映射关系，说明曾经访问过该文件的这部分数据，此映射关系当前仍然保留着。

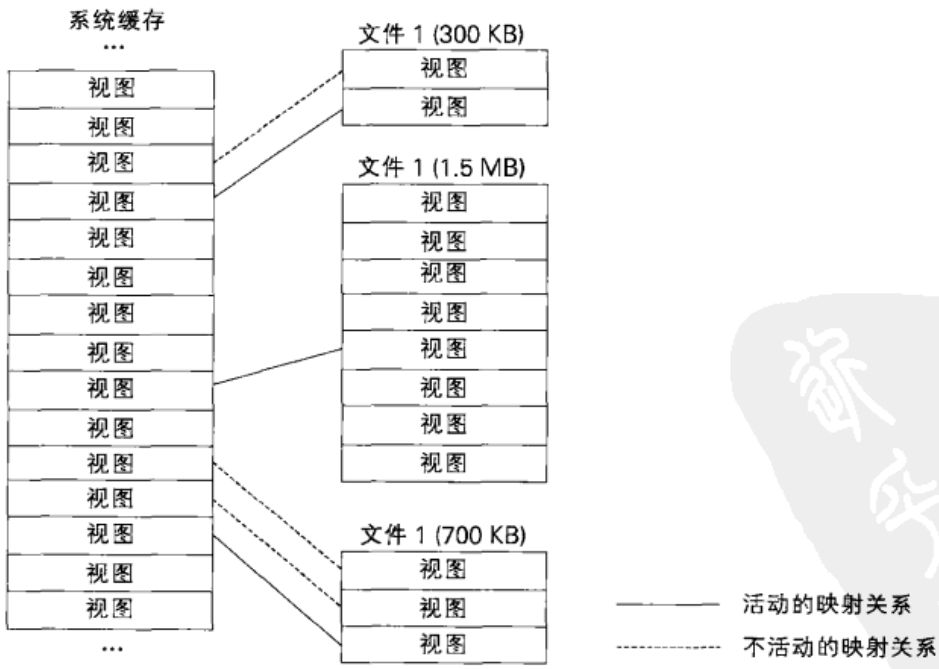


图 7.4 系统缓存中的视图与文件中的视图

当缓存管理器需要映射一个文件的视图，但系统缓存中又没有足够的空闲视图时，它将解除最早被映射的不活动视图，以试图满足当前文件的需要。若空闲视图仍然不够，则返回 I/O 错误。由于只有在读或写操作过程中视图才被标记为活动的，因此，因空闲视图不够而发生错误的可能性不大，毕竟系统缓存区域有几百 MB 乃至超过 1 GB 的地址范围。

下面我们来看一下 Windows 如何管理系统缓存和文件中的视图。系统缓存中的每个视图都有一个对应的数据结构，称为 VACB（Virtual Address Control Block，虚拟地址控制块），其定义如下：

```
typedef struct _VACB {
    PVOID BaseAddress;                                  // 基地址
    struct _SHARED_CACHE_MAP *SharedCacheMap;           // 指向使用此VACB的共享缓存表
    union {
        LARGE_INTEGER FileOffset;                       // 在共享缓存表中的文件偏移
        USHORT ActiveCount;                             // 当前正在使用此VACB的计数值
    } Overlay;
    LIST_ENTRY LruList;                                 // VACB链表项
} VACB, *PVACB;
```

VACB 的结构非常简单，它描述了与系统缓存中的一个视图相关的信息：BaseAddress 域记录了该视图的基地址；SharedCacheMap 域记录了当它被映射到一个文件中时，该文件对象的共享缓存表；FileOffset 域记录了该视图在文件中的偏移；ActiveCount 域表示它正在被使用的计数，即有多少读或写操作正在访问该视图。最后，LruList 域是一个链表项，当一个视图属于某一个链表（譬如，空闲视图链表或可重用视图链表）时，该项被加入到此链表中。

缓存管理器在初始化时，为系统缓存区域中的所有视图分配了一个 VACB 数组，即全局变量 CcVacbs，它可以描述系统缓存基本区域和额外区域中的所有视图。CcVacbs 是在 CcInitializeVacbs 函数中被初始化的，它指向一块从非换页内存池中分配得到的内存。另外，全局变量 CcVacbFreeList 是一个链表，包含所有空闲的视图；类似地，链表 CcVacbLru 包含了所有可被重用的视图。这里，CcInitializeVacbs 函数被缓存管理器的初始化函数 CcInitializeCacheManager 调用，而 CcInitializeCacheManager 则在系统阶段 1 初始化时被 Phase1InitializationDiscard 调用，参见 WRK 中这些函数的代码。

了解了系统缓存中视图的管理结构以后，我们再来看文件中的视图管理。我们首先回顾一下，6.2.3 节介绍的文件对象包含了两个与缓存管理有关的成员：PrivateCacheMap 域指向文件对象的私有缓存表；SectionObjectPointer 域的 SharedCacheMap 成员指向文件对象的共享缓存表。私有缓存表的数据结构为 PRIVATE_CACHE_MAP，它包含了最近两次读文件操作的位置以及当前预读的偏移和长度等信息。共享缓存表的数据结构为 SHARED_CACHE_MAP，它包含了针对一个文件的 VACB 索引数组，记录了该文件中哪

些视图已经被映射到系统缓存中了。因此，当缓存管理器接收到对一个文件的读操作时，它可以确定要读取的数据是否已被映射到系统缓存中。一个文件的共享缓存表同时也记录了所有指向该文件的文件对象中的私有缓存表，它们形成了一个链表。私有缓存表针对一个文件对象；而共享缓存表针对一个文件。如果两个文件对象是针对同一个文件而打开的，那么，它们的私有缓存表不同，但共享缓存表相同。图 7.5 显示了它们的关系。关于这两个缓存表的定义，请参考 base\ntos\cache\cc.h 文件。

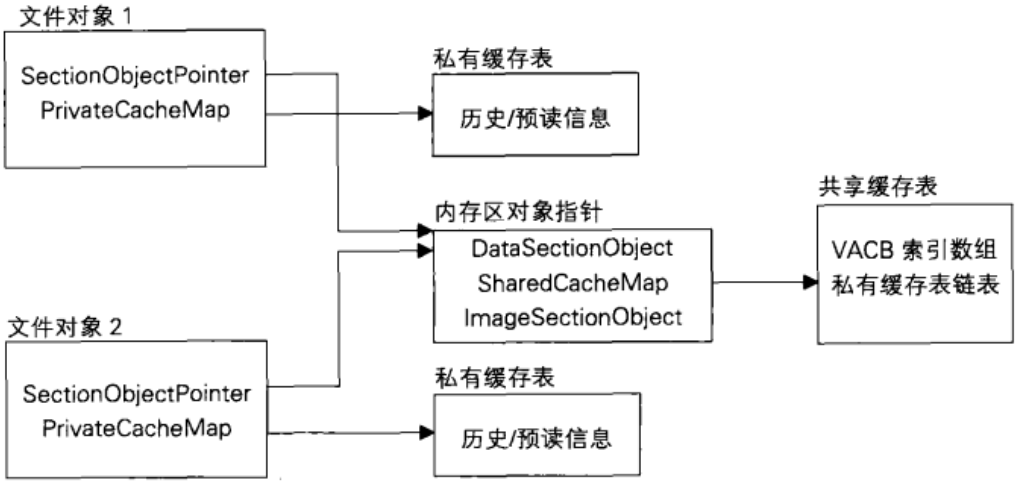


图 7.5 私有缓存表和共享缓存表

文件有大有小，共享缓存表中的 VACB 索引数组显然必须要适应文件的大小。VACB 索引数组的初始大小只包含 4 项，因此，对于不超过 1 MB 大小的文件，4 个项已经足够了。如果文件超过 1 MB，则缓存管理器从非换页内存池申请一个 VACB 索引数组，数组项个数等于该文件的大小除以 256 KB，然后，共享缓存表使用此 VACB 索引数组。

然而，当文件很大的时候，VACB 索引数组也会变得很大。而且，根据应用程序的访问模型，该数组有可能变成一个稀疏的索引数组，也就是说，虽然数组很大，但真正包含有效 VACB 的项数不多。为了应对空间使用效率的问题，缓存管理器使用多级索引数组来表达共享缓存表中的 VACB 索引。其做法是，当文件大小超过 32 MB 时，使用多级索引数组，每一级的每一个节点都是一个包含 128 项的数组，数组的每一项要么指向下一级的索引数组，要么指向系统缓存的 VACB 结构。

可以想象，若一个大文件的所有视图都同时被映射，那么，这一多级的索引数组将变成一个完全的 128 叉树，树上的叶子节点指向 VACB 结构。树的高度，即索引数组的级数，与文件大小的关系为：

$$Level = \lceil (\lceil \log_2 SizeOfFile \rceil - 18) / 7 \rceil$$

这里$\lceil \log_2 SizeOfFile \rceil - 18$代表文件 VACB 需要多少位来表达，减去 18 是因为每个 VACB 都表达了一个 256 KB 大小的文件块，然后除以 7 再向上取整即完全 128 叉树的高度。尽管这种表达方法在满映射的情况下，比单个索引数组需要更多的内存，但是，在只需映射部分文件数据的情况下，则可以节省内存空间。根据上述公式，级数与文件大小是指数关系，表 7.1 列出了 1~4 级索引数组所能表达的文件大小。譬如，只要文件的大小不超过 2^{39}=512 GB，则 3 级索引数组就足够描述它所有的 VACB 了。

表 7.1 VACB 索引数组的级数与文件大小的关系

共享缓存表的 VACB 索引数组	文件大小
固定的 4 项	0~1 MB
动态申请的索引数组（1 级）	1~32 MB
动态申请的索引数组（2 级）	32 MB~4 GB
动态申请的索引数组（3 级）	4 ~512 GB
动态申请的索引数组（4 级）	512 GB~64 TB

数据结构 SHARED_CACHE_MAP 中的 4 元素数组 InitialVacbs 域即内置的 VACB 索引数组，Vacbs 域指向真正使用的 VACB 索引数组。当文件大小不超过 1 MB 时，Vacbs 指向 InitialVacbs 的第一个元素。实现以上索引数组分配算法的函数是 CcCreateVacbArray 和 CcExtendVacbArray，请参考 base\ntos\cache\vacbsup.c 文件中的代码。其中 CcCreateVacbArray 函数分配索引数组，若是多级索引数组，则该函数仅分配第 1 级的 VACB 数组；CcExtendVacbArray 函数扩展一个已经建立好的共享缓存表。在代码中我们可以看到，当只需 1 级 VACB 索引数组（即文件大小为 1~32 MB）时，缓存管理器在构建 VACB 索引数组的同时，也建立了另一个称为 BCB（Buffer Control Block）链表的数组，其大小与 VACB 索引数组相同，关于 BCB 链表，后面 7.2.3 节将会进一步介绍。

系统缓存区域是系统地址空间中很大的一段地址范围，其中的视图被映射到相应文件中对应的视图上，全局的 VACB 数组变量 CcVacbs 描述了这一映射关系。当应用程序或系统读取系统缓存视图中的数据时，缓存管理器通过页面错误的途径将文件中的数据读入到缓存中。显然，系统缓存使用的物理内存属于操作系统，它们不被计算到应用程序的工作集中，而是系统工作集的一部分。注意，除了系统缓存中映射的物理页面以外，系统工作集还包含换页内存池、内核模块（ntoskrnl.exe）和设备驱动程序中可换页的代码和数据。

尽管在 Windows 源代码中，系统工作集的变量被称为 MmSystemCacheWs，但是，实际上，它是整个系统地址空间的工作集（当然，不包括非换页部分的内存）。系统工作集

链表为全局变量 MmSystemCacheWorkingSetList，该值是固定的，等于 0xc0c00000，如图 7.3 所示（图中称为系统缓存结构）。根据 4.6.1 节的介绍，工作集管理器对于全局的 MmWorkingSetExpansionHead 链表中的进程工作集按照统一的算法进行修剪，而系统工作集也加入到 MmWorkingSetExpansionHead 链表中，这是在缓存管理器初始化函数 MiInitializeSystemCache 中，作为最后一个步骤调用 MiAllowWorkingSetExpansion 函数来完成的，参见 MiInitializeSystemCache 函数的代码。因此，当系统物理内存紧缺时，内存管理器会一视同仁地回收系统缓存中占用的物理页面。

有一点值得在此说明一下，一个物理页面被从系统工作集（或进程工作集）中移除，并不意味着这个页面中的数据立即没有用了。相反，如果该页面中包含了有用的数据，那么，该页面将被转移到内存管理器的备用链表或修改链表中，通常不会立即变成空闲页面供系统挪作他用。关于物理页面的状态变换，参见 4.5.2 节。

Windows 内核中有一组变量记录了系统工作集与系统缓存中的内存使用情况，分别如表 7.2 和表 7.3 所示。关于表 7.2 中变量的使用情况，请参考 MiUpdateWsle 和 MiRemoveWsle 函数中为每个计数变量所设定的条件。

表 7.2 系统工作集的使用情况

内核变量	含义
MmSystemCachePage	系统缓存所占用的物理内存
MmSystemCodePage	内核模块（ntoskrnl.exe）可换页代码所占物理内存
MmSystemDriverPage	驱动程序可换页代码所占物理内存
MmPagedPoolPage	换页内存池所使用的物理内存

表 7.3 系统工作集 MmSystemCacheWs 中的信息

内核变量	含义
MmSystemCacheWs.WorkingSetSize	系统工作集总大小，等于表 7.2 中的 4 项之和
MmSystemCacheWs.PageFaultCount	系统工作集中发生的页面错误数量
MmSystemCacheWs.PeakWorkingSetSize	系统工作集的尖峰大小

在这一节的最后，我们来讨论一下缓存管理器中的视图与内存管理器中通过内存区对象建立的视图映射之间的区别。在 4.3.4 节介绍内存区对象时，我们已经看到，内存管理器可以将一个数据文件、可执行文件或者页面文件中的一部分映射到进程的虚拟地址空间中，从而允许应用程序通过访问内存的做法来访问文件。内存管理器在页面错误时将文件中的数据读入到物理内存中。当物理内存紧缺时，所占用的物理页面有可能被换出来，这

是通过修剪进程工作集来做到的。因此，这种视图映射完全是由内存管理器来实现的，与缓存管理器无关。它带来的好处有：实现了多个进程之间共享访问内存或文件；在内存访问时所用到的物理内存由工作集管理器统一调度。

缓存管理器则是介于文件系统与内存管理器之间的一个组件。它将文件数据以 256 KB 大小的“视图”为单位，动态地映射到系统缓存区域中，从而为文件系统提供了“按需缓存”的能力。缓存管理器还支持异步的预读（read ahead）和延迟写（lazy write）特性，并且支持快速 I/O 以提高读写操作的性能。缓存管理器对于应用程序是透明的，它的主要用户是文件系统驱动程序，其他的内核组件，像提供文件服务的内核模块，或者需要访问外存的模块（比如配置管理器），也可以使用缓存管理器来访问文件系统中的文件数据。

7.2.2 缓存管理器的数据访问路径

为了使用缓存管理器提供的文件缓存能力，当一个文件被第一次执行读或写操作时，如果该文件的属性指明了它支持以缓存方式来访问，那么，文件系统驱动程序调用缓存管理器的函数 CcInitializeCacheMap，为该文件建立起缓存访问环境。CcInitializeCacheMap 函数的原型如下：

```
VOID
CcInitializeCacheMap (
    __in PFILE_OBJECT FileObject,
    __in PCC_FILE_SIZES FileSizes,
    __in BOOLEAN PinAccess,
    __in PCACHE_MANAGER_CALLBACKS Callbacks,
    __in PVOID LazyWriteContext
    );
```

这里，FileObject 参数指向要被缓存的文件；FileSizes 参数告诉缓存管理器该文件的文件大小（FileSize）、占用设备存储空间的大小（AllocationSize）和有效数据长度（ValidDataLength）。布尔参数 PinAccess 表示是否将来使用锁定（pin）的方式（一组 CcPin*Xxx* 函数，见本节后文的介绍）来使用缓存。Callbacks 参数的回调结构包含了 4 个可能的异步回调例程，分别用于预读和延迟写出的回调通知。LazyWriteContext 参数指定了传递给 Callbacks 中例程的信息。

注意，CcInitializeCacheManager 是初始化缓存管理器的函数，它是在系统的阶段 1 初始化过程中被调用的，而 CcInitializeCacheMap 是针对每个要支持缓存的文件而被调用的，它的职责是建立起该文件的缓存环境，包括：创建共享缓存表；若有必要还创建内存区对象和 VACB 索引数组（分别通过 MmCreateSection 和 CcCreateVacbArray 函数），或者扩展内存区对象和 VACB 索引数组（分别通过 MmExtendSection 和 CcExtendVacbArray

函数）；另外，若有必要，为文件对象构造一个私有缓存表。CcInitializeCacheMap 函数的代码位于 base\ntos\cache\fssup.c 文件的 322~1 088 行。

此外值得提及的是，我们在上一章中看到，应用程序在访问设备对象时，利用一个名称来创建一个文件对象，并且获得一个文件句柄，以后通过文件句柄访问驱动程序中的例程。关于文件对象的创建过程，参见 6.2.3 节的介绍，其中文件对象是在 IopParseDevice 函数中被创建的。这里，文件对象是设备对象的一个接口。然而，在本章介绍缓存管理器（以及后文介绍文件系统）的上下文环境中，文件对象代表一个数据流，也即我们平时熟悉的文件系统中的文件。所以，文件对象表达了磁盘上的一些数据，至于磁盘上的数据是如何有序地组织起来的，则是文件系统的职责；而如何访问磁盘上的文件数据，则是磁盘驱动程序的职责。本章后面将会介绍文件系统和磁盘的数据管理，现在我们只需简单地认为文件对象表达了文件系统中的一个数据流，其接口与上一章介绍的 FILE_OBJECT 完全相同。

因此不难理解，CcInitializeCacheMap 函数是由文件系统驱动程序来调用的，对于绝大多数文件系统，除非文件创建时被指定了 FILE_NO_INTERMEDIATE_BUFFERING 标志，否则文件的缓存功能总是被允许的。在 NtReadFile 和 NtWriteFile 函数中，我们可以看到，若文件对象的 Flags 域包含了 FILE_NO_INTERMEDIATE_BUFFERING，那么，它们所发出的 I/O 请求的 IRP 对象的 Flags 域将包含 IRP_NOCACHE 标志，因而会阻止文件系统驱动程序为文件对象建立缓存环境。这里，文件对象的 FILE_NO_INTERMEDIATE_BUFFERING 标志对应于应用程序在调用 API 函数 CreateFile 时指定的 FILE_FLAG_NO_BUFFERING 标志。

此外，若系统需要从文件中读取数据到内存页面中，譬如在发生页面错误或加载可执行映像文件时，Windows 通过一个专门的 I/O 接口函数 IoPageRead 来完成这样的 I/O 操作。在 IoPageRead 函数发出的 I/O 请求中，IRP 对象的 Flags 域也包含 IRP_NOCACHE 标志，从而绕过了缓存管理器，直接通知低层驱动程序读取数据。类似地，I/O 管理器也提供了一个用于将页面写出到存储设备的接口函数 IoAsynchronousPageWrite，该函数也使用了 IRP_NOCACHE 标志，以指示文件系统驱动程序绕过缓存管理器直接通知低层驱动程序写数据。

图 7.6 显示了文件对象 I/O 操作的路径。当文件系统驱动程序接收到 I/O 请求时，若 IRP 对象的 Flags 域中未指定 IRP_NOCACHE，则文件系统驱动程序通常会使用缓存管理器来读或写文件数据，缓存管理器提供的接口函数为 CcCopyRead、CcCopyWrite、CcFastCopyRead 和 CcFastCopyWrite，这些函数的代码位于 base\ntos\cache\copysup.c 文件中。

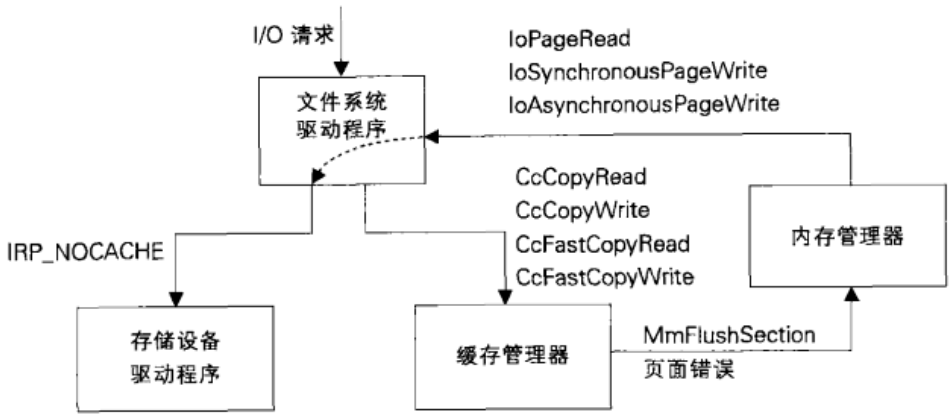


图 7.6 I/O 请求经过缓存管理器的路径

CcCopyRead 函数的功能是，把指定的文件数据从缓存拷贝到调用者指定的输出缓冲区中。调用者在参数中指定要读取的文件对象（FileObject）、偏移位置（FileOffset）和长度（Length）信息，并且指定此读操作是否要等待完成（Wait）。对于非等待的读请求，若缓存管理器能立即提供所要读取的数据，则 CcCopyRead 立即拷贝数据，并返回 TRUE；若不能立即提供所需数据，则函数返回 FALSE。若调用者要求等待，则该函数总是拷贝数据，并返回 TRUE，但有可能要阻塞一段时间。

CcCopyRead 函数的实现比较直截了当，它首先判断当前的活动视图，若要读取的数据部分或全部落在活动视图中，则将活动视图中的数据拷贝出来，并根据情况释放或保留该活动视图。若活动视图中的数据不足以满足要求，则调用 CcGetVirtualAddress 函数映射指定文件偏移处的视图。然后从新的视图中拷贝数据，这样循环进行，直至要读取的数据全部拷贝到输出缓冲区中。最后未读完数据的视图成为新的活动视图。有关代码的细节，请参考 copysup.c 文件的 33~589 行。

然而，映射视图的操作并不那么简单，缓存管理器需要内存管理器的紧密配合才能完成。在 CcGetVirtualAddress 函数中，如果指定的文件偏移处所在的视图尚未被映射，则调用 CcGetVacbMiss 来映射一个新的视图。这是缓存管理器中两个主要的内部函数，其代码位于 base\ntos\cache\vacbsup.c 文件中。CcGetVacbMiss 函数首先找到一个空闲的 VACB，然后调用 MmMapViewInSystemCache 函数，从系统缓存区域中申请一个 256 KB 大小的视图，并映射到目标文件的内存区对象中，建立起该视图的页表中的原型 PTE 项。关于内存区对象的管理机制，参见 4.3.4 节。内存管理器专门为缓存管理器提供了一组接口，包括 MmMapViewInSystemCache 和 MmUnmapViewInSystemCache 函数，参见 base\ntos\mm\mapcache.c 文件中的代码。一旦在该视图的页表中建立起原型 PTE，

CcCopyRead 即可直接对该视图的内存进行读访问，当页面错误发生时，内存管理器从文件中读取数据。

CcCopyWrite 执行写数据到文件的操作，首先将可能落在当前视图范围内的数据通过内存拷贝的方式写出。然后，对于未落在当前视图范围的数据，调用 CcMapAndCopy 函数来完成写数据任务。在 CcMapAndCopy 函数中，首先通过 CcGetVirtualAddress 函数，获得该文件中一个视图被映射后的虚拟地址，然后，或者通过拷贝操作，将用户缓冲区中的数据拷贝到系统缓存的视图页面中（有可能触发页面错误从而获得物理页面），或者调用内存管理器提供的函数 MmCopyToCachedPage，来完成物理页面的分配和拷贝操作。为了真正写出数据，CcMapAndCopy 或者调用 MmFlushSection 函数以便将写到系统缓存中的数据刷新到外部存储设备，或者调用 MmSetAddressRangeModified 或 CcSetDirtyInMask 函数标记这些数据已脏，请内存管理器或缓存管理器异步写出这些数据。在 MmFlushSection 函数中，内存管理器通过 IoAsynchronousPageWrite 或 IoSynchronousPageWrite 函数将页面写出。

CcFastCopyRead 是 CcCopyRead 的一个变形，它总是使用同步方式，相当于 CcCopyRead 中的 Wait 参数为 TRUE，另外它使用了 32 位文件偏移。CcFastCopyWrite 是 CcCopyWrite 的一个变形。同样地，CcFastCopyWrite 使用 32 位文件偏移，并且总是使用同步的文件写操作。有兴趣的读者可以参考这两个函数的代码，它们也位于 copysub.c 文件中，这里不再赘述。

因此，我们看到，当一个缓存的 I/O 请求（即，它的 IRP 对象的 Flags 域不包含 IRP_NOCACHE 标志）到达文件系统驱动程序时，它通过以上描述的缓存拷贝函数来读或写数据，而缓存管理器则利用内存管理器的页面管理能力，使得读写操作可以方便地在内存中进行。真正读写数据是由内存管理器调用 I/O 管理器的 IoPageRead、IoAsynchronousPageWrite 或 IoSynchronousPageWrite 函数来完成的，这些 I/O 函数发出的 I/O 请求都是非缓存的（包含 IRP_NOCACHE 标志），因此，在文件系统驱动程序这里 I/O 请求不会形成无限循环。缓存管理器介入其中从而形成一个圈，这种做法看似增加了 I/O 操作的复杂性，但却带来了两方面重要的好处：第一，增强了异步 I/O 的能力；第二，I/O 操作的中间缓存的统一管理，使得文件系统驱动程序无须为了 I/O 操作而额外地管理缓存。

7.2.3 直接使用系统缓存中的数据

上一小节介绍了当文件系统驱动程序要拷贝缓存中的数据时缓存管理器所提供的支持，然而，在有些情况下，文件系统驱动程序更期望直接访问缓存中的数据，比如，它们需要维护文件系统中的目录信息或者其他的元数据。试想，如果它们不能直接访问

缓存，那么，为了维护文件系统的元数据，一种做法是，每次修改或读取元数据，都从存储设备中获得目标数据，若修改的话，改完以后再写回去，此过程可能还要涉及一个中间缓冲区；另一种做法是，文件系统驱动程序自身维护一个缓存，修改或读取元数据的操作在缓存上进行，只有必要时才读写外部存储设备。幸运的是，缓存管理器提供了直接访问系统缓存视图的能力，因而文件系统驱动程序只需很少的努力就可以方便地操纵文件系统中的元数据。

缓存管理器提供了一组接口函数专门用于这一目的。如果文件系统驱动程序需要读取缓存中的数据，那么，它调用缓存管理器的映射函数 CcMapData；如果需要读/写缓存中的数据，则调用 CcPinRead、CcPreparePinWrite 或 CcPinMappedData 函数，映射并锁定（pin）缓存中的数据。文件系统驱动程序在写完了缓存中的数据以后，需要调用 CcSetDirtyPinnedData，以显式地通知缓存管理器缓存中的数据已经修改了。当文件系统驱动程序不再读或写缓存中的数据时，它调用 CcUnpinData，以便让这些数据解除锁定。

在介绍这些函数的实现以前，我们首先看一下共享缓存表中除了 VACB 以外的另一个数据结构 BCB（Buffer Control Block），其用途是描述被锁定（pin）在内存中的文件数据。如果缓存管理器要实现锁定功能，允许文件系统驱动程序直接访问缓存中的数据，则仅仅依赖于共享缓存表中的 VACB 是不够的，BCB 是 VACB 的一个补充，其定义如下：

```
typedef struct _BCB {

   union {
      MBCB Dummy;
      struct {
         CSHORT NodeTypeCode;                        // 类型码
         BOOLEAN Dirty;                              // 脏标记
         BOOLEAN Reserved;                           // 保留
         ULONG  ByteLength;                          // 整个缓冲区的字节长度
         LARGE_INTEGER FileOffset;                   // 整个缓冲区的文件偏移
         LIST_ENTRY BcbLinks;                        // 在共享缓存表内部的BCB链表
         LARGE_INTEGER BeyondLastByte;               // 缓冲区内最后一个字节的文件偏移
         LARGE_INTEGER OldestLsn;                    // 当此缓冲区被设置为脏时，最老的
                                                     // LSN（逻辑序列号）
         LARGE_INTEGER NewestLsn;                    // 当此缓冲区被设置为脏时，最新的LSN
         PVACB Vacb;                                 // 此BCB被映射在哪个视图上
         ULONG PinCount;                             // 锁定（pin）计数
         ERESOURCE Resource;                         // 执行体资源锁，保护对缓冲区的访问
         PSHARED_CACHE_MAP SharedCacheMap;           // 指向共享缓存表
         PVOID BaseAddress;                          // 缓冲区的基地址，位于系统地址空间
      };
   };
} BCB;
```

以上 BCB 结构中域的含义不难理解，唯一值得一提的是 LSN（逻辑序列号）的概念，它是文件系统日志服务用于标识元数据记录版本序列的一个值，值越大说明记录越新。从

BCB 结构的定义可以看出，每个 BCB 都描述了一个视图内部被映射和锁定的缓冲区，它指向该视图的 VACB，以及所属文件的共享缓存表。

我们已经知道，VACB 是一个全局数组，每个元素代表系统缓存区域中的一个视图，共享缓存表使用一个 VACB 索引数组来引用全局数组中的 VACB 元素。BCB 对象是从非换页内存池中动态申请的，每个 BCB 对象代表了一个视图内部被锁定的一段字节范围。一个视图内部可以有零个或多个 BCB 对象，这些 BCB 构成了一个双链表。实际上，一个文件中所有的 BCB 形成了一个双链表，共享缓存表的 BcbList 域指向此双链表的头。此外，缓存管理器以两个视图（即 512 KB 大小）为粒度来管理 BCB 对象，每 512 KB 可能会在 BCB 链表中插入一个链表头节点。所谓链表头节点，实际上是一个类型为 LIST_ENTRY 的对象。当文件大小位于 1 MB~32 MB 之间时，缓存管理器在初始化 VACB 索引数组的同时，也会建立起 BCB 链表，如图 7.7 所示。BCB 链表头数组部分的大小与 VACB 索引数组部分相同，因为每个 LIST_ENTRY 对象在 32 位系统上为 8 字节，PVACB 大小为 4 字节，而每个链表头负责 2 个视图的字节范围，PVACB 指向一个视图的 VACB 元素。CcCreateVacbArray 函数除了分配好 BCB 链表头，也将它们插入到以共享缓存表的 BcbList 域为头部的链表中。注意，BcbList 是一个双链表，其节点按字节范围的降序插入。以后，当有 BCB 对象被插入时，根据它的文件偏移位置，按降序插入在适当的链表头节点之间，如图 7.7 中的阴影节点所示。

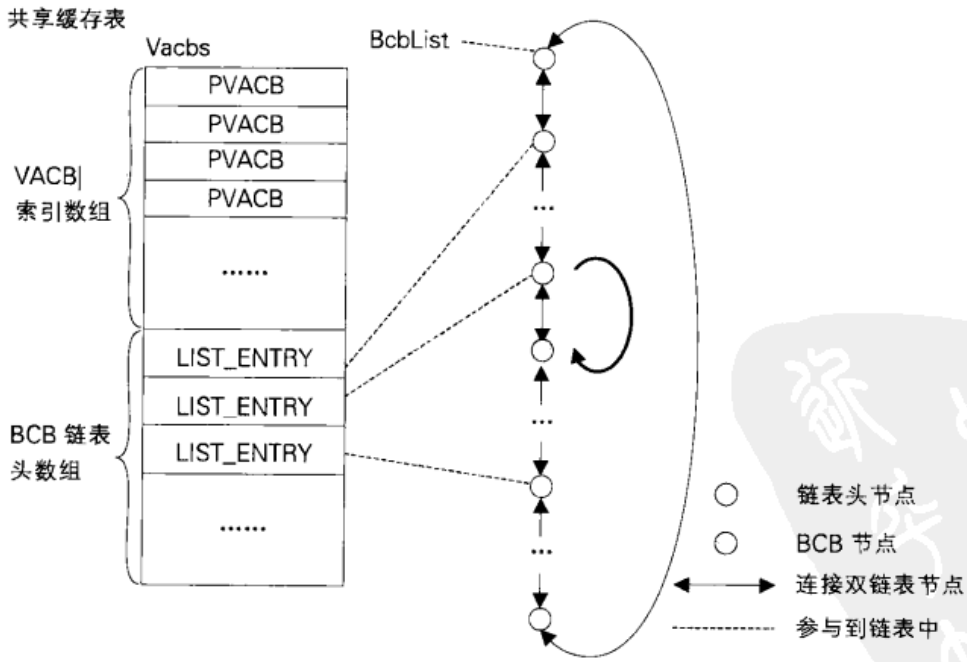


图 7.7 VACB 和 BCB 结构示意图

图 7.7 也可以扩展到多级 VACB 结构，除了第 1 级的 127 个 VACB 索引数组以外，其他级上所有的 127 项 VACB 索引数组都配有类似于图 7.7 那样的 BCB 链表头数组(实际上，在链表头节点数组后面还有一个称为 VACB_LEVEL_REFERENCE 的 8 字节结构）。多级 VACB 索引数组可以管理稀疏文件，或者虽然尺寸很大但只有少数位置被映射的文件，在这些情况下，对应的 BCB 链表头节点并不总是出现在共享缓存表的 BCB 链表中。图 7.8 从文件偏移分布的角度来显示 BCB 链表，适用于单级或多级 VACB 结构的共享缓存表，只不过，对于多级 VACB 结构的情形，并不是所有的 512 KB 整数倍位置处都有链表头节点。

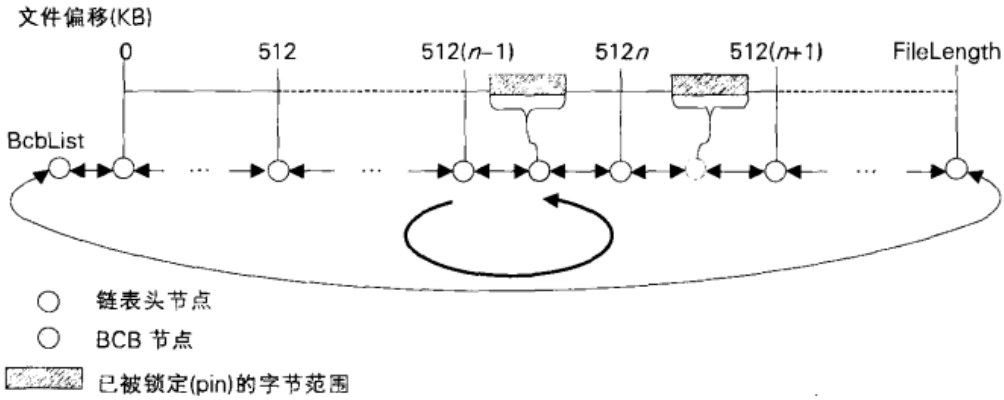


图 7.8 从文件偏移角度来看 BCB 链表

每个 BCB 节点描述了一段被映射并且被锁定的范围，其起始地址和结束地址均以页面为对齐边界，且不会跨越视图边界，即不会越过 256 KB 的边界，当然更不会越过 512 KB 的 BCB 链表头边界。而且，同一个 512 KB 区间内的多个 BCB 节点相互之间不会重叠，它们各自描述一段字节范围。在 Windows 内核中以下函数体现了 BCB 的管理机制：

- CcGetBcbListHead/CcGetBcbListHeadLargeOffset。CcGetBcbListHead 函数返回指定偏移处的链表头节点。其中，对于多级 VACB 索引数组的文件（即大于 32 MB 的文件对象），通过调用 CcGetBcbListHeadLargeOffset 来完成。这两个函数中的参数 FailToSuccessor 指明了当未能找到指定位置处的链表头时该往前还是往后找。注意，BcbList 链表是有方向的，如图 7.7 和图 7.8 所示。
- CcFindBcb。该函数用于找到一个描述了指定字节范围的 BCB。当它能找到一个描述了指定范围的 BCB，哪怕只是指定字节范围的前端一部分时，它返回 TRUE；若字节范围的前端部分未出现在任何一个 BCB 中，则返回 FALSE。CcFindBcb 调用 CcGetBcbListHead 先找到指定字节偏移前面处的链表头结构，然后从该链表头的下

一个节点开始搜索，因而 CcFindBcb 并不需要线性遍查 BcbList。注意，在 CcFindBcb 的代码中，我们可以看到，搜索的停止条件是 BCB 节点的 NodeTypeCode 域不等于 CACHE_NTC_BCB（宏定义，值为 0x2fd）。这里使用了一个小诀窍：若是 BCB 节点，NodeTypeCode 值等于 CACHE_NTC_BCB；若是链表头节点，则它本身只是一个 LIST_ENTRY 结构，而并非 BCB 结构，所以，从链表头节点的地址减去 16（即 BCB 结构中 BcbLinks 与 NodeTypeCode 两个域之间的偏移），结果位置处应该是一个地址值（参考图 7.7），其低 16 位不可能出现 0x2fd。

- CcAllocateInitializeBcb。该函数从非换页内存池中申请一个 BCB 对象，然后初始化其中的域，再插入到共享缓存表的 BcbList 链表中指定 BCB 对象的后面。

- CcPinFileData。这是用 BCB 机制来锁定文件字节范围的核心函数，它将指定范围的文件数据锁定在内存中，以便文件系统驱动程序直接以内存访问的方式来操纵文件数据。在执行该函数以前，指定的文件字节范围有三种可能的状态：尚无 BCB 对象描述此范围，已有 BCB 对象描述此范围但尚未映射，以及已有 BCB 对象描述此范围并且已经映射了这段范围。再根据参数 Flags 中指定是否要等待该锁定操作完成。CcPinFileData 函数共处理 6 种可能的情形，所以，该函数的代码实际上是这 6 种情形的组合，位于 base\ntos\cache\cachesub.c 文件的 119~965 行，其中映射操作是通过调用 CcMapAndRead 函数来完成的。

- CcUnpinFileDataEx/CcUnpinFileData。该函数执行与 CcPinFileData 相反的操作，它解除一个 BCB 对象的映射和锁定。其参数是一个通过 CcPinFileData 得到的 BCB 对象。另一个参数 UnmapAction 指定了这是一个 UNPIN 还是 SET_CLEAN 操作。BCB 对象是通过 PinCount 计数来维护其生命周期的，当 PinCount 减到 0 时，CcUnpinFileDataEx 调用 CcFreeVirtualAddress 释放虚拟地址，若 BCB 对象不再有用，则调用 CcDeallocateBcb 函数释放该对象。若是 SET_CLEAN 操作，则把所有的脏状态都清除。CcUnpinFileData 是一个宏，等于 CcUnpinFileDataEx。

映射和锁定功能为文件系统驱动程序提供了直接访问文件数据，特别是文件系统元数据的方便。将文件数据映射到内存中，这一能力是通过上一小节介绍的视图映射来实现的。CcPinFileData 也调用 CcGetVirtualAddress 函数来映射给定的视图，从而允许以内存访问的方式来操纵文件数据。VACB 视图的生命周期是通过其 ActiveCount 成员来控制的，CcGetVirtualAddress 函数增加 ActiveCount 计数值，对应地，CcFreeVirtualAddress 函数减小 ActiveCount 计数值，当 ActiveCount 计数减到 0 时，此 VACB 被转移到全局的

CcVacbFreeList 链表中。为了解除视图映射，缓存管理器必须显式地调用 CcUnmapVacbArray 或 CcUnmapVacb，而 CcUnmapVacb 则调用内存管理器提供的专门接口函数 MmUnmapViewInSystemCache 来解除映射。

锁定操作本身并没有提供额外的内存管理能力，每一个锁定操作都会增加 BCB 对象中的 PinCount 计数，相应地，每次解除锁定的操作都会减小 PinCount，当 PinCount 减到 0 时，触发 CcFreeVirtualAddress 调用，参见 CcUnpinFileDataEx 的代码。请注意，这里的锁定（pin）操作与内存管理器中通过 VirtualLock 将页面锁在物理内存中的锁定（lock）操作并不相同，在 VirtualLock 的情形下，一旦锁定，则页面将不会被换出到磁盘上。而这里的锁定含义是保持这一字节范围的页面映射，并且文件系统驱动程序通常将这些页面标记为“禁止修改写出”（共享缓存表的 Flags 域包含 MODIFIED_WRITE_DISABLED 标志，除非它们是只读页面），因而修改页面写出器或映射页面写出器不会将这些已修改的页面写出。当文件系统驱动程序显式地通知写出这些页面时，内存管理器才将所有的修改写到磁盘上。

Windows 内存管理器专门用一个链表（MmModifiedNoWritePageListHead）来记录这些修改但不写出的页面，只有当内核代码显式地将它们转移到譬如修改页面链表 MmModifiedPageListHead 中时，它们才会被刷新到磁盘上。CcUninitializeCacheMap 函数调用 MmEnableModifiedWriteOfSection 函数来实现内存区对象中这些页面的链表转移，请参考这两个函数的代码，以及 4.5.3 节关于修改但不写出页面的管理。

有了以上这些内部管理知识，现在我们来看一看缓存管理器提供给文件系统驱动程序直接操纵文件系统数据的接口函数，如下所列：

- CcMapData。它调用 CcMapDataCommon 函数来映射指定的文件数据。如果 CcMapData 的参数 Flags 指定了这是一个同步映射（即 MAP_WAIT），那么，CcMapDataCommon 调用 CcGetVirtualAddress 函数来映射视图，得到的 VACB 被当做 BCB 返回；否则，CcMapDataCommon 调用 CcPinFileData 函数，完成映射并获得 BCB。
- CcPinRead。此函数的用意是将指定的文件数据锁定在缓存中，它调用 CcPinFileData 来完成实际的映射和锁定操作。
- CcPreparePinWrite。该函数将指定的文件数据锁定在缓存中，并返回一个指针指向此文件数据。如果调用者指定了 PIN_CALLER_TRACKS_DIRTY_DATA 标志，则说明调用者自己负责跟踪脏页面，它将来会通过调用 MmSetAddressRangeModified 函数来刷新这些脏页面。在这种情况下，CcPreparePinWrite 调用 CcMapDataForOverwrite 函

数，而 CcMapDataForOverwrite 进一步调用 CcMapDataCommon 函数来映射文件数据。若 CcPreparePinWrite 的 Flags 参数不指定此特殊标志，则 CcPreparePinWrite 调用 CcPinFileData 来完成实际的映射和锁定工作。

- CcPinMappedData。该函数将一块已经映射的文件数据锁定在缓存中。该函数也通过调用 CcPinFileData 函数来完成实际的锁定工作。
- CcSetDirtyPinnedData。该函数的用意是将 CcPinFileData 返回的 BCB 设置为脏，从而使缓存管理器的延迟写出器（lazy writer，参见 7.2.5 节）考虑将脏页面写出到磁盘上。该函数要求提供一个 BCB 和一个 LSN 作为参数。
- CcUnpinData。该函数调用 CcUnpinFileDataEx 执行解除映射和锁定的工作。凡是调用过 CcMapData、CcPinRead、CcPreparePinWrite 和 CcPinMappedData 函数得到的 BCB，若不再使用该范围的文件数据，都需要调用 CcUnpinData。

以上提供的映射和锁定函数将文件数据映射到系统地址空间中，允许文件系统驱动程序以访问内存的方式来访问文件数据。除此以外，缓存管理器还提供了直接读或写系统缓存页面的接口函数，这对于使用物理地址作为访问途径的存储设备提供了方便，它可以避免一次数据拷贝，当传输数据量较大时尤其有意义。例如，远程文件系统可以直接在网络设备和物理内存之间传输数据，而无须经过一个中间缓冲区。

这一接口称为缓存管理器的 DMA 接口，它使用内存管理器的 MDL（内存描述符表）来描述物理内存。主要的接口函数如下：

- CcMdlRead。该函数首先为文件数据映射视图，这可以通过 CcGetVirtualAddress 函数来实现。然后调用 IoAllocateMdl 函数，申请一个 MDL 对象。接下来，调用内存管理器提供的专门接口函数 MmProbeAndLockPages，将该视图映射的物理页面锁定（lock）在内存中，并相应地更新 MDL 对象。然后，CcMdlRead 调用 CcFreeVirtualAddress，解除视图映射，但是这些物理页面仍然被锁定在内存中。此过程会一直进行，直到调用者指定的文件范围都已处理过。最后，调用者获得一个描述了这些物理页面的 MDL 链。
- CcMdlReadComplete/CcMdlReadComplete2。CcMdlReadComplete2 函数是在 CcMdlRead 之后被调用，它只是简单地调用 MmUnlockPages 函数以解除 MDL 对象中描述的物理页面的锁定，然后调用 IoFreeMdl 以释放 MDL 对象。CcMdlReadComplete 调用 CcMdlReadComplete2 来完成相同的任务。

- CcPrepareMdlWrite。该函数与 CcMdlRead 类似，它在调用了 CcGetVirtualAddress 函数以后接着调用 CcMapAndRead，以便映射文件数据并且分配零页面（并未从文件中读取数据）。然后分配 MDL 并执行 MmProbeAndLockPages，最后调用 CcFreeVirtualAddress 解除页面映射。同样地，调用者获得一个描述了物理页面的 MDL 链。
- CcMdlWriteComplete/CcMdlWriteComplete2/CcMdlWriteAbort。这三个函数对应于 CcPrepareMdlWrite 的后处理。首先看 CcMdlWriteAbort 函数，这是一个取消 MDL 中描述的文件数据的写操作，它调用 MmUnlockPages 函数解除对 MDL 页面的锁定，并调用 IoFreeMdl 释放 MDL。CcMdlWriteComplete2 函数调用 MmUnlockPages 函数解除对 MDL 对象中描述的物理页面的锁定，然后调用 MmFlushSection 或 CcSetDirtyInMask 函数以便将物理页面中的数据写到文件中，最后调用 IoFreeMdl 释放 MDL。CcMdlWriteComplete 调用 CcMdlWriteComplete2 来完成相同的任务。

以上我们看到了缓存管理器利用内存管理器提供的内存映射能力，为客户代码提供了直接访问虚拟内存或物理内存的能力，缓存管理器通过页面错误或 MmFlushSection 来读或写文件数据。下面的小节我们将讨论缓存管理器如何有效地管理实际的文件读写操作。

7.2.4 缓存管理器的预读处理

我们首先来看一下缓存管理器提供的文件预读能力。前面曾经提到，文件对象的私有缓存表记录了最近两次读访问的历史信息，缓存管理器利用这两次读访问的信息来预测下一次读请求，并提前完成这一读操作。另一方面，由于缓存管理器通过页面错误来读取文件数据，因此，它也指示页面错误处理例程，一次读入足够的页面数量。

除了最近两次读访问的信息，缓存管理器的预读算法也利用了文件对象被创建时设定的两个标志：FILE_FLAG_SEQUENTIAL_SCAN 和 FILE_FLAG_RANDOM_ACCESS。若客户代码指定了 FILE_FLAG_SEQUENTIAL_SCAN 标志，则缓存管理器不再使用历史信息，而是执行顺序预读；若客户代码指定了 FILE_FLAG_RANDOM_ACCESS 标志，则缓存管理器关闭预读特性，不再预测客户代码将来在哪个位置上读取数据。

对页面错误处理例程的指示是通过MmSetPageFaultReadAhead宏来实现的，其定义如下：

```
#define MmSetPageFaultReadAhead(Thread,ReadAhead) {                    \
            (Thread)->ForwardClusterOnly = TRUE;                       \
            if ((ReadAhead) > MM_MAXIMUM_READ_CLUSTER_SIZE) {          \
               (Thread)->ReadClusterSize = MM_MAXIMUM_READ_CLUSTER_SIZE;\
            } else {                                                   \
               (Thread)->ReadClusterSize = (ReadAhead);                \
            } }
```

这里的宏 MM_MAXIMUM_READ_CLUSTER_SIZE 为 15。若 ReadAhead 值为 0，则页面错误处理例程仅读入发生访问错误的页面，否则读入该页面以及后续的 ReadAhead 个页面。在页面错误处理过程中，MiResolveMappedFileFault 函数将参考线程对象的 ReadClusterSize 信息，以便一次读入多个页面的数据。

缓存管理器在执行读操作时，譬如在 CcCopyRead、CcFastCopyRead 和 CcMdlRead 中，都会使用 MmSetPageFaultReadAhead 宏来指示页面预读。内存管理器自身也会使用它来执行页面预读。在这些函数中，我们还会看到 MmSavePageFaultReadAhead 和 MmResetPageFaultReadAhead 宏，其职责是保存和恢复线程中的页面预读设置。

除了通过页面错误处理例程来执行顺序的预读，缓存管理器也根据历史信息来执行预读。首先，它在 CcCopyRead、CcFastCopyRead 和 CcMdlRead 函数中，每次都在私有缓存表中记录下这一次读操作的起始和结束位置。私有缓存表的 FileOffset2 和 BeyondLastByte2 域记录了最近一次读操作的信息，而 FileOffset1 和 BeyondLastByte1 域记录了前一次读操作的信息。这些读文件数据的函数调用 CcScheduleReadAhead 函数来安排针对同一个文件的预读，其中传递给该函数的参数指定了刚刚完成或刚刚发起的读操作的信息。

CcScheduleReadAhead 函数的代码位于 base\ntos\cache\cachesup.c 文件的 1 240~1 616 行，它根据三种情形来计算预读要求。第一种情形是顺序读，即文件对象指定了 FO_SEQUENTIAL_ONLY 标志，CcScheduleReadAhead 确保私有缓存表的 ReadAheadOffset[0]记录了当前所需要读取的文件位置，而 ReadAheadOffset[1]记录了顺序往后的文件位置；第二种情形是，当前的读操作恰好是三次连续的顺序读操作中的第三次读操作，CcScheduleReadAhead 设置 ReadAheadOffset[1]为当前操作之后的文件位置；第三种情形属于条带式访问模式，即当前读操作的位置减去上一次读操作的位置（私有缓存表的 FileOffset2 域），等于上一次读操作的位置减去再往前一次读操作的位置（私有缓存表的 FileOffset1 域），在这种情况下，CcScheduleReadAhead 预测下一次读操作也符合这样的等差关系，因而 ReadAheadOffset[1]等于当前读操作位置的两倍减去上一次读操作的位置。

计算了私有缓存表的预读位置 ReadAheadOffset[1]和预读长度 ReadAheadLength[1]以后，CcScheduleReadAhead 调用 CcAllocateWorkQueueEntry（实际上是一个宏），从一个全局的快查表 CcTwilightLookasideList 中申请一个工作队列项（数据类型为 WORK_QUEUE_ENTRY），并设置该工作队列项的函数类别（即 Function 域）为 ReadAhead，以及指定相应的文件对象参数。最后调用 CcPostWorkQueue，将工作队列项插入全局的工作队列链表 CcExpressWorkQueue 中。

简而言之，CcScheduleReadAhead 函数所做的工作是，根据当前读操作的信息，计算新的预读位置和长度，存放在私有缓存表的 ReadAheadOffset[1]和 ReadAheadLength[1]成员中，然后在全局的工作队列链表中插入一个队列项。那么，谁来处理这一队列项呢？

参考 CcPostWorkQueue 函数的代码（见 base\ntos\cache\lazyrite.c 文件），我们可以看到，CcPostWorkQueue 将工作队列项插入到参数中指定的工作队列链表中，然后判断当前是否正在进行节流控制（write throttling），以及是否有空闲的辅助线程，若满足条件，则激活一个辅助线程。缓存管理器的辅助线程实际上是系统工作项目（WORK_QUEUE_ITEM），它们在缓存管理器初始化时已经被分配好，见 CcInitializeCacheManager 函数的代码。对于大内存系统，缓存管理器的辅助线程数量等于系统全局的关键辅助线程数减去 2(ExCriticalWorkerThreads-2)。因此，缓存管理器的后台任务，包括预读操作和后文将要讲到的延迟写操作，是受严格控制的，以免集中执行时严重地影响系统的性能。

CcPostWorkQueue 将缓存管理器辅助线程链表（CcIdleWorkerThreadList）中的一项，即一个工作项目（WorkItem），通过调用 ExQueueWorkItem 函数由系统调度执行。由于 CcIdleWorkerThreadList 链表中的系统工作项目被初始化为 CcWorkerThread 函数，因此，实际被调度执行的是 CcWorkerThread 函数。在 CcWorkerThread 函数中，两个队列被依次处理，分别是 CcExpressWorkQueue 和 CcRegularWorkQueue，其中 CcExpressWorkQueue 首先被检查。若从队列中取出的项为 ReadAhead 类型，则调用 CcPerformReadAhead 函数执行真正的预读操作，并将 CcScheduleReadAhead 为该项设定的文件对象传递给它。

CcPerformReadAhead 函数的实现相对比较简单，它基本上重用了缓存管理器读取文件数据的功能。它首先从文件对象的私有缓存表中获得预读信息，即 ReadAheadOffset 和 ReadAheadLength 数组成员，然后对于由这些预读信息所定义的文件数据范围，通过调用 CcGetVirtualAddress 映射到系统缓存区域，并逐个页面调用 MmCheckCachedPageState 函数使其换入内存，最后调用 CcFreeVirtualAddress 函数。

如果文件系统驱动程序需要发起预读请求，它们应该使用 CcReadAhead 宏，该宏仅在当前读操作的长度超过 256 字节的情况下才会调用 CcScheduleReadAhead 函数。因为实验表明，对于小的读操作执行预读只会使系统性能退化。

图 7.9 显示了预读处理过程中的关键操作。

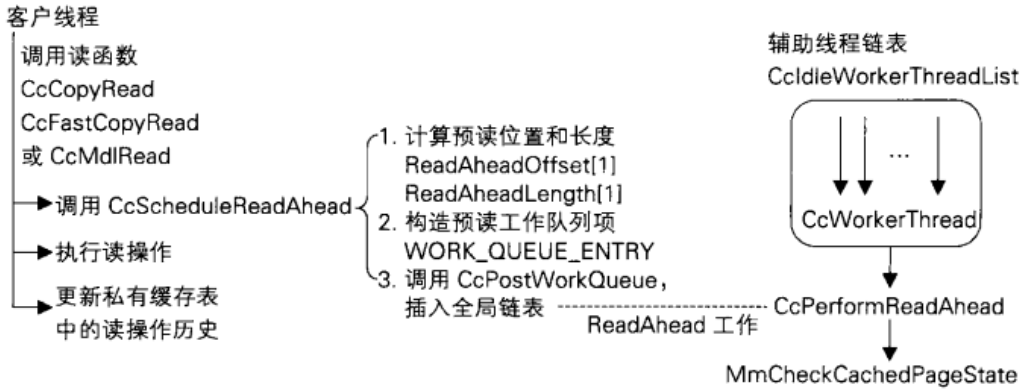


图 7.9 缓存管理器的预读处理

7.2.5 缓存管理器的延迟写

缓存管理器的写操作通常是在后台以异步方式完成的。这意味着被写到文件中的数据首先存放在视图页面中。除非缓存管理器显式地指示要写到磁盘上（譬如等待方式的 CcCopyWrite 调用），否则，这些数据将被统一安排，等累积到一定程度以后才被刷新到存储设备中，这样做的目的是减少硬件 I/O 操作的总量。

缓存管理器的延迟写出器（lazy writer）负责完成这一工作。与预读处理的做法相类似，缓存管理器也有一个安排延迟写操作的函数，名为 CcScheduleLazyWriteScan。当缓存管理器中的其他函数认为有必要安排延迟写操作时，譬如在 CcSetDirtyInMask 和 CcMdlWriteComplete2 函数中，它们会调用 CcScheduleLazyWriteScan 函数来安排延迟的写操作。CcScheduleLazyWriteScan 函数只是设置一个扫描定时器，如果延迟写出器的扫描动作已被激活（LazyWriter.ScanActive 为 TRUE），则扫描定时器的到期时间为 1 s（CcIdleDelay），否则 3 s 以后才激活扫描动作。定时器函数为 LazyWriter.ScanDpc，它被初始化为 CcScanDpc。在 CcScanDpc 函数中，它只是简单地调用 CcPostWorkQueue 函数，在 CcRegularWorkQueue 工作队列链表中插入一个类别为 LazyWriteScan 的工作队列项。

与预读操作类似，缓存管理器的工作队列链表中的队列项是由缓存管理器的辅助线程来执行的，于是又回到了 CcWorkerThread 函数中。在 CcWorkerThread 函数中，类别为 LazyWriteScan 的工作队列项是由 CcLazyWriteScan 函数来处理的，该函数不带任何参数。因此，缓存管理器的写操作函数调用了 CcScheduleLazyWriteScan，实际上是间接地触发了 CcLazyWriteScan 函数被调用。

在 CcLazyWriteScan 函数中，如果当前没有脏页面，也没有其他工作（指延迟的关闭，lazy close），那么，若全局的延迟写链表（即 CcDeferredWrites 链表）为空，则说明没有

客户模块在请求缓存管理器写出页面，因而延迟写出器不必被激活，否则，调用 CcPostDeferredWrites，以允许这些延迟的写请求，然后调用 CcScheduleLazyWriteScan，安排下一次激活扫描操作的定时器。CcLazyWriteScan 函数的代码位于 base\ntos\cache\lazyrite.c 文件的 224~713 行。关于文件系统驱动程序如何请求写出页面以及缓存管理器如何响应这些请求，这是一个协商过程，本节后面将进一步解释。

如果当前脏页面总数（CcTotalDirtyPages 变量）大于 0，那么，CcLazyWriteScan 计划将 1/8 的脏页面安排写出，并根据自上次扫描以来新生成脏页面的数量作调整，如果有必要的话，写出更多脏页面，以便在下一个间隔到来之前累积的脏页面不会超过某一个预设的值（即 CcDirtyPageTarget 全局变量，在 CcInitializeCacheManager 函数中被初始化，其值与物理页面总数有关）。然后从包含脏页面的共享缓存表链表中，逐个检查共享缓存表，看是否有脏页面要写出，以及写出多少脏页面，如此循环，直至扫描一圈。其中 lazyrite.c 文件的 418~663 行代码的 while 循环体现了一次扫描过程中对页面的选择策略。对于每个被选中要写出页面的共享缓存表，构造一个类别为 WriteBehind 的工作队列项，并调用 CcPostWorkQueue 函数插入到 CcExpressWorkQueue（若该共享缓存表正在被清除）或 CcRegularWorkQueue 链表中。

注意，缓存管理器将所有包含脏页面的共享缓存表记录在一个全局链表中，即 CcDirtySharedCacheMapList.SharedCacheMapLinks，并且用一个光标节点 CcLazyWriterCursor.SharedCacheMapLinks 来指示下一次扫描的开始。

在 CcLazyWriteScan 函数中，我们还看到，它在进入扫描循环以前，将全局链表 CcPostTickWorkQueue 中的项转移到了一个局部的链表 PostTickWorkQueue 中，待完成扫描循环以后，再逐项调用 CcPostWorkQueue 函数，将其插入到 CcRegularWorkQueue 队列中。这里，CcPostTickWorkQueue 链表中的项是类别为 EventSet 的工作队列项，可以认为是一种延迟的通知，参见 lazyrite.c 文件中 CcWaitForCurrentLazyWriterActivity 函数的代码。

最后，CcLazyWriteScan 函数在返回以前调用 CcScheduleLazyWriteScan 函数，以便安排下一次扫描操作在 1 s 以后进行，所以，只要缓存管理器有页面要写出，则至少每隔 1 s 就会安排一次扫描。

现在我们再来看类别为 WriteBehind 的工作队列项是如何被处理的。从 lazyrite.c 文件中 CcWorkerThread 函数的代码可以看到，这是由 CcWriteBehind 函数来完成的。CcWriteBehind 调用 CcFlushCache 函数来完成写出页面的任务，而 CcFlushCache 进一步针对文件对象中的连续脏区域，逐个调用 MmFlushSection 函数，将其刷新到存储设备中。

图 7.10 总结了缓存管理器的延迟写出器的工作过程。

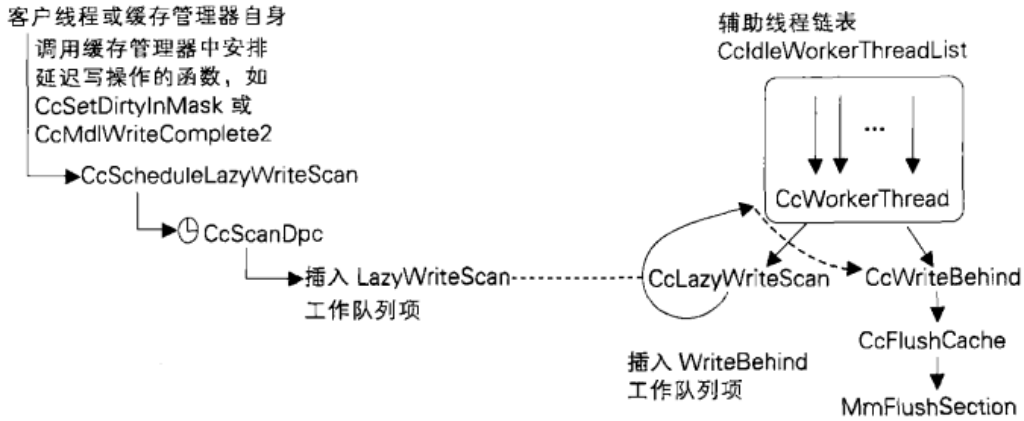


图 7.10 缓存管理器的延迟写处理

文件系统驱动程序为了更好地利用缓存管理器提供的延迟写特性，同时又不影响系统的性能，它们可以在发出写数据请求以前调用 CcCanIWrite 函数以询问缓存管理器，现在立即写出指定数量的字节是否会影响系统性能。如果 CcCanIWrite 返回 TRUE，则文件系统驱动程序可以立即调用像 CcCopyWrite 这样的写数据函数。

CcCanIWrite 函数的代码位于 base\ntos\cache\copysup.c 文件中，它的基本判断依据是，要写出的页面数加上当前的脏页面数（CcTotalDirtyPages 全局变量），是否超过了全局设定的脏页面阈值（CcDirtyPageThreshold 全局变量），以及是否有足够的可用物理内存。当缓存管理器认为立即写出这些数据将会影响系统性能，从而 CcCanIWrite 返回 FALSE 时，调用者接下来应该调用 CcDeferWrite 函数，发出一个回调请求。一旦缓存管理器写出了足够的页面以后，认为可以继续接受写请求，就会调用这一回调函数。

若文件系统驱动程序以阻塞方式来调用 CcCanIWrite 函数，即 Wait 参数为 TRUE，则 CcCanIWrite 会在全局的延迟写链表 CcDeferredWrites 中，插入一个 DEFERRED_WRITE 节点对象，然后调用 CcPostDeferredWrites 函数，并等待该节点对象中指定的事件对象。CcDeferWrite 函数也做了类似的工作，它先构造一个 DEFERRED_WRITE 对象，并填充一个回调函数，即 PostRoutine 域，然后将它插入到 CcDeferredWrites 链表中，再调用 CcPostDeferredWrites 函数。因此，CcDeferWrite 是一个完全异步方式的回调通知，而阻塞方式的 CcCanIWrite 调用则是一个同步等待版本，直至缓存管理器唤醒它可以继续写出页面。现在我们可以明白，为什么 CcLazyWriteScan 函数要在当前无脏页面的情况下检查 CcDeferredWrites 链表是否为空，以确定是否设置延迟写出器为不激活状态。

那么，谁来判断当前延迟写的条件并解除这些等待或发出回调呢？正是 CcPostDeferredWrites 函数。它检查 CcDeferredWrites 链表上的每一个节点，并且以非阻塞方式调用 CcCanIWrite，若满足继续写出页面的条件，则解除正在等待的写线程（调用 KeSetEvent 使 DEFERRED_WRITE 对象中的 Event 对象变成有信号状态），或者调用指定的回调函数（即 DEFERRED_WRITE 对象中的 PostRoutine 成员函数）。缓存管理器在很多地方调用了 CcPostDeferredWrites 函数，只要继续写页面的条件有所改变，就有可能释放一些正在为了写出页面而焦急等待的线程。

延迟写出器提供的这种交互式的流量控制机制，对于避免因为短时间内大量脏页面而造成的系统性能变差，有显著的改进意义，特别是针对慢速存储设备或者被重定向到网络上的存储设备，这一机制可以让整个系统运行得更加平稳。

除了以上介绍的延迟写功能，缓存管理器也支持直写（write-through）方式，只要应用程序在调用 CreateFile 函数来打开文件时指定了 FILE_FLAG_WRITE_THROUGH 标志，那么，缓存管理器的拷贝函数就会试图将要写的页面直接写出到存储设备中，参见 CcCopyWrite 和 CcFastCopyWrite 中关于直写方式的处理。另外，文件系统驱动程序也可以直接调用 CcFlushCache 函数来刷新一个文件的部分或全部。

尽管本章前面我们总是假设缓存管理器的客户是文件系统驱动程序，但实际上，任何访问文件数据流的内核组件都可以是缓存管理器的客户。除了文件系统驱动程序以外，像网络重定向器（network redirector）和网络文件服务器内核组件，以及一些需要使用文件系统接口的驱动程序和内核组件（比如配置管理器），也可以使用缓存管理器提供的接口函数。Windows DDK 中的 ntifs.h 文件包含了缓存管理器提供的接口函数的声明，并且这些接口函数已经被文档化。本书仅仅讨论缓存管理器的实现原理，有关这些接口函数的用法和说明，读者可以参考 Windows DDK 或 MSDN 中的文档。

7.3 Windows 中卷的管理

在现代操作系统中，大容量存储设备几乎都是按照“卷（volume）”来管理的。以磁盘为例，一个物理磁盘上的扇区可以划分成多个分区（partition），每个分区包含一组连续的扇区；而卷负责管理一个或多个分区中的扇区。在 Windows 中，卷有两种：简单卷（simple volume）和多分区卷（multipartition volume）。简单卷对应于一个分区，它管理该分区中的全部扇区或部分连续扇区；而多分区卷管理多个分区，将这些分区中的扇区统一起来管理。因此，卷是扇区的逻辑集合，这些扇区可能位于一个磁盘上，也可能来自多

个磁盘。文件系统是卷内部的一种逻辑结构，允许用户以文件（数据流）方式来访问和组织数据。这一节我们讨论 Windows 中卷的管理，首先看一下 Windows 管理磁盘设备所涉及的驱动程序结构。

7.3.1 Windows 存储栈结构

按照 Windows 的 I/O 模型设计理念，像磁盘这样的存储设备，不会由一个设备驱动程序来承担所有的管理任务。图 7.11 显示了在一个 Windows Server 2003 系统上管理一个基本磁盘所涉及的设备驱动程序：从 I/O 管理器接收到应用程序的 I/O 请求开始，经过三个设备栈到达磁盘的控制器。这些设备栈的驱动程序合起来构成了磁盘设备的存储栈。注意，为说明方便，这里用设备驱动程序的名称来代替它们所创建的设备对象。

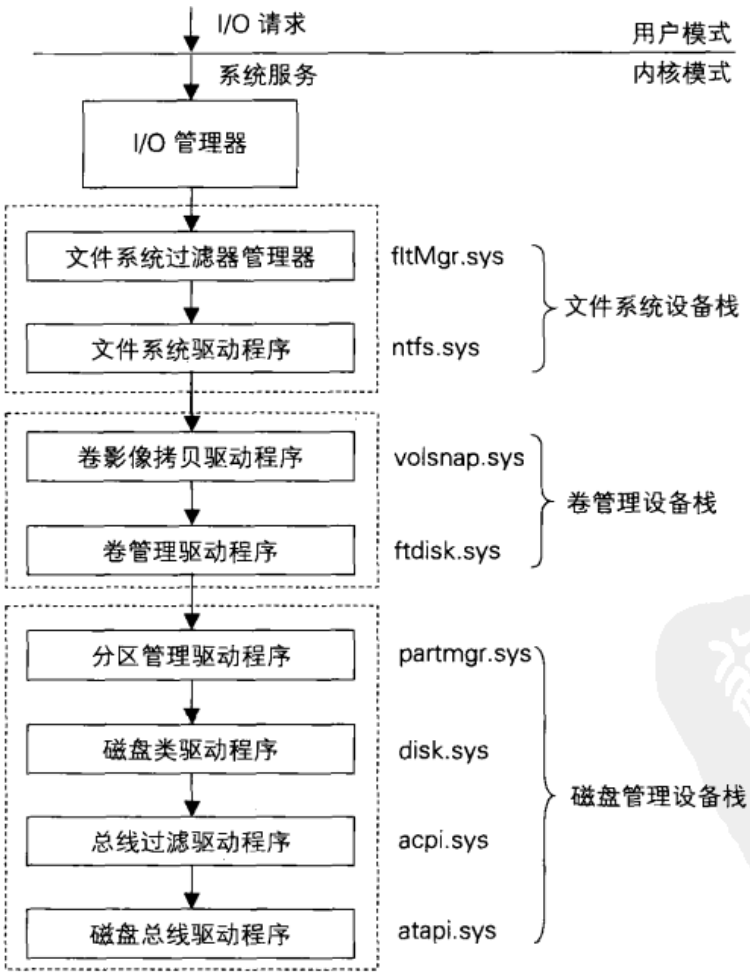


图 7.11 Windows 的存储栈结构

这里的磁盘管理设备栈和卷管理设备栈是在 I/O 系统初始化过程中建立起来的，参见 6.3.3 节关于设备列举和设备树的介绍。Atapi.sys 和 ftdisk.sys 分别是这两个设备栈的总线驱动程序，由即插即用管理器在列举设备过程中构建起相应的设备栈。如果系统只有一个 IDE 硬盘，那么，图中所示的磁盘管理设备栈只有一个，它负责处理发送给该 IDE 硬盘的 I/O 请求；若系统有多个 IDE 硬盘，则系统的设备树中将有多个这样的磁盘管理节点，分别对应于每个 IDE 硬盘。

Windows 对分区和卷的管理有一点复杂。分区和卷是两个相对独立的概念，它们之间并没有必然的对应关系。首先，Windows 支持三种类型的分区方案：MBR（Master Boot Record）分区、GPT（GUID Partition Table）分区，以及 LDM（Logical Disk Manager）分区。使用 MBR 或 GPT 分区方案的磁盘称为基本磁盘（basic disk），而使用 LDM 分区方案的磁盘称为动态磁盘（dynamic disk）。动态磁盘提供了更为灵活的分区方案，允许创建多分区卷，即一个卷包含来自多个不同分区的扇区。即使在基本磁盘上，分区和卷也不是一一对应的。在采用 MBR 分区方案的磁盘上，分区有两种类型：主分区（primary partition）和扩展分区（extended partition）。主分区只允许创建一个卷，扩展分区可以创建多个卷。GPT 磁盘采用了比 MBR 更为灵活的分区方案，它使用可扩展的分区表，并且维护了一个备份分区表。GPT 磁盘除了包含用户分区，还可能有隐藏分区，这些隐藏分区不存储用户数据，仅用于系统引导或系统恢复等管理用途。

在磁盘管理设备栈中，atapi.sys 是 IDE 总线设备驱动程序，负责向即插即用管理器列举所有使用 IDE 接口的设备，包括 IDE 硬盘。Windows 为 SCSI 磁盘使用 scsiport.sys 驱动程序。在图 7.11 中，ACPI 驱动程序在磁盘管理设备栈中作为总线过滤驱动程序而加入进来，其过滤设备对象是由 atapi.sys 指示而创建的。设备栈往上是 disk.sys 驱动程序，即 Windows 提供的磁盘类驱动程序。这是一种典型的类/端口/小端口驱动程序的划分方式，Microsoft 提供了类驱动程序，实现了对于各种磁盘通用的功能，而小端口驱动程序则针对不同类别的磁盘分别被提供。Windows 本身随带了常用的磁盘小端口驱动程序。

磁盘管理设备栈最上边是分区管理驱动程序，它负责通知即插即用管理器当前磁盘上有哪些分区，因而，卷管理驱动程序可以接收到有关分区创建和删除的通知。分区管理驱动程序调用 IoReadPartitionTable 或 IoReadPartitionTableEx 函数来获得磁盘的分区信息。另外，由于它位于设备栈的顶端，总是可以看到任何与修改分区表有关的 I/O 请求，所以，它可以将有关分区的变化通知即插即用管理器，即插即用管理器可以进一步指示卷管理驱动程序做相应的处理。IoReadPartitionTable 是一个遗留函数，在 Windows XP/Server 2003 以后，磁盘驱动程序应使用 IoReadPartitionTableEx 函数，其原型如下：

```
NTSTATUS
IoReadPartitionTableEx(
    IN PDEVICE_OBJECT DeviceObject,
    IN struct _DRIVE_LAYOUT_INFORMATION_EX** DriveLayout
    );
```

WRK 中不包含 IoReadPartitionTableEx 函数的代码，但是其实现原理并不难理解，它通过向磁盘设备对象 DeviceObject 发送读取扇区的 I/O 命令，并根据每一种分区方案的特点，读取到相应的分区表，然后分配并填充 DriverLayout 对象的信息。对于 MBR 风格的磁盘，读取分区表实际上是由 IoReadPartitionTable 函数来完成的，该函数的代码位于 base\ntos\fstub\drivesup.c 文件的 2 134~2 907 行。从代码可以看出，IoReadPartitionTable 函数向磁盘设备对象发送了一个 IRP_MJ_READ 请求，读取 MBR 扇区（偏移为 0，长度为 512 字节），然后解析其内容，并构造一个 DRIVE_LAYOUT_INFORMATION 对象。

为了说明如何建立起卷管理设备栈，我们首先看一下磁盘管理设备栈和卷管理设备栈在设备树中的位置，如图 7.12 所示。卷管理设备栈也是在即插即用管理器列举设备过程中建立起来的，或者当磁盘分区发生变化时，由分区管理驱动程序指示卷管理驱动程序创建相应的卷设备对象。

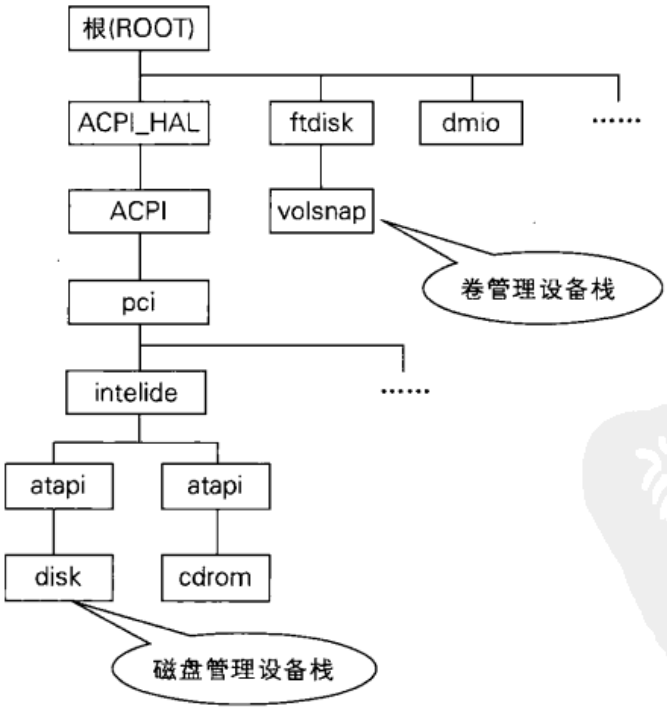


图 7.12 磁盘管理设备栈和卷管理设备栈在设备树中的位置

当系统引导时，分区管理驱动程序在初始化例程中，首先调用 IoAllocateDriver-

ObjectExtension 函数分配一个扩展区对象，这是在分区管理驱动程序中全局共享的一个对象；然后，它利用注册表的信息以及从磁盘引导扇区获得的信息来初始化扩展对象中的成员。然后，分区管理驱动程序调用 IoRegisterPlugPlayNotification 函数，向即插即用管理器注册一个针对卷管理器设备接口（VOLMGR_VOLUME_MANAGER_GUID）的加入或移除通知回调（EventCategoryDeviceInterfaceChange）。因此，分区管理驱动程序可以获得卷管理设备对象的实例变化通知。在 Windows Server 2003 系统上，基本磁盘的卷管理驱动程序为 ftdisk.sys（在 Windows Vista 下为 volmgr.sys）。分区管理驱动程序通过这种方法获得关于卷管理驱动程序的知识，实际上，它得到了卷管理驱动程序对象和设备对象（作为 FDO，即图 7.12 中 ftdisk 节点上的设备对象）的引用。

Ftdisk 是一个总线驱动程序，它在初始化例程中调用 IoRegisterDeviceInterface 函数来注册卷管理器设备接口，此设备接口与分区管理驱动程序在调用 IoRegisterPlugPlayNotification 来注册回调通知时指定的设备接口相同，因而分区管理驱动程序可以引用到 ftdisk 驱动程序。在即插即用管理器列举设备过程中，当分区管理驱动程序接收到 IRP_MN_QUERY_DEVICE_RELATIONS 子命令时，它检查所有的分区，并通过一个 IRP_MJ_INTERNAL_DEVICE_CONTROL 命令，指示相应的卷管理驱动程序为分区创建卷对象。这样可建立起磁盘存储栈结构中的卷管理设备栈。在此过程中，卷管理驱动程序和分区管理驱动程序相互获得对对方设备对象的引用，从而可以方便地进行通信。

Volsnap.sys 是卷管理设备栈中的功能驱动程序，它为卷提供了附加的功能，即基于软件的卷快照功能。该驱动程序位于文件系统和卷管理驱动程序之间，因而可以看到所有对于卷的 I/O 请求。当需要建立一个快照，比如备份工具启动一个备份操作时，volsnap 驱动程序为目标卷创建一个影像拷贝，于是，备份工具访问快照建立时的卷内容，而应用程序仍然可以修改和访问卷上的内容。Volsnap.sys 很好地满足了系统实时备份的需求。

有一点需要说明，虽然存储栈中的驱动程序位于不同的设备栈中，但是它们在处理 I/O 请求时仍然遵循“由上向下传递 IRP，由下往上执行完成处理”的规则。卷管理驱动程序（ftdisk.sys）根据需要，将 I/O 请求的 IRP 转发给分区管理驱动程序（partmgr.sys）。

与 ftdisk.sys 负责管理基本磁盘的卷相类似，在 Windows Server 2003 系统中，dmio.sys 驱动程序负责管理动态磁盘的卷（在 Windows Vista 系统中为 volmgrx.sys），它支持多分区卷。在多分区卷中，构成该卷的多个分区可以是同一个磁盘上不连续的分区，甚至可以来自不同的磁盘。不同类型的多分区卷有不同的用途，比如，有的卷可以提供更大的存储空间，有的卷则可以通过冗余的数据存储达到更高的可靠性和容错能力。在 Windows 中，典型的多分区卷包括跨距卷（spanned volume）、镜像卷（mirrored volume）、条带卷（striped

volume）和 RAID-5 卷。Dmio 驱动程序在动态磁盘中的角色如同 ftdisk 在基本磁盘中的角色一样，但它的管理工作相对要复杂很多。关于 dmio 驱动程序和多分区卷的工作原理，请参考相关书籍[WIN-INTERNALS]。

7.3.2 卷的挂载

为了允许应用程序和内核其他模块访问卷以及卷内部的文件，Windows 利用名字空间和文件对象两种抽象机制，提供了灵活的访问方法。首先，Windows 内部使用一个全局统一的名字空间，即 3.4.1 节讨论过的以 ObpRootDirectoryObject 为根的全局名字空间。其次，名字空间是一个树状结构，对象管理器在解析一个名称时，逐层递进，或者在对象管理器的目录中逐层匹配，或者碰到具有 Parse 方法的对象，然后将余下的尚未解析的名称串交给该对象以进一步解析。卷对象的命名和挂载正是充分利用了对象管理器的这一名字空间管理机制，从而为访问卷和卷内部的数据提供了极大的便利。

尽管 Windows 内部的对象管理使用了一个全局的名字空间，但它仍然兼容从 DOS 时代保留下来的以驱动器字母方式来访问卷和文件系统的习惯，诸如硬盘、CD-ROM、软盘驱动器和可移除的存储设备等，都可以被分配一个驱动器字母（譬如 C:、D:、A:等）。Windows 将卷的字母分配信息保存在注册表中（HKLM\SYSTEM\MountedDevices 键）。Windows 挂载管理器（mountmgr.sys 驱动程序）负责为卷分配驱动器字母，它在初始化例程中注册了设备接口（MOUNTDEV_MOUNTED_DEVICE_GUID）的变化通知，因而，每当卷管理器（ftdisk 或 dmio）创建一个卷时，挂载管理器总是会接收到通知，从而可为该卷分配一个驱动器字母，至于它如何选择驱动器字母，则取决于它的内部数据库和卷管理器的建议。

在 Windows Server 2003 系统中，卷管理驱动程序 ftdisk 在为基本磁盘的每一个卷创建设备对象时，指定的对象名称为“\Device\HarddiskVolume*X*”，这里 *X* 是一个标识该卷的数字，从 1 开始。譬如，如果一个系统有一块硬盘，该硬盘被分成两个分区，则这两个分区的卷设备对象分别为“\Device\HarddiskVolume1”和“\Device\HarddiskVolume2”。假设这两个卷的驱动器字母分别是 C:和 D:，那么，挂载管理器也为每个卷设备对象创建一个符号链接，在这个例子中，符号链接为“\DosDevices\C:”和“\DosDevices\D:”，分别指向“\Device\HarddiskVolume1”和“\Device\HarddiskVolume2”。这里“\DosDevices”也是一个符号链接，指向“\??”目录。

为便于理解，这里介绍一下对象管理器解析“\DosDevices”和“\??”的过程。Windows 系统支持多个会话，每个会话都可以有一个局部名字空间（local namespace），它是全局名字空间（global namespace）的一个私有视图。局部名字空间包括“\DosDevices”、“\Windows”

和“\BaseNamedObjects”三个目录。在全局名字空间中，“\Sessions\X”是会话 X 的一个目录，其中包含了与用户会话相关联的一些全局目录，这里的 X 是会话标识符，比如 0、1 等。Windows 的系统 DLL 在接收到应用程序传递过来的应该要在“\DosDevices”目录中解析的对象引用时，会在其名称前面加上“\??\”，比如，应用程序访问 C 盘的根目录，即“C:\”，则系统 DLL 将其转变成“\??\C:\”。

对象管理器在解析名称时，一旦看到“\??\”前缀，则使用进程对象 EPROCESS 中的 DeviceMap 域，其中 DeviceMap 的 DosDeviceDirectory 成员指向该进程局部名字空间的“\DosDevices”目录。于是，对象管理器将“\??\”转换成此 DosDeviceDirectory 成员所指的目录。若对象管理器在此目录中无法解析这一名称，则检查 DosDeviceDirectory 目录对象的 DeviceMap 成员是否有效，若有效，则在该 DeviceMap 对象的 GlobalDosDevicesDirectory 成员所指的目录中继续查找目标对象，而 DeviceMap 对象的 GlobalDosDevicesDirectory 成员总是指向全局名字空间的“\GLOBAL??”目录，所以，当对象管理器遇到“\??\”时，会首先尝试局部名字空间中的“\DosDevices”目录，然后尝试全局名字空间中的“\GLOBAL??”目录。注意，进程 EPROCESS 对象的 DeviceMap 域指向会话范围内共享的数据结构。因此，对象管理器的这一名称解析过程使得系统的会话（譬如终端服务）有机会定义它们自己的设备映射表。关于对象管理器的名称解析过程，请参考 base\ntos\ob\obdir.c 文件中的 ObpLookupObjectName 函数的代码，其中包括对于“\??\”前缀的处理。

如果运行在远程桌面中的应用程序想要访问全局名字空间中的对象，比如控制台会话中创建的对象，那么，它可以在名称前面加上“\Global”前缀。例如，运行在远程桌面中的应用程序要访问全局名字空间中的“\\.\MyObject”对象，则只需使用“\\.\Global\MyObject”即可。对象管理器将会绕过远程桌面会话的局部名字空间中的“MyObject”对象。对象管理器如何做到这一点呢？很简单，它只是在初始化时，在“\GLOBAL??”目录中创建了一个符号链接“Global”，让其指向“\GLOBAL??”目录自身。想一想为什么。

因此，挂载管理器创建的卷设备对象，譬如“\DosDevices\C:”和“\DosDevices\D:”，实际上被解析为“\GLOBAL??\C:”和“\GLOBAL??\D:”，它们分别指向“\Device\HarddiskVolume1”和“\Device\HarddiskVolume2”。图 7.13 显示了全局名字空间中与卷有关的目录、链接以及设备对象。其中虚线表示符号链接关系；“\??”并非对象，而是由对象管理器特殊处理的前缀字符串。

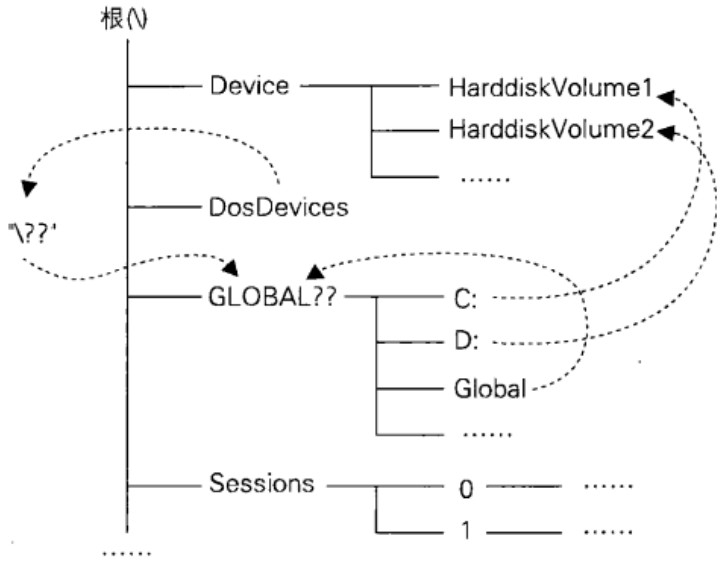


图 7.13 与卷设备对象有关的符号链接

Windows 操作系统也允许一个卷被挂载到另一个 NTFS 卷的目录上，从而将多个卷通过同一个文件系统名字空间链接起来。譬如，可以将一个包含有关工作项目数据的卷挂载到 C:\Work 目录；而将一个包含个人信息的卷挂载到 C:\MyData 目录。这样，三个卷将使用同一个名字空间，而且，工作卷和个人卷无须使用驱动器字母，就可以被应用程序访问。例如，工作卷上的文件\Projects\PrjList.txt，可通过 C:\Work\Projects\PrjList.txt 路径来访问。在三个卷中，系统卷 C:必须是 NTFS 文件系统，而工作卷和个人卷可以是 NTFS 文件系统，也可以是其他的文件系统，比如 FAT。关于文件系统的进一步介绍，参见 7.4 节。

将一个卷挂载到 NTFS 文件系统的目录上，此目录即被称为一个挂载点，挂载点是通过一项被称为重解析点（reparse point）的技术来实现的。为了实现挂载点，需要 NTFS 文件系统、I/O 管理器以及对象管理器相互协作。当对象管理器在解析名称过程中碰到一个设备对象时，它调用 IopParseDevice 函数来解析剩下的名称字符串。IopParseDevice 函数调用设备对象所在驱动程序的分发例程来打开一个文件对象（IRP_MJ_CREATE 命令），如果该分发例程返回 STATUS_REPARSE，则表明碰到了一个重解析点，因而对象管理器要用新的名称字符串重新进行解析。这样就从文件系统的一个目录转移到了另一个卷上。IopParseDevice 函数的代码位于 base\ntos\io\iomgr\parse.c 文件中。

在重解析点上，有一个数据结构 REPARSE_DATA_BUFFER 与 NTFS 文件或目录关联在一起，因此，当文件系统驱动程序解析名称到达该点上时，它可以告诉 I/O 管理器，该如何构造新的名称，读者可以参考 IopDoNameTransmogrify 函数的代码。除了 NTFS 文件系统可以定义重解析点以外，文件系统过滤驱动程序也可以定义重解析点，

它们可以按照自己的逻辑来转移或修改一个目录的路径。就本质而言，重解析点是一个可以由设备驱动程序自己来定义的符号链接。卷的挂载点只是重解析点技术的一个应用而已，它把卷的名称当做了重解析点的数据。Windows 自带的命令行工具 mountvol.exe（位于\Windows\System32 目录下）可用于创建或显示挂载点。

在 NTFS 卷上，挂载管理器在文件系统的元数据中记录了所有针对该卷而定义的挂载点，这样，当一个磁盘从一个系统移动到另一个系统中时，挂载点也会跟着移动。而且，NTFS 文件系统也记录了它自己的挂载点，从而当应用程序、管理工具或者系统需要列举一个卷的挂载点时，可以很快获得这些挂载点。

7.3.3 卷与文件系统

虽然上一小节的介绍隐含了一个卷上要有文件系统，该卷上的数据才可以被访问，但是，我们尚未解释卷设备对象如何与文件系统挂起钩来。本节将讨论文件系统是如何被应用到一个卷上的。

给卷分配一个驱动器字母并不等于该卷的数据有了文件格式，或者说卷有了对应的文件系统。卷上的数据是以扇区来管理的，而文件系统则为扇区中的数据定义了结构信息。因此，为了能以文件和目录的方式来访问一个卷上的数据，该卷必须被关联上一个文件系统，此过程称为卷识别（volume recognition）。

首先，所有的文件系统驱动程序在初始化过程中，都已经向 I/O 管理器注册过了。IoRegisterFileSystem 函数是注册函数，其代码位于 base\ntos\io\iomgr\iosubs.c 文件中，它根据文件系统设备对象的设备类型（DeviceType），将其插入到相应的文件系统链表中。Windows 有 4 个文件系统链表：IopNetworkFileSystemQueueHead、IopCdRomFileSystemQueueHead、IopDiskFileSystemQueueHead 和 IopTapeFileSystemQueueHead，分别对应于网络文件系统、CD-ROM 文件系统、磁盘文件系统和磁带文件系统。然后 IoRegisterFileSystem 通知那些已经注册过要接收文件系统变化通知（通过 IoRegisterFsRegistrationChange 函数）的客户。因此，I/O 管理器知道所有的文件系统驱动程序。

其次，挂载管理器并不在挂载一个卷的时候立即执行卷识别过程，而是会延迟到当第一次有客户代码（内核或应用程序）要访问该卷上的文件或目录的时候。譬如，对于系统卷，在引导时，当系统的初始化代码需要访问磁盘上的文件和目录时，一旦 IopParseDevice 解析到系统卷的路径名，就会调用 IopCheckVpbMounted 函数，进而调用 IopMountVolume 函数执行卷识别过程。

IopMountVolume 函数的代码位于 base\ntos\io\iomgr\internal.c 文件的 4 683~5 160 行，它首先根据设备对象的 DeviceType 定位到相应的文件系统链表，比如 IopDiskFileSystemQueueHead，然后通过一个 for 循环，向链表中的每一个文件系统对象发送一个命令码为 IRP_MJ_FILE_SYSTEM_CONTROL 的 I/O 请求，其子命令码为 IRP_MN_MOUNT_VOLUME，表明这是一个挂载请求。真正的识别过程由文件系统驱动程序来完成，如果此 I/O 请求返回成功，那么，IopMountVolume 调用 IopMountInitializeVpb 函数进一步初始化卷设备对象中的 VPB（Volume Parameter Block）对象。在 IopMountVolume 函数中，有以下三点值得注意：

- 如果在卷识别循环中，有文件系统驱动程序执行了文件系统注册（IoRegisterFileSystem）或注销（IoUnregisterFileSystem）的操作，则卷识别循环重新开始。
- Windows 对于文件系统驱动程序的加载做了优化，它并不是直接让每个文件系统驱动程序都加载进来，而是使用一个名为文件系统识别器（File System Recognizer）的代理驱动程序（fs_rec.sys），来执行初步的文件系统识别请求。文件系统识别器本身也被注册为一个文件系统对象。在 IopMountVolume 函数中，我们可以看到，在卷识别过程中，如果 I/O 请求的结果是 STATUS_FS_DRIVER_REQUIRED，则 IopMountVolume 调用 IopLoadFileSystemDriver 函数加载所要求的驱动程序。然后重新尝试卷识别循环。
- Windows 内核有一个内置的文件系统驱动程序“\FileSystem\RAW”，这也是第一个注册的文件系统，它被分别挂载到三个全局链表中。IopMountVolume 的参数 AllowRawMount 指明了是否允许挂载为 RAW 文件系统。由于 RAW 驱动程序并不执行任何名称解析的工作，所以，Windows 内核或应用程序不可能通过挂载 RAW 文件系统来访问卷上的文件或目录。关于 RAW 文件系统，参见 7.4.2 节。

文件系统驱动程序识别一个卷的依据是，检查该卷的引导扇区，看是否符合该文件系统所要求的必要条件，包括一些标识字符串和关键的结构信息。譬如，NTFS 文件系统驱动程序要求引导记录中包含名称域“NTFS”，FAT 文件系统驱动程序要求引导扇区的前三个字节必须是一条跳转指令，可以是“eb 58 90”（实际上是两条指令，90 是 nop 指令）或“e9 57 00”。通常文件系统驱动程序会执行尽可能严格的验证工作，以完成精确的识别。但是，RAW 文件系统不需要执行任何格式识别，参见它的 IRP_MN_MOUNT_VOLUME 命令处理函数 RawMountVolume。

文件系统驱动程序在识别出一个卷以后，会构造一个文件系统设备对象，以及一个

VCB（Volume Control Block）结构。此文件系统设备对象代表了这个卷的文件系统实例；VCB 结构由每个文件系统驱动程序自己定义，通常是设备对象扩展区的一部分。文件系统驱动程序在内部通过 VCB 与卷关联起来。

另外，一旦一个卷被文件系统驱动程序成功地识别出来，那么，代表该卷的文件系统设备对象实例就会包含已被正确初始化的 VPB 对象。VPB 对象是连接卷对象与文件系统对象的桥梁，只要在调用 IoCreateDevice 函数时指定了 FILE_DEVICE_DISK、FILE_DEVICE_TAPE、FILE_DEVICE_CD_ROM 或 FILE_DEVICE_VIRTUAL_DISK 设备类型，则 IoCreateDevice 就会调用 IopCreateVpb 函数创建一个 VPB 对象。VPB 结构的定义如下：

```
typedef struct _VPB {
    CSHORT Type;                                  // 类型域，为IO_TYPE_VPB
    CSHORT Size;                                  // VPB对象的大小
    USHORT Flags;                                 // 有关卷的标志信息
    USHORT VolumeLabelLength;                     // 卷标签的长度
    struct _DEVICE_OBJECT *DeviceObject;          // 指向文件系统设备对象
    struct _DEVICE_OBJECT *RealDevice;            // 指向卷设备对象
    ULONG SerialNumber;                           // 卷序列号
    ULONG ReferenceCount;                         // 引用计数
    WCHAR VolumeLabel[MAXIMUM_VOLUME_LABEL_LENGTH / sizeof(WCHAR)];
} VPB, *PVPB;
```

VPB 对象中的域，有些由 I/O 管理器来初始化，有些则由文件系统驱动程序来初始化。RealDevice 域指向卷设备对象，它是在 IoCreateDevice 调用 IopCreateVpb 函数时初始设置的；DeviceObject 域指向文件系统设备对象。当文件系统驱动程序接收到 IRP_MN_MOUNT_VOLUME 请求时，它会在验证了卷格式以后，创建文件系统设备对象，并且利用 IRP 参数对象附带的 IO_STACK_LOCATION 对象的 Parameters.MountVolume.Vpb->DeviceObject 成员，将文件系统设备对象存放到卷设备对象的 VPB 对象中。因此，当卷设备对象被成功挂载以后，它的 VPB 对象中的 DeviceObject 域指向文件系统设备对象。反过来，文件系统设备对象通常也会在它自己的 VCB 中记录下卷设备对象，I/O 管理器在挂载卷设备对象时，利用 IO_STACK_LOCATION 对象的 Parameters.MountVolume.DeviceObject 成员，将卷设备对象（实际上是卷设备栈顶部的设备对象）传递给文件系统驱动程序。关于这一挂载过程，读者可以参考 WRK 中的 IopMountVolume 和 RawMountVolume 函数，其中 RawMountVolume 是 Windows 内核内置的 RAW 文件系统处理 IRP_MN_MOUNT_VOLUME 请求的函数。

因此，卷设备对象与文件系统设备对象之间是通过 VPB 和 VCB 关联起来的，这里 VPB 是 I/O 管理器定义的数据结构，而 VCB 则是文件系统驱动程序内部定义的数据结构。一旦一个卷已被挂载，则其 VPB 的 Flags 域将包含 VPB_MOUNTED 标志。对象管理器在解析到卷设备对象时，如果碰到已被挂载的卷，则根据 VPB 中的 DeviceObject 域，转交给文件系统设备对象，以进一步解析剩余的名称字符串，从而深入到文件系统名字空间中。

了解了卷设备对象与文件系统设备对象之间的关系以后，我们再来看一看一个文件系统中的文件对象是如何被构造的。参考 6.2.3 节的介绍，如图 6.7 所示，创建文件对象需从 IopCreateFile，到 ObOpenObjectByName、ObpLookupObjectName，再经过一个循环，通过 ObpLookupDirectoryEntry 函数深入到对象管理器名字空间中的设备对象，最后调用 IopParseDevice 函数，完成对象创建过程。如果要创建一个卷设备中的文件对象，那么，在定位到卷设备对象（譬如“\Device\HarddiskVolume1”）以后，IopParseDevice 函数检查它已被挂载，于是转移到它的 VPB 对象的 DeviceObject 上，即文件系统设备对象，交给文件系统驱动程序来处理 IRP_MJ_CREATE 请求。IopParseDevice 函数调用 ObCreateObject 来构造一个 FILE_OBJECT 对象，并通过 IRP 的 IO_STACK_LOCATION 的 FileObject 成员将其传递给文件系统驱动程序，所以，文件系统驱动程序可以直接使用此 FILE_OBJECT 对象。文件系统驱动程序根据它的内部管理和命名规则，填充此文件对象信息，尤其是，设置其中的 Vpb 域，使之指向当前卷的 VPB 对象。因此，文件系统驱动程序从任何一个文件对象，都可以立即访问到卷设备对象。正如上一小节所讲，如果文件系统（或者文件系统的过滤驱动程序）支持重解析点，那么，IopParseDevice 发出的 I/O 请求可能会得到 STATUS_REPARSE 的结果值，在这种情况下，IopParseDevice 返回到 ObpLookupObjectName 函数，由它继续解析该重解析点所指示的名称。

7.3.4 文件对象的 I/O 处理

至此，我们已经分析了 Windows 中大容量存储设备的存储栈的组成部分，以及各部分之间如何建立起联系。一旦存储栈建立起来，I/O 管理器就可以沿着“文件系统设备栈→卷管理设备栈→磁盘管理设备栈”的路径处理 I/O 读写请求。图 7.14(a)和(b)分别显示了 IRP_MJ_CREATE 与 IRP_MJ_READ/IRP_MJ_WRITE 请求的处理路径。

图 7.14(a)显示了在磁盘上创建一个文件的流程，应用程序通过 NtCreateFile 系统服务，将创建文件的请求传递给 I/O 管理器，又经过对象管理器，到达 IopParseDevice 函数（参见 6.2.3 节），由它继续解析剩余的名称字符串“\Hello.txt”。在 IopParseDevice 函数中，它接收到的目标设备对象为卷设备对象“\Device\HarddiskVolume1”，它根据卷设备对象中的 VPB，得到文件系统设备对象的指针，并调用 IoGetAttachedDevice 函数，进一步得到文件系统设备栈的栈顶对象（譬如，fltMgr 设备对象），然后发出一个命令码为 IRP_MJ_CREATE 的 I/O 请求。

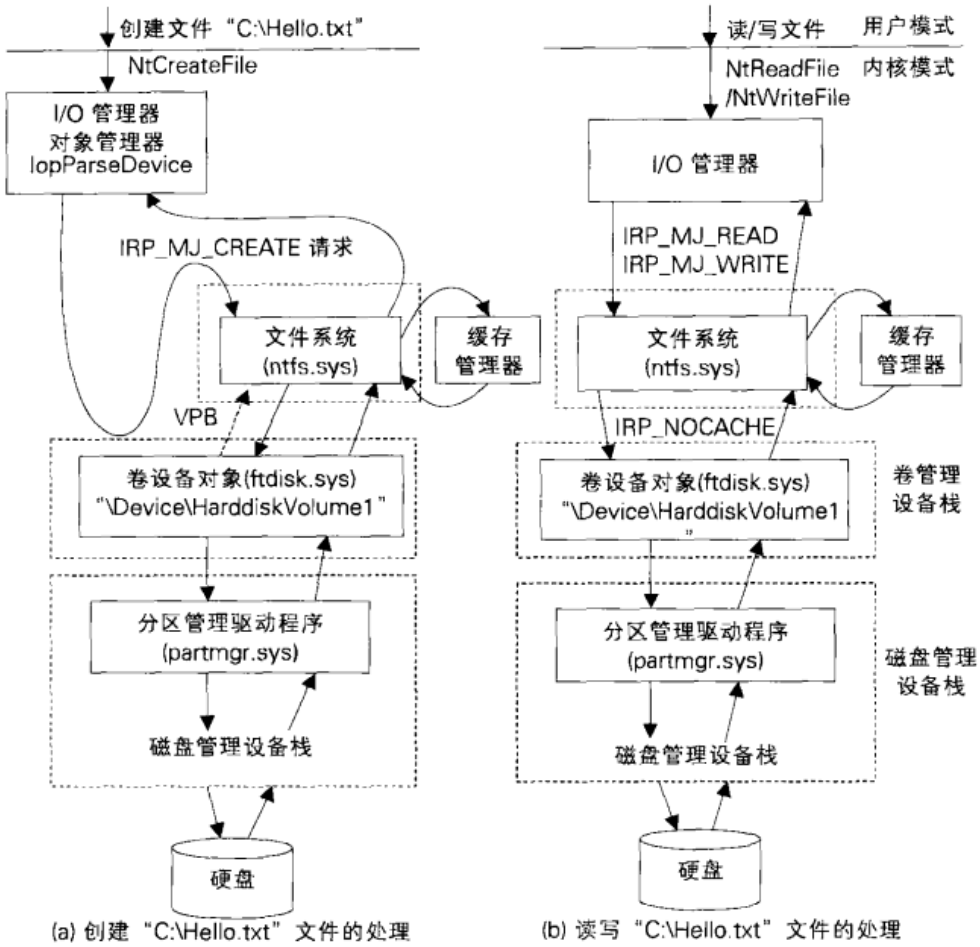


图 7.14 针对磁盘文件的 I/O 请求路径

文件系统驱动程序在接收到此创建文件对象的请求以后，根据文件系统的格式，以及文件系统元数据在缓存中的情况，在根目录中创建 Hello.txt 文件，若有必要，则通过 CcFlushCache 函数将目录信息刷新到磁盘上。在这种情况下，缓存管理器又会通过 I/O 管理器，发送一个带有 IRP_NOCACHE 标志的 IRP_MJ_WRITE 命令，文件系统驱动程序接收到此 I/O 命令，直接沿着卷管理设备栈和磁盘管理设备栈向下传递，直至将数据写到磁盘上，并完成此 I/O 请求。关于缓存管理器与文件系统的关系，参见 7.2.2 节。

图 7.14(b)读写文件的 I/O 请求的处理路径与创建文件请求的路径相似，只不过，由于 NtReadFile/NtWriteFile 接收目标文件句柄作为其参数，因而它们可以直接获得文件对象，并通过文件对象的 VPB 定位到文件系统设备对象，从而将 IRP_MJ_READ 或 IRP_MJ_WRITE 命令发送给它。文件系统驱动程序在接收到此 I/O 请求以后，根据文件

对象缓存的状态，要么在内存中满足此 I/O 请求，要么发送包含 IRP_NOCACHE 标志的 I/O 请求，以便真正从磁盘读数据或写数据到磁盘上。图 7.15 显示了 Windows Server 2003 SP1 标准配置环境中 I/O 读写操作的调用栈，从中可看出整个存储栈中驱动程序被调用的顺序。

(a) IoPageRead 读磁盘 I/O 操作　　(b) CcFlushCache 写磁盘 I/O 操作

图 7.15 读写磁盘的 I/O 操作的调用栈

最后，简单介绍一下在存储栈中出现的驱动程序 fltMgr。FltMgr 是文件系统过滤管理器，它提供了一个管理框架，以方便开发人员编写文件系统过滤驱动程序。FltMgr 自身附载到已挂载的卷上，所以，它会出现在文件系统的设备栈中，就像前面例子中所看到的那样。FltMgr 负责管理一个或多个小过滤驱动程序（minifilter driver），包括控制它们的加载顺序、提供有选择的过滤能力、缩短内核栈的深度要求等，关于小过滤驱动程序的开发，Windows DDK 提供了完整的文档以及一组例子驱动程序（位于 src\filesys\miniFilter 目录下）。图 7.16 显示了 fltMgr 驱动程序的工作模型，其中两个小过滤驱动程序分别负责病毒检测和文件复制服务。关于文件系统过滤的全面讨论，参见 7.4.3 节。

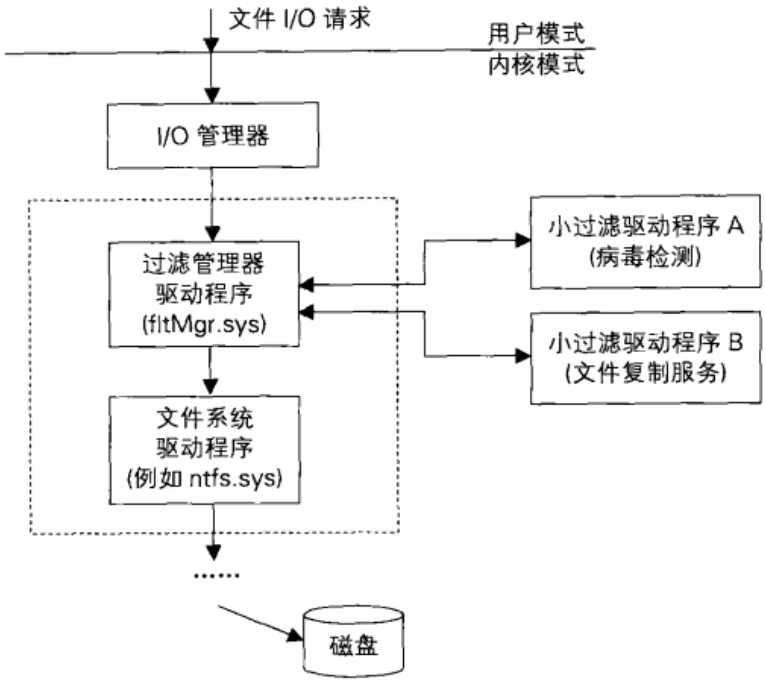


图 7.16 文件系统过滤管理器与小过滤驱动程序

7.4 Windows 文件系统

在现代操作系统中，文件系统是 I/O 的一个重要方面，它为存储设备提供了流方式的数据管理，允许应用程序共享卷的存储空间，同时又可以独享不同的数据流。在 Windows 中，文件系统提供了一个层次状的名字空间，并集成到了系统的全局名字空间中；文件对象既是设备对象的已打开实例的抽象，也代表了文件系统中一个已打开的数据流实例。Windows 的文件系统以卷设备对象为存储基础，不同种类的文件系统定义了不同的存储格式。文件系统格式的描述能力也决定了它的数据管理能力，比如文件的访问权限管理、是否支持压缩、单个文件的最大尺寸，等等。

正如上一节所讨论的，文件系统在 Windows 的存储管理结构中只是一个环节，其驱动程序位于存储栈的上部，它们通常依赖于缓存管理器来实现文件数据的缓存。这一节我们首先讨论文件系统驱动程序的结构，接着介绍 Windows 内核中实现的 RAW 文件系统，以及内核对于文件系统的支持库 FsRtl。然后介绍 Windows 提供的文件系统 I/O 过滤机制，最后分别讲述 FAT 和 NTFS 文件系统。

7.4.1 文件系统驱动程序结构

Windows 支持的原生文件系统是 NTFS（NT File System），其驱动程序为 ntfs.sys；另一种常用的文件系统是 FAT（File Allocation Table），这是从 DOS 时代遗留下来的，它的格式相对比较简单，目前仍被广泛用于移动介质，以方便数据交换。除了 NTFS 和 FAT 文件系统，Microsoft 还为 Windows 提供了其他一些文件系统驱动程序，包括 CDFS 和 UDFS。CDFS 即 CD-ROM 文件系统，这是一种只读文件系统，其驱动程序为 cdfs.sys；UDFS 即 UDF 文件系统，主要针对 DVD-ROM，它比 CDFS 更灵活，其驱动程序为 udfs.sys。在 Windows 的全局名字空间中，文件系统驱动程序对象被指定存放在“\FileSystem”目录下，例如，“\FileSystem\ntfs”和“\FileSystem\cdfs”等。当 Windows 加载这些文件系统驱动程序时，如果看到在驱动程序的注册表键（HKLM\SYSTEM\CurrentControlSet\Services\<*DriverName*>）中，Type 值为 2（宏 SERVICE_FILE_SYSTEM_DRIVER）或 8（宏 SERVICE_RECOGNIZER_DRIVER），那么，IopGetDriverNameFromKeyNode 函数将返回以“\FileSystem”为前缀的名称，否则返回以“\Driver”为前缀的名称。IopLoadDriver 函数调用 IopGetDriverNameFromKeyNode 来决定驱动程序对象的名称。然而，驱动程序也可以通过其注册表键下的“ObjectName”值来指定驱动程序对象的名称。读者可以参考 IopGetDriverNameFromKeyNode 函数的代码。第 6 章的表 6.2 列出了在 Windows Server 2003 SP1 虚拟机环境中“\FileSystem”目录下的驱动程序对象。我们可以看到，FltMgr（文件系统过滤管理器）、Fs_Rec（文件系统识别器驱动程序）、RAW 文件系统驱动程序，以及其他一些用于远程文件访问的驱动程序也位于此目录下。

文件系统驱动程序完全符合 6.5 节介绍的驱动程序结构模型，但是，在其功能和设计方法上，它们有诸多独特的地方，以至于被列为 Windows 设备驱动程序的一大类。而且，Windows DDK 为文件系统驱动程序提供了专门的开发支持，称为 IFS（Installable File System）。Windows DDK 提供了 CDFS 和 FAT 文件系统的源代码，读者可以编译这两个驱动程序（分别为 cdfs.sys 和 fastfat.sys 文件），然后替代 Windows 系统中自带的同名驱动程序。通过这种方法，可以跟踪和观察 Windows 文件系统的运行，并且，这种做法不依赖于 WRK，可以在 DDK 支持的任何 Windows 版本中使用。

下面我们来看文件系统驱动程序应该有哪些特点。首先从驱动程序的结构来看，文件系统驱动程序除了提供标准的分发例程，更倾向于提供快速 I/O 入口，因为经过快速 I/O 入口可避免“构造 IRP”的开销；其次，文件系统驱动程序通常使用缓存管理器来提供文件一级的缓存能力，虽然这不是必需的，但是缓存管理器本质上为文件系统提供了一套公共的缓存机制。只要文件系统驱动程序恰当地使用缓存管理器的缓存功能，就可以以相对

较小的代价获得性能、内存使用效率等方面的优势。文件系统驱动程序除了为每个打开的文件使用缓存管理器的缓存能力，还可以利用缓存管理器的 CcMapData、CcPinRead、CcPreparePinWrite 等接口函数，将卷上的元数据映射或/和锁定（pin）到系统地址空间，然后方便地操纵这些元数据。图 7.17 显示了一个文件系统驱动程序的结构，以及它与 I/O 管理器和缓存管理器之间的紧密关系。文件系统驱动程序不是即插即用驱动程序，不需要提供“增加设备”例程，从这一层意义上讲，它更像传统的遗留驱动程序。另一方面，它往往紧密依赖缓存管理器，而且，缓存管理器反过来调用文件系统驱动程序来执行真正的数据读写操作（参见图 7.15(b)给出的例子）。注意，图 7.17 省略了在缓存管理器与文件 I/O 操作之间内存管理器的角色。

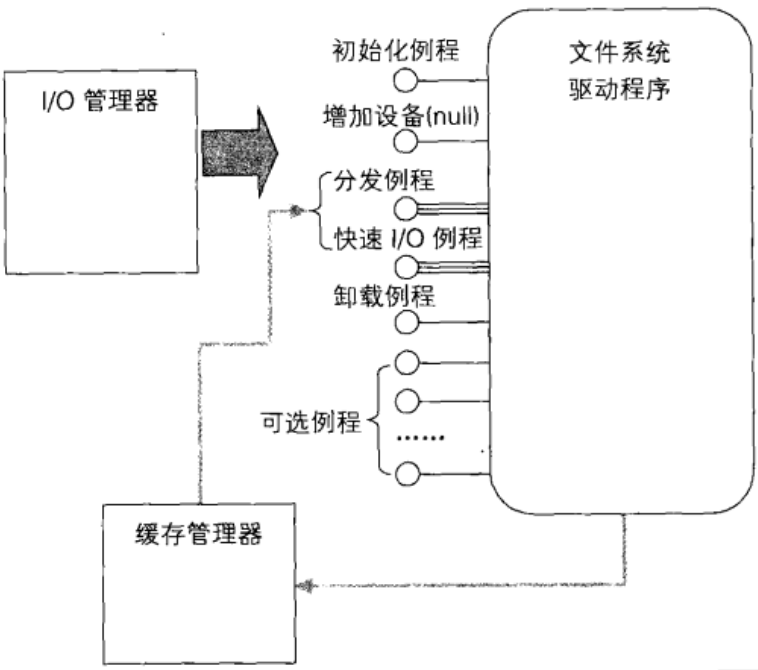


图 7.17 文件系统驱动程序的结构

为了有效地管理一个卷、文件系统驱动程序通常使用一些内部数据结构来维护卷、目录和文件的状态。一个最基本的数据结构是 VCB（Volume Control Block），如 7.3.3 节所述，VCB 通常包含了一个文件系统实例所对应的卷设备对象，因而发生在该文件系统中的 I/O 操作，若有必要，可通过卷设备对象传递到磁盘设备。VCB 是针对一个文件系统实例的，因此，通过 IRP_MN_MOUNT_VOLUME 命令创建的每一个已挂载的文件系统设备对象，都应该构造一个 VCB。通常的做法是在文件系统 DEVICE_OBJECT 后面紧跟一个 VCB 对象，这样驱动程序在处理 IRP 请求时，根据 IRP 请求中指定的设备对象，可以立即定位到 VCB 对象。

除了卷设备对象，VCB 还包含一些与文件系统实例相关的全局信息，比如根目录的结构信息、所有当前已打开的文件对象、反映当前卷状态的一些标志信息，以及用于同步访问 VCB 对象的锁，等等。我们可以把 VCB 对象看做该文件系统实例的全局信息容器。

在文件系统中，通常有两种类型的对象：目录（directory）和文件（file）。按照通常的定义，文件代表了一个数据流，应用程序可以从文件中读写数据。目录是文件系统中用于将多个文件有效地组织起来的容器对象，其本身并不包含用户数据流；相反地，作为一个容器对象，目录可以容纳文件，甚至目录（称为子目录）。任何一个文件必定存在于某一个目录之中；目录构成了文件系统的层次结构，根目录是该层次结构的根。目录和文件既有卷上的数据结构表示，称为磁盘上的表示（on-disk representation），也可能有内存中的表示（in-memory representation）。在磁盘上，最基本的管理数据单元是扇区，支持 Intel x86 系统的磁盘的扇区大小通常为 512 字节，然而，文件系统可以以更大的粒度来管理磁盘上的数据，此粒度称为簇（cluster），是扇区的整数倍大小。因此，从文件系统的管理角度，一个文件是由一组簇构成的，这些簇形成了文件数据流的存储空间；目录也包含一个或多个簇，这些簇中的数据描述了该目录中的文件，包括它们的文件属性（如文件名）和所属的簇的位置信息。不同的文件系统可以定义不同的磁盘结构，在本章后面我们将会看到 FAT 和 NTFS 文件系统的卷是被如何有效地组织和管理的。

磁盘是永久存储体，即使系统掉电或者磁盘迁移到不同的系统上，磁盘上的目录和文件依然存在。只要被挂载的文件系统按照恰当的格式来解析磁盘上的簇或扇区，则这些文件仍然有效。对于在系统运行过程中被引用的文件或目录，文件系统驱动程序必须使用恰当的内存数据结构来描述它们，分别称为 FCB（File Control Block）和 DCB（Directory Control Block）。文件系统驱动程序也可以使用统一的数据结构 FCB，既代表磁盘文件，也代表磁盘目录，这完全取决于驱动程序自身。

文件系统驱动程序通常不会为卷上的所有文件或目录都建立内存数据结构 FCB 或 DCB，而仅仅为已经打开（open）的文件或目录建立起内存表示。FCB 和 DCB 结构的定义完全取决于文件系统驱动程序的实现，一般而言，它们需要包含这样一些信息：文件或目录对象的基本属性，例如名称、大小、时间戳等；维护层次结构所需要的一些信息，比如指向该文件或目录所在目录的 DCB 对象；描述磁盘上存储结构的信息，比如第一个簇的编号；与 Windows 其他内核模块打交道时需要的一些信息，比如对应的内存区对象、各种引用计数、锁等；还可能有一些短时的状态信息，比如当前哪个线程正在执行此文件的 I/O 操作。每一个磁盘文件或目录必须至多只有一个 FCB 或 DCB 存在，文件系统驱动程序有责任保证此唯一性。这其实不难做到，因为所有的 FCB 或 DCB 都是由驱动程序来创建和管理的，并且每一个 FCB 或 DCB 都确定地代表了层次结构中的一个文件或目录。

对于每个文件只有一个数据流的文件系统而言，FCB 也包含了关于此数据流的描述。然而，有些文件系统支持一个文件包含多个数据流，例如 NTFS。在这种情况下，文件系统驱动程序还应该定义一个称为 SCB（Stream Control Block）的数据结构，用于表达一个数据流。此时，内核中的文件对象指向 SCB，而 SCB 又指向 FCB。文件系统驱动程序需要维护好 SCB 和 FCB 两者之间的关系。

在 Windows 文件系统驱动程序中，另一个数据结构是 CCB（Context Control Block），它代表了一个应用程序已打开一个文件。CCB 与 FCB 的不同之处是，FCB 针对每个文件而唯一存在，而 CCB 针对每一次打开文件而存在。例如，两个应用程序共享访问同一个文件，它们的 CCB 不同，但 CCB 中所指的 FCB 是相同的，而且，CCB 中记录了相应的应用程序正在操作该文件的偏移位置。在这种情况下，每个 CCB 关联不同的文件对象（即 FILE_OBJECT）。

以上介绍的所有这些控制块结构都是由文件系统驱动程序针对特定格式的文件系统而定义的，它们对于 Windows 内核和应用程序是透明的。Windows 内核使用文件对象来代表一个已打开的文件实例。在文件对象的数据结构 FILE_OBJECT（参见 6.2.3 节）中，我们看到有两个域：FsContext 和 FsContext2，它们的类型都是 PVOID。实际上，这是两个由文件系统驱动程序使用的域，FsContext 域表示此文件对象所代表的实际对象的数据结构，如果文件对象代表一个磁盘文件，则它指向 FCB（若文件支持多个数据流，则为 SCB）；如果文件对象代表一个磁盘目录，则它指向 DCB；如果文件对象代表整个卷，则直接指向 VCB。FILE_OBJECT 的 FsContext2 域指向 CCB 结构，代表了文件对象的已打开实例。FILE_OBJECT 对象的这两个成员是由文件系统驱动程序在处理 IRP_MJ_CREATE 请求时初始化的，也就是说，当应用程序通过 NtCreateFile 系统服务打开一个文件时，这两个成员已经被正确地设置好，因此，以后当应用程序在此文件对象实例上执行 I/O 操作时，文件系统驱动程序总是可以获得所访问文件（或目录）的 FCB（或 DCB）和 CCB。

尽管 FsContext 所指的 FCB、DCB 或 VCB 对象可以由文件系统驱动程序自行定义，但是，Windows 的缓存管理器需要依赖于这些对象中的一个公共头部才可以工作，此公共头部称为 FSRTL_COMMON_FCB_HEADER。如果文件系统驱动程序要使用缓存管理器提供的缓存功能，那么，它必须确保这些对象都以 FSRTL_COMMON_FCB_HEADER 作为头部成员，或者以 FSRTL_ADVANCED_FCB_HEADER 作为头部成员，这里后者是前者的一个扩展，它进一步支持文件系统过滤管理器的运行环境。关于这两个头部数据结构的定义，读者可以参考 WRK 中的代码，或者参考 Windows DDK 的文档，这里不再列出。

至此，我们已经描述了 Windows 文件系统驱动程序的数据结构框架，图 7.18 显示了所涉及数据结构之间的基本关系，这里假设文件只包含单个数据流，所以，文件对象的 FsContext 成员指向 FCB（或 DCB）。中间的虚线框代表了文件系统驱动程序的职责范围。图中列出了 4 个已经打开的文件对象，其中第一个和第二个文件对象代表了"\MyDocuments\MyDoc1.doc"文件的两个打开实例，它们所指的 FCB 相同，但是 CCB 不相同。文件对象 3 代表了"\MyProjects\Project1\Readme.txt"文件的打开实例，其 FsContext 和 FsContext2 成员分别指向 FCB-2 和 CCB-3。文件对象 4 代表了"\MyProjects\Project2"目录的打开实例，其 FsContext 和 FsContext2 成员分别指向 DCB-3 和 CCB-4 对象。所有的文件对象都指向卷设备对象的 VPB，而 VCB 对象则包含了指向卷设备对象的指针。另外，图中没有显示出 VCB 与 FCB 和 DCB 之间的关系。通常 FCB 和 DCB 都包含一个指向 VCB 的指针；但驱动程序如何通过 VCB 来管理这些已打开文件和目录的 FCB 或 DCB 对象，与驱动程序的具体实现相关。一般而言，VCB 包含指向根目录 DCB 的指针。

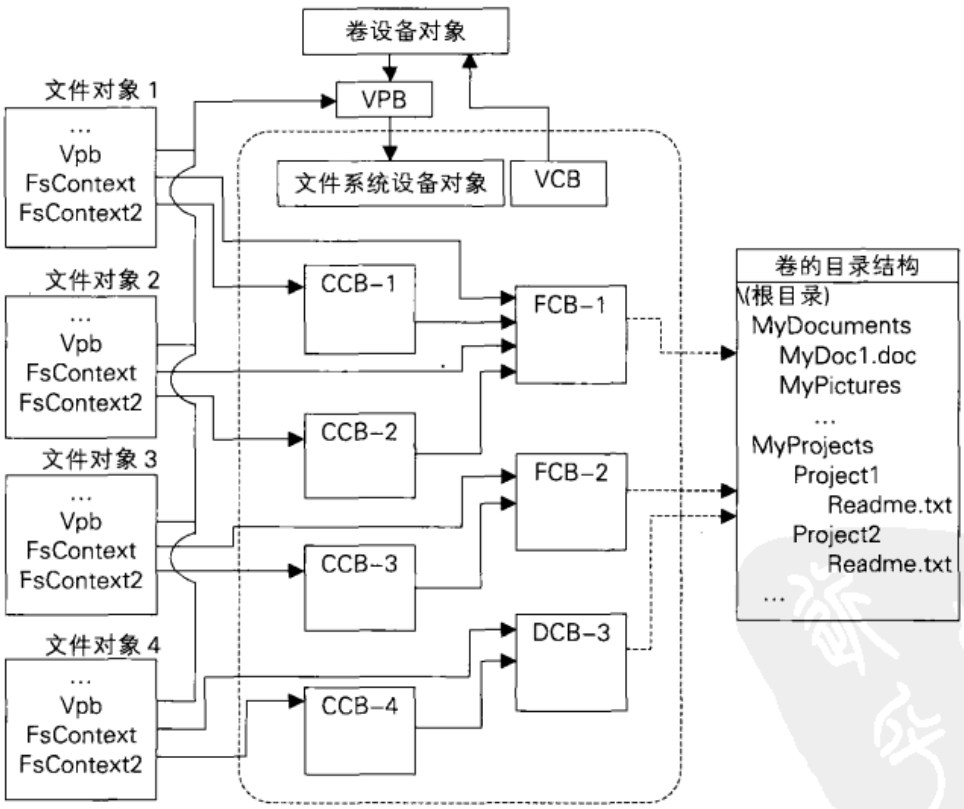


图 7.18 文件系统驱动程序中的数据结构关系示意图

最后，我们进一步理解一下在文件系统驱动程序中这些对象是被如何建立起来的。首先，在文件系统驱动程序的初始化例程，即 DriverEntry 函数中，它创建一个文件系统设

备对象。此设备对象并不代表任何一个卷，其用途是，通过 IoRegisterFileSystem 函数被注册到 I/O 管理器中，因而将来当 I/O 管理器执行卷挂载时，它有机会识别一个卷。例如，在 Windows 中，名为“\Ntfs”、“\Fat”和“\FatCdrom”的设备对象，就是被注册的文件系统设备对象。这些设备对象并不关联任何一个 VCB 对象，因为它们根本不代表一个卷。这样的设备对象也被称为文件系统的**控制设备对象**。

然而，在 IRP_MJ_FILE_SYSTEM_CONTROL 的子命令 IRP_MN_MOUNT_VOLUME 的处理函数中，文件系统驱动程序创建的设备对象则代表了一个卷，它有一个关联的 VCB 对象，并且在 VCB 中有一个成员指向卷设备对象，如图 7.18 所示。

当文件系统驱动程序接收到 IRP_MJ_CREATE 请求时，如果文件对象名称中指定的文件尚无对应的 FCB，那么，驱动程序将会创建一个 FCB；如果该文件已经有一个 FCB 了，那么，驱动程序将不会重复地为同一个文件创建 FCB，它会在 FCB 中用一个引用计数来控制 FCB 的生命周期。因此，驱动程序在接收到创建请求时，必须有办法检测到目标文件是否已存在 FCB。另外，驱动程序对 CCB 的处理方式与此不同，只要能成功地打开指定的文件，则驱动程序总是要构造 CCB 对象。

与 IRP_MJ_CREATE 请求相对应，当驱动程序接收到 IRP_MJ_CLOSE 请求时，首先，CCB 对象必须被删除，其次，它递减 FCB 中的引用计数，若计数减到 0，则删除 FCB。驱动程序可能还需要检查父目录的 DCB，如果父目录 DCB 的存在仅仅是为了当前文件或目录，则还需要连带删除父目录 DCB，此过程一直继续，直至定位到一个必须继续存在的父目录。因此，我们可以看到，CCB 对象的生命周期是从一次打开操作（IRP_MJ_CREATE）开始，直至在同一个文件对象上的关闭操作（IRP_MJ_CLOSE 请求）；而 FCB 或 DCB 的生命周期则是通过引用计数来控制的，从第一次被打开开始，直至引用计数为 0，意味着所有的打开都已经被关闭，不再有任何应用程序或内核组件引用该文件或目录。

7.4.2 RAW 文件系统与 FsRtl

Windows 内核本身提供了一个称为 RAW 的文件系统，其驱动程序对象为“\FileSystem\RAW”。RAW 文件系统代表了无格式的文件系统，也就是说，它不假定磁盘分区的扇区具有任何特定的格式。RAW 文件系统的驱动程序对象直接由 Windows 内核来创建。当 Windows I/O 系统在初始化时，IopInitializeBootDrivers 函数创建一个以 RawInitialize 函数为初始化例程的驱动程序。RawInitialize 函数注册了三个文件系统设备对象：RawDeviceDiskObject、RawDeviceCdRomObject 和 RawDeviceTapeObject，分别被

插入到针对磁盘、CD-ROM和磁带设备的全局文件系统链表中。关于RawInitialize函数，参见base\ntos\raw\rawinit.c文件中的代码。

RAW驱动程序创建的文件系统设备对象总是位于文件系统链表的末尾。当其他文件系统驱动程序通过IoRegisterFileSystem函数来注册文件系统时，如果所注册的文件系统设备对象的Flags成员包含DO_LOW_PRIORITY_FILESYSTEM标志，则该设备对象被插入到链表中最后一个文件系统设备对象的前面，即RAW文件系统设备对象的前面，否则插入到链表的最前面。因此，RAW驱动程序总是最后一个执行卷识别工作。

如7.3.3节所述，卷挂载函数IopMountVolume有一个布尔参数AllowRawMount，指明了当前这次挂载是否允许RAW文件系统参与进来。我们可以看到，从对象管理器调用IopParseDevice函数开始，进一步通过IopCheckVpbMounted函数调用IopMountVolume，在此过程中，AllowRawMount的实际传入值为"!RemainingName->Length && !op->RelatedFileObject"，这里RemainingName是指解析到卷设备对象（譬如"\Device\HarddiskVolume1"）后剩余的名称字符串，而op->RelatedFileObject则是指从一个相对的文件对象开始解析名称。在IopCreateFile函数的情形下，op->RelatedFileObject为NULL。因此，我们可以理解为，当纯粹解析一个卷设备对象，而无须进一步解析卷内部的结构时，AllowRawMount值为TRUE。在这种情况下，当任何一个已注册的文件系统驱动程序都未能识别该卷时，RAW被作为最后一个保底的文件系统识别该卷。

RAW文件系统处理各种I/O请求的例程是RawDispatch，唯有IRP_MJ_SHUTDOWN命令由RawShutdown函数来处理。在RawDispatch函数中，IRP_MJ_FILE_SYSTEM_CONTROL命令由RawFileSystemControl来处理。而在RawFileSystemControl函数中，IRP_MN_MOUNT_VOLUME子命令由RawMountVolume函数来处理。因此，RAW文件系统的卷识别工作最终由RawMountVolume来完成，根据base\ntos\raw\fsctrl.c文件中的代码，它只是简单地创建一个文件系统设备对象，并初始化相应的VCB。

RAW文件系统仅提供有限的卷访问功能。在IRP_MJ_CREATE命令的处理函数RawCreate中，它只允许打开指向卷设备本身的对象，其他情况下均返回失败。针对RAW文件系统的读写请求，即IRP_MJ_READ和IRP_MJ_WRITE命令，是由RawReadWriteDeviceControl函数来完成的。该函数只是简单地将读写请求发送给卷设备对象（即VCB中记录的目标设备对象TargetDeviceObject）。因此，RAW文件系统把整个卷当做单个文件来执行I/O操作，这也正体现了它的"无格式"思想。

RAW文件系统的主要用途是，对于不能被已注册的文件系统驱动程序识别的卷，接管这些卷的文件系统所有权。一方面，它仍然将这些卷纳入到Windows文件系统的

框架中，因而在各种磁盘管理工具中，可看到该卷的文件系统为 RAW；另一方面，它仍然允许一些系统工具，比如备份工具、文件系统恢复工具，能够读取卷上的数据。

Windows 内核除了支持 RAW 文件系统以外，还提供了一套针对文件系统的支持库，称为文件系统运行库（File System Runtime Libaray，简称 FsRtl）。FsRtl 中的例程实际上是执行体层面上的支持例程，只不过它们与文件系统的关系太过于密切了，以致于直接放在执行体中并不合适，放在 I/O 管理器中也不合适。这些例程的名称以“FsRtl”作为前缀，在 Windows DDK 中有关于它们的说明。

WRK 包含了这些 FsRtl 例程的实现代码（位于 base\ntos\fsrtl 目录下）。当 Windows 内核初始化时，它会调用 FsRtlInitSystem 函数，完成 FsRtl 的全局初始化。在 FsRtl 中，有两个数据类型被广泛使用：

- LBN（Logical Block Number，逻辑块号），逻辑块号用于标识磁盘上的物理块，假设磁盘上的扇区数为 N，则逻辑块号为 0~*N*–1。
- VBN（Virtual Block Number，虚拟块号），虚拟块号用于标识一个文件中的扇区，其值相对于文件起始处的扇区数。0 代表该文件的第一个扇区，1 代表第二个扇区，以此类推。

这两个数据类型均被定义为 ULONG。此外，上一小节提到的用于支持缓存管理器的 FCB、DCB 和 VCB 数据结构的公共头部 FSRTL_COMMON_FCB_HEADER 和 FSRTL_ADVANCED_FCB_HEADER 也被定义在 FsRtl 中，参见 base\ntos\inc\fsrtl.h 文件。

FsRtl 提供的支持例程主要包括以下类别：

- 对于名称解析的支持，尤其是针对 DBCS（双字节字符串）和 UNICODE 的支持，包括对于常用文件系统名称规范的解析。参见 base\ntos\fsrtl\dbcsname.c 和 name.c 文件中的代码。
- MCB（Mapping Control Block）支持，这里 MCB 记录了针对一个文件的“从 VBN 到 LBN”的映射。文件系统只要查过一次 VBN，并将它记录到 MCB 中，以后访问此 VBN 就可以直接从 MCB 中获得 LBN。参见 base\ntos\fsrtl\largemcb.c 文件中的代码。
- 快速 I/O 的支持，利用文件系统的快速 I/O 入口来执行各种缓存操作。参见 base\ntos\fsrtl\fastio.c 文件中的代码。
- 字节范围文件锁的支持。文件系统驱动程序可以在文件被打开的时候初始化一个 FILE_LOCK 类型的变量，然后利用 FsRtl 提供的函数（FsRtlProcessFileLock）来处

理 IRP_MJ_LOCK_CONTROL 命令的 I/O 请求。同样，当处理读或写请求时，文件系统驱动程序可以通过 FsRtl 函数（例如 FsRtlCheckLockForReadAccess 和 FsRtlCheckLockForWriteAccess）来检查是否可以执行读或写操作。参见 base\ntos\fsrtl\filelock.c 文件中的代码。

- 文件系统过滤回调的支持，参见 FsRtlRegisterFileSystemFilterCallbacks 函数的代码，以及文件 base\ntos\fsrtl\fsfilter.c 中的代码。
- 文件系统隧道能力的支持，这里所谓的隧道是指，若一个文件被删除，然后在很短的时间内又创建（或重命名）一个文件，其名称与原先被删除的文件名相同，那么，新文件的文件名和时间戳与原来的相同。原因是，当文件被删除以后，文件系统的元文件信息仍然保存在系统缓存中。文件 base\ntos\fsrtl\tunnel.c 中包含了 FsRtl 对于文件系统隧道能力的支持例程。有关文件名隧道机制的解释，读者可以参考相关资料[FN-TUNNELING]。
- MUP（Multiple UNC Provider，多 UNC 提供者）支持。当应用程序指定一个 UNC 名称来打开一个远程文件或设备时，MUP 驱动程序就会被请求到。它利用那些通过 FsRtlRegisterUncProvider 函数向它注册过的 UNC 提供者，由它们来识别 UNC 路径。FsRtl 包含了与 MUP 的处理有关的几个函数，参见 base\ntos\fsrtl\unc.c 文件中的代码。
- 目录变化通知的支持。支持目录变化通知的文件系统可以利用 FsRtl 提供的函数来简化这一工作。通常的做法是，文件系统驱动程序在 VCB 对象中维护一个通知块队列，然后在接收到变化通知请求时调用 FsRtl 函数来完成此请求。NTFS 和 FAT 文件系统均支持目录变化通知。读者可以参考 base\ntos\fsrtl\notify.c 文件中的代码，以及 fastfat 驱动程序中的用法。

以上这些能力正是文件系统通常需要提供的功能，Windows DDK 包含的 FAT 文件系统驱动程序（fastfat.sys）演示了大多数功能的用法，读者可以参考。

7.4.3 文件系统的 I/O 过滤

发送至文件系统的 I/O 请求在到达文件系统驱动程序以前，还可能经过 fltMgr 驱动程序或其他的过滤驱动程序。Windows 的文件系统框架提供了两种类型的过滤驱动程序：文件系统过滤驱动程序和文件系统小过滤驱动程序。这两种驱动程序的目的都是在文件系统驱动程序处理 I/O 请求以前，首先获得 I/O 请求的处理权，从而为文件系统提供增值服务，或者改变文件系统驱动程序的标准行为。两者的区别在于，文件系统小过

滤驱动程序被纳入到 fltMgr 驱动程序设定的过滤框架中，因而开发这种驱动程序相对要容易、便捷得多，而文件系统过滤驱动程序则是标准的驱动程序，它们创建设备对象并将其附载到文件系统设备栈中。下面首先介绍传统的文件系统过滤驱动程序。

(一) 文件系统过滤驱动程序（File System Filter Driver）

文件系统过滤驱动程序能够过滤针对文件系统或卷的 I/O 操作，它们可以监视（monitor）、记录（log）、修改或阻止这些 I/O 操作。典型的应用包括：防病毒检测、加密或压缩文件数据，引入一层额外的存储管理（譬如 HSM，即 Hierarchical Storage Management）。文件系统过滤驱动程序并不管理任何一个设备，它们使用了 Windows 设备驱动程序的框架结构，通常也有初始化例程、与文件相关的 I/O 请求的分发例程，以及相应的 I/O 完成例程等。

文件系统过滤驱动程序的工作方式与设备栈中其他驱动程序的工作方式类似，它们接收 IRP，执行自身的逻辑，再根据需要，设置完成例程并将 IRP 发送给低层的驱动程序。如果有必要，也可以在处理过程中创建自己的 IRP，并发送给低层的驱动程序。因此，从这一层意义上来讲，文件系统过滤驱动程序与 6.5.5 节介绍的设备栈中的过滤驱动程序是类似的。

然而，文件系统过滤驱动程序毕竟是针对文件系统的过滤驱动程序，它们有一些特殊之处：文件系统过滤驱动程序不是 WDM 驱动程序，它们不处理即插即用命令和电源命令，也不需要提供“增加设备（AddDevice）”例程。显然，文件系统过滤驱动程序往往只关心与文件和目录相关的 I/O 请求，对于其他的 I/O 请求，它们无须在 DriverEntry 中指定分发例程。

下面我们通过 Windows DDK 随带的例子驱动程序 filespy 来说明文件系统过滤驱动程序的安装、对文件系统设备栈的影响，以及对 I/O 请求的过滤处理。

在 Windows Server 2003 系统上，文件系统过滤驱动程序的安装可以通过一个.inf 文件来完成。Windows 已经为文件系统过滤驱动程序定义了一组类别（Class）和类别 GUID（ClassGuid），以及它们的过滤顺序，请参考相关资料[FS-FLT-CLS]。关于.inf 文件的编写方法，请参考相关文档[FS-FLT-INF]。Filespy 例子已经提供了.inf 文件，读者如果需要安装，只需在 Windows 资源管理器窗口中右键单击 filespy.inf 文件并选择“安装”菜单命令，Windows 安装程序就会按照.inf 文件中的指示，将文件拷贝到系统目录中，并且建立起相应的注册表键。注意，安装过程并不会加载 filespy 驱动程序，我们还需要使用控制台命令“net start filespy”将驱动程序加载到当前系统中。

在 filespy 驱动程序的初始化例程 DriverEntry 中，它利用 IoRegisterFsRegistrationChange 函数，注册一个接收“文件系统注册变化”的通知。I/O 管理器针对当前所有已注册的文件系统，逐个调用该注册函数中指定的回调函数（即 SpyFsNotification）。所以，filespy 通过这种方法，可以看到所有已注册的文件系统。然后，filespy 在回调函数中创建一个设备对象，附载到文件系统设备对象上。注意，在回调函数中，它直接看到的是文件系统的控制设备对象，并非用于管理卷的文件系统设备对象。为了进一步附载到卷的文件系统设备对象上，filespy 通过 IoEnumerateDeviceObjectList 函数来列举文件系统驱动程序的设备对象链表，以便获得所有管理卷的文件系统设备对象，并为每个卷的文件系统创建一个 filespy 设备对象，再附载到该文件系统的设备栈中。因此，filespy 在初始化例程中，已经完成了附载到所有文件系统设备对象的工作，从而可以监视所有有关文件操作的 I/O 请求。图 7.19 显示了一个 ntfs 卷被附载了 filespy 设备对象前后的设备栈。

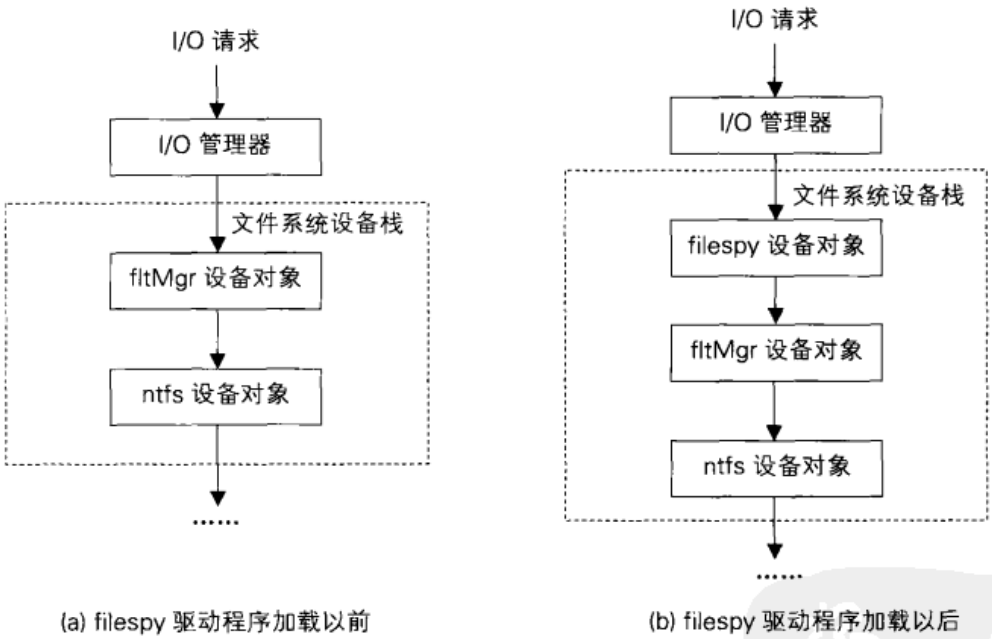


图 7.19 filespy 驱动程序加载前后的 ntfs 卷的设备栈

卸载 filespy 的过程相对要简单一些，我们可以使用控制台命令“net stop filespy”将驱动程序卸载，此时，filespy 的 DriverUnload 函数被调用，它的做法要简单很多，直接使用 IoEnumerateDeviceObjectList 函数，列举所有已附载的 filespy 设备对象，并逐个解除附载。因此，当 filespy 程序卸载以后，所有的文件系统设备栈又恢复原样。

与其他的设备驱动程序类似，filespy 驱动程序也包含两种类型的设备对象，一是控制

设备对象，用户模式的 filespy.exe 程序通过此控制设备对象来管理 filespy 的功能，包括解除 filespy 在特定卷上的附载、列出正在被监视的所有卷等。Filespy 用一个全局变量（gControlDeviceObject）记录下此控制设备对象。另一种对象是真正附载到各个文件系统实例的设备对象。在接收 I/O 请求的分发例程（SpyDispatch）中，控制设备对象处理 IRP_MJ_DEVICE_CONTROL 命令，因此，用户模式的 filespy.exe 程序通过 API 函数 DeviceIoControl，即 NtDeviceIoControlFile 系统服务，与 filespy 设备驱动程序进行通信；若附载于文件系统实例的 filespy 设备对象接收到 I/O 请求，则根据驱动程序当前的设置，可能输出到调试器，也可能将这一请求记录下来。

Filespy 还需要监视卷的动态挂载，所以，filespy 控制设备对象需要处理 IRP_MJ_FILE_SYSTEM_CONTROL 请求，若有新的卷被挂载到已有的文件系统驱动程序中，则 filespy 有机会为该卷的文件系统附载一个 filespy 设备对象。

Filespy 同样也记录下快速 I/O 的处理，它在初始化例程中登记所有它感兴趣的快速 I/O 例程。它登记的这些例程只是简单地记录下这些快速 I/O 操作，然后交给低层的驱动程序执行真正的快速 I/O 处理。

Filespy 驱动程序演示了文件系统过滤驱动程序中用到的典型技术（包括 FsRtl 提供的文件系统过滤回调），甚至还包含了一个内核调试器扩展模块 fspykd，用于在内核调试器中检查 filespy 的数据结构。Windows DDK 提供的 filespy 源代码有详尽的注释，可供读者参考。除了 filespy 例子，Windows DDK 还提供了另一个过滤驱动程序例子 sfilter。Sfilter 提供了类似于 filespy 的过滤驱动程序框架，但不执行任何记录 I/O 的动作。对于想编写文件系统过滤驱动程序的开发人员，sfilter 可能是一个很好的起点。

在介绍文件系统过滤驱动程序时，一个不得不提到的实用例子是 FileMon[FILEMON]。这是一个被广泛用于诊断各种与文件系统有关的系统性能问题的工具，它的核心技术正是上面介绍的文件系统过滤驱动程序。FileMon 在启动以后，会动态生成一个驱动程序，并将它加载到系统中，再附载到要监视的卷文件系统中，因而可以监听到所有发生在这些卷上的文件 I/O 操作。所以，虽然我们在 FileMon 工具中没有看到驱动程序，但是，在 FileMon 运行起来以后，总是可以通过某种方法看得到它所使用的过滤驱动程序的痕迹。

(二) 文件系统小过滤驱动程序（File System Minifilter Driver）

在 7.3.4 节我们已经看到，Windows 的过滤管理器驱动程序（fltMgr.sys）提供了一个 I/O 过滤框架，它允许一种被称为文件系统小过滤驱动程序的驱动程序加载到系统中，并向 fltMgr 注册它要过滤哪些 I/O 操作。小过滤驱动程序可以过滤的 I/O 操作包括：基于 IRP

的 I/O 请求、快速 I/O，以及文件系统过滤回调操作。对于每一个要过滤的操作，小过滤驱动程序可以注册一个“预操作回调例程（preoperation callback routine）”或一个“后操作回调例程（postoperation callback routine）”，或者两个都注册。当 fltMgr 处理一个 I/O 操作时，在处理该 I/O 操作之前或之后，它调用每一个已经注册了要过滤该 I/O 操作的小过滤驱动程序。

当有多个小过滤驱动程序注册了同样的 I/O 操作时，fltMgr 调用其回调例程的顺序是由每个小过滤驱动程序在附载时指定的一个高位值（altitude，整数值）来决定的。基本的原则是，高位值越大，则该小过滤驱动程序在文件系统的 I/O 驱动程序栈中就处于越高的位置，而越会被先执行过滤功能。相反地，高位值越小的小过滤驱动程序，则越是接近于文件系统驱动程序。为了与文件系统过滤驱动程序有较为一致的顺序关系，针对文件系统过滤驱动程序的加载顺序组（load order group）都有对应的高位值范围。Microsoft 为常见的过滤驱动程序类型预定义了相应的高位值范围，参见相关资料[FS-MINIFLT-ORDER]。

利用高位值来控制小过滤驱动程序在 I/O 驱动程序栈中的位置，这使得它们的位置不再像前面介绍的文件系统过滤驱动程序那样，简单地由加载顺序来决定。譬如 filespy，它被加载时总是附载到文件系统设备栈的顶部。相反地，小过滤驱动程序无论何时被加载，总是被插入到适合于其高位值的位置上。由于小过滤驱动程序并不直接附载到文件系统的设备栈中，它们依附于 fltMgr 驱动程序来实现过滤能力，因此会导致，在文件系统的设备栈中，fltMgr 驱动程序可能需要插入多个设备对象实例，才能有效地管理各种高位值的小过滤驱动程序。这样的 fltMgr 设备对象实例被称为过滤管理器帧（filter manager frame）。图 7.20 显示了有两个过滤管理器帧的文件系统设备栈。首先，过滤管理器帧 0 负责应用“FsFilter 压缩”和“FsFilter 加密”，即小过滤驱动程序 C 和 D 的过滤能力，它们的高位值最小，所以过滤管理器帧 0 接近于文件系统设备对象。其次，HSM 过滤驱动程序是一个传统的文件系统过滤驱动程序，没有定义高位值，但是根据 Microsoft 为各个加载顺序组指定的高位值范围，FsFilter HSM 组的高位值范围为 180 000~189 999，参见相关资料[FS-MINIFLT-ORDER]。所以，为了正确地应用小过滤驱动程序 A 和 B，fltMgr 驱动程序在 HSM 驱动程序之上插入一个过滤管理器帧 1，由它来应用“FsFilter 防病毒”和“FsFilter 复制”，即小过滤驱动程序 A 和 B 的过滤能力。因此构成了图 7.20 中的 I/O 驱动程序栈。

理解了小过滤驱动程序的工作模型以后，接下来我们仍然通过 Windows DDK 中的例子来说明文件系统小过滤驱动程序的结构和实现技术。Minispy 是一个在功能上类似于 filespy 的小过滤驱动程序，但由于使用了 fltMgr 提供的框架，所以，它的代码比 filespy 简洁很多。

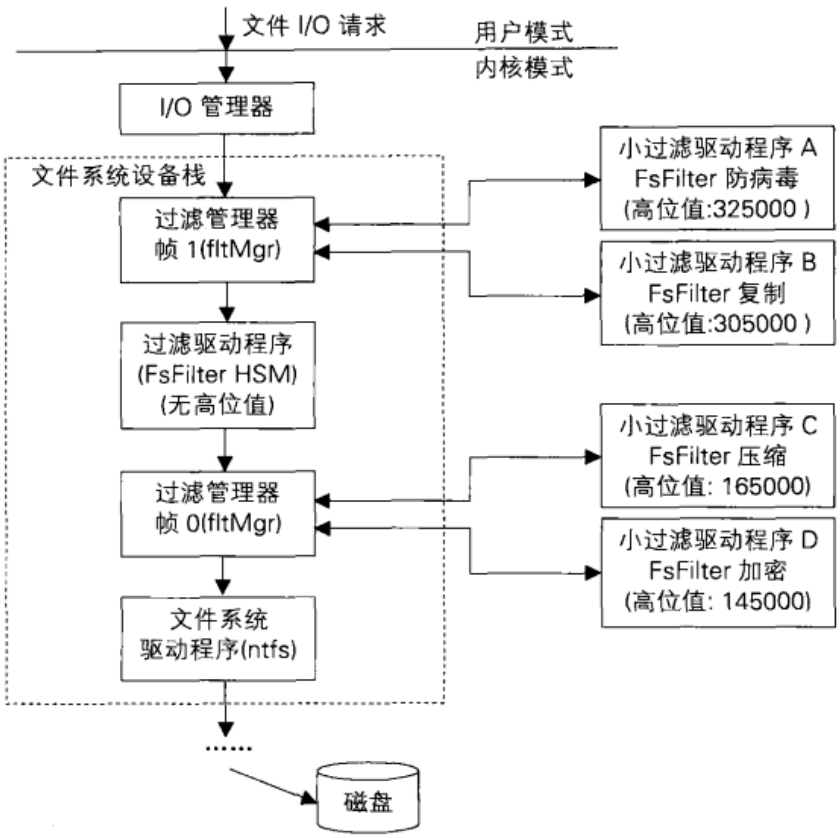


图 7.20 过滤管理器帧与小过滤驱动程序

小过滤驱动程序的安装可通过.inf 文件来进行，例如，在 Windows 资源管理器窗口中右键单击 minispy.inf 文件再选择“安装”菜单命令。此安装过程并不会自动将小过滤驱动程序加载到系统中。为了加载 minispy 驱动程序，用户可以通过“fltmc load minispy”或“net start minispy”命令来完成。注意，小过滤驱动程序的高位值是在.inf 文件中定义的。关于如何为小过滤驱动程序构造一个有效的.inf 文件，请参考相关资料[FS-MINIFLT-INF]。

小过滤驱动程序虽然也是一个驱动程序，但它本质上更像是 fltMgr 驱动程序所定义的过滤框架中的二进制插件。当小过滤驱动程序被加载到内存中时，在初始化例程中，它不像文件系统过滤驱动程序那样，要创建一个设备对象并附载到要过滤的文件系统的设备栈中，而是调用 FltRegisterFilter 函数，向 fltMgr 注册其自身，以便加入到全局的小过滤驱动程序链表中。在调用 FltRegisterFilter 函数时，小过滤驱动程序可以提供一组可选的回调例程，从而使 fltMgr 可以在特定事件发生时回调到小过滤驱动程序中。

仅仅向 fltMgr 注册自身还不够，为了达到过滤 I/O 操作的目的，小过滤驱动程序还需要调用 FltStartFiltering 函数，以便通知 fltMgr 它已经做好准备工作，可以开始过滤 I/O 操

作了。这通常是初始化例程在完成了全局初始化以后，返回 I/O 管理器之前调用的关键函数。Minispy 驱动程序也不例外，它的 DriverEntry 函数首先执行必要的初始化工作，然后调用 FltRegisterFilter 函数注册回调例程，最后调用 FltStartFiltering 函数启动过滤功能。

小过滤驱动程序不会出现在文件系统的设备栈中，它们并不使用设备对象来跟 I/O 管理器或者 fltMgr 打交道，相反，它们完全以回调例程的方式进行工作。在初始化例程中调用的注册函数 FltRegisterFilter 要求指定这些回调例程，以下是 FltRegisterFilter 函数的原型以及参数 Registration 的数据类型 FLT_REGISTRATION 的定义：

```
NTSTATUS
FLTAPI
FltRegisterFilter (
    __in PDRIVER_OBJECT Driver,
    __in CONST FLT_REGISTRATION *Registration,
    __deref_out PFLT_FILTER *RetFilter
    );

typedef struct _FLT_REGISTRATION {
  USHORT  Size;
  USHORT  Version;
  FLT_REGISTRATION_FLAGS  Flags;
  CONST FLT_CONTEXT_REGISTRATION  *ContextRegistration;
  CONST FLT_OPERATION_REGISTRATION  *OperationRegistration;
  PFLT_FILTER_UNLOAD_CALLBACK  FilterUnloadCallback;
  PFLT_INSTANCE_SETUP_CALLBACK  InstanceSetupCallback;
  PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK  InstanceQueryTeardownCallback;
  PFLT_INSTANCE_TEARDOWN_CALLBACK  InstanceTeardownStartCallback;
  PFLT_INSTANCE_TEARDOWN_CALLBACK  InstanceTeardownCompleteCallback;
  PFLT_GENERATE_FILE_NAME  GenerateFileNameCallback;
  PFLT_NORMALIZE_NAME_COMPONENT  NormalizeNameComponentCallback;
  PFLT_NORMALIZE_CONTEXT_CLEANUP  NormalizeContextCleanupCallback;
#if FLT_MGR_LONGHORN
  PFLT_TRANSACTION_NOTIFICATION_CALLBACK;
  PFLT_NORMALIZE_NAME_COMPONENT_EX  NormalizeNameComponentExCallback;
#endif // FLT_MGR_LONGHORN
} FLT_REGISTRATION, *PFLT_REGISTRATION;
```

FLT_REGISTRATION 数据结构包含了多个可选的回调函数，譬如 FilterUnloadCallback 成员例程在小过滤驱动程序被卸载之前调用。尤其值得一提的是 OperationRegistration 成员，这是一个指向可变长度的 FLT_OPERATION_REGISTRATION 数组的指针成员。数组中的每一项指定了该小过滤驱动程序要过滤哪个 I/O 操作，以及要在 I/O 操作之前还是之后，或者两者均要进行处理，FLT_OPERATION_REGISTRATION 数据结构的定义如下：

```
typedef struct _FLT_OPERATION_REGISTRATION {
    UCHAR MajorFunction;
    FLT_OPERATION_REGISTRATION_FLAGS Flags;
    PFLT_PRE_OPERATION_CALLBACK PreOperation;
    PFLT_POST_OPERATION_CALLBACK PostOperation;
    PVOID Reserved1;
} FLT_OPERATION_REGISTRATION, *PFLT_OPERATION_REGISTRATION;
```

这里 MajorFunction 域指定了 I/O 操作的主命令码（相当于 I/O 请求的主命令码），除了 I/O 管理器已经定义的 IRP_MJ_<*XXX*>（它们的值位于 0~0x1b 之间）以外，fltMgr 还针对文件系统的 I/O 特性定义了一些额外操作，比如 IRP_MJ_ACQUIRE_FOR_MOD_WRITE、IRP_MJ_ACQUIRE_FOR_CC_FLUSH、IRP_MJ_MDL_READ、IRP_MJ_VOLUME_MOUNT 等。FltMgr 也定义了作为数组结束标志项的 MajorFunction 为 IRP_MJ_OPERATION_END（0x80），因此，FltRegisterFilter 函数在解析 Operation-Registration 数组时，一看到 IRP_MJ_OPERATION_END 数组项，就知道这是数组的末尾。FLT_OPERATION_REGISTRATION 数据结构中的 PreOperation 和 PostOperation 域分别指定了该 I/O 操作的前操作回调例程和后操作回调例程；Flags 域定义了该 I/O 操作的回调标志，目前 fltMgr 定义了 FLTFL_OPERATION_REGISTRATION_SKIP_PAGING_IO（若是换页 I/O 操作，则不调用这一数组项中的回调例程）和 FLTFL_OPERATION_REGISTRATION_SKIP_CACHED_IO（若是缓存 I/O 操作，则不调用这一数组项中的回调例程）标志。关于以上这些数据结构以及相应的宏定义，请参考 Windows DDK 中的相关文档或者头文件 fltKernel.h。

现在回到 minispy 驱动程序，它的 OperationRegistration 数组，即源代码中的 Callbacks 数组（位于 RegistrationData.c 文件中），定义了所有对文件系统可能有意义的 I/O 操作，并且指定了前操作回调例程（SpyPreOperationCallback）和后操作回调例程（SpyPostOperationCallback）。因此，从过滤效果来看，minispy 借用 fltMgr 提供的过滤框架，达到了与文件系统过滤驱动程序 filespy 类似的效果。关于这两个回调例程的实现方法，这里不再介绍。Windows DDK 中有关于回调例程写法的完整文档，读者也可以参考 minispy 驱动程序的实现代码。

至此，我们已经看到了 Windows 文件系统使用的两种过滤机制：直接让文件系统过滤驱动程序创建的设备对象附载到文件系统的设备栈中，或者通过 fltMgr 提供的过滤框架以回调方式达到过滤 I/O 操作的目的。相比之下，Windows 更加推荐后一种过滤机制，因为它有以下一些优势[FS-MINIFLT-ADV]：

- 对小过滤驱动程序的加载顺序有更好的控制。
- 小过滤驱动程序可在系统运行时卸载。FltMgr 已经做好了同步和保护工作。
- 小过滤驱动程序可以只过滤感兴趣的 I/O 操作。
- 回调模型降低了内核栈的深度要求。

- 更少的冗余代码，并且降低了过滤代码的复杂性。小过滤驱动程序的主要代码在于过滤 I/O 操作。
- 允许方便地加入对于新 I/O 操作的过滤支持，而不会打破以前的小过滤驱动程序。
- 更好地支持多个平台。FltMgr 会决定是否调用小过滤驱动程序中的回调例程。
- FltMgr 也提供了常见的用户模式应用程序方面的支持。

7.4.4 FAT 文件系统

FAT 文件系统是 DOS 时代遗留的产物，现在仍然被广泛使用，其中一个原因是为了能够从早期的 Windows 文件系统升级或迁移过来，或者在多引导的系统中与其他的操作系统保持文件系统兼容；另一个原因是为了支持可移动或可插拔的存储介质（软盘、U 盘或移动硬盘等），作为一种跨操作系统共享存储体的解决方案。Windows 的 FAT 文件系统驱动程序是 fastfat.sys，Windows DDK 中的例子程序 fastfat 包含了完整的源代码。

FAT 文件系统将一个卷上的扇区划分成几个区域，如图 7.21 所示。引导扇区包含了 FAT 文件系统的标识和各种参数。文件分配表（File Allocation Table，FAT）正是 FAT 文件系统名称的由来，它实际上是一个固定长度的数组。卷上的每一簇在文件分配表数组中都有对应的一项，数组项的位长可以是 12、16 或 32，对应的文件系统格式分别称为 FAT12、FAT16 或 FAT32。当一个卷被格式化为 FAT 文件系统时，文件分配表区域的大小是确定的；第二个文件分配表区域是一个备份，当第一个文件分配表被破坏时，可以通过备份的文件分配表来恢复文件系统。随后是根目录区域，FAT12 和 FAT16 文件系统的根目录是预先指定大小的区域，因此，根目录中能存放的文件或目录有数量限制。在根目录以后是数据区，所有的文件和非根目录均存放在数据区中。在 FAT32 中，根目录并非位于固定的区域，而是在引导扇区的参数中指定了起始位置，而且它也像一个普通目录一样，可以动态地增长。

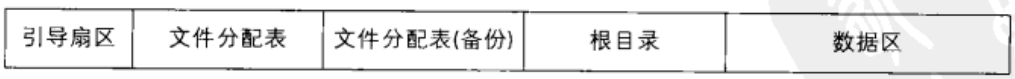


图 7.21 使用 FAT 文件系统的卷的格式

FAT 文件系统可以在格式化的时候指定簇的大小，比如 FAT16 的簇大小可以从 512 B（单个扇区）直至 64 KB（128 个扇区）。由于 16 位的簇标识符只能表达 2^{16} 个簇，所以，FAT16 能表达的卷的大小最多是 4 GB。FAT12 适合于存储量较小的卷，比如早期的软盘（只有 1.2 MB 或 1.44 MB 大小）。FAT32 使用 32 位簇标识符，但实际上只使用了 28 位，

并且FAT32的簇可以达到32 KB，所以，FAT32能表达的卷的大小最多是2^{28}×32 KB=8 TB。实际上，Windows 限制了 FAT32 卷最大只能为 32GB。关于 FAT 文件系统的磁盘格式规范，请参考相关资料[FAT]。

FAT 文件系统的引导扇区如图 7.22 所示，它以一条跳转指令为开始，前 3 个字节为“eb 58 90”或“e9 57 00”（分别对应于短跳转和近跳转指令，FAT 规范中允许这里的 58 或 57、00 为任何字节）。接下来 8 个字节为一个 OEM 名称字符串，规范中推荐的设置为“MSWIN4.1”，但 Windows 不检查此名称串中的内容。接下来从第 11 个字节，一直到第 89 个字节为 BIOS 参数块（BIOS Parameter Block，BPB），其中定义了簇的大小、根目录中的目录项数（对于 FAT32 卷，该值为 0）、卷的总大小、FAT 表的个数（总是 2）和大小、卷标签（volume label）等信息。若是 FAT32，BPB 还定义了根目录的第一个簇的簇编号。在引导扇区中，从第 90 个字节开始，是可执行代码，实际上，第一条指令即跳转到此处。这部分代码用于从该卷所在分区引导时的启动过程。最后，引导扇区的第 510~511 字节必须是“55 aa”标志字节。

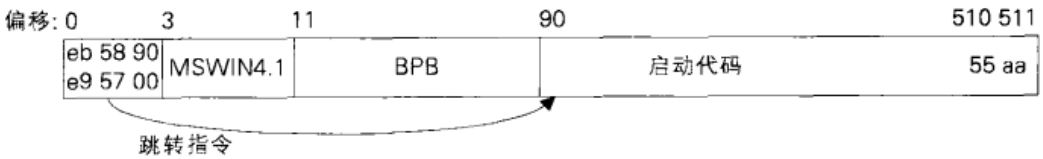


图 7.22 FAT 文件系统的引导扇区格式

BPB 中的信息决定了 FAT 文件系统的区域划分结构，包括每一部分的起始位置和大小。BPB 指定了在 FAT 区域之前的保留扇区的数量（其中包括引导扇区本身），在 FAT12 和 FAT16 中固定为 1，而在 FAT32 中通常为 32。这也相当于决定了文件分配表区域的起始扇区。在文件分配表中，每一项都描述了一个簇的当前状态。如果一个簇是空闲的，则对应的表项包含的值为 0；如果一个簇当前属于某一个文件，则对应的表项包含了该文件下一个簇的簇编号，因此，从一个文件的起始簇开始，利用文件分配表中的表项值，可以依次定位到该文件所有的簇，这些簇的表项构成了该文件的分配链。链尾项的值为 EOC（End Of Clusterchain），在 FAT12 中为 0x0fff，在 FAT16 中为 0xffff，在 FAT32 中为 0x0fffffff。图 7.23 显示了 FAT16 文件系统中文件分配表区域最前面表项的示意图。

注意，文件分配表中描述的簇是指数据区中的簇，即整个卷扣除保留扇区（包括引导扇区）、文件分配表区域和根目录区域以后的簇，而且，簇的编号从 2 开始，即数据区第一个簇的编号为 2。另外，如果一个簇已被确定为坏簇（bad cluster），则对应的表项也会被打上标记，以后 FAT 文件系统将不再使用这一簇。坏簇标记值在 FAT12 中为 0x0ff7，在 FAT16 中为 0xfff7，在 FAT32 中为 0x0fffff7。

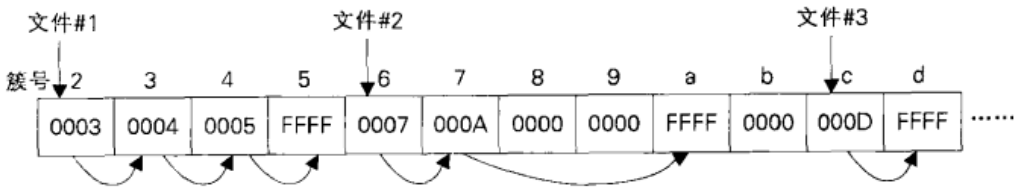


图 7.23 文件分配表中的文件分配链

在 FAT12 和 FAT16 文件系统中，根目录占据文件分配表区域之后一段单独的连续扇区。引导扇区的 BPB 指明了根目录下保留多少个目录项。因此，根目录所占据的扇区并不计算在文件分配表的管辖范围内。然而，FAT32 的 BPB 仅仅指明了根目录的起始簇编号（通常为 2，但允许其他值），而没有保留任何专用的扇区或簇。

无论是根目录，还是普通的目录，在磁盘上都是一个数据流，占据一个或多个簇。只不过，目录的数据流按照每 32 字节一个目录项来解析，而并非任意形式的数据流。每个目录项描述了该目录下的一个文件或子目录。目录项的格式如图 7.24 所示。

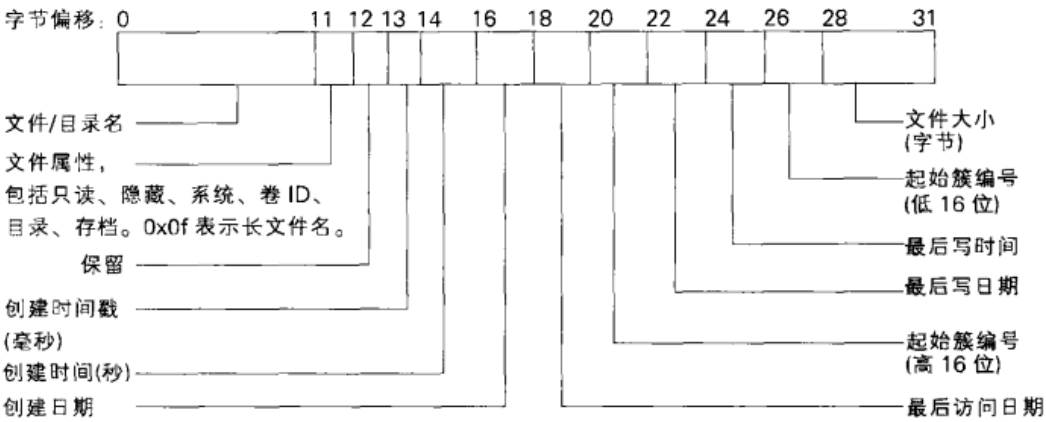


图 7.24 FAT 目录项的格式

我们可以看到，FAT 目录项包含了文件的名称、属性、大小，以及各种时间和日期信息，最重要的是，它包含了起始簇的编号，即该目录项所对应的文件或子目录的分配链的链头。因此，根据一个文件的目录项，我们可以从起始簇开始，沿着文件分配表中的分配链，顺序定位到该文件的所有簇，从而读取该文件的所有数据。若一个目录项对应的是子目录，则从起始簇开始，沿着文件分配表中的分配链，就可以访问该子目录中的所有目录项，从而构成了 FAT 文件系统中的目录层次结构。

理解了 FAT 文件系统的存储机制以后，我们再来看一下 FAT 文件系统中的文件和目录的命名规范。在历史上，FAT 文件系统使用 MS-DOS 8.3 命名规范，即 8 个字符的主名加上

3 个字符的扩展名，例如“config.sys”和“autoexec.bat”。FAT 目录项的首字符，即名称域的首字符，若为 0xe5，则说明这是一个空闲项；若为 0，则说明这是最后一个目录项，而且，此目录项后面的所有目录项都以 0 打头。名称域的 11 个字符可以是任意字母、数字或大于 127 的码（除了 0xe5），特殊字符“$ % ' - _ @ ~ ` ! ( ) { } ^ # &”也是合法的。FAT 文件系统的名称不区分大小写，同一个目录下的文件或子目录必须有唯一的名称。

在 FAT 文件系统中，这种 8.3 格式的名称被称为短名称，显然它的描述能力很有限。FAT 文件系统在不改变目录项存储格式的基础上，扩展了命名规范，从而支持长名称和 UNICODE 文件名。其做法是，用多个连续的目录项来描述一个文件，其中最后一个目录项是对应的短文件名目录项，包含了该文件的各种常规属性；在它之前的目录项存储了原始的长名称，保留大小写，并且其属性域（即第 11 字节）的值为 0x0f，表示这是一个长名称的目录项。FAT 文件系统总是使用 UNICODE 来存储长名称，每个字符占据 16 位。长名称的文件或目录有对应的短名称，关于短名称的生成算法，请参考 FAT 文件系统规范[FAT]，本书不再讲述。

FAT 是一个相对简单的文件系统格式，Windows DDK 提供了 FAT 文件系统驱动程序 fastfat 的源代码。尽管 FAT 的磁盘存储格式非常简单，但是，fastfat 驱动程序却是一个很具参考价值的、功能全面的文件系统驱动程序，它演示了编写文件系统驱动程序所涉及的各种关键技术，包括将数据存储到磁盘上、与缓存管理器协作，以及处理各种 I/O 请求，例如创建文件、读或写文件、设置文件属性等操作。

首先，fastfat 驱动程序在它的初始化例程中创建两个设备对象分别作为 FAT 磁盘文件系统和 FAT CD-ROM 文件系统的控制设备对象。注意，这两个设备对象不包含由驱动程序定义的扩展部分。然后，初始化例程完成例行的初始化工作，包括各种全局变量的初始化、I/O 分发例程的设置等。最后，初始化例程向 I/O 管理器注册这两个控制设备对象。

作为 FAT 文件系统的控制设备对象，一个很重要的任务是识别 FAT 格式的卷，即处理 IRP_MJ_FILE_SYSTEM_CONTROL 命令的 IRP_MN_MOUNT_VOLUME 子命令。Fastfat.sys 的 FatMountVolume 函数负责识别并挂载一个 FAT 卷。它先调用 IoCreateDevice 创建一个设备对象，该设备对象附带一个扩展部分，见 VOLUME_DEVICE_OBJECT 数据结构的定义，扩展部分的最后是 FAT 文件系统的 VCB 对象。然后，调用 FatInitializeVcb 函数以初始化该 VCB 对象。FatInitializeVcb 函数为访问卷对象建立起缓存环境。接着，FatMountVolume 函数读入引导扇区，并根据引导扇区的内容来检查该卷是否确实为 FAT 卷，若未通过检查，则释放缓存的卷文件，删除 VCB，返回 STATUS_UNRECOGNIZED_VOLUME 作为当前 I/O 请求的完成码。在通过了 FAT 卷的检查以后，FatMountVolume 再判断该卷是否已经被挂载了，若已被挂载，则执行重挂载操作，包括重新初始化卷文件、

检查脏位、与老的 VCB 建立连接，并完成此 I/O 请求；否则表明这是一次新的挂载操作，于是创建根目录的 DCB，并返回 STATUS_SUCCESS。

其次，我们来看一下 fastfat 驱动程序为 FAT 文件系统设备对象提供的支持，这里仅关心 IRP_MJ_CREATE 和 IRP_MJ_READ/IRP_MJ_WRITE 请求的处理过程。针对文件系统设备对象的 IRP_MJ_CREATE 请求是在 FatCommonCreate 函数中被处理的，它利用 FAT 文件系统的目录层次结构，从指定的一个相对目录或者根目录开始，层层递进，直至解释掉最长匹配的前缀。最后，如果这是一个创建目录的请求，则调用子函数 FatCreateNewDirectory 来完成创建工作；否则，调用另一个子函数 FatCreateNewFile 来创建一个文件。关于创建文件或目录的完整处理过程，请参考 fastfat 驱动程序的 create.c 文件中的代码。

IRP_MJ_READ 和 IRP_MJ_WRITE 请求分别在 FatFsdRead 和 FatFsdWrite 函数中被处理，它们又进一步调用 FatCommonRead、FatCommonWrite 或 FatPagingFileIo 函数，其代码位于 read.c、write.c 以及 deviosup.c 文件中。Fastfat 利用 FsRtl 提供的 MCB 支持来管理从 VBO（Virtual Byte Offset）到 LBO（Logical Byte Offset）的映射，每个文件或目录的 FCB/DCB 都包含一个类型为 LARGE_MCB 的 Mcb 成员。在处理 I/O 读写请求时，fastfat 使用缓存管理器来提高性能和简化 I/O 操作。另外，fastfat 在处理 IRP_MJ_READ 请求时，如果发现当前线程内核栈的可用空间已经低于一个阈值（在 Intel x86 平台上为 0xe00），则利用 FsRtl 提供的栈溢出保护机制（利用内核队列对象 KQUEUE 来实现，参见 5.4.2 节），将实际的读操作交给 FsRtl 中的辅助线程来完成，以避免当前线程的内核栈不够用。

Windows DDK 提供的 fastfat 驱动程序例子代码可以编译和运行。基于 fastfat 的源代码，读者可以在调试环境中观察 fastfat 的行为，以进一步理解 Windows 文件系统的特点。

7.4.5 NTFS 文件系统

NTFS 文件系统是 Windows 的原生文件系统格式。在 Windows NT 的设计之初，作为 MS-DOS 和 Windows 3.0/3.1 文件系统的 FAT 文件系统，显然不足以充当在市场上与 OS/2 和 UNIX 操作系统展开竞争的支撑文件系统，因此，Windows NT 有必要设计一个新的文件系统规范，以迎合 PC 软硬件长期发展的需要。以下是 NTFS 的设计目标：

- 安全性（security）和访问控制。文件系统中的文件和目录都是受保护的对象，以避免遭受未授权用户的访问。并且，文件系统的安全和访问控制机制直接集成在 Windows 内核本身的安全机制中，每个文件和目录的安全描述符都作为该文件或目录的元数据（或属性数据）存储在磁盘上。
- 可靠性（reliability）。文件系统可以从意外失败中恢复，防止文件系统元数据与文件

数据之间出现不一致性。NTFS 文件系统采用“事务（atomic transaction）”机制来确保一些关键磁盘操作的原子性。

- 可支持的卷容量足够大。在设计 NTFS 时，当时可使用的 FAT16 文件系统最大支持 4 GB 大小的卷（或分区）。NTFS 在这方面的目标是，支持足够大的磁盘卷，能支持持续增长的硬盘设备，以及 RAID 阵列。
- 长文件名。NTFS 从设计之初就摆脱了 MS-DOS 8.3 文件命名规范的限制，其文件名可长达 255 个字符，可以包含 UNICODE 字符、内嵌空格和句点号。

实际上，NTFS 文件系统伴随着 Windows 操作系统在 20 年的发展历程中，正变得越来越强大、健壮。自 Windows 2000 以后，除非硬件故障，Windows 操作系统已很少再发生“因为系统崩溃而导致文件系统被破坏，从而系统无法再引导”的情形，而这样的情形在早年“以 FAT 作为系统分区格式”的 Windows 95/98 时代屡有发生。NTFS 除了以上提到的这些设计目标均已实现以外，还融入了更多的高级特性，包括单个文件的多数据流支持、统一的索引机制、动态的坏簇重新映射、硬链接和交接（junction）、支持压缩文件和稀疏文件、变化日志（change journal）、针对单个用户的卷配额管理、链接跟踪、支持文件和目录的加密、对 POSIX 标准的支持，以及对只读卷的支持。关于 NTFS 这些特性的描述，读者可参考相关资料[WIN-INTERNALS]。

NTFS 是 Microsoft 私有的文件系统格式，尽管格式文档和 NTFS 驱动程序（ntfs.sys）的源代码并未公开，但 Microsoft 还是提供了较为全面的文档，阐述了 NTFS 文件系统的工作机理以及各种特性的实现原理[NTFS][WIN-INTERNALS]。NTFS 的一个重要设计思想是，卷上存储的所有数据都包含在文件中，包括用于管理文件数据流所涉及的各种数据结构、记录卷分配状态的位图等，甚至引导部分的数据也通过文件来管理。这种“一切皆文件”做法的好处是，文件系统很容易定位和维护这些数据，而且，每个文件都可以通过系统的安全描述符来加以保护。

NTFS 文件系统的规范使用 64 位簇编号，但是 NTFS 驱动程序限制了簇编号可用 32 位整数来表达，并且 NTFS 支持的簇大小为 512 B、1 KB、2 KB 或 4 KB。所以，虽然理论上 NTFS 可支持 2^{64}×4 KB=2^{76} B 大小的卷，但实际上它最多支持 2^{32}×4 KB =2^{44} B=16 TB 大小的卷。下面我们先来看 NTFS 的卷结构，如图 7.25 所示。

NTFS 引导扇区	主文件表(MFT)	文件数据	主文件表(MFT)的副本

图 7.25 使用 NTFS 文件系统的卷格式

NTFS 引导扇区的结构类似于 FAT 文件系统中的引导扇区，它位于卷的第一个逻辑扇区，

其中包含了有关该卷布局结构和文件系统结构的各种信息，以及引导时的启动代码。图 7.26 显示了 NTFS 文件系统的引导扇区格式。前三个字节为一条跳转指令，跳转到后面的启动代码部分。启动代码从第 84（0x54）个字节开始。第 3~10 字节为标志串，其中前 4 个字符为“NTFS”，后 4 个字符为空格。第 11~35 字节为 NTFS 文件系统的 BIOS 参数块（BPB），随后的第 36~83 字节为扩展的 BPB 区域。引导扇区的最后两个字节为“55”和“aa”。

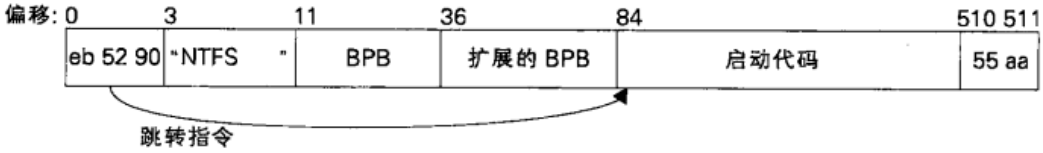


图 7.26 NTFS 文件系统的引导扇区格式

引导扇区的 BPB 和扩展 BPB 区域描述了卷的信息以及文件系统的结构信息，其中包括每个扇区有多少字节、每簇有多少扇区、介质类型、总共有多少扇区（64 位）、主文件表（$Mft）的逻辑簇编号（64 位）、主文件表副本（$MftMirr）的逻辑簇编号（64 位）、每个 MFT 记录的大小、每个索引缓冲区（index buffer）的大小，以及卷的序列号。关于这部分区域的特定格式，请参考相关资料[NTFS]。

NTFS 以文件的形式来描述所有的数据，它的主文件表（MFT，Master File Table）包含了文件的信息。MFT 本质上是一个关系数据库，对于 NTFS 卷上的任何一个文件，在 MFT 中都至少有一条记录。MFT 本身也是一个文件，它的第一条记录描述它自己。NTFS 文件系统在被格式化时，还会创建一些元数据文件。MFT 中前 16 条记录保留用于描述这些元数据文件。表 7.4 列出了这些元数据文件以及它们的用途。

表 7.4 NTFS 文件系统中的元数据文件

NTFS 文件名	MFT 中记录号	文件名称	用途
$Mft	0	主文件表	为卷上的每个文件和目录保存至少一个文件记录
$MftMirr	1	主文件表副本	主文件表的拷贝，包含$Mft 的前 4 个记录或第一簇。即使 MFT 的扇区被损坏，文件系统仍然可以访问
$LogFile	2	日志文件	包含了日志信息，以便于 NTFS 卷能快速恢复
$Volume	3	卷	包含了有关卷的信息，例如卷标签、卷版本等
$AttrDef	4	属性定义	列出了属性名称、属性编号，以及属性描述
.	5	根目录	根目录
$Bitmap	6	簇位图	代表卷的使用情况，指示了空闲的簇
$Boot	7	引导扇区	包括 BPB，如果这是一个可引导卷，则还包含引导时的启动加载代码

续表

NTFS 文件名	MFT 中记录号	文件名称	用途
$BadClus	8	坏簇文件	包含卷的坏簇
$Secure	9	安全文件	包含了该卷上所有文件的唯一安全描述符
$Upcase	10	大写字母表	用于将小写字符转换成 UNICODE 大写字符
$Extend	11	NTFS 扩展文件	用于各种扩展信息，例如配额信息、重解析点和对象标识符
保留	12~15		保留将来使用

为了存储一个文件，NTFS 并非简单地存储一个数据流，而是存储一组“属性/值”对。每个文件都有一对特殊的“属性/值”，包含了该文件的数据，此属性被称为未命名的数据属性（unnamed data attribute）。如果一个文件足够小，那么，它的所有属性可能都包含在 MFT 的文件记录中。对于大的文件，MFT 中仅包含一部分属性，其他的属性存储在 NTFS 卷的文件数据区部分。存放在 MFT 中的属性称为驻留的属性（resident attribute），例如像文件名和时间戳这样的属性总是驻留的。非驻留的属性（nonresident attribute）需要占据文件数据区的簇。尽管这些属性存储在非 MFT 的簇中，但是在 MFT 中仍然包含相应的线索，以便文件系统驱动程序能够通过 MFT 的文件记录访问到。表 7.5 列出了 NTFS 文件系统预定义的属性类型。

表 7.5 NTFS 文件系统预定义的属性类型

属性类型	属性名称	说明
标准信息	$STANDARD_INFORMATION	标准的信息，例如访问模式、时间戳和链接至该文件的数量
属性列表	$ATTRIBUTE_LIST	不在 MFT 记录中的属性的位置
文件名	$FILE_NAME	文件的名称，可以有多个文件名属性
对象标识符	$OBJECT_ID	文件或目录的 64 字节标识符，其中低 16 字节（128 位）是卷内唯一的标识符
安全描述符	$SECURITY_DESCRIPTOR	仅为兼容以前的 NTFS 版本。当前版本的 NTFS 将安全描述符存储在$Secure 元文件中
卷信息	$VOLUME_INFORMATION，$VOLUME_NAME	仅供$Volume 元数据文件使用，保存了卷的版本和标签信息
数据	$DATA	文件的数据，即未命名的数据属性
索引根	$INDEX_ROOT	用于目录或其他索引结构
索引分配	$INDEX_ALLOCATION	用于大目录或其他索引结构，实现 B-树结构
位图	$BITMAP	用于大目录或其他索引结构，簇位图索引
重解析点	$REPARSE_POINT	文件的重解析点数据。用于支持 NTFS 的交接（junction）和挂载点特性
扩展属性	$EA，$EA_INFORMATION	已不再使用。仅为了与 OS/2 程序保持兼容

并不是所有的文件都包含以上这些属性，一个典型的小文件包含$STANDARD_INFORMATION、$FILE_NAME和$DATA属性。有些属性总是驻留的，比如$STANDARD_INFORMATION 和$FILE_NAME，因而 NTFS 驱动程序在访问这些属性时，直接从 MFT 文件记录就可以获得数据。然而，当文件或目录中所有的属性无法被存放在一个 MFT 文件记录中时，NTFS 驱动程序将为某些属性分配专门的簇，这些簇不在 MFT 中，而是位于 NTFS 卷的数据区域中。NTFS 数据区域中的簇是按照**行串**（run）的概念来分配的。这里的**行串**是指在属性中连续的一段字节范围在磁盘上也占据一段连续的逻辑簇，所以行串的概念可以用“<VCN, LCN, LengthOfRun>”来表示，这里 VCN 和 LCN 代表了从虚拟簇号至逻辑簇号的映射关系，而 LengthOfRun 代表行串的长度（以簇为单位）。如果一个属性是非驻留的，那么，在 MFT 文件记录中该属性的属性头就指明了一个或多个行串。因此，当 NTFS 驱动程序访问一个文件时，根据 MFT 文件记录中数据属性头的行串信息，就可以访问整个数据流。图 7.27 显示了一个非驻留数据属性的 2 个行串。

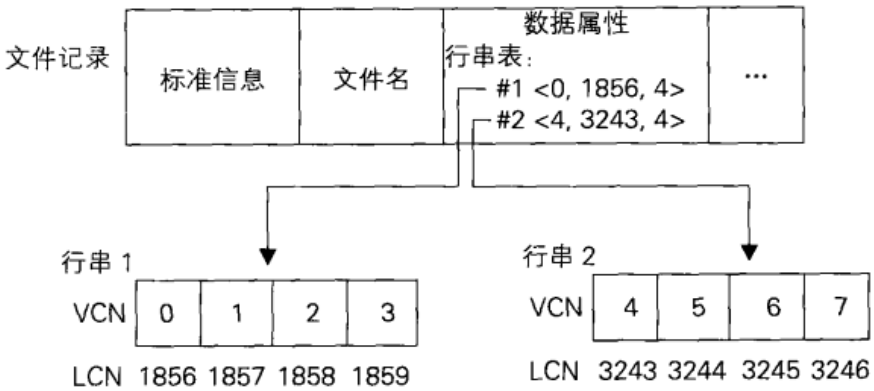


图 7.27 非驻留数据属性的行串示意图

对于非数据属性，如果在 MFT 文件记录中无足够的空间来存放，也可以通过这种方式存放到行串中。另一种可能是，一个文件有太多的属性，导致在一个 MFT 文件记录中放不下（MFT 文件记录的默认大小为 1 KB），在这种情况下，它可以使用额外的 MFT 文件记录来存放更多的属性（或非驻留属性的头）。属性$ATTRIBUTE_LIST 即用于此目的，它包含了该文件的每个属性的名称和类型代码，以及该属性所在 MFT 文件记录的编号（称为**文件引用**，见下一段落中的解释）。有时候，文件不停地增长，或者 NTFS 卷数据区域变碎，因而文件的属性包含了大量的行串，以至于这些属性的属性头本身变大，在这样的情况下，单个 MFT 文件记录可能无法存放所有的属性或属性头，此时 NTFS 可为该文件引入$ATTRIBUTE_LIST 属性，允许分配更多的文件记录来存放该文件的属性。在这种情况下，第一个 MFT 文件记录被称为基本文件记录（base file record）。

NTFS 使用 64 位的**文件引用**（file reference）来标识一个文件，其中低 48 位是指该文件在 MFT 中的文件记录的位置减 1（或基本文件记录的位置减 1），高 16 位则随着每次一个 MFT 文件记录位置被重新使用而增加 1，相当于此位置处的文件记录的版本号。NTFS 可以利用此版本号来执行一致性检查。

MFT 中的记录既可能是文件记录，也可能是目录记录。在 NTFS 中，目录是一组文件名和相应的文件引用按照一种便于快速访问的方式组织在一起而构成的 MFT 记录。表 7.5 中的$INDEX_ROOT、$INDEX_ALLOCATION 和$BITMAP 属性被用于建立起这一组织方式。对于小的目录，也就是说，目录中的文件和子目录数量相对比较小，则这些文件和子目录的文件引用信息被直接存放在$INDEX_ROOT 属性中。但是，当目录很大时，用这种方式来查找或列举其中的文件或子目录，显然比较低效。在这种情况下，NTFS 使用一个 B-树结构来存放文件和子目录信息，$INDEX_ROOT 属性存放的是 B-树的根。图 7.28 显示了一个目录记录的结构图。

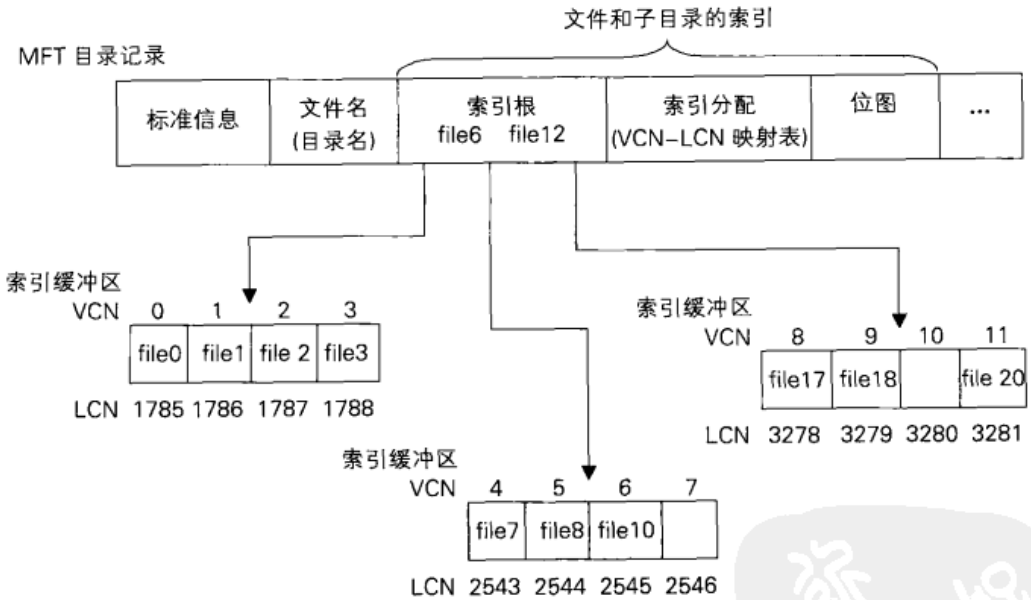


图 7.28 MFT 中针对大目录的目录记录

索引分配属性$INDEX_ALLOCATION 记录了 B-树的每个节点（称为索引缓冲区）的行串信息，即从 VCN 到 LCN 的映射关系，因而可以由 VCN 方便地换算出 LCN，以确定一个行串在磁盘卷上的位置。位图属性记录了在索引缓冲区中哪些 VCN 已经被使用了，哪些还是空闲的。在 NTFS 实际使用时，每个簇可以存放多个文件项，而并非如图 7.28 中所示每个簇只存放一个文件项。

B-树中的每一项除了包含文件名或目录名以外，还包含相应的 MFT 记录的文件引用，以及该文件或目录的时间戳和大小。因此，当 NTFS 文件系统接收到列举一个目录的请求时，可以遍历该目录的 B-树，以最少的磁盘访问次数，完成目录列举请求。这正是利用 B-树来组织目录项的优势。关于 B-树的结构和算法，读者可以参考相关资料[ALG]。

MTF 文件记录的$INDEX_ROOT、$INDEX_ALLOCATION 和$BITMAP 属性构建的是一般性的 B-树。它除了用于存放大目录中的文件或子目录信息，也可以用做其他的索引结构，例如，NTFS 也使用此 B-树来存放对象 ID、配额和安全信息等。

理解了 NTFS 文件系统在磁盘上的存储结构以后，现在我们从 NTFS 文件系统驱动程序的角度来看一看关键的数据结构，即 NTFS 文件系统的内存结构。图 7.29 显示了 NTFS 驱动程序内部数据结构之间的关系。NTFS 文件系统允许每个文件有多个流，前面提到的 $DATA 是未命名的数据流，此外每个文件还可以有命名的数据流。因此，文件对象 FILE_OBJECT 中的 FsContext 域指向 NTFS 文件的 SCB（Stream Control Block），而文件的 FCB（File Control Block）有一个链表成员，将已经打开的流链接起来；FsContext2 域仍然指向一个 CCB（Context Control Block）；Vpb 域指向卷挂载时传递进来的 VPB 对象。如图中所示，VCB、FCB、SCB、CCB 这些数据结构形成了 NTFS 文件系统的内存结构。

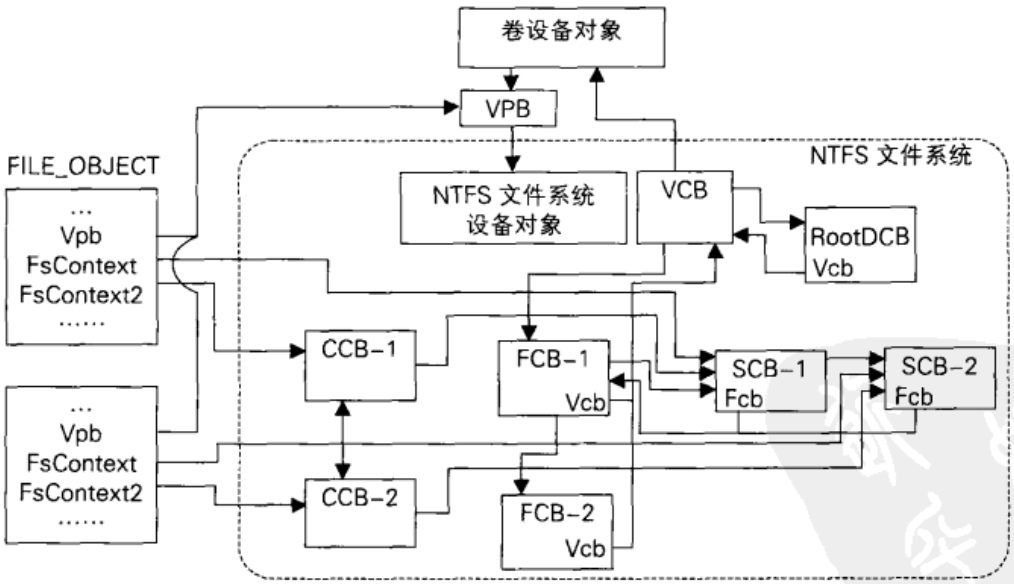


图 7.29 NTFS 驱动程序中的关键数据结构及其关系

在 NTFS 驱动程序中，DCB（Directory Control Block）与 FCB 相同，这符合 NTFS "一切皆文件"的思想。VCB 包含一张表，其中的元素指向当前已打开文件的 FCB；VCB 也包含根目录的 FCB（即图 7.29 中的 RootDCB）。FCB 包含了 SCB 链，而每个 SCB 又

反过来指向它所属的 FCB。每个 FCB 也指向它所属文件系统的 VCB。另外，CCB 总是与某一个 SCB 相关联，它指向此 SCB。VCB 也包含了一些常用文件的数据属性的 SCB，尤其是元数据文件中的数据属性，例如$Mft，以方便各种管理任务。

与 FAT 文件系统相比较，引入了 SCB 以后更能够显现出文件系统中的文件与 I/O 管理器中的文件对象（FILE_OBJECT）实际上是完全不同的概念，虽然这两者都是我们所熟知的“文件”。当 NTFS 文件系统驱动程序接收到一个读写文件的 I/O 请求时，通过 FILE_OBJECT 的 FsContext 域可以访问到目标文件中数据属性的 SCB，此 SCB 又指向所属文件的 FCB，而 FCB 含有该文件在 MFT 中的文件引用，从而可以通过它的 VCB 指针得到它的文件记录。因此，NTFS 文件系统驱动程序在接收到 I/O 请求时，只须沿着这样的几层引用关系，如图 7.30 所示，就可以很快地获得目标数据流的 VCN-LCN 映射关系，从而访问磁盘上的数据。

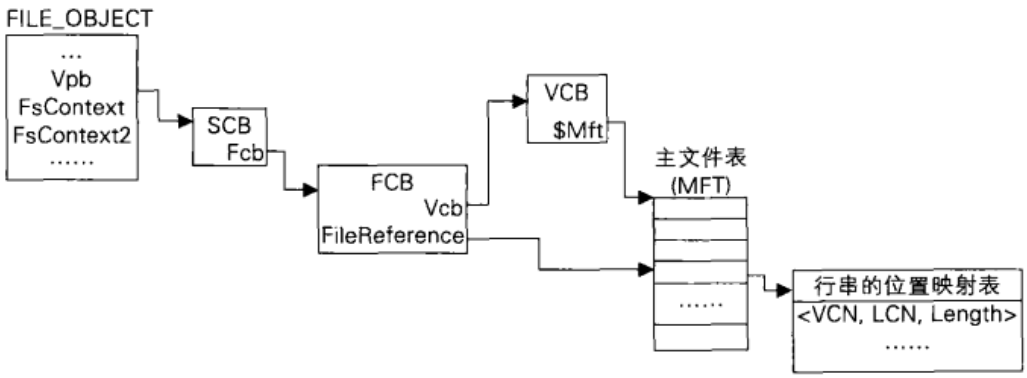


图 7.30 NTFS 访问磁盘数据的路径

以上介绍了 NTFS 文件系统的磁盘结构和内存结构，其灵活性和扩展性使得 NTFS 不仅实现了本节前文提到的设计目标，而且随着 Windows 的发展，实现了更多的高级特性。本节最后列出 NTFS 文件系统中的高级特性以及所涉及的一些关键技术：

- 多数据流。NTFS 的磁盘结构和内存结构都提供了多数据流的支持，这使得 NTFS 作为 Windows 的原生文件系统，能兼容更多其他的操作系统，以及赋予应用系统更强的存储能力。
- 统一的索引机制。正如在本节前面介绍 NTFS 目录结构时所讲，NTFS 为存储目录项而设计的、以 B-树为基础的索引机制也同样适用于其他的索引结构，例如整个文件系统范围的对象 ID 和安全描述符信息也使用了这一索引机制。
- 变化日志。NTFS 允许以变化日志的形式，使应用程序可以监视一个卷中的文件和目录的变化。

- 重解析点是 I/O 管理器提供的一种机制，NTFS 利用此机制，允许文件或目录关联一个称为重解析数据(reparse data)的数据块。在此基础上，NTFS 支持硬链接(hard link)和符号链接（ symbolic link，也称为交接或 junction ）特性。
- 安全性和访问控制。在 NTFS 中，文件和目录的安全描述符统一保存在一个元数据文件中，以方便实施各种安全控制。
- 针对单个用户的卷配额。NTFS 根据一个目录或文件的所有者来跟踪一个用户的空间使用量。
- 压缩文件和稀疏文件。NTFS 文件系统在处理一个文件属性的行串时，允许选择压缩原始数据从而减少存储空间使用量，或者不为稀疏文件中的零数据分配实际的簇，从而达到节省存储空间的目的。
- 加密。NTFS 包含一个称为 EFS（ Encrypting File System，加密文件系统 ）的内部服务，使得文件的数据都是经过密码学算法进行加密的，只有有权查看该文件的用户才可以解读其中的数据。
- 恢复能力。NTFS 使用事务机制来实现其恢复能力，它保证文件系统的数据可以被快速地恢复，但是不保证用户数据的可恢复性。NTFS 内置了一个称为 LFS（ Log File Service，日志文件服务 ）的服务来实现关键数据的事务特性。

关于以上这些特性以及实现技术的细节，读者可以参考相关资料[WIN-INTERNALS]。

7.5 本章总结

本章着重讨论了 Windows 存储管理体系中的三个核心组成部分：缓存管理器、卷的管理以及文件系统。缓存管理器与内存管理器、I/O 管理器和文件系统相互分工与合作，它既是内存管理器的延伸，也为文件系统提供了统一的缓存机制。本章结合 WRK 中的缓存管理器源代码，介绍了缓存管理器的内部数据管理及其数据访问接口，还讨论了缓存管理器的预读和延迟写机制。卷管理是 Windows 的存储栈中承上启下的一个重要组件，卷内部的扇区管理由文件系统驱动程序来负责，而卷的物理存储则由磁盘管理设备栈来完成。卷的管理与磁盘分区的管理不完全一致，卷管理器提供了卷的名称解析、卷快照等功能。本章还讨论了卷的挂载，以及它与分区管理器和文件系统之间的交互过程。

文件系统是现代操作系统中的重要组成部分，它使外部存储设备以一种能直接为应用程序理解的方式呈现出来，而且使它所定义的由目录和文件构成的层次结构，与操作系统

的全局名字空间以及文件对象的概念有机地结合在一起。本章讨论了文件系统的驱动程序结构，以及内核提供的 RAW 文件系统和 FsRtl 运行库，还介绍了 Windows 支持的两种文件系统过滤机制：一种是可直接附载于文件系统设备栈的过滤驱动程序，另一种是由文件系统过滤管理器（fltMgr）来统一管理的小过滤驱动程序。最后，本章还介绍了 Windows 平台上最常用的两种文件系统：FAT 和 NTFS。

针对本章所介绍的内容，尽管 WRK 并没有提供所有的代码，但是，通过 WRK 中已有的代码(包括缓存管理器、I/O 管理器、RAW 文件系统和 FsRtl 运行库等)，以及 Windows DDK 中提供的文档和例子驱动程序（例如 fastfat、cdfs、filespy 和 minispy 等），我们仍然可以全面地理解 Windowos 存储系统的原理与实现。

PDG

8

Windows 系统服务

在 Windows 的系统结构中，内核提供的服务都通过 ntdll.dll 模块被应用程序使用。Windows 应用程序调用一组系统 DLL 中的 API 函数，间接地通过 ntdll.dll 中的存根函数来调用内核提供的系统服务。本章我们首先讨论 Windows 系统服务的基本原理，包括 Intel x86 的模式切换过程和 Windows 的系统服务分发机制。然后讨论三个重要的系统服务：本地过程调用（LPC，Local Procedure Call）、命名管道和邮件槽服务，它们都可被用于跨进程的通信。最后介绍一个显示系统服务表的工具 SDTViewer。

8.1 Windows 系统服务原理

Windows 内核中的执行体层暴露了大量的功能供应用程序使用，那么，应用程序如何调用这些功能呢？譬如，NtCreateFile 是 Windows 内核中的“创建文件”服务例程，它运行在处理器的内核模式下，而应用程序的代码运行在用户模式下，所以，应用程序为了调用此“创建文件”服务，必须将处理器从用户模式切换到内核模式下。当然，模式切换工作并不需要由应用程序自己来完成，Windows 提供了一个系统模块 ntdll.dll，已经实现了所有系统服务的模式切换工作。这一模式切换依赖于硬件体系结构，本节我们将讨论 Windows 在 Intel x86 体系结构下的模式切换过程，以及 Windows 在此基础上实现的系统服务分发机制。

8.1.1 Intel x86 的用户模式-内核模式切换

Intel x86 处理器提供了 4 种处理器执行模式，称为 Ring0~Ring3，其中，Ring0 为特权模式，有些指令只能在特权模式下使用。Windows 仅使用两种模式：Ring0 和 Ring3，

分别称为内核模式和用户模式。另外，Intel x86 处理器还提供了“门（gate）”机制允许可执行代码从 Ring3 进入 Ring0 模式，以及从 Ring0 返回 Ring3 模式。Intel x86 支持 4 种“门”描述符：调用门（call gate）、陷阱门（trap gate）、中断门（interrupt gate）和任务门（task gate）[INTEL-253668]。Windows 使用中断门来实现模式切换，其中断号为 0x2e。因此，简单而言，当应用程序在用户模式下运行时，它通过“int 2e”指令，切换到内核中，IDT（中断描述符表）的 0x2e 表项指定了内核系统服务的入口点；当内核模式代码完成了指定的系统服务时，它通过“iret”或“iretd”指令，返回到用户模式代码。

从用户模式切换到内核模式的另一个途径是系统异常。当用户模式代码引发异常时，处理器切换至内核模式，并执行异常处理例程。关于在 Intel x86 体系结构下 Windows 的中断和异常处理机制，参见 5.2 节。这一小节仅讨论 Intel x86 处理器在执行模式切换时的代码段和栈的切换过程。

我们首先看一下“int 2e”指令的跳转流程，如图 8.1 所示。Intel x86 处理器在 IDT 中查找 2e 表项，IDTEntry 包含了一个段选择符和中断例程的段内偏移，所以，处理器还需要在 GDT 中再查找一次表项，得到段选择符指定的段的虚拟基地址。段基地址加上中断例程偏移，最终得到中断例程的虚拟地址。

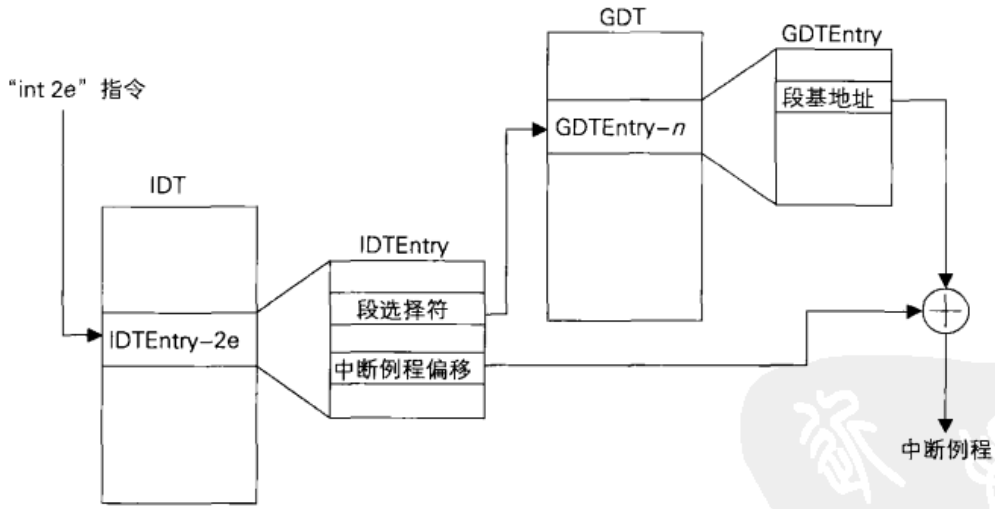


图 8.1 软件中断指令“int 2e”的跳转流程

在 Windows 中，IDT 的 2e 表项中的段选择符为 0x0008，中断例程偏移为 _KiSystemService 例程的地址，参见 5.2.1 节的介绍。这里段选择符 0x0008 指向第一个段，段基地址为 0x0，正是内核中使用的 cs 段，参见 4.2.1 节。因此，最终的中断服务例程为内核中的_KiSystemService 函数。

Intel x86 处理器在不同模式下使用不同的栈，用户模式代码使用用户栈，内核模式代码使用内核栈，因此，当一个线程通过“int 2e”指令从用户模式跳转到内核模式时，必然会伴随有栈的切换。图 8.2 显示了栈切换示意图。处理器在将控制权交给目标例程以前，首先将原来的 ss、esp、eflags、cs 和 eip 寄存器的值存放到内核栈中，并且让新的 esp 指向内核栈顶位置。

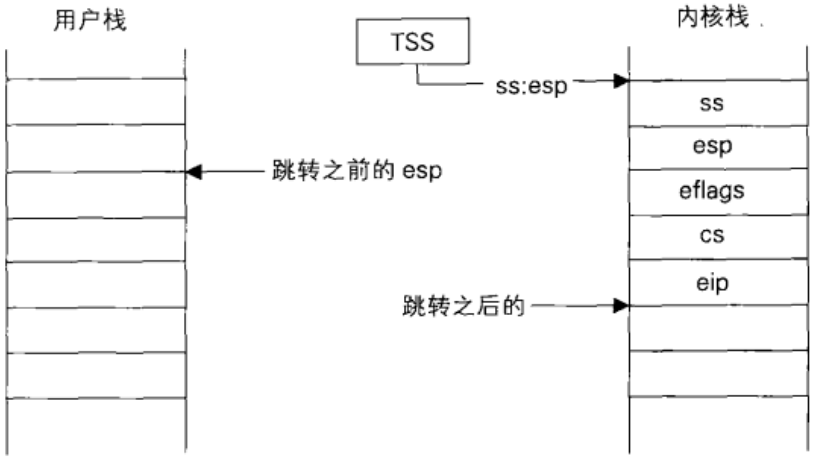


图 8.2 通过软件中断进入内核模式时发生的栈切换

读者可能会问一个问题：处理器在用户模式下执行，它如何知道内核栈的位置，即内核栈的 ss 和 esp？答案在于处理器的当前任务环境。每个处理器都总是在一个任务环境中运行，处理器的任务寄存器（task register）指向当前任务环境的 TSS（Task-State Segment），其中包含了每一特权级使用的栈，其中 Ring0 的 esp 位于 TSS+4 位置[INTEL-253668]。Windows 操作系统在每个处理器初始化时，会在 GDT 中为它构造一个 TSS 段，然后利用 ltr 指令，设置处理器的任务环境段，参见 base\ntos\ke\i386\newsysbg.asm 文件中的_KiSystemStartup 函数的代码。另外，Windows 在每次切换线程时，总是会设置好 TSS 中 Ring0 的 esp，使它指向当前线程的内核栈。读者可参考 base\ntos\ke\i386\ctxswap.asm 文件中的线程切换函数 SwapContext 的代码。

用户模式代码如果要向中断例程传递参数，可以使用寄存器或用户栈，Intel x86 处理器没有强加特别的规范。对于操作系统来说，唯一重要的是，触发中断的代码和处理中断的代码必须遵守同样的参数传递规则，比如在 Windows 中，中断处理代码在调用实际的系统服务例程以前，利用 edx 寄存器所指的栈位置信息，将用户栈中的参数信息拷贝到内核栈中，然后像普通的函数调用那样来调用内核例程。8.1.3 节将进一步介绍 Windows 系统服务的分发过程。

当内核模式下的中断例程完成了中断处理以后，它通过 iret/iretd 指令返回到用户模式

代码。Iret/iretd 指令从内核栈中弹出 eip、cs、eflags，以及 esp 和 ss，然后将控制权交给 eip 所指的用户模式代码。

从软件中断“int 2e”的工作过程，我们可以看出，尽管从用户模式进入内核模式只是通过一条指令来完成，但是，模式切换过程中涉及很多次内存访问，还有两次查表操作以及访问权限的检查，这导致模式切换的开销很大。从 Pentium II 处理器开始，Intel 引入一对新的指令 sysenter/sysexit，实现了快速的模式切换。接下来介绍 sysenter/sysexit 的工作原理。

在模式切换过程中，sysenter 要做的事情与“int 2e”指令类似，但是，它会尽可能地减少内存访问和权限检查的开销，其做法是，尽可能地避免内存访问，而通过处理器的内部寄存器来指定必要的信息。Sysenter 使用三个 MSR（Model Specific Register）寄存器来指定跳转目标地址和栈位置，如表 8.1 所示。操作系统可以在内核模式下通过 rdmsr/wrmsr 指令来设置这三个寄存器，当然必须在系统初始化时（第一次系统调用发生以前）完成。

表 8.1 sysenter/sysexit 指令用到的 MSR 寄存器

MSR 寄存器	MSR 地址	含义
IA32_SYSENTER_CS	174h	低16位值指定了特权级0的代码段和栈段的段选择符
IA32_SYSENTER_ESP	175h	内核栈指针的 32 位偏移
IA32_SYSENTER_EIP	176h	目标例程的 32 位偏移

Sysenter 指令的内部逻辑是：将 IA32_SYSENTER_CS 和 IA32_SYSENTER_EIP 分别装载到 cs 和 eip 寄存器中；将 IA32_SYSENTER_CS+8 和 IA32_SYSENTER_ESP 分别装载到 ss 和 esp 寄存器中；切换到特权级 0；清除 eflags 中的 VM 标志（虚拟 8086 模式）；执行目标例程。注意，sysenter 指令对于跳转目标的代码段和栈段有要求：首先，代码段和栈段在 GDT 中必须是相邻的，IA32_SYSENTER_CS 指向代码段的 GDT 表项，紧随其后的表项应该是栈段描述符；其次，代码段描述符指定的是一个基地址为 0、段范围可达 4 GB 的特权级 0 的段，具有执行和读访问权限；栈段描述符指定的也是一个基地址为 0、段范围可达 4 GB 的特权级 0 的段，但具有读写访问和向上扩展的权限。

再来看 sysexit 指令，它的内部逻辑是：将 IA32_SYSENTER_CS+16 装载到 cs 寄存器；将 edx 寄存器中的指针装载到 eip 寄存器中；将 IA32_SYSENTER_CS+24 装载到 ss 寄存器中；将 ecx 寄存器中的指针装载到 esp 寄存器中；切换到特权级 3；执行 eip 寄存器中指定的用户模式代码。类似地，sysexit 也要求 IA32_SYSENTER_CS+16 和 IA32_SYSENTER_CS+24 指定的用户模式代码段和栈段，即 GDT 中紧跟 sysenter 用到的两个段描述符之后的两个 GDT 表项，符合在特权级 3 下执行或读写方面的要求。

因此，MSR 寄存器 IA32_SYSENTER_CS 实际上指定了 GDT 中的 4 个段描述符符合上述 sysenter/sysexit 指令的要求。而且，用户模式代码在调用 sysenter 指令以前，必须将要返回的指令地址和栈指针值保存到 edx 和 ecx 寄存器中，否则，内核模式代码将来无法设置正确的值，以使 sysexit 还能回到用户模式代码原来的位置。另外，sysenter 指令并不负责将用户栈中的参数传递到内核栈中，参数的传递由用户模式代码和内核模式代码约定来完成，sysenter 仅提供跨模式跳转和栈切换的能力。图 8.3 显示了 sysenter/sysexit 指令的跳转示意图，sysenter 和 sysexit 指令旁边的粗箭头表示该指令的跳转依赖于这些寄存器。

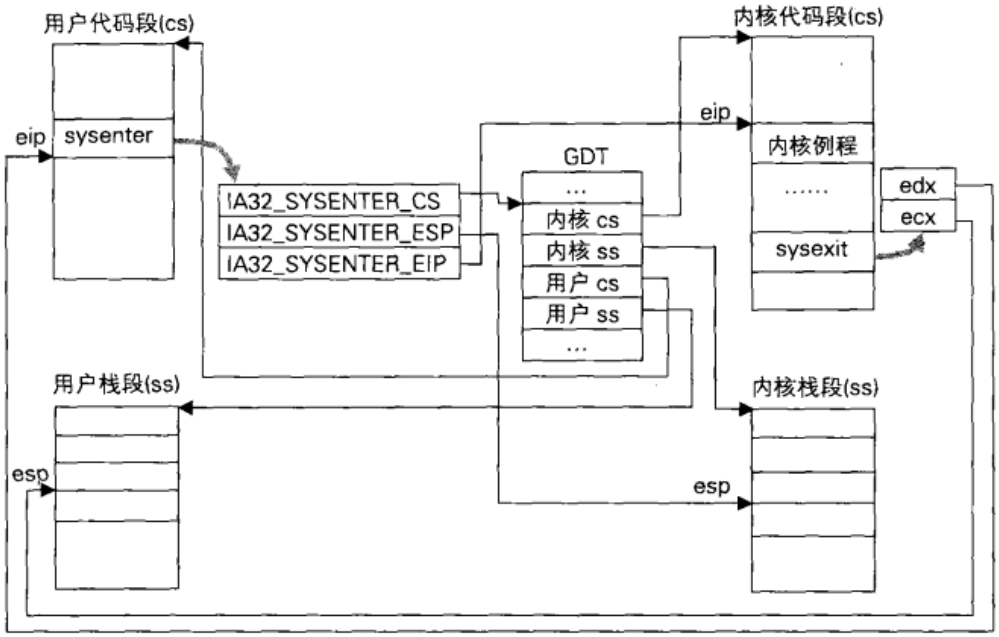


图 8.3 利用 sysenter/sysexit 实现模式切换和栈切换

从 sysexit 回到用户模式代码的地址和栈指针，即 sysexit 指令执行前的 edx 和 ecx 寄存器值，需要用户模式代码和内核例程来约定。也就是说，处理器在执行 sysenter 指令时并没有将现场信息保存在这两个寄存器中。所以，用户模式代码有责任将返回用户模式后的 eip 和 esp 通过某种方式传递到内核例程中，以便内核例程在发出 sysexit 指令前设置好 edx 和 ecx。

实际上，sysenter/sysexit 指令要求代码段和栈段必须指向满足相应的特权级保护要求的 0~4 GB 地址空间，即段的基地址为 0，大小可达 4 GB。Sysenter 指令在执行时，cs 寄存器被赋予 IA32_SYSENTER_CS 寄存器的值，但是，cs 寄存器内部的段描述符并非拷贝自 GDT 中的表项，而是硬编码的固定值，ss 寄存器也使用同样的处理方法。类似地，sysexit 指令在执行时，cs 和 ss 段寄存器虽然被赋予 IA32_SYSENTER_CS+16 或 IA32_SYSENTER_CS+24 的值，但是，内部的段描述符也是硬编码的固定值。所以 GDT 中的 4 个表项并未

真正被使用。关于这两条指令的详细信息，读者可以参考相关资料[INTEL-253667]。

因此，我们不难理解，sysenter 和 sysexit 只使用固定的值或者寄存器的值来完成跳转，从而达到快速切换代码段和栈段的目的。而且，使用固定内容的段描述符，省去了特权级检查的步骤，可进一步提升这两条指令的效率。对于频繁地进行模式切换的应用线程来说，使用 sysenter 指令比“int 2e”有显著的性能优势。

最后，操作系统如何知道一个处理器支持 sysenter/sysexit 与否呢？答案在于处理器的 cpuid 指令返回的特征信息。当通过 eax=1 来执行 cpuid 指令时，处理器的特征信息被存放在 ecx 和 edx 寄存器中，其中 edx 包含了一个 SEP 位（SysEnter/SysExit Present，第 11 位），该位指明了当前处理器是否支持 sysenter/sysexit 指令。因此，操作系统可以在初始化时，利用该位来决定应使用 sysenter/sysexit 还是“int 2e/iretd”作为模式切换的指令。下一小节我们将讨论 Windows 的做法。

8.1.2 Windows 的用户模式-内核模式切换

模式切换本质上是由处理器的体系结构来决定的，但是操作系统在处理器模式切换指令的基础上，需要正确地处理参数的传递，以及定义内核模式和用户模式的入口例程，从而使应用程序代码和内核代码可以无缝地一起工作。我们首先看一下在 Windows 中应用程序与 Windows 内核打交道的流程，如图 8.4 所示。

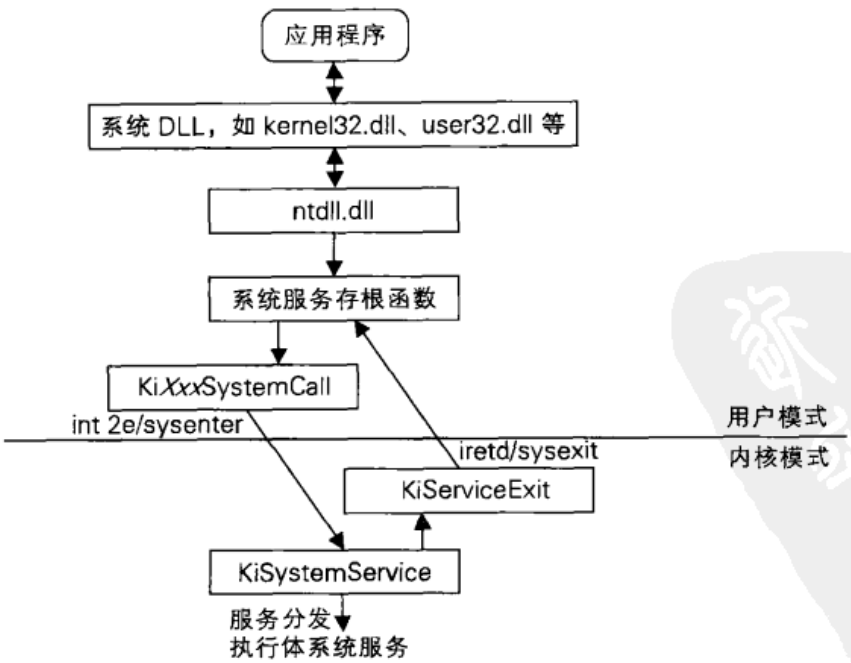


图 8.4 Windows 中用户模式和内核模式的交互流程

Windows 应用程序调用系统 DLL 中的函数，这是大量已经文档化的 API 函数；这些系统 DLL 函数可能又进一步调用 ntdll.dll 中的系统函数，这些系统函数要么是操作系统提供的支持函数，要么是一些系统服务存根（stub）。系统服务存根函数利用上一小节介绍的模式切换指令进入到内核模式，调用内核提供的系统服务来完成应用程序的请求。举例来说，Windows 应用程序调用 Windows API 函数 CreateFile 来创建文件，此 API 函数实际上是 CreateFileW，位于 kernel32.dll 模块中。CreateFileW 函数又进一步调用 ntdll.dll 中的 NtCreateFile 函数。Ntdll.dll 中的 NtCreateFile 函数是一个存根函数，它只是简单地将创建文件的请求转交给内核中的 NtCreateFile 函数。为了做到这一点，它通过 ntdll.dll 中的 KiIntSystemCall 或 KiFastSystemCall 函数执行“int 2e”或 sysenter 指令，以便切换到内核模式下，然后由内核模式的系统服务分发函数 KiSystemService 来调用 NtCreateFile 系统服务。待 NtCreateFile 系统服务执行完成以后，KiSystemService 调用 KiServiceExit 函数，最终通过 iretd 或 sysexit 指令返回到用户模式 ntdll.dll 模块中。

现在我们以此为例来展示此过程中的一些细节处理。首先，ntdll.dll 中的 NtCreateFile 存根函数的代码非常简单，在 Windows Server 2003 SP1 系统上，其汇编代码如下：

```
mov eax, 27h                                        ; 27h为NtCreateFile的系统服务号
mov edx, MM_SHARED_USER_DATA_VA+UsSystemCall        ; 见下文的解释
call [edx]
ret 2ch
```

为了说明以上代码中 edx 寄存器的内容，这里有必要特别解释一下，在 Windows 的虚拟地址空间管理中，系统空间有一个特殊的页面会被映射到每个进程地址空间中。也就是说，该页面是系统地址空间和进程地址空间共享的。它在系统地址空间的地址是 0xffdf0000，即宏定义 KI_USER_SHARED_DATA；在进程地址空间中的地址是 0x7ffe0000，即宏定义 MM_SHARED_USER_DATA_VA。该页面是在内存管理器初始化时分配的，全局变量 MmSharedUserDataPte 代表它的 PTE，参见 4.2.1 节。当一个进程在用户模式地址空间中访问该页面时，页面错误处理例程检测到此页面地址，并且直接使用 MmSharedUserDataPte 作为它的 PTE。参见 base\ntos\mm\pagfault.c 文件中 MiCheckVirtualAddress 函数的代码。

此共享页面记录了当前系统的一些关键状态信息，数据结构 KUSER_SHARED_DATA 定义了该页面内部的结构。其中两个成员与本节的讨论有关，即 SystemCall 和 SystemCallReturn 成员。这是两个函数地址，SystemCall 成员指示了从用户模式切换至内核模式的函数地址；SystemCallReturn 成员指示了从内核模式返回至用户模式的函数地址。因此，上面列出的汇编代码中的 edx 被赋予了 SystemCall 成员的值。

Windows 使用这种做法的一个好处是，系统在初始化时，根据当前处理器的特征信息，可以动态地选择使用 sysenter 还是“int 2e”作为模式切换指令。因而，同一份内核代码，无须任何修改或重编译，就既可以在支持 sysenter/sysexit 指令对的系统上运行，也可以在老式的系统上运行（这种软件特性称为多态，即 polymorphism，或动态绑定）。这两个成员的初始化是在 PspLookupKernelUserEntryPoints 函数中完成的，参见 base\ntos\ps\kulookup.c 文件中的代码。我们可以看到，当处理器支持快速系统调用时，SystemCall 成员指向 KiFastSystemCall 函数，SystemCallReturn 成员指向 KiFastSystemCallRet 函数；否则，SystemCall 成员指向 KiIntSystemCall 函数。

因此，ntdll.dll 中的 NtCreateFile 存根函数实际上调用了 KiFastSystemCall 或 KiIntSystemCall 函数，具体取决于当前处理器是否支持 sysenter/sysexit 指令对。以下是这两个函数的汇编代码：

```
KiIntSystemCall:
        lea edx, [esp+8]
        int 2e
        ret

KiFastSystemCall:
        mov edx, esp
        sysenter

KiFastSystemCallRet:
        ret
```

这里为说明方便，将 KiFastSystemCallRet 函数也列示了出来，事实上，该函数的代码是否紧跟在 KiFastSystemCall 函数之后并不重要，后面我们会看到 sysenter 指令是如何返回的。从以上代码可以看出，在切换至内核模式以前，edx 寄存器指向用户栈中的某个位置。显然，edx 的用意是向内核模式的系统服务函数传递有关参数的信息。接下来我们转到内核代码中，内核中哪个函数被执行呢？如果是由“int 2e”指令进入的，则 _KiSystemService 函数获得控制权，这是在 IDT 的 0x2e 项中指定的；如果是借由 sysenter 指令跳转过来的，则_KiFastCallEntry 函数获得控制权，这是由 MSR 寄存器 IA32_SYSENTER_EIP 指定的，参见 base\ntos\ke\i386\misc.c 中 KiLoadFastSyscallMachineSpecificRegisters 函数的代码。

_KiSystemService 和_KiFastCallEntry 函数的做法类似，它们首先建立一个陷阱帧（trap frame），参见 5.2.7 节关于陷阱帧的介绍。然后，这两个函数都转到一段公共的代码 _KiSystemServiceRepeat 处执行，见 base\ntos\ke\i386\trap.asm 文件中这两个函数的定义。在进入_KiSystemServiceRepeat 以前，这两个函数不仅在内核栈建立了陷阱帧，而且设置好了三个寄存器：eax 为系统服务号，edx 为调用者的当前栈指针，esi 为当前线程的地址。

从_KiSystemServiceRepeat 开始的代码是内核真正处理系统服务的代码，它利用当前线程所指的系统服务表，根据 eax 寄存器中指定的系统服务号，将该系统服务的参数从用户栈拷贝到内核栈，然后调用对应的系统服务例程。待系统服务例程返回以后，它转入_KiServiceExit 函数。

_KiServiceExit 函数在选择返回用户模式的指令方面，使用了一点技巧。它调用 DISPATCH_USER_APC 汇编宏来分发用户模式 APC，再调用 EXIT_ALL 汇编宏返回用户模式。在 EXIT_ALL 汇编宏的定义中，退出当前系统服务是由_KiSystemCallExitBranch 处的 jnz 指令来完成的，而该指令的目标地址是_KiSystemCallExit，_KiSystemCallExit 只是一条简单的 iretd 指令。实际上，如果当前系统支持快速系统调用，那么，这条指令将被修正为“jnz short _KiSystemCallExit2”。而_KiSystemCallExit2 所指的代码使用 sysexit 指令返回用户模式。这条指令的修正工作是在 KiEnableFastSyscallReturn 函数中完成的，该函数被 KiRestoreFastSyscallReturnState 调用，参见 base\ntos\ke\i386\misc.c 中的代码。以下是退出系统服务所执行的代码：

```
        test    dword ptr [esp+4], MODE_MASK
_KiSystemCallExitBranch:
          ; 若支持sysenter，下面指令的目标地址被修正为_KiSystemCallExit2
        jnz     short _KiSystemCallExit;

        ; 下面的指令用于返回内核模式，功能等同于iretd，但是比iretd快
        pop     edx                     ; 弹出eip
        pop     ecx                     ; 从栈中移除cs
        popfd                           ; 恢复eflags状态寄存器
        jmp     edx

_KiSystemCallExit:
        iretd                           ; 中断返回

_KiSystemCallExit2:
        test    dword ptr [esp+8], EFLAGS_TF ; 测试陷阱标志
        jne     short _KiSystemCallExit      ; 若是陷阱，则转到_KiSystemCallExit

        pop     edx                     ; 弹出eip，此EIP是_KiFastCallEntry填充的，见下文说明
        add     esp, 4                  ; 移除cs
        and     dword ptr [esp], NOT EFLAGS_INTERRUPT_MASK ; 禁止eflags中的中断标志
        popfd
        pop     ecx                     ; 弹出esp

        sti                             ; 恢复中断，因为sysexit指令不会恢复中断标志
        sysexit                         ; 返回用户模式
```

如果处理器不支持快速系统调用，则_KiSystemCallExitBranch 处的 jnz 指令不会被修正，于是当前线程通过“int 2e”指令返回用户模式。在这种情况下，用户模式代码的 cs 和 eip，以及用户栈的 ss 和 esp，都通过内核栈来传递，参考图 8.2 所示的内核栈结构。返

回到用户模式的指令正好是“int 2e”之后的指令，即 KiIntSystemCall 函数中的返回指令。所以，线程的控制流顺利地返回到 ntdll.dll 中的 NtCreateFile 函数中。

如果处理器支持快速系统调用，那么，_KiSystemCallExitBranch 处的跳转指令会将当前线程的控制流指引到_KiSystemCallExit2 处。注意，这里首先测试 eflags 标志寄存器中的 TF（陷阱）标志是必要的，因为，在 Pentium II 以后的 Windows 系统上，若应用程序绕过 KiFastSystemCall，而直接利用“int 2e”指令进入内核，那么，在此处将会被检测到，从而通过 iretd 返回，这样就不会发生进入内核模式和返回用户模式不对称的情形。代码最后的 sysexit 指令将控制流返回到由 edx 寄存器指定的用户模式地址处。这里有必要补充说明一下，_KiFastCallEntry 函数一开始已在栈中填充了目标例程，摘录此行代码如下：

```
push  dword ptr ds:[USER_SHARED_DATA+UsSystemCallReturn]     ; 填充返回地址
```

这里填充在栈中的返回地址是 KUSER_SHARED_DATA 共享页面中的 SystemCallReturn 成员，已被初始化为 KiFastSystemCallRet 函数的地址。所以，_KiSystemCallExit2 中的 sysexit 执行以后，线程的控制流返回到 KiFastSystemCallRet，该函数仅仅是一条 ret 语句，所以，控制流进一步返回到 ntdll.dll 中的 NtCreateFile 函数中。

至此，我们已经看到了应用程序在调用一个系统服务的 API 函数时，从用户模式进入内核模式，再回到用户模式的完整过程。Windows 利用 Intel x86 提供的中断和 sysenter/sysexit 指令对，巧妙地实现了快速、灵活的切换机制。图 8.5 总结了这一过程。

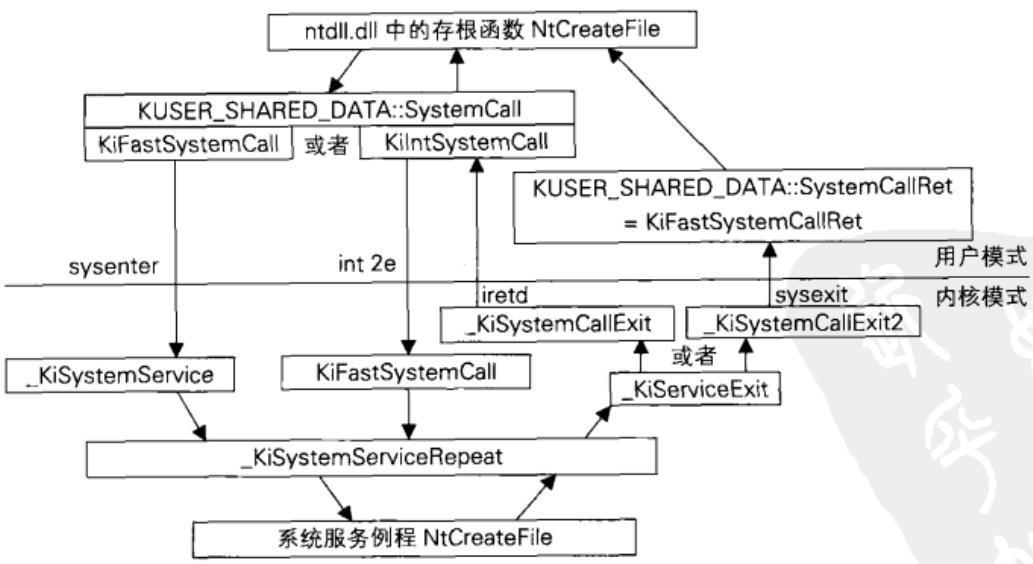


图 8.5 Windows 的模式切换全过程

8.1.3 Windows 中的系统服务分发

在上一小节我们看到 ntdll.dll 中的系统服务存根函数指定了一个系统服务号，比如 NtCreateFile 的系统服务号为 27h。在内核模式下，_KiSystemServiceRepeat 根据此系统服务号，知道该调用内核中哪个系统服务例程，以及从用户栈拷贝多少数据到内核栈中。这一过程被称为系统服务分发（System Service Dispatching）。

Windows 实现系统服务分发的关键是一个称为 SDT（Service Descriptor Table，服务描述符表）的表。Windows 支持多个 SDT，系统服务存根函数指定的系统服务号，即 eax 寄存器的值，包含了表的索引和表内的服务索引。eax 的低 12 位（即 0~11 位）指定了表内的系统服务索引，12~13 位指定了表的索引。内核全局变量 KeServiceDescriptorTable 是一个数组，每个元素指向一个 SDT，所以，系统服务分发代码根据系统服务号，首先定位到 KeServiceDescriptorTable 的元素，然后找到特定的系统服务项。仍然以 NtCreateFile 为例，服务号 0x27 说明 NtCreateFile 服务在 KeServiceDescriptorTable 数组第 0 个表的第 39 项。

以下是 WRK 中与系统服务描述符表相关的定义：

```
typedef struct _KSERVICE_TABLE_DESCRIPTOR {
    PULONG_PTR Base;
    PULONG Count;
    ULONG Limit;
    PUCHAR Number;
} KSERVICE_TABLE_DESCRIPTOR, *PKSERVICE_TABLE_DESCRIPTOR;

#define NUMBER_SERVICE_TABLES 2

KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTable[NUMBER_SERVICE_TABLES];
KSERVICE_TABLE_DESCRIPTOR KeServiceDescriptorTableShadow[NUMBER_SERVICE_TABLES];
```

从以上代码可以看出，Windows Server 2003 SP1 仅支持 2 个 SDT。另外，KeServiceDescriptorTableShadow 是一个内部数组，它的第 2 个元素，即 KeServiceDescriptorTableShadow[1]表项，专门用于 Windows 子系统，其他的表项与 KeServiceDescriptorTable 完全相同。KeServiceDescriptorTable[1]元素不使用。因此，在 Windows Server 2003 SP1 的情况下，Windows 内核的系统服务号范围为 0x00000000~0x00000fff；而 Windows 子系统的系统服务号范围为 0x00001000~0x00001fff。至于系统在分发服务时如何确定该使用哪个 SDT 数组，是由每个线程的属性决定的，本节后文将会进一步介绍。

在 KSERVICE_TABLE_DESCRIPTOR 数据结构描述的每一个 SDT 中，Base 域指向一个由系统服务例程构成的 32 位地址数组；Count 域是一个指针成员，指向一个计数器数组，数组中的每个元素记录了相应的系统服务被调用的次数，此域仅用于内核的调试版

本；Limit 域记录了该 SDT 中系统服务的数量；Number 域也是一个指针，指向一个字节数组，数组中的每个元素对应于一个系统服务所需要传递的参数区的长度。内核入口处 _KiSystemServiceRepeat 的代码，根据 Number 成员数组中的字节值来确定应该从用户栈拷贝多少数据到内核栈中。

系统使用的基本 SDT，即 KeServiceDescriptorTable[0]，是在 KiInitSystem 函数中被初始化的，该 SDT 同时也被复制到 KeServiceDescriptorTableShadow[0]。此 SDT 的原始数据分别来自于内部变量 KiServiceTable、KiServiceLimit 和 KiArgumentTable，其中 Count 成员被初始化为 NULL，参见 base\ntos\ke\kiinit.c 文件中 KiInitSystem 函数的代码。如果当前内核是调试版本，则执行体的初始化函数 ExpInitializeExecutive 分配该 SDT 的 Count 数组成员，以便记录每个系统服务被调用的次数。表 8.2 列出了在 Windows Server 2003 SP1 系统上此 SDT 中的系统服务（共 296 个，读者可以参考 base\ntos\ke\i386\systable.asm 中的 TABLE_ENTRY 项）。

表 8.2 Windows Server 2003 SP1 的基本 SDT 中的系统服务

编号	系统服务	编号	系统服务
0	NtAcceptConnectPort	148	NtQueryDebugFilterState
1	NtAccessCheck	149	NtQueryDefaultLocale
2	NtAccessCheckAndAuditAlarm	150	NtQueryDefaultUILanguage
3	NtAccessCheckByType	151	NtQueryDirectoryFile
4	NtAccessCheckByTypeAndAuditAlarm	152	NtQueryDirectoryObject
5	NtAccessCheckByTypeResultList	153	NtQueryDriverEntryOrder*
6	NtAccessCheckByTypeResultListAnd-AuditAlarm	154	NtQueryEaFile
7	NtAccessCheckByTypeResultListAnd-AuditAlarmByHandle	155	NtQueryEvent
8	NtAddAtom	156	NtQueryFullAttributesFile
9	NtAddBootEntry*	157	NtQueryInformationAtom
10	NtAddDriverEntry*	158	NtQueryInformationFile
11	NtAdjustGroupsToken	159	NtQueryInformationJobObject
12	NtAdjustPrivilegesToken	160	NtQueryInformationPort
13	NtAlertResumeThread	161	NtQueryInformationProcess
14	NtAlertThread	162	NtQueryInformationThread
15	NtAllocateLocallyUniqueId	163	NtQueryInformationToken
16	NtAllocateUserPhysicalPages	164	NtQueryInstallUILanguage
17	NtAllocateUuids	165	NtQueryIntervalProfile

续表

编号	系统服务	编号	系统服务
18	NtAllocateVirtualMemory	166	NtQueryIoCompletion
19	NtApphelpCacheControl	167	NtQueryKey
20	NtAreMappedFilesTheSame	168	NtQueryMultipleValueKey
21	NtAssignProcessToJobObject	169	NtQueryMutant
22	NtCallbackReturn	170	NtQueryObject
23	NtCancelDeviceWakeupRequest*	171	NtQueryOpenSubKeys
24	NtCancelIoFile	172	NtQueryOpenSubKeysEx
25	NtCancelTimer	173	NtQueryPerformanceCounter
26	NtClearEvent	174	NtQueryQuotaInformationFile
27	NtClose	175	NtQuerySection
28	NtCloseObjectAuditAlarm	176	NtQuerySecurityObject
29	NtCompactKeys	177	NtQuerySemaphore
30	NtCompareTokens	178	NtQuerySymbolicLinkObject
31	NtCompleteConnectPort	179	NtQuerySystemEnvironmentValue
32	NtCompressKey	180	NtQuerySystemEnvironmentValueEx*
33	NtConnectPort	181	NtQuerySystemInformation
34	NtContinue	182	NtQuerySystemTime
35	NtCreateDebugObject	183	NtQueryTimer
36	NtCreateDirectoryObject	184	NtQueryTimerResolution
37	NtCreateEvent	185	NtQueryValueKey
38	NtCreateEventPair	186	NtQueryVirtualMemory
39	NtCreateFile	187	NtQueryVolumeInformationFile
40	NtCreateIoCompletion	188	NtQueueApcThread
41	NtCreateJobObject	189	NtRaiseException
42	NtCreateJobSet	190	NtRaiseHardError
43	NtCreateKey	191	NtReadFile
44	NtCreateMailslotFile	192	NtReadFileScatter
45	NtCreateMutant	193	NtReadRequestData
46	NtCreateNamedPipeFile	194	NtReadVirtualMemory
47	NtCreatePagingFile	195	NtRegisterThreadTerminatePort
48	NtCreatePort	196	NtReleaseMutant
49	NtCreateProcess	197	NtReleaseSemaphore
50	NtCreateProcessEx	198	NtRemoveIoCompletion

续表

编号	系统服务	编号	系统服务
51	NtCreateProfile	199	NtRemoveProcessDebug
52	NtCreateSection	200	NtRenameKey
53	NtCreateSemaphore	201	NtReplaceKey
54	NtCreateSymbolicLinkObject	202	NtReplyPort
55	NtCreateThread	203	NtReplyWaitReceivePort
56	NtCreateTimer	204	NtReplyWaitReceivePortEx
57	NtCreateToken	205	NtReplyWaitReplyPort
58	NtCreateWaitablePort	206	NtRequestDeviceWakeup
59	NtDebugActiveProcess	207	NtRequestPort
60	NtDebugContinue	208	NtRequestWaitReplyPort
61	NtDelayExecution	209	NtRequestWakeupLatency
62	NtDeleteAtom	210	NtResetEvent
63	NtDeleteBootEntry*	211	NtResetWriteWatch
64	NtDeleteDriverEntry*	212	NtRestoreKey
65	NtDeleteFile	213	NtResumeProcess
66	NtDeleteKey	214	NtResumeThread
67	NtDeleteObjectAuditAlarm	215	NtSaveKey
68	NtDeleteValueKey	216	NtSaveKeyEx
69	NtDeviceIoControlFile	217	NtSaveMergedKeys
70	NtDisplayString	218	NtSecureConnectPort
71	NtDuplicateObject	219	NtSetBootEntryOrder*
72	NtDuplicateToken	220	NtSetBootOptions*
73	NtEnumerateBootEntries*	221	NtSetContextThread
74	NtEnumerateDriverEntries*	222	NtSetDebugFilterState
75	NtEnumerateKey	223	NtSetDefaultHardErrorPort
76	NtEnumerateSystemEnvironmentValues Ex*	224	NtSetDefaultLocale
77	NtEnumerateValueKey	225	NtSetDefaultUILanguage
78	NtExtendSection	226	NtSetDriverEntryOrder*
79	NtFilterToken	227	NtSetEaFile
80	NtFindAtom	228	NtSetEvent
81	NtFlushBuffersFile	229	NtSetEventBoostPriority
82	NtFlushInstructionCache	230	NtSetHighEventPair
83	NtFlushKey	231	NtSetHighWaitLowEventPair

续表

编号	系统服务	编号	系统服务
84	NtFlushVirtualMemory	232	NtSetInformationDebugObject
85	NtFlushWriteBuffer	233	NtSetInformationFile
86	NtFreeUserPhysicalPages	234	NtSetInformationJobObject
87	NtFreeVirtualMemory	235	NtSetInformationKey
88	NtFsControlFile	236	NtSetInformationObject
89	NtGetContextThread	237	NtSetInformationProcess
90	NtGetDevicePowerState	238	NtSetInformationThread
91	NtGetPlugPlayEvent	239	NtSetInformationToken
92	NtGetWriteWatch	240	NtSetIntervalProfile
93	NtImpersonateAnonymousToken	241	NtSetIoCompletion
94	NtImpersonateClientOfPort	242	NtSetLdtEntries
95	NtImpersonateThread	243	NtSetLowEventPair
96	NtInitializeRegistry	244	NtSetLowWaitHighEventPair
97	NtInitiatePowerAction	245	NtSetQuotaInformationFile
98	NtIsProcessInJob	246	NtSetSecurityObject
99	NtIsSystemResumeAutomatic	247	NtSetSystemEnvironmentValue
100	NtListenPort	248	NtSetSystemEnvironmentValueEx*
101	NtLoadDriver	249	NtSetSystemInformation
102	NtLoadKey	250	NtSetSystemPowerState
103	NtLoadKey2	251	NtSetSystemTime
104	NtLoadKeyEx	252	NtSetThreadExecutionState
105	NtLockFile	253	NtSetTimer
106	NtLockProductActivationKeys	254	NtSetTimerResolution
107	NtLockRegistryKey	255	NtSetUuidSeed
108	NtLockVirtualMemory	256	NtSetValueKey
109	NtMakePermanentObject	257	NtSetVolumeInformationFile
110	NtMakeTemporaryObject	258	NtShutdownSystem
111	NtMapUserPhysicalPages	259	NtSignalAndWaitForSingleObject
112	NtMapUserPhysicalPagesScatter	260	NtStartProfile
113	NtMapViewOfSection	261	NtStopProfile
114	NtModifyBootEntry*	262	NtSuspendProcess
115	NtModifyDriverEntry*	263	NtSuspendThread
116	NtNotifyChangeDirectoryFile	264	NtSystemDebugControl

续表

编号	系统服务	编号	系统服务
117	NtNotifyChangeKey	265	NtTerminateJobObject
118	NtNotifyChangeMultipleKeys	266	NtTerminateProcess
119	NtOpenDirectoryObject	267	NtTerminateThread
120	NtOpenEvent	268	NtTestAlert
121	NtOpenEventPair	269	NtTraceEvent
122	NtOpenFile	270	NtTranslateFilePath*
123	NtOpenIoCompletion	271	NtUnloadDriver
124	NtOpenJobObject	272	NtUnloadKey
125	NtOpenKey	273	NtUnloadKey2
126	NtOpenMutant	274	NtUnloadKeyEx
127	NtOpenObjectAuditAlarm	275	NtUnlockFile
128	NtOpenProcess	276	NtUnlockVirtualMemory
129	NtOpenProcessToken	277	NtUnmapViewOfSection
130	NtOpenProcessTokenEx	278	NtVdmControl
131	NtOpenSection	279	NtWaitForDebugEvent
132	NtOpenSemaphore	280	NtWaitForMultipleObjects
133	NtOpenSymbolicLinkObject	281	NtWaitForSingleObject
134	NtOpenThread	282	NtWaitHighEventPair
135	NtOpenThreadToken	283	NtWaitLowEventPair
136	NtOpenThreadTokenEx	284	NtWriteFile
137	NtOpenTimer	285	NtWriteFileGather
138	NtPlugPlayControl	286	NtWriteRequestData
139	NtPowerInformation	287	NtWriteVirtualMemory
140	NtPrivilegeCheck	288	NtYieldExecution
141	NtPrivilegeObjectAuditAlarm	289	NtCreateKeyedEvent
142	NtPrivilegedServiceAuditAlarm	290	NtOpenKeyedEvent
143	NtProtectVirtualMemory	291	NtReleaseKeyedEvent
144	NtPulseEvent	292	NtWaitForKeyedEvent
145	NtQueryAttributesFile	293	NtQueryPortInformationProcess
146	NtQueryBootEntryOrder*	294	NtGetCurrentProcessorNumber
147	NtQueryBootOptions*	295	NtWaitForMultipleObjects32

注：* 函数仅仅简单地返回 0xc0000002（STATUS_NOT_IMPLEMENTED，意为“此功能未实现”），不同函数的出栈字节长度不同（即 ret 指令的操作数不同）。

表 8.2 中的服务例程实际上也代表了 Windows 内核向用户模式应用程序提供的完整接口面，然而，Microsoft 并没有将这些接口例程文档化，而且也不保证同样的功能在不同的版本上使用相同的系统服务编号。事实上，从表中可以看出，这些系统服务基本上是按照字母顺序来排列的，除了少数一些未实现的服务以及最后几个系统服务以外。随着 Windows 的发展，内核的系统服务总数也呈现上升的趋势，譬如，在 Windows Server 2003 SP1 上，总数为 296；到了 Windows Vista SP1，总数为 391；而到了 Windows 7，系统服务总数达到了 401。由于从用户模式到内核模式的系统服务调用总是要经过 ntdll.dll 模块，所以，在同一个系统上，ntdll.dll 与内核模块 ntoskrnl.exe 必须使用相同的系统服务表（包括每个系统服务传递的参数总长度）。只要保证这一点，则 Windows 内核的 SDT 机制总是可以让用户模式的系统服务请求得以正确地处理。

我们现在再来看系统服务分发代码，即从_KiSystemServiceRepeat 开始的代码。这一段代码并不直接使用 KeServiceDescriptorTable 全局变量来查找系统服务例程，相反地，它使用当前线程对象（KTHREAD）中的一个指向系统服务表的指针成员（ServiceTable），来确定应该使用哪个 SDT。在 KeInitThread 函数中，此指针成员被初始化为 KeServiceDescriptorTable，参见该函数的代码。如果这是一个图形用户界面（GUI）线程，那么该线程对象的 ServiceTable 成员被替换成全局变量 KeServiceDescriptorTableShadow。系统服务分发代码根据线程的 ServiceTable 成员，利用系统服务号的表索引部分，找到相应的 SDT。再根据系统服务号的表内服务索引，确认该索引值不超过 SDT 的 Limit 成员的值。然后通过 SDT 的 Base 数组成员即可找到系统服务例程，最后调用该例程。

系统服务分发代码位于 base\ntos\ke\i386\trap.asm 文件中，我们还可以注意到另一个细节，当系统服务号的表内服务索引超过了 SDT 的 Limit 成员值时，控制流转到 Kss_ErrorHandler 标记的代码处。在 Kss_ErrorHandler 标记处的代码进行错误处理时，若系统服务号的表索引等于 1，则调用 PsConvertToGuiThread 函数，试图将当前线程转换成 GUI 线程，一旦转换成功则继续回到系统服务分发代码开始处。PsConvertToGuiThread 函数的代码位于 base\ntos\ps\psquery.c 文件中，读者可以看到一个线程如何从系统标准线程被转换成 Windows 子系统的线程，这其中可能会涉及线程内核栈的转换（通过 KeSwitchKernelStack 函数），参见 9.2.2 节。在 Windows Server 2003 SP1 系统上，Windows 子系统提供的 SDT，即 KeServiceDescriptorTableShadow[1]，共包含 665 个服务例程；在 Windows Vista SP1 系统上，该 SDT 共包含 772 个服务例程；而在 Windows 7 系统上，该 SDT 共包含 825 个服务例程。

8.5 节将介绍一个显示 SDT 的工具 SDTViewer，它可以列出当前系统上正在使用的基本 SDT 和 Windows 子系统 SDT 中所有的系统服务例程。

8.1.4 增加系统服务表或表项

理解了 Windows 的系统调用和系统服务分发机制以后，我们来看一看，Windows 的内核驱动程序或者应用程序如何扩展 Windows 的系统调用机制。首先，Windows 内核提供了一个增加系统服务表的接口函数 KeAddSystemServiceTable，以下是该函数的原型：

```
BOOLEAN
KeAddSystemServiceTable (
    IN PULONG_PTR Base,
    IN PULONG Count OPTIONAL,
    IN ULONG Limit,
    IN PUCHAR Number,
    IN ULONG Index
    );
```

参数 Base、Count、Limit 和 Number 分别对应于系统服务表描述符 KSERVICE_TABLE_DESCRIPTOR 数据结构的成员，Index 参数是指加入到 KeServiceDescriptorTable 数组的哪个元素中。从上一小节所列出的代码片段我们知道，KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow 是两个固定大小的数组，其元素个数由宏定义 NUMBER_SERVICE_TABLES 指定。在 Windows Server 2003 SP1 中，该宏定义等于 2。因此，除了系统本身已经定义的基本 SDT 以外，KeAddSystemServiceTable 只有一次被调用的机会，这正好是 Windows 子系统定义其专用系统服务表的机会，此时 Index 为 1。正如上一小节所讲，KeServiceDescriptorTable 数组中下标为 1 的项并不使用，而 KeServiceDescriptor-TableShadow 数组中下标为 1 的项则专门为 Windows 子系统保留。当 Windows 子系统的内核模块 win32k.sys 被加载时，它的入口例程调用 KeAddSystemServiceTable 函数，定义这一专用 SDT。

KeAddSystemServiceTable 函数的代码位于 base\ntos\ke\miscc.c 文件的 728~804 行，它只是简单地填充由 Index 指定的数组项。如果该数组项已经被填充过，即它的 Base 成员不为 NULL，那么，增加 SDT 的操作不成功，函数返回 FALSE。此外，另一个函数 KeRemoveSystemServiceTable 执行相反的操作，它将一个已经加入的 SDT 从数组 KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow 中移除。

尽管内核提供了这两个接口函数，但是，对于 Windows Server 2003 SP1 系统来说，系统服务表数组中的两项均已被占用，所以，增加 SDT 的功能对于第三方软件来说形同虚设。然而，对于其他版本的 Windows 系统，若系统服务表数组允许容纳更多的 SDT，那么，驱动程序就有机会定义它们自己的 SDT。一个典型的例子是，在 Windows 2000 上安装了 IIS（Internet Information Services）以后，IIS 的驱动程序（spud.sys）就在初始化例程中定义了一个私有的 SDT。

仅仅定义一个 SDT 并通过 KeAddSystemServiceTable 函数加入到系统中还不够，在用户模式部分，还需要为每个新增加的系统服务编写一个存根函数。这就如同为内核的 NtCreateFile 服务编写一个存根函数（即 ntdll.dll 中的 NtCreateFile）一样。存根函数可以由一个简单的模板生成，唯一重要的是，通过 eax 寄存器来传递的系统服务号必须与内核中的表索引和表内服务索引保持一致。另外，对于每个新增的系统服务，参数区间的长度也必须保持两端一致，否则，内核的服务例程可能访问不到用户程序传递进来的参数。Windows 子系统的 win32k.sys 驱动程序定义了它自己的 SDT，其系统服务的存根函数并不在 ntdll.dll 模块中，而是在 user32.dll 和 gdi32.dll 中，例如下面是 user32.dll 中的 NtUserGetMessage 函数的汇编代码：

```
mov     eax,11a4h
mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
call    dword ptr [edx]
ret     10h
```

以上代码与 8.1.2 节介绍的 NtCreateFile 例子代码相仿，这里第一条指令中的 11a4h 表示 NtUserGetMessage 系统服务号：表索引为 1，表内服务索引为 1a4h。第二条指令中的操作数等同于 8.1.2 节介绍的 KUSER_SHARED_DATA 共享页面中的 SystemCall 成员。

接下来我们看一看，是否有可能在 Windows 已有 SDT 的基础上，修改或者新增系统服务。在 Windows 的发展历程中，系统调用机制一直是许多系统软件厂商和操作系统爱好者的关注目标，他们试图截取某些或全部系统调用，以便实现一些特殊的系统任务，例如监视注册表的行为[NT-SYSCALL]、让用户程序可以访问内核中的例程[W2K-SECRETS]。截取系统调用无疑是实现高级系统任务（譬如，收集系统调用日志以便分析和学习系统的行为、绕过操作系统安全检查）或者扩展应用程序功能（例如，绕过 API 函数直接调用内核中的例程）的极具吸引力的手段，然而，从操作系统完整性以及稳定性等角度来考虑，这些做法值得商榷。

从系统调用机制本身来看，Windows 提供了增加或移除 SDT 的能力，但是并不是所有的系统都有这一功能空间，譬如 Windows Server 2003 SP1 就没有多余的 SDT 空间可供插入。另外，Windows 支持的基本 SDT 和子系统 SDT 并非是可增长的，这意味着，并不能直接在这两张表的后面再增加新的系统服务项，因为 SDT 中的数组成员并没有预留空间。然而，由于 WRK 提供了与系统服务分发有关的全部源代码，并且可以重新编译生成可运行的内核，因此，若基于 WRK 来增加系统服务则是有可能的，参见附录 B 中的做法或有关材料[HPI]。

即使系统定义的 SDT 没有预留的空间，有些软件也可以通过置换此 SDT 来达到增加系统服务的目的。它们的做法是，按照 KSERVICE_TABLE_DESCRIPTOR 数据结构的定

义，构造一个新的 SDT，其中不仅容纳了原来的系统 SDT 中的表项，还增加了新的表项。然后，巧妙地将它替换为 KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow 数组的第 0 项，因而原来的系统服务全部保留，新的系统服务也加入了进来。真正的替换过程只是 KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow 数组中的 SDT 项的赋值过程。只要新的 SDT 保留原来的系统服务项，并且确保原来 SDT 的那些数组成员不会立即被清除，那么，尽管内核模式的系统服务分发代码（即 8.1.2 节介绍的 KiFastSystemCall 或 KiSystemService 函数）并没有使用任何原子操作来保护对 SDT 的访问，这一置换过程也仍然是安全的。但是，如果要移除 SDT 尾部的一个系统服务，则只有等所有调用此系统服务的线程都离开此系统服务以后，这一置换才是安全的。

增加系统服务是有安全风险的，因为新系统服务运行在内核模式下，如果它绕过系统安全检查机制而直接访问受保护的资源，那么，这种做法有可能会打开一个安全缺口。因此，在衡量是否要增加系统服务时，务必谨慎考虑系统安全要素。鉴于 Windows 的各种组件往往都有较好的可扩展性，通常不会有不得已的理由要新增一个系统服务。

倒是许多软件想要截取系统调用（intercepting system calls），以便监控被调用的系统服务。这种截取技术也称为“system service hooking”，很受信息安全产品和系统工具的亲睐。为了做到这一点，这些软件往往通过一个内核模式驱动程序，找到 KeServiceDescriptorTable 和 KeServiceDescriptorTableShadow，前者是内核模块直接导出的全局变量，总是可以被访问到，后者需要额外的符号信息才能访问。然后替换掉要截取的系统服务。对于每个被替换的系统服务而言，所谓替换，是指将新例程的地址赋给 SDT 的 Base 数组中对应的元素。这一替换过程本身是安全的，但是服务例程要编写正确并不简单。

有多方面的原因造成了新服务例程编写上的困难。首先，对原系统服务的功能规范必须有正确的理解，甚至还需要对 Windows 系统的内核工作机理有充分的了解，以避免漏掉一些要监视的系统服务，或者选择不确切的系统服务集。其次，对系统服务例程的参数传递，包括参数的类型和含义，要有正确的理解；如果要监视系统服务的执行结果，那么还需要解释返回值或者输出参数。对于指针类型的参数，是否要深入到所指的对象，以及如何获取对象的信息，更需要谨慎处理。另外，若新的系统服务例程除了调用原来的系统服务以外，还需要新增功能，甚至改变原系统服务的执行结果，那么，新增功能部分的编写也同样可能会打开安全缺口，或者带来不稳定的因素。总之，与编写内核驱动程序中处理 I/O 请求的例程相比，编写系统服务例程往往更微妙，更难编写正确。

最后要提及的一点是，所有的系统服务都是系统相关的。在 Windows 平台上，系统服务是不公开的，所以，不要指望有文档来介绍你所关心的系统服务。逆向工程是一条获

得系统服务知识的途径，而 WRK 则是一个更好的参考资源。而且，你不能假定这些系统服务在不同的 Windows 版本上其编号不变，因为连它们的参数或调用规范都有可能发生变化。因此，一旦你的软件使用了截取系统服务的技术，那么你必须要做好系统版本检查工作，并且你仍然无法保证它还可以在将来的 Windows 版本上正常地工作[WIN64-PATCHING]。

8.2 LPC（本地过程调用）服务

这一节我们来讨论本地过程调用（LPC，Local Procedure Call），有时也称为 Lightweight Procedure Call。LPC 是一种直接由内核支持的进程间通信机制，它非常高效，主要用于操作系统各个组件之间进行通信，或者用户模式程序与系统组件之间通信。

LPC 涉及两方通信，其基本工作方式是消息传递。一个进程创建一个 LPC 端口对象，然后等待其他进程连接过来；当其他进程成功地连接到一个 LPC 端口以后，两者便可以开始通信。LPC 允许两个进程进行双向通信，它在 Windows 中的主要应用如下：

- Windows 应用程序与系统进程，包括 Windows 环境子系统之间的通信。这通常发生在一些 Windows API 函数的内部。
- 用户模式程序与内核模式组件之间的通信。比如在 8.2.4 节中将要介绍的 lsass（Local Security Authority SubSystem）进程与安全引用监视器（SRM，Security Reference Monitor）之间的通信就是通过 LPC 来完成的。
- 当 RPC（远过程调用，Remote Procedure Call）的两端在同一个系统中时，RPC 通信转化为 LPC 通信。

在 5.2.7 节介绍异常处理时，我们看到过，进程对象的两个端口，分别称为调试端口和异常端口，有机会接收和处理该进程中的异常，包括调试事件。这里的异常端口正是 LPC 端口，Windows 子系统通过此端口可以截获进程中发生的异常。

LPC 通信是在内核中完成的，WRK 包含了 LPC 的完整实现代码。本节首先介绍 LPC 的结构模型，并描述 LPC 端口和 LPC 消息，然后讨论 LPC 通信的建立过程和消息传输，最后介绍两个 LPC 应用实例。

8.2.1 LPC 结构模型

通过 LPC 进行通信的两个进程本质上是客户-服务器模型，其工作方式与基于连接的 socket 编程模型相仿。服务器进程创建一个 LPC 连接端口对象，然后在该端口上监听连接请求。客户进程根据已知的端口名称，请求连接到此端口对象上。当服务器接收到连接

请求时，它创建一个通信端口对象，用以代表与该客户进程之间的 LPC 连接。而客户进程也会创建一个通信端口对象，代表它与服务器进程之间的 LPC 连接。通信端口对象没有名称，它们是私有对象，所以，除了服务器进程和客户进程通过通信端口句柄来访问它们以外，其他进程无法访问。LPC 连接端口对象属于服务器进程，它们有公开的名称，通常被加入到系统的对象管理器目录中，所以，其他进程可以通过名称访问它们，从而建立起单独的 LPC 连接。连接端口对象和通信端口对象的关系如图 8.6 所示。

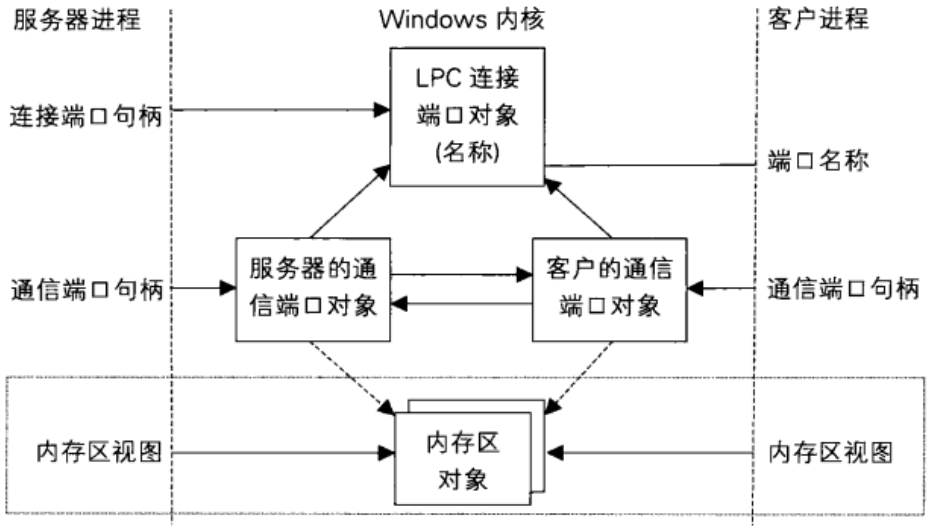


图 8.6 LPC 通信模型

LPC 服务器进程可以同时连接多个客户，每个客户通过名称连接到服务器的连接端口对象，而服务器为每个接受的客户进程创建一个通信端口对象。所以，LPC 允许一对多通信模型。LPC 连接端口对象是一个中心的连接点，它只接受连接请求，不接受数据请求。而通信端口可进行任意的数据请求和服务。这是连接端口和通信端口的职责区别。

当服务器进程和客户进程建立了 LPC 连接以后，它们可以给对方发送消息。每个端口对象都有一个消息队列，用来保存发送给该端口对象的消息。LPC 允许使用两种方法来传输数据：

- 当最大消息长度不超过 256 字节时，要传输的数据直接跟在消息头后面。发送进程将数据拷贝到系统地址空间中，然后接收进程将数据拷贝到它的进程地址空间中。
- 若最大消息长度超过 256 字节，那么，客户在连接到端口对象上时，可以指定一个内存区对象，并且在客户进程中映射一个视图，以用于向服务器发送大数据；服务器进程在接受连接时，也映射一个视图。而且，服务器也可以指定一个内存区对象，用于向客户发送大数据。因此，两个进程通过共享内存区来传输数据，如图 8.6 所示。

Windows 实现的 LPC 使用消息队列来保存发送给一个端口对象的消息，使用信号量对象来同步发送和接收操作。而且，Windows 已经将 LPC 的基本操作直接以系统服务的形式暴露给用户模式代码，因而用户程序可以利用 LPC 实现极为高效的通信。表 8.3 列出了 Windows 中的 LPC 系统服务，8.2.3 节将会介绍这些系统服务的实现原理。

表 8.3 LPC 系统服务（适用于 Windows 2000 以后所有的 Windows 版本）

系统服务	简要说明
NtAcceptConnectPort	服务器进程利用该服务来接受或拒绝一个连接请求
NtCompleteConnectPort	服务器进程在调用了 NtAcceptConnectPort 以后，再调用该服务以便唤醒客户线程
NtConnectPort	通过该服务，客户进程可通过名称来连接到一个服务器进程
NtCreatePort	服务器进程利用该服务来创建一个 LPC 连接端口
NtCreateWaitablePort	服务器进程利用该服务来创建一个 LPC 连接端口，它允许异步方式等待 LPC 消息，即等待客户连接请求的到来
NtListenPort	这是服务器线程调用的系统服务，它利用 NtReplyWait-ReceivePort 服务来等待来自客户端的连接请求
NtReplyPort	发送一个应答消息
NtReplyWaitReceivePort	（可选地）发送一个应答消息，等待接收一个客户消息
NtReplyWaitReceivePortEx	功能同 NtReplyWaitReceivePort，可指定超时值
NtReplyWaitReplyPort	发送一个应答消息，并等待此应答消息的应答消息
NtRequestPort	发送一个请求消息
NtRequestWaitReplyPort	发送一个请求消息，并等待此请求的应答消息
NtSecureConnectPort	通过此服务，客户进程可通过名称来连接到一个服务器进程，它允许指定服务器进程的安全标识符

8.2.2 LPC 端口和 LPC 消息

LPC 端口代表了 LPC 通信中的一个端点，它是内核中的对象，维护了通信所需要的状态信息。用户模式代码不能直接访问 LPC 端口对象，只能指定所使用的端口的句柄。端口对象可以有名称，也可以没有名称。在上一节介绍的 LPC 通信模型中，我们已经看到了三种不同的 LPC 对象，即连接端口对象、服务器通信端口对象和客户通信端口对象。实际上，除此以外，还有一种非连接的通信端口对象，它主要用于在同一个进程中的两个线程之间进行通信。虽然这些端口对象的类型不同，但 Windows 使用统一的数据结构 LPCP_PORT_OBJECT 来予以描述，以下是其定义，见 base\ntos\inc\lpc.h 文件：

```
typedef struct _LPCP_PORT_OBJECT {
   struct _LPCP_PORT_OBJECT *ConnectionPort;
   struct _LPCP_PORT_OBJECT *ConnectedPort;
```

```
    LPCP_PORT_QUEUE MsgQueue;
    CLIENT_ID Creator;
    PVOID ClientSectionBase;
    PVOID ServerSectionBase;
    PVOID PortContext;
    PETHREAD ClientThread;                    // 仅适用于服务器通信端口
    SECURITY_QUALITY_OF_SERVICE SecurityQos;
    SECURITY_CLIENT_CONTEXT StaticSecurity;
    LIST_ENTRY LpcReplyChainHead;             // 仅适用于通信端口
    LIST_ENTRY LpcDataInfoChainHead;          // 仅适用于通信端口
    union {
        PEPROCESS ServerProcess;              // 仅适用于服务器连接端口对象
        PEPROCESS MappingProcess;             // 仅适用于通信端口
    };
    USHORT MaxMessageLength;
    USHORT MaxConnectionInfoLength;
    ULONG Flags;
    KEVENT WaitEvent;                         // 仅适用于可等待的端口
} LPCP_PORT_OBJECT, *PLPCP_PORT_OBJECT;
```

可以看出，LPC 端口的定义较为简洁，下面仅对它的域作简单说明。ConnectionPort 域指向当前端口对象的连接对象，对于服务器通信端口或客户通信端口，该域指向相关联的连接端口对象；连接端口的 ConnectionPort 域指向其自身。ConnectedPort 域指向通信的对方，所以，通过一个通信端口对象可以很方便地定位到对方的通信端口对象。服务器通信端口的 ConnectedPort 域指向客户通信端口，反之亦然；连接端口的 ConnectedPort 域指向其自身。MsgQueue 域表示端口对象的消息队列。Creator 域表示创建线程的 ID。ClientSectionBase 和 ServerSectionBase 域分别指客户和服务器的内存区对象的视图基地址。PortContext 域是一个由服务器进程管理和解释的指针域。ClientThread 域仅用于服务器通信端口对象，在建立连接过程中用于记录客户线程对象。SecurityQos 和 StaticSecurity 域用于建立端口对象的安全环境。LpcReplayChainHead 和 LpcDataInfoChainHead 是两个链表头，用于通信端口存放应答消息。ServerProcess 域仅用于连接端口，存放服务器进程对象指针。MappingProcess 域仅用于通信端口，存放映射内存区视图的进程，以便销毁端口对象时可以解除映射。MaxMessageLength 和 MaxConnectionInfoLength 域指示了该端口的最大消息长度和最大连接信息长度。Flags 域指示了该端口对象的类型和状态标志，其最低 4 位代表端口的类型，即前面提到的 4 种类型。WaitEvent 域仅用于可等待的端口（连接对象的一个变种），此事件的信号状态表示该端口是否有消息到达，因此，一个线程可以在该事件对象上等待消息的到来。

LPC 端口对象是 Windows 系统的内核对象，由对象管理器来管理。LPC 子系统在初始化时，创建了两种 LPC 端口对象类型，名称为“Port”和“WaitablePort”。两者的一个主要区别是，后者包含了 LPCP_PORT_OBJECT 的 WaitEvent 域，并且从非换页内存池分配；而前者不包含 WaitEvent 域，从换页内存池分配。读者可以参考 base\ntos\lpc\lpcinit.c 文件中的 LpcInitSystem 函数的代码。这里，LpcInitSystem 函数是在内核的阶段 1 初始化

过程中被调用的。因此，当内核代码需要创建 LPC 端口对象时，只需调用 ObCreateObject 函数即可，然后，可以调用 ObInsertObject 将 LPC 端口对象插入到当前进程的句柄表中。而用户模式代码可以通过句柄来引用一个 LPC 端口对象。

接下来看一看 LPC 通信中用到的消息结构，其定义 LPCP_MESSAGE 也位于 base\ntos\inc\lpc.h 文件中。

```
typedef struct _LPCP_MESSAGE {
   union {
      LIST_ENTRY Entry;
      struct {
         SINGLE_LIST_ENTRY FreeEntry;
         ULONG Reserved0;
      };
   };

   PVOID SenderPort;
   PETHREAD RepliedToThread;          // 发送应答时填充，因而接收方可以解除对该线程的引用
   PVOID PortContext;                 // 从发送方的通信端口中获得
   PORT_MESSAGE Request;
} LPCP_MESSAGE, *PLPCP_MESSAGE;
```

LPCP_MESSAGE 定义的是消息头，真正的消息体附在消息头的后面。Entry 域是一个消息被插入消息队列时的链表节点对象。SenderPort 域指向发送方的端口对象。RepliedToThread 域指向正在等待应答的线程，当发送应答消息时，此域需填充，因而接收方可以解除对该线程的引用。PortContext 域指向发送方端口对象的 PortContext。Request 域是一个 PORT_MESSAGE 结构，其中指定了消息的长度、类型、消息 ID 等信息，其定义位于 public\sdk\inc\ntlpcapi.h 文件中，这里不再列出。

8.2.3 LPC 通信模型的实现

LPC 通信过程可以分为两个阶段：建立连接阶段以及数据传输阶段。建立连接的阶段实际上是一次握手过程，服务器进程在一个公开名称的端口上等待连接请求，客户根据名称连接到服务器的连接端口上。连接阶段的结果是，客户端的通信端口与服务器端的通信端口相互链接起来，并准备好发送或接收消息。在数据传输阶段，客户和服务器可以依据其功能逻辑的需求发送或接收消息，发送消息可以要求应答，也可以不要求；如果接收方不在等待数据，或者未能及时地处理消息，则消息可以留存在目标端口对象的消息队列中，以等待被处理。

在建立连接阶段，首先服务器进程调用 NtCreatePort 或 NtCreateWaitablePort 函数创建一个 LPC 端口，这两个函数的原型如下：

```
NTSYSCALLAPI
NTSTATUS
NTAPI
```

```
NtCreatePort(
    __out PHANDLE PortHandle,
    __in POBJECT_ATTRIBUTES ObjectAttributes,
    __in ULONG MaxConnectionInfoLength,
    __in ULONG MaxMessageLength,
    __in_opt ULONG MaxPoolUsage
    );

NTSYSCALLAPI
NTSTATUS
NTAPI
NtCreateWaitablePort(
    __out PHANDLE PortHandle,
    __in POBJECT_ATTRIBUTES ObjectAttributes,
    __in ULONG MaxConnectionInfoLength,
    __in ULONG MaxMessageLength,
    __in_opt ULONG MaxPoolUsage
    );
```

函数的代码位于 base\ntos\lpc\lpccreat.c 文件中。实际上，这两个函数都只是简单地调用位于同一文件中的 LpcpCreatePort 函数。LpcpCreatePort 根据参数中指定的名称，创建一个 LPC 端口对象。它首先检查参数，然后调用 ObCreateObject 创建一个 LpcPortObjectType 或者 LpcWaitablePortObjectType 类型的内核对象。接着初始化该对象中的域，包括最大消息长度和最大连接信息长度。最大消息长度不得超过预设的长度，即 PORT_MAXIMUM_MESSAGE_LENGTH，为 256 字节。如果调用者指定了名称，则此端口对象为连接端口对象（SERVER_CONNECTION_PORT），否则是非连接的通信端口对象（UNCONNECTED_COMMUNICATION_PORT）。最后，调用 ObInsertObject 函数将新建的端口对象插入到当前进程的句柄表中。这些创建函数，包括 NtCreatePort、NtCreateWaitablePort 和 LpcpCreatePort 函数，都向调用者返回一个指向 LPC 端口对象的句柄。

创建了 LPC 端口对象以后，服务器进程调用 NtListenPort 函数监听连接请求，NtListenPort 的监听是一个同步过程，它调用 NtReplyWaitReceivePort 函数，直至该函数接收到一个 LPC 连接请求，或者返回不成功。下面是这两个函数的原型：

```
NTSTATUS
NtListenPort (
    __in HANDLE PortHandle,
    __out PPORT_MESSAGE ConnectionRequest
    );

NTSTATUS
NtReplyWaitReceivePort (
    __in HANDLE PortHandle,
    __out_opt PVOID *PortContext ,
    __in_opt PPORT_MESSAGE ReplyMessage,
    __out PPORT_MESSAGE ReceiveMessage
    );
```

监听过程主要是在 NtReplyWaitReceivePort 函数中完成的。NtReplyWaitReceivePort

是一个通用的函数，LPC 服务器和客户进程都可以使用该函数来接收消息。NtReplyWaitReceivePort 调用 NtReplyWaitReceivePortEx 函数来完成其功能，这两个函数的代码位于 base\ntos\lpc\lpcrecv.c 文件中。NtReplyWaitReceivePortEx 函数首先检查其参数，尤其是 ReplyMessage 和 PortHandle 参数。接下来，确定接收者的端口对象，即局部变量 ReceivePort。如果指定了 ReplyMessage 参数，则从此消息中定位到目标线程，并调用 KeReleaseSemaphore 函数，唤醒正在等待的目标线程。NtListenPort 将 ReplyMessage 参数设置为 0，所以，服务器监听连接请求时不需要处理 ReplyMessage 参数。待完成参数处理以后，NtReplyWaitReceivePortEx 函数在接收者端口对象的消息队列的信号量对象上等待，即代码中的 ReceivePort->MsgQueue.Semaphore 对象。等待成功以后，从 ReceivePort 的消息队列中提取出消息，并复制到参数 ReceiveMessage 指定的消息对象中。

因此，NtListenPort 函数直接在 LPC 连接端口对象的消息队列信号量上等待，直至有客户向此端口对象发送连接请求消息。在客户进程中，这是通过 NtConnectPort 函数来完成的：

```
NTSYSAPI
NTSTATUS
NTAPI
NtConnectPort (
    __out PHANDLE PortHandle,
    __in PUNICODE_STRING PortName,
    __in PSECURITY_QUALITY_OF_SERVICE SecurityQos,
    __inout_opt PPORT_VIEW ClientView,
    __inout_opt PREMOTE_PORT_VIEW ServerView,
    __out_opt PULONG MaxMessageLength,
    __inout_opt PVOID ConnectionInformation,
    __inout_opt PULONG ConnectionInformationLength
    );
```

NtConnectPort 的代码位于 base\ntos\lpc\lpcconn.c 文件中，它调用同一文件中的另一函数 NtSecureConnectPort。NtSecureConnectPort 要求一个额外的服务器 SID 信息，NtConnectPort 将其简单地设置为 NULL。参数 PortName 指明了客户要连接的 LPC 端口名称，这正是服务器在 NtCreatePort 中指定的 LPC 端口名称。如果客户发送的连接请求被服务器接受，则该函数返回时，客户获得一个 LPC 通信端口对象。此端口没有名称，属于客户进程私有，以后客户利用该端口与服务器传送数据。如果 ClientView 参数包含一个有效的内存区句柄，则该句柄所指的内存区对象被映射到客户进程地址空间和服务器进程地址空间中。如果连接请求被服务器拒绝，或者 PortName 指定的端口不存在，或者客户没有权限访问指定的端口，则 NtConnectPort 的返回值会指明错误原因，ConnectionInformation 参数可能包含进一步的失败原因细节。

NtSecureConnectPort 首先检查参数的有效性，并作相应处理。它根据 PortNmae 获得目标连接对象，然后调用 ObCreateObject 创建一个新的 LPC 端口对象。如果参数 ClientView

指定了内存区对象，则在当前进程中映射一个视图。然后初始化新建的 LPC 端口对象，并构造一个请求连接的 LPC 消息，将它插入到 LPC 连接对象的消息队列中。接下来针对服务器连接对象的消息队列的信号量对象调用 KeReleaseSemaphore，因此，唤醒服务器进程中正在监听该连接对象的线程。然后，在当前线程的 LpcReplySemaphore 信号量成员上等待。服务器进程在处理客户连接请求的最后阶段（在 NtCompleteConnectPort 函数中）会向客户线程的 LpcReplySemaphore 信号量发送信号，稍后我们会讲到。NtSecureConnectPort 函数得到了服务器对象的信号通知以后，将客户的通信端口对象插入到进程的句柄表中。如果连接不成功，也需要做相应的返回处理。

因此，服务器的 NtListenPort 在连接端口对象的消息队列的信号量对象上等待，而客户在发送连接请求时会唤醒服务器线程。而且，NtListenPort 返回一个连接请求消息，其中描述了客户正在进行的连接请求。服务器进程接下来调用 NtAcceptConnectPort 和 NtCompleteConnectPort 函数，以便接受或拒绝客户的连接请求。这两个函数的代码位于 base\ntos\lpccompl.c 文件中，其原型如下：

```
NTSTATUS
NtAcceptConnectPort (
    __out PHANDLE PortHandle,
    __in_opt PVOID PortContext,
    __in PPORT_MESSAGE ConnectionRequest,
    __in BOOLEAN AcceptConnection,
    __inout_opt PPORT_VIEW ServerView,
    __out_opt PREMOTE_PORT_VIEW ClientView
    );

NTSTATUS
NtCompleteConnectPort (
    __in HANDLE PortHandle
    );
```

NtAcceptConnectPort 函数中的 ConnectionRequest 参数是 NtListenPort 返回的连接请求消息，AcceptConnection 参数指明了应该接受还是拒绝此请求，ServerView 和 ClientView 参数用于内存区对象的视图映射。如果服务器进程接受此连接请求，那么，该函数创建一个通信端口，并且将它插入到进程的句柄表中。此通信端口与客户进程创建的通信端口相互链接起来。如果 NtAcceptConnectPort 函数的 ServerView 参数是一个有效值，那么，服务器进程分配的内存区对象被用于从服务器向客户进程传送大数据消息。这不同于在客户请求连接时指定的内存区对象和视图。

NtAcceptConnectPort 首先检查参数，根据连接请求消息中的信息找到客户线程，并且构造一个应答消息。然后，如果服务器接受此连接请求，则构造一个服务器通信端口对象。若客户已经指定了内存区对象，则服务器也映射一个视图。如果服务器已经分配了一个内存区对象用于向客户发送数据，则分别为客户进程和服务器进程映射两个视图。然后，将

通信端口对象插入到进程的句柄表中，再设置有关的输出参数并返回。注意，如果 NtAcceptConnectPort 成功接受连接，则它并不唤醒正在线程对象的 LpcReplySemaphore 信号量成员上等待的客户线程。相反，若不接受此客户的连接请求，则调用 KeReleaseSemaphore 唤醒客户线程。

因此，如果服务器线程调用 NtAcceptConnectPort 函数返回成功，则必须再调用 NtCompleteConnectPort 函数，才能完成连接过程。NtCompleteConnectPort 函数的 PortHandle 参数指向服务器的通信端口对象，它再次检查客户线程的状态和端口对象的一些状态信息，然后针对客户线程的 LpcReplySemaphore 信号量成员调用 KeRelease-Semaphore 函数，以唤醒客户线程。

至此，客户进程与服务器进程之间的 LPC 连接已经建立起来，双方各自得到一个句柄指向自己这一端的通信端口对象。如果客户或服务器在创建过程中指定了内存区对象，则现在双方已经有了各自的映射视图，从而为后续的数据传输做好了准备。图 8.7 显示了建立 LPC 连接的交互过程。

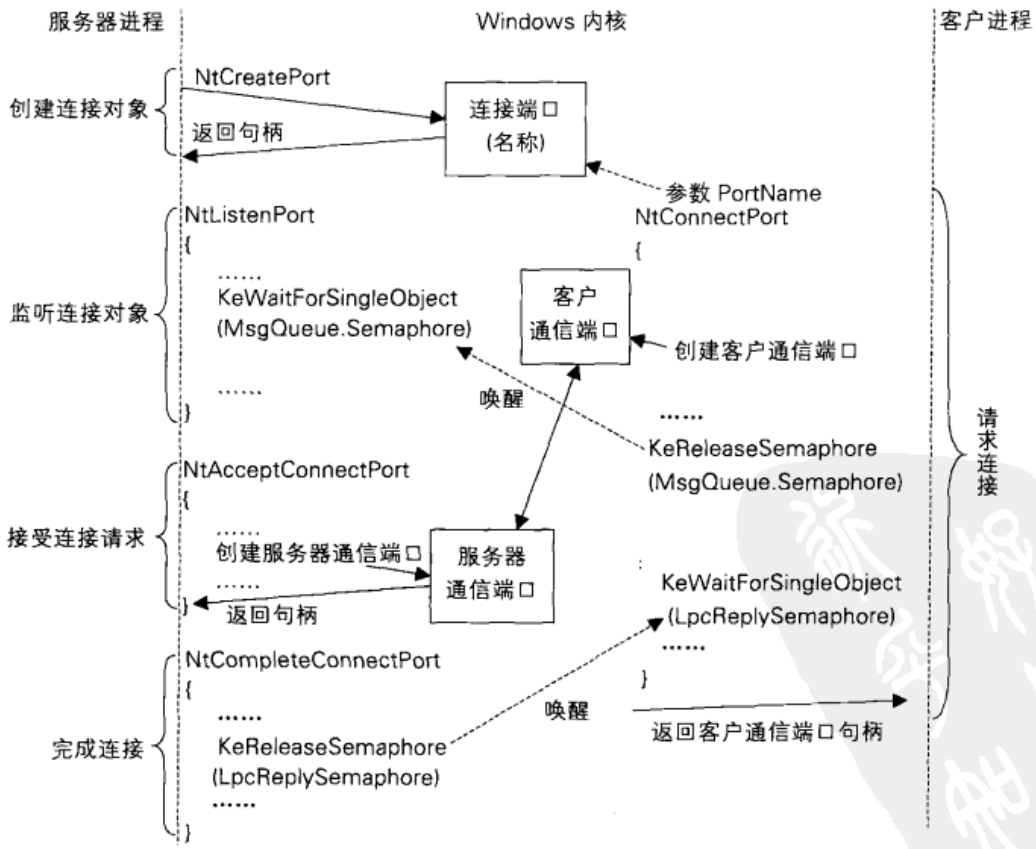


图 8.7 建立 LPC 连接的交互过程示意图

接下来我们讨论数据传输阶段。一旦两个进程建立连接以后，任何一方都可以向另一方发送消息，或者接收应答消息。以下是一些用于 LPC 通信的函数的简要说明：

- NtRequestPort，向指定的端口发送一个请求消息。该函数首先找到目标端口，将消息插入到该端口的消息队列中，然后调用 KeReleaseSemaphore 使消息队列的信号量计数器增 1。
- NtRequestWaitReplyPort，向指定的端口发送一个请求消息，然后等待应答。该函数首先检查要发送的消息，并利用消息中的信息找到要唤醒的线程，将请求消息告诉对方，并唤醒它，然后等待对方的应答。
- LpcRequestPort，此函数与 NtRequestPort 类似，但是它直接使用端口对象的地址，而不是句柄。
- LpcRequestWaitReplyPort/LpcRequestWaitReplyPortEx，这两个函数与 NtRequestWait-ReplyPort 类似，但是它们直接使用端口对象的地址，而非句柄。
- NtReplyWaitReceivePort/NtReplyWaitReceivePortEx，如果参数中指定了应答消息，则首先根据应答消息中的信息，向应答目标方传送应答消息，并唤醒对方。然后在端口对象的消息队列信号量上等待，以接收消息。如前所述，NtListenPort 使用此函数来监听客户进程的连接请求。
- NtReplyPort，发送一个应答消息。参数 PortHandle 指定了原来接收到请求消息的端口对象，参数 ReplyMessage 指定了要送回的应答消息。
- NtReplyWaitReplyPort，发送一个应答消息，并且等待对此应答消息的应答。

这些函数的实现代码并不复杂，它们位于 base\ntos\lpc\lpcsend.c、lpcrecv.c 和 lpcreply.c 文件中。在阅读代码时，以下几点请留意：

1. 在传输消息时，不管是请求消息还是应答消息，都要进行消息拷贝，这是通过 LpcpMoveMessage 函数来完成的。以上这些传输函数的参数中提供的消息（注意，是 PORT_MESSAGE 对象而非 LPCP_MESSAGE 对象）并不被传递到另一方。相反地，真正被传递的消息是从一个全局的快查表（LpcpMessagesLookaside）中分配的，参见 LpcpAllocateFromPortZone 的代码。收到消息的一方，调用 LpcpFreeToPortZone 函数将它放回全局快查表。
2. 应答消息必定是针对前一个已经传输的消息，线程对象 ETHREAD 的 LpcReplyMessageId 域记录了该消息的 ID，因此，在发送应答消息时，须确保应答消

息的 MessageId 与目标线程中记录的消息 ID 一致。应答消息通过 ETHREAD 对象的 LpcReplyMessage 域来传递，而通知信号则通过其 LpcReplySemaphore 信号量成员来传递。此信号量的最大计数值为 1，所以，它实质上被当做互斥体来使用。

3. 消息 ID 通过一个简单的计数器来产生，参见 C 宏 LpcpGenerateMessageId。

4. 如果一个函数要等待应答消息，则在等待之前，要将当前线程的 LpcReplyChain 链表节点加入到适当端口对象的 LpcReplyChainHead 链表中。正常情况下，应答方在发送应答消息前会将它移除。当端口对象被删除时，它会遍历此链表，并唤醒这些等待线程。当线程结束时，如果它的 LpcReplyChain 成员仍然位于一个链表中，则将它从链表中移除。参见 LpcpDestroyPortQueue 函数（被 LpcpClosePort 调用）和 LpcExitThread 函数（被 PspExitThread 调用）中的处理。

5. LPC 使用了全局锁 LpcpLock，这是一个守护互斥体。参见 C 宏 LpcpAcquireLpcpLock、LpcpAcquireLpcpLockByThread 和 LpcpReleaseLpcpLock。线程对象的 LpcReplyMessage 和 LpcReplyChain 成员、LPC 端口对象和 LPC 消息的一些操作和处理都受此全局锁的保护。

6. 如果这些函数的调用者是用户模式代码，则必须严格检查参数的有效性，我们在大多数函数的开始处都可以看到像 ProbeForXxx 这样的调用。

最后，我们提一下端口对象的删除和资源回收。LPC 子系统在注册 LPC 端口对象类型时，已经指定了它的 Close 方法是 LpcpClosePort 函数，Delete 方法是 LpcpDeletePort 函数。当一个端口对象被关闭或删除时，如果它的引用计数已经为 0，则各种资源都需要回收，包括消息队列中尚未接收的消息、消息队列本身、已经映射的内存区视图，等等。另外，当一个线程退出时，PspExitThread 函数会调用 LpcExitThread 函数，以便回收该线程中可能尚未被接收的应答消息（即 LpcReplyMessage 成员）。关于这些后处理函数，请参考 base\ntos\lpc\lpcclose.c 文件中的代码。

8.2.4 LPC 应用

我们已经看过了 LPC 通信的实现机理，现在来看 LPC 的应用。虽然前面介绍的许多 LPC 函数都以系统服务的形式暴露到了用户模式下，但是，在 Windows SDK 中并没有关于 LPC 系统服务或 API 函数的参考文档。因此，如果 Windows 应用程序想直接使用 LPC，则必须弄清楚 ntdll.dll 中对应 LPC 函数的原型，才可以调用它们。正如前面所讲，Windows 有一些 API 函数的内部已经使用了 LPC 来跟系统进程或者 Windows 环境子系统进行通信。

关于 Windows 程序如何使用 LPC 系统服务进行通信的话题，已经超出了本书的范围。相信读者在理解了以上小节介绍的 LPC 通信机制以后，不难编写出正确使用 LPC 的程序。

接下来介绍两个在内核中使用 LPC 的例子。第一个例子是，当系统的异常处理例程在分发用户模式异常时，它可能会将异常分发到进程的异常端口，即 EPROCESS 的 ExceptionPort 成员，参见图 5.11(b)。在 KiDispatchException 函数中，这是通过调用 DbgkForwardException 函数来完成的。而 DbgkForwardException 函数又调用 DbgkpSendApiMessageLpc 来发送 LPC 消息。最后，DbgkpSendApiMessageLpc 调用 LpcRequestWaitReplyPortEx 函数发送异常请求消息，并等待应答。

进程的 ExceptionPort 成员是在进程创建时被赋值的，参见 PspCreateProcess 中的代码。由 Windows 子系统创建的进程，其 ExceptionPort 指向一个由子系统创建的 LPC 端口，Windows 子系统进程（csrss.exe）负责接收并处理该端口上的请求消息。所以，Windows 子系统有机会处理应用程序中未被处理的异常。

第二个例子是内核模式的安全引用监视器（SRM）与用户模式的 lsass 进程之间的通信。SRM 是执行体内部的一个组件，负责安全访问检查、管理用户特权等与安全有关的事务，WRK 包含有它的代码。Lsass 则负责本地系统的安全策略、用户认证以及发送安全审计消息到系统日志中等任务。SRM 和 lsass 之间需要相互通信，其通信模型如图 8.8 所示。

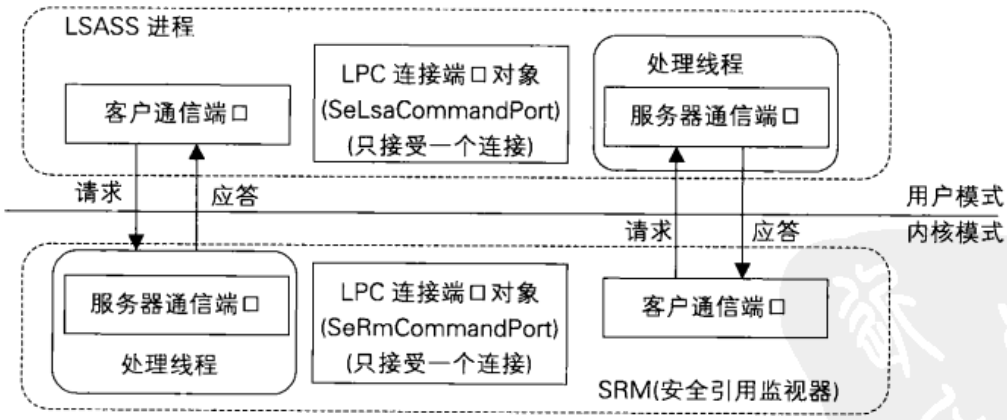


图 8.8 SRM 和 lsass 之间的通信模型

SRM 在初始化时，会创建一个名为“SeRmCommandPort”的 LPC 端口对象，其句柄存放在全局变量 SepRmState 的 RmCommandServerPortHandle 成员中。当 SRM 的线程 SepRmCommandServerThread 初始运行时，会在此端口对象上监听，直至 lsass 连接上来，然后调用 ZwAcceptConnectPort 和 ZwCompleteConnectPort 函数，完成连接过程，得到一

个服务器通信端口，其句柄存放在全局变量 SepRmState 的 RmCommandPortHandle 成员中。接着，SRM 再调用 ZwConnectPort 函数连接至 lsass 进程创建的名为“SeLsaCommandPort”的端口对象上。这样，SRM 和 lsass 之间的通信就建立了起来。以后，SepRmCommandServerThread 线程就在 RmCommandPortHandle 句柄所引用的通信端口上接收和处理消息。读者可以参考 base\ntos\se\rmmain.c 文件中的代码。这里以 Zw 打头的 LPC 函数是上一小节介绍的 LPC 系统服务的一个功能等价版本（用于内核模式的调用者），读者可以将其理解成以 Nt 打头的等价函数。

可以想见 lsass 也经过了一个类似的 LPC 连接建立过程。由于 SRM 和 lsass 在建立了连接以后，便不再在“SeRmCommandPort”和“SeLsaCommandPort”连接端口上监听和接受连接请求，因此，其他的进程和内核组件无法与它们进行 LPC 通信。这正是 SRM 和 lsass 保证其安全性的一个基础。

8.3 命名管道（Named Pipe）服务

上一节介绍的 LPC 机制是 Windows 系统中不同进程之间的高效通信手段，这一节我们介绍一种可用于在不同系统之间进行网络通信的机制，即 Windows 命名管道服务。命名管道提供了双向通信的能力，与 Windows 的文件对象管理紧密地集成在一起。它使用了 Windows 安全机制，所以，命名管道的服务器端可以控制哪些客户有权与它建立连接。

利用命名管道机制，不同机器上的进程可以相互进行通信。命名管道依赖于低层的网络接口，包括 DNS 服务、TCP/IP 协议等。这一节我们忽略对于 Windows 网络机制的讨论，而只关注本地系统中的命名管道通信。

8.3.1 命名管道的名称解析

自然地，命名管道有名称，服务器在创建一个命名管道时指定该管道的名称，客户通过名称连接到此管道上，所以，名称就成了服务器和客户之间的联系纽带。在 Windows 中，管道的名称遵从 Windows 统一命名规范（UNC，Universal Naming Convention），这是一种标识 Windows 网络资源的方法。

命名管道的名称格式为“\\<Server>\Pipe\<PipeName>”，这里<Server>指定了一个命名管道的服务器所在的计算机的名称，既可以是 DNS 名称，也可以是 NetBIOS 名称，或者字符串形式的 IP 地址；<PipeName>是该命名管道的唯一名，这一部分可以是层次结构中的一个全路径名，即允许包含多级目录。以下就是一个合法的命名管道名称：

```
\\MyServer\Pipe\MyAppServer\ConnectionPipe
```

我们首先来看一下 Windows 解析 UNC 名称的过程。Windows 有一个专门解析 UNC 名称的网络组件，称为 MUP（Multiple UNC Provider，多 UNC 提供者），它负责把针对 UNC 名称的网络 I/O 请求传递给处理该 UNC 名称的组件，这些组件称为 UNC 提供者或网络重定向器（network redirector）。

MUP 是一个内核设备驱动程序，在系统引导时被加载到系统中，创建一个名为“\Device\Mup”的设备对象。MUP 也建立符号链接名“\Global??\UNC”，让它指向设备对象“\Device\Mup”。当一个 UNC 提供者驱动程序被加载到系统中时，它创建一个命名的设备对象，并且调用 FsRtlRegisterUncProvider 函数，向 MUP 注册，使其成为一个 UNC 提供者。在 Windows 2000 以后的系统中，rdbss.sys 驱动程序为网络重定向器提供了一个缓存框架，它实现了网络重定向器所需要的大部分公共功能，包括与 I/O 管理器、缓存管理器和内存管理器打交道的代码逻辑，因而简化了网络重定向器的编写。Rdbss.sys 框架中的网络重定向器称为网络小重定向器（network mini-redirector），一个典型的例子是 mrxsmb.sys 驱动程序，它负责 SMB（Server Message Block）协议的远程文件系统的重定向解析。Windows DDK 提供了关于 MUP 和网络重定向器的完整文档[MSDN-UNC]。

在 Windows 平台上，MUP 驱动程序本身也是分布式文件系统（DFS，Distributed File System）的客户端实现，所以，它也是一个重定向器，负责解析 DFS 路径（遵从 UNC 格式）。如果 MUP 接收到的 I/O 请求并非 DFS 路径，那么，MUP 会将它交给其他已注册的重定向器。重定向器通过低层的协议驱动程序与远程计算机系统进行通信。

当应用程序通过指定一个 UNC 路径来打开一个远程文件（或者设备、命名管道等）时，Windows 的系统 DLL（kernel32.dll）将在 UNC 路径的前面附加上“\DosDevices\UNC\”前缀，然后调用 NtCreateFile 来打开指定文件。参考 7.3.2 节关于对象管理器解析“\DosDevices”、“\??”和“\GLOBAL??”目录的过程，以“\DosDevices\UNC\”为前缀的 UNC 路径被解析到“\Device\Mup”，从而通过恰当的重定向器驱动程序被进一步解析，最终经过协议驱动程序建立网络连接。

现在回到命名管道的名称解析上来。假设一个命名管道指定的是本地的名称，例如，当命名管道服务器程序调用 CreateNamedPipe API 函数来创建命名管道时，它必须指定一个形如“\\.\Pipe\<PipeName>”的本地名称，这里“.”代表本地系统。内核中的系统服务 NtCreateNamedPipeFile 接收到的名称为“\??\PIPE\<PipeName>”，对象管理器会将此名称变换为“\Global??\PIPE\<PipeName>”，由于“\Global??\PIPE”是一个符号链接，指向

设备对象“\Device\NamedPipe”，所以，对象管理器找到该设备对象，由它所属的驱动程序进一步解析剩下的子串，这就进入到了命名管道驱动程序中。

Windows 中的命名管道客户程序本质上通过文件接口来使用命名管道，它们调用 CreateFile API 函数来创建连接。客户程序可以指定一个远程系统上的命名管道，在这种情况下，命名管道名称的解析过程需要经过 MUP 驱动程序。如果它指定了本地命名管道，如上面介绍的形式“\\.\Pipe\<PipeName>”，那么，其名称解析过程也如同服务器创建命名管道一样：对象管理器接收到“\??\PIPE\<PipeName>”，将<PipeName>交给设备对象“\Device\NamedPipe”所属的驱动程序。

因此，如果我们不考虑远程命名管道的情形，那么，命名管道实际上仅仅是本地的一个文件系统，这也正是命名管道驱动程序（npfs.sys）对象位于“\FileSystem”目录中的原因，而且 npfs 本身就代表了命名管道文件系统（named pipe file system）。每个命名管道等同于一个文件对象，与普通文件系统中的文件不同的是，命名管道是动态对象，它没有永久状态。一个命名管道的存在总是伴随着一个服务器和一个客户，其职责是为服务器和客户提供一个双向交换数据的通道。

8.3.2 命名管道的通信模型

如同 LPC 一样，命名管道的通信也是以连接方式进行的，服务器创建一个命名管道对象，然后在此对象上等待连接请求。一旦客户连接过来，则两者都可以通过命名管道读或写数据。与 LPC 不同的是，命名管道本质上使用文件接口，它的实现使用了文件系统驱动程序的形式，而且它支持远程通信。图 8.9 显示了客户和服务器建立命名管道的通信模型。

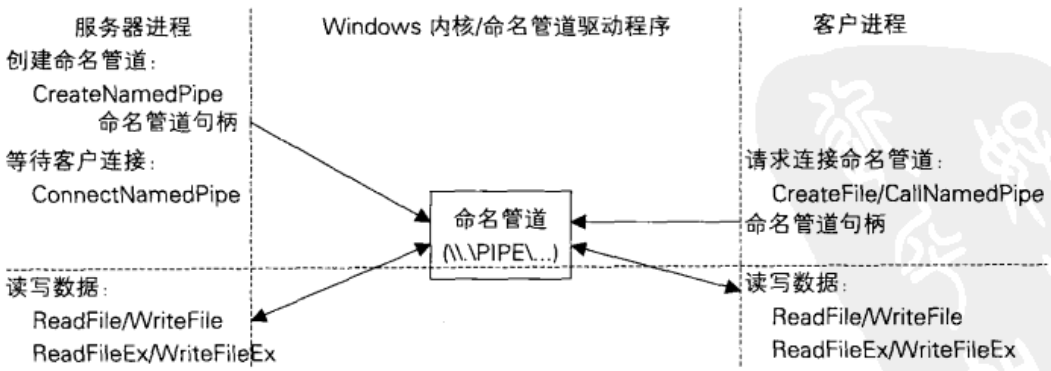


图 8.9 命名管道通信模型

图 8.9 的上面部分体现了服务器与客户建立命名管道的过程。服务器进程调用 API 函数 CreateNamedPipe 来创建一个有名称的命名管道，正如上一小节所讲，它必须指定一个本地的命名管道名称。Windows 允许同一个名称有多个命名管道实例，所以，服务器进程在调用 CreateNamedPipe 函数时必须指定最大允许的实例数（1~255）。该函数返回以后，服务器进程得到一个指向一个命名管道实例的句柄。然后，服务器进程调用 API 函数 ConnectNamedPipe，以等待客户的连接请求，该函数既支持同步形式（阻塞），也支持异步形式（要求在创建命名管道时指定了 FILE_FLAG_OVERLAPPED 标志）。若服务器进程以同步形式调用 ConnectNamedPipe 函数，那么，当该函数返回时，客户与服务器之间的命名管道连接已经建立起来。

客户进程通过调用 CreateFile 函数连接到一个正在等待连接的命名管道上。由于客户指定了命名管道的名称，所以，内核中的对象管理器会把此请求转给 MUP（若是远程命名管道），或者交给本机的命名管道文件系统驱动程序（npfs.sys）。当 CreateFile 成功返回以后，客户进程得到一个指向已建立连接的命名管道实例的句柄。至此，服务器进程的 ConnectNamedPipe 函数也已完成其建立连接的任务。

因此，每个已建立连接的命名管道实例总是伴随着一对服务器进程和客户进程，在这两个进程中分别有一个句柄指向该管道实例。这两个句柄也被称为命名管道的服务器端句柄和客户端句柄。服务器进程可以调用 DisconnectNamedPipe 函数，将一个管道实例与当前建立连接的客户进程断开，从而可以重新连接到新的客户进程。无论是服务器进程还是客户进程，总是可以调用 CloseHandle 来关闭一个已建立连接的命名管道实例。

客户进程除了直接调用 CreateFile 来建立管道连接以外，还可以调用 WaitNamedPipe 函数来测试指定名称的管道实例是否可用。该函数等待一段指定的时间间隔，若在此间隔中，服务器进程正在执行一个 ConnectNamedPipe 调用，那么，客户端的 WaitNamedPipe 调用就会完成并返回一个非 0 值，从而客户可以立即调用 CreateFile 来创建命名管道实例。否则，WaitNamedPipe 的等待超时，返回 0，表明该命名管道当前无法建立实例连接。

一旦服务器和客户进程建立了命名管道实例连接，它们就可以通过文件接口来读写该命名管道实例，即调用 ReadFile/WriteFile 或 ReadFileEx/WriteFileEx 函数，从管道中读取数据或者向管道写入数据，如图 8.9 的下面部分所示。对于每一方而言，针对命名管道的读写操作就如同文件操作一样。值得一提的是，命名管道支持字节模式和消息模式，服务

器进程在创建命名管道时指定它的默认模式。一旦建立连接以后，服务器和客户进程都可以调用 SetNamedPipeHandleState 函数来设置命名管道实例的读操作模式和等待模式。模式的设置是针对命名管道一端的。若服务器端指定了管道为字节模式，则客户端不能设置消息模式；反之，若服务器端指定了消息模式，则客户端允许将它这一端设置为字节模式。模式的设置会影响设置端的读操作行为。

Windows 的命名管道设施提供了两个包装函数：TransactNamedPipe 和 CallNamedPipe。TransactNamedPipe 函数只能用于消息模式的命名管道，它将一个写消息操作和读消息操作合并为一个网络操作，从而提高客户与远程服务器之间的网络通信性能。客户进程和服务器进程都可以使用 TransactNamedPipe 函数。CallNamedPipe 函数是一个更大范围的功能包装函数，仅用于客户进程。该函数的功能等价于：先调用 CreateFile 函数建立命名管道实例连接，然后调用 TransactNamedPipe 函数来发送和接收消息，最后调用 CloseHandle 函数关闭命名管道实例。关于以上提到的命名管道函数的用法，MSDN 中有详细的文档和例子[MSDN-NP]，读者可以参考这些例子中的代码。

8.3.3 命名管道的实现

现在我们来看 Windows 中命名管道机制的实现。支持远程连接是命名管道功能强大的地方，不过本节的讨论仅限于命名管道服务器和客户进程位于本地系统的情形，也就是说，客户进程连接的是一个本地的命名管道，因而对象管理器在解析该命名管道的名称时无须触及 MUP 驱动程序。

首先看服务器创建命名管道的过程，API 函数 CreateNamedPipe 将创建命名管道的请求交给内核的系统服务 NtCreateNamedPipeFile。正如 8.3.1 节所讲，NtCreateNamedPipeFile 接收到的命名管道的名称为“\??\PIPE\<PipeName>”。NtCreateNamedPipeFile 函数的代码位于 base\ntos\io\iomgr\create.c 文件中，它只是简单地把函数的参数组织到一个数据结构 NAMED_PIPE_CREATE_PARAMETERS 中，然后调用 IoCreateFile 函数来完成创建任务。IoCreateFile 有一个 CREATE_FILE_TYPE 参数，指明了要创建的文件类型：普通的文件、命名管道或者邮件槽。6.2.3 节已经描述过文件对象的创建过程：IoCreateFile 调用 IopCreateFile 函数，而 IopCreateFile 调用对象管理器的 ObOpenObjectByName 函数来完成创建工作。

ObOpenObjectByName 函数进一步调用 ObpLookupObjectName 函数来创建目标对象。ObpLookupObjectName 判断命名管道的名称以“\??\”为前缀，于是从进程对象的

DeviceMap 域的 DosDeviceDirectory 成员开始解析余下的名称。关于“\??\”前缀的解析过程，参见 7.3.2 节。

经过“\??\”前缀的解析以后，ObpLookupObjectName 接着从“\Global??”目录中解析“PIPE”，这是一个指向设备对象“\Device\NamedPipe”的符号链接对象。符号链接对象的数据类型为 OBJECT_SYMBOLIC_LINK，其 Parse 方法为 ObpParseSymbolicLink。因而 ObpLookupObjectName 调用 ObpParseSymbolicLink 函数来解析余下的名称“<PipeName>”。ObpParseSymbolicLink 函数返回 STATUS_REPARSE，告诉 ObpLookupObjectName 重新解析新的名称“\Device\NamedPipe\<PipeName>”。于是，ObpLookupObjectName 又重新从根目录开始解析，后面的过程如同 6.2.3 节所介绍的那样，在根目录中查找“Device”项，然后在“Device”目录中查找“NamedPipe”项，因而找到设备对象“\Device\NamedPipe”。接着调用设备对象的 Parse 方法 IopParseDevice，将解析命名管道的任务交给它。

在 IopParseDevice 函数中，它调用 ObCreateObject 函数创建一个文件对象（FILE_OBJECT），然后通过 IoCallDriver，向设备对象的驱动程序 npfs.sys 发送一个 IRP_MJ_CREATE_NAMED_PIPE 请求，将创建命名管道的任务交给驱动程序 npfs.sys。

命名管道驱动程序的初始化函数创建设备对象“\Device\NamedPipe”，并设置驱动程序对象的 I/O 分发例程，包括 IRP_MJ_CREATE_NAMED_PIPE 请求的处理例程。虽然在系统中它也被当做一个文件系统驱动程序，但是，它不需要像真正的文件系统驱动程序那样调用 IoRegisterFileSystem 来注册自身。所以，I/O 管理器在挂载卷时不会来打扰命名管道驱动程序，毕竟命名管道驱动程序不跟任何一个卷关联。尽管如此，npfs.sys 也实现了一套层次状的名称管理。由于它不需要像 fastfat.sys 或 ntfs.sys 那样维护磁盘卷的格式，以及为每个文件或文件中的流维护行串（run）的 VCN-LCN 映射关系，所以，npfs.sys 的实现相对简单很多。

Npfs.sys 的内部实现如同文件系统驱动程序一样，有一个全局的 VCB 和根目录的 DCB，每个管道对象有一个 FCB，命名管道的每个实例有一个 CCB，每个 CCB 有两个文件对象指针，分别指向该实例的服务器端和客户端文件对象。因为每个命名管道对象可以有多个实例，所以 FCB 有一个链表成员，将它的所有实例的 CCB 链接起来。为了提供双向数据传输能力，每个 CCB，即命名管道实例，包含两个队列结构，用做数据传输的缓冲区。WRK 并没有包含命名管道驱动程序的代码，但是，不难想象这是一个简化的文件系统驱动程序，如图 8.10 右侧所示。

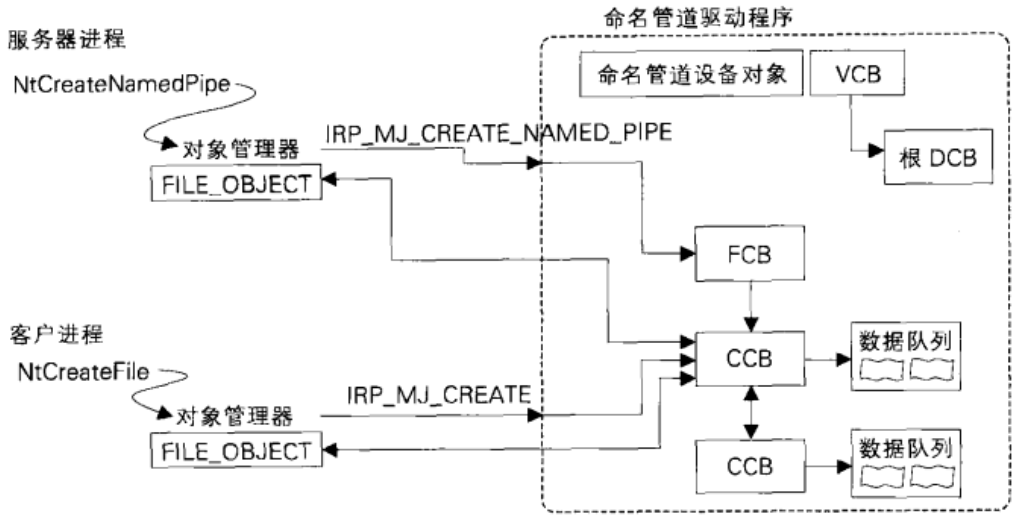


图 8.10 命名管道的实现示意图

图 8.10 的左侧显示了服务器进程和客户进程创建并连接命名管道的示意图。服务器进程通过系统服务 NtCreateNamedPipe 来创建命名管道，如本节前文所述。它通过对象管理器，向命名管道驱动程序发送 IRP_MJ_CREATE_NAMED_PIPE 请求。Npfs.sys 接收到该请求时，若这是一个新的命名管道，则创建一个 FCB 和一个 CCB，并初始化 FCB，以及 CCB 中有关服务器端的状态；若这是一个已有的命名管道，则 npfs 创建一个 CCB，将其加入到 FCB 的 CCB 链表中，并初始化该 CCB 中的服务器端状态。

当客户进程通过系统服务 NtCreateFile 来连接命名管道时，它通过对象管理器，向命名管道驱动程序发送 IRP_MJ_CREATE 请求。Npfs.sys 接收到此请求时，通过解析指定的名称，找到 FCB，然后找到一个处于监听状态的 CCB，并初始化它的客户端状态信息，最后将它置成“已建立连接”的状态。

服务器进程和客户进程之间的命名管道实例一旦建立起来，则任何一方都可通过标准的文件读写服务来操纵命名管道。内核中标准的文件读写服务，例如 NtReadFile 和 NtWriteFile 函数，根据文件对象的指示，向驱动程序发送 IRP_MJ_READ 或 IRP_MJ_WRITE 请求。Npfs 驱动程序利用 CCB 中的两个数据队列来实现基于管道实例的读写操作，并且支持字节模式和消息模式。这也正说明了命名管道实例实际上是一个内存中的文件对象，供不同程序之间传递数据或消息。

以上讨论了命名管道的基本实现框架，命名管道的真正实现还涉及更多复杂的内容，包括读写数据的缓冲区管理、命名管道的安全检查等。其中绝大多数细节是文件系统驱动程序必须要面对的，有关这些细节的处理技术，读者不妨参考 Windows DDK 中提供的文件

系统驱动程序例子，包括 fastfat.sys 和 cdfs.sys。最后，表 8.4 列出了 WRK 随带的 Windows Server 2003 SP1 虚拟系统在启动过程中创建的命名管道。

表 8.4 Windows Server 2003 SP1 启动过程中创建的命名管道

TerminalServer\AutoReconnect	net\NtControlPipe0
InitShutdown	net\NtControlPipe7
Lsass	net\NtControlPipe8
protected_storage	net\NtControlPipe9
Ntsvcs	net\NtControlPipe10
Scerpc	Keysvc
net\NtControlPipe1	PCHHangRepExecPipe
net\NtControlPipe2	net\NtControlPipe11
net\NtControlPipe3	Srvsvc
Winsock2\CatalogChangeListener-290-0*	net\NtControlPipe12
net\NtControlPipe4	Trkwks
net\NtControlPipe5	W32TIME_ALT
net\NtControlPipe6	Winsock2\CatalogChangeListener-300-0*
Winsock2\CatalogChangeListener-1b0-0*	PIPE_EVENTROOT\CIMV2SCM EVENT PROVIDER
Epmapper	Browser
wkssvc	

注：* 这 3 个 Winsock2 的命名管道的名称每次引导时其尾部的数字可能与表中的不同。

8.4 邮件槽（Mailslot）服务

邮件槽是 Windows 提供的另一种跨进程通信手段，与命名管道不同的是，它提供的是不可靠的单向数据传输服务，并且支持广播。如同命名管道服务一样，邮件槽服务也与 Windows 的文件对象管理与安全检查机制紧密地集成在一起。这一节我们首先讨论邮件槽的名称解析，接着介绍邮件槽的通信模型以及在 Windows 中的实现。

8.4.1 邮件槽的名称解析

邮件槽的名称解析过程几乎与命名管道完全相同，只不过名称中的 PIPE 被替换成了 Mailslot。服务器只能创建本地的邮件槽，邮件槽的名称遵从 UNC 命名规范，服务器

可指定的有效邮件槽名称为“\\.\Mailslot\<MailslotName>”，这里“\\.\”代表本地计算机，<MailslotName>是本地机器上邮件槽的唯一名，它可以是层次结构中的一个路径名。

邮件槽客户也使用类似的名称“\\<Server>\Mailslot\<MailslotName>”来指定本地的或远程的邮件槽，这里“<Server>”是邮件槽服务器所在计算机的名称，可以是 DNS 名、NetBIOS 名或者 IP 地址。由于邮件槽支持广播传输，所以，客户在打开一个邮件槽时可以指定一组满足特定条件的邮件槽，以便向它们发送广播消息。在 Windows 中，客户可以按照以下 4 种形式来指定邮件槽：

- “\\.\Mailslot\<MailslotName>”，指定本地系统中的邮件槽<MailslotName>。
- “\\<Server>\Mailslot\<MailslotName>”，指定服务器<Server>上的邮件槽<MailslotName>。
- “\\<DomainName>\Mailslot\<MailslotName>”，指定在域<DomainName>中所有名为<MailslotName>的邮件槽。
- “*\Mailslot\<MailslotName>”，在当前主域中所有名为<MailslotName>的邮件槽。

前两种形式不难理解，如同命名管道中的本地命名管道或远程命名管道一样；后两种形式与 Windows 域有关，其名称解析过程将涉及与 Windows 域控制器的通信。实际上，邮件槽的这一特性也正好为域管理提供了方便，例如，邮件槽的一个典型用法是域内时间同步服务，时间服务器每隔一定时间在域范围内广播一次源时间，域内的机器接收到源时间消息后，可以校正本地系统的时钟。尽管邮件槽是不可靠服务，但对于像时间同步这样的服务来说，即使丢掉一些广播消息，通常也并不会影响系统的正常运行。

对于远程的邮件槽，其名称解析过程如同远程命名管道一样，也涉及 MUP 和网络重定向器。对于本地邮件槽，服务器端的名称解析与命名管道的解析遵循同样的流程：创建邮件槽的系统服务 NtCreateMailslotFile 接收到的名称为“\??\MailSlot\<MailslotName>”，对象管理器将此名称变换为“\Global??\Mailslot\<MailslotName>”，由于“\Global??\Mailslot”是一个符号链接，指向设备对象“\Device\Mailslot”，所以，对象管理器找到该设备对象，交给它的驱动程序（msfs.sys）来解析剩下的子串并创建邮件槽对象。客户程序调用 CreateFile 来打开一个本地邮件槽，内核中的创建函数通过对象管理器，向邮件槽驱动程序发送 IRP_MJ_CREATE 请求，最终由邮件槽驱动程序来创建邮件槽对象。

8.4.2 邮件槽的通信模型

邮件槽只支持单向数据传输，即服务器等待接收数据，客户发送数据。服务器进程先

调用 CreateMailslot 函数创建一个本地邮件槽，然后，它只需简单地在邮件槽的句柄上执行 ReadFile 函数，就可以监听来自客户的消息。它既可以同步方式，也可以异步方式调用 ReadFile。客户进程调用 CreateFile 函数打开一个邮件槽，然后，它只需在邮件槽的句柄上执行 WriteFile 函数，就可以向一个或多个服务器发送消息。图 8.11 显示了邮件槽的通信模型。

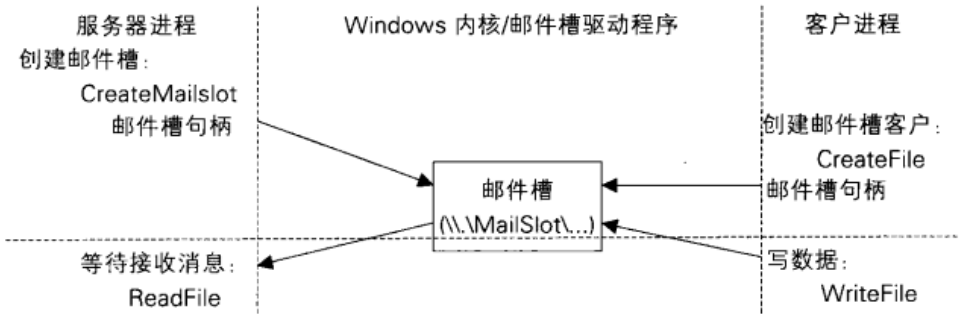


图 8.11 邮件槽通信模型

由于邮件槽实现上的原因，客户端发送的消息，只有当小于 425 字节时，才可以被广播给多个服务器。如果消息长度大于 425 字节，那么，邮件槽将使用可靠的通信机制，这就要求建立起客户/服务器之间的端到端连接，因而在这种情形下邮件槽不支持广播通信。

服务器进程在读消息以前，也可以调用 GetMailslotInfo 函数，获得指定邮件槽的当前状态，包括邮件槽中有多少消息、下一个消息的长度等信息，因而服务器进程可以分配足够大小的缓冲区来读取消息。另外，服务器进程还可以调用 SetMailslotInfo 函数来设置邮件槽读操作的超时值。

8.4.3 邮件槽的实现

邮件槽的实现与命名管道类似，服务器端调用的 CreateMailslot 函数将创建邮件槽的请求交给内核的系统服务 NtCreateMailslotFile。NtCreateMailslotFile 也经历类似于 NtCreateNamedPipeFile 的步骤到达邮件槽驱动程序 msfs.sys。NtCreateMailslotFile 调用 IoCreateFile 函数，然后 IoCreateFile 将创建邮件槽的请求交给 IopCreateFile 函数；在 IopCreateFile 函数中，它调用对象管理器的 ObOpenObjectByName 函数来完成创建工作；ObOpenObjectByName 又进一步调用 ObpLookupObjectName 函数来创建目标对象。ObpLookupObjectName 函数解析邮件槽的名称：将“\??\”解析到“\GLOBAL??”目录，然后在该目录中找到“Mailslot”，这是一个指向设备对象“\Device\Mailslot”的符号链接，

于是，ObpLookupObjectName 函数将邮件槽的唯一名部分交给设备对象的 Parse 方法 IopParseDevice。IopParseDevice 函数创建一个文件对象（FILE_OBJECT），然后通过 I/O 管理器，向该设备对象的驱动程序 msfs.sys 发送一个 IRP_MJ_CREATE_MAILSLOT 请求，将创建邮件槽的工作交给 msfs.sys 来完成。

如同命名管道驱动程序 npfs.sys 一样，msfs.sys 虽然也被看做一个文件系统驱动程序，但它并不与任何一个磁盘卷关联。Msfs.sys 的初始化例程创建设备对象"\Device\Mailslot"，并设置驱动程序对象的 I/O 分发例程，包括 IRP_MJ_CREATE_MAILSLOT 请求的处理函数 MsFsdCreateMailslot。Msfs.sys 实现了一套层次状的名称管理，但是，它并不需要像真正的文件系统那样，维持永久的目录结构以及为每个邮件槽对象维护行串和 VCN-LCN 映射关系，所以，与真正的文件系统驱动程序相比，msfs.sys 的实现要简捷很多。

由于邮件槽的服务器和客户之间没有连接关系，所以，服务器和客户的文件对象实例分别引用（通过文件对象的 FsContext 成员）到邮件槽对象的 FCB 和 CCB，这两个文件对象是松耦合关系。服务器创建的是 FCB 对象，它有一个数据队列成员。服务器只需监视此数据队列就可以实现对邮件槽的读操作。客户通过 NtCreateFile 创建的是 CCB 对象，此 CCB 对象代表该客户创建的邮件槽实例。以后，客户通过此 CCB 来操纵邮件槽。图 8.12 显示了邮件槽的实现示意图。

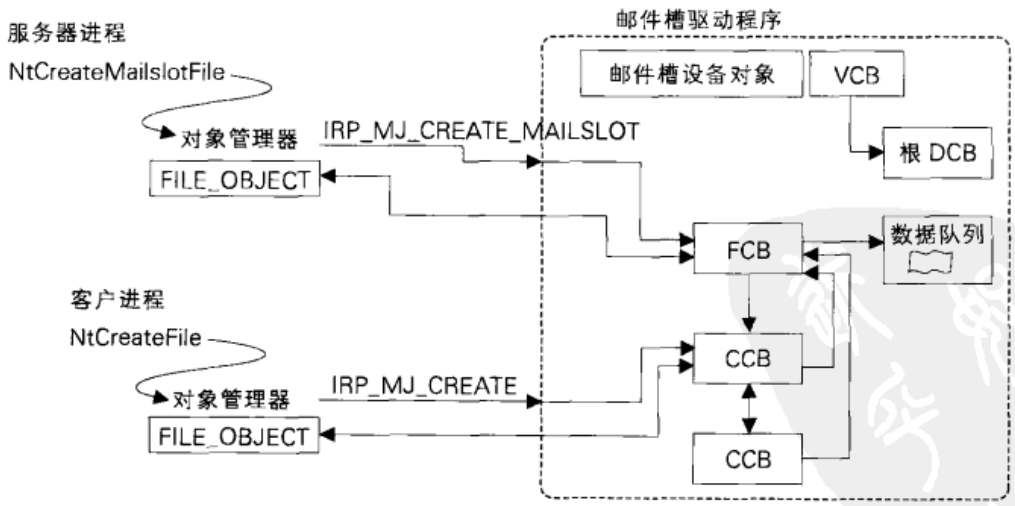


图 8.12 邮件槽的实现示意图

在邮件槽驱动程序内部，VCB 对象是全局的，依附在邮件槽设备对象之后；VCB 有

一个指向根目录的成员，所以，邮件槽驱动程序在接收到创建或查询邮件槽请求时，可以沿着根目录，按层次搜索，直至找到指定的目录或邮件槽对象。每个邮件槽有一个 FCB 对象，它有一个指向服务器端文件对象的成员以及一个 CCB 链表成员。一个邮件槽可以同时有多个客户打开它，每个客户在打开了该邮件槽以后，得到一个指向 CCB 实例的文件对象。这些 CCB 都将被加入到 FCB 的 CCB 链表中。并且，每个 CCB 都指向 FCB，因而当客户向邮件槽写数据时，邮件槽驱动程序根据 CCB 中的 FCB 指针，可以访问到该邮件槽的数据队列。

不像命名管道的 CCB 代表了一对服务器和客户之间的共享对象（包括两个数据队列），邮件槽的 CCB 仅代表一个客户，真正的数据队列位于 FCB 中。服务器和所有的客户共享同一个数据队列，这也正反映了邮件槽的松散通信模型，并且区别于命名管道的基于连接的通信模型。

WRK 包含了邮件槽系统服务 NtCreateMailslotFile 的代码，但并没有提供邮件槽驱动程序的实现代码。然而，基于以上的介绍，读者不难想象邮件槽的实现要点。MSDN 提供了关于邮件槽的文档，以及相应的服务器和客户例子程序[MSDN-MS]。

8.5 SDT 显示工具 SDTViewer

SDTViewer 是一个显示当前系统中 SDT 的工具，它根据 8.1.3 节中介绍的系统服务分发机制，首先定位到系统内部的 SDT，包括系统的基本 SDT 和 Windows 子系统 SDT，然后依次解析表中的每一个系统服务项，并将其在一个友好的用户界面中列示出来，因此，使用 SDTViewer 可以很方便地看到用户模式和内核模式之间的接口层。

8.5.1 SDTViewer 使用介绍

SDTViewer 是一个需要特权才能运行的应用程序，它的功能几乎全部集中在初始化阶段。当 SDTViewer 启动时，它加载一个内核驱动程序，以便访问系统的 SDT。如果运行 SDTViewer 的系统能够访问 Microsoft 公开的符号服务，那么，SDTViewer 将可以显示 Windows 的基本 SDT 和子系统 SDT；否则，SDTViewer 仅显示基本 SDT。图 8.13 是 SDTViewer 正在显示 Windows 基本 SDT 和子系统 SDT 的界面。

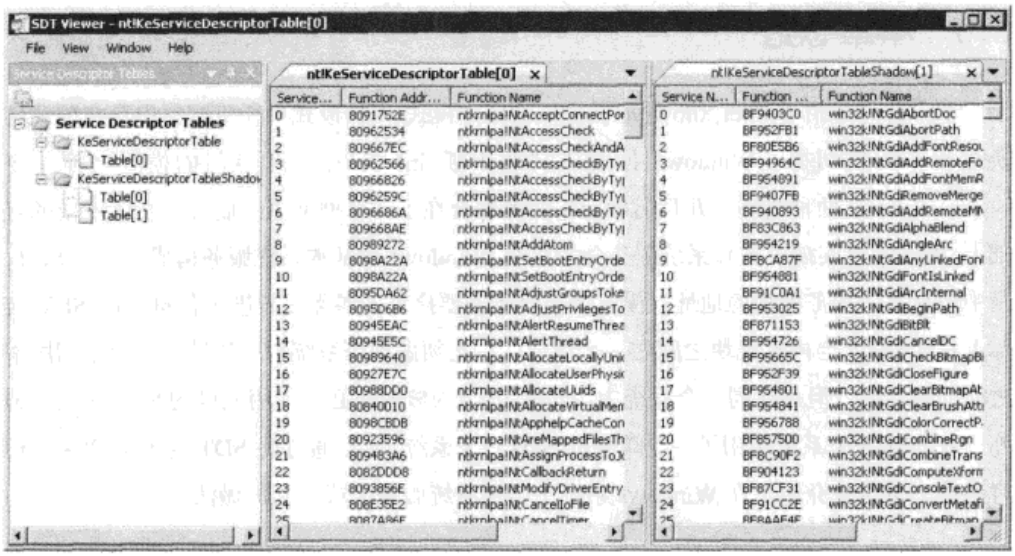


图 8.13 SDTViewer 显示基本 SDT 和子系统 SDT（在 Windows Server 2003 SP1 系统上）

8.5.2 SDTViewer 实现原理

SDTViewer 的原理相对比较简单，它在初始化阶段，通过一个内核驱动程序，首先定位到内核全局变量 KeServiceDescriptorTable。由于这是一个导出变量，所以，内核模块的导出表中已经包含了它的地址。Windows 的基本 SDT 即为 KeServiceDescriptorTable[0]，SDTViewer 根据 8.1.3 节描述的 KSERVICE_TABLE_DESCRIPTOR 数据结构，可以依次解析 SDT 中的每一个系统服务项，包括它的服务名称、参数块长度等信息。

接下来 SDTViewer 定位到 KeServiceDescriptorTableShadow。这是一个非导出的全局变量，因此，除非使用一些启发式的方法在内核模块的系统内存区找到它的位置，否则，我们就要依赖于内核模块的符号信息。对于标准的 Windows 版本，可通过 Microsoft 提供的符号服务来获得相应的符号信息。只要运行 SDTViewer 的系统能连接 Microsoft 的符号服务，则它就可以获得 KeServiceDescriptorTableShadow 的地址，进而获得 Windows 子系统 SDT，即 KeServiceDescriptorTableShadow[1]。如果当前系统是 WRK 内核，那么，只要 WRK 编译得到的.pdb 文件位于符号搜索路径中，则 SDTViewer 就会使用 WRK 内核的符号信息。得到了 KeServiceDescriptorTableShadow[1]以后，SDTViewer 依次解析每一个系统服务项。类似地，为了解析每一个服务项的名称，需要用到 win32k.sys 的符号信息。

如果 SDTViewer 无法定位到 KeServiceDescriptorTableShadow 变量，那么，它仅显示 Windows 的基本 SDT。

8.6 本章总结

本章首先介绍了 Intel x86 处理器提供的用户模式-内核模式切换机制，以及在模式切换过程中的栈切换。Windows 同时支持软中断 int 2e/iretd 和专门的模式切换指令 sysenter/sysexit 两种机制，并且巧妙地将两者融合在了一个框架中。而且，在该框架的基础上，Windows 实现了一套系统服务分发机制。Windows 的基本系统服务构成了一个 SDT，其中包括每个服务例程的地址、模式切换时所需要拷贝的参数区长度等信息。此 SDT 实际上是 ntdll.dll 与内核模块之间的一个约定，它无须做到跨系统的一致性，以及不同版本之间的兼容性，但是在同一个系统中，ntdll.dll 与内核模块必须使用同样的 SDT 定义。另外，Windows 子系统使用了一个单独的 SDT，子系统 DLL 通过此 SDT 与 win32k.sys 进行通信。本章也介绍了在 Windows 系统中增加系统服务表或表项的做法。

本章还介绍了 Windows 中三个重要的系统服务：LPC、命名管道和邮件槽服务。LPC 是同一个系统中不同进程之间进行通信的重要手段，它直接由内核提供支持，非常高效。Windows 的 LPC 利用了操作系统提供的基本同步能力，允许两个进程相互之间建立连接和传输数据。内核组件和 Windows 子系统进程利用 LPC 来实现协作和数据通信。

命名管道服务是 Windows 支持的系统间通信机制，它提供了双向通信的能力，与 Windows 的文件对象管理和安全机制紧密地集成在一起。服务器进程和客户进程一旦建立命名管道连接，两者就可以像读写文件一样操纵命名管道。在 Windows 中，命名管道的名称解析依赖于对象管理器的名称解析能力，而命名管道的数据传输则通过文件接口和命名管道文件系统驱动程序（npfs.sys）来完成。

邮件槽也是 Windows 支持的系统间通信机制，它提供不可靠的单向数据传输服务，并且支持广播。其名称解析与命名管道对象的解析过程类似，但它的数据传输服务相对要简单很多。Windows 通过一个邮件槽文件系统驱动程序（msfs.sys）来提供邮件槽的数据传输功能。同样地，应用程序在使用邮件槽对象时，也通过文件接口来发送或接收数据。

本章最后介绍了一个显示系统服务表的工具 SDTViewer，它利用可获得的符号信息，将用户模式与内核模式之间的系统服务接口以直观友好的方式呈现出来。

9

Windows 系统高级话题

Windows 是一个复杂的操作系统，本书前面的章节仅仅介绍了 Windows 内核中最核心的一些组件和机制。作为一个完整的操作系统，Windows 中还有更多的功能模块在发挥作用。在这一章中，我们将简要地介绍三个重要的系统部件：网络、Windows 子系统，以及 Windows 内置的内核日志机制。本书前面的内容一直以 Windows Server 2003/XP 的内核作为参照，但自从 WRK 发布以来，Windows 已经历了两次重要的升级：Windows Vista 和 Windows 7（也伴随着 Windows Server 2008 的发布），因此，在本章最后，我们将讨论 Windows 最新版本中的一些重要变化。

9.1 网络

Windows 从设计之初就考虑了对于网络的支持，它有一个定义良好的结构，容纳了各种网络特性，并且与 Windows 的 I/O 系统和标准 API 紧密地集成在一起。从最上层的 Windows 网络应用程序，一直到内核模式的网络设备驱动程序，中间还包括网络 API、网络服务，以及各种网络协议，自上而下构成了一个网络栈。Windows 为网络栈中的每一层都定义了良好的接口，从而允许第三方网络模块融入到网络栈中。本节我们首先介绍网络栈的总体结构，然后依次讨论网络栈中两个重要的接口规范：TDI 和 NDIS，最后讨论在 Windows Vista 及以后的版本中网络体系结构的变化。

9.1.1 Windows 网络体系结构

尽管我们可以从用户模式和内核模式的角度来看待 Windows 对于网络的支持，但是，

这样的划分并不能反映出 Windows 网络结构的本质。另外，Windows 的网络支持并不完全遵从于国际标准化组织（ISO，International Organization for Standardization）定义的 7 层 OSI 参考模型，或者如计算机网络教科书中所倡导的 5 层参考模型[COMP-NW]。Windows 的网络结构同样采用了垂直分层模型，并且在每一层上既定义了接口规范，也提供了系统模块来支持上下层之间的通信，这些系统模块或者以系统 DLL 的形式被加载到网络进程空间中，或者以内核模式驱动程序的形式被加载到系统地址空间中。图 9.1 显示了 Windows 网络体系结构，这一垂直结构也被称为 Windows 的网站栈。第三方软件厂商可以在恰当的层次上插入自己的模块，或者扩充 Windows 的网络支持，或者截取系统的网络流量，以实现一些网络附加服务，比如防火墙、网络流量分析等。

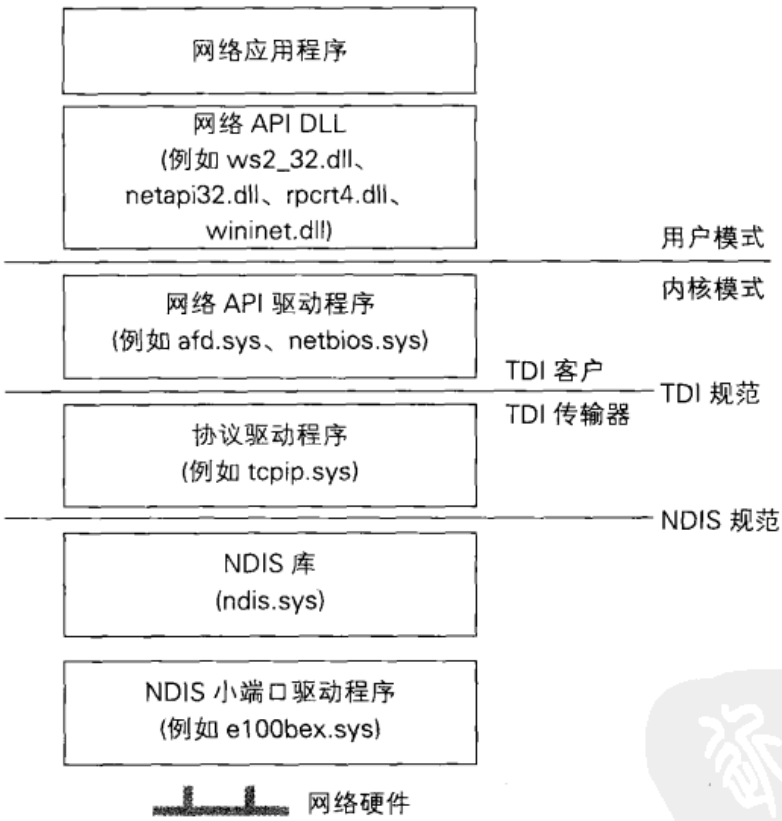


图 9.1 Windows 网络体系结构

Windows 应用程序可以通过多个网络 API 进行网络通信，这一方面是为了兼容老式的、遗留的应用程序，另一方面也是为了为不同的网络应用提供最为方便的网络接口。因此，不同类型的网络应用，应该选择恰当的网络 API。提供多种选择的好处是显而易见的，但有可能给软件设计人员带来额外的负担，因为他们必须知晓不同网络 API 的优缺点。下面简要地列出 Windows 平台上典型的网络 API：

- Windows 套接字，即 Windows Socket，或简称 Winsock。网络应用的标准通信接口是 BSD 套接字（Berkeley Software Distribution Sockets），Winsock 实现了 BSD 套接字标准，从而便于将 UNIX 网络应用程序移植到 Windows 平台上；此外，Winsock 2.0 版本还扩充了大量新特性，包括异步网络 I/O、服务质量（QoS）规范、允许扩充额外的协议、可扩展的名字空间，以及支持多点传输消息等。
- WinInet。这是一个高层网络 API 组件，支持多个协议，包括 Gopher、FTP 和 HTTP。Microsoft 的浏览器 Internet Explorer 使用 WinInet 来实现数据传输。WinInet 中的 FTP 部分提供了在 FTP 服务器上列举目录、创建目录和文件，以及更改文件名称等能力，以方便应用程序操纵 FTP 资源；WinInet 的 HTTP 部分提供了 cookie 管理、拨号连接、客户端缓存，以及自动代理和异步传输等能力。
- 命名管道和邮件槽。如第 8 章所介绍，命名管道和邮件槽可以用于不同进程之间进行通信，这些进程可以在同一个系统上，也可以位于不同的系统上。命名管道和邮件槽都使用了文件接口，所以应用程序可以以文件操作的方式来进行通信。命名管道支持连接方式的通信模型，客户进程必须先与服务器进程建立连接，然后才能与之通信；邮件槽支持非连接方式的通信模型，客户进程可以发送广播消息。邮件槽的消息传输是不可靠的，常用于一些非关键的网络服务。
- NetBIOS。这是一个早期的网络编程 API，Windows 支持 NetBIOS 是为了兼容老的应用程序。每台机器被分配一个 NetBIOS 名称，服务器和客户通过 NetBIOS 名称进行通信。传统上，NetBIOS 的名称解析（为了获得计算机的 IP 地址）依赖于 Windows Internet 名称服务（WINS）或者本地广播消息来完成。现在 Windows 主要依赖于 DNS 来解析计算机名称。NetBIOS 既支持有连接的通信，也支持无连接的通信。
- RPC。这是网络编程的一个标准，往往是分布式系统基础设施中的一个重要组件。RPC 建立在其他的网络 API 基础之上，比如命名管道或者 Winsock，它试图将网络传输的细节隐藏起来，让应用开发人员在编写基于网络的过程化代码时，无须涉及网络传输等琐碎的细节，而只需利用 RPC 设施，就好像调用本进程中的过程一样来调用远程过程。Windows 的 RPC 也支持异步调用方式，因而发起调用的线程不必阻塞和等待结果，而可以当服务器完成了被调用的过程时，由 RPC 运行库通知客户线程此调用已完成。

以上这些网络 API 都可以直接被应用程序使用。当网络 API 的用户模式 DLL 接受了应用程序的网络请求后，它必须将这些请求传递给内核模式的相应驱动程序。典型情况下可有两种途径来完成这一从用户模式到内核模式的传递任务：

- 通过内核中与特定网络 API 相关的系统服务，如上一章介绍的 NtCreateNamedPipeFile 和 NtCreateMailslotFile，在 API 调用中指定远程命名管道或邮件槽，然后对象管理器将名称解析到设备对象“\Device\Mup”上，由 MUP 驱动程序进一步解析名称，定位到适当的 UNC 提供者驱动程序（也称为网络重定向器），并通过协议驱动程序发送远程的网络请求。
- 通过标准的文件接口，如 NtReadFile、NtWriteFile，以及 NtDeviceIoControlFile 系统服务，直接与内核驱动程序打交道。例如，当 NetBIOS API 函数接收到应用程序的请求时，它们通过 NtDeviceIoControlFile 将请求传递给 NetBIOS 驱动程序。

Winsock 是 Windows 中最重要的网络 API，它并非简单地由一个用户模块 ws2_32.dll 加上内核中的驱动程序构成；相反地，它定义了一套可扩展的框架，允许第三方在框架中加入传输服务提供者（transport service provider）和名字空间服务提供者（namespace service provider）。顾名思义，传输服务提供者的职责是，提供诸如建立连接、传送数据、控制数据流、错误控制等与传输服务相关的功能；而名字空间服务提供者则是将网络协议所使用的地址规范与一个或多个友好的名称关联起来，它解决的是名称解析或地址映射的问题。Ws2_32.dll 通过 SPI（服务提供者接口）与这些服务提供者进行通信；传输服务提供者可以按层次叠加起来，形成 LSP（Layered Service Provider，层次的服务提供者）模型。图 9.2 显示了 Winsock2 的结构图。

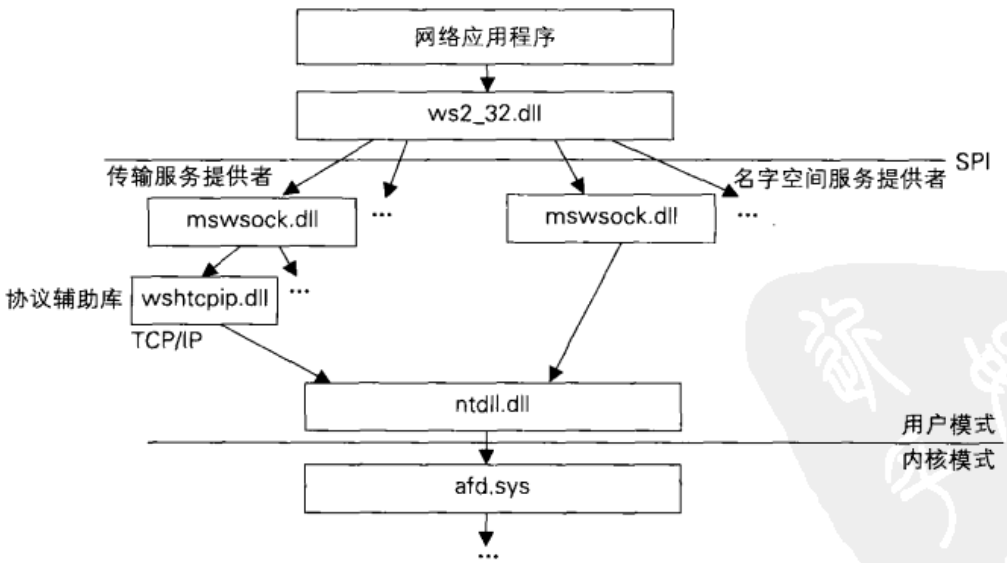


图 9.2 Winsock2 结构图

Winsock 的 ws2_32.dll 提供了相应的函数以允许动态地加载传输服务提供者和名字空

间服务提供者。Winsock 默认支持的协议由传输服务提供者 mswsock.dll 来提供，mswsock 又借助于一组协议辅助库来跟内核模式的协议驱动程序打交道，例如，针对 TCP/IP 协议的辅助库为 wshtcpip.dll。Windows 还提供了其他一些协议的辅助库，包括 IPX/SPX、AppleTalk、ATM 和红外协议（IrDA）等。另外，mswsock.dll 也是一个名字空间服务提供者，Winsock 通过它实现了以下协议的名字空间服务：DNS、IPX/SPX 和活动目录。

Winsock 默认提供的传输服务和名字空间服务都通过内核模式驱动程序 afd.sys 真正实现网络通信（这里 AFD 代表 Ancillary Function Driver，即辅助功能驱动程序）。所以，mswsock 和协议辅助库都通过 ntdll.dll 的文件操作函数，例如 NtCreateFile、NtReadFile、NtWriteFile 和 NtDeviceIoControlFile，进入到内核中并与 afd.sys 通信。

Winsock2 是一个成熟、稳定的网络 API，是在 Windows NT 4.0 中引入的，在此后的所有 Windows 版本（包括 Windows 9x 系统）上都可以使用。有关 Winsock2 的详细资料，可以参考 MSDN 中的介绍[MSDN-WINSOCK2]；关于 SPI 和 LSP，读者可参考有关资料[MSJ-LSP]。

9.1.2 TDI（传输驱动程序接口）

我们已经知道，Winsock 以网络协议为基础来访问网络，比如 TCP/IP 和 IPX/SPX 等，其相应的内核驱动程序为 afd.sys。那么，像 afd.sys 这样的网络 API 驱动程序在接收到针对特定协议的网络请求时，如何将请求转交给相应的协议驱动程序呢？它如何适应不同网络协议的实现差异呢？Windows 在网络 API 驱动程序与协议驱动程序之间规定了一个接口标准，称为传输驱动程序接口（TDI，Transport Driver Interface），从而使网络 API 驱动程序可以按照 TDI 接口来调用更低层的协议驱动程序。

定义 TDI 带来的好处是显而易见的，它消除了网络 API 与低层的传输协议之间的紧耦合关系。按照 TDI 接口来实现的传输协议可以被多个 TDI 客户使用。例如，TCP/IP 协议驱动程序为 tcpip.sys，既可以被 Winsock 的驱动程序 afd.sys 使用，也可以被 netbt.sys 驱动程序（NetBT 代表 NetBIOS over TCP/IP）使用。符合 TDI 接口的传输协议驱动程序被称为 TDI 传输器（TDI Transport）。

在 TDI 这一层上，网络 API 的内核驱动程序是 TDI 客户，而协议驱动程序则是 TDI 传输器。网络 API 驱动程序接收各种形式的网络请求，例如，名称解析、建立连接、发送或接收数据等，TDI 使得这些网络请求被统一描述，因而各种传输协议只要按照 TDI 的规范来实现，就可以被网络 API 使用。另外，由于 TDI 传输器也是内核驱动程序，它接收的请求一定是 IRP 数据结构，所以，本质上，TDI 是将各种网络请求转变成一种规范的 IRP 描述。

在 Windows 中，内核模块 tdi.sys 提供了一组支持函数，从而将 TDI 客户和 TDI 传输器联系起来。这些支持函数，以及 TDI 客户和传输器在通信时用到的一些数据结构和宏定义，构成了 TDI 接口。有关 TDI 接口的这些定义，请参考 MSDN 中的相关文档[MSDN-TDI]或 Windows DDK 中的头文件 tdikrnl.h。为了便于理解 TDI，表 9.1 列出了 TDI 为传输器定义的一组控制码（TDI IOCTL），这可以看做是 TDI 在 I/O 管理器定义的 I/O 模型基础上扩展的 I/O 控制码。通过这一组 I/O 控制码，我们可以大略地理解，TDI 的基本通信模型与 Winsock 非常相似。在客户端，为了建立一个连接，TDI 客户需要发出 TDI_CONNECT 请求，这类似于在 Winsock 中客户通过 connect 函数建立连接。而在服务器端，TDI 客户首先通过 TDI_ASSOCIATE_ADDRESS 请求，将本地节点地址与一个连接端点关联起来，这类似于在 Winsock 中服务器调用 bind 函数；然后，TDI 客户发出 TDI_LISTEN 请求，使该连接端点进入监听状态，再进一步发出 TDI_ACCEPT 请求，以等待连接请求。这后面的两个步骤类似于 Winsock 中服务器调用 listen 和 accept 函数。

表 9.1 TDI IOCTL 设备控制码

TDI I/O 控制码	说明
TDI_ACCEPT	TDI 客户发出此 TDI 请求，请低层的 TDI 传输协议驱动程序接受一个进入的连接，从而可以在本地机器和远程机器之间进行端到端的数据传输
TDI_ACTION	TDI 客户发出一个由传输器来定义和解释的动作请求
TDI_ASSOCIATE_ADDRESS	TDI 客户请低层传输器，将一个本地节点地址与一个连接端点关联起来
TDI_CONNECT	TDI 客户通过此 TDI 命令，请低层传输器与一个远程节点建立一个连接
TDI_DISASSOCIATE_ADDRESS	TDI 客户请低层传输器，将一个本地节点地址与一个连接端点之间的关联关系解除
TDI_DISCONNECT	TDI 客户通过此命令，指示 TDI 传输器断开与远程节点的连接，或者拒绝远程节点的连接请求
TDI_LISTEN	TDI 客户通过此命令，请低层传输器监听来自远程节点的端到端连接请求
TDI_QUERY_INFORMATION	TDI 客户要求低层传输器返回客户指定的信息，比如广播地址、传输器的能力、在特定连接上的 I/O 统计信息等
TDI_RECEIVE	TDI 客户要求低层传输器，在一个已建立的连接上接收远程节点的数据
TDI_RECEIVE_DATAGRAM	TDI 客户要求低层传输器，在一个指定的地址上接收数据作为一个数据报
TDI_SEND	TDI 客户要求低层传输器，在一个已建立的连接上向远程节点发送数据

续表

TDI I/O 控制码	说明
TDI_SEND_DATAGRAM	TDI 客户要求低层传输器，向一个指定的地址发送数据作为一个数据报
TDI_SET_EVENT_HANDLER	TDI 客户通过此命令，指示低层的传输器当特定的网络事件发生时调用指定的例程
TDI_SET_INFORMATION	TDI 客户请低层传输器，在特定的地址、连接或控制通道上设置指定的信息

除了基于连接的通信模型，TDI 也支持无连接的通信模型，这反映在 TDI_SEND_DATAGRAM 和 TDI_RECEIVE_DATAGRAM 两个请求上。TDI 客户可以在不建立连接的情况下，直接向远程节点发送数据，或者在指定的地址上接收数据。

在 TDI 接口中，另外值得一提的是 TDI_SET_EVENT_HANDLER 请求，TDI 客户可以向低层的传输器注册事件通知，即当特定的事件发生时，传输器直接调用 TDI 客户指定的函数。TDI 传输器提供的事件包括四种类型：传输器建立或断开连接、传输器接收到远程数据、传输器能够发送数据，以及传输器或底层网络发生错误，关于这些事件的详细说明，读者可以参考有关文档[MSDN-TDI-EVENT]。例如，当 TDI 传输器接收到来自远程节点的数据时，它可以调用一个已注册的客户通知函数，从而让客户知道有数据到达。这种事件回调通知的一个明显好处是，TDI 客户在接收网络数据时可以更加有效地管理缓冲区，它们可以清楚地知道数据是什么时候进来并可以使用的，甚至可以重用 TDI 传输器的缓冲区。

在现代网络操作系统中，TCP/IP 无疑是最为重要的网络协议，Windows 也不例外。TCP/IP 是一个协议套件，包括多个协议，并支持多种传输模型。Windows 的 TCP/IP 协议的驱动程序是 tcpip.sys，它实现了基于 IPv4（即 32 位 IP 地址）的 TCP/IP 协议。Tcpip.sys 创建了 5 个设备对象：\Device\RawIp、\Device\Udp、\Device\Tcp、\Device\IPMULTICAST 和\Device\Ip。TDI 客户可以通过 TDI 接口与这些设备对象打交道。如果其他的内核驱动程序附载到这些设备对象的设备栈中，那么，它们将有机会监听到相应网络协议的通信流量。

Tcpip.sys 实现了基本的 TCP/IP 协议套件，然而，在实际的网络系统中，仅仅基本的 TCP/IP 协议还不够，因此，Windows 在 tcpip.sys 的外围又提供了一些扩展。这些协议扩展也是内核驱动程序，它们通过一些私有的接口与 tcpip.sys 打交道。图 9.3 显示了 Windows 中 TCP/IP 协议的实现结构。

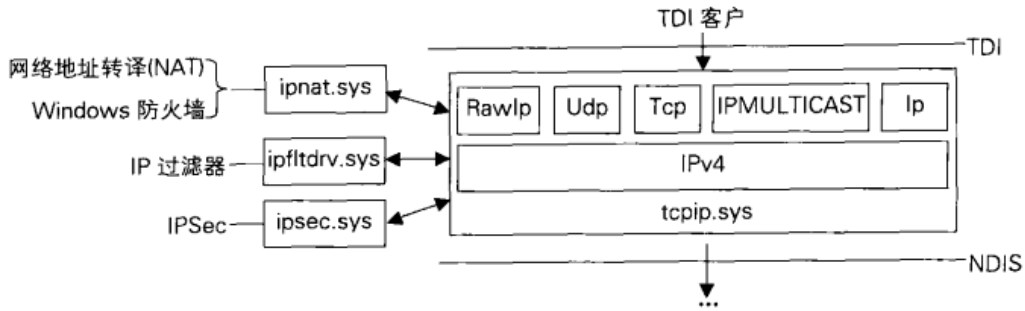


图 9.3 Windows 中 TCP/IP 协议的实现结构

网络地址转译（NAT，Network Address Translation）是 IPv4 解决地址短缺的主要方案，它允许多个私有 IP 地址被映射到单个公共 IP 地址，从而使这些私有 IP 地址所在的系统都可以连接到 Internet。Windows 中的 NAT 是由内核驱动程序 ipnat.sys 来实现的，它与 tcpip.sys 协作，为 TCP/IP 协议栈提供了 NAT 的能力。

Ipnat.sys 除了为 TCP/IP 协议提供地址转译的功能，还提供了防火墙的功能。其做法是，向 TCP/IP 驱动程序注册一个防火墙钩子（firewall hook），因此，当 tcpip.sys 在处理进来的或者出去的 IP 包时，它会调用每一个防火墙钩子的回调函数。如果防火墙钩子回调函数修改一个包的 IP 地址信息，那么，它能够提供地址转译功能；如果它向 TCP/IP 驱动程序返回一个状态代码，指示 TCP/IP 协议接受或丢弃该包，那么，它提供的是防火墙功能。Windows 的防火墙服务实现了一个有状态的防火墙，可以跟踪网络流量，区分本地 LAN 上的网络流量和从 Internet 上闯入的流量。

Tcpip.sys 还提供了另一种专门用于 IP 过滤的机制，以类似于防火墙钩子的交互方式，为每个进来的或出去的数据包执行一个由 IP 过滤驱动程序（ipfltdrv.sys）指定的函数。IP 过滤驱动程序允许系统中的一个驱动程序注册成过滤器钩子驱动程序（filter hook driver），从而在接收到网络流量事件时，可以调用一个由该过滤器钩子驱动程序指定的回调函数，让它参与对网络数据包的修改、拒绝、接受的决策。这一由 IP 过滤驱动程序提供的过滤器钩子功能使得第三方软件也可以在系统中加入地址转译、防火墙、流量监视等常见的网络功能。

TCP/IP 协议的安全性是通过 IPSec 规范来实现的。在 Windows 中，IPSec 是一个涉及操作系统多个方面的组件，而不仅仅简单地通过密码学原理来保护网络流量。Windows 的 IPSec 实现包括三个部分：IPSec 策略引擎、IKE（Internet Key Exchange）协议和 IPSec 驱动程序 ipsec.sys。其中策略引擎和 IKE 分别负责管理 IPSec 策略和协商安全密钥。真正与 TCP/IP 协议驱动程序打交道的是 IPSec 驱动程序 ipsec.sys，它监视和保护 tcpip.sys 驱

动程序发送出去的 IP 数据包，并且也监视、解密和验证接收进来的 IP 数据包。Ipsec.sys 通过策略引擎获得 IPSec 策略，通过 IKE 获得 SA（Security Association 或安全关联，其中包含了通信双方已达成一致的 IPSec 策略设置和密钥组合）。关于 IPSec 协议规范，请参考文档[IPSEC-RFC]；关于 Windows 的 IPSec 配置和用法，请参考 MSDN 文档[MSDN-IPSEC]。

9.1.3 NDIS（网络驱动程序接口规范）

在 Windows 的网络栈中，网络协议与网络适配器是分离的，协议驱动程序并不针对特定的网络适配器而设计，然而，当协议驱动程序真正运行时，它必须通过一个网络适配器才能发送和接收数据。协议驱动程序通过统一的接口与网络适配器驱动程序进行通信，此接口即为 Microsoft 和 3COM 于 1989 年联手开发的 NDIS（Network Driver Interface Specification，网络驱动程序接口规范）。符合 NDIS 的网络适配器驱动程序称为 NDIS 驱动程序，或 NDIS 小端口驱动程序（NDIS miniport driver）。Windows 提供了 NDIS 库，即 ndis.sys，作为协议驱动程序与 NDIS 驱动程序之间的桥梁。随 Windows XP 和 Windows Server 2003 一起发行的 NDIS 库是 NDIS 5.1。

NDIS 客户，即 TDI 传输器，利用 NDIS 库提供的功能，对将要发送给 NDIS 驱动程序的命令进行格式化，并发送给 NDIS 驱动程序；而 NDIS 驱动程序则利用 NDIS 库，接收请求和送回应答。图 9.4 显示了 Windows 实现的 NDIS 结构。

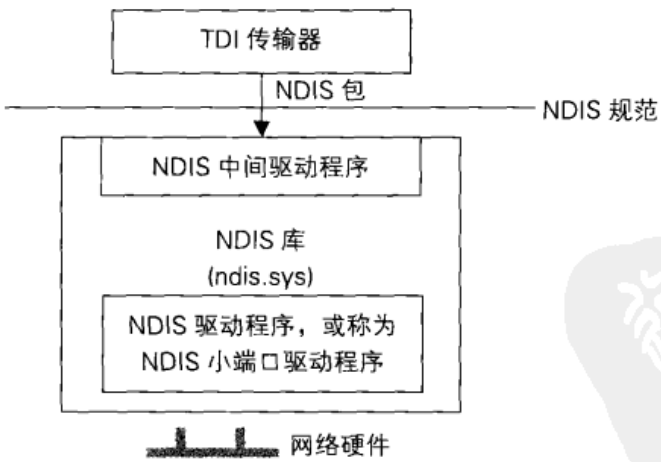


图 9.4 Windows 的 NDIS 结构

NDIS 库为 NDIS 驱动程序提供了一个完全的开发环境，它并没有使用 Windows I/O 管理器的接口规范，而是自己定义了一套开发接口。所以，NDIS 驱动程序不是真正的 Windows 驱动程序，它们并不接收和处理 IRP。相反地，TDI 传输器调用 NDIS 库的函数

（例如 NdisAllocatePacket）来构造一个 NDIS 包（NDIS Packet），再通过 NDIS 库的发送函数（例如 NdisSend）将 NDIS 包传送给 NDIS 驱动程序。NDIS 驱动程序在初始化时向 NDIS 库注册自身，因而当 NDIS 库接收到针对特定 NDIS 驱动程序的请求时，知道将请求转发给它。因此，在 NDIS 的体系结构中，TDI 传输器和 NDIS 驱动程序都要调用 NDIS 库的函数，由于所有这些模块都位于系统地址空间，所以，这些调用关系都是直接进行的。而且，TDI 传输器和 NDIS 驱动程序都要链接（link）NDIS 库。

在 NDIS 体系结构中，有一种特殊的驱动程序起着协议过滤器的作用，这便是 NDIS 中间驱动程序（NDIS Intermediate Driver）。它们位于 TDI 传输器（即协议驱动程序）和 NDIS 驱动程序之间：对于 NDIS 驱动程序来说，它们就好像是一个协议驱动程序；而对于协议驱动程序来说，它们就好像是一个 NDIS 驱动程序。因此，NDIS 中间驱动程序必须同时暴露这两种驱动程序所要求的编程接口，以便能够与上层的协议驱动程序和下层的 NDIS 驱动程序进行通信。

NDIS 中间驱动程序可看到系统中的所有网络流量。而且，在一个系统中，可以有多个中间驱动程序叠加起来。中间驱动程序的一种典型用法是，当系统中有多个网络适配器时，利用中间驱动程序来实现网络流量负载均衡。负载均衡驱动程序向上层传输协议驱动程序暴露一个虚拟的适配器，然后使用适当的负载均衡算法在多个物理适配器之间分配流量。

NDIS 驱动程序，即 NDIS 小端口驱动程序，有两方面的基本功能：管理网络接口卡（NIC，Network Interface Card），包括通过网络接口卡发送和接收数据；与上层驱动程序进行通信。为了完成这些功能，NDIS 驱动程序需要调用 NDIS 库中的函数（NdisM<*Xxx*>和其他的 Ndis<*Xxx*>），同时也需要向 NDIS 库注册一些回调函数（Miniport<*Xxx*>函数），以便 NDIS 库在必要时调用这些函数。下面通过发送数据和接收数据两个操作来说明 NDIS 驱动程序与 NDIS 库和上层协议驱动程序（或 NDIS 中间驱动程序）的协作：

- 当协议驱动程序要发送一个数据包时，它调用 NDIS 库的发送函数，比如 NdisSend。NDIS 库通过调用 NDIS 驱动程序的 Miniport<*Xxx*>函数，比如 MiniportSend，将数据包传给 NDIS 驱动程序。然后，NDIS 驱动程序把数据包插入到底层网络接口卡的发送队列中，交给它发送出去。
- 当网络接口卡接收到一个发送给它的数据包时，它可以触发一个硬件中断，让 NDIS 库或 NDIS 驱动程序来处理该数据包的接收工作。若 NDIS 库响应此中断，则它会在中断服务例程或一个 DPC 中调用 NDIS 驱动程序的接收函数。NDIS 驱动程序在接收了数据包以后，再调用 NDIS 的函数，比如 NdisMIndicateReceivePacket，以通知高层的驱动程序。

如前所述，NDIS 驱动程序并没有采用 Windows I/O 管理器的接口，相反地，它直接在初始化例程（即 PE 文件格式中的入口函数）中，调用 NdisMRegisterMiniport 函数，向 NDIS 库注册它的 Miniport<*Xxx*>函数。因此 NDIS 库将来可以调用这些函数。NDIS 驱动程序也可以向系统注册自己的中断对象，或者设置定时器对象，或者创建额外的系统线程等，以便在未被 NDIS 库调用的情况下也可以获得执行权。NdisMRegisterMiniport 的 NDIS_MINIPORT_CHARACTERISTICS 参数实际上规定了 NDIS 驱动程序可向 NDIS 库暴露的基本接口，读者可以参考 Windows DDK 中 NdisMRegisterMiniport 函数的文档。

关于 NDIS 驱动程序的开发，通常涉及 NDIS 库和 NIC 的编程，Windows DDK 提供了一些例子，可作为最佳参考。这些例子程序展示了 NDIS 驱动程序的一些开发要点，例如 coisdn 演示了面向连接的 NDIS 客户，e100bex 演示了“Intel EtherExpress PRO/100+ Ethernet PCI”适配器的非序列化的小端口驱动程序。这里非序列化的概念是指由驱动程序自己来处理网络请求重叠的情形，而并非采用 NDIS 提供的序列化（即逐个处理）机制。通常非序列化的处理方法可以提高一个网络系统的吞吐量，尤其在多处理器的系统中。关于 Windows DDK 的例子，请参考页面[DDK-NW-SAMP]。

自 Windows Vista 以后，NDIS 库升级到了 NDIS 6，其中主要的变化体现在两方面：

- 增强了网络系统的性能和可扩展性（scalability）。主要的改进表现在：使用了 NET_BUFFER 数据包结构；发送和接收数据的处理路径做了优化；支持运行时重新配置（暂停、重启动）的能力；接收数据包的处理过程针对多核和多处理器做了优化；数据包的发送支持新的分散/聚集（Scatter/Gather）DMA；支持 NDIS 过滤驱动程序；完全支持 TCP 负载外移（TCP offload）。
- 简化了 NDIS 驱动程序的开发模型。主要表现在：初始化接口更加容易使用；一些关键数据结构支持版本更新，许多以前的函数参数现在被放到了数据结构中，这样做的好处是这些函数参数即使以后发生改变，也不会影响到接口的变化；提供了发送取消和适配器 OID 请求取消的功能，从而简化了重置（reset）的处理；提供了查询 NDIS 接口信息的标准接口；新的 NDIS 过滤驱动程序比以前的 NDIS 中间驱动程序更易于编写。

有关 NDIS 更多的信息（包括 NDIS 5 和 NDIS 6）以及 NDIS 驱动程序的开发，请参考 MSDN 中的文档[MSDN-NDIS]以及 Windows DDK 中提供的例子驱动程序[DDK-NW-SAMP]。

9.1.4 Windows Vista 及以后版本的网络结构

在前面的小节中，我们讨论了在 Windows Server 2003 及以前版本中的网络体系结构，

此结构是十多年以来随着 Windows NT 的早期版本而发展过来的。在这十多年间，Windows 操作系统所处的网络环境已经发生了很大的变化，网络连接也趋向于多样化，因此，Microsoft 在 Windows XP/Server 2003 以后重新设计了网络体系结构，所以，Windows Vista 及以后系统的网络结构和接口有所不同。图 9.5 显示了新的网络体系结构示意图。

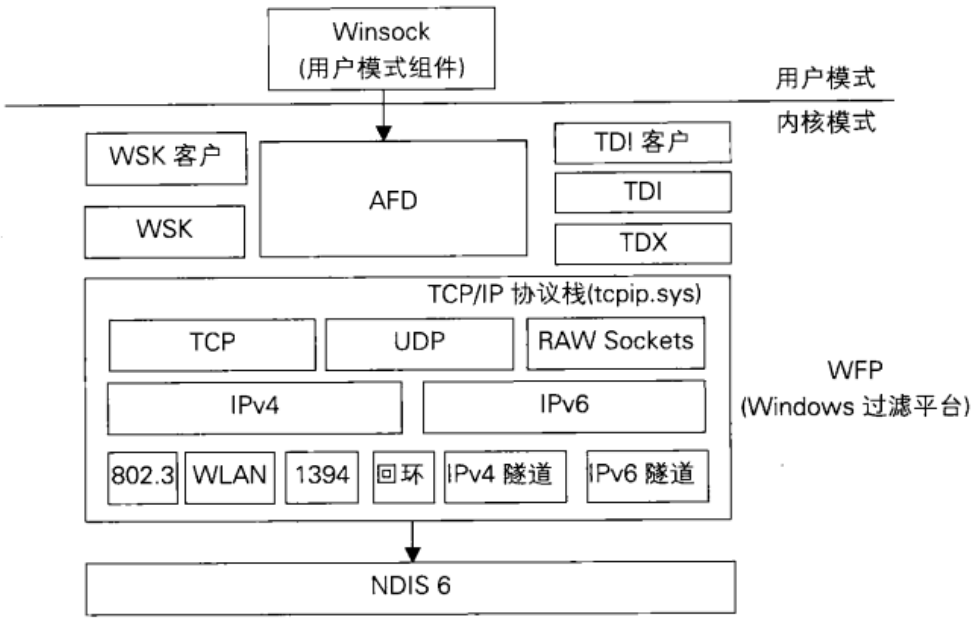


图 9.5 Windows Vista 及以后系统的网络体系结构

在新的网络体系结构中，有几点值得特别指出：

- IPv4 和 IPv6 两个协议栈合二为一，它们共享同样的传输层、网络层和成帧（framing）层，避免了维护两套协议栈带来的额外开销以及不一致性。新的网络系统优先使用 IPv6，这也有利于 IPv6 的广泛部署。
- WSK（Winsock Kernel）是一个内核模式的网络编程接口。通过 WSK，内核模式组件可以使用与用户模式 Winsock 相同的编程模型，来执行网络 I/O 操作。WSK 提供了类似的套接字接口，比如创建套接字、地址绑定、建立连接，以及发送和接收数据等。在以前的网络体系结构中使用 TDI 的内核模式模块，如果专为 Windows Vista 以后的系统设计，则应该使用 WSK，而不再使用 TDI。在新的体系结构中，对 TDI 的支持仅仅是为了兼容以前的软件和协议。
- WFP（Windows Filtering Platform）是新的 TCP/IP 协议驱动程序提供的过滤机制，它允许 WFP 出调驱动程序（WFP callout driver）向 WFP 的过滤引擎（filter engine）注册它的出调函数（callout function）。过滤引擎是 WFP 的核心，在协议栈的一些关键

点上网络数据会经过过滤引擎，由它调用满足条件的出调函数。WFP 的目标是替代原来网络结构中的防火墙钩子和过滤机制，提供更为灵活并且统一的过滤平台。

- NDIS 部分已升级到 NDIS 6，主要增强了网络系统的性能和可扩展性，并简化了 NDIS 驱动程序的开发模型，参见 9.1.3 节。
- TDI 接口在新的体系结构中不再支持，但为了兼容起见，Windows Vista 提供了一个 TDI 翻译驱动程序（TDX），以支持 TDI 客户（和 TDI 过滤驱动程序）。移除 TDI 的主要理由是，TDI 与 Windows 的 I/O 管理器和对象管理器绑得太紧，从而导致 TDI 对象（也是文件对象）不是一个轻量的对象，它们需要较多内存，各种操作相对较重；另一方面，尽管 TDI 客户与传输器之间是松耦合关系，但 TDI 客户实际上需要 TDI 传输器的知识，自动发现 TDI 传输器很难，而且改变 TDI 传输器要求重新启动系统。

关于新体系结构中 WSK、WFP 以及 NDIS 6 的详细介绍，请参考 MSDN 或 Windows DDK 的文档[MSDN-NW]。

9.2 Windows 子系统

正如在第 2 章中所讲，Windows 对应用程序的支持是通过 Windows 的环境子系统来做到的，任何一个用户应用程序都运行在特定的子系统环境中。早期的 Windows 版本支持三个环境子系统：OS/2、POSIX 和 Windows。到了 Windows XP 以后，只有 Windows 子系统随 Windows 操作系统一起发行，而且，在 Windows 系统中，即使是没有交互用户登录的服务器系统，Windows 子系统也是必须运行的。相反，另外两个子系统被配置成按需启动。我们可以这样简单地理解，Windows 子系统是 Windows 操作系统不可分割的一部分，它在 Windows 内核的基础上，为应用程序提供了一个图形用户界面（GUI）环境；OS/2 和 POSIX 则是为了兼容 OS/2 和 UNIX 应用程序而提供的模拟环境。本节将讨论 Windows 子系统的组成结构以及它的两个关键功能部件：窗口管理和 GDI（Graphics Device Interface，图形设备接口），最后简单地介绍在 Windows Vista 及以后的系统中 Windows 子系统的重要变化。

9.2.1 Windows 子系统结构

Windows 支持的应用程序为 PE 文件格式，在 PE 文件格式规范中，有一个域指明了该应用程序属于哪个环境子系统。使用 Visual Studio 随带的工具 dumpbin.exe 或者 Windows 资源工具箱中的 exetype.exe 工具可以查看一个 PE 程序的子系统类型。当使用开

发工具生成一个 Windows 可执行文件时，链接器将填充此信息，例如，Visual Studio 的 link 选项“/SUBSYSTEM”就用于指定一个程序的子系统类型[MSDN-SUBSYSTEM]。

在第 8 章中，我们了解了 Windows 内核提供的系统服务是如何被用户模式代码调用的，而且，应用程序通常并不直接使用操作系统提供的系统服务，而是通过调用系统 DLL 所提供的 API 函数，来间接地使用各种系统服务。Windows 子系统也使用了类似的模块结构，供应用程序直接调用的 API 函数位于一组子系统 DLL 中，这些子系统 DLL 再根据需要调用内核模式组件（win32k.sys）的功能。Windows 子系统的组成结构如图 9.6 所示。注意，这些子系统 DLL 有时候也被简单地统称为系统 DLL。

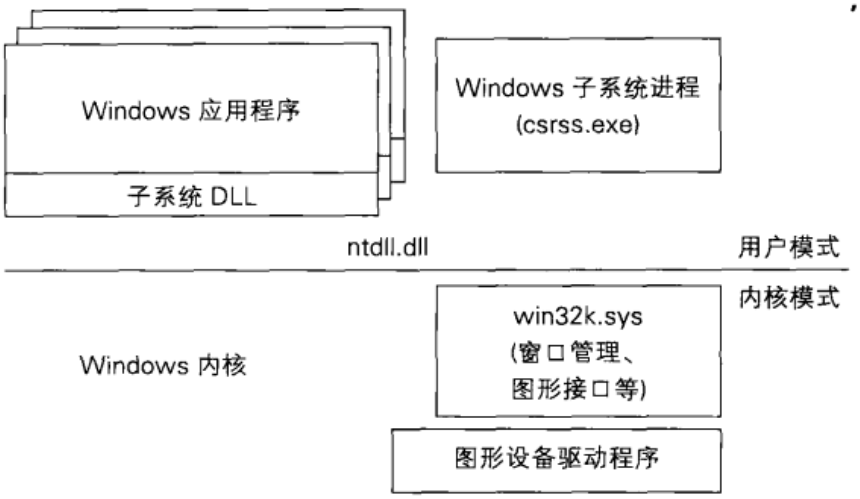


图 9.6 Windows 子系统结构

Windows 子系统中既有用户模式组件，也有内核模式组件；既有注入到每个应用程序进程中的组件模块，也有独立的进程空间。下面列出这些组件的职责：

- 内核模块 win32k.sys，虽然它的名称像是一个驱动程序，但实际上，win32k.sys 并不遵从 I/O 管理器定义的程序模型，而仅仅是 Windows 内核的扩展而已。Win32k.sys 包含两大功能组成部分：
  - 窗口管理器（window manager）：负责控制窗口显示、管理屏幕输出、收集来自键盘、鼠标和其他设备的输入，以及将用户消息传递给应用程序。
  - GDI：这是一个针对图形输出设备的函数库，包含了有关线、文本和图形绘制，以及操纵各种图形的函数。
- 图形设备驱动程序，这是一些与硬件相关的显示驱动程序、打印驱动程序，以及视频小端口驱动程序。

- Windows 环境子系统进程（csrss.exe），它包含以下支持：
  - 控制台窗口。
  - 创建和删除进程和线程。
  - 支持 16 位虚拟 DOS 机（VDM，Virtual DOS Machine）进程。
  - 其他一些函数，比如 GetTempFile、DefinedDosDevice、ExitWindowsEx，以及少量自然语言支持函数。
- 子系统 DLL，例如 user32.dll、advapi32.dll、gdi32.dll 和 kernel32.dll，它们实现了已经文档化的 Windows API 函数，可能需要调用内核模块 ntoskrnl.exe 和 win32k.sys 中未文档化的系统服务，甚至与环境子系统进程通信。

Windows 是一个以图形用户界面为主要输入输出手段的操作系统，但图形用户界面功能并非位于内核模块 ntoskrnl.exe 中，而是在 Windows 子系统中，这是 Windows 体系结构的一个重要特性。在 Windows 的发展历史上，图形用户界面的窗口管理和 GDI 曾经位于用户模式 Windows 子系统进程中，但从 Windows NT 4 开始，这些代码被移到了内核模式下（win32k.sys），而直接在系统地址空间中运行。根据第 8 章的介绍，我们已经知道，win32k.sys 注册了一个 SDT（系统服务描述符表），将它所提供的功能直接以系统服务的形式暴露给用户模式程序。因此，Windows 应用程序在使用窗口管理和 GDI 功能时，通过子系统 DLL 可直接调用 win32k.sys 中的系统服务。关于一个系统上 win32k.sys 所包含的系统服务，读者可以使用第 8 章介绍的 SDTViewer 工具查看。

在 Windows 子系统进程中，由于窗口管理和 GDI 已经移出，所以，它介入 Windows 应用程序运行过程的机会并不多，然而，特别值得一提的是，进程和线程的创建和删除过程仍然受子系统进程管理。在第 3 章中，我们看到，Windows 的进程和线程数据结构中有一些域是保留给 Windows 子系统使用的。当进程和线程内核对象被创建起来以后，子系统 DLL 要给 Windows 子系统进程发送一个 LPC 消息，以通知它有关新进程和新线程的信息。Windows 子系统进程维护了所有属于该子系统的进程和线程的列表，并且设置进程的异常端口和调试端口，以便接收该进程中发生的异常和调试事件。类似地，当线程或进程退出或终止时，Windows 子系统进程也会被通知到，从而维护子系统内部信息的一致性。关于 Windows 进程和线程的创建过程，参见 3.4.3 节。

现在我们从应用程序的角度来看，它通过子系统 DLL 发出的服务请求（即 API 调用）是如何被满足的。首先，由于这些子系统 DLL 被加载到应用程序进程中，所以，这些服务请求是直接的函数调用（同一进程地址空间中跨模块的函数调用）。当子系统 DLL 接收到一个函数调用以后，根据该函数功能的复杂程度，可能有以下的处理方式：

- 直接由子系统 DLL 包揽所有工作，这一类函数总是在用户模式下完成，不涉及处理器模式切换。这类函数的典型例子有：像 PtInRect、IsRectEmpty 这样的简单函数，无须内核模块或 Windows 子系统进程的介入即可完成；像 GetCurrentProcess 函数，简单地返回一个伪句柄值（-1）代表当前进程；像 GetCurrentProcessId 函数，可以从一个缓存的数据结构中获得当前进程的 ID，无须每次调用都进入到内核中，因为进程 ID 在进程的生命周期中保持不变。

- 需要调用 Windows 内核一次或多次。在这种情形下，可能存在一次或多次模式切换，子系统 DLL 或者通过 ntdll.dll 调用到 Windows 执行体，或者通过 win32k.sys 注册的系统服务表调用到 win32k.sys 中。这类 API 函数的例子有：像 ReadFile 和 WriteFile 函数，调用底层的 NtReadFile 或 NtWriteFile 系统服务；像 PostMessage 和 BitBlt 这样的窗口管理和 GDI 函数，调用 win32k.sys 中的 NtUserPostMessage 或 NtGdiBitBlt 来完成其功能（它们也可能会调用 Windows 执行体函数，比如为了获取用以保护共享资源的锁）。

- 需要 Windows 子系统进程的协助来完成其功能。在这种情况下，子系统 DLL 向 csrss.exe 进程发送一个请求（以 LPC 消息的形式），然后等待应答消息，直至收到应答并完成所有功能之后再返回调用者。这类 API 函数最典型的例子是 CreateProcess 和 CreateThread，它们需要通知子系统进程，以维护子系统环境中进程和线程的状态。

以上三种可能的执行方式仅仅代表了子系统 DLL 在执行一个服务请求时可能的执行路径，并非简单地将 API 函数分成三类。有些 API 函数在某些条件下要切换到内核模式，在另外的条件下则不需要；有些函数既要多次进入内核，又要与子系统进程通信。然而，从这三种可能的执行方式来理解一个 API 函数的性能是有意义的。与子系统进程通信要涉及进程之间的同步以及线程切换和进程切换，所以，包含这种执行路径的 API 函数调用会有相对较大的系统开销。这也是当初将窗口管理和 GDI 功能从子系统进程移出到内核中的原因，这样做可以避免大量的跨进程通信开销。现在的 Windows 系统中，仅有少量的 API 函数还需要与子系统进程通信。从用户模式切换到内核模式也会带来额外的开销，但是，若一个 API 函数真正需要调用内核的系统服务，那么，这种模式切换开销也是不可避免的。Windows 子系统 DLL 实现了一些优化或者简单的替代功能，尤其针对一些被频繁调用的小函数，以避免过多的模式切换开销。例如，kernel32.dll 中实现的临界区（critical section）同步机制，只有在真正发生多线程访问冲突时才调用内核的等待函数；kernel32.dll 还实现了一组互锁操作，用于保护用户模式下的简单整数运算。

9.2.2 Windows 子系统初始化与 GUI 线程

Windows 子系统为操作系统提供了 GUI 的能力，这也正是在 Windows 体系结构中，其他的环境子系统依赖于 Windows 子系统的原因。在这一节中我们首先看一看 Windows 子系统的初始化过程，然后看一下 GUI 线程，以便将 Windows 子系统与整个操作系统关联起来。

Windows 的会话管理器（smss.exe）是系统中第一个被创建的用户模式进程，它是在 Windows 执行体和内核初始化完成以后被启动的。Smss 进程的任务之一是，启动 Windows 环境子系统进程 csrss.exe 和 Windows 登录进程 winlogon.exe。此外，smss 还通过系统服务 NtSetSystemInformation，指示内核加载 win32k.sys 模块。它向 NtSetSystemInformation 函数的 SystemInformationClass 参数传递的值为 SystemExtendServiceTableInformation，其处理过程可参考 WRK 中 base\ntos\ex\sysinfo.c 文件的 3 348~3 505 行代码。NtSetSystemInformation 函数调用 MmLoadSystemImage 函数以加载指定的“\SystemRoot\System32\win32k.sys”文件，然后调用其 PE 头中指定的入口函数 EntryPoint。所以，尽管 win32k.sys 看起来像是一个驱动程序名称，但实际上它并非驱动程序，而是内核的动态链接库。

在 win32k.sys 的入口函数中，除了对内部的一些全局变量进行初始化（包括初始化同步事件、申请内存等）以外，它主要做了以下的初始化工作：

- 调用 KeAddSystemServiceTable 函数，加入一个新的 SDT（服务描述符表）。
- 调用 PsEstablishWin32Callouts 函数，向内核注册一组出调函数，以便内核在适当时候调用这些出调函数。
- 调用 MmPageEntireDriver 函数，使得 win32k.sys 的所有代码页面和数据页面可被换出内存（pageable）。
- 调用 InitializeGre 函数，初始化 Windows 子系统的图形引擎。此初始化函数是 win32k.sys 的内部函数。
- 调用 Win32UserInitialize，初始化 Windows 子系统的窗口管理器。此初始化函数也是 win32k.sys 的内部函数。

正如 8.1.4 节所讲，win32k.sys 在它的入口例程中，调用 KeAddSystemServiceTable 函数，将 win32k.sys 中的系统服务表加入到了 SDT 数组 KeServiceDescriptorTableShadow 中，因而以后 Windows 子系统 DLL 通过此 SDT（对应的系统服务号范围为 0x1000~0x1fff）来访问 win32k.sys 中的系统服务。在 Windows Server 2003 SP1 上，win32k.sys 提供了 665 个系统服务，这里不一一列出这些服务名称，读者可以通过 SDTViewer 工具查看它们。

表 9.2 说明了 win32k.sys 所提供的系统服务的名称分类。

表 9.2 Win32k.sys 提供的系统服务的名称分类（适用于 Windows Server 2003）

名称	功能说明
NtUser<*Xxx*>	窗口管理服务，例如 NtUserFindWindowEx 和 NtUserGetMessage
NtGdi<*Xxx*>	GDI 服务，例如 NtGdiGetDcObject 和 NtGdiSetBitmapBits
NtGdiEng<*Xxx*>	图形引擎服务，例如 NtGdiEngBitBlt 和 NtGdiEngTextOut
NtGdiDd<*Xxx*>	DirectDraw 服务，例如 NtGdiDdGetDxHandle 和 NtGdiDdAlphaBlt
NtGdiDvp<*Xxx*>	与视频端口（video port）打交道的 DirectDraw 服务，例如 NtGdiDvpCreateVideoPort 和 NtGdiDvpGetVideoPortInputFormats
NtGdiD3d<*Xxx*>	Direct3D 服务，例如 NtGdiD3dCreateSurfaceEx 和 NtGdiD3dDrawPrimitives2

Win32k.sys 在初始化阶段通过 PsEstablishWin32Callouts 函数向内核注册一组出调（callout）函数，数据结构 WIN32_CALLOUTS_FPNS 定义了这些出调函数，读者可参考 PsEstablishWin32Callouts 函数的代码（位于 base\ntos\ps\psquery 文件中）。Win32k.sys 将这些出调函数暴露给了内核，因而内核可以在适当的时候调用它们。以下是 WIN32_CALLOUTS_FPNS 的定义：

```
typedef struct _WIN32_CALLOUTS_FPNS {
    PKWIN32_PROCESS_CALLOUT ProcessCallout;          // 创建或删除进程时调用
    PKWIN32_THREAD_CALLOUT ThreadCallout;            // 转成 GUI 线程或删除时被调用
    PKWIN32_GLOBALATOMTABLE_CALLOUT GlobalAtomTableCallout;
                // 内核的全局原子对象管理器调用该函数来获得当前线程的全局原子对象表的地址
    PKWIN32_POWEREVENT_CALLOUT PowerEventCallout;    // 子系统通过它来接收电源事件
    PKWIN32_POWERSTATE_CALLOUT PowerStateCallout;    // 子系统通过它来接收电源状态
    PKWIN32_JOB_CALLOUT JobCallout;                  // win32k.sys 通过它参与作业管理
    PVOID BatchFlushRoutine;                         // 刷新 GDI 批任务处理
    PKWIN32_OBJECT_CALLOUT DesktopOpenProcedure;     // 打开桌面
    PKWIN32_OBJECT_CALLOUT DesktopOkToCloseProcedure;        // 是否可以关闭桌面
    PKWIN32_OBJECT_CALLOUT DesktopCloseProcedure;    // 解除桌面映射
    PKWIN32_OBJECT_CALLOUT DesktopDeleteProcedure;   // 释放桌面
    PKWIN32_OBJECT_CALLOUT WindowStationOkToCloseProcedure;  // 是否可以关闭窗口站
    PKWIN32_OBJECT_CALLOUT WindowStationCloseProcedure;      // 从全局列表中去除窗口站
    PKWIN32_OBJECT_CALLOUT WindowStationDeleteProcedure;     // 删除窗口站，释放资源
    PKWIN32_OBJECT_CALLOUT WindowStationParseProcedure;      // 解析一个窗口站路径
    PKWIN32_OBJECT_CALLOUT WindowStationOpenProcedure;       // 打开一个窗口站路径
} WIN32_CALLOUTS_FPNS, *PKWIN32_CALLOUTS_FPNS;
```

Win32k.sys 针对每个会话被加载一次，它没有创建任何一个设备对象，但是创建了驱动程序对象“\Driver\win32k”，因此，当其他模块需要引用 win32k 时，可以通过调用对象管理器的 ObReferenceObjectByName 函数获得此驱动程序对象。

Windows 内核创建的线程不是 GUI 线程，在 8.1.3 节，我们曾经提到，当一个非 GUI 线程调用的系统服务号超出该线程所指的系统服务表数组中定义的界限，并且系统服务号的表索引为 1 时，系统服务分发代码试图将它转换为 GUI 线程，这是通过调用

PsConvertToGuiThread 函数来完成的。因此，当一个非 GUI 线程调用任何一个子系统系统服务（服务号为 0x1000~0x1fff）时，由于该线程的 ServiceTable 成员初始指向 KeServiceDescriptorTable，而 KeServiceDescriptorTable[1]项的系统服务表不包含任何系统服务，所以，此调用必定导致 PsConvertToGuiThread 函数被调用，从而该线程被转换为 GUI 线程。

PsConvertToGuiThread 函数将一个线程转换为 GUI 线程，其代码位于 base\ntos\ps\psquery.c 文件中，它所做的工作包括：

- 将线程的内核栈转换为一个大内核栈。它先调用 MmCreateKernelStack 函数创建一个新的大内核栈，然后调用 KeSwitchKernelStack 函数替换内核栈。在 Intel x86 平台上，大内核栈的大小为 61 440 字节，标准的内核栈大小为 12 288 字节。注意，内核栈的实际内存分配是按页面来计数的，而且还要加上一个不可访问（no-access）的保护页面，以便检测到内核栈溢出的错误。
- 调用 win32k.sys 注册的出调函数 PspW32ProcessCallout。Windows 子系统为它的进程维护了一套平行的数据结构，进程对象 EPROCESS 中的 Win32Process 域指向 Windows 子系统分配的进程数据结构。PspW32ProcessCallout 函数创建此进程结构，并调用内核的 PsSetProcessWin32Process 函数，将其设置到执行体进程对象中；然后，若有必要，则初始化此进程结构，并执行 GDI 初始化。
- 将线程的 ServiceTable 域切换到指向 KeServiceDescriptorTableShadow，从而使该线程以后可以使用 win32k.sys 提供的系统服务。
- 调用 win32k.sys 的出调函数 PspW32ThreadCallout。同样地，Windows 子系统也为每个 GUI 线程维护了一个数据结构，线程对象 KTHREAD 中的 Win32Thread 域指向 Windows 子系统分配的线程数据结构。PspW32ThreadCallout 函数创建此线程结构，并调用内核的 PsSetThreadWin32Thread 函数，将其设置到执行体线程对象中；然后，初始化该线程中与 Windows 子系统相关的成员（包括线程的 TEB 中的信息），并且在 win32k.sys 内部维护该线程的信息，同时也执行 GDI 初始化。

将一个内核线程转换为 GUI 线程，这表明了 Windows 子系统已接纳该线程及其所属的进程。两个出调函数 PspW32ProcessCallout 和 PspW32ThreadCallout 分别执行了当该线程和所属进程加入到 Windows 子系统中时的初始化工作（当然，当线程或进程结束时，这两个出调函数也负责相应的清除工作）。一旦线程被接纳到 Windows 子系统中，它就可以方便地使用 Windows 子系统提供的两个重要功能：窗口管理和 GDI。

9.2.3 窗口管理

窗口管理是 Windows 作为桌面操作系统成功的基础，也是 Windows 子系统的重要职责。尽管绝大多数应用程序只需简单地按照 Windows SDK 中介绍的那样，通过窗口句柄和消息机制来访问和操纵 Windows 应用程序中的用户界面元素，但 Windows 子系统内部维护了一套复杂的窗口管理机制，并且此管理机制贯穿一个用户会话的始终。当用户登录到 Windows 系统中时，winlogon 进程会创建一个交互式窗口站（interactive window station）和三个桌面。应用程序所创建的窗口，一定属于某一个桌面。图 9.7 显示了 Windows 中窗口站和桌面的关系。

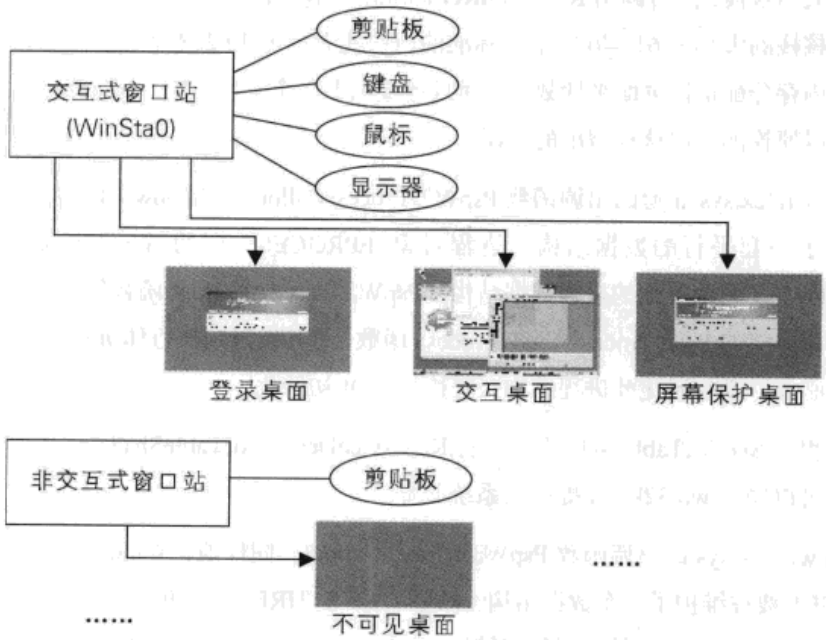


图 9.7 Windows 中的窗口站和桌面

交互式窗口站包含剪贴板、键盘、鼠标、显示器和三个桌面。这三个桌面分别对应于当用户登录时、当用户可正常操作应用程序时，以及当屏幕保护程序启动时的情形。这些桌面都可以显示图形界面元素，并且可以接收用户输入，但任何时候只有一个桌面是活动的，该桌面称为输入桌面（input desktop），只有它可以往屏幕上输出以及接收用户输入。非交互式窗口站是针对非交互式登录会话而存在的窗口站，它们不拥有输入/输出设备，所以这些窗口站中的桌面都是不可见的，其中的应用程序无法接收用户输入。

另外，在支持远程桌面服务的系统上，当用户通过远程桌面连接到一个 Windows 系统中时，系统会针对此登录用户启动一个会话，该会话也有自己的交互式窗口站（图 9.7 未显示远程会话的窗口站和桌面）。

接下来我们看一下窗口站和桌面与进程和线程之间的关系：

1. 窗口站可通过 Windows 子系统的系统服务 NtUserCreateWindowStation 来创建，当一个进程调用该系统服务来创建一个窗口站时，此窗口站与该调用进程相关联，并且属于该进程所在的会话。例如，当交互式用户登录时，winlogon 进程调用此系统服务来创建窗口站 WinSta0，所以，WinSta0 与 winlogon 进程关联，并属于该交互式用户的会话。

2. 桌面可通过 Windows 子系统的系统服务 NtUserCreateDesktop 来创建，当一个进程调用该系统服务来创建一个桌面时，所创建的桌面属于该进程所属的窗口站，并且与调用线程相关联。

3. 当一个进程第一次调用 Windows 子系统的系统服务（并非窗口站或桌面的系统服务）时，它自动与一个窗口站和桌面建立连接。至于连接到哪个窗口站，则依据以下规则来确定：

   a. 该进程已调用过 SetProcessWindowStation 函数，指定了窗口站。

   b. 从父进程继承一个窗口站。

   c. 如果该进程没有调用过 SetProcessWindowStation 函数，也不能从父进程获得窗口站，那么，Windows 子系统试图按以下的顺序来选择窗口站：

      i. 进程创建函数 CreateProcess 的 STARTUPINFO 参数中指定的窗口站名称。

      ii. 若进程运行在交互式用户的登录会话中，则连接到交互式窗口站。

      iii. 若运行在非交互式的登录会话中，则根据登录会话的标识信息，构造一个窗口站名称，并试图打开该窗口站。如果打开操作不成功，则创建此名称的窗口站以及一个默认桌面。

4. 当进程连接到一个窗口站以后，Windows 子系统给调用线程分配一个桌面。至于分配哪个桌面，则依据以下规则来确定：

   a. 该线程已调用过 SetThreadDesktop 函数，指定了桌面。

   b. 从父进程继承一个桌面。

   c. 如果该线程没有调用过 SetThreadDesktop 函数，也不能从父进程获得桌面，那么，Windows 子系统试图按以下的顺序来选择桌面：

      i. 进程创建函数 CreateProcess 的 STARTUPINFO 参数中指定的桌面名称。

      ii. 否则，连接到该进程所连接的窗口站的默认桌面。

窗口站被注册到对象管理器名字空间的“\Windows\WindowStations”目录下或“\Sessions\<*X*>\Windows\WindowStations”目录下，这里 *X* 代表会话 ID，例如 1、2 等。举例来说，在一个 Windows Server 2003 SP1 系统中，“\Windows\WindowStations”目录包含以下 5 个窗口站：

```
WinSta0
Service-0x0-3e4$
Service-0x0-3e5$
Service-0x0-3e7$
SAWinSta
```

这里 WinSta0 是登录用户的默认窗口站，Service-0x0-<*xxx*>$是非交互式窗口站。正如前文所讲，当一个运行在非交互式登录会话中的进程试图连接到一个窗口站时，它根据登录会话的标识符来构造一个窗口站名称，此名称的形式为“Service-0x<*yyy*>-<*xxx*>$”，这里 *xxx* 和 *yyy* 分别是登录会话标识符的高位部分和低位部分。若两个进程运行在同一个用户账户下，它们共享一个会话，其登录会话标识符当然相同，因而构造出来的窗口站名称也是一样的，从而它们将运行在同一个窗口站中。在上述例子中，窗口站 Service-0x0-3e7$是运行在 SYSTEM 账户下的进程所对应的窗口站；Service-0x0-3e4$窗口站对应 NETWORK SERVICE 账户；Service-0x0-3e5$窗口站对应 LOCAL SERVICE 账户。这些窗口站都包含一个“Default”桌面。窗口站 SAWinSta 是由任务调度器（Task Scheduler，一个 svchost 进程）创建的，所有通过任务调度器启动的进程都连接到此窗口站。如果系统的任务调度器服务未启动，则窗口站 SAWinSta 不会被创建。

在第 4 章中，我们曾经介绍过会话空间，这里的会话（session）是 Windows 操作系统支持的概念，它代表了一个用户登录到系统中以后的执行环境。任何一个进程都属于一个会话。会话空间是指系统地址空间中特定的地址范围，在 Windows Server 2003 系统中为 0xbd000000~0xbfffffff。属于同一个会话的进程共享这段地址空间所映射的页面；属于不同会话的进程，其地址空间中这一段范围被映射到不同的页面。正如 4.2.1 节所介绍，会话空间包含 4 部分：会话映像文件区包括 win32k.sys、视频驱动程序和打印驱动程序的映像；会话视图区域包含了该会话中映射的视图，包括桌面堆（desktop heap，见本节下文介绍）；SESSION_WS 区域包含了内存管理数据结构，包括该会话的工作集链表；会话内存池是一个换页内存池，允许分配整个会话范围的内存块。

在 Windows 中，会话是按编号来区分的。在系统控制台登录的用户会话为 Session 0，其他通过远程桌面或终端服务登录到系统中的会话可以是 Session 1、Session 2 等。每个会话包含一个或多个窗口站，每个窗口站包含一个或多个桌面。值得注意的一点是，在 Windows Server 2003 中，像系统服务或者按特定账户（Run As）来运行的进程，虽然不

在 WinSta0 窗口站中，但是它们运行在 Session 0 中。所以，前面例子中列出的 Service-0x0-xxx$和 SAWinSta 窗口站都位于 Session 0 中。

不难看出，每一个会话有一个 win32k.sys 实例，每个 win32k.sys 实例又有自己的状态。远程桌面或终端服务会话也包含 win32k.sys 实例，它们会重新加载 win32k.sys 模块；而且，这些会话也有 WinSta0 窗口站，这是它们的交互式窗口站，用于产生屏幕输出和接收输入。但是，像 Windows 服务进程所连接的额外窗口站，在这些会话中则不再出现。

桌面必定属于一个窗口站，它代表了一个逻辑输出表面。桌面包含了各种用户界面对象，例如我们通常见到的应用程序窗口、按钮、菜单、工具条等。为了维护这些对象，每个桌面都有一个相应的桌面堆，所有这些对象都从它的桌面堆中分配。通过 API 函数 CreateDesktop 来创建的桌面都有默认的桌面堆大小；而 CreateDesktopEx 函数则允许调用者指定桌面堆的大小。一个会话中所有窗口站的桌面堆都从该会话的会话视图区域申请。在 Windows Server 2003 中，会话视图区域的大小为 20 MB，该值可通过修改注册表中会话管理器的内存配置参数而改变。桌面堆的默认大小也可通过修改注册表中会话管理器的内存配置参数而改变，对于交互式窗口站和非交互式窗口站中的桌面堆，其默认值分别为 3 072 KB 和 512 KB。显然，在会话视图区域确定的情况下，桌面堆的大小决定了该会话能支持多少个桌面；而在桌面堆确定的情况下，该桌面中能构建的用户界面对象也是有限的。如果一个桌面堆中的内存被用光了，那么，在该桌面上运行的进程想要创建窗口对象，必定招致失败，在这种情况下，可能连错误消息框都显示不出来了。

Microsoft 提供了一个工具 Desktop Heap Monitor（dheapmon），可以用来显示一个系统中桌面堆的分配和使用情况[DH-MON]。图 9.8 显示了该工具的一个输出例子（使用 dheapmon 8.1 版本），目标系统是运行在 Virtual PC 中的 Windows Server 2003 SP1 系统。

```
C:\kktools\dheapmon8.1\x86>dheapmon.exe
Desktop Heap Information Monitor Tool (Version 8.1.2925.0)
Copyright (c) Microsoft Corporation.  All rights reserved.
-------------------------------------------------------------
 Session ID:    0 Total Desktop: (  5312 KB -    7 desktops)

 WinStation\Desktop          Heap Size(KB)    Used Rate(%)
-------------------------------------------------------------
 WinSta0\Default                        3072            2.5
 WinSta0\Disconnect                     64              4.0
 WinSta0\Winlogon                       128             6.1
 Service-0x0-3e7$\Default               512             3.2
 Service-0x0-3e4$\Default               512             3.2
 Service-0x0-3e5$\Default               512             0.4
 SAWinSta\SADesktop                     512             0.4
-------------------------------------------------------------
```

图 9.8 使用 Desktop Heap Monitor 的例子

该系统的注册表中的会话管理器内存配置参数均为默认设置。Winlogon 和 Disconnect 两个桌面并没有使用前面提到的默认设置，它们是默认规则的两个例外；其余桌面均符合默认设置。Dheapmon 工具也显示了每个桌面的内存使用百分比。

至此，我们了解了 Windows 子系统的窗口管理系统中的概念：会话、窗口站和桌面。有关 Windows 子系统的窗口站和桌面这两个概念的更多信息，读者可以参考 MSDN 中的文档[MSDN-WINSTA]。Windows 应用程序可以通过 EnumWindowStations 函数列举当前会话中的窗口站；通过 EnumDesktops 函数列举指定窗口站中的桌面；更进一步，还可以通过 EnumDesktopWindows 函数来列举指定桌面中的所有顶级窗口。这三个函数都是 Windows 的标准 API 函数，读者可以参考 Windows SDK 或 MSDN 中的文档。

在一个桌面环境下的窗口管理，这是 Windows 应用程序开发教程中广泛涵盖的话题。窗口是 Windows 子系统中的对象，是由 win32k.sys 来管理的内核对象，并且以句柄（HWND）的方式暴露给应用程序代码。Windows 子系统的每个会话实例都维护了一个句柄表，注意，这是会话空间的句柄表，与每个进程的句柄表没有直接关系。Windows 子系统的句柄由两部分构成：一是该对象在句柄表中的索引，二是确保句柄唯一性的部分。当 win32k.sys 接收到用户模式系统调用时，若参数中包含句柄值，则利用句柄的索引部分可直接定位到句柄表中的项，从而获得对应的内核对象的指针。窗口与线程有亲和性（affinity），而线程又与桌面有亲和性，所以，窗口也必定属于一个桌面。

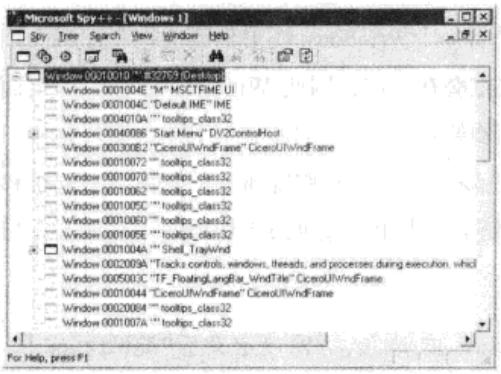
Windows 子系统以窗口为基本图形单元来管理可视的界面元素，它支持以下 5 种窗口：

- 可重叠窗口（overlapped window），是桌面上的顶级窗口，通常有标题、边框和客户区域。可重叠窗口往往作为 Windows 应用程序的主窗口，可以有一个系统菜单、最大化和最小化按钮，以及滚动条。这类窗口通常由应用程序指定而创建。
- 弹出式窗口（pop-up window），是一种特殊的可重叠窗口，往往用做对话框、消息框或者临时跳出应用程序主窗口的窗口。
- 子窗口（child window），顾名思义，它们有父窗口，并且，它们受限于父窗口的客户区域。应用程序通常利用子窗口把父窗口的客户区域分隔成不同的功能区域。父窗口可以是可重叠窗口、弹出式窗口，甚至子窗口。
- 层次窗口（layered window），用于实现不规则形状的窗口，或者窗口的形状有动画效果，或者窗口有半透明效果。
- 消息窗口（message-only window），这类窗口仅仅用于发送和接收消息，它们不可见，也不参与桌面上窗口的层次交叠，并且也不会被列举到。

比较典型和常见的是前面 3 种窗口，几乎每一个 Windows 图形界面应用程序都会用到。由于一个桌面上的窗口可能会以各种方式重叠在一起，因此 Windows 子系统为桌面中的所有窗口定义了一个 z-序（z-order），即深度顺序。在 z-序中，最上面的窗口是用户可以看到的，而下面的窗口则极有可能被隐藏起来了；子窗口跟随所属的父窗口一起被移动。因此，桌面上的窗口按 z-序形成了一个单链表。当用户操纵或者切换应用程序的窗口时，它们的 z-序实际上也在作相应的调整。

Windows 子系统定义了窗口类(window class)的概念，为窗口定义了“类型(type)”。窗口类为一类窗口定义了它们的各种窗口属性，包括风格、菜单、背景颜色和填充刷子等，尤为重要的是，它定义了一个窗口过程（window procedure）用于处理所有该类窗口的消息。Windows 子系统内置了 7 种窗口类：按钮（Button）、组合框（ComboBox）、编辑框（Edit）、列表框（ListBox）、多文档界面中的子窗口（MDIClient）、滚动条（ScrollBar）和静态文本（Static），以及一些供 Windows 内部使用的窗口类：组合框内含列表框（ComboLBox）、DDE 管理库事件（DDEMLEvent）、消息窗口（Message）、菜单（#32768）、桌面窗口（#32769）、对话框（#32770）、任务切换窗口（#32771）和图标（#32772）。

Windows 子系统为每个窗口维护了一些重要的信息，包括：窗口的风格和位置信息、父窗口、窗口过程、滚动信息、菜单等，并且每个窗口也包含一个指向其窗口类的引用。除了这些固有的属性以外，窗口也可以有以字符串来命名的属性，因而应用程序可以通过窗口的已命名属性来保存与应用相关的状态信息。Windows 应用程序调用 API 函数 CreateWindow 或 CreateWindowEx 来创建一个窗口，对应的 win32k.sys 系统服务为 NtUserCreateWindowEx。应用程序可以调用 EnumWindows 函数列举得到桌面上所有顶级窗口；调用 EnumThreadWindows 列举一个线程创建的窗口；调用 EnumChildWindows 列举某一指定窗口的所有子窗口；调用 FindWindow 查找指定类名称或窗口名称的窗口，或调用 FindWindowEx 查找子窗口对象；调用 SetProp 和 GetProp 设置或获取一个命名属性；调用 EnumProps 列举一个窗口的命名属性。一个桌面上的窗口按照层次结构的形式有序地组织在一起。Microsoft 提供了一个工具 spy++（随 Visual Studio 一起提供），可以显示桌面上的窗口，以及进程、线程与窗口的关系，还可以查找窗口，以及实时地监视窗口处理的消息。图 9.9(a)显示了 spy++在列举一个 Windows Server 2003 系统桌面的窗口，图 9.9(b)显示了 spy++正在监视一个窗口的消息处理。



(a) 列举桌面上的所有窗口

(b) 监视一个窗口的消息处理

图 9.9 spy++使用示例

Windows 为应用程序提供了消息驱动的编程模型，负责处理用户界面的线程的主体逻辑通常是一个消息循环，如下面的代码所示：

```
for(;;)
{
    if (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)   // 从消息队列中获得一个消息
    {
        if (bRet == -1) goto ErrorExit;      // 严重错误处理
        TranslateMessage(&msg);              // 处理按键消息，将虚拟键消息转译为字符消息
        DispatchMessage(&msg);               // 将消息分发到负责处理该消息的窗口
    } else {
        break;                               // 线程接收到 WM_QUIT 消息，退出循环
    }
}
```

这里 GetMessage 是线程向 Windows 子系统要一个消息，然后经过 TranslateMessage 之后，通过 DispatchMessage 将消息传递到该线程中接收此消息的窗口的窗口过程，所以，在 DispatchMessage 中，一个消息的目标窗口的窗口过程会被自动调用。（注意，这里 DispatchMessage 实际上是 DispatchMessageA 或 DispatchMessageW，下文同。）

与窗口数据结构不同的是，消息数据结构直接被暴露给应用程序，我们在 Windows SDK 中可以看到消息的定义如下：

```
typedef struct {
    HWND hwnd;              // 接收该消息的窗口的句柄
    UINT message;           // 消息标识符，WM_USER(0x400)以下的消息为系统保留
    WPARAM wParam;          // 该消息的参数，其含义取决于 message 成员的值
    LPARAM lParam;          // 该消息的另一个参数，其含义也取决于 message 成员的值
    DWORD time;             // 该消息被寄到队列中的时间
    POINT pt;               // 记录了当该消息被寄到队列中时的光标位置（按屏幕坐标）
} MSG, *PMSG;
```

Windows 子系统为 GUI 线程维护了一个消息队列。线程的主体逻辑不断地获取消息并交给消息的目标窗口的窗口过程来处理。这些消息可能来自于系统的输入设备，比如键盘和鼠标，也可能来自于系统中其他的线程。图 9.10 显示了 Windows 子系统的消息流。在每个会话中，Windows 子系统进程（csrss.exe）都会创建一个 RIT（Raw Input Thread），该线程负责从设备驱动程序获得原始的输入，然后将消息寄到正确的队列中。其中鼠标的输入是由另一个称为桌面线程（desktop thread）的线程来接收的，它将来自驱动程序的原始鼠标信息转变成一个内部鼠标事件，然后交给 RIT 线程分发到应用线程中。键盘事件或其他的 HID（Human Input Device）事件则直接由 RIT 线程从设备驱动程序中获得。

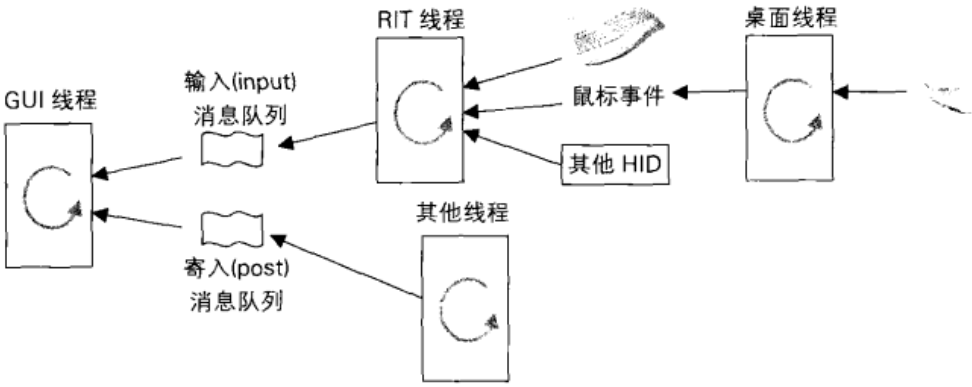


图 9.10 Windows 子系统的消息流

RIT 或桌面线程获得输入事件的做法是这样的：第一次，向设备发起一个异步读操作，在读操作中指定一个 APC 例程，因而当设备驱动程序有数据可提供时，I/O 管理器在完成处理时将此 APC 例程插入到原线程中；此 APC 例程在获得了当前的输入事件以后，再次发起一个异步读操作，然后返回；下次当设备驱动程序有输入数据时，会再次触发 APC 例程；依此类推，可通过 APC 例程不停地接收输入设备的数据。关于 I/O 处理过程，包括完成处理时如何触发 APC 例程，参见 6.6.3 节。

在 Windows 系统中，每个会话都有一个 RIT 线程。为了与输入设备驱动程序建立联系，RIT 线程调用 IoRegisterPlugPlayNotification 函数，向即插即用管理器注册它感兴趣的输入设备的即插即用通知。RIT 线程在完成了初始化以后，其主体逻辑是，根据当前桌面上所有 GUI 线程的状态，确定每一个接收到的输入事件应该属于哪一个线程，从而将它插入到该线程的输入消息队列中。例如，对于鼠标事件，它需要确定当前桌面的焦点窗口、鼠标事件的光标所在窗口等，然后才能确定目标线程；对于键盘事件，它需要根据键盘布局结构信息，确定是否为控制键，并且还要处理键盘的状态，最后将键盘事件插入到当前前台线程的消息队列中。

在一个 GUI 线程的消息循环中，当该线程要获取一个消息时，win32k.sys 中的系统服务 NtUserGetMessage 必须检查两个队列。它首先检查系统队列，即图 9.10 中的输入消息队列；如果该队列中没有满足条件的消息，则再检查应用队列，即图中的寄入消息队列。如果 GUI 线程以阻塞方式（即 GetMessage 而非 PeekMessage）调用 NtUserGetMessage 系统服务，那么，该线程进入等待，在一个内部事件对象上等待，直至有满足条件的消息到来。

除了 HID 生成的输入事件以外，系统中其他的线程也可以向一个线程以“邮寄(post)”的方式送出一个消息，Windows 子系统把这类消息放在一个单独的寄入消息队列中。Win32k.sys 中相应的系统服务为 NtUserPostMessage 和 NtUserPostThreadMessage。注意，这样寄入的消息可能并非针对一个窗口，而是针对一个线程，在这种情况下，线程的消息循环必须直接处理它们，而不能交给 DispatchMessage 来处理。而且，这种通过消息来实现跨进程通信的方式仅限于在两个同属于一个桌面的进程之间才可以进行。

当应用程序调用 DispatchMessage 来处理一个消息时，如果此消息必须交给内核模式的代码来处理，那么，user32.dll 中的 DispatchMessage 函数将通过 win32k.sys 中的系统服务 NtUserDispatchMessage 来处理此消息；如果 DispatchMessage 能够处理该消息，那么，它可以直接调用窗口过程来处理此消息。有两种情况下是必须要进入内核的：这是一个系统定时器消息；消息的窗口类注明了这是一个内核窗口类型。Windows 子系统在用户模式下为每个窗口类和窗口对象维护了状态信息，因此，发送给用户模式窗口的消息通常可直接由用户模式的代码来完成，从而避免了模式切换的开销。

以上介绍了 Windows 子系统中的窗口管理和消息处理，在这一小节的最后，我们介绍一下 Windows 子系统的消息钩子（hook）机制。在 Windows 平台上，消息钩子是一种被广泛使用的编程手段，它使得一个程序可以将一段代码插入到当前线程或当前桌面上所有其他线程的消息处理过程中，前者称为局部钩子，后者称为全局钩子。钩子实际上是一个函数，全局钩子必须位于动态链接库中，因为所有包含 GUI 线程的进程都要加载全局钩子所在的模块，以便能调用钩子函数。Windows 子系统提供的安装钩子的 API 函数是 SetWindowsHookEx，对应的卸载钩子的 API 函数是 UnhookWindowsHookEx。每个线程可以有多个钩子，它们按照不同的类型形成一个或多个钩子链（hook chain）。Windows 子系统支持多种类型的钩子，如表 9.3 所列。

表 9.3 Windows 子系统支持的钩子类型

钩子类型	说明
WH_CALLWNDPROC	在系统发送一个消息到目标窗口过程之前调用的钩子函数
WH_CALLWNDPROCRET	在一个消息被目标窗口过程处理之后调用的钩子函数
WH_CBT	在接收 CBT（Computer-Based Training）事件之前调用的钩子函数，这里 CBT 事件是指系统在处理窗口重要事件、同步消息队列等时引发的各种消息
WH_DEBUG	用于调试其他的钩子函数
WH_FOREGROUNDIDLE	当应用程序的前台线程变成空闲时调用的钩子函数
WH_GETMESSAGE	当应用程序调用 GetMessage 或 PeekMessage 从消息队列中获取消息时调用的钩子函数
WH_JOURNALPLAYBACK	应用程序使用此钩子函数回放一系列由 WH_JOURNALRECORD 钩子记录下来的鼠标和键盘消息
WH_JOURNALRECORD	当子系统从系统消息队列中移除消息时，调用此钩子函数，因而该钩子函数可以记录下消息序列
WH_KEYBOARD	当应用程序调用 GetMessage 或 PeekMessage 获取一个键盘消息时调用的钩子函数
WH_KEYBOARD_LL	当一个键盘输入事件被插入到线程的输入队列中时调用的钩子函数
WH_MOUSE	当应用程序调用 GetMessage 或 PeekMessage 获取一个鼠标消息时调用的钩子函数
WH_MOUSE_LL	当一个鼠标输入事件被插入到线程的输入队列中时调用的钩子函数
WH_MSGFILTER	当对话框、消息框、菜单或滚动条中的键盘或鼠标操作导致产生的消息被处理时调用的钩子函数
WH_SHELL	通过此钩子函数可以接收系统的 Shell 事件的通知
WH_SYSMSGFILTER	当对话框、消息框、菜单或滚动条中的键盘或鼠标操作导致产生的消息被处理时调用的钩子函数，是全局钩子

Windows 子系统实现这些钩子的方式非常直截了当，它为每个 GUI 线程维护了一个钩子数组，允许各种类型的钩子加入到线程的钩子链中。注意，所有的钩子都是在内核模式下被激发的，但钩子函数本身却是用户模式函数。为了在内核模式下调用用户模式函数，Windows 子系统通过 KeUserModeCallback 函数来调用指定的钩子函数，必要时还要将钩子函数所在的模块加载到当前进程中。关于这一跨模式调用机制，读者可以参考 WRK 中 KeUserModeCallback 和 KiCallUserMode 函数的代码，这两个函数分别位于 base\ntos\ke\

i386 目录的 callback.c 和 callout.asm 文件中。简而言之，它们构造一个陷阱帧，利用标准的系统服务返回机制（参见 8.1.2 节），指定运行 ntdll.dll 的 KeUserCallbackDispatcher 函数，由它调用指定的用户模式回调函数（这里即钩子函数）。待钩子函数完成以后，KeUserCallbackDispatcher 函数通过 NtCallbackReturn 系统服务返回内核模式，回到 KeUserModeCallback 函数。图 9.11 显示了当钩子事件被激发时内核模式代码回调到用户模式钩子函数并返回内核的过程。

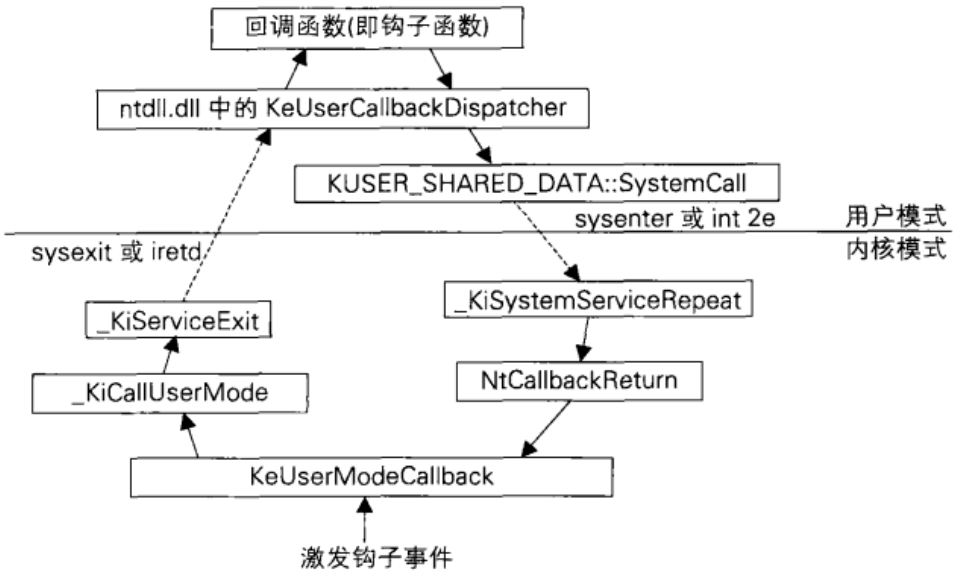


图 9.11 Windows 中从内核回调到用户模式再返回内核的执行过程

Windows 子系统在激发钩子事件时，并非简单地循环遍历钩子链上的每个钩子函数，相反地，它调用钩子链上的第一个钩子函数，该链上的后续钩子函数是否被调用到，取决于第一个钩子函数是否调用 API 函数 CallNextHookEx。此函数在用户模式下被调用，它又会进入到内核模式中，交给子系统处理该链上的下一个钩子。这是一个嵌套过程，但不会发生栈溢出，因为 KiCallUserMode 确保了线程的内核栈还可以再执行一次系统调用。如果因为内核栈无法增长而不能保证这一点，则 KiCallUserMode 直接返回（因而无法执行后面的钩子函数）。

9.2.4 GDI（图形设备接口）

图形用户界面是 Windows 操作系统的重要特色，这也是在 Windows 子系统中提供的。概括而言，Windows 的图形系统有两方面特点：首先，它提供了一套与设备无关的编程接

口，即 GDI，这使得应用程序可以适应各种底层显示设备的差异；其次，应用程序与图形设备驱动程序之间的通信足够高效，从而即使在频繁输出和刷新图形元素的情况下，Windows 也能够为用户提供良好的视觉效果。

从技术角度而言，Windows 子系统定义了一个稳定的图形体系结构，以便第三方的图形设备硬件厂商可以方便地将他们的视频显示器和打印设备集成到 Windows 中。图 9.12 显示了 Windows 子系统的图形显示部分的体系结构（适用于 Windows 2000 以后的系统）。

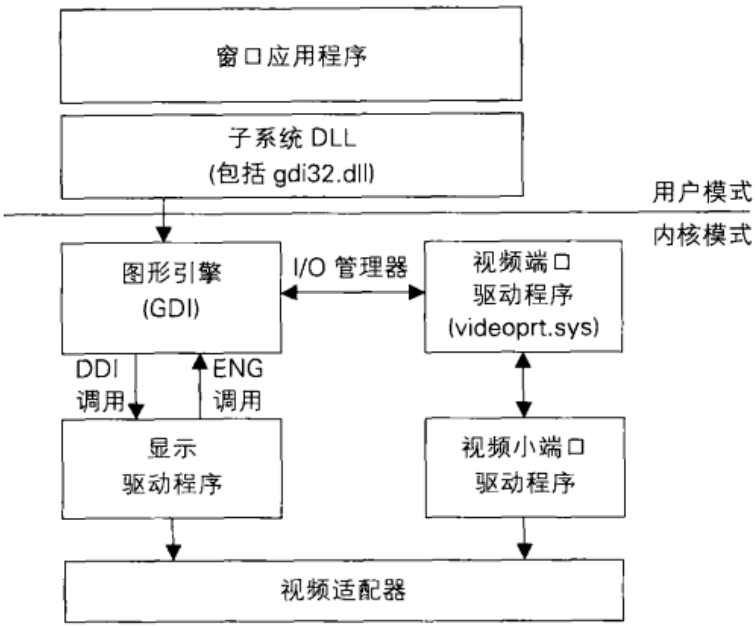


图 9.12 Windows 子系统的图形显示体系结构

正如 9.2.1 节所讲，在 Windows 的发展历史上，在 Windows NT 4 以前，图形显示是在子系统的用户模式进程（即 csrss.exe）中被处理的，而从 Windows NT 4 开始，这部分功能被转移到了内核中，允许应用程序的图形请求被直接通过显示驱动程序转发到视频适配器。在图 9.12 中，子系统 DLL 被动态链接到应用程序进程中，它们接收应用程序的图形请求，并将其转发到内核模式下的图形引擎中。

内核模式的图形引擎（即 Windows 子系统的 GDI）位于 win32k.sys 中，其职责是接受来自用户模式的图形请求，并且完成通用的、不涉及硬件的图形绘制功能，而把涉及硬件的或者需要特殊支持的图形功能交给显示驱动程序或者转发给视频端口驱动程序来完成。图形引擎在一个标准格式的位图（单个平面[plane]，像素可以是 1、4、8、16 或 32 位，每行上的像素连续存放，并且每行数据以 4 字节为对齐边界）上执行各种绘制操作。

如有必要，则通过 DDI（Device Driver Interface，设备驱动程序接口）调用显示驱动程序来完成相应的功能。图形 DDI 函数的名称都以“Drv”为前缀，例如显示驱动程序的入口例程，即初始化例程，为 DrvEnableDriver。

虽然显示驱动程序也被称为驱动程序，但是它并不遵守 I/O 管理器定义的接口标准，图形引擎与显示驱动程序之间的通信并没有采用 IoCallDriver 的调用方式。显示驱动程序本质上是一个动态链接库，它暴露了一组 DDI 函数，供图形引擎调用其功能；而图形引擎也暴露了一组 ENG 函数给显示驱动程序，从而使显示驱动程序可以调用图形引擎的功能。显示驱动程序支持的 DDI 函数都通过其入口函数（即 DrvEnableDriver）的输出参数（一个函数数组）被传递给图形引擎。关于 DDI 和 ENG 函数用法的一个典型例子是：图形引擎调用显示驱动程序的 DrvBitBlt 函数，将标准格式位图中的矩形块拷贝到显示器设备的图形表面（surface）中；显示驱动程序调用 EngBitBlt 函数，将它管理的图形表面拷贝到 GDI 管理的标准格式位图中。

图形引擎和显示驱动程序之间建立联系的初始化过程是这样来完成的：

- 在一个会话中，当 GDI 被第一次调用来创建一个设备环境（device context）时，图形引擎调用显示驱动程序的入口函数 DrvEnableDriver，从而获得显示驱动程序的版本，以及它所支持的 DDI 函数。
- 图形引擎接着调用 DrvEnablePDEV，以获得关于该驱动程序的物理设备的一个特征描述。对于 GDI，显示驱动程序代表了一个逻辑设备，它可以管理多个物理设备，每个物理设备有它自己的硬件类型、逻辑地址和图形表面（surface）。对于每个物理设备，显示驱动程序可以管理多个内存表示，即 PDEV 数据结构，这些内存表示描述了此设备的状态以及它的图形能力。在同一个物理设备的多个 PDEV 中，同一时刻只能使用一个。当图形引擎调用 DrvEnablePDEV 时，显示驱动程序根据指定的设备模式创建一个 PDEV。
- 当物理设备的 PDEV 被创建以后，图形引擎调用 DrvCompletePDEV，把它为该 PDEV 生成的一个句柄告诉显示驱动程序，以便将来显示驱动程序调用 ENG 接口函数时指定此 PDEV。
- 最后，图形引擎调用 DrvEnableSurface，让显示驱动程序为该物理设备创建一个图形表面，以代表最终的图形输出硬件。显示驱动程序可以有两种不同的方法来建立图形表面：

- 自己管理图形表面。做法是，调用 ENG 接口函数 EngCreateDeviceSurface，获得一个图形表面句柄。若图形硬件不支持标准格式位图，那么，显示驱动程序只能自己来管理图形表面。
- 完全由图形引擎来管理图形表面（即设备无关的位图）。如果图形硬件设备支持标准格式位图，那么，显示驱动程序可以调用 EngModifySurface 将设备管理的位图转换成图形引擎管理的位图。在这种情况下，显示驱动程序仍然可以截取任何一个绘制操作。

显示驱动程序配合图形引擎来实现图形绘制功能，图形引擎已经实现了在标准格式位图上的各种绘制能力。显示驱动程序可以把性能要求不高的图形操作，或者图形适配器不支持的复杂图形操作，转交给子系统的图形引擎来完成；然而，如果图形适配器支持一些特殊的加速操作，那么，显示驱动程序可以选择只实现跟这些加速操作相关的图形功能，而其他的都转交给图形引擎来完成。在编写显示驱动程序时，两个概念“hooking”和“punting to GDI”，分别指“显示驱动程序截取 GDI 的图形绘制操作”和“将图形绘制操作转交给 GDI 来完成”。简而言之，图形引擎完成了在标准格式位图上的所有图形操作，而显示驱动程序则往往有选择地实现部分图形操作，它的代码尺寸相对比较小。

以基本的画线操作为例，应用程序调用 gdi32.dll 中的 LineTo 函数，该函数将画线操作传递到 win32k.sys 中的 NtGdiLineTo 函数。如果当前使用的图形表面要求截取（hook）LineTo 操作，即显示驱动程序在调用 EngModifySurface 函数时指定了 HOOK_LINETO 标志，那么，NtGdiLineTo 函数将画线请求交给显示驱动程序的 DrvLineTo 函数。DrvLineTo 函数可以直接操纵显示适配器硬件，画出指定的线条，并返回 TRUE；它也可以返回 FALSE，以便将此画线任务传回给 GDI，即“punting to GDI”。如果当前图形表面未设置 HOOK_LINETO 标志（并且也没有设置 HOOK_STROKEPATH），那么，NtGdiLineTo 函数将画线请求交给 GDI 内核函数 EngLineTo。至于 EngLineTo 如何实现画线功能，有两种可能：若这是图形引擎自己管理的表面（必须是标准格式位图的表面），则它知道如何在当前表面上画线；若这是由显示驱动程序管理的图形表面，那么，画线操作最终还得由显示驱动程序来完成。由于 Windows 要求显示驱动程序至少实现三个绘制操作：DrvCopyBits、DrvStrokePath 和 DrvTextOut，所以 EngLineTo 可以将画线请求转变成 DrvStrokePath 调用。图 9.13 显示了画线操作几种可能的执行路径：①是交给显示驱动程序来决定画线操作；②是交给 GDI 内核函数 EngLineTo；③是交给显示驱动程序的 DrvStrokePath 函数，这是一个比 LineTo 画单条线更加通用的绘制函数；④是由 GDI 自己在标准格式位图上绘制一条线。

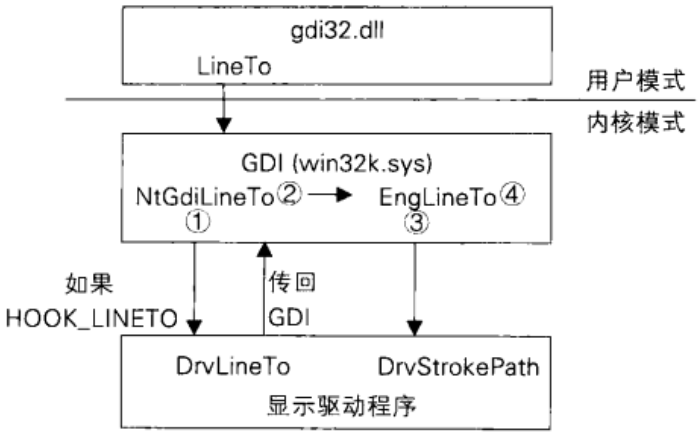


图 9.13 GDI 中画线功能 LineTo 的可能执行路径

在 Windows 子系统的图形体系结构中，显示驱动程序与视频小端口驱动程序是成对出现的，也就是说，对于一个视频适配器，硬件厂商除了要提供一个显示驱动程序以外，还需要提供一个配套的视频小端口驱动程序。图 9.12 中的视频端口驱动程序是由 Windows 操作系统本身提供的，在 Windows Server 2003 中视频端口驱动程序为 videoprt.sys，该文件实际上并非一个驱动程序，而是一个动态链接库。然而，视频小端口驱动程序是真正的驱动程序，而且往往是即插即用驱动程序，所以，它们由即插即用管理器在系统引导时加载到系统中，参见 6.3 节关于即插即用管理器的介绍。视频端口驱动程序暴露了一组以“VideoPort”为前缀的函数，而视频小端口驱动程序则以回调函数的形式暴露了一组以“HwVid”为前缀的函数。例如，VideoPortMapMemory 函数将视频存储区的一段物理地址范围映射到系统地址空间或者进程地址空间中；HwVidInitialize 函数执行适配器的初始化工作。

显示驱动程序和视频小端口驱动程序都可以访问视频适配器硬件，它们的分工原则是：显示驱动程序执行一些对用户可见的并且性能紧急的图形操作，例如，它直接访问适配器的视频存储区或者寄存器；小端口驱动程序执行一些不常用的图形操作，或者一些不能被中断打断或被其他进程抢占的关键图形操作。显示驱动程序不能处理设备中断。只有小端口驱动程序才可以配置视频适配器的存储区，并且将存储区映射到显示驱动程序可以访问的虚拟地址空间中。总体上，视频小端口驱动程序负责直接的硬件资源管理和控制，例如注册中断服务例程、定时器例程、DPC 例程，管理电源状态以及 DMA 传输等；视频端口驱动程序负责与图形处理有关的各种事项，包括缓存区管理、设备内存区管理与映射、读写 I/O 寄存器等，它也为小端口驱动程序提供一个与系统内核和执行体打交道的环境。

GDI 或显示驱动程序调用图形引擎的 EngDeviceIoControl 函数来跟视频小端口驱动程序进行通信。虽然视频小端口驱动程序是一个真正的设备驱动程序，但它的大部分功能逻辑是由视频端口驱动程序来完成的。在小端口驱动程序的初始化例程中，它必须调用视频

端口驱动程序的 VideoPortInitialize 函数，将自己的一些特殊处理例程通过一个数据结构 VIDEO_HW_INITIALIZATION_DATA 告诉视频端口驱动程序。至于创建设备对象、I/O 请求分发等设备驱动程序的常规逻辑，均由视频端口驱动程序代劳。小端口驱动程序链接到视频端口驱动程序上，它除了暴露一个入口函数作为初始化例程以外，其他所有的接口函数均以函数指针的形式告诉视频端口驱动程序。小端口驱动程序也可以调用视频端口函数 VideoPortGetProcAddress，以便动态地获知当前系统的视频端口驱动程序是否支持某个特定的视频端口例程，这一能力的好处是，为 Windows 某个版本开发的小端口驱动程序也可以在更早以前的 Windows 版本上运行。

因此，视频端口驱动程序是一个动态链接库，它不仅提供了控制视频适配器所需要的大部分功能，而且会将视频小端口驱动程序包装成一个真正的 I/O 设备驱动程序。小端口驱动程序除了在入口函数中调用 VideoPortInitialize 函数外，只需提供一组回调函数即可。图 9.14 显示了两者之间的关系。

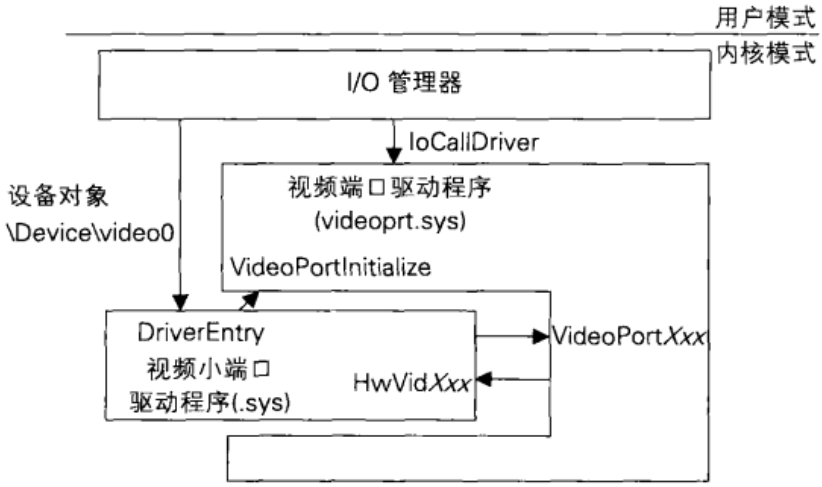


图 9.14 视频端口驱动程序与视频小端口驱动程序之间的关系

以上介绍了 Windows 子系统中的 GDI 部分以及与之相关联的显示驱动程序、视频端口驱动程序和视频小端口驱动程序。更详细的信息，读者可以参考 MSDN 中关于 Windows 2000 显示驱动程序模型的文档资料[MSDN-W2K-GDI]，以及 Windows DDK 提供的驱动程序例子 mirror。Windows XP 和 Windows Server 2003 仍然沿袭了 Windows 2000 的显示驱动程序模型。

在 Windows 子系统的显示驱动程序模型中，除了 GDI 部分，它还提供了对 Microsoft DirectX 的显示支持，允许像游戏、多媒体播放器等应用软件直接操纵显示器硬件，绕过前面所提到的 GDI 图形引擎，从而获得更快的显示速度，并且避免屏幕抖动。如表 9.2 所示，Windows 子系统的 NtGdiDd<*Xxx*>和 NtGdiD3d<*Xxx*>系统服务正是为这类应用软件提供的 DirectDraw 和 Direct3D 支持。

DirectDraw 是 DirectX 中的显示部件，是一套与 Windows 子系统 GDI 平行的图形编程接口。由于 DirectDraw 驱动程序充分发挥了硬件的图形加速能力，所以，DirectDraw 在快速运动和持续屏幕变化等图形特性方面比 GDI 有更强的处理能力和执行效率。我们可以认为 DirectDraw 是 GDI 的一个增强，更加适用于像场景游戏和多媒体播放器这样的应用软件。在 Windows 2000 及以后的系统上，DirectDraw 包含以下组件：

- 用户模式 DirectDraw，这是一个动态链接库（ddraw.dll），它提供了硬件仿真、管理各种 DirectDraw 对象、管理显示存储区和显示器硬件的服务。DirectDraw 为应用程序提供了基于 COM（Component Object Model）的接口，所以，使用 DirectDraw 的软件必须是一个 COM 应用。
- 内核模式 DirectDraw，如同 GDI 一样，它也是 Windows 子系统的 win32k.sys 中的一部分。内核模式 DirectDraw 的职责是，检查 DirectDraw 系统调用的参数的有效性，并且负责与 GDI 的同步，以及维护各种跨进程的状态。它提供的系统服务以“NtGdiDd”和“NtGdiDvp”为前缀。
- DirectDraw 驱动程序，这是由图形硬件厂商提供的软件组件。从形式来看，它既可以是显示驱动程序中的一个组成部分，也可以是一个独立的驱动程序。DirectDraw 驱动程序的核心是利用硬件的能力来提供更快、更强的图形显示功能。它提供各种回调函数供子系统调用，这组回调函数涉及图形表面（surface）、颜色控制和调色板、运动补偿（motion compensation）、视频端口，以及驱动程序内部的数据对象管理等。在 Windows DDK 文档中这些回调函数的前缀为“Dd”，例如 DdCreateSurface。

DirectDraw 驱动程序的结构与显示驱动程序很相似，它并非 I/O 设备驱动程序，而是一个动态链接库，其初始化函数 DrvEnableDirectDraw 负责硬件初始化。DrvEnableDirectDraw 返回 DirectDraw 驱动程序实现的三组回调函数，分别对应于调色板回调、图形表面回调，以及另外几个基本的回调函数。此外，Windows 子系统通过 DdGetDriverInfo 函数可以进一步获得其他的回调函数，通过 DrvGetDirectDrawInfo 函数可以获得图形硬件的能力。从 DrvEnableDirectDraw 和 DrvGetDirectDrawInfo 这两个函数的命名也可以看出，Windows 子系统对 DirectDraw 的支持实际上也是 GDI 的一个扩展或延伸（这两个关于 DirectDraw 的函数以“Drv”为前缀），DirectDraw 驱动程序是显示驱动程序的扩展。关于 Windows 子系统对 DirectDraw 的支持的更多信息，可参考 MSDN 中的文档[MSDN-DD]。

DirectX 中也包含了对于三维图形的支持，称为 Direct3D。如同 DirectDraw 一样，Windows 子系统也支持 Direct3D。以“NtGdiD3d”为前缀的系统服务代表了 Windows 子系统提供给应用程序的 3D 显示支持。Direct3D 驱动程序也是一个动态链接库，它可以是

显示驱动程序或者 DirectDraw 驱动程序的一部分。Windows 子系统通过 DirectDraw 函数 DdGetDriverInfo 来获得 Direct3D 的回调函数，因此，Direct3D 也可以看做是 DirectDraw 的一个自然扩展。关于 Windows 子系统对 Direct3D 支持的更多信息，可参考 MSDN 中的文档[MSDN-D3D]。

9.2.5 Windows Vista 及以后的子系统变化

尽管 Windows 子系统一直保持相对稳定的接口，从而使 Windows 应用程序总是能够在新的系统平台上运行，但是，它同时也在不断进化和发展。前面小节中关于 Windows 子系统的描述适用于 Windows 2000/XP/Server 2003，并且在其后的操作系统（包括 Windows Vista/Server 2008/7）中仍然保持兼容。然而，从 Windows Vista 开始，Windows 子系统又有了新的发展，其中比较突出的两个变化是，引入了桌面窗口管理器（DWM，Desktop Window Manager）和一套新的显示模型。下面对此作一简单介绍。

（一）DWM——桌面窗口管理器

Windows Vista 新引入了一种称为“桌面合成（desktop composition）”的特性，它彻底改变了应用程序在屏幕上显示像素的方法。一旦桌面合成特性被启用，则每个窗口不再像以前那样，直接在屏幕或显示设备上绘制，而是先在一个后台的视频存储区图形表面（surface）上绘制，然后才被渲染到一个桌面图像并显示出来。正是这种桌面合成特性，使得 Windows 可以支持各种视觉效果，例如透明窗口框、三维的窗口变换动画效果、窗口翻转（甚至三维翻转），以及高分辨率支持。

Windows 的桌面合成特性是通过 DWM 来实现的。DWM 提供了一组专门的 API，用于控制和管理窗口的各种效果。应用程序调用 DwmEnableComposition 函数可以启用或关闭桌面合成特性。注意，桌面合成特性是自动启用的，除非有一个程序显式地关闭它；而且，当所有关闭桌面合成特性的应用程序都退出以后，系统又会自动启用它。如果一个应用程序试图直接在主显示器的图形表面上绘制，那么 DWM 会自动关闭桌面合成特性，直至该图形表面被释放。关闭桌面合成特性对于有些应用程序是非常有意义的，比如游戏或多媒体播放器可能需要运行在全屏模式下。

DWM 的典型视觉效果是窗口非客户区域（包括标题栏和边框等）的半透明（translucent）和背景模糊（blur behind）效果。应用程序也可以通过调用 DwmEnableBlurBehindWindow 函数，让整个窗口或窗口中特定的区域具有背景模糊效果，或者调用 DwmExtendFrameIntoClientArea 函数，让窗口框扩展到窗口的客户区域中。

另外，DWM 还支持一个新特性，即窗口的动态缩略图（DWM Thumbnail）。DWM

缩略图并非一个窗口的静态快照，而是一个源窗口与一个目标窗口之间的渲染关系。应用程序调用 DwmRegisterThumbnail/DwmUnregisterThumbnail 函数可以注册/注销这一关系；当应用程序调用 DwmUpdateThumbnailProperties 函数时，目标窗口的缩略图就会被更新。DWM 缩略图的两个典型应用是：鼠标在任务栏上移动时，自动出现每个应用程序的当前缩略图；使用 Alt-Tab 键切换应用程序时，可以看到每个应用程序当前的运行状况。

尽管 DWM 使用了 GPU（图形处理单元）来协助处理各种效果，但是在有些情形下仍然是十分消耗资源的，包括存储和计算资源。例如，客户区域的背景模糊往往需要较多计算资源，因而应用程序须谨慎使用。有关 DWM 的详细信息，请参考 MSDN 中的文档[MSDN-DWM]。

（二）Windows 子系统新的显示模型

Windows Vista 采用了一种新的显示驱动程序模型，该模型要求显示设备硬件厂商提供一个用户模式的显示驱动程序和一个内核模式的显示小端口驱动程序。用户模式显示驱动程序是一个动态链接库，由 Direct3D 运行时库加载到应用进程中；显示小端口驱动程序与 DirectX 图形内核子系统进行通信。图 9.15 显示了 Windows Vista 的显示驱动程序模型。

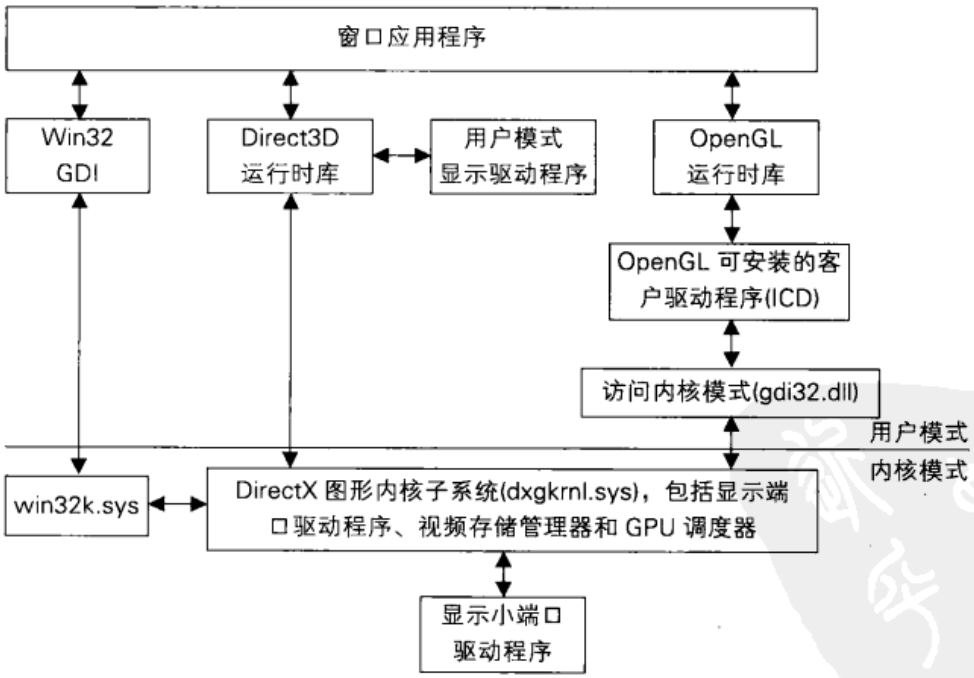


图 9.15 Windows Vista 引入的显示驱动程序模型

Windows Vista 显示驱动程序模型也支持 OpenGL 接口。OpenGL 运行时库通过注册表来决定应该加载哪个“OpenGL 可安装的客户驱动程序”（ICD，Installable Client Driver）。

OpenGL ICD 与用户模式显示驱动程序类似，它们可以获得相同的内核模式服务的支持。然而，两者使用系统服务的方式和路径不同。用户模式显示驱动程序通过 Direct3D 运行时库提供的回调函数来调用系统服务，而 OpenGL ICD 必须加载 gdi32.dll，通过它的导出函数来调用一组内核模式的 DirectX 系统服务。

Windows Vista 显示驱动程序模型倾向于将更多的代码逻辑提升到用户模式下，从图 9.15 的结构模型可以看出，显示驱动程序已转移到用户模式下。该模型的另一个显著特点是，Direct3D 运行时库和 DirectX 图形内核子系统承担了更多的显示处理任务，包括管理视频存储区、为 GPU 管理和调度 DMA 缓冲区等。DirectX 图形内核子系统也兼并了以前模型中的视频端口驱动程序。显示小端口驱动程序的工作方式与以前模型中的视频小端口驱动程序类似，它在初始化例程中调用 DxgkInitialize 函数，将自己实现的回调函数告诉 DirectX 图形内核子系统。在新的模型中，DirectX 图形内核子系统和显示小端口驱动程序更加一致地控制和管理显示适配器。有关 Windows Vista 显示驱动程序模型的更多详细信息，请参考 MSDN 中的文档[MSDN- VISTA-DDM]。

9.3 内核日志

正如 2.5.3 节所讲，Windows 内核中的 WMI 驱动程序（其名称为“WMIxWDM”）实现了一个内核日志记录器。该内核日志记录器也是一个 ETW 提供者，可以记录内核中发生的各种事件，例如，我们在 WRK 中可以看到线程环境切换事件（CSwitch）的记录过程。本节我们首先看一下内核日志记录器，然后讨论如何利用内核日志信息来分析系统的性能问题。

9.3.1 内核日志记录器

在 Windows Server 2003 SP1 中，尽管内核日志记录器并没有提供所有的内核事件支持（即 public\internal\base\inc\ntwmi.h 文件中的 PERF_<*XXX*>定义），但是从 WRK 的代码中我们仍然可以看到一些关键的内核事件是如何记录下来的。表 9.4 列出了在 WRK 中支持的主要（并非全部）内核事件。

表 9.4 WRK 中内核日志记录器支持的主要内核事件

内核事件类	记录相应事件的出调函数	触发事件的模块或内核例程
磁盘 I/O	WmipTraceIo	分区管理驱动程序
文件 I/O	WmipTraceFile	若 EVENT_TRACE_FLAG_DISK_FILE_IO 被指定，则随磁盘 I/O 一起记录文件 I/O 事件

续表

内核事件类	记录相应事件的出调函数	触发事件的模块或内核例程
映像加载	WmipTraceLoadImage	当加载可执行映像文件时记录事件，参见 MiMapViewOfImageSection 调用 PsCallImageNotifyRoutines 函数的代码
页面错误	WmipTracePageFault	当发生硬页面错误时由内存管理器触发（MmAccessFault）
进程创建/删除	WmiTraceProcess	PspCreateProcess 和 PspExitProcess
线程创建/删除	WmiTraceThread	PspCreateThread 和 PspExitThread
注册表事件	WmipTraceRegistry	注册表服务例程，例如 NtOpenKey、NtCreateKey、NtQueryKey 等
网络事件	WmipTraceNetwork	TCP/IP 驱动程序（tcpip.sys）
线程切换	WmiTraceContextSwap	当内核切换线程时（SwapContext 汇编函数）
性能采样	PerfProfileInterrupt	性能中断服务例程 KeProfileInterrupt（可由 Kernrate 工具设定）
中断	PerfInfoLogInterrupt	在中断分发例程中被触发
DPC/TimerDPC	直接调用 PerfInfoLogBytes	当 DPC 例程被执行时，见 KiExecuteDpc、KiRetireDpcList 和 KiProcessTimerDpcTable

从表 9.4 可以看出，内核日志记录器中的事件触发点散布在内核各处。所有的内核事件在内存中都有一个头，头后面是数据体。下面是事件头数据结构 PERFINFO_TRACE_HEADER 的定义（见 public\internal\base\inc\ntwmi.h）：

```
typedef struct _PERFINFO_TRACE_HEADER {
   union {
      ULONG      Marker;                    // 标记信息，由三部分构成
      struct {
         USHORT  Version;                   // 版本
         UCHAR   HeaderType;                // 头部类型
         UCHAR   Flags;                     // 标志，WMI 使用它来标识事件类型
      };
   };
   union {
      ULONG          Header;                // Packet 成员的大小必须与 ULONG 相等
      WMI_TRACE_PACKET Packet;              // 内有一个 Size 域和一个 HookId 成员
                                            // HookId 又可以分成 Type 和 Group 两个子域
   };
   union {
      PERFINFO_TIMESTAMP TS;                // 时间戳
      LARGE_INTEGER      SystemTime;        // 系统时间
   };
   UCHAR Data[1];                           // 事件数据
} PERFINFO_TRACE_HEADER, *PPERFINFO_TRACE_HEADER;
```

在事件头后面的数据的结构随事件类型的不同而不同，关于如何记录每一种事件，请参考表 9.4 中列出的出调函数。Xperf 工具[MSDN-WPT]在转储出一个事件痕迹文件（.etl 文件）时也会将每种事件的表结构（schema）列示出来，如图 9.16 所示。关于这些内核事件中各个域的含义，请参考相关的文档[MSDN-ET]或 WRK 中的有关代码。

```
CSwitch,  TimeStamp, New Process Name ( PID),    New TID, NPri, NQnt, TmSinceLast, WaitTime, Old Process Name ( PID),    Old TID, OPri, OQnt,
                     续上行 OldState,      Wait Reason, Swapable, InSwitchTime, CPU, IdealProc,  OldRemQnt, NewPriDecr, PrevCState

P-Start,  TimeStamp,     Process Name ( PID),  ParentPID,  SessionID,  UniqueKey, UserSid, Command Line
P-End,  TimeStamp,     Process Name ( PID),  ParentPID,  SessionID,  UniqueKey,     Status, UserSid, Command Line

T-Start,  TimeStamp,     Process Name ( PID),   ThreadID,  StackBase, StackLimit, UsrStkBase,  UsrStkLmt,    TebBase, SubProcessTag,  Image!Function
T-End,  TimeStamp,     Process Name ( PID),   ThreadID,  StackBase, StackLimit, UsrStkBase,  UsrStkLmt,    TebBase, SubProcessTag,    Image!Function

ImageId,  TimeStamp,     Process Name ( PID),   BaseAddr,  ImageSize, TimeDateStamp, OrigFileName

DiskRead,  TimeStamp,     Process Name ( PID),   ThreadID,     IrpPtr,   ByteOffset,     IOSize, ElapsedTime,  DiskNum, IrpFlags, DiskSvcTime,
                     续上行 I/O Pri,  VolSnap, FileObject, FileName
DiskWrite,  TimeStamp,     Process Name ( PID),   ThreadID,     IrpPtr,   ByteOffset,     IOSize, ElapsedTime,  DiskNum, IrpFlags, DiskSvcTime,
                     续上行 I/O Pri,  VolSnap, FileObject, FileName
DiskFlush,  TimeStamp,     Process Name ( PID),   ThreadID,     IrpPtr,                                ElapsedTime,  DiskNum, IrpFlags, DiskSvcTime, I/O Pri

FileIoCreate,  TimeStamp,     Process Name ( PID),   ThreadID, CPU,     IrpPtr, FileObject,    Options, Attributes,ShareAccess, FileName,
                     续上行 ParsedOptions, ParsedAttributes, ParsedShareAccess
FileIoClose,  TimeStamp,     Process Name ( PID),   ThreadID, CPU,     IrpPtr, FileObject, FileName
FileIoFlush,  TimeStamp,     Process Name ( PID),   ThreadID, CPU,     IrpPtr, FileObject, FileName
FileIoRead,  TimeStamp,     Process Name ( PID),   ThreadID, CPU,     IrpPtr, FileObject,   ByteOffset,  Size,  Flags, Priority, FileName, ParsedFlags
FileIoWrite,  TimeStamp,     Process Name ( PID),   ThreadID, CPU,     IrpPtr, FileObject,   ByteOffset, Size,  Flags, Priority, FileName, ParsedFlags

PageFault,  TimeStamp,     Process Name ( PID),   ThreadID,VirtualAddr,   PrgrmCtr,        Type,            Image!Function
HardFault,  TimeStamp,     Process Name ( PID),   ThreadID,VirtualAddr,   ByteOffset,     IOSize, ElapsedTime, FileObject, FileName,
                     续上行 Hardfaulted Address Information

RegCreateKey,  TimeStamp,        KCB,     Process Name ( PID),   ThreadID,     Status, ElapsedTime, Key Name
RegOpenKey,  TimeStamp,        KCB,     Process Name ( PID),   ThreadID,     Status, ElapsedTime, Key Name
RegDeleteKey,  TimeStamp,        KCB,     Process Name ( PID),   ThreadID,     Status, ElapsedTime, Key Name
RegQueryKey,  TimeStamp,        KCB,     Process Name ( PID),   ThreadID,     Status, ElapsedTime, InfoClass
RegSetvalue,  TimeStamp,        KCB,     Process Name ( PID),   ThreadID,     Status, ElapsedTime, Key Name

TcpSend,  TimeStamp,     Process Name ( PID), Transfer Size, Remote IPv4 Addr, Remote IPv4 Port, Local IPv4 Addr, Local IPv4 Port
TcpRecv,  TimeStamp,     Process Name ( PID), Transfer Size, Remote IPv4 Addr, Remote IPv4 Port, Local IPv4 Addr, Local IPv4 Port
TcpConnect,  TimeStamp,     Process Name ( PID), Transfer Size, Remote IPv4 Addr, Remote IPv4 Port, Local IPv4 Addr, Local IPv4 Port
TcpDisconnect,  TimeStamp,     Process Name ( PID), Transfer Size, Remote IPv4 Addr, Remote IPv4 Port, Local IPv4 Addr, Local IPv4 Port
TcpRetransmit,  TimeStamp,     Process Name ( PID), Transfer Size, Remote IPv4 Addr, Remote IPv4 Port, Local IPv4 Addr, Local IPv4 Port
TcpAccept,  TimeStamp,     Process Name ( PID), Transfer Size, Remote IPv4 Addr, Remote IPv4 Port, Local IPv4 Addr, Local IPv4 Port

DPC,  TimeStamp, ElapsedTime, CPU,ServiceAddr,            Image!Function
DPCTmr,  TimeStamp, ElapsedTime, CPU,ServiceAddr,            Image!Function
Interrupt,  TimeStamp, ElapsedTime, CPU,     Status,ServiceAddr,            Image!Function
SampledProfileNmi,  TimeStamp,     Process Name ( PID),   ThreadID,   PrgrmCtr, CPU, ThreadStartImage!Function,     Image!Function, Interrupts Masked
SampledProfile,  TimeStamp,     Process Name ( PID),   ThreadID,   PrgrmCtr, CPU, ThreadStartImage!Function, Image!Function, Count, SampledProfile type
```

图 9.16　WRK 中记录的内核事件的表结构

随着 Windows 的内核日志记录器的日益完善，自 Windows Vista 起，可被记录的事件更为全面，几乎覆盖到了内核中所有重要的模块。而且，对于许多重要的事件，事件发生时的调用栈（包括内核栈和用户栈）也可被记录下来。下一小节我们讨论如何使用内核日志信息来诊断系统性能问题。

9.3.2 利用内核日志信息诊断性能问题

Windows Vista 及以后的系统能够记录非常全面的内核事件，加之 Microsoft 发布了 WPT（Windows Performance Toolkit，包含在 Windows SDK 中）和 xperf 作为对性能分析的工具支持[MSDN-WPT]，因此，许多复杂的性能问题有望通过内核中捕获的信息得以正确诊断并解决。这可以看做是 Windows Vista 以后的一个重要改进措施，也使内核日志记录器得以真正发挥效用。这甚至可以看成是内核技术走向成熟的一个标志。

随着操作系统和应用软件的功能日益丰富，软件功能模块趋向于更加复杂，而软件模块之间的交互也日趋频繁。在 Windows 平台上，内核、系统 DLL、Windows 子系统是 Windows 应用程序运行所必需的模块。尽管 Windows 程序只需简单地调用 Windows API 就可以与它们打交道，但不恰当地使用 Windows API 有可能造成应用程序未能按照预期的性能指标来运行。在很多情况下，单单从应用程序的角度难以诊断性能问题，而利用系统范围的日志数据则有可能相对容易暴露出问题的根源。譬如，以下就是一些典型的可通过 ETW 内核日志来诊断的性能问题：

- 对用户动作的响应迟缓。当用户触发输入事件（比如鼠标或键盘）时，应用程序并不能立即响应。常见的“没有响应（Not Responding）”现象属于这一类性能问题。
- 应用程序中的长时间延迟。有些任务偶尔需要很长时间才能完成，譬如，启动一个应用程序、刷新一个窗口等。有时候也会造成“没有响应”现象。
- 高 CPU 使用率。系统的 CPU 使用率达到 100%，导致系统一段时间内不能响应。
- 高磁盘使用率。大量的磁盘操作导致系统变慢。
- 开关状态转移很慢。这里的开关状态转移包括开机启动、系统停机、休眠或睡眠，以及系统恢复（resume）等系统层面上的状态变化。
- 电池寿命比预期的要短。由于某些原因（例如，无法进入低耗电或睡眠状态），导致电量被很快消耗光。

显然，以上这些情形往往涉及一个过程，而并非一个出错点，而且可能涉及多个应用程序或者系统组件，所以，用调试器难以有效地找到问题所在。相反地，通过 ETW 的内核日志记录器或更多 ETW 提供者，记录下性能问题发生过程中的系统日志信息，并转储出来进行分析，则有可能较为方便地（自动或手动）找到问题的根源所在。这便是 ETW 和 WPT 所提供的性能分析方法。

首先，xperf 作为一个 ETW 控制器，它可以启动一个记录会话；当会话结束时，它将缓冲区中的日志数据转储到一个.etl 文件中。记录会话一旦被启动，内核日志记录器本身也会造成一定的额外开销，包括处理器负荷和内存开销。ETW 的事件记录和缓冲区管理已经做了充分的优化，因此日志记录的开销相对较轻，例如，在 2 GHz 处理器的 Windows Vista 系统中每秒钟记录 10 000 个事件，CPU 使用率大约为 2.5%。

Xperf 在启动记录会话时可以指定要记录哪些内核事件，也可以指定缓冲区的数量范围，以及日志文件的最大限制。例如，下面的命令：

```
xperf -on DiagEasy
```

启动内核记录会话，这里 DiagEasy 是一个事件组，根据“xperf–providers k”命令的提示：

```
DiagEasy : PROC_THREAD+LOADER+DISK_IO+HARD_FAULTS+DPC+INTERRUPT+CSWITCH+PERF_COUNTER
```

DiagEasy 包含常见的内核事件，通常而言，这些事件足以分析很多性能问题了。为了将某些关键事件的调用栈也记录到事件中，可以使用-stackwalk 选项。例如，我们可以使用下面的命令来打开 CSwitch 和 ReadyThread 事件的调用栈开关：

```
xperf -on DiagEasy+DISPATCHER -stackwalk CSWITCH+READYTHREAD
```

这里，DISPATCHER 是一个内核记录标志，它包含了 ReadyThread 事件。注意，按上述命令启动了记录会话以后，日志数据将被转存到默认的文件“\kernel.etl”中。为了停止记录会话，并且将日志数据合并和转存到指定的日志文件中，可以使用以下命令：

```
xperf -d mytrace.etl
```

经过这一步骤，我们便得到了日志文件，其中包含了在此会话过程中发生的内核事件。注意，.etl 文件是二进制文件格式，xperf 也提供了将.etl 文件转成文本文件的能力，下面的命令可以做到这一点：

```
xperf -i mytrace.etl -o mytrace.txt -symbols verbose
```

为了将内核事件中的函数地址转储成更为友好的符号名称，而非文本形式的十六进制函数地址，xperf 支持符号解析，它使用环境变量_NT_SYMBOL_PATH 设置的符号路径来解析符号。环境变量_NT_SYMBOL_PATH 也是 windbg 所使用的符号路径，关于如何

设置符号路径，请参考附录 A 关于使用 windbg 来调试 Windows 内核的介绍。上面命令行中的“-symbols verbose”选项指示 xperf 解析详细的符号信息。这里我们假设 _NT_SYMBOL_PATH 指向 Microsoft 提供的公共符号服务器：

```
srv*C:\symbols*http://msdl.microsoft.com/downloads/symbols
```

文本格式的日志文件便于查看和检查每一个细节，xperf 转储出来的文本日志文件的头部是该文件中的事件的表结构，图 9.16 中显示的表结构正是从 xperf 转储一个 Windows Server 2003 日志文件的文本输出中摘取的。下面的一对 ReadyThread 和 CSwitch 事件是从 Windows 7 的一个日志文件中摘录下来的，为便于理解这两个事件的域的含义，它们的表结构也列示了出来：

```
CSwitch, TimeStamp, New Process Name ( PID), New TID, NPri, NQnt, TmSinceLast, WaitTime,
Old Process Name ( PID), Old TID, OPri, OQnt, OldState, Wait Reason, Swapable,
InSwitchTime, CPU, IdealProc, OldRemQnt, NewPriDecr, PrevCState

ReadyThread, TimeStamp, Process Name ( PID), ThreadID, Rdy Process Name ( PID), Rdy
TID, AdjustReason, AdjustIncrement, CPU, InDPC
……

ReadyThread, 236493, csrss.exe ( 436), 524, iexplore.exe (5364), 4032, Unwait, 2, 1,
Stack, 236493, 524, 1, 0x82a8c59a, ntkrnlpa.exe!KiReadyThread
Stack, 236493, 524, 2, 0x82a8c567, ntkrnlpa.exe!KiProcessThreadWaitList
Stack, 236493, 524, 3, 0x82a867b1, ntkrnlpa.exe!KiExitDispatcher
Stack, 236493, 524, 4, 0x82a86777, ntkrnlpa.exe!KeSetEvent
Stack, 236493, 524, 5, 0x9474aa8e, win32k.sys!SetWakeBit
Stack, 236493, 524, 6, 0x947491fe, win32k.sys!TimersProc
Stack, 236493, 524, 7, 0x9469843c, win32k.sys!RawInputThread
Stack, 236493, 524, 8, 0x94698586, win32k.sys!xxxCreateSystemThreads
Stack, 236493, 524, 9, 0x9475c2c8, win32k.sys!NtUserCallNoParam
Stack, 236493, 524, 10, 0x82a6242a, ntkrnlpa.exe!KiFastCallEntry
Stack, 236493, 524, 11, 0x75a93ed1, winsrv.dll!NtUserCallNoParam
Stack, 236493, 524, 12, 0x75a93188, winsrv.dll!StartCreateSystemThreads
Stack, 236493, 524, 13, 0x7797b3c8, ntdll.dll!_RtlUserThreadStart

CSwitch, 236559, iexplore.exe (5364), 4032, 12, -1, 91992, 0, csrss.exe (436), 524, 15,
-1, Waiting, WrUserRequest, NonSwap, 84, 1, 1, 14359552, 2, 1
Stack, 236559, 4032, 1, 0x82a86ce2, ntkrnlpa.exe!SwapContext_XRstorEnd
Stack, 236559, 4032, 2, 0x82a8c403, ntkrnlpa.exe!KiCommitThreadWait
Stack, 236559, 4032, 3, 0x82a862cf, ntkrnlpa.exe!KeWaitForSingleObject
Stack, 236559, 4032, 4, 0x94750d75, win32k.sys!xxxRealSleepThread
Stack, 236559, 4032, 5, 0x94750e10, win32k.sys!xxxSleepThread
Stack, 236559, 4032, 6, 0x9470bd56, win32k.sys!xxxRealWaitMessageEx
Stack, 236559, 4032, 7, 0x94741eba, win32k.sys!NtUserWaitMessage
Stack, 236559, 4032, 8, 0x773f93f3, user32.dll!NtUserWaitMessage
Stack, 236559, 4032, 9, 0x69f80703, ieframe.dll!CTabWindow::_TabWindowThreadProc
Stack, 236559, 4032, 10, 0x69f90446, ieframe.dll!LCIETab_ThreadProc
Stack, 236559, 4032, 11, 0x763349bd, iertutil.dll!CIsoScope::RegisterThread
Stack, 236559, 4032, 12, 0x75f81174, kernel32.dll!BaseThreadInitThunk
Stack, 236559, 4032, 13, 0x7797b3f5, ntdll.dll!__RtlUserThreadStart
Stack, 236559, 4032, 14, 0x7797b3c8, ntdll.dll!_RtlUserThreadStart
```

这两个事件都发生在 1 号 CPU 上，ReadyThread 事件是由 csrss.exe 进程的 524 号线程所触发，因为该线程执行了 KeSetEvent 函数，使一个 KEVENT 事件变成有信号状态；而恰好 iexplorer.exe 进程的 4032 号线程当时正在等待此对象，因而 524 号线程使 4032 号线程的等待条件可以满足。这正是 ReadyThread 事件的含义。接下来的 CSwitch 事件说明了 4032 号线程很快获得了处理器资源。该处理器上原来的执行线程是 524，现在切换到了 4032 号线程。按照这样的方式顺序地分析一个日志文件，我们实际上可以了解一个处理器在一段时间内是如何在不同线程之间切换的。由于这些事件包含了调用栈，所以，如果能够自动地把性能问题发生过程中关键事件的调用栈呈现给软件开发人员，则软件中的不合理设计或不正确的实现代码将有可能被快速地定位出来，并得以修正。

Xperf 除了作为控制器生成.etl 日志文件，还提供了强大的分析功能，使得日志文件中的系统行为能够以可视化的形式呈现出来，从而方便日志分析人员发现和追踪问题。例如，图 9.17 显示了在 Windows 7 系统上运行 xperf 打开一个.etl 日志文件的界面。

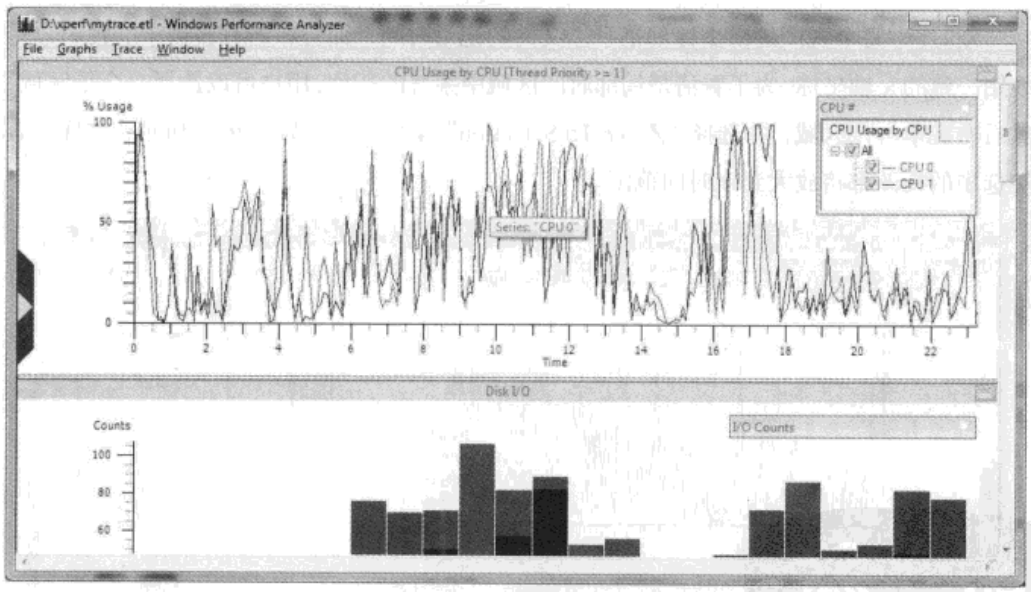


图 9.17 在 xperf 中打开一个.etl 日志文件

Xperf 在显示一个.etl 日志文件时支持多种视图，图 9.17 上半部分显示的是 CPU 使用率视图，其中每个处理器一条曲线，横坐标为日志文件中记录的时间，以秒为单位。图 9.17 下半部分显示的是磁盘 I/O 视图。单击窗口左侧的图表选择按钮，可以列出 xperf 支持的各种视图，如图 9.18 所示。

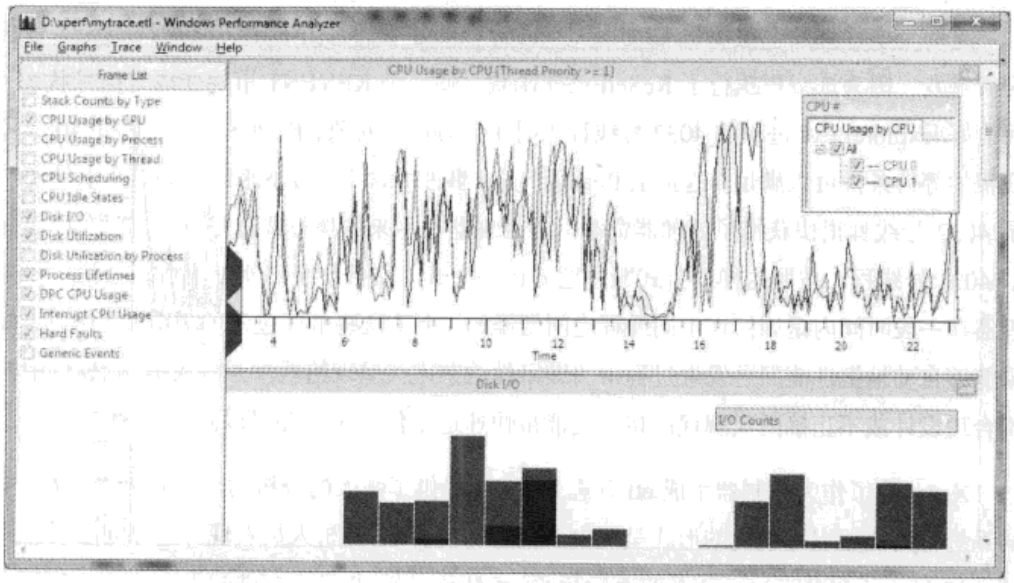


图 9.18 xperf 的图表选择功能

Xperf 中每个视图的横坐标都是日志文件的记录时间，而且，所有被选中的视图总是使用一致的 x 轴坐标。为了看清楚局部时间区域中系统的行为，用户可以选择一个子区域，然后右键单击该区域，并选择“Zoom To Selection”菜单命令，如图 9.19 所示，这样所有被选中的视图都将放大到该时间范围。

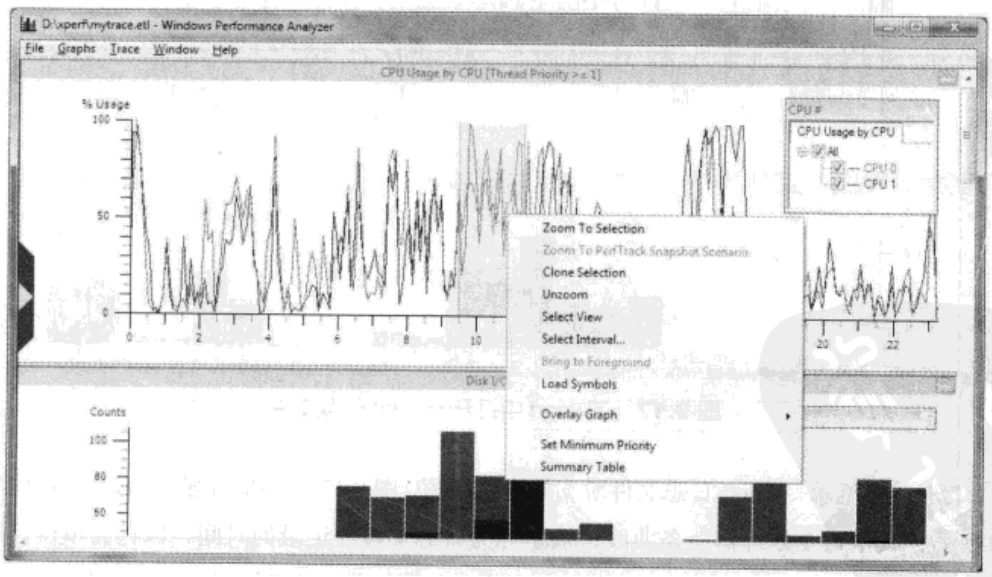


图 9.19 选择一个时间区域并放大至该时间范围

除了独立地显示每一个视图以外，用户也可以让这些视图叠加在同一个显示区域（通过右键菜单中的“Overlay Graph”命令），例如图 9.20 就显示了在 CPU 使用率视图中叠

加上磁盘 I/O 视图的结果。同样地，在这种情况下，放大至子区域（Zoom To Selection）的功能仍然可以使用。

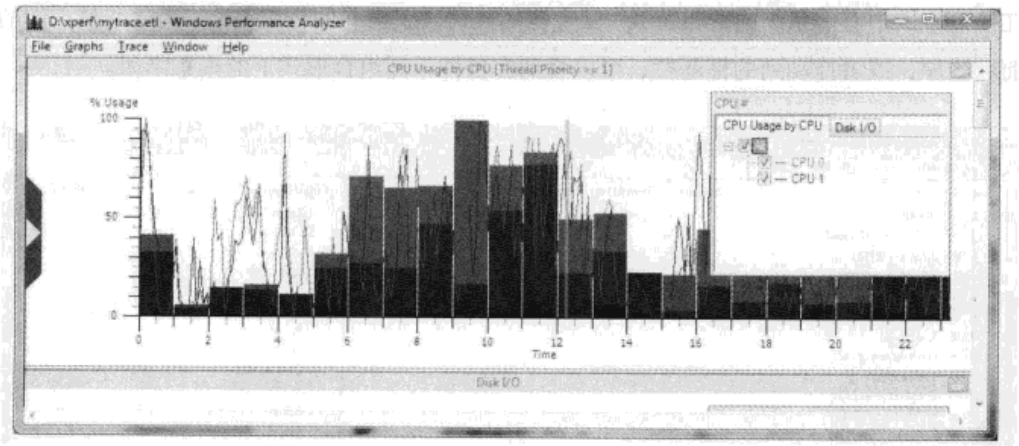


图 9.20 CPU 使用率和磁盘 I/O 两个视图叠加在一起

Xperf 不仅能清晰地显示一段时间内系统的总体情况，也能显示一些细节行为。例如 xperf 的磁盘 I/O 细节视图显示了一段时间内的磁盘访问模式，用户从中可以看出磁盘 I/O 操作的偏移位置情况。这些偏移位置基本上反映了磁盘读写头的移动情况。图 9.21 显示了磁盘 I/O 细节视图，图中的每个点代表了一个或一系列磁盘 I/O 操作，鼠标移动到这些点上时可直接提示相关信息。

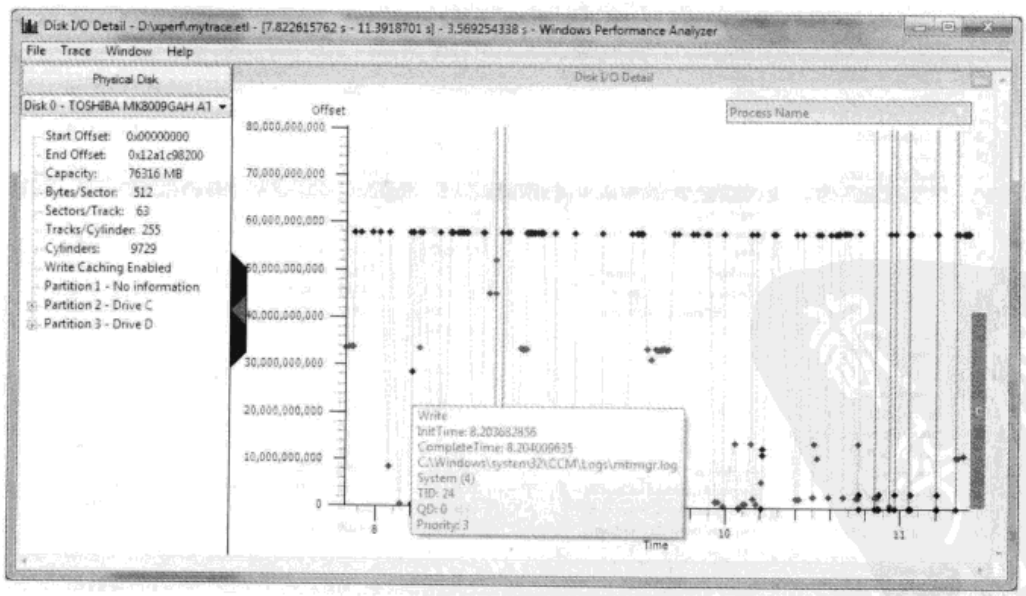


图 9.21 磁盘 I/O 细节视图

Xperf 还提供了摘要视图，允许用户从各个不同角度来聚集一个日志文件中的事件信

息。譬如，用户可以看到一段时间内每个进程和线程的 CPU 使用情况。磁盘 I/O 也可以按进程和线程来聚集。图 9.22 和图 9.23 显示了 CPU 使用率和磁盘 I/O 的摘要视图。注意，在摘要视图中有一条纵向的粗黄线，其左侧的域代表了当前的分组策略。通过调整它们的前后顺序，或者拖动列的标题栏，可以看到在各种不同组合形式下的聚集结果。

CPU Scheduling Aggregate Summary Table - D:\xperf\mytrace.etl - [0 s - 23.2596408 s] - 23.2596408 s - Windows Performance Analyzer

File Columns Trace Window Help

Line	Process	Thread ID	Thread Start Module	Thread Start Function	Cpu Usage (ms)	% Cpu Usage	% Relative Cpu
1	Idle (0)	0	ntkrnlpa.exe	KiIdleLoop	31,235.187 923	67.14	6
2	OUTLOOK.EXE (3848)				5,370.476 388	11.54	1
3	dwm.exe (3776)				1,954.266 273	4.20	
4	explorer.exe (1108)				1,526.487 980	3.28	
5	xperf.exe (5500)				1,492.387 374	3.21	
6	svchost.exe (676)				555.022 994	1.19	
7	System (4)				510.492 082	1.10	
8	taskmgr.exe (4052)				478.927 857	1.03	
9	csrss.exe (436)				475.620 732	1.02	
10	CcmExec.exe (2224)				333.145 690	0.72	
11	svchost.exe (1036)				323.574 162	0.70	
12	MsMpEng.exe (820)				313.531 241	0.67	
13	svchost.exe (992)				180.182 968	0.39	
14	xperf.exe (3988)				176.081 971	0.38	
15	iexplore.exe (4676)				136.860 739	0.29	
16		6,132	iertutil.dll	CloseScope::RegisterThread	134.492 014	0.29	
17		3,412	ntdll.dll	TppWorkerThread	2.037 669	0.00	
18		5,920	iexplore.exe	wWinMainCRTStartup	0.178 787	0.00	
19		2,248	GdiPlus.dll	DllRefCountSafeThreadTh...	0.152 269	0.00	
20	conhost.exe (4780)				121.046 139	0.26	
21	SearchIndexer.exe (3116)				111.609 743	0.24	
22	iexplore.exe (4336)				85.421 156	0.18	

Total CPU Usage: Non-Idle/DPC/ISR: 31.83% Idle time: 67.14% DPC/ISR time: 1.03%

图 9.22 CPU 使用率摘要视图

Disk Summary Table - D:\xperf\mytrace.etl - [0 s - 23.2596408 s] - 23.2596408 s - Windows Performance Analyzer

File Columns Trace Window Help

Line	Process	Path Name	Service Time (us)	Size	Pri	Read Counts	Read Service Time (us)
1	OUTLOOK.EXE (3848)		5,161,701.775	7,875,072		411	4,977,766.850
2	svchost.exe (676)		2,380,014.570	2,446,336		127	2,380,014.570
3	System (4)		1,328,882.824	24,273,920		16	245,195.825
4	MsMpEng.exe (820)		1,037,772.948	1,579,008		69	1,037,772.948
5	explorer.exe (1108)		197,976.183	368,128		13	197,976.183
6		C:\pagefile.sys	69,279.814	323,584		8	69,279.814
7		C:\Windows\AppPatch\pcamain....	40,170.780	4,608	0x 03	1	40,170.780
8		C:\$Mft	39,226.370	4,096	0x 03	1	39,226.370
9		C:\Windows\AppPatch\sysmain.s...	21,507.565	28,672	0x 03	1	21,507.565
10		C:\Windows\winsxs\x86_microsof...	16,158.476	4,096	0x 03	1	16,158.476
11		C:\ProgramData\Microsoft\Wind...	11,633.178	3,072	0x 03	1	11,633.178
12	xperf.exe (3988)		144,004.534	57,344		7	144,004.534
13	svchost.exe (1036)		138,302.996	306,176		0	0.000
14	xperf.exe (5500)		123,967.480	28,672		7	123,967.480
15	svchost.exe (992)		52,805.684	14,336		4	52,805.684
16	csrss.exe (436)	C:\pagefile.sys	49,850.981	8,192	0x 03	1	49,850.981
17	WakeUpAgt.exe (2184)		39,311.055	70,656		2	34,871.396
18	svchost.exe (936)		36,199.810	36,864		2	36,199.810
19	csrss.exe (372)	C:\pagefile.sys	23,426.325	4,096	0x 03	1	23,426.325
20	taskmgr.exe (4052)	C:\Windows\system32\wevtapi.dll	15,612.703	32,768	0x 03	1	15,612.703
21	services.exe (544)	C:\Windows\system32\services.exe	14,649.474	30,208	0x 03	1	14,649.474
22	CcmExec.exe (2224)	C:\Windows\system32\CCM\Serv...	266.042	4,096	0x 03	0	0.000

图 9.23 磁盘 I/O 摘要视图

此外，如果将内核日志记录器的性能采样标志位（PERF_PROFILE）打开，则日志文

件中将包含性能采样中断的事件。对于这样的日志文件，xperf 的性能采样摘要视图可以列出每个函数的时间权重，图 9.24 显示了 xperf 的性能采样摘要视图。

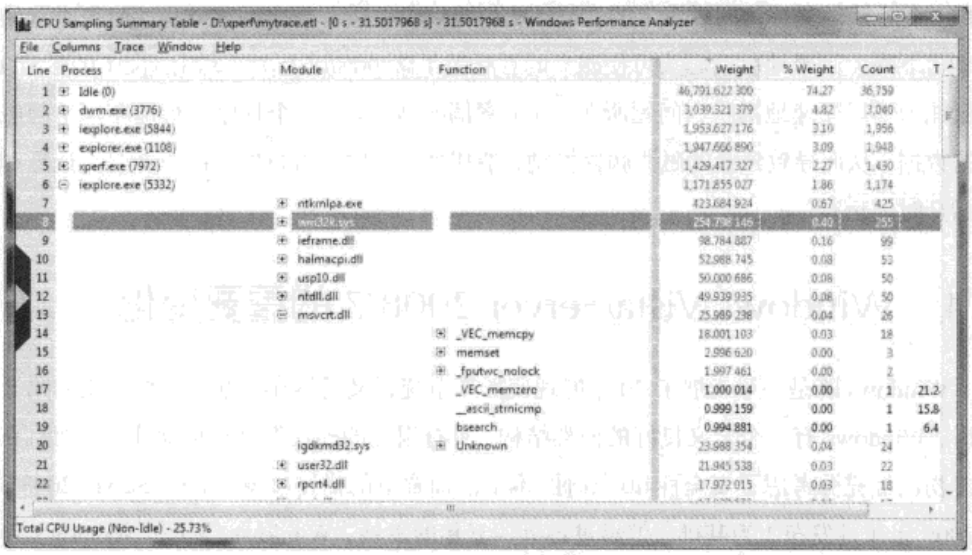


图 9.24 性能采样摘要视图

以上介绍了 xperf 的基本日志分析能力以及它的一些界面图示，实际上，它还提供了更为细致的分析功能，比如，线程等待关系的分析、锁竞争的自动检测等。由于.etl 日志文件提供了丰富的内核行为信息，因此，只要有足够的耐心和内核知识，绝大多数性能问题都可以通过分析.etl 日志文件来诊断和解决。Xperf 只是一个工具，它可以帮助使用者快速地找到问题的症结所在，但真正起决定作用的还是内核日志记录器。

Xperf 将系统的很多方面以直观的形式呈现出来，毫无疑问这对于分析性能问题非常有用。这一节最后举两个实际案例[WPA-WPT]来说明 xperf 用于诊断系统性能问题的能力。

第一个案例的症状是，当用户在 Windows 资源管理器中浏览一个 SD 卡，并打开其中一个图片文件时，声音播放有明显的抖动。使用 xperf 来检查这一段时间的.etl 日志文件时，发现 DPC 的 CPU 使用率非常高（通过查看“DPC CPU Usage”视图），差不多在 60%~90%之间；进一步检查该日志的性能采样摘要视图，可确认 SD 卡总线驱动程序 sdbus.sys 消耗了 25%左右的处理器资源。这个案例的结论是，SD 卡驱动程序消耗了太多的处理器资源在它的 DPC 例程中，导致系统性能受到影响。可行的解决方案是，驱动程序应该利用 DMA 而不是处理器 I/O（PIO，Processor I/O）来传输数据。

第二个案例的症状是，在 Outlook 2007 中标记 875 条消息为“已读”时，该程序出奇地慢。使用 xperf 来检查 CPU 使用率和磁盘使用率（Disk Utilization）视图，发现 CPU 使

用率其实并不很高，但磁盘使用率非常高，在性能异常的这段时间内几乎达到 100%。进一步查看磁盘 I/O 细节视图，发现几个进程（包括 Outlook.exe、SearchIndexer.exe 和 System 进程）在频繁地交替访问磁盘上的文件，从而导致磁盘读写头来回移动，这使得磁盘的性能显著下降（频繁地移动磁头可使磁盘的吞吐量下降 90%以上）。利用 xperf 的磁盘 I/O 分析能力可以快速地揭示出问题所在。这个案例的结论是，几个程序并行地密集访问磁盘上的数据，从而导致磁盘的磁头频繁摆动，造成性能下降。可行的解决方案是，让这些程序错开时间运行。

9.4 Windows Vista/Server 2008/7 的重要变化

Windows 既是一个发展了 20 年的现代操作系统，又仍然在不断演进中。正如第 2 章所讲，Windows 有一个定义良好的框架结构，具有很好的模块设计和扩展能力，而且在设计之初已经充分考虑了安全性和可靠性。本书前面章节的描述以 Windows Server 2003 SP1（2005 年 4 月发布）为基础，但经过最近四五年的发展，Windows 经历了 Windows Vista 和 Windows 7 两次重大的发布（以及 Windows Server 2008 的发布），其内核又有了新的发展。尽管结构模型上并无本质变化，但是在一些关键组件上却有显著的改进。这一节首先介绍 Microsoft 内部的一个 MinWin 工程项目，然后分别从进程和线程管理、内存管理以及 I/O 处理的角度来讨论自 Windows Vista 以后的重要改进。关于 Windows Vista 及以后系统的内核的更多介绍，读者可以参考有关资料[WIN-INTERNALS-5][VISTA-CHG][WIN08-CHG]。

9.4.1 MinWin 工程

MinWin 是一个工程项目，其目标是将 Windows（不仅仅内核，但内核是极为重要的部分）的模块梳理出来，以求理解和驾驭模块之间的依赖性，从而可以构建出一个最小的操作系统核，并在此基础上根据需要添加新的功能和模块。MinWin 项目本质上是代码重构，它试图将 Windows 系统中的模块以一种更加清晰和自然的方式组合在一起，因而当 Windows 继续发展时，Microsoft 的开发人员可以更加清楚地知道哪些模块依赖于其他哪些模块，以及它们对于特定用途的 Windows 系统是否必要。目前，MinWin 项目的结果已经融入到了 Windows Server 2008 和 Windows 7 系统中。

从历史来看，Windows 是一个发展了 20 年的操作系统，它的基本体系结构并没有大的变化，但是它的模块结构，特别是模块之间的依赖性已经非常复杂。在过去，为了在 Windows 中增加新的功能，开发人员通常只是在已有模块结构上用最直接的办法引入这些新功能，而不会顾及是否会引入额外的模块依赖性。因此，当为了构造一个小印迹（small

footprint）的 Windows 版本，比如 Server Core[MSDN-SC]、嵌入式 Windows 或 Windows PE（Preinstallation Environment）时，Microsoft 开发人员的做法是，从一个功能完全的 Windows 开始，将各种不需要的功能和服务从系统中去掉。然而，由于模块之间的依赖性不清楚，所以，这一过程并不轻松，而且往往无法彻底剥离掉不需要的服务或模块。

一个典型的例子是构建 Server Core[MSDN-SC]。Server Core 是在 Windows Server 2008 中引入的，其目的是构造一个运行最小数量服务的服务器操作系统，它包含仅仅为了向外部提供一个网络服务器而需要的那些服务。Server Core 支持常见的网络服务，但是并不提供图形界面能力，甚至连外壳程序 Windows 资源管理器也不提供；它仅提供命令行界面窗口或通过远程 MMC（Microsoft Management Console）连接到机器上进行服务配置。然而，Server Core 的服务仍然依赖于一些涉及图形功能的库，不过这些图形功能并没有真正被使用。

MinWin 是构建 Windows 小核的一个重要步骤，也是 Windows 操作系统在发展了多年以后重新回到体系结构层次来完善自身的一个转折点。MinWin 目前的最新结果是，包含 150 个文件，在磁盘上占 25 MB 大小，能够在 Virtual PC 中引导起来并提供 Web 服务（注意，没有命令行提示环境），引导起来后占内存大约 40 MB。本质上，MinWin 运行一个内核，提供基本的系统服务，以及运行 TCP/IP 协议栈[PDC-09]。

MinWin 不仅仅是对内核的代码重构，也是对 API 的重构。在历史上，Windows 基本 API 是从 Windows 3.1 和 Windows 95 时代继承过来的，包括 kernel32、user32 和 gdi32 中的函数，分别对应于系统核心功能、用户输入和交互功能，以及图形和显示功能。这每一个模块中的 API 函数都构成了一个大的 API 集合，这种大 API 集合的做法显然不利于构建精巧的系统，因为这些大 API 集合的模块本身给低层的功能模块强加了可能不必要的依赖性。

MinWin 采用了一种新的 API 分割模型，将 API 集（API set）的定义与实现分开。当应用程序调用一个 DLL 中的函数时，它调用的是虚拟 *DLL* 中的一个函数，而此虚拟 DLL 在一个实际的系统中将被映射到一个真正实现了此函数功能的逻辑 *DLL* 中。由于引入了虚拟 DLL，因而不同的系统可以以不同的方式将虚拟 DLL 组合到逻辑 DLL 中。MinWin 所要求的基本服务是在 KernelBase.dll 中提供的，其他的服务在一个称为 apisetschema.dll 的文件中定义了从虚拟 DLL 到逻辑 DLL 的映射表结构（schema）。图 9.25 显示了虚拟 DLL 与逻辑 DLL 的关系。

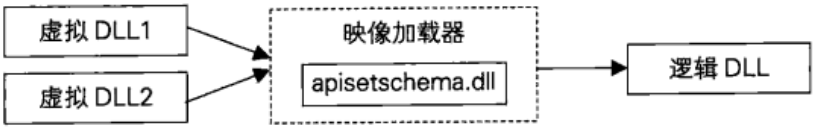


图 9.25 虚拟 DLL 被映射到逻辑 DLL

内核在系统引导时读入 apisetschema.dll 中的表结构，并且将它映射到每个进程中，因而，模块加载器（loader）在加载模块（包括 DLL 和 EXE 文件等）时可以直接引用此表结构。在 MinWin 的 API 模型中，API 集成了真正的接口，调用者在编译时面对的只是虚拟 DLL，而接口实现者（即逻辑 DLL）则要留到运行时或者软件发布时才确定下来，这为模块的灵活组合提供了方便。而且，MinWin 也引入了 API 层次结构，例如，kernel32.dll 所依赖的逻辑 DLL 都是 L1 层。图 9.26 显示了 Windows 7 中的 kernel32.dll 在 Dependency Walker 工具（depends.exe，随 Visual Stuido 一起提供）中的依赖关系。Kernel32.dll 除了依赖于 KernelBase.dll 和 ntdll.dll 以外，还依赖于一组名为 api-ms-win-<xxx>-<yyy>-l1-1-0.dll 的逻辑 DLL，这些逻辑 DLL 名称的最后三个数字分别代表了 API 集的层次数、主版本和次版本号。Kernel32.dll 实际上将它接收到的 API 调用转发给了 KernelBase.dll 和这些逻辑 DLL。新的 MinWin 应用程序可以直接建立在 KernelBase.dll 和 API 集上。

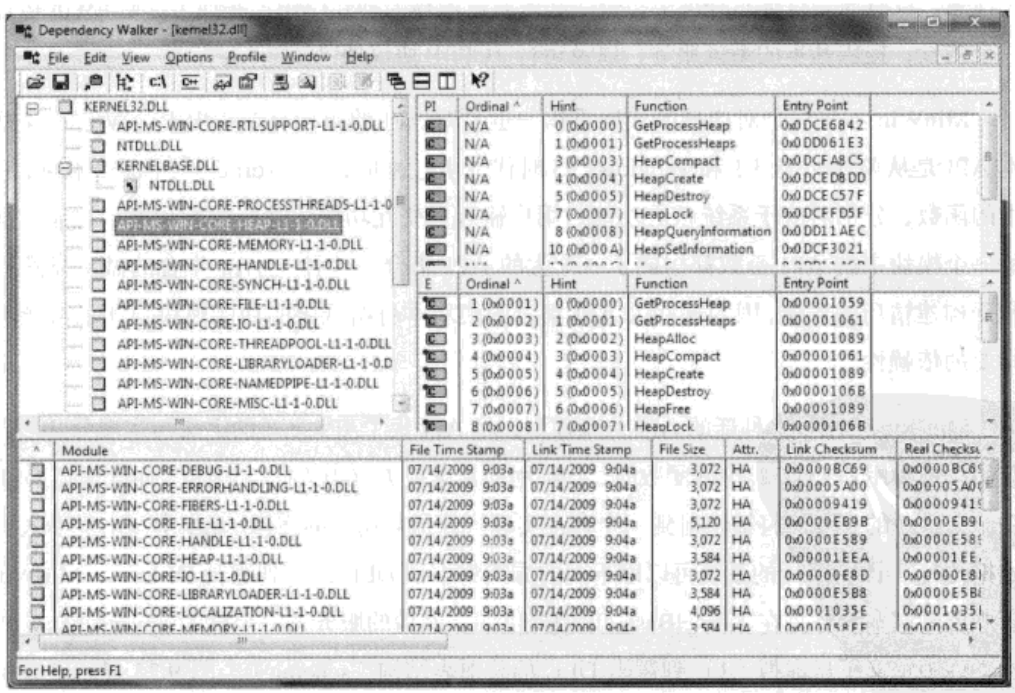


图 9.26 在 Dependency Walker 工具中显示 kernel32.dll 的模块依赖关系

此外，在 MinWin 中一个有关命令行控制台工具的变化是，cmd.exe 的窗口消息循环被从 Windows 子系统进程（csrss.exe）移到了一个称为 conhost.exe 的进程中。Conhost.exe

是 csrss.exe 的子进程，负责处理键盘输入，而真正的命令由 cmd.exe 来处理。这一变化的目的是，让 cmd.exe 尽可能地接近最必要的核心服务，而避免不必要的模块依赖性。

尽管 MinWin 的代码重构结果已经被应用到了 Windows 的最新版本中，但是，这还只是一个开始，相信未来版本的 Windows 会有一个更加整齐的模块结构，模块之间的依赖关系将更加清晰。尤其是，模块的拆分和组合将更为灵活，同样的 Windows 代码基可以配置出不同功能层次的 Windows 发布版本。

9.4.2 进程和线程管理

进程和线程的管理一直是 Windows 系统的亮点，尤其它的优先级调度算法已经充分考虑到了各种可能的需求，既能很好地支持桌面系统中应用软件的需求，也能适应服务器系统软件的要求。尽管如此，在 Windows Vista 及以后的系统中，进程和线程的管理又有了新的增强，本节接下来介绍三个重要的改进：线程时限计量更加精确、多媒体类别的调度服务，以及对用户模式调度的支持。

（一） 线程时限计量更加精确

在第 3 章介绍 Windows 的线程调度机制时，我们曾经介绍过，Windows 利用时钟中断来计量一个线程的时限，每个时钟中断间隔按照 CLOCK_QUANTUM_DECREMENT（3）个时限单位来计量。当时钟中断发生时，从当前线程的时限中减去 3，然后判断当前线程是否已经用完了分配给它的时限。其实，这种做法并不能保证线程的公平性，因为在一个时钟中断间隔期间，处理器的时间可能并不都用在当前线程上，甚至可能绝大部分时间被中断或其他线程用掉了。譬如，在一个多核 Windows XP 系统上，后台线程的时限值为 2 个时钟中断间隔，其值等于 6，对应的时间长度为 30 ms。而它们实际使用的处理器时间可能是比 0 多一点点，最多不超过 30 ms。因此，优先级相等的两个线程，它们获得的处理器时间可能并不相同，甚至差异很大。这种调度不公平性对于桌面操作系统也许并不那么严重，但毕竟造成了处理器资源分配的不公平。

自 Windows Vista 开始，Windows 的调度器利用现代处理器的指令周期计数寄存器（Intel x86 处理器的 rdtsc 指令）来精确地计量一个线程执行的指令周期数。它跟踪一个线程实际获得的指令周期，然后在时钟中断到来时，判断一个线程是否获得了足够时间的指令周期。按照这种计算方法，一个线程实际获得的时间长度为：至少等于它的时限值，譬如 30 ms，但最多不超过此时限值加上一个时钟中断间隔，譬如 45 ms。因此，相同优先级的线程获得的处理器时间的误差不超过一个时钟中断间隔。显然，线程调度的公平性比原来的计量方案好了很多。

而且，采用这种方案以后，原来调度算法中考虑每次等待醒来的线程减去 1 个时限单位的做法已经不再必要了，即 3.5.3 节中提到的 WAIT_QUANTUM_DECREMENT 宏的用法，因为每个线程的时限已经被精确计量了，所以，无论在执行过程中等待与否，它都无法偷取处理器资源。

（二）多媒体类别的调度服务

多媒体应用程序是桌面系统中一类常用的应用程序，例如常见的媒体播放器，它们既有大量的 I/O 操作，也要占用相当数量的处理器资源。当系统中有其他的程序抢夺资源时，它们很可能会出现媒体流播放抖动（glitch）的情形。为了提供更好的多媒体播放体验，Windows Vista 在原有 Windows 线程调度算法的基础上，为这一类应用提供了特殊的支持，称为多媒体类别的调度服务（MMCSS，Multimedia Class Scheduler Service）。

本质上，MMCSS 是用户模式的服务，它允许应用程序向 MMCSS 注册，因而系统的线程调度器在分配处理器资源时，可以为它们的线程提供所要求的资源保证。MMCSS 并没有改变 Windows 线程调度器的算法和结构，但是，它在处理线程的优先级提升时额外考虑了 MMCSS 指定的进程的要求，作为另一种线程优先级提升的情形。

应用程序通过注册表告诉 MMCSS 它们所要求的服务，线程调度器将按照它们指定的调度类别（Scheduling Category）来决定这些进程中的线程的优先级。这些指定了 MMCSS 调度类别的线程将在 80%的时间内获得所要求的线程优先级，剩下的 20%时间则降低到普通优先级，从而其他的线程也有机会得到处理器执行权。

这种优先级提升的做法并不能缓解在多媒体程序执行过程中 I/O 流量带来的影响（因为处理 I/O 流量需要 DPC 时间，而 DPC 优先于任何一个 PASSIVE_LEVEL 的线程），尤其是网络突发流量的干扰。因此，MMCSS 还通知网络栈，让它在多媒体流播放期间实施网络包节流（network package throttling）。这种设计的基本思想是，在多媒体流播放期间，允许一定程度（少量）的网络数据丢失。

（三）用户模式调度

Windows 7（64 位版本）引入了一种新的线程模型，称为用户模式调度（UMS，User-Mode Scheduling），允许应用程序调度它的工作线程。应用程序可以在用户模式下切换 UMS 线程，而无须涉及系统的线程调度器。当一个 UMS 线程在内核中阻塞时，应用程序仍然可以获得处理器的执行权，来执行其他的 UMS 线程。

为了使用 UMS，应用程序必须注册一个 UMS 调度器（UMS Scheduler），它负责创建、管理和删除 UMS 线程，并决定该哪个 UMS 线程运行。应用程序可以为每个参与调

度的处理器创建一个 UMS 调度器线程。每个 UMS 调度器线程都有一个相关联的完成链表（completion list），新创建的 UMS 线程以及从阻塞状态被唤醒的 UMS 线程都加入到指定的完成链表中。UMS 调度器以回调函数的方式进行工作，在下列情况下，系统会调用 UMS 调度器注册的回调例程：

- 当一个非 UMS 线程被转换为 UMS 调度器线程时。
- 当一个 UMS 线程调用 UMS 的 API 函数放弃执行权时。
- 当一个 UMS 线程在内核中阻塞时，比如它调用了系统服务或产生了页面错误。

在 UMS 调度器的回调例程中，它从完成链表中获得那些已经完成了内核部分的处理，且用户模式部分已经就绪的线程环境，将它们转移到调度器的就绪线程队列中，然后运行就绪队列中的线程。因此，与系统原生线程不同，UMS 线程不会自动执行，而必须由 UMS 调度器选中才能运行。UMS 线程一旦被选中并开始运行，就将一直运行到它显式地放弃执行权，或者被阻塞，或者终止。

UMS 线程不同于纤程（fiber），每个 UMS 线程都有自己的线程环境，而纤程共享使用一个线程的环境。因此，UMS 线程可以调用任意 API 函数，就好像一个原生的系统线程一样。

UMS 带来的好处是在多处理器或多核的系统上，应用程序在处理大量任务时的并发性可以显著提高。然而，UMS 要求软件开发人员必须实现一个 UMS 调度器来管理 UMS 工作线程，并决定它们何时被执行。因此，是否使用 UMS，以及它带来的性能好处是否值得开发一个 UMS 调度器，这需要针对具体问题来进行权衡。关于 UMS 的更详细信息，请参考 MSDN 中的相关资料[MSDN-UMS]。

9.4.3 内存管理

内存管理是操作系统的核心功能之一，Windows Vista 在内存管理上的改进涉及很多方面，例如大量地使用无锁（lock-free）同步技术和更细粒度的锁，从而提高内存管理器在多核和多处理器环境下的性能；更大的换页 I/O；支持 GPU 内存结构；更有效地使用硬件 TLB；使用闪存作为磁盘和内存之间的一个中间缓存（此技术称为 ReadyBoost），等等。本节我们介绍 3 项在 Windows Vista 中使用的较为基础的内存管理技术。

（一）动态内核地址空间

根据 4.2 节的介绍，我们知道，在 Intel x86 体系结构下，Windows 的系统地址空间在

内核初始化时已经被划分成多个固定的区域。尽管注册表中的一些设置项以及物理内存的数量可能会影响到最终的系统地址空间布局，但本质上，Windows 的系统地址空间布局结构相对比较固定，而且一旦初始化完成，各个区域的位置将不再变化。这有可能带来这样的问题：系统由于某个区域的虚拟空间被用完而失败，但实际上其他区域可能还很空闲。

在 32 位版本的 Windows Vista 系统中，内存管理器以动态方式来管理系统地址空间，它可以根据需要分配和回收系统地址空间中的地址范围。以换页内存池为例，当设备驱动程序大量地申请换页内存时，内存管理器会增加换页内存池的地址范围，而当设备驱动程序释放了这些换页内存时，内存管理器也相应地缩减换页内存池的地址范围。因此，Windows Vista 能够更加有效地使用系统地址空间中的虚拟地址范围，以适应更为广泛的系统负荷。

动态内核地址空间解决的是地址范围紧缺的问题，这对于 32 位系统有现实意义，但这一问题对于 64 位系统尚不足为虑。因此，64 位 Windows Vista 系统仍然使用静态的系统地址空间划分方案。

（二）内存优先级

在 4.5 节介绍 Windows 的物理内存管理时，我们曾经提到，Windows 有一个备用链表（standby list）记录了被系统回收的页面，也就是说，一个页面原来属于某个进程或系统工作集，但现在已经被从该工作集中移除，可任由系统作他用。实际上，Windows 内存管理器的备用链表支持函数已经用一个链表数组（MmStandbyPageListByPriority）来代替单个链表 MmStandbyPageListHead 了。因此，从这一层意义上讲，Windows Server 2003 的内存管理器已经支持内存优先级了。而且，用于管理物理页面的数据结构 MMPFN 也包含一个 3 位的优先级域。只不过，在 Windows Server 2003 系统中，所有页面的优先级都是 3，因此体现不出页面优先级调度的真正意义。

在 Windows Vista 中，页面的优先级管理真正得到了使用。一个页面的优先级通常由分配该页面的线程的优先级（ETHREAD::ThreadPagePriority）来决定，而线程的页面优先级继承自它所在的进程（EPROCESS::DefaultPagePriority）。进程的页面优先级的默认值为 5。页面优先级的有效性只有当不同进程或不同用途的页面被赋予不同的优先级时，才能够真正体现出来。如果要让应用程序或用户来指定内存的优先级，则显然这一机制的可用性将大打折扣。Windows Vista 的 SuperFetch 机制提供了一种理想的页面优先级分配方法，它可以在无须人工干预的情况下“学习”一个系统的历史，从而逐渐积累起关于页面用途的知识。

（三）SuperFetch

SuperFetch 是 Windows Vista 引入的一种物理内存优化管理方案。它搜集关于页面的用途以及会话、进程等信息，记录下用户行为对文件页面访问的历史信息，并根据这些信息来评价每个页面，重新分配页面的优先级，从而建立起真正支持优先级的备用链表。

SuperFetch 是一种主动式内存管理机制（proactive memory management），而 Windows Vista 以前的内存管理方案并非主动式的。例如，在支持优先级的备用链表管理算法中，内存优先级只反映了一个进程过去的行为，而不代表任何将来的内存需求；另外，它管理的对象仅限于一个进程所属的页面，一旦一个页面离开了进程（比如，被换出到磁盘文件中），则只有当将来该页面被真正使用时才再次带入内存。体现这种内存管理缺陷的一个典型情景是“饭后综合症”，即当用户离开一个系统去吃饭时，一个消耗内存的程序（比如病毒扫描或桌面搜索引擎建立索引）运行起来了，从而导致用户正在运行的程序被换出内存，当用户吃完饭回来时，需要经历一段“慢反应”期，以便把换出的代码和数据陆续换回到物理内存中。其实，任何时候当用户运行了一个消耗大量内存的应用程序再回到其他程序中时这种情形就可能发生。

SuperFetch 可以有效地解决这样的性能问题。它跟踪并记录页面的使用历史，不仅包括当前物理页面的使用信息，也包括曾经出现在内存中的数据和代码。另一方面，它也能够指示内存管理器将磁盘文件或页面文件中的数据和代码预加载（preload）到备用链表中，并分配优先级。当系统中内存变得充裕时，比如当应用程序退出或释放了大量内存时，SuperFetch 可以指示内存管理器将最近由于内存紧张而换出的代码和数据再取回来。当然，这一任务是在后台以低优先级、低速率逐渐完成的，因而不会影响前台交互程序。

SuperFetch 支持场景（scenario）的概念，每个场景规定了应该跟踪哪些页面，使用什么规则来分配页面优先级，以及采取哪些行动来提高系统的性能。不同的场景有不同的目标，但本质上都是为了尽可能地减少硬页面错误（hard page fault）。SuperFetch 预定义的场景包括休眠场景、待机（standby）场景和快速用户切换（FUS，fast user switching）场景。

9.4.4 I/O 处理的改进

在 Windows Vista 及以后的系统中，I/O 系统的基本结构没有变化，但是一些局部设计和实现有了改进。在 I/O 请求的处理方面，I/O 完成端口的处理更加高效，对 I/O 取消的支持也更为全面，并且引入了 I/O 优先处理机制。另外，NTFS 文件系统开始支持基于文件的符号链接。

（一）I/O 完成端口的处理

I/O 完成端口主要被服务器端软件用于提高系统的处理能力，在 Windows Vista 以前的系统中，对于一个关联了 I/O 完成端口的文件对象，当它的 I/O 请求被完成时，发起 I/O 请求的线程总是要介入该 I/O 请求的完成处理过程，所以，这意味着必须要切换到该线程，甚至可能会伴随有进程切换操作。然后，在完成例程（IopCompleteRequest 函数）中，它将一个完成包插入到完成端口的队列中，以便由完成端口线程池中的线程来进一步处理此 I/O 请求。

从 Windows Vista 开始，I/O 请求的完成处理不必非得由发起 I/O 请求的线程来执行，而可以由一个正在等待完成端口的线程直接接手完成处理工作。这种 I/O 被称为未知线程的 I/O（thread agnostic I/O）。这样做可避免不必要的线程调度开销，甚至地址空间切换的开销。

（二）全面支持 I/O 取消

在 6.6.3 节，我们讨论了 I/O 管理器对于 I/O 取消的支持，但是，在以前版本的 Windows 系统中，在有些情况下，用户根本不能取消 I/O。例如，当用户在资源管理器的窗口中试图打开一个不可用的共享文件夹时，用户只能等到网络请求超时才能继续操作该窗口；或者，当用户在打开文件对话框中输入一个不可用的网络路径时，在网络请求超时以前该对话框将停住，而不再响应任何用户输入。从 Windows Vista 开始，资源管理器和标准的打开文件对话框都改进了图形界面，以允许用户显式地取消一个正在进行中的 I/O。应用程序可以通过调用 API 函数 CancelIo、CancelIoEx 或 CancelSynchronousIo 来取消一个异步方式或同步方式的 I/O 请求。根据 6.6.3 节中的讨论，我们知道，这实际上加速了一个 I/O 请求的完成处理（其 I/O 返回状态码为 STATUS_CANCELLED）。

另一个典型的取消 I/O 的情形是当一个线程被终止的时候。如果一个线程发出的 I/O 请求未能被及时地取消，那么，该线程也无法完成其终止过程。注意，只有当驱动程序为一个 I/O 请求设置了取消例程时，该 I/O 请求才可被取消。

（三）I/O 优先处理

Windows 在处理器资源的分配上，采用了优先级模型，这使得一些紧急的任务可以优先得到处理器的执行权。然而，在 Windows Vista 以前的系统中，I/O 的处理不分优先级。因此，一个显而易见的问题是，后台任务的 I/O，比如病毒扫描、桌面搜索引擎的索引创建等，会不可避免地影响到前台任务的执行。Windows Vista 引入了两种 I/O 优先处理机制来帮助前台的 I/O 操作获得优先处理权。

第一种机制是，针对单独的 I/O 操作定义其优先级。Windows Vista 的 I/O 系统定义了 5 种 I/O 优先级，分别为“紧急（Critical）”、“高（High）”、“普通（Normal）”、“低（Low）”、“很低（Very Low）”。其中 4 种优先级已经被一些组件或应用程序所使用：内存管理器使用“紧急”级别的 I/O，以便在内存紧张的情况下将脏页面写出到磁盘上，从而尽快腾出更多的物理内存；默认的 I/O 优先级为“普通”级别，所以，普通应用程序发出的 I/O 处于中间优先级；Windows 的任务调度器（Task Scheduler）将它的任务的 I/O 默认设置为“低”级别；专门为 Windows Vista 及以后的系统而设计的后台处理任务的 I/O 使用“很低”级别，比如 Windows Defender 和桌面搜索引擎使用这一最低级别。

Windows 定义的 I/O 操作优先级在实际处理的时候采用了两种策略：层次优先策略和空闲优先策略。从“紧急”到“低”这 4 种优先级使用层次优先处理策略，即高优先级的 I/O 总是先于低优先级的 I/O 被处理；“很低”优先级采用空闲处理策略，即系统使用一个定时器，确保每个时间单位内至少有一个“很低”级别的 I/O 操作被处理，这样可避免后台任务发生饥饿的情形。

由于 I/O 请求最终是由驱动程序来处理的，所以，I/O 优先级需要驱动程序的支持。对于存储设备，层次优先策略由端口驱动程序来实现，而空闲优先策略由类驱动程序来实现。因此，Microsoft 可以以相对较小的工程代价，支持尽可能多和广泛的存储设备。

第二种优先处理机制称为带宽预留（bandwidth reservation）。像多媒体应用程序可以向 I/O 系统要求按指定的速率来读取外部设备的数据。如果设备确实有这个能力，那么 I/O 系统就会向应用程序提供带宽保证。如同层次优先策略一样，带宽预留机制也是在端口驱动程序中实现的。

（四）基于文件的符号链接

7.4.5 节曾经提到，NTFS 文件系统支持硬链接（hard link）和符号链接（symbolic link，也称为交接或 junction）。这里的硬链接是指不同的路径指向同一个文件，注意，硬链接仅支持文件，不支持目录。应用程序可通过 CreateHardLink 函数来创建一个硬链接。符号链接是指将一个目录重定向到另一个本地目录上。在 Windows Vista 以前，符号链接只支持目录形式。Windows Vista 引入了针对文件的符号链接，或称为软链接（soft link）。

无论是针对文件的符号链接，还是针对目录的符号链接，NTFS 文件系统驱动程序都是利用 I/O 管理器的重解析点技术来实现的，关于重解析点的简要介绍，参见 7.3.2 节。在 Windows Vista/Server 2008 及以后的系统上，应用程序可以通过 CreateSymbolicLink 函数来创建符号链接（文件或目录均可）。

9.5 本章总结

Windows 是一个功能全面的现代操作系统，不仅具备桌面操作系统所要求的各种功能要素，同时也是一个功能强大的 PC 服务器操作系统。本书前面的章节介绍了 Windows 内核中的核心机制，而在本章中，我们从一个完整的操作系统的角度出发，来讨论一些更加高级的话题，包括 Windows 的网络结构、Windows 子系统和内核日志及性能分析。网络结构和 Windows 子系统虽然不在 Windows 内核模块 ntoskrnl.exe 中，但对于 Windows 的运行本质上是不可或缺的，并且它们在 Windows 的发展历程中确实起到了至关重要的作用。内核日志对于分析和诊断 Windows 系统中存在的问题非常有帮助，它超越了调试器能够做到的问题诊断能力，即使对一些微妙的、偶尔发生的错误，通过分析内核日志，也有可能揭示出其根源所在。尤其在最近的 Windows 版本中，内核日志记录器提供了更加全面的日志记录能力，从而有助于解决一些积累已久的性能问题。Windows 最新版本不断提升的性能已经充分说明了这一点。

Windows 也是一个仍然在不断发展的操作系统，因此，本章还讨论了自 Windows Server 2003 SP1 以后的一些重要变化。MinWin 是 Microsoft 的一个内部工程项目，其目的是梳理 Windows 系统的组织结构，使系统中的模块以更加清晰和自然的方式协同工作。目前 MinWin 项目的结果已经融入到了 Windows Server 2008 和 Windows 7 产品中。对于 Windows 系统中的每一个组件，自 Windows Vista 以后几乎都有相应的改进，本章最后讨论了这些系统在进程和线程管理、内存管理和 I/O 处理方面的一些重要改进。

附录 A

建立 WRK 工作环境

正如 2.3 节所介绍，WRK 不仅提供了 Windows 内核模块的绝大部分代码，而且还提供了编译工具以及 Windows Server 2003 SP1 的虚拟机映像。WRK 的 readme.txt 文件已经清楚地列出了编译 WRK 的步骤，尽管如此，本附录还是将从实际操作的角度来全面地介绍编译和调试 WRK 的过程。

A.1 编译 WRK

WRK 的发布包提供了相应的编译工具，因此，用户无须额外的编译器就可以将 WRK 源代码编译成内核可执行模块。虽然 WRK 的内核针对的是 Windows Server 2003 SP1（x86）或者 Windows XP Professional（AMD64）系统环境，但是编译 WRK 的过程并不限于在这两种系统版本上进行，而可以在任何一个可运行 Microsoft 编译工具 nmake 的系统环境中进行（实际上，Windows XP/Server 2003 以后的任何 Windows 操作系统都可以）。有两种方法可以编译 WRK 内核模块：命令行或 Visual Studio 集成环境，下面分别介绍。

第一种方法是在命令行环境中编译 WRK。首先打开一个控制台窗口，即启动 cmd.exe 程序。注意，编译 WRK 无须管理员特权，所以，在 Windows Vista 或 Windows 7 中只需按常规做法启动控制台程序即可。在控制台窗口中，首先将当前目录切换到 WRK 源代码主目录下，然后设置 PATH 环境变量，使它包含 WRK 的 tools 目录中的 x86 子目录或 amd64 子目录。然后切换到 WRK 的 base\ntos 目录下，并输入“nmake –nologo x86=”或“nmake –nologo amd64=”命令。图 A.1 演示了这一过程，这里 WRK 源代码主目录为“D:\WRK\WRK-v1.2”（假定要编译的是 x86 版本的 WRK 内核）。

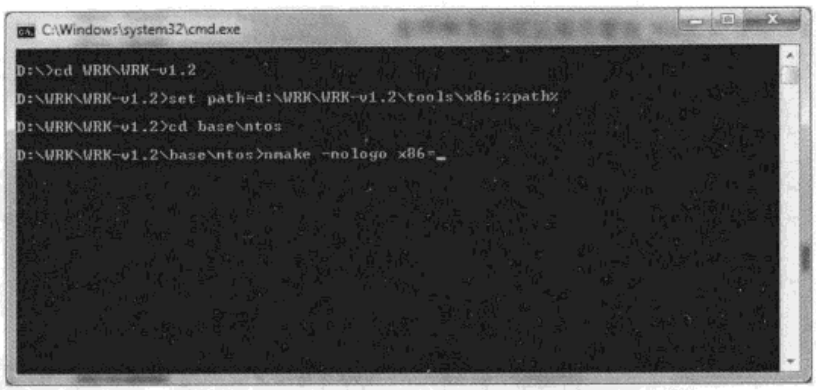


图 A.1 在命令行下编译 WRK 的命令序列

Nmake 的编译过程大约需要几分钟，具体时间长短取决于系统性能配置。编译完成以后，可得到图 A.2 所示提示。最终生成的内核模块文件名为 wrkx86.exe，位于 WRK 的 base\ntos\BUILD\EXE 目录下。

图 A.2 编译 WRK 完成提示

在编译 WRK 时，要注意的两点是：第一，编译工具必须包含在可执行程序的搜索路径中，即 PATH 环境变量要包含 WRK 的 tools\x86 或 tools\amd64 目录，而且，由于在编译过程中要用到 xcopy 工具，所以，Windows 的系统目录 Windows\System32 也要出现在 PATH 环境变量中；第二，须在 ntos 目录下输入 nmake 命令。

第二种方法是使用 Visual Studio 来编译 WRK。在 WRK 的主目录下包含 WRK.sln 和 WRK.vcproj 文件。用户可以在 Visual Studio 中打开 WRK.sln 文件，然后在工具条的“Solution Configurations”列表框中选择 x86 或 amd64。接下来，选择“Build Solution”或按下 F7 键以后 Visual Studio 开始编译 WRK，待编译结束后可看到图 A.3 所示界面。同样地，用 Visual Studio 编译后得到的内核模块为 wrkx86.exe，位于 base\ntos\BUILD\EXE 目录中。

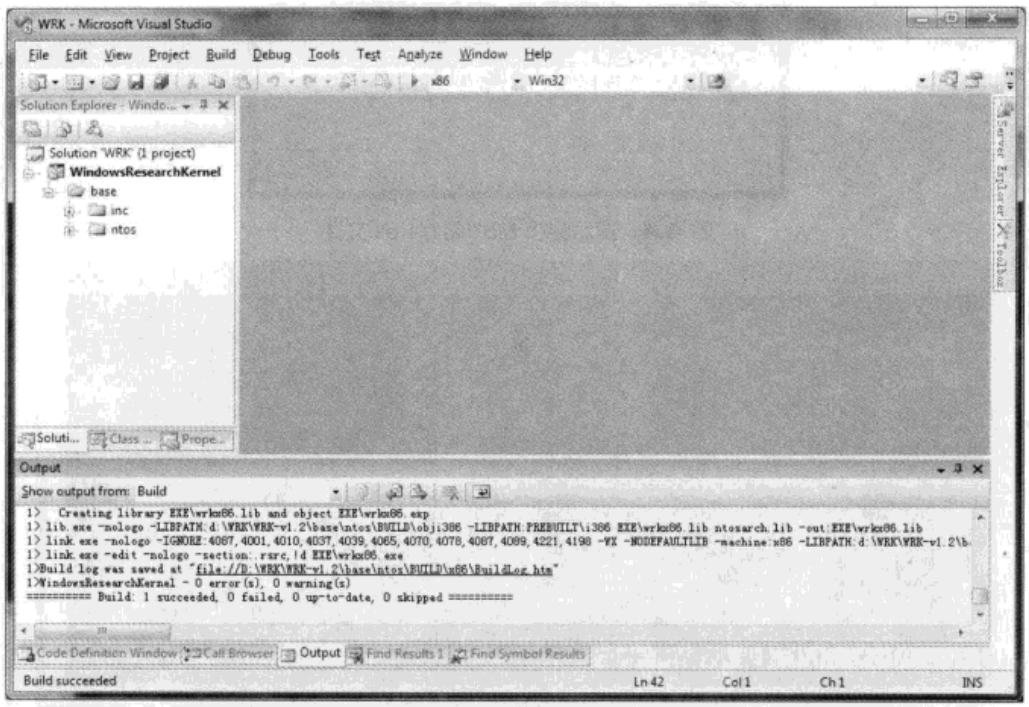


图 A.3 在 Visual Studio 中编译 WRK

使用 Visual Studio 来编译 WRK，本质上调用了与命令行类似的 nmake 命令，参见与 WRK.sln 位于同一目录下的 build.bat，如下：

```
@ECHO OFF
set path="%2\tools\%1";%path%
cd base\ntos\
nmake -nologo %1=
cd ..\..
```

用 Visual Studio 编译 WindowsResearchKernel 工程时，按照该工程中的设置，其编译命令行为 build.bat。所以，就编译过程而言，以上两种方法实际上是完全等价的。但是，在 Visual Studio 环境中，用户可以方便地浏览或修改代码，甚至在不同文件的代码之间方便地切换和跳转。而在命令行下，用户不得不依赖于其他一些文本搜索工具（如 grep 或 findstr）或编辑器来阅读和维护 WRK 代码。两者孰优孰劣，完全取决于用户个人的喜好和习惯。

最后，顺便提一下，笔者在从两个不同途径拿到的 WRK 发布包中，均发现在 WRK-v1.2\tools\x86 目录中缺少 MSVCR71.dll 文件，从而导致编译过程不能如预期般进行，具体表现如下所述。如果您已经能够编译 WRK，则请跳过本节以下内容。

在命令行情形下，当用户在 ntos 目录下输入“nmake x86=”命令时，会弹出图 A.4 所示对话框，当用户在该对话框中单击 OK 按钮以后，命令行如图 A.5 所示。

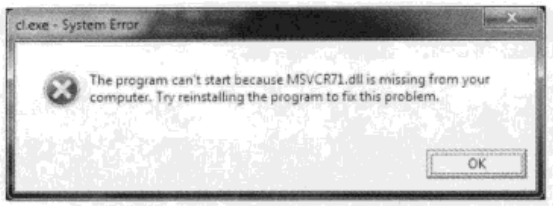


图 A.4 提示缺少 MSVCR71.dll 文件

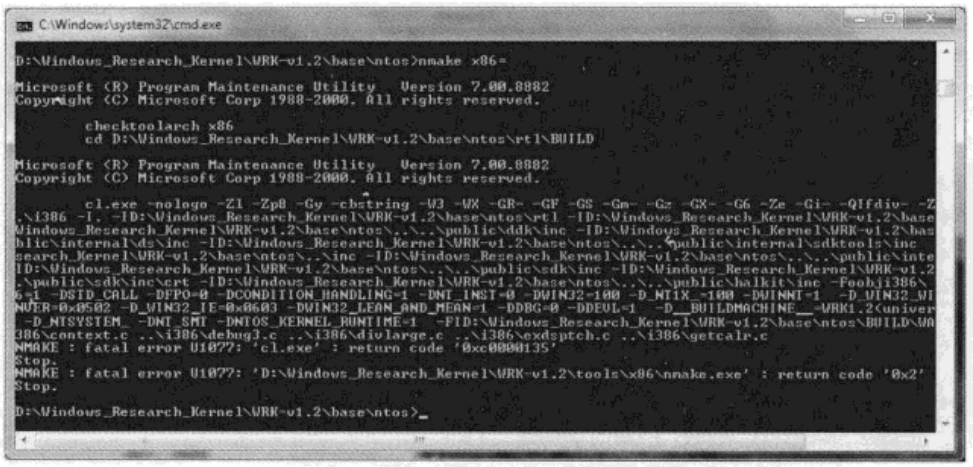


图 A.5 由于缺少 MSVCR71.dll 文件而未能成功编译 WRK

在 Visual Studio 中，若缺少 MSVCR71.dll 文件，那么，当选择“Build Solution”命令时，在 Output 窗口中将出现图 A.6 所示提示。

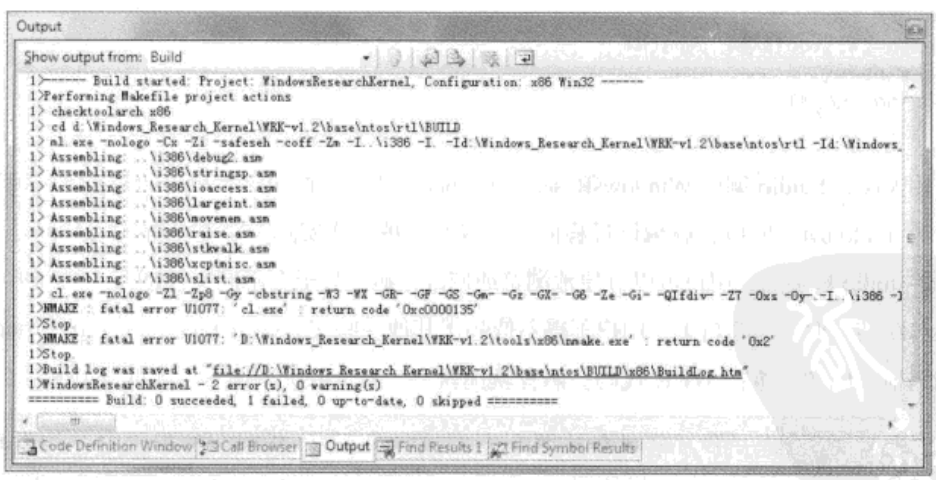


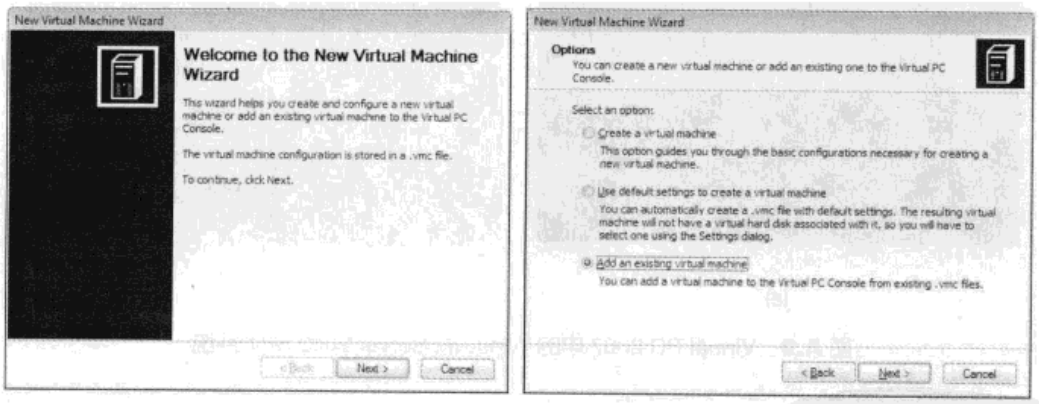
图 A.6 由于缺少 MSVCR71.dll 文件在 Visual Studio 中未能成功编译 WRK

显然，在 Visual Studio 中，由于错误提示对话框被抑制了，所以，仅仅从 NMAKE 的 U1077 错误或 cl.exe 的错误返回码 0xc0000135，并不能诊断出缺少 MSVCR71.dll 这一根源，相反地，在命令行环境下编译 WRK，则马上就能发现原因所在。

A.2 启动 WRK

按照上一节介绍的方法编译得到 WRK 的内核可执行模块以后，我们可以在一台安装了 Windows Server 2003 SP1（x86）或 Windows XP Professional（AMD64）的机器上启动 WRK。目标系统既可以在一台实际的机器上，也可以在虚拟机环境中。由于 WRK 的发行光盘本身已经随带了 Virtual PC，以及一块安装了 Windows Server 2003 SP1 的虚拟硬盘，因此，启动 WRK 实际上变成了增加一个引导选项以及替换内核映像和 HAL 映像的过程。本节我们以 x86 版本 WRK 为例来介绍如何在虚拟机环境中启动 WRK。

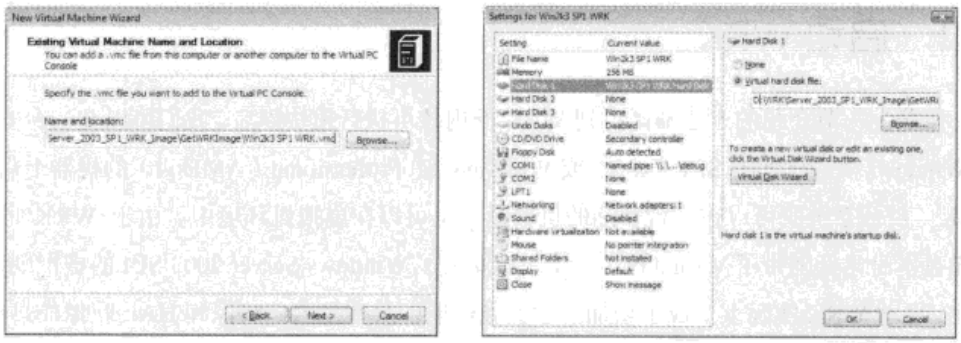
首先在一台 Windows 机器上安装 Virtual PC 2007，WRK 的发行光盘中包含了 Virtual PC 2007 安装包，读者也可以从 Microsoft 的网站下载。安装完成以后，启动 Virtual PC 2007，导入 WRK 的 Windows Server 2003 SP1 映像。假设该映像位于 D:\WRK\Server_2003_SP1_WRK_Image\GetWRKImage 目录下，其中包括“Win2k3 SP1 WRK.vmc”和“Win2k3 SP1 WRK Hard Disk.vhd”文件。启动 Virtual PC 2007，然后选择“New...”。当第一次启动 Virtual PC 2007 时，它会自动进入“New Virtual Machine Wizard”，如图 A.7(a)所示。选择“Add an existing virtual machine”选项，如图 A.7(b)所示。



(a) (b)

图 A.7 在 Virtual PC 2007 中新建一个虚拟机

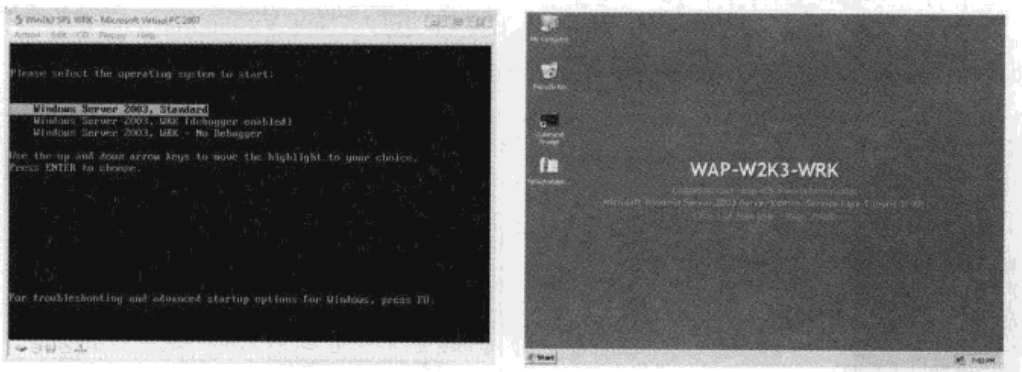
单击 Next 按钮，然后选择“Win2k3 SP1 WRK.vmc”文件，如图 A.8(a)所示。在接下来的页面中单击 Finish 按钮，完成导入过程。Virtual PC 2007 将显示此映像系统的基本设置，如图 A.8(b)所示。我们可以看到，此虚拟机系统内存大小的默认设置为 256 MB，并且其 COM1 端口已被设定为宿主机器上的命名管道“\\.\pipe\debug”。用户在此“Settings for Win2k3 SP1 WRK”界面中可以改变虚拟机系统的配置信息，包括内存大小、网络等设备，也可以创建或导入额外的虚拟硬盘。



(a) (b)

图 A.8 选择和配置 WRK 中提供的虚拟机映像

单击 OK 按钮完成导入过程。现在可以启动 Windows Server 2003 SP1 了。在 Virtual PC 2007 中双击“Win2k3 SP1 WRK”，稍等片刻，虚拟机中的 Windows Server 2003 SP1 开始启动，如图 A.9(a)所示。第一次启动我们选择默认的“Windows Server 2003, Standard”引导选项。启动进入 Windows Server 2003 SP1 系统以后的界面如图 A.9(b)所示。



(a) (b)

图 A.9 Virtual PC 2007 中的 Windows Server 2003 SP1 系统

进入虚拟机系统中后，我们检查一下 C:\boot.ini 文件（在控制台窗口中输入“notepad c:\boot.ini”即可）的内容，如下所列：

```
[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="Windows Server 2003, Standard"
    /noexecute=optout /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, WRK"
    /noexecute=optout /fastdetect /debug /debugport=com1 /kernel=wrkx86.exe
    /hal=halacpim.dll
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, WRK - No Debugger"
    /noexecute=optout /fastdetect /kernel=wrkx86.exe /hal=halacpim.dll
```

可以看出，系统的引导菜单包含三项：第一项是标准引导选项，使用 Windows Server

2003 SP1 本身自带的内核和 HAL 模块；第二项是调试模式的 WRK 引导选项，指定内核使用 wrkx86.exe，HAL 使用 halacpim.dll 文件；第三项是非调试模式的 WRK 引导选项，同样使用 wrkx86.exe 和 halacpim.dll 作为系统的内核和 HAL 模块。因此，当用户在引导菜单中选择第二或第三项时，Windows 将会加载指定的 wrkx86.exe 和 halacpim.dll 作为它的内核和 HAL 模块。这两个文件位于虚拟机系统的 C:\Windows\System32 目录下。WRK 的虚拟机映像中已经包含了一份编译出来的 wrkx86.exe，所以，即使用户不编译 WRK，也可以让虚拟机系统启动 WRK 内核。

要让虚拟机系统使用我们自己编译的 WRK 内核，我们必须把编译得到的 wrkx86.exe 模块拷贝到虚拟机系统的 C:\Windows\System32 目录中。由于 WRK 中的虚拟机系统已经安装了 Virtual Machine Additions，所以，我们可以直接将要拷贝的文件拖放到虚拟机内部的文件夹中。最简单的做法是，在虚拟机内部，将目标文件夹（比如 System32）打开并最大化，然后在宿主系统中，将编译得到的 wrkx86.exe 文件拖放到 Virtual PC 虚拟机窗口中。

最后，关闭虚拟机系统（如正常关机一样在 Start 菜单中选择“Shut Down”命令）。下次启动虚拟机时，若选择第二或第三引导选项，则虚拟机中的 Windows Server 2003 SP1 系统将使用我们自己编译的 WRK 内核。如果我们在编译 WRK 以前修改了源代码，那么这些修改将在新启动的系统中起作用。

WRK 已经可以启动，接下来我们可以通过调试器来观察内核中的代码是如何运行的了，在进入下一节以前，关于启动 WRK 还有以下三点说明：

(1) 在以上的描述中，由于使用了 WRK 提供的虚拟机映像，所以有些关键步骤已经被简化了。如果用户自己安装 Windows Server 2003 SP1 系统，则需要先创建一个空的虚拟硬盘，然后安装 Windows Server 2003 SP1 系统。待安装完成以后，还需要将 wrkx86.exe 和 halacpim.dll 文件拷贝到新安装的虚拟机系统中，然后修改 boot.ini 文件如前所述。注意，WRK 要求使用多处理器版本的 HAL 模块，在虚拟机环境下，它必须使用 halacpim.dll。如果用户安装后的系统中没有此文件，请从 WRK 主目录的 WRK-v1.2\WS03SP1HALS\x86\halacpim 目录下，将 halacpim.dll 拷贝到虚拟机系统的 C:\Windows\System32 目录中。一旦修改了 boot.ini 文件，则接下来的步骤与前面的描述相同。唯一值得一提的是，如果用户不能用拖放的方式来拷贝文件，那么，要么安装 Virtual Machine Additions，要么通过网络共享方式在虚拟机系统和宿主系统之间拷贝文件。

(2) Windows 7 自带了 Virtual PC，所以，在 Windows 7 中使用 WRK 更加方便。为了往虚拟机系统中拷贝文件，用户可以使用 Virtual PC 中新的特性“Integration Features”，从而可以在虚拟机系统和宿主系统之间共享各种资源，包括声卡、打印机、剪贴板、磁盘等。

(3) 在实体机器中启动 WRK 的步骤并不复杂。首先安装一个 Windows Server 2003 SP1 系统，然后通过某种途径将 wrkx86.exe 和相应的多处理器 HAL 模块拷贝到目标系统中。注意，实体机系统的 HAL 有三种可能：如果是支持 ACPI 的 PIC（中断控制器）机器，则使用 halacpim.dll；如果是支持 ACPI 的 APIC（高级可编程中断控制器）机器，则使用 halmacpi.dll；如果是非 PIC 和 APIC 的机器，则使用 halmps.dll。WRK 的 WRK-v1.2\WS03SP1HALS\x86\目录下包含了这三个 HAL 模块。

A.3 调试 WRK

为了调试 WRK，首先要安装一个调试器，WRK 光盘中包含了 windbg 软件，其版本为 6.6.7.5。读者也可以从 Microsoft 的网站直接下载，到本书写作时最新的版本为 6.12.2.633（http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx）。用 windbg 来调试 Windows 内核有两种典型配置：一种是在被调试的系统内部启动 windbg，然后使用“Local Kernel Debugging”的能力来调试本地系统的内核；另一种是在另一个系统中通过 COM、USB 或 1394 连接线调试一个 Windows 系统。这两种调试配置如图 A.10(a)和(b)所示。第一种方法实际上并不能完全调试一个系统，很多调试命令在这种配置下无法使用，因为一旦一个命令导致系统停止（halt），系统将无法自己恢复运行，但有关内存输入输出的命令都可以使用。所以，如果只是简单地查看系统内存，则使用 Local Kernel Debugging 模式不失为一种简单的办法。第二种方法要求配置两台机器，并且进行物理连线以便 windbg 控制另一个被调试的 Windows 系统，这种做法虽然不方便搭建，但是 windbg 的调试能力可以全面发挥，用户不但可以检查被调试系统的内存，甚至它的寄存器状态，还可以单步执行每一条指令，以及其他更高级的调试功能。

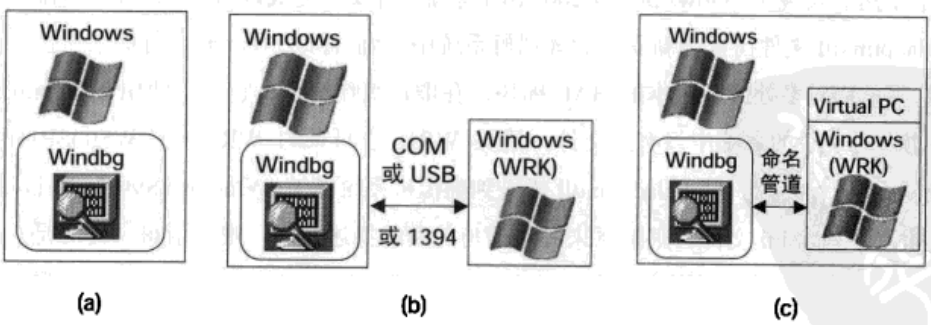


图 A.10 用 windbg 来调试 Windows 内核的三种配置

然而，利用虚拟机技术我们可以将这两种配置的优势结合起来。如图 A.10(c)所示，windbg 通过命名管道来控制一个运行于 Virtual PC 中的 Windows 系统。用这种方法，无须额外的物理机器，就可以调试一个 Windows 系统的运行过程。这种配置方案的调试能力近似于（并不

等同于）前面提到的第二种方案，但是操作起来要方便很多。除非有一些依赖于特定硬件特性（如多处理器、中断等）的功能需要调试，否则用这种配置方案来调试 Windows 内核或设备驱动程序往往已经足够了。本节接下来讨论用这种配置方案来调试 WRK。

假设我们已经能够在 Virtual PC 中启动 WRK 了。上一节提到的引导选项“Windows Server 2003, WRK”指定了在虚拟机中以调试模式引导 WRK 内核，并且还指定了引导参数“/debug /debugport=com1”，这说明一旦选择了此引导选项，内核模式的调试功能将被打开，并且允许远程内核调试器连接到 COM1 端口上。这里“/debugport”除了指定 COM1 或 COM2 这样的 COM 端口外，也可以指定 IEEE 1394 端口（如/debugport=1394）。然而，宿主系统中的 windbg 如何连接到虚拟机中的 COM1 端口呢？答案在于，虚拟机的参数设置中 COM1 端口是如何定义的。图 A.11 显示了 WRK 虚拟机的 COM1 端口定义。

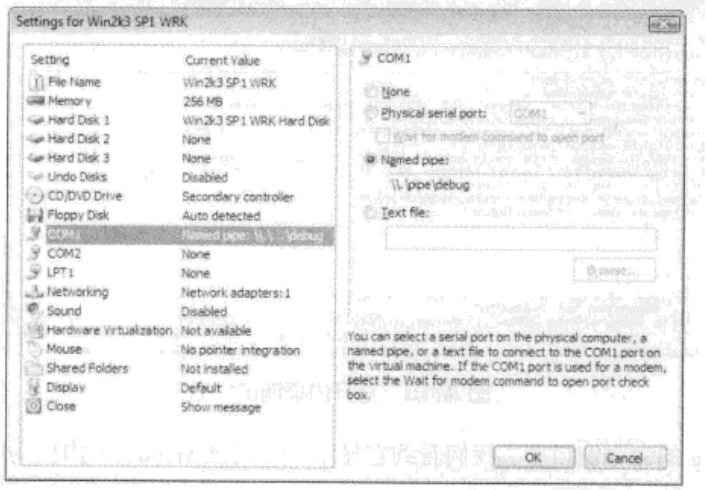


图 A.11 WRK 虚拟机的 COM1 端口定义

虚拟机的 COM1 端口可以被定义为空（None）、物理串口、命名管道或者一个文本文件。WRK 虚拟机的 COM1 端口已被定义为宿主系统中的命名管道“\\.\pipe\debug”。因此，宿主系统中的 windbg 可以通过命名管道连接到该 WRK 虚拟机的内核调试组件。

在 windbg 中，在 File 菜单中选择“Kernel Debug...”命令，然后在 COM 标签页中选中“Pipe”并指定虚拟机 COM1 端口的命名管道名称，如图 A.12 所示。

为了建立起基于命名管道的调试会话，虚拟机和 windbg 的启动顺序须要特别注意。命名管道对象是由 Virtual PC 创建的，当用户在 Virtual PC 中启动 WRK 虚拟机时，Virtual PC 创建“\\.\pipe\debug”命名管道对象，并且一直等待有内核调试器连接到该对象上。Windbg 是发起连接一方，它尝试连接到指定名称的命名管道对象上。所以，windbg 只有等到 Virtual PC 启动 WRK 虚拟机之后才能打开调试会话。关于命名管道的通信模型，参见 8.3 节。

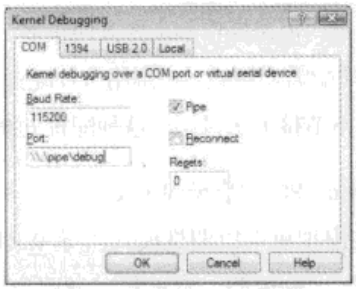


图 A.12 在 windbg 中打开调试会话并连接到虚拟机 COM1 端口

当 windbg 打开命名管道对象并连接到 WRK 虚拟机以后，其界面如图 A.13 所示。

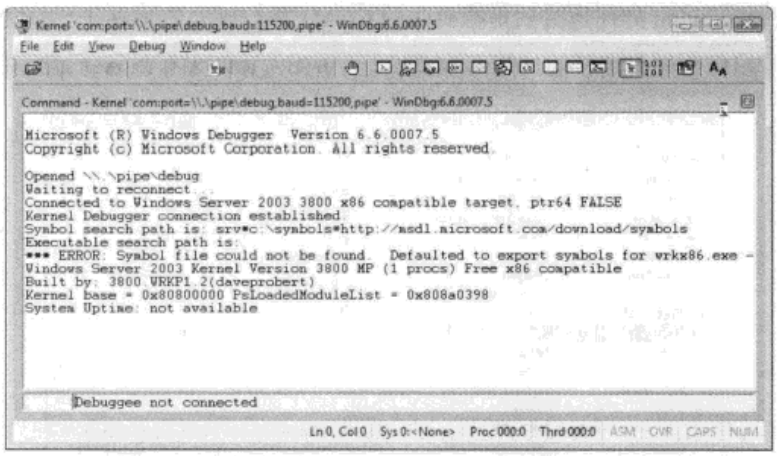


图 A.13 打开内核调试会话

在 windbg 的命令窗口中，我们看到它显示了被调试 Windows 内核的基本信息，包括内核基地址（Kernel base）、模块链表头（PsLoadedModuleList），以及内核版本信息。注意，在命令窗口底下的状态栏显示了“Debuggee not connected”，用户还无法通过命令窗口来控制被调试的 Windows 系统。但是，被调试内核中的调试输出（即内核代码中的 DbgPrint 调用）将显示在此窗口中。

要想使用各种调试命令来检查系统的状态或者控制内核的执行过程，用户首先要让被调试系统停下来。有两种方法可以停止被调试的内核：设定引导时断点，或者在被调试系统执行过程中让它立即停下。

设定引导时断点要求在 Windows 引导早期建立调试会话，即 windbg 必须在引导开始时使用“Kernel Debug...”命令连接到被调试系统，并使用一次或多次“Debug|Kernel Connection|Cycle Initial Break”菜单命令（或者按下“Ctrl+Alt+K”组合键），以便设定引导时断点。随着每一次使用“Cycle Initial Break”命令，windbg 的输出窗口会循环显示以下提示信息的每一行：

```
Will request initial breakpoint at next boot.
Will breakin on first symbol load at next boot.
Will NOT breakin at next boot.
```

如果用户能够赶在内核引导过程的初始断点或第一次符号加载发生之前设定好引导断点，那么，当被调试系统的指令流到达此断点处时，该系统的内核将停止执行，于是用户可以使用各种调试命令来控制被调试系统。反之，若用户错过了这两个断点时机，则必须使用后面介绍的第二种方法来停止被调试内核。对照 WRK 的代码，我们可以看到，当用户使用“Cycle Initial Break”命令来设定初始断点（initial breakpoint）以停止内核执行时，实际上此初始断点是由执行体初始化函数 ExpInitializeExecutive 来判断并响应的，见 base\ntos\init\initos.c 文件第 678 行的 if 语句。如果 KdBreakAfterSymbolLoad 变量的值为 TRUE，则调用 DbgBreakPointWithStatus 函数，触发 int 3 中断。这里 KdBreakAfterSymbolLoad 变量的值是由内核调试子系统在初始化时（即 KdInitSystem 函数中），通过调用 KdPollBreakIn 函数与外部的调试器通信而获得的。

如果用户设定在引导时当第一次加载符号时（breakin on first symbol load）停止内核，则该断点是由 Windows 的内核调试子系统初始化函数（即 KdInitSystem）发起的。KiSystemStartup 调用 KdInitSystem 函数，而 KdInitSystem 调用 DbgLoadImageSymbols 函数以通知调试器加载符号文件，所以调试器可以在这个点上获得控制权。

第二种方法非常简单，它可以在调试会话建立起来以后的任何时刻让被调试系统停止执行。用户使用菜单命令“Debug|Break”（或按下“Ctrl+Break”键），被调试系统便会在当前执行点上停下。图 A.14 显示了当用户在 windbg 中使用“Break”命令来中断 WRK 虚拟机时的情形。

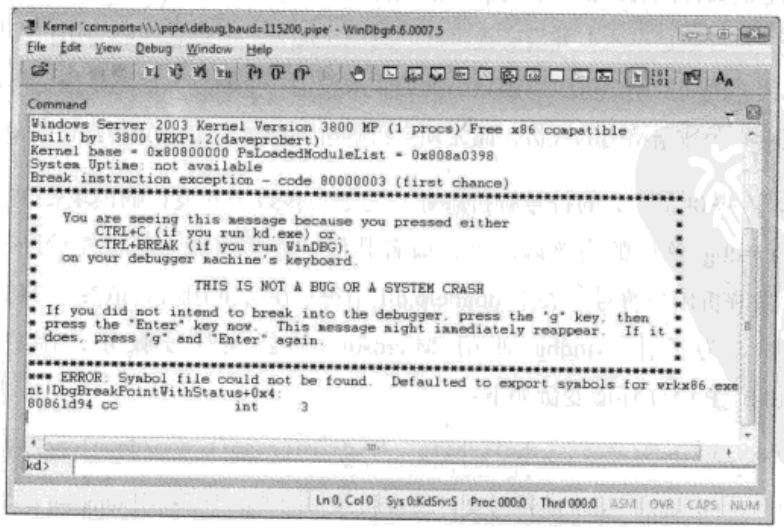


图 A.14 让被调试系统停下来

一旦被调试的系统停了下来，用户就可以使用各种调试命令了，一些常用的调试命令如表 A.1 所列。Windbg 支持大量的调试命令。除了基本调试命令，它还支持各种扩展命令，允许一些插件模块提供特定的调试功能。关于 windbg 的基本调试命令和各种扩展调试命令，请参考 windbg 的联机帮助。此外，在 windbg 集成环境中，对于大多数常用调试命令，用户也可以通过菜单命令或其他用户界面元素来完成同样的功能。

表 A.1 一些常用的调试命令

调试命令	含义	调试命令	含义
bp、bu、bm	设置一个或多个断点	r	显示或修改寄存器
bl	列出当前已设置断点的信息	rdmsr、wrmsr	读或写 MSR 寄存器
bc	清除断点	dd、db 等	显示数据
bd、be	禁止或启用一个或多个断点	u	显示汇编代码
lm	列出已加载的模块	dt	显示数据结构的信息
ld	加载指定模块的符号	p	单步执行一条指令或一行代码
x	显示符号，支持通配符	t	单步执行，进入被调用例程
.reload	为指定的模块重新加载符号	g	恢复执行
!object	显示系统对象的信息	k、kp 等	显示调用栈
!process	显示进程的信息	!thread	显示线程的信息

在利用 windbg 来调试 Windows 内核或者 Windows 应用程序时，符号信息往往起着至关重要的作用。诸如断点设置、变量内容查看、调用栈解析等常用的调试行为，往往都会涉及符号解析。这里的符号信息是指编译器在生成一个二进制模块时额外提供的符号数据，通常是一个与二进制文件同名但扩展名为.pdb 的文件，其中包括函数地址和参数、各种变量的地址和类型等。符号信息本质上并没有提供额外的调试能力，但是它能够为用户提供更为友好的输入和输出。例如，用户可以以函数名的形式来设定一个断点，而不是指定代码地址；可以方便地查看一个数据结构的成员，而无须自己解析数据区中每一个字节的含义。

Microsoft 提供了公共的符号解析服务，为绝大多数公开发行的模块提供了在线符号信息。像 windbg 这样的内核调试器，或者其他一些常用的系统性能诊断工具，使用 dbghelp.dll 来解析用户符号，关于 dbghelp.dll 中符号函数的用法，请参考 MSDN 中的文档[MSDN-DBHP]。为了让 windbg 使用 Microsoft 的公共符号服务，用户可以设置 _NT_SYMBOL_PATH 环境变量如下：

```
set _NT_SYMBOL_PATH=srv*C:\Symbols*http://msdl.microsoft.com/download/symbols
```

这里两个星号之间的路径是本地符号文件缓存目录。用户也可以利用 windbg 的“File|Symbol File Path...”菜单命令（或按 Ctrl+S 键），在“Symbol Search Path”对话框中设定符号搜索

路径，用与上述环境变量中同样的 srv 语法来指向 Microsoft 的公共符号服务。

如果用户要设定多个符号搜索路径，可以在_NT_SYMBOL_PATH 环境变量或“Symbol Search Path”对话框中使用分号将其隔开。Dbghelp 会依次检查这些搜索路径。这对于在调试过程中涉及多个不同来源的模块非常有意义。值得特别一提的是，为了调试 WRK 的自编译内核模块，读者可以将编译 WRK 时生成的 wrkx86.pdb 文件放在符号搜索路径范围中。如果用户是在编译 WRK 的机器上使用 windbg，并且.pdb 文件仍保留在 WRK 的 BUILD\EXE 目录下，那么，无须任何额外设置，dbghelp 就可以利用 wrkx86.exe 中的信息定位到 wrkx86.pdb，从而使 windbg 能够解析 WRK 内核符号。

接下来演示几个简单的调试命令的用法。假设 windbg 的符号设置已经正确，并且被调试的 WRK 系统已经停止。若读者在命令窗口中输入“lm”命令，则在输出的模块列表中关于“nt”的行应该如下所示：

```
80800000 80a42000   nt        (private pdb symbols)
     D:\WRK\WRK-v1.2\base\ntos\BUILD\EXE\wrkx86.pdb
```

这说明内核模块的符号已被正确加载。

接下来设定一个断点。比如我们想看一下线程创建过程，于是在 PspCreateThread 函数开始处设置一个断点。以下命令可以做到这一点：

```
kd> bp nt!PspCreateThread
```

或者用户可以在 windbg 中打开 base\ntos\ps\create.c 文件，然后移动光标至 PspCreateThread 函数开始处，按下 F9 键设置一个新的断点。如果用户想要验证新断点是否已经生效，可以使用 bl 命令查看当前断点列表。

然后输入“g”命令或按下 F5 键，系统运行，一旦 PspCreateThread 函数被调用，则系统又会停下来，输入焦点又回到命令窗口。此时，用户可以查看 PspCreateThread 的参数，还可以检查当前的调用栈，访问当前进程或线程的各种信息，等等。例如，下面是当 Windows Server 2003 SP1 的 explorer.exe 创建一个线程时的调用栈：

```
kd> kp
ChildEBP RetAddr
f4f7fcc8 809318f1 nt!PspCreateThread(void ** ThreadHandle = 0x00c2d81c, unsigned long
         DesiredAccess = 0x1f03ff, struct _OBJECT_ATTRIBUTES * ObjectAttributes =
         0x00000000, void * ProcessHandle = 0x0000045c, struct _EPROCESS *
         ProcessPointer = 0x00000000, struct _CLIENT_ID * ClientId = 0x00c2d76c, struct
         _CONTEXT * ThreadContext = 0x00c2d870, struct _INITIAL_TEB * InitialTeb =
         0xf4f7fd08, unsigned char CreateSuspended = 0x01 '', <function> * StartRoutine
         = 0x00000000, void * StartContext = 0x00000000)
         [D:\WRK\wrk-v1.2\base\ntos\ps\create.c @ 295]
f4f7fd3c 80882a2c nt!NtCreateThread(void ** ThreadHandle = 0x00c2d81c, unsigned long
         DesiredAccess = 0x1f03ff, struct _OBJECT_ATTRIBUTES * ObjectAttributes =
```

```
        0x00000000, void * ProcessHandle = 0x0000045c, struct _CLIENT_ID * ClientId
        = 0x00c2d76c, struct _CONTEXT * ThreadContext = 0x00c2d870, struct _INITIAL_TEB
        * InitialTeb = 0x00c2d4f4, unsigned char CreateSuspended = 0x01 '')+0xdd
        [D:\WRK\wrk-v1.2\base\ntos\ps\create.c @ 168]
f4f7fd3c 7c82ed64 nt!KiFastCallEntry(<HRESULT 0x80004005>)+0xfc
        [D:\WRK\WRK-v1.2\base\ntos\ke\i386\trap.asm @ 1369]
00c2d4a0 7c8212f4 ntdll!KiFastSystemCallRet
00c2d4a4 77e5174c ntdll!ZwCreateThread+0xc
00c2dcc4 77e42474 kernel32!CreateProcessInternalW+0x1d66
00c2dcfc 7c92829b kernel32!CreateProcessW+0x2c
00c2e780 7c927ff9 shell32!_SHCreateProcess+0x387
00c2e7d4 7c927ab4 shell32!CShellExecute::_DoExecCommand+0xb4
00c2e7e0 7c927106 shell32!CShellExecute::_TryInvokeApplication+0x49
00c2e7f4 7c92703c shell32!CShellExecute::ExecuteNormal+0xb1
00c2e808 7c926fd4 shell32!ShellExecuteNormal+0x30
00c2e824 7c8fb591 shell32!ShellExecuteExW+0x8d
00c2ec84 7c8fb3f7 shell32!ShellExecCmdLine+0x143
00c2f150 7c8fb2ad shell32!CRunDlg::OKPushed+0x179
00c2f160 7739c3e7 shell32!RunDlgProc+0x121
00c2f18c 77394e13 USER32!InternalCallWinProc+0x28
00c2f208 77394cb6 USER32!UserCallDlgProcCheckWow+0x147
00c2f250 7738b2c4 USER32!DefDlgProcWorker+0xa8
00c2f280 7739d035 USER32!SendMessageWorker+0x37b
```

从调用栈中我们可以看出，系统服务 NtCreateThread 是如何被应用程序调用的。

调试 Windows 内核通常不像调试一个应用程序那样简单和直截了当。有时候，为了捕捉一个调用点，须要非常谨慎地理解断点处的执行条件。多方面的因素导致了内核调试的复杂性：第一，内核底层许多代码是异步执行的，有些例程的发起点和执行点在时间上是分离的，而在空间上不仅相隔很远，甚至可能有额外模块介入；第二，内核代码在执行时可能处在不同的进程环境中，如果要关联到用户模式逻辑，最好以用户程序或系统服务作为切入点；第三，低中断级别（IRQL）的代码可能会被高中断级别的代码打断，从而导致不连贯的指令流；另外还有一些不确定因素也会造成代码逻辑被打断，例如多处理器或多核、时钟中断、线程切换等。可以这样讲，只有熟悉 Windows 内核中的各种基本系统机制，才可能灵活地驾驭 Windows 内核中的代码逻辑，并充分利用内核调试器 windbg 的调试能力，以达到梳理代码逻辑的目的，或者确定内核代码或驱动程序中的逻辑错误（bug）。总之，相比应用程序调试，内核调试需要更多的创新性方法，而前提条件则是对内核知识有足够的了解。

附录 B

内核代码插入工具 KInjectToolKit

在 Windows 中，内核模式代码要么由操作系统的内核模块（包括 ntoskrnl.exe 或它的变体，以及 HAL）提供，要么位于设备驱动程序中。设备驱动程序可以看做是 Windows 内核的动态链接库，它们在系统运行时被链接到内核中，其入口函数是初始控制点。内核与设备驱动程序之间的通信按照 Windows I/O 模型来进行：设备驱动程序设置 I/O 分发例程，以处理它所感兴趣的 I/O 请求，或者它也可以注册中断服务例程（ISR）、定时器例程或回调例程等，以便当系统中特定事件发生时它的例程会被调用。本附录将讨论另外一种注入内核代码的方式：在用户模式下将代码链接好，然后注入到内核的特定执行点上。这样，当内核模式的控制流到达这些执行点时，用户注入的代码就会被执行。

本附录介绍的工具 KInjectToolKit 实现了这样一套代码注入机制。KInjectToolKit 允许用户程序将 DLL 中的代码插入到内核中特定的执行点上，当处理器的控制流到达这些点上时，用户插入的代码就会被执行。KInjectToolKit 能够将一个 DLL 中的函数按照内核代码来重定位，并动态链接到当前内核模块上。KInjectToolKit 提供了一种不同于设备驱动程序的内核代码加载方法。从内核事件的角度来看，KInjectToolKit 为用户程序提供了一个动态响应内核事件的执行环境，这超越了 ETW（Event Tracing for Windows）提供的内核事件日志记录能力。在某种程度上，它更接近于 SUN Solaris 的 DTrace 机制所提供的内核代码动态监控（instrumentation）[1]能力[DTRACE]。

[1] 这里 instrumentation 是指在内核原有功能代码的基础上附加上额外的代码来完成事件响应能力。翻译成监控并不十分恰当，但通常可作监控之用。

B.1 KInjectToolKit 功能介绍

为了让用户程序可以往内核中插入代码，并且在特定控制点上执行这些代码，KInjectToolKit 包含内核模式和用户模式两部分，其结构如图 B.1 所示。图中内核模块内部的回代表内核事件点，Injector 是一个用户模式程序，它通过内核事件管理器提供的系统服务将 DLL 中的 EventHandler 事件函数关联到相应的事件点上。

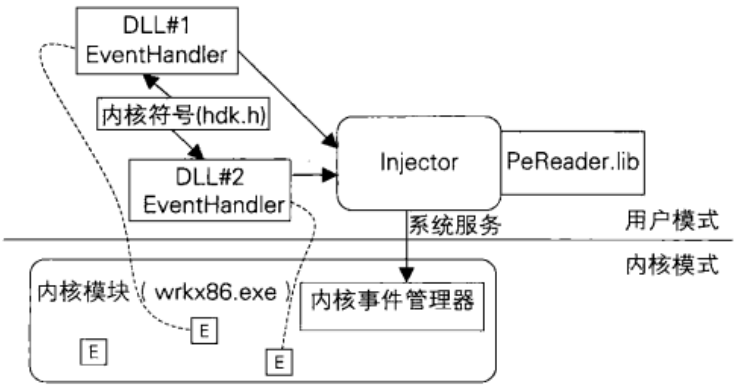


图 B.1 KInjectTooKit 结构示意图

在内核模式下有一个内核事件管理器，它维护着一组数据结构，记录了各个内核事件的相关信息。KInjectToolKit 要求预先定义一组事件触发点（即图中的回点），并为它们编号。KInjectToolKit 不支持在内核的任意点上触发事件，而只能在一些预定义的点上触发事件。为了触发事件，KInjectToolKit 要求修改 Windows 的源代码，比如，线程切换事件发生在 KiSwapThread 函数中某个恰当的代码点上，而进程创建事件发生在 PspCreateProcess 函数内部的某个点上。因此，为了实现内核事件响应机制，我们必须先定义好内核事件集合，这一步骤称为内核事件布点。

内核事件管理器的初始化可以在操作系统内核初始化时完成，比如我们将这一初始化过程放在 Windows 内核的阶段 1 初始化中（由 Phase1InitializationDiscard 函数来调用），它的主要任务是初始化内核事件数据结构。

此外，内核事件管理器也要提供一个事件响应例程，供事件触发点调用。每个触发点在调用此例程时都会指明该触发哪种事件。内核事件管理器还提供一组系统服务，让用户程序可以插入事件函数。为了向内核事件管理器注册一个事件函数，用户程序需要分两步进行：首先请内核事件管理器申请一块足以放得下此事件函数的内存；得到了此内存地址以后，可以对该函数中的某些指令进行必要的重定位处理，然后请内核事件管理器将事件函数的代码拷贝到系统地址空间（即上一步申请到的内存）中。因此，内核事件管理器提

供了两个系统服务，NtAllocEventHandler 和 NtAddEventHandler，分别完成内核事件的内存分配和代码插入操作。另外一个系统服务，NtRemoveEventHandler，用于移除一个已注册的内核事件函数，包括从内核事件管理结构中移除它，并释放该内核事件函数所占内存。

在用户模式部分，KInjectToolKit 对于内核事件函数的编写提供了开发上的支持，包括三部分，如下所列：

- 首先，读者可以想象，在一个内核事件函数中，它需要使用内核中已有的系统功能，比如调用 KeSetEvent 和 ExAllocatePool 之类的内核 API 函数，所以，KInjectToolKit 提供了一个工具，来找到需要用到的这些内核函数（位于 hdk.h 头文件中）在内核模块中的地址，因而内核事件函数可以简单地调用这些内核函数。
- 其次，KInjectToolKit 提供了对 PE 文件格式的解析支持，包括解析 PE 文件的头信息、计算加载 PE 文件所需要的内存大小、对 PE 文件进行重定位处理、在 PE 文件的导出表（export table）中查找一个符号的地址。在 KInjectToolKit 中，PeReader 是一个静态链接库。
- 此外，KInjectToolKit 提供了一个用于插入 DLL 模块的工具 Injector。它将一个 DLL 中名为“EventHandler”的导出函数，注册到指定编号的内核事件上。Injector 使用 PeReader 来解析 DLL 文件，并调用前面提到的三个系统服务来注册或注销一个内核事件响应函数。

一旦 Injector 工具将一个 DLL 中的事件函数注册到内核事件管理器中，以后，当该事件被触发时，此 DLL 中的事件函数就会在内核模式下被调用。Injector 工具也提供了注销一个事件函数的能力。

B.2 KInjectToolKit 的代码实现

上一节介绍了 KInjectToolKit 的各个组件，下面我们来看一下它们的代码实现。首先，在内核模式部分，所有预定义的内核事件构成了一个 EVENT_HANDLER 数组。每个内核事件对应于一个 EVENT_HANDLER 结构，其定义如下：

```
typedef struct _EVENT_HANDLER {
        PVOID HandlerAddress;           // 对齐处理后的内核模式内存地址
        PFUNC HandlerEntry;             // 事件响应函数的地址
        SIZE_T HandlerSize;             // 代码区的大小（按字节计）
} EVENT_HANDLER, *PEVENT_HANDLER;
```

内核事件管理器只是一个虚体，其最基本的状态是 EVENT_HANDLER 全局数组

GlobalEventHandlers。初始化函数 EhInitialize 完成对该数组的零初始化。它在内核的阶段 1 初始化函数 Phase1InitializationDiscard 中被调用。注意，每个预定义的事件都有一个 ID，作为一种简单的实现方案，我们直接使用每个事件在全局数组中的索引作为它的 ID，所以，给定一个事件 ID，内核事件管理器可以立即定位到该事件的 EVENT_HANDLER 结构。

在每个内核事件触发点上，它调用事件触发函数 EhTouchEvent。EhTouchEvent 函数根据传递给它的事件 ID，定位到相应的 EVENT_HANDLER 结构，只要它的 HandlerEntry 成员不为 0，则调用 HandlerEntry 所指的函数。在当前的实现中，我们假定事件函数只带一个 void *参数。以线程切换事件和进程创建事件为例，KiSwapThread 和 PspCreateProcess 函数都调用了 EhTouchEvent 函数，此外，PspCreateProcess 在调用该函数时会将进程的映像文件名称传递给它，所以，进程创建事件的响应函数可以在参数中获得此名称。

在内核中，另有三个系统服务 NtAllocEventHandler、NtAddEventHandler 和 NtRemoveEventHandler 被加入到系统服务表。这三个系统服务的原型如下：

```
HANDLER_ERROR
NtAllocEventHandler(
    IN EVENT EventId,                   // 事件 ID
    IN ULONG CodeAlignment,             // 代码对齐要求
    IN SIZE_T CodeSizeInBytes,          // 代码大小（字节数）
    OUT PPVOID HandlerBase              // 返回内核中申请到的代码地址
    );

HANDLER_ERROR
NtAddEventHandler(
    IN EVENT EventId,                   // 事件 ID
    IN PVOID HandlerAddr,               // 用户空间中的函数地址
    IN PFUNC HandlerEntry               // 内核空间中的函数地址
    );

HANDLER_ERROR
NtRemoveEventHandler(
    IN EVENT EventId                    // 事件 ID
    );
```

这三个系统服务函数的实现比较直截了当：NtAllocEventHandler 调用 ExAllocatePool 函数从非换页内存池中申请一块指定大小的内存，并设置到相应事件的 EVENT_HANDLER 结构中；NtAddEventHandler 负责将用户程序指定的函数代码拷贝到事件结构中已分配好的内存块中，然后设置事件结构的 HandlerEntry 成员；NtRemoveEventHandler 负责释放内存块，并重置事件结构的状态。

Windows 内核的系统服务表是在 base\ntos\ke\i386\systable.asm 文件中定义的，WRK 中原来的系统服务表尾部如图 B.2(a)所示，新加入三个系统服务以后如图 B.2(b)所示，其中斜体的三行是新增的。

```
TABLE_ENTRY WaitForKeyedEvent, 1, 4
TABLE_ENTRY QueryPortInformationProcess, 0, 0
TABLE_ENTRY GetCurrentProcessorNumber, 0, 0
TABLE_ENTRY WaitForMultipleObjects32, 1, 5

TABLE_END 295
```

(a) WRK 原始的系统服务表

```
TABLE_ENTRY WaitForMultipleObjects32, 1, 5
TABLE_ENTRY AllocEventHandler, 1, 4
TABLE_ENTRY AddEventHandler, 1, 3
TABLE_ENTRY RemoveEventHandler, 1, 1

TABLE_END 298
```

(b) 在尾部加入三个系统服务

图 B.2 在 WRK 中加入三个系统服务前后的系统服务表

这里每个表项后面的数字，第一个数字 0 或 1 表示该系统服务是否带参数；第二个数字表示带几个参数。接下来还需要在 systable.asm 文件随后的参数表（ARGTBL）中加入一行，以指示系统服务分发代码将用户地址空间中的参数拷贝到内核栈中。图 B.3 显示了修改参数表前后的对照，其中图 B.3(b)中的斜体行代表新增的 ARGTBL_ENTRY（可对应 8 个系统服务）。

```
ARGTBL_ENTRY 4,8,8,20,16,8,8,16
ARGTBL_ENTRY 20,12,4,4,36,36,24,20
ARGTBL_ENTRY 0,16,12,16,16,0,0,20

ARGTBL_END
```

(a) WRK 原始的 ARGTBL 定义

```
ARGTBL_ENTRY 20,12,4,4,36,36,24,20
ARGTBL_ENTRY 0,16,12,16,16,0,0,20
ARGTBL_ENTRY 16,12,4,0,0,0,0,0

ARGTBL_END
```

(b) 在尾部加入一个 ARGTBL_ENTRY

图 B.3 在 WRK 中加入三个系统服务前后的参数表

现在我们来看用户模式部分的代码实现。首先来看一个简单的 hdk.h 样例文件：

```
#ifndef _HDK_
#define _HDK_

#define EXPORT __declspec(dllexport)
#pragma warning(disable:4047)

extern unsigned int (*DbgPrint)(char *Format, ...);

#endif
```

用户可以进一步扩展 hdk.h 中的函数声明，即加入更多的 extern 行，KInjectToolKit 中的 hdk 工具可以扫描 hdk.h 文件，为其中的每一个 extern 声明，输出类似下面的行：

```
unsigned int (*DbgPrint)(char *Format, ...) = 0x8080c59c;
```

Hdk 工具利用 dbghelp.dll 提供的符号解析能力，从指定的.pdb 文件中获得符号信息。有关 dbghelp.dll 的符号解析能力，这里不作讨论，读者可以参考 MSDN 的文档[MSDN-DBHP]。此后，当用户编写事件 DLL 时，只要同时包含 hdk.h 和以上输出文件，就可以在事件响应函数中方便地调用像 DbgPrint 这样的内核函数了。

接下来看一下 KInjectToolKit 中的 PeReader 库，它的功能是解析 PE 文件，并为 Injector 提供代码重定位、PE 头信息提取和导出表搜索等功能。关于 PE 文件格式的更多信息，

请参考 2.2.2 节中的介绍以及有关文章[MSDN-PE1][MSDN-PE2]，这里仅列出 Injector 中用到的 4 个函数：

- PeReaderParsePe：利用已读入的 PE 文件数据，构造内部结构。
- PeReaderGetLoadedSize：计算出为加载 PE 文件而需要的内存大小。
- PeReaderLoad：完成加载动作，包括代码重定位。
- PeReaderSeekExportedName：搜索 PE 文件导出表，返回指定名称的函数的地址。

最后我们看一下 Injector 工具的实现。它提供了注册和注销内核事件响应函数的能力。为了注册一个内核事件函数，Injector 读入一个 DLL 文件，利用 PeReader 库提供的 PE 文件解析功能，获得该 DLL 文件中的 EventHandler 导出函数的地址，并重定位到通过 NtAllocEventHandler 系统服务分配的系统内存上，然后调用 NtAddEventHandler 系统服务将事件函数拷贝到系统内存中，并注册到内核事件管理器中。注销内核事件的实现要简单得多，Injector 直接调用 NtRemoveEventHandler 来完成注销操作。值得注意的一点是，为了调用这些系统服务，Injector 为每个服务编写了一个存根函数，下面是存根函数的汇编代码实现：

```
SysEntry macro name, id
   public _Nt&name
_Nt&name:
   mov eax, id
   call _SystemCallStub
   ret
   endm

   public _SystemCallStub
_SystemCallStub:
   mov edx, esp
   db 0fH, 34H // sysenter 指令

SysEntry AllocEventHandler, 128H
SysEntry AddEventHandler, 129H
SysEntry RemoveEventHandler, 12AH
```

这一做法等同于 ntdll.dll 中为内核本身的系统服务而提供的存根函数（譬如 NtCreateProcess），因此，Injector 的 C 代码只要按照常规的做法来调用 NtAllocEventHandler、NtAddEventHandler 和 NtRemoveEventHandler 系统服务即可。

KInjectToolKit 还提供了两个样例事件 DLL，分别响应线程切换和进程创建事件。它们的代码非常简单，如图 B.4 所列，这里 kernfunc.inc 是根据 hdk.h 生成的内核符号定义头文件，其中包含了内核函数 DbgPrint 的地址。

```
#include "hdk.h"
#include "kernfunc.inc"

EXPORT void EventHandler(void *Arg)
{
   DbgPrint("Calling Swap Thread…\n");
}
```

(a) 线程切换事件的 EventHandler

```
#include "hdk.h"
#include "kernfunc.inc"

EXPORT void EventHandler(void *Arg)
{
    char *ImageFileName = (char *)Arg;
    char SafeName[16];

    strncpy_s(SafeName, 16, ImageFileName, 16);
    SafeName[15] = '\0';
    DbgPrint("Create Process: %s\n", SafeName);
}
```

(b) 进程创建事件的 EventHandler

图 B.4 KInjectToolKit 随带的两个样例事件 DLL 的 EventHandler 代码

B.3 KInjectToolKit 的限制

KInjectToolKit 只是一个学生项目，并非一个完善的内核事件响应机制。但是，它的意义在于，提供了一种不同于设备驱动程序的内核代码注入方案。KInjectToolKit 的目标是，让用户像编写普通用户模式 DLL 一样来编写内核中的代码。仔细观察图 B.4 中所列示的代码就会发现，这一目标已经初步达到。

然而，目前实现的 KInjectToolKit 还有诸多限制，有些是该方案的本质性问题，而另一些则是由于工程方面的原因。下面列出其中一些重要的限制：

- 为了在内核中进行事件布点，需要修改 Windows 内核源代码，而 Windows 不是一个开放源代码的操作系统，因此，KInjectToolKit 仅仅在 WRK 上可以工作。目前这是一个无法解决的问题，除非使用内核逆向工程或者内核钩子的方法在一个非 WRK 的系统上进行布点。然而，将来有可能的变化是，操作系统会提供布点的机会，但从安全性的角度来考虑，操作系统内核总是要禁止插入未经验证的代码。
- 事件函数代码的保护。事件函数直接运行在内核中，即使简单的程序错误也有可能导致系统崩溃，因此，事件函数必须在保护环境下运行，最起码常见的异常应该能被捕捉到。本质上讲，事件函数与设备驱动程序中的函数是一样的，事件函数必须遵从所对应的事件触发点处的约束，比如 IRQL 约束、内存是否可换页等。因此，这些约束构成了每个事件的执行条件。
- 增加系统服务或者增加系统服务表只是为了让用户模式程序以相对简捷的途径与内核中的功能代码进行通信。KInjectToolKit 中使用系统服务来注册或注销内核事件响应函数并不必要，它也可以使用设备驱动程序来完成。
- 在事件函数中能调用哪些内核函数（即 hdk.h 中的 extern 声明），此范围受内核模块符号信息的影响。目前是在编译时解析这些内核函数的地址，从而为一台机器编写的

内核事件 DLL 可能无法在另一台机器上运行。如果改成在运行时动态地解析所调用的函数的地址，则可望解决或缓解这一问题。此外，若能利用 Microsoft 提供的公共符号服务，那么，事件函数也可以调用一些内部函数。然而，调用内部函数有可能会导致事件 DLL 在不同 Windows 版本上的兼容性问题，而且，如何获得内部函数的详细原型信息，也需要通过其他的途径来解决。

尽管存在以上这些限制，KInjectToolKit 仍然是一个值得学习的项目包。如果读者能访问 WRK 的代码，建议读一读 KInjectToolKit 中的源代码，并动手增加一些自己感兴趣的内核事件。

参考资料

[ALG] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithm*, Second Edition, The MIT Press, 2001.

[CHANDY-MISRA] Chandy, K.M.; Misra, J., *The Drinking Philosophers Problem*, ACM Transactions on Programming Languages and Systems, 1984.

[COFFMAN] E. G. Coffman, Jr. M. J. Elphick, and A. Shoshani, *System Deadlocks*, Computing Surveys, Vol. 3, No. 2, pp.67-78, June 1971.

[COMP-NW] Andrew S. Tanenbaum, *Computer Networks*, Fourth Edition, Prentice Hall PTR, 2003.

[COREY] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang and Zheng Zhang, *Corey: An operating system for many cores*, In Proceedings of the 8th Symposium on OS Design and Implementation (OSDI), 2008.

[DDK-NW-SAMP] *Network Samples*.
http://msdn.microsoft.com/en-us/library/dd163322.aspx

[DH-MON] *Desktop Heap Monitor*.
http://www.microsoft.com/downloads/thankyou.aspx?familyId=5cfc9b74-97aa-4510-b4b9-b2dc98c8ed8b&displayLang=en

[DIJKSTRA] Dijkstra E. W., *Co-operating Sequential Processes*, in Programming Languages, Genuys, F. (Ed.), London, Academic Press, 1965.

[DTRACE] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal, *Dynamic Instrumentation of Production Systems*, Proceedings of the 2004 USENIX Annual Technical Conference.

[EXCEPT-1] Matt Pietrek , *A Crash Course on the Depths of Win32™ Structured Exception Handling*, Microsoft Systems Journal, 1997, January.
http://www.microsoft.com/msj/0197/exception/exception.aspx

[EXCEPT-2] Matt Pietrek , *New Vectored Exception Handling in Windows XP*, MSDN Magazine, 2001, Sept.
http://msdn.microsoft.com/en-us/magazine/cc301714.aspx

[EXOKERNEL] Dawson Engler, Frans Kaashoek, James O'Toole, *Exokernel: An Operating System Architecture for Application-Level Resource Management*, Proceedings of the fifteenth ACM symposium on Operating systems principles, 1995. p251–266.

[FAT] *Microsoft Extensible Firmware Initiative FAT32 File System Specification.*
http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx

[FILEMON] *FileMon for Windows.*
http://technet.microsoft.com/en-us/sysinternals/bb896642.aspx

[FN-TUNNELING] *File Name Tunneling* or *FltGetTunneledName.*
http://support.microsoft.com/?kbid&id=172190
http://msdn.microsoft.com/en-us/library/aa488608.aspx

[FS-FLT-CLS] Windows DDK: *File System Filter Driver Classes and Class GUIDs.*
http://msdn.microsoft.com/en-us/library/ms790747.aspx

[FS-FLT-INF] Windows DDK: *Creating an INF File for a File System Filter Driver.*
http://msdn.microsoft.com/en-us/library/ms790727.aspx

[FS-MINIFLT-ADV] *Advantages of the Filter Manager Model.*
http://msdn.microsoft.com/en-us/library/aa488086.aspx

[FS-MINIFLT-INF] *Creating an INF File for a Minifilter Driver.*
http://msdn.microsoft.com/en-us/library/ms793586.aspx

[FS-MINIFLT-ORDER] Windows DDK: *Load Order Groups and Altitudes for Minifilter Drivers.*
http://msdn.microsoft.com/en-us/library/aa488218.aspx

[GOOGLE-OS] *Google Chrome.*
http://www.google.com/chrome

[HPI] Michael Schöbel, *Howto: Implementation of new system service calls* (I-III).
http://www.dcl.hpi.uni-potsdam.de/research/WRK/?p=28
http://www.dcl.hpi.uni-potsdam.de/research/WRK/?p=29
http://www.dcl.hpi.uni-potsdam.de/research/WRK/?p=44

[IBM-S370] A. Padegs, *System/370 Extended Architecture: design considerations*, IBM Journal of Research & Development (IBM) 27 (3), May 1983, p198–205.

[INTEL-253666] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference A-M*, Nov. 2008.
http://www.intel.com/assets/pdf/manual/253666.pdf

[INTEL-253667] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference N-Z*, Nov. 2008.
http://www.intel.com/assets/pdf/manual/253667.pdf

[INTEL-253668] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, Nov. 2008.
http://www.intel.com/Assets/PDF/manual/253668.pdf

[IPSEC-RFC] Kent S., Atkinson R., *Security Architecture for the Internet Protocol*, RFC 2401, Nov. 1998.
http://www.ietf.org/rfc/rfc2401.txt

[L4-OS] *The L4 μ-Kernel Family*.
http://os.inf.tu-dresden.de/L4/

[MACH] Black D.L., Golub D.B., Julin D.P., Rashid R.F., Draves R.P., Dean R.W., Forin A., Barrera J., Tokuda H., Malan G., and Bohman D, *Microkernel Operating System Architecture and Mach*, Journal of Information Processing, 14 (4), March 1992.

[MARS-PATHFINDER] *Explanation of priority inversion problem experienced by Mars Pathfinder*.
http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html

[MINIX] *The Minix 3 Operating System*.
http://www.minix3.org

[MOS] Andrew S. Tanenbaum, *Modern Operating Systems*, Second Edition, Prentice Hall, 2001.

[MSDN-BOOT] *Introduction to Boot Options*.
http://msdn.microsoft.com/en-us/library/ms791478.aspx

[MSDN-D3D] *Direct3D*.
http://msdn.microsoft.com/en-us/library/ms793073.aspx

[MSDN-DBHP] *Windows Debug Help Library*.
http://msdn.microsoft.com/en-us/library/ms679309(VS.85).aspx

[MSDN-DD] *DirectDraw*.
http://msdn.microsoft.com/en-us/library/ms796178.aspx

[MSDN-DEADLOCK] *Deadlock Detection*.
http://msdn.microsoft.com/en-us/library/ms792855.aspx.

[MSDN-DWM] *Desktop Window Manager*.
http://msdn.microsoft.com/en-us/library/aa969540(VS.85).aspx

[MSDN-ET] *Event Tracing*.
http://msdn2.microsoft.com/en-us/library/aa363787.aspx.

[MSDN-KL] *Configuring and Starting the NT Kernel Logger Session*.
http://msdn.microsoft.com/en-us/library/aa363691(VS.85).aspx

[MSDN-IPSEC] *Deploying IPSec.*
http://technet.microsoft.com/en-us/library/cc737024(WS.10).aspx

[MSDN-MS] *Mailslots.*
http://msdn.microsoft.com/en-us/library/aa365576(VS.85).aspx

[MSDN-NDIS] *NDIS 5.1 and NDIS 6.*
http://msdn.microsoft.com/en-us/library/ms797120.aspx
http://msdn.microsoft.com/en-us/library/aa938296.aspx

[MSDN-NP] *Named Pipes.*
http://msdn.microsoft.com/en-us/library/aa365590(VS.85).aspx

[MSDN-NW] *Network Design Guide.*
http://msdn.microsoft.com/en-us/library/aa938292.aspx

[MSDN-PE1] Matt Pietrek, *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format.*
http://msdn.microsoft.com/en-us/library/ms809762.aspx

[MSDN-PE2] Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format.*
http://msdn.microsoft.com/en-us/magazine/cc301805.aspx
http://msdn.microsoft.com/en-us/magazine/cc301808.aspx

[MSDN-PO] Windows Driver Kit: Kernel-Mode Driver Architecture, *Implementing Power Management.*
http://msdn.microsoft.com/en-us/library/ms798306.aspx

[MSDN-REG] *Registry Data Types.*
http://msdn.microsoft.com/en-us/library/bb773476(VS.85).aspx

[MSDN-SC] *Server Core.*
http://msdn.microsoft.com/en-us/library/ms723891(VS.85).aspx

[MSDN-SERVICE] *Windows Services.*
http://msdn.microsoft.com/en-us/library/ms685141(VS.85).aspx

[MSDN-SUBSYSTEM] *Visual C++ Link Options: /SUBSYSTEM (Specify Subsystem).*
http://msdn.microsoft.com/en-us/library/fcc1zstk.aspx

[MSDN-TDI] *TDI Drivers.*
http://msdn.microsoft.com/en-us/library/ms800790.aspx

[MSDN-TDI-EVENT] Windows Driver Kit: *TDI_SET_EVENT_HANDLER.*
http://msdn.microsoft.com/en-us/library/ms801156.aspx

[MSDN-UMS] *Windows User-Mode Scheduling.*
http://msdn.microsoft.com/en-us/library/dd627187(VS.85).aspx

[MSDN-UNC] *Network Redirector Drivers.*
http://msdn.microsoft.com/en-us/library/ms794161.aspx

[MSDN-VISTA-DDM] *Windows Vista Display Driver Model.*
http://msdn.microsoft.com/en-us/library/ms796819.aspx

[MSDN-W2K-GDI] *Display Drivers and Video Miniport Drivers (Windows 2000 Model).*
http://msdn.microsoft.com/en-us/library/ms797859.aspx
http://msdn.microsoft.com/en-us/library/ms800267.aspx

[MSDN-WINSOCK2] *Windows Sockets 2.*
http://msdn.microsoft.com/en-us/library/ms740673(VS.85).aspx

[MSDN-WINSTA] *About Window Stations and Desktops.*
http://msdn.microsoft.com/en-us/library/ms681928(VS.85).aspx

[MSDN-WMI] *Windows Management Instrumentation.*
http://msdn.microsoft.com/en-us/library/aa384642(VS.85).aspx

[MSDN-WPA] *Windows Performance Analysis.*
http://msdn.microsoft.com/en-us/performance/default.aspx

[MSDN-WPT] *Windows Performance Analysis Tools.*
http://msdn.microsoft.com/en-us/performance/cc825801.aspx

[MSJ-LSP] Wei Hua, Jim Ohlund and Barry Butterklee, *Unraveling the Mysteries of Writing a Winsock 2 Layered Service Provider*, Microsoft System Jounal, May 1999.
http://www.microsoft.com/msj/0599/LayeredService/LayeredService.aspx

[NT-SYSCALL] Mark Russinovich and Bryce Cogswell, *Windows NT System Call Hooking*, Dr. Dobb's Journal, January 1997.
http://www.ddj.com/184410109

[NTFS] *How NTFS Works: Local File Systems.*
http://technet.microsoft.com/en-us/library/cc781134(WS.10).aspx

[PDC-09] Mark Russinovich, *Windows 7 and Windows Server 2008 R2 Kernel Changes.*
http://microsoftpdc.com/Sessions/P09-20

[PE-SPEC] *Microsoft Portable Executable and Common Object File Format Specification.*
http://www.microsoft.com/whdc/system/platform/firmware/PECOFFdwn.mspx

[PETERSON] Patterson G. L., *Myths about the Mutual Exclusion Problem*, Information Processing Letters, Vol. 12, pp. 115-116, June 1981.

[QNX] Dan Hildebrand, *An Architectural Overview of QNX*, Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, 1992, p113–126.

[SINGULARITY] Galen C. Hunt and James R. Larus, *Singularity: Rethinking the Software Stack*, in ACM SIGOPS Operating Systems Review, vol. 41, no. 2, pp. 37-49, ACM SIGOPS, April 2007.

[SPACE] Probert, D., Bruno, J. and Karaorman, M., *Space: A new approach to operating system abstraction*, in Proceedings of the International Workshop on Object Orientation in Operating Systems, pages 133-137, Oct. 1991.

[TAOCP-1] Donald Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Third Edition, Addison-Wesley, 1997, pp. 435-455.

[TAOCP-2] Donald Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*, Second Edition, Addison-Wesley, 1998, pp. 458-481.

[TAOCP-3] Donald Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*, Second Edition, Addison-Wesley, 1998, pp. 525-534.

[TLB-MISS] Harvey G. Cragon, *Memory Systems and Pipelined Processors*, First Edition Jones & Bartlett Publishers, 1996, pp. 162-178.

[TOASTER] *WDM Sample: Toaster*.
http://msdn.microsoft.com/en-us/library/dd163450.aspx

[VISTA-CHG] Mark Russinovich, *Inside the Windows Vista Kernel: Part 1-3*, TechNet Magazine, Jan, Feb and March, 2007.
http://technet.microsoft.com/en-us/magazine/2007.02.vistakernel.aspx
http://technet.microsoft.com/en-us/magazine/2007.03.vistakernel.aspx
http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx

[W2K-SECRETS] Sven Schreiber, *Undocumented Windows 2000 Secrets*, Addison-Wesley Professional, May 2001.

[WALTER] Walter Oney, *Programming the Microsoft Windows Driver Model*, Second Edition, Micrososft Press, 2003.

[WIN-INTERNALS] Mark Russinovich and David Solomon, *Microsoft Windows Internals*, Fourth Edition, Microsoft Press, 2004.

[WIN-INTERNALS-5] Mark Russinovich and David Solomon, *Microsoft Windows Internals*, Fifth Edition, Microsoft Press, 2009.

[WIN08-CHG] Mark Russinovich, *Inside the Windows Server 2008 Kernel*, TechNet Magazine, March, 2008.
http://technet.microsoft.com/en-us/magazine/2008.03.kernel.aspx.

[WIN64-PATCHING] *Patching Policy for x64-Based Systems*.
http://www.microsoft.com/whdc/driver/kernel/64bitPatching.mspx

[WPA-WPT] Michael Milirud, *Windows Performance Analysis: Using Windows Performance Tools.*
http://download.microsoft.com/download/5/E/6/5E66B27B-988B-4F50-AF3A-C2FF1E62180F/COR-T594_WH08.pptx

[WRK] *Windows Research Kernel.*
http://www.facultyresourcecenter.com/curriculum/ResourceKits/os.aspx
http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx

[WSCLOCK] Richard W. Carr and John L. Hennessy, *WSClock – A Simple and Effective Algorithm for Virtual Memory Management*, Proc. of the 8th Symposium on Operating Systems Principles, ACM, 1981, pp. 87-95.

[XEN] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, *Xen and the art of virtualization*, In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP) 2003.

PDG

术语对照表

（英–中）

A

abort　中止

access violation　访问违例

Advanced Configuration and Power Interface, ACPI　高级配置和电源接口

affinity　亲和性

Asynchronous Procedure Call, APC　异步过程调用

attach　附载

B

balance set manager　平衡集管理器

base priority　基本优先级

basic disk　基本磁盘

bin　巢箱

boost　提升

boot　引导

boot partition　引导分区

boot sector　引导扇区

bucket　桶

buddy system　伙伴系统

bug check　错误检查

bus driver　总线驱动程序

busy-wait　忙等待

C

cache line　缓存线

cache manager　缓存管理器

call stack　调用栈

callout function　出调函数

cell　巢室

cell index　巢室索引

Central Processing Unit, CPU　中央处理单元

class driver　类驱动程序

client ID handle table　CID 句柄表

clock interrupt　时钟中断

cluster　簇（用于磁盘存储单元）

commit　提交

concurrency　并发性

configuration manager　配置管理器

context　环境

context switch　环境切换

copy-on-write　写时复制

crash dump　崩溃转储

critical section　临界区

D

deadlock　死锁

debugger　调试器

Deferred Procedure Call, DPC　延迟过程调用

deliver　交付（针对 APC 和 DPC）

desktop composition　桌面合成

Desktop Window Manager, DWM　桌面窗口管理器

device driver　设备驱动程序

device enumeration　设备列举

device object　设备对象

device stack　设备栈

device tree　设备树

Direct Memory Access, DMA 直接内存访问
Discretionary Access Control, DAC 自主访问控制
dispatcher 分发器，调度器
dispatcher object 分发器对象
driver object 驱动程序对象
driver verifier 驱动程序检验器
dynamic disk 动态磁盘
Dynamic Link Library, DLL 动态链接库

E

Encrypting File System, EFS 加密文件系统
environment subsystem 环境子系统
event 事件
event manager 事件管理器
Event Tracing for Windows, ETW Windows 事件追踪
exception 异常
exception dispatching 异常分发
exception handler 异常处理器，异常处理例程
executive 执行体
executive resource 执行体资源
exokernel 外核
extended partition 扩展分区
external fragmentation 外碎片

F

fast I/O 快速 I/O
fast mutex 快速互斥体
fault 错误
fiber 纤程
File Allocation Table, FAT 文件分配表
file object 文件对象
file system 文件系统
file system driver 文件系统驱动程序
file system recognizer 文件系统识别器
file system runtime library, FsRtl 文件系统运行库
filter driver 过滤驱动程序
footprint 印迹
frame-based exception handler 基于帧的异常处理器
function driver 功能驱动程序
Functional Device Object, FDO 功能设备对象

G

gate 门
gate object 门对象
Global Descriptor Table, GDT 全局描述符表
global namespace 系统全局名字空间
graphical identification and authentication, GINA 图形化标识和认证
Graphics Device Interface, GDI 图形设备接口
guard page 守护页面
GUID 全局唯一标识符
GUID Partition Table, GPT GUID 分区表

H

handle 句柄
handle table 句柄表
hard fault 硬错误
Hardware Abstraction Layer, HAL 硬件抽象层
heap manager 堆管理器
hive 储巢
hook 钩子
hybrid kernel 混合内核

I

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
internal fragmentation 内碎片
interlocked operation 互锁操作
Inter-Process Communication, IPC 进程间通信
Inter-Processor Interrupt, IPI 处理器间的中断
interrupt 中断
interrupt controller 中断控制器
Interrupt Descriptor Table, IDT 中断描述符表
interrupt object 中断对象
interrupt request level, IRQL 中断请求级别
Interrupt Service Routine, ISR 中断服务例程
interrupt vector number 中断向量编号

I/O cancellation　I/O 取消
I/O completion　I/O 完成
I/O completion port　I/O 完成端口

J

job　作业
junction　交接

K

key　键
key control block　键控制块
kernel-level thread　内核级线程
kernel logger　内核日志记录器
kernel mode　内核模式

L

lazy writer　延迟写出器
lightweight process　轻量进程
linear address　线性地址
loader　加载器，加载程序
Local Descriptor Table, LDT　局部描述符表
Local Procedure Call, LPC　本地过程调用
local security authority subsystem, lsass, LSASS　本地安全权威子系统
lock-free singly-linked list　无锁的单链表
logical address　逻辑地址
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

M

macrokernel (monolithic kernel)　大内核
mailslot　邮件槽
mandatory access control　强制访问控制
mapped page writer　映射页面写出器
Master Boot Record, MBR　主引导记录
Master File Table, MFT　主文件表
Memory Descriptor List, MDL　内存描述符链表
memory hierarchy　存储体系
Memory Management Unit, MMU　内存管理单元
memory mapped I/O　内存映射 I/O
microkernel　微内核
micro-kernel　微内核
minifilter driver　小过滤驱动程序
miniport driver　小端口驱动程序
modified page writer　修改页面写出器
mount　挂载
mount point　挂载点
mutant　突变体
mutex　互斥体
Multiple UNC Provider, MUP　多 UNC 提供者
multithreaded programming　多线程程序设计

N

named pipe　命名管道
native　原生的
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
nonpaged pool　非换页内存池
NT File System　NTFS 文件系统
NUMA, non-uniform memory architecture　非一致的内存结构

O

object body　对象体
object creation　对象创建，对象构造
object header　对象头
object manager　对象管理器

P

page　页，页面
page aging algorithm　页面老化算法
page directory　页目录
Page Directory Entry, PDE　页目录项
page fault　页面错误
page fault handler　页面错误处理例程
page file, paging file　页面文件
page frame　页帧
Page Frame Number, PFN　页帧编号
page replacement algorithm　页面替换算法

page table 页表
Page Table Entry, PTE 页表项
pageable 换页的
paged pool 换页内存池
paravirtualization 仿虚拟化
partition 分区
partition table 分区表
performance counter 性能计数器
physical address 物理地址
Physical Address Extension, PAE 物理地址扩展
Physical Device Object, PDO 物理设备对象
pin 锁定
plug and play, PnP 即插即用
plug and play manager 即插即用管理器
port driver 端口驱动程序
portable executable PE 文件格式
power manager 电源管理器
preemptive 抢先式
primary partition 主分区
priority boost 优先级提升
priority inversion 优先级反转
procedure 过程
process 进程
process address space 进程地址空间
process creation 进程创建
process swapper 进程交换器
process termination 进程终止
processor 处理器
processor control block, PRCB 处理器控制块
processor control region, PCR 处理器控制区
prototype page table entry, prototype PTE 原型页表项，原型 PTE
push lock 推锁

Q

quantum 时限
queue 队列，队列对象

R

race condition 竞争条件
raise IRQL 提升 IRQL
Random-Access Memory, RAM 随机访问存储器
Raw Input Thread, RIT RIT 线程
read-ahead 预读
reference count 引用计数
registry 注册表
registry key 注册表键
registry value 注册表值
reparse 重解析
reparse point 重解析点
reserve 保留
routine 例程
run 行串（用于 NTFS 结构中）

S

scheduler 调度器
schema 表结构
section, memory section 内存区
section object 内存区对象
section table 段表
Security Accounts Manager, SAM 安全账户管理器
Security Descriptor, SD 安全描述符
security identifier, SID 安全标识符
Security Reference Monitor, SRM 安全引用监视器
segment 段
segment selector 段选择符
semaphore 信号量
Service Control Manager, SCM 服务控制管理器
Service Descriptor Table, SDT 服务描述符表
session manager 会话管理器
session space 会话空间
shutdown 停机
socket 套接字
soft fault 软错误
spinlock 自旋锁
stable 稳定的
stack frame 栈帧
stack location 栈单元
stack swapper 栈交换器

standby 备用

startup 启动
starvation 饥饿
storage stack 存储栈
Structured Exception Handling, SEH 结构化异常处理
stub 存根
surface 图形表面
symbolic link 符号链接
synchronization 同步
system address space 系统地址空间
system service 系统服务
system service dispatching 系统服务分发
system view 系统视图

T

TDI transport TDI 传输器
thread 线程
thread creation 线程创建
thread scheduling 线程调度
thread termination 线程终止
timer 定时器
timer object 定时器对象
transition 转移
Translation Look-aside Buffer, TLB 地址转译快查缓冲区
Transport Driver Interface, TDI 传输驱动程序接口
trap 捕获（动词），陷阱（名词）
trap handler 陷阱处理器
trim 修剪

U

user mode 用户模式
user-level thread 用户级线程
User-Mode Scheduling, UMS 用户模式调度

V

Value 值
Vectored Exception Handling, VEH 向量化异常处理
view 视图
virtual address 虚拟地址
Virtual Address Descriptor, VAD 虚拟地址描述符
volatile 易失的、易失性
volume 卷
volume recognition 卷识别

W

wait block 等待块
window management 窗口管理
window station 窗口站
Windows Driver Model, WDM Windows 驱动程序模型
Windows Management Instrumentation,WMI Windows 管理规范
Windows Research Kernel Windows 研究内核
Windows Service Windows 服务
Windows socket, Winsock Windows 套接字
Windows subsystem Windows 子系统
working set 工作集
working set manager 工作集管理器
working set trimming 工作集修剪
WorkItem 工作项目
write-through 直写

Y

yield 放弃

Z

zero page thread 零页面线

PDG

索引

CID 句柄表, 134
GUID Partition Table（GUID 分区表）, 501
I/O 请求包, 440
I/O 请求完成, 456, 464
I/O 取消, 460
I/O 完成端口, 465, 648
NTFS 文件系统, 534
PE 文件格式, 25
RIT 线程, 34, 617
TDI 传输器, 31, 595
Windows 服务, 94, 394
Windows 管理规范, 72, 391
Windows 驱动程序模型, 29, 391, 423
Windows 事件追踪, 72
Windows 套接字, 593, 594
Windows 研究内核, 38
Windows 子系统, 19, 32, 603, 607

A

安全标识符, 77
安全描述符, 77
安全引用监视器, 76, 404, 576
安全账户管理器, 76

B

保留, 235
备用, 154, 267
本地安全权威子系统, 37, 76, 576
本地过程调用, 565
并发性, 51, 298

C

巢室, 64
巢室索引, 64
巢箱, 65
重解析点, 506
储巢, 61
出调函数, 33
处理器控制区, 195
处理器控制块, 88
处理器间的中断, 311
传输驱动程序接口, 31, 595
窗口站, 34, 610
窗口管理, 33, 610
簇, 516
存储体系, 472
存储栈, 500
错误, 314

D

大内核, 14
等待块, 52, 355
地址转译快查缓冲区, 180, 252
电源管理器, 392, 414
调度器, 166
调用栈, 46
定时器, 49, 327, 367
定时器对象, 53
动态磁盘, 501
段, 176, 181
段表, 26
段选择符, 182
端口驱动程序, 425
堆管理器, 193
队列（队列对象）, 52, 364
对象创建（对象构造）, 58
对象管理器, 53, 504
对象体, 54, 361
对象头, 54, 361
多 UNC 提供者, 522, 578
多线程程序设计, 103

F

放弃, 159, 169
访问违例, 257
仿虚拟化, 15
非换页内存池, 44, 209
分发器对象, 52, 361
分区, 499
分区表, 501
服务分发表, 555
服务控制管理器, 92, 94, 438
附载, 332

G

高级配置和电源接口, 414
功能驱动程序, 423
功能设备对象, 413
工作集, 46, 190

工作集管理器, 193, 273, 274, 286
工作集修剪, 193
工作项目, 149, 366, 381
钩子, 618, 623
挂载, 475, 504
挂载点, 506
过滤驱动程序, 424, 434, 523

H

互斥体, 302, 363
互锁操作, 51, 345
缓存管理器, 476
缓存线, 378, 473
换页内存池, 45, 212
环境切换, 165
环境子系统, 19, 32
会话管理器, 91
会话空间, 198
混合内核, 14
伙伴系统, 186

J

饥饿, 306
基本优先级, 48
基本磁盘, 501
即插即用, 391, 408
即插即用管理器, 391, 408
基于帧的异常处理器, 50
加密文件系统, 542
加载器（加载程序）, 82, 642
交付, 50, 324, 333
交接, 535, 542, 649
键, 61
键控制块, 70
结构化异常处理, 341
进程, 46, 366
进程创建, 136
进程地址空间, 44, 229
进程间通信, 103
进程交换器, 46, 194, 280
进程终止, 148
竞争条件, 301
句柄, 129
句柄表, 130
局部描述符表, 182
卷, 499
卷识别, 507

K

空闲进程, 36
空闲线程, 86
快速 I/O, 452
快速互斥体, 370
扩展分区, 501

L

类驱动程序, 425
临界区, 303
零页面线程, 194
逻辑磁盘管理器, 501
逻辑地址, 176
逻辑序列号, 487
逻辑预取器, 191

M

忙等待, 51, 304, 386
门, 546
门对象, 53, 369
命名管道, 577
模仿, 77

N

内存管理单元, 185
内存描述符链表, 279
内存区对象, 45, 241
内存映射 I/O, 386
内核级线程, 102
内核模式, 43, 182, 546
内核日志记录器, 72
内碎片, 186

P

配置管理器, 61
平衡集管理器, 46, 193, 194, 292

Q

抢先式, 48
强制访问控制, 75
亲和性, 48, 116, 150
轻量进程, 102
驱动程序对象, 399
驱动程序检验器, 379
全局唯一标识符, 433
全局描述符表, 182

R

软错误, 258

S

死锁, 305, 378
设备对象, 401
设备列举, 411
设备驱动程序, 25, 383, 388, 422, 427
设备树, 412, 432
设备栈, 392, 403, 432, 461
事件, 52, 362
事件管理器, 73
视图, 245, 478
时限, 163, 643
时钟中断, 327
锁定, 487
守护页面, 258
随机访问存储器, 3

T

提交, 235
提升 IRQL, 51, 343
停机, 75
同步, 51

图形表面, 35, 622
图形化标识和认证, 93
图形设备接口, 33, 603, 620
推锁, 373
突变体, 52, 363

W

外核, 14
外碎片, 186
网络地址转译, 598
网络驱动程序接口规范, 599
稳定的, 66
文件对象, 404, 510
文件分配表, 530
文件系统, 475, 513
文件系统驱动程序, 475, 514, 519
文件系统识别器, 508
文件系统运行库, 366, 521
微内核,14, 20, 23
物理地址, 176
物理地址扩展, 181
物理设备对象, 413
无锁的单链表, 51, 346

X

系统地址空间, 44, 194, 206
系统服务, 545
系统服务分发, 555
系统全局名字空间, 58
系统视图, 198
纤程, 48
线程, 46, 367
线程创建, 140
线程调度, 150
线程终止, 146
陷阱, 314
陷阱处理器, 250
线性地址, 176
向量化异常处理, 341
小端口驱动程序, 425
小过滤驱动程序, 512, 525
写时复制, 44, 258, 263
信号量, 52, 302, 364
行串, 538
性能计数器, 73
修改页面写出器, 194, 274, 280
虚拟地址, 176
虚拟地址描述符, 45, 192

Y

延迟写出器, 496
延迟过程调用, 49, 318, 323
页（页面）, 177
页表, 179
页表项, 179, 251
页面错误, 44, 193, 250
页面错误处理例程, 250
页面老化算法, 189
页面替换算法, 187
页目录, 179
页目录项, 179
页帧, 177
页帧编号, 45, 192, 266
异步过程调用, 49, 330, 381
异常, 48, 336
异常处理器（异常处理例程）, 50
异常分发, 50, 336
易失的（易失性）, 66
引导, 81
引导分区, 82
引导扇区, 82
印迹, 640
硬错误, 258
硬件抽象层, 22
映射页面写出器, 274, 280
用户级线程, 102
用户模式, 43, 182, 546
用户模式调度, 644
邮件槽, 584
优先级反转, 307
优先级提升, 152
预读, 493
原型页表项（原型 PTE）, 252, 255

Z

栈单元, 441, 442
栈交换器, 46, 194, 280
栈帧, 50, 341
值, 61
直接内存访问, 386
直写, 499
执行体, 20
执行体资源, 370
中断, 48, 311
中断对象, 49, 317, 320
中断服务例程, 49, 317, 320
中断控制器, 311
中断描述符表, 49, 312, 546
中断请求级别, 49, 317
注册表, 61
注册表键, 62
注册表值, 62
主分区, 501
主文件表, 536
主引导记录, 82, 501
桌面合成, 627
自旋锁, 51, 304, 349
自主访问控制, 75
总线驱动程序, 391, 423
作业, 48