微软公司核心技术书库

Microsoft

书签

附 赠
CD-ROM

Windows 2000
内部揭密

Inside Microsoft Windows 2000, Third Edition

(美) David A. Solomon
Mark E. Russinovich 著

詹剑锋 张文耀 黄 艳 等译

机械工业出版社
China Machine Press

{ — ° ç Ô s < œ

ì { 2 ! > h Ú t X + h t ÿì U ì Y ' q Ł \ ÿÔ h t ô ! l F '
r Ł ¸ & ì · ÿp ¨ M äh r u , ÿì p Q ' l p » ÿq Ó { h 2 % 8 1 ¸ X
+
§ » öl Ó « · ¸ ì ì ÿ & ö ÷ ì O u ÷ Ñ r : h Ø Ò [v h ß O - ¸ [r : h { ò V Œ
l 2 ì Þ ß O - ÿ ¸ [! . 3 + / Ø Ò [v K [v = l ‹ { { 2 2 C ÿ { l Ó ¢ ‘ ÿ Õ Þ
ì k ¸ À t ! I » ÿq Ó { h ì F O L @ K J A P I B ? R > ÿ " \ î À I ‡ Ò ° Ö N â ì
Þ » W ì F O L @ K J A P À ï f fl ÿ _ w ı ý t »
r ¸ ÿ Æ H { 2 § \ t r ! t b < § h 1 ü + $! t b § { Ò h
ÿw ì ' ì = O L J A P * ì ¨ y Ü ! J A P F A A L D L § h Úh 4 ¶ N _ ðl h r
ÿe v y Y u · ì ÿO M H O A N R A N ? O O = F = T v ß [> h H t ß ì W Ö ø | l
H t ß W ” ª l ¸ y ˇ ˇ » + r W r § ˇ ˇ ' æ ? ì ’ æ q ? Úæ
I ? ì ! ') h § J h I ') Þ s ¥ ÿx Ł ðÓÓ b r k k ‡ à ! e
Æ Úh « J A P ì Þ ˇ h Æ F A A ÿs 7 h Æ = L E { 2 h Ø Ò [v % ‡ [y 4 ì
ÿ ¢ ı r Ł ¸ & ì · ÿM ä l » ÿq Ó { ! ¸ À
´ ¨ J A P F = R = ÿN _ ? t Øß ! | { Ø Ò [v I ° t Õ Þ l » ! f ¸ » ÿh Óh
ý ¨ t ý ! ¨ h ' I B ? J A P p ‹ Ù ¶ Ö . . § k æ Þ h f ÿÚh ° » k Ü t h ! '
r à k à t ! I » Þ t b ï ÿV Œ t b ÿV ' ì J A P ÿ Æ 9 À Æ Ø e Ù ¶
÷ t ÷ ð ¥ ü r ð t b § l t âð Ó ° N _ Ót b ï ÿV Œ Ók Ü t
h ° ýß E ÷ ð ÿh t ¨ ¥ N _ ï < h ° Ó Üï ÿV Œ y W ð À f ÿÚ
Ó { ¾ ÿ – w J Ó ß ÿ » h ß > ÷ Ñ r : O ì ÿt Þ h x 2 C ÿ { J ˇ ⅜ x J
‹ S D W ` u U ÿ ÷ Ñ r : ß > t Ü Ø Ò [v h x J Ül N _ ’ þÿV Œ ï · k » Ø Ò [v
ÿ t Ü | { h { ò V Œ h x b 2 C E ˇ Ó ¸ Ó ¸ Ø h t { ° ò ÿ –
ß > ! ì Þ » Ö N â W ì J A P h F = R = I # b y À Úh # s + h ï · [ÿ ' , — ¸ h w t °
' þ h q h I R ? K N I I A : h Óh À ð t ý § ðh Úl p À h Ú ì Þ Ó «
‹ { h 4 ¶ ’ 2 h Œ ß ^ R I » ì Þ W I k ‡ Öt b E p b
¸ t Ó ¸ þx æ þ ' , P , # 2 6 + * I I ' , / ¨ ß ¥ h k ß > , ÿ » t ß h ì '
p ƒ p ˇ Ö J h h ! & h t ! K ÿ “ ÿ I þ ¸ ÿ ' , Û Ł ¸ F ÿV Œ h Ó f
ÿì ' , H s C h ß ß ÿ « » ï ì Þ h ! I » h Ó Õ Þ e þ ' , f
N â { 2 E ÷ ò ì ! ! [v = L E { 2 h ï · Ø F F h ì fl · Æ ¥ Z V Œ ì Þ ? !
{ 2 » h W ì I B ? J A P I À s 7 5 7 h w f . ß ä Œ ì 4 { 2 § Ó h ì » 2
4 ! ÿ 2 C Œ ß § Ó > e ß > Ó ò ì 4 h 4 ì y h t À h H > h ^ C 4 h ß > ò ì
y h 4 ì 4 h Ł Óh § Ó ’ ì h “ 9 ¸ k

{2 Þs k - h ß> ì J A P I B ? R > I þ¸ ÿ ø Þ ' þ ° - ‰ R h » ÿì - h q
‰ R - t ß t k ° ÿ f ¸ ¥ Àà fi ´ ¯ Æ b § k ! ôô h ¯ b
ôÿ¸ ‹ 4 ¯ h / ß k - I h ô ¾ o y N âk Õ = ÷ Õ âÿ I ¢ U ÿ
- h } À¸ ¨ œ ¯ Y . r k H \ ´ Óÿ p ° . - Ñ] H ' y h n k ˘ k ‹ ÿi
- h æ g k Õ ¯ Æ b ´ Ós ô ÿ » ÿh q ÓÿÖ { 2 Þs ° [h L » Þ !
Ö À H Ôÿh w h # ° h ˘ Ò # 8 q I F ï + t ! ð t ° ¢ ÿt ì h
· m ‡ ÕÞÿ»
I ¥ ÿ ; Õh üÿt f ¥ Z h ¸ k ‡ I & ì · ÿp ¨ h Ý óy ¸ Ö À! ¸ À
ÿ ÿÚ¸ h « I & ì · ÿM äÿø B ł Œ L @ B ¸ ' s ‹ s 6 2 % ¸ ! 8 1 L @ B
“ › D P P L > > O P D A E P D K I A ? K I N A = @ D P I P E @ D P I H
ü I ¸ “ Ø¸ Ó Þ ß ÿ» h t Óh · R { ÿ ' , h Þ» W ì e ' ,
X . ” ª h / ÿ_ À Þÿ
» I q Óx J ì Þï öl Úh ì Þï · E ÷ É! » 2 [ÿÿ ‹ ¸ Ö p ÿ § ì O
Ø { 2 ç – ¸ ö þ¸ “ o ' + Ø ‡ q + ÿ — – p I I p ÿ § t ¤ 8 1 · Þ» h w
· ÿ2 C ¥ ï ï ‡ ¸ ÿ_ h Ýà 6 óI § y h · ÿ2 C + À= † ı ¨ I
§ ? t Þ ! î \ h ¥ ÿ¸ ì ðI \ ô0 ÿ t ¸ I Y å { Œ Ø , ! 4 ôÀ
! ^ ! M Ö Ø ‡ h ¸ œ w Æ Ø ì I § h À ì ÿ . ß Ô ! % I J
‡ y ÿì s ß ÿ»
+ Ø ‡ × § j > > O P D A E P D K I A ? K I
ßO - ì » t — ß ÿ " \ ¨ ÿ { h Ó q -
ì Út p ÿ § h ¨ E § h x J k -
ì Ý × Ö ÿ “ ÿh ì Þh t Àe ^
q + B Æ Ø h v h À ì ÿ . ß Ô
Æ Ø ÿk p ÿ § h q Ö + k Àâ · a î q + Æ Ø f ! ÿ

目　录

序言
艾奇序
前言
第 1 章　概念和工具 …… 1
1.1　基础概念和术语 …… 1
1.1.1　Win32 API …… 1
1.1.2　服务、函数和例程 …… 2
1.1.3　进程、线程和作业 …… 3
1.1.4　虚拟内存 …… 4
1.1.5　内核模式与用户模式 …… 6
1.1.6　对象和句柄 …… 8
1.1.7　安全 …… 9
1.1.8　注册表 …… 10
1.1.9　Unicode …… 10
1.2　深入 Windows 2000 精髓 …… 11
1.2.1　配套光盘中的工具 …… 12
1.2.2　Performance 工具 …… 13
1.2.3　Windows 2000 支持工具 …… 13
1.2.4　Windows 2000 资源包 …… 13
1.2.5　内核调试工具 …… 14
1.2.6　软件开发包平台 …… 16
1.2.7　设备驱动程序包 …… 16
1.2.8　系统内部工具 …… 16
1.3　小结 …… 16
第 2 章　系统体系结构 …… 17
2.1　要求与设计目标 …… 17
2.2　操作系统模型 …… 19
2.2.1　可移植性 …… 20
2.2.2　对称式多处理 …… 20
2.2.3　可伸缩性 …… 21
2.3　总体结构 …… 22
2.4　Windows 2000 的产品包 …… 24
2.4.1　检查链接编译 …… 25
2.4.2　多处理器系统文件 …… 26
2.5　关键系统组件 …… 29
2.5.1　环境子系统与子系统 DLL …… 30
2.5.2　Ntdll.dll …… 37
2.5.3　执行程序 …… 38
2.5.4　内核 …… 39
2.5.5　硬件抽象层 …… 40
2.5.6　设备驱动程序 …… 41
2.5.7　查看没存档的接口 …… 44
2.5.8　系统进程 …… 46
2.6　小结 …… 55
第 3 章　系统机制 …… 56
3.1　陷阱调度 …… 56
3.1.1　中断调度 …… 57
3.1.2　异常调度 …… 71
3.1.3　系统服务调度 …… 76
3.2　对象管理器 …… 79
3.2.1　执行程序对象 …… 80
3.2.2　对象结构 …… 82
3.3　同步 …… 96
3.3.1　内核同步 …… 97
3.3.2　执行程序同步 …… 99
3.4　系统工作者线程 …… 105
3.5　Windows 2000 全局标记 …… 107
3.6　本机过程调用 …… 108
3.7　小结 …… 112
第 4 章　启动与关闭 …… 113
4.1　引导进程 …… 113
4.1.1　引导前的准备 …… 113
4.1.2　引导扇区和 Ntldr …… 115
4.1.3　初始化 Kernel 和执行程序子系统 …… 121
4.1.4　Smss、Csrss 和 Winlogon …… 124
4.2　安全模式 …… 125
4.2.1　在安全模式中加载的驱动程序 …… 125
4.2.2　安全模式意识用户程序 …… 127
4.2.3　安全模式中的引导日志 …… 127
4.3　恢复控制台 …… 128

4.4 关机 …… 130
4.5 系统崩溃 …… 131
4.5.1 Windows 2000 为什么会崩溃 …… 132
4.5.2 蓝色屏幕 …… 132
4.5.3 崩溃转储文件 …… 134
4.6 小结 …… 135
第 5 章 管理机制 …… 136
5.1 注册表 …… 136
5.1.1 注册表数据类型 …… 136
5.1.2 注册表逻辑结构 …… 137
5.1.3 注册表内部 …… 141
5.2 服务 …… 150
5.2.1 服务应用程序 …… 150
5.2.2 服务帐号 …… 154
5.2.3 服务控制管理器 …… 156
5.2.4 服务启动 …… 158
5.2.5 启动错误 …… 161
5.2.6 接受引导和所知最近的正确配置 … 161
5.2.7 服务错误 …… 163
5.2.8 服务关闭 …… 163
5.2.9 共享的服务进程 …… 164
5.2.10 服务控制程序 …… 166
5.3 Windows 管理装置 …… 167
5.3.1 WMI 体系结构 …… 168
5.3.2 提供程序 …… 169
5.3.3 通用信息模型和管理对象格式语言 …… 170
5.3.4 WMI 名字空间 …… 172
5.3.5 类关联 …… 173
5.3.6 WMI 实现 …… 174
5.3.7 WMI 安全性 …… 174
5.4 小结 …… 175
第 6 章 进程、线程和作业 …… 176
6.1 进程的本质 …… 176
6.1.1 数据结构 …… 176
6.1.2 内核变量 …… 183
6.1.3 性能计数器 …… 183
6.1.4 相关函数 …… 184
6.1.5 相关工具 …… 185
6.2 CreateProcess 流程 …… 191
6.2.1 阶段 1：打开要执行的映像 …… 193
6.2.2 阶段 2：创建 Windows 2000 执行程序进程对象 …… 195
6.2.3 阶段 3：创建初始线程及其堆栈和环境 …… 198
6.2.4 阶段 4：向 Win32 子系统通知新进程 …… 198
6.2.5 阶段 5：开始初始化线程的执行 …… 199
6.2.6 阶段 6：在新进程环境中完成进程初始化 …… 199
6.3 线程的本质 …… 200
6.3.1 数据结构 …… 200
6.3.2 内核变量 …… 204
6.3.3 性能计数器 …… 205
6.3.4 相关函数 …… 205
6.3.5 相关工具 …… 206
6.4 CreateThread 流程 …… 207
6.5 线程调度 …… 210
6.5.1 Windows 2000 调度概述 …… 210
6.5.2 优先级 …… 212
6.5.3 Win32 调度 API …… 214
6.5.4 相关工具 …… 214
6.5.5 实时优先级 …… 216
6.5.6 中断级与优先级对比 …… 216
6.5.7 线程状态 …… 217
6.5.8 时间片 …… 218
6.5.9 调度数据结构 …… 220
6.5.10 调度方案 …… 221
6.5.11 环境切换 …… 224
6.5.12 空闲线程 …… 224
6.5.13 提高优先级 …… 224
6.5.14 对称式多处理系统上的线程调度 … 229
6.6 作业对象 …… 233
6.7 小结 …… 236
第 7 章 内存管理 …… 237
7.1 内存管理器组件 …… 237
7.1.1 配置内存管理器 …… 238
7.1.2 检查内存的使用 …… 240
7.2 内存管理器提供的服务 …… 243
7.2.1 保留和提交页面 …… 244

9.5.4 I/O 完成端口操作 …… 386
9.5.5 同步 …… 388
9.6 小结 …… 389
第 10 章 存储管理 …… 391
10.1 Window2000 存储的演化历史 …… 391
10.2 分区 …… 392
10.2.1 基本分区 …… 392
10.2.2 动态分区 …… 393
10.3 存储驱动程序 …… 397
10.3.1 磁盘驱动程序 …… 398
10.3.2 设备命名 …… 398
10.3.3 基本磁盘管理 …… 399
10.3.4 动态磁盘管理 …… 400
10.3.5 磁盘性能监视 …… 402
10.4 多分区卷管理 …… 402
10.4.1 跨越卷 …… 403
10.4.2 带区卷 …… 403
10.4.3 镜像卷 …… 404
10.4.4 RAID－5 卷 …… 406
10.4.5 卷的 I/O 操作 …… 407
10.5 卷名字空间 …… 408
10.5.1 装配管理器 …… 408
10.5.2 装配点 …… 409
10.5.3 卷装配 …… 412
10.6 小结 …… 415
第 11 章 高速缓存管理器 …… 416
11.1 Windows 2000 高速缓存管理器的主要特性 …… 416
11.1.1 单个、集中的系统高速缓存 …… 416
11.1.2 内存管理器 …… 417
11.1.3 高速缓存的一致性 …… 417
11.1.4 虚拟块高速缓存 …… 418
11.1.5 基于流的高速缓存 …… 419
11.1.6 可恢复文件系统支持 …… 419
11.2 高速缓存结构 …… 420
11.3 高速缓存的大小 …… 421
11.3.1 高速缓存的虚拟大小 …… 422
11.3.2 高速缓存的物理大小 …… 422
11.4 高速缓存数据结构 …… 425
11.4.1 系统范围的高速缓存数据结构 …… 425
11.4.2 每个文件的高速缓存数据结构 …… 426
11.5 高速缓存操作 …… 429
11.5.1 写回高速缓存和延迟写 …… 429
11.5.2 智能预读 …… 432
11.5.3 系统线程 …… 433
11.5.4 快速 I/O …… 433
11.6 高速缓存支持例程 …… 435
11.6.1 复制到高速缓存和从高速缓存复制 …… 436
11.6.2 带映射和牵制接口的高速缓存 …… 437
11.6.3 带直接存储器存取接口的高速缓存 …… 438
11.6.4 写入调整 …… 439
11.7 小结 …… 440
第 12 章 文件系统 …… 441
12.1 Windows 2000 文件系统格式 …… 441
12.1.1 CDFS …… 442
12.1.2 UDF …… 442
12.1.3 FAT12、FAT16 和 FAT32 …… 442
12.1.4 NTFS …… 445
12.2 文件系统驱动程序体系结构 …… 445
12.2.1 本机 FSD …… 445
12.2.2 远程 FSD …… 446
12.2.3 文件系统操作 …… 448
12.3 NTFS 设计目标和特性 …… 452
12.3.1 高端文件系统需求 …… 452
12.3.2 NTFS 的高级特性 …… 453
12.4 NTFS 文件系统驱动程序 …… 460
12.5 NTFS 磁盘结构 …… 463
12.5.1 卷 …… 463
12.5.2 簇 …… 463
12.5.3 主文件表 …… 464
12.5.4 文件引用数 …… 469
12.5.5 文件记录 …… 469
12.5.6 文件名 …… 471
12.5.7 驻留和非驻留属性 …… 473
12.5.8 索引 …… 475
12.5.9 数据压缩和稀疏文件 …… 476
12.5.10 重分析点 …… 480
12.5.11 更改日志文件 …… 480

12.5.12 对象 ID …… 481
12.5.13 配额跟踪 …… 481
12.5.14 统一的安全 …… 482
12.6 支持 NTFS 恢复 …… 482
12.6.1 文件系统设计演变 …… 482
12.6.2 日志 …… 484
12.6.3 恢复 …… 488
12.7 NTFS 坏簇恢复 …… 491
12.8 文件系统安全加密 …… 494
12.8.1 注册回调 …… 496
12.8.2 第一次加密文件 …… 496
12.8.3 解密过程 …… 500
12.8.4 备份加密的文件 …… 501
12.9 小结 …… 502
第 13 章 连网机制 …… 503
13.1 OSI 参考模型 …… 503
13.1.1 OSI 层 …… 504
13.1.2 Windows 2000 连网组件 …… 504
13.2 网络 API …… 505
13.2.1 命名管道和邮箱 …… 506
13.2.2 Windows Socket …… 510
13.2.3 远程过程调用 …… 514
13.2.4 通用网络文件系统 …… 517
13.2.5 NetBIOS …… 520
13.2.6 其他网络 API …… 522
13.3 网络资源名字解析 …… 524
13.3.1 MPR …… 525
13.3.2 MUP …… 527
13.3.3 域名系统 …… 528
13.4 协议驱动程序 …… 528
13.5 NDIS 驱动程序 …… 530
13.5.1 NDIS 小端口特征 …… 535
13.5.2 面向连接的 NDIS …… 535
13.6 绑定 …… 538
13.7 分层网络服务 …… 539
13.7.1 远程访问 …… 539
13.7.2 活动目录 …… 539
13.7.3 网络负载平衡 …… 540
13.7.4 文件复制服务程序 …… 541
13.7.5 分布式文件系统 …… 542
13.7.6 TCP/IP 扩展 …… 543
13.8 小结 …… 545
术语表 …… 546

第1章　概念和工具

本章介绍 Microsoft Windows 2000 主要的概念和术语，这些概念和术语将在全章使用，如 Microsoft Win32 API、进程、线程、虚拟内存、核心模式与用户模式、对象、句柄、安全以及注册表。我们还将介绍一些可以用来研究 Windows 2000 内部结构的工具，如 Performance 工具、内核调试程序（kernel debugger）、配套 CD 上的特殊工具和各种附加工具包如 Windows 2000 Support Tools、Windows 2000 调试工具、Windows 2000 资源工具包和 Platform Software Development Kit (SDK)。另外，我们还将介绍如何将 Windows 2000 Device Driver Kit（DDK）作为查看 Windows 2000 内部更进一步信息的资源。

--定要理解本章的所有内容，本书的其余章节是基于已经理解了本章内容。

1.1　基础概念和术语

本书中，我们将提到一些读者可能不熟悉的结构和概念。本部分将定义后面使用的术语。在进入后面章节之前你应该熟悉它们。

1.1.1　Win32 API

Win32 应用程序编程接口（application programming interface—API）是 Microsoft Windows 操作系统系列中的主要编程接口，Microsoft Windows 操作系统系列包括 Windows 2000、Windows 95、Windows 98、Windows Millennium Edition 和 Windows CE。尽管在本书中没有描述 Win32 API，但解释了主要 Win32 API 化数的内部行为和实现。有关 Win32 API 编程的全面指导，参见 Jeffrey Richter 的《*Programming Applications for Microsoft Windows*》（第 4 版，微软出版社，1999）。

不同的操作系统实现不同的 Win32 子集。总的来说，Windows 2000 是所有 Win32 实现的超集。哪些服务在哪些相应平台上实现的规定包括在 Win32 API 的参考文档中。该文档可以从 *msdn.microsoft.com* 免费获得，也可以从 MSDN Library CD－ROM 中获得。该文档中的信息在文件 \ Program Files \ Microsoft Platform SDK \ Lib \ Win32api.csv（一个以逗号分隔的文本文件）中也有详细描述，它作为 Platform SDK 的一部分被安装。Platform SDK 来自 MSDN Professional 或从 *msdn.microsoft.com* 免费下载（参见 1.2.6 节）。

注意：MSDN 表示 Microsoft Developer Network，即供微软的开发商使用的支持程序。它有三个 CD－ROM 预订程序：MSDN 库、MSDN 专业版、MSDN 普及版。MSDN 库的内容也可以在 MSDN Web 免费获得。更为详细的信息，参阅 *msdn.microsoft.com*。

在本书中，Win32 API 指基本的函数集，其内容覆盖了进程、线程、存储管理、安全、I/O、窗口和图形等领域。Win32 API 是 Platform SDK 的一部分。Platform SDK 中其他主要内容，如事务处理、数据库、通信、多媒体和网络服务，不在本书中介绍。

尽管 Windows 2000 被设计为支持多个编程接口，但 Win32 是主要的或首选的操作系统接口。这是因为：在三个环境子系统（Win32、POSIX 和 OS/2）中，它对主要的 Windows 2000 系统服务提供了最大程度的访问。第 2 章将介绍，Windows 2000 的应用程序并不直接调用本机的 Windows 2000 系统服务，而必须利用环境子系统提供的 API。

Win32 API 的历史

有趣地是，Win32 并不是 Microsoft Windows NT 的最初编程接口。因为 Windows NT 是作为第二版 OS/2 的替代品开发的，基本的编程接口是 32 位 OS/2 Presentation Manager API。项目开发的一年后，Microsoft Windows 3.0 开始出击市场并获得成功。结果是，微软改变方向，将 Windows NT 作为 Windows 系列产品的替代品，而不是 OS/2 的替代品。这时，需要确定 Win32 API 的开发方向。在此之前，Windows API 仅开发为 16 位接口。

尽管 Win32 API 引进了许多 Windows 3.1 没有的新函数，微软仍决定使新的 API 在函数名字、语义和数据类型的使用上尽可能与 16 位 Windows API 兼容，以减少将已有的 16 位 Windows 应用程序移植到 Windows NT 的负担。因此有些人第一次查看 Win32 API 时，会对一些函数名与接口的不一致感到迷惑，应该记住不一致的一个原因是保证 Win32 API 与旧的 16 位 Windows API 兼容。

1.1.2 服务、函数和例程

Windows 2000 用户和编程文档中的一些术语在不同的上下文中具有不同的意义。例如：服务（service）可以指操作系统中一个可调用的例程、设备驱动程序或服务器进程。下面列出了本书中某些术语的意义。

■ Win32 API 函数。Win32 API 中文档化的、可调用的子例程。如 *CreatProcess*、*CreatFile* 和 *GetMessage*。

■ 系统服务（或执行程序系统服务）。Windows 2000 操作系统中可以从用户模式调用的本机函数（本机函数的定义，请参见 3.1.3 节“系统服务调度”）。例如，*NtCreatProcess* 是 Win32 *CreatProcess* 函数调用的内部系统服务，它创建新的进程。

■ 内核支持函数（或例程）。Windows 2000 操作系统内核模式（本章后面定义）中的子例程。例如，*ExAllocatePool* 是设备驱动程序调用的例程，它从 Windows 2000 系统堆中分配内存。

■ Win32 服务由 Windows 2000 服务控制管理器启动的进程（尽管注册表将 Windows 2000 设备驱动程序定义为“服务”，但在本书中我们并不这样称呼它们）。例如，Task Scheduler 服务是支持 *at* 命令的用户模式进程（它与 UNIX 命令 *at* 或 *cron* 相似）。

■ DLL（动态链接库）。一系列可调用的子例程连接到一起作为二进制文件，可以由使用子例程的应用程序动态装载。例如 Msvcrt.dll（C 运行时库）与 Kernel32.dll（Win32 API 子系统库之一）。Windows 2000 用户模式组件和应用程序广泛使用 DLL。DLL 相对静态库的优点在于应用程序可以共享 DLL，Windows 2000 保证在引用它的应用程序中，只有一个 DLL 代码的内存副本。

1.1.3 进程、线程和作业

尽管程序与进程表面看起来是相似的，但它们在本质上是不同的。程序是一系列静态指令，而进程是由执行程序实例的线程使用的一系列资源的包容器（container）。从最高抽象层上来说，Windows 2000 进程的组成如下：

- 专用的虚拟地址空间，它是进程使用的一系列虚拟存储地址。
- 可执行程序，它定义初始代码和数据，并映射到进程的虚拟地址空间。
- 各种系统资源的开放句柄列表，如信号量、通信端口和文件，它们对进程的所有线程都是可访问的。
- 标识用户并称为“访问令牌”的安全环境、安全组和同进程相关的特权。
- 称为“进程 ID”（内部称为“客户 ID”）的惟一标识符。
- 至少执行一个线程。

线程是进程中 Windows 2000 调度执行的实体。没有线程，进程的程序就不能运行。线程包括下面的必要组件：

- CPU 寄存器的内容，它表示处理器的状态。
- 两个栈，一个是以内核模式执行时被线程使用，另一个则是以用户模式执行时被线程使用。
- 被称为“本机线程存储区”（TLS）的专用存储区，供子系统、运行时库和 DLL 使用。
- 称为“线程 ID”的唯一标识符（内部也称为“客户 ID”，进程 ID 和线程 ID 产生于不同的名字空间，因此它们不会重叠）。
- 线程有时有自己的被多线程服务器应用程序使用的安全环境，多线程服务器应用程序模拟（impersonate）它们服务的客户的安全环境。

易失寄存器，栈和专用存储区被称为线程的环境。因为这些信息对 Windows 2000 运行的每个机器体系结构来说是不同的，这种结构取决于特定体系结构。实际上，由 Win32 GetThreadContext 函数返回的 CONTEXT 结构是唯一与机器相关的 Win32 API 的公共数据结构。

尽管线程有它们自己的执行环境，但进程中的每个线程都共享进程的虚拟地址空间（加上属于进程的其余资源），这意味着进程中的所有线程都可以写入或读取彼此的内存。除非其他的进程把它的专用地址空间的可用部分作为共享内存区域（在 Win32 API 中称为“文件映射对象”），或者一个进程打开另一个进程并使用 ReadProcessMemory 和 WriteProcessMemory 函数，否则线程不能引用另一个进程的地址空间。

除了一个专用地址空间和一个或更多的线程，每个进程还有一个安全标识符和一个对象的开放句柄列表，这些对象可以是文件，共享内存区域或其中的一个同步对象如互斥，事件或信号量，如图 1-1 所示。

每个进程有一个存储在称为“访问令牌”的对象中的安全环境。进程访问令牌包括安全标识符和进程凭证。默认情况下，线程没有自己的访问令牌，但是它们可以得到一个，这样允许单个线程模拟另一个进程的安全环境（包括运行在远程 Windows 2000 系统上的进程），而不影响进程中的其他线程。（关于进程和线程安全的更多细节参见第 8 章。）

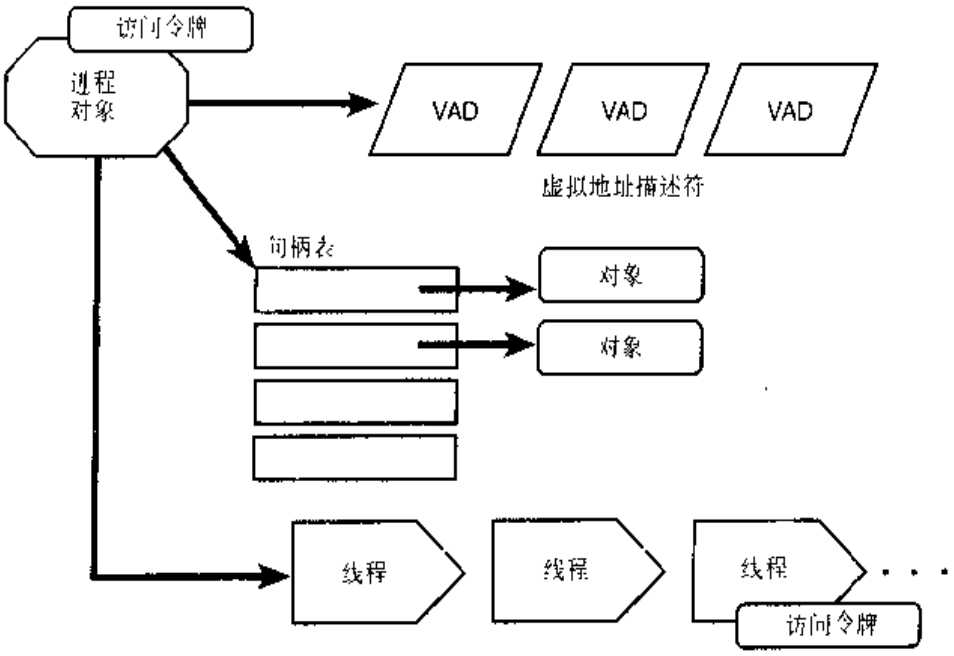


图 1-1 进程和它的资源

虚拟地址描述符（VAD）是内存管理器用于追踪进程正在使用的虚拟地址的数据结构。这些数据结构将在第 7 章进行更深入的讨论。

Windows 2000 引入了进程的扩展模型，称为作业（job）。作业对象的主要功能是允许进程组被作为一个单元来管理和控制。作业对象允许控制一定的属性和提供与作业相关的进程或进程组的限制。作业对象也记录同作业相关的所有进程和同作业相关但已被终止的所有进程的基本说明信息。在某些方面，作业对象弥补了 Windows 2000 结构进程树的缺乏。尽管在很多方面，Windows 2000 结构进程树比 UNIX 型进程树的功能强。

第 6 章将详细介绍作业、进程和线程、进程机制和线程建立、线程调度算法和作业的内部结构。

1.1.4 虚拟内存

Windows 2000 采用基于平面的（线形的）32 位地址空间的虚拟内存系统。32 位地址空间可转化为 4GB 虚拟内存。在大多数系统中，Windows 2000 分配一半地址空间（4GB 虚拟地址空间一半的低位地址空间，从 x00000000 到 x7FFFFFFF）作为进程唯一的专用存储区，而其余的一半（高位地址，从 x80000000 到 xFFFFFFFF）作为被保护的操作系统内存。低位地址的映射改变为反映当前执行进程的虚拟地址空间，但是较高一半的映射总是由操作系统的虚拟内存组成。Windows 2000 Advanced Server 和 Datacenter Server 支持引导时选项（Boot.ini 中的/3GB 限定项），让进程在一个 3GB 专用地址空间（为操作系统留下 1GB）运行被特别标记的程序（大地址空间标记必须在可执行映像的头里设置）。这个选项允许应用程序（如数据库服务器）保留进程地址空间数据库的较大的部分，从而减少了对映射数据库子集视图的需要。图 1-2 显示了两个 Windows 2000 支持的虚拟地址空间布局。

尽管 3GB 比 2GB 更好，但对于映射非常大的（多于千兆字节的）数据库，3GB 仍然不是足够的虚拟地址空间。为满足映射非常大的数据库的需要，Windows 2000 有一个新机制称为地

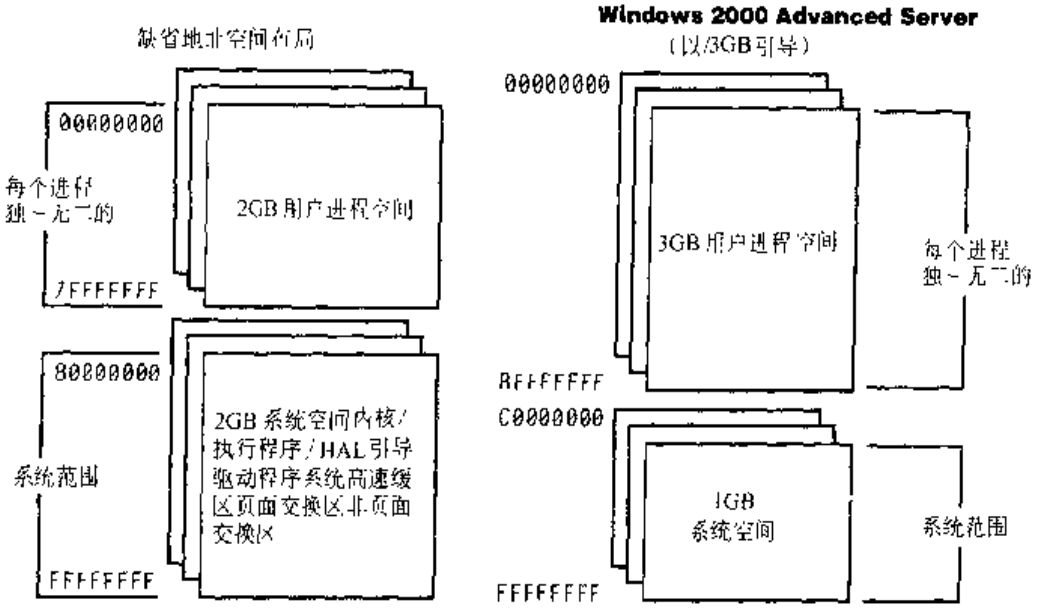


图 1-2 Windows 2000 支持的地址空间布局

址窗口扩展（AWE），它允许一个 32 位应用程序分配高达 64GB 的物理内存，然后将视图或窗口映射到应用程序的 2GB 虚拟地址空间。尽管用 AWE 将管理虚拟内存到物理内存的映射的负担加在程序设计员身上，但它确实立即解决了能直接访问比在 32 位进程地址空间任何时候映射的物理内存还要多的内存需要。地址空间极限的长期解决方案是使用 64 位的 Windows。

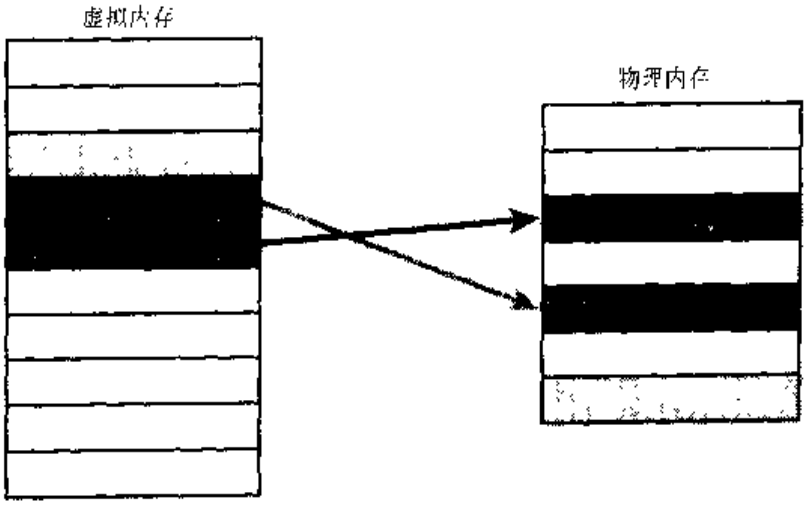


图 1-3 把虚拟内存映射为物理内存

回顾一下，进程的虚拟地址空间是进程的线程使用的可用的地址集。虚拟内存提供了内存的逻辑视图，逻辑视图可能不和它的物理布局相对应。运行时，内存管理器在硬件的帮助下，将虚拟的地址转换或映射到物理地址，数据实际上存储在物理地址中。通过控制保护和映射，操作系统能保证单个的进程不会彼此冲突或覆盖操作系统数据。图 1-3 说明了三个虚拟的连续页映射到物理内存中的三个不连续的页。

因为大多数系统的物理内存比正在运行的进程（每个进程 2GB 或 3GB）使用的总的内存少得多，所以内存管理器会把一些内存的内容转移或页面调出到磁盘。把数据调页磁盘后即可释

放物理内存，这样物理内存就可以用于其他的进程或操作系统本身。当线程访问已经调页到磁盘的虚拟地址时，虚拟内存管理器将信息从磁盘加载回内存。使用调页的优点是应用程序不必进行任何方式的改变，因为硬件支持使得内存管理器进行调页而无须进程或线程的知晓或帮助。

内存管理器实现的细节，包括地址转换是如何工作的和 Windows 2000 是如何管理物理内存的，将在第 7 章进行详细的描述。

1.1.5 内核模式与用户模式

为了避免用户应用程序访问和（或）修改重要的操作系统数据，Windows 2000 采用两种处理器访问模式态（即使正在运行 Windows 2000 的计算机的处理器不止支持两种）：用户模式和内核模式。用户应用程序在用户模式下运行，而操作系统代码（如系统服务和设备驱动程序）则在内核模式中运行。内核模式是指处理器的执行方式，而该处理器授予所有系统内存和所有 CPU 指令访问权限。通过为操作系统软件提供高于应用程序软件的特权级别，处理器就为操作系统设计人员提供了必要的基础，以确保错误应用不会破坏整个系统的稳定性。

注意 Intel x86 处理器体系结构定义了四个级别的权限（或“环”）来保护系统代码和数据，以免被更低级权限代码无意或恶意的重写。Windows 2000 中内核模式的权限级别为 0（或 0 环），而用户模式的权限级别为 3（或 3 环）。Windows 2000 仅仅使用两种权限的原因是过去被支持的体系结构的一些硬件（如 Compaq Alpha 和 Silicon Grahics MIPs）仅实现两级权限。

尽管每个 Win32 进程都有其自身的专用内存空间，但内核模式操作系统和设备驱动程序代码共享同一虚拟地址空间。虚拟地址空间的每一页都有标记以便确定处理器读出或写入该页时的访问模式。系统空间中的页只能用内核模式访问，而用户空间中的所有页都只能从用户模式访问。只读页（如那些包含可执行代码的页）在任何模式下都不可写。

Windows 2000 并没有为在内核模式运行的组件所使用的专用读/写系统内存提供任何保护。换句话说，一旦处于内核模式，操作系统和设备驱动程序代码就拥有对系统空间内存的完全访问，并能绕过 Windows 2000 安全系统而访问对象。由于大量 Windows 2000 操作系统代码在内核模式运行，因此小心地设计和测试在内核模式运行的组件以确保它们不破坏系统安全就显得十分重要。

因此在加载第三方设备驱动程序时应该小心，因为一旦处于内核模式，软件就能完全访问所有的操作系统数据。这一弱点导致了在 Windows 2000 中引入驱动程序信号机制。该机制能在试图添加非授权（非标记）驱动程序时对用户提出警告（关于驱动程序信号的更多信息，参见第 9 章）。称为“驱动程序验证器”的机制也有助于设备驱动程序编写人员查找错误（如内存泄漏）。驱动程序验证器将在第 7 章解释。

正如将要在第 2 章了解到的，用户应用程序在调用系统服务时从用户模式切换到内核模式。如 Win32 ReadFile 函数最终需要调用处理从文件读出数据的 Windows 2000 内部例程，该例程由于访问内部系统数据结构，而必须在内核模式中运行。从用户模式到内核模式的切换是通过使用特殊处理器指令而实现的，该指令导致处理器切换到内核模式。操作系统俘获该指令，

注意到系统服务正在被请求，验证线程传递给系统函数的参数，然后执行内部函数。在将控制返回给用户线程之前，处理器模式再切换回用户模式。以这种方式，操作系统就能保护其自身和其数据以免，被用户进程读取和修改。

注意 从用户模式到内核模式的切换（和返回）不影响本身的线程调度——模式切换并不是环境切换。系统服务调度的更多细节在第 3 章讲述。

因此，用户线程将其部分时间用于在用户模式执行，而将一部分时间在内核模式执行是很正常的。事实上，由于大部分的图形和视窗系统要在内核模式中运行，因此图形密集的应用程序在内核模式花费的时间多于在用户模式中的时间。测试这点的一个简单方法就是运行图形密集的应用程序如 Microsoft Paint 或 Microsoft Pinball，并利用表 1-1 中所列出的任意一个性能计数器观察用户模式和内核模式之间的时间差。

表 1-1 与模式相关的性能计数器

对 象	计数器	函 数
处理器	% Privileged Time	在指定时间间隔内单个 CPU（或所有 CPU）在内核模式运行的时间百分比
处理器	% User Time	在指定时间间隔内单个 CPU（或所有 CPU）在用户模式运行的时间百分比
进程	% Privileged Time	在指定时间间隔内一个进程的线程在内核模式运行的时间百分比
进程	% User Time	在指定时间间隔内一个进程的线程在用户模式运行的时间百分比
线程	% Privileged Time	在指定时间间隔内线程在内核模式运行的时间百分比
线程	% User Time	在指定时间间隔内线程在用户模式运行的时间百分比

实验：内核模式与用户模式

可以利用 Performance 工具来查看系统在内核模式与用户模式所花费的时间。步骤如下：

1）通过打开 Start 菜单并选择 Programs/Administrative Tools/Performance 来运行 Performance 工具。

2）单击工具条上的 Add 按钮（+）。

3）选定处理器性能对象，单击 % Privileged Time 计数器，在按下 Ctrl 键的同时，单击 % User Time 计数器。

4）单击 Add 按钮，然后单击 Close 按钮。

5）快速来回移动鼠标。在来回移动鼠标时，可以观察到 % Privileged Time 线在上升，这反映了花费在服务鼠标中断的时间和花费在视窗系统的图像部分（将在第 2 章解释，主要以设备驱动程序形式在内核模式中运行）的时间（见图 1-4）。

6）结束后，单击工具条上的 New Counter Set 按钮（或关闭该工具）。

读者可以利用任务管理器快速查看该活动。只要单击 Performance 标签，然后从 View 菜单中选择 Show Kernel Times，CPU 使用情况栏就将用户模式时间显示为绿色，而将内核模式时间显示为红色。

为了查看 Performance 工具本身如何使用内核模式时间和用户模式时间，再一次运行它，但为系统中的每个进程添加单独的进程计数器 % User Time 和 % Privileged Time：

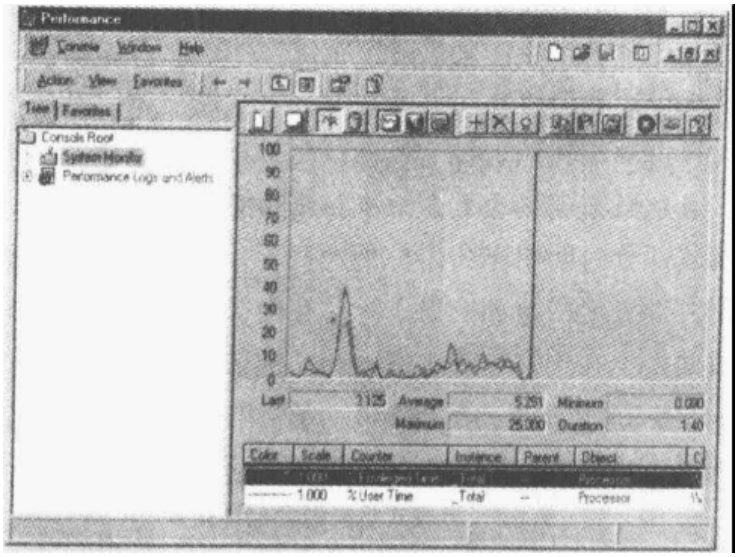


图 1-4 显示内核模式和用户模式时间区别的 Performance 工具

1）如果它还不在运行中，再一次运行 Performance 工具。（如果它在运行，按下工具栏上的 New Counter Set 按钮以一空白显示开始

2）单击工具栏中的 Add 按钮（+）。

3）将 Performance Object 改为进程。

4）选定 % Privileged Time 和 % User Time 计数器。

5）选定 Instance 框中的所有进程（_Total 进程除外）。

6）单击 Add 按钮，然后单击 Close 按钮。

7）快速来回移动鼠标。

8）按下 Ctrl + H 打开高亮度显示方式。这时当前选定的计数器高亮度显示为白色。

9）将计数器滚动到显示屏底端以确定移动鼠标时其线程正在运行的进程，并观察它们是在用户模式还是在内核模式运行。

应该可以看到当移动鼠标时，Performance 工具进程（在 mmc 进程的 Instance 栏中可以看到）内核模式和用户模式时间在上升，因为它在用户模式执行应用代码并调用在内核模式运行的 Win32 函数。在移动鼠标时，还将看到一个称为 csrss 的进程中的内核模式线程的活动。出现该活动是因为 Win32 子系统的内核模式原始输入线程与该进程相联，而该内核模式原始输入线程处理键盘和鼠标输入（对于有关系统线程的更多信息，参见第 2 章）。最后一个进程称为空闲（Idle）进程，正如读者所看到的，几乎将其全部时间花费在内核模式，它实际上是一个用于计算空闲 CPU 周期的伪进程。从空闲进程中线程运行的模式可以观察到，当 Windows 2000 无事可做时，它就在内核模式运行。

1.1.6 对象和句柄

在 Windows 2000 操作系统中，对象是静态定义的对象类型的单个运行时实例。对象类型包括系统定义的数据类型、在数据类型实例上操作的函数以及一组对象属性集。编写 Win32 应用程序时，会遇到要命名的进程、线程、文件和事件对象。这些对象都以 Windows 2000 创建和管

理的低级对象为基础。在 Windows 2000 中，进程就是进程对象类型的实例，文件就是文件对象类型的实例，其他类推。

对象属性是对象中的数据字段，它部分定义了对象状态。例如，进程类型对象的属性包括进程 ID、基本的调度优先级以及指向访问令牌对象的指针。对象方法，即管理对象的方法，通常读入或更改对象的属性。如进程的打开方法接受进程标识符作为输入，并将指针返回给该对象作为输出。

注意　尽管存在一个名为 ObjectAttributes 的参数，当使用 Win32 API 或者本机对象服务创建对象时，它由调用程序提供，但这个参数不应该与本书使用的这个术语的一般意义相混淆。

对象与通常的数据结构之间最基本的区别在于对象的内部结构是隐藏的。要想从对象中取得数据或将数据存放到对象中，就必须调用对象服务，而无法直接读出或更改对象中的数据。这种差别就将对象的基础实现与仅仅使用对象的代码区分开，这种技术可以方便地改变对象实现。

对象为完成下列四种重要的操作系统任务提供了方便的方法：

- 为系统资源提供可读的名字。
- 在进程间共享资源和数据。
- 保护资源以免非授权访问。
- 引用跟踪，它允许系统确知对象什么时候不再使用以便自动被释放。

在 Windows 2000 操作系统中，并不是所有的数据结构都是对象。只有需要被共享、保护、命名或对用户模式程序可见（借助系统服务）的数据才被放置在对象中。仅被操作系统的一个组件用来实现内部函数的结构并不是对象。对象和句柄（对象实例的引用）将在第 3 章详细介绍。

1.1.7　安全

Windows 2000 支持美国国防部计算机安全系统评估标准（DoD 5200.28 - STD，1985 年 12 月）所定义的 C2 级安全。该标准包括对所有可共享系统对象（如文件、目录、进程、线程等等）的保护、安全审计（主体或用户的责任以及他们引发的动作）、登录时的密码验证以及防止用户访问另一用户已释放的未初始化资源（如空闲内存或磁盘空间）。

Windows NT 4 被正式评估为 C2 级，并且是通过美国政府评估的产品（Windows 2000 仍处于评估阶段）。此外，Windows NT 4 达到了欧洲组织 ITSEC（IT 安全评估标准）安全级的 FC2/E3 级（功用级 C2，可靠级 E3，其它仅与 B 级系统相当）。只有达到政府批准的安全级别，操作系统才被允许在该领域中参与竞争。当然，在这些必要性能中，很多都是对任何多用户系统有利的特点。

Windows 2000 有两种访问控制形式。第一种形式——自由访问控制，就是很多人在 Windows 2000 下考虑保护时所想到的保护机制。它是对象所有者（如文件或打印机）授予或拒绝他人访问的方法。当用户登录时，他们被授予安全凭证，或者安全环境。当他们试图访问对象时，就将其安全环境与他们试图访问的对象中的访问控制表进行比较，以确定他们是否被允许

执行该请求操作。

特权访问控制在自由访问控制不够时是必要的。它是确保当主人不在时其他人能访问受保护对象的方法。比如，当雇员离开公司时，管理人员需要通过某种方式取得对那些可能只有该雇员才有访问权的文件的访问。在这种情况下，在 Windows 2000 中，管理人员必须取得文件的所有权，这样才能必要地行使其管理权利。

安全在 Win32 API 接口中十分普遍。Win32 子系统使用与操作系统相同的方式实现，基于对象的安全性。Win32 子系统通过将 Windows 2000 安全描述符放置在共享 Windows 对象中，而实现对它们的保护以防止非授权访问。应用程序第一次试图访问共享对象时，Win32 子系统校验应用程序的访问权限。如果安全检查成功通过，Win32 子系统就允许应用程序继续进行。

Win32 子系统在很多共享对象上都实现对象安全，其中一些建立在本机 Windows 2000 对象的最顶端。Win32 对象包括桌面对象、窗口对象、菜单对象、文件、进程、线程以及一些同步对象。

对于 Windows 2000 安全性的详细描述，参见第 8 章。

1.1.8 注册表

如果使用过 Windows 操作系统，你可能听说或看过注册表。不涉及注册表，就无法讨论 Windows 2000 精髓，因为注册表是系统数据库，该数据库包含启动和配置系统所必需的信息、控制 Windows 2000 操作系统范围内的软件设置、安全数据库以及每个用户的配置设置（如选用的屏幕保护程序）。

此外，注册表也是内存中易失数据的窗口，易失数据如系统的当前硬件状态（所加载的设备驱动程序和它们所使用的资源等等）和 Windows 2000 性能计数器。性能计数器事实上并不在注册表中，它是通过注册表函数而被访问的。关于如何从注册表访问性能计数器信息的细节论参见第 5 章。

尽管很多 Windows 2000 用户和管理人员都不需要直接查看注册表（因为可以利用标准管理实用程序来查看或更改大多数配置设置），但注册表仍是 Windows 2000 内部信息的有用资源，因为它包含了很多能影响系统性能和行为的设置（直接修改注册表设置必须十分小心。任何改动都可能对系统性能产生负面影响，甚至会更糟糕，导致系统无法正常启动）。贯穿本书始终，读者将发现要涉及到单个注册表键，它们与所描述的组件有关。本书中所涉及的大部分注册表键都在 HKEY _ LOCAL _ MACHINE 之下，以后一律简写为 HKLM。

有关注册表和其内部结构的进一步信息，参见第 5 章。

1.1.9 Unicode

Windows 2000 和其他大多数操作系统的区别在于它的大部分内部文本串以 16 位的统一编码标准字符的形式被保存和处理。Unicode 是国际字符集标准，它为大多数世界已知字符集定义了唯一的 16 位值（有关 Unicode 的更多信息，浏览 www.unicode.org 站点或参考 MSDN 库中的编程文档）。

由于很多应用程序处理 8 位（单字节）ANSI 字符串，因此接受字符串参数的 Win32 函数

有两个入口点：Unicode（宽 16 位）和 ANSI（窄 8 位）版本。Windows 95、Windows 98 以及 Windows Me 的 Win32 都没有实现将所有 Unicode 用于所有 Win32 函数，因此设计在这些操作系统以及 Windows 2000 上运行的应用程序通常使用窄版本。如果调用 Win32 函数的窄版本，在系统处理之前，输入的字符串参数被转换成 Unicode，并在返回给应用之前将输出参数从 Unicode 转换成 ANSI。因此，如果有旧的服务程序或代码段要在 Windows 2000 中运行，而该代码是用 ANSI 字符文本串编写的，那么 Windows 2000 将把 ANSI 字符转换为 Unicode 来供使用。然而 Windows 2000 从不对文件中的数据进行转换——它由应用程序来决定是否将数据保存为 Unicode 还是 ANSI。

在 Windows NT 的早期版本中，亚洲和中东版本是美国和欧洲版本的核心扩充版本，它们含有一些额外的 Win32 函数用来处理更复杂的文本输入和布局要求（如从右到左的文本输入）。在 Windows 2000 中，各种语言版本都包含相同的 Win32 函数。虽然没有独立语言版本，但 Windows 2000 中却存在单独的国际二进制编码使得单独安装能支持多种语言（通过添加语言包）。应用程序也能利用 Win32 函数，这些 Win32 函数允许支持多种语言的单独的国际应用程序二进制。

1.2 深入 Windows 2000 精髓

尽管本书中的很多信息是以 Windows 2000 源代码为基础的，但读者不必熟记所有信息。很多有关 Windows 2000 内部的细节都可利用各种可用工具显示出来，这些工具包括 Windows 2000 中自带的，Windows 2000 支持工具、Windows 2000 资源包以及 Windows 2000 调试工具。这些工具包将在本节后文进行简单介绍。

为深入 Windows 2000 内部，我们在本书中还包括了“实验”工具条，它们描述在考察 Windows 2000 内部行为的某个特定方面时应采取的步骤（已在前面的一些章节中涉及到）。建议你试着做做这些实验，以便看到本书中所讲的很多主题的运行情况。

此外，本书附带一张 CD－ROM，其中包含有来自 www.sysinternals.com（是个 32 位 Windows 内部相关工具和信息的大众网站）的工具的最新版本以及只有本书才出现的工具。

表 1-2 列出了本书中所使用的工具及其来源。尽管从它们所能显示的信息来看，这些工具中很多的功能相互重叠，但是每个工具都至少显示一条其他任何实用程序都没有的独一无二的信息。

表 1-2 查看 Windows 2000 内部的工具

工 具	映像名	来 源
Dependency Walker	DEPENDENS	Support Tools，Platform SDK
Dump Check	DUMPCHK	Support Tools，调试工具、Platform SDK、Windows 2000 DDK
EFS Information Dumper	EFSDUMPV	www.sysinternals.com
File Monitor	FILEMON	www.sysinternals.com

（续）

工　具	映像名	来　源
Get SID 工具	GETSID	资源包
Global Flags	GFLAGS	Support Tools、Platform DDK、Windows 2000 DDK
Handle/DDL	HANDLEEX	www.sysinternals.com
Viewer	NTHANDLE	
Junction 工具	JUNCTION	www.sysinternals.com/misc.htm
Kernel	I386KD	调试工具、platform SDK
调试程序	WINDBG，KD	Windows 2000 DDK
Object Viewer	WINOBJ	Platform SDK、www.sysinternals.com
Open Handle	OH	资源包
Page Fault Monitor	PFMON	资源包、Platform SDK
Performance 工具	PERFMON	Windows 2000
PipeList 工具	PIPELIST	www.sysinternals.com/tips.htm
Pool Monitor	POOLMON	Support Tools Windows 2000 DDK
Process Explode	PVIEW	www.reskit.com
Process Statistics	PSTAT	Platform SDK，www.reskit.com
Process Viewer	PVIEWER（在 Support Tools 中） PVIEW（在 Platform SDK）	Support Tools，Platform SDK
Quantum	QUANTUM	配套 CD
Quick Slice	QSLICE	资源包
Register Monitor	REGMON	www.sysinternals.com
Service Control 工具	SC	资源包
Task（Process）List	TLIST	Support Tools
Task Manager	TASKMAN	Windows 2000
TDImon	TDIMON	www.sysinternals.com

注：① 所有可以从 www.sysinternals.com 获得的工具也包含在配套光盘中

1.2.1 配套光盘中的工具

配套光盘含有如下有助于深入 Windows 2000 内部的独特工具：

■ LiveKd　该工具允许使用标准 Microsoft 内核调试程序，I386kd.exe 和 Windbg.exe（以及新的 Kd.exe，在调试工具的新版本中它取代了前两种工具），以便从当前运行系统中显示内部信息，而不需第二台计算机充当主机（通过空值调制解调器电缆）。该工具将在本章后文的“内核调试工具”一节中讲述。

■ 内核变量性能计数器扩展 DDL 该 Windows 2000 Performance 工具的扩展允许检查从核心内核映像 Ntoskrnl.exe 中输出的任何内核变量值。

本书中的很多实验使用内核调试程序，因为它能轻易地显示很多 Windows 2000 内部数据结构以及不能从任何用户模式实用程序获得的其他细节。因此，LiveKd 使得做这些实验更容易，因为它允许在实时系统中使用内核调试程序，而不需第二台计算机。

而 LiveKd 显示内部内核变量，内核变量性能计数器扩展 DLL 在该期间内监测这些变量的值。例如，这些变量可以包含一些有趣的数值，而它们可能无法使用任何 Windows 2000 性能计数器访问。

有关这些工具的更多信息，参见光盘中提供的参考文档，它们也是工具安装的一部分。注意，只有购买了本书的人才能安装和使用这些工具。它们不能进一步发布（有关细节参见本书附录后面的许可协议）。

1.2.2 Performance 工具

在本书中，我们将讨论 Performance 工具，它能在启动菜单（或通过控制面板）上的管理工具文件夹中找到。Performance 工具有三种作用：系统监测、查看性能计数器日志以及设置警告。

Performance 工具提供的有关系统如何运行的信息比其他任何单个的实用程序提供的要多。它包括数百个不同对象的计数器。对于本书所讲述的每个主题，都包括有关 Windows 2000 性能计数器列表。

Performance 工具含有每个计数器的简单描述。为查看这些描述，选定 Add Counter 窗口中一个计数器并单击 Explain 按钮。或者是打开资源包中的 Performance Counter Reference 帮助文件。关于如何解释这些计数器以检测瓶颈问题或规划计算效率，参见 Windows 2000 服务器操作指南中的“性能监测”一节，它是 Windows 2000 服务器资源包的一部分。这些章节为那些对 Windows 2000 性能有兴趣的读者提供了很好的解释。

注意所有 Windows 2000 性能计数器都可通过程序访问。第 5 章“HKEY_PERFORMANCE_DATA”一节中简要的讲述了涉及到通过 Win32 API 获取性能计数器的组件。

1.2.3 Windows 2000 支持工具

Windows 2000 支持工具包括大约 40 种工具，它们对于 Windows 2000 系统的管理和故障的查找排除十分有用。这些工具中很多都是 Windows NT 4 资源包的正式部分。

通过运行任何 Windows 2000 产品发布 CD 上 \ Support \ Tools 文件夹中的 Setup.exe 文件，可以安装支持工具（也就是说，支持工具在 Windows 2000 Professional 、Server 和 Advanced Server 上都是相同的）。

1.2.4 Windows 2000 资源包

Windows 2000 资源包是对支持工具的补充，它另外增加了大约 200 种工具。除了很多对显示内部系统状态十分有用的工具外，它们还包括有用的 Windows 2000 内部参考文档，如注册表

索引和性能计数器帮助文件。

资源包有两种版本：Windows 2000 Professional 资源包和 Windows 2000 Server 资源包。尽管后者是前者的扩展集且能够安装在 Windows 2000 Professional 系统上，但在本书实验中，没有任何一个使用仅存在于 Windows 2000S Server 源包中的工具。注意浏览 www.reskit.com 以便更新工具并获得一些新工具。

1.2.5 内核调试工具

内核调试程序是设备驱动程序开发人员用来调试其驱动程序、维护个人使用以查找并排除悬挂系统故障以及检查崩溃转储（存储在能被分析以确定系统崩溃原因的文件中的系统内存拷贝）的工具。尽管内核调试程序主要用来分析崩溃转储或调试设备驱动程序，但它也是研究 Windows 2000 内部情况的有用工具，因为它能显示任何标准实用程序所不可见的 Windows 2000 内部系统信息。例如，它能转储线程块、进程块、页面表、I/O 以及库结构等内部数据结构。本书中，有关内核调试程序命令和输出包括在它们所应用的每个讨论主题中。

1. Microsoft 内核调试程序

Microsoft 内核调试程序有两个版本：命令行版本（用于 x86 系统的 I386kd.exe[○]）和图形用户界面（GUI）版本（Windbg.exe）。另有一新版本 Kd.exe，它可代替以上两种版本。这些工具是调试程序工具包的一部分，在三个不同的地方发行：

- Windows 2000 Customer Support Diagnostics（可从 www.microsoft.com 站点下载）
- Platform SDK（Professional 和 Universal 版 MSDN 的一部分，可从 msdn.microsoft.com 下载）
- Windows 2000 DDK（设备驱动程序包——也是 MSDN 的一部分，可从 www.microsoft.com/hwdev 站点免费下载）。

注意 调试工具包的新版本和 Windows 2000 新版本的发行彼此独立。因此，应不时看看 Microsoft 网站下载区，以获得这三个软件包的最新版本。调试工具包括了一个称为 OEM Support Tools 的工具包，其更新独立于调试工具，因此可能有与调试相关工具（如 Kdex2x86.dll，它是一个具有额外调试命令的内核调试程序扩展 DLL）的更新版本。

调试工具帮助文件（上述三个软件包都提供）解释如何安装和使用内核调试程序（以及其他作为软件包中一部分的调试和支持工具）。关于如何使用主要面向设备驱动程序编写人员的内核调试程序的其他细节，可参见 Windows 2000 DDK 参考文档。此外还有一些有用的关于内核调试程序的知识库文章。从 www.support.microsoft.com 的 Windows 2000 知识库（在线技术文章数据库）中搜索“debugref”即可看到。

内核调试程序具有两种操作方式：

- 打开由于 Windows 2000 或 Windows NT4 系统崩溃而创建的崩溃转储文件（有关崩溃转储的更多知识，参见 4.5% “系统崩溃”）。

○ 即使 Windows 2000 不在 Intel 80386 处理器上运行（Windows NT 的早期版本在其上运行），由于历史原因，Windows 2000 发布媒体上的 x86 目录仍称为 i386，因此，x86 内核调试程序称为 I386kd.exe。

■ 连接到实时运行系统上并检查系统状态（或设置断点，如果在调试设备驱动程序代码）。该操作需要两台计算机——目标计算机和主机。目标计算机是被调试系统，主机是运行调试程序的系统。目标系统可以是本机的（通过空值调制解调器连接到主机上）或远程的（通过调制解调器连接到主机上）。目标系统必须带/DEBUG 限定词启动（可通过在启动过程中按下 F8 并选择调试模式，也可通过在 C：\ Boot.ini 入口中添加启动选项）。

详细的安装指令，可见前面所引用的调试工具参考文档。

2. LiveKd 工具

配套光盘中含有一个称为 LiveKd 的工具，它允许使用实时系统上的标准 Microsoft 内核调试程序，而不需两台计算机。LiveKd 能用于本书中的大多数实验，因此是深入 Windows 2000 内部的有用工具。

可以和运行 I386kd、Windbg 或 Kd 一样运行 LiveKd。LiveKd 将指定的命令行选项传递给所选择的调试程序。在缺省情况下，LiveKd 运行新的命令行内核调试程序（Kd）。如果 Kd 不在当前目录中，LiveKd 就试着运行 I386kd。要运行 GUI 调试程序（Windbg），应使用 - W 开关。要查看有关 LiveKd 开关的帮助，应使用 - ? 开关。

LiveKd 为调试程序提供了模拟崩溃转储文件，因此可以在 LiveKd 中执行任何崩溃转储所支持的操作。由于 LiveKd 需要物理内存来支持模拟转储，因此内核调试程序可能会陷入这样的情况，即数据结构会被系统改变，而且前后不一致。每次启动调试程序时，它都会取得系统的瞬间状态，因此，如果想刷新该瞬间状态，只要退出调试程序（用“q”命令），LiveKd 将询问是否重新启动调试程序。如果调试程序进入了打印输出循环，按下 Ctrl + C 以中断输出，退出并再次运行它。如果它已挂起，按下 Ctrl + Break，它将会终止调试程序进程并询问是否需要重新运行调试程序。

3. SoftICE

对于实时内核调试，另一个不需两台机器的调试工具是称为 SoftICE 的第三方内核调试程序，它可从 Compuware NuMega 买到（有关细节浏览 www.numega.com 站点）。

4. 内核调试符号

为利用前述内核调试工具以深入研究 Windows 2000 数据结构（如进程列表、线程块、加载驱动程序列表、内存使用信息等等），必须至少要有内核映像的正确符号文件，Ntoskrnl.exe（第 2 章“体系结构概述”一节详细地解释了该文件）。符号是客户支持诊断包（正如前述，它可从 www.microsoft.com 下载）的一部分。其安装独立于调试工具，在缺省时存在于 \ Winnt \ Symbols 文件夹中。

在本书后面，读者将看到如何使用这些符号表文件来显示 Windows 2000 内部系统例程的名字以及全局变量。

注意 符号表文件必须与它们所取自的映像版本匹配。如，如果安装了 Windows 2000 服务包，就必须至少要获得与内核映像相匹配的、更新了的符号文件；否则的话，在试图将它们与调试程序一同加载时，会出现校验和错误。这些更新符号文件在从 www.microsoft.com 下载和安装服务包时，通常并不包括在其中或被安装——它们必须单独下载（如果有 MSDN 专业版或 TechNet，它们则包括在服务包 CD - ROM 中）。

1.2.6 软件开发包平台

软件开发包（SDK）平台是 MSDN 专业版（和通用版）预定的一部分，且能从 msnd.microsoft.com 站点免费下载。它含有编译和链接 Win32 应用程序所必需的 C 头文件和库（尽管 Microsoft Visual C++有这些头文件的拷贝，但 SDK 平台中的版本通常与 Windows 操作系统最新版本相匹配，而 Visual C++中的版本可能是较老的版本，虽然在 Visual C++发行时是最新版本）。从内部的角度来说，SDK 平台中有趣的内容包括 Win32 API 头文件（\Program Files\Microsoft Platform SDK\Include）以及一些实用程序（Pfmon.exe、Pstat.exe、Winobj.exe）。SDK 平台中的一些工具还有资源包。其中的一些工具还以 SDK 平台和 MSDN 库中实例源代码的形式发行。

1.2.7 设备驱动程序包

Windows 2000 设备驱动程序（DDK）是 MSDN 专业版（和通用版）预定的一部分，但也可从 www.microsoft.com/hwdev 站点免费下载。尽管 DDK 是面向设备驱动程序开发人员的，但它是 Windows 2000 内部信息的丰富源泉。如，DDK 参考文档同时以教学形式和参考形式给出了 Windows 2000 I/O 系统的全面描述，包括设备驱动程序所使用的内部系统例程和数据结构。

除了参考文档外，DDK 还含有定义关键内部数据结构和常数以及很多内部系统例程的接口的头文件（特别是，Ntddk.h）。这些文件在利用内核调试程序研究 Windows 2000 内部数据结构时十分有用，因为尽管书中给出了这些结构的一般布局和内容，但并没有给出它们在字段层次上（如大小和数据类型）的详细描述。然而，这些数据结构中的很多（如对象调度程序头、等待块、事件、互斥、信号量等等）都没有在 DDK 中完整定义。此外，内核调试程序中的!dso 命令可以显示 DDK 头文件中所没定义的很多 Windows 2000 内部数据结构的格式。

1.2.8 系统内部工具

本书中的很多实验使用免费工具，它们可从 www.sysinternals.com 站点下载。本书的作者之一 Mark Russinovich 编写了大多数的工具。这些工具的拷贝存在于配套 CD 的\Sysint 目录中。另外，CD 中也有 www.sysinternals.com 网站的完整拷贝（注意：尽管配套 CD 中的 www.sysinternals.com 版本有本书出版时这些工具的最新版本，但却没有后来添加的更新了的工具）。这些实用程序中，很多需要安装和执行内核模式设备驱动程序，因此需要管理特权。

1.3 小结

本章介绍了 Windows 2000 中一些重要的技术概念和术语，这些都将在本书中用到。你也大致地了解了对于深入 Windows 2000 内部十分有用的工具。现在我们将着手研究系统的内部设计，在此之前将概述一下系统体系结构和其重要组件。

第2章　系统体系结构

前面已经讲述了你需要熟悉的术语、概念和工具，现在开始剖析 Microsoft Windows 2000（原来的 Windows NT）的内在设计目标和结构。本章将阐述系统的总体结构——主要组件、它们如何相互作用，以及它们运行的环境。为了给出一个框架以理解 Windows 2000 的内部情况，我们首先回顾一下构成系统最初设计和规范的要求与目标。

2.1　要求与设计目标

下述要求于 1989 年推动了 Windows NT 规范的形成：

- 提供一个真正 32 位、抢占式、可重入的虚拟内存的操作系统。
- 在多种硬件结构和平台上运行。
- 在对称式多处理系统上很好地运行、具有良好的伸缩性。
- 是优秀的分布式计算平台，既可以作为网络客户机也可以作为服务器。
- 运行大多数现有的 16 位 MS－DOS 和 Microsoft Windows 3.1 应用程序。
- 满足政府对 POSIX 1003.1 一致性（compliance）的要求。
- 满足政府与业界对操作系统安全性的要求。
- 通过支持 Unicode，适应全球市场的要求。

为了创建一个满足以上要求的系统，就必须作出众多决策，为了统一这些决策，Windows NT 设计小组在项目开始时采用了下面的设计目标：

- 可扩展性（Extensibility）代码必须易于扩充并根据市场需求变化而改变。
- 可移植性（Portability）系统必须能在多种硬件体系结构上运行，必须能够较容易地移植到市场要求的新硬件上。
- 可靠性与健壮性（Reliability and robustness）系统必须保护自己免受内部故障和外部破坏，应用程序不能危害操作系统和其他应用程序。
- 兼容性（Compatibility）尽管 Windows NT 应该扩展现有的技术，但它的用户界面和 API 应该与老版本的 Windows 和 MS－DOS 兼容。它还应与其他系统如 UNIX、OS/2 和 NetWare 很好地交互操作。
- 性能（Performance）在满足其他设计目标的情况下，系统应在每一硬件平台上尽可能的快速地运行和响应。

在我们剖析 Windows 2000 内部结构和操作细节时，你会明白这些最初的设计目标和市场要求是如何成功地融入系统的实现的。在我们开始剖析之前，先考查一下 Windows 2000 的整体设计模型并将它与其他现代操作系统作一比较。

Windows 2000 与 Consumer Windows

Windows 2000 与 Consumer Windows（Windows 95、Windows 98 和 Windows Millennium Edition）是“Windows 操作系统系列”的一部分，共享同一子集 API（Win32 和 COM），并在一些情况下共享操作系统代码。Windows 2000、Windows 98 与 Windows Millennium Edition 还共享同一被称为“Windows 驱动程序模型”（Windows Driver Model，WDM）的子集设备驱动程序模型，这一模型将在第 9 章中详细说明

从 Windows NT 宣布问世起，微软就一直声明这一操作系统是未来的战略平台——不仅对服务器与商业桌面，最终也是用户系统的桌面。下面列出的几点突出了 Windows 2000 相对于 Consumer Windows 的一些结构上的不同和优点。

■ Windows 2000 支持多处理器系统-- Consumer Windows 做不到。

■ Windows 2000 的文件系统支持安全性（如自由存取控制），Consumer Windows 文件系统做不到。

■ Windows 2000 是一个完全的 32 位操作系统——它不包含 16 位的代码，除了支持运行 16 位 Windows 应用程序的代码。Consumer Windows 包含了早期产品（Windows3.1 和 MS - DOS）的大量老的 16 位代码。

■ Windows 2000 是完全可重入的（reentrant） —— Consumer Windows 的重要部分是不可重入的（主要是来自 Windows 3.1 的 16 位代码）。这种不可重入的代码包括大多数的图形和窗口管理函数（GDI 与 USER）。当 Consumer Windows 中的 32 位应用程序试图调用通过不可重入的 16 位代码实现的系统服务时，应用程序必须首先获得一个全系统的锁定（或互斥）来阻塞其他线程进入不可重入的代码库。更糟的是，16 位的应用程序在运行时保持这一锁定。结果是，虽然 Consumer Windows 的内核拥有抢占式的 32 位的多线程调度程序，但应用程序经常以单线程运行，原因是系统的大部分仍旧通过不可重入的代码实现。

■ Windows 2000 允许 16 位的 Windows 应用程序在自己的地址空间上运行，而 Consumer Windows 始终在共享的地址空间上运行 16 位的 Windows 应用程序，这样它们会相互破坏和挂起。

■ Windows 2000 中的共享内存只对映射同一共享内存区域的进程是可见的（在 Win32 API 中，共享内存区域称为“文件映射对象”）。在 Consumer Windows 中，所有共享内存对所有进程是可见并可写的。这样，任一进程都可以写到任意的文件映射对象中。

■ Consumer Windows 的一些关键操作系统页从用户模式是可写的，这样就允许用户应用程序来破坏系统或使系统崩溃。

Consumer Windows 能做但 Windows 2000 做不到的一件事情是运行所有早期的 MS - DOS 和 Windows 3.1 应用程序（尤其是需要直接访问硬件的应用程序）以及运行 16 位的 MS - DOS 设备驱动程序。100%与 MS - DOS 和 Windows 3.1 兼容是 Windows 95 的一个强制性目标，*Windows NT* 的最初目标是运行大多数现有的 16 位应用程序，并维护系统的完整性与可靠性

2.2 操作系统模型

在大多数多用户操作系统中，应用程序与操作系统本身是分离的——操作系统代码以处理器特权态运行（本书称为内核模式），能够访问系统数据和硬件；应用程序代码以处理器非特权模式运行（称为用户模式 *user mode*），具有有限的可用接口，有限的访问系统数据，不能直接访问硬件。当用户模式的程序调用系统服务时，处理器俘获该调用并将调用线程切换为内核模式。当系统服务完成后，操作系统将线程环境切换为用户模式，并允许调用者继续进行。

Windows 2000 与大多数 UNIX 操作系统是相似的，大量操作系统和设备驱动程序代码共享同一由内核模式保护的内存空间，它们是单一操作系统。这意味着任一操作系统组件或设备驱动程序能够潜在地破坏正在被其他操作系统组件使用的数据。

Windows 2000 是基于微内核的系统吗？

尽管有些宣传如此，但就微内核的典型定义而言，Windows 2000 不是基于微内核的操作系统，微内核系统是指主要的操作系统组件（如存储管理器，进程管理器，I/O 管理器）作为分开的进程在它们各自的地址空间上运行，并建立在微内核提供的服务原始集之上。例如，Carnegie Mellon 大学的 Mach 操作系统是一个当代的微内核结构的例子，它实现由线程调度、信息传递、虚拟内存和设备驱动程序构成的微小内核。其他的一切，包括不同的 API、文件系统和网络则以用户模式运行。但是，Mach 微内核操作系统的商业实现至少要以内核模式运行所有的文件系统、网络及存储管理代码。原因很简单：纯粹的微内核设计在商业上是不切实际的，因为它效率太差。

Windows 2000 的大部分以内核模式运行的事实是否意味着它比纯粹的微内核操作系统更易于崩溃呢？不是这样，考虑下面的情况：假定操作系统的文件系统代码有一个错误，使得系统不断崩溃。在传统的操作系统或改进的微内核操作系统中，内核模式代码的错误，如存储管理器或文件系统的错误可能使整个操作系统崩溃。在纯粹的微内核操作系统中，这些组件以用户模式运行，因此理论上一个错误只意味着退出该组件的进程。但实际上，系统会崩溃，因为恢复如此关键进程的失败简直是不可能的。

当然，所有这些操作系统组件是不会被错误程序破坏的，因为应用程序无法直接访问操作系统中具有特权的代码和数据（尽管它们可以迅速调用其他内核服务）。从内核操作系统服务角度来看，如虚拟内存管理、文件 I/O、网络、文件和打印共享，这一保护是使 Windows 2000 作为应用程序服务器和工作站平台拥有健壮、稳定、快速、敏捷等优点的原因之一。

Windows 2000 的内核模式组件也体现了面向对象设计的基本原则。例如，它们不能进入其他数据结构来访问个别组件拥有的信息。但是，它们通过规范的接口来传递参数并访问或修改数据结构。

尽管 Windows 2000 广泛使用对象来表示共享系统资源，但它并不是严格意义上的面向对象的系统。由于 C 的开发工具广泛应用并且 C 语言代码具有可移植性，操作系统的大部分代码是用 C 编写的。C 并不直接支持面向对象的结构，如数据类型的动态绑定、多态函数或类继承。因此，Windows 2000 中基于 C 的对象实现是借用（而不是依赖）面向对象语言的特征。

2.2.1 可移植性

Windows 2000 是为在各种硬件上运行而设计的，包括基于 Intel 的 CISC 系统和 RISC 系统。最初发行的 Windows NT 支持 x86 和 MIPS 结构。对 Digital Equipment Corporation（DEC）的 Alpha AXP 的支持是后来添加的。为支持第四代处理器结构，Motorola PowerPC 被添加到 Windows NT 3.51 中。由于市场需求的变化，在开发 Windows 2000 前，对 MIPS 和 PowerPC 的支持已经下降。后来，Compaq 取消对 Alpha AXP 结构的支持，结果使得 Windows 2000 只被 x86 结构支持

注意 Windows 2000 下一版本支持的是新的 Intel Itanium 处理器家族，是由 Intel 和 Hewlett - Packard 联合开发的，首次实现 64 位体系结构家族，称为 IA - 64（Intel Architecture 64）。Windows 的 64 位版本将为用户进程和系统提供更大的地址空间。尽管这在明显提高系统可伸缩性方面是一个重要步骤，但是到目前为止，将 Windows 2000 移植到 64 位平台上并不需要大大改变系统内核体系结构（当然，存储管理器的支持除外）。使应用程序更方便地移植到 64 位 Windows 中这方面的信息请参阅 Platform SDK 部分的资料，标题为“Win64 Programming Preview”（也可在线查看 *msdn.microsoft.com*）。为获得 64 位 Windows 的一般信息，可在 www.microsoft.com/windows 上查关键词“64 - bit”。

Windows 2000 通过两种主要方式在硬件体系结构和平台上实现可移植性：

■ Windows 2000 使用分层设计，系统的低层部分，即处理器体系结构或平台被分成独立的模块。这样系统的高级层可以被屏蔽在千差万别的体系结构和硬件之外。为操作系统提供可移植性的两个主要组件是内核（包含在 Ntoskrnl.exe 中）和硬件抽象层（包含在 Hal.dll 中）（这两个组件将在本章后面详细介绍）。体系结构相关的函数（如线程环境切换（context switching）和陷阱调度）在内核中实现。在同一体系结构内（如不同主板），随系统而不同的函数在 HAL 中实现。

■ Windows 2000 的大部分是用 C 编写的，部分用 C + + 编写。汇编语言仅仅用在操作系统的某些部分，如需要直接与系统硬件通信（如中断陷阱处理程序）的部分或性能特别敏感部分（如环境切换）。汇编语言代码不仅存在于内核与 HAL 中，还存在于内核操作系统的某些其他地方（如实现 interlock 指令的例程，本机过程调用功能的模块），在 Win32 子系统的内核模式部分，甚至在一些用户模式库中，如 Ntdll.dll 中的进程启动代码（Ntdll.dll 系统库将在本章后面介绍）。

2.2.2 对称式多处理

多任务是多个执行线程共享一个处理器的操作系统技术。当计算机有多个处理器时，它可以同时执行两个线程。多任务操作系统只是看起来像在同时执行多个线程，而多处理操作系统则能真正做到在每一处理器中执行一个线程。

正如本章开头所提到的，Windows NT 的一个主要设计目标是在多处理器计算机系统上运行良好。Windows 2000 也是对称式多处理（*symmetric mutiprocessing* —— SMP）操作系统。它没有主处理器——操作系统与用户线程可以调度到任一处理器上运行。并且，所有处理器共享一

个存储空间。这一模型与非对称式多处理（*asymmetric mutiprocessing* —— ASMP）相对，在 ASMP 中，操作系统选择一个处理器来执行操作系统代码，其他处理器只运行用户代码。这两种多处理模型的区别见图 2-1。

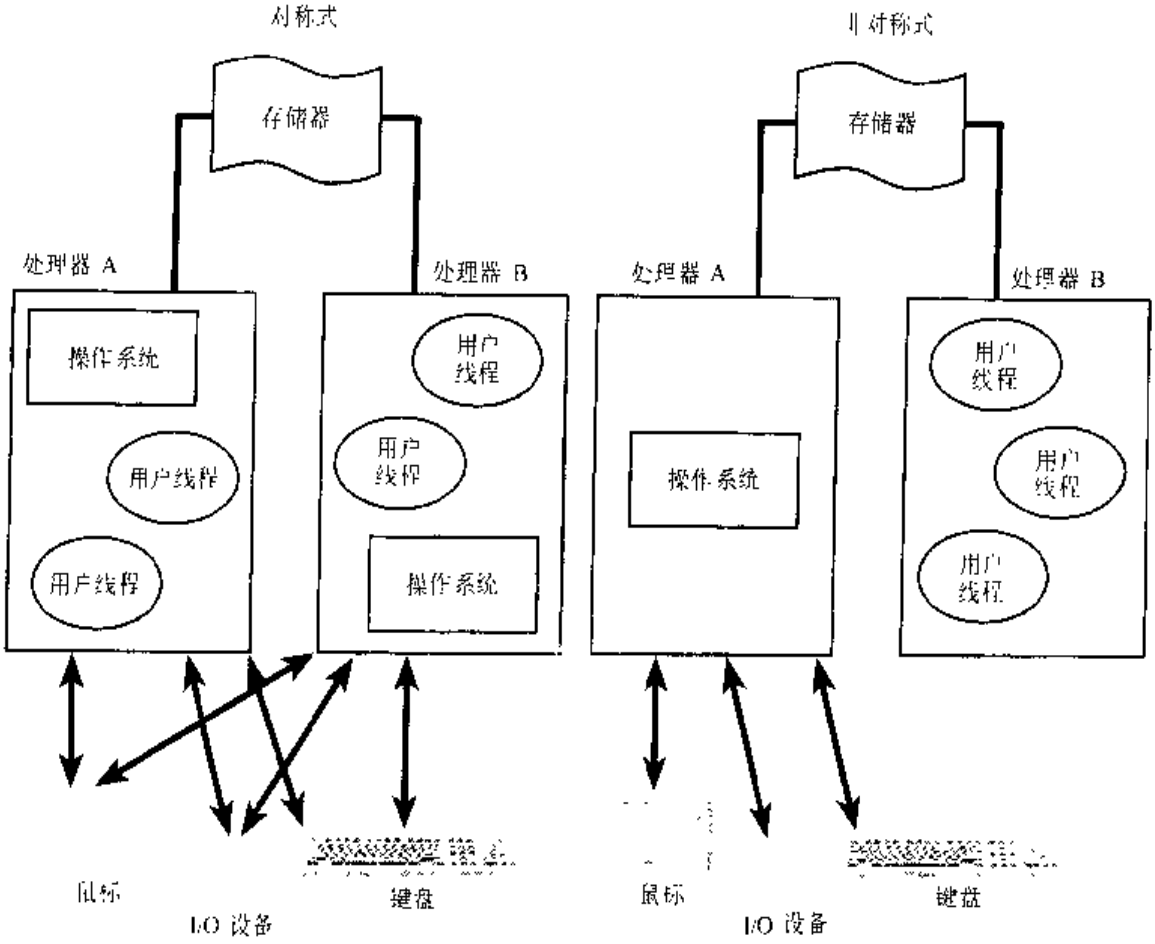


图 2-1　对称式与非对称式多处理

尽管 Windows NT 原本设计支持多达 32 个处理器，但在多处理器设计中，并没有限制处理器的个数只能为 32 ——这一数字仅仅是为了明显和方便，因为 32 个处理器可以容易地代表使用本机 32 位数据类型的位屏蔽。

实际支持的处理器个数依赖于所使用的 Windows 2000 的版本（各种 Windows 2000 版本将在以下部分介绍）。这一数字存储在注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session \ Manager \ Licensed Processors 中。请记住擅自改动此数据是对微软注册的侵犯，将可能导致在重新启动时系统的崩溃，因为修改注册表不仅仅改变此键值，而且涉及允许使用更多的处理器。

2.2.3　可伸缩性

多处理器操作系统的一个关键问题是可伸缩性。要在 SMP 系统上准确运行，操作系统代码必须遵守严格的准则与规则。在多处理系统中，资源的争用与其他性能问题比单处理器系统更复杂，这些必须在系统设计中加以考虑。Windows 2000 综合了下述特点，这是确保它作为成功多处理器操作系统的关键所在：

■ 在任一处理器和多处理器上同时运行操作系统代码的能力。

■ 在单个进程内执行多个线程，每一线程能在不同的处理器上同时执行。

■ 在内核、设备驱动程序和服务器进程中实现细粒度同步，使得更多组件在多处理器上同时运行

另外，Windows 2000 提供了机制（如 I/O 完成端口，在第 9 章中介绍）使得多线程序服务器进程的有效实现在多处理器系统上有更好的可伸缩性。

多处理器同步将在第 3 章中介绍。多处理器的线程调度细节在第 6 章中介绍。

2.3 总体结构

简要介绍 Windows 2000 的设计目标和组件后，再来看一下构成其体系结构的主要系统组件，该体系结构的简图见图 2-2。注意该简图是最基本的介绍，并没有显示所有内容。Windows 2000 各种组件的情况将在本章后面详细介绍。

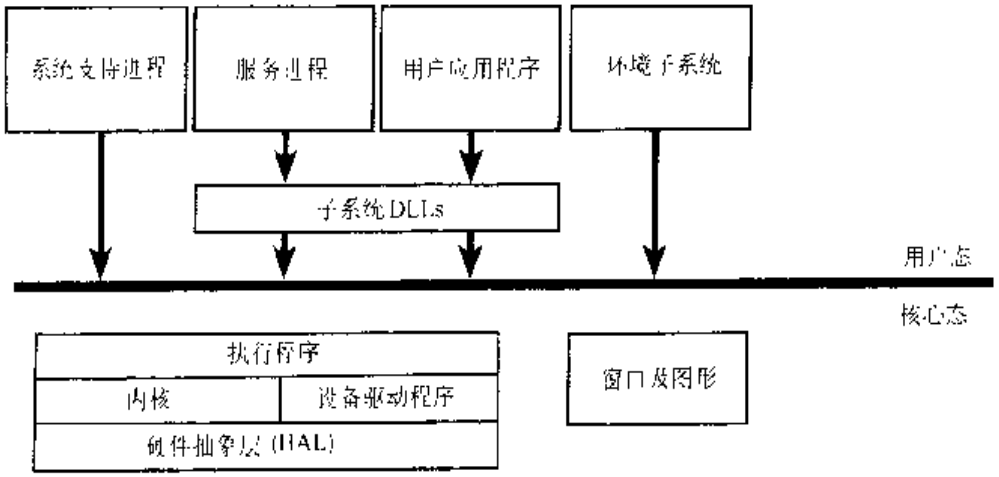


图 2-2 Windows 2000 体系结构简图

在图 2-2 中，注意将 Windows 2000 操作系统的用户模式与内核模式分开的横线。横线上的方框代表用户模式进程，横线下的组件是内核模式操作系统服务。正如第 1 章所提到的，用户模式线程在受保护的进程地址空间执行（尽管当它们在内核模式执行时能够访问系统空间）。这样，系统支持进程、服务进程、用户应用程序和环境子系统都有它们各自的专用进程地址空间。

四种基本类型的用户模式进程分述如下：

■ 固定的（或硬连线的（hardwired））系统支持进程（*system support process*），如登录进程（logon process）和会话管理器（session manager），它不是 Windows 2000 的服务（不是通过服务控制管理器启动的）

■ 服务进程（*service process*）提供 Win 32 服务，如 Task Scheduler 和 Spooler 服务。许多 Windows 2000 服务器应用程序，如 Microsoft SQL Server 和 Microsoft Exchange Server，也包括作为服务运行的组件

■ 用户应用程序（*user application*）可以是 Win32、Windows 3.1、MS－DOS、POSIX 或 OS/2 1.2 五种类型之一。

■ 环境子系统（*environment subsystem*），通过一系列可调用函数，将本机操作系统服务提供给用户应用程序，提供操作系统环境或个性（personality）。Windows 2000 装有 3 种环境子系统：Win32、POSIX 和 OS/2。

在图 2-2 中，注意“服务进程”图框和“用户应用程序”图框下面的图框“子系统 DLL”。在 Windows 2000 中，用户应用程序不直接调用本机 Windows 2000 操作系统的服务，而是通过一个或多个子系统动态链接库（*subsystem dynamic - link libraries*，DLL）。子系统 DLL 的功能是将一个存档的函数翻译为适宜的内部（并且没存档）Windows 2000 系统服务调用。这种转换包括或不包括将 -信息传送到正服务于用户应用程序的环境子系统进程。

Windows 2000 的内核模式组件如下：

■ Windows 2000 的执行程序（*executive*）包括基本的操作系统服务，如存储管理、进程和线程管理、安全、I/O 和进程间通信。

■ Windows 2000 的内核（*kernel*）包括低级的操作系统函数，如线程调度、中断和异常调度和多处理器同步。它还提供一系列的例程和基本对象，供其他执行程序用来实现高级构造。

■ 设备驱动程序（*device driver*）包括将用户 I/O 函数调用翻译为特定的硬件设备 I/O 请求的硬件设备驱动程序以及文件系统和网络驱动程序。

■ 硬件抽象层（*hardware abstraction layer*，HAL）是代码层，它将内核，设备驱动程序和其他 Windows 2000 执行程序与特定平台硬件差别（如母板间的不同）隔离。

■ 窗口和图形系统（*windowing and graphics system*）实现图形用户界面（GUI）函数（Win32 USER 和 GDI 函数），如处理窗口、用户界面控制和绘图

表 2-1 列出了 Windows 2000 操作系统内核组件的文件名（你需要了解这些名字，因为我们将通过它们提到一些系统文件）。每一组件将在本章的后面和下章中详细讲解。

表 2-1 Windows 2000 内核系统文件

文件名	组 件
Ntoskrnl.exe	执行程序和内核
Ntkrnlpa.exe	支持物理地址扩展（Physical Address Extension，PAE）的执行程序与内核，允许多达 64GB 的物理内存的寻址
Hal.dll	硬件抽象层
Win32k.sys	Win32 子系统的内核模式部分
Ntdll.dll	内部支持函数和执行程序函数的系统服务调度内存程序（stub）
Kernel32.dll	内核 Win32 子系统 DLL
Advapi32.dll	
User32.dll	
Gdi32.dll	

在深入剖析这些系统组件的细节之前，让我们研究一下 Windows 2000 Professional 与 Win-

dows 2000 Server 各种版本的区别。

2.4 Windows 2000 的产品包

Windows 2000 有四个版本：Windows 2000 Professional、Windows 2000 Server、Windows 2000 Advanced Server、Windows 2000 Datacenter Server。这些版本的区别在于：

- 支持处理器的个数。
- 支持物理内存的数量
- 支持并行网络连接的数目。
- Server 版本有分层服务，而 Professional 版本没有。

这些差别在表 2-2 中作了总结。

表 2-2 Windows 2000 Professsional 与 Server 版本的差别

版 本	支持处理器数	支持物理内存	并行客户网络连接数[1]	附加分层服务
Windows 2000 Professional	2	4GB	10	
Windows 2000 Server	4	4GB	无限制	域控制器（domain controller），Active Directory 服务，基于软件 RAID，动态主机配置协议（DHCP）服务器，域名系统（DNS）服务器，分布式文件系统（DFS）认证服务，远程安装，终端业务
Windows 2000 Advance Server	8	8GB	无限制	2－结点（node）集群
Windows 2000 Datacenter Server	32	64GB[2]	无限制	4－结点（node）集群，进程控制管理器工具[3]

注：(1) Windows 2000 Professional 的终端用户许可协议（包括在 \ Winnt \ System32 \ Eula. txt 中）声明："可以允许最多为 10 台计算机或其它电子设备（每个为一台设备）连接到工作站计算机上以实现针对文件和打印服务、Internet 信息服务以及远程访问（包括连接共享和电话服务）的产品服务。"这种限制作用于文件、打印共享及远程访问，而 Internet 信息服务则不受此限。

(2) 理论限制；由于商业硬件的性能使得支持限制可能少于此值。

(3) 要了解更多内容，参见第 6 章

Windows 2000 各种版本之间没有区别的是内核系统文件：内核映像 Ntoskrnl.exe（PAE 版本的 Ntkrnlpa.exe）、HAL 库、设备驱动程序、基本系统工具和 DLL。对 Windows 2000 所有版本，这些文件都是相同的。例如，没有 HAL 的特殊 Server 版本。

但是，由于运行版本的不同，许多组件的操作是不同的。作为高性能的应用服务器，Windows 2000 Server 系统对系统吞吐量进行了优化，而 Windows 2000 Professional 尽管有服务器功能，但是为了交互桌面使用，对响应时间进行了优化。例如，根据产品类型不同，在系统引导时，一些资源分配决策是不同的，如操作系统堆（或交换区）的大小与数量、内部系统工作者线程的数量，系统数据高速缓存的大小。还有，运行时（run－time）策略的决定，如内存管理器对

系统和进程内存需求的平衡。Windows 2000 Server 版本与 Windows 2000 Professional 版本是有区别的。在这两个版本系列中，甚至一些线程调度细节也有不同的默认行为。两种产品有明显的操作上的区别，这些将在本书后面的相关章节中讲述。除了特别声明以外，本书的所有内容对 Windows 2000 Server 版本与 Windows 2000 Professional 版本都是适用的。

如果内核映像在 Windows 2000 不同版本的产品中是一样的，那么系统如何识别所引导的版本呢？通过查询注册表 HKLM \ SYSTEM \ CurrentControlSet \ Control \ ProductOptions 键下的键值 ProductType 和 ProductSuite。ProductType 用来区分系统是 Windows 2000 Professional 系统还是 Windows 2000 Server 系统（任一版本）。这一有效键值列于表2-3中。结果存储在系统全局变量 MmProductType 中，它可以从设备驱动程序中通过内核模式支持函数 MmIsThisAnNtAsSystem 来查询，该函数存档在 Windows 2000 DDK 中。

表 2-3 ProductType 注册表键值

Windows 2000 版本	ProductType 键值
Windows 2000 Professional	WinNT
Windows 2000 Server（域控制器）	LanmanNT
Windows 2000 Server（仅服务器）	ServerNT

不同的注册表键值 ProductSuite 区分 Windows 2000 Server、Advanced Server 和 Datacenter Server 以及是否安装了终端服务（Terminal Service）（仅在 Server 系统中）。在 Windows 2000 Professional 系统中，该值为空（blank）。

如果用户程序需要确定所运行的是 Windows 2000 的哪一版本，它们可以调用存档在 Platform SDK 中的 Win32 *VerifyVersionInfo* 函数。设备驱动程序可以调用存档在 Windows 2000 DDK 中的内核模式函数 *RtlGetVersion*。

2.4.1 检查链接编译

在 Windows 2000 Professional 中有一个专门的调试（debug）版本，称为检查链接编译（checked build）。这一版本仅在 MSDN Professional（或 Universal）CD 中发行。它是为帮助设备驱动程序开发者而提供的——检查链接编译能检测被设备驱动程序或其他系统代码调用的内核模式函数的许多严格错误。例如，如果驱动程序（或某些内核模式代码段）对正在检测参数的系统函数进行了无效调用（如在错误中断级获得自旋锁（spinlock）），系统会在查出问题后中断执行，而不是允许数据结构被破坏，以防止系统以后可能的崩溃。

检查链接编译是 Windows 2000 源代码的重编译，将编译时间标志（compile - time flag）DEBUG 设置为 TRUE，检查链接编译二进制代码中的大部分附加代码是使用 ASSERT 宏的结果，ASSERT 宏在 DDK 的头文件 Ntddk.h 中定义，它存档在 DDK 文档中。该宏测试条件（如数据结构或参数的有效性），如果表达式值为 FALSE，宏调用内核模式函数 RtlAssert，该函数调用 DbgPrint 将调试信息的文本传递到内核调试程序（kernel debugger）（如果已附加（attach））来显示，然后提示用户要做的事情（设置断点、忽略、终止进程或终止线程）。如果系统不是通过内核调试程序引导的（通过 Boot.ini 中/DEBUG 开关（switch）），并且当前没有连接内核调试程序，ASSERT 检测的失败会使系统崩溃。

尽管微软没有提供 Windows 2000 Server、Advanced Server、Datacenter Server 的检查链接编译版本，但可以人工地将内核映像的检测（调试）版本复制到 Windows 2000 Server 系统中，重新启动，并以检测内核运行。（也可以对其他系统文件这样做，但大多数使用检查链接编译的开发者，实际上仅需要内核映像的检测版本，而并不需要所有设备驱动程序、工具和 DLL 的检测版本。）

2.4.2 多处理器系统文件

多处理器系统和单处理器系统相比有 6 个系统文件[○]不同（参见表 2-4）。在安装时，合适的文件被选定并复制到本机 \ Winnt \ System32 目录下。要确定哪些文件需要复制，请参见文件 \ Winnt \ Repair \ Setup.log，该文件列出了需要复制到本机系统盘上的所有文件和它们从何处来到分布式媒介。

表 2-4 多处理器和单处理器系统文件

系统盘上的文件名字	CD 上单处理器版本的名字	CD 上多处理器版本的名字
Ntoskrnl.exe	\ I386 \ Ntoskrnl.exe	\ I386 \ Ntkrnlmp.exe
Ntkrnlpa.exe	\ I386 \ Driver.cab 目录下的 Ntkrnlpa.exe	\ I386 \ Driver.cab 目录下的 Ntkrpamp.exe
Hal.dll	与系统类型有关（请看表 2-5 的 HAL 列表）	与系统类型有关（请看表 2-5 的 HAL 列表）
Win32k.sys	\ I386 \ UNIPROC \ Win32k.sys	\ I386 \ Driver.cab 目录下的 Win32k.sys
Ntdll.dll	\ I386 \ UNIPROC \ Ntdll.dll	\ I386 \ Ntdll.dll
Kernel32.dll	\ I386 \ UNIPROC \ Kernel32.dll	\ I386 \ Kernel32.dll

注：如果在 Windows 2000 CD 的 \ I386 \ UNIPROC 的文件夹上查找，你会发现名字为 Winsrv.dll 的文件——虽然该文件存在于名为 UNIPROC 的文件夹里，意味着是单处理器版本，但实际上它是多处理器和单处理系统映像的仅有唯一版本。

实验：查看多处理器支持的文件

你可以通过查看在 Device Manager 中的 Computer 驱动程序器细节看到多处理器系统的不同文件。

1）打开 System 属性（可以通过在 Control Panel 中选择 System 或右击桌面上 My Computer 图标，选择 Properties）

2）点击 Hardware 选项。

3）点击 Device Manager。

4）展开 Computer 对象。

5）双击 Computer 下面的子节点（child node）。

6）点击 Driver 选项。

7）点击 Driver Details。

你可以看到多处理器系统的对话框（如图 2-3 所示）。

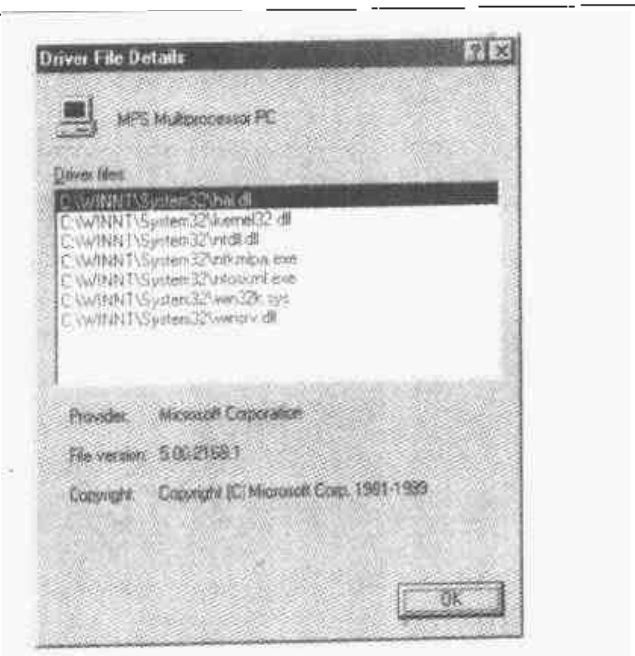


图 2-3 查看多处理器支持的文件

这些关键系统文件具有单处理器版本的原因是性能问题——多处理器同步比使用单处理器更加复杂、耗时，因此关键系统文件通过具有专门的单处理器版本，在单处理器系统上避免了这一开销（它组成了运行 Windows 2000 系统的大部分）。

有趣的是，尽管 Ntoskrnl 的单处理器与多处理器版本是通过有条件地编译源代码生成的，Ntdll.dll 和 Kernel32.dll 的单处理器版本却是通过修补 x86 LOCK 和 UNLOCK 指令生成的，它用来使多线程与 no-operation（NOP）指令（它不做任何事情）同步。

在单处理器与多处理器系统中，组成 Windows 2000 的其他系统文件（包括所有的工具、库和设备驱动程序）具有相同的版本（也就是说，它们正确地处理多处理器的同步问题）。你可以将这一方法用于你所编译的任一软件，不论它是 Win32 应用程序还是设备驱动程序——当你设计软件时，请记住多处理器的同步问题，并且在单处理器与多处理器系统上都要检测软件。

在检查链接编译 CD 上，如果比较 Ntoskrnl.exe 与 Ntkrnlmp.exe 或 Ntkrnlpa.exe 和 Ntkrpamp.exe，你将发现它们是相同的——它们都是相同文件的多处理器版本。换句话说，没有提供了检查链接编译的内核映像的单处理器调试版本。

实验：查看你所运行的 Ntoskrnl 版本

在 Windows NT4 中，通过点击 Windows NT Diagnostics（Winmsd.exe）中的 Version 选项，你可以知道你所运行的 Ntoskrnl 的版本。在 Windows 2000 中，没有工具显示这一信息。但是，在每次系统引导时，一个 Event Log 项被写入，它记录了内核映像的类型（单处理器还是多处理器，是自由链接编译还是检查链接编译），如图 2-4 所示。在 Start 菜单选择 Programs/Administrative Tools/Event Viewer，选择 System Log，双击 Event Log 的 Event ID 6009 项，表明该条目在系统启动时被写入。

Event Log 项并不指出你是否引导了 PAE 版本的支持 4GB 以上物理内存的内核映像（Ntkrnlpa.exe）。但是，可以通过查看注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \

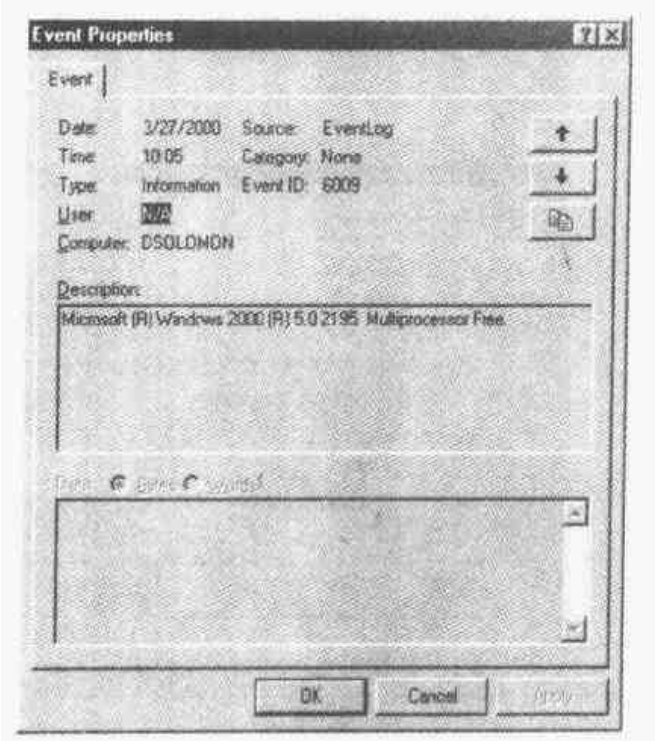


图 2-4 查看你所运行的 Ntoskrnl 版本

System StartOptions，来查看你是否引导了 PAE 内核。另外，如果你引导了 PAE 内核，注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ Memory Management \ PhysicalAddressExtension 设定为 1。

通过检查文件属性，你可以确定你是否安装了 Ntoskrnl（或 Ntkrnlpa）的多处理器版本：运行 Windows Explorer，在 \ Winnt \ System32 文件夹中右击 Ntoskrnl.exe、选择 Properties。然后点击 Version 选项，选择 Original Filename 属性——如果你正在运行多处理器版本，你将会看到如图 2-5 所示的对话框。

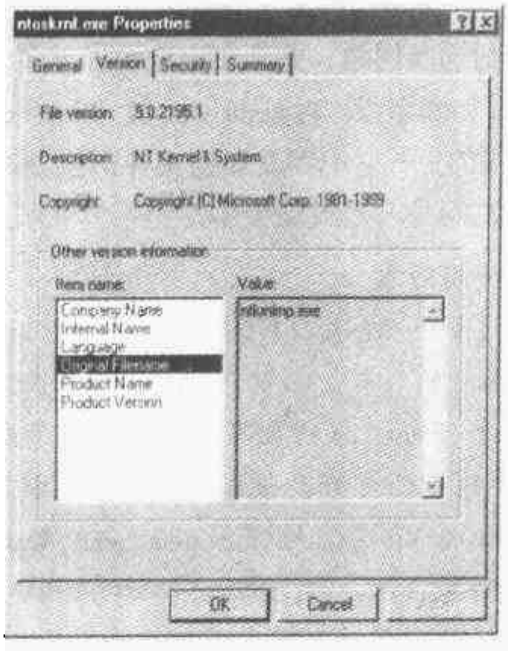


图 2-5 查看文件属性

最后，如前面所提到的，通过查看文件 \ Winnt \ Repair \ Setup.log，你将清楚的看到在安装时所选择的内核映像和 HAL。

2.5　关键系统组件

到目前为止，我们已经了解了 Windows 2000 的高级体系结构，让我们再深入了解一下系统的内部结构和各关键操作系统组件所起的作用。与本章前面的图 2-2 比较，图 2-6 是 Windows 2000 系统体系结构和组件更为详细的完整图示。

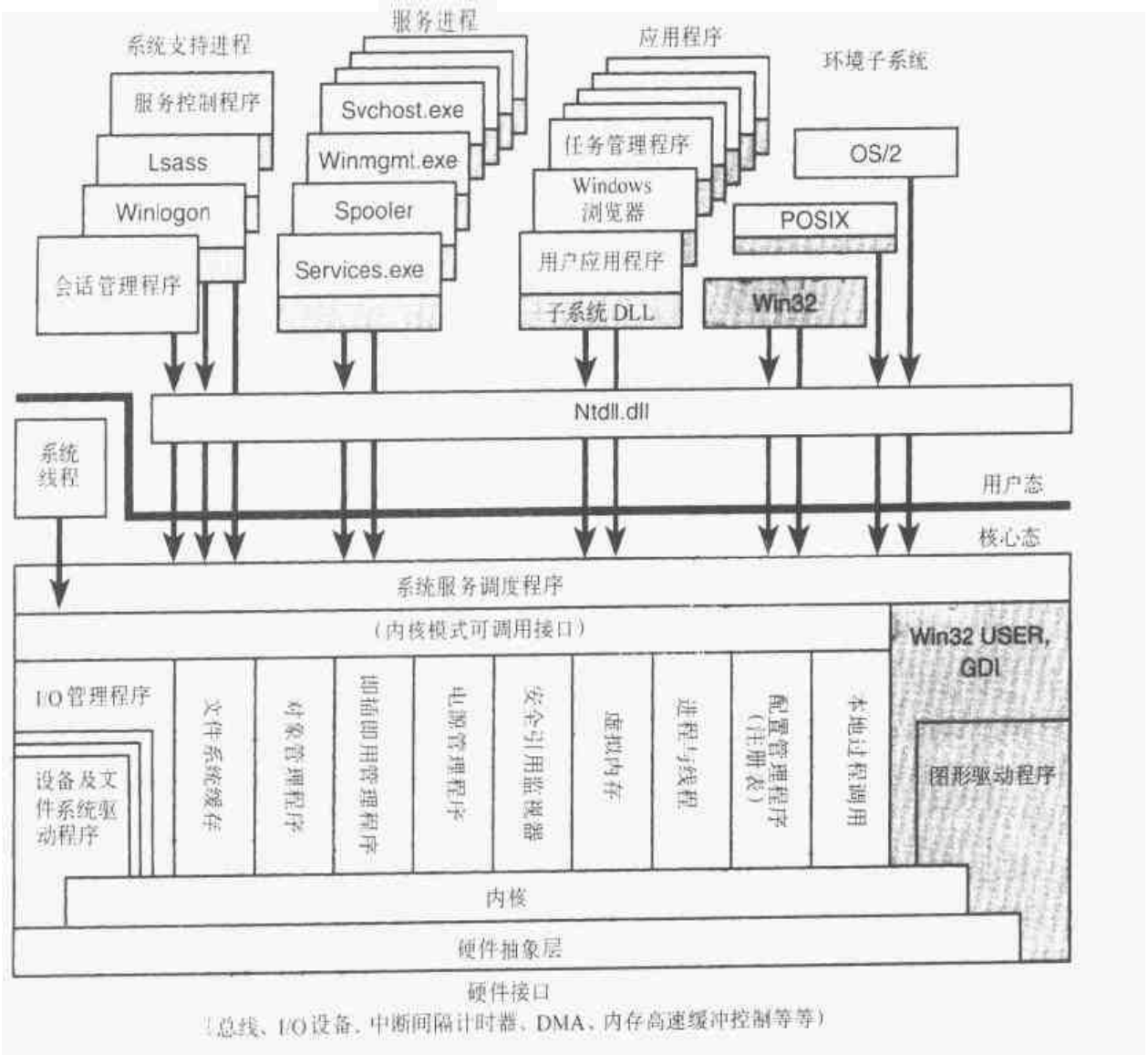


图 2-6　Windows 2000 体系结构

下面部分将深入研究该图的每一重要部分。第 3 章讲述系统使用的主要控制机制（如对象管理器、中断等等）。第 4 章讲述 Windows 2000 的启动与关机过程，第 5 章详述管理机制（如注册表、服务进程和 Windows 管理装置（Windows Management Instrumention，WMI）。其余章节更为详细地剖析内部结构和关键区域操作，如进程与线程、存储管理、安全性、I/O 管理器、高速缓存管理器、Windows 2000 文件系统（NTFS）和网络。

2.5.1 环境子系统与子系统 DLL

如图 2-6 所示，Windows 2000 有 3 个环境子系统：OS/2、POSIX 和 Win32。我们随后将说明，在这 3 个子系统中，Win32 子系统是特殊的，因为 Windows 2000 离开它将无法运行（它拥有键盘、鼠标、显示器，甚至在没有交互用户登录的服务系统上，也需要它存在）。实际上，其他两个子系统根据需要配置，但 Win32 子系统必须始终运行。

子系统的启动信息存储在注册表键 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ SubSystems 中。图 2-7 显示了该键下的键值。

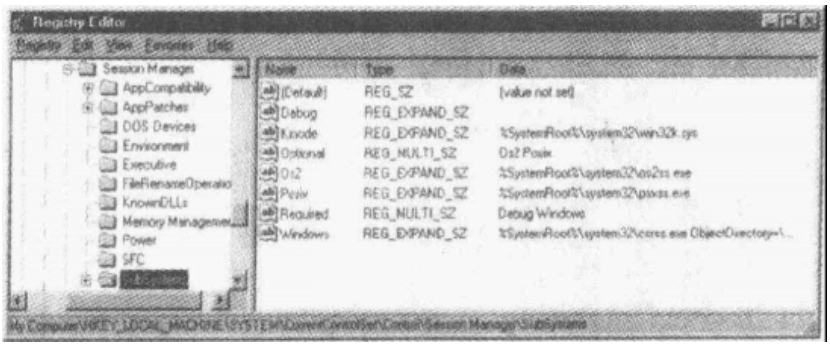


图 2-7 注册表编辑器显示 Windows 2000 启动信息

当系统引导时，Required 键值列出所装入的子系统。该键值有两个字符串：Windows 和 Debug。Windows 键值包含了 Win32 子系统 Csrss.exe 的文件规格，它代表 Client/Server Run - Time Subsystem。[1] Debug 为空（blank）（它用于内部检测），所以不做任何事情。Option 键值表明 OS/2 和 POSIX 子系统将根据需要启动。注册表键值 Kmode 包含了 Win32 子系统的内核模式部分的文件名，Win32k.sys（本章后面将说明）。

环境子系统的作用是向应用程序提供 Windows 2000 的基本执行系统服务的某些子集（subset）。在 Windows 2000 中，每一子系统可以访问本机服务的不同子集。这意味着某些事情可以通过建立在一个子系统上的应用程序来完成，但不能通过建立在其他子系统上的应用程序来完成。例如，一个 Win32 的应用程序不能使用 POSIX 的 fork 函数。

每一可执行映像（.exe）仅和一个子系统相联系。当一个映像运行时，进程生成代码检测在映像的头（header）中的子系统的类型代码，这样就可以知道新进程所属的子系统。这一类型代码通过在 Microsoft Visual C++ 中使用带/SUBSYSTEM 的限定词的 link 命令指定，它可以在 Windows 2000 的资源工具包中用 Exetype 查看。

函数调用不能在子系统间混用。换句话说，一个 POSIX 应用程序只能调用 POSIX 子系统输出的服务，而一个 Win32 的应用程序只能调用 Win32 子系统输出的服务。后面你将会看到，这一限制的原因是 POSIX 子系统实现极有限的函数集（仅有 POSIX1003.1），对移植 UNIX 应用程序来说，它不是一个有用的环境。

---

[1] 作为历史记载，Win32 子系统进程被称为 Csrss.exe 的原因是在最初设计 Windows NT 的时候，所有子系统都将作为线程在单个操作系统环境的子系统进程中执行。当 POSIX 和 OS/2 子系统被转移到它们自己的进程中时，Win32 子系统进程的名字从没有改变。

如前面所提到的，用户应用程序不能直接调用 Windows 2000 系统服务，而是通过一个或多个子系统 DLL。这些库输出链接到该子系统的程序可以调用的存档的接口。例如，Win32 子系统 DLL（如 Kernel32.dll、Advapi32.dll、User32.dll 和 Gdi32.dll）实现 Win32 API 函数。POSIX 子系统 DLL 实现 POSIX1003.1 API。

实验：查看映像子系统类型

通过使用 Windows 2000 资源工具包的 Exetype 工具或 Windows 2000 Support Tools 和 Platform SDK 中的 Dependency Walker 工具（Depends.exe），你可以看到映像子系统类型。例如，注意两种不同的 Win32 映像，Notepad.exe（简单文本编辑）和 Cmd.exe（Windows 2000 的命令提示符）的映像类型：

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

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

实际上，只有一个 Windows 子系统，而没有分头对于图形映像和基于字符或控制台的映像。另外，Windows 2000 不支持 Intel 386 处理器（或 486）——因为 Exetype 程序输出的文本还未更新。

当应用程序调用子系统 DLL 的函数时，可能会发生以下三种情况：

■ 函数完全在子系统 DLL 用户模式中实现。换句话说，没有信息传送到环境子系统进程，也没有 Windows 2000 的执行系统服务被调用。函数在用户模式中完成，结果返回调用程序。这种函数的例子包括 *GetCurrentProcess*（它始终返回 -1，定义该值用来引用所有相关进程函数的当前进程）和 *GetCurrentProcessId*（对一个运行进程，进程 ID 并不改变，所以该 ID 在缓冲位置检索以避免调用内核）。

■ 函数需要对 Windows 2000 的执行程序进行一次或多次调用。例如，Win32 的 *ReadFile* 和 *WriteFile* 函数分别调用底层的内部（没存档的）Windows 2000 I/O 系统服务 *NtReadFile* 和 *NtWriteFile*。

■ 函数需要在环境子系统进程中做些工作（环境子系统进程以用户模式运行，负责维护在其控制下运行的客户应用程序的状态）。在这种情况下，通过向子系统传递执行某些操作的信息，将客户/服务器请求传送给环境子系统。然后，在返回调用程序之前，子系统 DLL 等待应答。

某些函数可能是以上两项或三项的结合，如 Win32 的 *CreatProcess* 和 *CreatThread* 函数。

尽管 Windows 2000 设计支持多个独立的环境子系统，但从实际的角度看，让每一子系统实现处理窗口和显示 I/O 的所有代码将导致系统函数的大量复制，最终将对系统规模和性能产生负面影响。因为 Win32 是主要的子系统，Windows 2000 的设计者决定将基本的函数放置在这里，让其他的子系统在 Win32 子系统上调用来执行显示 I/O。这样，POSIX 和 OS/2 子系统调用 Win32 子系统的服务来执行显示 I/O（实际上，如果检查这些映像的子系统类型，可以看到它们是 Win32 可执行程序）。

让我们更加深入地研究每一个环境子系统。

1.Win32 子系统

Win32 子系统包括下面的主要组件：

■ 环境子系统进程（Csrss.exe）包括支持：

□ 控制台（文本）窗口。

□ 创建、删除进程与线程。

□ 支持 16 位虚拟 DOS 机器（VDM）进程的部分。

□ 其他多种函数，如 *GetTempFile* 、*DefineDosDevice*、*ExitWindowsEx* 和几种自然语言支持函数。

■ 内核模式设备驱动程序（Win32k.sys）包括：

□ 窗口管理器，它控制窗口显示，管理屏幕输出，收集来自键盘、鼠标和其他设备的输入，并将用户信息传送给应用程序。

□ 图形设备接口（Graphics Device Interface—GDI），它是一个图形输出设备函数库。它包括线条、文本、图形绘制和图形操作函数。

■ 子系统 DLL（如 Kernel32.dll，Advapi32.dll，User32.dll 和 Gdi32.dll）通过对 Ntosknl.exe 和 Win32.sys 的调用将存档的 Win32 API 函数翻译为合适的非存档的内核模式系统服务。

■ 图形设备驱动程序是硬件相关的图形显示驱动程序、打印机驱动程序和视频小型端口（video miniport）驱动程序。

应用程序调用标准 UESR 函数，在显示器上生成用户界面控件，如窗口和按钮。窗口管理器将这些请求传到 GDI，GDI 再将它们传递给图形设备驱动程序，在此，它们按照显示设备的要求将其格式化。显示驱动程序与视频小型端口驱动程序一起完成视频显示支持。

GDI 提供一系列标准二维函数，在不了解设备的任何情况下，函数使应用程序与图形设备通信。GDI 函数是应用程序与图形设备的中介，如显示驱动程序和打印机驱动程序。GDI 解释应用程序对图形输出的请求，并将请求传送给图形显示驱动程序。它还为使用不同的图形输出设备的应用程序提供标准接口。该接口能使应用程序代码独立于硬件设备和它们的驱动程序。GDI 根据设备能力处理它的信息，经常将请求分为可管理的部分。例如：某些设备知道绘制椭圆的方向；某些要求 GDI 将命令译为一系列安放在特定坐标上的象素。要获得更多关于图形和视频驱动程序结构的信息，参见 Windows 2000 DDK 的 *Graphics Drivers* 一书的 *Design Guide* 部分。

在 *Windows NT 4 以前*，窗口管理器与图形服务是用户模式 Win32 子系统进程的一部分。在 Windows NT 4 中，大量的窗口和图形代码从在 Win32 子系统进程的环境运行转移到在内核模式运行的一系列可调用的服务（在文件 Win32k.sys 中）。这一变化的重要原因是提高系统的整体

性能。由于分开的包含 Win32 图形子系统的服务器进程要求多线程和进程环境切换（context switch），这样会消耗相当的 CPU 周期循环和内存资源，尽管原有设计是高度优化的。

例如，对于客户端的每一线程，在 Win32 子系统进程中都有一个专用的、配对的服务器线程等待客户线程的请求。一个特殊的被称为快速 LPC 的进程间通信工具用来传送线程间的信息。和一般的线程环境切换不同，使用快速 LPC 的配对线程间切换在内核中不产生重新调度事件，所以在内核的抢先型线程调度程序中，服务器线程在获得它的时间片之前可以在客户线程的剩余时间片中运行。而且，共享内存缓冲器允许大型数据结构如位图的快速传递，客户机直接以只读方式访问关键服务器数据结构，以减少客户与 Win32 服务器之间的线程/进程切换的需要。此外，GDI 操作是成批的。批处理是指 Win32 应用程序的一系列图形调用不会被发送到服务器，也不在输出设备上绘出，直到一个 GDI 批处理对列被填充满为止。通过使用 Win32 的 *GdiSetBatchLimit* 函数，可以设置队列的大小，也可以随时通过 *GdiFlush* 函数来刷新队列。相反地，GDI 的只读特性和数据结构一旦被 Win32 的子系统进程获得就会缓存在客户端以便随后快速访问。

尽管具有这些优化，系统的总体性能对图形密集的应用程序依然不够。直接的解决方法是通过将窗口和图形系统移到内核模式，以减少对附加线程及其导致的环境切换的需要。另外，一旦应用程序调用了窗口管理器和 GDI，这些子系统不通过用户模式或内核模式的切换，就能直接访问其他的 Windows 2000 执行程序组件。GDI 调用视频驱动程序时，进程涉及到高频率和高带宽的视频硬件间相互作用，在这种情况下，这种直接访问尤其重要。

将 Win32 USER 和 GDI 移入内核模式会削弱 Windows 2000 的稳定性吗？

有人想知道将这么多代码移到内核模式是否会对系统的稳定性产生很大影响。对系统稳定性的影响已经最小化的原因是，在 Windows NT 4 之前，用户模式 Win32 子系统进程（Csrss.exe）的错误（如存取违例（access violation））会导致系统的崩溃。这种崩溃发生是因为，Csrss 的父进程（会话管理器 Smss，将在后面讲述）对 Csrss 的进程句柄做了等待操作，一旦等待返回，Smss 就会使系统崩溃——因为 Win32 子系统进程过去是（现在仍是）系统运行的至关重要的进程，因为该进程包括了描述显示器上窗口的数据结构，所以该进程的破坏将会导致用户界面的破坏。然而，甚至作为服务器的 Windows 2000 操作系统，即使没有交互进程，缺少该进程也不能运行，因为服务器进程可能使用窗口消息（window messaging）来驱动应用程序的内部状态。在 Windows 2000 中，在内核模式运行的相同代码存取违例会简单地使系统更快崩溃，因为内核模式的异常会导致系统崩溃。

然而，将窗口和图形系统移到内核模式之前并不存在附加的理论上的危险，因为这些代码正在内核模式中运行，一个错误（如使用错误指针）会破坏内核模式保护的数据结构。在 Windows NT 4 之前，这种引用会导致存取违例，因为内核模式页对用户模式是不可写的。但系统崩溃会随后发生，如前所述。现在代码在内核模式运行，对某些内核模式页进行写操作的错误指针引用不会立即导致系统崩溃，但如果它破坏数据结构，崩溃会随后发生。但是，这种引用有极小的可能会破坏内存缓冲区（而不是数据结构），可能导致向应用程序返回破坏的数据或将错误数据写到磁盘上。

另一种可能性来自将图形驱动程序移到内核模式。以前，图形驱动程序的一部分在Csrss内运行，其余部分在内核模式运行。现在，整个驱动程序都在内核模式运行。尽管微软没有开发所有的支持Windows 2000的图形设备驱动程序，但它的确直接与硬件生产商合作来确保他们生产出可靠、有效的驱动程序。同其他执行程序组件一样，所有安装在系统上的驱动程序都经过了同样的严格测试。

最后，了解这种设计（窗口和图形子系统在内核模式运行）从根本上来说不冒风险是很重要的。它与其他许多设备驱动程序所用方法是相同的（如网卡驱动程序和硬盘驱动程序）。自Windows NT开始，所有这些驱动程序就在内核模式操作，并具有高度的可靠性。

有些人推测将窗口管理器和GDI移到内核模式会破坏Windows 2000的抢先型多任务能力，理由是所有附加的Win32在内核模式耗费处理时间，其他线程抢先运行的机会将减少。这种观点是基于对Windows 2000体系结构的误解。在其他许多名义上是抢先型的操作系统中，在内核模式执行的程序从来被操作系统调度程序抢先——或者说只在某些有限数量的内核可重入的预定义点上是抢先的。但是，在Windows 2000中，在执行程序任何地方运行的线程都是抢先的，与在用户模式运行的线程一起调度，执行程序内的所有代码都是完全可重入的。其他理由之一就是这种能力对在SMP硬件上获得高度的系统可伸缩性是必要的。

另一种推测是SMP的可伸缩性将被这一变化破坏。理由是这样的：以前，应用程序与窗口管理器或GDI的交互作用涉及到两个线程，一个在应用程序中，一个在Csrss.exe中。所以，在SMP系统中，这两个线程并行运行以提高吞吐量。这种分析源于对Windows NT 4以前的版本的工作原理的误解。在大多数情况下，客户应用程序对Win32子系统进程的调用同步运行，即客户线程完全阻塞服务器线程的等待，只有等服务器线程完成调用后它才会重新运行。因此，SMP硬件上的并行从未实现。通过SMP系统的Performance工具，这种现象在负荷大的图形应用程序中很容易看到。观察者会发现，在两处理器系统中，每一处理器的负荷大约是50%，相对容易查找与正在执行的应用程序线程配对的单Csrss线程。实际上，因为这两个线程是相当密切的，并共享状态，所以处理器的高速缓存必须不断刷新以保持一致性。这种不断刷新正是在Windows NT 3.51中，单线程的图形应用程序一般在SMP机上比在单处理机系统上运行稍微慢些的原因。

结果是，Windows NT 4的这种变化提高了大量使用图形管理器和GDI应用程序的SMP吞吐量，特别是不只一个应用程序线程在执行时。在基于Windows NT 3.51的双处理器计算机上，当两个应用程序线程执行时，共有四个线程（两个在应用程序中，两个在Csrss中）在两个处理器上争时间。在任一时间，尽管只有两个线程准备运行，线程运行的一致模式的缺少导致访问局部性和缓存一致性的损失。这种损失是因为忙线程可能会在一个处理器和另一个处理器之间穿梭。在Windows NT4的设计中，两个应用程序线程本质上有各自的处理器，Windows 2000的自动线程相似性（thread affinity）往往不确定地在同一处理器上运行同一线程，这样使引用局部性最大化，使同步专用的各处理器存储缓存的需要最小。

总起来说，将窗口管理器和GDI从用户模式移到内核模式提高了性能，并没有明显降低系统的安全性或可靠性。

那么，在 Win32 子系统的用户模式进程部分还剩下些什么呢？控制台的绘制与更新或文本窗口由它处理，因为控制台应用程序没有重画窗口的概念。很容易看到这一行为——只要打开命令提示符，在它上面拖动另一个窗口，就可以看到在它重绘控制台窗口时，Win32 子系统进程疯狂运行。除了控制台窗口的支持，只有少数的 Win32 函数导致向 Win32 子系统进程传送信息：进程与线程的创建与终止、网络驱动符映射和临时文件的创建。一般的，一个运行的 Win32 应用程序不会对（即使会的话）Win32 子系统进程产生许多环境切换（context switch）。

2. POSIX 子系统

POSIX，" a portable operating system interface based on UNIX" 的首写字母缩写，指 UNIX 操作系统接口国际标准的汇集。POSIX 标准鼓励厂商实现 UNIX 风格的接口以使它们兼容，这样程序员就能容易地将他们的应用程序从一个系统上移到另一个系统上。

Windows 2000 仅实现了许多 POSIX 标准中的一个——POSIX.1，正式称为 ISO/IEC 9945－1：1990 或 IEEE POSIX 标准 1003.1－1990。包含该标准主要是为满足美国政府的要求，该要求于 80 年代中后期制定，它要求与联邦信息处理标准（Federal Information Processing Standard —— FIPS）151－2 指定的 POSIX.1 保持一致性（compliance），该标准由国家标准技术协会（National Institute of Standards and Technology）制定。Windows NT3.5，3.51 和 4 已经正式通过 FIPS 151-2 的测试与认证。

因为 POSIX.1 一致性是 Windows 2000 的一个强制性目标，所以操作系统设计要保证需要的基本系统支持存在以允许 POSIX.1 子系统的实现（如 fork 函数，它在 Windows 2000 的执行程序中和 Windows 2000 文件系统的硬文件链接的支持中实现）。但是因为 POSIX.1 只定义了有限的服务（如进程控制、进程间通信和简单字符单元 I/O 等等），所以 Windows 2000 的 POSIX 子系统不是一个完整的程序设计环境。因为应用程序不能在 Windows 2000 的子系统间混合调用，所以在默认情况下，POSIX 应用程序只能限制在由 POSIX.1 定义的严格的服务。这种限制意味着 Windows 2000 的 POSIX 执行不能创建线程或窗口或使用远程过程调用（RPC）和套接字。

为了解决这一问题，微软提供了一种称为 Interix 的产品。该产品包括了增强的 POSIX 子系统环境，它提供近 2000 个 UNIX 函数和 300 个类似 UNIX 的工具和实用程序（如需了解更多的有关 Microsoft Interix 产品的信息，请查看 *www.microsoft.com/WINDOWS 2000/guide/server/solutions/interix.asp*）。这一提高，使得将 UNIX 应用程序移植到 POSIX 子系统上更可行。但是，由于程序仍然作为 POSIX 可执行程序链接，它们不能调用 Win32 函数。

为了将 UNIX 应用程序移植到 Windows 2000 上并且允许使用 Win32 函数，你可以购买 UNIX 到 Win32 的移植库，如包含在 MKS NuTCRACKER 专业产品中的移植库，它可从 Mortice Kern System Inc（www.mks.com）获得。通过这一方法，UNIX 应用程序可以作为 Win32 可执行程序重编译、重链接，可以慢慢地集成对本机 Win32 函数的调用。

实验：查看 POSIX 子系统的启动

在默认情况下，首次运行 POSIX 执行程序时，POSIX 子系统配置启动，所以可以通过运行 POSIX 程序来观察这一启动，如 Windows 2000 资源工具包的 POSIX 实用程序之一。（可以在资源工具包 CD 的 \ Apps \ POSIX 文件夹中找到这些 POSIX 实用程序它一们并不

作为资源工具包工具安装的一部分进行安装)。按下面的步骤来查看 POSIX 子系统的启动:

1) 启动命令提示符。

2) 输入 *tlist/t*, 确定 POSIX 子系统未运行 (即在 Smss.exe 下面没有 Psxss.exe 进程)。

3) 运行 Windows 2000 资源工具包 CD 上的 POSIX 实用程序,如 \ Apps \ POSIX \ Ls.exe。

4) 当 POSIX 子系统启动, Ls 命令显示目录内容时, 你可以注意到有一个短暂的暂停。

5) 重新运行 *tlist/t*。这次, 注意 Psxss.exe 作为 Smss.exe 的子进程存在。

6) 第二次再运行 Ls.exe; 你将注意到一个快速响应 (到此, POSIX 子系统已经启动)。

7) 再运行 Ls.exe, 但按下 Ctrl + S 使输出暂停; 从另一命令提示符输入 tlist/t, 注意 POSIX 支持图象 (Posix.exe) 是由第一次命令提示符创建的进程, 它又创建了 Ls.exe 进程。你可以看到与下面注释输出相似的一些内容。

```
System (2)
  smss.exe (23)————————————— 会话管理器
    csrss.exe (31)————————————— Win32子系统
    .
    .
    .
    psxss.exe (187)————————————— POSIX子系统
explorer.exe (69) Program Manager
  CMD.EXE (93) Command Prompt - ls
    posix.exe (178)————————————— POSIX支持进程
      ls.exe (97)————————————— POSIX应用程序
                                  开始运行
```

在 Windows 2000 中编译和链接 POSIX 应用程序需要 Platform SDK 的 POSIX 头文件和库。POSIX 的可执行程序与 POSIX 的子系统库 (Psxdll.dll) 链接。因为在默认情况下, Windows 2000 根据需要配置启动 POSIX 子系统, 首运行 POSIX 应用程序时, 必须启动 POSIX 子系统进程 (Psxss.exe)。它会一直保持运行直到系统重新启动 (如果取消了 POSIX 子系统进程, 则不能再运行 POSIX 应用程序直到重新启动)。POSIX 映像本身并不直接运行, 而是运行一个称为 Posix.exe 的特殊支持映像, 它创建一个子进程以运行 POSIX 应用程序。

要了解 POSIX 子系统和将 UNIX 应用程序移植到 Windows 2000 的更多信息, 在 MSDN 库中查找 POSIX 和 UNIX。

3. OS/2 子系统

OS/2 环境子系统同内建的 POSIX 子系统一样, 使用上有很多局限性, 因为它只支持 OS/2 1.2 的 16 位基于字符的或视频 I/O (VIO) 的应用程序。尽管微软的确为 Windows NT 4 出售了替代品 OS/2 1.2 Presentation Manager 子系统, 但它并不支持 OS/2 2.x (或更高版本) 的应用程序 (对 Windows 2000 同样也不支持)。

因为 Windows 2000 不允许用户应用程序直接访问硬件, 所以也不支持包含试图执行 IN/OUT 指令 (访问某些硬件设备) 的 I/O 特权段的 OS/2 程序和高级视频 I/O (AVIO)。支持使用 CLI/STI 命令的应用程序—— 但系统中所有其他 OS/2 应用程序和发送 CLI 指令的 OS/2 进程中的其他所有线程都将被悬挂, 直到执行 一条 STI 指令。另外值得注意的是在 Windows 2000 中, 对

OS/2 16位应用程序调用32位DLL的特殊支持在移植程序时很有用。

本机OS/2 1.2的16-MB内存限制不会影响Windows 2000，OS/2子系统使用Windows 2000的32位虚拟地址空间，为OS/2 1.2应用程序提供高达512MB的内存，如图2-8所示。

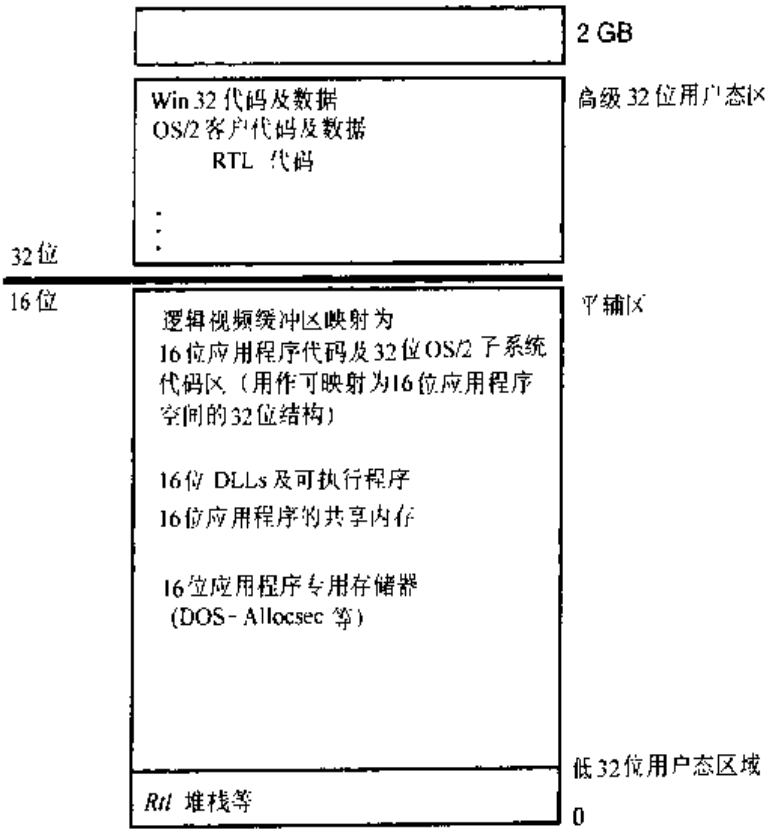


图2-8 OS/2子系统虚拟内存布局

平铺区域是保留的512MB的虚拟地址空间，当16位应用程序需要段时，平铺区域就会被提交或回收。OS/2子系统为每一进程保留一个局部描述符表（Local descriptor table，LDT），所有OS/2进程在同一LDT存储槽中具有共享的存储段。

我们将在第6章详细介绍，线程是执行程序的单元，因此要对它们进行调度以分配处理器时间。尽管Windows 2000的优先级从0到31，但64 OS/2优先级（从0到63）被映射为Windows 2000的动态优先级（从1到15）。OS/2线程从不接收Windows 2000从16到31的实时优先级。

同POSIX子系统一样，当你第一次激活一个兼容OS/2映像时，OS/2子系统就会自动启动。它一直运行直到系统重新启动。

有关Windows 2000如何处理运行POSIX和OS/2应用程序的信息，参见“6.2节 Creet Process流程”开头的部分。

2.5.2 Ntdll.dll

Ntdll.dll主要是用于子系统DLL的一个特殊系统支持库。它包含两种类型的函数：

- Windows 2000执行系统服务的系统服务调度占位程序（dispatch stub）。
- 子系统、子系统DLL和其他本机映像使用的内部支持函数。

第一组函数提供Windows 2000执行系统服务的接口以便它从用户模式调用。有200多个这

样的函数，如 *NtCreatFile*、*NtSetEvent* 等等。如前面所述，这些函数的大部分功能可以通过 Win 32 API 访问（但有一些不是，它们只为微软内部使用）。

针对每个函数，Ntdll 包含相同名字的入口点（entry point）。函数中的代码包含了指定体系结构的（architecture - specific）指令，它导致切换为内核模式来调用系统服务调度程序（在第 3 章中详细介绍），调度程序随后验证某些参数，并调用包含在 Ntoskrnl.exe 中的实代码的真正内核模式系统服务。

Ntdll 还包括许多支持函数，如映像加载程序（loader）（以 *Ldr* 开头的函数）、堆管理器（heap manager）、Win32 子系统进程通信函数（以 *Csr* 开头的函数）以及一般运行时库例程（以 *Rtl* 开头的函数）。它还包括用户模式异步过程调用（asynchronous procedure call—APC）调度程序和异常调度程序（APC 和异常将在第 3 章中介绍）。

2.5.3 执行程序

Windows 2000 执行程序是 Ntoskrnl.exe 的上层（内核是下层）。执行程序包括下列函数类型：

■ 从用户模式输出调用的函数。这些函数被称为系统服务（*system service*），通过 Ntdll 输出。大多数服务可以从 Win32 API 或另一环境子系统的 API 访问。但是一些服务不能通过任何存档的子系统函数访问（例子包括 LPC 和各种查询函数，如 *NtQueryInformationxxx*，专用函数如 *NtCreatPagingFile* 等）。

■ 只能从内核模式调用的函数，它们输出并存档在 Windows 2000 DDK 或 Windows 2000 的 Installable File System（IFS）Kit 中（有关 Windows 2000 IFS Kit 的信息，参见 *www.microsoft.com/ddk/ifskit*）。

■ 能从内核模式输出并调用的函数，但不存档在 Windows 2000 DDK 或 IFS Kit 中（如由引导视频驱动程序调用的函数，以 *Inbv* 开头）。

■ 作为全局符号定义的函数，但不输出。这些包括在 Ntoskrnl 内调用的内部支持函数，如以 *Iop*（内部 I/O 管理器支持函数）或以 *Mi* 开头的（内部内存管理器支持函数）。

■ 不作为全局符号定义的模块内部函数。

执行程序包括下面的主要组件，每一部分将在本书后面章节中详细介绍。

■ 配置管理器（*configuration manager*）（在第 5 章中介绍）负责实现并管理系统注册表。

■ 进程与线程管理器（*process and thread manager*）（在第 6 章中介绍），创建并终止进程与线程。进程与线程的底层支持在 Windows 2000 的内核中实现；执行程序将附加的语义和函数加到这些低级对象中。

■ 安全引用监视器（*security reference monitor*）（在第 8 章中介绍），将安全策略施加于本机计算机。它监控操作系统资源、执行运行时对象的保护和监视。

■ I/O 管理器（*I/O manager*）（在第 9 章中介绍）实现设备无关的 I/O 并负责为进一步处理调度到合适的设备驱动程序。

■ 即插即用（PnP）管理器（*Plug and Play manager*）（在第 9 章中介绍）确定支持特定设备所需要的驱动程序，并装入这些驱动程序。它在枚举（enumeration）时检索每一设备的硬件资

源需求。根据每一设备的资源需求，PnP 管理器分配合适的硬件资源，如 I/O 端口、IRQ、DMA 通道和存储单元。它还负责为设备变动（添加或删除设备）传送正确的事件通知。

■ 电源管理器（*power manager*）（在第 9 章介绍）协调电源事件，并为设备驱动程序产生电源管理 I/O 通知。当系统空闲时，电源管理器通过将 CPU 置于睡眠状态来减少电力消耗。个别设备耗电量的改变由设备驱动程序处理，但由电源管理器协调。

■ WDM 窗口管理装置例程（*WDM Windows Management instrumentation routine*）（在第 5 章介绍）公布设备驱动程序程性能和配置信息，并接收用户模式 WMI 服务命令。WMI 信息既可以从本机上也可以通过网络远程获得。

■ 高速缓存管理器（*cache manager*）（第 11 章介绍）提高基于文件 I/O 的性能，通过将最近引用的磁盘数据驻留在主存储器以便快速访问（在发送给磁盘之前，通过将更新数据在存储器内保留一般短暂的时间来延迟磁盘写）。你将会看到，这将通过使用内存管理器对映射文件的支持来完成。

■ 虚拟内存管理器（*virtual memory manager*）（第 7 章介绍）实现虚拟内存（*virtual memory*），它是一个存储管理方案，能为每一进程提供超过可用物理内存大小的专用地址空间。存储管理器还为高速缓存管理器提供基本支持。

另外，执行程序包括四组主要的支持函数，它们由上面列出的执行程序组件使用。1/3 的支持函数存档在 DDK 中，因为设备驱动程序也要使用它们。下面是四类支持函数：

■ 对象管理器（*object manager*），它创建、管理并删除 Windows 2000 执行对象和用来代表操作系统资源的抽象数据类型，如进程、线程和各种同步对象。对象管理器在第 3 章中介绍。

■ LPC 功能（*facility*）（在第 3 章介绍）在同一计算机上传递客户进程与服务器进程间的信息。LPC 是远程过程调用（RPC）的灵活、优化的版本，是客户和服务器进程通过网络通信的的商业标准。

■ 一系列广泛的公用运行时库（*run - time library*）函数，如串处理、算术运算、数据类型转化和安全结构处理。

■ 执行程序支持例程（*executive support routine*），如系统存储分配（页交换区和和非页交换区）、互锁内存存取（interlocked memory access）以及两种特殊类型的同步对象：资源（resource）和快速互斥（fast mutex）。

2.5.4 内核

内核由 Ntoskrnl.exe 中的一系列函数组成，它提供执行程序组件所使用的基本机制（如线程调度和同步服务）以及低级硬件体系结构的相关支持（如中断和异常调度），它们在每一处理器体系结构上是不同的。内核代码主要用 C 编写，为了访问不容易用 C 访问的专门处理器指令与寄存器，保留了汇编代码。

如前面部分所提到的各种执行支持函数，内核中的许多函数存档在 DDK 中（查找以 Ke 开头的函数），因为需要它们实现设备驱动程序。

1. 内核对象

内核提供定义好的低级库、可预测的操作系统原语（primitive）和机制，以便执行程序的

高级层组件做它们需要做的事情。内核通过实现操作系统机制和避免作出决策（policy making）将它本身与执行程序的其他部分分开。它将几乎所有的策略决定留给执行程序，除了由内核实现的线程安排与调度。

在内核外，执行程序将线程和其他可共享资源表达为对象。当它们创建时，这些对象需要一些策略开销，如操作它们的对象句柄（object handle）、保护它们的安全检查和演绎的资源配额。在内核中消除了这种开销，内核实现一些较简单的称为“内核对象”（*kernel object*）的对象，它帮助内核控制中心处理并支持执行程序对象的创建。大部分执行级对象封装一个或多个内核对象，合并它们的内核定义属性。

一组称为控制对象（control object）的内核对象建立控制各种操作系统函数的语义（semantic）。它包括 APC 对象、延迟过程调用（*deferred procedure call—DPC*）对象和 I/O 管理器使用的几个对象，如中断对象。

另一组称为调度程序对象（*dispatcher object*）的内核对象，集成了改变或影响线程调度的同步能力。调度程序对象包括内核线程、互斥体（mutex）（内部称为 mutant）、事件，内核事件对、信号量、计时器和可等待的计时器（waitable timer）。执行程序利用内核函数创建内核对象实例、操作它们、构造更为复杂的对象提供给用户模式。对象将在第 3 章详细介绍，进程与线程在第 6 章中介绍。

2. 硬件支持

内核的另一重要工作是将执行程序与设备驱动程序从 Windows 2000 支持的硬件体系结构的差异中抽象或隔离。该工作包括处理函数差异，如中断处理、异常调度和多处理器同步。

甚至对这些硬件相关的函数，内核设计也试图使公用代码数量最大。内核支持一系列在结构上语义相同并能移植的接口。实现可移植接口的大部分代码在体系结构上也是相同的。

一些接口在不同体系结构上的实现是不同的，或者说一些接口以特定结构代码部分实现。这些体系结构无关的接口可以在任何机器上调用，不论代码是否随体系结构而变化，接口语义是相同的。一些内核接口（如自旋锁（spinlock）例程，在第 3 章中介绍）实际上在 HAL 中实现（在下部分介绍），因为它们的实现在同一体系结构系列的系统中也会发生变化。

内核还包括少量特定 x86 接口的代码，它用来支持老的 MS - DOS 程序。这些 x86 接口不能移植，因为它们不能从基于其他体系结构的机器上调用；它们不会存在。例如，X86 特定代码支持操作全局描述符表（GDT）、LDT 和 X86 硬件特征的调用。

内核中的体系结构相关代码的其他实例包括提供变换缓冲器与 CPU 高速缓存器支持的接口。由于高速缓存器实现的方式，该支持对不同体系结构需要不同的代码。

另一实例是环境切换（context switching）。尽管在高级层上，线程选择与环境切换所用的算法是相同的（前一线程的环境（context）被保存，新线程的环境被加载，新线程启动），但在不同处理器上的实现存在结构差别。因为环境由处理器状态（寄存器等）描述，所以保存什么与加载什么随体系结构变化而变化。

2.5.5 硬件抽象层

在本章开头提到过，Windows 2000 设计的重点之一是在不同的硬件平台上移植。硬件抽象

层（HAL）是使该移植可行的关键部分。HAL 是可加载的内核模式模块（Hal.dll），它为 Windows 2000 运行的硬件平台提供低级接口。它隐藏硬件相关的细节，如 I/O 接口、中断控制器和多处理器通信机制——特定结构和机器相关的函数。

所以当 Windows 2000 需要平台相关信息时，不直接访问硬件，内部组件以及用户编写的设备驱动程序通过调用 HAL 例程获得移植。由于这一原因，HAL 例程存档在 Windows 2000 DDK 中。有关 HAL 的更多信息和设备驱动程序对它的使用参考 DDK。

尽管 Windows 2000 CD 包含几个 HAL（参见表 2-5），在安装时只有一个被选定并复制到系统盘上，文件名为 Hal.dll（其他操作系统如 VMS，在系统启动时选择等效的 HAL）。所以，如果支持另一处理器的 HAL 不同，不能假定从 x86 安装的系统盘会在不同处理器上引导。

表 2-5　HAL 列表

HAL 文件名	支持系统
Hal.dll	标准（Standard）PC
Halacpi.dll	高级配置和电源接口（Advanced Configuration and Power Interface—ACPI）PC
Halapic.dll	高级可编程中断控制器（Advanced Programmable Interrupt Controller—APIC）PC
Halaacpi.dll	APIC ACPI PC
Halmps.dll	多处理器 PC
Halmacpi.dll	多处理器 ACPI PC
Halborg.dll	Silicon 图形工作站（Graphics Workstation）
Halsp.dll	Compaq SystemPro

实验：确定你所运行的 HAL

有两种方法确定你所运行的 HAL：

1）打开文件 \ Winnt \ Repair \ Setup.log，找到 Hal.dll，查看等号后的文件名，这是分布式媒体上 HAL 的名字，来自 Driver.cab。

2）在 Device Manager 中（在桌面上右击 My Computer 栏，选择 Properties，点击 Hardware 选项，然后点击 Device Manager），查看计算机设备类型下的“driver”名字，例如，图 2-9 为运行 ACPI HAL 系统：

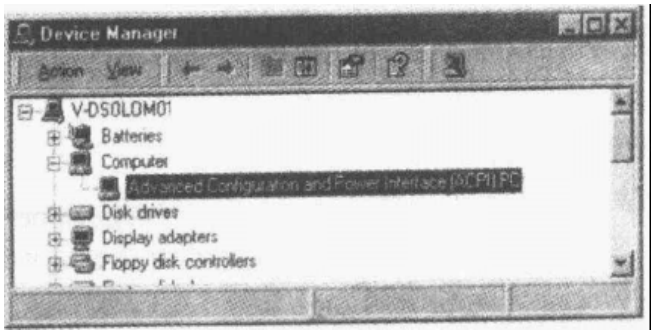


图 2-9　查看计算机设备类型下的“driver”名字

2.5.6　设备驱动程序

尽管设备驱动程序将在第 9 章中详细介绍，但在此我们有必要简单地讲述驱动程序类型的

简要配置，并解释如何列出系统上安装和加载的驱动程序。

设备驱动程序是可加载的内核模式模块（通常以 .sys 结尾），它是 I/O 管理器与相关硬件的接口。它们在内核模式的三种环境中运行：

■ 初始化 I/O 函数的用户线程环境。

■ 内核模式系统线程环境。

■ 作为中断的结果（所以不是任何特定进程或线程的环境——当中断发生时无论是什么样的当前进程或线程）。

如前部分所述，Windows 2000 设备驱动程序不直接操作硬件，但它们调用 HAL 中的函数来与硬件接口。驱动程序主要用 C 语言编写（有时用 C++），由于正确利用 HAL 例程，所以可以是在 Windows 2000 上支持 CPU 结构的可移植的源代码，或体系结构系列内可移植的二进制。

有几种类型的设备驱动程序：

■ 硬件设备驱动程序（*hardware device driver*）操作硬件（利用 HAL）输出或从物理设备或网络上输入。有许多类型的硬件设备驱动程序，如总线驱动程序（bus driver）、人机接口驱动程序、大容量存储驱动程序等等。

■ 文件系统驱动程序（*file system driver*）是 Windows 2000 驱动程序，它接收面向文件的 I/O 请求并将它们翻译为特定设备的 I/O 请求。

■ 文件系统过滤驱动程序（*file system filter driver*），如完成磁盘镜像（mirroring）和加密，截获（intercept）I/O，并在将 I/O 传送给下一层前完成一些增值处理。

■ 网络重定向器和服务器（*network redirector and server*）是文件系统驱动程序，它将文件系统 I/O 请求传送到网络上的机器，并相应地接收那些请求。

■ 协议驱动程序（*protocol driver*）实现网络协议如 TCP/IP、NetBEUI 和 IPX/SPX。

■ 内核流过滤驱动程序（*kernel streaming filter driver*）连接在一起完成数据流的信号处理，如记录或显示音频和视频。

因为安装设备驱动程序是将用户编写的内核模式代码添加到系统的唯一方法，一些程序员已经编写了设备驱动程序，它访问不能由用户模式访问的（但在 DDK 中存档并支持）内部操作系统函数或数据结构。例如 www.sysinternals.com 的许多实用程序组合了 Win32 GUI 应用程序与设备驱动程序，用于收集内部系统状态，但不能从 Win32 API 访问。

Windows 2000 设备驱动程序的增强

Windows 2000 添加了对即插即用（Plug and Play）、电源选项（Power Option）和称为 Windows Driver Model（WDM）的 Windows NT 驱动程序模型扩充的支持。Windows 2000 可以运行有效的 Windows NT 4 驱动程序，但因为它们不支持即插即用和电源选择，系统运行这些驱动程序将会在这两方面降低能力。

从 WDM 角度看，有三种类型的驱动程序：

■ 总线驱动程序（*bus driver*），维护总线控制器、适配器、桥或任何有子设备的设备。总线驱动程序是必要的驱动程序，微软一般提供；系统上每一类型的总线（如 PCI，PCMCIA，USB）有一个总线驱动程序。第三方可以写总线驱动程序以支持新的总线，如 VMEbus、Mutibus 和 Futurebus。

■ 功能驱动程序（*function driver*）是主要的设备驱动程序，它为其设备提供操作接口。它是必要的驱动程序，除非设备被原始的使用（一种 I/O 由总线驱动程序和任一总线过滤驱动程序完成的实现，如 SCSI PassThru）。功能驱动程序在定义上是指知道特定设备最多的驱动程序，它通常是能访问特定设备寄存器的唯一驱动程序。

■ 过滤驱动程序（*filter driver*）用来对设备（或已存在的驱动程序）添加函数功能或修改其他驱动程序的 I/O 请求和响应（通常用来调整提供错误硬件资源需求信息的硬件）。过滤驱动程序是可选的，以任意数量存在，存放在功能驱动程序上面或下面，总线驱动程序的上面。通常情况下，系统原始设备制造商（OEM）或独立硬件商（IHV）提供过滤驱动程序。

在 WDM 驱动程序环境中，没有一个单独驱动程序控制设备的所有方面：总线驱动程序主要将总线上的设备报告给 PnP 管理器，而功能驱动程序操作设备。

在大多数情况下，低级层过滤驱动程序改变设备硬件的行为。例如，如果设备向总线驱动程序报告它需要 4 个 I/O 端口，而实际上需要 16 个 I/O 端口，低级层特定设备操作过滤驱动程序将截取由总线驱动程序报告给 PnP 管理器的硬件资源，并更新 I/O 端口数。

高级层过滤驱动程序通常为设备提供增值特征。例如，键盘的高级层设备过滤驱动程序可以实施附加安全检查。

中断处理在第 3 章中介绍。I/O 管理器、WDM、即插即用及电源选项的详情将在第 9 章介绍。

实验：查看所安装的设备驱动程序

通过运行 Computer Management，可以列出所安装的驱动程序（在 Start 菜单中，选择 Programs，Administrative Tools，然后选择 Computer Management；或者在 Control Panel 中打开 Administrative Tools 并选择 Computer Management。）在 Computer Management 中，展开 System Information，在 Software Environment 中打开 Drivers。图 2-10 是所装驱动程序列表输出的例子。

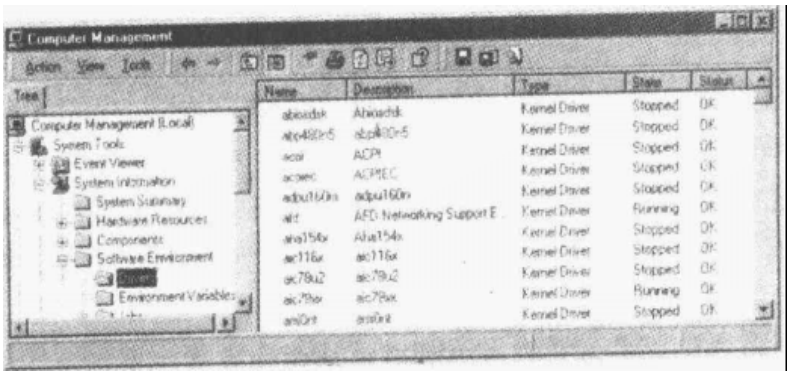


图 2-10　查看所安装的设备驱动程序

该窗口显示了在注册表中定义的设备驱动程序和它们的类型、状态（运行或停止）。设备驱动程序和 Win32 服务进程都定义在同一地方：HKLM \ SYSTEM \ CurrentControlSet \ Services。但是，它们由一个类型代码区分——类型 1 为内核模式设备驱动程序；类型 2 为文件系统驱动程序。有关存储在注册表中的设备驱动程序的更多细节请参见 Technical Reference to the Windows 2000 的 Registry 帮助文件（Regentry.chm），在 Windows 2000 资源工具包主题 HKEY – LOCAL – MACHINE \ SYSTEM \ CurrentControlSet \ Service 中。

另外，可以通过 Drivers 工具（Windows 2000 资源工具包中的 Drivers.exe）或 Pstat 工具（Platform SDK 中的 Pstat.exe）列出当前装载的设备驱动程序。下面是 Drivers 工具的部分输出，显示了每一装载的内核模式组件（Ntoskrnl、HAL 和设备驱动程序）及每一映像的部分的大小。

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

Pstat 工具也能显示装载的驱动程序列表，但要首先显示进程列表和各进程中的线程。Pstat 包括 Drivers 工具没有的一个重要信息——模块在系统空间中的装入地址。我们在后面将解释，这一地址对将运行系统线程映射到它们所在的设备驱动程序是很重要的。

2.5.7 查看没存档的接口

研究主要系统映像中的输出名字或全局符号，（如 Ntoskrnl.exe，Hal.dll 或 Ntdll.dll）会有启发——你会对 Windows 2000 所做事情的类型有所了解。在今天，被存档和被支持意味着什么呢？当然，你知道这些函数的名字，并不意味着你能调用它们——接口是没存档的，很容易变化。我们建议你查看这些函数，只为了深入了解 Windows 2000 执行内部函数的种类，不要绕过支持的接口。

例如查看 Ntdll.dll 中的函数列表，会看到给出的所有系统服务列表，它们是 Windows 2000 提供给用户模式子系统 DLL 的相应于每一子系统提供的子集。尽管许多函数明确映射到存档并受到支持的 Win32 函数，但有几个函数并不通过 Win32 API 提供（从 www.sysinternals.com 参见文章“Inside the Nature API”，它的副本在本书配套的 CD 中）。

研究 Win32 子系统 DLL（如 Kernel32.dll 或 Advpi32.dll）的导入和它们在 Ntdll 中调用哪些函数是很有意思的。

另一有趣的转储映像是 Ntoskrnl.exe——尽管内核模式设备驱动程序所用的许多输出例程存档在 Windows 2000 DDK 中，但有些不是。查看 Ntoskrnl 和 HAL 的导入表，会发现很有意思；

该表显示了 Ntoskrnl 使用的 HAL 函数列表，反之亦然。

表 2-6 列出了执行程序组件通常使用的大部分函数名前缀。每一主要的执行程序组件还利用前缀变化来表示内部函数——前缀的第一个字母后面接 i（表示 *internal*）或整个前缀后面接 p（表示 *private*）。例如，Ki 表示内部内核函数。*Psp* 表示内部进程支持函数。

表 2-6 常用的前缀

前缀	组 件
Cc	高速缓存管理器
Cm	配置管理器
Ex	执行程序支持例程
FsRtl	文件系统驱动程序运行时（run-time）库
Hal	硬件抽象层
Io	I/O 管理器
Ke	内核
Lpc	本机过程调用
Lsa	本机安全验证
Mm	存储管理器
Nt	Windows 2000 系统服务（大部分作为 Win32 函数输出）
Ob	对象管理器
Po	电源管理器
p	PnP 管理器
Ps	进程支持
Rtl	运行时（run-time）库
Se	安全
Wmi	Windows 管理装置
Zw	系统服务（以 Nt 开头）的镜像入口点，它为内核设置以前的访问方式、它消除参数验证、因为 Nt 系统服务只有当以前访问模式是用户时才验证参数

实验：列出没存档的函数

通过使用包含在 Windows 2000 Support Tools Platform SDK 中的 Dependency Walker 工具（Depends.exe），可以转储（dump）映像的输出与输入表。要想在 Dependency Walker 中查看一个映像，在 File 菜单中选择 Open 来打开所需的映像文件。

图 2-11 是通过该工具查看 Ntoskrnl 的 dependencies 的结果输出：

注意 Ntoskrnl 与 HAL 连接，HAL 依次与 Ntoskrnl 连接。（它们都使用各自的函数。）Ntoskrnl 也与 Bootvid.dll 连接，它是用来显示 Windows 2000 中新的 GUI 启动屏幕的引导视频驱动程序。

要想了解通过该工具显示信息的详细情况，参见 Dependency Walker 帮助文件（Depends.hlp）。

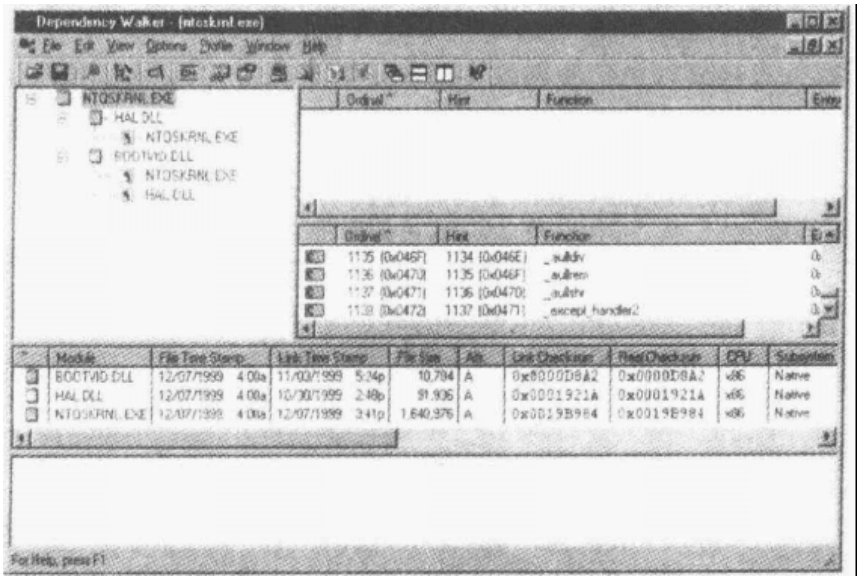


图 2-11 通过 Dependency Walk 工具查看 dependencies 的结果输出

了解 Windows 2000 系统例程的命名约定可以很容易地理解这些输出函数的名字。其一般格式为：

< 前缀（prefix）> < 操作（operation）> < 对象（object）>

在该格式中，前缀是输出例程的内部组件，操作通知对对象和资源所做的工作，对象标识操作的对象

例如：*ExAllocatePoolwithTag* 是执行程序支持例程，它在页交换区或非页交换区中分配，*KeInitializeThread* 是分配并设置内核线程对象的例程。

2.5.8 系统进程

下面的系统进程出现在所有 Windows 2000 系统上。（其中的两个——空闲（Idle）和系统（System）——不是完整的进程，它们不运行用户模式执行程序。

- 空闲进程（Idle process）。（在每个 CPU 包含一个线程来负责空闲 CPU 时间。）
- 系统进程（System process）（包括大部分的内核模式系统线程）。
- 会话管理器（Session manager— Smss.exe）。
- Win32 子系统（Csrss.exe）。
- 登录进程（Winlogon.exe）。
- 服务控制管理器（Services.exe）和它创建的子服务进程。
- 本机安全验证服务器（Lsass.exe）。

为了帮助理解这些进程间的关系，在 Windows 2000 Support Tool 中，使用 *tlist/t* 命令来显示进程“树”，即进程间的父/子关系。下面是通过 *tlist/t* 命令，得到的部分带注释的输出：

```
System Process (0)              Idle process
System (8)                      System process (default home for system threads)
  smss.exe (144)                Session Manager
    csrss.exe (172)             Win32 subsystem process
    winlogon.exe (192)          Logon process (also contains NetDDE service)
```

```
services.exe (220)      Service control manager
  svchost.exe (384)     Generic service host image
  spoolsv.exe (480)     Spooler service
  regsvc.exe (636)      Remote registry service
  mstask.exe (664)      Task Scheduler service
lsass.exe (232)         Local security authentication server
```

下面解释上面显示的主要系统进程。尽管该部分简要介绍了进程启动的次序，第 4 章还将详细介绍 Windows 2000 引导与启动所涉及的步骤。

1. 空闲进程（idle process）

在前面示例 *tlist/t* 输出中，列出的第一个进程（process ID 0）实际上是系统空闲进程。我们将在第 6 章介绍，进程通过它们的映像名来标识。但是该进程（还有称为 System 的 process ID 8,）并不运行实用户模式映像。因此，通过不同系统显示工具显示的名字是不同的。尽管大多数工具称 process ID 8 为“System”，但并不是所有的都是。表 2-7 给出了空闲进程（process ID 0）的几个名字。空闲进程将在第 6 章详细介绍。

表 2-7　不同实用工具中的 Process ID 0 的名字

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

现在让我们来了解系统线程和运行实映像（real image）的每一系统进程的目的。

2. 系统进程与系统线程

系统进程（始终是进程 ID 8）是只在内核模式运行的特殊类型线程的宿主：内核模式系统线程（*kernel - mode system thread*）。系统线程具有普通用户模式线程的所有属性和环境（如硬件环境、优先权等）。不同的是系统线程仅在内核模式运行，执行装载在系统空间的代码，不论它是在 Ntoskrnl.exe 中还是在其他装载的设备驱动程序中。另外，系统线程没有用户进程地址空间，因此，必须从操作系统存储堆如从页交换区或非页交换区分配动态存储区。

系统线程由 *PsCreateSystemThread* 函数创建（存档在 DDK 中），它只能从内核模式调用。Windows 2000 和其他设备驱动程序在系统初始化时创建系统线程，执行需要线程环境的操作，如发送和等待 I/O 或其他对象及轮询设备。例如内存管理器利用系统线程来实现如下功能，将无效页（dirty page）写到页文件或映射文件、在内存内外交换进程等等。内核创建称为平衡集管理器（*balance set manager*）的系统进程，它每秒钟都唤醒一次并可能初始化各种调度和内存管理相关事件。高速缓存管理器也使用系统线程来实现先读后写 I/O。文件服务器设备驱动程序（srv.sys）利用系统线程响应网络上共享的磁盘分区上的文件数据的网络 I/O 请求。甚至软盘驱动程序也有系统线程来轮询软盘设备（轮询在这种情况下更有效，因为中断驱动的软盘驱

动程序消耗大量的系统资源)。有关特定系统线程的更多信息将在本章中的每一组件中介绍。

在默认情况下，系统线程属于系统进程，但设备驱动程序能在任一进程中创建系统线程。如 Win32 子系统设备驱动程序（Win32k.sys）在 Win32 子系统进程（csrss.exe）中创建系统线程。这样，它们就能很容易地访问该进程的用户模式地址空间的数据。

当排除故障或检查系统分析时，能将个别系统线程的执行映射回驱动程序或包含该代码的子例程将是很有用的。例如，在负载超重的文件服务器上，系统进程可能会消耗相当多的 CPU 时间。但是，当系统进程运行时，知道一些系统在运行系统线程并不能足以确定哪一设备驱动程序或操作系统组件在运行。

因此，如果系统进程在运行，查看该进程的线程执行（例如，使用 Performance 工具）。一旦找到在运行的线程，查看该系统线程在哪一驱动程序开始执行（它至少告诉你哪一驱动程序可能创建该线程），或者查询线程的调用栈（call stack）（或至少当前地址），它会指出线程当前在哪执行。

这两项技术将在下面的实验中说明。

实验：识别系统进程中的系统线程

你将会发现系统进程中的线程一定是内核模式系统线程，因为每一线程的起始地址大于系统空间的起始地址（默认情况下，它起于 0x80000000，除非系统是通过/3GB Boot.ini-switch 引导的）。同时，如果你查看这些线程的 CPU 时间，你将会发现那些累积所有 CPU 时间的线程仅在内核模式运行。

要找到创建系统线程的驱动程序，查看线程的起始地址（可以通过 Pviewer.exe 显示），并查找其基地址与线程起始地址最接近（但在线程起始地址前面）的驱动程序。Pstat 工具（在它输出的最后）和！*drivers* 内核调试程序命令都会列出加载的每一设备驱动程序的基地址。

为了快速找到线程的当前地址，使用内核调试程序中的！*stacks* 0 命令。下面是一个正在运行的系统的输出例子（使用 *LiveKd*）：

```
kd> !stacks 0
Proc.Thread  Thread  ThreadState  Blocker
                                  [System]
 8.000004  8146edb0  BLOCKED  ntoskrnl!MmZeroPageThread+0x5f
 8.00000c  8146e730  BLOCKED  ?? Kernel stack not resident ??
 8.000010  8146e4b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000014  8146d030  BLOCKED  ?? Kernel stack not resident ??
 8.000018  8146ddb0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00001c  8146db30  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000020  8146d8b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000024  8146d630  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000028  8146d3b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00002c  8146c030  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000030  8146cdb0  BLOCKED  ntoskrnl!ExpWorkerThreadBalanceManager+0x55
 8.000034  8146b470  BLOCKED  ntoskrnl!MiDereferenceSegmentThread+0x44
 8.000038  8146b1f0  BLOCKED  ntoskrnl!MiModifiedPageWriterWorker+0x31
```

```
8.00003c  8146a030  BLOCKED  ntoskrnl!KeBalanceSetManager+0x7e
8.000040  8146adb0  BLOCKED  ntoskrnl!KeSwapProcessOrStack+0x24
8.000044  8146a5b0  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
8.000048  8146a330  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
8.00004c  81461030  BLOCKED  ACPI!ACPIWorker+0x46
8.000050  8143a770  BLOCKED  ntoskrnl!MiMappedPageWriter+0x4d
8.000054  81439730  BLOCKED  dmio!voliod_loop+0x399
8.000058  81436c90  BLOCKED  NDIS!ndisWorkerThread+0x22
8.00005c  813d9170  BLOCKED  ltmdmntt!WakeupTimerThread+0x27
8.000060  813d8030  BLOCKED  ltmdmntt!WriteRegistryThread+0x1c
8 000070  8139c850  BLOCKED  raspptp!MainPassiveLevelThread+0x78
8.000074  8139c5d0  BLOCKED  raspptp!PacketWorkingThread+0xc0
8.00006c  81384030  BLOCKED  rasacd!AcdNotificationRequestThread+0xd8
8.000080  81333330  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x6f
8.000084  813330b0  BLOCKED  rdbss!RxSpinUpRequestsDispatcher+0x58
8.00008c  81321db0  BLOCKED  ?? Kernel stack not resident ??
8.00015c  81205570  BLOCKED  IN0_FLTR+0x68bd
8.000160  81204570  BLOCKED  IN0_FLTR+0x80e9
8.000178  811fcdb0  BLOCKED  irda!RxThread+0xfa
8.0002d0  811694f0  BLOCKED  ?? Kernel stack not resident ??
8.0002d4  81168030  BLOCKED  ?? Kernel stack not resident ??
8.000404  811002b0  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x5f
8.000430  810f4990  READY    parallel!ParallelThread+0x3e
8.00069c  80993030  READY    rdbss!RxpWorkerThreadDispatcher+0x5f
```

第一栏是进程ID和线程ID（"process ID.thread ID"的形式）；第二栏是线程的当前地址；第三栏表明线程是等待状态、就绪状态还是运行状态；（线程状态的描述参见第6章。）最后一栏是在线程栈的最前面地址。最后一栏的信息使你容易明白各个线程在哪一驱动程序开始，对于Ntoskrnl中的线程，函数的名字进一步提示了线程所做的事情。

但是，如果运行线程是系统工作者线程（system worker thread）之一（ExpWorkerThread），则不知道线程在做什么，因为任一设备驱动程序向系统工作者线程提交作业。因此，跟踪系统工作者线程活动的唯一方式是在 *ExQueueWorkItem* 中设置断点（breakpoint）。当到达断点时，键入！*dso work _queue _item esp +4*。该命令将转储给 *ExQueueWorkItem* 的第一个参数（工作队列结构），它依次包含了在工作线程环境中被调用的工作者例程地址。或者，你可以通过内核调试程序中的k命令查看调用程序，它将显示当前调用栈。当前调用栈将显示把工作加入工作者线程的驱动程序（与被工作者线程调用的例程相对）。

实验：将系统线程映射为设备驱动程序

在该实验中，我们将查找Raw Mouse Input线程，它是Win32子系统设备驱动程序（Win32k.sys）中的线程，决定将鼠标移动和事件通知给哪些线程。要运行该系统线程，只要来回快速移动鼠标并监视进程CPU时间（使用Task Manager、Performance工具或Windows 2000资源工具的QuickSlice工具），注意Csrss进程运行的时间很短暂。

下面的步骤表明如何降低线程粒度，以找到包含正在运行系统线程的驱动程序。尽管该例演示Csrss进程中的系统线程活动，该技术也适用于将系统进程中的系统线程映射回创建该线程的设备驱动程序。

首先，我们需要设置 Performance 工具，在 System Monitor 中观察各个系统线程的活动：

1）运行 Performance 工具。

2）选择 System Monitor，并点击 Add 按钮（工具栏中的 plus 符号）。

3）在 Performance Object 下拉列表框中选择 Thread 对象

4）选择% Processor Time 计数器或% Privileged Time，它们具有相同值。

5）在 Instance 框中，向下滚动鼠标到名为 csrss/0 的线程。在该项上点击，向下拖鼠标，直到你选中 csrss 进程中的所有线程（如果正在监视系统进程中的线程，要选择系统进程中从 System/0 到最后标号的线程）。

6）点击 Add 按钮。

7）这时，你的屏幕看起来如图 2-12 所示。

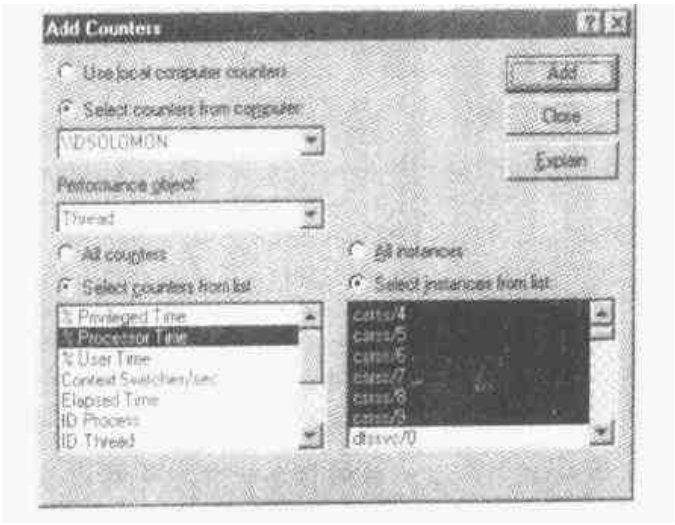


图 2-12 设置 Performance 工具

8）点击 Close。

现在，从 100 到 10 改变竖向比例最大值。这种改变将会更容易地看到系统线程活动，因为系统线程一般运行很短的时间周期。要改变竖向比例最大值，按下面的步骤进行：

9）在图形上右击鼠标。

10）选择 Properties。

11）点击 Graph 选项。

12）将 Vertical Scale Maximum 从 100 改为 10。

13）点击 OK。

现在我们将监视 Csrss 进程，通过移动鼠标，人工地产生一些活动，以使得 Win32k.sys 设备驱动程序中的线程运行。然后，我们将确定当移动鼠标时，哪一驱动程序创建了正在运行的系统线程。

14）快速来回移动鼠标，直到在 Performance 工具的显示中看到有一个或两个运行的系统线程。

15）按下 Ctrl + H 以打开加亮状态（它将当前选中的计数器加亮为白色）。

16）在计数器中滚动鼠标以确定移动鼠标时运行的线程（可能不只一个线程运行）。

17）在 Performance 工具的图形视窗的底部，注意 Instance 栏中的相关线程号（图 2-13csrss 中的线程 7）。

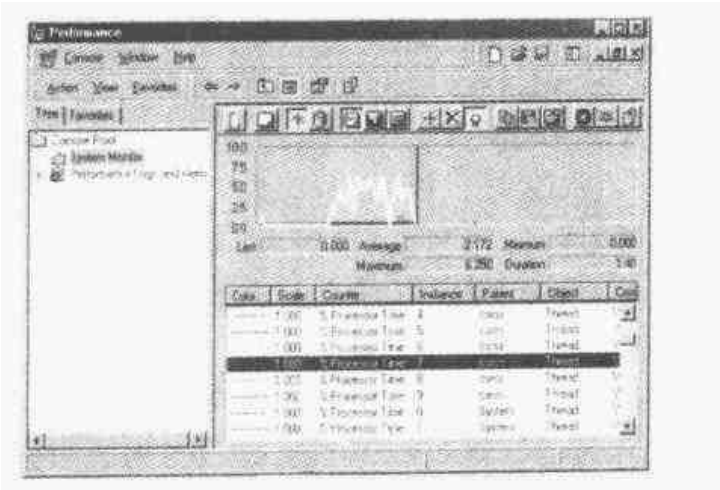


图 2-13　监视 Csrss 进程，通过移动鼠标，确定哪一驱动程序创建了正在运行的系统线程

在该例中，csrss 中的线程 7 是运行的。在 Windows 2000 的 Support Tools 中，通过 Process Viewer，将会找到该线程的起始地址。

18）运行 Process Viewer。

19）在进程列表框显示中，点击 csrss 进程。

20）在线程列表中滚动鼠标，直到你找到与第 17 步具有相同的相关线程号的线程。

21）选择该线程。

如果你找到 csrss 中的一个系统线程，它的起始地址（显示在 Process Viewer 视窗的 Thread Information 区域底部）应该是 0xa0009cbf。（如果运行的是 Windows 2000 的更新版本，该地址可能会不同。）

22）最后，运行 Pstat.exe，它可以从 Platform SDK 或 www.reskit.com 网站获得。

在 Pstat 输出的最后，是所加载的各个设备驱动程序的列表，包括它在系统虚拟内存中的起始地址。找到该驱动程序，它在所要询问线程起始地址之前，具有最接近的起始地址。下面是 Pstat 工具最后一部分的摘录：

```
ModuleName   Load Addr   Code    Data   Paged          LinkDate
-------------------------------------------------------------------
ntoskrnl.exe 80400000  429184   96896  775360 Tue Dec 07 18:41:11 1999
     hal.dll 80062000   25856    6016   16160 Tue Nov 02 20:14:22 1999
 BOOTVID.DLL F7410000    5664    2464       0 Wed Nov 03 20:24:33 1999
    ACPI.sys BFFD8000   92096    8960   43488 Wed Nov 10 20:06:04 1999
  WMILIB.SYS F75C8000     512       0    1152 Sat Sep 25 14:36:47 1999
     pci.sys F7000000   12704    1536   31264 Wed Oct 27 19:11:08 1999
  isapnp.sys F7010000   14368     832   22944 Sat Oct 02 16:00:35 1999
:
  win32k.sys A0000000 1520960   54944       0 Tue Nov 30 03:51:03 1999
   rdbss.sys BECBF000   27B08    1952   86656 Tue Nov 30 03:52:29 1999
  mrxsmb.sys BEBF7000   91616   21824  237568 Tue Nov 30 03:52:10 1999
```

```
:
   ntdll.dll 77F80000  294912   12288   16384 Wed Oct 27 16:06:08 1999
-------------------------------------------------------- -------
      Total          4375648  547040 3164960
```

所要询问线程的起始地址 0xa0009cbf，很明显是 Win32k.sys 的一部分，因为没有其他驱动程序具有更接近的起始地址。

如果起始地址在 Ntoskrnl.exe 内，那么线程在执行主内核映像的子例程。仅有该信息并不能足以确定线程所做的事情，你还需要找到该地址上的函数名。可以通过查找全局符号列表来实现，它包含在关联符号表文件 Ntoskrnl.dbg 中。

在 Ntoskrnl 中生成全局符号列表的最简单方法是启动内核调试程序（通过连接到在线系统或通过打开崩溃转储文件），并在装有 Ntoskrnl.dbg 的内核调试程序中键入 *x nt!* * 命令。在键入该命令前，请用 *.logopen* 命令创建调试内核会话登录文件。这样，你可以将输出保存到一个文件中，然后查找要询问的地址。也可以用 Visual C++ Dumpbin 工具（键入 *dumpbin/symbols ntoskrnl.dbg*），但是必须查找减去 Ntoskrnl 基地址的地址，因为它只列出了偏移量。

或者和前面的实验一样，内核调试程序的 *!stacks 0* 命令也可以用来显示线程当前地址上的驱动程序和函数名（假如已加载了合适的符号）。

3. 会话管理器（Session Manager—Smss）

会话管理器（\Winnt\System32\Smss.exe）是系统创建的第一个用户模式进程。完成执行程序和内核初始化最后步骤的内核模式线程创建实际的 Smss 进程。

会话管理器负责启动 Windows 2000 的一些重要步骤，如打开附加页面文件、执行延迟文件重命名和删除操作、创建系统环境变量。它还运行子系统进程（通常指 Csrss.exe）和 Winlogon 进程，反过来创建其他的系统进程。

注册表中驱动程序 Smss 初始化步骤的大部分配置信息可以在 HKLM\SYSTEM\CurrentControlSet\Control\Session Manager 下找到。你会发现，查看存储在这里的数据类型是很有趣的。（关于键与键值的描述请查看 Registry Entries 帮助文件，Regentry.chm 在 Windows 2000 的资源工具包中。）

完成这些初始化步骤之后，Smss 的主要线程将在 Csrss 和 Winlogon 进程句柄上永远等待。当进程意外中断时，Smss 将使系统崩溃，因为 Windows 2000 依赖于它们的存在。同时，Smss 等待装载子系统、新子系统启动和调试事件的请求。它还作为应用程序与调试程序之间的转换器和监视程序。

4. 登录（Logon—Winlogon）

Windows 2000 的登录进程（\Winnt\System32\Winlogon.exe）处理交互式的用户登录与退出。当安全提示序列（*secure attention sequence*—*SAS*）按键组合被输入时，Winlogon 被通知用户登录请求。在 Windows 2000 中，默认 SAS 是 Ctrl + Alt + Delete 的组合。SAS 用来保护用户，以防口令捕捉程序模拟登录进程。一旦用户名与口令被捕捉，它们将被发送到本机安全验证服务器进程（将在下部分介绍）加以验证。如果它们匹配，Winlogon 在注册表键 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon 下抽取 Userinit 注册表键值，并创建…

进程以运行该键值列出的每一可执行映像。在默认情况下运行名为 Userinit.exe 的进程。

实验：查看多重会话

如果安装了终端服务（terminal service），会话管理器会为每一连接用户会话创建单独的 Csrss 和 Winlogon 实例。会话列表可以用内核调试程序中的！session 命令显示。在下面的例子中，系统有三个会话活动（！session 命令仅显示每一会话中的 Csrss.exe 进程，尽管在每一会话中有很多进程）。

```
kd> !session
**** NT ACTIVE SESSION DUMP ****
PROCESS 813531e0  Cid: 00c8    Peb: 7ffdf000  SessionId: 00000000
    DirBase: 03639000  ObjectTable: 81353508  TableSize: 371.
    Image: csrss.exe

PROCESS 81180c80  Cid: 0364    Peb: 7ffdf000  SessionId: 00000001
    DirBase: 00fcc000  ObjectTable: 812c7288  TableSize: 115.
    Image: csrss.exe

PROCESS 811358e0  Cid: 0618    Peb: 7ffdf000  SessionId: 00000002
    DirBase: 040f0000  ObjectTable: 8114bcc8  TableSize: 111.
    Image: csrss.exe
```

该进程完成用户环境的某些初始化（如恢复映射的驱动程序器名、运行注册过程、应用组策略（group policy）），然后在注册表中查看 Shell 键值（参考前面，在同样的 Winlogon 键下面），并创建进程以运行系统定义的 shell（默认情况下为 Explorer.exe）。然后，Userinit 退出。这就是显示的 Explorer.exe 没有父进程的原因——它的父进程已经退出，如前面所解释的，Tlist 左侧列出父进程不在运行的进程（另一种情况是 Explorer 是 Winlogon 的孙进程）。

登录进程的验证在名为 GINA（Graphical Identification and Authentication）的可替代的 DLL 中实现。标准的 Windows 2000 GINA，Msgina.dll，实现默认的 Windows 2000 登录接口。然而，开发者可以提供它们自己的 GINA DLL 来实现其他的验证机制，以代替标准的 Windows 2000 用户/口令方式（如基于声纹的方式）。另外，Winlogon 可以安装附加的需要实现二次验证的网络提供程序 DLL。这种能力允许多个网络提供程序在正常登录情况下同时收集验证信息。

Winlogon 不仅在用户登录与退出时是活动的，而且在截获来自键盘的 SAS 时也是活动的。例如，在登录时，当你按下 Ctrl + Alt + Delete 时，Windows Security 对话框就会出现，提供选项退出、启动 Task Manager、锁定工作站、关闭系统等等。Winlogon 是处理这种交互的进程。

有关 Winlogon 的详细情况，参见第 8 章。

5. 本机安全验证服务器（Local Security Authentication Server，LSASS）

本机安全验证服务器进程（\Winnt\System32\Lsass.exe）接收来自 Winlogon 的验证请求，并调用合适的验证程序包（与 DLL 实现一样）来执行实际的验证，如检查口令是否与 active directory 或 SAM（注册部分中包含用户与组定义的部分）中所存储的相匹配。

一旦成功验证后，Lsass 生成一个包含用户安全配置文件（profile）的访问令牌对象。然后 Winlogon 利用该访问令牌创建初始 shell 进程。从 shell 启动的进程在默认情况下继承该访问令牌。

有关 lsass 和安全验证的详细情况，参见第 8 章。有关与 Lsass 接口的可调用函数（以 *Lsa* 开头的函数）的详细情况，参见 Platform SDK 下的文档。

6. 服务控制管理器（Service Control Manager，SCM）

回想本章前面的 Windows 2000 "服务"，可以参考服务器进程或者设备驱动程序。该部分讨论用户模式进程的服务。服务如同 UNIX 的 "守护进程（daemon process）" 或 VMS 的 "分离进程（detached process）"，因为在系统引导时不需要交互登录，可以设置为自动启动。也可通过人工启动（如运行服务管理工具或调用 Win32 StartService 函数）。一般来说，服务对登录的用户没有影响，尽管在一些特殊情况下是可能的（参见第 5 章）。

服务控制管理器是一个特殊的系统进程，它运行 \ Winnt \ System32 \ Services.exe 映像，该映像负责启动、停止服务进程以及和服务进程相互作用。服务程序是真正的 Win32 映像，它调用特殊 Win32 函数与服务控制管理器交互作用，以完成如下活动：如记录服务的成功启动，对状态请求作出响应、暂停或关闭服务。服务定义在 HKLM \ SYSTEM \ CurrentControlSet \ Services 下的注册表中。资源工具包 Registry Entries 帮助文件（Regentry.chm）记录服务的子键和键值。

记住服务有 3 个名字：在系统上运行时显示的进程名字、注册表中的内部名字、在 Services 管理工具中显示的名字（并非所有的服务都有显示名字，如果服务没有显示名字，就会显示内部名字）。在 Windows 2000 中，服务还会有一个描述区，它包含多达 1024 个字符来描述服务所做事情的细节。

使用 *tlist /s* 命令将服务进程映射为包含在进程中的服务。注意在服务进程与运行服务间并非总是一一对应的映射，因为某些服务与其他服务共享同一进程。在注册表中，类型代码表明了哪些服务在自己的进程中运行，哪些服务与映像上的其他服务共享一个进程。

Windows 2000 的一些组件是作为服务实现的，如 spooler、事件日志（Event Log）、任务调度程序（Task Scheduler）以及其他各种网络组件。

实验：列出安装的服务

要列出安装的服务，在 Control Panel 中选择 Administrative Tools，然后选择 Services。你将会看到如图 2-14 所示的输出：

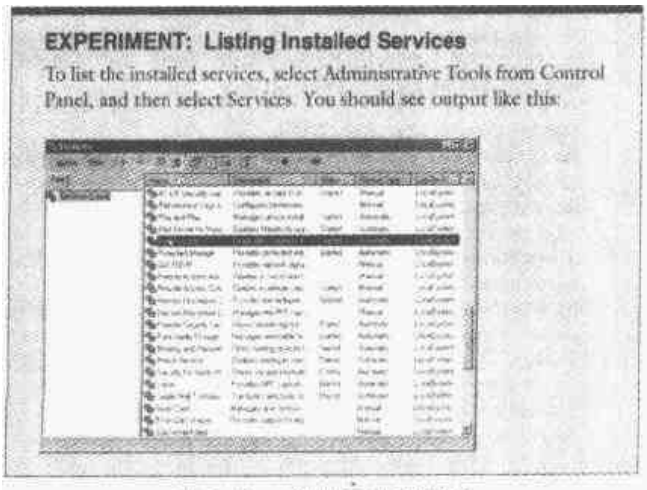


图 2-14 列出安装的服务

要查看一个服务的详细属性，在服务上右击并选择 Properties。例如，如图 2-15 所示是 Print Spooler 服务（在前图加亮的服务）的属性。

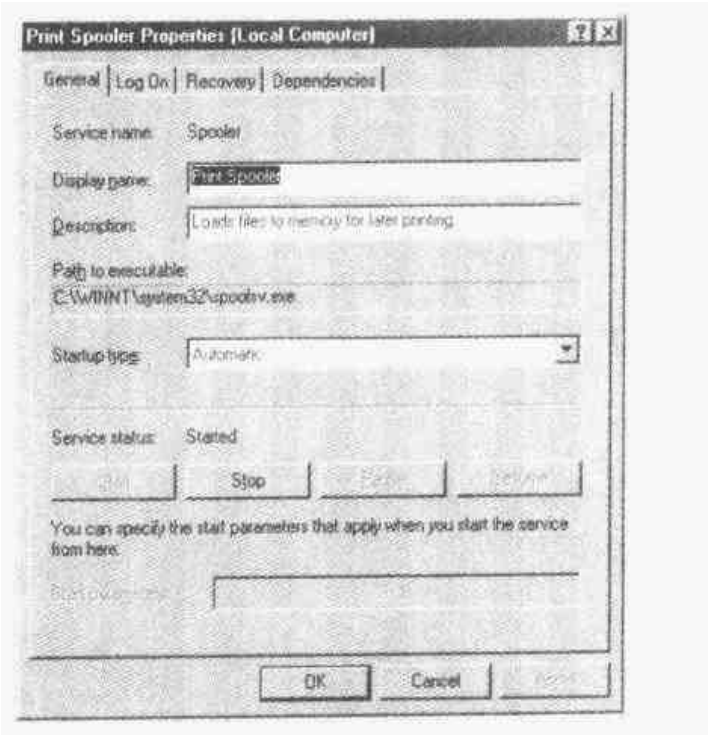


图 2-15 查看 Print Spooler 服务的详细属性

注意 Path To Executable 字段，以确定包含该服务的程序。请记住某些服务与其他服务共享同一进程——映射并非总是一一对应的。

要了解有关服务的更详细情况，请参见第 5 章。

2.6 小结

本章广泛分析了 Windows 2000 的总体体系结构，研究了 Windows 2000 的主要组件以及它们是如何相互联系的。在下章中，将更详细地研究这些组件所建立的内核系统机制，如对象管理器和同步。

第3章 系统机制

Microsoft windows 2000 提供了诸如执行程序、内核以及设备驱动程序等内核模式组件所使用的一些基本机制。本章将对以下系统机制进行解释并讲述它们的使用方法：

- 陷阱调度，包括中断、延迟过程调用（DCP）、异步过程调用（APC）、异常调度和系统服务调度
- 执行程序对象管理器。
- 同步，包括自旋锁、内核调度程序对象和等待的实现方式。
- 系统工作者线程。
- 各种各样的机制如 Windows 2000 全局标记（global flag）。
- 本机过程调用（LPC）。

3.1 陷阱调度

中断和异常是操作系统使处理器转向正常控制流以外的编码的情况。无论是硬件还是软件都能检测到。“陷阱”一词是指当异常或中断发生时，处理器捕捉正在执行的线程，并将控制转向操作系统固定位置的机制。在 Windows 2000 中，处理器将控制转向陷阱处理程序（trap handler），它是与特定中断或异常相关的函数。图 3-1 列出了一些激活陷阱处理程序的条件。

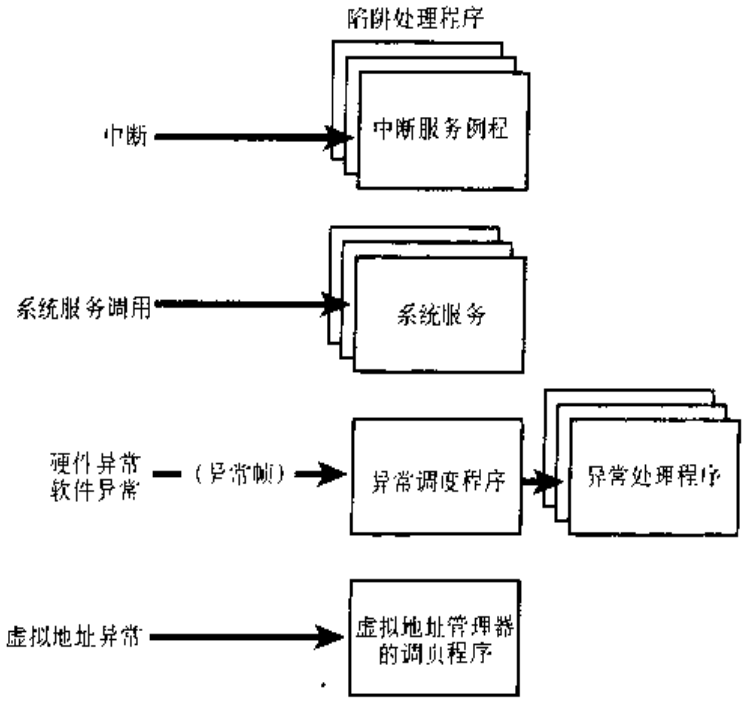


图 3-1 陷阱调度

内核以下列方式区分中断和异常。中断是异步事件（可在任何时候发生），它与处理器正在执行什么无关。中断主要是由 I/O 设备、处理器时钟或定时器引发的，而且能被允许（开）

或取消（关）。相反，异常是由于执行特定指令而引起的同步事件。在相同条件下，第二次运行相同数据的程序能重复异常。异常现象包括存储器存取违规、一定的调试程序指令和被0除错误。内核也把系统服务调用视为异常（尽管从技术上说它们是系统陷阱）。

无论是硬件还是软件都会引起异常和中断。例如，总线错误异常是由硬件出故障引起的，而除0异常是软件错误引起的。类似的，I/O设备也会产生中断，或者内核本身也能引起软中断（如本章后文将要讲到的ACP或DCP）。

当产生硬件异常或中断时，处理器记录足够的机器状态以使能返回控制流中的该点、而恢复执行，好像异常和中断没发生一样。为达到这一目的，处理器在中断线程的内核栈中创建一陷阱帧（trap frame），并将线程的执行状态存储在中断线程的内核栈中。陷阱帧通常是全部线程环境的子集（线程环境将在第6章介绍）。当线程调用与软中断相关的内核函数时，内核将软中断作为硬件中断处理的一部分进行处理或者同步处理。

大多数情况下，内核在将控制转移给其他产生处理陷阱的现场的函数之前以及之后，都要安装前端陷阱处理函数，用以完成一般的陷阱处理任务。例如，如果是设备中断，内核硬件中断陷阱处理程序将控制转移给中断服务例程（ISR），它由设备驱动程序为中断设备提供。如果中断是由调用系统服务引起的，一般系统服务陷阱处理程序就将控制转移给正在执行中的特定系统服务函数。内核也为无法预测或无法处理的陷阱建立陷阱处理程序。这些陷阱处理程序通常执行系统函数KeBugCheckEx。只要内核检测到有问题或错误的行为，便停止计算机而如果不对其进行检查，将引起数据出错（关于错误检查的进一步资料，参见4.5节“系统崩溃”）。以下几节将更详细的讲述中断、异常和系统服务调度。

3.1.1 中断调度

硬件引起的中断通常来自I/O设备，它们在需要服务时必须通知处理器。中断驱动设备允许操作系统通过中央处理与I/O操作重叠而最大限度地使用处理器。线程启动从或到某一设备的I/O传输，在设备完成传输过程中执行有效工作。当该设备传输完成后，它就中断处理器请求服务。定点设备、打印机、键盘、磁盘驱动器和网卡一般都是中断驱动的。

系统软件也能产生中断。比如，内核能引起软中断，从而初始化线程调度并异步地中断线程的执行。为使处理器不被中断，内核也能禁止中断，但这是很少出现的——只有在临界点期间，如当它在处理某个中断或调度某一异常时。

内核安装了陷阱处理程序来响应设备中断。中断陷阱处理程序将控制转移给处理中断的外部例程（ISR）或者转移给响应中断的内部内核例程。设备驱动程序将中断服务例程（ISR）提供给服务设备中断，而内核则为其他类型的中断提供中断处理例程。

在以下的章节中，你将会了解到硬件如何向处理器通知设备中断，内核所支持的中断类型、设备驱动程序与内核交互的方式（作为中断处理的一部分）、以及内核所能识别的软中断（加上用来实现这些的内核对象）。

1. 硬件中断处理

在x86系统上，外部I/O中断进入到中断控制器的某条线中。接着控制器中断某条线上的处理器。一旦处理器被中断，它就查询控制器以获取中断请求（IRQ）。中断控制器再将IRQ翻译为一个中断号，用它作为在中断调度表（IDT）结构的索引，并把控制转移给适当的中断调度例程。

在系统启动时，Windows 2000 在 IDT 中填入指向处理中断和异常的内核例程的指针。

实验：查看 IDT

利用在Kdex2x86.dll调试器扩展库中实现的!idt命令，读者可以查看IDT的内容，包括有关 Windows 2000 分配给中断（异常和 IRQ）陷阱处理程序的信息。将 0 标记传递给！idt 命令表示为硬件设备中断注册的设备驱动程序的 ISR

下例说明了如何装载 Kdex2x86. dll 调试器扩展库以及！idt 命令的输出结果：

```
kd> .load kdex2x86
Loaded kdex2x86 extension DLL
kd> !idt 0
    00: 80463440 (ntkrnlmp!KiTrap00)
    01: 80463590 (ntkrnlmp!KiTrap01)
    02: 0000144e
    03: 8046386c (ntkrnlmp!KiTrap03)
    04: 804639d4 (ntkrnlmp!KiTrap04)
    05: 80463b18 (ntkrnlmp!KiTrap05)
    06: 80463c78 (ntkrnlmp!KiTrap06)
    07: 804641bc (ntkrnlmp!KiTrap07)
    08: 000014a8
    09: 80464558 (ntkrnlmp!KiTrap09)
    0a: 80464660 (ntkrnlmp!KiTrap0A)
    0b: 8046478c (ntkrnlmp!KiTrap0B)
    0c: 80464a90 (ntkrnlmp!KiTrap0C)
    0d: 80464c9c (ntkrnlmp!KiTrap0D)
    0e: 80465708 (ntkrnlmp!KiTrap0E)
    0f: 80465aac (ntkrnlmp!KiTrap0F)
    10: 80465bb4 (ntkrnlmp!KiTrap10)
    11: 80465cd8 (ntkrnlmp!KiTrap11)
    12: 80465aac (ntkrnlmp!KiTrap0F)
    :
    29: 00000000
    2a: 804628fe (ntkrnlmp!KiGetTickCount)
    2b: 804629f0 (ntkrnlmp!KiCallbackReturn)
    2c: 80462b10 (ntkrnlmp!KiSetLowWaitHighThread)
    2d: 8046375c (ntkrnlmp.KiDebugService)
    2e: 80462420 (ntkrnlmp K'SystemService)
    2f: 80465aac (ntkrnlmp!KiTrap0F)
    30: 80461a50 (ntkrnlmp!KiStartUnexpectedRange)
    :
    51: 80461b9a (ntkrnlmp!KiUnexpectedInterrupt33)
    52: 813dbbe4 (Vector:52,Irql:4,SyncIrql:5,Connected:TRUE, →
                  No:0,ShareVector:FALSE,Mode:Latched,        →
                  ISR:i8042prt!I8042KeyboardInterruptService(f04b10cc))
    53: 80461bae (ntkrnlmp!KiUnexpectedInterrupt35)
    54: 80461bb8 (ntkrnlmp!KiUnexpectedInterrupt36)
    :
    c0: 80462090 (ntkrnlmp!KiUnexpectedInterrupt60)
    c1: 800638d4 (halmps!HalpClockInterrupt)
    d2: 804620a4 (ntkrnlmp!KiUnexpectedInterrupt62)
    :
```

例子中一些有趣的中断号在0x0到0x10范围和0x2a到0x2e的范围内，前面的范围包括x86异常中断（如页面错误是0xe异常，由KiTrap0E处理），后者则包括系统服务调度程序和其他被内核在内部用作从环境子系统到内核的快速入口点的软中断。在用来提供这个实验结果的系统上，时钟中断程序的中断号是0xd1，键盘设备驱动程序的键盘ISR的中断号是0x52。

Windows 2000将硬件IRQ映射为IDT中的中断号，系统也利用IDT为异常配置陷阱处理程序。例如，页面错误（当线程作业试图访问没定义或不存在的虚拟内存页时出现的异常）的x86异常号是0xe。因此，IDT中的0xe入口就指向系统页面错误处理程序。尽管x86体系结构能支持多达256个IDT入口，但具体机器所能支持的IRQ数量却由机器所使用的中断控制器的设计决定。

大多数x86系统要么依赖于i8259A可编程中断控制器（PIC），要么依赖于i8259A高级可编程控制器（APIC）的变种。多数新型计算机都含有APIC。PIC标准起源于最初的IBM PC。而APIC在多处理器上运行，带有256根中断线。Intel和其他公司都确定了多处理器规范（MP Specification），它是围绕使用APIC的x86多处理器系统的设计标准。为了能与单处理器操作系统兼容和提供在单处理器模式下启动多处理器系统的引导代码，APIC支持带有15个中断并仅把中断发送给主处理器的PIC兼容模式。图3-2画出了APIC的结构。事实上APIC包括以下几部分：接收来自设备的中断的I/O APIC，接收来自I/O APIC的中断和与它们相关的CPU的本机APIC以及可兼容i8259A的中断控制器（它把APIC输入转换成等价的PIC信号）。I/O APIC负责实现中断路由算法——该算法是软件可选择的（在Windows 2000里，HAL做出选择）——既平衡了处理器之间的负载，又尽量利用局部性，并将中断发送给刚产生前一相同类型中断现场的同一处理器。

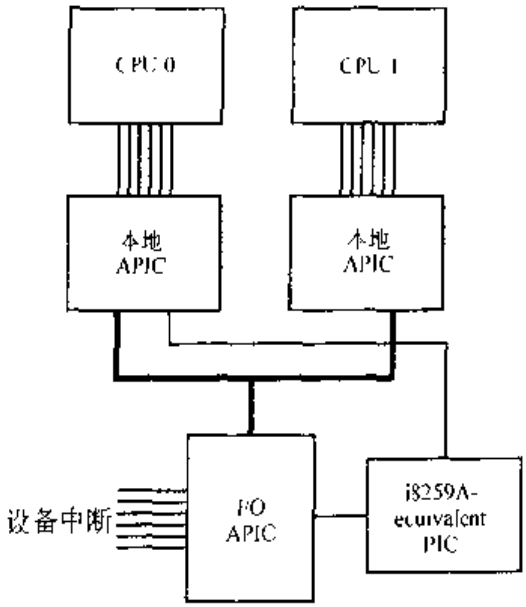


图3-2 x86 APIC结构

每个处理器都有单独的IDT，因而合适的话，不同的处理器可以运行不同的ISR。例如，在一多处理器系统中，每个处理器都接收时钟中断，但只有一个处理器更新系统时钟来响应该中断。然而所有的处理器，当线程时间片（Quantum）结束时，都采用中断来度量线程时间片而准备重新调度，一些系统配置可能要求特定的处理器来处理一定的设备中断。

实验：查看PIC和APIC

分别利用!pic命令和!apic I386kd（或kd）命令可以查看单处理器的PIC和多处理器上的APIC的配置（本实验不能利用LiveKd，因为它不能访问硬件）。以下是单处理器上运行!pic命令的输出（注意如果系统使用APIC HAL，该命令无效）。

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

以下是在运行MPS HAL的系统上使用!apic命令的输出结果。I386kd提示符之前的"0:"前缀表明命令在0处理器上运行，因此这是0处理器的I/O APIC：

```
0: kd> !apic
Apic @ fffe0000  ID:1 (40011) LogDesc:01000000 DestFmt:ffffffff  TPR FF
TimeCnt: 03f66780clk  SpurVec:1f  FaultVec:e3  error:80
Ipi Cmd: 000008el  Vec:E1  FixedDel  Lg:02000000       edg
Timer..: 000300fd  Vec:FD  FixedDel    Dest=Self       edg        masked
Lint0.: 0001001f  Vec:1F  FixedDel    Dest=Self       edg        masked
Lint1.: 000084ff  Vec:FF  NMI         Dest=Self       lv'
TMR: 93, a3
IRR: 41, d1, e3
ISR: d1
```

下面是!ioapic命令的输出结果，它显示了与设备相连的中断控制器组件I/O APIC的配置。

```
0: kd> !ioapic
IoApic @ ffd02000  ID:8 (11)  Arb:0
Inti00.: 000180ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti01.: 00000962  Vec:62  LowestDl  Lg:03000000  edg
Inti02.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti03.: 00000971  Vec:71  LowestDl  Lg:03000000  edg
Inti04.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti05.: 00000961  Vec:61  LowestDl  Lg:03000000  edg
Inti06.: 00010982  Vec:82  LowestDl  Lg:02000000  edg  masked
Inti07.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti08.: 000008d1  Vec:D1  FixedDel  Lg:01000000  edg
Inti09.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0A.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0B.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0C.: 00000972  Vec:72  LowestDl  Lg:03000000  edg
Inti0D.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti0E.: 00000992  Vec:92  LowestDl  Lg:03000000  edg
Inti0F.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti10.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti11.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti12.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti13.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti14.: 0000a9a3  Vec:A3  LowestDl  Lg:03000000  lvl
Inti15.: 0000a993  Vec:93  LowestDl  Lg:03000000  lvl
Inti16.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
Inti17.: 000100ff  Vec:FF  FixedDel  PhysDest:00  edg  masked
```

大多数处理中断的例程驻留在内核中。例如，内核更新时钟时间。而键盘、定点设备和磁盘驱动器等外部设备也能产生很多中断，而且设备驱动程序在发生中断时需要一种途径通知内核调用哪一个例程。

2. 软件中断请求级别（IRQL）

尽管中断控制器采用中断优先级，但Windows 2000有其自身的中断优先方案，称为中断请求级别（IRQL）。内核在内部将IRQL表示成从0到31的数字，数字越大，优先级别越高。内核为软件中断定义了标准的IRQL集，而HAL把硬件中断号映射成IRQL。图3-3给出了x86体系结构中定义的IRQL。

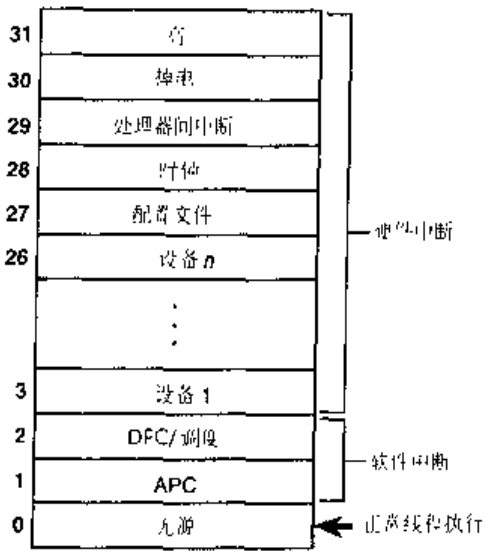


图3-3 中断请求级（IRQL）

中断按照优先级顺序获得服务。优先级高的中断先于优先级低的中断获得服务。当优先级高的中断出现时，处理器保存被中断的线程的状态，并调用与中断相关的陷阱调度程序。陷阱调度程序提升IRQL，并调用中断服务例程。当服务例程执行完后，中断调度程序将处理器的IRQL降低到中断发生之前的级别，然后加载保存的机器状态。中断的线程继续从其停止处执行。内核降低IRQL后，被屏蔽的低优先级中断可能触发。如果这确实发生，则内核将重复处理新中断的过程。

IRQL优先级与线程调度优先权截然不同（后者在第6章讲述）。调度优先权是线程的属性，而IRQL是键盘或鼠标等中断源的属性。此外，每个处理器都有随操作系统代码执行而改变的IRQL设置。

实验：查看IRQL

称为处理器控制区（PCR）的数据结构和其扩展处理器控制块（PRCB）包括系统中有关每个处理器状态的信息，如当前的IRQL、指向硬件IDT的指针、当前运行的线程和下一个选定要运行的线程。内核和HAL则利用这些信息执行体系结构和机器相关的动作。PCR和PRCB的一部分在Windows 2000的设备驱动程序包（DDK）的头文件Ntddk.h中公开定义，因此当需要这些结构的全部定义时，可以查看该文件。

利用!pcr命令可以查看具有内核调试程序的PCR的内容：

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

```
                SelfPcr: ffdff000
                   Prcb: ffdff120
                   Irql: 00000000
                    IRR: 00000000
                    IDR: ffff28e8
      InterruptMode: 00000000
                    IDT: 90036400
                    GDT: 80036000
                    TSS: 802b5000

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

每个处理器的 IRQL 设置决定了处理器所能接收的中断。IRQL 也可用来同步访问内核模式数据结构（在本章后面将了解到更多有关同步的知识）。在内核模式线程运行时，它通过直接调用 KeRaiseIrql 和 KeLowerIrql 或更一般地通过间接调用获取内核同步对象的函数来提高或降低处理器的 IRQL。如图 3-4 所示，来自 IRQL 高于当前中断源的中断将中断处理器，而来自 IRQL 等于或低于当前中断源的中断将被屏蔽，直到当前执行线程降低 IRQL。

由于访问 PIC 的操作相对较慢，使用 PIC 的 HAL 实现了性能优化，称为“延迟”IRQL，它避免了访问 PIC。当 IRQL 提高时，HAL 在内部记下新的 IRQL，而不是改变中断屏蔽。如果低优先级的中断随后出现，HAL 将中断屏蔽设置为符合第一个中断的设置，并延迟低优先级中断直到 IRQL 被降低。因此当 IRQL 升高时，如果没有低优先级中断出现，HAL 就不需要修改 PIC。

内核模式的线程根据它所要达到的目的提高或降低它所在的处理器的 IRQL。例如，当中断出现时，陷阱处理程序（或者处理器）就将处理器的 IRQL 提升到中断源所分配的 IRQL，结果屏蔽了所有等于和低于此 IRQL（仅在该处理器上）的中断，从而确保了该处理器不会因同一级别或更低级别的中断而防碍对此中断的服务。被屏蔽的中断要么由另一处理器处理，要么被延迟直到 IRQL 降低。因此，系统的所有组件（包括内核和设备驱动程序）都试图将 IRQL 保持在无源级别（有时称为低级）。因为只要 IRQL 没有长时间地保持在不必要的高级别，设备驱动程序就能及时地对中断做出反应。

由于改变处理器的 IRQL 对系统操作有很大的影响，因而只能在内核模式下更改——用户模式线程不能改变处理器的 IRQL。这意味着当处理器在执行用户模式代码时，处理器的 IRQL 处于无源级别。因而仅当处理器执行内核模式代码时，IRQL 才会处于较高水平。

每个中断级别都有特定的目的。例如，内核发送处理器间中断（IPI）请求另一处理器执行动作，如调度执行某一特定线程或更新其变换监视缓冲区高速缓存。系统时钟以固定时间间

隔产生中断，内核则通过更新时钟和测定线程执行时间做出响应。如果硬件平台支持两个时钟，内核就需添加另一时钟中断级别来度量性能。HAL提供了一系列的中断级别供中断驱动的设备使用，具体数目随处理器和系统设置的不同而不同。内核使用软件中断（本章后文将讲述）来启动线程调度，并异步中断线程的执行。

映射中断到IRQL 这些IRQL级别与x86系统的中断请求（IRQ）级别不同——x86体系结构没有在硬件中实现IRQL概念。那么Windows 2000如何将IRQL分配给中断呢？答案就在于HAL。在Windows 2000中，称为总线驱动程序的设备驱动程序确定设备在总线上（PCI、USB等等）是否存在，并决定什么样的中断可以分配给该设备。总线驱动程序将这些信息汇报给即插即用管理器，即插即用管理器在考虑所有其他设备的可接受的中断分配后，决定分配给每一设备的中断，然后它再调用HAL函数HalpGetSystemInterruptVector——把中断映射到IRQL。

对于Windows 2000所包括的单处理器和多处理器HAL，它们的分配算法是不同的。在单处理器系统上，HAL只做简单的转换：用27减去中断向量而得到给定中断的IRQL。因此，如果中断所使用的级别为5，那么其ISR在IRQL为22时执行。在多处理器系统上，映射却不这么简单。APIC支持100个以上的中断向量，因此就没有足够的IRQL进行一对一映射。所以多处理器HAL用循环算法（在整个设备IRQL（DIRQL）范围内循环）将IRQL分配给中断向量。结果在多处理器系统中，没有简单的方法来预测或确定Windows 2000分配给APIC IRQ的IRQL。但可以用!idt内核调试程序命令（如本章前文所述）来查看硬件IRQL的赋值。

预定义IRQL 让我们更进一步看看图3-4所示的从最高级别开始的预定义IRQL的使用：

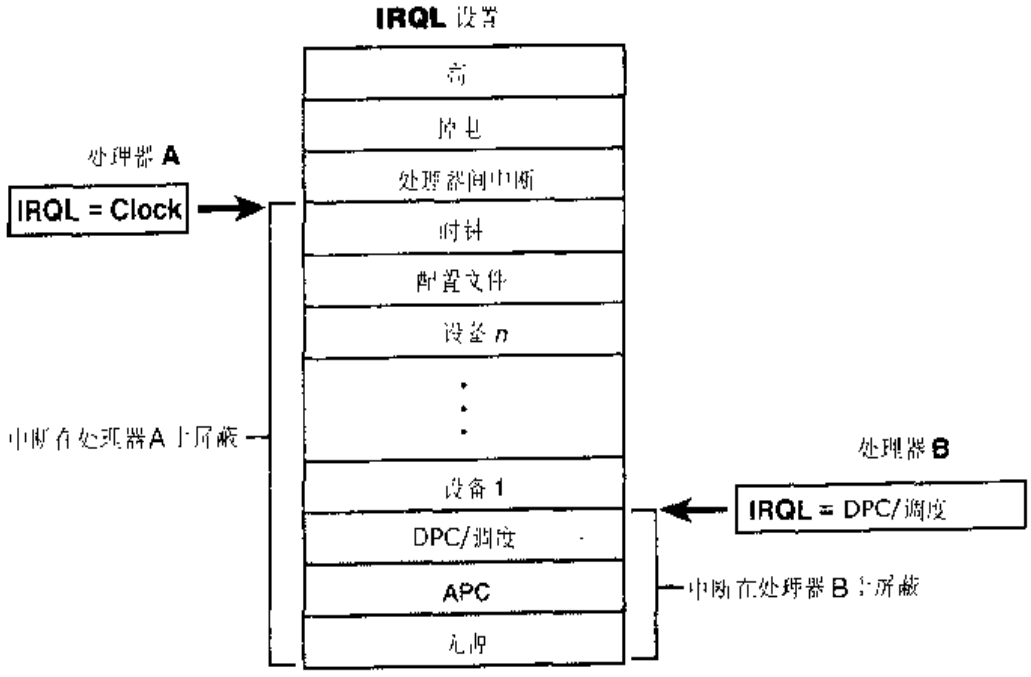


图3-4 从最高级别开始的预定义IRQL的使用

■ 只有当内核在KeBugCheckEx中终止系统并屏蔽所有中断时，内核才使用高级别。

■ 电源失败级别起源于最初的Microsoft Windows NT设计文档，它说明了系统电源失败代码的行为，但该IRQL从未被使用过

■ 处理器间中断级别用来请求另一处理器执行动作，如调度执行某一具体线程，更新处理器变换后备缓冲区（TLB）高速缓存、系统关机或系统崩溃。

■ 时钟级别用于系统时钟，内核利用它来跟踪时间同时测定并分配线程的 CPU 时间。

■ 在内核截面图（Profiling）这种性能测定机制被允许时，系统实时时钟使用 profile 级别。在内核截面图活跃时，若中断发生，内核截面图陷阱处理程序记录当中断发生时正在运行的代码的地址。在一段时间范围内可以构造地址样本表，它可以使用工具提取和分析。Windows 2000 资源包括 Kernel Profiler（Kernrof. exe）工具，可用它来配置和查看截面图产生的数字统计。有关 Kernprof 的更详细资料参见 Kernel Profiler 实验。

■ 设备 IRQL 用于设备中断优先级（参见上节硬件中断优先级到 IRQL 的映射）。

■ DPC/调度级别和 APIC 级别中断是内核和设备驱动器产生的软件中断（DPC 和 APC 将在本章后文做更详尽的解释）。

■ 最低级 IRQL 无源级别，它实际上不是真正的中断级别。它是正常线程执行并且允许所有中断发生的设置。

实验：利用 Kernel Profiler 查看执行

可用Windows2000资源包随带的KernelProfiler工具来允许系统profiling定时器，收集定时器响铃时正在执行代码的样本和显示映射文件和函数频率分布的概要。Kernel Profiler 在关键性能代码重复运行时十分有用，而且在代码运行时可以获得系统耗时的关键所在。为使输出有用，Kernel Profiler 要求系统必须安装 Windows 2000 符号（Symbol）。以下是 Kernel Profiler 在相对比较空闲的系统收集 30 秒的资料后输出的样本（注意大多数的例子都在 *KiIdleLoop*（空闲线程循环）中，这将在第 6 章解释）。

```
C:\kernprof  a  d -x -p -s -t 5 30
Symbols loaded  80400000  ntoskrnl exe
.
delaying for 30 seconds... report on values with 5 hits
end of delay
Processor 0: 30404 Total hits
30404 Total hits

PROCESSOR 0

  26708        ntoskrnl.exe --Total Hits--
     5  0  ntoskrnl.exe memmove 0x80045AEF0 0 3
     5  0  ntoskrnl.exe ExAcquireResourceExclusiveLite 0x080413A00 0 3
     6  0  ntoskrnl.exe ExReleaseResourceLite 0x080413E5t 0 3
    79  0  ntoskrnl.exe KiXMMIZeroPageNoSave 0x080430080 0 3
     8  0  ntoskrnl.exe MiInsertPageInList 0x080442E1E 0 3
     6  0  ntoskrnl.exe MiRemovePageByColor 0x080443756 0 3
     6  0  ntoskrnl.exe ObReferenceObjectByHandle 0x080449B6C 0 3
  26384  0  ntoskrnl.exe KiIdleLoop 0x08045E98C 0 3
     9  0  ntoskrnl.exe KiSystemService 0x08045F4A0 0 3
     6  0  ntoskrnl.exe ExAllocatePoolWithTag 0x080465080 0 3
     5  0  ntoskrnl.exe SepPrivilegeCheck 0x0804F7638 0 3
     5  0  ntoskrn  exe IopParseDevice 0x0804B0F20 0 3

   563           hal.dll   Total Hits--
     5  0  hal.dll KfRaiseIrql 0x080062E90 0 2
    22  0  hal.dll KfLowerIrql 0x080062F00 0 2
```

```
    5  0  hal.dll READ_PORT_UCHAR 0x080067960 0 2
   14  0  hal.dll WRITE_PORT_UCHAR 0x0800679C8 0 2
  503  0  hal.dll HalAcpiC1Idle 0x080068334 0 2

    6           atapi.sys --Total Hits--

   15         Fastfat.sys --Total Hits--

  109           Ntfs.sys   Total Hits
    6  0  Ntfs.SYS NtfsInitializeIrpContext 0x0F99CE432 0 2
   15  0  Ntfs.SYS NtfsCommonWrite 0x0F99CF480 0 2
    7  0  Ntfs.SYS NtfsCommonCleanup 0x0F99D6190 0 2
    7  0  Ntfs.SYS NtfsQueryDirectory 0x0F99DC430 0 2

    8          win32k.sys --Total Hits--

  229           ntdll.dll --Total Hits--
   22  0  ntdll.dll RtlIsValidHandle 0x077F9ADEE 0 2
    5  0  ntdll.dll ZwReleaseSemaphore 0x077F8872C 0 2
   14  0  ntdll.dll RtlpFreeToHeapLookaside 0x077FB0358 0 2
   17  0  ntdll.dll RtlpInterlockedPushEntrySList 0x077FB7620 0 2
    7  0  ntdll.dll RtlTimeToTimeFields 0x077FA7BFE 0 2
    8  0  ntdll.dll RtlEnterCriticalSection 0x077F87B30 0 2
    8  0  ntdll.dll wcslen 0x077FB3D6E 0 2
    8  0  ntdll.dll RtlpAllocateFromHeapLookaside 0x077FB02E4 0 2
   24  0  ntdll.dll RtlpInterlockedPopEntrySList 0x077FB75FC 0 2
   17  0  ntdll.dll RtlpFindAndCommitPages 0x077FC8DF6 0 2
   33  0  ntdll.dll RtlAllocateHeap 0x077FC68F8 0 2
   21  0  ntdll.dll RtlFreeHeap 0x077FC7426 0 2
    5  0  ntdll.dll RtlAllocateHeap 0x077FC68F8 0 2

  756           User Mode --Total Hits-- (NO SYMBOLS)

Context Switch Information
    Find any processor            0
    Find last processor           0
    Idle any processor            0
    Idle current processor        0
    Idle last processor           0
    Preempt any processor         0
    Preempt current processor     0
    Preempt last processor        0
    Switch to idle                0

    Total context switches     3255
```

在DPC/调度级别或以上级别运行代码的一条重要限制就是它不能在对象上等待，如果它在对象上等待就使得调度程序需要选择运行另一线程。另一条限制是只有非页式内存才能在

IRQLDPC/调度级别或更高级别被访问。这一规则事实上是第一条限制的负效应，因为试图访问非常驻内存会引起页面错误。当出现页面错误时，内存管理器初始化磁盘 I/O，然后需等待文件系统驱动程序从磁盘将该页读入。该等待接着请求调度程序执行环境切换（如果没有用户线程在等待运行就可能切换到空闲线程），因此，违反该规则就无法调用调度程序（因为在读磁盘时，IRQL 仍处于 DPC/调度或更高级别）。如果违反以上两条限制中的任何一条，系统将出现崩溃，并产生 IRQL – NOT – LESS – OR – EQUAL 崩溃代码（参见 4.5 节系统崩溃的详细讨论）。违反这些限制是设备驱动程序中常见的错误。Windows 2000 驱动程序检验器，在 7.3.2 节“驱动程序校验器”中解释，有一项可以设置用来帮助查找这类特殊错误的选项。

中断对象 内核提供了一种可移植机制，一种称为中断对象的内核控制对象，它允许设备驱动程序为其设备注册 ISR。中断对象含有一切内核所需要用来联系设备 ISR 与具体中断级别的信息，包括 ISR 地址、设备中断的 IRQL 以及内核的 IDT 中与 ISR 相联系的项。当中断对象被初始化时，称为调度代码的汇编语言代码指令就从中断处理模板 KiInterruptTemplate 中复制并存储到该对象中。若发生中断，就执行这些代码。

这种中断对象常驻代码调用实际的中断调度程序，内核的 KiInterruptDispatch 例程，并将指向中断对象的指针传递给它。该中断对象包含有第二个调度程序例程需要的信息，用于定位和合适地调用设备驱动程序提供的 ISR。

中断对象还存储了与中断相关的 IRQL，这样另一例程 KiDispatchInterrupt 能在调用 ISR 之前将 IRQL 提升到正确级别，并在 ISR 返回之后降低 IRQL。这两步过程是必须的，因为初始调度是由硬件来完成的，没有办法将指针传递给中断对象（或必须的其他任何参数）。在多处理器系统中，内核为每一 CPU 分配并初始化一中断对象，同时允许该 CPU 的本机 APIC 以接收具体中断。

Windows 2000 与实时处理

无论是硬实时系统还是软实时系统，具有不可逾越的界限是实时环境的典型特征。硬实时系统（如核电厂控制系统）不可逾越的界限是系统必须满足避免设备损坏或生命损失。软实时系统（如汽车燃料经济优化系统）的不可逾越的界限是系统可以出错，但需保证具有实时性。在实时系统中，计算机具有传感器输入设备和输出控制设备。实时计算机系统设计人员必须知道在最坏情况下，输入设备产生中断时与设备驱动持程序控制输出设备做出响应时之间的延迟。最坏情况分析必须考虑操作系统导致的延迟以及应用程序和设备驱动程序产生的延迟。

由于Windows2000不以任何可控方式优化设备IRQ，并且用户级应用程序只有当处理器的 IRQL 为无源级别时才执行，因此 Windows 2000 作为实时操作系统并不合适。系统的设备和设备驱动程序——而不是 Windows 2000 操作系统本身——最终决定了最坏情况下的延迟。当实时系统设计人员使用现成的硬件时这种情况就成为了最大的问题。设计人员很难确定每个现成的设备的 ISR 或 DPC 在最坏情况下可能需要的时间。即使经过测试，设计人员还是无法保证实时系统中某些特殊情况不会导致系统超过不可逾越的界限。此外，系统的 DPC 和 ISR 所引入的延迟总和通常远远超过了时间敏感系统的容许值。

尽管很多类型的嵌入式系统(如打印机和汽车用计算机)都有实时要求,但WindowsNT的特殊嵌入式版本却没有实时特征(但目前基于 Windows NT 4 的嵌入式版本最终将会被 Windows 2000 的未来版本所替代)。利用 Microsoft 经 VenturCom 公司授权的系统设计技术,有可能开发出适用于在有限资源设备上运行的微型 Windows NT 版本。例如,没有网络能力的设备可能忽略所有 Windows NT 组件,包括网络管理工具、适配器和与网络相关的协议堆栈设备驱动程序。

另外,还有为WindowsNT和Windows2000提供内核的第三方组件。它们所采用的途径是将其实时内核嵌入用户 HAL,并作为实时操作系统中的一项任务运行 Windows 2000 或 Windows NT。运行 Windows 2000 或 Windows NT 的任务就充当了系统的用户界面,其优先级低于管理设备的任务。要想知道 Windows NT 或 Windows 2000 的第三方实时内核扩展的例子,可以浏览 VenturCom 的网址:www.venturcom.com

将 ISR 与一定级别的中断联系起来的过程称为连接一个中断对象,而将 ISR 与 IDT 入口断开的过程称为断开一个中断对象。这些操作通过调用内核函数 IoConnectInterrupt 和 IoDisconnectInterrupt 实现,允许设备驱动程序在被加载到系统中时“打开”ISR,而在卸载时“关闭”ISR。

使用中断对象注册 ISR,防止设备驱动程序直接干扰中断硬件(在不同处理器体系结构上不同)和了解 IDT 的细节。这种内核特征有助于创建可移植的设备驱动程序,因为它不需用汇编语言编程或反映设备驱动程序的处理器差异。

中断对象还有其他优点。使用中断对象,内核可以在设备驱动程序的其他部分可能与 ISR 共享数据时同步执行 ISR(关于设备驱动程序如何响应中断的进一步资料参见第 9 章)。

此外,中断对象使得内核可以轻易地为任何中断级别调用多个 ISR。如果多个设备驱动程序创建中断对象,并将它们连接到相同的 IDT 项,当中断在具体中断线上发生时,中断调度程序调用每个例程。这种能力使得内核能方便地支持“菊花链”(daisy-chain)配置,从而多个设备能共享相同的中断线。当其中的一个 ISR 占据中断时,它就将状态返回给中断调度程序从而断开菊花链。如果共享相同中断的多个设备要求同时服务,则一旦中断调度程序降低 IRQL,不被各自 ISR 应答的设备就会再一次中断系统。只有希望使用相同中断的所有设备向内核表明它们可以共享中断时,菊花链才是许可的,如果它们不能共享中断,即插即用管理器就重新安排它们的中断分配以确保它满足了每个共享要求。

3. 软件中断

尽管大多数中断是硬件引起的,Windows 2000 内核也为多种任务产生软件中断,如下:

- 初始化线程调度。
- 非时间关键性中断处理。
- 处理定时器过期。
- 在特定线程的环境中异步执行过程。
- 支持异步 I/O 操作。

这些任务将在下面章节中描述。

调度或延迟过程调用（DPC）中断 当线程不能继续执行时，可能由于它已经结束或自动进入等待状态，内核就直接调用调度程序实现立即环境切换。然而，有时当内核处于多层代码的深层次时，检测到应进行重新调度。这时，一个理想的解决办法就是请求调度，但延迟直到内核完成其当前作业。使用 DPC 软件中断就是实现这种延迟的便利方法。

当内核需要同步访问共享的内核结构时，它总是将处理器的 IRQL 提升到 DPC/调度或更高级别。这就禁止了其他的软件中断和线程调度。当内核检测到应当发生调度时，它就请求 DPC/调度级中断；但由于 IRQL 处于或高于该级别，处理器就挂起中断。当内核完成其当前活动时，它认为要将 IRQL 降低为低于 DPC/调度级别，并查看是否有调度中断悬而不决，如果有，IRQL 就降低到 DPC/调度级，调度中断就得到处理。利用软件中断来激活线程调度程序是将调度延迟直到条件满足的一个好办法，Windows 2000 也用软件中断来延迟其他类型的处理。

除了线程调度外，内核也在该 IRQL 级别处理延迟过程调用（DPC）。DPC 实现系统任务的功能，该任务的重要紧迫性要低于当前任务。这些功能之所以被称为延迟的是因为它们可能不被立即执行。

DPC 为操作系统提供了在内核模式下产生中断并执行系统函数的能力。内核利用 DPC 处理定时器过期（并释放在定时器上等待的线程），同时在线程时间片过期后重新调度处理器。设备驱动程序利用 DPC 来完成 I/O 请求。为了给硬件中断提供及时的服务，Windows 2000 与设备驱动程序协调尽量将 IRQL 保持在设备 IRQL 以下。实现这一目标的方法就是让设备驱动程序 ISR 在 DPC/调度 IRQL 以延迟过程调用（DPC）的方式执行完成应答设备、保存易变中断状态、延迟数据传输以及其他非时间关键中断处理活动所必须的最小工作量（DPC 和 I/O 系统的详细资料参见第 9 章）。

DPC 是由 DPC 对象表示的。DPC 对象是内核控制对象，它对于用户模式的程序是不可见的，而对于设备驱动程序和其他系统代码是可见的。DPC 对象所包含的最重要的信息是当内核处理 DPC 中断时它所要调用的系统函数的地址。等待执行的 DPC 例程存储在内核管理的队列中，每个处理器一个队列，称为 DPC 队列。为请求 DPC，系统代码调用内核初始化 DPC 对象，并将其置于 DPC 队列。

缺省情况下，内核将 DPC 对象放在 DPC 请求的处理器（通常是 ISR 被执行的处理器）的 DPC 队列的尾端。然而，通过指定 DPC 优先级（低、中、高，缺省时为中）并为 DPC 指定处理器，设备驱动程序能替换该行为。指向具体处理器的 DPC 就是定向的 DPC（targeted DPC）。如果 DPC 只有低或中优先级，内核就将 DPC 对象加入队列的尾部，如果 DPC 的优先级很高，内核就将 DPC 对象插入队列的前端。

当处理器的 IRQL 要从 DPC/调度或更高级别降到更低级 IRQL（APC 或无源级别）时，内核就处理 DPC。Windows 2000 确保 IRQL 保持在 DPC/调度级，并将 DPC 对象从当前处理器队列中清除直到队列为空（即内核清空队列），依次调用每个 DPC 函数。只有当队列为空时，内核才允许 IRQL 降到 DPC/调度级别以下，并恢复常规线程执行。DPC 处理如图 3-5 所示。

DPC 优先级能以另一种方式影响系统行为。内核通常以 DPC/调度级别中断开始 DPC 队列的清空。只有当 DPC 指向了 ISR 请求的处理器并且 DPC 具有很高或中等优先级时，内核才引发这样的中断。如果 DPC 优先级很低，内核只有当处理器未完成的 DPC 请求数量达到阈值时，

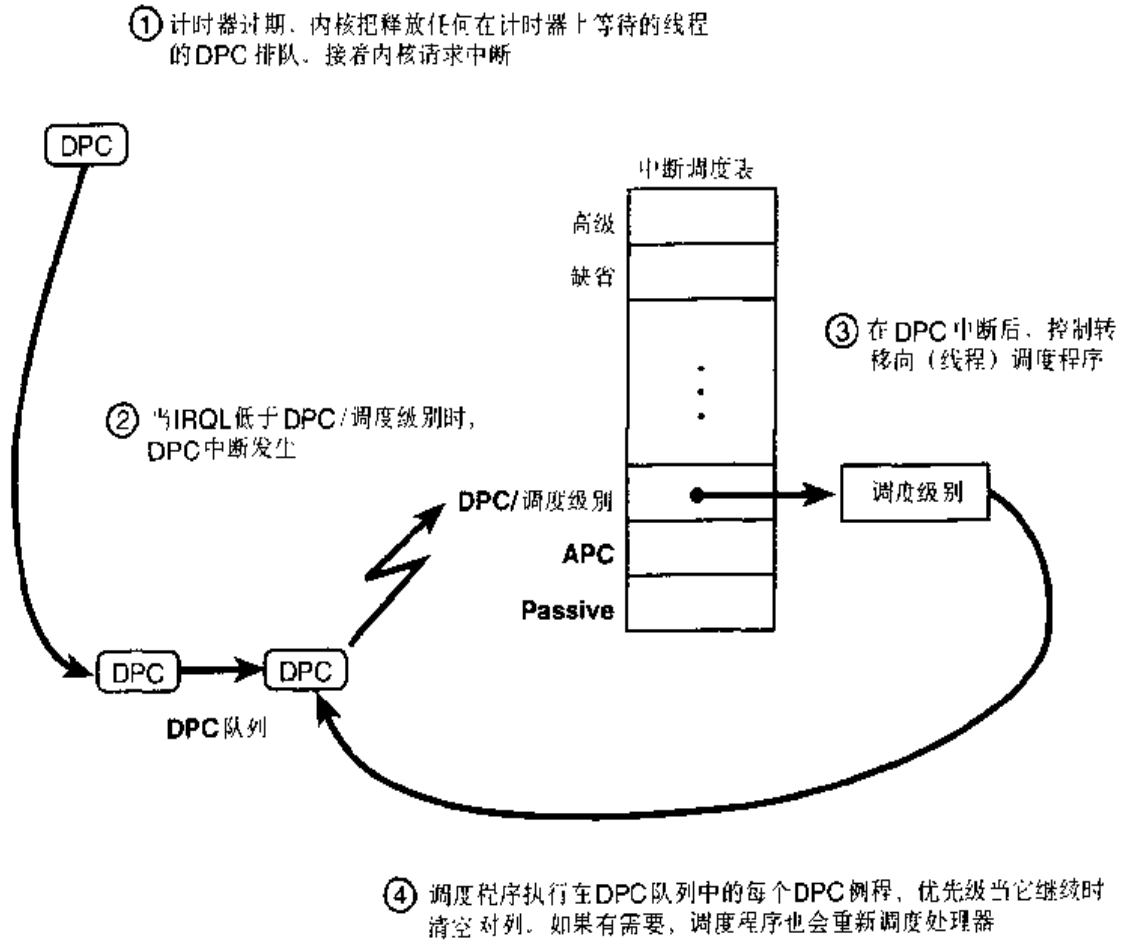


图 3-5 发送 DPC

或在处理器上被请求的 DPC 数量在某时间范围内仍很低时才请求中断。如果 DPC 所指向的 CPU 不是 ISR 所在运行的 CPU，而 DPC 的优先级又较高，内核就立即向目标 CPU 发出信号（通过向它发送调度 IPI）以清空其 DPC 队列。如果优先级中等或很低，那么要使内核引发 DPC/调度级中断，目标处理器队列上的 DPC 数量必须超过阈值。系统空闲进程也在它运行的处理器上清空 DPC 队列。尽管 DPC 的定向和优先级是可变的，设备驱动程序却很少需要改变它们 DPC 对象的缺省行为。表 3-1 总结了引起 DPC 队列清空的情况。

表 3-1 DPC 中断产生规则

DPC 优先级	目标为 ISR 运行的处理器的 DPC	目标为另一处理器的 DPC
低	DPC 队列长度超过最大 DPC 队列长度或 DPC 请求速率小于最小 DPC 请求速率	DPC 队列长度超过最大 DPC 队列长度或系统空闲
中	始终	DPC 队列长度超过最大 DPC 队列长度或系统空闲
高	始终	始终

由于用户模式的线程在低 IRQL 执行，因此 DPC 很有可能中断普通用户线程的执行。DPC 例程执行时不管当前运行什么线程，这意味着当 DPC 例程运行时它无法假定当前映射的是什么进程地址空间。DPC 例程能调用内核函数，但它们却不能调用系统服务、产生页面错误，创

建或在调度程序对象上等待（在本章后文解释）。然而它们能访问非页式系统内存地址，因为不论当前进程是什么，系统地址空间总是进行了映射。

DPC 主要是为设备驱动程序提供的，但内核也使用它们。内核频繁地使用 DPC 来处理时间片过期。在系统时钟的每一滴答中，在时钟 IRQL 发生一次中断。时钟中断处理程序（在时钟 IRQL 运行）就更新系统时间，然后递减一个跟踪当前线程运行状况的计数器。当计数器达到 0 时，线程时间片就过期了，内核可能需要重新调度一个应在 DPC/调度 IRQL 执行的低优先级任务给处理器。时钟中断处理程序将一个 DPC 加进队列以初始化线程调度，然后完成其作业并降低处理器的 IRQL。由于 DPC 中断优先级低于设备中断，所以任何在时钟中断完成之前出现的悬而不决地设备中断在 DPC 中断出现之前被处理。

实验：监控中断和 DPC 活动

利用 Windows2000 Performance 工具，可以查看系统处理中断和 DPC 所耗时间的百分比。处理器对象有 % Interrupt Time 和 % DPC Time 计数器，包括总的和每个处理器的情况，这说明可以监控每个 CPU 或系统范围内的活动。这些对象同时具有用来测量每秒的中断和 DPC 数目的计数器。

当在内核模式下系统花费过多的时间，而又不能将所有内核模式 CPU 时间归于进程时，你可能想查看这些计数器。如果总的内核模式 CPU 时间大于所有进程的总内核模式 CPU 时间，剩余的时间就应是给中断或 DPC 的，因为消耗在中断级和 DPC 级的时间并不受任何线程 CPU 时间性能计数器的支配。注意在 Windows 2000 解决 CPU 时间的方式存在固有的不精确性，这种不精确性与系统定时器的间隔尺寸有关（参见 6.5.8 节“时间片”关于时间统计的解释）。

异步过程调用（APC）中断 异步过程调用（APC）允许用户程序和系统代码在特定用户线程（因而在特定进程地址空间）的环境中执行。由于 APC 在特定的线程环境中排队等待执行，而且在 IRQL 小于 2 时，它们不受 DPC 的限制条件限制。一个 APC 例程可以获取资源（对象）、在对象句柄上等待、引发页面错误以及调用系统服务。

APC 由称为 APC 对象的内核控制对象描述。等待执行的 APC 驻留在内核管理的 APC 队列中。与系统范围内的 DPC 队列不同，APC 队列是线程范围内的——每个线程都有其自身的 APC 队列。当要求把一个 APC 加入队列时，内核将它插入到属于将要执行 APC 例程的线程的队列中，而且当线程最终开始运行时，它就执行 APC。

APC 有两种：内核模式 APC 和用户模式 APC。内核模式 APC 在线程中运行，不需从目标线程请求“许可”，而用户模式 APC 则需要。内核模式 APC 中断线程，并且执行过程时不需线程的干预或同意。

执行程序利用内核模式 APC 完成必须在特定线程的地址空间完成的操作系统任务。例如，它可以利用内核模式 APC 引导线程停止执行可中断的系统服务或记录线程地址空间中的异步 I/O 操作。环境子系统利用内核模式 APC 挂起线程或终止它本身，获取或设置其用户模式执行环境。POSIX 子系统利用内核模式 APC 模拟 POSIX 信号到 POSIX 进程的传送。

设备驱动程序也使用内核模式 APC。例如当一个 I/O 操作被初始化并且线程进入等待状态

时，另一进程中的线程就可被调度运行。当设备完成数据传输时，I/O 系统无论如何必须回到初始化 I/O 的线程环境中，以便它能将 I/O 操作结果复制到包含该线程的进程地址空间的缓冲区中。I/O 利用内核模式 APC 完成这一动作。（APC 在 I/O 系统中的使用将在第 9 章更详尽地讨论）。

很多 Win32 API 如 ReadFileEx、WriteFileEx 和 QueueUserAPC 使用用户模式 APC。如 ReadFileEx 和 WriteFileEx 函数允许调用程序在 I/O 操作完成后指定被调用的完成例程。I/O 的完成是通过将 APC 加进引发该 I/O 的线程队列中实现的。然而当 APC 加进队列后，完成例程的回调未必发生，因为只有当线程处于可改变的等待状态时，APC 才被发送到线程中。线程既可以通过在对象句柄上等待，也可以通过指定其等待是可改变的（利用 Win32 WaitForMultipleObjectsEx 函数）或直接检测它是否具有悬而不决的 APC（利用 SleepEx）而进入等待状态。在以上两种情况下，如果用户模式 APC 悬而不决，内核就中断（改变）线程，将控制转移给 APC 例程，并在 APC 例程完成时恢复线程的执行。

APC 发送能重新排列等待队列——线程在什么上等待以及等待的次序（等待处理在本章“同步执行”一节中讲述）。如果 APC 发送时，线程正处于等待状态，在 APC 例程结束后，就再引发等待或再执行等待。如果等待仍未得到解决，线程就回到等待状态，但此时处于它所等待的对象链表的的末尾。如 APC 用来挂起线程的执行，如果线程在任一对象上等待，那么直到线程恢复后，其等待才被删除，此后，该线程就处于等待访问该对象的线程链表尾端。

3.1.2 异常调度

与可在任何时候出现的中断相比，异常只能由运行程序的执行引起。Win32 引进了结构化异常处理功能，它允许应用程序在异常出现时取得控制。然后，应用程序确定情况并返回到异常发生的地方，展开堆栈（这样就终止了引发异常的子程序的执行）或者向系统说明该异常无法识别，系统就继续搜索可能能够处理该异常的异常处理程序。本节假定你熟悉一些 Win32 结构化异常处理的基本概念——如果不熟悉的话，在继续学习之前可参阅 SDK 平台上的 Win32 API 参考文档中的综述或者是 Jeffery Richter 的书“Programming Application for Microsoft Windows”（第 4 版，Microsoft 出版社，2000）的第 23～25 章。记住尽管异常处理可通过语言扩展（如 Microsoft Visual C++）访问，但它是一种系统机制，因此它不是语言相关的。使用 Windows 2000 异常处理的其他例子包括 C++ 和 Java 异常。

在 x86 上，所有的异常都预定义了直接相应于 IDT 中项的中断号，该项指向处理具体异常的陷阱处理程序。表 3-2 显示了 x86 定义的异常和分配给它们的中断号。由于 IDT 中的头几项是用于异常的，所以正如上面所说的，分配给硬件中断的项在表的后头。

除了那些能被陷阱处理程序解决的简单异常外，所有异常都由称为异常调度程序的内核模块服务。异常调度程序的任务就是查找能够处理异常的异常处理程序。内核所定义的独立于体系结构的异常例子包括内存访问违例、整数被 0 除、整数溢出、浮点异常和调试程序断点。要想知道独立于体系结构异常的完整列表请参阅 Win32 API 参考文档。

内核透明地俘获和处理一些用户程序异常。例如，在调试程序时，遇到断点就会产生异常，内核则通过调用调试程序来处理。内核还通过向调用程序返回失败状态码来处理其他异常。

表 3-2 x86 的异常和其中断号

中断号	异 常
0	除法错误
1	调试陷阱
2	NMI/NPX 错误
3	断点
4	溢出
5	BOUND/Print Screen
6	无效操作码
7	NPX 不可用
8	双精度异常
9	NPX 段超时运行
A	无效任务状态段（TSS）
B	段不存在
C	堆栈错误
D	一般保护
E	页面错误
F	Intel 预留
10	浮点
11	对齐方式检查

少量的异常被允许原封不动地过滤回到用户模式。如内存访问违例或操作系统无法处理的计算溢出异常。环境子系统能创建基于帧的异常处理程序来处理这些异常。“基于帧”一词是指异常处理程序与具体的过程激活相联系。当过程被激活时，一个代表过程被激活的栈帧被压入到栈中。一页栈帧能有一个或更多的与之相联的异常处理程序，每个处理程序都保护源程序的特定代码块。当出现异常时，内核就搜索与当前栈帧相联系的异常处理程序。如果不存在，内核就搜索与前一个栈帧相联系的异常处理程序，如此下去，直到它找到一个基于帧的异常处理程序。如果没找到异常处理程序，内核就调用默认的异常处理程序。

当出现异常时，无论是直接由软件产生的还是硬件隐含产生的，都将在内核中产生一系列事件。CPU 硬件将控制转移给内核陷阱处理程序，它再创建陷阱帧（和出现中断时一样）。如果异常得到解决，陷阱帧就允许系统继续工作。陷阱处理程序还创建一个包括异常出现的原因和其他相应信息的异常记录。

如果异常在内核模式下出现，异常调度程序仅仅调用例程来查找处理异常的基于帧的异常处理程序。因为没处理的内核模式异常被视为操作系统的致命错误，所以可以认为调度程序总能找到异常处理程序。

如果异常在用户模式下出现，异常调度程序要做些更复杂的事。在第 6 章你将看到，*Win32* 子系统中包括调试程序端口和异常端口用来接收 Win32 进程中用户模式异常的通知。内核在其默认异常处理中使用这些，如图 3-6 所示。

调试程序断点是异常的常见源。因此，异常调度程序所采取的第一步策略就是查看引发异

常的进程是否有与之相联系的调试程序进程。如果有，它就向与引发异常的进程联系的调试程序端口发送第一次机会调试消息（通过 LPC 端口）。（该消息被发送到会话管理器进程，后者接着将它分配给适当的调试程序进程。）

如果进程没有与其附接（attach）的调试程序进程或调试程序没有处理该异常，异常调度程序就切换到用户模式，调用例程来查找基于帧的异常处理程序。如果没找到或没能处理该异常，异常调度程序就会切换到内核模式，并重新调用调试程序以便允许用户做更多的调试（这叫做第二次机会通知）。

所有 Win32 线程都有一个在无法处理异常的栈顶声明的异常处理程序。该异常处理程序声明在 Win32 内部的 start - of - process 或 start - of - thread 函数中。前者在进程中的第一个线程开始执行时运行，它调用映像中的主入口点。后者在用户创建另外的线程时运行，它调用在 CreateThread 调用中指定的由用户提供的线程启动例程。

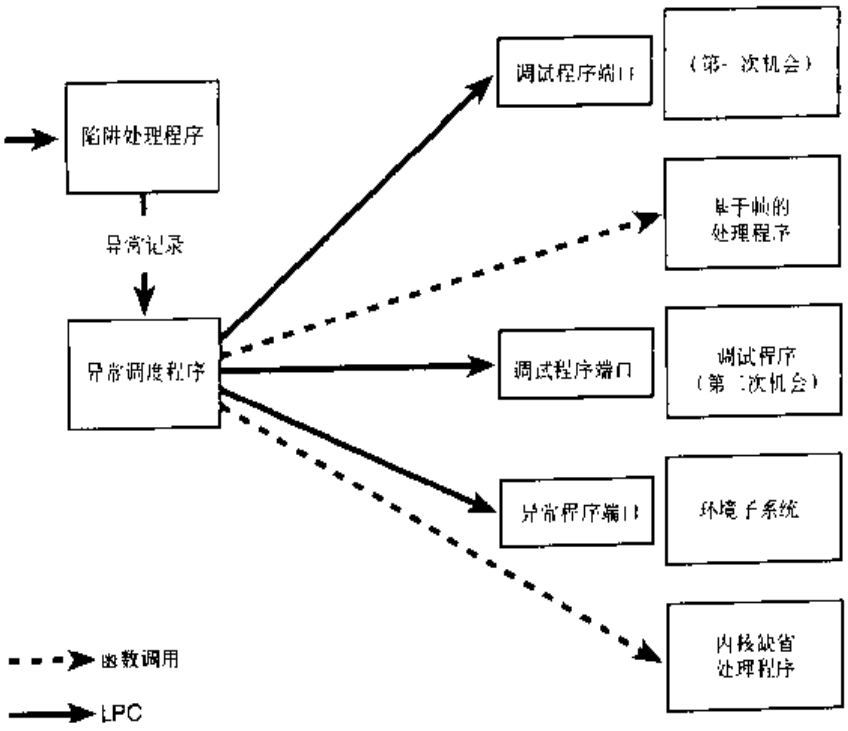


图 3-6 调度异常

实验：Win32 线程的实际用户起始地址

每个Win32线程都是以系统提供的函数开始执行的(而不是用户提供的函数),这一事实解释了为什么线程 0 的起始地址对于系统中的每一个 Win32 线程都是相同的（以及为什么第二个线程的起始地址也是相同的）。Win32 进程中的 0 线程起始地址就是 Win32 start - of - process 函数，任何其他线程的起始地址都应该是 Win32 start - of - thread 函数。为查看用户提供的函数地址，可使用 Windows 2000 维护工具中的 Tlist 实用程序。敲入 tlist process - name 或 tlist procee - id 就可以获得含该信息的详细进程输出。如比较一下 Pstat 和 Tlist（在 SDK 平台上）所报告的 Windows Explorer 进程的线程起始地址：

```
C:\> pstat
:
pid:3f8 pri: 8 Hnd:  329 Pf:  80043 Ws:   4620K explorer.exe
 tid pri Ctx Swtch StrtAddr   User Time kernel Time  State
  7c   9     16442 77E878C1 0:00:01.241 0:00:01.251 Wait:UserRequest
 42c  11    157888 77E92C50 0:00:07.110 0:00:34.389 Wait:UserRequest
 44c   8      6357 77E92C50 0:00:00.070 0:00:00.140 Wait:UserRequest
 1cc   8      3318 77E92C50 0:00:00.030 0:00:00.070 Wait:DelayExecution
 :

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

Pstat所报告的线程0的起始地址是Win32内部的start – of – process函数；线程1到3的起始地址Win32内部的start – of – thread函数。而Tlist，则显示了用户提供的Win32起始地址（用户函数由内部Win32起始函数调用）。

这些内部起始函数的一般代码显示如下：

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

    __try {

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

    } __except(UnhandledExceptionFilter(
            GetExceptionInformation())) {

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

注意，如果线程有它未处理的异常，它就会调用Win32未处理的异常过滤程序。该函数在HKLM \ SOFTWARE \ Microsoft \ Windows NT \ CurrentVersion \ AeDebug Key的注册表中查找用来确定是立即运行调试程序还是先询问用户。

Windows 2000上的默认“调试程序”是 \ Winnt \ System32 \ Drwtsn32. exe (Dr. Watson)，它实际上不是调试程序，而是事后工具（postmortem check），用于捕捉应用程序“崩溃”的状态，并

在日志文件（Drwtsm32. log）以及进程崩溃文件（User. dmp）中记录下来，它们都可以缺省地在 \ Document And Settings \ All Users \ Documents \ DrWatson 文件夹中找到。如果想观察或修改 Dr. Watson 的配置，交互地运行它——它显示一个带有当前设置的窗口，如图 3-7 所示。

日志文件包括一些基本信息。如异常代码、失败的映像名、加载的 DLL 列表以及栈和引发异常的线程的指令跟踪。要想知道日志文件内容的详细描述，运行 Dr. Waston 并单击图 3-7 所示的帮助按钮。

崩溃转储文件包括异常出现时进程中的私用页（该文件不包括 EXE 或 DLL 的代码页）。崩溃转储文件能用 Window 2000 调试程序 WinDbg 打开。Window 2000 调试程序随带在 Windows 2000 调试工具软件包中（是 Windows 2000 Customer Support Diagnostics Platform SDK 和 DDK 一部分）。由于 User. dmp 文件在每次进程崩溃时被重写，因此除非在每次进程崩溃后重命名或复制该文件，否则系统上只有该文件的最新版本。

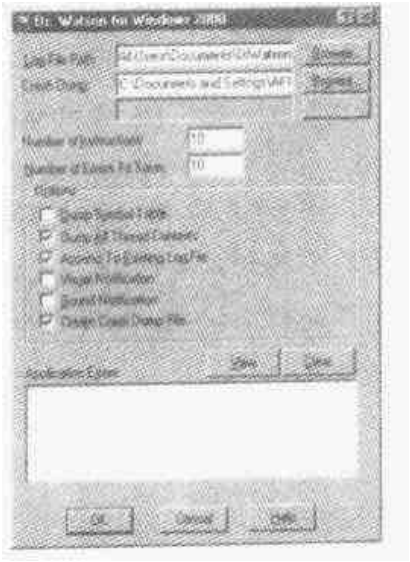


图 3-7 Dr. Watson 缺省设置

如果安装了某个 Microsoft Visual Studio 编译器，AeDebug 注册表键的调试程序值就会变为 Msdev. exe（包括路径），这样才能调试引发了无法处理的异常的程序。另一会改变调试程序键值的产品是 Lotus Notes ——它运行称为 Qnc. exe 的 Notes 相关的事后工具。

如果调试程序不在运行且没查找到基于帧的处理程序，内核就向与线程的进程相关联的异常端口发送消息。如果该异常端口存在，则由控制该线程的环境子系统注册。异常端口使得正在该端口倾听的环境子系统有机会将异常转换成环境相关的信号或异常。例如，当 POSIX 收到来自内核的消息，该内核的一个线程引发了异常，POSIX 子系统就向引发异常的线程发送--个 POSIX 风格的信号。然而，如果内核在处理异常时到了这种地步而子系统无法处理异常，内核就执行默认异常处理程序，它仅仅终止所属线程引发了异常的进程。

实验：未处理的异常

要想了解Dr.Waston日志文件例子，请运行本书配套光盘的 \ Tools \ Accvio.exe程序。该程序试图在 0 地址写入时产生了存储器存取违例，而 0 地址在 Win32 进程中始终是无效地址（参见第 7 章的表 7-6）。

1）运行注册表编辑器，查找HKLMV \ SOFTWARE \ Microsoft \ WindowsNT \ CurrentVersion \ AeDebug。

2）如果 Debugger的值为“drwtsn32 - p %ld - e %ld - g”，则系统已设置好，可将Dr. Waston 作为默认调试程序运行。继续步骤 4。

3）如果Debugger的值还没设置好，不能运行Drwtsn32.exe，则仍需通过暂时安装Dr. Waston 来测试它，然后恢复原来的调试程序设置：

□ 保存当前值（如在 NotePad 文件或当前粘贴缓冲区中）。

□从任务栏中Start菜单中选择Run，然后键入drwtsn32 - i。(这是初始化Debugger字段来运行 Dr. Waston)。

4) 运行测试程序 \ Tools \ Accvio. exe。

5)程序错误信息框应当弹出——如果日志文件和崩溃转储文件已经创建好，按钮就从Cancel 变为 Ok。(注意：只有在 Windows 2000 Professional 的 Dr. Waston 安装中才会缺省地出现错误信息框——缺省 Windows 2000 服务器不会显示信息框，但转储文件仍然会创建。如果 Dr. Waston 不是默认调试程序，读者也可能看到不同的错误信息。)

6) 单击 Ok 退出信息框。

7) 运行 Drwtsn32. exe (在 Start 菜单上选择 Run，然后键入 drwts32)。

8) 在应用程序错误列表中，单击最后一项并单击View按钮— 包括Accvio.exe产生的存取违例细节的 Dr. Waston 日志文件就会显示出来。(要想知道日志文件格式的具体细节，按 Dr. Waston for Windows 2000 对话框的 Help 按钮，并选择 Dr. Waston Log File Overview)。

9) 如果 Debugger 的初始值不是缺省的 Dr. Waston 设置，恢复步骤 1 中的保存值。

作为另一个实验，试着将Debugger的值改为另一程序，如NotePad.exe(NotePad编辑器)或 Sol. exe (Solitaire)。再运行 Accio. exe，注意看在 Debugger 值指定的运行程序到底是什么——即实际上 Debugger 中定义的程序是调试程序，注意确保恢复了注册表设置。(如步骤 3b 所示，要恢复系统缺省 Dr. Waston 设置，在 Run 对话框中或在命令提示符下键入 drwtsn32 - i。)

3.1.3 系统服务调度

如图 3-1 所示，内核的陷阱处理程序调度中断、异常和系统服务调用。在前面的章节中，读者已经了解了中断和异常处理的工作方式。在本节中，你将学习有关系统服务的知识。系统服务调度在 x86 处理器上是通过执行 int 0x2e 指令（十进制的 46）触发的。由于执行 int 指令会导致陷阱，所以 Windows 2000 在 IDT 中填入项 46 指向系统服务调度程序（参见表 3-1)。陷阱导致执行线程切换到内核模式并进入系统服务调度程序。在 EAX 处理机寄存器上传递的数值参数显示的正是被请求的系统服务号。EBX 寄存器指向调用程序传递给系统服务的参数列表。下面是系统服务请求的通用代码：

```
NtWriteFile:
  mov  eax, 0x0E ; build 2195 system service number for NtWriteFile
  mov  ebx, esp  ; point to parameters
  int  0x2E      ; execute system service trap
  ret  0x2C      ; pop parameters off stack and return tc caller
```

如图 3-8 所示，内核利用该参数来定位系统服务调度表中的系统服务信息。该表类似于本章早些时候讲述的中断调度表，只是这里每个项所包含的指针指向系统服务，而不是中断处理例程。

注意 系统服务号能在服务包中改变—— Microsoft 有时添加或删除系统服务，而且系

统服务号作为内核编译的一部分自动产生。

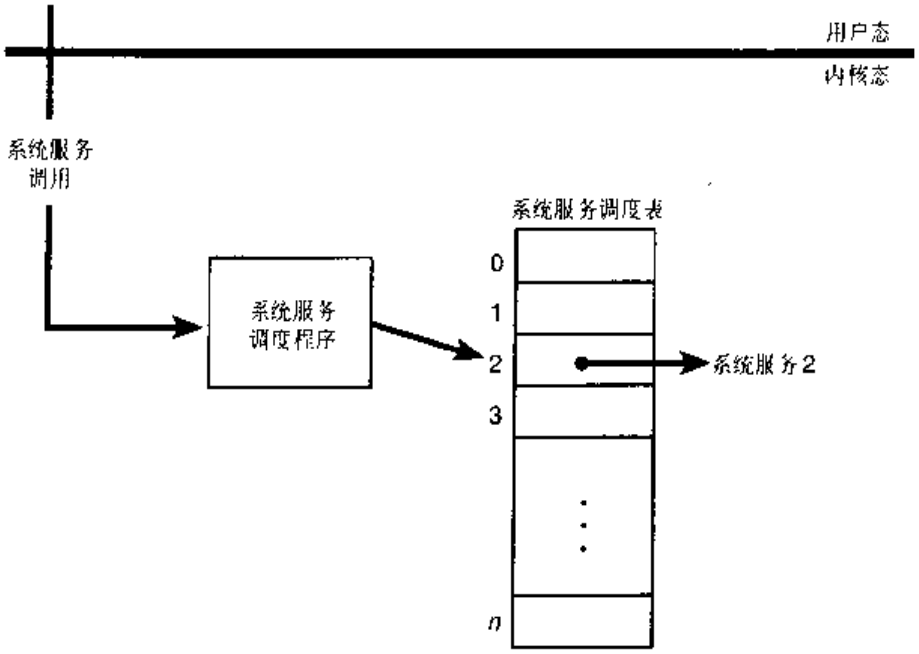


图 3-8 系统服务异常

系统服务调度程序 KiSystemService 校验参数正确的最小数目，并且将调用程序的参数从线程的用户模式栈复制到内核模式栈（以便在内核访问它们时用户无法改变这些变量），然后执行系统服务。如果传递给系统服务的参数指向用户空间的缓冲区，在内核模式代码复制数据到缓冲区或从缓冲区复制数据之前，就必须检测这些缓冲区的可访问性。

正如在第 6 章中所了解的，每个线程都有一个指向系统服务表的指针。Windows 2000 有两个内建的系统服务表，但最多支持 4 个。系统服务调度程序通过将 32 位系统服务号中的 2 位字段解释为表的索引来确定包含被请求服务的表。系统服务号的低 12 位作为表索引所指定的表的索引。字段如图 3-9 所示。

主要的缺省数组表 KeServiceDescriptorTable，定义了在 Ntosrknl. exe 中实现的执行程序系统服务。另一个表数组 KeServiceDescriptorTableShadow，包括了 Win32k. sys，作为 Win32 子系统内核模式部分实现的 Win32 USER 和 GDI 服务。当 Win32 线程第一次调用 Win32 USER 或 GDI 服务，线程系统服务表的地址被改为指向包括了 Win 32 USER 和 GDI 服务的表。函数 KeAddSystemServiceTable 允许 Win32. sys 和其他设备驱动程序添加系统服务表。如果在 Windows 2000 中安装了 Internet 信息服务（IIS），其支持的驱动程序（Spud. sys）一旦加载后就定义另外的服务表，仅留下一个表由第三方定义。除了 Win32k. sys 服务表外，另外用 KeAddSystemServiceTable 添加的服务表会被同时复制到 KeServiceDescriptorTable 数组和 KeServiceDescriptorTableShadow 数组中。

Windows 2000 执行程序服务的系统服务调度指令存在于系统库 Ntdll. dll 中。子系统 DLL 调用 *Ntdll* 中的函数实现它们存档的函数。Win32 USER 和 GDI 函数是例外，其中的系统服务调度指令直接在 User. dll 和 Gdi32. dll 中实现——没有 Ntdll. dll。这两种情况如图 3-10 所示。

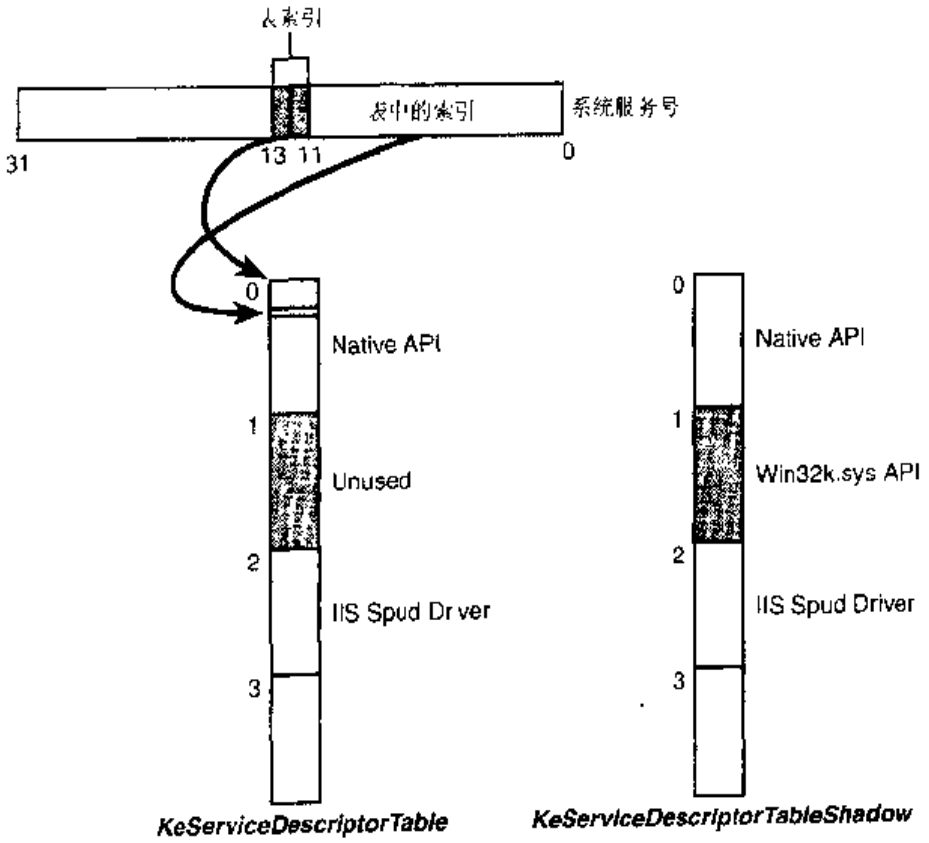


图 3-9 系统服务转换的系统服务号

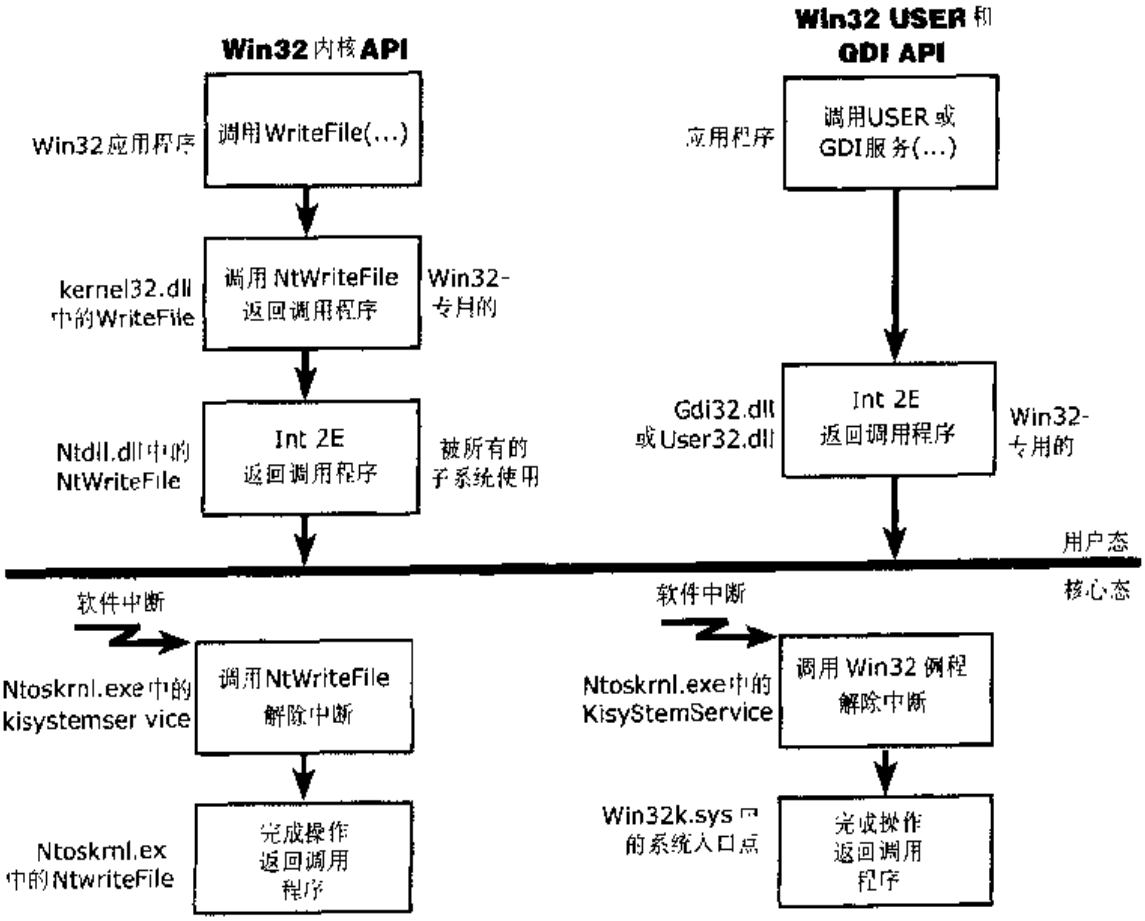


图 3-10 系统服务调度

如图 3-10 所示，Kernl32. dll 中 Win32 WriteFile 函数调用 Ntdll. dll 中的 NtWriteFile 函数，后者接着执行相应的指令引起系统服务陷阱，同时传递代表 NtWriteFile 的系统服务号。系统服务调度程序（Ntoskrnl. exe 中的函数 KiSystemService）随后调用实时的 NtWriteFile 来处理 I/O 请求。对于 Win32 USER 和 GDI 函数，系统服务调度调用 Win32 子系统可加载内核模式部分中的函数 Win32k. sys。

实验：查看系统服务活动

通过查看系统对象中SystemCalls/Sec性能计数器，可以监视系统服务活动。运行Performance 工具，并在图表中单击 Add 按钮，向该图表添加一个计数器，选定 System 对象，选定系统 System Calls/Sec 计数器，然后单击 Add 按钮将该计数器添加给该图表。

3.2 对象管理器

正如第 2 章所提到的，Windows 2000 为了能够一致而安全地访问在执行程序中实现的各种内部服务而实现了对象模型。本节将描述 Windows 2000 的对象管理器，它是负责创建、删除和跟踪对象的执行程序组件。对象管理器集中了资源控制操作，使其不会在整个操作系统中分散。

实验：研究对象管理器

贯穿本节始终，你将发现显示如何查看对象管理器数据库的实验。这些实验要使用以下工具，若不熟悉请首先熟悉这些工具：

- Object Viewer 该工具有两种版本：一种来自 www. sysinternals. com 网站（本书配套光盘中的 \ Sysint \ Winobj. exe），另一种不同版本来自 Platform SDK 中（在 \ Program Files \ Microsoft Platform SDK \ Bin \ Winnt \ Winobj. exe 中）。与后者相比，前者提供了有关对象的更确切信息（如引用记数器、打开的句柄数、安全描述符等等）。
- Open Handle www. sysinternals. com 中的两种工具显示打开的句柄：GUI 工具（本书配套光盘中 \ Sysint \ Handleex. exe）和命令行工具（配套光盘的 \ Sysint \ Nthandle. exe）。Windows 2000 资源包含有称为 Oh. exe 的另一个工具可显示打开的句柄。
- 内核调试程序！handle 命令。

对象浏览器可以遍历对象管理器维护的名字空间（后面将解释并不是所有的对象都有名字）。试着运行配套光盘中的 WinObj 对象管理器实用程序，并检查布局，如图 3-11 所示。

在Windows2000资源工具包关于OH的帮助文件中，你将发现如果对象跟踪（执行程序的一个内部调试特性）没被激活，通过设置 Windows 2000 注册表中的全局标记然后重新启动系统，OH 将被允许。不论是 Nthandle 还是配套光盘中 Handleex 都不需要对象跟踪（可以通过手工设置标记并重新启动系统来允许对象跟踪）。由于该标记占用额外内存（在跟踪对象使用信息时），那么在完成 OH 实验后，应使用 Gflag 实用工具取消它并重新启动系统。

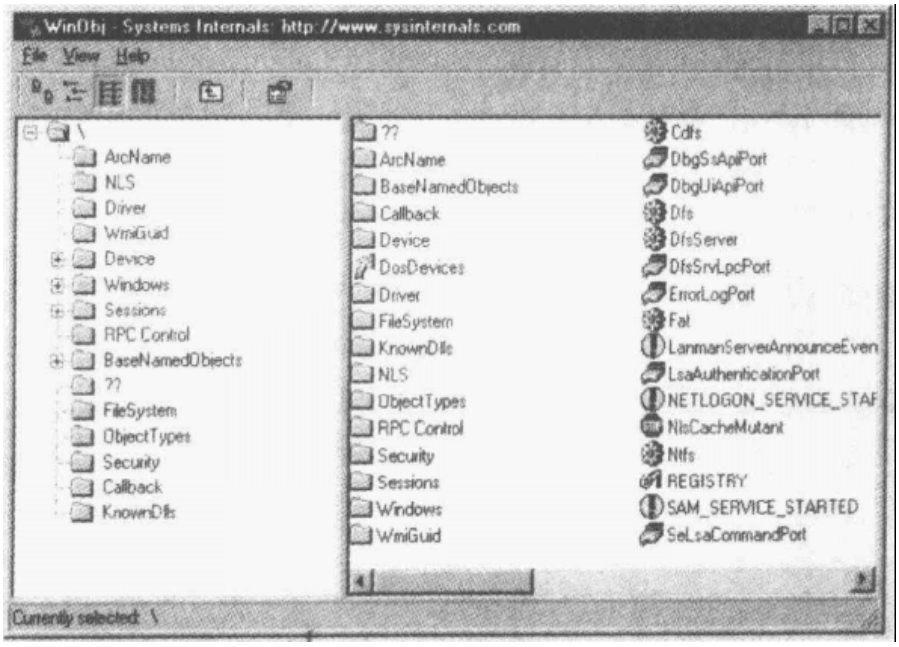


图 3-11 运行配套光盘中的 WinObj 对象管理器实用程序

对象管理器的设计应满足以下要求：

- 提供公共的统一的系统资源使用机制。
- 在操作系统的某个位置隔离对象保护以便能实现 C2 安全一致性。
- 提供机制来控制进程使用对象以便系统管理员可以设置使用系统资源的限制。
- 建立对象——名字方案以便能有准备地合成现有对象，如设备、文件以及文件系统目录或对象的其他独立集合。
- 支持不同操作系统环境的要求，如进程继承父进程资源的能力（Win32 和 POSIX 需要）以及创建大小写敏感文件名的能力（POSIX 需要）。
- 建立对象保留的统一规则（也就是保持对象可用，直到所有进程不再使用它）。

在内部，Windows 2000 有两种对象：执行程序对象和内核对象。执行程序对象就是由执行程序的不同组件实现的对象（如进程管理器、内存管理器和 I/O 子系统等等）。内核对象则是由 Windows 2000 内核实现的原语对象集。这些对象在用户模式代码下是不可见的，仅在执行程序内被创建和使用。内核对象提供了基本能力，如同步，执行程序对象的创建以同步为基础。因此，很多执行对象含有一个或更多内核对象，如图 3-12 所示。

对于有关内核对象的结构以及它们如何用来实现同步的细节将在本章后文讲解。本节接下来将集中讲述对象管理器的工作原理以及执行程序对象的结构、句柄和句柄表。这里将简要描述对象是如何涉及到实现 Windows 2000 安全访问检查的。第 8 章将全面讨论该问题。

3.2.1 执行程序对象

每个 Windows 2000 环境子系统都将应用程序映射为不同的操作系统映像。执行程序对象和对象服务就是环境子系统用来构建它们自身的对象版本和其他资源的基本原语。

执行程序对象通常或者由代表用户应用程序的环境子系统创建，或者由操作系统的不同组件作为它们正常操作的一部分创建。例如，要创建文件，Win32 应用程序将调用在 Win32 子系统 DLL

Kernel32.dll 中实现的 CreateFile 函数。经过一定检验和初始化后，CreateFile 接着调用本机 Windows 2000 服务 NtCreateFile 来创建执行程序文件对象。

环境子系统提供给其应用程序的对象集可能大于或小于执行程序提供的对象集。Win32 子系统利用执行程序对象来输出其自身的对象集，其中很多直接对应于执行程序对象。例如，Win32 互斥和信号量就是直接建立在执行程序对象基础上的（它又以相应的内核对象为基础）。此外，Win32 子系统还提供了名字管道（named pipes）和邮箱（mailslot）以及基于执行程序文件对象的资源。一些子系统，如 POSIX，根本不支持对象。POSIX 子系统利用执行程序对象和服务作为为其应用提供 POSIX 风格进程、管道以及其他资源的基础。

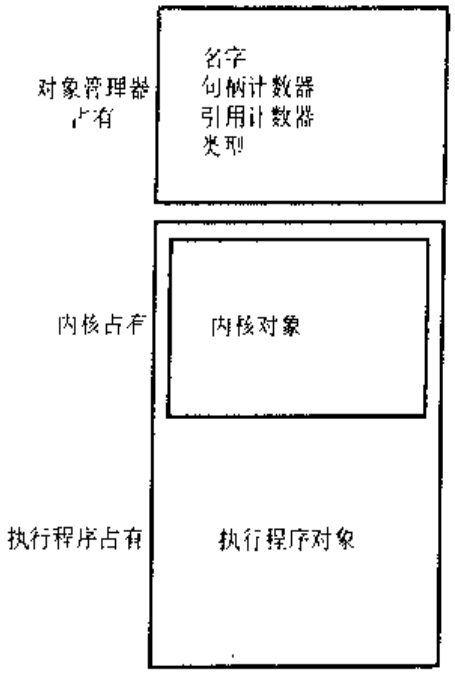


图 3-12 包含内核对象的执行程序对象

表 3-3 列出了执行程序提供的主要对象，并简要地描述了它们所代表的意义。读者可以在描述有关执行程序组件（或者在 Win32 API 参考文档中，执行程序对象被直接输出到 Win32 的情况）的章节中看到执行程序对象的进一步细节。

注意 执行程序在 Windows 2000 中一共实现了 27 个对象类型，其中很多仅由定义它们的执行程序组件使用，而不能由 Win32 API 直接访问。这些对象例子包括驱动程序、设备和事件对。

表 3-3 提供给 Win32 的执行程序对象

对象类型	描 述
符号链接	间接引用对象名字的机制
进程	执行线程对象集所必须的虚拟地址空间和控制信息
线程	进程内的可执行程序实体
作业	通过作业可作为单一实体管理的进程集合
区域	共享内存的区域（在 Win32 中称为文件映射对象）
文件	打开的文件或 I/O 设备的实例
访问令牌	进程或线程的安全配置文件（安全 ID、用户权限等等）
事件	可用于同步或通知的具有持久状态的对象（信号和非信号的）
信号量	一个记数器，它通过仅允许不超过某一最大数量的线程数目去访问由信号量保护的资源来提供资源门
互斥	用于序列化访问资源的同步机制
定时器	固定时间段过后通知线程的机制
IoCompletion	线程对 I/O 操作结束后通知加入队列和退出队列的方法（在 Win32 API 中称为 I/O 完成端口）
键	引用注册表中数据的机制。尽管键出现在对象管理器名字空间中，但它们由配置管理器管理，类似于文件对象由文件系统驱动程序管理。没有或更多的键值与键对象相关，键值包括与键相关的数据

（续）

对象类型	描　述
窗口站	包括粘贴板、全局原子操作集以及桌面对象组的对象
桌面	包括在窗口站内的对象、桌面有逻辑显示面，并包括窗口、菜单和钩

注：① 在 Win32 API 外部，mutant 称为互斥，在内部，构成互斥的内核对象称为 mutant

3.2.2 对象结构

如图 3-13 所示，每个对象都有一个对象头和一个对象体。对象管理器控制对象头，而所属执行程序组件控制它们所创建对象类型的对象体。除此以外，每个对象头都指向让对象处于打开状态的进程列表和一个称为类型对象的特殊对象，它包含对象每个实例共有的信息。

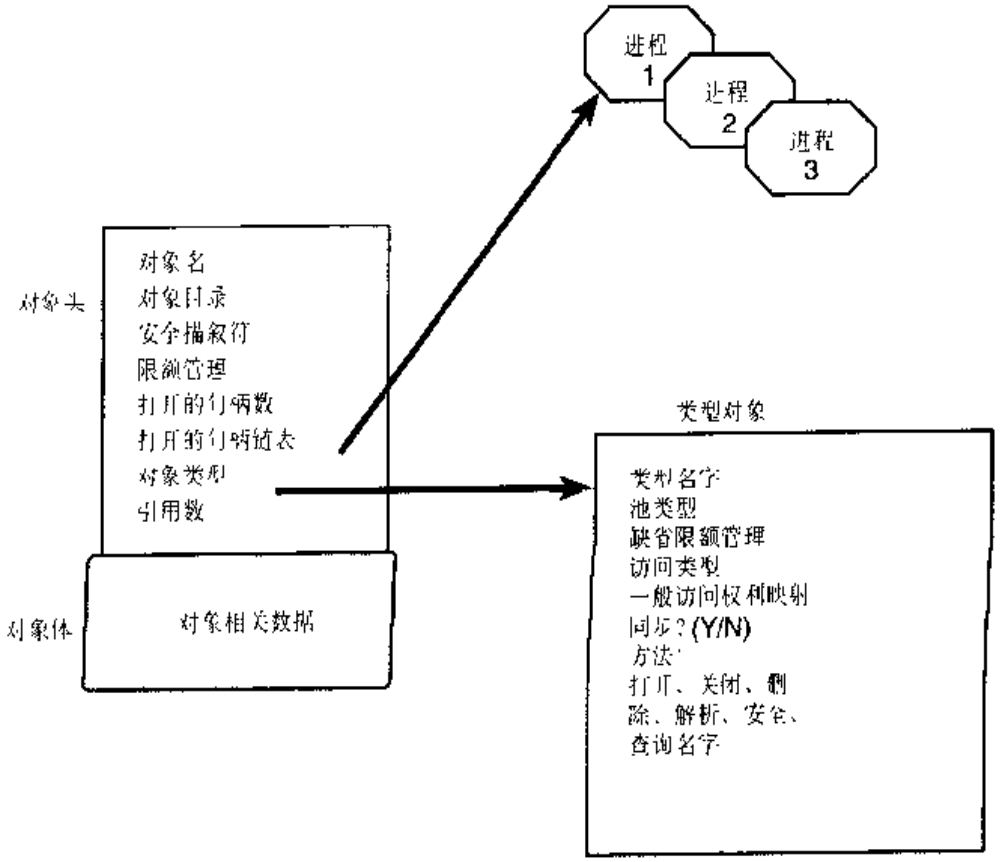


图 3-13　对象结构

1. 对象头和对象体

对象管理器利用存储在对象头的数据来管理对象，而不需考虑它们的类型。表 3-4 简单地描述了对象头属性。

除了对象头外，每个对象都有一个对象体，其格式和内容对于其对象类型来说都是惟一的；相同类型的所有对象具有相同的对象体格式。通过创建对象类型并为其提供服务，执行程序组件能控制该类型的所有对象体中的数据操作。

对象管理器提供一些通用服务，对存储于对象头的属性进行操作，并能用于任何类型的对

象（尽管一些通用服务对于某些对象没有什么意义）。这些通用服务列在表3-5中，Win32子系统使得其中一些能为Win32应用程序所用。

表3-4 标准对象头属性

属性	目的
对象名字	使对象对其他共享进程可见
对象目录	提供存储对象名字的层次结构
安全描述符	决定谁能使用对象和对其进行什么样的操作
配额量	进程打开对象句柄时，列出针对进程的资源配额
打开的句柄计数	计算一个对象的句柄被打开的次数
打开的句柄列表	指向打开对象句柄的进程列表
对象类型	指向包括该类型对象的公共属性的类型对象
引用计数	计算内核模式组件引用对象地址的次数

虽然所有对象类型都支持这些通用对象服务，但每个对象都有其自身的创建、打开和查询服务。例如，I/O系统为其文件对象实现了创建文件服务，而进程管理器为其进程对象实现创建进程服务。尽管单独的创建对象服务也能被实现，但这样的例程会十分复杂。因为初始化文件对象所需的参数集与初始化进程对象所需的就截然不同。况且，在线程每次调用对象服务以确定句柄所指的对象类型和调用相应的服务方案时，对象管理器会引起额外的处理系统开销。由于以上和其他原因，每个对象类型都单独实现了创建、打开和查询服务。

表3-5 通用对象服务

服务	目的
关闭	关闭对象句柄
复制	通过复制句柄并把它交给另一个进程来共享对象
查询对象	取得有关对象标准属性的信息
查询安全	取得对象安全描述符
设置安全	改变对对象的保护
等待单个对象	用一个对象同步一个线程执行
等待多个对象	用多个对象同步一个线程执行

2. 类型对象

对象头包含了所有对象的公共数据，但对象的每个实例都可对该数据取不同的值。如，每个对象只有惟一的名字，并且只能拥有惟一的安全描述符。但是，对象还包括对于特定类型的所有对象保持不变的数据。如打开该类型的对象句柄时，可以从对应于某对象类型的访问权限集合中进行选择。执行程序为线程对象（及其他之间）提供了终止和挂起访问，为文件对象（及其他之间）提供了读、写、附加和删除访问。与对象类型属性相关的另一个例子就是同步，它将在后面介绍。

为了节省内存，一旦创建了新对象类型，对象管理器就将这些静态的与对象类型相关的属性存储起来。它利用其自身的一个对象，即类型对象来记录该数据。正如图3-14所示，如果设置了对象跟踪调试标记，类型对象就同时将相同类型的所有对象连接起来（在这种情况下是进程类型），并在必要时允许对象管理器对它们进行查找和枚举。

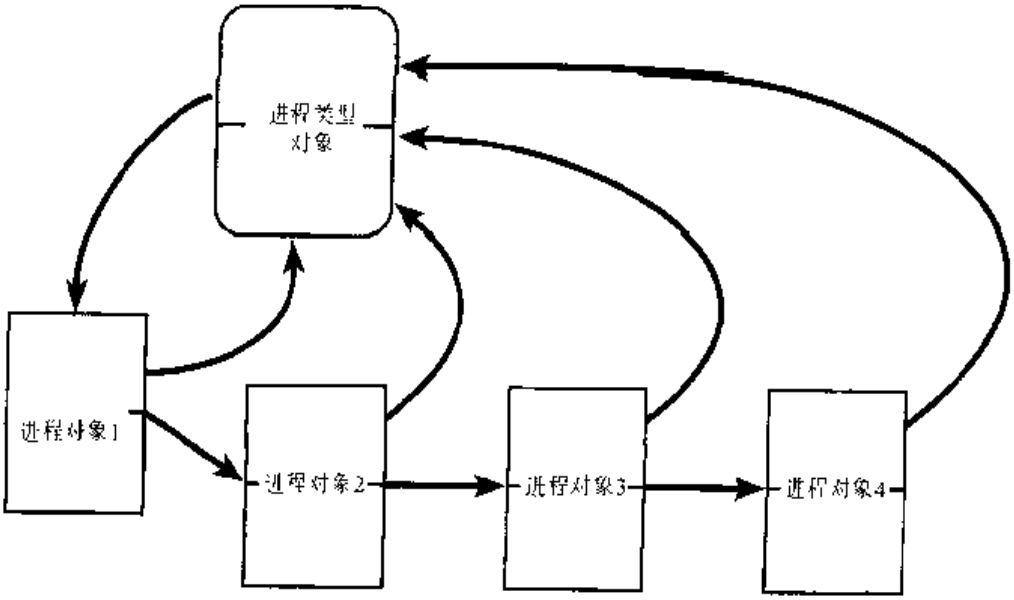


图 3-14 进程对象和进程类型对象

实验：查看类型对象

利用本书配套光盘中的ObjectViewer实用程序能够查看向对象管理器声明的类型对象列表。运行 \ Sysint \ Winobj. exe，然后在 Winobj 对象管理器中打开 \ ObjectType 目录，如图 3-15 所示。

图 3-15 查看类型对象列表

类型对象不能在用户模式下操作，因为对象管理器未给它们提供服务。然而，通过一定的本机服务和 Win32 API 例程，它们所定义的一些属性却是可见的。存储在类型对象中的属性如表 3-6 所述。

表 3-6 类型对象属性

属 性	目 的
类型名字	类型对象的名字（“进程”、“事件”、“端口”等等）
交换区类型	对象类型是否应该从页式或非页式内存分配
缺省配额量	对进程配额征收的缺省页面和非页交换区值
访问类型	当打开该类型对象句柄（读、写、终止和挂起等等）时，线程所能请求的访问类型
一般访问权限映射	四种一般访问权限（读、写、执行和所有）与类型相关访问权限之间的映射
同步	线程是否能在该类型对象上等待
方法	在对象生存期内某点对象管理器自动调用的一个或更多的例程

同步是 Win32 应用程序可见的一种属性，是指线程通过等待对象从一种状态改变到另一种状态来同步执行其程序的能力。线程可以与执行程序作业、进程、文件、事件、信号、互斥以及定时器对象同步。其他执行程序对象不支持同步。对象支持同步的能力是基于对象是否包括嵌入的调度程序对象的基础上的，该对象是一内核对象，将在本章后文的“执行程序同步”一节中讲述。

3. 对象方法

表 3-6 中的最后一项属性，也就是方法，包括了类似于 C++ 的构造函数和析构函数的内部例程集——即在对象创建或破坏时被自动调用的例程。对象管理器通过在其他情况下调用对象方法推广了这一思想，如当某人打开或关闭对象句柄或试图改变对象保护时，一些对象类型指定了方法，而其他的则没有，这依赖于对象类型的使用。

当执行程序组件创建新的对象类型时，它能在对象管理器中注册一种或更多的方法。此后，对象管理器就在该类型对象的生命期内已经定义好的地方调用这些方法，在一定程度上通常是对象被创建、删除或修改时。对象管理器支持的方法列在表 3-7 中。

表 3-7 对象方法

方 法	何时被调用
打开	当对象句柄被打开时
关闭	当对象句柄被关闭时
删除	对象管理器删除对象之前
查询名字	当线程请求存在于次级对象域的对象名字时，如文件
解析	当对象管理器搜索存在于次级对象域的对象名字时
安全	当进程读入或更改存在于次级对象域的对象（如文件）的保护时

对象管理器在创建对象句柄的任何时候调用打开方法，而当对象被创建或打开时它就要创建对象句柄。然而，只有一种对象类型即桌面对象类型定义了打开方法。桌面对象类型要求打开方法，使得 Win32k 能与作为桌面相关的内存交换区的进程共享一块内存。

关闭方法使用的例子出现在 I/O 系统中。I/O 管理器为文件对象类型注册了关闭方法，而且对象管理器在它每次关闭文件对象句柄时调用关闭方法，该关闭方法检查正在关闭文件句柄的进程在文件上是否有没打开的锁，如果有，就将它们消除。对文件锁的检查并不是对象管理器本身能做或应该做的。

如果注册了删除方法，在对象管理器从内存中删除临时对象时，它就调用删除方法。如内存管理器为区域对象类型注册了删除方法。区域对象类型释放被区域所使用的物理页。它也在区域对象被删除之前，校验内存管理器为区域分配的数据结构是否被删除。对象管理器在此也无法完成该作业，因为它对内存管理器的内部工作一无所知。其他对象类型的删除方法也有类似功能。

解析方法（查询名字方法也类似）允许对象管理器在发现对象管理器名字空间外存在的对象时，将查找对象的控制下放给二级对象管理器。当对象管理器查找对象名字时，如果它遇到路径中具有相关的解析方法的对象，就将其搜索挂起。对象管理器调用解析方法，同时将它要查找的对象名字的其余部分传递给它。除了对象管理器的名字空间外，Windows 2000 中还有两种名字空间：配置管理器实现的注册表名字空间和 I/O 管理器在文件系统驱动程序协助下实现的文件系统名字空间（配置管理器的进一步信息，参见第 5 章；I/O 管理器和文件系统驱动程序的进一步信息参见第 9 章）。

例如，当进程打开一名为 \Device\Floppyv0\docs\resume.doc 的对象句柄时，对象管理器就遍历其名字树，直到它到达称为 Floppy0 的设备对象。当它发现了与该对象相关的解析方法，就调用该方法，并将正在寻找的对象名字的其余部分传递给该方法——在这种情况下，就是字符串 \docs\resume.doc。设备对象的解析方法是一个 I/O 例程，因为 I/O 管理器定义了设备对象类型，并为它注册了解析方法。I/O 管理器的解析例程获取名字字符串，并将它传递给相应的文件系统，文件系统再在磁盘上查找文件，并打开它。

类似于解析方法，I/O 系统也使用安全方法。无论线程在什么时候试图查询或更改保护文件的安全信息，它都会被调用。该信息对于文件和其他对象有差异，因为安全信息是存储在文件本身当中，而不是在内存中。因此，为查找安全信息、阅读或更改它，就必须调用 I/O 系统。

4. 对象句柄和进程句柄表

当进程以名字创建或打开对象时，它就接收代表访问对象的句柄。用句柄访问对象比用名字访问要快，因为对象管理器不需要查找名字，而是直接查找对象。进程也能通过在进程创建时继承句柄而获得对象句柄（如果创建程序在 CreateProcess 调用中指定了继承句柄标记，或者在创建时或之后利用 Win32 SetHandleInformation 函数将句柄标记为可继承的），或从其他进程接收复制的句柄（参见 Win32 DuplicateHandle 函数）。

所有用户模式进程在线程使用对象之前，必须拥有对象句柄。利用句柄来操作系统资源并不是新思想，如 C 和 Pascal（或其他语言）运行库就是将句柄返回给打开文件。句柄起着系统资源的间接指针作用。这种间接作用避免了应用程序直接随便地操作系统数据结构。

注意 执行程序组件和设备驱动程序能直接访问对象，因为它们在内核模式下运行，从而能访问系统内存中的对象结构。然而，它们必须通过递增打开的句柄计数或引用计数来说明它们对对象的使用，这样能够防止对象在使用时不会被释放（进一步细节参见本章后面所讲的“对象保留”）。

对象句柄还有其他优点。首先，除了文件句柄、事件句柄以及进程句柄的指向不同外，它们之间没有什么区别。这种相似性为引用对象提供了一致的接口，而不管对象的类型是什么。

其次，对象管理器具有创建句柄并定位句柄引用对象的独一无二的权利。这意味着对象管理器能仔细检查影响对象的每个用户模式操作，以查看调用程序的安全配置文件是否允许在有关对象上请求的操作。

实验：用 Nthandle 查看打开句柄

如下面的例子所示，Nthandle工具(也就是本书配套光盘上的 \ Sysint \ Nthandle.exe)可以显示任何或者所有进程打开的句柄：

```
C:\>nthandle -a -p system

Handle V1.2
Copyright (C) 1997-2000 Mark Russinovich
Systems Internals - http://www.sysinternals.com
-------------------------------  -------------------------------
System pid: 8
   4: Process
   8: Key   \REGISTRY
   c: Thread
  10: Key   HKLM\SYSTEM\ControlSet003\Control\ProcuctOptions
  14: Key   HKLM\SYSTEM\Setup
  18: Key   HKLM\SYSTEM\ControlSet003\Control\IDConfigDB\CurrentDockInfo
  1c: Key   HKLM\SYSTEM\ControlSet003\Hardware Profiles\Current
  20: Key   HKLM\HARDWARE\DESCRIPTION\SYSTEM\MultifunctionAdapter
```

上面显示了系统进程中的前八个打开的句柄。最先显示的是进程名和ID，紧接着每个句柄后面跟着一条线。每个句柄的句柄值、对象类型以及对象名字都被显示。因为指定了 -a 标记，没有名字的对象句柄（句柄号 0x4、0xc 和 0x18）也在其中。

对象句柄是与进程相关的句柄表的索引；执行程序进程（EPROCESS）块（第 6 章所述）指向该句柄表。第一个句柄的索引是 4，第二个是 8 等等。进程的句柄表包含了指向所有进程已打开句柄的对象的指针。句柄表是以 3 级方案实现的，类似于 x86 内存管理器实现虚拟到物理地址转换的方式（有关 x86 系统中内存管理的详细内容，参见第 7 章）。当进程被创建时，对象管理器就分配句柄表的顶级，其中包括指向中级表的指针；而中级表又包括指向次级句柄表的第一个指针数组。最低级包含有第一个次级句柄表。每级中的数组包括 256 个项，允许进程的初始句柄表拥有最多 255 个句柄，只有 255 个句柄项，而不是 256 个的原因是次句柄表中的最后一项已被初始化为 -1。-1 向对象管理器的句柄分配例程表明它已到达次句柄表的末项，而必须转到下一项或者在当前次句柄表中没有自由项时再分配新项。对象管理器将对象句柄值的低 24 位处理成 3 个 8 位字段，以此作为句柄表中 3 级的索引。图 3-16 解释了 Windows 2000 中句柄表的结构。

注意 在 Windows NT 4 中，句柄表由固定表头和可变大小部分组成。可变大小部分是句柄表项的数组，每个项描述一个打开的句柄。如果进程打开的句柄数比可变部分所能容纳的多，系统就分配新的、更大的数组，并将原来的数组复制到其中。Windows 2000 中的变化就是通过避免复制操作，并将整个句柄表必须锁定的地方最小化从而

提高句柄表的性能。

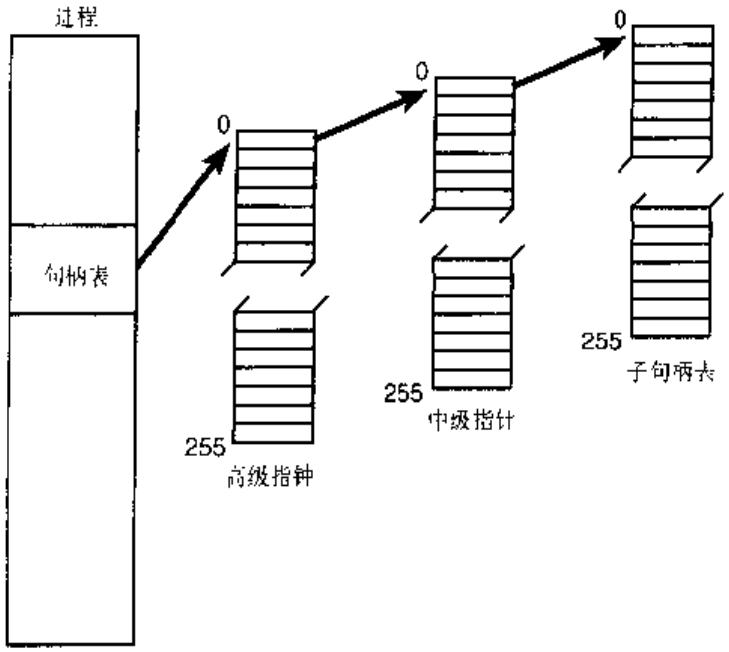


图 3-16 Windows 2000 中进程句柄表的结构

如图 3-17 所示，每个项由具有两个 32 位成员的结构组成。第一个 32 位成员含有同时指向对象头和四个标记的指针。由于对象头总是以 32 位对齐的，因而该字段的低 3 位可用做标记。项的高位可用做锁。当对象管理器将句柄转换成对象指针时，它在转换过程中锁定句柄项。因为所有对象都位于系统地址空间，所以对象指针的高位是固定的（即便是在具有/3GB 引导开关的系统上也可保证地址高于 0x80000000）。这样，对象管理器就能在句柄表项没锁定时，也很清楚，而在锁定项过程中固定该位并取得对象正确指针值。只有当进程创建新句柄或关闭现有句柄时，对象管理器需要利用与每个进程相关的句柄表锁来锁定进程的整个句柄表。句柄表项的第二个成员是对象的访问屏蔽（访问屏蔽在第 8 章讲述）。

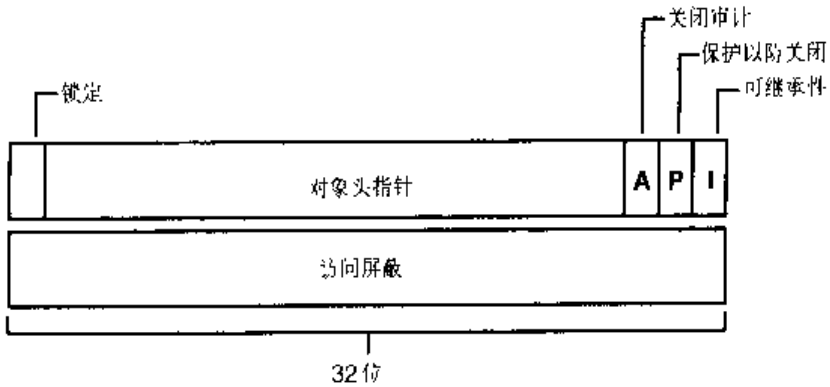


图 3-17 句柄表项结构

第一个标记是继承说明（inheritance designation）——即由该进程创建的进程是否将在它们的句柄表中得到该句柄的副本。正如前面所提到的，句柄继承可在句柄创建时或创建后用 SetHandleInformation 函数来指定。第二个标记显示调用程序是否被允许关闭该句柄（它也可用 Win32 的 SetHandleInformation 函数来指定）。第三个标记表明关闭该对象时是否产生审计消息（该标记不向 Win32 公开，对象管理器在内部使用它）。

系统组件和设备驱动程序经常需要打开用户模式应用程序无法访问的对象句柄。在 Windows NT 4 中，这样的句柄必须在 System 进程中创建，该进程是预留给系统线程和内核模式句柄的。

当内核模式函数在用户模式线程执行时，也就是在用户模式进程的环境中运行，而句柄表实际上属于用户模式进程，为了从 Windows NT 4 的系统进程引用句柄，函数必须以某种方式切换到系统进程。驱动程序和执行程序组件对上面的实现既可通过请求系统工作者线程（在本章后文的“系统工者线程”一节中讲述），它在系统进程的环境中执行并且代表对相应句柄引用的函数；也可通过 KeAttachProcess API 函数将当前线程的进程环境切换到系统进程环境。这两种方法都很烦琐，而且对性能有负面影响。

Microsoft 在 Windows 2000 中引入了一个称为内核句柄表（kernel handle table）的特殊句柄表（以 ObpKernelHandleTable 的名义在内部引用）。该表中的句柄只能从内核模式和各种进程环境来访问。这就是说，内核模式函数能在任何进程环境中引用句柄，而不影响性能。内核句柄表中的句柄与当前进程句柄表中句柄是有差别的，因为句柄的高位是固定的——即所有内核句柄表中句柄的值都大于 0x80000000

实验：利用内核调试程序查看句柄表

内核调试程序中的！handle 命令具有三个参数：

！handle <句柄索引> <标记> <进程 id>

句柄索引标识了句柄表中的句柄项(0表示显示所有句柄)。第一个句柄的索引是4，第二个是 8，以此类推。如，键入！handle 4 将会显示当前进程中的第一个句柄。

你能指定的标记是位屏蔽，其中位0表示仅仅显示句柄项中的信息，位1表示显示自由句柄（不是刚使用的句柄），而位 2 表示显示句柄所引用的对象的信息。下面的命令显示了有关进程 ID 是 0x408 的句柄表的所有细节。

```
kd> !handle 0 7 408

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

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

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

5. 对象安全

当打开文件时，必须指定是要读还是要写。如果试图对读访问打开的文件进行写，就会出错。类似地在执行程序中，当进程创建对象或打开现有的对象时，进程必须说明一组“期望的访问权限”(desired access rights)——也就是说，它想对对象所做的操作。可以请求一组适用于所有对象类型的标准访问权限（如读、写和运行），也可请求随对象类型的变化而变化的特定访问权限。例如，进程可以请求文件对象的删除访问或添加访问。类似的，它可能需要挂起或终止线程对象的能力。

当进程打开对象句柄时，对象管理器就调用安全引用监视器（security reference monitor），它是内核模式的安全系统的一部分，向它发送进程的期望的访问权限。安全引用监视器就检查进程安全描述符（security descriptor）是否允许进程请求的访问类型。如果允许，引用监视器就返回一组进程授权访问权限（granted acess rights），同时对象管理器将它们存储在所创建的对象句柄中。第8章将探讨安全系统如何确定谁取得对哪个对象的访问。

因此，不论进程的线程何时使用句柄，对象管理器都能快速检查存储在句柄中的授权访问权限是否对应于线程所调用的对象服务所隐含的用法。比如，如果调用程序请求对区域对象的读访问，但接着又调用对它的写服务，那么该服务就要失败。

6. 对象保留

由于所有访问对象的进程必须首先打开其句柄，因此对象管理器能轻易的跟踪这些进程的数量，甚至哪些在使用对象。跟踪句柄是实现对象保留的一部分——即保留对象到它们使用完再删除。

对象管理器分两个阶段实现对象保留。第一个阶段称为名字保留，它由存在的对象打开的句柄的数量控制。每次进程打开一个对象的句柄，对象管理器就在对象头中递增已打开的句柄计数器。当进程完成对象的使用，并关闭它的句柄时，对象管理器就减少已打开句柄计数器。当计数器减少到0时，对象管理器就从其全局名字空间中删除对象名，这一删除操作防止了新进程打开该对象的句柄。

对象保留的第二个阶段是当对象不再使用时，停止保留对象本身（即删除它们）。由于操作系统代码通常利用指针而不是句柄访问对象，因此对象管理器还必须记录它分配给操作系统进程的对象指针的数目。每次分给对象一个指针，就为该对象增加一个引用计数；而当内核模式组件用完该指针后，就调用对象管理器减少对象的引用计数。系统在递增句柄计数的同时递增引用计数，类似在句柄计数减少时减少引用计数，因为句柄也是对必须跟踪的对象的引用(对于对象保留的进一步细节，参见DDK文档中有关函数ObReferenceObjectByPointer和ObDereferenceObject的部分)。

图3-18显示了两个正被使用的事件对象。进程A的第一个事件是打开的。而进程B的两个事件都是打开的。此外，第一个事件正被某个内核模式结构引用。它的引用记数是3。因此即使进程A和进程B关闭了它们的第一个事件对象的句柄，事件对象将继续存在，因为它的引用计数是1。然而当进程B关闭了它的第二个事件对象的句柄时，该对象就会被释放

因此，即使对象的打开句柄计数为0时，对象的引用数可能仍是正值，这表明操作系统还在使用该对象。最终，引用计数也将减少到0，而对象管理器就从内存中删除对象。

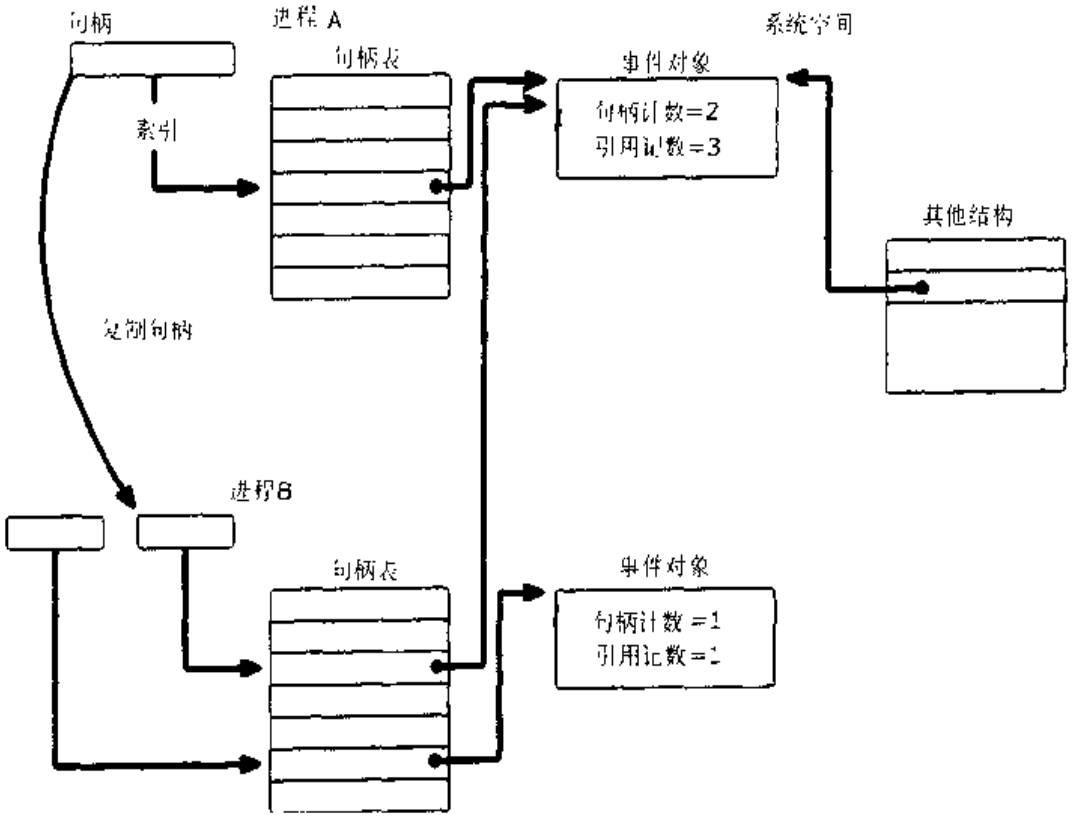


图 3-18 句柄和引用计数

因为对象保留的工作方式，仅仅通过打开对象的句柄，应用程序就能保证对象和其名字保留在内存中。程序员在编写包括两个协作进程的应用程序时，不需考虑其中一个进程可能会在另一个进程用完对象之前将它删除。此外，如果操作系统仍在使用一个对象，那么关闭应用程序的对象句柄不会导致删除。如一个进程可能创建第二个进程在后台执行某程序，然后立即关闭该进程的句柄。由于操作系统需要第二个进程来运行程序，它就保留对其进程对象的引用。只有当后台程序运行完毕，对象管理器才会递减第二个进程的引用计数，然后将它删除。

7. 资源统计

和对象保留一样，资源统计（resource accounting）与对象句柄的使用紧密相关。已打开的句柄计数为正，表明某个进程正在使用该资源。它也表明某个进程因为对象占用内存而被统计。当对象句柄计数减至为 0 时，正在使用该对象的进程就不再被统计。

很多操作系统采用配额系统来限制进程对系统资源的访问。然而强加给进程的配额类型有时是多样而复杂的，并且跟踪配额的代码分布在整个操作系统中。如在某些操作系统中，I/O 组件可能记录并限制进程所能打开的文件数目，而存储器组件可能对进程的线程所能分配的内存大小加以限制。一个进程可能限制用户所能创建的新进程的最大数目，或在一个进程之内的最大线程数目。这些限制都会被跟踪并在操作系统的不同部分实现。

相对而言，Windows 2000 对象管理器为资源统计提供了一个重要工具。每个对象头都有一称为配额量（quota charge）的属性，它在进程的线程打开对象句柄时，记录对象管理器从分配给进程的页面或非页交换区配额中减去了多少。

Windows 2000 中的每个进程都指向配额数据结构，它记录了非页交换区、页交换区以及页

式文件使用的限制和当前值。然而，交互式会话中的所有进程共享同一配额块（不存在有公开文档的方法来创建具有自身配额块的进程），而系统进程（如服务）没有配额限制。

尽管系统实现了跟踪配额的代码，但它并没有体现这些。进程的页交换区和非页交换区配额缺省为 0（没限制）。可用注册值来覆盖这些缺省值，但是限制是“软的”，因为当超过配额时，系统会试图自动增加进程配额。如果打开一个对象会超过页式或非页式配额，内存管理器将被调用来查看配额是否能增加。内存管理器则根据系统交换区中剩余内存大小来做决策。如果它决定配额无法增加，那么对象的打开请求就会因“配额超支”而失败。但在多数系统中，配额随需要持续增加。

实验：查看进程配额

利用 Process Explode 实用程序，Pview.exe（在 www.reskit.com 网站上可获得）可查看进程的页交换区、非页交换区和页式文件的当前使用、峰值使用以及配额（极限）。（Performance 工具只显示了使用信息，而没有配额）。在图 3-19 中，所选进程的页交换区使用峰值为 1062KB，当前使用为 1028KB，配额为 1504KB：

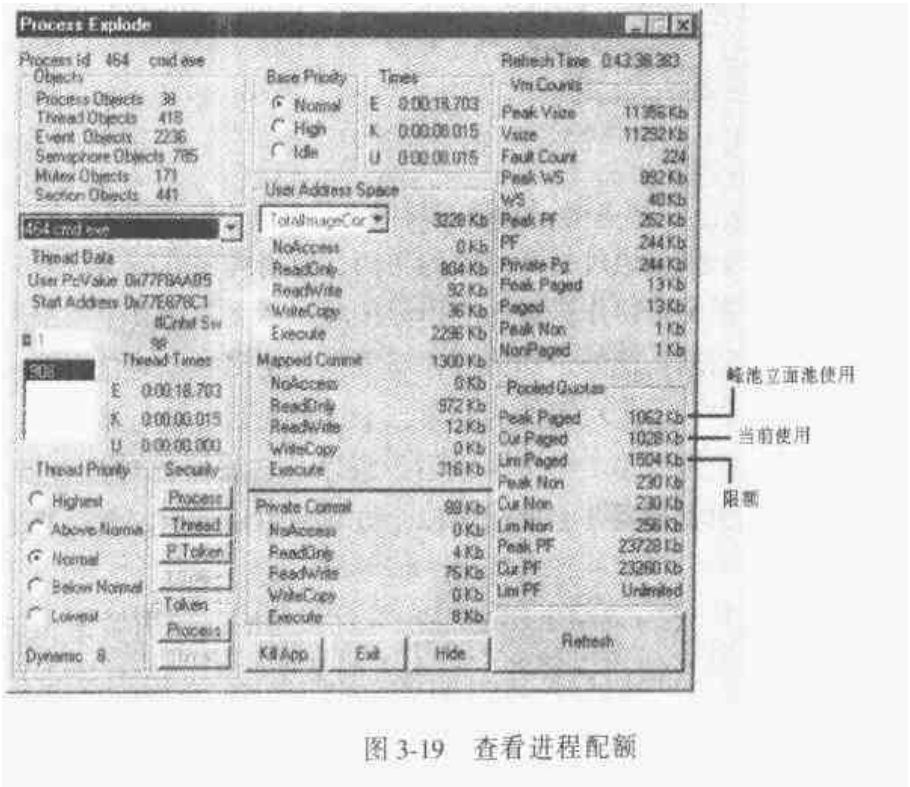


图 3-19 查看进程配额

8. 对象名字

创建多对象时的一个重要考虑是设计能对对象成功跟踪的系统。要达到这一点，对象管理器需要下列方法：

- 区分不同对象的方法。
- 查找和检索特定对象的方法。

第一条要求可通过允许给每个对象赋名实现。这是对多数操作系统所提供的能力的扩充——如对选择的资源、文件、管道或共享内存块等进行命名的能力。相反，执行程序允许任

何由对象表示的资源有一个名字。第二条要求，查找和检索对象，也可由对象名字来实现。如果对象管理器以名字存储对象，它就能通过搜索其名字来查找对象。

对象名字也能满足第三条要求，即允许进程共享对象。执行程序的对象名字空间是全局的，对系统中的所有进程是可见的。一个进程能创建对象，并将其名字放置在全局名字空间里，而第二个进程通过指定对象名字能打开该对象的句柄。如果对象不需要以这种方式共享，其创建程序就不需要给它赋名。

为提高效率，每次使用对象时对象管理器并不搜索对象名字。相反，它只在两种情况下搜索对象名字。第一种是当进程创建一个有名字对象时，对象管理器搜索名字并验证在它将新名字存储到全局名字空间之前，该名字确实不存在。第二种情况是当进程打开有名字对象句柄时，对象管理器搜索该名字，查找该对象，然后将对象句柄返回给调用程序；此后调用程序利用该句柄来引用对象。在搜索名字时，对象管理器允许调用程序选择大小写敏感或大小写不敏感搜索，这是 POSIX 和使用大小写敏感文件名字的其他环境所支持的一个特性。

对象名字存储的地方取决于对象类型。表 3-8 列出了所有 Windows 2000 系统中的标准对象目录以及存储在其中的有名字的对象类型。在所列出的目录中，只有 \ BaseNamedObjects 和 \ ?? 对用户程序是可见的。

由于基本的内核对象如互斥、事件、信号、可等待定时器和区域都将它们的名字存储在同一个的目录中，所以这些对象中的任何两个都不能有相同的名字，即使它们属于不同类型。这一限制强调了小心选择名字并避免与其他名字冲突的需要（如前缀名字用公司和产品名）。

表 3-8 标准对象目录

目 录	所保存对象名的类型
\ ??	MS - DOS 设备名（\ DosDevice 是到该目录的符号链接
\ BaseNamedObjects	互斥、事件、信号量、可等待定时器和区域对象
\ Callback	回调对象
\ Device	设备对象
\ Driver	驱动程序对象
\ FileSystem	文件系统驱动程序对象和文件系统识别程序设备对象
\ KnownDlls	区域名和已知 DLL（启动时由系统映射的 DLL）的路径
\ Nls	映射的国家语言支持表的区域
\ ObjectTypes	对象类型名字
\ RPC Control	远端过程调用（RPC）所使用的端口对象
\ Security	安全子系统相关的对象名字
\ Windows	Win32 子系统端口和窗口站

对象名字对单台计算机（或对多处理器计算机上的所有进程）来说是全局性的，但它们在网络上是不可见的。然而，对象管理器的解析方法使得有可能访问存在于其他计算机上的有名字对象。如 I/O 管理器提供的文件对象服务就将对象管理器的函数扩展到了远程文件。当被请求打开远程文件对象时，对象管理器调用解析方法，从而允许 I/O 管理器截取请求并将它发送给网络重定向器，网络重定向器是通过网络访问文件的驱动程序。远程 Windows 系统中的服务器代码将调用该系统上的对象管理器和 I/O 管理器查找文件对象，并通过网络返回信息。

实验：查看基本的有名字对象

利用本书配套光盘上的ObjectViewer实用程序可以查看有名字的基本对象列表(在Platform SDK 中也有该实用程序的另一版本)。运行 \ Sysint \ Winobj. exe，并单击 \ BaseNamedObjects，如图 3-20 所示：

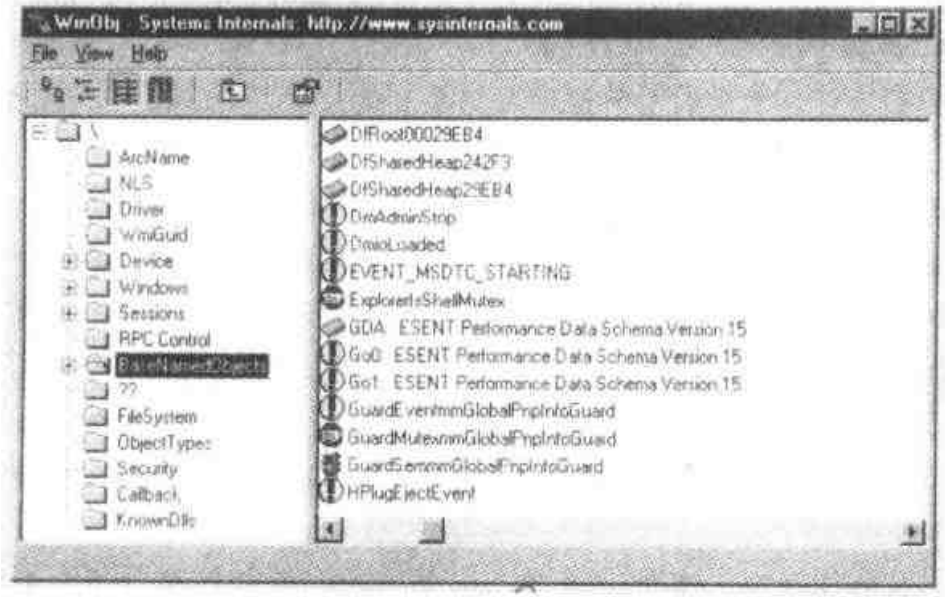


图 3-20 查看基本的有名字对象列表

- 有名字对象显示在右端。图标显示对象类型。
- 互斥用停止符号表示。
- 区域（Win32 文件映射对象）显示为内存芯片。
- 事件用感叹号表示。
- 信号量用类似于交通指示灯的图标表示。
- 符号链接的图标则为曲线箭头。

对象目录 对象目录对象（Object directory object）是支持层次名字结构的对象管理器方法。该对象类似于文件系统目录，并包括其他对象的名字，甚至可能还有其他对象目录。对象目录对象保留足够的信息将这些对象名字转换为指向对象本身的指针。对象管理器利用这些指针构建它所返回给用户模式调用程序的对象句柄。内核模式代码（包括执行程序组件和设备驱动程序）和用户模式代码（如子系统）都能创建对象目录用来存储对象。如 I/O 管理器创建名为 \ Device 的对象目录，其中包含代表 I/O 设备的对象名。

符号链接 在文件系统中（例如在 NTFS 和一些 UNIX 系统中），符号链接允许用户创建一个文件名或目录名，操作系统在使用它们时将其转换成不同文件或目录名。使用符号链接能方便地允许用户间接地共享文件或目录内容，在一般层次目录结构的不同目录之间创建交叉链接。

对象管理器实现称为“符号链接对象”的对象，为在它的对象名字空间中对象名字完成一个相似的功能。符号链接能在对象名字符串中的任何地方出现。当调用程序引用符号链接对象名时，对象管理器就遍历对象名字空间，直到它找到符号链接对象。它在符号链接中搜索查找用以代替符号链接名字的字符串，然后重新开始查找名字。

执行程序使用符号链接的一个体现是将 MS－DOS 风格的设备名转换成 Windows 2000 的内部设备名。在 Win32 中，用户用 A:、B:、C: 等等来引用软驱和硬盘驱动器。此外，用户能用 subst（替换）命令创建伪驱动符或将驱动符映射为网络共享。一旦被创建，这些驱动器名必须对系统上所有进程可见。

Win32 子系统通过将驱动符放置在对象管理器名字空间的 \ ?? 对象目录下（在 Windows NT 4 之前，该目录名为 \ DosDevices，由于名字按字母顺序放置，出于性能原因，它被重新命名为 \ ??）来保护它们。当用户或应用程序创建一新驱动符时，Win32 子系统就在 \ ?? 对象目录下添加另一对象。

9. 终端服务名字空间

开发 Windows NT 时已经假设只有一个用户会交互式地登录到该系统，而系统只运行任何交互式应用程序的一个实例。当安装 Windows 2000 终端服务时，就违反了这些假定，因此为使 Windows 2000 支持多交互式用户需要改变对象管理器。

登录到控制台的用户可以访问全局名字空间，该名字空间作为名字空间的第一个实例。远程登录到终端的用户被允许查看名字空间的视图，称之为局部名字空间。为远程用户本机化的名字空间包括 \ DosDevices、\ Windows 和 \ BaseNamedObjects。分别拷贝名字空间的相同部分就是人们所熟悉的名字空间实例。\ DosDevices 实例使得每位用户可拥有不同的驱动符和 Win32 对象如串行端口。\ Windows 目录是 Win32k. sys 创建交互式窗口站的地方，\ WinSta0. A 终端服务环境能支持多交互式用户，但每位用户都需要 WinSta0 的单独版本，使用户感觉就像在 Windows 2000 中访问预定义的交互式窗口站。最后，应用程序和系统在 \ BaseNamedObject 中创建共享对象，包括事件、互斥和内存区域。如果两个用户同时运行一个应用程序，而该应用程序创建一个有名字对象 ApplicationInitialized，那么每个用户会话都必须拥有该对象的私有版本以便该应用程序的两个实例不会因为访问相同对象而相互影响。

通过创建提到的三个目录的私有版本，这三个目录在与用户会话相关的 \ Session \ X（其中 X 是用户的会话标识）目录下，对象管理器实现局部名字空间。如，当远程会话 2 的 Win32 应用程序创建了一有名字事件，对象管理器透明地将对象名从 \ BaseNamedObjects 重定向到 \ Session \ 2 \ BaseNamedObjects。

所有与名字空间管理相关的对象管理器函数都能意识到实例目录，并使非控制台会话好像在使用与控制台会话相同的名字空间。为了优化，进程对象具有一名为 DeviceMap 的字段，它指向同一会话中其他进程共享的数据结构。它确定了属于该会话的 \ DosDevices 对象管理器目录的位置以及对该会话有效的驱动符表。对象管理器在 \ DosDevices 中搜索对象时利用该数据结构。

在一定情况下，终端服务察觉的应用程序需要访问控制台会话中的对象，即使应用程序在远程会话中运行。应用程序在同步执行在其他远程会话或控制台会话中运行的实例时，可能需要实现这一点。对于这些情形，对象管理器提供了特殊的重载“\ Global”，应用程序能把它作为前缀加到任何对象名并访问全局名字空间。例如，打开了名为 \ Global \ ApplicationInitialized 的对象的会话 2 中的应用程序会被引导到 \ BasedNamedObjects \ ApplicationInitialized 而不是 \ Session \ 2 \ BaseNamedObjects \ ApplicationInitialized。

实验：查看名字空间实例

可以查看安装了终端服务的 Windows 2000 Server、Advanced Server 或 Datacenter Server 系统上的对象管理器实例。利用终端服务客户机器登录到服务器，并运行本书配套光盘上的 \ Sysint \ Winobj. exe 实用程序。

点击 \ Sessions 目录，能看到一子目录，其每个被激活的远程会话都有一个数字名。如果打开其中一个目录，就能看到名为 \ DosDevices、\ Windows 和 \ BaseNamedObjects 的子目录，它们是会话的局部名字空间子目录。如图 3-21 所示显示了局部名字空间：

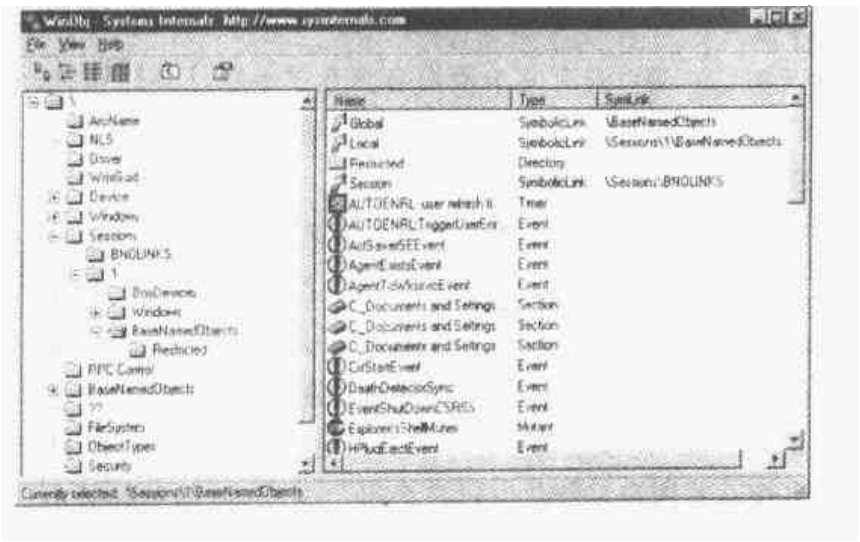


图 3-21 查看名字空间

3.3 同步

“互斥”是操作系统开发的一个重要概念。它保证一次有且只有一个线程能访问一定的资源。当资源本身不能用于共享访问或共享会引起无法预测的后果时，互斥就是必要的。例如，如果两个线程同时将文件复制到打印机端口，它们的输出可能会被分散。类似的，如果一个线程读一个存储单元，而另一个则向它写，第一个线程就会收到无法预测的数据。一般来说，可写资源不能无限制的被共享，而不需修改的资源则能共享。图 3-22 显示了当在不同处理器上运行的两个线程都向一个环形队列写数据时，发生的情况。

因为第二个线程在第一个线程完成更新队列尾端指针之前已取得队列尾端指针的值，因此，第二个线程在第一个线程已经占用的相同位置插入数据，这就重写了数据使得一个队列位置为空。尽管该图显示的是多处理器系统上可能发生的情况，但只要操作系统在第一个线程更新队尾指针前需将环境切换到第二个线程，在单处理器上也会出现同一错误。

访问非共享资源的代码段称为临界区。为确保代码正确，一次只有一个线程在临界区中执行。当一个线程在写文件、更新数据库或修改共享变量时，其他的线程不允许访问该资源。图 3-22 所示的伪代码就是临界区，它在无互斥的情况下错误地访问了共享数据结构。

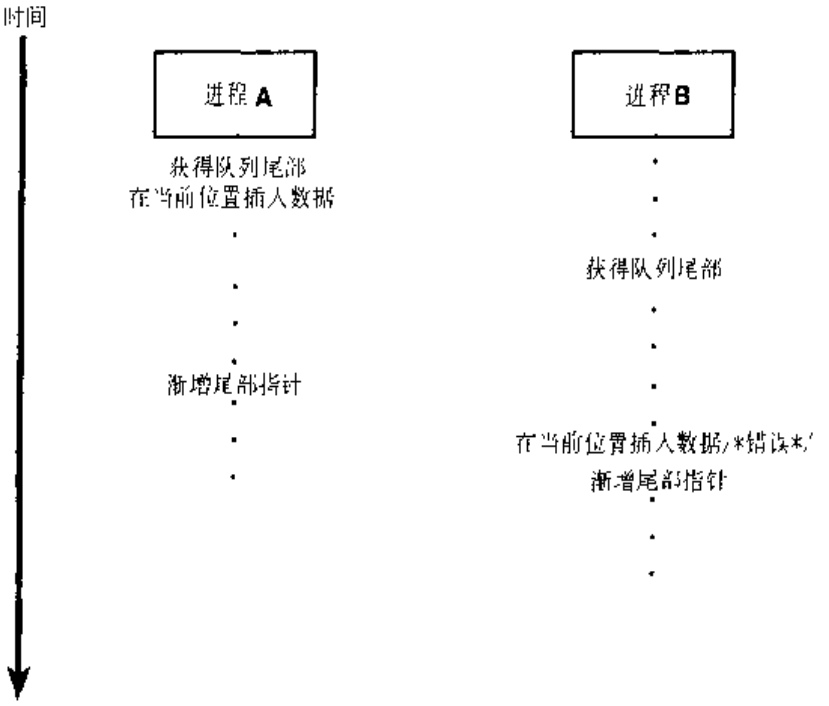


图 3-22 内存的错误共享

互斥问题对所有操作系统都十分重要，而对紧耦合的对称多处理（SMP）操作系统如 Windows 2000 尤为重要，在这样的系统中，相同系统代码同时在一个以上的处理器上运行而共享存储在全局内存上的一定数据结构。Windows 2000 的内核任务就是提供系统代码能使用的机制以防止两个线程同时修改同一结构。内核提供了自身和其他执行程序用来同步访问全局数据结构的互斥原语。

在下列章节中，我们将看到内核如何使用互斥来保护全局数据结构以及内核为执行程序提供了互斥和同步机制，而执行程序依次又为用户模式提供了互斥和同步机制。

3.3.1 内核同步

在执行过程中的不同阶段，内核必须保证有且仅有一个处理器在临界区内执行。内核临界区就是修改全局数据结构的代码段，如内核调度程序数据库或其 DPC 队列等。除非内核能保证线程以互斥方式访问这些数据结构，否则操作系统就无法正确工作。

涉及最多的领域就是中断。如当发生中断时，内核可能正在更新全局数据结构，而其中断处理例程也在修改该结构。简单的单处理器操作系统有时能在它们每次访问全局数据时通过禁用所有中断而避免这种情况。但 Windows 2000 内核提供了更完善的解决方法。在使用全局资源之前，内核暂时屏蔽中断，中断处理程序也使用该资源。通过将处理器的 IRQL 提升到任何可能访问全局数据的中断源所使用的最高级别，就能实现这一点。如处于 DPC/调度级的中断导致调度程序运行，而调度程序使用调度程序数据库。因此，内核中任何其他使用该调度程序的部分都将 IRQL 提升到 DPC/调度级，在使用调度程序数据库之前屏蔽 DPC/调度级的中断。

这种策略对于单处理器系统来说是很好的，但对多处理器系统配置却还有所不足。在一个处理器上提升 IRQL 并不能防止中断在另一个处理器上出现，内核还要保证在好几个处理器上进行互斥访问。

内核用来实现多处理器互斥的机制称为自旋锁。自旋锁是与全局数据结构相联系的锁定原语，诸如图 3-23 所示的 DPC 队列。

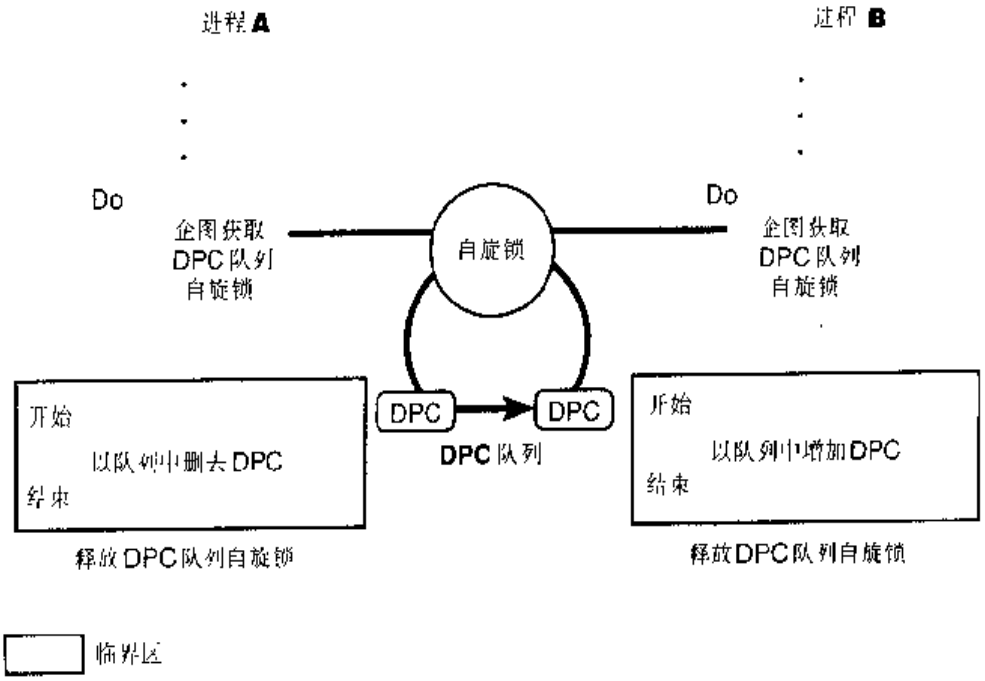


图 3-23 使用自旋锁

在进入图中任何临界区之前，内核必须获得与受保护的 DPC 队列相联系的自旋锁。如果自旋锁繁忙，内核就一直企图取得该锁直到成功。自旋锁的取名来源于以下事实：内核（以及处理器）在取得该锁之前处于中间过度状态——“旋转”。

自旋锁和它们保护的数据结构一样，驻留在全局内存中。由于速度以及为利用底层处理器体系结构所提供的任何锁定机制，获取和释放自旋锁的代码都是用汇编语言编写的。在很多体系结构中，自旋锁是由硬件支持的测试和设置操作实现的，它测试锁变量的值并在基本指令中获取该锁。在一个指令中测试并获得自旋锁防止了第二个线程在第一个线程从测试变量到取得该锁的时间内抢夺该锁。

Windows 2000 中的所有自旋锁都有相关的 IRQL，它总是处于 DPC/调度或更高级。这样，当一个线程试图获取自旋锁时，所有其他处于自旋锁的 IRQL 或更低的活动都在该处理器上停止。由于线程调度在 DPC/调度级发生，因此持有自旋锁的线程从来不会因为 IRQL 屏蔽调度机制而被抢先。这种屏蔽允许执行由自旋锁保护的临界区的代码继续执行以便能快速释放自旋锁。内核使用自旋锁十分小心，在它持有自旋锁时，对指令数目进行最小化。

注意 IRQL 是单处理器上有效的同步机制，因此单处理器 HAL 的自旋锁获取和释放函数不实现自旋锁——它们仅仅是提升或降低 IRQL。

内核通过一系列内核函数包括 KeAcquireSpinlock 和 KeReleaseSpinlock，使得执行程序的其他部分能得到自旋锁。例如设备驱动程序需要自旋锁来保证设备注册表以及其他全局数据结构一次仅被设备驱动程序的一部分（和仅从一个处理器）访问。用户程序并不使用自旋锁——它们使用下节将要描述的对象。

内核自旋锁对于使用它们的代码是有限制的。如前所述，由于自旋锁的 IRQL 总处于 DPC/调度或更高级，占有自旋锁的代码如果试图使调度程序执行调度操作或引起页面错误，会使系统崩溃。

Windows 2000 引入了称为队列化自旋锁的特殊自旋锁类型，它仅供内核使用而不输出到执行程序组件或设备驱动程序。队列化自旋锁在多处理器上比标准自旋锁可伸缩性更好。队列化自旋锁工作方式如下：当处理器想取得当前被占用的队列化自旋锁时，它将其标识符放在与自旋锁相关的队列中。当占用自旋锁的处理器释放它时，它就将该锁传递给在队列中标识的第一个处理器。同时，等待繁忙自旋锁的处理器并不检查自旋锁本身的状态，而是检查队列中它之前的处理器所设置的每个处理器标记的状态，用来表示是否轮到等待处理器的次序。

队列化自旋锁引起每个处理器标记而不是全局自旋锁的旋转这一事实有两种结果。首先处理器的总线不会因为处理器之间的同步而严重阻塞。其次等待组中的随机处理器不会获得自旋锁，队列化自旋锁对锁实行先进先出（FIFO）的原则。先进先出原则意味着在处理器之间访问相同锁的性能更趋一致。

Microsoft 还没将所有的内核锁转变为队列化自旋锁，而只有保护内核数据结构的大约六个锁，如缓冲管理器的数据库、调度程序的线程数据库以及内存管理器的物理内存数据库。

实验：查看队列化自旋锁

利用！qlock 内核调试程序命令可以查看队列化自旋锁的状态。该命令只在多处理器系统上才有意义，因为单处理器 HAL 不实现自旋锁。在下面的例子中，调度程序数据库队列化自旋锁由处理器 1 占用，而其他队列化自旋锁未被获取（调度程序数据库在第 6 章描述）。

```
kd> !qlocks
Key: O = Owner, 1-n = Wait order, blank = not owned/waiting, C = Corrupt

                         Processor Number
    Lock Name      0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

KE - Dispatcher       0
KE - Context Swap
MM - PFN
MM - System Space
CC - Vacb
CC - Master
```

3.3.2 执行程序同步

在多处理器环境中，内核外的执行程序软件也需要同步访问全局数据结构。如内存管理器只有一个页面帧数据库，它以全局数据结构存取，而设备驱动程序需要确保它们能获得对设备的惟一访问。通过调用内核函数，执行程序能创建自旋锁、获取并释放它。

然而自旋锁只能部分地满足执行程序同步机制的需要。由于在自旋锁上等待实际上会导致

终止处理器，因此，自旋锁只能在下列严格限制的情况下使用：

■ 受保护的资源必须快速地被访问而不需要与其他代码进行复杂的交互。

■ 临界区代码不能从内存中调页、不能引用可调页数据、不能调用外部过程（包括系统服务）以及不能产生中断或异常。

这些限制条件是有限的，而且不能在所有情况下满足。此外，执行程序除了实现互斥还要实现其他类型的同步，而且还必须为用户模式提供同步机制。

内核以内核对象的形式为执行程序提供了其他的同步机制，即人们所熟知的调度程序对象（dispatcher object）。用户可见的同步对象从这些内核调度程序对象中获得同步能力。每个支持同步的用户可见对象都至少封装了一个内核调度程序对象。通过 WaitForSingleObject 和 WaitForMultipleObjects 函数，执行程序的同步语义对 Win32 程序员是可见的。这两个函数是 Win32 子系统通过调用对象管理器提供的类似系统服务而实现的。Win32 应用程序的线程能与 Win32 进程、事件、信号量、互斥、可等待定时器、I/O 完成端口或者文件对象同步。

另一类值得注意的执行程序同步对象称为“执行程序资源”（executive resource）。执行程序资源同时提供了独占访问（如互斥）和共享读访问（多个读者共享对结构的只读访问）。但它们只能被内核模式代码使用，因此不能从 Win32 API 访问。执行程序资源不是调度程序对象，而是数据结构，它们由非页交换区直接分配，这些非页交换区由其自身的专门服务（specialized services）对其进行初始化、锁定、释放、查询等等。执行程序资源结构在 Ntddk.h 中定义，而其支持例程在 DDK 参考文档中有叙述。

下面的章节描述在调度程序对象上等待（waiting on dispatcher object）的实现细节。

1. 在调度程序对象上等待

线程通过在对象的句柄上等待的方式与调度程序对象同步。这样做导致内核挂起线程，并将其调度程序状态从运行变为等待，如图 3-24 所示。内核将线程从调度程序就绪队列中取消，而不再考虑执行它。

注意 图 3-24 是突出就绪、等待和运行状态的进程状态迁移图（在对象上等待相关的状态）间的过渡图。其他状态在第 6 章讲述。

在任意特定时刻，同步对象都处于两种状态中的一种，要么是“有信号状态”（signaled state）要么是“无信号状态”（nonsignaled state）。线程只有在内核将其调度程序状态从等待变为就绪时才能继续执行。当线程正在其句柄上等待的调度程序对象也经历状态改变，即从无信号状态改变为有信号状态时（如线程设置事件对象），改变发生。为了和对象同步，线程调用对象管理器提供的等待系统服务之一，将句柄传递给它想与之同步的对象。线程能在一个或几个对象上等待，而且只要在一定时间内还没结束，就能被指定取消等待。不论内核何时设置对象为有信号状态，内核的 KiWaitTest 函数都检查是否有线程在对象上等待。如果有，内核就将线程中的一个或几个从它们的等待状态中释放以便它们能继续执行。

下列设置事件的例子说明了同步如何与线程调度交互：

1）用户模式线程在事件对象的句柄上等待。

2）内核将线程的调度状态从就绪改为等待，然后将该线程添加到线程等待事件的链表中。

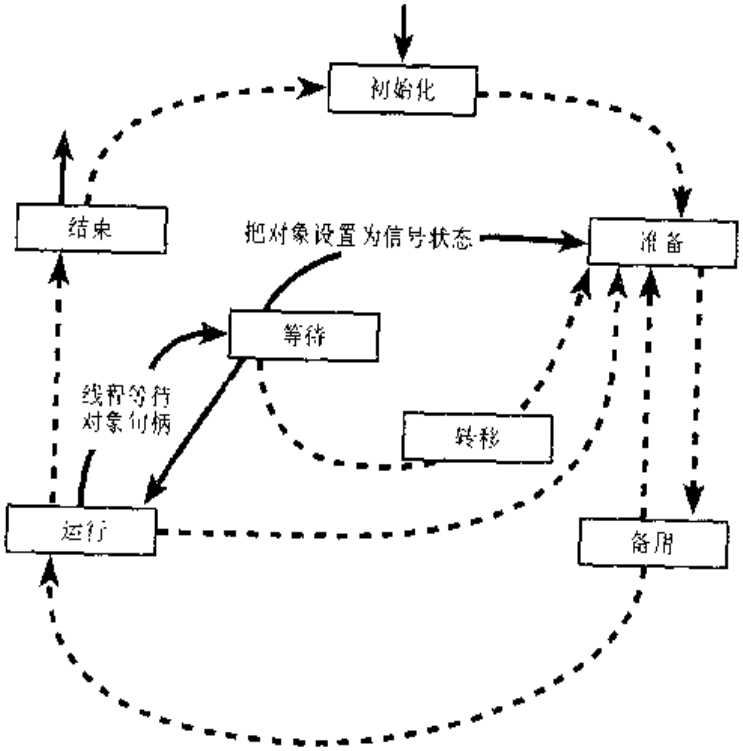


图 3-24 访问调度程序对象

3）另一线程设置事件。

4）内核沿着事件等待的线程链表搜索。如果线程的等待条件被满足，内核将线程的状态由等待变为就绪。如果它是可变优先级线程，内核也可能增加其执行优先级。

5）因为新线程已经就绪准备执行，调度程序重新调度。如果它发现正在运行的线程优先级低于新就绪线程的优先级，它就抢先低优先级线程，并发出软件中断以引发到高优先级线程的环境切换。

6）如果没有处理器被抢先，调度程序将就绪线程置于调度程序就绪队列中供以后调度。

2. 什么情况把对象置为有信号状态

不同对象的有信号状态的定义是不同的。线程对象在其生存期内处于无信号状态，在线程终止时被内核设置为有信号状态。类似的，当进程的最后线程终止时，内核设置进程对象为有信号状态。相反，定时器对象就象闹钟一样被设置为在一定时间内“响铃”。当其时间过期，内核就将定时器对象设置为有信号状态。

在选择同步机制时，应用程序必须考虑支配不同同步对象行为的规则。当对象被设置为有信号状态时，线程的等待状态是否结束取决线程正在其上等待的对象类型，如表 3-9 所示。

当对象被设置为有信号状态时，等待线程通常立即从其等待状态中被释放。一些导致状态变化的内核调度程序对象和系统事件如图 3-25 所示。

表 3-9 信号状态的定义

对象类型	何时设置为有信号状态	对等待线程的影响
进程	最后线程终止	都释放
线程	线程终止	都释放
文件	I/O 操作完成	都释放

○ 一些线程可能不止在一个对象上等待，所以它们继续等待。

（续）

对象类型	何时设置为有信号状态	对等待线程的影响
事件（通知类型）	线程设置事件	都释放
事件（同步类型）	线程设置事件	一个线程被释放，事件对象被重设置
信号量	信号量计数减 1	一个线程被释放
定时器（通知类型）	设置时间到达或时间间隔过期	都释放
定时器（同步类型）	设置时间到达或时间间隔过期	一个线程被释放
互斥	线程释放互斥	一个线程被释放
文件	I/O 完成	所有线程释放
队列	队列中放置项	一个线程被释放

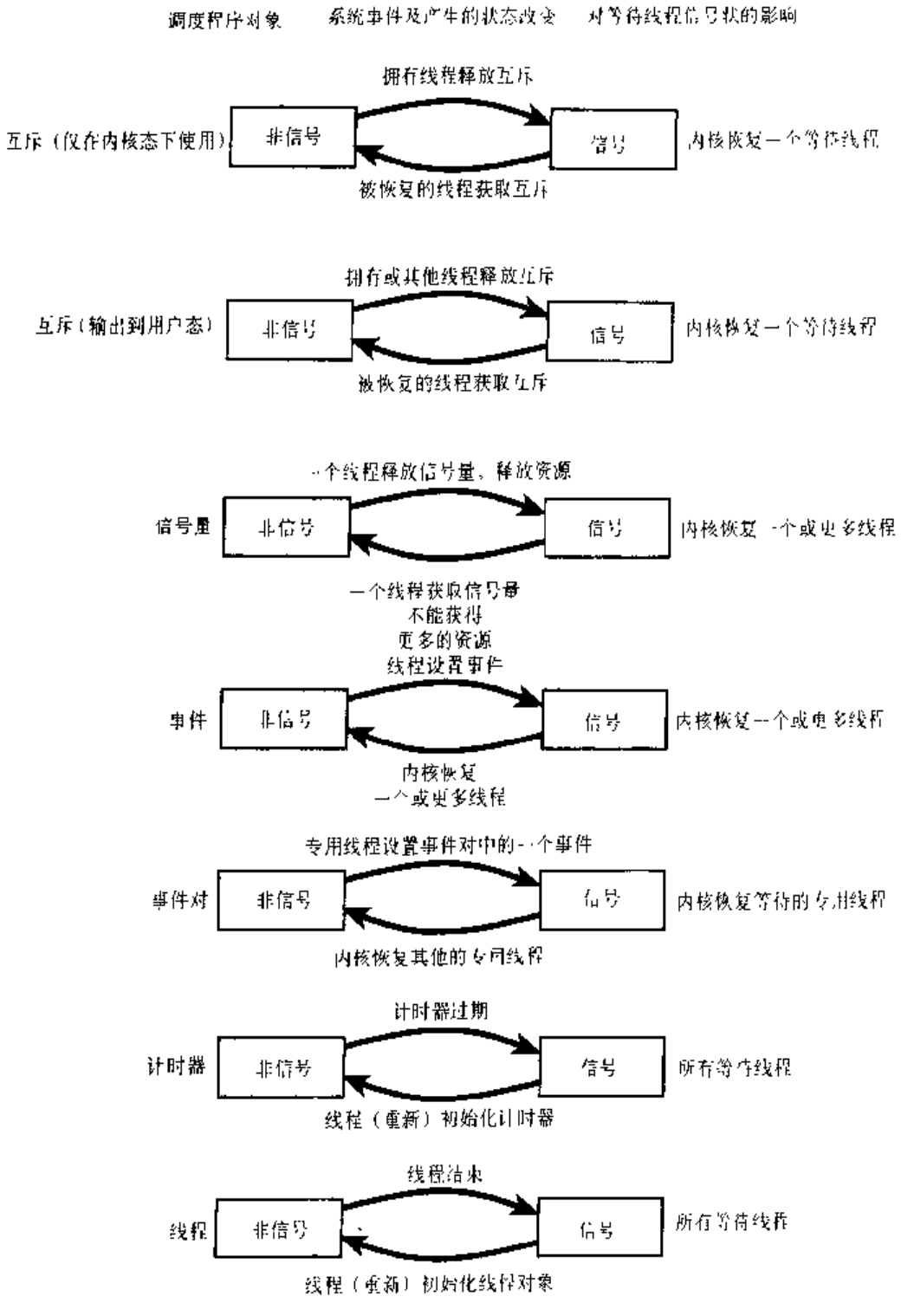


图 3-25 选定的内核调度程序对象

例如，通知事件对象（在 Win32 API 中称为手工重设置事件）用来通知一定事件的发生。当事件对象被设置为有信号状态时，所有在事件上等待的线程都被释放。但一次同时在好几个对象上等待的线程除外，这种线程可能需要继续等待，直到其他对象进入有信号状态。

不同于事件对象，互斥对象拥有与其相关的自主权。它用来获得对资源的互斥访问，而且一次只有一个线程能占用互斥。当互斥对象空闲时，内核将其设置为有信号状态，然后选择一等待线程执行。内核所选定的线程获得互斥对象，而所有其他线程继续等待。

这些简单讨论并不是在枚举使用不同执行程序对象的所有理由和应用，而是要列出它们的基本功能和同步行为。关于如何在 Win32 程序中使用这些对象，参见有关同步对象的 Win32 参考文档或 Jeffery Richter 的“Microsoft Windows 的编程应用”（Programming Application for Microsoft Windows）。

3. 数据结构

跟踪谁在什么上等待的两个关键数据结构是调度程序头（dispatcher header）和等待块（wait blocks），这两个结构都公开定义在 DDK 引用文件 Ntddk.h 中。为方便起见，定义重列出如下：

```
typedef struct _DISPATCHER_HEADER {
    UCHAR Type;
    UCHAR Absolute;
    UCHAR Size;
    UCHAR Inserted;
    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;

typedef struct _KWAIT_BLOCK {
    LIST_ENTRY WaitListEntry;
    struct _KTHREAD *RESTRICTED_POINTER Thread;
    PVOID Object;
    struct _KWAIT_BLOCK *RESTRICTED_POINTER NextWaitBlock;
    USHORT WaitKey;
    USHORT WaitType;
} KWAIT_BLOCK, *PKWAIT_BLOCK, *RESTRICTED_POINTER PRKWAIT_BLOCK;
```

调度程序头包含了对象类型、有信号状态和在该对象上等待的线程列表。等待块表示在对象上等待的线程。每个处于等待状态的线程都有等待块链表，它表示线程在其上等待的对象。而每个调度程序对象都有等待块链表，它代表了哪些线程在该对象上等待。设置该链表使得当调度程序对象被置为有信号状态时，内核能快速地确定谁在该对象上等待。等待块有一个指向被等待对象的指针，一个指向在该对象上等待的线程的指针以及一个指向下一个等待块的指针（如果线程在一个以上的对象上等待）。它还记录等待类型（任何或所有）以及在 WaitForMultipleObjects 调用时，该项在线程所传递的句柄数组中的位置（如果线程仅在一个对象上等待则为 0）。

图 3-26 显示了调度程序对象与等待块和线程的关系。在该例中，线程 1 在对象 B 上等待，线程 2 在对象 A 和 B 上等待。如果对象 A 被置为有信号状态，内核将发现由于线程 2 也在另

一对象上等待，因此线程 2 不能就绪执行。另一方面，如果对象 B 被置为有信号状态，内核立即让线程 1 就绪以备执行，因为它不在另一个对象上等待。

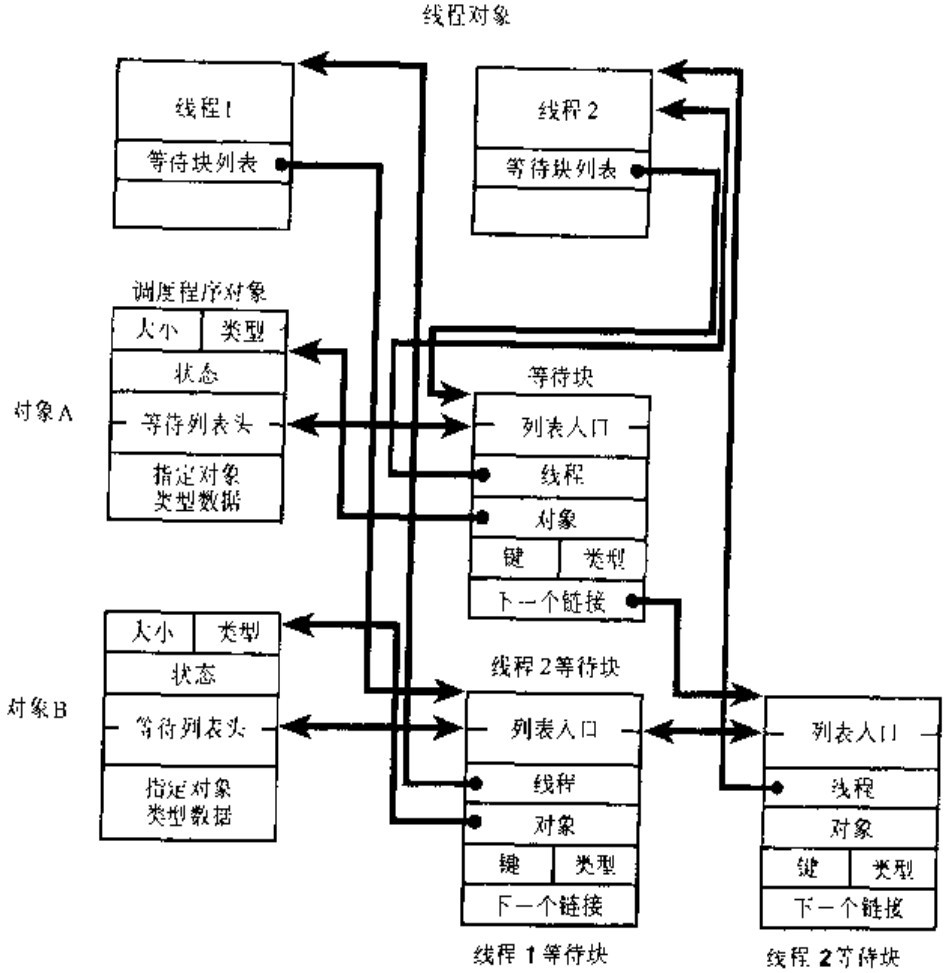


图 3-26　等待数据结构

实验：查看等待队列

尽管有很多进程查看实用程序可以显示线程是否处于等待状态（如果是，并指出等待的类型），但只要利用内核调试程序！thread 命令就能查看线程正在等待的对象列表。例如下面从！thread 命令的输出中摘录的一段显示了在事件对象上等待的线程：

```
kd> !process
:
THREAD 80618030  Cid 97.7f  Teb: 7ffde000  Win32Thread: e199cea8
WAIT: (WrUserRequest) UserMode Non-Alertable
            805b4ab0  SynchronizationEvent
```

虽然内核调试程序没有对调度程序头的内容进行格式化，但由于了解其布局，因此，可以手工地解释其内容：

```
kd> dd 805b4ab0
0x805B4AB0  00040001 00000000 8061809c 8061809c ..........a...a.
```

由此，我们可以确知没有其他线程在该事件对象上等待，因为等待列表头向前指针和向后指针（第三和第四个32位值）指向同一位置（单一等待块）。清除该等待块（在地址0x8061809c）产生如下结果：

```
kd> dd 8061809c
0x8061809C  805b4ab8 805b4ab8 80618030 805b4ab0 .J[..J[.0.a..J[.
0x806180AC  8061809c 00010000 00000000 00000000 ..a.............
```

头两个32位值指向调度程序头的等待块链表头。第三个32位值是指向线程对象的指针。第四个值指向调度程序对象本身。第五个值（0x8061809c）是指向下一等待块的指针。由此，我们可以推断线程不在等待任何其他对象，因为下一等待块字段指向等待块本身。

3.4 系统工作者线程

在系统初始化过程中，Windows 2000在System进程中创建的几个线程称为系统工作者线程（System worker thread）。它们以代表其他线程工作的方式存在。在很多情况下，在DPC/调度级执行的线程需要执行只能在更低IRQL上运行的函数。例如DPC例程，在任意线程环境的DPC/调度级IRQL运行（DPC执行能抢夺系统中的任何线程），它可能需要访问页交换区，或在用来与应用线程同步执行的调度程序对象上等待。由于DPC例程不能降低IRQL，因此它必须将这样的处理传递给在低于DPC/调度级的IRQL上执行的线程。

设备驱动程序和执行程序组件创建它们自身的线程以专门处理无源级别上的工作；然而，大多数使用系统工作者线程，它避免了不必要的调度和在系统中需要额外的线程而引起的内存开销。设备驱动程序或执行程序组件通过调用执行程序函数ExQueueWorkItem或IoQueueWorkItem来请求系统工作者线程的服务。这些函数将工作项（work item）放置在队列调度程序对象中，而线程就在其中查找工作（队列调度程序对象在第9章中做详细介绍）。工作项包括指向例程的指针和在线程处理工作项时由线程传递给例程的参数。例程则由请求无源级别执行的设备驱动程序或执行程序组件实现。

例如，必须在调度程序对象上等待的DPC例程对指向在调度程序对象上等待的驱动程序例程的工作项进行初始化，该工作项也可能指向一个指向对象的指针。在一定阶段，系统工作者线程将会把工作项从其队列中删除，并执行驱动程序的例程。当驱动程序例程结束后，系统工作者线程就检查是否还有其他工作项需要处理。如果没有，系统工作者线程就阻塞直到工作项被放在队列中。当系统工作者线程处理工作项时，DPC例程可能已经结束执行或还没结束（在单处理器系统中，DPC例程总是在其工作项被处理之前结束执行，因为线程调度在IRQL处于DPC/调度级时并不发生）。

有三种类型的系统工作者线程：

■ 延迟工作者线程在优先级为12时执行，处理那些不被认为是时间关键的工作项，并在它们等待工作项时，能将它们的栈调页到调页文件。

■ 关键工作者线程在优先级为13时执行，处理时间关键工作项，在Windows 2000 Server版中，它们的栈始终出现在物理内存中。

■ 单独的超关键工作者线程在优先级为 15 时执行，同时其栈出现在内存中。进程管理器利用超关键工作项执行线程“reaper”函数，该函数释放终止了的线程。

通过执行程序的 ExpWorkerInitialization 函数（在引导过程的早期调用）创建的延迟和关键工作者线程的数目取决于系统中现有内存的大小以及系统是否为服务器。表 3-10 显示了在不同系统配置上创建线程的缺省数量。利用注册表项 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SessionManager \ Executive 下的 AdditionalDelayedWorkerThreads 和 AdditionalCriticalWorkerThreads 值可以指定 ExpInitializeWorker 创建可达 16 个额外的延迟工作者线程和 16 个额外的关键工作者线程。

表 3-10 系统工作者线程的数量

工作者线程	系统内存		
	12 – 19MB	20 – 64MB	> 64MB
延迟	3	3	3
关键	3	Professional：3 Server：6	Professional：5 Server：10
超关键	1	1	1

执行程序在系统执行时，试图将关键工作者线程与变化的工作量相匹配。每隔一秒钟，执行函数 ExpWorkerThreadBalanceManager 就决定它是否应该创建一个新的关键工作者线程。由 ExpWorkerThreadBalanceManage 创建的关键工作者线程被动态工作者线程调用，而在该线程创建时必须满足以下条件：

■ 工作项存在于关键工作队列中。

■ 非活动关键工作者线程的数量（即在等待工作项时被阻塞或在执行工作例程时被阻塞在调度程序上的工作者线程）必须小于系统处理器的数目。

■ 少于 16 个动态工作者线程。

动态工作者线程在处于非活动状态 10 分钟后就退出。因此，当工作量有要求时，执行程序能创建多达 16 个动态工作者线程。

实验：系统工作者线程的列出

可以使用！exqueue 内核调试程序命令来查看按照类型分类的系统工作者线程列表：

```
kd> !exqueue
Dumping ExWorkerQueue: 8046A5C0

**** Critical WorkQueue( current = 0 maximum = 1 )
THREAD 818a2d40  Cid 8.c   Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2ac0  Cid 8.10  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2840  Cid 8.14  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a25c0  Cid 8.18  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a2340  Cid 8.1c  Teb: 00000000  Win32Thread: 00000000 WAIT

**** Delayed WorkQueue( current = 0 maximum = 1 )
THREAD 818a20c0  Cid 8.20  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1020  Cid 8.24  Teb: 00000000  Win32Thread: 00000000 WAIT
THREAD 818a1da0  Cid 8.28  Teb: 00000000  Win32Thread: 00000000 WAIT

**** HyperCritical WorkQueue( current = 0 maximum = 1 )
THREAD 818a1b20  Cid 8.2c  Teb: 00000000  Win32Thread: 00000000 WAIT
```

3.5 Windows 2000 全局标记

Windows 2000 具有存储在系统范围内的全局变量的标记集，称为 NtGlobalFlag，它允许多种内部调试、跟踪以及操作系统中的检校支持。在系统引导时，系统变量 NtGlobalFlag 从注册表键 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SessionManager 中初始化。在缺省情况下，该注册值为 0，因此在系统中，可能没使用任何全局标记。此外，每个映像都有全局标记集，它也能打开内部跟踪和检校代码（尽管这些标记的位格式完全不同于系统范围内的全局标记）。这些标记不被存档或支持供客户使用，但它们却是发掘 Windows 2000 内部操作的有用工具。

值得庆幸的是，Platform SDK 和调试工具包括名为 Gflags.exe 的实用程序，它可以查看和改变系统全局标记（在注册表中或者在运行系统中）以及映像全局标记。Gflags 既有命令行界面又有 GUI 界面。为查看命令行标记，键入 gflags/?。若你不带任何开关（switch）就运行实用程序，则显示的对话框如图 3-27 所示。

可以在注册表中的设置（单击系统注册表）与系统内存中变量的当前值（单击内核模式）之间进行切换。要想实施这些改变必须按 Apply 按钮（如果按 OK 键将退出）。尽管可以改变运行系统上的标记设置，但大多数标记需要重新启动方能生效。而且没有文档记录哪些需要重新启动而哪些不需要。因此若有疑问，改变全局标记再重新启动。

Image File Options 要求填入有效可执行映像的文件名。该选项用来改变应用于单个映像（而不是整个系统）的全局标记集。在图 3-28 中，注意其中的标记与图 3-27 所示的操作系统的标记是不同的。

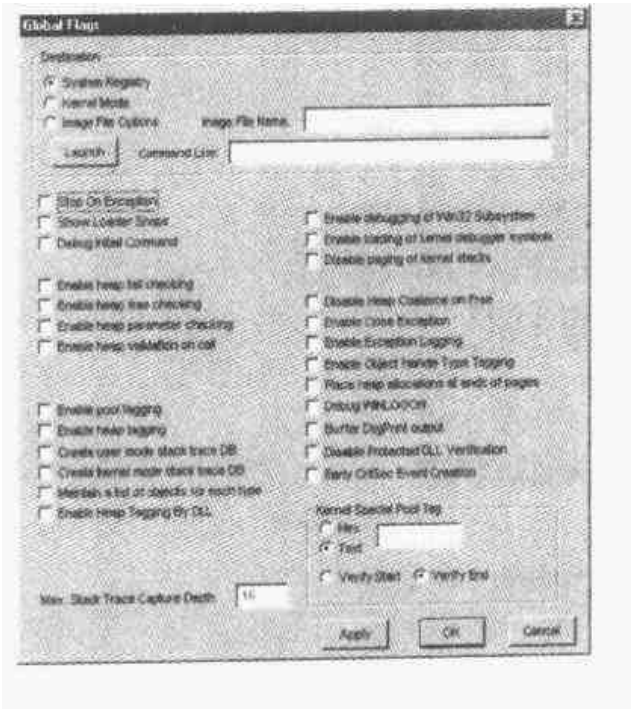


图 3-27 利用 Gflags 设置系统调试选项

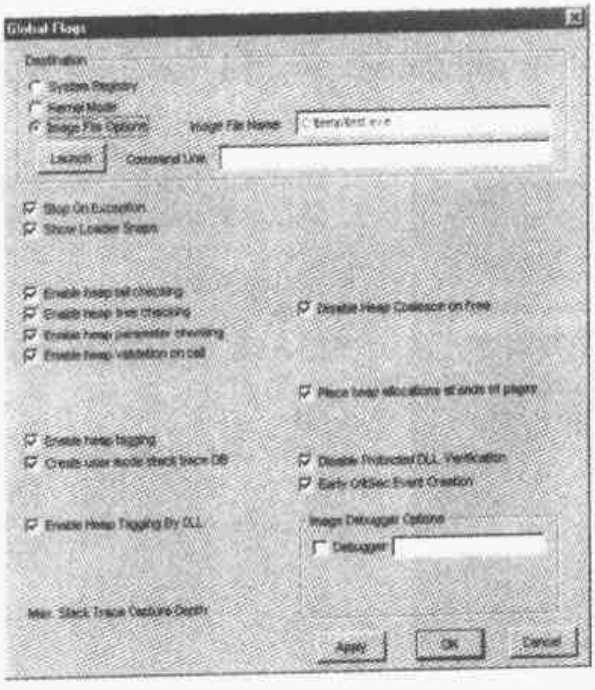


图 3-28 用 Gflags 设置映像全局标记

实验：允许映像加载程序跟踪和查看 NtGlobalFlag

为查看在设置全局标记时所获得的详细跟踪信息的例子，可试着运行系统上随内核调试程序导入的 Gflags。该内核调试程序与运行 Kd、Windbg 或运行 LiveKd 的主机系统相连。

作为例子，试着激活 Show Loader Snaps 标记。为做这一步，选择内核模式，单击 Show Loader Snaps 复选框，并单击 Apply 按钮。然后运行该机器上的映像，在内核调试程序中将看到如下的输出：

```
LDR: PID: 0xb8 started - 'notepad'
LDR: NEW PROCESS
     Image Path: C:\WINNT\system32\notepad.exe (notepad.exe)
     Current Directory: C:\ddk\bin
     Search Path: C:\WINNT\System32;C:\WINNT\system;C:\WINNT
LDR: notepad.exe bound to comdlg32.dll
LDR: ntdll.dll used by comdlg32.dll
LDR: Snapping imports for comdlg32.dll from ntdll.dll
    :
LDR: KERNEL32.dll loaded. - Calling init routine at 77f01000
LDR: RPCRT4.dll loaded. - Calling init routine at 77e1b6d5
LDR: ADVAPI32.dll loaded. - Calling init routine at 77dc1000
LDR: USER32.dll loaded. - Calling init routine at 77e78037
```

可以利用！gflags 和！gflag 内核调试程序命令来查看 NtGlobalFlag 内核变量的状态。！gflags 命令列出所有标记，同时标明哪些被激活；而！gflag 只列出被激活的标记。

```
kd> !gflags

NT!NtGlobalFlag 0x4400

  STOP_ON_EXCEPTION               SHOW_LDR_SNAPS
  DEBUG_INITIAL_COMMAND           STOP_ON_HUNG_GUI
  HEAP_ENABLE_TAIL_CHECK          HEAP_ENABLE_FREE_CHECK
  HEAP_VALIDATE_PARAMETERS        HEAP_VALIDATE_ALL
 *POOL_ENABLE_TAGGING             HEAP_ENABLE_TAGGING
  USER_STACK_TRACE_DB             KERNEL_STACK_TRACE_DB
 *MAINTAIN_OBJECT_TYPELIST        HEAP_ENABLE_TAG_BY_DLL
  ENABLE_CSRDEBUG                 ENABLE_KDEBUG_SYMBOL_LOAD
  DISABLE_PAGE_KERNEL_STACKS      HEAP_DISABLE_COALESCING
  ENABLE_CLOSE_EXCEPTIONS         ENABLE_EXCEPTION_LOGGING
  ENABLE_HANDLE_TYPE_TAGGING      HEAP_PAGE_ALLOCS
  DEBUG_INITIAL_COMMAND_EX        DISABLE_DBGPRINT

kd> !gflag
NtGlobalFlag at 8046a164
Current NtGlobalFlag contents: 0x00004400
    ptg - Enable pool tagging
    otl - Maintain a list of objects for each type
```

3.6 本机过程调用

本机过程调用（LPC）是实现高速消息传递的进程间通信机制。它不能通过 Win32 API 直接可用，而是内部机制，仅可用于 Windows 2000 操作系统组件。下面是些使用 LPC 的例子：

■ 使用远程过程调用（RPC）存档的 API 的 Win32 应用程序在它们指定本机 RPC 时，间接使用 LPC。本机 RPC 是用于相同系统上进程间通信的 RPC 的一种形式。

■ 一些 Win32 API 导致向 Win32 子系统进程发送消息。

■ Winlogon 利用 LPC 与本机安全验证服务器进程（Lsass）通信。

■ 安全引用监测器（在第 8 章解释执行程序组件）利用 LPC 与 Lsass 进程通信。

实验：查看 LPC 端口对象

利用本书配套光盘中的 Object Viewer 实用程序可以查看名为 LPC 的端口对象。运行 \ Sysint \ Winobj. exe 并选择根目录，加号图标表示端口对象，如下图 3-29 所示：

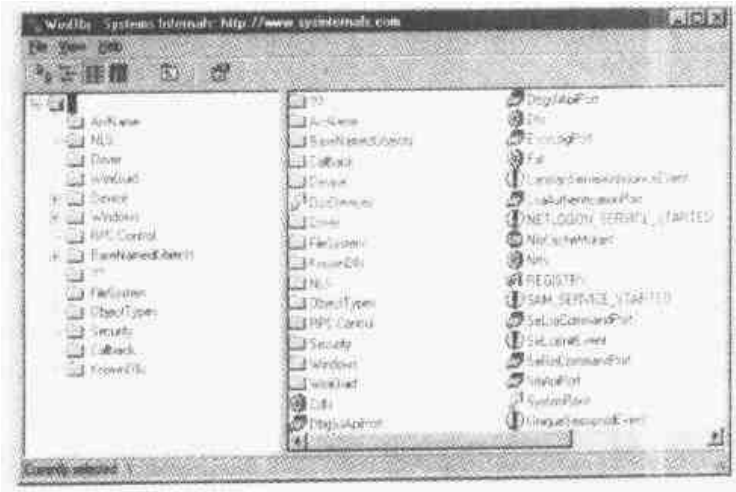


图 3-29 查看名为 LPC 端口对象

为了查看由 RPC 所使用的 LPC 端口对象，选取 \ RPC Control 目录，如下图 3-30 所示：

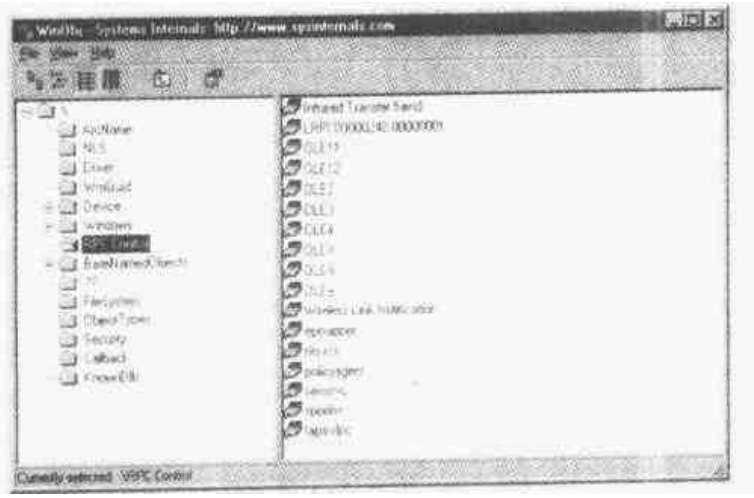


图 3-30 选取 \ RPC Control 目录

也可以利用 ! lpc 内核调试程序命令来查看 LPC 端口对象。该命令接受参数来显示 LPC 端口、LPC 消息以及在等待或发送 LPC 消息的线程。为了查看 Lsass 验证端口（Winlogon 向其发送登录请求的端口），先获取系统的端口列表：

```
kd> !lpc
Usage:
    !lpc                        - Display this help
    !lpc message [MessageId]    Display the message with a
                                given ID and all related
                                information
                                If MessageId is not
                                specified, dump all messages
    !lpc port [PortAddress]     - Display the port information
    !lpc scan PortAddress       - Search this port and any
                                connected port
    !lpc thread [ThreadAddr]    - Search the thread in rundown
                                port queues and display the
                                port info
                                If ThreadAddr is missing,
                                display all threads marked
                                as doing some lpc operations
kd> !lpc port
Scanning 206 objects
       1  Port: 0xe1360320 Connection: 0xe1360320
          Communication: 0x00000000 'SeRmCommandPort'
       1  Port: 0xe136bc20 Connection: 0xe136bc20
          Communication: 0x00000000 'SmApiPort'
       1  Port: 0xe133ba80 Connection: 0xe133ba80
          Communication: 0x00000000 'DbgSsApiPort'
       1  Port: 0xe13606e0 Connection: 0xe13606e0
          Communication: 0x00000000 'DbgUiApiPort'
       :
       1  Port: 0xe205f040 Connection: 0xe205f040
          Communication: 0x00000000 'LsaAuthenticationPort'
       :
```

在输出中找到名为 LsaAuthenticationPort 的端口，然后将地址传递给！lpc 命令，如下所示：

```
kd> !lpc port 0xe205f040
Server connection port e205f040 Name: LsaAuthenticationPort
    Handles: 1   References: 37
    Server process  : ff7d55c0 (lsass.exe)
    Queue semaphore : ff7bfcc8
    Semaphore state 0 (0x0)
    The message queue is empty
    The LpcDataInfoChainHead queue is empty
```

通常，LPC 用于服务器进程和服务器的一个或多个客户进程之间。LPC 连接可在两个用户模式进程之间或一个内核模式组件和一个用户模式进程之间建立。例如，正如第 2 章指出的，Win32 进程通过使用 LPC 而向 Win32 子系统发送间或的消息。另外，一些系统进程利用 LPC 进行通信，如 Winlogon 和 Lsass。内核模式组件利用 LPC 与用户进程会话的一个例子就是安全性能监测器与 Lsass 进程之间的通信。

LPC 的设计允许三种交换消息方式：

■ 对于小于256字节的消息可以用包含消息的缓冲区调用LPC发送。随后该消息从发送进程的地址空间被复制到系统地址空间，最后被复制到接收进程的地址空间。

■ 如果客户机和服务器交换大于256字节的数据，就选择使用它们都能被映射到的共享区域。发送方将消息数据放在共享区域，然后发送短消息给接收方，并将指针指向可以查找到数据的共享区域。

■ 当服务器要读或写大于共享区所能容纳的数据量时，数据能直接从客户地址空间读或写。LPC组件提供了两个函数供服务器实现这一目标。第一个函数所发送的消息被用来同步消息传递。

LPC输出一个称为端口对象的执行程序对象来维持通信所需的状态。尽管LPC使用单一的对象类型，但它有好几种端口类型：

■ 服务器连接端口　该命名端口是服务器连接请求点。客户能通过连接到该端口而连接到服务器上。

■ 服务器通信端口　服务器用来与特殊客户通信的无名字端口。在每个活动客户中，服务器都有这样的端口。

■ 客户通信端口　特定客户线程用来与特定服务器通信的无名端口。

■ 无名通信端口　该无名端口的创建用来供同一进程中的两个线程使用。

LPC通常以如下方式使用：服务器创建一个命名服务器通信端口对象。客户向该端口发出连接请求。如果请求同意，就创建两个新无名端口，客户通信端口和服务器通信端口。客户取得客户通信端口的句柄，而服务器取得服务器通信端口的句柄。客户和服务器随后使用这些新端口进行它们的通信。

客户和服务器之间的完整连接如图3-31所示。

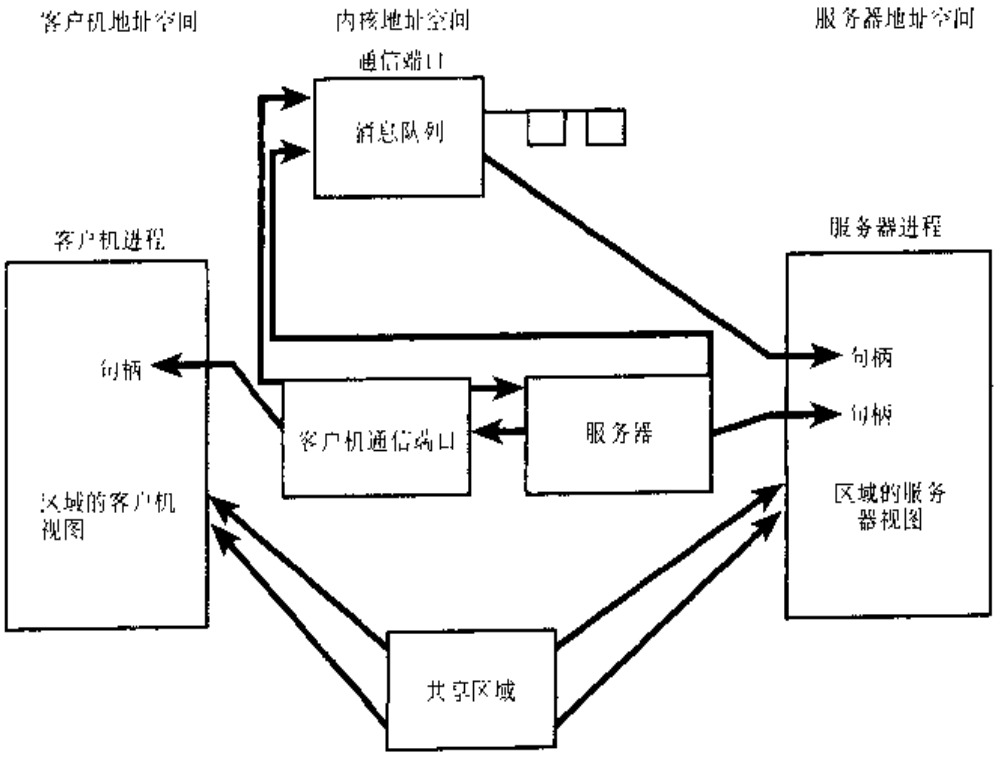


图3-31　LPC端口的使用

3.7 小结

在本章中，我们研究了 Windows 2000 执行程序所依赖的重要基本系统机制。下一章将详细讲述与 Windows 2000 启动有关的步骤，并解释为什么 Windows 2000 有时出现崩溃以及如何处理等问题。

第4章　启动与关闭

本章将描述引导 Microsoft Windows 2000 的步骤和能够影响系统启动的选项。接着将解释正常的系统关闭时发生的情况。最后将讨论可能导致 Windows 2000 崩溃的原因和处理方法。对引导过程细节的理解将有助于诊断所发生的问题。

4.1　引导进程

首先从 Windows 2000 的安装开始，讲述 Windows 2000 的引导过程。接着讲述 Ntldr 和 Ntdetect 的执行过程。设备驱动程序是引导过程中的一个关键部分，因此我们将解释它们在加载和初始化时控制引导过程的方法。然后将描述执行程序子系统的初始化和内核如何通过启动会话管理器程序（Session Manager Process，Smss．exe）、Win32 子系统和登录进程（Winlogon）启动 Windows 2000 的用户模式部分。总之，我们将关注屏幕上出现的各种文本并帮助你将内部进程与引导 Windows 2000 时所见的屏幕显示联系起来。表 4-1 汇总了引导过程的组件和它们的执行模式与责任。

4.1.1　引导前的准备

表 4-1　引导进程组件

组　件	处理器执行	功　能
主引导记录代码（MBR）	16 位实模式	读取和加载分区引导扇区
引导扇区	16 位实模式	读取根目录并加载 Ntldr
Ntldr	16 位实模式和 32 位保护模式；打开调页	读取 Boot．ini，显示引导菜单并加载 Ntoskrnl．exe、Bootvid．dll、Hal．dll 和引导－启动（boot－start）设备驱动程序
Ntoskrnl．exe	带调页的 32 位保护模式	初始化执行程序子系统并引导和系统－启动（system－start）设备驱动程序，为系统运行本机应用程序做准备并运行 Smss．exe
Smss	32 位本机应用程序	加载 Win32 子系统，包括 Win32k．sys 和 Csrss．exe，并启动 Winlogon 进程
Winlogon	32 位本机应用程序	启动服务控制管理器（SCM），本机安全子系统（Lsass），并显示交互式登录对话框
服务控制管理器（SCM）	32 位本机应用程序	加载并初始化自动启动（auto－start）设备驱动程序和 Win32 服务程序

当你打开计算机电源或按下复位键时，Windows 2000 引导进程并没有开始。它开始于你安装 Windows 2000 的时刻。在 Windows 2000 Setup 开始运行时，系统的主要硬盘准备好了在引导进程中运行的代码。在了解这些代码的作用之前，首先了解 Windows 2000 将这些代码存放在磁

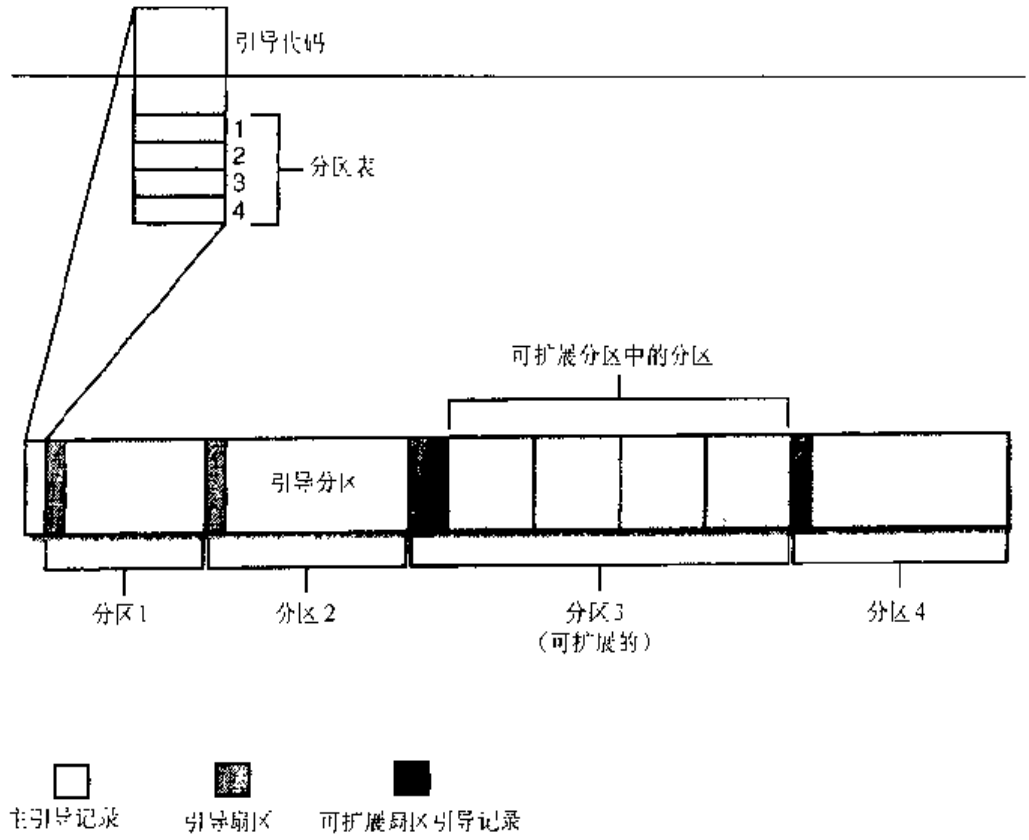


图 4-1 硬盘布局示例

盘何处以及存放的方式。从早期 MS－DOS 时期开始，X86 系统就存在着将物理硬盘划分成卷的标准。Microsoft 操作系统将硬盘划分为称作“分区”（partition）的离散空间，并且使用文件系统（如 FAT 和 NTFS）将每一部分格式化为卷。一个硬盘能够包含最多 4 个基本分区。因为这种分配模式会将磁盘限制为 4 个卷，一种称为可扩展分区（extended partition）的特殊分区类型会将每个基本分区进一步划分为另外最多 4 个分区。就这样，可扩展分区进一步包括可扩展分区，可以使一个操作系统能够在硬盘上划分无限多的卷。图 4-1 显示了一个硬盘布局的例子（在第 10 章中你能够了解更多关于 Windows 2000 分区的内容）。

物理盘以扇区为单位进行编址。典型的 IBM－PC 兼容机的一个硬盘扇区容量是 512 比特。为硬盘定义逻辑驱动器的实用程序，包括 MS－DOS Fdisk 或 Windows 2000 Setup 程序，将一个称为主引导记录的（MBR）的数据扇区写入硬盘的第一个扇区。MBR 包含一个固定的空间，其中存有可执行指令（称为引导代码）和一个定义了磁盘上主分区位置的带有四个项的表（称为分区表）。当 IBM 兼容机引导时，它执行的第一个程序是在 ROM 中编码的 BIOS。BIOS 将 MBR 读入内存并将控制权转移给 MBR 中的代码。

Microsoft 分区工具（如集成进 Windows 2000 Setup 和 Disk Management MMC 插件的分区工具）编写的 MBR，进行了类似的读取和转移控制。首先，MBR 代码扫描主分区表直至找到一个带有可引导标记的分区为止。当 MBR 至少找到一个这样的标记时，它从有这个标记的分区读取第一个扇区的内容到内存并将控制转移给分区内的代码。这种类型的分区称为引导分区，而这种分区的第一个扇区称为引导扇区。

操作系统通常不需要用户干预就将引导扇区写入磁盘。例如，当 Windows 2000 Setup 在硬

盘上写入 MBR 时，它同时在这个磁盘的第一个可引导分区写入引导扇区。你可能曾经使用 MS－DOS sys命令人为地在磁盘上建立引导扇区。Windows 2000 Setup 将检测它要覆盖的引导扇区是否是有效的 MS－DOS 引导扇区。如果是的话，Windows 2000 安装程序将把引导扇区的内容复制到这个分区根目录中的文件 Bootsect．dos 中。

在写入分区的引导扇区之前，Windows 2000 Setup 将确保按照你指定的文件系统来格式化引导分区（和其他任何分区），以确保这个分区已根据某一 Windows 2000 支持的文件系统（FAT、FAT32 或 NTFS）格式化。如果各个分区已经格式化，你可能指示 Setup 忽略这一步骤。Setup 在格式化引导扇区后，将把 Windows 2000 所用的文件拷贝到逻辑磁盘驱动器，包括两个引导文件 Ntldr 和 Ntdetect．com。

Setup 的另外作用是在引导分区的根目录中建立引导菜单文件 Boot．ini。这个文件包含了启动 Windows 2000 的版本选项，如安装程序所安装的或任何以前存在的 Windows 2000 安装。如果 Bootsect．dos 包含一个有效的 MS－DOS 引导扇区，Boot．ini 创建项之一就是引导进入 MS－DOS。以下的输出显示了一个来自双重引导计算机的 Boot．ini 文件例子，这台机器在安装 Windows 2000 之前已经安装了 MS－DOS。

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)
\WINNT="Microsoft Windows 2000 Professional" /fastdetect
C:\="Microsoft Windows"
```

4.1.2 引导扇区和 Ntldr

安装程序在写引导扇区之前必须知道分区格式，因为引导扇区内容根据格式而变化。例如，如果引导分区是一个 FAT 分区，那么 Windows 2000 在引导扇区写入能够理解 FAT 文件系统的代码。但是如果分区是 NTFS 格式，Windows 2000 将写入能够理解 NTFS 文件系统的代码。引导扇区代码的作用是向 Windows 2000 提供逻辑磁盘驱动器的结构和格式信息并且从逻辑磁盘驱动器根目录中读取 Ntldr 文件。这样，引导扇区代码包括足以完成这一任务的只读文件系统代码。在引导扇区代码将 Ntldr 加载到内存后，它把控制权转移给 Ntldr 的入口点。如果引导扇区代码在逻辑磁盘驱动器根目录中无法找到 Ntldr，若引导文件系统是 FAT 格式，它就显示错误信息“BOOT：无法找到 NTLDR”，若引导文件系统是 NTFS 格式，它就显示错误信息“NTLDR 丢失”。

当系统以称为“实模式”（real mode）的 X86 操作模式执行时，Ntldr 开始存在。在实模式中，不会出现内存地址的虚拟－物理转换过程，这意味着把内存地址解释为物理地址和仅有 1MB 计算机物理内存可用访问。简单的 MS－DOS 程序在实模式环境中执行。然而 Ntldr 所采取的第一个行动是将系统切换为保护模式。在引导过程中仍然没有内存地址的虚拟－物理转换过程出现，但可以访问完全的 32 位内存。系统处于保护模式时，Ntldr 能够访问所有的物理内

存。在建立足够的页面表后并设置调页（paging）使 16MB 以下内存可以访问，Ntldr 允许调页。可以调页的保护模式是 Windows 2000 在正常情况下执行的正常操作。

Ntldr 允许调页后，它就完全可以操作。然而，它仍然依赖引导代码提供的功能去访问基于 IDE 的系统和引导磁盘以及显示。引导代码功能简单地关闭调页，并将处理器切换到能够执行 BIOS 提供的服务模式。如果引导盘或系统盘是基于 SCSI 的，Ntldr 加载名为 Ntbootdd. sys 的文件并使用它取代引导代码的磁盘访问功能。下一步 Ntldr 使用内建（built - in）文件系统代码从根目录读取 Boot. ini 文件。像引导扇区代码一样，Ntldr 包括只读 NTFS 和 FAT 代码；和引导扇区代码不同的是，Ntldr 文件系统代码能够读取子目录。

Ntldr 清除屏幕，如果 Boot. ini 中存在不止一种引导选项，它就向用户提供引导选择菜单（如果只有一个条目，Ntldr 忽略菜单并继续显示启动程序条）。Boot. ini 中的选择项指引 Ntldr 到所选择安装 Windows 2000 系统目录（典型的情况下是 \ Winnt）驻留的分区去。这个分区可能与引导分区相同，或者它是另外一个主要分区。

如果 Boot. ini 项指定 MS - DOS 安装（就是说，通过指定 C: \ 作为系统分区），Ntldr 读取 Bootsect. dos 的内容到内存，切换到 16 位实模式，并调用 Bootsect. dos 中的 MBR 代码。这种操作导致 Bootsect. dos 执行就好像 MBR 已经从磁盘中读取了代码。Bootsect. dos 中的代码继续进行 MS - DOS 指定的引导进程，就象在同时安装了 Windows 2000 的计算机上引导 Microsoft Windows98 或 Micorsoft Windows 95那样。

Boot. ini 中的项能够包括 Ntldr 和其他在引导进程中涉及的组件的可选参数。表 4-2 包括这些选项和其作用的完整列表。

表 4-2 Boot. ini 开关

Boot. ini 限定词	意 义
/3GB	将用户进程地址空间从 2GB 增至 3GB（因此系统空间由 2GB 减至 1GB），给虚拟内存密集应用程序（如数据库服务器）更大的地址空间以改善其性能。然而对应用程序来说使用这一特性必须满足两个附加的条件：系统必须运行在 Windows 2000 Advanced Server 或 DataCenter Server 上，并且 application. exe 必须被标记为 3GB—意识的应用程序。（详见第 7 章“地址空间布局”部分）
/BASEVIDEO	使 Windows 2000 使用标准的 VGA 显示驱动器进行 GUI 模式操作
/BAUDRATE =	允许内核模式调试，并为远程内核调试程序主机连接的缺省波特率（19200）指定一个可以替代的值。如：/BAUDRATE = 115200
/BOOTLOG	导致 Windows 2000 在文件 %SystemRoot% \ Ntbtlog. txt 中写入引导日志
/BREAK	导致硬件抽象层（HAL）在 HAL 初始化的断点上停止。Windows 2000 内核在初始化时所做的第一件事就是初始化 HAL，因此这个断点可能是最早的。HAL 在这个断点上无限期等待直到连接上内核调试程序。如果在没带有 /DEBUG 开关的情况下使用这个开关，系统将处于蓝屏状态，此时的停止代码为 0x00000078（PHASE0 - EXCEPTION）
/BURNMEMORY =	指定 Windows 2000 不能使用的内存数量（类似于 /MAXMEM 开关）。这个值以兆字节指定。如：/BURNMEMORY = 128 显示 Windows 2000 不能使用机器上全部物理内存的 128MB

（续）

Boot. ini 限定词	意 义
/CLKLVL	导致标准的X86多处理器HAL（Halmps. dll）将自己设置成一个电平敏感（leve. - sensitive）系统时钟，而不是一个边沿触发（edge - triggered）时钟。电平敏感和边沿触发是用来描述硬件中断类型的名词
/CRASHDEBUG	系统启动时，加载内核调试程序。除非发生系统崩溃否则它并不运行。这将允许内核调试程序使用的串行口可以被系统使用直到系统崩溃（与/DEBUG比较，导致内核调试程序在系统会话期间使用串行端口）
/DEBUG	允许内核模式调试
/DEBUGPORT	允许内核模式调试，并为远程内核调试程序主机连接的缺省串行口（COM1）指定一个可替代的值，如：/DEBUGPORT = COM2
/FASTDETECT	Windows 2000默认的引导选项，替代Windows NT 4开关/NOSERIALMICE。这个限定词的存在使得NTDETECT能够支持引导Windows NT 4。（在仅有NTDETECT的情况下，会缺省地会完成此项操作）。Windows 2000即插即用设备驱动器检测并行和串行设备，但Windows NT 4希望NTDETECT完成这种检测。这样指定/FASTDETECT导致NTDETECT忽略并行和串行设备枚举（在引导Windows 2000时不需要这些操作），与之相反忽略这一开关会导致NTDETECT进行这种枚举（这在Windows NT 4时是必须的）
/INTAFFINITY	指引标准X86多处理器HAL（Halmp. dll）设置中断相似性（affinity），使得仅有最大编号的处理器收到中断。没有这些开关，HAL默认的正常行为是让所有处理器接收到中断
/KERNEL = /HAL =	允许为内核映像文件（Ntoskrnl. exe）和/或HAL（Hal. dll）重载Ntldr的缺省文件名。这些选项在可调试的内核环境（checked kernel environment）和纯运行版本的（零售）内核环境之间更替或手工选择不同的HAL时很有用。如果你想启动仅由可调试的内核和HAL组成的可调试环境（一般来说，这是调试设备程序必须的），请在纯运行版本的系统上遵循如下步骤： 1）从可调试版本（checked bulid）的CD上拷贝内核映像的可调试版本到\Winnt\System32目录，给映像文件赋予缺省名字以外的名字。例如，在单处理器上拷贝Ntoskrnl. exe为Ntoschk. exe并拷贝Ntkrnlpa. exe为Ntoschkpa。内核文件名必须是一个8.3类型的短文件名 2）从可调试版本的CD的\I386\Driver. cab目录拷贝可调试版本的合适HAL到\Winnt\System32目录，并将它命名为Halchk. dll。为了确定需要拷贝哪个HAL，打开\Winnt\Repair\Setup. log并搜索Hal. dll；你会找到象\WINNT\system32\hal. dll = "halacpi. dll", "1d8a1"这样的行。等号右边的名字就是你应拷贝的HAL文件名。HAL文件名必须是一个8.3类型的短文件名 3）拷贝系统的Boot. ini文件中的默认行 4）在引导选择的字符串中添加显示新的选项是为可调试版本的环境准备的描述。（例如"Windows 2000 Professional Checked"） 5）在这一新的选项行后添加如下内容：/KERNEL = NTOSCHK. EXE/HAL = HALCHK. DLL 现在，在启动过程中，当选项菜单出现时，你能够选择新的条目引导可调试环境版本的项或选择你正在引导纯运行版本环境的项

（续）

Boot.ini 限定词	意 义
/MAXMEM =	限制 Windows 2000 使用超过规定的物理内存量，这个数量以兆字节进行说明。如：/MAXMEM = 32 会限制系统仅使用物理内存的第一个 32MB，即使有更多内存
/MAXPROC SPERCLUSTER =	对于标准的 X86 多处理器 HAL(Halmps.dll)，强制簇模式的 Advanced Programmable Interrupt Controller(APIC)寻址(在带 82489DX 外部 APIC 中断控制器的系统上不支持)
/NODEBUG	防止初始化内核模式调试。重载与调试有关的二个开关的规约：/DEBUG、/DEBUGPORT 和/BAUDRATE
/NOGUIBOOT	在启动过程中指示 Windows 2000 不要初始化 VGA 视频驱动程序，它负责显示位图图形。这个驱动程序用来显示引导过程信息，因此关闭它将使 Windows 2000 无法显示这种信息
/NOLOWMEM	要求/PAE 开关存在并且系统具有 4GB 以上的物理内存 如果这些条件满足，允许 PAE 的 Windows 2000 内核版本，Ntkrnlpa.exe 版本，将不再使用第一个 4GB 物理内存。取而代之的是它将加载所有的应用程序和设备驱动程序，并从这一内存范围以外分配所有的内存地址。这一开关具有在大容量内存系统中测试设备驱动程序的兼容性时有用
/NOPAE	即使已检测出系统支持 X86 PAEs 并具有多于 4GB 的物理内存，仍然强制 Ntldr 加载 Windows 2000 内核非物理地址扩展(PAE)版本
/NOSERIALMICE = [COMx|COMx,y,z…]	废弃的 Windows NT 4 限定词——相当于没有/FASTDETECT 开关的情况。禁止指定的 COM 端口上的串行鼠标检测。如果在启动过程中你有一个不是鼠标的设备联在一个串行端口上，这个开关就有用了。在不指出某个 COM 端口的情况下，使用/NOSERIALMICE 参数会禁止所有 COM 端口上的串行鼠标检测工作。详见 Microsoft Knowledge Base 的第 Q131976 条
/NUMPROC =	指定多处理器系统上可以使用的 CPU 数量。例如 在 4 路系统上，/NUMPROC = 2 将限制 Windows 2000 仅使用 4 个处理器中的 2 个
/ONECPU	导致 Windows 2000 在多处理器系统上仅使用一个 CPU
/PAE	导致 Ntldr 加载 Ntkrnlpa.exe，它是能够利用 X86 PAE 的 X86 内核版本。内核的 PAE 版本为设备驱动程序提供了 64 位物理地址，因此这个参数对于测试支持大容量内存的设备驱动程序有帮助
/PCILOCK	停止 Windows 2000 向 PCI 设备动态分配 IO/IRQ 资源，并使这些设备由 BIOS 进行设置。详见 Microsoft Knowledge Base 的第 Q148501 条

（续）

Boot.ini 限定词	意 义
/SAFEBOOT:	指定安全引导选项。当使用 F8 菜单进行安全引导时，Ntldr 已经设定了此项，你不要手工地指定这些选项。（安全的引导指 Windows 2000 仅加载在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SafeBoot 下的 Minimal 或 Network 注册表键下用名或组指定的驱动程序和服务）在这一选项的冒号后必须指定二个参数 MINIMAL、NETWORK 或 DSREPAIR 中的一个。MINIMAL 和 NETWORK 标记分别与没有网络和具备网络的安全引导相对应。DSREPAIR（Directory Services Repair）导致 Windows 2000 引导进入由你指定的备份媒介中的恢复 Active Directory 目录服务的模式。另一个你能添加的选项是（ALTERNATE - SHELL），它让 Windows 2000 使用由 HKLM \ SYSTEM \ CurrentControlSet \ SafeBoot \ AlternateShell 值指定的程序作为图形的 shell 程序，而不是使用默认的 Windows Explorer
/SCSIORDINAL:	向 Windows 2000 指示控制器的 SCSI ID。（给一个带有主板上（on - board）SCSI 控制器的系统增加一个新的 SCSI 设备能够导致控制器的 SCSI ID 改变。）详见 Microsoft Knowledge Base 的第 Q103625 条
/SOS	导致 Windows 2000 列出标记为在引导时加载的设备驱动程序并且显示系统的版本号（包括创建号）、物理内存的数量和处理器的数量
/TIMERES =	设置标准 X86 多处理器 HAL（Halmps.dll）上的系统计时器的精确度。参数是解释为以十亿分之一百秒为单位的数字，但这个等级被定为 HAL 支持的不超过所要求方案的最接近的方案。HAL 支持如下方案： 十亿分之一百秒 毫秒（ms） 9766 0.98 19532 2.00 39063 3.90 78125 7.80 默认的精确度是 7.8ms。系统计时器精确度影响到可等待计时器的精确度。如：/TIMERES = 21000 将计时器设置为 2.0ms
/UFS8254	指引 HAL 把 8254 计时器芯片作为基本计时器（base timer）（采用老式 BOIS 的系统）。详见 Microsoft Knowledge Base 的第 Q169901 条
/WIN95	指引 Ntldr 从保存在 Bootsect.w40 中的 Consumer Windows 引导扇区进行引导。这个参数只有在具备三重引导的系统上：MS - DOS、Consumer Windows 和 Windows 2000 上才是适用的。详见 Microsoft Knowledge Base 的第 Q157992 条
/WIN95DOS	指引 Ntldr 从保存在 Bootsect.dos 中的 MS - DOS 引导扇区进行引导。这个参数只有在具备三重引导的系统：MS - DOS、Consumer Windows 和 Windows 2000 上才是适用的。详见 Microsoft Knowledge Base 的第 Q157992 条
/YEAR =	指引 Windows 2000 核心时间函数忽略计算机的实时时钟报告，并用所指定的值代替它。这样，这个开关所用的年份影响到系统中每个软件，包括 Windows 2000 内核。如：/YEAR = 2001（这个开关被用来测试 Y2K 问题）。

如果用户没有在 Boot.ini 指定的超时范围内从选项菜单中选择一项的话，Ntldr 会选取默认的选项。一但引导选项确定后，Ntldr 加载和执行 Ntdetect.com，这是一个使用系统的 BIOS 进行查询计算机基本设备和设置信息的 16 位实模式程序。这些信息包括如下内容：

- 存储在系统 CMOS(非易失性存储器)中的时间和日期信息。
- 系统的总线类型(如 ISA、PCI、EISA、Micro Channel Architecture[MCA])和连接在总线上的设备标识符。
- 系统上磁盘驱动器的数量、尺寸和类型。
- 与系统连接的鼠标输入设备类型。
- 系统中配置的并行端口的数量和类型。

这些信息收集在内部数据结构中，在引导的后期被保存在 HKLM \ HARDWARE \ DESCRIPTION 注册表键下。

然后 Ntldr 开始清除屏幕并显示“Starting Windows”进度栏。这个进度栏保持空白直至 Ntldr 开始加载引导驱动程序(参见下表所列出的第 5 步)。在进度栏的下面是信息“For troubleshooting and advanced startup options for Windows 2000, press F8”。如果用户按下 F8 键，高级引导菜单出现，这将允许用户选择如下引导：所知最近的正确模式(last known good)、安全模式(safe mode)和调试模式(debug mode)引导等等。

下一步，Ntldr 开始从启动内核初始化所需的引导分区加载文件：

1)加载合适的内核和 HAL 映像文件(缺省为 Ntoskrnl.exe 和 Hal.dll)。如果 Ntldr 未能成功加载那些文件，它会显示信息“Windows 2000 could not start because the following file was missing or corrupt”，紧跟着是未成功加载的文件名。

2)读入 SYSTEM 注册表 hive 文件，\ Winnt \ System32 \ Config \ System，以便确定需要加载哪些设备程序来完成引导(hive 是一种包含注册表子树的文件。在第 5 章注册表中你可以找到更多细节)。

3)扫描内存中的 SYSTEM 注册表 hive 文件并找到所有的引导设备驱动程序。引导设备驱动程序是引导系统所必须的驱动程序。这些驱动程序在注册表中通过 SERVICE - BOOT - START 的启动值指出来。每个设备驱动程序在 HKLM \ SYSTEM CurrentControlSet Services 下有一个注册表子键。如 Services 中有一个关于 Logical Disk Manager 驱动程序的名为 Dmio 的子键，这个可以参见图 4-2(服务程序注册表选项的详细描述参见 5.2 节“服务”)。

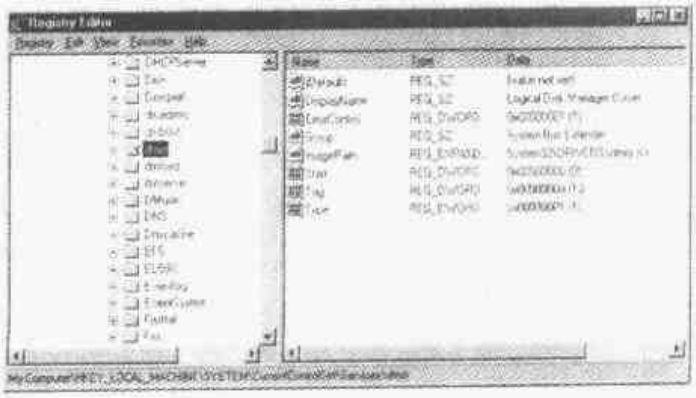


图 4-2 Logical Disk Manager 驱动程序服务设置

4)添加负责为某一分区类型(FAT、FAT32 或 NTFS)实现代码的文件系统驱动程序,安装目录驻留在加载的引导驱动程序的列表上。此时 Ntldr 必须加载这个驱动程序;否则,kernel 将要求驱动程序加载它们,这项要求会导致循环依赖。

5)加载引导驱动程序。为了显示加载过程的进度,Ntldr 在文本"Starting Windows"下显示出一个不断更新的进度栏。进度栏随着每个驱动程序的加载而变化(它假定有 80 个引导设备驱动程序—每加载成功一个文件,进度栏增加1.25%)。如果/SOS 参数在 Boot. ini 选项里已被选定,Ntldr 将不再显示进度栏,而是显示每个引导驱动程序的文件名。记住此时是在加载驱动程序而不是初始化驱动程序——它们在引导过程中的晚些时候进行初始化。

6)为 Ntoskrnl. exe 的执行准备 CPU 寄存器。

这是 Ntldr 在引导进程中所做最后一个操作。这时,Ntldr 调用 Ntoskrnl. exe 中的主函数去执行系统初始化的其他工作。

4.1.3 初始化 Kernel 和执行程序子系统

当 Ntldr 调用 Ntoskrnl 时,它传递包括 Boot. ini 中代表本次引导所选择的菜单选项的一行语句拷贝、指向 Ntldr 产生用于描述系统物理内存的内存表和 HARDWARE 和 SYSTEM 注册表 hive 的内存拷贝以及 Ntldr 加载的引导驱动程序列表的指针的数据结构。

然后 Ntoskrnl 开始其两阶段初始化进程中的第一阶段,这两阶段称为 phase0 和 phase1。大部分执行程序子系统有一个接收描述执行阶段参数的初始化函数。

在 phase0 中,中断被禁止。这个阶段的目的是建立一个基本结构以满足 phase1 阶段所需要的服务程序被调用。Ntoskrnl 的主要函数调用 KiSystemStartup,KiSystemStartup 接下来为每个 CPU 调用 HalInitializeProcessor 和 KiInitializeKernel。如果 KiInitializeKernel 运行在引导 CPU 上,它完成系统范围内的内核初始化,诸如初始化内部 listheads 和其他所有 CPU 共享的数据结构。然后 KiInitializeKernel 的每个实例调用协调 phase0 的函数,ExpInitializeExecutive。

ExpInitializeExecutive 从调用 HAL 函数 HalInitSystem 开始,这让 HAL 在 Windows 2000 进一步完成有效的初始化之前有机会获得系统控制。HalInitSystem 的责任之一是为每个 CPU 中断准备系统中断控制器并配置用于 CPU 时间统计的间隔时钟计时器中断(interval clock timer interrupt)。(更多 CPU 时间统计的情况参见 6.5.8 节)。

在引导处理器上 ExpInitializeExecutive 完成的仅是初始化而不是调用 HalInitSystem。当 HalInitSystem 返回控制权时,引导 CPU 上的 ExpInitializeExecutive 继续处理 Boot. ini 开关/BURNMEMORY(如果这个开关来自 Boot. ini 文件中的某一行,而 Boot. ini 文件与用户在选择哪种安装引导时确定的菜单选项相对应)并放弃这个开关所指定的内存量。

下一步,ExpInitializeExecutive 为内存管理器(memory manager)、对象管理器(object manager)、安全引用监视器(security reference monitor)、进程管理器(process manager)和即插即用管理器(Plug and Play manager)调用 phase0 阶段初始化例程。这些组件完成以下初始化步骤:

1)内存管理器创建页面表和提供基本内存服务所需的内部数据结构。内存管理器还要为系统文件的高速缓存(cache)建立并保留区域,并且为页交换区(paged pool)和非页交换区(nonpaged pool)建立内存区域。其他执行程序子系统、内核和设备驱动程序使用这两个内存交

换区分配它们的数据结构。

2）在对象管理器初始化阶段，建立对象管理器名字空间所必须的对象被定义以便其他子系统能够把对象插入其中。一个句柄表被创建用于跟踪资源。

3）安全引用监视器初始化令牌类型对象，然后使用这个对象创建并准备分配给初始化进程的第一个令牌。

4）进程管理器在 phase0 阶段完成大部分的初始化工作，定义了进程和线程对象类型并设置了跟踪活跃的进程和线程的链表。进程管理器还为初始化进程创建了一个进程对象并命名为空闲（Idle）。作为最后一步，进程管理器创建 System 进程，并建立一个系统线程去执行例程 Phase1Initialization。这个线程并不立即运行，因为中断还处在被禁止状态。

5）接着即插即用管理器的 phase0 阶段初始化开始，它仅简单地初始化一个用来同步总线资源的执行程序资源。

当控制返回到每个处理器的 KiInitializeKernel 函数时，控制继续 Idle 循环，这将导致在以上描述过程的第 4 步创建系统线程并开始执行 phase1 阶段（辅助处理器直到下文描述的 phase1 的第 5 步才开始初始化，这将在下面的列表中描述）。Phase1 阶段由以下步骤组成。（屏幕上进度栏产生的更新变化的步骤如下所示。）

1）调用 HalInitSystem 为系统接受来自设备的中断并允许中断。

2）调用引导视频驱动程序（\ Winnt \ System32 \ Bootvid. dll），它依次显示 Windows 2000 启动屏幕。

3）初始化电源管理器。

4）初始化系统时间（通过调用 HalQueryRealTimeClock）并保存为系统引导时的时间。

5）在多处理器系统上，剩余的处理器初始化并开始执行。

6）进度栏显示被设定为 5%。

7）对象管理器创建了名字空间根目录、\ ObjectTypes 目录、\ ?? 目录和链接到 \ ?? 目录的 \ DosDevices。

8）执行程序被调用以创建执行程序对象类型，包括信号量、互斥、事件和计时器。

9）内核初始化调度程序（dispatcher）数据结构和系统服务调度表。

10）安全引用监视器在对象管理器名字空间中创建 \ Security 目录，并且如果审计（auditing）功能被允许的话，初始化审计数据结构。

11）进度栏显示被设置为 10%。

12）内存管理器被调用以建立区域对象和内存管理器的系统工作者线程（system worker thread）（第 7 章里进行解释）。

13）国家语言支持表（NLS）被映射到系统空间。

14）Ntdll. dll 被映射到系统地址空间。

15）高速缓存管理器初始化文件系统高速缓存数据结构，并创建它的工作者线程。

16）配置管理器在对象管理器名字空间中创建 \ Registry key 对象，并拷贝由 Ntldr 传送的初始注册表数据到 HARDWARE 和 SYSTEM hive 中。

17）初始化统一文件系统驱动程序数据结构。

18）即插即用管理器调用即插即用 BIOS。

19）进度栏显示被设置为 20%。

20）本机过程调用（LPC）子系统初始化 LPC 端口类型对象。

21）如果系统采用引导日志进行引导（/BOOTLOG），引导日志文件被初始化。

22）进度栏显示被设置为 25%。

23）现在 I/O 管理器进行初始化。这个阶段是系统启动的一个复杂阶段，占进度栏中进度的 50%。每成功加载一个驱动程序，I/O 管理器进度栏增加已完成引导进度的 2%（如果有多于 25 个驱动程序需要加载，进度栏会停在 75% 位置）。

I/O 管理器首先初始化各种内部结构，并建立驱动程序和设备对象类型。然后它调用即插即用管理器、电源管理器和 HAL 以开始进行动态设备枚举和初始化的各个阶段（因为这个过程极为复杂并特定于 I/O 系统，第 9 章将讲述详细的细节）。然后 Windows 管理装置（Windows Management Instrumentation（WMI））子系统被初始化，它为遵循 Windows 驱动程序模型（Windows Driver Model，WDM）的设备驱动程序提供 WMI 支持（详见第 5 章 Windows 管理装置（Windows Management Instrument）一节）。下一步，所有引导 - 启动（boot - start）驱动程序被调用以完成驱动程序相关的初始化，系统启动（system - start）设备驱动程序被加载并初始化（第 9 章也描述了驱动程序在注册表中加载控制信息过程的细节）。最后，MS - DOS 设备名在对象管理器的名字空间中被作为符号链接创建。

24）进度栏显示被设置为 75%。

25）如果在安全模式中引导，这个事实将在注册表中记录下来。

26）除非明确地在注册表中禁止，否则将允许内核模式代码的调页（在 Ntoskrnl 和设备驱动程序中）。

27）进度栏显示被设置为 80%。

28）电源管理器被调用以初始化各种电源管理结构。

29）进度栏显示被设置为 85%。

30）安全引用监视器被调用以建立与 Lsass 通信的 Command Server Thread（在第 8 章安全系统组件一节中，可以进一步了解在 Windows 2000 中安全是如何体现的）。

31）进度栏显示被设置为 90%。

32）最后一步是创建会话管理器子系统（Smss）过程（在第 2 章中已介绍）。Smss 负责创建向 Windows 2000 提供可见界面的用户模式环境，其初始化步骤在下一节中描述。

33）进度栏显示被设置为 100%。

作为考虑执行程序（executive）和内核初始化完成前的最后一步，phase1 初始化线程在会话管理器进程的句柄上等待，等待时间为 5 秒。如果会话管理器进程在 5 秒钟时限之前退出，系统崩溃并显示 SESSION5_INITIALIZATION_FAILED 错误检测代码。

如果 5 秒钟等待时间过去了（就是说 5 秒钟已经用完），会话管理器被认为已经成功启动，并且 phase1 初始化函数调用内存管理器的清零页面线程（Zero Page Thread）函数（在第 7 章解释）。这样，系统线程会成为系统剩下时间内的清零页面线程。

4.1.4 Smss、Csrss 和 Winlogon

除了两点不同以外，Smss 和任何其他用户模式进程一样：第一，Windows 2000 认为 Smss 是操作系统的一个可信任的部分。第二，Smss 是一个本机应用程序。因为它是一个可信任的操作系统组件，Smss 能够执行很少有其他进程能够执行的操作，如创建安全令牌。由于它是一个本机应用程序，Smss 不使用 Win32 API 而仅使用内核执行程序（core executive）API，统称为 Windows 2000 本机 API。Smss 不使用 Win32 API，因为当 Smss 启动时 Win32 子系统还没有执行。事实上，Smss 的首要任务之一就是启动 Win32 子系统。

接着 Smss 调用配置管理器执行程序子系统来完成初始化注册表，更新注册表以包括所有键。配置管理器被设计成能够知道内核注册表 hives 保存在磁盘的什么地方（不包括与用户配置文件对应的 hive）并记录它在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ hivelist 中加载的 hive 路径。

Smss 的主要线程完成如下初始化步骤：

1）创建 LPC 端口对象（\ SmApiPort）和两个线程以等待客户请求（如加载新的子系统或创建一个会话）。

2）为 MS - DOS 设备名（如 COM1 和 LPT1）定义符号链接。

3）如果安装了终端服务，在对象管理器名字空间中创建 \ Sessions 目录（对于多重会话）。

4）运行在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ BootExecute 中定义的任何程序，一般来说这个值包括一个运行 Autochk 的命令（Chkdsk 的引导时版本）。

5）按 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ PendingFileRenameOperations 的指示完成延迟文件更名操作。Pending 文件删除在 PendingFileRenameOperations2 中进行。

6）打开已知的 DLL。

7）创建其他页面文件。

8）初始化注册表。配置管理器通过为 HKLM \ SAM、HKLM \ SECURITY 和 HKLM \ SOFTWARE 键加载注册表 hives 来刷新注册表。尽管 HKLM \ SYSTEM \ CurrentControlSet \ Control \ hivelist 在磁盘上查找到 hive 文件，配置管理器却被设定为在 \ Winnt \ System32 \ Config 中查找它们。

9）创建系统环境变量。

10）加载 Win32 子系统（Win23k. sys）的内核模式部分。Smss 通过在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager 中查找它们的路径来确定 Win32k. sys 和它要加载的其他组件位置。Win32k. sys 中的初始化代码根据系统缺省配置（default profile）文件定义的分辨率使用视频驱动程序来开启屏幕显示。因此在这一时刻，屏幕从引导视频驱动程序使用的 VGA 模式改变为选定的系统缺省分辨率

11）启动子系统进程，包括 Csrss（见第 2 章注释，POSIX 和 OS/2 子系统被定义成根据要求启动）。

12）启动登录进程（Winlogon）。Winlogon 的启动步骤稍后简短描述。

13）为调试事件消息创建 LPC 端口（DbgSsApiPort 和 DbgUiApiPort），并创建线程以倾听这

些端口。

执行了这些初始化步骤后，Smss中的主线程就一直在Csrss和Winlogon的句柄上等待。既然Windows 2000依赖于它们的存在，如果这些进程中的任何一个意外地结束，Smss导致将系统崩溃。

然后Winlogon执行其启动步骤，如创建初始的窗口位置和桌面对象，加载GINA DLL等等。然后它创建服务控制管理器（SCM）进程（Winnt \ System32 \ Services. exe），其加载被标记为自动启动（auto - start）的所有服务程序和设备驱动程序和本机安全验证子系统（Lsass）进程（\ Winnt \ System32 \ Lsass. exe）（Winlogon和Lsass的启动顺序的更多细节见第8章）。

在SCM初始化自动启动（auto - start）服务程序和驱动程序并且用户在控制台成功登录后，SCM认为引导成功。注册表中所知最近的正确控制集（last known good control set）（由HKLM \ SYSTEM \ Select \ LastKnownGood指出）被更新以匹配 \ CurrentControlSet。如果用户在引导过程中的第一步选择引导所知最近的正确（last known good）菜单或驱动程序返回一个严重或关键的错误，则系统使用LastKnownGood值作为当前的控制集。这样做增加了系统成功引导的机会，因为至少一个先前使用的所知最近的正确（last known good）配置的引导是成功的

这样就结束了整个引导过程。

4.2 安全模式

Windows 2000系统无法引导的最常见原因是设备驱动程序在引导的过程中导致系统崩溃。因为软件或硬件配置能够随时间改变，潜在的错误能够随时在驱动程序中出现。Windows 2000为管理员提供一种解决这些问题的方法：在安全模式中引导。安全模式是Windows 2000从Consumer Windows中借用的一个概念——由设备驱动程序和服务最小集合组成的引导配置。通过仅依赖引导所必需的驱动程序和服务，Windows 2000避免加载可能崩溃的第三方和其他非关键的驱动程序。

Windows 2000引导时，按下F8键进入包含安全模式引导选项的特殊引导菜单。从3个安全模式变种中选取：安全模式、网络安全模式和命令提示安全模式。标准安全模式由成功引导所需的最少量的设备驱动程序和服务组成。网络安全模式在标准安全模式的设备驱动程序和服务中添加了网络驱动程序和服务。除了在系统允许GUI模式时Windows 2000运行命令提示符应用程序（Cmd. exe）取代Windows Explorer作为shell以外，在其他方面，命令提示安全模式和标准安全模式完全一样。

Windows 2000包括第4种安全模式——目录服务恢复模式（Directory Service Restore Mode）——这一点不同于标准和网络安全模式。使用目录服务恢复模式引导时，系统从备份媒体中恢复域控制器的Active Directory目录服务。所有的驱动程序和服务在目录服务恢复模式引导的过程中加载；因此，不要使用目录服务恢复模式去引导不可引导的系统。

4.2.1 在安全模式中加载的驱动程序

Windows 2000是如何确定哪些设备驱动程序和服务是标准和网络安全模式的一部分呢？答案存在于HKLM \ SYSTEM \ CurrentSet \ Control \ SafeBoot注册表键中。这个键包括Minimal和

Network 子键。每个子键包括更多指定设备驱动程序、服务或驱动程序组名的子键。如 vga.sys 子键标识启动配置中包括的 VGA 显示驱动程序。VGA 显示驱动程序为任何 PC 兼容的显示适配器提供基本的图形服务。系统使用这个驱动程序作为安全模式的显示驱动程序，代替一个可能利用适配器高级硬件特性但也可能阻碍系统引导的驱动程序。SafeBoot 键下的每个子键有一个描述子键标识内容的默认值；vga.sys 子键的默认值是 Driver。

引导文件系统子键有默认值 Driver Group。当开发人员设计设备驱动程序的安装脚本时，他们能够指定设备驱动程序属于一个驱动程序组。系统定义的驱动程序组被列表在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ ServiceGroupOrder 键 List 值下。开发人员指定某个驱动程序为组的成员，向 Windows 2000 说明这个驱动程序开始于引导过程中的哪一点。ServiceGroupOrder 键的主要目的是定义驱动程序组加载的顺序；某些类型的驱动程序必须在其他类型的驱动程序之前或之后加载。驱动程序的配置注册表键下的 Group 值将这个驱动程序与一个组关联起来。驱动程序和服务配置键驻留在 HKLM \ SYSTEM \ CurrentControlSet \ Services 下。如果在该键下查看，你会发现 VGA 显示设备驱动程序的 VgaSave 键在注册表中作为 Video Save 组的成员存在。Windows 2000 要求用来访问 Windows 2000 系统驱动的任何文件系统驱动程序存在于 Boot 文件系统组中。如果系统驱动是 NTFS，NTFS 驱动程序是这个组的成员（Ntfs 键下的 Group 值是 Boot 文件系统）；否则，Fastfat 文件系统驱动程序（在 Windows 2000 中支持 FAT12、FAT16 和 FAT32 驱动器）是这个组的成员。其他文件系统驱动程序为文件系统组的部分，它也被包括在标准和网络安全模式配置中。

当你引导进入安全模式配置时，引导加载程序（Ntldr）把一个作为命令行参数的相关开关（Switch）连同在 Boot.ini 中为正在引导的安装所指定的其他开关传递给内核（Ntoskrnl.exe）。如果你引导进入任何安全模式，Ntldr 传递/SAFEBOOT：开关。根据你所选择的安全模式类型，Ntldr 会添加一个或更多其他的字符串到/SAFEBOOT：中。对于标准安全模式，Ntldr 添加 MINIMAL；对于网络安全模式，Ntldr 添加 NETWORK；对于命令提示安全模式，Ntldr 添加 MINIMAL（ALTERNATESHELL）；目录服务恢复模式中添加 DSREPAIR。

Windows 2000 内核早在引导过程中扫描引导参数以查找安全模式开关，并把内部变量 InitSafeBootMode 设置为一个反映内核所找到的开关的值。内核把 InitSafeBootMode 的值写入注册表的值 HLKM \ SYSTEM \ CurrentControlSet \ Control \ SafeBoot \ Option \ OptionValue 中，以便用户模式组件（如 SCM）能够确定系统处在什么引导模式中。除此以外，如果系统正在进行命令提示安全模式引导，内核将 HLKM \ SYSTEM \ CurrentControlSet \ Control \ SafeBoot \ Option \ UseAlternateShell 的值设为 1。内核将 Ntldr 传递给它的参数记录在值 HLKM \ SYSTEM \ CurrentControlSet \ Control \ SystemStartOptions 中。

当 I/O 管理器内核子系统加载 HLKM \ SYSTEM \ CurrentControlSet \ Services 指定的设备驱动程序时，I/O 管理器执行函数 IopLoadDriver。当即插即用管理器检测到一个新设备并想动态地加载设备驱动程序时，即插即用管理器执行函数 IopCallDriverAddDevice。这两个函数在加载所需的驱动程序前都调用函数 IopSafeBootDriverLoad。IopSafeBootDriverLoad 检查 InitSafeBootMode 的值并确定这个驱动程序是否应该加载。例如，如果系统在标准模式中引导而驱动程序在 Minimal 键下有一个组，IopSafeBootDriverLoad 查找驱动程序的组。如果 IopSafeBootDriverLoad 找到了

所列出的驱动程序的组，IopSafeBootDriverLoad 向其调用者说明这个驱动程序能够加载。否则，IopSafeBootDriverLoad 在 Minimal 键下查找这个驱动程序的名。如果这个驱动程序无法找到驱动程序组或驱动程序名子键，这个驱动程序无法加载。如果系统在网络安全模式中引导，IopSafeBootDriverLoad 在 Network 子键下搜索。如果系统不在安全模式中引导，IopSafeBootDriverLoad 加载所有的驱动程序。

考虑到安全模式在引导过程中排除驱动程序，因而存在一个循环陷阱：Ntldr，而不是内核，加载在注册表键中 Start 值为 0 的任何驱动程序，它指定在引导时加载驱动程序。因为 Ntldr 不检查用来标识哪些驱动程序需要加载的 SafeBoot 注册表键，Ntldr 加载所有引导 - 启动（boot - start）驱动程序。

4.2.2 安全模式意识用户程序

当服务控制管理器（SCM）用户模式组件（Services. exe 实现）在引导进程中初始化时，SCM 检查 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SafeBoot \ Option \ OptionValue 的值用来确定系统完成的是否是安全引导。如果是的话，SCM 镜像 IopSafeBootDriverLoad 的操作。尽管 SCM 处理列于 HKLM \ SYSTEM \ CurrentControlSet \ Services 下的服务，但它仅加载那些用适当的安全模式子键通过名字指定的服务。在第 5 章服务程序部分，你能够找到更多关于 SCM 初始化进程的信息。

Userinit（ \ Winnt \ System32 \ Userinit. exe）是另一种需要知道系统是否在安全模式中引导的用户模式组件。Userinit 在用户登录时初始化用户环境的组件，检查 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SafeBoot \ Option \ UseAlternateValue。如果这个值已被指定，Userinit 运行在值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SafeBoot \ AlternateShell 中作为用户 shell 指定的程序，而不是执行 Explorer. exe。在安装过程中，Windows 2000 将程序名 Cmd. exe 写入 AlternateShell 值，使 Win32 命令提示成为命令提示安全模式默认的 Shell。尽管命令提示已经成为 shell，你仍然能够在命令提示符下键入 Explorer. exe 以启动 Windows Explorer，并且在命令提示符下也能够运行任何其他 GUI 程序。

应用程序是如何确定系统是否在安全模式中引导的呢？可以通过调用 Win32 GetSystemMetrics（SM_CLEANBOOT）函数。系统在安全模式中引导时，需要执行某种操作的批脚本查找 SAFEBOOT_OPTION 环境变量，因为系统仅在安全模式引导中定义这个环境变量。

4.2.3 安全模式中的引导日志

当你指示系统在安全模式中引导时，Ntldr 将由/BOOTLOG 选项指定的字符串作为参数与安全模式要求的参数一起传递给 Windows 2000 内核。当内核初始化时，无论是否存在安全模式的参数，它都检查出现的引导日志参数。如果内核检测到一个引导日志字符串，内核记录它所考虑加载的每个设备驱动程序上的操作。例如，如果 IopSafeBootDriverLoad 告诉 I/O 管理器不要加载一个驱动程序，I/O 管理器调用 IopBootLog 去记录没有加载的驱动程序。同样，在 IopLoadDriver 成功加载作为安全模式配置一部分的驱动程序后，IopLoadDriver 调用 IopBootLog 去记录加

载的驱动程序，你能够检查引导记录去查看哪些设备驱动程序是引导配置的一部分。

因为内核希望在 Chkdsk 运行前避免修改磁盘，所以在引导进程的后期，IopBootLog 无法简单地将消息转储到日志文件。取而代之的是，IopBootLog 记录 HKLM \ SYSTEM \ CurrentControlSet \ Bootlog 注册表值中的信息。作为在引导中加载的第一个用户模式组件，会话管理器（\ Winnt \ System32 \ Smss. exe）执行 Chkdsk 以确保系统驱动器的一致性并随后通过执行 NtInitializeRegistry 系统调用完成注册表初始化。内核以这个操作作为它能够安全打开一个日志文件的线索，这涉及到函数 IopCopyBootLogRegistryToFile。这个函数在 Windows 2000 系统目录（默认为 \ Winnt）中建立文件 Ntbtlog. txt，并拷贝 BootLog 注册表值的内容到这个文件。IopCopyBootLogRegistryToFile 也为 IopBootLog 设置标记使 IopBootLog 直接写入日志文件，而不是在注册表中记录信息，到此为止已基本成功。以下输出显示了引导日志的部分内容：

```
Microsoft (R) Windows 2000 (R) Version 5.0 (Build 2195
 2 11 2000 10:53:27.500
Loaded driver \WINNT\System32\ntoskrnl.exe
Loaded driver \WINNT\System32\hal.dll
Loaded driver \WINNT\System32\BOOTVID.DLL
Loaded driver ACPI.sys
Loaded driver \WINNT\System32\DRIVERS\WMILIB.SYS
Loaded driver pci.sys
Loaded driver isapnp.sys
Loaded driver compbatt.sys
Loaded driver \WINNT\System32\DRIVERS\BATTC.SYS
Loaded driver intelide.sys
Loaded driver \WINNT\System32\DRIVERS\PCIIDEX.SYS
Loaded driver pcmcia.sys
Loaded driver ftdisk.sys
Loaded driver Diskperf.sys
Loaded driver dmload.sys
Loaded driver dmio.sys
:
Did not load driver Media Control Devices
Did not load driver Communications Port
Did not load driver Audio Codecs
:
```

4.3 恢复控制台

在引导过程中，安全模式对由设备驱动程序崩溃造成的系统无法引导的情况来说是一个满意的依靠，但是在某些情况下，安全模式引导无法帮助系统的引导。如果一个妨碍系统引导的驱动程序是 Safe 组的成员，安全模式引导将会失败。安全模式无法帮助系统引导的其他例子是在引导时加载的第三方驱动程序妨碍引导，如病毒扫描驱动程序（无论系统是否处于安全模式，引导启动（boot - start）驱动程序都要加载）。其他在安全模式中引导会失败的情况是系统模块或安全模式配置中的关键设备驱动程序文件损坏或当系统驱动器主引导记录（MBR）被损坏时。通过使用 Windows 2000 恢复控制台你能够避免这些问题。恢复控制台允许你从 Windows

2000 CD 或引导盘中引导进入一个受到限制的命令行 shell 去修复一个安装程序，而不必引导安装。

当你从 Windows 2000 CD 或引导盘中引导系统时，你最终会看到一个屏幕显示，让你选择是安装 Windows 2000 还是修复现有的安装。如果你选择修复现有的安装，系统提示你插入 Windows 2000 CD（如果没有在系统的 CD 驱动器中放入它的话）并在两个修复选项中选择：启动恢复控制台或启动紧急修复进程。如果你在 Setup 欢迎屏幕显示中按下 F10 键，你就绕过了菜单选项而通过快捷键直接进入恢复控制台。

当你启动恢复控制台时，它给出 Windows NT 和 Windows 2000 的安装列表，从中选择在其扫描计算机硬盘时编译。做出选择之后，系统提示你键入管理员帐号口令而作为管理员登录安装。如果你登录成功，系统让你进入一个类似 MS－DOS 环境的命令 shell。这个命令集灵活并使你能够进行简单的文件操作（如拷贝、重命名和删除）、允许和禁止服务和驱动程序、甚至修改 MBR 和引导记录。然而，除了根目录、你所登录的安装的系统目录或可以卸下的设备如 CD 或 3.5 寸软盘外，恢复控制台不让你访问其他目录。这种禁令为管理员通常不能访问的数据提供了一定程度的安全。

恢复控制台使用本机 Windows 2000 系统调用界面去完成文件 I/O 以支持如 Cd、Rename 和 Move 等命令。让你改变设备驱动程序和服务程序启动模式的 Enable 和 Disable 命令以不同的方式工作。如，当你让恢复控制台关闭某一设备驱动程序时，它访问安装程序的 Service 键并操作指定驱动程序键的 Start 值，将这个值改为 SERVICE_DISABLED。下次安装程序引导时，那个设备驱动程序不会加载。（恢复控制台也为你登录的安装加载 SYSTEM hive［\Winnt\System32\Config\System］。这个 hive 包含存贮在 HKLM\SYTEM\CurrentControlSet\Services 注册表键中的信息。

当你从 Windows 2000 CD 或引导盘中引导时，在系统让你选择安装或修复 Windows 2000 时，CD 已经引导了 Windows 2000 内核的拷贝，包括所有必要的支持设备驱动程序（如 NTFS、FAT 驱动程序、SCSI 驱动程序和视频驱动程序）。在 x86 系统中，位于 Windows 2000 CD 的 I386 目录中的 Txtsetup.sif 文件指引从 CD 中引导；这个文件包括标识哪些文件需要加载及其在 CD 中位置的指令。正如你从硬盘引导 Windows 2000 那样，内核执行的第一个用户模式程序是会话管理器（Smss.exe），位于 I386\System32 文件夹中。Windows 2000 Setup 使用的会话管理器不同于标准安装会话管理器。前一个组件向你显示选择进行安装或者修复 Windows2000 的菜单并询问你想进行哪种类型修复。如果你正在安装 Windows 2000，会话管理器就是指导你选择安装分区并拷贝文件到硬盘上的那个组件。

当你运行恢复控制台时，会话管理器加载并启动两个实现恢复控制台的设备驱动程序：Spcmdcon.sys 和 Setupdd.sys。Spcmdcon.sys 显示交互式的命令提示符并执行高层次的命令处理。Setupdd.sys 是一个支持驱动程序，它为 Spcmdcon.sys 提供了一系列函数让 Spcmdcon.sys 管理磁盘分区、加载注册表 hive 和显示与管理视频输出。Setupdd.sys 也与磁盘驱动程序进行通信以管理磁盘分区和使用在 Windows 2000 内核内建的基本视频支持去显示屏幕上的信息。

当你选择了一个安装登录并且恢复控制台接受了你的口令，尽管此时安装程序的 Windows 2000 安全子系统尚未完成安装，恢复控制台必须验证你的登录尝试。这样，恢复控制台必须

独自确定你的口令与系统管理员帐号是否匹配。在这个过程中，恢复控制台采取的第一步骤是使用 Setupdd. sys 从硬盘中加载安装程序的安全帐号管理器（SAM）注册表 hive，它保存有口令信息。SAM hive 位于 \ Winnt \ System32 \ Config \ Sam。加载这个 hive 后，恢复控制台在安装程序的注册表中查找系统键并使用系统键去解密 SAM 的内存拷贝。SAM hive 加密是在 Windows NT 4 Service Pack 3 中引入的新特征，它增加了针对那些试图直接从 hive 文件中读入口令的基于 MS－DOS 口令的偷窃者（snoopers）的保护。

下一步，恢复控制台（Spcmdcon. sys）在 SAM 中查找系统管理员帐户口令，并在最后的验证阶段，恢复控制台使用 RC4 散列算法——Windows 2000 登录进程使用的同样的算法　　散列键入的口令并与 SAM 存贮的散列的口令进行比较。如果恢复控制台认为匹配，系统认为你登录上了，如果恢复控制台认为不匹配，系统将拒绝你访问恢复控制台。

4.4 关机

如果某个人已经登录并且进程通过调用 Win32 ExitWindowsEx 函数初始化了关机操作，信息被送入 Csrss 指示它完成关机。Csrss 接着模拟调用者并传递一个 Windows 信息到 Winlogon 所拥有的隐藏窗口，告诉它去完成关机操作。然后 Winlogon 模拟当前登录的用户（他可能具有也可能不具有与启动系统的那个用户相同的安全背景）并且采用特殊的内部标记去调用 ExitWindowsEx。这个调用再一次导致消息发送到请求系统关机的 Csrss。

这时，Csrss 注意到请求来自 Winlogon 并且在交互式用户（而不是要求关机的那个用户）的登录会话中的所有进程中循环。对于每一个拥有顶层窗口的进程，具有 Windows 消息循环的进程中的 Csrss 发送 WM_ QUERYENDSEEION 消息到每个线程。如果线程返回 TRUE，系统关机继续。然后 Csrss 发送 WM_ ENDSESSION Windows 消息到要求它退出的线程。Csrss 按照 HKCU \ ControlPanel \ Desktop \ HungAppTimeout 中定义的秒钟数等待线程退出（默认值为 5000 毫秒）。

如果线程在超时前没有退出，Csrss 显示挂起程序的对话框，见图 4-3（你能够通过把注册表值 HKCU \ ControlPanel \ Desktop \ AutoEndTasks 设置为 1 的方式来禁止这个对话框）。这个对话框显示程序没有及时地关闭，并给用户一个选择，要么结束进程要么放弃关机操作（这个对话框没有超时概念，这意味着关机请求能够在这一点永远等待）。

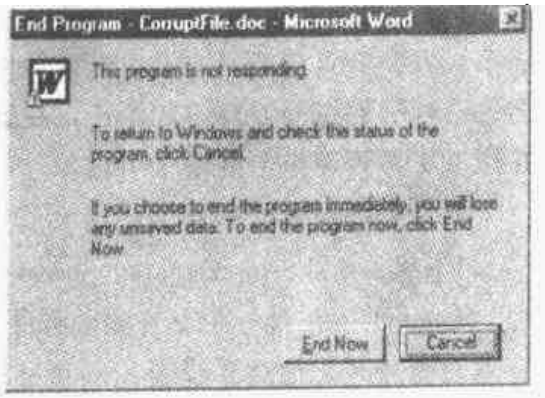


图 4-3　挂起程序对话框

如果线程在超时前退出，Csrss 继续发送 WM_ QUERYENDSESSION/WM_ ENDSESSION 消息对到拥有窗口的这个进程的其他线程。一旦这个进程中所有拥有窗口的线程退出，Csrss 便结束这个进程并继续交互式会话中的下一个进程。

如果 Csrss 找到一个控制台应用程序，它通过发送 CTRL_ LOGOFF_ EVENT 事件调用控制台控制处理程序（只有服务进程在关机时收到 CTRL_ SHUTDOWN_ EVENT 事件）。如果处理程序返回 FALSE，Csrss 取消这个进程。如果处理程序返回 TRUE 或在 KHCU \ ControlPanel \ Desktop \ WaitToKillAppTimeout

定义的秒钟数内不响应的话（默认值为20000毫秒），Csrss显示如图4-3所示的挂起程序对话框。

下一步，Winlogon调用ExitWindowsEx让Csrss结束作为任何交互式用户的会话部分的COM进程。

在这一点，交互式用户会话中的所有进程已经结束。Winlogon再次调用ExitWindowsEx，但是这次在系统进程环境中，它再次发送消息到Csrss，查看所有属于系统环境的进程并完成和发送WM_QUERYENDSESSION/WM_ENDSESSION消息到GUI线程（如以前一样）。它发送CTRL_SHUTDOWN_EVENT而不是CTRL_LOGOFF_EVENT到已经注册了控制处理程序的控制台应用程序。注意到SCM是一个注册了控制处理程序的控制台应用程序。当它收到关机请求时，它依次发送服务关闭控制信息到所有注册了关机通知的服务程序。关于服务关闭的更多细节（如Csrss为SCM使用的关闭超时），参见5.2节"服务"。

尽管Csrss在结束用户进程时执行同样的超时，但它不显示任何对话框并且不取消任何进程（系统进程超时的注册表值可以从默认的用户配置（user profile）文件中获取）。这些超时仅给系统进程一次在系统关闭前清除和退出的机会。因此，实际上许多系统进程在系统关闭时仍在运行，如Smss、Winlogon、SCM和Lsass。

一旦Csrss完成通知系统进程系统正在关机任务，Winlogon就通过调用执行程序子系统函数NtshutdownSystem完成关机过程。这个函数调用函数NtSetSystemPowerState去指挥驱动程序和剩下的执行程序子系统（即插即用管理器、电源管理器、执行程序、I/O管理器、配置管理器和内存管理器）的关闭。

例如，NtSetSystemPowerState调用I/O管理器发送关闭I/O包到所有已请求关机通知的设备驱动程序。这个操作给设备驱动程序一个机会去执行任何它们的设备在Windows 2000退出前可能需要的特殊处理。配置管理器刷新任何修改过的数据到磁盘，并且内存管理器把包含有文件数据的所有修改过的页面写回它们各自的文件。如果在关闭时，清除调页文件的选项是允许的，内存管理器此时要清除调页文件。I/O管理器第二次被调用去通知文件系统驱动程序系统正在关闭。系统关机结束了电源管理器。电源管理器采取的行为依赖于用户是否指定了关机、重新引导或关闭电源。

4.5 系统崩溃

几乎每个Windows NT和Windows 2000用户都听说过"蓝屏死机（blue screen of death）"，因为这个不祥的名词指的是当Windows 2000崩溃时所显示的蓝色的屏幕，或由于灾难性的错误和妨碍系统继续运行的内部条件造成的停止执行。

在本节中，我们讲述导致Windows 2000崩溃的基本问题，描述蓝色的屏幕上显示的信息，并且解释可用于创建崩溃转储（crash dump）的各种配置选项，崩溃转储是系统崩溃时的系统记录，它能够帮助你估计哪个组件导致崩溃。本节不打算提供详细的关于分析、查找和解决Windows 2000系统崩溃的问题的信息。

4.5.1 Windows 2000 为什么会崩溃

以下原因会导致 Windows 2000 崩溃（停止执行并显示蓝屏）：

■ 驱动程序或运行在内核模式的操作系统函数产生了不能处理的异常，如内存访问违例（尝试写入只读页面或尝试读当前没有映射的地址）。

■ 对内核支持例程的调用导致再调度，如在无信号的调度程序对象上等待，当中断请求级（IRQL）是 DPC/调度或更高级时。（关于 IRQL 的细节参见第 3 章）

■ 由调页文件中的数据或内存映射文件产生的内存中的页面错误发生在 DPC/调度级或更高的 IRQL 上（它将请求内存管理器等待 I/O 操作发生——如刚才所说的，由于会要求一个再调度，等待不能发生在 DPC/调度或更高级）

■ 驱动程序或操作系统函数明确地使系统崩溃（通过调用系统函数 KeBugCheckEx）因为它检测到一个内部条件，这个内部条件显示破坏或系统不能在不破坏数据的情况下继续执行的其他情况。

■ 硬件错误，如机器检查或不可屏蔽的中断（NMI）的发生。

当内核模式设备驱动程序或子系统导致了非法的异常时，Windows 2000 面临着进退两难的困境。它检测到有能力访问任何硬件设备和任何合法的内存的操作系统的组件执行了它没有料到的操作。

但是为什么那意味着 Windows 2000 不得不崩溃呢？难道它不能忽略这个异常，并让设备驱动程序或子系统继续，就如同什么事也没发生一样？错误被隔离并且组件以某种方式恢复的可能性是存在的。但是最可能的是检测到的异常来自于更深刻的问题——如来自一般的内存损坏或不能正常工作的硬盘设备。允许系统继续执行或许会产生更多的异常和存贮在磁盘的数据或其他外围设备的损坏从而产生更大的风险。

4.5.2 蓝色屏幕

不考虑系统崩溃的原因，实际执行崩溃的函数是 KeBugCheckEx（存档在 Windows 2000 DDK 中）。这个函数接收停止代码（stop code）（有时叫做查错代码），和 4 个在停止代码基础上解释的参数。在 KeBugCheckEx 屏蔽了系统所有处理器上的所有的中断后，它把显示转入蓝屏模式（80 列 × 50 行文本模式），显示一个蓝色背景然后显示停止代码，跟着的是一些说明用户能做什么的文本（系统数据结构已经严重破坏到无法显示蓝屏的情况也是可能的）。图 4-4 显示一个蓝屏的例子。

第一行列出了停止代码和 4 个传给 KeBugCheckEx 的附加的参数。停止代码下的文本行提供了与停止代码数字标识符相同意义的文本。如图 4-4，停止代码 *0x0000000A* 是一个 *IRQL_NOT_LESS_OR_EQUAL* 崩溃。当一个参数包括操作系统或设备驱动程序代码的地址（如图 4-4）时，

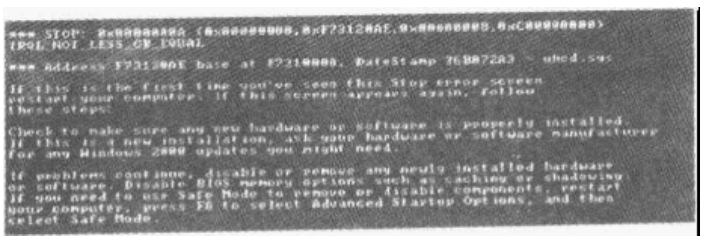


图 4-4 蓝屏示例

Windows 2000 显示这个模块所在的基本地址，日期邮戳和设备驱动程序的文件名。这条信息可能会帮助你查找出错的组件。

尽管有一百条以上的不同停止代码，如果可能的话，大部分也很少能在产品系统中见到。实际上，少数常见的停止代码就代表了 Windows 2000 系统中的大多数的崩溃事例。4 个附加的参数的意义依赖于停止代码（并非所有的停止代码都有可扩展的参数信息）。然而，查找停止代码和参数的意义（如果适用的话）可能至少会帮助你诊断出错的组件（或引起崩溃的硬件设备）。你能够在以下地方找到停止代码信息。

- 测试帮助文件中的（Ddkdbg.chm）“查错（蓝屏）”小节，它放在了 3 个地方：Windows 2000 调试工具（用户支持诊断）、Platform SDK 和 Windows 2000 DDK。
- Windows 2000 服务器操作指南中的“查找和解决问题”一章的“Windows 2000 停止消息”小节（Windows 2000 Server Resource Kit 中的一部分）。该节包括详细的描述，如常见的停止代码参数的含义。
- 你也能查找 Microsoft Online Knowledge 库（suport.microsoft.com）中的停止代码和可能有问题的硬件或程序名。你可能找到关于工作环境的信息，一个及时更新的能够修正你的问题的服务包。Knowledge 库第 Q103059 条列出了大部分停止代码并提供了详细的参数含义。（适用于 Windows NT，但对于 Windows 2000 也成立。）
- Windows 2000 DDK 中的 Bugcodes.h 文件包含了 150 个左右的停止代码及其原因的一个完整的列表。

经常在安装了新的软件或新的硬件后看到蓝屏（blue screen）。如果你只是添加了一个驱动程序，重新引导后在系统初始化时出现蓝屏，你可以重新启动机器，按下 F8 键，然后选择所知最近的正确配置（Last Known Good Configuration）。允许所知最近的正确配置使 Windows 2000 可以从上次成功的引导（在你安装这个驱动程序前）恢复到注册表的设备驱动程序注册键（HKLM \ SYSTEM \ CurrentControlSet \ Services）的拷贝。从所知最近的正确配置的角度来看，成功的引导中所有服务和驱动程序已经完成了加载并且至少有一个成功的登录。

如果还显示蓝屏的话，一个明显的方法是卸载你刚刚在蓝屏出现之前添加的组件。如果距你添加新程序已有一段时间或你同时添加了多个程序，你需要记下在参数中引用的驱动程序名字。如果发现了任何与你添加的东西有关的名字，或许你就找到了出错的对象。

许多设备驱动程序有隐藏的名字，一个用来判断应用程序或硬件设备与某个名字相关的方法是通过在 HKLM \ SYSTEM \ CurrentControlSet \ Services 键下查找设备驱动程序名字，在注册表中查找与设备驱动程序有关的服务名字。注册表的这个分枝是 Windows 2000 为系统中每个设备驱动程序存贮注册信息的地方。如果你找到一个匹配项，查找名为 DisplayName 和 Description 的值。一些驱动程序填入这些值以描述设备驱动程序的目的。例如你可能在 DisplayName 值中找到字符串“Virus Scanner”，这显示你正在运行反病毒软件。驱动程序的列表能够显示在计算机管理（Computer Management）工具中（从 Start 菜单选择 Programs/Administrative Tools/Computer Management）。在 Computer Management 中，打开 System Tools、System Information 和 Software Environment，然后选择 Drivers。

然而，更多的情况是停止代码和 4 个相关参数的信息不足以提供充足的信息用于查找和解

决系统崩溃问题。例如你可能需要检查内核模式调用堆栈以找出触发崩溃的驱动程序或系统组件。同时，因为 Windows 2000 默认的操作是在系统崩溃后自动重新引导，你不太可能有时间记下显示在蓝屏上的信息。这就是 Windows 2000 为什么在默认状态下试图把系统崩溃的信息记录到磁盘供今后分析的原因，这也同时涉及到了本章的最后一个专题，崩溃转储文件。

4.5.3　崩溃转储文件

默认情况下，所有 Windows 2000 系统都被配置成在系统崩溃时尽可能去记录系统状态的信息。你能够通过在控制面板中打开 System 工具看到这些设置，然后在 System Properties 对话框中，点击 Advanced 按钮，然后再点击 Startup And Recovery 按钮。Windows 200 Professional 系统的默认设置见图 4-5。

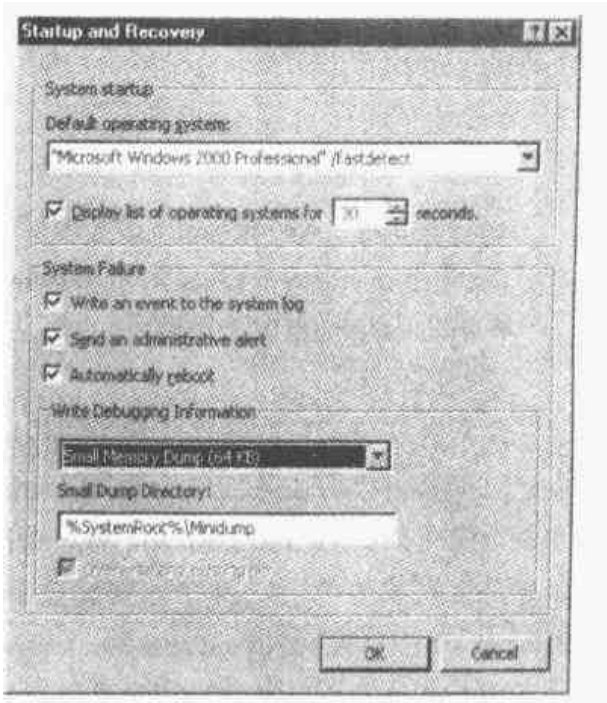


图 4-5　崩溃转储设置

系统崩溃时，有三个层次的信息可以记录下来：

■ **完整内存转储**　完整内存转储包括了崩溃时所有的物理内存记录。这种类型的转储要求页面文件应至少是物理内存的大小。因为在大容量内存系统中要求一个极大的页面文件，这种类型的转储是最不常用的设置。Windows NT 4 仅支持这种类型的转储文件。

■ **内核内存转储**　内核内存转储（Windows 2000 Server 系统中的默认值）仅包括在崩溃时出现在物理内存中的内核模式读/写页面。这种转储不包括属于用户进程的页面。因为只有内核模式程序能够直接导致 Windows 2000 崩溃，然而并不是说用户进程页面对于调试崩溃是不必要的。没有办法去预测内核内存转储的大小，因为它的大小取决于操作系统分配的内核模式内存总量和机器中的驱动程序。作为例子，在 128MB 的便携机中运行 Windows 2000 的测试系统，内核内存占用了 35MB。

■ **小内存转储**　一个 64KB 大小的小内存转储（Windows 2000 专业版中的默认值）包括停止代码和参数、加载的设备驱动程序的清单、描述当前进程和线程（EPROCESS 和 ETHREAD ——第 6 章描述）的数据结构和导致崩溃的线程的内核堆栈。

当 Windows 2000 配置成可以记录崩溃转储信息时，它在调页文件中写入信息，因为试图在磁盘上建立一个新文件更多地要取决于系统数据结构完整无缺。（如果有不止一个调页文件，使用第一个或主页面文件）系统重新引导后，登录进程（Winlogon. exe）创建一个子进程（Savedump. exe）从页面文件中拷出崩溃转储信息并拷入新文件。小内存转储缺省地在目录 \ Winnt \ Minidump 中创建并赋予独一无二的文件名，文件名由字符串“Mini”和日期再加上序列号组成（如，Mini031000 - 01. dmp）。内核内存和完整内存被拷贝到名为 Winnt \ Memory. dmp 的文件中，这意味着只有最新的转储文件保存在磁盘上。

早先曾经提过，由于用于访问调页文件的数据结构本身可能损坏并导致系统无法在磁盘上写入任何东西，所以无法保证崩溃转储信息一定会记录下来。如果系统不能记录崩溃转储信息，你可以试着用内核调试器引导正在崩溃的系统，以使在系统崩溃时能够从主机调试器中取得控制权。那样的话，你能够使用交互式内核调试器去查看内核堆栈以确定崩溃的原因。关于如何建立内核调试器，参见早先提到的 Windows 2000 调试帮助文件（Ddkdbg.chm）。

一但你拥有了崩溃转储文件（不管它是小内存转储、内核内存转储或是全部内存转储），你怎样才能获取停止代码或完成进一步的分析？最简单的工具是 Dumpchk（在 Windows 2000 支持工具、Platform SDK、Windows 2000 DDK 和调试工具中可以使用）。缺省情况下，Dumpchk 打开转储文件并显示关于崩溃的基本信息，诸如操作系统版本、停止代码和参数。如果你用“-e”参数调用，它显示更多的细节，如加载的设备驱动程序、当前的进程和线程以及内核堆栈列表。（这个选项要求 Ntoskrnl.exe 的符号文件匹配崩溃的 Windows 2000 版本。符号文件详见第一章。）

最后，称为内核内存空间分析器（Kernel Memory Space Analyzer，Kanalyze.exe）的高级工具在调试崩溃转储中可能有用。这个工具是调试工具包、Windows 2000 DDK 和 Platform SDK 的一部分并且编辑在单独叫做 OEM Tool Help 的 Microsoft Word 文件中（如果已安装了 Windows 2000 调试工具，你能够在 \ ProgramFiles \ Debuggers \ bin \ kanalyze \ usedocs.doc 中找到这个文件。你也能在 Platform SDK 和 DDK 目录树中找到它）。

不幸的是，你不能运行一个神奇的程序来标识蓝屏的准确原因或消除它们。即便充分了解 Windows 2000 内部和设备驱动程序，分析蓝屏或崩溃转储都是非常困难的。然而，能够获得停止代码和参数至少能够为你指出正确的方向。

实验：强产生制崩溃并获取停止代码

为了产生一个崩溃转储并试验这部分讲述的转储分析工具，你可以运行本书配套的 CD 中的 \ sysint \ Bsod.exe 工具或通过打开 Windows 2000 增加的用来强制产生系统崩溃的支持工具（通过按下右 Ctrl 键同时按下 Scroll Lock 两次）来强制产生 Windows 2000 崩溃。为了允许这个特性，增加一个名为 *CrashOnCtrlScroll* 的 DWORD 值 1 到注册表键 HKLM \ SYSTEM \ CurrentControlSet \ Services \ i8042prt \ Parameters。为了这些修改生效，你必须重新引导系统。

一旦你创建了崩溃转储文件，试着用 Dumpchk 显示基本崩溃转储信息，然后试试 Dumpchk -e 选项去显示可扩展的消息。最后，试着用交互式内核调试器（Kd，I386kd.exe 或 Windbg.exe）打开崩溃转储文件，并使用内建的内置内核调试器扩展中的一些命令，如！process、！thread 和！drivers。

关于如何使用这些工具，请参见 Debugging 帮助文件（Ddkdbg.chm）。

4.6 小结

在本章里，我们已经观察了启动和关闭 Windows 2000 的详细的步骤（正常状态下或有错误的情况）。在下一章，我们将了解 Windows 2000 的底层基础设施的三个机制：注册表、服务和 Windows 管理装置（WMI）。

第5章 管 理 机 制

本章介绍 Microsoft Windows 2000 的三个基本机制，它们对系统管理和配置来说是极为关键的：

- 注册表（registry）。
- 服务（service）。
- 窗口管理装置（windows management instrumentation）。

5.1 注册表

注册表在 Windows 2000 系统配置与控制中发挥重要的作用，它是系统范围和每个用户设置的存储库。尽管许多人认为注册表是作为静态数据存储在硬盘上的，但在本节中你将明白，注册表是了解 Windows 2000 执行程序与内核维护的各种内存中结构的窗口。本节不是 Windows 2000 注册表内容的完整参考，更详细的信息在 Windows 2000 资源工具包的 *Technical Reference to the Windows 2000 Registry* 帮助文件（Regentry.chm）中可以找到。

我们将向你展现注册表结构的总体结构，它支持的数据类型，以及简要介绍 Windows 2000 保存在注册表中的重要信息。然后深入探讨配置管理器（*configuration manager*）的内部结构，它是实现注册表数据库的执行程序组件。本章要讲述的主题是注册表在磁盘上的内部结构，当应用程序发送请求时，Windows 2000 如何检索配置信息以及 Windows 2000 采取什么措施来保护这一关键系统数据库。

5.1.1 注册表数据类型

注册表是一个数据库，其结构与逻辑磁盘驱动程序（logical disk drive）结构相似。注册表中包含键（*key*）与键值（*value*）。键与磁盘的目录相似，键值与磁盘上文件相当。一个键中包含其他键（子键 *subkey*）或键值。另一方面，键值存储数据。顶级键是根键（*root key*）。在本节中使用的子键与键这两个词是一个意思（只有根键不是子键）。

键与键值借用了文件系统的命名约定。这样，你可以通过名字 mark 来标识键值，其名字标记存储在称为 trade 的键中，名字为 trade \ mark。这一命名模式的例外是每个键的未命名键值。两个注册表编辑器工具：Regedit 与 Regedt32，以不同方式显示键值。Regedit 显示未命名的键值为 < Default > ；Regedt32 使用 < no name > 。

键值存储的数据类型如表 5-1 所示，有 11 种类型。常用的注册表键值是 REG_DWORD，REG_BINARY 或 REG_SZ。REG_DWORD 类型的键值可以存储数字或布尔值（Boolean）（on 或 off 键值）；REG_BINARY 键值可以存储大于 32 位的数字或原始数据，如加密口令；REG_SZ 键值存储字符串（当然是 Unicode），可以表示名字、文件名、路径和类型等元素。

表 5-1 注册表键值类型

键值类型	描 述
REG_NONE	无键值类型
REG_SZ	以 Null 结束的固定长 Unicode 字符串
REG_EXPAND_SZ	以 Null 结束的可变长 Unicode 字符串，它可以有嵌入环境变量
REG_BINARY	任意长二进制数据
REG_DWORD	32 位数
REG_DWORD_LITTLE_ENDIAN	32 位数的第一个低位字节，它相当于 REG_DWORD
REG_DWORD_BIG_ENDIAN	32 位数的第一个高位字节
REG_LINK	Unicode 符号链接
REG_MULTI_SZ	以 NULL 结束的 Unicode 字符串数组
REG_RESOURCE_LIST	硬件资源描述
REG_FULL_RESOURCE_DESCRIPTOR	硬件资源描述
REG_RESOURCE_REQUIREMENTS_LIST	资源需求

REG-LINK 类型特别有趣，因为它允许一个键值透明地指向另一键或键值。如果你通过链接（link）遍历注册表，路径搜索在链接目标上还会继续。例如：如果 \Root1\Link 有一个 \Root2\RegKey 的 REG-LINK 键值，并且 RegKey 包含 RegValue 键值，那么有两个路径标识 RegValue：\Root1\Link\RegValue 和 \Root2\Regkey\RegValue。在下节将讲述，Windows 2000 大量地使用注册表链接：6 个注册表根键中的 3 个链接到 3 个非链接根键的子键。链接不保存，它们必须在每次重启动后动态创建。

5.1.2 注册表逻辑结构

你可以通过它存储的数据来描绘注册表的组织。有 6 个根键（你不能添加新的根键或删除存在的根键），它们存储的信息如下：

- HKEY_CURRENT_USER 存储与当前登录用户相关的数据。
- HKEY_USER 存储该机上所有帐号的信息。
- HKEY_CLASSES_ROOT 存储文件关联和 COM（Component Object Model）对象注册信息
- HKEY_LOCAL_MACHINE 存储与系统相关的信息。
- HKEY_PERFORMANCE_DATA 存储性能信息。
- HKEY_CURRENT_CONFIG 存储与当前硬件配置相关的信息。

为何根键名字以 H 开头？因为根键名代表键（KEY）的 Win32 句柄。第一章提到过，HKLM 是 HKEY_LOCAL_MACHINE 的缩写。表 5-2 列出所有根键和它们的缩写。下面将详细介绍 6 个根键的内容和目的。同样，参考 Windows 2000 资源工具包中 Technical Reference to the Windows 2000 Registry 帮助文件（Regentry.chm），可以获得这些键内容的详细信息。

表 5-2 注册表根键

根 键	缩 写	描 述	链 接
HKEY_CURRENT_USER	HKCU	指向当前登录用户的用户配置文件（profile）	与当前登录用户对应的 HKEY_USERS 下的子键
HKEY_USERS	HKU	包括所有加载的用户配置文件的子键	不是链接
HKEY_CLASSES_ROOT	HKCR	包括文件关联和 COM 注册表信息	HKLM \ SOFTWARE \ Classes
HKEY_LOCAL-MACHINE	HKLM	位置包容器（Placeholder）——包含其他键	不是链接
HKEY_CURRENT_CONFIG	HKCC	当前硬件配置文件	HKLM \ SYSTEM \ CurrentControlSet \ Hardware Profiles \ Current
HKEY_PERFORMANCE_DATA	HKPD	性能计数器	不是链接

1. HKEY_CURRENT_USER

HKCU 根键包含本机登录用户的首选项和软件配置的数据。它指向当前登录用户的用户配置文件，位于硬盘 \ Documents and Settings \ < username > \ Ntuser.dat（参考本章 5.1.3 节“注册表内部”--以了解根键如何被映射为硬盘上的文件）。当用户配置文件装入后（如在登录时或当服务进程在特定用户名环境运行时），HKCU 作为指向 HKER_USER 下用户键的链接被创建。表 5-3 列出了 HKCU 下的一些子键。

表 5-3 HKEY_CURRENT_USER 子键

子 键	描 述
AppEvents	声音/事件关联
Console	命令窗口设置（如宽度、高度和颜色）
Control Panel	屏幕保护程序、桌面图案、键盘、鼠标设置以及访问（accessibility）和地区设置
Environment	环境变量定义
Keyboard Layout	键盘布局设置（如 U.S 或 U.K）
Network	网络驱动器映射和设置
Printers	打印机连接设置
Software	用户相关的软件首选项（preference）
UNICODE Program Groups	用户指定起始菜单组定义
Windows 3.1 Migration Status	由 Windows 3.x 升级到 Windows 2000 的文件状态数据

2. HKEY_USER

HKU 包含了系统上加载的各个用户配置文件和用户类注册数据库的子键。它还包含名为 HKU \ .DEFAULT 的子键，它链接到默认工作站配置文件（它用于在本机系统帐号下运行的进程，在本章后面的“服务”部分将详细介绍）。

3. HKER _ CLASSES _ ROOT

HKCR 包含两类信息：文件扩展关联和 COM 类注册。键为每一注册的文件名的扩展存在。大多数键包括 REG _ SZ 键值，它指向 HRCR 下的另一键，该键包含扩展名代表的文件的类关联信息。例如，HKCR \ .xls 将指向在键 HKCU \ Excel.sheet.8 下的 Microsort Excel 文件的信息。另一些键包含注册在系统上的 COM 对象的配置细节。

HKEY _ CLASSES _ ROOT 的数据来自两个地方：

■ HKCU \ SOFTWARE \ Classes 中的每个用户类注册数据（映射到硬盘文件 \ Documents and Settings \ < username > \ Local Setting \ Application Data \ Microsoft \ Windows \ Usrclass.dat）。

■ HKLM \ SOFTWARE \ Classes 中的系统范围的类注册数据

每个用户类注册数据的添加对 Windows 2000 来说都是新的。这种变化是为了将各用户的注册数据与系统范围的状态分离，以使配置文件包括这些定制。它还避免了一个安全漏洞：在 Microsoft Windows NT 4 中，非特权用户可以修改或删除 HKER _ CLASSES _ ROOT 的键，这样会影响系统上应用程序的操作。在 Windows 2000 中，非特权用户和应用程序可以读取系统范围的数据，但只能修改它们的私有数据。

4. HKEY _ LOCAL _ MACHINE

HKLM 是包含系统范围内的配置子键的根键，子键有 HARDWARE、SAM、SECURITY、SOFTWARE 和 SYSTEM。

HKLM \ HARDWARE 子键包含系统范围内的硬件和所有硬件设备—驱动程序映射的描述。设备管理器（Device Manager）工具（在 Control Panel 中，点击 Hardware 选项，然后点击 Device Manager，运行 System）允许你查看注册表硬件信息，只要简单地阅读 HARDWARE 键的键值即可。

HKLM \ SAM 包含本机帐号和组信息，如用户口令、组定义和域关联（domain association）。作为域控制器（domain controller）操作的 Windows 2000 Server 系统在 Active Directory 中存储域帐号和分组，Active Directory 是一个存储全域设置和信息的数据库（Active Directory 不在本书介绍）。在默认情况下，SAM 的安全描述符是这样配置的，即使是系统管理员帐号也不能访问。如果你想了解其内部，可以修改安全描述符允许对系统管理员进行读访问，但这种查看不能揭示太多，因为数据不存档并且口令是通过单向映射加密的——也就是说不能从加密形式确定口令。

HKLM \ SECURITY 存储系统范围的安全策略和用户权限分配。HKLM \ SAM 链接到 HKLM \ SECURITY \ SAM 下的 SECURITY 子键。在默认情况下，你不能查看 HKLM \ SECURITY 或 HKLM \ SECURITY \ SAM 的内容，因为这些键的安全设置只允许系统帐号访问（系统帐号在本章后面详细讨论）。

HKLM \ SOFTWARE 是 Windows 2000 存储引导系统时不需要的系统范围的配置信息的地方。另外，第三方应用程序在这里存储它们的系统范围的设置，如应用程序文件的路径、目录、授权和截止日期信息。

HKLM \ SYSTEM 包括引导系统时需要的系统范围的配置信息，如安装哪一设备驱动程序和启动哪一服务。因为这些信息对启动系统是关键的，所以 Windows 2000 在该键下保存该部分信息的副本，称为所知最近的正确控制集（last known good control set）。当 CurrentControlSet 键的改变导致系统不能启动时，副本的维护允许系统管理员选择以前的工作控制设置。有关 Windows 2000 何时认为 CurrentControlSet 键是“正确”的详情，请参见后面 5.2.6 节“接受引导和所

知最近的正确配置”小节。

5.HKEY _ CURRENT _ CONFIG

HKEY CURRENT CONFIG 链接到当前的硬件配置文件，它存储在 HKLM \ SYSTEM \ CurrentControlSet \ Hardware Profiles \ Current 下。硬件配置（hardware profile）文件允许系统管理员配置基本系统驱动程序设置的变化。尽管基本配置文件可能会因引导的不同而变化，但通过该键应用程序可以一直引用当前活动配置文件。

实验：查看 SAM 和 SECURITY 键

尽管 SAM 和 SECURITY 键是受安全设置保护的，它只允许通过系统帐号访问。但你可以使用一个技巧，在不改变它的安全性的情况下使用注册表编辑器（Registry Editor）查看它的内容。*at* 命令在系统帐号下运行你指定的应用程序。因此，如果你指定 Regedit.exe 为应用程序，让 *at* 交互地启动 Regedit，你将有一个 Regedit 实例，它可以查看 SAM 和 SECURITY 键的内部。

6.HKEY PERFORMANCE DATA

该注册表是访问 Windows 2000 性能计数器（performance counter）键值的机制，不管它们来自操作系统组件还是服务应用程序。通过注册表提供性能计数器访问的好处是远程性能监视器可以自由地工作，因为通过正常的注册表 API 很容易远程访问注册表。

通过打开名为 HKEY _ PERFORMANCE DATA 的特殊键并查询下面的键值，你可以直接访问注册表性能计数器信息。在 Registry Editor 中查看你将找不到该键；该键只能通过编程方式 Win32 注册表函数得到，如 *RegQueryValueEx*。性能信息并没有实际存储在注册表中；注册表函数使用该键查找来自性能数据提供程序的信息。

你也可以使用从 Performance Data Helper API（Pdh.dll）中可以获得的 Performance Data Helper（PDH）函数来访问性能计数器信息。图 5-1 显示了访问性能计数器信息所涉及的组件。

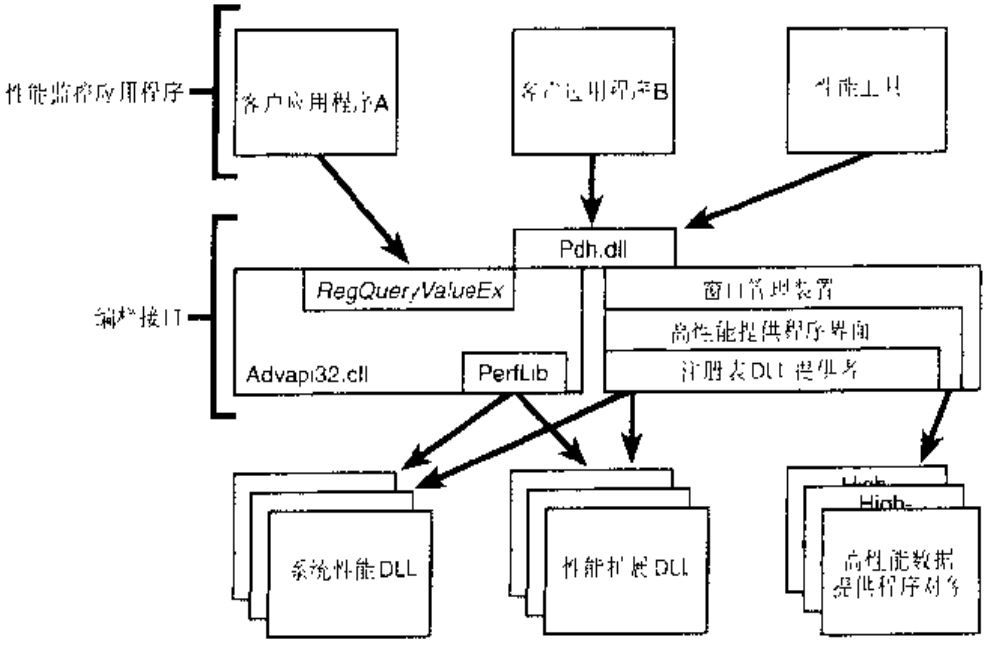


图 5-1 注册表性能计数器体系结构

实验：查看注册表活动

注册表监视器（registry Monitor）工具（在配套光盘的 \ Sysint \ Regmon.exe）可用来监视注册表发生的活动。对于每一个注册表访问，Regmon 向你显示执行访问的进程、访问的时间、类型及结果。该信息对了解应用程序与系统依赖注册表的方式，了解应用程序及系统存储配置设置的地方，以及丢失注册表键或键值的应用程序的故障诊断等问题是有用的。如图 5-2 所示的快照中，Regmon 显示了当启动 Computer Management 时，Microsoft Management Console（MMC）完成的某些注册表访问（Regmon 还包括显示访问时间的一栏，但我们省略该栏以节省空间）。

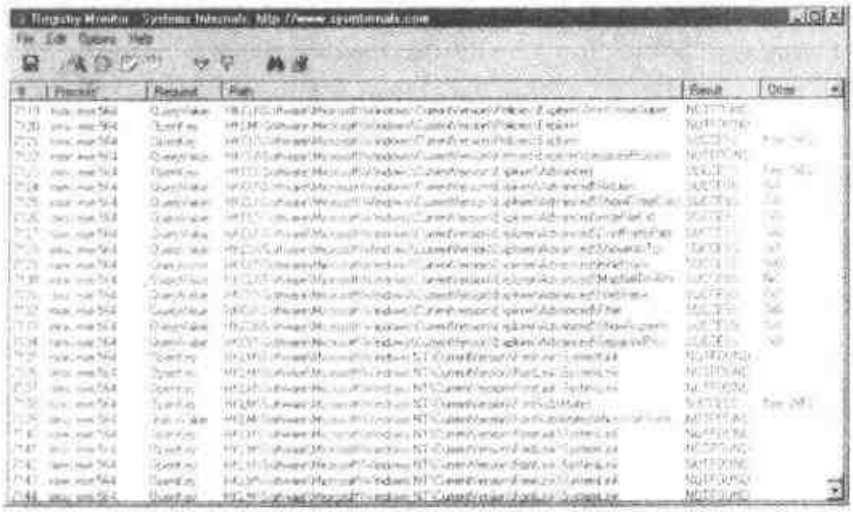


图 5-2 查看注册表活动

Regmon 依赖于系统调用工作簿（*system-call booking*）技术，通过该技术内核模式驱动程序代替系统服务调度表中注册表相关函数的项。当应用程序执行 Win32 注册表调用（如 *RegCreatKey*）时，相应的系统服务（在该情况下即 *NtCreatKey*）被激活，导致执行 Regmon 的异常分支（hook）例程。

5.1.3 注册表内部

本节介绍配置管理器（实现注册表的执行程序子系统）如何组织注册表的磁盘上文件（on-disk file）。当应用程序和其他操作系统组件读取并改变注册表键和键值时，我们将考查配置管理器如何管理注册表。我们还将讨论即使在注册表被修改导致系统崩溃的情况下，配置管理器确保注册表始终是可恢复状态的机制。

1. Hive

在磁盘上，注册表并不是一个简单的大文件，而是一系列称为 hive（配置单元）的离散文件。每一个 hive 包含一个注册表树，该树有一个键代表根或树的开始点，子键和其键值在根的下面。你可能认为通过注册表编辑器显示的根键与 hive 中的根键相互有联系，但情况并非如此。表 5-4 列出了注册表 hive 和它们的磁盘上文件名。除用户配置文件外，所有 hive 的路径名都被编码在配置管理器中。当配置管理器装载 hive（包括系统配置）时，它在 HKLM \ SYS-

TEM \ CurrentControlSet \ Control \ hivelist 子键的键值中记下每一个 hive 路径，当 hive 未装载时，删除路径（当不被引用时，用户配置不被装载）。它创建根键，链接这些 hive 在一起创建你所熟悉的注册表编辑器显示的注册表结构。

表 5-4 与注册表路径对应的磁盘上文件

Hive 注册表路径	Hive 文件路径
HKEY_LOCAL_MACHINE \ SYSTEM	\ Winnt \ System32 \ Config \ System
HKEY_LOCAL_MACHINE \ SAM	\ Winnt \ System32 \ Config \ Sam
HKEY_LOCAL_MACHINE \ SECURITY	\ Winnt \ System32 \ Config \ Security
HKEY_LOCAL_MACHINE \ SOFTWARE	\ Winnt \ System32 \ Config \ Software
HKEY_LOCAL_MACHINE \ HARDWARE	Volatile hive
HKEY_LOCAL_MACHINE \ SYSTEM \ Clone	Volatile hive
HKEY_USERS \ 〈Security ID of username〉	\ Documents and Settings \ 〈username〉 \ Ntuser.dat
HKEY_USERS \ 〈Security ID of username〉 Classes	\ Documents and Settings \ 〈username〉 \ Local settings \ Application Data \ Microsoft \ Windows \ Usrclass.dat
HKEY_USERS \ .DEFAULT	\ Winnt \ System32 \ Config \ Default

你会注意到表 5-4 列出的某些 hive 是非永久性的（volatile）并且没有关联文件。系统在内存中创建并管理这些 hive；所以这些 hive 是临时的。每次启动时，系统创建非永久性的 hive。一个非永久性 hive 的例子是 HKLM \ HARDWARE hive，它存储有关物理设备和设备分配的资源信息。资源分配和硬件检测在系统每次引导时发生，因此不将这些数据存储在硬盘上是合理的

实验：查看 hive 句柄

通过内核句柄表（在第 3 章讲述），配置管理器能打开 hive，因此它能从任一进程环境中访问 hive。使用内核句柄表是通过使用驱动程序或执行程序从系统进程内访问不能被用户进程访问的句柄的有效的可替换方法，你可以使用 HandleEx 工具（本书配套光盘的 \ Sysint \ Handleex.exe）来查看 hive 句柄。因为当句柄在 System Idle 进程被打开时，对象管理器报告内核句柄表，在上面窗格中选择 System Idle Process 来查看 hive 句柄，如图 5-3 所示（一定在 View 菜单中选择 View Handles）。

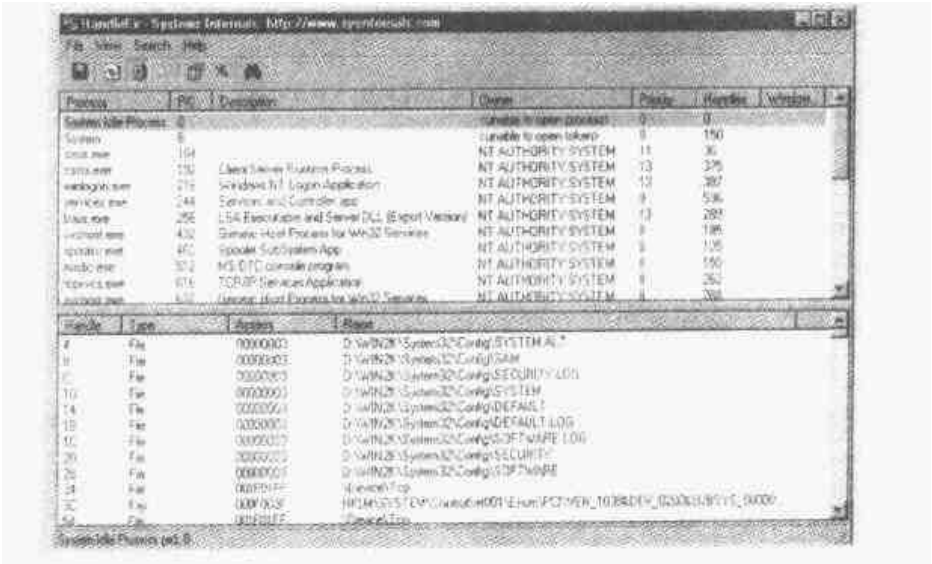


图 5-3 查看 hive 句柄

一种称为符号链接（*symbolic link*）的特殊类型的键使得配置管理器有可能把 hive 连接起来组织成注册表。符号链接是重定位配置管理器到另一键的键。这样 HKLM \ SAM 键是到 SAM hive 根键的符号链接。

2. hive 结构

配置管理器逻辑上将 hive 划分为称为“块”（block）的分配单元，很象文件系统将磁盘划分为簇的方法。根据定义，注册表块大小为 4096 字节（4KB）。当新数据扩充 hive 时，hive 始终以块粒度渐增。hive 的第一个块为基块（*base block*）。基块包括 hive 的全局信息，包括签名（signature）*regf*，它将文件标识为 hive、更新的序列号、表明在 hive 上最后进行写操作的时间戳（time stamp）、hive 格式版本号、检校和（checksum）以及 hive 文件的内部文件名（例如 \ Device \ HarddiskVolume1 \ WINNT \ CONFIG \ SAM）。在描述数据是如何写到 hive 文件中时，将说明更新序列和时间戳的重要性。Hive 格式版本号指定了 hive 内的数据格式。Hive 格式从 Windows NT 3.51 到 Windows NT 4 始终在变化，因此如果想将 Windows NT 4 或 Windows 2000 的 hive 加载到 Windows NT 的早期版本，你会失败。

Windows 2000 在称为“单元”（cell）的包容器（container）中组织 hive 存储的注册表数据。单元包括键、键值、安全描述符、子键列表或键值列表。单元数据的开头字段描述数据类型。表 5-5 详细描述了每一单元的数据类型。单元的头是用于指定单元大小的字段。当单元加入 hive 时，hive 必须扩充以容纳该单元，系统创建一个称为 bin 的分配单元。bin 的大小为新的单元到下一个块边界的大小。系统考虑单元端与分配给其他单元的 bin 自由空间端之间的任何空间。bin 也有头（header），它包括签名、*hbin* 和记录 *bin* 对 hive 文件的偏置以及 bin 大小的字段。

表 5-5 单元数据类型

数据类型	描述
键单元（key cell）	该单元包括注册表键，也称为键结点（key node）。键单元包含签名（kn 代表键，kl 代表符号链接）、最近更新的键的时间戳、键的父键单元的单元索引、标识键的子键的子键列表单元的单元索引、键的安全描述符单元的单元索引、指定键类名的字符串键的单元索引以及键的名字（例如 CurrentControlSet）
键值单元（value cell）	该单元包括键的键值的信息。该单元包括签名（kv）、键值的类型（例如 REG_DWORD 或 REG_BINARY）、键值的名字（如 Boot-Execute）。键值单元还包括包含键值数据的单元的单元索引
子键列表单元（subkey-list cell）	由同一父键的所有子键的键单元的单元索引列表组成的单元
键值列表单元（value-list cell）	由同一父键的所有键值的键值单元的单元索引列表组成的单元
安全描述符单元（security-descriptor cell）	该单元包括安全描述符。安全描述符单元在单元头包括签名（ks）以及记录共享同一安全描述符的键结点数的引用计数。多个键单元可以共享同一安全描述符单元

通过使用 bin 代替单元（cell）来跟踪注册表的活动部分，Windows 2000 使管理变得简单。例如，通常系统分配与释放 bin 比分配与释放单元的频率要低，这使得配置管理器管理存储更为有效。当配置管理器将注册表 hive 读入内存时，它可以选择只读入包括单元的 bin（也就是

活动的（active）bin），并忽略空的 bin。当系统添加或删除 hive 中的单元时，hive 可以包含空 bin 以及活动的 bin。这种情况与磁盘碎片相似，它发生在当系统在磁盘上创建或删除文件时。当 bin 为空时，配置管理器将空的 bin 与相邻的空 bin 链接到一起以形成一个尽可能大的连续的空 bin。配置管理器也将相邻删除的单元链接到一起形成更大的空单元（配置管理器不会试图压缩注册表 hive，你可以通过使用 Win32 *RegSaveKey* 和 *RegReplaceKey* 函数（它由 Windows Backup 工具使用）的备份和恢复来压缩注册表）。

创建 hive 结构的链接称为单元索引（*cell index*）。单元索引是单元在 hive 文件上的偏置（offset）。这样，单元索引如同从一个单元到另一个单元的指针，而配置管理器相对 hive 的启始位置来解释它。例如，你在表 5-5 中看到的，描述键的单元包含了指定父键的单元索引的字段；子键的单元索引指定了描述从属于指定子键的子键的单元。子键列表单元包含访问子键的键单元的单元索引列表。因此，如果你想查找子键 A 的键单元，它的父键为键 B。你必须首先使用键 B 单元的子键列表单元索引定位包含键 B 的子键列表的单元。然后通过子键列表单元中的单元索引列表定位键 B 的各子键单元。对于各子键单元，你要检查键单元存储的子键名字与你所要定位的子键是否匹配，在本例中为子键 A。

单元（cell）、bin 和块（block）的概念很容易混淆，因此让我们了解简单注册表 hive 布局的例子以便区分它们。图 5-4 中的简单注册表 hive 文件例子中包含一个基块和两个 bin。第一个 bin 是空的，第二个 bin 包含几个单元。在逻辑上，一个 hive 只有两个键：根键 Root 及其子键 Sub 键。Root 有两个键值，Val1 与 Val2。子键列表单元定位根键的子键，键值列表单元定位根键的键值。第 2 个 bin 的自由空间是空单元。该图没有显示这两个键的安全单元，它应该在 hive 中出现。

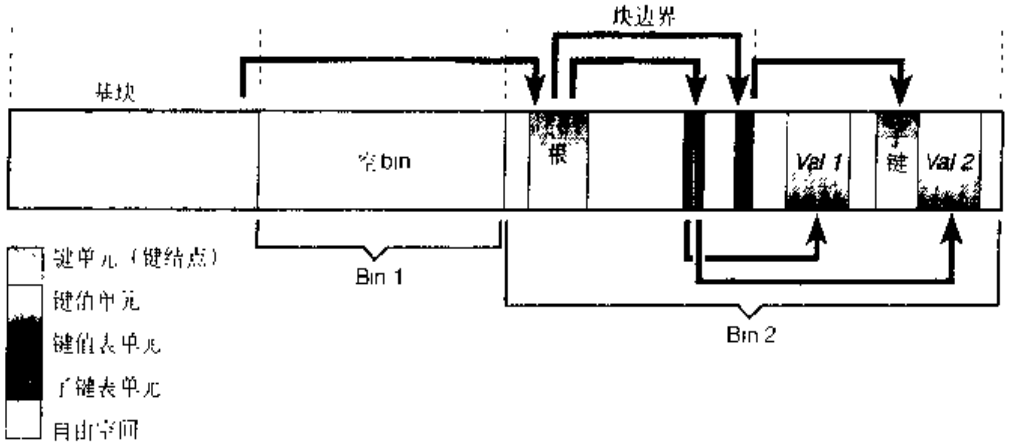


图 5-4 注册表 hive 的内部结构

图 5-5 显示了 Disk Probe 工具（Dskprobe.exe）检查 System hive 第一个 bin 的例子。请注意图形右上方的 bin 签名（signature）hbin。在 bin 签名的下面，你将看到签名 *nk*。该签名是键单元（*kn*）的签名。该签名之所以反向显示是由 x86 计算机存储数据的方式决定的。该单元为 SYSTEM hive 的根单元，配置管理器在内部将其命名为 $ $ $ PROTO.HIV，由 *nk* 签名后面的名字指定。

为了优化对键值与子键的查找，配置管理器将子键列表单元按字母顺序排序。当它查找子键列表中的子键时，配置管理器可以进行二分法查找。配置管理器可以从列表中检索子键，如果配置管理器查找的子键名字的字母顺序在中间子键名字之前，那么配置管理器就会知道该子键在子键列表的前半部分；否则，该子键在子键列表的后半部分。这种分割过程一直进行，直

到配置管理器找到该子键或找不到匹配的子键为止。键值列表单元并不排序，因此新的键值可以一直添加到列表的尾端。

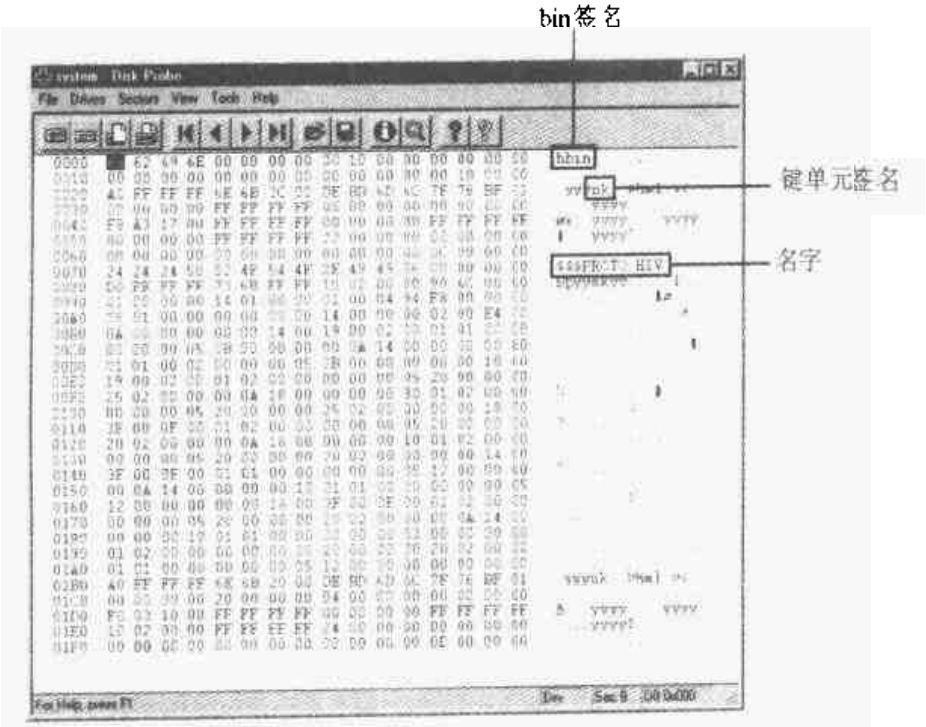


图 5-5 SYSTEM hive 中的第一个 bin 的二进制内容

3. 单元映射（cell map）

配置管理器并不是在每次访问注册表时访问 hive 在磁盘上的映像。Windows 2000 在内核地址空间上保留了每一 hive 的版本。当 hive 初始化时，配置管理器确定 hive 文件的大小，在页交换区（paged pool）分配足够的内存来存储它并将 hive 文件读入内存（有关页交换区的详细信息，请参见第 7 章）。因为所有加载的注册表 hive 都被读入页交换区中，所以一般来说注册表数据是页交换区的最大消耗者（要检查页交换区的分配，请使用 Poolmon 工具，该工具在第 7 章“监视页交换区的使用”一节的实验中有介绍）。

如果 hive 从未增长，配置管理器将内存上的 hive 版本当做文件来完成它的注册表管理。给定一个单元索引，配置管理器将通过简单地将单元索引（它是 hive 偏移量）和 hive 映像在内存中的基址相加来计算单元在内存中的位置。在系统引导早期，Ntldr 对 SYSTEM hive 所做的操作如下：Ntldr 将整个 SYSTEM hive 作为只读 hive 读入内存，并将单元索引和 hive 映像在内存中的基址相加来定位单元。不幸的是，当它们加入新的键和键值时，hive 就会增长，这意味着系统必须分配页交换区内存来存储包含添加的键与键值的新 bin。这样，内存中保存注册表数据的页交换区不必是连续的。

实验：查看 hive 页交换区的使用

没有管理级的工具向你显示 hive（包括用户配置文件）所消耗页交换区的数量。但是，!regpool 内核调试程序命令不仅向你显示所加载的每一 hive 所消耗页交换区的页数，而且显示有多少页被用来存储易失和非易失数据。在输出的末端，该命令显示所有 hive 内存的使用（该命令仅显示 hive 名字的最后 32 个字符）。

```
kd> !regpoo'

dumping hive at e20d66a8 (a\Microsoft\Windows\UsrClass.dat)
  Stable Length = 1000
  1/1 pages present
  Volatile Length = 0

dumping hive at e215ee88 (ettings\Administrator\ntuser.dat)
  Stable Length = f2000
  242/242 pages present
  Volatile Length = 2000
  2/2 pages present

dumping hive at e13fa188 (\SystemRoot\System32\Config\SAM)
  Stable Length = 5000
  5/5 pages present
  Volatile Length = 0

dumping hive at e142fe68 (stemRoot\System32\Config\DEFAULT)
  Stable Length = 24000
  36/36 pages present
  Volatile Length = 0

dumping hive at e13fbb48 (temRoot\System32\Config\SOFTWARE)
  Stable Length = b40000
  1856/2880 pages present
  Volatile Length = 1000
  1/1 pages present

dumping hive at e13fa948 (temRoot\System32\Config\SECURITY)
  Stable Length = b000
  11/11 pages present
  Volatile Length = 1000
  1/1 pages present

dumping hive at e128c008 (NONAME)
  Stable Length = d000
  10/13 pages present
  Volatile Length = 4000
  4/4 pages present

dumping hive at e1008528 (SYSTEM)
  Stable Length = 274000
  628/628 pages present
  Volatile Length = 36000
  54/54 pages present

dumping hive at e10088e8 (NONAME)
  Stable Length = 1000
  1/1 pages present
  Volatile Length = 0
Total pages present = 2852 / 3879
```

为了解决内存中用不连续的内存缓冲区存储 hive 数据的问题，配置管理器采用一种与 Windows 2000 内存管理器将虚拟内存地址映射为物理内存地址的相似的策略。配置管理器采用两级模式，如图 5-6 所示，它将单元索引（即 hive 文件偏移量）作为输入，而输出单元索引所在块（Block）的内存地址与单元所在 bin 的内存地址。请记住一个 bin 可以包括一个或多个块，并且 hive 在块中增长，因此 Windows 2000 总是将一个 bin 表示为连续的内存缓冲区。因此，bin 的所有块出现在页交换区（paged pool）的同一部分。

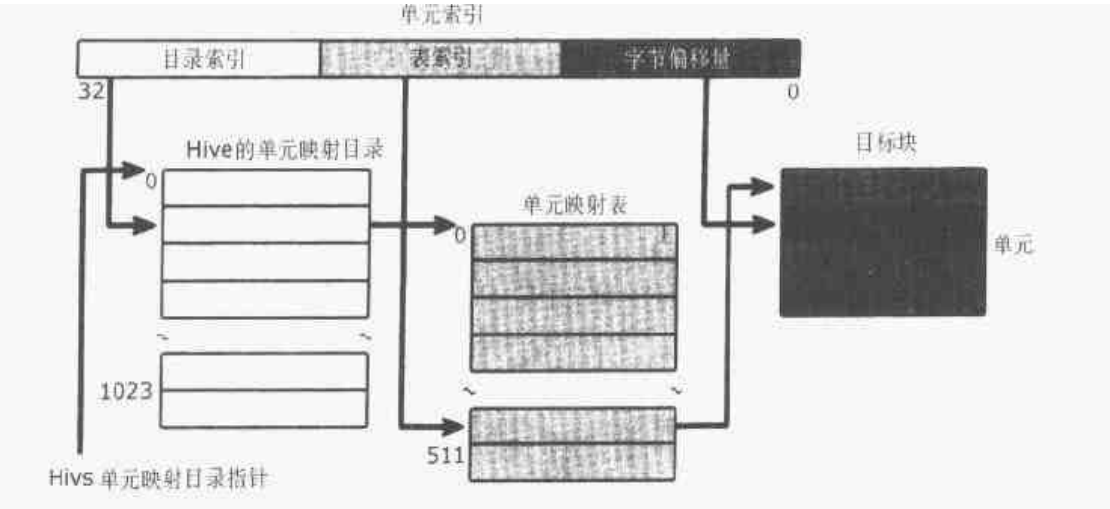


图 5-6 单元索引结构

为了实现映射，配置管理器在逻辑上将单元索引划分为字段，就象内存管理器将虚拟地址划分为字段一样。Windows 2000 将单元索引的第一个字段解释为指向 hive 映射趴录的索引。hive 映射目录包含 1024 个项，每一项指向包含 512 个映射项的单元映射表。该单元映射表中的项由单元索引的第二个字段指定。该项定位 bin 和单元的块内存地址。在变换过程的最后一步，配置管理器将单元索引的最后字段解释为标识块的偏移量以精确定位在内存中的单元。当 hive 初始化时，配置管理器动态创建映射表，它为 hive 中的每一块指定映射项，并且随 hive 大小的变化而在单元目录中添加、删除表。

4. 注册表名字空间和操作

配置管理器定义一个称为“键对象”（key object）的对象类型，它将注册表的名字空间与内核的一般名字空间相统一起来。配置管理器将名为 Registry 的键对象插入 Windows 2000 名字空间的根（root）而作为注册表的入口点。注册表编辑器（Regedit）以 HKEY-LOCAL-MACHINE \ SYSTEM \ CurrentControlSet 的形式显示键名，但 Win32 子系统将这些名字转换为对象名字空间形式（例如：\ Registry \ Machine \ System \ CurrentControlSet）。当 Windows 2000 对象管理器分析（parse）该名字时，它首先通过名字 Registry 找到该键对象，并将名字的其他部分传递给配置管理器。配置管理器负责名字分析，在它的内部 hive 树中查找需要的键或键值。在描述典型注册表操作控制流（flow of control）之前，需要讨论一下键对象和键控制块（*key control block*）。无论什么时候当应用程序打开或创建一个注册表键时，对象管理器都会给出一个句柄（handle）借由它来访问应用程序的键。在对象管理器的帮助下，句柄对应于配置管理器分配的键对象。通过使用对象管理器的对象支持，配置管理器能利用对象管理器提供的安全与引用记

数（reference-counting）的功能。

对每一打开的注册表键，配置管理器将分配一个键控制块。键控制块存储键的路径全名，包括控制块引用的键结点（key node）的单元索引，还包括一个标记（flag），它注释当最后键句柄关闭时配置管理器是否需要删除键控制块引用的键单元。Windows 2000 将所有的键控制块放入按字母排序的二分树中使得可以通过名字快速搜索现有的键控制块。键对象指向它相对应的键控制块，因此如果两个应用程序打开同一注册表键，每一个应用程序将获得一个键对象，并且两个键对象将指向同一键控制块。

当应用程序打开一个存在的注册表键时，控制流起始于一个应用程序，它指定在注册表 API 中调用对象管理器名字分析例程的键名。对象管理器一旦在名字空间找到配置管理器的注册表键对象，它便把路径名传递给配置管理器。配置管理器利用内存 hive 数据结构在键与子键中搜索以找到指定的键。如果配置管理器找到键单元，它将搜索键控制块树以确定键是否打开（通过同一或另一应用程序）。搜索例程是这样优化的，它总是从已打开的键控制块的最近的祖先开始。例如如果应用程序打开 \ Registry \ Machine \ Key1 \ Subkey2，而 \ Registry \ Machine 已经打开，分析例程（parse routine）利用 \ Registry \ Machine 注册表控制块作为起始点。如果键打开，配置管理器增加存在的键控制块的引用记数。如果键没打开，配置管理器分配一个新的键控制块并将它插入树中。然后配置管理器分配一个键对象，它指向键控制块中的键对象，对象管理器重新获得控制，它向应用程序返回句柄。

当应用程序创建新的注册表键时，配置管理器首先为新键的父键找到键单元，然后搜索容纳新键的 hive 的空闲单元列表，以确定足以容纳新键单元的单元是否存在。如果没有，配置管理器分配新的 bin 供单元使用，并将 bin 末端的空间放在空闲单元列表上。新键单元填充相关信息（包括键的名字）配置管理器将键单元添加到父键的子键列表单元的子键列表。最后，系统将父单元的单元索引存储在新子键的键单元中。

配置管理器利用键控制块的引用记数来确定何时删除键控制块。当键控制块中引用键的所有句柄关闭时，引用记数为 0，它表明该键控制块不再需要。如果调用 API 来删除键的应用程序设置删除标志，配置管理器可以从键的 hive 中删除关联键，因为它知道没有应用程序打开该键。

5. 稳态存储

为了保证非永久性注册表 hive（每一个都对应有磁盘上的文件）始终处于可恢复状态，配置管理器使用日志 hive（*log hive*）。每一非永久性 hive 都有一个关联的日志 hive，它是一个隐式文件，有着 hive 和 .log 扩展相同的基址名。例如，如果你查看 \ Winnt \ System32 \ Config 目录（选择 Show Hidden Files And Folders 文件夹选项），会看到 System.log、Sam.log 和其他的 .log 文件。当 hive 初始化时，配置管理器分配一个位数组（bit array），其中的每一位表示 hive 的 512 字节区域或扇区。该数组称为脏扇区数组（*dirty sector array*），因为数组中的 on 位表明系统已经修改了 hive 在内存中的对应扇区，而且必须将该扇区写到 hive 文件。（off 位代表对应的扇区已经用内存中 hive 的内容更新）。

当创建新键或键值、修改现有键或键值时，配置管理器在 hive 的脏扇区数组中记录改变的 hive 扇区。然后配置管理器调度延迟写操作，或 hive 同步（*hive sync*）。在请求同步 hive 并为

所有 hive 从内存到磁盘 hive 文件写脏 hive 扇区后，延迟书写器（lazy writer）系统线程在 5 秒后唤醒。这样，系统同时刷新发生在请求 hive 同步请求和 hive 同步发生之间的注册表修改。当 hive 同步发生时，下一个 hive 同步将在随后的 5 秒发生。

如果延迟书写器仅简单地将所有 hive 的脏扇区写到 hive 文件并且系统在中间操作状态崩溃，那么 hive 文件将是不一致的（破坏的）并且不可恢复。为了防止这种事情发生，延迟书写器首先将 hive 的脏扇区数组和所有的脏扇区转储到 hive 的日志文件，如果有必要会增加日志文件的大小。接着迟书写器更新 hive 基址块中的序列号，并将脏扇区写到 hive。当延迟书写器结束时，它更新基址块中的第二个序列号。这样，如果系统在 hive 的写操作期间发生系统崩溃，在第二次启动时，配置管理器将注意在 hive 基址块中的两个序列号不匹配。配置管理器利用 hive 日志文件中的脏扇区更新 hive，hive 将向前滚动。这样 hive 及时更新并保持一致。

为更好地保护关键 SYSTEM hive 的整体性，配置管理器在磁盘上保存 SYSTEM hive 的镜像（mirror）。如果你在 \ Winnt \ System32 \ Config 目录中查看非隐藏文件，你将看到三个具有基址名 *system* 的文件：System、System.alt 和 System.sav。System.alt 是备用 hive（*alternate hive*）。无论什么时候当 hive 同步刷新脏扇区到 SYSTEM hive，hive 同步更新 System.alt hive。在系统启动时，如果配置管理器检测到 SYSTEM hive 已被破坏，配置管理器将试图加载 hive 的备用版本。如果 hive 是可用的，它将使用备用的 hive 来更新原有的 SYSTEM hive。

在 Windows 2000 完成安装后，System.sav 是 SYSTEM hive 的副本。该副本是可用的，一般仅在极端情况下，它用来恢复计算机的配置为初始状态。

6. 注册表优化

配置管理器采取了一些值得注意的性能优化。首先，几乎每一注册表键都有一个安全描述符来保护对键的访问。将各键的唯一安全描述符副本保存在 hive 中效率是极低的，但是因为相同的安全设置通常应用于注册表的整个子树。因此当系统将安全施加于键时，配置管理器首先检查与该键的父键相关联的安全描述符及其所有父键的子键。如果任意的安全描述符与系统应用于键的安全描述符相匹配，配置管理器将简单地使键共享现有的描述符，并用引用记数来跟踪有多少键共享同一描述符。

配置管理器还优化它在 hive 中存储键与键值名的方式。尽管注册表完全采用 Unicode，并且用 Unicode 约定来指定所有的名字，但如果名字中仅包括 ASCII 字符，配置管理器在 hive 中以 ASCII 形式存储名字。当配置管理器读取名字时（例如当执行名字查找时），在内存中将名字转换为 Unicode 形式。将名字以 ASCII 形式存储可以显著地减小 hive 的大小。

为了减少内存的使用，键控制块并不存储全部的键注册表路径名。它们仅引用键的名字。例如，访问 \ Registry \ System \ Control 的键控制块并不访问全部路径，而是访问名字 *Control*。更进一步的存储优化是配置管理器利用键名控制块来存储键名，并且具有相同名字的所有键控制块共享同一键名控制块。要优化性能，配置管理器将键控制块名字存储在散列表中以便快速查找。

为了提供对键控制块的快速访问，配置管理器将频繁访问的键控制块存储在高速缓存表中，它设置为一个散列表。配置管理器需要查找键控制块时，它首先检查高速缓存表。最后，配置管理器还有另一个高速缓存——延迟关闭表（delayed close table），它存储应用程序关闭的

键控制块。这样应用程序能够很快的重新打开它最近关闭的键。配置管理器删除延迟关闭表中最老的键控制块，将最新关闭的块添加到表中。

5.2 服务

几乎每个操作系统都在系统启动的时候拥有启动进程的机制，它可以提供没有与交互用户结合在一起的服务。在 Windows 2000 中，该进程叫服务，或 Win32 服务，因为它们依赖于 Win32 API 与系统交互。服务与 UNIX 新进程相似并且经常被实现为客户机/服务应用的服务端。Web 服务可算是 Win32 服务的一个例子，因为无论是否有人登录该机器它都必须运行，而且当系统启动时，Web 也须同时启动，这样系统管理员无须去记忆，甚至在启动时不需在场。

Win32 服务由三部分组成：服务应用程序、服务控制程序（SCP）和服务控制管理器（SCM）。首先，我们介绍服务应用程序、服务帐号和 SCM 的操作。然后我们介绍当系统引导时，自动启动（auto-start）服务是如何启动的。我们也会介绍服务在启动过程中失败时 SCM 采取的步骤以及 SCM 关闭服务的方法。

5.2.1 服务应用程序

服务应用程序，例如 Web 服务，至少由一个可执行的程序组成，即 Win32 服务。用户通过 SCP 可以启动、停止或运行服务。尽管 Windows 2000 提供内建的 SCP，用来保证通常的开、关、暂停和继续功能，但许多服务仍使用其自身的 SCP，这样允许系统管理员指定特定于所管理的服务的配置设置。

服务应用程序是带有附加代码用来接收来自 SCM 的命令或把应用程序状态反馈给 SCM 的简单的 Win32 可执行程序（GUI 或控制台）。由于大多数服务没有用户界面，所以它们都被建成了控制台程序。

当你安装一个包括服务的应用程序时，应用程序的安装程序必须向系统注册服务。为了注册服务，安装程序会调用 Win32 的 CreatService 函数，这是一个与服务相关的用 Advapi32.dll（\ Winnt \ system32 \ Advapi32.dll）实现的函数。Advapi32，即“Advanced API”DLL，实现所有的客户端 SCM API。

当安装程序调用 CreatService 注册服务时，消息被发送给服务所在的机器上的 SCM，然后 SCM 在 HKLM \ SYSTEM \ CurrentControlSet \ services 中为服务创建注册表键。注册表键是 SCM 的数据库的永久性表示。对于每个服务的键都定义了包括服务、参数和配置选项的可执行映像的路径。

在创建服务后，安装或管理应用程序可通过 StartService 函数启动服务，由于在引导过程期间，基于服务的应用程序也须初始化，因此安装程序经常把服务注册为自动启动服务，让用户重新启动系统以完成安装以及当系统引导时让 SCM 启动服务。

当程序调用 CreatService 时，它必须指定一系列描述服务特征的参数。这些特征包括服务的类型（它是否是运行在自己的进程中而不是与其他服务共享进程的服务）、服务的可执行映像文件的位置、可选的显示名称、可选的帐号名称、在特定帐号安全环境中用以启动服务的可选帐号名称和密码、当系统引导或者在 SCP 引导下手工启动时用来显示服务是否自动启动的

启动类型（start type）、描述当系统启动时如果发现错误如何反应的错误代码以及当服务自动启动时指定服务相对于其他服务何时启动的可选信息。

在服务的注册表键中，SCM把每个特征保存为键值。图表5-7所示为一个服务注册表键的例子。

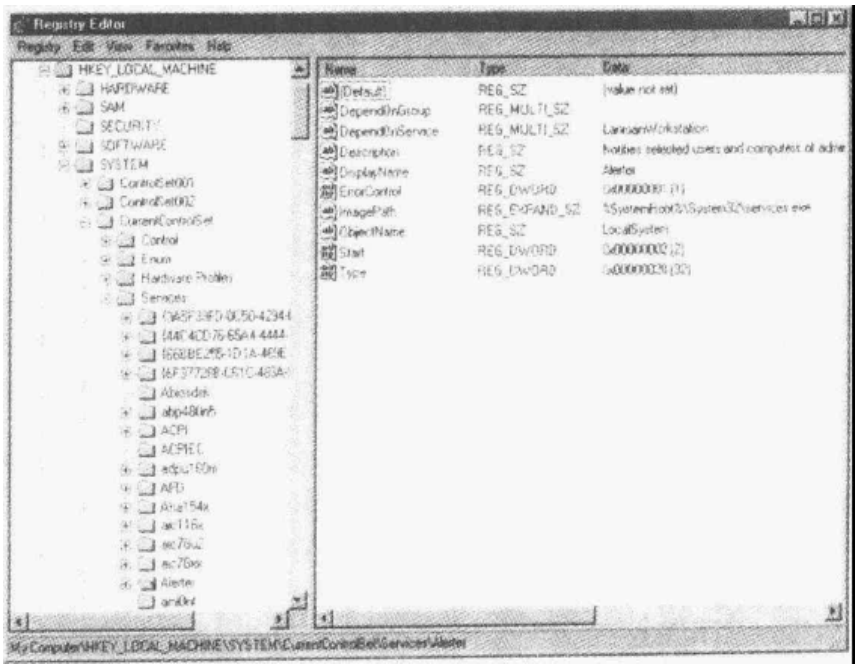


图5-7 服务注册表键的例子

表5-6列出了所有的服务特征（不是每一个特征都适用于每一种类型的服务）。如果服务需要保存专属于服务自身的配置信息，常用的方法是在它的Service键下创建名为Parameter的子键，然后在该子键中以键值形式存放配置信息。服务可通过使用标准注册函数来重新获取那些键值。

表5-6 服务和驱动程序注册表参数

键值名称	键值设置	键值设置描述
Start	SERVICE_BOOT_START（0）	Ntldr或Osloader预加载驱动程序以便在引导时它们在内存之中。这些驱动程序可以在SERVICE_SYSTEM_START驱动程序之前初始化
	SERVICE_SYSTEM_START（1）	驱动程序在SERVICE_BOOT_START驱动程序完成初始化之后加载和初始化
	SERVICE_AUTO_START（2）	SCM启动驱动程序或服务
	SERVICE_DEMAND_START（3）	SCM必须按需要启动驱动程序或服务
	SERVICE_DISABLED（4）	驱动程序或服务没有加载或初始化
ErrorControl	SERVICE_ERROR_IGNORE（0）	I/O管理器忽略驱动程序返回的错误，不显示和记录警告
	SERVICE_ERROR_NORMAL（1）	若驱动程序报告出错，显示警告
	SERVICE_ERROR_SEVERE（2）	若驱动程序返回错误并且所知最近的正确模式（last known good）没被使用，则按照所知最近的正确模式重新引导；否则，继续引导

（续）

键值名称	键值设置	键值设置描述
	SERVICE _ ERROR _ CRITICAL (3)	如果驱动程序返回错误，并且所知最近的正确模式没被使用，在所知最近的正确模式下重新引导；否则，显示蓝屏终止引导
Type	SERVICE _ KERNEL _ DRIVER (1)	设备驱动程序
	SERVICE _ FILE _ SYSTEM _ DRIVER (2)	内核模式文件系统驱动程序
	SERVICE _ RECOGNIZER _ DRIVER (8)	文件系统识别驱动程序
	SERVICE _ WIN32 _ OWN _ PROCESS (16)	进程中运行的服务仅提供一种服务
	SERVICE _ WIN32 _ SHARE _ PROCESS (32)	进程中运行的服务提供多种服务
	SERVICE _ INTERACTIVE _ PROCESS (256)	允许服务在控制台上显示窗口并且接受用户输入
Group	组名	当组初始化时驱动程序或服务初始化
标签	标签数	在组初始化序列中指定的位置，此参数不适用于服务
ImagePath	服务或驱动程序的可执行文件的路径	若 ImagePath 不是指定的，I/O 管理器在 \ Winnt \ System32 \ Drivers 下查找驱动程序，SCM 在 \ Winnt \ System32 \ 下查找服务
DependOnGroup	组名	除非驱动程序或服务从指定组处加载，否则驱动程序或服务不会加载
DependOnService	服务名	服务只有在特定服务加载之后才会加载，该参数不适用于设备驱动程序
ObjectName	通常是 LocalSystem，但也可是帐号名。如 \ Administrator	指定服务运行的帐号。若不指定 ObjectName，则 LocalSystem 为工作帐号，该参数不适用于设备驱动程序
DisplayName	服务名	服务应用程序用这个名来显示服务，如果没有指定名字，服务注册表键的名字成为该指定名字
Discription	服务说明	多达 1024 字节的服务说明
FailureAction	当服务进程以未预料的方式退出时，SCM 应该采取的行为的描述	故障行为包括重新启动服务进程，重引导系统和运行指定的程序，该值不适用于驱动程序
FailureCommand	程序命令行	仅当 FailureAction 指定了当服务失败时应该执行的程序，SCM 才读此值，该值不适用于驱动程序
Security	安全描述符	该值包含定义了谁拥有访问服务的权限的安全描述符

注意 Type 值包含可用于设备驱动程序的三个方面：设备驱动程序、文件系统驱动程序和文件识别程序。Windows 2000 设备驱动程序使用它们，同时它们的参数也在服务注册表键中以注册表数据形式存储。SCM 负责启动 Start 值为 SERVICE _ AUTO _ START 的驱动程序，所以对 SCM 数据库而言包含驱动程序是很自然的事情。服务使用其他类型：SERVICE _ WIN32 _ OWN _ PROCESS 和 SERVICE _ WIN32 _ SHARE _ PROCESS，它们是互相排斥的。提供不止一个服务的可执行程序指定 SERVICE _ WIN32 _ SHARE _ PROCESS 类型。让一个进程运行不止一个服务的优点在于它将节约需要运行多个不同的进程所需要的系统资源，而其潜在的缺点在于如果在同一进程上运行的多个服务中的一个出现故障而终止进程工作，那么所有该进程上的服务都将终止。

当 SCM 启动服务进程时，该进程就会立即调用 StartServiceCtrlDispather 函数，StartServiceC-

trlDispatcher接受一列服务的进入点，进程中的一个进入点对应一个服务。每个进入点由进入点对应的服务的名称来标记。在创建好到SCM的称为“名字管道”（namedpipe）的通信连接之后，StartServiceCtrlDispatcher在循环中等待来自SCM通过名字管道（namedpipe）发出的命令。SCM每次发送一个服务启动（Service－Start）命令，该命令启动进程所拥有的服务。每次接到启动命令，StartServiceCtrlDispatcher函数就会创建一个线程（即服务线程）来调用正在启动的服务的进入点并且实现服务的命令循环。StartServiceCtrlDispatcher随时等待SCM的命令并且只有当所有进程的服务线程都终止后，才将控制返回到进程的主函数，允许服务进程在退出前清除资源。

服务进入点的首要行为是调用RegisterServiceCtrlHandler函数、该函数接受并且存储服务实现的函数表以处理来自于SCM的众多命令。RegisterServiceCtrlHandler不与SCM通信，但它为StartServiceCtrlDispatcher函数将此表存储在当前进程的内存中。服务进入点继续初始化服务，包括分配内存、创建通信端点和从注册表中读取专有的配置数据。一个大多数服务遵循的约定是将它们的参数存储在服务的注册表键称为Parameter的子健下。当进入点初始化服务时，它可能周期性地将状态信息传送给SCM显示服务启动的进展。在进入点完成初始化后，服务线程通常在循环中等待来自客户机应用程序的请求。例如，Web服务器初始化TCP倾听socket，并且等待进入的HTTP连接请求。

在StartServiceCtrlDispatcher函数中执行的服务进程的主线程接收目标为进程中的服务的SCM指令、并且使用服务的处理程序函数表（由RegisterServiceCtrlHandler存储）来定位和调用负责响应命令的服务函数。SCM命令包括停止、暂停、恢复、询问、关机或应用程序定义的命令，图5-8显示了服务进程的内部组成，图中是构成一个进程来共同提供一个服务的两个线程：主线程和服务线程。

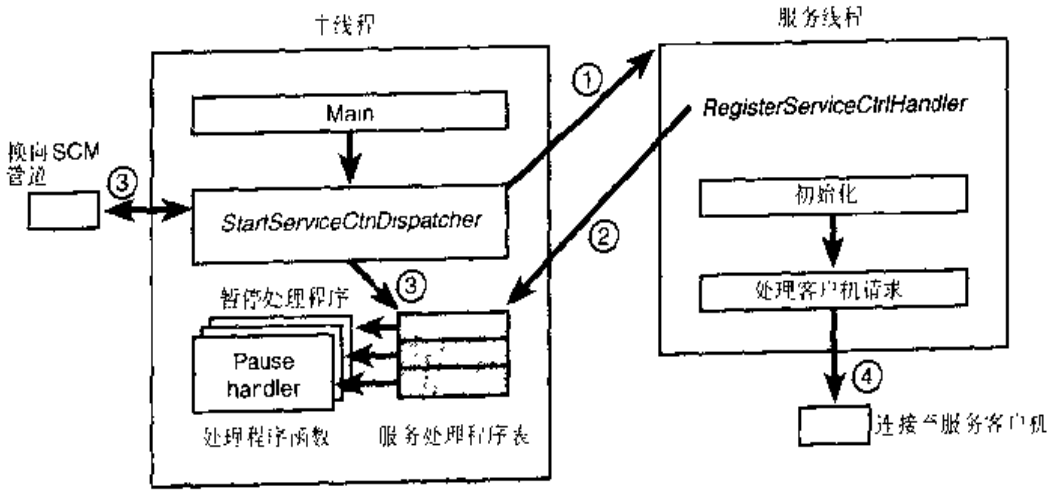


图5-8 服务进程的内部构造

SrvAny 工具

如果你有一个想当作服务运行的程序，你需要修改启动代码以遵从本节中列出的服务的需求。如果你没有源代码，可以使用 Windows 2000 资源箱中的 SrvAny 工具，SrvAny 允许你将任何应用程序作为服务运行，它读服务文件的路径，该文件必须从服务注册表键的 Parameter 子键中加载。当 SrvAny 启动时，它通知 SCM 它正在提供一个特殊的服务，并且当接收到启动命令时，把服务可执行程序作为一个子进程。该子进程接收 SrvAny 进程的访问令牌（access token）的拷贝和对于同一窗口站（windows station）的引用，以便在配置 SrvAny 进程时该可执行程序能在相同安全帐号和相同的交互设置下运行。SrvAny 服务没有共享的进程 Type 值，所以你使用 SrvAny 作为服务安装的应用程序都将以 SrvAny 宿主程序的不同实例的方式运行在不同的进程中。

5.2.2 服务帐号

服务的安全环境对于服务开发人员和系统管理员来说是十分重要的事情，因为它指导进程存取何种资源。除非服务的安装程序或服务管理员特别指定，否则服务在本机系统帐号的安全环境下运行（有时作为 SYSTEM 显示，有时作为 LocalSystem 显示）。下面从两个方面来介绍本帐号的特征。

1. 本机系统帐号

本机系统帐号是所有 Windows 2000 用户型操作系统组件使用的同一帐号，这些组件包括会话管理器（\ Winnt \ System32 \ Smss.exe）、Win32 子系统进程（Csrss.exe）、本机安全权限子系统（\ Winnt \ System32 \ Lsass.exe）和 Winlogon 进程（\ Winnt \ System32 \ Winlogon.exe）。

从安全的角度来看，本机系统帐号在当前系统执行安全任务时是相当强大的——比任何本机或域帐号都强大。该帐号有以下特点：

■ 它是本机系统管理员组的成员。

■ 它有权利去设置几乎所有的特权（甚至是那些通常不授予本机系统管理员的特权，如创建安全令牌）。

■ 大多数文件和注册表键都被完全地授予本机系统帐号的权利（即使未赋予使用本机系统帐号的完全权利，在本机系统帐号下运行的进程也可以尝试获取使用本机系统帐号的权利）。

■ 在本机系统帐号下进程以缺省用户配置使用（HKU \ .DEFAULT）。因此它们不能访问存储在另一帐号下的用户配置文件中的配置信息。

■ 当一个系统是 Windows 2000 域的成员时，对于服务进程正在运行的计算机来说，本机系统帐号包括机器的安全标记符。因此运行在本机系统帐号下的服务将被同一森林（forest）（森林指一组域）中的其他计算机使用它的计算机帐号自动验证。

■ 除非机器帐号被特别授予存取资源的权利（如网络共享命名管道（namedpipe）等等）进程都可访问允许空会话的网络资源，也就是那些不需要信用的连接。你可以在特定的计算机上指定共享（share）和管道，这些计算机允许在 NullSessionPipes 和 HKLM \ SYSTEM \ CurrentControlSet \ Services \ Lanmanserver \ Parameters 下的 NullSessionShares 注册表键值中允许空会话。

2. 在可选帐号下运行服务

由于刚才提到的限制，一些服务需要用户帐号的安全凭证才可以运行。当服务被创建或通过指定在 Windows 2000Service MMC 插件下运行的服务的帐号和密码时，你可以配置在可选帐号下运行的服务。在服务的插件中，右击服务并选择 Propertities，点击 Log On 标签，接着选择 This Account 选项，如图 5-9 所示。

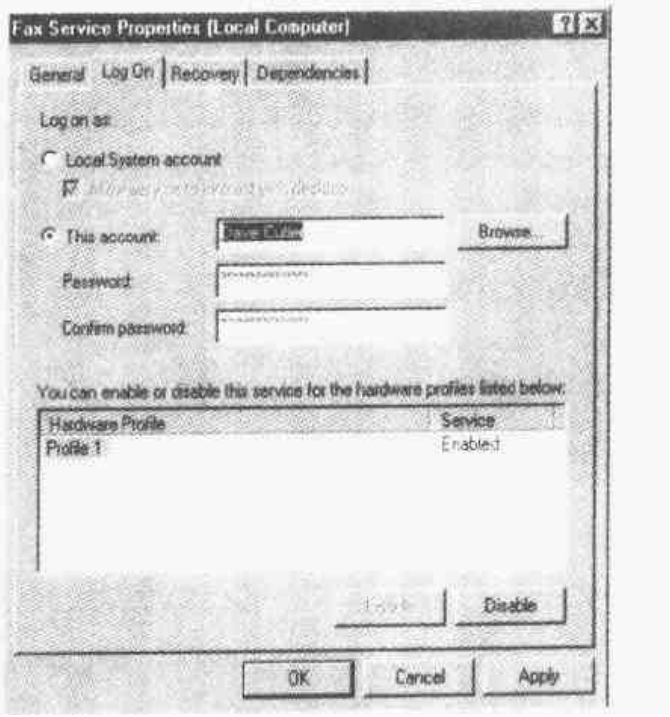


图 5-9 服务帐号设置

3. 交互服务

服务在本机系统帐号下运行的另一个限制是它们不能显示对话框或交互用户的桌面窗口（没有使用在 MessageBox 函数中的特别标记，过一会再讨论）。这一限制不是在本机系统帐号下运行的直接结果，而是服务控制器向窗口站分派服务进程方式的结果。

Win32 子系统将每一 Win32 进程与窗口站关联。窗口站包括桌面，而桌面又包括窗口。在控制台上只可以看见一个窗口站，正是它接受用户鼠标和键盘输入。在终端服务环境中，每一个会话对应的窗口站是可见的，但服务都作为控制台会话的一部分来运行。Win32 把可见的窗口站命名为 WinSta0 工作站，所有的交互进程都访问 WinSta0。

除非另有指定，SCM 把所有非交互式的服务共享的名为 Service-0x0-3e7$的不可见窗口站与服务相关联。名字中的数字 3e7 代表登录会话标识符，该标识符由 Lsass 分配给登录会话，而 SCM 则把登录会话用于在本机系统帐号下运行的非交互服务。

配置在用户帐号（也就是说不是本机系统帐号）下运行的服务运行在不同的不可见的窗口站中，它以分配给服务登录会话的 Lsass 登录标识符命名。图 5-10 给出了一个在 \ Sysint \ Winobj 下显示的例子（这可以在本书的配套 CD 上找到），它显示了 Win32 放置窗口站对象的对象管理器目录。交互式窗口站（WindSta0）、非交互式系统服务窗口站（Service-0x0-3e7$）和分配给作为用户登录的服务进程（Service-0x0-6368f $）的非交互窗口站都是可见的。

无论服务是否在用户帐号或在本机系统帐号下运行，不在可见窗口站上运行的服务不能接受来自用户或控制台显示窗口的输入。事实上，如果服务在窗口站上弹出一个常用对话框，因为没有用户可以看见这个对话框，服务会被挂起，当然这将妨碍用户使用键盘或鼠标输入而忽略它以及允许服务继续执行。（一个例外是如果特殊的标记 MB _ SERVICE _ NOTIFICATION 或 MB _ DEFAULT _ DESKTOP _ ONLY 在消息框调用中被设置——如果 MB SERVICE _ NOTIFICATION 被指定，消息框将在交互式窗口站上显示，即使服务没有配置与用户交互的权限；如果 MB _ DEFAULT _ DESKTOP _ ONLY 被指定，消息框在交互式窗口站的缺省桌面上显示。

尽管比较少，但有些服务有合理的理由允许通过对话框或 Windows 与用户联系。一个有着这种需求的内建 Windows 2000 服务是 Windows 安装器——交互用户需要看与软件安装相关的信

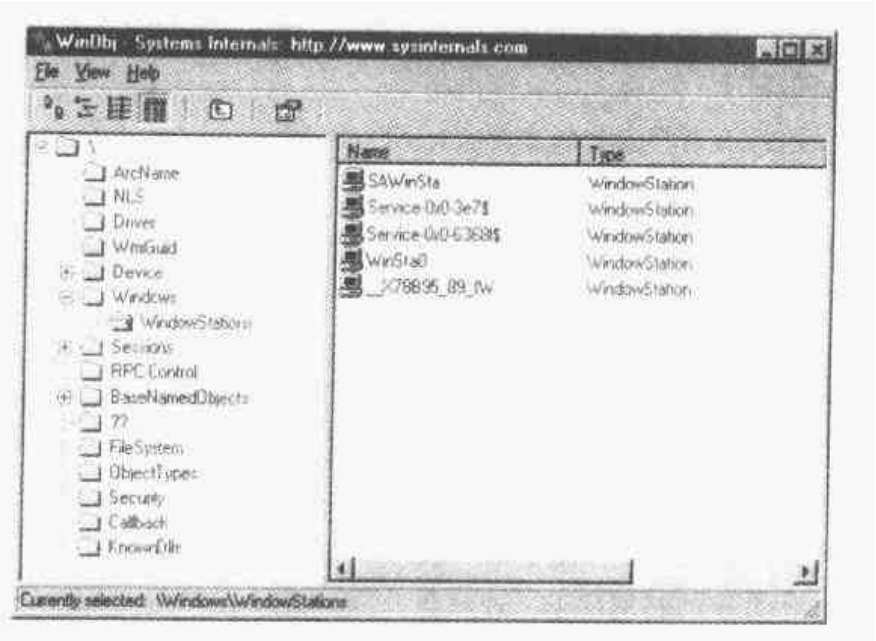


图 5-10 窗口站列表

息。为配置与用户交互权限的服务，SERVICE _ INTERACTIVE _ PROCESS 限定词必须在服务注册表键的 Type 参数中出现（注意配置在用户帐号下运行的服务不能被标记为交互式）。当 SCM 启动被标记为交互式的服务时，它在本机系统帐号安全环境下启动服务进程，但用 WinSta0 而不是非交互式窗口站连接服务。与 WinSta0 的连接允许服务在控制台上显示对话框和窗口，同时允许这些窗口响应用户输入。

5.2.3 服务控制管理器

SCM 的可执行文件是 \ Winnt \ System32 \ Services.exe，像大多数服务进程一样，它作为 Win32 控制台程序运行。在系统引导期间，Winlogon 进程提前启动 SCM（详情参见第 4 章的引导进程）。SCM 的启动函数 SvcCtrMain 协调为自动启动配置的服务。SvcCtrlMain 通常在 Winlogon 加载代表着登录对话框的图形识别和验证界面（GINA）之前和在屏幕被切换为空白桌面后立即执行。

SvcCtrMain 首先创建一个作为无信号状态初始化的名字为 SvcCtrlEvent _ A3752DX 的同步事件。仅仅在 SCM 完成准备接收来自 SCM 命令的必须步骤之后，SCM 才会将事件设置为有信号状态（signaled state）。SCP 用来建立带有 SCM 的对话框的函数是 OpenSCManager，OpenSCManager 通过等待 SvcCtrlEvent _ A3752DX 成为有信号状态来防止 SCM 在初始化之前和 SCM 通信。

接着，Svc CtrlMain 开始着手工作并且调用 ScCreateServiceDB，这是建立 SCM 的内部服务数据库的函数。ScCreateServiceDB 读取和存储 HKLM \ SYSTEM \ CurrentControlSet \ Control \ ServiceGroupOrder \ List 的内容，一个列出了定义的服务组的名字和顺序的 REG _ MULTI _ SZ 键值。如果服务或设备驱动程序需要控制来自其他组的服务的启动顺序，则服务的注册表键包含一个可选的 Group 值。例如，Windows 2000 网络栈是自底向上创建的，所以网络服务必须确定 Group 键值，这时它们的启动顺序晚于网络设备驱动程序。SCM 在内部创建一个组的列表来保存它从注册表中读取的组的顺序。组包括（并不是局限于）NDIS、TDI、主磁盘、键盘端口和键盘类。

添加的和第三方的应用程序能够定义它们自己的组而且把它们添加到列表。例如 Microsoft Transaction Server 添加称为 MS Transactions 的组。

然后 ScCreateServiceDB 扫描 HKLM \ SYSTEM \ Cuirentcontrolset \ Services 的内容，在服务数据库为它遇到的每一个键建立项。数据库项包括为服务定义的所有与服务相关的参数和跟踪服务状态的字段。SCM 为驱动程序和服务增加项，因为 SCM 启动标记为自动启动（auto-start）的服务和驱动程序，而且检测标记为引导启动和系统启动的驱动程序的启动错误。I/O 管理器在任何用户模式应用程序执行之前加载标记为引导启动和系统启动的驱动程序，因此有这些启动类型的任何驱动程序在 SCM 启动前都须加载。

ScCreateServiceDB 为了确定组的成员关系并把已在组列表中创建的组的项与这个键值联系，须读取服务的 Group 键值。这一函数也通过查询 DependOnGroup 和 DependOnService 注册表键值，读取和记录在数据库中的服务的组和服务的从属关系。图 5-11 显示了 SCM 组织服务项和组顺序列表的方式。记住，服务列表是按字母顺序排序的。该表以字母顺序排序是因为 SCM 从服务注册表键中创建该列表，而且 Windows 2000 按照字母表顺序存储注册表键。

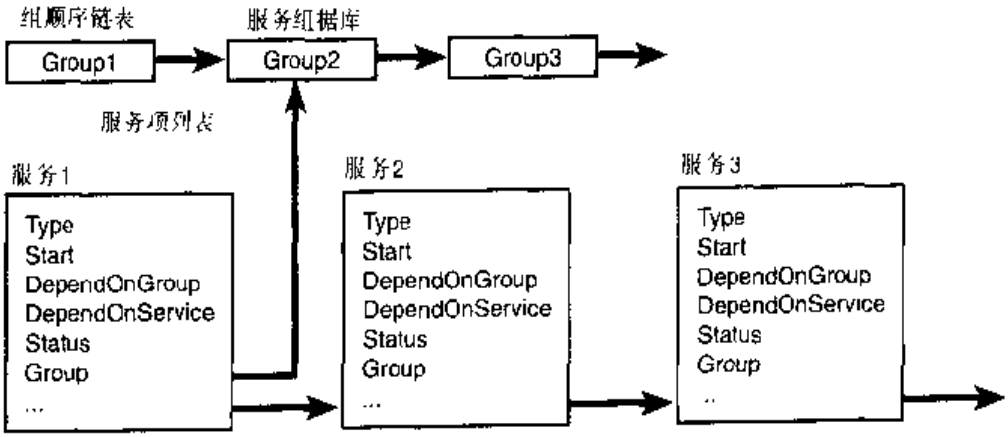


图 5-11 服务数据库的组织

在服务启动期间，SCM 可能需要调用 Lsass（例如以用户帐号注册服务），所以 SCM 等待 Lsass 发送 LSA _ RPC _ SERVER _ ACTIVE 同步事件信号，当 Lsass 完成启动时它才能做。Winlogon 也启动 Lsass 进程，所以 Lsass 的初始化是与 SCM 并发的，而且 Lsass 和 SCM 完成初始化的顺序可以变化，然后 SvcCtrlMain 调用 ScGetBootAndSystemDriverState 来扫描服务数据库查找引导启动和系统启动设备驱动程序的项。ScGetBootAndSystemDriverState 通过浏览它在对象管理器名字空间称为 \ Driver 的目录中的名字来决定驱动程序是否成功启动。当设备驱动程序成功加载后，I/O 管理器在此目录下的名字空间中插入驱动程序的对象。所以如果它的名字不存在，它就不会被加载。图 5-12 显示了显示 Driver 目录内容的 Winobj。如果驱动程序未加载，SCM 在 PnP _ DeviceList 函数返回的驱动程序列表中查找它的名字。PnP _ DeviceList 提供了包括在系统当前硬件配置文件中的驱动程序。SvcCtrlMain 记录了尚未被启动的服务的名字和称为 ScFailedDriver 的列表中的当前配置（profile）文件的部分。

启动自动启动服务之前，SCM 要完成一些步骤。它要创建远程过程调用（RPC）命名管道（namedpipe），它被命名为 \ Pipe \ Ntsvcs，然后启动线程来监听来自 SCP 的输入信息。然后它

发送初始化完成事件 SvcCtrlEvent _ A3752DX 信号。通过 RegisterServiceProcess 注册控制台关机事件处理程序以及注册 Win32 子系统进程来为系统关机准备 SCM。

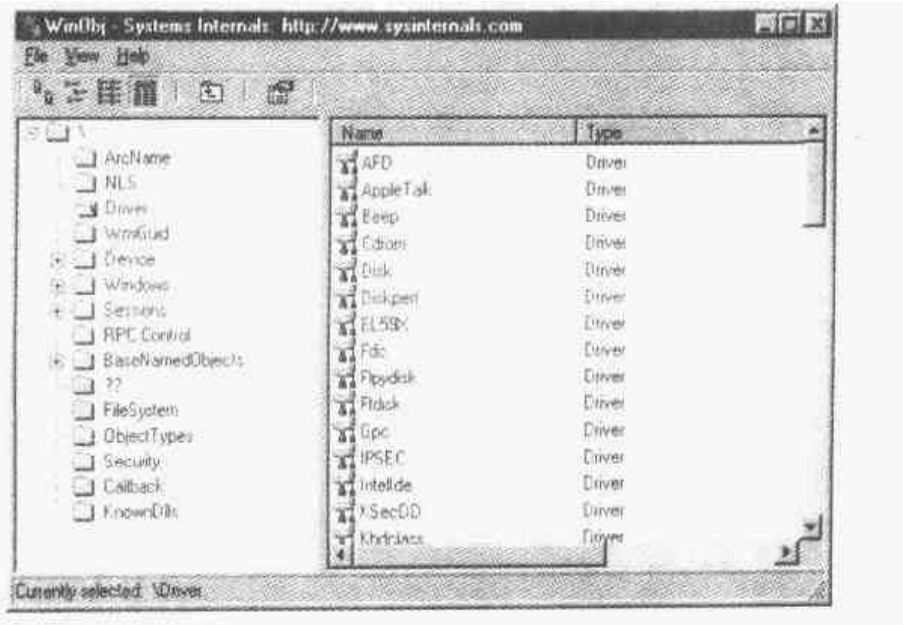


图 5-12 驱动程序对象列表

网络驱动符

除了作为服务界面的重要作用外，SCM 有另一个完全无关的责任：无论什么时候当系统创建或删除网络驱动符连接时，它都会通知系统中的 GUI 应用程序。SCM 等待 LAN Manager 工作站服务发送命名事件 ScNetPrvMsg 的信号。只要应用程序分配驱动符给远程网络共享或删除远程共享驱动符分配时，工作站服务就会发送该信号。当工作站服务发送这一事件信号时，SCM 调用 GetDriveType Win32 函数来查询连接的网络驱动符列表。如果列表在事件信号发送过程中发生改变，SCM 发送 WIM _ DEVICECHANGE 类型的 Windows 广播消息。SCM 使用 DBT _ DEVICEREMOVECOMPELTE 或 DBT _ DEVICEARRIVAL 作为消息的子类型。这一消息主要供 Windows Explorer 使用以便它更新任一打开的 My Computer 窗口从而显示网络驱动符或使之消失。

5.2.4 服务启动

为启动 Start 值标记为自动启动（auto - start）的所有服务，SvcCtrlMain 调用 SCM 函数 ScAutoStartSevices。该函数也启动标记为自动启动的设备驱动程序。为避免混淆，你应认为术语“服务”同时指服务与驱动程序，除非另有说明。为了能以正确顺序启动服务，ScAutoStartSevices 中的算法分阶段进行，一个阶段对应于一个组，而且分阶段按照在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ ServiceGroupOrder \ List 注册表值中存储的组顺序进行。List 键值，如图 5-13 所示，按 SCM 启动它们的顺序包含组的名字。因此，给组分配服务没有意义，除非根据属于不同组的其他服务来细调它的启动。

当一个阶段启动时，ScAutoStartSevices 标记所有属于启动阶段的组的服务项。然后 ScAu-

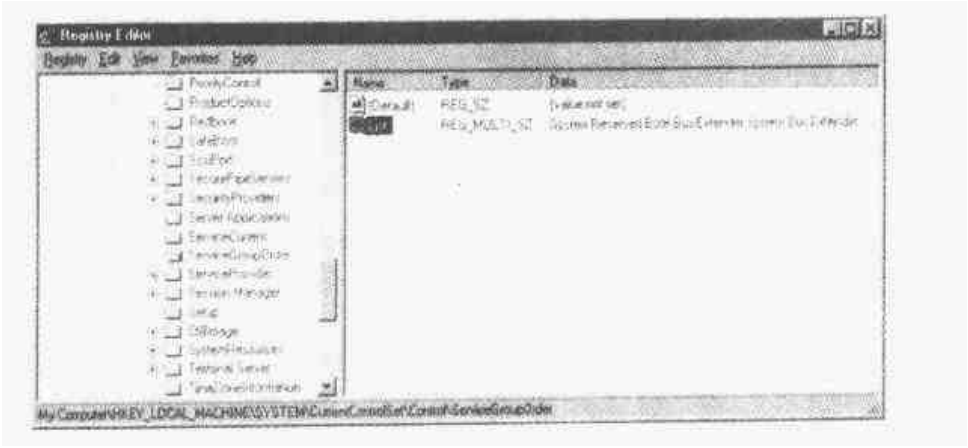


图 5-13 ServiceGroupOrder 注册表键

toStartSevices 在标记的服务中循环检查是否可以启动每一个服务。这个检查过程的一部分是确定服务是否与其他组相关，这通过在服务注册表键 DependOnGroup 值是否存在来指定。如果相关性存在，该服务的相关组必须已经初始化，该组的至少一个服务已经完成启动，如果服务依赖于一个启动顺序比服务所在的组晚的服务组，则 SCM 为服务记下“循环相关”错误。如果 ScAutoStartSevices 考虑的是 Win32 服务而不是设备驱动程序，并且如果那些服务已经启动，那么它接着检查服务是否依赖于一个或更多的其他服务。服务的相关性是在服务的注册表键中用 DependOnService 注册表键值来显示。如果一个服务与在 ServiceGroupOrder \ List 中居后的组的其他服务相关，SCM 同样会产生“循环相关”错误并且不启动服务。如果服务与那些尚未启动的同一组的任意服务相关，该服务被忽略。

当服务的相关性得到满足时，ScAutoStartSevices 做最后一次检查以在启动服务之前确定服务是否是当前引导配置的一部分。当系统以安全模式引导时，SCM 确保服务既可通过名字也可通过适当的安全引导注册表键中的组来标识。在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SafeBoot 下有两个安全引导键，Minimal 和 Network，而 SCM 访问哪个键取决于用户引导的安全模式。如果用户在特殊的引导菜单中（你可以在引导过程中按下 F8 键访问它）选择安全模式（Safe Mode）或带命令提示的安全模式（Safe Mode With Command Prompt），SCM 访问 Minimal 键；如果用户选择的是网络安全模式（Safe Mode With Networking），SCM 访问 Network 键。在 SafeBoot 键下的名为 Option 的字符串键值的存在不仅表明系统以安全模式引导，而且表明用户选择的安全模式类型。要想知道有关安全模式的更多信息，请参见 4.2 节“安全模式”部分。

一旦 SCM 决定启动服务，它会调用 ScStartService，ScStartService 为服务采用与设备驱动程序不同的步骤。当 ScStartService 启动 Win32 服务时，它首先通过从服务注册表键处读取 ImagePath 键值来确定运行服务的进程的文件名。然后它检查服务的 Type 键值，而且如果该键值是 SERVICE _ WIN32 _ SHARE _ PROCESSL（0x20），SCM 确保服务运行的进程（如果已经启动）对正在启动的特定的服务以用同样的帐号登录。服务的 ObjectName 注册表键值存储服务应该运行的用户帐号。没有 ObjectName 或本机系统的 ObjectName 的服务在本机系统帐号下运行。

SCM 通过检查服务的 ImagePath 键值在称为“映像数据库”（image database）的内部 SCM 数据库中是否存在项来核实服务的进程有没有以不同的帐号启动。如果映像数据库没有 ImagePath 键值的项，SCM 就会创建一个。当 SCM 创建新项时，它将存储用于服务的登录帐号名字

和来自服务的 ImagePath 键值的数据。SCM 要求服务具有 ImagePath 键值。如果服务没有 ImagePath 键值，SCM 报告错误声明它不能找到服务路径并且不能启动服务。如果 SCM 用匹配的 ImagePath 数据来定位已存在的映像数据库项，SCM 确信它正在启动的服务的用户帐号信息与数据库项中存储的信息是相同的——进程仅仅只能作为一个帐号登录，所以当服务指定一个不同于在同一进程中已经启动的另一服务的帐号名字时，SCM 报告出错。

如果服务的配置被指定，SCM 用 ScLogonAndStartImage 登录服务并启动服务进程。SCM 通过调用 Lsass 函数 LsaLogonUser 来注册不在系统帐号下运行的服务。LsaLogonUser 通常需要一个口令，但 SCM 向 Lsass 显示这个口令作为服务的 Lsass "秘密"存储在注册表中的键 HKLM \ SECURITY \ Policy \ Secrets 下（记住，因为 SECURITY 的缺省安全设置只允许从系统帐号下访问，所以它的内容通常是看不见的）。当 SCM 调用 LsaLogonUser 时，它指定服务登录为注册类型，所以 Lsass 在 Secret 子键中查找 SC _ < service name > 形式的密码。当 SCP 配置服务的登录信息时，SCM 指示 Lsass 将登录密码作为 Secret 存储。当登录成功时，LsaLogonUser 将返回给调用者的访问令牌（access token）的句柄，Window2000 使用该访问令牌代表用户的安全环境，并且 SCM 稍后用此访问令牌与实现服务的进程相联系。

在登录成功后，如果帐号的配置文件（profile）信息没有加载，SCM 通过调用 UserEnv DLL 的（Winnt \ System32 \ Userenv.dll）LoadUserProfile 函数来加载。< HKLM \ SOFTWARE \ Microsoft \ Windows NT \ CurrentVersion \ ProfileList \ < user profile key > \ Profile Image Path 键值包括 LoadUserProfile 加载到注册表的注册表 hive 在磁盘中的位置，使得 hive 中的信息成为服务的 HKEY _ CURRENT _ USER 键。

交互式服务必须打开 WinSta0 窗口站，但在 ScLogonAndStartImage 允许交互式服务存储 Winsta0 之前，它会检查 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Windows \ NoInteractiveServices 键值是否被设置。系统管理员设置该键值以阻止标记为交互式的服务在控制台上显示窗口。这个选项在服务器环境中是可取的，因为这种环境中不存在户响应来自交互服务的弹出式窗口。

如果进程尚未启动（例如在为另一个服务工作），作为下一步 ScLogonAndStartImage 继续启动服务的进程。SCM 调用 CreatProcessAsUser Win32 函数把进程启动为挂起状态。SCM 接下来创建一个命名管道（pipe），通过它与服务进程相联系，并且它分配给管道（namedpipe）名 \ Pipe \ Net \ NetControlPipeX，此处的 X 是每次 SCM 创建管道时增加的数。SCM 通过 ResumeThread 函数恢复服务进程，并且等待服务与 SCM 管道连接。如果它存在，注册表键 HKLM \ SYSTEM \ CurrentControlSet \ Control \ ServicesPipeTimeout 确定在 SCM 放弃之前它等待服务调用 StartServiceCtrlDispatcher 并且连接、终止进程并且断定服务启动失败的时间长短。如果 ServicePipeTimeout 不存在，SCM 使用缺省的 30 秒作为超时设置。SCM 为所有服务通信使用同一超时设置。

当服务通过管道与 SCM 连接时，SCM 向服务发送启动指令。如果服务在超时之内无法主动响应启动指令，SCM 会放弃转而启动下一个服务。当服务未响应启动请求时，SCM 并不终止进程，正如它在超时之内未调用 StartServiceCtrlDispatcher 一样，SCM 也不终止进程；取而代之，它在系统事件日志中记录错误，显示服务未能以及时的方式启动服务。

如果 SCM 调用 ScStartService 启动的服务有一个 SERVICE _ KERNEL _ DRIVER 或 SERVICE _ FILE _ SYSTEM _ DRIVER 的 Type 注册表键值，服务实际上是设备驱动程序，所以 ScStartService

调用 ScLoadDeviceDriver 来加载驱动程序。ScLoadDeviceDriver 允许为 SCM 进程加载驱动程序的安全权限，然后调用内核服务 NtloadDirver，在驱动程序的注册表键的 ImagePath 键值的数据中传递 驱动程序不像服务，不需要确定 ImagePath 键值，并且如果没有这个键值，SCM 会通过将驱动程序的名字附加给字符串 \ Winnt \ System32 \ Drivers \ 来创建一个映像路径。

ScAutoStartServices 在属于组的服务中继续循环，直到所有的服务已经启动或产生相关错误。这种循环是 SCM 根据服务的 DependOnService 的相关性自动排列组中服务的次序的方法，SCM 将启动在早期循环中被其他服务依赖的服务，忽略依赖的服务直到后来的循环到来。注意 SCM 忽略 Win32 服务的 Tag 键值，在 HKLM \ SYSTEM \ CurrentControlSet \ Services 键下的子键中可以找到它；I/O 管理器用 Tag 键值来指示为引导的设备驱动程序启动和系统启动驱动程序排序。

一旦 SCM 完成将所有组列入 ServiceGroupOrder \ List 键值中的阶段，它进入属于组却未列入值中的服务对应的阶段以及为没有组的服务完成的最后阶段。

5.2.5 启动错误

如果驱动程序或服务报告一个错误以响应 SCM 的启动命令，服务注册表键的 ErrorControl 键值决定 SCM 如何反应。如果 ErrorControl 值是 SERVICE _ ERROR _ IGNORE (0) 或者 ErrorControl 键值没有被指定，SCM 就简单地忽略错误而继续启动服务。如果 ErrorControl 键值是 SERVICE _ ERROR _ NORMAL (1)，SCM 将事件写入系统事件日志，此事件被记录为“〈服务名〉服务由于下列错误无法启动:”。SCM 包含 Win32 错误代码的文本表示，服务将此文本表示作为事件日志中记录启动失败的原因返回给 SCM。图 5-14 显示了代表服务启动失败的事件日志项。

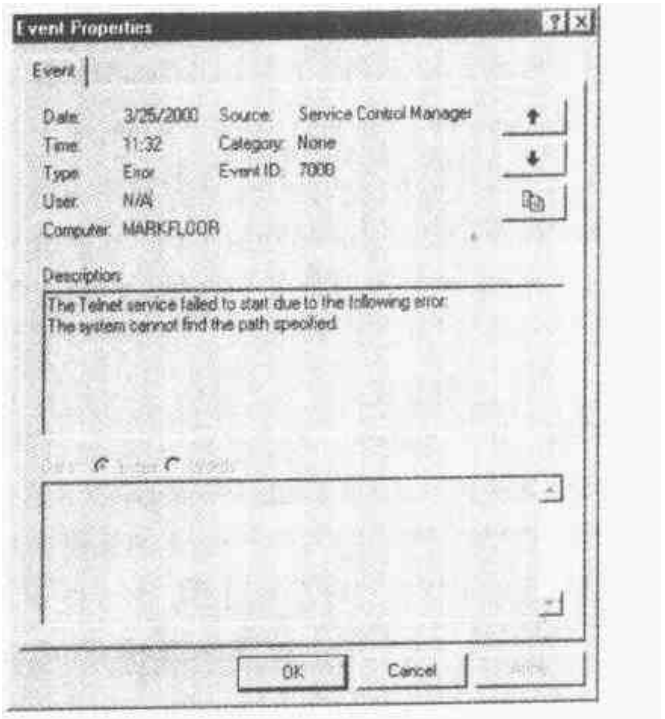


图 5-14 服务启动失败事件日志项

如果带有 SERVICE _ ERROR _ SEVERE (2) 或 SERVICE _ ERROR _ CRITICAL (3) ErrorControl 键值的服务报告启动错误，SCM 在事件日志中添加记录，然后调用内部函数 ScRevertToLastKnownGood。该函数将系统的注册表配置转换成最近的称为“所知最近的正确版本”的版本。然后它用 NTShutdownSystem 系统服务重启动系统，这在执行程序中实现。如果系统已经在用所知最近的正确版本来引导，系统就能重新引导。

5.2.6 接受引导和所知最近的正确配置

除了启动服务，系统还让 SCM 确定系统的注册表配置 HKLM \ SYSTEM \ CurrentControlSet 何时应保存为所知最近的正确控制集。CurrentControlSet 键包含作为子键的 Service 键，所以 CurrentControlSet 键包括 SCM 数据库的注册表表示。它也包含 Control 键，它存储许多内核模式和用户模式子系统的配置设置。成功的引导在缺省方式下由自动启动服务的成功启动和成功的用户

登录构成。如果系统因为设备驱动程序在引导期间导致系统崩溃而停止引导，或如果带有 SERVICE _ ERROR _ SEVERE 或 SERVICE _ ERROR _ CRITILA 的 ErrorControl 键值的自动启动服务报告启动错误，则引导失败。

SCM 显然知道何时已经完成对自动启动服务的成功启动，但有成功的登录时，Winlogon（\ Winnt \ System32 \ Winlogon.exe）必须通知 SCM 。当一个用户登录时，Winlogon 启动 NotifyBootConfigStatus 函数，NotifyBootConfigStatus 将一个信息传递给 SCM。成功启动自动启动服务或收到来自 NotifyBootConfigStatus 的消息（它来得较晚）之后，SCM 调用系统函数 NtInitializeRegistry 存储当前注册表的启动配置。

第三方软件开发者可以用他们自己的定义取代成功登录的 Winlogon 的定义。例如，运行 Microsoft SQL 服务的系统只有在 SQL 服务可以接收和处理事务之后才可能认为引导成功。开发者通过编写一个引导校验程序并且通过用存储在注册表键 HKLM \ SYSTEM \ CurrentControlSet \ Control \ BootVerificationProgram 中的键值指向安装程序在磁盘上的位置来强行定义自己的成功引导。另外，引导校验（boot-verification）程序的安装必须通过设置 HKLM \ SOFTWARE \ Microsoft \ windows NT \ CurrentVersion \ Winlogon \ ReportBootOK 为 0 来禁止对 NotifyBootConfigStatus 的调用。当引导校验程序已经被安装，在完成自动启动服务并等待程序在保存所知最近的正确控制集之前调用 NotifyBootConfigStatus 后，SCM 启动该引导检校程序。

Windows 2000 有多个 CurrentControlSet 的副本，并且 CurrentControlSet 键实质上是一个指向某个备份的符号注册表链接。控制集有着诸如 HKLM \ SYSTEM \ ControlSetnnn 形式的名字，此处 nnn 是一个数字诸如 001 或 002。HKLM \ SYSTEM \ Select 键包含表示每一个控制集作用的键值。例如，如果 CurrentControlSet 键指向 CurrentControlSet001，则在 Select 下的 Current 键值有一个为 1 的键值。在 Select 下的 LastKnownGood 键值包含所知最近的正确控制集的数目，这些控制集最近成功地用于引导。另一个在你的系统上可以发现的 Select 键下的键值是 Failured，它指向那些引导注定不成功的最近控制集。图 5-15 显示了系统的控制集和 Select 键值。

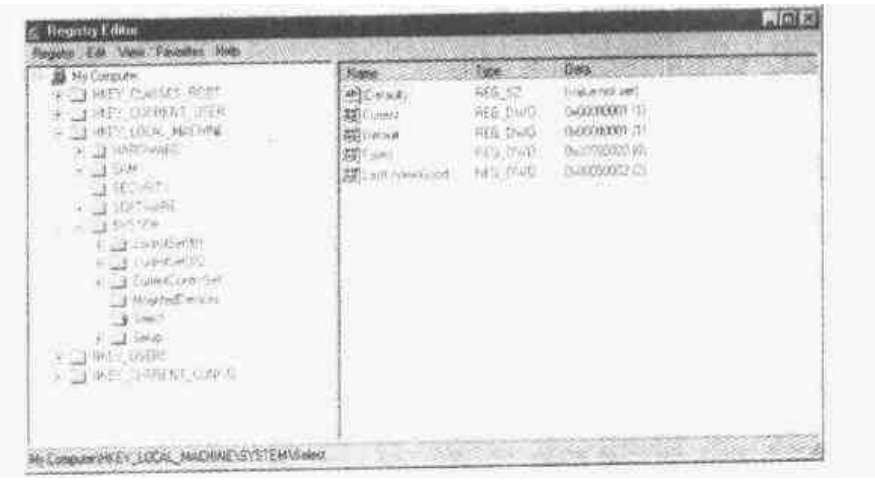


图 5-15 控制集 selection 键

NtInitializeRegistry 接收所知最近的正确控制集的内容，并且使之和 CurrentControlSet 键的树同步。如果这是系统的第一次成功引导，所知最近的正确控制集将不存在，并且系统会为它创建一个新的控制集。如果所知最近的正确控制集树存在，系统简单地更新它与 CurrentControlSet

键之间的差异。

在改变 CurrentControlSet 的情况下，例如在 HKLM \ SYSTEM \ Control 下系统性能微调值的修改以及添加服务或设备驱动程序导致随后的引导失败的情况下，所知最近的正确控制集是有用的。用户可以在引导过程中按 F8 键来产生一个菜单，这个菜单引导使用所知最近的正确控制集恢复回系统的注册表配置到它最近一次系统成功引导的方式下。

5.2.7 服务错误

当服务启动时 SCM 记录的注册表键中有可选的 FailureAction 和 FailureCommand 键值。SCM 注册系统，这样当服务进程退出时系统给 SCM 发送信号。当服务进程以未预料的方式结束时，SCM 确定哪些服务在进程中运行并且采取错误相关的注册表键值指定的恢复步骤。

对于 SCM 来说，服务能采取的行为包括重新启动服务、运行程序和重新引导计算机。此外，服务能指定服务进程第一次、第二次和后来的失败时发生的错误行为，而且如果服务需要重新启动，在启动前服务能指明 SCM 所等待的时间。IIS Admin 服务的服务错误行为导致 SCM 运行 IISReset 应用程序，这会进行清除工作然后重新启动服务。在 Service MMC 插件服务的 Properties 对话框的 Recovery 标签中，你能很容易地完成服务的恢复，如图 5-16 所示。

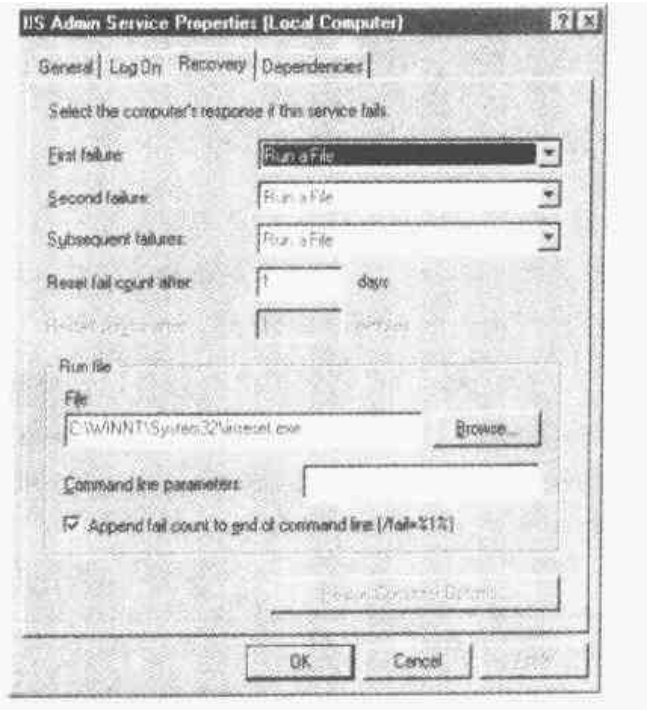


图 5-16 服务恢复选项

5.2.8 服务关闭

当 Winlogon 调用 Win32 ExitWindowsEx 函数时，ExitWindowsEx 发送消息到 Win32 子系统进程 Csrss，来调用 Csrss 的关机例程。Csrss 在当前活跃进程中循环并且通知系统正在关机。对于除 SCM 外的任一系统进程，Csrss 在移向下一次进程之前，等待由 HKU \ DEFAULT \ Control Panel \ Desktop \ WaitToKillAppTimeout（缺省为 20 秒）指定的时间达到一定秒数而退出进程。当 Csrss 遇到 SCM 进程时，它也通知 SCM 进程系统正在关闭，只是它使用 SCM 限定的超时设置。在系统初始化过程中，当 SCM 用 RegisterServiceProcess 函数注册 Csrss 时，Csrss 使用 Csrss 存储的进程 ID 来识别 SCM。SCM 的超时不同于其他进程，因为 Csrss 知道当服务关闭时，SCM 与需要执行清除任务的服务通信，因此这样管理人员也许只需要微调 SCM 的超时设置。SCM 的超时键值在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ WaitToKillServiceTimeout 注册表键值中，而且一般缺省为 20 秒。

当所有请求关闭通知的服务与 SCM 一起初始化时，SCM 的关闭处理程序负责将关机通知传递给这些服务。SCM 函数 ScShutdownAllServices 在 SCM 服务数据库中循环寻找期望关闭通知的服务，并且向每个服务发送关闭命令。SCM 对于它发送关闭命令的每一服务，都记录这个服

务的等待提示（wait hint）值，这个值在服务用 SCM 注册时服务也已经指定。SCM 跟踪它收到的最长的等待提示。发送关闭信息之后，SCM 一直等待直到它通知关闭的服务中的某个退出或最长的等待提示指定的时间过去。

如果等待提示已经过期，而服务未退出，SCM 确定它正等待退出的一个或多个服务是否已经发送信息给 SCM 以通知 SCM 服务正在关闭过程之中。如至少一个服务正在退出，SCM 再次等待一段等待提示指定的的时间。SCM 继续等待，直到所有的服务都已退出或是它等待的服务在等待提示指定的时间段中没有通知进程。

当 SCM 忙于通知服务关闭并等待它们退出时，Csrss 等待 SCM 退出。如果 Csrss 已结束等待而 SCM 尚未退出（WaitToKillService 过期），Csrss 就简单地继续关闭。这样，当系统关闭时，没有及时关闭的服务就同 SCM 一道留下来继续运行。不幸的是，对于系统管理员来说，没有办法知道它们是否应在系统上增加 WaitToKillServiceTimeout 值，从而在该系统关闭前让服务有机会在系统上完全关闭。

5.2.9 共享的服务进程

在各自的进程中运行每个服务而不是让服务共享一个进程，在任何时候都可能是系统资源的浪费。然而，共享进程意味着一旦某一服务在处理过程中出现导致进程退出的错误，所有在该进程中的服务都会中断。

Windows 2000 内建服务有些可以在单独的进程中运行，而有些则可与其他服务共享一个进程。例如，SCM 进程支持事件日志（Event Log）服务，文件服务器服务（LanmanServer）和 LAN Manager 名字解析服务。在 Windows 2000 中 SCM 支持的服务列在表 5-7 中。（不是所有的服务都在每个系统上活跃）。

表 5-7 运行在 SCM 中的 Windows 2000 服务

服 务	服务说明
Alerter	向选定的用户和计算机通知管理警告信息
AppMgmt	提供软件安装服务，例如 Assign、Publish 和 Remove 等
Browser	在网络上维护计算机的更新列表并向请求它的程序提供列表
Dhcp	通过注册和更新 IP 地址和域名系统（DSN）的名字来管理网络配置
Dmserver	逻辑磁盘管理器的监视服务（Logical Disk Manager Watchdog Service）
Dnscache	解析和高速缓存 DNS 名
Eventlog	应用程序和 Windows 发送的日志事件信息。事件日志报告包括对诊断问题有用的信息。在事件浏览器（Event Viewer）中可以看到报告
LanmanServer	提供远程过程调用（RPC）支持和文件，打印和管道（pipe）的共享
LanmanWorkstation	提供网络连接和通信
LmHosts	允许在 TCP/IP（NetBT）上支持 NetBIOS 的服务和允许支持 NetBIOS 名字解析
Messenger	发送和接收管理员或 Alerter 服务传递的信息
PlugPlay	管理设备安装和配置并且将设备的改变通知程序
ProtectedStorage	为敏感数据提供保护性存储，如专用密匙、禁止未授权的服务、进程和用户的访问

（续）

服 务	服务说明
Seclogon	在可替换的信用下允许启动进程
TrkSvr	存储信息以使在卷之间移动的文件能在域的每一个卷上被跟踪
TrkWks	在网络域中的 NTFS 卷间移动的文件间发送通知
W32Time	设定计算机时钟
Wmi	从驱动程序或向驱动程序处提供系统管理信息

安全的相关服务，诸如安全帐号管理器（SamSs）服务、Net Logon（Netlogon）服务和 IPSec 策略代理（Policy Agent）服务，共享 Lsass 进程。

还有一个叫做 Service Host（SvcHost _ \ Winnt \ System32 \ Svchost.exe）的“一般”进程包括多种服务。SvcHost 的多个实例可以运行在不同进程中。在 SvcHost 进程里运行的服务包括电话（TapiSrv）、远程过程调用（RpcSs）和远程访问连接管理器（Remote Access Connection Manager，RasMan）。Windows 2000 将运行在 SvcHost 中的服务实现为 DLL，并且包括在服务的注册表键中以“%SystemRoot%System32 \ Svchost.exe-knetsvcs”形式定义的 ImagePath 中。服务的注册表键也必须有一个在 Parameters 子键下指向服务 DLL 文件的名为 ServiceDLL 的注册表键值。

所有共享相同 SvcHost 进程的服务指定同一个参数（在前一段的例子中的“-knetsvcs”）以便它们在 SCM 的映像数据库中有单独的项。当 SCM 在服务启动过程中遇到第一个服务时，该服务有一个带特殊参数的 SvcHost ImagePath，SCM 创建一个新的映像数据库项，并且用该参数启动一个 SvcHost 进程。这个新的 SvcHost 进程接收该参数并查找一个和 HKLM \ SOFTWARE \ Microsoft \ Windows NT \ Current Version \ SvcHost 下的参数有同样名字的一个值。SvcHost 读取该键值的内容，将它解释成服务名称列表，并且当 SvcHost 用 SCM 注册时，通知 SCM 它支持这些服务。图 5-17 所示为 Svchost 注册表键的例子，该键显示用“-knetsvcs”参数开始的一个 SvcHost 进程准备支持多个不同的与网络相关的服务。

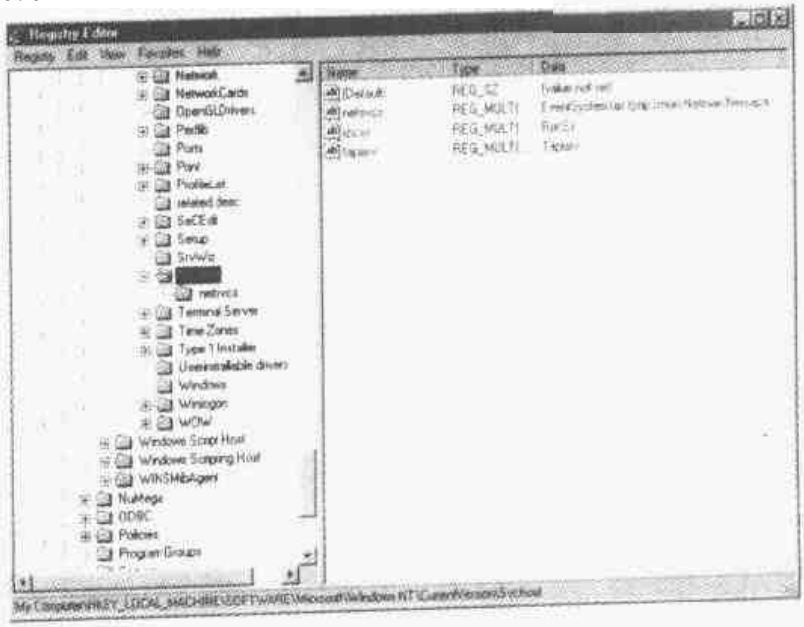


图 5-17 Svchost 注册表键

在服务启动期间，当 SCM 通过匹配映像数据库中已有的项的 ImagePath 找到一个 SvcHost 服务时，它不启动第二个进程而只是为服务将启动命令发送给它已经为 ImagePath 键值启动的 SvcHost。存在的 SvcHost 进程在服务的注册表键中读取 serviceDLL 参数，并在它的进程中加载 DLL 以启动服务。

实验：浏览进程中运行的服务

在 Windows 2000 支持工具中的 Tlist 实用程序指定/s 命令行选项。通过指定这个选项，Tlist 显示服务列表，如果有的话这些服务在进程中执行，如下输出的例子所示。名字后标有“Svcs:”而不是“Title:”的过程是服务进程，所列名字为进程提供的服务的名字。

```
C:\>tlist /s
   0 System Process
   8 System
 144 smss.exe
 172 csrss.exe      Title:
 192 winlogon.exe   Title: NetDDE Agent
 220 services.exe   Svcs:  Browser,Dhcp,dmserver,Dnscache,
                           Eventlog,lanmanserver,lanmanworkstation,
                           LmHosts,Messenger,PlugPlay,ProtectedStorage,
                           seclogon,TrkWks,Wmi
 232 lsass.exe      Svcs:  PolicyAgent,SamSs
 392 svchost.exe    Svcs:  RpcSs
 428 spoolsv.exe    Svcs:  Spooler
 456 ati2plab.exe   Svcs:  Ati HotKey Poller
 472 svchost.exe    Svcs:  EventSystem,Netman,NtmsSvc,RasMan,SENS,TapiSrv
 508 regsvc.exe     Svcs:  RemoteRegistry
 544 WinMgmt.exe    Svcs:  WinMgmt
 836 Explorer.Exe   Title: Program Manager
 712 RealPlay.exe   Title:
 748 Atiptaxx.exe   Title: ATI Tray Icon Application
 724 DESKTOPS.EXE   Title: MultiDesk
 824 Explorer.Exe   Title: Program Manager
 900 Explorer.Exe   Title: Program Manager
 564 cmd.exe        Title: Command Prompt - tlist /s
 752 ntvdm.exe
 668 calc.exe       Title: Calculator
 848 calc.exe       Title: Calculator
 308 calc.exe       Title: Calculator
 660 tlist.exe
```

5.2.10 服务控制程序

服务控制程序是标准的 Win32 应用程序，它使用 SCM 函数 CreatService、Open Service、StartService、ControlService、QueryServiceStatus 和 DeleteService。。为了使用 SCM 函数，SCP 必须首先通过调用 OpenSCManager 函数打开一个到 SCM 的通信通道。调用时，SCP 必须指定它想执行的行为类型。例如，如果 SCP 只想枚举和显示当前在 SCM 数据库里的服务，它在调用 OpenSCManager 时请求枚举服务权限。在它初始化过程中，SCM 创建代表 SCM 数据库的内部对象，并

且使用 Windows 2000 安全函数来保护带有安全描述符的对象，此描述符确定用什么访问权限和什么帐号可以打开对象。例如，安全描述符显示 Everyone 组（每一个帐号都是该组的成员）能用枚举服务权限打开 SCM 对象。然而，只有管理员才能用创建或删除服务所需的权限打开对象。

当它为 SCM 数据库工作时，SCM 为服务自身实现安全。当 SCP 使用 CreatService 函数创建服务时，它指定 SCM 在内部将其与服务数据库中的服务项相联的安全标识符。SCM 将安全描述符作为 Security 键值存储在服务注册表键中，并且在初始化过程中当它扫描到注册表的 Service 键时读取该键值，以便安全设置在重新引导时保持一致。同样 SCP 必须确定在它调用 OpenSCManager 时，它对于 SCM 数据库所需要的访问类型，SCP 必须告诉 SCM 在调用 openService 时它对于服务所需的访问类型。SCP 请求的访问包括查询服务状态、配置、停止和启动服务的能力。

你可能非常熟悉的 SCP 是包含在 Windows 2000 中的 Service MMC 插件，它驻留在 \ Winnt \ System32 \ Filemgr.dll 中。Windows 2000 资源工具箱包括名为 SC.exe（服务控制器工具）的命令行 SCP 和名为 Srvinstw.exe（服务创建向导，Service Creation Wizard）的 GUI SCP。

有时 SCP 将服务策略置于 SCM 所实现的顶层。一个很好的例子是当服务手工启动时，Service MMC 插件实现的超时，插件显示了表示服务启动的进度的进度栏。与 SCM 无限期地等待响应启动命令的服务相反，服务插件在进度栏达到 100% 并且插件宣布服务没有及时地启动之前，仅等待 2 分钟。当服务响应 SCM 的命令如开始命令时，通过设置它们的配置状态来反映它们的进程，服务与 SCP 不直接地交互。SCP 使用 QueryServiceStatus 函数查询状态。它们能判断何时一个服务主动地更新状态，何时服务被挂起，并且 SCM 能采取适当行为通知用户服务正在做什么。

5.3 Windows 管理装置

Windows NT 已有集成的性能和系统事件监视工具。应用程序和系统使用事件管理器来报告错误和诊断消息。事件浏览器实用程序让程序管理员从本地计算机或网络的另一台计算机上查看事件输出内容。同样，性能计数器机制让应用程序和操作系统组件将与性能相关的统计资料报告给性能监视应用程序，例如性能监视器。

尽管 Windows NT4 事件监视和性能监视功能达到了它们的设计目标，但它们仍有局限性。例如，程序界面彼此不同，并且这种差异增加了那些同时使用事件和性能监视来收集数据的应用程序的复杂性。性能计数器机制提供的粒度层次更糟糕，特别是通过网络时，因为它获取的是系统上定义的所有性能计数器而不只是你感兴趣的对象。这是个要么全有要么全无的命题：对于应用程序来说没有办法查询具体组件的性能信息。也许对 Windows NT 4 监视功能来说最大的弊端是它们很少有甚至没有可扩展性，并且事件日志和性能数据的收集都没有提供在一个管理 API 中必需的双向交互作用。应用程序必须以预定义格式提供数据。性能 API 没有为应用程序提供获得与性能相关的事件通知的方法，而且那些请求事件管理器事件通知的应用程序不能将此通知限定在指定的事件类型或资源上。最后，任何一个收集设备的客户机都不能通过事件管理器或性能 API 与事件数据或性能数据提供程序（Performance data provider）通信。

为了解决这些局限以及为其他的数据源类型提供管理能力，Windows 2000 有一个新的管理

机制：Windows 管理装置（Windows Management Instrumentation，WMI）。WMI 是一个基于 Web 的企业管理工具（Web Based Enterprise Management，WBEM）的实现，是 Distributed Management Task Force（DMTF——一个工业组织）定义的标准。WBEM 标准包括了可扩展的企业数据收集和数据管理设备的设计，该设计具备管理那些由任意组件组成的本机和远程系统所要求的灵活性和可扩充性。

WMI 的支持添加在 Service Pack 4 里的 Windows NT 4 里。Windows 95 OSR2 和 Windows 98 也支持它。尽管该部分适用于支持 WMI 的所有 Windows 平台，实现细节则是特定于 Windows 2000 的。

5.3.1 WMI 体系结构

正如图 5-18 所示，WMI 包括四个主要的组件：管理应用程序（management application），WMI 基础设施（WMI infrastructure）、提供程序（provider）和管理对象（managed object）。管理应用程序是访问、显示或处理应用程序获得的有关管理对象的数据的 Windows 应用程序。Performance 工具替换是一个管理应用程序的简单例子，它依赖 WMI 而不是性能 API 来得到性能信息。更为复杂的例子是企业管理工具，它让管理人员完成企业内每台计算机的软件和硬件配置的自动化目录。

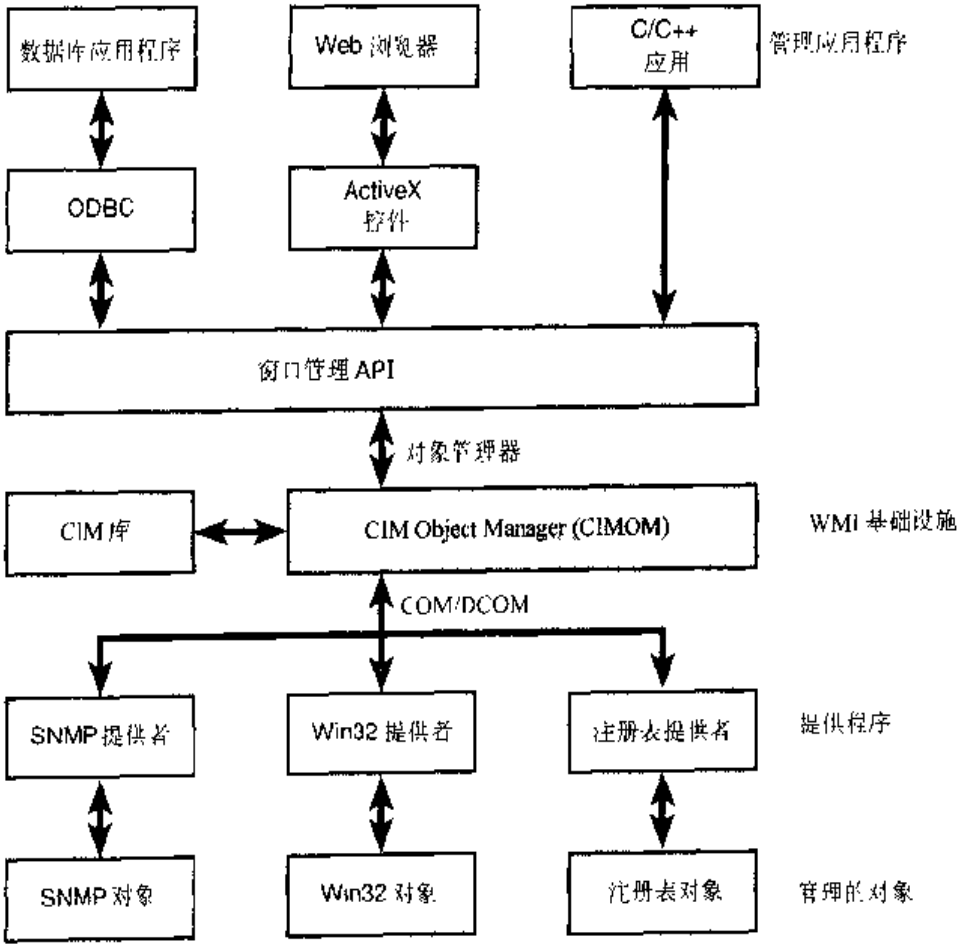


图 5-18 WMI 体系结构

开发者必须以管理应用程序作为收集数据和管理具体对象的目标。一个对象也许代表一个

组成部分，诸如网络适配器设备或组件集，例如一台计算机。（计算机对象可能包括网络适配器对象）提供程序需要指定并且输出管理应用程序感兴趣的对象的表示。例如，网络适配器产商可能想添加适配器相关的属性到 Windows 2000 包括的网络适配器 WMI 支持上，按照管理应用程序的指定来查询和设置适配器的状态和行为。在某些情况下（例如对于设备驱动程序）Microsoft 提供了一个有自己 API 的提供程序来帮助开发者用最小的编码努力为自己的管理对象利用提供程序的实现。

WMI 基础设施是将管理应用程序和提供程序结合起来的胶合剂，其内核是公共信息模型（CIM）对象管理器（CIMOM）（CIM 在本章后面部分进行描述）。这个基础设施作为对象类的存储并且在很多情况下也作为永久对象属性的存储管理器。WMI 以磁盘上（on-disk）数据库的方式来实现这个存储或库，该数据库的名字为 CIMOM Object Repository。作为它的基础设施的一部分，WMI 支持很多 API，通过这些 API 管理应用程序访问对象数据以及提供程序提供数据和类定义。

WIN32 程序使用主管理 API（WMI COM API）与 WMI 直接交互。其他的 API 在 COM API 层次之上并且包括 Microsoft Access 数据库应用程序的开放式数据库互连（Open Database Connection，ODBC）适配器。数据库开发者使用 WMI ODBC 适配器在开发者数据库里嵌入对对象数据的引用。这样开发者就能利用包含基于 WMI 的数据的数据库查询很容易地产生报告。WMI ActiveX 控件支持其他层次的 API。Web 开发者使用 ActiveX 控件来创建基于 Web 的 WMI 数据的接口。另一个管理 API 是 WMI 脚本 API，适用于基于脚本的应用程序和 Microsoft Visual Basic 程序。WMI 脚本支持所有 Microsoft 编程语言技术。

就象对管理应用程序一样，WMI COM 界面构成了提供程序主要的 API。然而，不象作为 COM 客户机的管理应用程序，提供程序是 COM 或分布式 COM（DCOM）服务器（也就是说，提供程序实现 WMI 交互的 COM 对象）。一个 WMI 提供程序的可能体现包括加载到 WMI 管理器进程和独立的 Win32 应用程序或 Win32 服务里的 DLL。Microsoft 包括许多表示来自众所周知的数据源的数据的内建提供程序，例如性能 API、注册表、Event Manager、Active Directory、SNMP 和 Windows 驱动程序模型（Windows Driver Model，WDM）设备驱动程序。WMI SDK 让开发者开发第三方 WMI 提供程序。

5.3.2 提供程序

WBEM 的内核是 DMTF 设计的 CIM 的规格说明。CIM 从系统管理角度指定了管理系统如何表达计算机中的任何东西乃至计算机上的应用程序或设备。提供程序的开发者使用 CIM 表示那些组成开发者想管理的应用程序各部分的组件。开发者使用 Managed Object Format（MOF）语言实现 CIM 的表示。

除了定义表示对象的类以外，提供程序必须将 WMI 作为对象的接口。WMI 按照提供程序提供的接口特性对提供程序进行分类。表 5-8 列出了 WMI 提供程序分类。注意一个提供程序能实现一种或多种特性；因此，一个提供程序既可以是类提供程序，又可以是事件提供程序。为了明确阐明表 5-8 中的特性定义，让我们看一个实现几种特性的提供程序。事件日志提供程序定义了几个对象，包括事件日志计算机（Event Log Computer），事件日志记录（Event Log

Record）和事件日志文件（Event Log File），事件日志提供程序是一个类提供程序，因为它通过使用类来定义这些对象并且必须将这些类定义传给 WMI。这个提供程序也是一个实例提供程序，因此它能为它的几个类定义多个实例。事件日志提供程序为之定义多个实例的类是事件日志文件类；事件日志提供程序为系统的每个事件日志定义这个类的一个实例（也就是系统事件日志、应用事件日志和安全事件日志）。

表 5-8 提供程序分类

分类	描 述
类	能提供、修改、消除和枚举提供程序相关的类。也能支持查询处理。Active Directory 是一个较少见的类提供程序的例子
实例	能提供、修改、消除和枚举系统和提供程序相关的类的实例。一个实例代表一个管理对象，也能支持查询处理
属性	能提供和修改单独的对象属性值
方法	为提供程序相关的类提供方法
事件	产生事件通知
事件用户	将物理用户映射为逻辑用户以支持事件通知

事件日志提供程序定义实例数据并且让管理应用程序枚举这些记录。为了让管理应用程序使用 WMI 备份和恢复事件日志文件，事件日志提供程序为事件日志文件对象实现备份和恢复方法。这样做使得事件日志提供程序成为了一个方法提供程序（Method Provider）。最后，无论什么时候当新记录被写到事件日志中时，管理应用程序都能注册来接收通知。这样，当事件日志提供程序使用 WMI 事件通知来告诉 WMI 事件日志记录已经收到时，它就作为一个事件提供程序。

5.3.3 通用信息模型和管理对象格式语言

CIM 遵循的是诸如 C++ 和 Java 这样的面向对象语言的做法，在该语言内部，建模者以类的方式来表达设计。采用类的方法让开发者能够利用具有强大的继承和组合能力的建模技术。子类能继承父类的属性、添加自己的特性，并且能重载它们从父类继承的特性。从其他类派生（derive）的类可以继承特性。类也能组合：开发者也能创建包含其他类的类。

DMTF 提供众多的类作为 WBEM 标准的一部分。这些类是 CIM 的基本语言，并且适用于所有管理范围的对象。这些类是 CIM 内核模型的一部分。CIM _ ManagedSystemElement 是内核类的例子。这个类包含一些标识物理组件诸如硬件设备和逻辑组件诸如进程和文件的基本特性。这些特性包括标题、描述、安装日期和状态。这样，CIM _ LogicalSystemElement 和 CIM _ PhysicalElement 这两个类继承了 CIM _ ManagedSystemElement 类的属性。这两个类也是 CIM 内核模型的一部分。WBEM 标准称这些类为抽象的类，因为它们作为其他类继承的类单独存在（即一个抽象的类不存在对象实例）。因此你可以将抽象的类看作是定义了其他类使用的属性的模板。

类的另一个分类表示特定于管理范围但与特定的实现不相关的对象。这些类构成通用模型，并且被认为是内核模型的扩展。CIM _ FileSystem 类是一个通用模型类的例子，它继承了 CIM _ LogicalElement 的属性。因为实际上每个操作系统，包括 Windows 2000、Linux 和其他各种 UNIX，都依赖基于文件系统的结构化存储，所以 CIM _ FileSystem 类是通用模型的一个合适的

组成部分。

类的最后一个分类包括通用模型的技术相关的增加部分。Windows 2000 定义了一大批这样的类来表示特定于 Win32 环境的对象。因为所有的操作系统都在文件里存储数据，所以 CIM 通用模型包括 CIM _ LogicalFile 类。CIM _ DataFile 类继承了 CIM _ LogicalFile 类，并且 Win32 为 Win32 文件类型增加了 Win32 _ PageFile 类和 Win32 _ ShortCutFile 类。

事件日志提供程序广泛地使用了继承。图 5-19 显示了 WMI CIM 工作室的外观，这是一个用 WMISDK 安装的类浏览器。(Microsoft 在 MSDN 软件和桌面 SCK 提供 WMI SDK)。你可以看到事件日志提供程序在何处依赖提供程序的 Win32 _ NTEventLogFile 类里的继承，这个类从 CIM _ DataFile 派生。事件日志文件是数据文件，这些文件有附加的事件日志相关的属性，诸如日志文件名（Logfile Name）和文件包含的记录数的计数（NumberOfRecords）。类浏览器显示的树揭示了 Win32 _ NTEventlogfile 基于几个继承层次之上，其中 CIM-DataFile 派生于 CIM _ LogicalFile，CIM _ LogicalFile 派生于 CIM _ LogicalElement，而 CIM _ LogicalElement 派生于 CIM _ ManagedSystemElement。

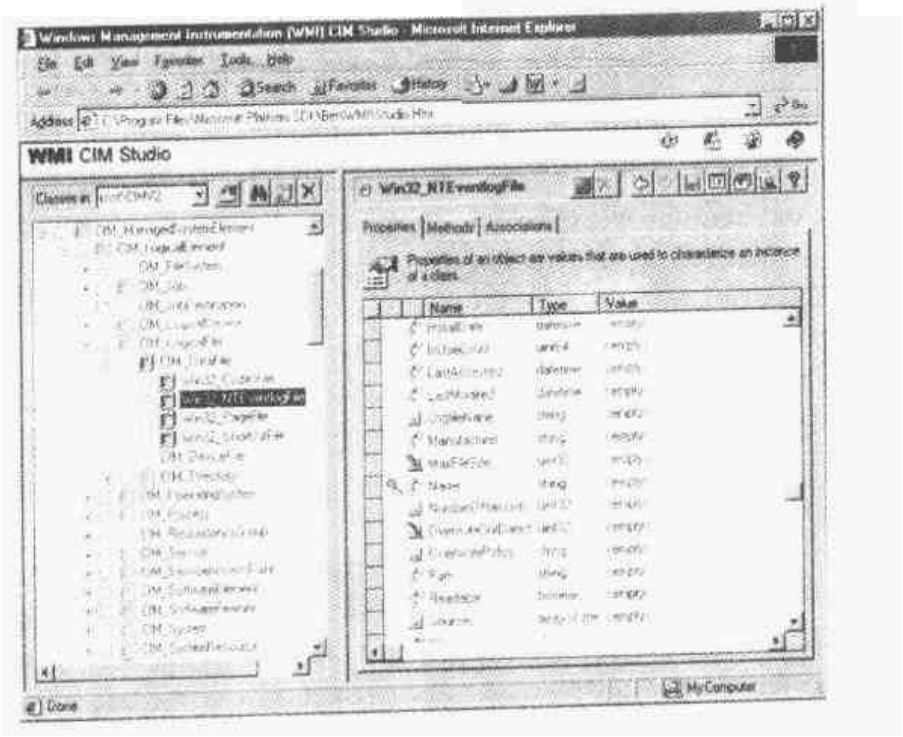


图 5-19 WMI CIM 工作室

如前所述，WMI 提供程序开发者用 MOF 语言编写他们的类。下面的输出显示了事件日志提供程序的 Win32 _ NTLogEventFile 的定义，它是在图 5-19 中选择的。请注意，图 5-19 列表中右边窗格的属性和下面的 MOF 文件里的那些属性定义之间的联系。CIM 工作室使用黄色箭头标记类继承的那些属性。因此，你看不见在 Win32 _ NTEventlogFile 的定义中指定的那些属性。

```
[dynamic,provider("MS_NT_EVENTLOG_PROVIDER"),Locale(1033),
UUID("{8502C57B-5FBB-11D2-AAC1-006008C78BC7}")]
class Win32_NTEventlogFile : CIM_DataFile
{
  [read] string LogfileName;
  [read,write] uint32 MaxFileSize;
```

```
    [read] uint32 NumberOfRecords;
    [read,volatile,ValueMap{"0", "1..365", "4294967295"}]
      string OverWritePolicy;
    [read,write,Units("Days"),Range("0-365 | 4294967295")]
      uint32 OverwriteOutDated;
    [read] string Sources[];
    [implemented,Privileges{"SeSecurityPrivilege", "SeBackupPrivilege"}]
      uint32 ClearEventlog([in] string ArchiveFileName);
    [implemented,Privileges{"SeSecurityPrivilege", "SeBackupPrivilege"}]
      uint32 BackupEventlog([in] string ArchiveFileName);
};
```

值得评论的一个术语是动态性（dynamic），这是前面输出的 MOF 文件中显示的 Win32_NTEventlogFile 类的一个描述性符号。动态性（Dynamic）意味着无论什么时候管理应用程序询问对象的属性时，WMI 基础设施就向 WMI 提供程序查找和那个类的对象相关的属性值。一个静态类在 WMI 库中；WMI 基础设施引用这个库得到这些值而不是向提供程序请求这些值。因为更新库是一个相对复杂的操作，对于那些经常有改变的属性来说动态性提供程序更有效。

在 MOF 中建立了类之后，WMI 开发者就能通过几种方法为 WMI 提供类定义。WDM 提供程序开发者将一个 MOF 文件编译成一个二进制的 MOF（BMF）文件——一个比 MOF 文件更紧凑的二进制的描述，并且将 BMF 文件传送给 WDM 基础设施。对提供程序来说另一个方法是编译 MOF 并且使用 WMI COM API 将这个类定义传给 WMI 基础设施。最后，提供程序可以利用 MOF 编译器（Mofcomp.exe）工具直接将一个编译过的类表示传给 WMI 基础设施。

5.3.4 WMI 名字空间

类定义了对象的属性，而对象是系统上的类的实例。WMI 使用包括几个子名字空间的名字空间，WMI 有层次地使用它们去组织对象。在应用程序访问名字空间中的对象之前，管理应用程序必须与名字空间建立连接。

WMI 将名字空间根目录命名为 root。WMI 的安装全部都有四个事先定义好的位于根目录下的名字空间：CIMV2、Default、Security 和 WMI。在一些名字空间内还有其他的名字空间。例如 CIMV2 包含作为子名字空间的 Application 和 MS_409 名字空间。提供程序有时定义它们自己的名字空间；你在 Windows 2000 根目录下可以看见 WMI 名字空间（由 windows 设备驱动程序 WMI 提供程序定义）。

不像文件系统名字空间包含多层次的目录和文件，WMI 空间只有一个层次深度。与文件系统使用名字相反，WMI 使用对象属性来标识对象。它将对象属性定义为标识对象的键。管理应用程序用键名指定类名以便在名字空间内定位相关的对象。因此，一个类的每个实例必须被它的键值唯一地标识。例如事件日志提供程序使用 Win32_NTLogEvent 类来代表事件日志里的数据。这个类有两个键：Logfile 是字符串；另一个为无符号整数 RecordNumber。向 WMI 查询事件日志记录实例的管理应用程序会从标识记录的提供程序键对中获得实例。应用程序使用下面对象路径例子所示的语法引用记录。

```
\\PICKLES\CIMV2:Win32_NTLogEvent.Logfile="Application",
                                    RecordNumber="1"
```

名字里第一部分（\\PICKLES）确定了对象所在的计算机，第二部分（\CIMV2）是对象所在的名字空间。类名跟在冒号后面，句号后面是键名和它们的相关值。逗号将这些键值分隔开。

WMI提供界面让应用程序枚举特殊类中所有的对象或查询并返回符合查询条件的类的实例。

5.3.5 类关联

许多对象类型在多方面是相关联的，例如计算机有处理器、软件、操作系统和活动的进程等等。WMI让提供程序创建关联的类来代表两个不同类之间的一个逻辑连接。这些关联类将一个类与另一个类相联，所以这些类只有两个属性：一个类名和Ref修饰符。下面的输出内容显示了一个关联，在这个关联里，事件日志提供程序的MOF文件将WIN32_NTLOGEVENT类与Win32_ComputerSystem类相关联，给定一个对象，一个管理应用程序就能查询相关联的对象。这样，提供程序定义了对象的层次。

```
[dynamic,provider("MS_NT_EVENTLOG_PROVIDER"),
 ThumPrivileges{"SeSecurityPrivilege"},Locale(1033),
 UUID("{8502C57F-5FBB-11D2-AAC1-006008C78BC7}"),
 Associator : ToInstance]
class Win32_NTLogEventComputer
{
  [key,read] Win32_ComputerSystem Ref Computer;
  [key,read] Win32_NTLogEvent Ref Record;
};

Instance of __Win32Provider as $EventProv
{
  Name = "MS_NT_EVENTLOG_EVENT_PROVIDER";
  ClsId = "{F55C5B4C-517D-11d1-AB57-00C04FD9159E}";
};
```

图5-20给出了显示CIMV2名字空间根的WMI对象浏览器（WMI SDK包含的另一个开发工具）。Win32系统组件典型地将它们的对象放在CIMV2名字空间里。对象浏览器首先查找Win32_ComputerSystem对象实例DSOLOMON，这是个代表本计算机的对象。然后对象浏览器获得Win32_ComputerSystem相关联的对象并且将它们显示在DSOLOMON下面。对象浏览器用户界面用双箭头文件夹图标显示关联对象。关联类类型对象显示在文件夹下面。

在对象浏览器中你能看见对象日志提供程序的关联类Win32_NTLogEventComputer在DSOLOMON下面而Win32_NTLogEvent类的大量实例都存在。为证实MOF文件定义了Win32_NTLogEventComputer类将Win32_ComputerSystem类与Win32_NTLogEvent类相关联，请查阅前面的输出。选择对象浏览器中的一个Win32_NTLogEvent的实例，在右手窗格中的Properties标签下显示那个属性。微软试图让对象浏览器帮助WMI开发者检查它们的对象，但是管理应用程序将执行同样的操作，并且更加灵活地显示属性或者显示收集到的信息。

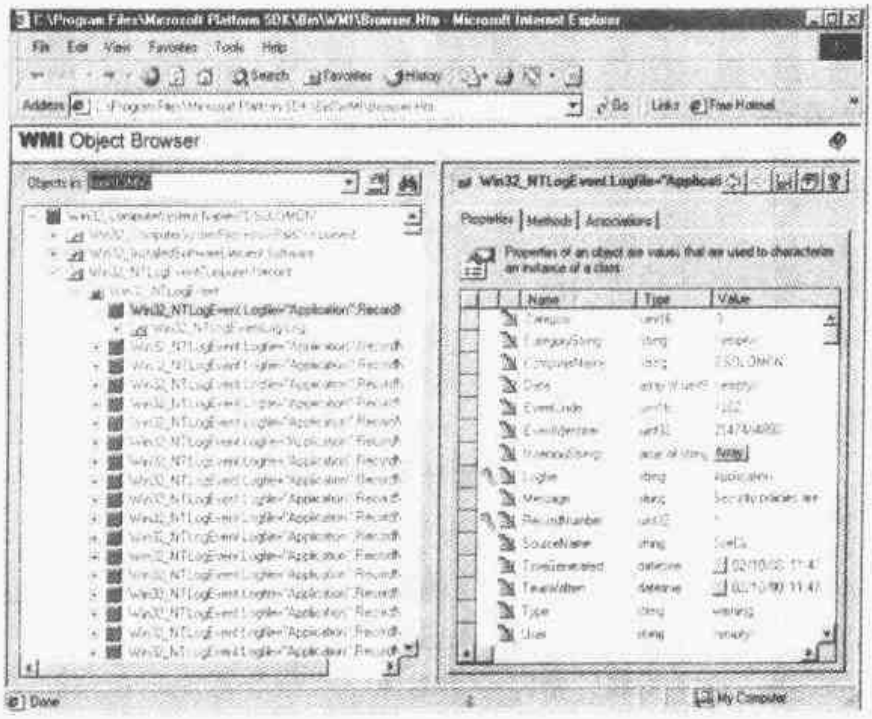


图 5-20 WMI 对象浏览器

5.3.6 WMI 实现

WMI 基础结构主要由 \ winnt \ System32 \ Wbem \ Winmgmt.exe 文件实现。这个文件作为 Win32 服务运行，当管理应用程序或 WMI 提供程序第一次试图访问 WMI API 时，Windows 2000 SCM 开启这个服务。大多数 WMI 组件缺省地驻留在 \ Winnt \ System32 和 \ Winnt \ System32 \ Wbem 里，包括 Win32 MOF 文件、内建提供程序 DLL 和管理应用程序 WMI DLL。查看 \ Winnt \ System32 \ Wbem 目录，你会发现 Ntevt.mof，这是个对象日志提供程序 MOF 文件。你也许会看到 Ntevt.dll 文件，这是事件日志提供程序 DLL 文件，它由 Winmgmt.exe 安装。

\ Winnt \ System32 \ Wbem 下的目录存储了存储库、日志文件以及第三方 MOF 文件。WMI 以文件 \ Winnt \ System32 \ Wbem \ Repository \ Cim.rep 的方式实现了该库，并命名为 CIMOM。Winmgmt 承认了许多与此存储库有关的注册表设置（包括各种内部性能参数，例如 CIMOM 备份位置和区间），这些设置由此库的 HKLM \ SOFTWARE \ Microsoft \ WBEM \ CIMOM 注册表键来存储。

设备驱动程序使用特殊的接口将数据提供给 WMI，并且从 WMI 那儿接收称为 WMI 系统控制命令的命令。这些接口是 WMI 的一部分，这在第 9 章加以解释。因为这些接口是平台无关的，所以它们位于 \ root \ WMI 名字空间。

5.3.7 WMI 安全性

WMI 在名字空间层实现安全。如果管理应用程序成功地与名字空间建立连接，这个应用程序就能浏览并访问名字空间里的所有对象的属性。系统管理员能利用 WMI Control 应用程序控制用户访问名字空间的权限。要启动 WMI Control 应用程序，首先从 Start 菜单选择 Programs、

Adminastrative Tools 和 Computer Management。然后打开 Service And Application 子菜单。右击 WMI Control，选择 Properties，启动 WMI Control Properties 对话框，如图 5-21 所示。为给名字空间配置安全性，点击 Security 标签，选择名字空间并点击 Security。WMI Control Properties 对话框中其他的标签键让你更改注册表存储的性能和备份设置。

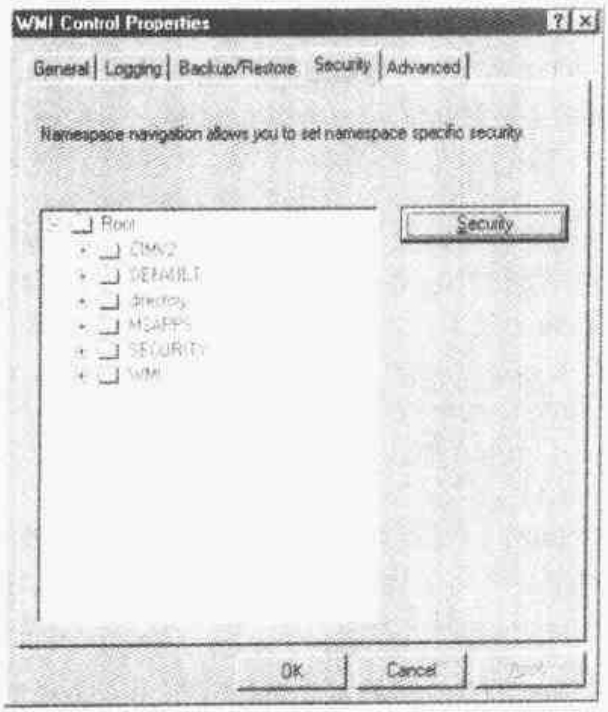


图 5-21 WMI 安全属性

5.4 小结

迄今为止，我们已检查了 Windows 2000 整体结构和内核系统机制，这些系统机制使系统启动、运行和关闭。在此基础上，我们准备从进程和线程开始讨论每一个执行程序组件。

第 6 章　进程、线程和作业

本章将解释在 Windows 2000 中处理进程和线程的数据结构与算法。第一节着重讲述组成进程的内部结构。第二节简要讲述创建进程（及其初始线程）的步骤。接下来的部分将讨论线程的内部和线程调度。本章还包括对作业对象的描述。

另外，本章也提及了与本章有关的性能计数器和内核变量。虽然本书不是微软 Win32 编程手册，但也列出了与进程和线程有关的 Win32 函数，这样你就可以继续得到有关使用它们的额外信息。

在 Windows 2000 中，由于进程和线程涉及到许多组件，所以本章提及了大量的术语和数据结构（例如工作集、对象和句柄、系统内存堆等等），而它们在本书的其他部分有详细的解释。为了能全面理解本章内容，你要熟悉本书的第 1、2 章所解释的术语和概念，例如进程和线程的区别、Windows 2000 虚拟地址空间布局、用户模式和内核模式的区别。

6.1　进程的本质

本节将描述关键的 Windows 2000 进程数据结构，还列出了关键的内核变量、性能计数器和与进程有关的函数和工具。

6.1.1　数据结构

每个 Windows 2000 进程都由一个执行程序进程（EPROCESS）块表示。EPROCESS 块不仅包括进程的许多相关属性，还包括并指向许多其他相关的数据结构。例如，每个进程都有一个或多个由执行程序线程（ETHREAD）块表示的线程（线程数据结构将在“线程的本质”一节中解释）。除了进程环境块（PEB）存在于进程地址空间中以外（因为它包含用户模式代码更改的信息），EPROCESS 块及其相关的数据结构都存在于系统空间中。

除了 EPROCESS 块，Win32 子系统进程（Csrss）为执行 Win32 程序的每个 WINDOWS 2000 进程维持一个并行结构。同样，Win32 子系统的内核模式部分（Win32k. sys）有一个每进程（per – process）数据结构，它在线程第一次调用在内核模式实现的 Win32 USER 或 GDI 函数时被创建。

图 6-1 是进程和线程的数据结构简图。本章将详细讨论图中的每个数据结构。

首先让我们着重讨论进程块（我们将在 6.3 节“线程的本质”一节中讨论线程块）。图 6-2 显示了 EPROCESS 的关键字段。

表 6-1 较为详细地解释了上述实验中的某些字段，并且包括本书中可以获得更多信息的章节位置。正如前面所说的，进程和线程是 Windows 2000 整体的一部分，讨论它们时不涉及系统的其他部分是不可能的。然而，为了使本章更为紧凑，与之相关的一些问题（例如，内存管理、安全、对象和句柄）将在其他章节中讨论。

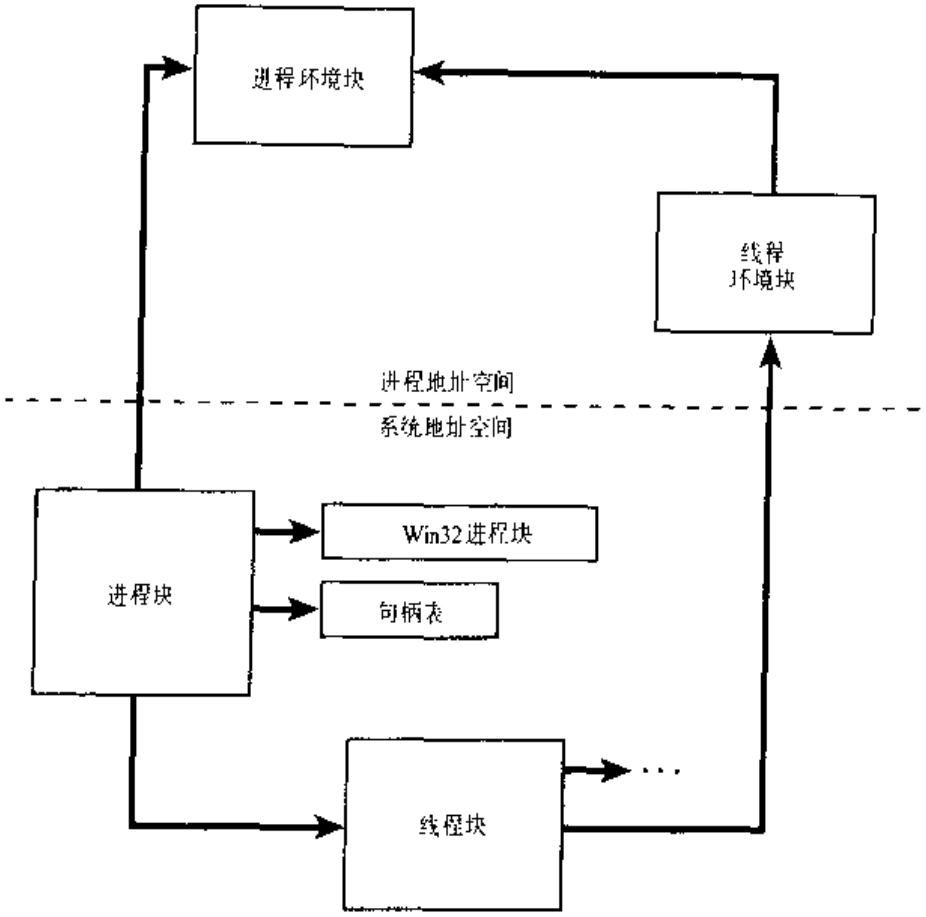


图 6-1 与进程和线程相关的数据结构

实验：显示 EPROCESS 块的格式

为了列出组成 EPROCESS 块的大多数字段及其十六进制偏移量，在内核调试器中键入！processfields（要了解内核调试器的设置和使用，请参阅第 1 章关于内核调试器的内容）。输出如下：

```
kd> !processfields
 EPROCESS structure offsets:
    Pcb:                          0x0
    ExitStatus:                   0x6c
    LockEvent:                    0x70
    LockCount:                    0x80
    CreateTime:                   0x88
    ExitTime:                     0x90
    LockOwner:                    0x98
    UniqueProcessId:              0x9c
    ActiveProcessLinks:           0xa0
    QuotaPeakPoolUsage[0]:        0xa8
    QuotaPoolUsage[0]:            0xb0
    PagefileUsage:                0xb8
    CommitCharge:                 0xbc
    PeakPagefileUsage:            0xc0
    PeakVirtualSize:              0xc4
    VirtualSize:                  0xc8
    Vm:                           0xd0
    DebugPort:                    0x120
```

```
ExceptionPort:                  0x124
ObjectTable:                    0x128
Token:                          0x12c
WorkingSetLock:                 0x130
WorkingSetPage:                 0x150
ProcessOutswapEnabled:          0x154
ProcessOutswapped:              0x155
AddressSpaceInitialized:        0x156
AddressSpaceDeleted.            0x157
AddressCreationLock.            0x158
ForkInProgress:                 0x17c
VmOperation:                    0x180
VmOperationEvent:               0x184
PageDirectoryPte:               0x1f0
LastFaultCount:                 0x18c
VadRoot:                        0x194
VadHint:                        0x198
CloneRoot:                      0x19c
NumberOfPrivatePages:           0x1a0
NumberOfLockedPages:            0x1a4
ForkWasSuccessful:              0x1a2
ExitProcessCalled:              0x1ad
CreateProcessReported:          0x1ab
SectionHandle:                  0x1ac
Peb:                            0x1b0
SectionBaseAddress:             0x1b4
QuotaBlock:                     0x1b8
LastThreadExitStatus:           0x1bc
WorkingSetWatch:                0x1c0
InheritedFromUniqueProcessId:   0x1c8
GrantedAccess:                  0x1cc
DefaultHardErrorProcessing      0x1d0
LdtInformation:                 0x1d4
VadFreeHint:                    0x1d8
VdmObjects:                     0x1dc
DeviceMap:                      0x1e0
ImageFileName[0]:               0x1fc
VmTrimFaultValue:               0x20c
Win32Process:                   0x214
Win32WindowStation:             0x1c4
```

命令！processfields 显示的是进程块的格式而不是内容（！process 命令实际上转储进程块的内容，在本章第 172 页，包括这个命令的输出的带注释的例子）。虽然有些字段名是不需要加以说明的，但输出并没有给出字段的数据类型，也没有显示包含在 EPROCESS 块中或由 EPROCESS 块指向的结构的格式（例如内核进程块、配额块等等）。然而，通过检测偏移量，至少可以得到字段的长度（提示：那些长度为 4 个字节并涉及到某些其他结构的字段可能是指针）。

你也可以使用！strct 命令（在第二个内核调试扩展库 Kdex2x86.dll）显示进程块的格式。这个命令显示每个字段的值和它的数据类型（而！processfields 命令只显示部分字段并不显示数据类型信息），部分输出如下：

```
kd> !kdex2x86.struct eprocess
Loaded kdex2x86 extension DLL
struct   _EPROCESS (sizeof=648)
+000 struct   _KPROCESS Pcb
+000    struct   _DISPATCHER_HEADER Header
+000       byte     Type
+001       byte     Absolute
+002       byte     Size
+003       byte     Inserted
+004       int32    SignalState
+008       struct   _LIST_ENTRY WaitListHead
+008          struct   _LIST_ENTRY *Flink
+00c          struct   _LIST_ENTRY *Blink
+010    struct   _LIST_ENTRY ProfileListHead
+010       struct   _LIST_ENTRY *Flink
+014       struct   _LIST_ENTRY *Blink
+018    uint32   DirectoryTableBase[2]
+020    struct   _KGDTENTRY LdtDescriptor
+020       uint16   LimitLow
+022       uint16   BaseLow
+024       union    __unnamed9 HighWord
+024          struct   __unnamed10 Bytes
+024             byte     BaseMid
+025             byte     Flags1
+026             byte     Flags2
+027             byte     BaseHi
+024          struct   __unnamed11 Bits
+024             bits0-7 BaseMid
+024             bits8-12 Type
+024             bits13-14 Dpl
+024             bits15-15 Pres
+024             bits16-19 LimitHi
+024             bits20-20 Sys
+024             bits21-21 Reserved_0
+024             bits22-22 Default_Big
+024             bits23-23 Granularity
+024             bits24-31 BaseHi
+028    struct   _KIDTENTRY Int21Descriptor
+244    int32    HighPart
+240    struct   __unnamed3 u
+240       uint32   LowPart
+244       int32    HighPart
+240    int64    QuadPart
+248 union    _LARGE_INTEGER OtherOperationCoun
+248    uint32   LowPart
+24c    int32    HighPart
+248    struct   __unnamed3 u
+248       uint32   LowPart
+24c       int32    HighPart
+248    int64    QuadPart
+250 union    _LARGE_INTEGER ReadTransferCount
+250    uint32   LowPart
+254    int32    HighPart
+250    struct   __unnamed3 u
+250       uint32   LowPart
```

```
+254        int32    HighPart
+250    int64    QuadPart
+258 union    _LARGE_INTEGER WriteTransferCount
+258    uint32   LowPart
+25c    int32    HighPart
+258    struct   __unnamed3 u
+258       uint32   LowPart
+25c       int32    HighPart
+258    int64    QuadPart
+260 union    _LARGE_INTEGER OtherTransferCount
+260    uint32   LowPart
+264    int32    HighPart
+260    struct   __unnamed3 u
+260       uint32   LowPart
+264       int32    HighPart
+260    int64    QuadPart
+268 uint32   CommitCharge_imit
+26c uint32   CommitChargePeak
+270 struct   _LIST_ENTRY ThreadListHead
+270    struct   _LIST_ENTRY *Flink
+274    struct   _LIST_ENTRY *Blink
+278 struct   _RTL_BITMAP *VadPhysicalPagesBitMa
+27c uint32   VadPhysicalPages
+280 uint32   AweLock
```

表 6-1 EPROCESS 块的内容

元素	目的	附加参考
内核进程(KPROCESS)块	公用调度程序对象头，指向进程页面目录的指针，列出属于此进程的内核线程（KTHREAD）块列表、缺省基本优先级、时间片、相似性掩码以及用于进程中线程的内核模式和用户模式总时间	线程调度
进程标识	唯一的进程 ID，创建的进程 ID，正在运行的映像的名字，进程正在运行的窗口站	
配额块	非页交换区，页交换区以及页面文件的使用加上当前和峰值进程的非页交换区和页交换区的使用限制。（注：很多进程共享这个结构；所有的系统进程指向单个的系统范围内的缺省配额块；在交互会话中所有进程共享一个由 Winlogon 建立的单一配额块）	
虚拟地址描述符(VAD)	描述存在于进程中的地址空间部分的状态的一系列数据结构	内存管理（第 7 章）
工作集信息	指向工作集列表（MMWSL 结构）的指针；当前的、峰值的、最小的和最大的工作集大小，上次裁减时间；页面错误计数；内存优先级；交换出标记；页面错误历史	内存管理（第 7 章）
虚拟内存信息	当前和峰值虚拟值，页面文件使用，用于进程页面目录的硬件页面表项	内存管理（第 7 章）

（续）

元 素	目 的	附加参考
异常本机过程调用（LPC）端口	当进程中的一个线程导致异常时，进程管理器发送消息的进程间通信通道	本机过程调用（第3章）
调试LPC端口	当进程中的一个线程引起调试事件时，进程管理器发送消息的进程间通信通道	本机过程调用（第7章）
访问令牌（ACCESS-TOKEN）	描述这个进程的安全配置的执行程序对象	安全（第8章）
句柄表	每个进程句柄表的地址	对象句柄（第3章）
设备映射	解析引用的设备名的对象目录地址（支持多用户）	对象管理器（第3章）
进程环境块（PEB）	映像信息（基地址、版本号、模块列表）、进程堆信息和线程本机存储使用（注：在PEB后开始的第一个字节是进程堆的指针）	
Win 32子系统进程块（W32PROCESS）	Win 32子系统的内核模式组件所需要的进程细节	

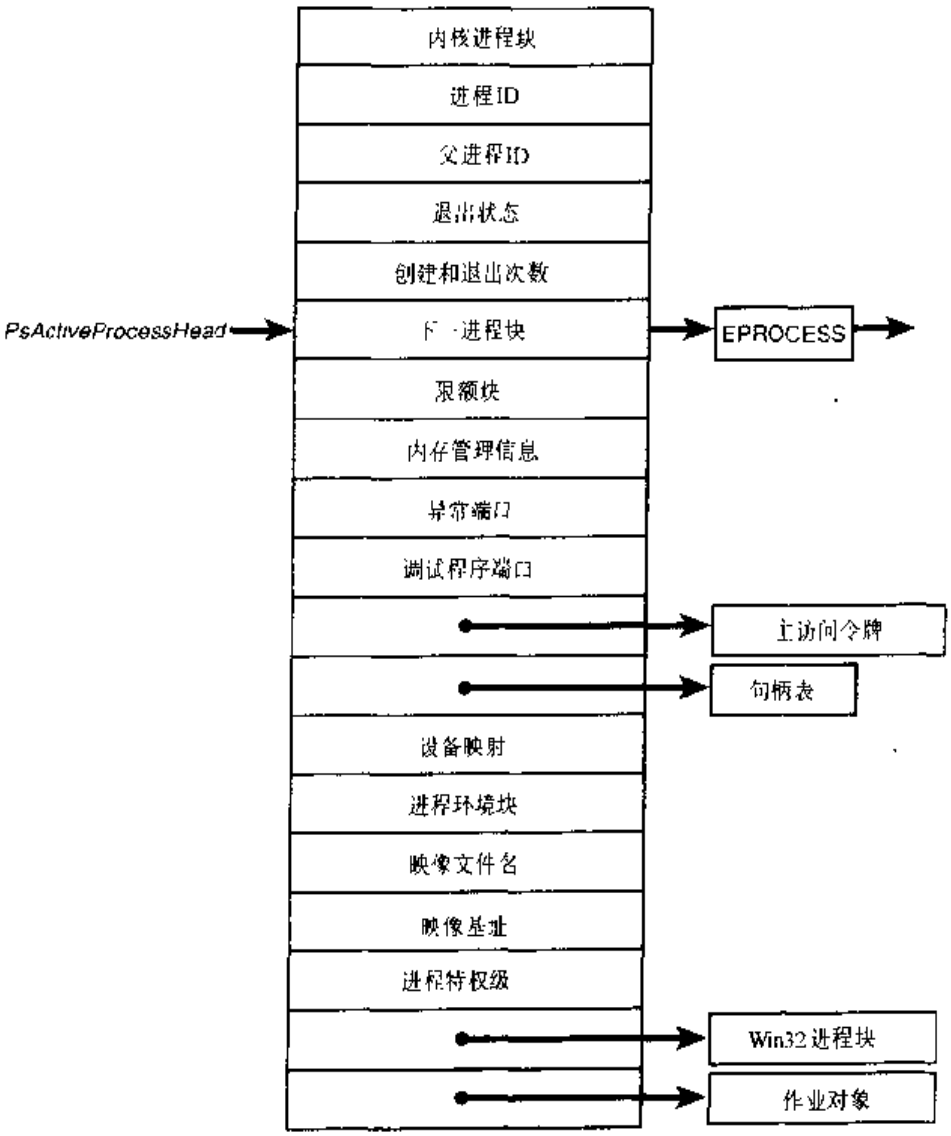


图6-2 执行程序进程块的结构

内核进程块（KPROCESS）（EPROCESS 块的一部分）和 EPROCESS 块指向的进程环境块（PEB）包含了关于进程对象的其他信息。图 6-3 显示了 KPROCESS 块（有时称作 PCB 或进程控制块）。它包含了 Windows 2000 内核调度线程所需的基本信息。（在第 7 章介绍页面目录，在本章后面将介绍内核线程块）。

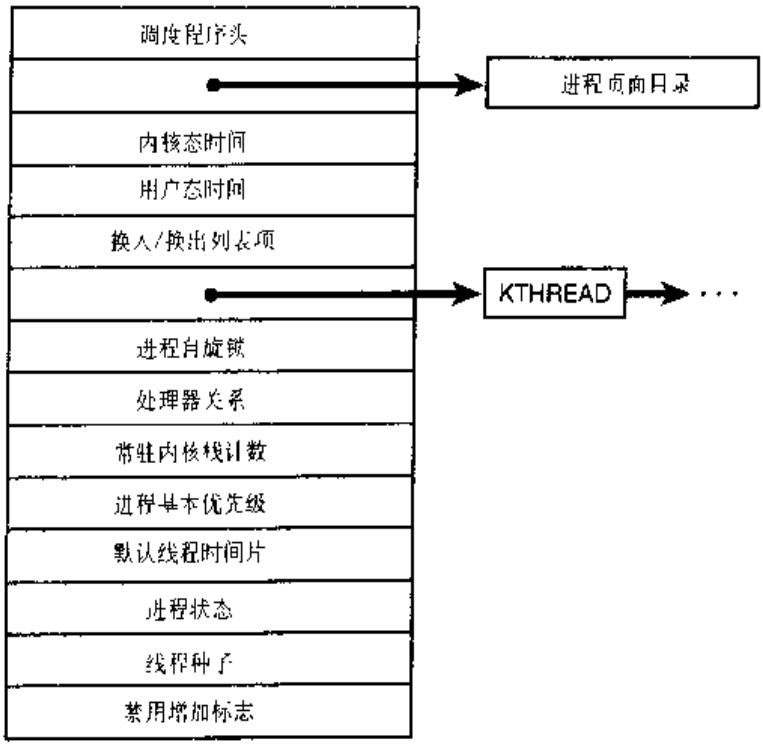


图 6-3 内核进程块结构

存在于用户进程地址空间的 PEB 包含映像加载程序、堆管理器和其他需要在用户模式下可以写的 Win32 系统 DLL 的信息。（EPROCESS 和 KPROCESS 块只能从内核模式访问）。PEB 总是映射到地址 0x7FFDF000 处。图 6-4 说明了 PEB 的基本结构，它将在本章稍后详细解释。

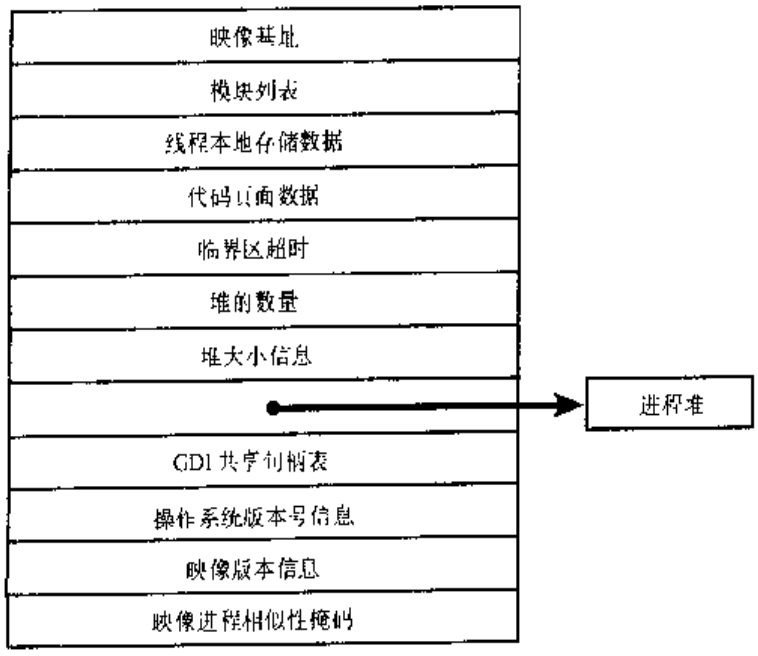


图 6-4 进程环境块字段

实验：检查 PEB

你可以利用内核调试器的！peb 命令显示 PEB 结构，下面的例子使用 LiveKd 显示 LiveKd 进程的 PEB。

```
kd> !peb
PEB at 7FFDF000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            No
    ImageBaseAddress:         00400000
    Ldr.Initialized: Yes
    Ldr.InInitializationOrderModuleList: 131f40 . 134b98
    Ldr.InLoadOrderModuleList: 131ec0 . 134b88
    Ldr.InMemoryOrderModuleList: 131ec8 . 134b90
        00400000 C:\nt\livekd.exe
        77F80000 C:\WINNT\System32\ntdll.dll
        77920000 C:\WINNT\system32\IMAGEHLP.dll
        78000000 C:\WINNT\system32\MSVCRT.DLL
        77EB0000 C:\WINNT\system32\KERNEL32.dll
        77E10000 C:\WINNT\system32\USER32.dll
        77F40000 C:\WINNT\system32\GDI32.DLL
        77DB0000 C:\WINNT\system32\ADVAPI32.dll
        77D40000 C:\WINNT\system32\RPCRT4.DLL
        72A00000 C:\WINNT\system32\DBGHELP.dll
    SubSystemData:     0
    ProcessHeap:       130000
    ProcessParameters: 20000
        WindowTitle:  '\nt\livekd'
        ImageFile:    'C:\nt\livekd.exe'
        CommandLine:  '\nt\livekd'
        DllPath:      'C:\nt;.;C:\WINNT\System32;
                       C:\WINNT\system;C:\WINNT;
                       C:\WINNT\system32;C:\WINNT;
                       C:\WINNT\system32\WBEM;
                       C:\Program Files\Support Tools\;
                       C:\Program Files\Resource Kit\'
        Environment:  0x10000
```

6.1.2 内核变量

表 6-2 列出了几个与进程相关的关键的系统全局变量。这些变量在本章稍后描述，创建进程的时候会涉及。

6.1.3 性能计数器

Windows 2000 维持一定数量的计数器，通过它们可以跟踪在系统中运行的进程。可以使用编程方式得到这些计数器，也可以使用“性能监视器”实用程序来查看它们。表 6-3 列出了与进程有关的性能计数器（第 7 章和第 9 章分别描述与内存管理和 I/O 有关的计数器除外）。

表 6-2 与进程有关的内核变量

变 量	类 型	说 明
PsActiveProcessHead	队列头	进程块表头
PsIdleProcess	EPROCESS	空闲进程块
PsInitialSystemProcess	指向 EPROCESS 的指针	指向包含系统线程的初始系统进程（进程 ID2）的进程块的指针
PspCreateProcessNotifyRoutine	32 位指针数组	指向在进程创建和删除时被调用的例程的指针的数组（最多 8 个）
PspCreateProcessNotifyRoutineCount	DWORD	注册的进程通知例程的数量
PspLoadImageNotifyRoutine	指针数组	指向在映像加载时调用的例程的指针数组
PspLoadImageNotifyRoutineCount	DWORD	注册的映像加载通知例程的计数
PspCidTable	指向 HANDLE-TABLE 的指针	用于进程和线程客户 ID 的句柄表

表 6-3 与进程有关的性能计数器

对象：计数器	功 能
Process：% Privileged Time	描述了在特定的时间内，进程中的线程运行在内核模式的时间的百分数
Process：% Processor Time	描述了在特定的时间内，进程中的线程使用的 CPU 时间的百分数。这个数是 % Privileged 和 % User Time 的和
Process：% User Time	描述了在特定的时间内，进程中的线程运行在用户模式的时间的百分数
Process：% Elapsed Time	描述自从进程被创建后，以秒为单位的总时间
Process：ID Process	返回进程 ID。此 ID 仅在进程存在时应用，因为进程 ID 是重复使用的
Process：Creating Process ID	返回创建进程的进程 ID。如果创建进程存在则该值不更新
Process：Thread Count	返回进程中的线程数
Process：Handle Count	返回进程中打开的句柄数

6.1.4 相关函数

为了提供参考，表 6-4 描述了用于进程的 Win32 函数。要得到详细信息，请参阅 MSDN 库中的 Win32 API 文档。

表 6-4 与进程相关的函数

函 数	描 述
CreateProcess	使用调用程序的安全标识，创建新的进程和线程
CreateProcessAsUser	使用指定的交替的安全令牌，创建新的进程和线程
CreateProcessWithLogonW	使用指定的交替的安全令牌，创建新的进程和线程，允许加载用户配置文件
OpenProcess	返回指定进程对象的句柄
ExitProcess	退出当前进程
TerminateProcess	终止进程
FlushInstructionCache	清除另一个进程的指令高速缓存
GetProcessTimes	得到另一个进程的时间信息，描述进程在用户模式和内核模式所用的时间
GetExitCodeProcess	返回另一个进程的退出代码，显示关闭这个进程的方法和原因
GetCommandLine	返回传递给进程的命令行字符串
GetCurrentProcessID	返回当前进程的 ID
GetProcessVersion	返回指定进程希望运行的 Windows 的主要和次要版本
GetStartupInfo	返回在 createprocess 时指定的 STARTUPINFO 结构的内容
GetEnvironmentStrings	返回环境块的地址
GetEnvironmentVariable	返回一个指定的环境变量
Get/SetProcessShutdownParameters	为当前进程定义关闭优先级和重试次数
GetGuiResource	返回 USER 和 GDI 句柄的计数

6.1.5 相关工具

一些工具可以用于查看（和修改）进程和进程的信息。这些工具包含在 Windows 2000 自身中，在 Windows 2000 资源工具包、平台软件开发工具包（SDK）和设备驱动程序工具包（DDK）中也有这些工具。可是问题在于，你不能使用一个工具得到所需要的全部信息——大多数信息都要通过多个工具才能得到，并且有时某个数据在不同的工具中被标识为不同的名字（并且有时会赋给不同的值）。为了帮助确定使用哪种工具来得到所需的基本进程信息，请参考表 6-5。这个表不是关于进程全部有用信息的全面列表——例如，你在第 7 章中可以找到用来收集内存管理信息的工具，但如果你需要基本的东西，可以找到它们。

表 6-5 进程相关工具

对象	Taskman	Perfmon	Pview	Pviewer	Qslice	Pmon	Pstat	Pulist	Tlist	KD! process
Process ID	√	√	√	√	√		√	√	√	√
Image Name	√		√	√	√	√	√	√	√	√
Total CPU Time	√	√				√				√
%CPU	√	√			√	√				

（续）

对象	Taskman	Perfmon	Pview	Pviewer	Qslice	Pmon	Pstat	Pulist	Tlist	KD! process
time										
Handle Count	√	√								
Thread Count	√	√	√		√	√	√	√		
View Priority Class	√		√	√						
% User Time		√				√				
% Privileged Time		√			√					
Total UserTime			√	√			√			√
Total Privileg - ed Time			√	√			√			√
Quota Limits			√	√						
Elapsed Time		√	√	√						√
Creating Process									√	
Current Directory									√	
Command Line		√							√	
Security ID								√		

实验：利用任务管理器（Task Manager）查看进程信息

内建的 Windows 2000 任务管理器提供了正在系统中运行的进程快速列表，但其中没有进程的线程。可以使用下面的三种方式之一来启动 Task Mannager：1）按 Ctrl - Shift - Esc；2）右键单击任务条并选择 Task Manager；3）按下 Ctrl + Alt + Del 并单击 TaskManager 按钮。一旦任务管理器启动，点击 Process 标签查看正在运行的进程列表。注意进程是用该实例所在的映像的名字来标识。不像 Windows 2000 中的其他对象，进程没有给予全局名字。为了显示其他细节，从 View 菜单中选择 Select 栏并选择要添加的列，如图 6-5 所示。

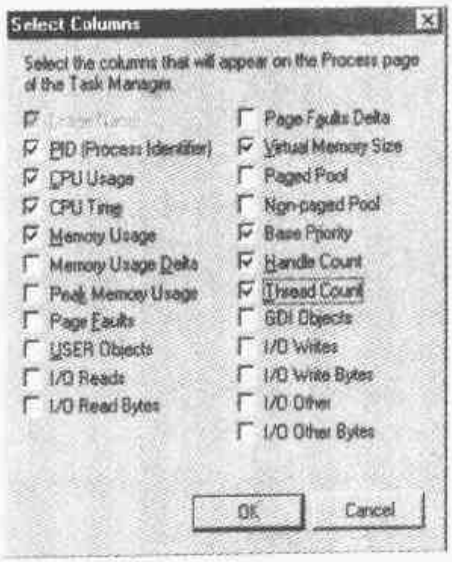


图 6-5　显示其他细节

虽然你在 Task Manager Processes 标签中所看到的很明显是一组进程列表，但 Application 标签中显示的不是这样清楚。Application 标签列出了在交互窗口站上所有桌面的顶层可见窗口（缺省地，只有两个桌面对象 – 你可以利用 Win32 CreateDesktop 函数创建更多的）。Status 栏表示拥有窗口的线程是否处于 Windows 消息等待状态。“Running”表示线程等待窗口输入；“Not Responding”表示线程没有等待窗口输入（例如，线程可能在运行或等待 I/O 或一些 Win32 同步对象），如图 6-6 所示。

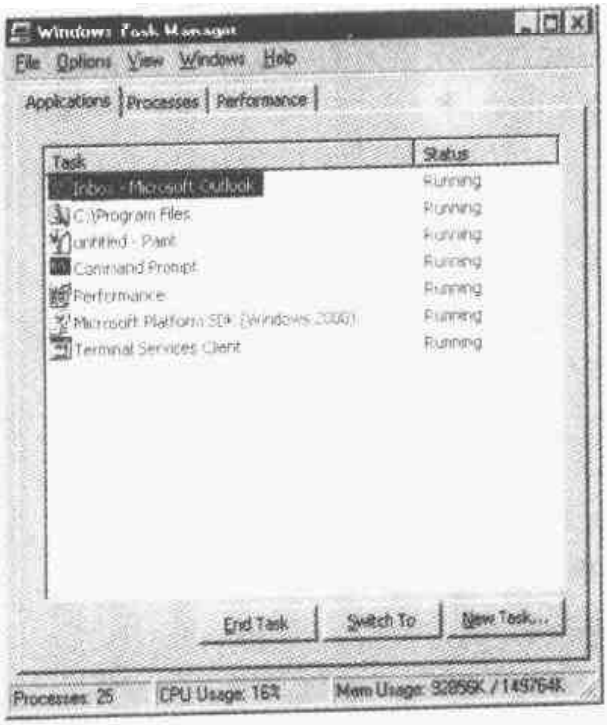


图 6-6　利用任务管理器查看进程信息

从 Application 标签，通过在任务名称上右击你可以将任务和拥有线程的进程关联，该线程拥有任务窗口，然后选择 Go To Process。

实验：查看进程树

多数工具不能显示进程的唯一属性是父进程或创建者进程的 ID。你可以利用 Performance Tool（或编程），通过查询创建进程 ID 来获取这个值。Windows 2000 Support Tool 命令 tlist/t 使用属性中的信息来显示进程树，该进程树表明进程和父进程的关系。下面例子是 tlist/t 的输出结果：

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

Tlist 通过缩进来表示父子关系。父进程不活动的进程是左对齐的，因为即使一个祖父进程存在，也没有办法找出它们的关系。Windows 2000 仅维持创建者进程 ID，而不是一个联接关系来指向创建者的创建者。

为了显示 Windows 2000 确实没有跟踪除了父进程 ID 的其他 ID，遵循如下几步骤：

1）打开命令行输入窗口。

2）键入 start cmd（启动第二个命令行输入程序）。

3）启动任务管理器。

4）切换到第二个命令行输入窗口。

5）键入 mspaint（它运行 Microsoft Paint）。

6）点击中间的（第二个）命令行窗口。

7）键入 exit（注意到 Paint 仍在运行）

8）切换到任务管理器。

9）点击 Application 标签。

10）右击命令行任务，选择 Go To Process。

11）点击 Cmd. exe 进程使它处于灰色高亮状态。

12）右击这个进程，选择 End Process Tree。

13）在 Task Manager Warning 消息框中点击 Yes。

第一个命令输入窗口将会消失，但你仍会看到 Paintbrush 窗口存在，因为它是你终止的

命令行进程的孙进程；因为中间的进程（Paintbrush 的父进程）已经结束，在父进程和孙进程之间就不存在任何关系。

实验：利用 QuickSlice 查看线程的活动状态

QuickSlice 给出了当前在系统中运行的进程所使用的系统时间之间的快速、动态的比例视图。联机时，条的红色部分显示用于内核模式的 CPU 时间，蓝色部分显示用户模式时间。（虽然在下图显示的是黑色和灰色，但联机中的条是蓝色和红色的。）显示在 QuickSlice 窗口中的所有条加起来应该为百分之百的 CPU 时间。要运行 QuickSlice，请单击 Start 按钮，选择 Run 并输入 Qslice. exe（假设 Windows 2000 资源工具在你的路径里）。例如，运行一个有大量图形的应用程序，如 Windows Paint（MSPAINT.EXE）。同时打开 QuickSlice 和 Paint，在 Paint 窗口中随便画一些东西。当你这样做的时候，MSPAINT. EXE 就会如图 6-7 所示：

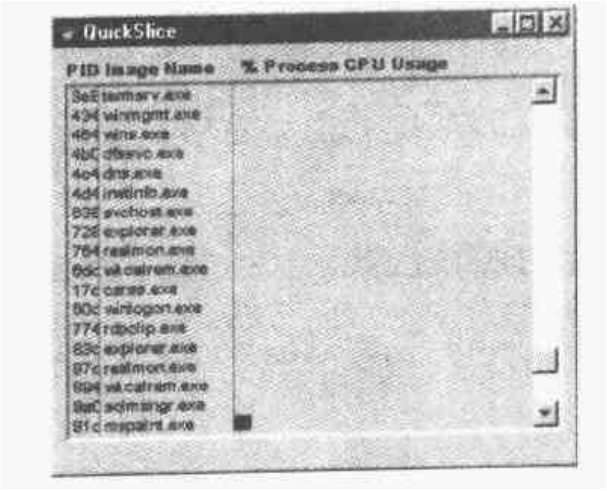


图 6-7 利用 Quick Slice 查看线程的活动状态

要获得关于进程中线程的其他信息，可以双击一个进程（在进程名称或颜色条上）。你可以看到这个进程中的线程和相应的被每个线程使用的 CPU 时间（不通过系统），见图 6-8。

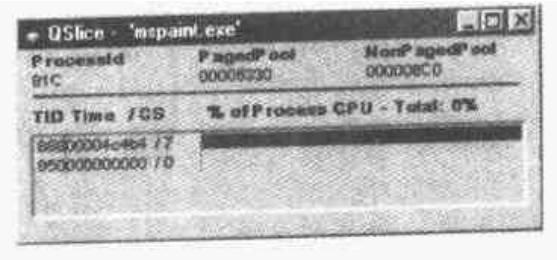


图 6-8 查看线程的其他信息

实验：用 Process Viewer 查看进程细节

Windows 2000 资源工具包自带的 Process Viewer（进程查看器）（Pviewer. exe）允许查

看有关正在运行的线程和进程以及终止进程和改变进程优先级的信息。可以使用这个工具在本机计算机或通过网络在远程 Windows 2000 计算机上查看进程。这个工具可以在 Platform SDK 和 Visual C++ 中找到。（在这两者中都被称作 Pview.exe）进程查看器在 windows 2000 支持工具帮助里有详细的文档，这里提供你一个有用的选项快速浏览。Process Viewer 的基本界面如图 6-9 所示。

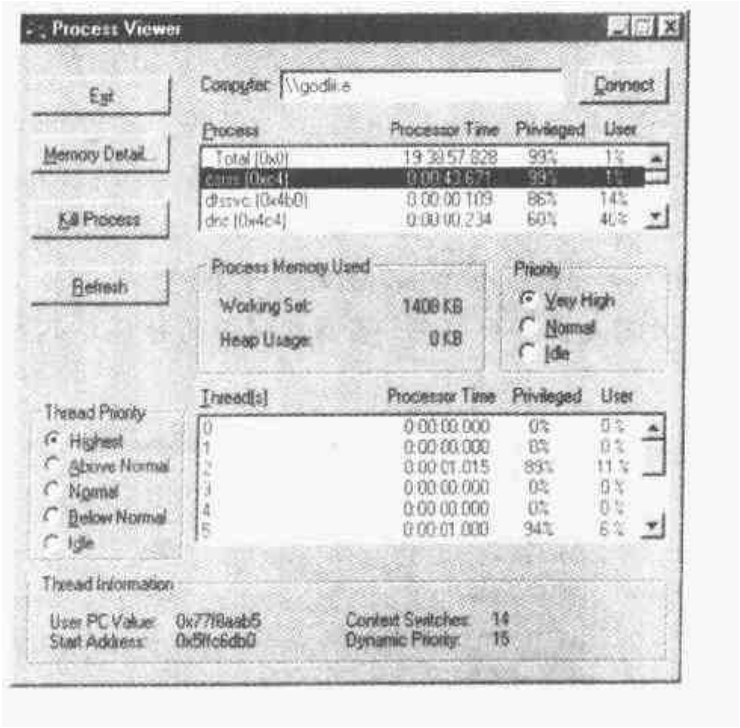


图 6-9 Process Viewer 的基本界面

下面是不同的选项的作用：

■ 计算机文本框显示当前显示的进程所属的计算机的名字。点击 Connect 按钮浏览另外一个计算机。

■ Memory Detail 按钮显示关于被选择的进程的内存管理信息，例如分配给该进程的内存大小，工作集的大小等等。

■ Kill Process 按钮终止所选择的进程。终止进程的时候要小心，因为这个时候进程没有机会完成任何清除工作。

■ Refresh 按钮刷新显示——除非你发出请求，否则 Process Viewer 不会更新信息。

■ 进程和线程列表框中的 Processor Time 栏显示进程或线程在它们创建以后所消耗的 Processor Time。

■ Priority 单选按钮集合调节所选进程的优先级别（不会显示实时优先权），而 Thread Priority 集合显示进程中的线程的相对线程优先级别。

■ 在窗口的底部，显示的是环境切换的数量、线程的动态优先权，起始地址和显示的当前 PC。

如果需要查看进程和线程的其他信息显示，运行 Tlist.exe（支持工具）、Pstat.exe（平台 SDK 或 www.reskit.com）、Pmon.exe（支持工具）或 Pulist.exe（资源包）。

实验：使用内核调试器！process 命令

内核调试器（已在第1章讲述）的！process 命令显示 EPROCESS 块中的信息子集。对于每个进程，这一输出分成两部分。首先你将看到如图 6-10 所示的关于进程的信息。（这里没有标注所有的输出字段——只标注了与本节密切相关的部分。）

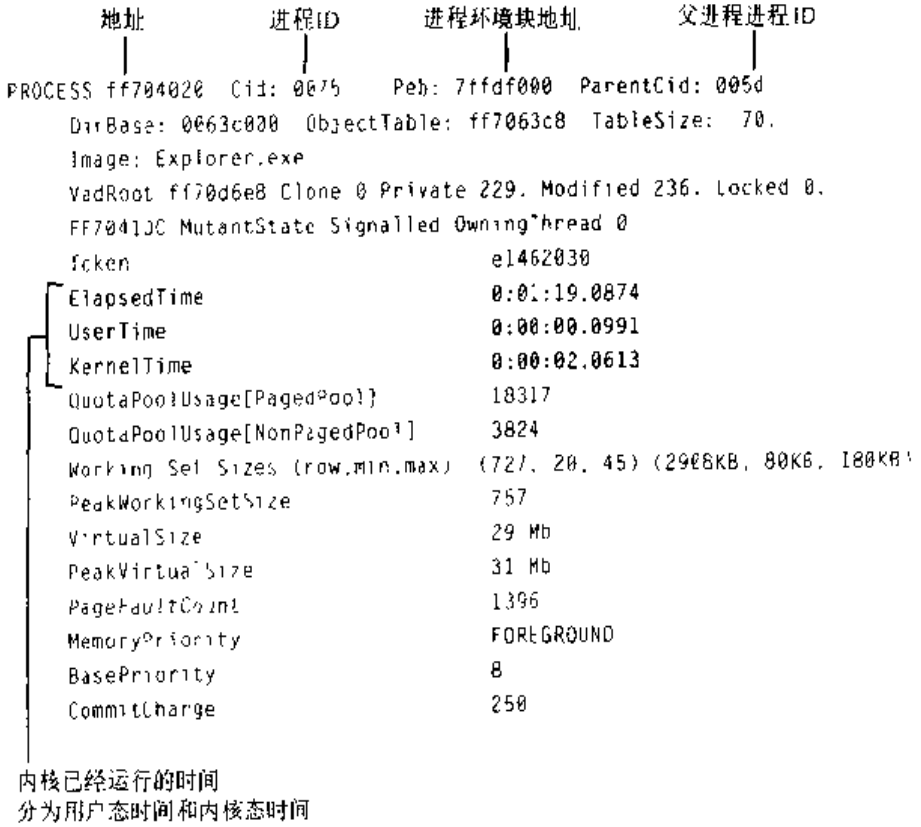


图 6-10 关于进程的信息

紧接着基本进程输出的是进程中的线程列表。在后面“实验：内核调试器命令！thread”中有对输出的解释。显示进程信息的其他命令包括转储进程句柄表的！handle（在第3章的“对象句柄和进程句柄表”中有详细描述）。在第8章描述了进程和线程的安全性结构。

另外可以用！strct 转储 EPROCESS 块。参见“实验：显示 EPROCESS 块的格式”获取更多的信息。

6.2 CreateProcess 流程

本章到目前为止，已经了解了进程的部分结构以及你（或操作系统）能用来操作进程的 API 函数，还找到了怎样利用工具来查看进程是如何与操作系统交互作用的。但是这些进程是如何产生的，以及当它们完成任务后是怎样退出的呢？在下面的几节中就将讨论 Win32 进程是如何产生的。

当应用程序调用 Win32 进程创建函数之一，诸如 CreateProcess、CreateProcessAsUser 或 CreateProcessWithLogon 函数时，就将创建一个 Win32 进程。创建 Win32 进程是由在操作系统的三个部分中完成的几个阶段组成的。这三个部分是：Win32 客户端的 Kernel32.dll 库、Windows 2000 执行程序和 Win32 子系统进程（Csrss）。由于 Windows 2000 的多环境子系统体系结构，Windows

2000 的执行程序进程对象（它可以被其他的子系统使用）的创建是同创建 Win32 进程中的工作分开的。所以，尽管下面对 Win32 CreateProcess 函数流程的描述是复杂的，但是请记住，与创建 Windows 2000 执行程序进程对象的内核工作相反，该工作的某些部分是与 Win32 子系统添加的语义相关的。

下面是使用 Win32 CreateProcess 创建进程的主要阶段的总结。每个阶段执行的操作将在随后的几节中详细描述。

1）打开将在进程中被执行的映像文件（.EXE）。

2）创建 Windows 2000 执行程序进程对象。

3）创建初始线程（堆栈、环境和 Windows 2000 执行程序线程对象）。

4）向 Win32 子系统通知新进程以便它可以设置新的进程和线程。

5）启动初始线程的执行（除非 CREATE-SUSPENDED 标记被指定）。

6）在新进程和线程的环境中，完成地址空间的初始化（例如加载所需的 DLL）并开始程序的执行。

图 6-11 显示了 Windows 2000 在创建一个进程的过程中的各阶段的概况。

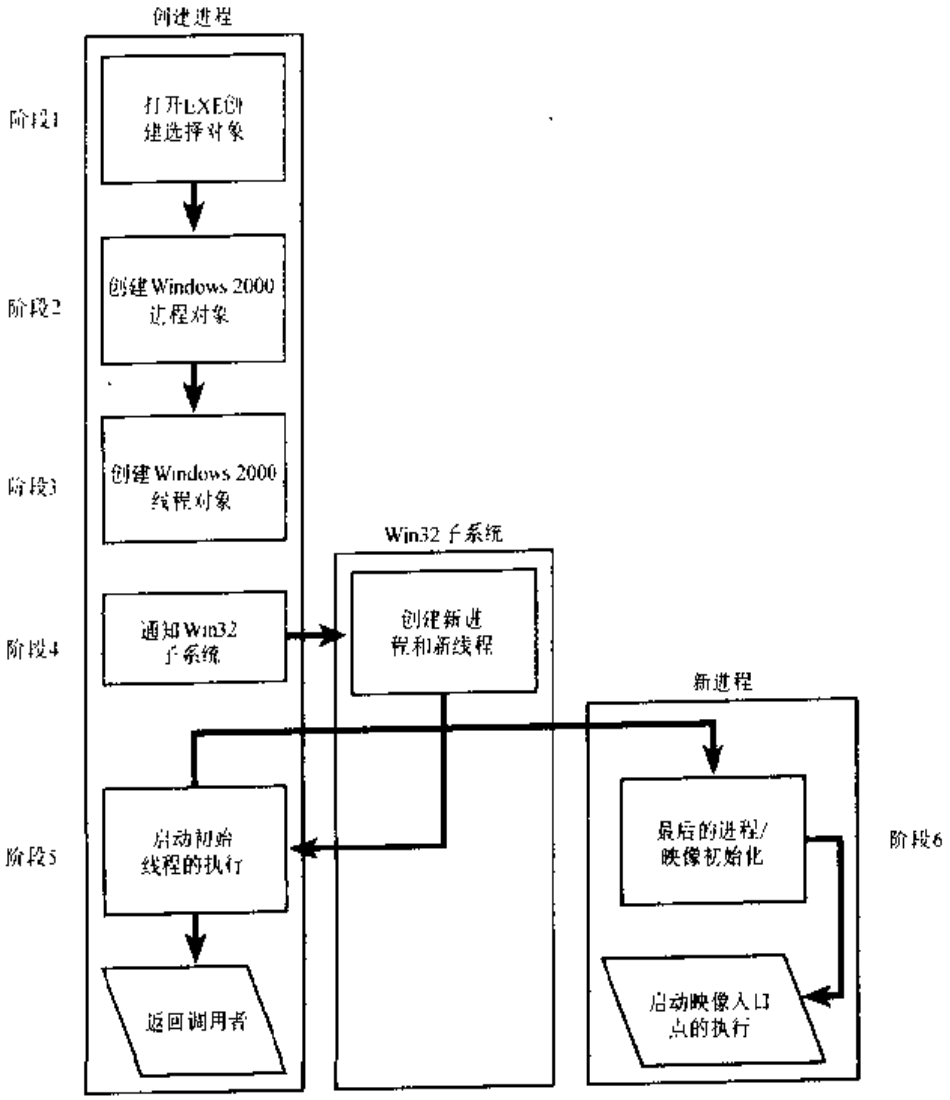


图 6-11　进程创建的主要阶段

在详细描述这些阶段之前，本书将提到一些适合于所有阶段的简略说明。

■ 在 CreateProcess 中，新进程的优先级等级由 CreationFlags 参数中的独立位指定。因此，可以对单个 CreateProcess 调用指定多个优先级等级。WINDOWS 2000 通过选择最低的优先级等级设置来解决分配给进程的优先级问题。

■ 如果没有为新进程指定优先级等级，那么优先级等级就默认为“Normal”，除非创建它的进程的优先级等级是“Idle”或“Below Normal”，这样新进程的优先级同创建它的进程的优先级相同。

■ 如果给新进程指定的优先级等级是“Real time（实时）”，并且进程的调用程序没有“Increase Scheduling Priority（增加调度优先级）”的特权，则用“High”优先级等级代替。换句话说，CreateProcess 不会失败，因为调用程序没有足够的特权以“实时”优先级等级来创建进程；新的进程不会有像“实时”这样的优先级。

■ 所有的窗口都与桌面（工作空间的图形化表示）相关联。如果在 CreateProcess 中没有指定桌面，则进程将与调用程序的当前桌面关联。

现在你的背景知识已经足够了，在下面的几节中将详细介绍 CreateProcess 的步骤。

注意 CreateProcess 的许多步骤与进程虚拟地址空间的设置有关，因此将会引用第 7 章定义的内存管理的术语和结构。

6.2.1 阶段 1：打开要执行的映像

如图 6-12 所示，CreateProcess 的第一步是找到合适的 Win32 映像，这个 Win32 映像将运行由调用程序指定的可执行文件，并创建一个区域对象以便稍后把它映射到新进程的地址空间中。如果没有指定映像名称，那么命令行的开始标记（被定义为命令行字符串以空格或制表符结尾的开始部分，是一个有效的文件规格说明）被用作映像文件名称。

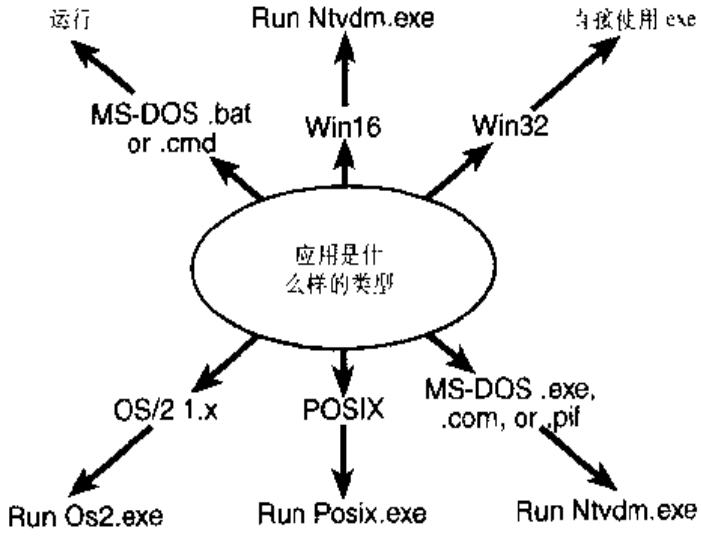


图 6-12 选择要激活的 WIN32 映像

如果指定的可执行文件是 Win32 应用程序，它就可以直接使用。如果不是 Win32．exe（例如，如果它是 MS－DOS、Win16、POSIX 或 OS/2 应用程序），则 CreateProcess 要经过一系列步骤以找到一个 Win32 的支持映像来运行它。这个进程是必需的，因为非 Win32 应用程序不能直接运行——Windows 2000 改为使用少数特殊的支持映像中的某一个，这些支持映像负责实际执行非 Win32 应用程序。例如，如果我们尝试去运行 Posix．exe 应用程序，CreateProces 就以这样的身份识别它，并且把映像改变为在 Win32 可执行程序 Posix．exe 上运行。如果我们尝试去运行 MS－DOS 或 Win16 可执行程序，将要运行的映像就成为 Win32 可执行的 NTVDM.EXE 程序。简而言之，我们不可以直接创建一个“非”Win32 进程。如果 Windows 2000 不能找到一种方法以解决如何按 Win32 进程激活映像（如表 6－6 所示），那么 CreateProcess 就失败了。

表 6-6 CreateProcess 阶段 1 的决策树

如果映像是	这个映像将运行	结 果
POSIX 可执行文件	Posix.exe	CreateProcess 从阶段 1 重新开始
OS/2 1.x 映像	Os2.EXE	CreateProcess 从阶段 1 重新开始
以 .exe、.com、.pif 为扩展名的 MS-DOS 应用程序	Ntvdm.exe	CreateProcess 从阶段 1 重新开始
Win16 应用程序	Ntvdm.exe	CreateProcess 从阶段 1 重新开始
以 bat 或 .cmd 为扩展名的命令程序的 MS-DOS 应用程序	Cmd.exe	CreateProcess 从阶段 1 重新开始

具体的说，为运行一个映像，CreateProcess 的决策树如下：

■ 如果映像是 OS/2 1.x 应用程序，将改为运行 Os2.EXE 并且 CreateProcess 从第一阶段重新开始，

■ 如果映像是以 .exe、.com 或 .pif 为扩展名的 MS-DOS 应用程序，就会将消息发送到 Win32 子系统，以验证用于这次会话的 MS-DOS 支持进程（Ntvdm.exe，在注册表键值 HKLM\System…\Control\WOW\cmdline 中指定）是否已经创建了。如果创建了支持进程，它就会被用于运行 MS-DOS 应用程序（Win32 子系统把该消息发送到 VDM（虚拟 DOS 机）进程来运行新映像），并且返回 CreateProcess。如果没有创建支持进程，Windows 2000 将改为运行 NTvDM.EXE，并且 CreateProcess 从阶段 1 重新开始。

■ 如果运行的文件以 .bat 或 .cdm 为扩展名，则映像将要在 Windows 2000 命令提示符下运行 Cmd.exe，并且 CreateProcess 从阶段 1 重新开始（批处理文件名作为第一个参数传递给 Cmd.exe）。

■ 如果是一个可执行的 Win16（Windows 3.1）映像，CreateProcess 必须决定是否创建一个新的 VDM 进程来运行该映像，或者映像是否使用默认的在系统范围内共享的 VDM 进程（该 VDM 进程或许还没有创建）。这个决定由 CreateProcess 标志 CREATE_SEPARATE_WOW_VDM 和 CREATE_SHARED_WOM_VDM 控制。如果没有指定这些标志，则默认的操作由注册表键值 HKLM\System…\Control\WOW\DefaultSeparateVDM 指定。如果应用程序在单独的 VDM 下运行，则将要运行的映像变为 HKLM\System…\Control\WOW\wowcmdline 键值，并且 CreateProcess 从阶段 1 重新开始。否则，Win32 子系统发送消息来查看系统范围内的 VDM 进程是否存在以及是否能够使用（如果 VDM 进程在不同的桌面下运行或 VDM 进程不能在与调用程序同等的安全级下运行，就不能使用它。并且必须创建一个新的 VDM 进程）。如果系统范围内的 VDM 进程能够使用，Win32 子系统就会给它发送一条消息以运行新的映像，并返回 CreateProcess。如果还没有创建 VDM 进程（或者它虽然存在但却不能够使用），Windows 2000 将改为运行 VDM 支持映像，并且 Createprocess 从阶段 1 重新开始。

到此为止，CreateProcess 已经成功地打开了一个有效的 Windows 2000 可执行文件，并且为它创建了一个区域对象。这个对象还没有映射到内存中，但它是打开的。然而，仅仅因为成功地创建了区域对象并不意味着文件是有效的 Win32 映像。它可能是可执行的 DLL 或 POSIX 文件。如果是一个可执行的 POSIX 文件，Windows 2000 将改为运行 POSIX.EXE，并且 CreateProcess 从阶段 1 的起点重新开始。如果文件是 DLL，CreateProcess 将会失败。

现在 CreateProcess 已经找到了有效的 Win32 可执行映像，它就在 \ HKLM \ SOFTWARE \ Microsoft \ Windows NT \ CurrentVersion \ Image File Execution Options 下的注册表中查看是否有一个带有可执行映像的文件名和扩展名的子键（但没有目录和路径的信息，例如 IMAGE.EXE）。如果该子键存在，CreateProcess 就为那个键寻找一个名为 Debugger 的值。如果此键值非空，将要运行的映像会成为键值中的字符串，并且 CreateProcess 从阶段 1 重新开始。

注意 可以利用 CreateProcess 的行为在 Windows 2000 服务进程启动前调试它们的启动代码，而不是在启动服务后附接调试器，因为那时不允许调试启动代码。如果你偶尔想搞个恶作剧，就可以使用这个行为来迷惑人们，当它们要运行指定的文件时却运行了另外的文件。

6.2.2 阶段 2：创建 Windows 2000 执行程序进程对象

到此为止，CreateProcess 已经打开了一个有效的 Windows 2000 可执行文件，并且为了把它映射到新的进程地址空间而创建了区域对象。下一步它要通过调用内部系统函数 NtCreateProcess 来创建运行映像的 Windows 2000 执行程序进程对象。创建执行程序进程对象包括下面几步：

1）设置 EPROCESS 块。

2）创建初始进程地址空间。

3）创建内核进程块。

4）决定进程地址空间的设置。

5）设置 PEB。

6）完成执行程序进程对象的设置。

注意 仅在系统初始化阶段没有父进程。但在那以后总需要父进程为新进程提供安全环境。

1. 阶段 2A：设置 EPROCESS 块

这一部分包括五步：

1）分配和初始化 Windows 2000 EPROCESS 块。

2）把新进程配额块移动到父进程配额块的地址处，并增加父进程配额块的引用计数。

3）把父进程的进程 ID 和会话 ID（如果可用）存储到新进程对象的 InheritedFromUniqueProcessID 字段中。

4）设置新进程的退出状态为 STATUS_PENDING。

5）创建进程的主访问令牌（它的父进程主令牌的副本）。新的进程继承了其父进程的安全配置文件（如果 CreateProcessAsUser 函数被用来为新进程指定不同的访问令牌，则令牌将会适当的改变）。

2. 阶段 2B：创建初始进程地址空间

初始进程地址空间由三个页面构成：

- 页面目录。
- 超空间页。
- 工作集列表。

要创建这三个页面，将执行下列步骤：

1）在适当的页面表中创建页面表项以映射上面列出的三个初始页面。

2）考虑新的页面，从内核变量 MmTotalCommittedPages 中减 3，并在 MmProcessCommit 中加 3。

3）从 MmResidentAvailablePages 中减去系统范围内默认的进程最小工作集大小（PsMinimumWorkingSet）。

4）页表为系统空间的非页部分调页，并且系统高速缓存被映射到进程中。

5）进程的最小和最大工作集大小分别设置为 PsMinimumWorkingSet 和 PsMaximumWorkingSet 的值。

3. 阶段 2C：创建内核进程块

CreateProcess 下一步要做的是 KPROCESS 块的初始化。KPROCESS 块包含了指向内核线程列表的指针（内核中没有句柄的内容，因此它忽略了对象表）。内核进程块也指向进程的页面表目录（用于跟踪进程的虚拟地址空间）、进程的线程执行的总时间、进程默认的基本调度优先级（它作为 Normal 或 8 启动，除非父进程被设置为 Idle 或 Below Normal，在这种情况下设置被继承）、用于进程中的线程的默认处理器相似性和进程默认时间片的初始值，时间片的初始值从在系统范围内的时间片数组中的第一项 PspForegroundQuantum［0］得来。

注意 Windows 2000 Professional 和 Windows 2000 Server 的初始时间片是不相同的。关于线程时间片的详细信息，请参阅第 6.5 节的“线程调度”。

4. 阶段 2D：决定进程地址空间的设置

设置新进程的地址空间有些复杂，所以让我们看一看这一步中同时包含了什么。为了从本节中得到更多的东西，你应该比较熟悉 Windows 2000 内存管理器的内部情况，这将在第 7 章中描述。

1）虚拟内存管理器把进程的上一次剪裁时间设置为当前时间值。工作集管理器（它在平衡集管理器（Balance Set Manager）系统线程环境中运行）使用此值来决定什么时候启动工作集剪裁。

2）初始化用于页面目录的页面帧号（PFN）数据库以及映射超空间的页面目录项。

3）内存管理器初始化进程工作集列表——现在可以获得页面错误。

4）把主版本号和次版本号从可执行文件复制到 EPROCESS 块。

5）将区域（当映像文件被打开时创建）映射到新的进程地址空间，进程区域基地址被设置为映像的基址。

6）Ntdll.dll 被映射到进程。

7）系统范围的国家语言支持表被映射到进程地址空间。

注意 POSIX 进程复制它们父进程的地址空间，因此它们不必经过这些步骤去创建新的地址空间。对于 POSIX 应用程序，新进程区域基地址被设置为它们父进程的区域基地址。父进程的 PEB 被复制到新进程。

8）如果父进程被包含在作业（job）中，新进程被添加到作业（作业在本章的末端描述）

9）CreateProcess 把新进程块插入到 Windows 2000 活动进程列表的末端（PsActiveProcessHead）。

5. 阶段2E：设置PEB

CreateProcess为PEB分配页面并且初始化一些字段如表6-7所示。

表6-7 PEB字段的初值

字 段	初始值
ImageBaseAddress	主域的基址
NumberOfProcessors	KeNumberProcessors内核变量
NtGlobalFlag	NtGlobalFlag内核变量
CriticalSectionTimeout	MmCriticalSectionTimeOut内核变量
HeapSegmentReserve	MmHeapSegmentReserve内核变量
HeapSegmentCommit	MmHeapSegmentCommit内核变量
HeapDeCommitTotalFreeThreshold	MmHeapDeCommitTotalFreeThreshold内核变量
HeapDeCommitFreeBlockThreshold	HeapDeCommitFreeBlockThreshold内核变量
NumberOfHeaps	0
MaximumNumberOfHeaps	(页面大小 - PEB大小) /4
ProcessHeaps	PEB后的第一个字节
OSMajorVersion	NtMajorVersion内核变量
OSMinorVersion	NtMinorVersion内核变量
OSBuildNumber	NtBuildNumber内核变量 &0x3FFF
OSPlatformId	2

如果映像文件指定明确的Win32版本值，就要用该信息取代表6-7中的初值。从映像版本信息字段到PEB字段的映射如表6-8所示。

表6-8 初始化PEB值的Win32取代值

字段名称	从映像文件头取得的值
OSMajorVersion	OptionalHeader。Win32VersionValue& 0xFF
OSMinorVersion	(OptionalHeader。Win32VersionValue > > 8) &0xFF
OSBuildNumber	(OptionalHeaderWin32VersionValue > > 16) &0x3FFF
OSPlatformId	(OptionalHeader。Win32VersionValue > > 30) 0x2

6. 阶段2F：完成执行程序进程对象的设置

在新进程的句柄可以返回之前，必须完成最后的几步设置：

1) 初始化进程句柄表：如果为父进程设置了复制句柄标志，把任何可继承的句柄从父进程对象句柄表复制到新进程。(要了解关于对象句柄表的详细信息，请参阅第3章。)

2) 如果仅在运行Windows 2000 Professional，并且映像头指定为IMAGE_FILE_AGGRESIV_WS_TRIM，则在进程块内设置PS_WS_TRIM FROM EXE_HEADER标志。如果在小内存的x86系统上运行Windows 2000 Professional，则在进程块内设置PS_WS_TRIM_BACKGROUND ONLY APP，它限制了没有与前台窗口相关的进程的过度修剪(agressive trimming)。这些工作集标志没有为运行Windows 2000 Server的系统上创建的进程设置。

3) 如果设置了映像头特有的标志IMAGE_FILE_UP_SYSTEM ONLY(表示映像只能在单处理器系统上运行)，则为新进程的所有线程选择单个CPU来运行。通过在可用处理器之间简单地循环来完成这个选择进程——每次运行这种映像类型时，使用下一个处理器。通过该方法，这

些映像类型将被均匀分布到多个处理器上。

4) 如果映像指定了明确的处理器相似性掩码（例如，配置头中的字段），则该值就被复制到 PEB，并且在这之后被设置为默认进程相似性掩码。

5) 如果在父进程的 PEB 中有一个事件日志区域，则事件日志将被复制到新进程，并且为新进程复制一个句柄到区域中。

6) 如果启用在系统范围内的进程审计（process auditing）（通过 Group Policy 插件完成），则进程的创建将被写到审计日志中。

7) 设置进程的创建时间，把新进程的句柄返回给调用程序（在 Kernel32.dll 中的 CreateProcess），然后返回到 CreateProcess 的原始调用程序。

6.2.3 阶段 3：创建初始线程及其堆栈和环境

到现在为止，已经完成了设置 Windows 2000 执行程序进程对象。然而，它现在还没有线程，因此还不能做任何事情。在可以创建线程之前，它需要堆栈和环境以在其中运行。因此首先要设置它们。用于初始线程的堆栈大小来自映像——没有办法去指定其他的大小。

现在通过调用 NtCreateThread 就可以创建线程（要了解创建线程的细节，请参阅第 6.4 节“CreateThread 流程”一节的内容）。线程参数（不能在 CreateProcess 中指定，但可以在 CreateThread 中指定）是 PEB 的地址。运行在这个新线程的环境中的初始化代码将使用此参数（如同阶段 6 中所描述的）。然而，这个线程还不能够做任何事情——它是在挂起状态下创建的，它要等到进程被完全初始化后才能继续运行（如同在阶段 5 中所描述的）。

6.2.4 阶段 4：向 Win32 子系统通知新进程

在所有必要的执行程序进程和线程对象创建以后，Kernel32.dll 将消息发送到 Win32 子系统，这样 Win32 子系统就能对新的进程和线程进行设置。这条消息包括下列信息：

- 进程和线程句柄。
- 创建标志中的项。
- 进程创建者的 ID。
- 表明进程是否属于 Win32 应用程序的标志（以便 Csrss 能够决定是否显示启动光标）。

在 Win32 子系统收到这条信息后，它将执行下列步骤：

1) CreateProcess 为进程和线程复制句柄。这个步骤把进程和线程的使用计数从 1（在创建时设置）增加到 2。

2) 如果没有指定进程优先级类别，CreateProcess 按 6.2.1 节描述的算法描述。

3) 分配 Csrss 进程块。

4) 新进程的异常端口被设置为 Win32 子系统的通用函数端口，以便当进程中发生异常时，Win32 子系统将收到消息。（要了解异常处理的详细信息，请参阅第 3 章）

5) 如果在调试进程（也就是说，如果进程附接于调试器进程），则进程调试端口被设置为 Win32 子系统的通用函数端口。这项设置将确保 Windows 2000 把新进程内发生的调试事件如线程的创建和删除、异常等等）以消息的方式发送到 Win32 子系统，这样它就能够调度事件到作

为新进程调试器的进程。

6）分配和初始化 Csrss 线程块。

7）CreateProcess 将线程插入进程的线程列表。

8）递增在这一会话中的进程计数。

9）进程关闭级别被设置为 x280（默认的进程关闭级别——请参阅 MSDN 库中更多有关 SetProcessShut - DownParameters 的信息）。

10）将新的进程块插入 Win32 子系统范围内的进程列表。

11）分配和初始化由 Win32 子系统内核模式部分使用的每个进程（per - process）的数据结构（W32PROCESS 结构）。

12）显示应用程序的开始光标。光标是常见的带有沙漏的箭头—— Windows 2000 和用户会话的方式，“我正在启动某事，但你在其间可以使用光标”。如果在两秒钟后进程不做 GUI 调用，光标将变回标准的指针。如果进程在分配的时间内进行了 GUI 调用，CreateProcess 为应用程序等待五秒的时间来显示窗口。此后，CreateProcess 将重新设置光标。

6.2.5 阶段 5：开始初始化线程的执行

到此为止，已经确定了进程环境，并且分配了它的线程所使用的资源；进程有了一个线程，并且 Win32 子系统知道这个新进程。除非调用程序指定 CREATE _ SUSPENDED 标志，初始线程现在被恢复执行，这样它就可以开始运行并完成产生在新进程环境中的进程初始化工作的剩余部分（阶段 6）。

6.2.6 阶段 6：在新进程环境中完成进程初始化

由 KeInitializeThread 调用的 KiInitializeContextThread 创建了线程的初始化环境以及内核栈。运行内核模式线程启动例程 KiThreadStartup 时，新线程就开始运行（关于线程启动步骤的详细描述，请参阅 6.4 节“CreateThread 流程”）。KiThreadStartup 例程执行下列步骤：

1）把 IRQL 级别从 Dispatch/DPC 级别降低到 APC（异步过程调用）级别。

2）允许工作集扩展。

3）为新线程的用户模式 APC 排队来执行用户模式在 Ntdll. dll 内的线程启动例程 LdrInitializeThunk。

4）把 IRQL 级别降低到 0，启动 APC 并调用 LdrInitializeThunk。LdrInitializeThunk 例程初始化加载程序、堆管理器、NLS 表、线程本机存储（TLS）数组和临界区结构。然后，它加载任何需要的 DLL，并使用 DLL _ PROCESS _ ATTACH 函数代码调用 DLL 入口点。

5）如果创建的进程是一个调试器进程，则该进程中的所有线程都将被挂起（在步骤 3 中可能创建线程）。然后一条创建进程的消息被发送到进程的调试端口（Win32 子系统函数端口，因为这是一个 Win32 进程），这样子系统就能够将进程启动调试事件（CREATE PROCESS _ DEBUG _ INFO）传递到适当的调试器进程。然后 LiThreadStartup 等待 Win32 子系统以从调试器中得到答复（借助于 ContinueDebugEvent 函数）。在 Win32 子系统答复后，所有的线程将被继续执行。

6）最后，映像在用户模式开始执行。这是通过创建一个陷阱帧来完成，该陷阱帧指定了

先前的模式作为用户模式并且该地址作为映像的主要入口点而返回。这样，当解决了引起线程在内核模式下开始运行的陷阱后，程序在用户模式下的适当位置开始运行。

6.3 线程的本质

现在我们已经仔细研究过了进程，就让我们把注意力转到线程的结构上来。除非在特别指明的情况下，否则，我们就认为本节所讲内容应用于正常的用户模式线程和内核模式系统线程（在第 3 章中描述）。

6.3.1 数据结构

在操作系统级别上，Windows 2000 线程是由执行程序线程（ETHREAD）块代表的，该线程块在图 6-13 中阐明。ETHREAD 块和它所指向的结构位于系统地址空间内，但线程环境块（TEB）除外，它位于进程地址空间内。另外，Win32 子系统进程（CSRSS）为在 Win32 进程中创建的每个线程保持一个并行结构。而且，对于曾经调用 Win32 子系统 USER 或 GDI 函数的线程，Win32 子系统（WIN32K.SYS）内核模式的部分维持一个由 ETHREAD 块指向的每线程数据结构（叫作 W32THREAD 结构）。

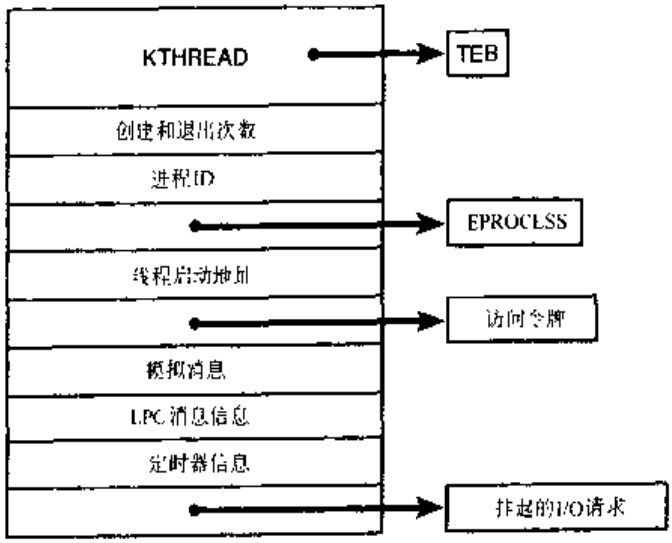


图 6-13 执行程序线程块的结构

构造（Fiber）和线程

构造允许一个应用程序调度它拥有的线程的执行而不依赖与 Windows 2000 内嵌的基于优先权的调度机制。构造通常又称为“轻重量”级的线程，并且在调度方面，它们对操作系统是不可见的，因为它们是在 Kernel32.dll 中以用户模式实现的。为了使用构造，首先需要调用 Win32 的 ConvertThreadToFiber 函数。这个函数将线程转换为一个正在运行的构造。然后，这个转换得来的新构造调用 CreateFiber 函数创建其他构造（每个构造可以有它自己的构造）。然而，与线程不同，一个构造不能开始执行，除非它被通过调用 SwitchToFiber 函数人为选择。新构造运行直到它退出或者它调用 SwitchFiber 选择其他构造运行。为了获得更多的信息，请参见有关 Fiber 的平台 SDK 文档。

图6-13 中举例说明的大多数字段是不需要加以说明的。第一个字段是内核线程〈KTHREAD〉块。接下来是线程标识信息、进程标识信息（包括一个指向拥有该线程的进程的指针，这样就可以访问该进程的环境信息），以指向访问令牌的指针为形式的安全信息和模拟信息，最后是与LPC消息和挂起的I/O请求有关的字段。如表6-9所示，一些关键字段将在本书的其他章节中详细描述。

表6-9 执行程序线程块的关键内容

元 素	说 明	参考章节
KTHREAD块	请参见表6.10	第6章
线程时间信息	线程的创建和退出时间	
进程标识	进程ID和指向线程所属进程的EPROCESS块的指针	
开始地址	线程启动例程的地址	
模拟信息	访问令牌和模拟级别（如果线程模拟一个客户）	安全性（第8章）
LPC信息	线程正在等待的消息ID和消息地址	第3章"本机过程调用"
I/O信息	挂起的I/O请求数据包（RP）列表	I/O系统（第9章）

要更详细地了解EHREAD块的内部结构，可以使用内核调试器！threadfields命令来以十六进制形式显示此结构中几乎每个字段的偏移量。虽然许多字段名称是不需要加以说明的，但是输出既不给出字段的数据类型，也不显示包含在EHTREAD块内或EHREAD块指向的结构的格式。

让我们进一步看一下上面提到过的两个关键的线程数据结构：KTHREAD块和TEB。KTHREAD块包含Windows 2000内核需要访问的信息，这些信息用于执行线程调度和使运行的线程同步。它的结构如图6-14所示。

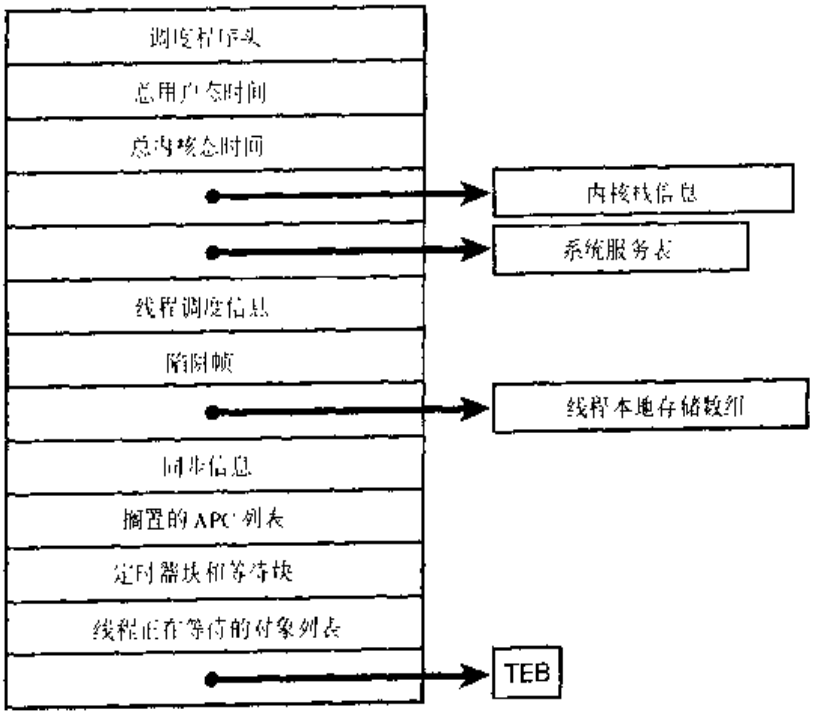


图6-14 内核线程块的结构

KTHREAD 块的关键字段在表 6-10 中做了简要描述。

表 6-10 KTHREAD 块的主要内容

元 素	说 明	参考章节
调度程序头	因为线程是能够等待的对象，所以它以标准的内核调度程序对象头开始	调度程序对象（第 3 章）
执行时间	总的用户模式和内核模式 CPU 时间	
内核堆栈信息指针	内核堆栈的栈底和栈顶地址	内存管理（第 7 章）
指向系统服务表的指针	每一个线程都从这个字段开始，指向主系统服务表（KeServiceDescriptorTable）。当线程初次调用 Win32 GUI 服务时，它的系统服务表变成了一个包括在 Win32k.sys 中的 GDI 和 USER 服务的系统服务表	系统服务调度（第 3 章）
调度信息	基本的和当前的优先级、时间片、相似性掩码、理想处理器、调度状态、冻结计数、挂起计数	线程调度
等待块	线程块包含四个内置的等待块，这样就没有必要在每一次线程处于等待状态时都分配和初始化等待块（其中一个用于定时器）	同步（第 3 章）
等待信息	线程等待的对象列表、等待原因、线程进入等待状态的时间	同步（第 3 章）
变异列表	属于线程的变异对象的列表	同步（第 3 章）
APC 队列	挂起的用户模式和内核模式的 APC 列表、可报警标志	APC 队列（第 3 章）
定时器块	内置的定时器块（也是相应的等待块）	
队列列表	线程等待的对象的队列列表	同步（第 3 章）
指向 TEB 的指针	线程 ID、TLS 信息、PEB 指针、GDI 和 OpenGL 信息。	

实验：显示 ETHREAD 和 KTHREAD 结构

虽然你可以使用内核调试器命令！threadfields 查看一些信息，类似于本章实验“显示 EPROCESS 块的格式”的！processfields 的输出，但如果用！ethread 你可以获得关于 ETHREAD 和 KTHREAD 块的字段更详细的输出。！ethread 命令是由 Kdex2x86.dll 中提供的命令之一。为了使用这些扩展命令，你必须首先加载这个 DLL，如下输出所示。！ethread 命令取得一个 ETHREAD 块的地址，然后显示其后的 KTHREAD 和 ETHREAD 块。在下面的例子中，我们使用！process 命令转储一个进程并找到进程中包含线程对应的线程块的地址。输出结果如下：

```
kd> .load kdex2x86
kd> !process 2c8 2
Searching for Process with Cid == 2c8
```

```
+1fc void     *LpcReplyMessage =                00000000
+200 uint32   LpcReplyMessageId =               00000000
+204 uint32   PerformanceCountLow =             00000000
+208 struct _PS_IMPERSONATION_INFORMATION *ImpersonationInfo = 00000000
+20c struct   _LIST_ENTRY IrpList
+20c    struct   _LIST_ENTRY *Flink =           8112F91C
+210    struct   _LIST_ENTRY *Blink =           8112F91C
+214 uint32   TopLevelIrp =                     00000000
+218 struct   _DEVICE_OBJECT *DeviceToVerify =  00000000
+21c uint32   ReadClusterSize =                 00000007
+220 byte     ForwardClusterOnly =              00
+221 byte     DisablePageFaultClustering =      00
+222 byte     DeadThread =                      00
+223 byte     HideFromDebugger =                00
+224 uint32   HasTerminated =                   00000000
+228 uint32   GrantedAccess =                   001f03ff
+22c struct   _EPROCESS *ThreadsProcess =       810BAD70
+230 void     *StartAddress =                   77E92C50
+234 void     *Win32StartAddress =              77D48759
+234 uint32   LpcReceivedMessageId =            77d4b759
+238 byte     LpcExitThreadCalled =             00
+239 byte     HardErrorsAreDisabled =           00
+23a byte     LpcReceivedMsgIdValid =           00
+23b byte     ActiveImpersonationInfo =         00
+23c int32    PerformanceCountHigh =            00000000
+240 struct   _LIST_ENTRY ThreadListEntry
+240    struct   _LIST_ENTRY *Flink =           810BAFE0
+244    struct   _LIST_ENTRY *Blink =           810A4550
```

在图 6-15 中说明的 TEB 是在本节中讲述的唯一存在于进程地址空间（与系统空间相对）

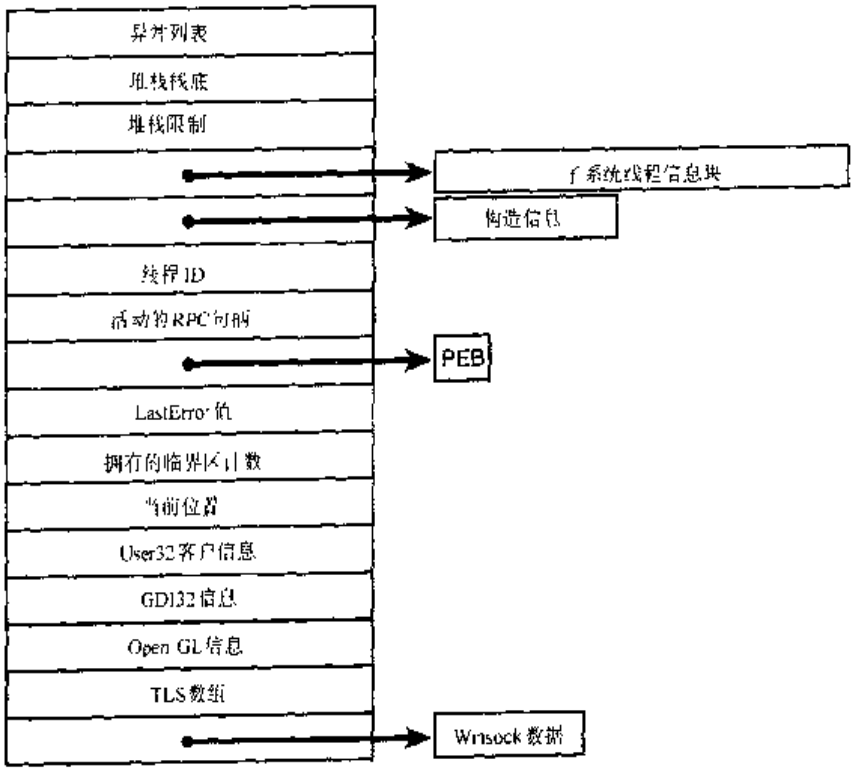


图 6-15 线程环境块的字段

的数据结构

TEB 存储了用于映像加载程序和各种 Win32 DLL 的环境信息。因为这些组件在用户模式下运行，它们需要一种从用户模式可写的数据结构。这也是为什么这种结构存储于用户地址空间而没有存储在系统空间，因为系统空间只有在内核模式下才可写。可以使用内核调试器的!thread 命令找到 TEB 的地址。

实验：观察 TEB

你可在内核调试器中用!teb 转储 TEB 结构；结果输出如下：

```
kd> !teb
TEB at 7FFDE000
    ExceptionList:    12ffb0
    Stack Base:       130000
    Stack Limit:      12d000
    SubSystemTib:     0
    FiberData:        1e00
    ArbitraryUser:    0
    Self:             7ffde000
    EnvironmentPtr:   0
    ClientId:         490.458
    Real ClientId:    490.458
    RpcHandle:        0
    Tls Storage:      0
    PEB Address:      7ffdf000
    LastErrorValue:   0
    LastStatusValue:  0
    Count Owned Locks:0
    HardErrorsMode:   0
```

6.3.2 内核变量

和进程一样，线程的运行方式受到许多 Windows 2000 内核变量的控制。表 6－11 中列出了与线程相关的内核模式内核变量。

表 6-11　与线程相关的内核变量

变　量	类　型	说　明
PspCreateThreadNotifyRoutine	指针数组	指向在线程创建和删除过程中被调用例程的指针数组（最大值 8）
PspCreateThreadNotifyRoutineCount	DWORD	注册的线程通知例程的计数
PspCreateProcessNotifyRoutine	指针数组	指向在线程创建和删除过程中被调用例程的指针数组（最大值 8）
PspCreateProcessNotifyRoutineCount	DWORD	注册的线程通知例程的计数
PspCidTable	句柄表	存储进程和线程对象的句柄表。当线程和进程 ID 保证唯一性时，使用线程或进程的句柄值

6.3.3 性能计数器

线程数据结构中的大部分关键信息是通过性能计数器输出的，表 6-12 列出了这些计数器。可以通过使用标准的 Performance Monitor（性能监视器）实用程序来得到大量的线程内部的信息。

表 6-12 与线程相关的性能计数器

对象：计数器	功 能
Process: Priority Base	返回当前进程的基本优先级。这是在该进程内创建的线程的起始优先级
Thread: % Privileged Time	描述在指定的时间间隔内，线程在内核模式运行时所占时间的百分数
Thread: % Processor Time	描述在指定的时间间隔内，线程使用 CPU 时间的百分数。这个数值是特权时间的百分数与用户时间的百分数之和
Thread: % User Time	描述在指定的时间间隔内，线程在用户模式运行所占时间的百分数
Thread: Context Switches/sec	返回每一秒钟系统上在执行的环境切换的次数。此值越大，有相同优先级的试图执行的线程数量就越多
Thread: Elapsed Time	返回线程消耗的 CPU 时间（以秒为单位）
Thread: ID Process	返回线程所在进程的进程 ID。此 ID 仅在进程生存期内有效，因为进程 ID 是可以重复使用的
Thread: ID Thread	返回线程的线程 ID。此 ID 仅在该线程的生存期内有效，因为线程 ID 是可以重复使用的
Thread: Priority Base	返回线程当前的基本优先级。这个数值可能与线程的起始基本优先级不同
Thread: Priority Current	返回线程当前的动态优先级
Thread: Start Address	返回线程的起始虚拟地址（注：对大多数线程来说，这个地址是相同的）
Thread: Thread State	返回线程当前状态的数值，取值范围从 0 到 7
Thread: Thread Wait Reason	返回线程处在等待状态的原因，数值范围从 0 到 19

6.3.4 相关函数

表 6-13 中列出了创建和使用线程的 Win32 函数。这个表中不包括与线程调度和优先级有关的函数，这部分包括在本章后面的“线程调度”一节中。

表 6-13 Win32 线程函数

函 数	说 明
CreateThread	创建新线程
CreateRemoteThread	在另一个进程中创建线程
ExitThread	正常地结束线程的执行
TerminateThread	终止线程
GetExitCodeThread	获得另一个线程的退出代码
GetThreadTimes	返回另外一个线程的定时信息
Get/SetThreadContext	返回或更改线程的 CPU 寄存器
GetThreadSelectorEntry	返回另一个线程的描述符表项（仅在 x86 系统中使用）

6.3.5 相关工具

除了 Performance Monitor 外，其他的几个工具软件也能揭示 Windows 2000 线程状态的各种元素。（显示线程调度信息的工具列在本章 6.5“线程调度”一节中）这些工具在表 6-14 中分别列出。

说明 为了使用 Tlist 显示线程细节，必须输入 tlist xxx，xxx 是指进程映像名或窗口标题（支持通配符）。

表 6-14 有关线程的工具及其功能

对 象	Perfmon	Pview	Pviewer	Pstat	Qslice	Tlist	KD! thread
线程 ID	√	√		√		√	√
实际起始地址	√	√	√	√			√
Win32 起始地址						√	√
当前地址	√	√	√				√
环境切换数	√	√	√	√			
全部用户时间		√		√			√
全部特权时间		√		√			√
消耗的时间	√	√					√
线程状态	√			√		√	√
等待状态理由	√			√		√	√
最后一次错误						√	
安全描述符			√				
访问令牌			√				
CPU 时间百分比	√				√		
用户时间百分比	√		√		√		
特权时间百分比	√		√		√		
TEB 地址							√
ETHREAD 地址							√
正在等待的对象							√

实验：内核调试器！thread 命令

内核调试器！thread 命令转储线程数据结构中信息的子集。内核调试器显示的一些关键元素的信息不能够由 Windows 2000 实用程序显示。这些信息包括：内部结构地址、优先级详细资料、堆栈信息、挂起的 I/O 请求列表；对于处于等待状态的线程，还包括它所等待的对象的列表（参考表 6-14）。

为了显示线程信息，可以使用！process 命令（它在显示完进程块后显示所有的线程块），也可以使用！thread 命令来转储指定的线程。线程信息的输出和一些关键字段的注解显示如图 6-16 所示。

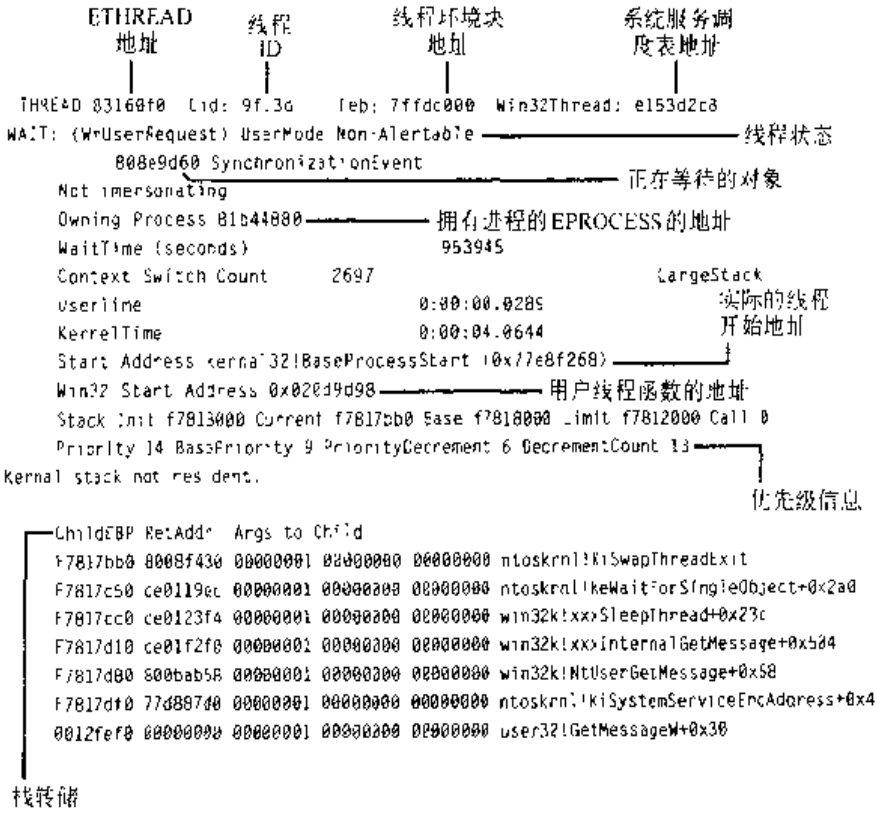


图 6-16 线程信息的输出及一些关键字段的注解

实验：查看线程信息

使用 Windows 2000 资源工具包中的 Tlist 实用程序，显示的进程的详细输出如下。注意线程列出了实际的 Win32 的起始地址。（所有其他的显示进程起始地址的实用程序显示实际的起始地址，而非 Win32 的起始地址。）

```
C:\> tlist winword
 155 WINWORD.EXE       Document1 - Microsoft Word
   CWD:     C:\book\
   CmdLine: "C:\Program Files\Microsoft Office\Office\WINWORD.EXE"
   VirtualSize:    64448 KB   PeakVirtualSize:   106748 KB
   WorkingSetSize:  1104 KB   PeakWorkingSetSize:  6776 KB
   NumberOfThreads: 2
    156 Win32StartAddr:0x5032cfdb LastErr:0x00000000 State:Waiting
    167 Win32StartAddr:0x00022982 LastErr:0x00000000 State:Waiting
                      0x50000000  WINWORD.EXE
     5.0.2163.1 shp  0x77f60000  ntdll.dll
     5.0.2191.1 shp  0x77f00000  KERNEL32.dll
         :
         list of DLLs loaded in process
```

6.4 CreateThread 流程

线程的生存周期是从程序创建线程时开始的。创建线程的请求过滤到 Windows 2000 执行程

序，在此处进程管理器为线程对象分配空间并调用内核以初始化内核线程块。在 Kernel32.dll 的 Win32 函数 CreateThread 内执行了下列步骤来生成 Win32 线程。发生在 Windows 2000 执行程序内的工作是步骤 3 中的子步骤，发生在新线程环境中的工作是步骤 7 的子步骤。因为进程创建包括创建线程，下面的内容的一部分是先前的 CreateProcess 流程描述的重复。

1）CreateThread 在进程的地址空间内为线程创建用户模式堆栈。

2）CreateThread 初始化线程的硬件环境（CPU 体系结构特定的）。要了解线程环境块的进一步内容，请参阅 Win32 API 参考文档中有关 CONTEXT 结构的部分。

3）调用 NtCreateThread 创建处于挂起状态的执行程序线程对象。在 Windows 2000 执行程序和内核之中，下列步骤在内核模式下执行：

a）渐增进程对象中的线程计数。

b）创建并初始化执行程序线程块（ETHREAD）。

c）为新线程生成线程 ID。

d）从非页交换区分配线程的内核堆栈。

e）在进程的用户模式地址空间设置 TEB。

f）把线程起始地址（KiThreadStartup）和用户指定的 Win32 起始地址存储在 ETHREAD 块中。在 KiThreadStartup 处线程开始执行，它完成线程特定的初始化，然后由 CreateThread 的调用程序所指定的实际线程例程将被启动。

g）调用 KeInitializeThread 设置 KTHREAD 块。线程的初始的和当前的基本优先级被设置为进程的基本优先级，并且它的相似性和时间片被设置为进程的相似性和时间片。该函数同样基于进程的线程种子（在 CreateProcess 中设置的随机数）来设置初始的线程的理想处理器。然后这个“种子”递增。这样假定系统不只有一个处理器，进程的每一个线程将有一个不同的理想处理器。KeInitializeThread 接着设置线程的状态为“已初始化”并且为线程初始化机器的硬件环境，包括环境、陷阱和异常帧。线程的环境被设置，这样线程将由在内核模式下系统范围内的启动例程 KiThreadStartup 启动（见步骤 6a）。

h）调用任何的在系统范围内注册的线程创建通知例程。

i）线程的访问令牌被设置为指向进程的访问令牌，并且进行访问检查以决定调用程序是否有权创建线程。如果在本机进程中创建线程，这个检查就总是成功的，但若使用 CreateRemoteThread 在另一个进程中创建线程，检查就可能不会成功。

4）CreateThread 通知 Win32 子系统有关新线程的信息，并且子系统为新线程做一些设置工作。

5）线程句柄和线程 ID（在步骤 3 中产生）被返回到调用程序。

6）除非调用程序使用 CREATE_SUSPENDED 标志设置创建线程，否则现在线程将被恢复以便能够被调度执行。当线程开始运行时，它在调用实际的用户指定的起始地址之前执行下面的额外步骤（在新线程的环境中）（线程创建的这个最后部分的流程图在图 6.17 中显示）。

a）KiThreadStartup 把线程的 IRQL 级别从 Dispatch/DPC 级别降低到 ANC 级别，然后调用系统初始线程例程 PspUserThreadStartup。用户指定的线程起始地址作为参数传递给这个例程。

b）系统初始线程例程启用工作集扩展，然后把用户模式 APC 排队来运行映像加载程序初始化例程（Ntdll.dll 中的 LdrInitializeThunk）。IRQL 降低到 0，这样就引起挂起的 APC 启动。

c）加载程序初始化例程然后执行一些额外的线程特定的初始化步骤，例如调用已加载的DLL以把新线程通知它们（Win32子系统DLL，如USER32、KERNEL32和GDI32的初始化细节不包括在本书的这个版本内）。

d）如果进程有附接的调试器，线程的启动例程就挂起进程中其他所有的活动线程，并通知Win32子系统，这样它就能够发送线程启动调试事件（CREATE_THREAD_DEBUG_INFO）到相应的调试器进程。然后它等待Win32子系统接收来自调试器的回答（通过ContinueDebugEvent函数），当Win32子系统答复后，所有的线程继续执行。

e）最后，主线程在用户模式映像运行的入口点开始执行。当使用陷阱帧（在初始化内核线程块的早些时候创建的）启动线程执行的陷阱被取消后，执行开始。该陷阱帧指定先前的模式为用户模式，并指定PC作为线程的起始地址。

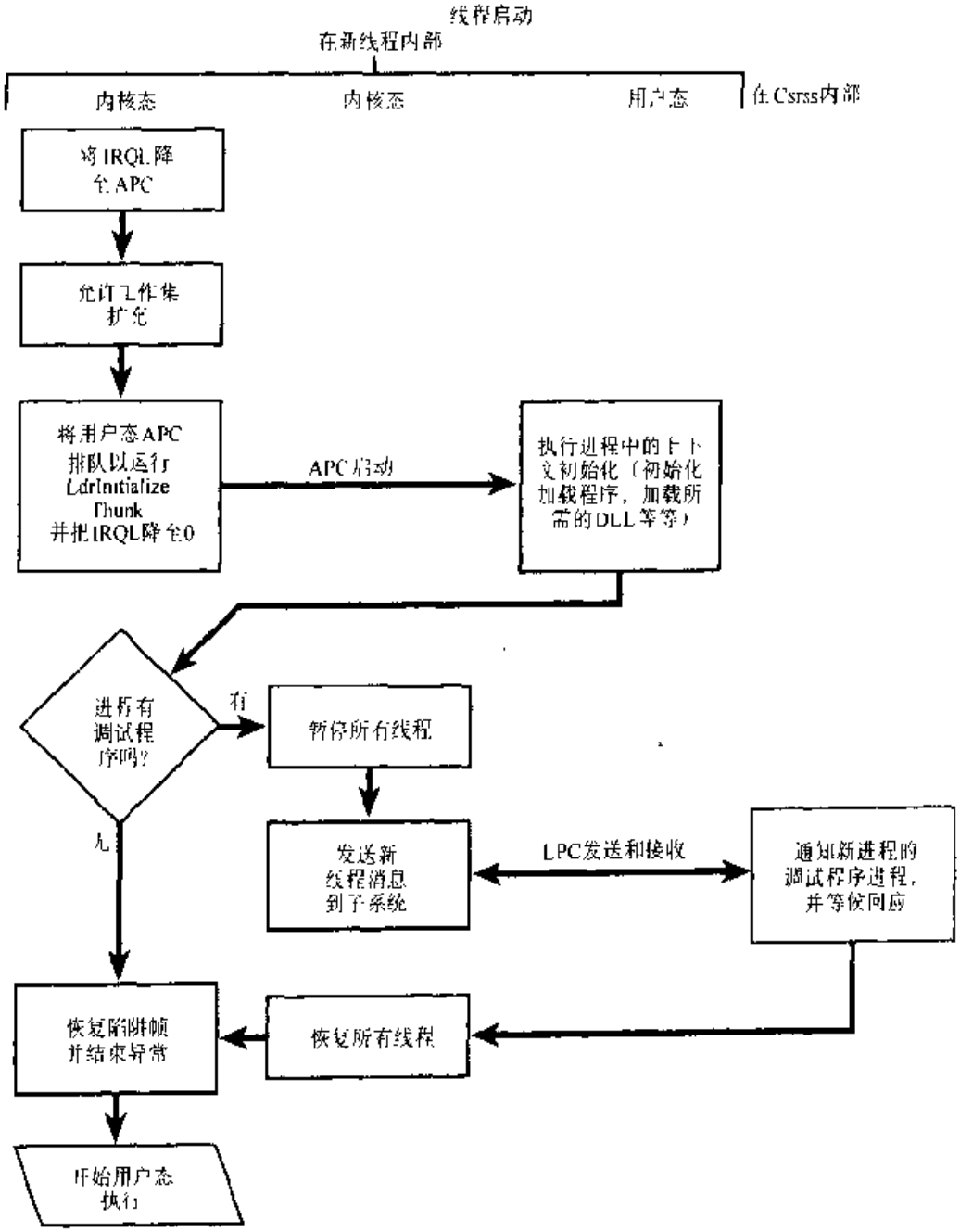


图6-17 在环境中初始化线程

6.5 线程调度

本节将介绍 Windows 2000 的调度规则和算法。前一部分集中介绍了在 Windows 2000 上的调度工作以及主要术语的定义，然后从 Win32 API 和 Windows 2000 内核两个角度介绍了 Winodws 2000 的优先级。在回顾了相关的 Win32 函数以及 Windows 2000 实用程序和与调度有关的工具之后，将介绍组成 Windows 2000 调度系统详细的数据结构和算法。

6.5.1 Windows 2000 调度概述

Windows 2000 实现了一个优先级驱动的、可抢先的调度系统——即总是运行优先级最高的（就绪（ready））线程，此时可能出现警告，所选择运行的线程可能受允许线程在它上面运行的处理器的限制，这种现象称作“处理器相似性”（processor affinity）。默认情况下，线程可在任何可用的处理器上运行，但你可以通过使用 Win32 调度函数中的一个来改变处理器相似性。

实验：查看就绪线程

你可以用内核调试器！ready 命令查看就绪线程列表。这个命令显示在每一个优先级就绪的线程或线程列表。在下面的例子中，有两个线程在优先级 10 就绪，并且有 6 个在优先级 8 就绪。因为这个输出是用 LiveKd 在单处理器上产生的，所以当前线程将始终是内核调试程序（Kd 或者 I386kd）。

```
kd> !ready 1
Ready Threads at priority 10
 THREAD 810de030  Cid 490.4a8 Teb: 7ffd9000  Win32Thread: e297e008 READY
 THREAD 81110030  Cid 490.48c Teb: 7ffde000  Win32Thread: e29425a8 READY
Ready Threads at priority 8
 THREAD 811fe790  Cid 23c.274 Teb: 7ffdb000  Win32Thread: e258cda8 READY
 THREAD 810bec70  Cid 23c.50c Teb: 7ffd9000  Win32Thread: e2ccf748 READY
 THREAD 8003a950  Cid 23c.550 Teb: 7ffda000  Win32Thread: e29a7ae8 READY
 THREAD 85ac2db0  Cid 23c.5e4 Teb: 7ffd8000  Win32Thread: e297a9e8 READY
 THREAD 827318d0  Cid 514.560 Teb: 7ffd9000  Win32Thread: 00000000 READY
 THREAD 8117adb0  Cid 2d4.338 Teb: 7ffaf000  Win32Thread: 00000000 READY
```

当一个线程被选择运行时，它将运行被称作“时间片”（quantum）的一段时间。时间片指的是在 Windows 2000 中断线程以查找是否有其他相同优先级的线程在等待运行或者线程的优先级需要降低之前，允许一个线程运行的时间长度。线程不同，它的时间片值也不同（在 Windows 2000 Professional 与 Windows 2000 Server 之间也不同）。（6.5.8 节“时间片”详细描述了时间片）。线程有可能不会用完它的时间片。由于 Windows 2000 实现一个可抢先的调度程序，如果具有较高优先级的其他线程准备运行，那么当前运行的线程会在用完它的时间片之前被抢先。实际上，可以选择线程在下一次运行，甚至在它的时间片开始前被抢先！

Windows 2000 调度代码在内核中实现。没有单个的“调度程序”模块或例程，相反代码散布在发生调度相关事件的内核中，执行这些任务的例程被一起称作内核的“调度程序”（dispatcher）。线程调度发生在 DPC/调度一级，它可由以下任何一个事件触发：

- 线程由就绪变为执行——例如，新创建的线程或刚从等待状态释放的线程。

- 由于线程的时间片结束，所以它离开运行状态，终止运行或进入等待状态。
- 系统服务调用或者 Windows 2000 自身改变了优先级值所造成的线程优先级的改变。
- 正在运行的线程的处理器相似性的改变。

在每个这样的接合点，Windows 2000 必须决定哪一个线程是要执行的下一个线程。在 Windows 2000 选择运行一个新的线程时，将对它进行“环境切换”（Context Switch）。环境切换是保存与正在运行的线程相关的非永久性的机器状态，加载另一个线程的非永久性状态并开始执行新线程的过程。

实验：改变线程调度状态

可以使用 Performance 工具来查看线程调度状态的改变。在调试一个多线程应用程序时，如果你不能确定进程中线程运行的状态，这个功能是很有用的，为了能够使用 Performance 工具观察线程调度阶段变化，遵循如下步骤：

1）运行 Microsoft Notepad 程序（Notepad.exe。）

2）从 Start 菜单中选择 Programs，启动 Performance 工具，然后选择 Administrative Tools/Performance。

3）如果你处于其他视图中，请选择图表视图。

4）右击图形，选择 Properties。

5）点击 Graph 标签，改变图形垂直范围调节最大值为 7（正如你在性能计数器上的解释文本看到的那样，线程状态从数字 0 变化到 7）。点击 OK。

6）点击工具栏上的 Add 按钮启动弹出 Add Counters（添加计数器）对话框。

7）选择线程性能对象，然后选择 Thread State 计数器，点击 Explain 按钮查看有关值的定义，见图 6-18。

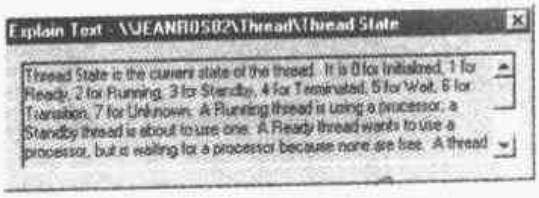


图 6-18 查看有关值的定义

8）在 Instance 框中，向下滚动直到你看见 Notepad 进程（notepad/0）；选择它，然后点击 Add 按钮。

9）在 Instance 框中回滚倒 Mmc 进程（运行了 Monitor ActiveX 控制的 Microsoft Management Console 进程），选择所有的线程（mmc/0，mmc/1 等等），并通过点击 Add 按钮将它们添加到图中。在你点击 Add 按钮之前，你应该看到类似图 6-19 所示的对话框。

10）现在通过点击 Close 关闭 Add Counter 对话框。

11）你应该能够看到 Notepad 线程的状态（图 6-20 中最上面的一行）为 5，如同在图 6-18 对话框中的解释文本所示，它代表等待状态（线程正在等待 GUI 输入）。

12）注意 Mmc 进程的一个线程（运行 Performance Tool 插件）处于运行状态（状态 2）。这是一个可以查询线程状态的线程，因此总是显示处于运行状态。

13）你将不会看到处于运行状态的 Notepad（除非在多处理器系统中），这是由于在收集你正在监视的线程的状态时，Mmc 总是处于运行状态。

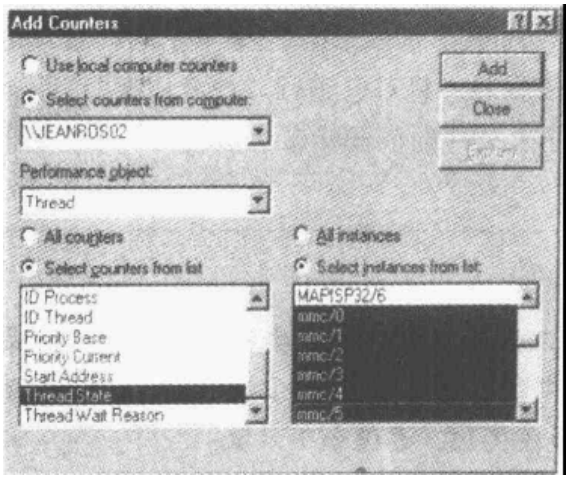


图 6-19　Add Counter 对话框

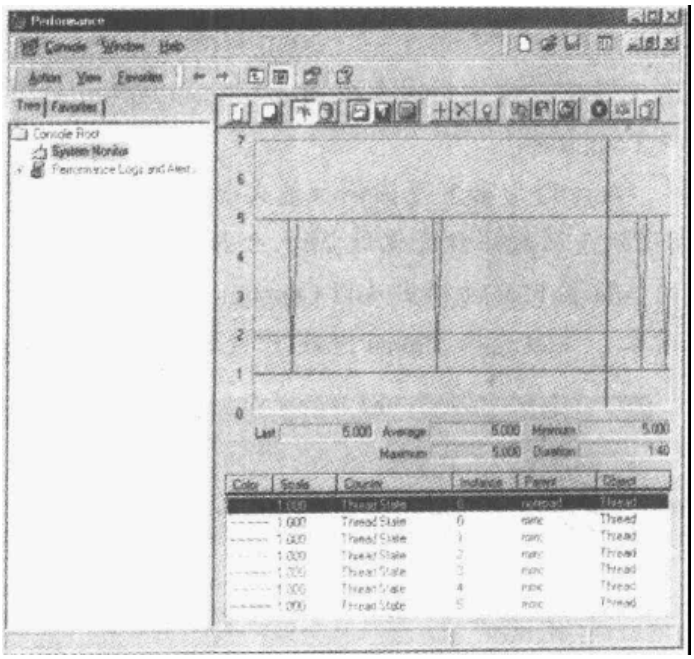


图 6-20　代表等待状态的解释文本

6.5.2　优先级

如图 6－21 所示，Windows 2000 在内部使用 32 个优先级，范围从 0 到 31。这些值的划分如下：

- 16 个实时级别（16～31）。
- 15 个变量级别（1～15）。
- 1 个系统级别（0），为清零页面线程保留。

线程优先级是从两个不同的角度分配的：Win32 API 角度和 Windows 2000 内核角度。Win32 API 首先是依据在创建进程时分配的优先级等级（Real time、High、Above Normal、Below normal 和 Idle）组织进程，然后依据进程中各个线程的相对优先级（Time－critical、Highest、Above－

normal、Normal、Below－normal、Lowest 和 Idle）组织进程。

在 Win32 API，每个线程的优先级是以它的进程优先级等级及其相对线程优先级的结合为基础的。图 6-22 给出了从 Win32 的优先级到内部 Windows 2000 数字优先级的映射。

图 6-22 所示的优先级是线程的基本优先级。线程启动时就继承进程的基本优先级，它可以使用 Task Manager（任务管理器）（在下一页“相关工具”一节中介绍）来改变，也可以使用 Win32 函数 SetProcessPriority 来改变。

一般地，进程基本优先级（并从此启动线程基本优先级）默认将是每个进程优先级

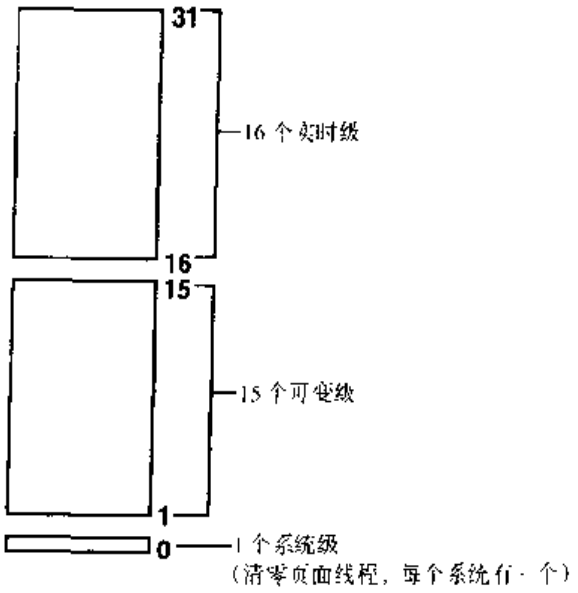


图 6－21 线程优先级

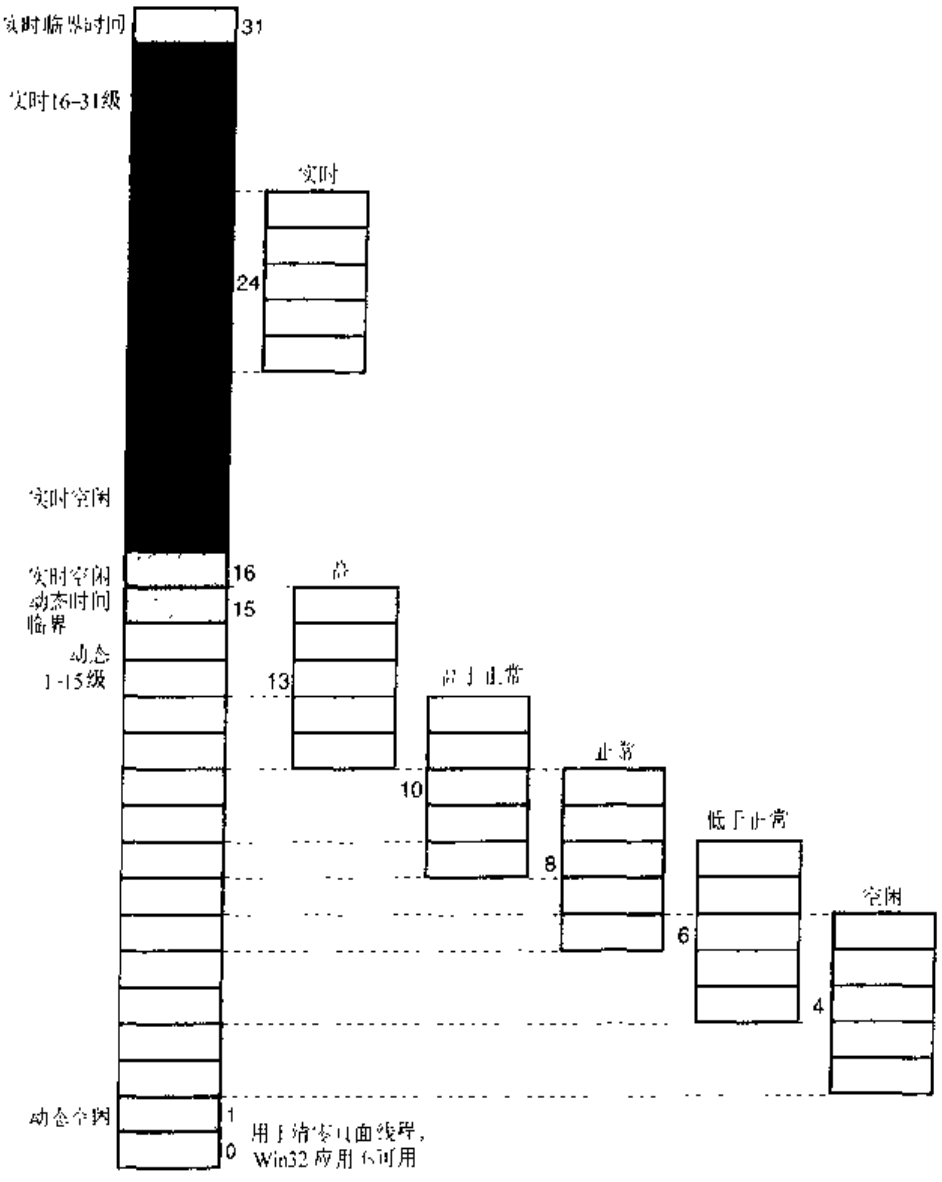


图 6-22 Win32 和 Windows2000 内核优先级

范围（24、13、10、8、6 或 4）中间的某个值。然而，一些 Windows 2000 系统进程（例如会话管理器、服务控制程序以及本机安全验证服务器）具有的基本进程优先级要比默认的 Normal 等级（8）高一些，因此，在这些进程中的线程就能以高于默认值 8 的优先级启动。系统进程通过使用内部的 Windows 2000 函数将其进程基本优先级设置为一个数字值，而不是默认启动 Win32 基本优先级来实现此功能。

一个进程只能有一个单一优先级值（基本优先级），而每个线程却具有两个优先级值：当前优先级和基本优先级。对于处在动态范围（1 到 15）内的线程的当前优先级可能（通常总是）高于基本优先级。处于实时范围（16 到 31）内的线程，它们的优先级不会被 Windows 2000 调整，所以它们的基本优先级和当前优先级总是相同的。

6.5.3 Win32 调度 API

表 6－15 给出了与线程调度有关的 Win32 API 函数。详细信息请参阅 Win32 API 参考文档。

表 6-15 与调度有关的 API 和它们的函数

API	功 能
Suspend/ResumeThread	从执行中挂起或恢复暂停的线程
Get/SetPriorityClass	返回或设置进程的优先级等级（基本优先级）
Get/SetThreadPriority	返回或设置线程的优先级（相对于进程的基本优先级）
Get/SetProcessAffinityMask	返回或设置进程的相似性掩码
SetThreadAffinityMask	为一个特殊的处理器集设置线程的相似性掩码（必须是进程相似性掩码的子集）来限制它只能运行在这些处理器上
Get/SetThreadPriorityBoost	返回或设置 Windows 2000 的暂时提高线程优先级的能力（仅适用于处于动态范围内的线程）
SetThreadIdealProcessor	为特殊线程建立一个理想处理器，但并不将线程限制在该处理器
Get/SetProcessPriorityBoost	返回或设置当前进程的默认优先级提高控制状态。（此函数用来在创建线程时设置线程的优先级提高控制状态）
SwitchToThread	给在当前处理器上准备运行的另一线程一个时间片来让它执行
Sleep	以指定的时间间隔（以毫秒计算［msec］使当前线程进入等待状态，值为 0 时就放弃剩余的线程时间片
SleepEx	使当前线程进入等待状态，直到 I/O 完成回调结束，一个 APC 进入了线程队列，或者是指定的时间间隔结束

6.5.4 相关工具

你可以使用 Task Manager、Pview 或 Pviewer 来查看（并改变）基本继承优先级。你可以使用 Performance Tool 或 Pstat 来查看基本进程优先级的数值。你可以使用 Performance Tool、Pview、Pviewer 和 Pstat 查看线程优先级。然而没有通用的使用程序来改变相对的线程优先级别。表 6.16 列出了与线程调度相关的工具。

表 6-16 与调度相关的工具

对 象	Taskman	Perfmon	Pviewer	Pview	Pstat	KD! thread
进程优先级	√		√	√		
进程基本优先级		√			√	
线程基本优先级		√				
线程当前优先级		√	√	√	√	√

要为一个进程指定起始优先级等级的唯一方法是在 Windows 2000 命令提示符下使 start 命令。

实验：检查并指定进程和线程的优先级

尝试以下实验：

1）在命令提示符下键入 start/realtime notepad，打开 Notepad。

2）运行支持工具中的进程查看器（Pviewer.exe），从进程列表中选择 Notepad.exe，如图 6-23 所示。注意到 Notepad 中线程的动态优先级是 24。这与图 6-22 所示的 real-time 值相符合。

图 6-23 进程查看器查看线程的动态优先级

3）Task Manager（任务管理器）可以显示类似的信息。按下 Ctrl + Shift + Esc 启动 Task Manager，然后来到 Processes 选项。右击 Notepad.exe 进程，选择 Set Priority 选项。你可以看到 Notepad 的进程优先级是实时的，如图 6-24 所示：

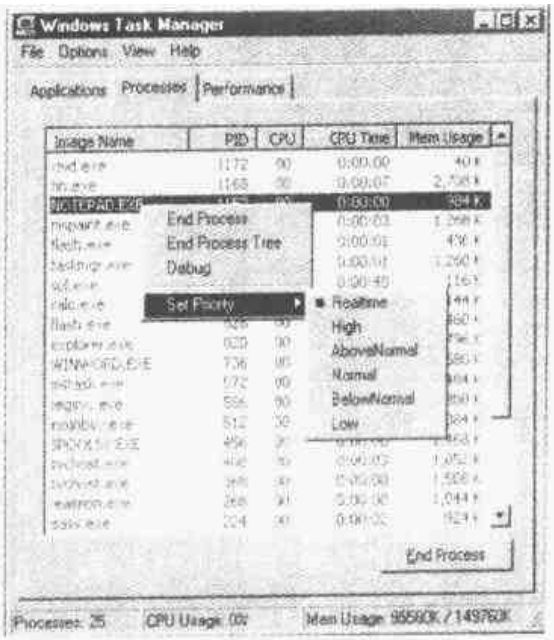


图 6-24 Taxk Manage 显示 Notepad 的进程优先级

6.5.5 实时优先级

在任何应用程序中都可以提高或降低动态范围内的线程优先级，然而必须具有“增加调度优先级”（increase scheduling priority）特权才能进入实时范围。（如果在不具备该特权的情况下试图使进程进入实时优先级等级，操作并不会失败——将使用“高级”等级的优先级。）

请注意，许多重要的 Windows 2000 内核模式系统线程都是在实时优先级范围内运行的。如果在此范围内运行时间过长，就可能阻塞内存管理器、高速缓存冲管理器、本机和网络文件系统，甚至其他设备驱动程序中的关键系统函数。如同在前面提到的，你不能阻塞硬件中断，因为它们具有比任何线程都高的优先级，但你可以阻塞正在运行的系统线程。

对于处于实时范围的线程存在一种行为差异（将在后面“优先”一节提到）：如果它们被抢先，它们的线程时间片（quantum）将会重置。

注意 虽然 Windows 2000 具有被称作“实时”（real－time）的优先级，但它并不是真正的实时操作系统，它并不提供保证的中断等待时间，或者线程获得保证的执行时间而提供某种方法。详细信息请参阅第 3 章，还有 MSDN 库中的文章“Real－Time and Microsoft Windows NT”。

6.5.6 中断级与优先级对比

图 6-25 中的所有线程（无论运行在用户模式还是运行在内核模式）都在 IRQL 0 或 1 上运行（有关 Windows 2000 使用中断级的信息请参阅第 3 章）。用户模式线程通常在 IRQL 0 上运行。只有内核模式的 APC 才会在 IRQL 1 上执行（关于 APC 的详细信息请参阅第 3 章）。而且，

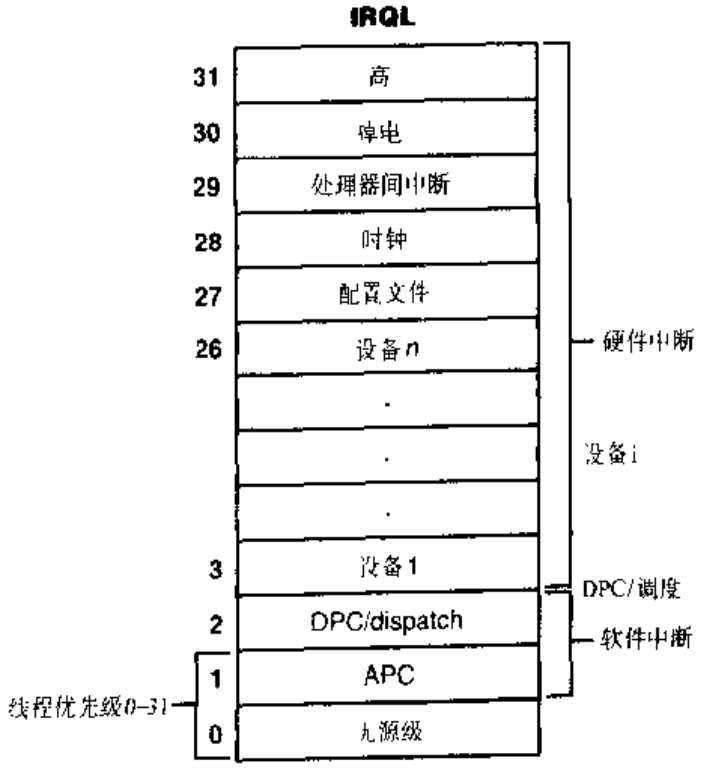


图 6-25 中断优先级和线程优先级对比

运行在内核模式的线程可以提高 IRQL。因为这个原因，不管它的优先级是多少，没有用户线程可以阻塞硬件中断（尽管高优先级实时线程可以阻塞重要的系统线程的执行）。

线程调度决定是在 DPC/调度上做出的。这样，在内核决定下一个将运行哪一个线程时，不会有线程被执行，并且与调度有关的信息（例如优先级）也不能被改变。在多处理器系统中，访问线程调度数据结构是通过“调度程序”自旋锁（KiDispatcherLock）来同步的。

6.5.7 线程状态

在理解线程调度算法与数据结构之前，你需要理解线程可能处于的不同的执行状态。图 6-26 举例说明了 Windows 2000 线程状态的转变。关于在每一个转变中将发生什么的细节将在稍后描述。

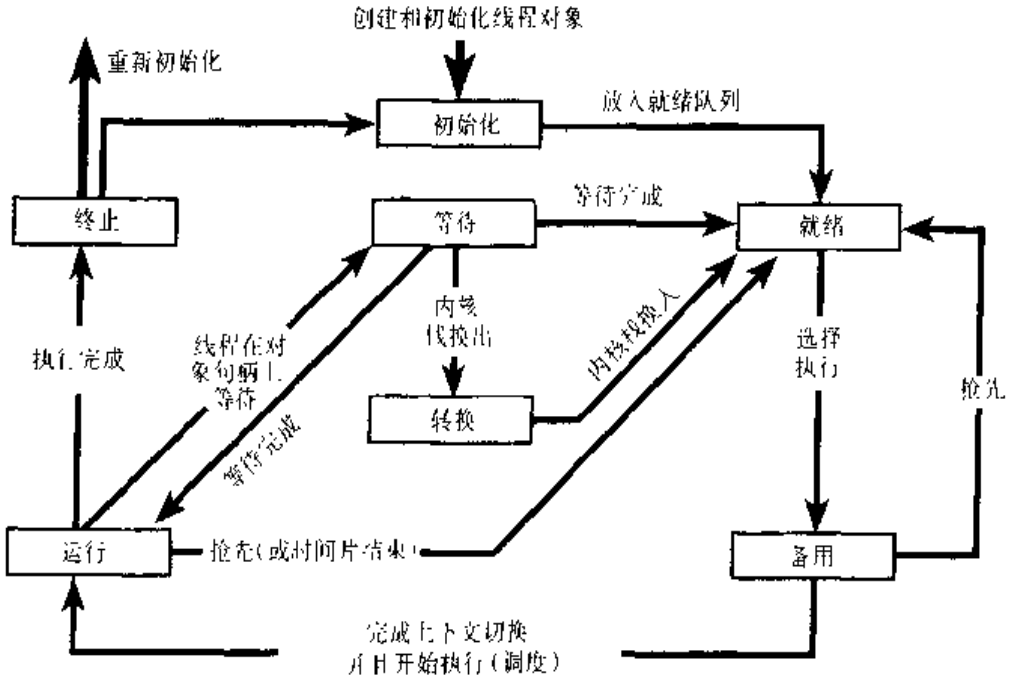


图 6-26 线程状态

线程状态有如下几种：

■ 就绪 当查找要执行的线程时，调度程序只考虑处于就绪状态的线程交换区。这些线程只是简单地等待执行。

■ 备用 处于备用状态的线程是某一特定处理器的下一个执行对象。当条件合适时，调度程序对该线程执行环境切换。系统中每个处理器上只能有一个处于备用状态的线程。

■ 运行 一旦调度程序对线程执行完环境切换，线程就进入运行状态并开始执行。线程的执行持续下去，直到内核抢先它去运行一个更高优先级的线程，或是它的时间片结束，或是它已终止，或是它自动进入等待状态。

■ 等待 一个线程可以按几种方式进入等待状态：一个线程可以自动等待一个对象以便同步它的执行，操作系统（如 I/O 系统）代表线程等待，或者是环境子系统引导线程将自己挂起。当线程的等待状态结束时，根据其优先级或者开始运行或者返回到就绪状态。

■ 转换 在线程准备执行而其内核堆栈被调页出内存之外，线程进入转换状态。例如，线

程的内核堆栈可能被调页出内存之外。一旦其内核堆栈被调回内存，线程就进入就绪状态。

■ 终止 线程执行完就进入终止状态。一旦结束，线程对象可能被删除，也可能不被删除（对象管理器设定何时删除对象的策略）。如果执行程序有一指向线程对象的指针，它就可以将线程对象重新初始化，并再次使用它。

6.5.8 时间片

本章前面提到过，时间片是在 Windows 2000 检查具有相同优先级的其他线程是否应该开始运行之前，一个线程开始运行的一段时间。如果一个线程完成了它的时间片，而且没有其他具有相同优先级的线程，则 Windows 2000 将重新调度线程来运行另一个时间片。

每个线程都有一个时间片值来代表时间片结束之前线程可运行的时间长度。这个值并非时间长度而仅仅是一个整数值，称作“时间片单位”（quantum units）。

1. 时间片统计

默认情况下，在 Windows 2000 Professional 上线程启动的时间片值为 6，而在 Windows 2000 Server 上线程启动的时间片值为 36（后面我们将会解释如何改变这些值）。在 Windows 2000 Server 上时间片值较大的原因是为了减少环境切换的时间。通过具有较长时间片，作为客户请求的结果而被唤醒的服务器的应用程序有足够的时间完成请求并在它的时间片结束之前恢复到等待状态。

每次时钟中断发生时，时钟中断例程都会从线程的时间片中减去一个固定的值（3）。如果没有剩余的线程时间片，那么将触发时间片结束处理，而另一个线程就可能被选择去运行。在 Windows 2000 Professional 中，由于时钟中断每发生一次就减去 3，所以线程将运行 2 个时钟间隔；而在 Windows 2000 Server 上线程将运行 12 个时钟间隔。

当时钟中断发生的时候，即使系统处于 DPC/调度级别或更高（例如，如果一个 DPC 或者一个中断服务例程正在执行），当前线程仍旧会减少它的时间片，即使它已经在一个完整的时钟间隔内运行了。如果它没有完成或者在时钟间隔定时器中断发生之前设备中断或 DPC 发生了，线程可能就不能将它们的时间片减少。

时钟间隔的长度随着硬件平台不同而变化。时钟中断的频率同 HAL 保持一致，而不是内核。举例，对大多数 x86 单处理器的时钟间隔是 10 微秒，而对大多数 x86 多处理器是 15 微秒。

实验：决定时钟间隔频率

Win32 的 GetSystemTimeAdjustment 函数返回时钟间隔。为了决定时钟间隔，从 www.sysinternals.com（附带 CD 上 \ Sysint \ Clockres）运行 Clockres 程序。下面是单处理器 x86 系统的输出：

```
C:\>clockres

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

The system clock interval is 10 ms
```

时间片以每个时钟滴答的三个时间单元的倍数而不是单个单元来表达的原因是为了考虑在等待完成时损失的部分时间片。当一个线程具有小于14的基本优先级执行一个等待函数（例如 WaitForSingleObject 或者 WaitForMultipleObjects），它的时间片将减少1个时间片单元。（那些优先级高于14的线程在一个等待后将恢复它们的时间片）。

这样的部分减少表明一种情况：一个线程在时钟间隔定时器发生之前进入等待状态。如果不作这种调整，那么线程就不可能将它们的时间片减少。例如，如果一个线程运行，进入等待状态，再运行，然后再进入等待状态，但当时钟间隔定时器发生时却不是当前运行的线程，它就不会因为它的运行而损失任何时间片。

2. 控制时间片

注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ PriorityControl \ Win32PrioritySeparation 允许你定义线程时间片的相对长度（长或短）以及是否处于前台运行的线程应该增加它们的时间片（如果是，增加的数量）。这个值包含6个位，划分为3个2-位字段，如图6-27所示。

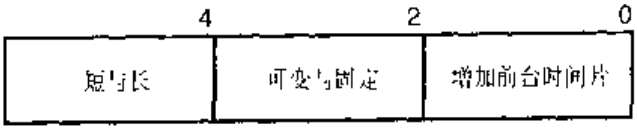


图6-27 Win32PrioritySeparation 注册表键值的字段

■ 短与长，值1表示长，值2表示短。值0或者3说明使用缺省的值（Windows 2000 Professional 为短，Windows 2000 Server 为长）。

■ 可变与固定，值1表示前台进程的时间片可变，值2表示前台进程的时间片不能变化，值0或3表示使用缺省的值（Windows 2000 Professional 为可变，Windows 2000 Server 为固定）。

■ 增加前台时间片，这个字段的值为0，1或2（3是无效的，等价于2），它是一个具有三个选项的时间片表的索引，这个时间片表用作获取前台进程中的线程的时间片。后台进程中线程的时间片为表中的第一项。该字段的值保存在内核变量 PsPrioritySeparation 中。

前台进程包含一个具有在焦点中的窗口的线程。当前台窗口改变为具有比 Idle 优先级高的进程中的某个线程中时，Win32 子系统通过使用 Win32PrioritySeparation 注册表键值的低两位作为一个名为 PspForegroundQuantum 的三元素数组的索引，来改变那个进程中所有线程的时间片值。这个数组包含了由 Win32PrioritySeparation 注册表键值的其他两个位决定的值。表6-17给出了 PspForegroundQuantum 的可能的设置。

表6-17 时间片值

	短			长		
可变	6	12	18	12	24	36
固定	18	18	18	36	36	36

Windows 2000 增加前台线程的时间片而不是优先级的原因可以用下面的例子说明，这个例子表明如果采用增加优先级的策略可能导致的问题。假定你启动一个长时间运行的电子数据表格计算，然后切换到频繁使用 CPU 的应用程序（比如图形丰富的游戏），如果在前台运行的游戏进程增加它的优先级，那么在后台运行的电子数据表格将获得很少的 CPU 时间。而增加游戏进程的时间片则不会妨碍电子数据表格程序的运行，但是又对游戏进程运行有利。如果你确实想要运行一个具有比其他交互式进程更高优先级的交互应用程序，你总可以利用任务管理器来改变其优先

级为 Above Normal 或 High（或者在命令行启动应用时，采用 start/abovenormal 或者 start/high）。

当你通过直接修改 Win32PrioritySeparation 注册表键值来设置时间片长度，你可以选择任意组合。当你使用 Performance Options 对话框，你只能从两个组合中选择。为了看到这个对话框（如 6-28 所示），打开控制面板的系统功能（右击 My Computer 并且选择 Properties），点击 Advanced 标签，然后点击 Performance Options 按钮！

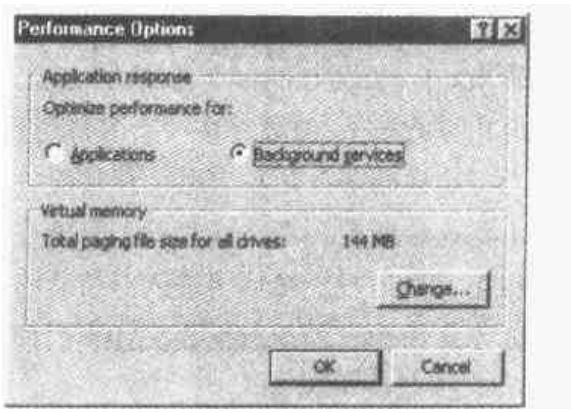


图 6-28 调整时间片设置

在 Optimize Performance For 下的 Applications 选项指定使用短的、可变的时间片——这是 Windows 2000 Professional 的缺省值。Background Services 选项指定使用长的、固定的时间片——Windows 2000 Server 的缺省值。如果你在 Windows 2000 Advanced Server 或 Windows 2000 Datacenter Server 上安装 Terminal Services（终端服务）并配置服务器为应用服务器，这个设置就作相应改变使得对应用程序而言是最优的。

6.5.9 调度数据结构

为了作出线程调度的决策，内核维护了一组数据结构，总称为调度程序数据库（dispatcher database），如图 6-29 所示。调度程序数据库跟踪哪些线程等待执行，哪个进程正在执行那些

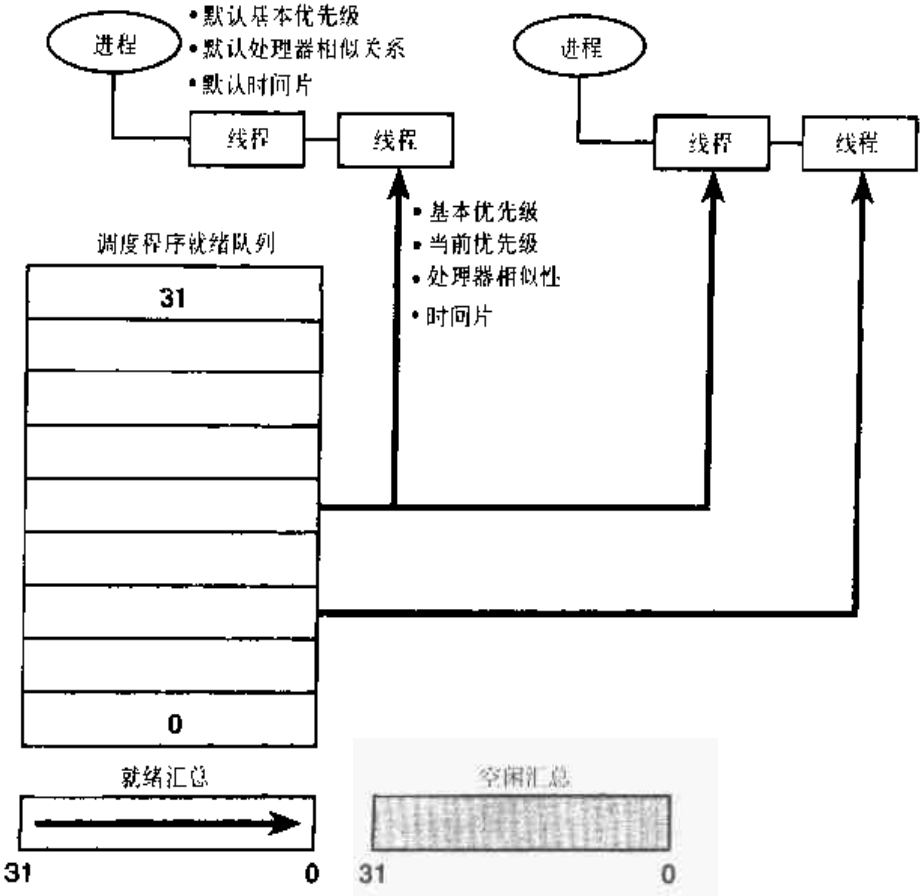


图 6-29 调度程序数据库

线程。在调度程序数据库中最重要的结构是调度程序就绪队列（dispatcher ready queue）（在KiDispatcherReadyListHead中）。这个队列事实上是一组队列，每个调度优先级一个队列。队列包含那些处于就绪状态的线程，这些线程正在等待调度执行。

为了更快地选择哪个线程运行或抢先，Windows 2000维护一个32位的掩码称为就绪汇总（ready summary）（KiReadySummary）。每一个位说明相应的优先级是否有一个或多个线程处于就绪队列中（位0表示优先级为0，如此类推）。Windows 200还维护另一个掩码空闲汇总（idle summary）（KiIdleSummary），其中的每一位表示一个空闲处理器。

如前面提到的，线程调度发生在DPC/调度级。除了防止其他线程运行，处于DPC/调度级别上可以同步对调度程序数据库的访问。然而，在一个多处理器系统下，对调度程序数据库的改变还需要获得内核调度程序的自旋锁（KiDispatcherLock）。表6-18给出了与线程调度相关的内核模式内核变量。

表6-18 给出了与线程调度有关的内核模式内核变量

变 量	类 型	说 明
KiDispatcherLock	自旋锁	调度程序自旋锁
KeNumberProcessors	字节	系统中活动处理器的个数
KeActiveProcessors	位掩码（32位）	系统中活动处理器的位掩码
KiIdleSummary	位掩码（32位）	空闲处理器的位掩码
KiReadySummary	位掩码（32位）	拥有一个或多个就绪线程的优先级的位掩码
KiDispatcherReadyListHead	32个列表项的数组	用于32个就绪队列列表头

6.5.10 调度方案

Windows 2000把“谁将得到CPU?”这个问题建立在线程优先级的基础上，实际中它是如何实现的呢？以下部分举例说明了在线程级别上如何实现优先级驱动的可抢先的多任务机制。注意到Windows 2000处理调度决定时在多处理器系统和单处理器系统是不一样的。这些区别将在本章后面“对称多处理器的线程调度”一节中解释。

1. 自愿切换

首先，通过调用许多Win32等待函数（例如WaitForSingleObject或WaitForMultipleObjects中的一个）进入对某些对象（例如事件、互斥体、信号量、I/O完成端口、进程、线程、窗口消息等等）的等待状态，线程或许会自动放弃对处理器的使用。第3章详细描述了等待对象。

自愿切换大致相当于线程点了一个在快餐柜台上没有准备就绪的项目。在准备第一个线程的汉堡包时，线程不会阻碍其他就餐者进入队列，它将会退让并让下一个线程执行它的例程。在汉堡包准备好之后，第一个线程就进入这个优先级就绪队列的末尾。然而，在本章稍后你会看到，大多数等待操作都会暂时提高优先级以便线程可以立刻得到它的汉堡包并开始享用。

图6-30说明在一个线程进入等待状态的同时，Windows 2000选择一个新的线程来运行。

图6-30中，最上面的方块（线程）自动放弃了处理器以便处于就绪状态的下一个线程（在“运行”列中以光环标记的）可以运行。从图中看上去放弃线程的优先级降低了，其实并非如此——它只是被移到了线程正在等待的对象的等待队列中去了。用于线程的任何剩余的时间片怎样了？当线程进入等待状态时，时间片值并没有被重新设置——事实上，当等待条件满

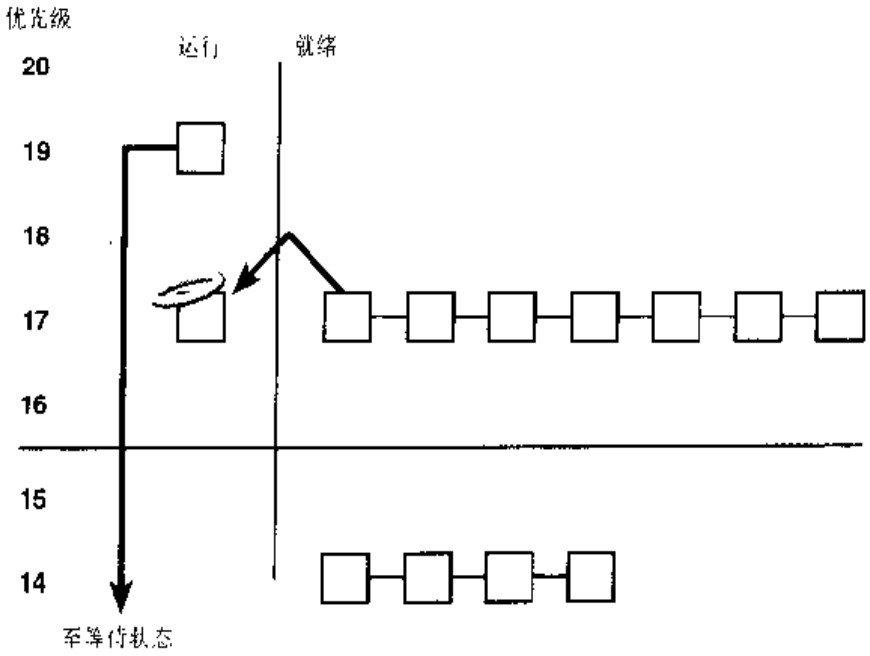


图 6-30 自愿切换

足时，线程的时间片值会减少一个时间片单位，这相当于一个时钟间隔的三分之一（除非线程运行的优先级为 14 或更高——它们在经过一次等待后时间片就会重置）。

2. 抢先

在这个调度方案中，当较高优先级的线程变成就绪而准备运行时，较低优先级的线程将被抢先。有两种原因可使得这个情况发生：

- 较高优先级线程的等待完成（其他线程所等待的事件已经发生）。
- 线程的优先级被增加或降低。

在以上两种情况中，Windows 2000 必须决定是要继续运行当前的线程，还是要使它被抢先以便允许较高优先级的线程运行。

注意 运行在内核模式的线程可以被在用户模式运行的线程抢先——这与线程正在运行的状态无关。线程的优先级是决定性的因素。

当线程被抢先之后，就被放在它所运行的优先级就绪队列的前面。这样，再次开始运行时，它就可以完成时间片。当运行在动态优先级的线程完成它们的时间片后，在实时级范畴运行的线程在它们重新运行后将它们的时间片设置为完全的时间片。图 6-31 说明了这个情况。

如图 6-31 所示，优先级为 18 的线程从等待状态出现并重新占有了 CPU，导致已经在运行的线程（优先级为 16）被返回到就绪队列的前端。注意，被返回的线程不会回到队列的尾部，而是在队列的前端。在抢先的线程完成运行之后，被返回的线程就可以完成它的时间片了。在本例中，线程处于实时范围，如同在“提高优先级”解释的那样，在实时范围内，为线程提高动态优先级是不允许的。

如果自愿切换大致相当于这样一个过程，即在第一个线程等待进餐时，让另一线程点它的午餐，那么抢先就可以近似地比作由于这时美国总统刚好进来并且点了份汉堡包，线程被返回到它的位置。被抢先的线程并不会被移到行的后边，而只是在总统得到进餐时被简单地移到一边。一旦总统离开，第一个线程点它的菜。

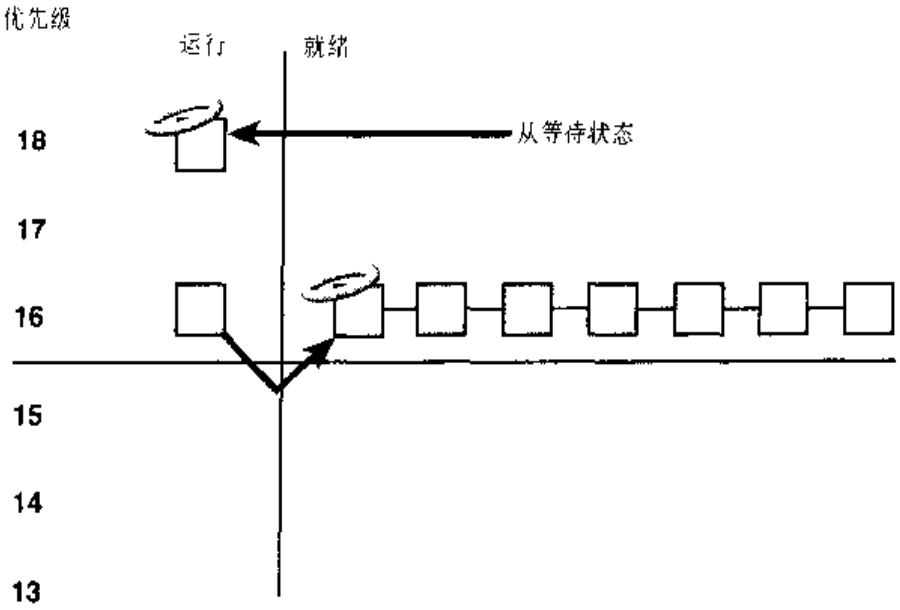


图 6-31 抢先线程调度

3. 时间片结束

当运行的线程用完它的 CPU 时间片以后，Windows 2000 必须决定是否应该降低线程的优先级，然后决定是否在该处理器上调度另一个线程。

如果线程的优先级降低了，Windows 2000 就会寻找一个更合适的线程来调度（例如，更合适的线程可能是处于就绪队列中的某个线程，它比当前运行的具有新优先级线程的优先级还要高）。如果线程的优先级没有降低，则 Windows 2000 将选择处于同一优先级就绪队列中的下一个线程，并且将以前运行的线程移到队列的尾部（赋予它一个新时间片值，并把它从运行状态改变为就绪状态）。图 6-32 说明了这种情况。如果在同一优先级上没有其他线程可以运行，则该线程就运行另一个时间片。

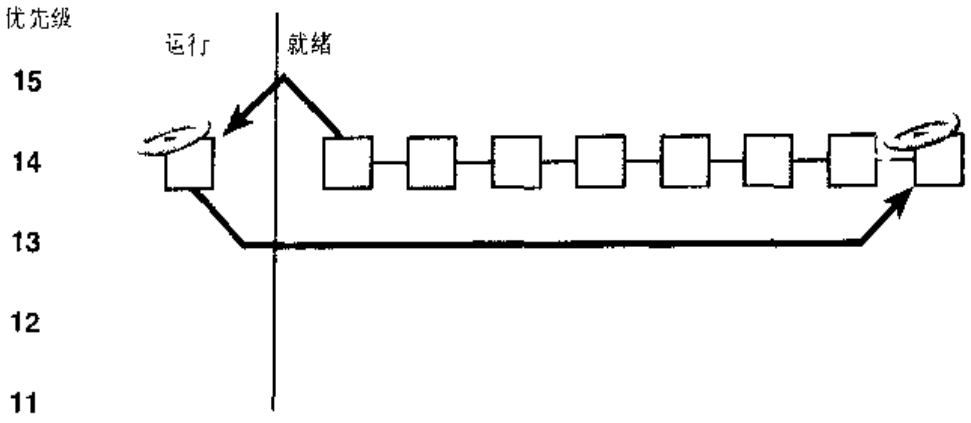


图 6-32 时间片结束线程调度

4. 终止

当线程完成运行以后（或者因为调用 ExitThread 而从主例程返回，或是使用 TerminateThread 将其终止），它就会从运行状态移动至终止状态。如果在线程对象上没有打开句柄，线程就会从进程线程列表中被删除，并且与它相关的数据结构将被重新分配和释放。

6.5.11 环境切换

线程环境和环境切换的过程因处理器体系结构的不同而不同。一个典型的环境切换要求保存并重新加载下面的数据：

- 程序计数器。
- 处理器状态寄存器。
- 其他寄存器内容。
- 用户及内核堆栈指针。
- 指向线程运行的地址空间的指针（进程的页表目录）。

通过将以上信息推入当前（旧线程）内核模式堆栈，更新堆栈指针并且在旧线程的KTHREAD块中保存堆栈指针，内核保存旧线程中的信息。内核堆栈指针被设置为新线程的内核堆栈并且内核加载新线程的环境。如果新线程在不同的进程中，它还要加载其页表目录地址到特殊的处理器寄存器，这样它的址空间才可用（详见第7章的地址变换）。如果需要传送的内核APC正被挂起，则申请一个IRQL 1中断。否则，控制将传递给新线程的被恢复的程序计数器，线程将恢复执行。

6.5.12 空闲线程

当CPU内没有可运行的线程时，Windows 2000将调度每个CPU的空闲线程。因为在多处理器系统中，一个CPU可能正在执行一个线程，而其他CPU没有线程在执行，所以每个CPU都被分配了一个空闲线程。Windows 2000报告空闲线程的优先级为0。然而，实际上这样的线程是没有优先级的，因为只在CPU中没有线程运行时它们才运行。（请记住，每个Windows 2000系统中实际上只有一个线程在0优先级运行——清零页面线程。）事实上，空闲在IRQL 2上循环，轮询需要做的工作：传送延迟过程调用（DPC）或调度线程。尽管因体系结构间的差异而使流程的一些细节不同，但空闲线程控制的基本流程如下：

1）启用或禁用中断（允许任何挂起的中断被传送）。

2）检查在处理器上是否有DPC（将在第3章描述）处于挂起状态。如果有DPC处于挂起状态，则清除挂起的软件中断并传送它们。

3）检查处理器上是否有下一个被选择运行的线程，如果有，就调度该线程。

4）调用HAL处理器空闲例程（如果需要执行任一个电源管理功能）。

不同的Windows 2000进程查看器实用程序使用不同的名称报告空闲进程。任务管理器称它为“System Idle Process”，Process Viewer称它为“Idle”，Pstat称它为“Idle Process”，Process Explode和Tlist称它为“System Process”，Qslice称它为“System Process”。

6.5.13 提高优先级

在五种情况下，Windows 2000可以提高线程的当前优先级的值：

- I/O操作完成。
- 在等待执行程序事件或信号量以后。

■ 在前台进程的线程完成一次等待操作之后。

■ 当 GUI 线程由于窗口活动而被唤醒。

■ 当一个准备运行的线程一直没能运行（CPU 饥饿）。

这些调整的目的是为了提高整个系统的吞吐能力和反应能力，同时解决潜在的不公平的调度方案。然而，如同任何调度算法一样，这些调整并不是完美的，它们不可能有利于所有应用程序。

注意 Windows 2000 不会提高处于实时范围（16～31）的线程的优先级。因此，调度同处于实时范围内的其他线程相比是可以预测的。Windows 2000 假定如果你正在使用实时进程优先级，你已经知道你在做什么。

1. 在 I/O 完成后提高优先级

Windows 2000 对某些 I/O 和等待操作的完成将给予临时的优先级提高，以便等待 I/O 的线程有更多的机会立即运行处理正在等待的任何事情，记得当一个线程苏醒时，它的剩余时间片会被减掉 1 个时间片单元，这样 I/O 线程就不会遭受不公平的对待了。虽然你可以在 DDK 头文件中（在 Wdm.h 或 Ntddk.h 中搜索 "#define IO" — 这些值列在表 6-19 中）找到推荐的增加值，实际增加的值是由设备驱动程序决定的。当设备驱动程序通过调用内核函数 IoCompleteRequest 完成一次 I/O 请求时，由它来定义增加值。在表 6-19 中，注意到那些对响应快的设备的 I/O 请求具有更高的增加值。

表 6-19 推荐的增加值

设 备	增加值
磁盘、CD-ROM、并行设备、视频	1
网络、邮箱、名字管道、串行设备	2
键盘、鼠标	6
声音	8

增加值都是在线程的基本优先级上增加的，而不是当前的优先级。如图 6-33 所示，在增加值获得以后，该线程在提升的优先级别上运行一个时间片。在线程完成它的时间片后，它就降低一个优先级别，然后运行另一个时间片。这个循环过程继续直到该线程的优先级被降回到基本优先级。一个具有较高优先级的线程仍然能够抢先于优先级提高的线程执行，但是在被中断的线程降低到下一个优先级以前，它会在被提升的优先级上继续完成它的时间片。

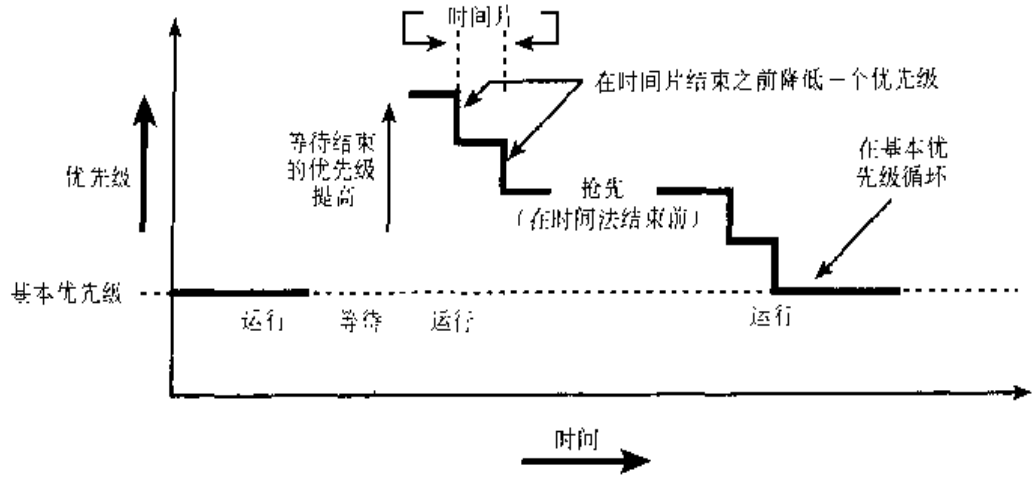


图 6-33 优先级提高和降低

正如前面所提到的，这些提高只适用于动态优先级范围（0 到 15）内的线程。不管提高幅度有多大，线程的优先级永远不会超过级别 15 而进入实时优先级范围。换句话说，一个优先级为 14 的线程如果得到 5 个提高，那么将只提升到优先级 15。优先级为 15 的线程优先级提高后优先级仍为 15。

2. 在事件和信号量的等待完成后提高优先级

当一个正在等待某个执行程序事件或信号量对象的线程的等待满足以后（因为调用某个函数 SetEvent、PulseEvent 或 ReleaseSemaphore 的调用），它的优先级提高了 1。（查看 DDK 头文件中 EVENT-INCREMENT 和 SEMAPHORE-INCREMENT 的值）等待某个事件和信号量的线程同那些等待 I/O 操作的线程有同样的理由要求提高优先级——那些因为等待事件而阻塞的线程请求 CPU 的频率要比正在使用 CPU 的线程低，这样的调整也是为了有利于平衡比例。

这种优先级的提高操作方法同前面一节中描述的 I/O 完成后的优先级提高是相同的：增加值都是在线程的基本优先级上增加的，而不是当前的优先级，提高后的优先级也绝不会超过 15，在线程降低一个优先级别之前，线程将在它剩余的时间片内在提升后的优先级上继续运行（如前面描述的那样，当线程退出一个等待后，它的时间片的减少 1），直到它的优先级别降低到初始的基本优先级。

3. 在等待之后前台线程的优先级的提高

无论何时当前台进程的线程完成了对一个内核对象的等待操作后，内核函数 KiUnwaitThread 将在当前优先级的基础上增加值 PsPrioritySeparation。（窗口系统将决定哪个进程将处于前台）。如同在时间片控制一节中描述的，PsPrioritySeparation 表示的是时间片表的索引，用来选择前台应用程序线程的时间片。

提高优先级的原因是为了增加交互式应用程序的响应能力——当前台应用程序完成一次等待，通过小小地增加它的优先级，它就会有机会立刻运行，尤其在是可能其他具有相同基本优先级的应用程序正在后台运行的时候。

与其他类型的优先级提高不同的是，这种优先级的提高同时适用于 Windows 2000 Professional 和 Windows 2000 Server，即使你采用 Win32 SetThreadPriorityBoost 函数禁止优先级提高，你也不能禁止这种优先级的提高。

实验：观察前台优先级提高和降低

利用 CPU 的 Stress 工具（在资源包和平台 SDK 中），你能够实时观察到优先级的提高。采取如下步骤：

1）打开控制面板的 System 功能（或右击 My Computer 并选择 Property），点击 Advanced 标签，并点击 Performance Options 按钮 选择 Applications 选项，这将使得 PsPrioritySeparation 获得值 2。

2）运行 Cpustres.exe。

3）运行 Windows NT 4 的 Performance Monitor（Windows 2000 资源包中的 Perfmon4.exe）。我们需要采用这个旧版本的 Performance 工具来做这个实验，因为它查询 performance counter 的值的速度要比 Windows 2000 Performance 工具快（后者的最大间隔是每秒一次）。

4）点击 Add Counter 工具条按钮（或按 Ctrl + I）弹出 Add To Chart 对话框。

5）选择 Thread 对象，然后选择 Priority Current counter。

6）在 Instance 框内，滚动列表直到你能看到 cpustres 进程。选择第二个线程（线程 1）（第一个线程是 GUI 线程）。你可以看到如图 6-34 所示：

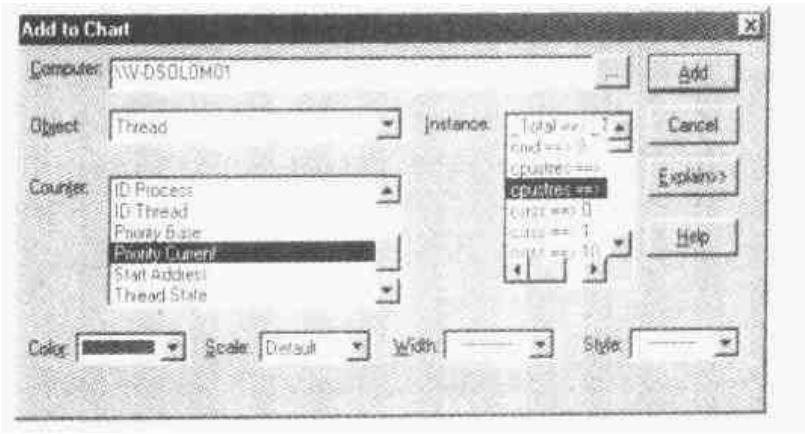


图 6-34 查看第二个线程

7）点击 Add 按钮，然后点击 Done 按钮。

8）从 Options 菜单中选择 Chart，改变垂直最大值为 16，间隔为 0.01，如图 6-35 所示，点击 OK：

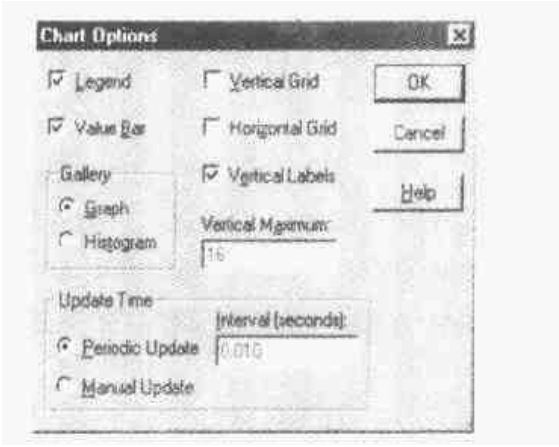


图 6-35 改变 Chart Option 选项卡数值

9）现在把 Cpustres 进程带到前台。你可以看到 Cpustres 线程的优先级提高了 2，然后又回到了基本优先级，如图 6-36 所示

10）Cpustres 周期性的获得一个值为 2 的优先级提高的原因在于你正在监视的线程在大约 75% 的时间内休眠然后苏醒——当线程苏醒时优先级就会提高。为了更频繁地观察到线程获得优先级提高，将 Activity 级别从 Low 增加到 Medium，再到 Busy。如果你将 Activity 级别设置为 Maximum，你就不会看到任何优先级的提高，因为 Cpustres 的 Maximum 使得线程进入无限循环。因此，线程不会激活任何 wait 函数，也就不会获得任何优先级的提高。

11）完成后，退出 Performance Monitor 和 CPU Stress。

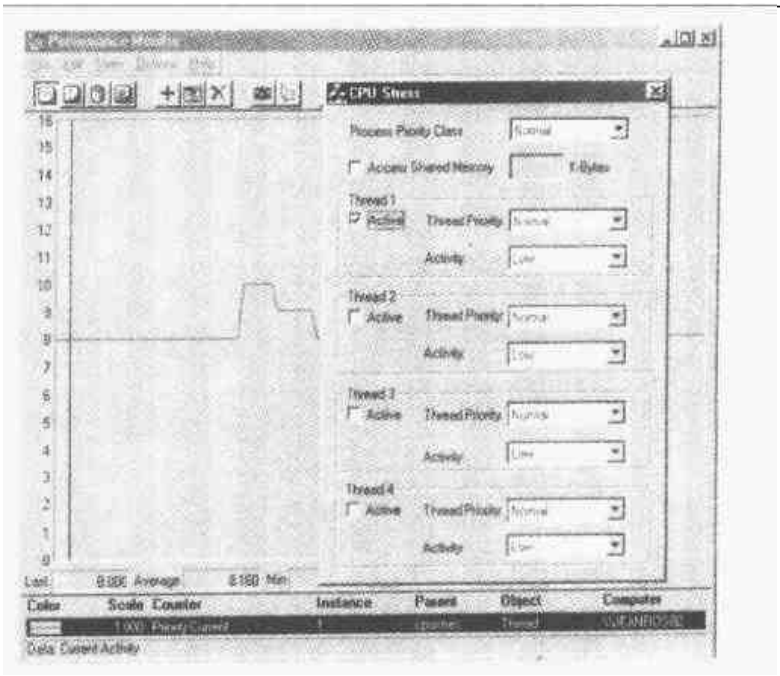


图 6-36 观察前台优先级提高和降低

4、唤醒 GUI 线程再提高优先级

当拥有窗口的线程由于窗口活动（如窗口消息的到来）而唤醒时，它的优先级会提高一个 2 的值。当窗口系统调用一个 KeSetEvent 设置一个事件来唤醒某个 GUI 线程，它就会使用这种优先级提高。这样做的原因与前面的类似——都是为了有利于交互式应用程序。

实验：观察 GUI 线程的优先级提高

通过监视一个 GUI 应用的当前优先级并在它的窗口上移动鼠标，你可以看到窗口系统给唤醒以便处理窗口消息的 GUI 线程增加优先级 2。只需遵循如下步骤：

1）打开控制面板的 System 功能（或右击 My Computer 并选择 Properties），点击 Advanced 标签，并点击 Performance Options 按钮。确信 Applications 选项已经选择。这将使得 PsPrioritySeparation 获得值 2。

2）选择 Programs/Accessories/Notepad，从启动菜单中运行 Notepad。

3）运行 Windows NT 4 的 Performance Monitor（Windows 2000 资源包中的 Perfmon4.exe）。

我们需要采用这个旧版本的 Performance 工具来做这个实验，因为它查询 performance counter 的值的速度要比 Windows 2000 Performance 工具快（后者的最大间隔是每秒一次）。

4）点击 Add Counter 工具条按钮（或按 Ctrl + I）弹出 Add To Chart 对话框。

5）选择线程对象，然后选择 Priority Current counter。

6）在 Instance 框内，滚动列表直到你能看到 Notepad 线程 0。点击 Add 按钮，然后选择 Done 按钮。

7）从 Options 菜单中选择 Chart。改变垂直最大值为 16，间隔为 0.010，如下所示，点击 OK。

8）你就可以看到 Notepad 中线程 0 的优先级为 8、9 或 10。Notepad 在它的前台进程的线程收到一个优先级增加 2 后，立刻就进入等待状态，因此它可能还没有来得及从 10 降低

到9，再减低到8。

9）利用前台的 Performance Monitor，移动鼠标到 Notepad 窗口（使得两个窗口在桌面上都是可见的）。你会看到优先级有时是10，有时是9，原因和上面解释的一样。（你不可能捕捉到 Notepad 在8的时候，因为它在接受到 GUI 线程的优先级增加2后运行的很少，以至于在它再次唤醒后就没有经历多于一个优先级别的降低，而由于其他窗口的活动优先级又增加了2。

10）现在将 Notepad 带到前台。你可以看到其优先级增加到12并保持（或降到11，因为可能经历一次正常的优先级降低）。因为线程收到两次优先级的增加：当它苏醒来处理窗口消息时增加了2；由于 Notepad 处于前台又增加了2。

11）如果你将鼠标移动到 Notepad（当它仍然在前台时），你会看到优先级降到11（或者甚至10），这是由于在时间片结束之前经历的优先级的降低过程。然而，只要 Notepad 仍旧在前台，因为是前台进程它的优先级就会增加2。

12）完成后，退出 Performance Monitor 和 Notepad。

5. 用于 CPU 饥饿的优先级提高

想象下面的情况：你得到一个优先级为7的正在运行的线程，它将禁止优先级为4的线程在任何时候得到 CPU；然而，一个优先级为11的线程在等待优先级为4的线程锁住的资源。但是，因为在中间的优先级为7的线程占据了所有的 CPU 时间，优先级为4的线程将永远不会得到足够长的运行时间以完成它的工作并释放阻塞了优先级为11的线程的资源。Windows 2000 是如何处理这种情况的呢？平衡集管理器（balance set manager）（主要用于执行内存管理函数的系统线程，第7章将详细描述）每秒扫描一次所有处于就绪状态（即没有运行）的超过300个时钟滴答（大约有3～4秒，这取决于时钟间隔）的线程的就绪队列。如果它发现这样的线程，它就会把该线程的优先级提高到15并给它两倍的正常时间片。一旦两倍的时间片结束，该线程的优先级会立刻衰减到它原先的基本优先级。若该线程没有完成而同时有一个更高优先级的线程准备运行，被衰减的线程将返回到就绪队列中，在那儿如果它又等待了300个时钟滴答，那么它将再一次有资格进行另外一次提高。

事实上，平衡集管理器并不是每次运行时都扫描所有的就绪线程。为了使所使用的 CPU 时间最少，它只扫描16个就绪线程；如果在该优先级上有更多的线程，它将记住它离开的位置并在下一次扫描时从那里开始。而且它每次扫描时只提高10个线程的优先级——如果它发现有10个线程值得这种特殊的提升（这意味着这是一个不同寻常的繁忙的系统），它将在这一点停止扫描并在下一次扫描时从此点开始。

这种算法总能解决这种优先级冲突问题吗？不，不是在任何时候它都是十全十美的。但总的来说，处于饥饿状态的线程应得到足够的 CPU 时间以完成它们正在处理的工作，并再次进入等待状态。

6.5.14 对称式多处理系统上的线程调度

如果对系统处理器的调度访问是基于线程的优先级，那么对于多处理器系统会发生什么事

情呢？总的来说，Windows 2000 总是试图在所有的处理器上调度优先级最高的可运行线程。然而，有几个因素影响选择线程应在哪一个 CPU 上运行。在讲述算法之前，需要定义一些术语：

1. 相似性

每个线程都有一个相似性掩码（affinity mask），它指定允许该线程运行的处理器。线程的相似性掩码继承进程的相似性掩码。默认时，所有的进程（同样对于所有的线程）都开始于一个相似性掩码，它等于系统中活动的处理器集——也就是说，全部线程都能够在所有的处理器上运行。

有两件事可以改变该掩码，它们是：

■ 应用程序调用 SetProcessAffinityMask 或 SetThreadAffinityMask 函数。

■ 在映像头中指定的映像范围内的相似性掩码（关于 Windows 2000 映像格式的详细信息，参见 MSDN 库中的文章“Portable Executable and Common Object File Format Specification”）。

实验：观察 CPU 饥饿的优先级提高

使用 CPU Stress 工具（在资源工具箱和 Platform SDK 中），你可以观察实际的优先级提高。在本实验中，当线程的优先级提高时，你会看到 CPU 使用的改变）。采取以下的步骤：

1）运行 Cpustress.exe，改变活动线程的活动级别从 Low 到 Maximum（默认为 Thread 1）。改变线程的优先级从 Normal 到 Below Normal，如图 6-31 所示：

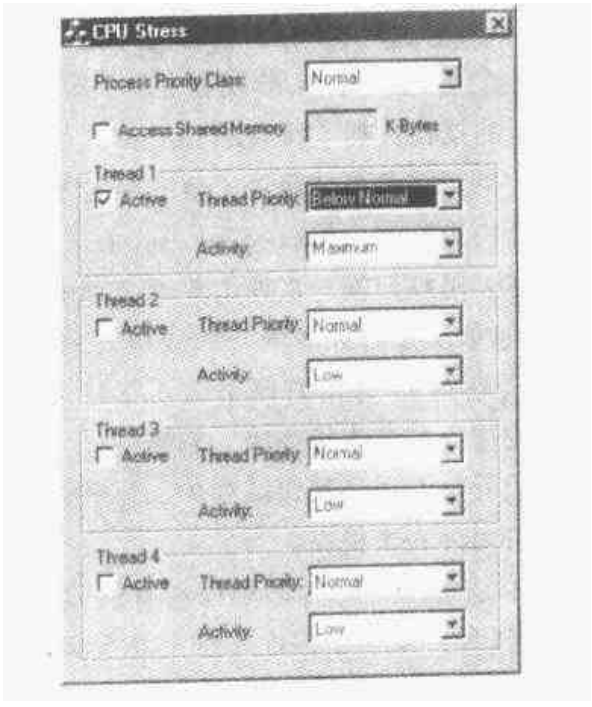


图 6-37 改变线程的优先级

2）运行 Windows NT 4 Performance Monitor（在 Windows 2000 工具箱里为 Perfmon4.exe）。接着，你需要这个实验的老版本，因为它可以以快于每秒钟一次的速度查询性能计数器。

3）点击 Add Counter 工具条按钮（或者按 Ctrl + I），出现 Add To Chart 对话框。

4）选择 Thread 对象，接着选择%Processor Time 计数器。

5）在 Instance 对话框中，滚动列表直到你看到 cpustress 进程。选择第二个线程（thread 1）（第一个线程是 GUI 线程）。你可以看到如图 6-38 的对话框：

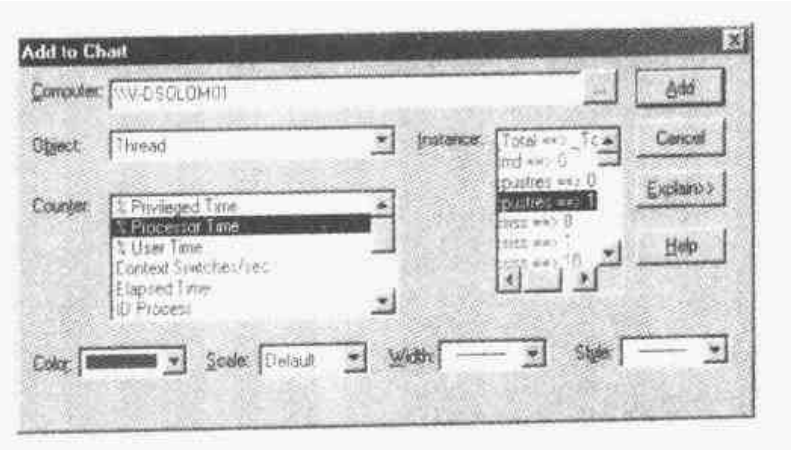


图 6-38 查看 cpustress 进程

6) 点击 Add 按钮，接着点击 Done 按钮。

7) 通过运行 Task Manager，提高 Performance Monitor 的优先级，点击 Process 标签，接着选择 Perfmon4.exe 进程。右击进程，选择 Set Priority，接着选择 Realtime。(如果你收到一个 Task Manager Warning 消息框警告你系统不稳定，点击 Yes 按钮。)

8) 运行 CPU Stress 复本。在该复本中，从 Low 到 Maximum 改变 Thread 1 的活动级。屏幕看起来应该像图 6-39。

9) 现在切换回到 Performance Monitor。你应该大约每四秒钟观察一次 CPU 活动，因为线程已经提高到级别 15。

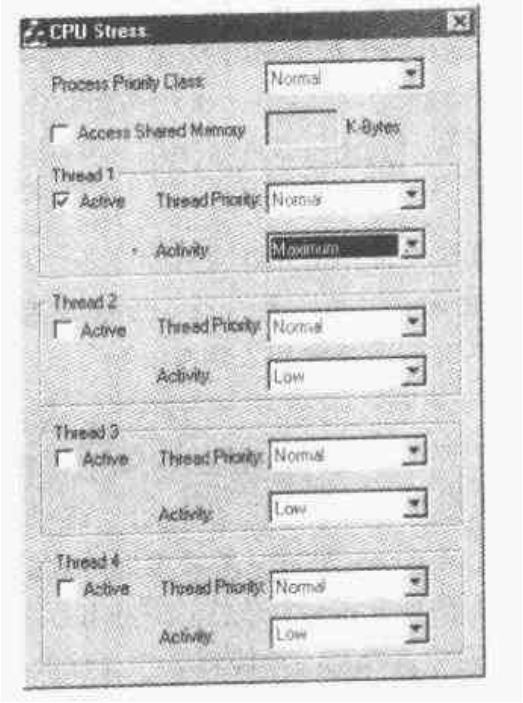


图 6-39 查看 CPU Stress 副本

当你已经完成，退出 Performance Monitor 和 CPU Stress 的两个副本。

2. 理想处理器和次选处理器

每个线程有两个 CPU 号存储在内核线程块中：

■ 理想处理器（ideal processor 或 preferred processor），线程应在它上面运行的处理器。

■ 最后处理器（next processor），线程运行的最后处理器。

在创建过程中，理想处理器是随机选择的，它基于在进程块内的种子（seed）。每次创建线程时这个种子都递增以便用于进程中每一个新进程的理想处理器在系统的有效处理器上轮流被选择。一旦线程被创建，Windows 2000 就不再更改理想处理器；然而，应用程序可以使用 SetThreadIdealProcessor 函数为线程更改理想处理器值。

3. 为就绪线程选择处理器

当线程变为就绪而运行时，Windows 2000 将首先试图调度线程在一个空闲的处理器上运行。如果有可选的空闲处理器，优先选择是给该线程的理想处理器，其次是该线程的最后处理器，然后是当前执行的处理器（也就是说，正在运行调度代码的 CPU）。如果所有的这些 CPU 都忙，则 Windows 2000 通过从最低到最高 CPU 号来扫描空闲处理器掩码，来选择第一个有效的空闲处理器。

如果所有的处理器现在都忙，而且有一个线程已经就绪，则 Windows 2000 就看它是否能够抢先其中一个 CPU 上的处于运行或备用状态的线程。应该检查哪一个 CPU 呢？首先选择线程的理想处理器，其次是线程的最后处理器。如果这些 CPU 都不在线程的相似性掩码内，Windows 2000 就选择在线程能够运行的活动的处理器掩码内的最高位的处理器。

如果要选择的处理器已经选择了下一个要运行的线程（处于备用状态并等待调度），并且那个线程的优先级低于准备执行线程的优先级，则新线程将取代处于备用状态的线程，而成为那个 CPU 的下一个要运行的线程。如果那个 CPU 没有下一个要选择运行的线程，则 Windows 2000 就检查当前运行线程的优先级是否低于准备执行的线程的优先级。如果真是如此，那么当前运行的线程就标记为已被抢先，然后 Windows 2000 加进一个处理器间中断来取消当前在运行的线程以便支持新的线程。

注意 Windows 2000 并不查看所有 CPU 上当前的和下一个线程的优先级——仅是在上面描述的一个被选择的 CPU 上查看。如果在那一个 CPU 上没有线程能够被抢先，新线程将被加入它的优先级上的就绪队列，等待被依次调度。

4. 选择线程在指定的 CPU 上运行

在一些情况下（例如在线程降低了它的优先级、更改它的相似性或者延迟或放弃执行时），Windows 2000 必须找到一个新线程在指定的 CPU 上运行，此 CPU 是当前被执行的线程正在运行的 CPU。在单处理器系统上，Windows 2000 只是简单地挑选就绪队列中的第一个线程，从至少有一个线程的有最高优先级的队列开始，并依次往下查找。然而，在多处理器系统中，Windows 2000 不是简单地从就绪队列中选择第一个线程。它寻找的是符合下列条件之一的线程：

■ 最近一次在指定处理器上运行过。

■ 它的理想处理器被设置为所指定的处理器。

■ 为运行而等待的时间超过了两个时间片。

■ 有一个不小于 24 的优先级。

很明显地，那些在它们的硬相似性掩码中没有特定处理器的线程将被跳过。如果 Windows

2000不能找到任何线程满足这些条件中的一个，它就从就绪队列头部的线程开始搜索。

为什么关心哪一个是线程最近一次运行所在的处理器呢？像通常一样，答案是速度——之所以优先选择线程最近执行的处理器是因为线程数据仍保留在正被讨论的处理器的二级缓存中的机会最大。

5. 什么情况下最高优先级就绪线程不运行

如同在前面说明的，多处理器系统中，Windows 2000并不是总选择优先级最高的线程在给定的CPU上运行。这样，在给定的CPU上，比当前运行的线程优先级高的线程能够处于就绪状态，但并不一定立刻抢先当前线程。

还有另外一种具有最高优先级的线程不抢先当前线程的情况：当线程的相似性掩码被设置为可获得的CPU的子集时。这种情况下，与线程有相似性关系的处理器当前正在运行最高优先级的线程，线程就必须等待这些处理器中的一个——即使另外一个处理器是空闲的，或正在运行可以被抢先的较低优先级的线程。Windows 2000不会把一个可以在其他处理器上运行的线程从一个CPU移动到第二个处理器上，以便让和第一个处理器有相似性关系的线程在第一个处理器上运行。

例如，考虑这种情况：CPU 0正在运行一个优先级为8的可以在任何处理器上运行的线程，CPU 1正在运行一个优先级为4的可以在任何处理器上运行的线程。一个只能在CPU 0上运行的优先级为6的线程处于就绪状态。那么将会发生什么情况呢？Windows 2000不会为了运行优先级为6的线程而把优先级为8的线程从CPU 0移到CPU 1（抢先优先级为4的线程）；优先级为6的线程不得不等待。

6.6 作业对象

作业（Job）是一个可命名的、安全的和可共享的内核对象，它允许以一个组的形式来控制一个或多个进程。作业的基本功能是允许以一个单元的形式管理和处理进程组。一个进程可以仅仅是一个作业的成员。缺省地，它和作业的关系不能断开而且所有由该进程和它的后代创建的进程都与同一个作业关联。作业也可以为所有与它关联的进程以及曾经与之关联但已经终止的进程记录基本的统计信息。表6-20列出了用来创建和操作作业的Win32函数。

表6-20 作业相关Win32 API函数

函 数	描 述
CreateJobObject	创建一个作业（带一个可选的名字）
OpenJobObject	用名字打开一个已经存在的作业
AssignProcessToJobObject	将一个进程加入到一个作业
TerminateJobObject	终止一个作业中包含的所有对象
SetInformationJobObject	设置限制
QueryInformationJobObject	获取作业的信息，例如CPU时间、页面缺省计数、进程的数目、进程ID列表、配额或者限制以及安全限制

下面是一些你可以为一个作业设定的与CPU和内存相关的限制：

- 活动线程的最大数目 限制作业内同时执行的进程数目。
- 作业范围内的用户模式的CPU时间限制 限制作业内的进程在用户模式下总共可以使

用的最大 CPU 时间（包括那些已经运行和激活的进程）一旦达到这种限制，缺省情况下作业中所有的进程将会终止并返回一个错误代码，而且不能再创建新的进程（除非限制重新设定）。作业被设为有信号的，这样所有等待该作业的线程都会被释放。你可以通过调用 EndOfJobTimeAction 来改变这种缺省的行为。

■ 每一进程在用户模式下的 CPU 的时间限制 允许作业中每个进程在用户模式下运行 CPU 的最大时间限制。当达到最大值时，该线程就会终止（没有时间完成退出前必要的清理工作）。

■ 作业调度类别 设置作业内进程包含的线程的时间片长度。这个设置只适用于长的且固定的时间片系统（如 Windows 2000 Server）。作业调度类别的值决定时间片，如下所示：

调度类别	时间片单元
0	6
1	12
2	18
3	24
4	30
5	36
6	42
7	48
8	54
9	如果实时，就是无穷；否则为 60

■ 作业处理器相似性 为作业中每个进程设置处理器相似性掩码（单个线程可以将它的相似性改变为作业相似性的任何子集，但是进程不能改变它们的相似性设置）。

■ 作业进程优先级类别 为作业中的每个进程设置优先级。线程不能提高相对于进程优先级的自身的优先级（虽然通常情况下可以）。那些提高线程优先级的企图都会被忽略（虽然对 SetThreadPriority 的调用不会返回任何错误，但是实际上优先级并没有改变）。

■ 缺省的工作集合（working set）最大和最小 为作业中的每个进程定义特定的工作集合最大和最小（这个设置不是针对整个作业范围的——每个进程有它们自己的有相同的最大和最小值的工作集）。

■ 进程和作业提交（commit）的虚拟内存限制 定义单个进程或整个作业可以提交的最大内存数量。

作业也可以设置为将 I/O 完成端口对象的项排队，而其他线程可能由于调用 Win32 GetQueuedCompletionStatus 函数而处于等待状态。

你也可以将安全限制加到某个作业的进程中。你可以设置一个作业使得每个进程都在同一个作业范围的访问令牌下运行。你可以创建一个作业用来限制进程模拟或创建包含本机管理者组的访问令牌的进程。而且，你可以使用安全过滤使得当一个作业中某个进程内的线程模拟客户线程的时候，某些特权和安全 ID 将从模拟令牌中消减。

最后，你可以给作业中的进程加上用户接口限制。这些限制包括不允许进程打开不属于该作业的线程所拥有的窗口句柄，读/写剪贴板，利用 Win32 函数 SystemParametersInfo 来改变许多用户接口的系统参数。

Windows 2000 Datacenter Server 有一个工具叫 Process Control Manager，它允许管理员定义作业

对象、为作业指定的不同的配额和限制以及哪个进程在运行的时候应该加入到作业中。一个服务组件监视进程活动并把指定进程添加到作业中。

实验：查看作业对象

你可以利用 Performance tool 来查看命名的作业对象（参见 Job Object 和 Job Object Details 性能对象）为了查看没有名字的作业对象，你必须使用内核调试命令！job，按照以下步骤创建和查看一个无名字的作业对象：

1）在命令行输入 runas 命令创建一个运行命令行处理程序（Cmd. exe）的进程。例如，键入 runas /user: <domain> \ <username> cmd。你会被提示输入密码。输入密码后，一个命令行窗口就会出现。在 runnas 命令后的服务创建一个无名的作业来包含所有进程（这样它就可以在 logoff 的时候终止所有进程。

2）在内核调试器中（例如 LiveKd），使用！process 命令显示进程列表，找出最近创建的运行 Cmd. exe 的进程。然后，利用！process <processID> 显示进程块，找出作业对象的地址，最后，利用！job 命令显示作业对象。

下面是第 2 步中描述的命令序列在调试器中的部分输出结果：

```
kd> !process 0 8
**** NT ACTIVE PROCESS DUMP ****
    :
PROCESS 84eab6d0  SessionId: 0  Cid: 0478  Peb: 7ffdf000 ParentCid: 0240
    DirBase: 03834000  ObjectTable: 8097ef88  TableSize:  42.
    Image: livekd.exe

PROCESS 857e0d70  SessionId: 0  Cid: 0550  Peb: 7ffdf000 ParentCid: 00dc
    DirBase: 05337000  ObjectTable: 82273ac8  TableSize:  22.
    Image: cmd.exe

PROCESS 83390710  SessionId: 0  Cid: 0100  Peb: 7ffdf000 ParentCid: 0478
    DirBase: 05b3b000  ObjectTable: 81bb7e08  TableSize:  34.
    Image: i386kd.exe

kd> !process 550
Searching for Process with Cid == 550
PROCESS 857e0d70  SessionId: 0  Cid: 0550  Peb: 7ffdf000 ParentCid: 00dc
    DirBase: 05337000  ObjectTable: 82273ac8  TableSize:  22.
    Image: cmd.exe
    :
    Job                                 85870970
    :
kd> !job 85870970 7
Job at 85870970
  TotalPageFaultCount      0
  TotalProcesses           1
  ActiveProcesses          1
  TotalTerminatedProcesses 0
```

```
LimitFlags              0
MinimumWorkingSetSize   0
MaximumWorkingSetSize   0
ActiveProcessLimit      0
PriorityClass           0
UIRestrictionsClass     0
SecurityLimitFlags      0
Token                   0
Processes assigned to this job:
 PROCESS 857e0d70 SessionId: 0 Cid: 0550 Peb: 7ffdf000 ParentCid: 00dc
     DirBase: 05337000  ObjectTable: 82273ac8  TableSize:  22.
     Image: cmd.exe
```

6.7 小结

在本章中，我们研究了进程和线程的结构、了解它们是如何创建和销毁的以及 Windows 2000 是如何决定线程是否应该运行和运行多长时间。

在本章中参考引用了许多与内存管理相关的主题。因为线程是在进程中运行的而进程定义了一个地址空间，所以下面一个逻辑主题是 Windows 2000 执行虚拟和物理内存管理——第 7 章的主题。

第7章 内存管理

在本章中，你将学习 Microsoft Windows 2000 实现虚拟内存和管理保留在物理内存中的虚拟内存子集的方法。这些工作包括两项主要任务：

■ 转换或映射进程的虚拟地址空间到物理内存以便当在进程环境中运行的线程读取或写入虚拟地址空间时，引用正确的物理地址（一个驻留进程的虚拟地址空间子集称为“工作集”，“工作集”在本章后面详细介绍）。

■ 当内存被过量使用时，也就是说，当运行的线程或系统代码试图使用比当前的可用内存更多的物理内存时，页面调度一些内存的内容到磁盘中，并且在需要的时候再把它写回到物理内存中。

正如第 2 章看到的（表 2-2），Windows 2000 Professional 和 Windows 2000 Server 支持最多 4GB 的物理内存。Windows 2000 Advanced Server 支持最多 8GB 的物理内存。而 Windows 2000 Datacenter Server 达到 64GB（实际上 Windows 2000 Datacenter Server 支持的内存依赖于可获得的硬件。写本书时 Windows 2000 Datacenter Server 还没有发行，所以不能证实当前的硬件是否支持最多 64GB 的操作系统）。

因为 Windows 2000 是 32 位操作系统，可以提供用户进程一个 4GB 的 32 位平面虚拟地址空间。本章后面的“地址窗口扩展”将说明一个 32 位进程如何分配和使用大量的物理内存。

注意 微软公司宣布计划提供支持英特尔 Itanium 系列处理器的真正 64 位 Windows 版本。新的 64 位 API，称为 Win64，支持真正的 64 位地址。支持这个平台的原因和从 16 位转移到 32 位地址空间的原因一样，是为了满足在内存中处理和存储不断增长的大量数据的需要。64 位版本的 Windows 突破 Windows 2000 的空间限制，提供给进程一个 64 位虚拟地址空间平面。关于 64 位 Windows 的详细内容参见平台 SDK 的“Getting ready for 64-bit Windows”或微软站点 www.microsoft.com/ Windows 2000/ guide/ platform/ strategic/ 64bit.asp。

除了提供虚拟内存管理外，内存管理器还提供了一组内核服务，在它们上面将建立不同的 Windows 2000 环境子系统。这些服务包括内存映射文件（在内部叫“区域对象”（section object））、写时复制（copy－on－write）内存和对使用大量的、稀疏的地址空间的应用程序支持。在这一章中，将总结这些基本的服务并回顾有关概念，例如保留内存与提交内存的对比和共享内存。同时还将描述组成内存管理器的内部结构和组件，包括关键的数据结构和算法。

7.1 内存管理器组件

内存管理器是 Windows 2000 执行程序的一部分，因此它存在于 Ntoskrnl.exe 中，在硬件抽象层（HAL）中没有内存管理器的任何部分。内存管理器由下列组件构成：

■ 一组执行程序系统服务程序，用于虚拟内存的分配、释放和管理，它们中的大多数通过

Win32 API 或内核模式设备驱动程序接口显露。

■ 一个转换无效（translation - not - valid）和访问错误陷阱处理程序，用于解决硬件检测到的内存管理异常事件，并代表一个进程使虚拟页驻留。

■ 运行在 6 个不同内核模式系统线程的环境中的几个关键组件：

□ 工作集管理程序（优先级为 16），由平衡集管理程序（内核创建的系统线程）每秒调用一次，或在空闲内存低于某个阈值时调用，驱动所有的内存管理规则，例如工作集的休整、年龄（aging）和已修改页的写入。

□ 进程/栈交换程序（优先级为 23），执行进程和内核线程的栈换入和换出操作。在需要执行换入和换出操作时，平衡集管理程序和在内核中的线程调度代码可以唤醒这个程序。

□ 修改页面的书写器（优先级为 17）把在修改链表上的脏页面写入到正确的调页文件中。当需要减小修改链表的大小时可以唤醒这个线程（参见 7.9.2 节“修改页面的书写器”了解如何修改默认值）。

□ 映射页面书写器（优先级为 17）把映射文件中的脏页写到磁盘。当修改链表的大小减小时和当映射文件的页面在修改链表上超过 5 分钟时，唤醒这个线程。这个第二个修改页面书写器是必需的，因为它可以产生页面错误，导致对空闲页的请求。如果没有空闲页并且只有一个修改页面书写器，系统由于等待空闲页而死锁。

□ 废弃段线程（优先级为 18）负责系统高速缓存和页面文件的增加和减少（例如，没有虚拟地址满足页交换区的增长时，废弃段线程减少系统高速缓存）。

□ 清零页面线程（优先级为 0）将空闲链表内的页清零以便使零页高速缓存能用于满足将来的零页错误。

本章的后续内容将详细讲述各个组件的细节内容。

像所有其他 Windows 2000 执行程序组件一样，内存管理器在多处理器系统上是完全可重入的并支持同时执行——也就是说，它允许两个线程获取资源而不破坏对方数据。为实现完全可重入，内存管理器利用了多种不同的内部同步机制，用它们来控制对它的内部数据结构的访问，例如自旋锁和执行程序资源（同步对象在第 3 章已经讨论过）。

内存管理器必须同步访问系统范围的资源，这些资源包括页面帧号（PFN）数据库（由自旋锁控制）、区域对象和系统工作集（由执行程序系统资源控制）、页面文件的创建（由互斥控制）以及其他内部结构。每进程（per - process）内存管理数据结构使用两个每进程（per - process）互斥进行同步：工作集锁（工作集链表被修改时将被加锁）和地址空间锁（地址空间改变时加锁）。

7.1.1 配置内存管理器

像大多数 Windows 2000 程序一样，为了对不同的工作量以及系统大小和类型提供优化的系统性能，内存管理器自动提供优化的性能。可以使用注册表键 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ Memory Management 下的键值覆盖默认的性能值，表 7-1 列出了其中部分键值。细节参见 Windos 2000 资源开发包技术参考的注册表帮助文件。

表 7-1 影响内存管理器的注册表变量

注册表键值	说明
ClearPageFileAtShutdown	指定是否在系统终止时，将调页文件（paging file）中的不活动的页面用0填满。这是一个安全特性
DisablePagingExecutive	指定用户模式和内核模式下的驱动程序和内核模式系统代码是否在它们不工作时可以被调页到磁盘上。如果此项的值为0（默认值），那么驱动程序和内核代码必须保留在物理内存中；如果此项的值为1，它们就可以在需要时被调页到磁盘上
IoPageLockLimit	指定用于I/O操作的用户进程可以锁定的字节数的限制。当值为0时，系统使用默认值（512K）。它的最大值近似等于系统物理内存减去7MB。在Windows 2000 Datacenter Server中不使用，在以Windows 2000 Service Pack 1开始的Windows 2000中不再使用
LargePageMinimum	显示Ntoskrnl和HAL使用大页面（4MB）需要映射的最小兆字节数（此注册值不在文档中也没有默认值。必须手工自己添加）
LargeSystemCache	在综合权衡内存时，影响文件系统的高速缓存或进程的工作集是否可以给定优先级，同时影响文件系统的高速缓存大小（在Windows 2000 Server上可以通过设置文件服务器的属性间接调整这个值，参见第11章）
NonPagedPoolQuota	显示可以通过任何一个进程分配的最大非页交换区（以兆字节表示）。若此项的值为0，系统将计算此值
NonPagedPoolSize	显示非页交换区的初始大小（以字节表示）。当此值为0时，系统将计算此值
PagedPoolQuota	显示可以通过任何一个进程分配的最大页交换区（以兆字节表示）。若此项的值为0，系统将计算此值
PagedPoolSize	显示页交换区的最大值（以字节表示）。当此项的值为0时，系统将计算此值。若它的值为-1表示选择可能的最大值，允许有利于系统页面表项（PTE）的大页交换区
SystemPages	显示为映射I/O缓冲区、设备驱动程序、内核线程堆栈或用户程序控制的I/O占用系统地址空间而保留的系统页面表项的数量。若它的值为0，系统将计算此值。若它的值为-1，保留最大数的系统PTE（必须支持此值，例如支持需要大量系统PTE的设备，如一个显卡的512MB的显示内存必须一次映射完成）

影响内存管理器策略的大部分有趣的调整量或控制是内核变量，包含了在系统启动时根据内存大小和产品类别计算的各种阈值和限制（Windows 2000 Professional为用于桌面交互而进行了优化，Windows 2000 Server为运行服务程序而优化）。调整量包括：系统内存的大小（页交换区、非页交换区、系统高速缓存、系统页面表项的数量）、页面读取的大小、触发工作集休整的计数器以及修改页面的书写器的阈值。关于这些调整量，请查找在Ntoskrnl.exe中以Mm开头的包含“maximum”和“minimum”的变量。

注意 尽管你可以找到这些调整量的参考值，但是请不要改变它们，对这些可以进行计算的数值的当前可能的排列，Windows 2000已经进行了测试并运行正常。在一个正运行的系统上改变系统变量的值会造成不可预见的系统行为，包括系统挂起或者系统

崩溃。

表 7-2 列出了当前内存的大小，它被 Windows 2000 用来判断系统是否有小的、中等的或大的内存。内存管理器经常在启动的计算中使用此值。

表 7-2 决定系统内存大小的值物理内存

系统内存大小	物理内存
小	< 19 MB
中等	20 ~ 32 MB
大	> 32 MB，Windows 2000 Professional
	> 64 MB，Windows 2000 Server

决定系统内存大小

因为设备驱动程序以及其他一些 Windows 2000 内核模式组件会以这些值为基础决定资源的分配和运行期策略，所以提供了下面一些内核模式例程（已经记录在 DDK 文档中）。

函数	说明
MmQuerySystemSize	返回该计算机是否有可用的小的、中等的或大的内存。
MmIsThisAnNtAsSystem	对于 Windows 2000 Server，Windows 2000 Advanced Server 和 Windows 2000 Datacenter Server 返回 TRUE，对于 Windows 2000 Professional 返回 FALSE。（这个例程名称源于早期产品 Microsoft Windows NT 的服务器版本，Windows NT Advanced Server）。

7.1.2 检查内存的使用

内存和进程性能计数器对象提供了对关于系统和进程内存利用的大部分细节。在整个这一章中，将引入一些特定的性能计数器，它们包含与所讨的组件相关的信息。

除了性能工具，Windows 2000 支持工具（Support Tool）和 Windows 2000 资源工具包（resource kit）中的许多工具显了内存使用信息的不同子集。本章中包括了相关的范例和实验。提醒一句，不同的应用程序在显示内存信息时，使用不同的有时是不一致和易混淆的名称。以下例子说明了这一点（本章后面详细解释例子中的术语）。

实验：查看系统内存信息

如图 7-1，Windows 2000 Task Manager Performance 标签显示基本的系统内存信息。这些信息是通过性能计数器可以获得的详细信息的子集。

Pmon.exe（在 Windows 2000 Support Tool 中）和 Pstat.exe（在 Platform SDK 中）显示系统和进程的内存信息。在图 7-2 所示的输出注解中解释了 Pstat 所报告的信息（对于提交总和与限制的解释参见表 7-8）。

查看页交换区和非页交换区的使用情况，请使用 Poolmon 实用程序。在后面的实验“监视交换区的使用”中描述。

最后，内核调试器中的！vm 命令显示了通过内存相关的性能计数器可以获得的基本内存的管理信息。在观察崩溃转储和挂起系统时，该命令极其有用，这里有一个输出的例子。

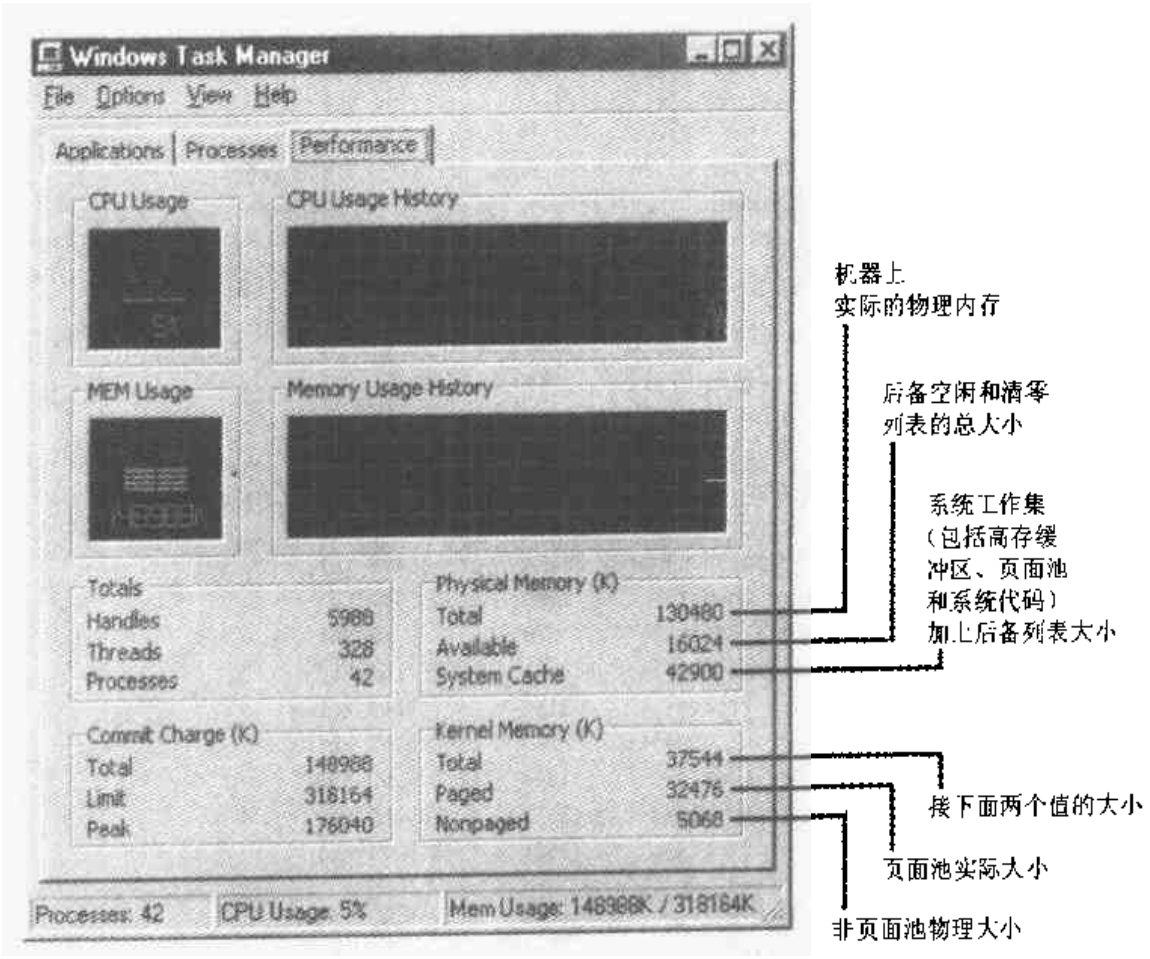


图 7-1 查看 Windows 2000 Task Manager Performance 标签

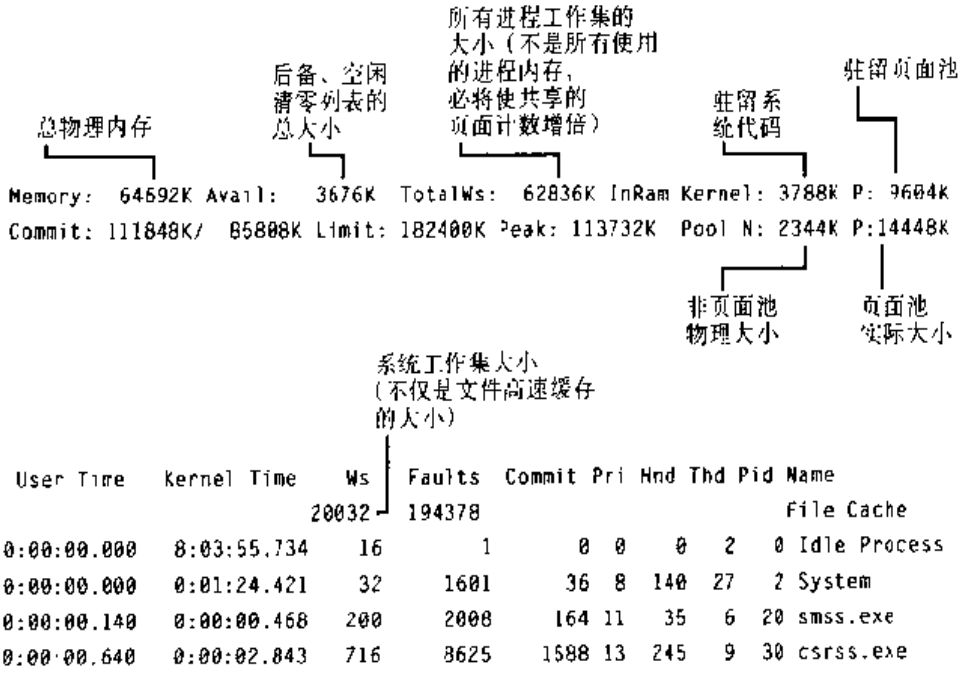


图 7-2 输出 Pstas 报告信息的注解

```
kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:     32620   (  130480 Kb)
        Page File: \??\C:\pagefile.sys
           Current:    204800Kb Free Space:    101052Kb
           Minimum:    204800Kb Maximum:       204800Kb
        Available Pages:      3604   (   14416 Kb)
        ResAvail Pages:      24004   (   96016 Kb)
        Modified Pages:        758   (    3072 Kb)
        NonPagedPool Usage:   1436   (    5744 Kb)
        NonPagedPool Max:    12940   (   51760 Kb)
        PagedPool 0 Usage:    6817   (   27268 Kb)
        PagedPool 1 Usage:     982   (    3928 Kb)
        PagedPool 2 Usage:     984   (    3936 Kb)
        PagedPool Usage:      8783   (   35132 Kb)
        PagedPool Maximum:   26624   (  106496 Kb)
        Shared Commit:        1361   (    5444 Kb)
        Special Pool:            0   (       0 Kb)
        Free System PTEs:   189291   (  757164 Kb)
        Shared Process:       3165   (   12660 Kb)
        PagedPool Commit:     8783   (   35132 Kb)
        Driver Commit:        1098   (    4392 Kb)
        Committed pages:     45113   (  180452 Kb)
        Commit limit:        79556   (  318224 Kb)

        Total Private:       30536   (  122144 Kb)
            IEXPLORE.EXE      3028   (   12112 Kb)
            svchost.exe       2128   (    8512 Kb)
            WINWORD.EXE       1971   (    7884 Kb)
            POWERPNT.EXE      1905   (    7620 Kb)
            Acrobat.exe       1761   (    7044 Kb)
            winlogon.exe      1361   (    5444 Kb)
            explorer.exe      1300   (    5200 Kb)
            livekd.exe        1015   (    4060 Kb)
            hh.exe             960   (    3840 Kb)
            :
```

实验：计算物理内存使用

综合性能计数器的信息和内核调试器命令的输出，可以在运行 Windows 2000 的计算机上计算物理内存使用情况，为了观察通过性能计数器获得的内存使用情况，运行 Performance 工具添加计数器观察以下的信息。(如果把图中的垂直标尺最大值设置到 1000，更容易看到结果)

■ 总进程工作集大小 观察此信息，请选择进程性能对象和 Total 进程实例的 Working Set 计数器。因为计算了每个进程工作集共享页面数，进程工作集的大小比实际的进程使用的总内存要大。从计算机总物理内存减去空闲内存（可用字节）、使用了的操作系统内存（非页交换区、驻留页交换区、驻留操作系统和驱动程序代码）和机器上全部物理内存的修改链表

大小，剩下的就是更准确的进程内存利用情况。通过与性能工具报告的总进程工作集大小的比较，可以表示出进程共享内存的数量。尽管观察进程的物理内存很有趣，进程的专有提交的虚拟内存的使用情况更值得关注，这是由于内存泄漏引起专有虚拟内存而不是工作集的增长。（尽管进程虚拟内存的大小可以增长到提交的限制——系统可用的最大专有提交的虚拟内存，但内存管理器会在某点停止增长。更多信息参见7.6.5节“页面文件”部分）

■ 总系统工作集大小 观察此信息，选择 Memory 处理器对象和 Cache Bytes 记数器。在“系统工作集”部分解释过：总系统工作集大小包括的不止是高速缓存，还有驻留页交换区和驻留操作系统和驱动程序代码。

■ 非页交换区大小 观察此信息，要添加 Memory：Pool Nonpaged Bytes 计数器。

■ 空闲、清零和后备链表的大小 观察此信息，要添加 Memory：Available Bytes 计数器（使用！memusage 内核调试器命令分别得到各链表的大小）。

现在你的图中包括除了下列两个不能从性能计数器得到的组件以外的全部物理内存的表示：

■ 非调页的操作系统和驱动程序代码。

■ 修改的和修改未写入的页面链表。

尽管通过内核调试命令！memusage 很容易得到修改和修改未写入页的链表的大小，但没有简单的办法得到非调页操作系统和驱动程序代码的实际大小。然而使用内核调试命令！drivers 1 可以得到非调页和调页操作系统和驱动代码的总物理内存量以及 Resident 栏的总量，它表示驻留在每个可装载的内核模式模块中的页面数。

7.2 内存管理器提供的服务

内存管理器为环境子系统提供了一组系统服务来分配和释放虚拟内存、在进程间共享内存、映射文件到内存、刷新虚拟页面到磁盘、获取关于虚拟页面范围的信息、更改对虚拟页面的保护以及把虚拟页面锁入内存。

像其他的 Windows 2000 执行程序服务一样，内存管理服务允许它们的调用程序提供进程句柄，以指示其虚拟内存将被操作的特殊进程。这样，调用程序就可以操作它自己的内存或（拥有适当的权限）操作另一个进程的内存。例如，一个进程可以创建一个子进程，并在缺省情况下拥有操作子进程的虚拟内存的权力。因此，父进程可以通过调用虚拟内存服务并以参数的形式传递句柄到子进程中来代表子进程分配、释放、读取和写入内存。子系统使用这个特性来管理它们的客户进程的内存，同时这个特性对于实现调试程序也很关键——因为调试程序必须能够读取和写入被调试进程的内存。

这些服务的大部分通过 Win32 API 提供。Win32 API 有三组函数用于管理应用程序：页面粒度虚拟内存函数（Virtualxxx）、内存映射文件函数（CreatFileMapping，MapViewofFile）和堆函数（Heapxxx 和早期的接口 Localxxx 以及 Globalxxx）（在这一节的后面将描述堆管理器）

内存管理器也为设备驱动程序提供了许多其他服务（和其他内核模式系统代码），例如分配和释放物理内存，以及为了直接存储器访问（DMA）传送而锁定物理内存页面。这些函数以

前缀 Mm 开头。另外，尽管不是严格的内存管理器的一部分，但以 Ex 开头的执行程序支持例程被用于从系统堆（页交换区和非页交换区）分配和释放内存，也被用于操作“后备链表”。在本章“系统内存交换区”一节中，将讨论这些主题。

尽管在本书中将引用为设备驱动程序提供的 Win32 函数和内核模式内存管理以及内存分配例程，但因为这本书是描述这些函数的内部操作的，所以将不涉及接口和编程的细节。对于可用函数和它们的接口的完整描述，请参考在 MSDN 上的 Win32 应用程序编程接口（API）和设备驱动程序工具包（DDK）文档。

7.2.1 保留和提交页面

进程的页面是空闲的、保留的或是提交的。应用程序可以首先保留（reserve）地址空间，然后提交（commit）在那个地址空间中的页面。或者它们可以在同一个函数中调用保留和提交。这些服务通过 Win32 的 VirtualAlloc 和 VirtualAllocEx 函数提供。

保留内存是一种简单的方法，线程保留虚拟地址的一个范围以供将来使用。企图访问保留内存导致访问违例，因为页面没有映射到可以解析引用的存储

提交的页面在访问时最终转换到物理内存中的有效页面。提交的页面可以是专用的和非共享的，或者是映射到区域的视图（它可以被其他进程映射，也可以不被其他进程映射）区域将在下一节和“区域对象”一节中描述。

如果页面是进程专有的并且以前未访问过，在第一次访问时作为清零初始化页面（要求清零）创建。如果内存命令明确指出，那么专有提交页面可以由操作系统后来自动写入调页文件。除非使用进程间内存函数，如 ReadProcessMemory 和 WriteProcessMemory，提交页面是专有的不能被其他进程访问。如果提交页面映射到映射文件的一部分，当访问时，必须从磁盘取回，除非以前访问页面的进程或有相同映射文件的其他进程已经读过。

页面从进程工作集移到修改链表，并最终通过正常修改页写入磁盘。（本章稍候解释工作集和修改链表）映射文件页面也可以通过显式调用。
FlushViewofFile 函数完成写回磁盘。

你可以使用 VirtualFree 或 VirtualFreeEx 函数回收（decommit）页面和释放地址空间。回收和释放之间的差别与保留和提交之间的差别相似——回收的内存仍然是保留的，但是释放的内存既不是提交的也不是保留的（它是空闲的）。

分两步过程来保留和提交内存能够通过在需要提交页面之前推迟提交来减少内存的使用。在 Windows 2000 下，保留内存是一种又快又便宜的操作，因为它不消耗任何提交的页面（一个宝贵的系统资源）或者进程页面文件配额（进程可以消耗的提交页面数的限制，不一定是页面文件空间的限制）。所有需要被更新或构造的是相对较小的代表进程地址空间状态的内部数据结构（将在本章的后面解释这些称为“虚拟地址空间描述符”或 VAD 的数据结构）。

保留然后提交内存，这对于需要潜在、大量和连续的内存缓冲区的应用程序是很有用的。地址空间可以被保留，然后在需要的时候再提交，而不是为整个区域提交页面。在操作系统中这项技术被用于每个线程的用户模式栈。当创建线程时，就保留一个栈（默认值是 1MB；可以使用 CreateThread 函数调用或在映像范围的基础上使用/STACK 链接程序标志重新设置其大小）。

默认情况下，提交一个栈中的初始化页面，下一个页面标记为保护页而不提交，用来捕获对超过栈提交部分末端的引用并自动扩展栈。

7.2.2 锁住内存

有两种方式锁住内存中的页面：

■ 设备驱动程序调用内核模式函数 MmProbeAndLockPages、MmLockPagableCodeSection、MmLockPagable DataSection 或 MmLockPagableSectionByHandle。以这种机制锁住的页面一直保留在内存中直到显式开锁。尽管没有对设备在内存中可以锁住的页面数的配额限制，一个设备不能锁住多于驻留的可用页面数。并且，每页都使用系统页面表项（PTE），它是一种有限资源。（本章后面描述 PTE。）

■ Win32 应用程序调用 VirtualLock 函数在其进程工作集中锁住页面。注意这样的页面仍参加调页度。如果进程中所有线程处于等待状态并且内存命令明确指出，内存管理器可以自由的从工作集中删除这样的页面（对于修改页面，最终将导致页面写入磁盘）。在这种情况下，由于当线程被唤醒运行时，在线程开始执行前内存管理器必须先读入所有的锁住页面，因此在工作集中锁住页面实际降低了性能。一般来说，最好让内存管理器决定哪些页留在物理内存中。进程可以锁住的最大页面数不能超过它的最小工作集大小减去 8 页。

7.2.3 分配粒度

Windows 2000 以系统分配间隔尺寸值所定义的整数边界为启始地址排列每个保留的进程地址空间，系统分配间隔尺寸值可以由 Win32GetsystemInfo 函数获取。目前，系统值是 64KB。选择这个大小是因为如果将来添加支持大页面尺寸（例如，达到 64KB）的处理器时，减少由于应用程序假设的分配边界而引起修改的可能性。（Windows 2000 内核模式代码不受这种限制；它能够在单页粒度上保留内存。）

最后，当你保留一个地址空间区域时，无论有多大，Windows 2000 会确保该区域的尺寸是系统页的倍数。例如，由于 x86 系统使用 4KB 的页面，如保留 18KB 大小的区域，在 x86 系统上实际保留的数量将是 20KB。

7.2.4 共享内存和映射文件

像大多数现代操作系统一样，Windows 2000 提供了在进程和操作系统之间共享内存机制。"共享内存"（shared memory）可以定义为对一个以上的进程是可见的内存或存在于多个进程的虚拟地址空间。例如，如果两个进程使用相同的 DLL，只把 DLL 的代码页装入内存一次，其他所有映射这个 DLL 的进程只要共享这些代码页就可以了，如图 7-3 所示。

每一个进程将一直保留它的存储专用数据的专用内存区域，程序指令和不修改的数据页面能被共享而不会损坏。正如在稍后将要解释的，这种共享是自动发生的，因为可执行的映像中的代码页被映射为只执行（execute - only）、并且可写页面被映射为写时复制（详细信息请参阅 7.2.6 节"写时复制"）。

在内存管理器中用于实现共享内存的基本原语叫作"区域对象"。在 Win32 API 中，它们

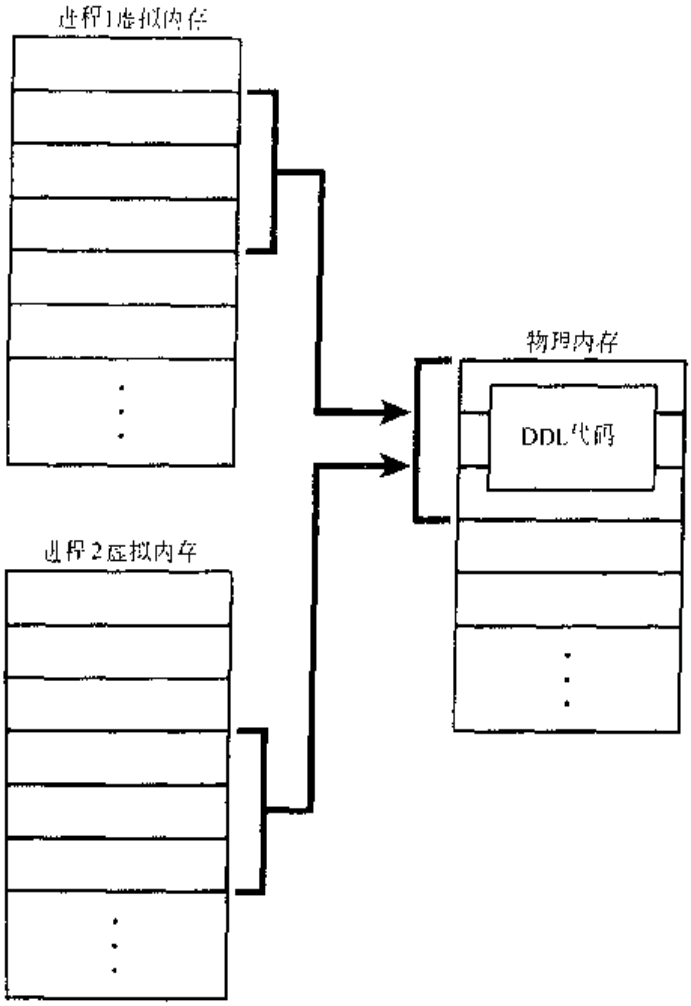


图 7-3 进程间共享的内存

被称作“文件映射对象”。内部结构和区域对象的实现将在本章后面描述。

内存管理器中的这个基本原语被用来映射虚拟地址，不管它是在主内存、页面文件还是在应用程序想要访问的其他文件中，虚拟地址仿佛就在内存中。一个区域可以被一个进程打开也可以被多个进程打开；换句话说，区域对象并不一定意味着共享内存。

区域对象可以被连接到在磁盘上已打开的文件（称为映射文件）或提交内存（以提供共享内存）上。如果内存命令指示，页面可以写到调页文件，所以映射到提交内存的页称为支持页面文件的区域（page file backed section）（由于 Windows 2000 在没有调页的情况下也可以运行，支持页面文件的区域实际上可能仅仅由物理内存“支持”）。作为专有的提交页面，当它们第一次被访问时，共享的提交页面总是由零填充。

要创建区域对象，可以调用 Win32 的 CreateFileMapping 函数，其参数包括它被映射到文件的文件句柄（或 INVALID-HANDLE-VALUE 用于支持页面文件的区域）、任选的名称以及安全描述符。如果区域有名称，其他的进程就能够使用 OpenFileMapping 打开它或者可以通过句柄继承（在打开或创建句柄时，通过指定句柄为可继承）或句柄复制（通过使用 DuplicateHandle）准予对区域对象的访问。设备驱动程序可以使用 ZwOpensection、ZwMapViewOfSection 和 ZwUnmapViewOfSection 函数操作区域对象

区域对象可以引用比进程地址空间能够容纳的文件大得多的文件（如果调页文件支持区域对象，就必须有足够的空间来容纳它）。进程通过调用 MapViewOfFile 函数映射它需要的区域对象的部分（称为区域的一个视图（view）），然后指定映射的范围，可以访问一个非常大的区域对象。映射视图允许进程保留地址空间，因为在这个时候只有区域对象的视图必须被映射到内存。

Win32 应用程序可以通过简单地使映射文件出现在它们的地址空间来完成到文件的 I/O。但是用户应用程序并不是区域对象的唯一用户：映像装载器使用区域对象映射可执行映像、动态链接库（DLL）和设备驱动程序到内存中，高速缓存管理器使用区域对象访问缓存文件的数据（高速缓存管理器与内存管理器集成的信息请参阅第 11 章。）

在本章的后面部分将解释根据地址转换和内部数据结构来实现共享内存区域的方法。

7.2.5 保护内存

正如第 1 章解释的，Windows 2000 为进程和操作系统本身提供内存保护，这样就没有用户进程能够偶然或故意破坏另一个进程或操作系统本身的地址空间。Windows 2000 通过四种主要的方法提供保护：

首先，所有系统范围内的被系统内核模式组件使用的数据结构和内存交换区只能在内核模式下被访问——用户模式线程不能够访问这些页面。如果它们试图这样做，硬件将产生错误，内存管理器会报告线程访问违例。

注意 与之相反，Microsoft Windows 95、Microsoft Windows 98 和 Windows 2000 在地址空间中有一些在用户模式下可写的页面，这样就使错误的应用程序能够破坏关键的系统数据结构并使系统崩溃。

其次，每一个进程拥有一个独立的、专用的地址空间，禁止其他进程的任何线程访问，除非进程与另一个进程使用共享页面或另一个进程使用虚拟内存读写访问进程对象，并且调用 ReadProcessMemory 或 WriteProcessMemory 函数。每次线程引用一个地址，虚拟内存硬件和内存管理器共同工作，干预并转换虚拟地址到物理地址。通过控制虚拟地址如何转换，Windows 2000 能够确保在一个进程内运行的线程不会错误地访问属于其他进程的页面。

再次，除了提供虚拟地址到物理地址转换的隐含保护，Windows 2000 支持的所有处理器都提供了某些形式的硬件控制的内存保护（如读/写、只读等等），这种保护的具体细节因处理器的不同而不同。例如，在进程地址空间内的代码页面被标记为只读，这样就禁止了用户线程对它们的修改。已加载的设备驱动代码页面类似地标记为只读。

注意 默认时，在带有 128MB 或更多物理内存的系统上 Ntoskrnl.exe 或 Hal.dll 不使用系统代码页保护。在这样的系统上，Windows 2000 映射带有大页面（4MB）的系统地址空间的前 512MB 以便提高转换后备链表的效率。（本章后面解释）由于映像区域使用 4KB 粒度映射，意味着映像的代码部分可以和数据部分驻留在同一页面。这样，把页面标记为只读将防止修改页面的数据。可以增加 DWORD 注册表键值 HKLM \ SYSTEM \ CurrentContolSet \ Control \ SessionManager \ MemoryManagement \ LargePageMinimum 覆盖此值，作为系统映射带大页面的 Ntoskrnl 和 HAL 使用的兆字节数。

表 7-3 列出了 Win32 API 中定义的内存保护选项（请参阅 VirtualProtect、VirtualProtectEx、VirtualQuery 和 VirtualQueryEx 函数）。

表 7-3 Win32 API 定义的内存保护选项

属 性	描 述
PAGE-NOACCESS	任何试图对这个区域中代码的读取、写入或执行操作都会产生访问违例
PAGE-READONLY	任何试图对在内存中代码的写入或执行操作都会产生访问违例，但允许读取
PAGE-READWRITE	该页面是可以读写的 没有操作会产生访问违例
PAGE-EXECUTE[1]	任何试图对在这个区域中的在内存中代码的读取 写入操作都会产生访问违例，但允许执行
PAGE-EXECUTE-READ[2]	任何试图对在这个区域中的在内存中代码的写入访问违例，但允许读取和执行
PAGE-EXECUTE-READWRITE[2]	该页面是可读的、可写的和可执行的——没有操作会产生访问违例
PAGE-WRITECOPY	任何试图对这个区域中内存的写入将导致系统给进程一个该页面的专用副本。试图执行这个区域中的代码会产生访问违例
PAGE-EXECUTE-WRITECOPY	任何试图对这个区域中内存的写入将导致系统给进程一个该页面的专用副本
PAGE-GUARD	任何试图读取或写入保护页面（guard page）的操作都会产生 EXCEPTION-GUARD- PAGE 异常并将关闭保护页状态这样保护页就作为一次性报警。注意：除了 PAGE-NOACCESS 以外在表中列出的任何页保护项都可以指定这个标志

注：(1) - x86 体系结构未实现只执行访问（即，代码可以在任何可读的页面执行），因此 Windows 2000 不在任何实际意义上支持只执行访问（尽管 IA - 64 支持）。Windows 2000 把 PAGE-EXECUTE-READ 处理为 PAGE-READONLY，把 PAGE-EXECUTE-READWRITE 处理为 PAGE-READWRITE

最后，共享内存区域对象具有标准的 Windows 2000 访问控制表（ACL），当进程试图打开共享内存区域对象时会检查 ACL，这样就把对共享内存的访问限制在那些有适当权限的进程之内。当线程创建一个区域来包含一个映射文件时，安全性就会起作用。要创建该区域，线程必须至少对此文件对象具有读的访问权限，否则该操作将失败。

一旦线程成功地为区域打开了句柄，它的行为仍将服从前面所讲的内存管理器和基于硬件的页面保护。如果对页面级保护的更改不违反区域对象在 ACL 中的权限，线程就可以在区域内更改对虚拟页面的页面级保护。例如，内存管理器允许线程把只读区域的页面更改为写时复制访问权限，但不能具有读/写访问权限。允许写时复制权限是因为它不影响共享数据的其他进程。

这四种主要的内存保护机制是 Windows 2000 拥有健壮、可靠的操作系统声誉的部分原因，

它不受应用程序错误的影响，并可以从应用程序错误中恢复。

7.2.6 写时复制

写时复制页面保护是内存管理器用来节约物理内存的优化技术。当进程映射包含读/写页面的区域对象的写时复制视图，而不是在映射视图的同时制作进程专用副本时（如同 Compaq OpenVMS 操作系统所做的那样），内存管理器延迟制作页面的副本直到页面被修改后。所有现代的 UNIX 系统也使用这种技术。例如，如图 7-4 所示，两个进程共享三个页面，每个页面都被标记为写时复制。但是两个进程都不会试图修改页面的数据。

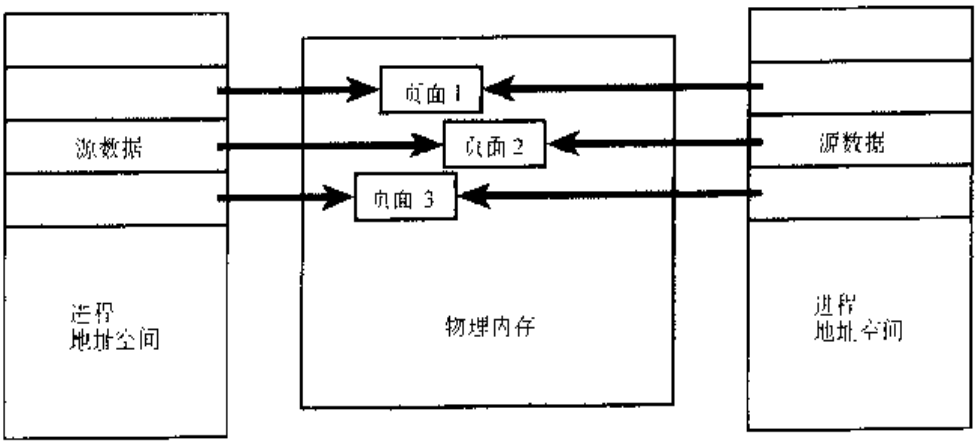


图 7-4 写时复制之前

如果任何一个进程中的线程写入该页面，就会产生内存管理错误。内存管理器观察到写操作是作用于写时复制页面，因此它不是以访问违例报告该错误，而是在物理内存中分配一个新的读/写页面，复制原始页面的内容到新页面，更新在这个进程中相应的页面映射信息（将在本章后面解释）来指向新的位置并取消异常，这样导致产生错误的指令重新被执行。此时就可以成功地进行写操作，但是如图 7-5 所示，新复制的页面是进行写操作的进程专用的，并且对共享写时复制页面的其他进程是不可见的。每个写到相同共享页面的进程都将得到它的专用副本。

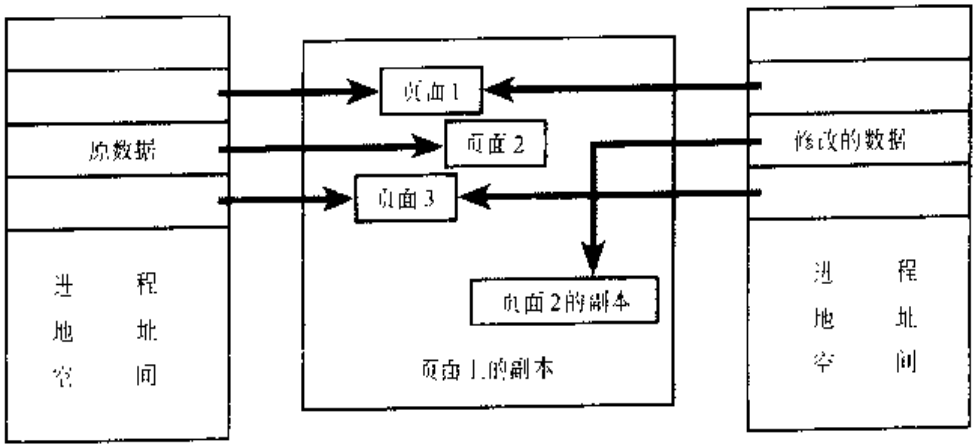


图 7-5 写时复制之后

写时复制被用于实现调试程序中的断点支持。例如，在默认状态下，代码页作为只能执行（execute - only）开始。然而，如果程序员在调试程序时设置一个断点，调试程序就必须添加断

点指令到代码中。要完成这项工作，它首先要更改页面的保护为 PAGE-EXECUTE-READWRITE，然后更改指令流。因为代码页是映射区域的一部分，内存管理器为带有断点设置的进程创建专用的副本，而其他的进程仍然继续使用未更改的页面。

写时复制技术是一个称为“惰性求值”(lazy evaluation) 的求值技术的例子，它是内存管理器经常使用的。惰性求值算法避免完成昂贵的操作，除非是在绝对需要时——如果从不需要此项操作，在它上面就不会浪费任何时间。

POSIX 子系统利用了写时复制实现 fork 函数。典型的，当 UNIX 应用程序调用 fork 函数创建另一个进程时，新进程做的第一件事就是调用 exec 函数用可执行程序重新初始化地址空间。新进程通过标记父进程中的页面为写时复制来共享这些页面，而不是在 fork 中复制整个地址空间。如果子进程要向页面写数据，就制作进程的专用副本。否则，两个进程将继续共享并且不进行复制。无论如何，内存管理器只复制进程中试图写入的页面而不是整个地址空间。

要检验写时复制错误比例，请观察 Memory: Copies/Sec 性能计数器。

7.2.7 堆函数

堆是保留的一个或多个页面的区域，通过堆管理器 (heap manager) 提供的一组函数可以把这个区域分开并分配成较小的块。堆管理器是一组用于分配和回收可变的内存量的函数（不需要象 VirtualAlloc 函数一样在页面粒度上分配）。堆管理器函数存在于两个地方：Ntdll.dll 和 Ntoskrnl.exe。子系统 API (例如 Win32 堆 API) 调用 Ntdll 中的函数，而各种执行程序组件和设备驱动调用 Ntoskrnl 中的函数。

每个进程以一个默认进程堆开始，通常是 1MB 大小（除非在映像文件中使用/HEAP 链接程序标志设定了其他值），这个大小是初始保留值——在需要的时候它会自动扩展（可以指定映像文件中的初始提交值）。一些可能需要分配临时内存块的 Win32 函数和 Win32 应用程序将使用这个进程默认堆。进程也可以使用 HeapCreate 函数创建另外的专用堆。当进程不再需要专用堆时，它可以通过调用 HeapDestroy 函数恢复虚拟地址空间。在进程的生存期内，只有使用 HeapCreate 创建的专用堆——而不是默认堆——能够被破坏。

为了从默认堆中分配内存，线程必须通过调用 GetProcessHeap 得到一个指向它的句柄（这个函数返回描述这个堆的数据结构的地址，但调用程序从来也不依赖它）。有了堆句柄以后，线程就可以调用 HeapAlloc 和 HeapFree 来从堆中分配和释放内存。堆管理器也为每个堆提供一个选项来串行化分配和回收，这样多线程就可以同时调用堆函数而不会破坏堆数据结构。默认进程堆被默认地设置为具有此串行化功能（尽管可以通过 call - by - call 原则重载）。对于另外的专用堆，传递给 HeapCreate 函数的标志被用来指定是否应该完成串行化功能。

堆管理器支持大量的内部确认检查。虽然当前没有添加到文档中，但是利用 Windows 2000 的 Support Tools、Platform SDK 和 DDK 中的工具 Global Flags (Gflags.exe)，你可以在系统范围或单个映像的基础上进行这种确认检查。按照它们引起堆管理器的行为，许多标记都是自解释的。一般，允许这些标记通过使用异常或返回的错误代码将导致堆的无效使用或堆的破坏向应用程序产生异常通知。

关于堆函数的详细信息，请参阅 MSDN 中的 Win32 API 函数参考文档。

7.2.8 地址窗口扩展

虽然 Windows 2000 系统能够支持高达 64GB 的物理内存（如表 2-2 所示），但是每个 32 位的用户进程都只有 2GB 或 3GB 的虚拟地址空间（这取决于是否打开了/3GB 引导开关）。为了让 32 位进程能够分配并访问更多的物理内存，Windows 2000 系统提供了一组称为地址窗口扩展（Address Windowing Extensions，AWE）的函数。例如，在具有 8GB 物理内存的 Windows 2000 Advanced Server 系统中，数据库服务器应用程序能够利用 AWE 来分配并使用接近于 8GB 的内存作为数据库高速缓存。

通过 AWE 函数分配并使用内存有三个步骤：

- 分配将要使用的物理内存。
- 创建一个虚拟地址空间的区域，以此作为一个窗口来映射物理内存的视图。
- 将物理内存的视图映射到该窗口。

为了分配物理内存，应用程序调用 Win32 函数 AllocateUserPhysicalPages（该函数需要内存用户权限中的锁定页面），然后使用带 MEM-PHYSICAL 标志的 Win32 VirtualAlloc 函数在该进程地址空间的专用部分创建一个窗口，该窗口被映射到前面分配的部分或全部物理内存。然后，AWE 分配的内存几乎可以被所有的 Win32 API 函数使用。（例如，Microsoft DirectX 函数不能使用 AWE 内存。）

如果应用程序在其地址空间创建一个 256MB 的窗口，并分配 4GB 的物理内存（系统物理内存多于 4GB），那么通过将内存映射到 256MB 的窗口，应用程序借助 Win32 函数 MapUserPhysicalPages 或 MapUserPhysicalPagesScatter 访问物理内存的任何位置。应用程序虚拟地址空间窗口的大小决定了应用程序在给定的映射中能够访问的物理内存的数量。图 7-6 显示了服务器应

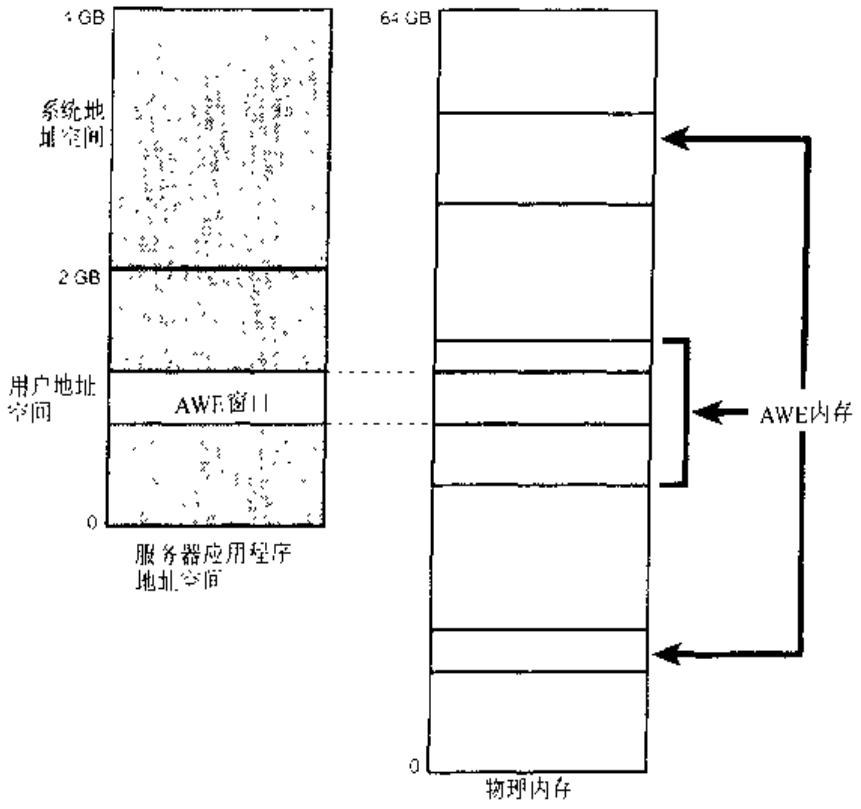


图 7-6 使用 AWE 映射物理内存

用程序地址空间中的一个 AWE 窗口，该窗口映射到先前通过 AllocateUserPhysicalPages 分配的物理内存的一部分。

AWE 函数存在于所有的 Windows 2000 版本中，并且不管系统具有多少物理内存都能够使用。然而，AWE 在物理内存多于 2GB 的系统中最常用，因为这是 32 位的进程直接访问 2GB 以上内存的唯一方法。

最后，对 AWE 函数分配和映射的内存有些限制：

- 页面不能在进程间共享。
- 同一物理页面不能在同一进程中映射到多个虚拟地址。
- 页面保护限制为读/写。

对于在物理内存多于 4GB 的系统中用来映射内存的页面表数据结构的描述，请参阅 7.5.7 节“物理地址扩展”。

7.3 系统内存交换区

在系统初始化阶段，内存管理器将创建两个可以动态调整大小的内存交换区，内核模式组件使用这两个内存交换区分配系统内存：

- 非页交换区　由系统虚拟地址的范围组成，在任何时候都要保证它们驻留在物理内存中，这样就可以在任何时候访问（从任何 IRQL 级和任何进程环境）而不发生页面错误。第 2 章已经描述了需要非页交换区的准则：在 DPC/调度或更高级别上页面错误不能满足。
- 页交换区　在系统空间中的虚拟内存区域，它能够被页面调入和页面调出，设备驱动不需要从 DPC/调度或更高级别中访问内存而可以使用页交换区。从任何进程环境都可以访问页交换区。

这两个内存交换区都位于地址空间的系统部分，并且被映射到每一个进程中的虚拟地址（可以在页面表 7-10 里找到它们在系统内存中的起始位置）。执行程序提供了从这些交换区中分配和回收内存的例程。关于这些例程的详细信息，请参阅 Windows 2000 DDK 文档中从 ExAllocatePool 开始的函数。

这里有两种类型的非页交换区：一种是在通常情况下使用的，另一种较小的交换区（四个页面）被保留在当非页交换区已满并且调用程序又不能容忍分配失败时的紧急情况下使用（第二种交换区类型将不再使用，应该编写设备驱动程序合适地处理低内存的情况。驱动程序检验器（driver verifier），本章后面讨论，将使测试这些条件变得更为容易）。单处理器系统有三个页交换区，多处理器系统有五个。不止一个页交换区减少了系统代码在同步调用交换区例程时发生堵塞的频率。这些交换区的初始大小是在系统初始化时计算的，并且它们依赖于内存的大小。必要时页交换区和非页交换区会自动增加到在系统引导时计算的系统定义的最大值。可以通过将注册表键 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manger \ Memory Management \ 中的 NonPagedPoolSize 和 PagedPoolSize 的键值从 0（这将导致系统计算大小）改变为希望的字节大小来重新设置这些交换区的初始大小。然而，不能超过表 7-4 列出的交换区大小的最大值。

表 7-4 最大交换区的大小

交换区类型	最大值
非页式	256MB（如果用/3GB 启动则为 128MB）
页式	491 875MB

计算完的大小存储在四个内核变量中，其中的三个能够通过性能计数器查询，这些变量和计数器以及两个能够更改大小的注册表键列在表 7-5 中。

表 7-5 系统交换区变量大小和性能计数器

内核变量	性能计数器	重载的注册表键	描述
MmSizeOfNonPaged – PoolIn Bytes	Memory: Pool NonPaged Bytes	没有	非页交换区的当前大小
MmMaximumNon – PagedPoolIn Bytes	没有	HKLM \ SYSTEM \ Current-ControlSet \ Control \ Session Manger \ Memory Management \ NonPagedPoolSize	非页交换区的最大值
没有	Memory: Pool Paged Bytes	没有	页交换区的当前虚拟大小
MmpagedPoolPage（页面数）	Memory: Pool Paged Resident Bytes	没有	页交换区的当前物理（驻留）大小
MmSizeOfPagedPool – InBytes	没有	HKLM \ SYSTEM \ Current-ControlSet \ Control \ Session Manger \ Memory Management \ PagedPoolSize	页交换区的最大值（虚拟）

实验：确定最大交换区

由于页交换区和非页交换区代表着关键的系统资源，因此了解系统何时接近最大值以便决定是否需要使用适当的注册值重载默认的最大值是很重要的。然而，交换区大小性能计数器只报告当前值，而不是最大值。因此在消耗交换区之前，你不能知道何时接近最大值。

利用 LiveKd，检查列举在表 7-5 中的内核变量的值，你可以很容易地查看最大值。下面是一个例子：

```
kd> dd mmmaximumnonpagedpoolinbytes l1
804/f620  0328c000
kd> ? 328c000
Evaluate expression: 53002240 = 0328c000
kd> dd mmsizeofpagedpoolinbytes l1
804/0a98  06800000
kd> ? 6800000
Evaluate expression: 109051904 = 06800000
```

从这个例子，你可以看到非页交换区的最大值为 53，002，240 字节（约 50MB），页交换区的最大值为 109，051，904 字节（104MB）。在本例使用的该测试系统中，当前使用的非页式交换区为 5.5MB，页交换区是 34MB，因此两个交换区都未接近最大值。（要快速查看这些数字，请运行 Task Manager，单击 Performance 标签，查看 Kernel Memory 部分）。

实验：监视交换区的使用

对于非页交换区和页交换区（虚拟的和物理的）的大小，内存性能计数器对象都有各自的计数器。除此以外，Poolmon 工具（在 Windows 2000 Support Tools 中）允许你监视非页交换区和页交换区的详细使用情况。要实现这一功能，你必须允许内部的 Enable Pool Tagging 选项（在可调试版本中，Pool 标签总是打开的，因此你只需在纯运行版本中打开它）。如果你要打开 Pool 标签，请运行 Windows 2000 Support Tools、Platform SDK 或 DDK 中的 Gflags 工具，并选择如图 7-7 所示的 Enable Pool Tagging：

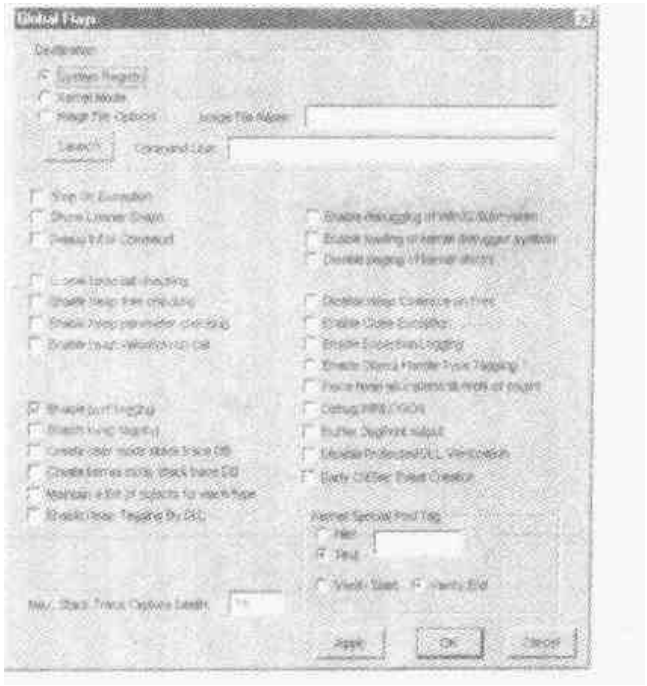


图 7-7 Enable Pool Tagging 显示

然后单击 Apply，并重新引导系统。启动系统后，运行 Poolmon，你将看到类似图 7-8 的显示：

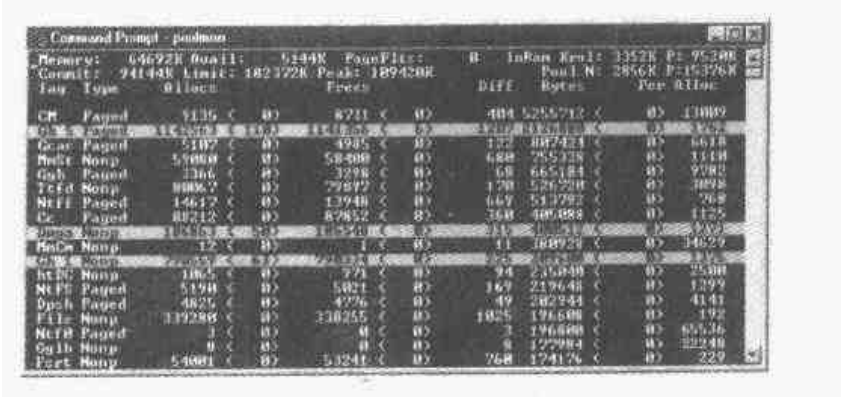


图 7-8 显示 Poolmon 运行结果

加亮的行代表对显示的改变（你可以在运行 Poolmon 时输入 l 来取消加亮特性）。在运行 Poolmon 时输入？键，将弹出它的帮助屏幕。你可以配置所要监视的交换区（页式、非页式，

或者两者混合）以及排序顺序。同时还显示了用来监视具体结构（或某个结构类型以外的所有类型）的命令行选项。例如，命令 poolmon -iCM 将仅监视 CM 类型（配置管理器，它管理注册表）的结构 各栏的含义如下：

栏	解释
Tag	给予交换区分配的四字节标签
Type	交换区类型（页交换区或非页交换区）
Allocs	所有分配交换区的总数（括号中数字显示自最近一次更新以来 Allocs 栏的差异）
Frees	所有 Frees 的总数（括号中数字显示自最近一次更新以来 Frees 栏的差异）
Diff	Allocs 减去 Frees
Bytes	该结构类型消耗的字节总数（括号中数字显示自最近一次更新以来 Bytes 列的差异）
Per Alloc	该结构类型的单一实例的字节大小

在这个例子中，CM 结构占据了大部分页式交换区，而 MmSt 结构（用于映射文件的内存管理相关的数据结构）则占据了大部分非页交换区。

你还可以使用内核调试器命令！poolused 和！xpool 查看交换区的使用情况。（！xpool 在辅助的内核调试器扩展 DLLKdex86.dll 中。）命令！poolused 2 按照使用大多数交换区的结构标签的排序显示非页交换区的使用，命令！poolused 4 列举页交换区的使用，也是按照使用大多数交换区的结构标签排序。下面的例子显示了这两个命令的部分输出结构：

```
kd>  .load kdex2x86
kd>  !poolused 2
   Sorting by NonPaged Pool Consumed
   Pool Used:
            NonPaged            Paged
  Tag    Allocs     Used    Allocs     Used
  File     1702   326784         0        0
  Thre      376   240640         0        0
  Vad      3167   202638         0        0
  Ntf0        3   196608       367    51264
  Even     2869   187254         0        0
  Npfs      335   154080       221    41504
  Devi      249   148480         0        0
  WDMA      333   146176       319    49984
  Pool        3   134368         0        0
  Irp       305   134272         0        0
  AmlH        2   131072         0        0
.
kd> !poolused 4
   Sorting by Paged Pool Consumed

   Pool Used:
            NonPaged            Paged
  Tag    Allocs     Used    Allocs     Used
```

```
CM          11      704      5146 22311904
Gh o         0        0       727  2523648
MmSt         0        0       968  1975872
Nt'f        10     2240       790   682560
Glal         1      128       383   600544
Tt'd         0        0       265   545440
Glab         1      128      1099   457184
NtFB         0        0         2   376832
IXMK         0        0       948   319680
Gla:         1      128       444   298368
Obtb        52     6656        68   278528
NtfF         0        0       250   248000
Gcac         0        0        54   243424
Mmpp         0        0        55   225280
Grgb         0        0         5   163840
:
```

命令！xpool – map 显示非页交换区（！xpool – map 1 显示页交换区的情况）中每个页面的状态以及使用的总页面。以下是该命令的部分例子：

```
kd> !xpool -map

Status Map of Pool Area Pages
=============================
 '0': one page in use                                ('P': paged out)
 '<': start page of contiguous pages in use          ('[': paged out)
 '>': last page of contiguous pages in use           (']': paged out)
 '-': intermed·ate page of contiguous pages in use   (' ': paged out)
 '.': one page not used

Non-Paged Pool Area Summary
--- -----------------------
Maximum Number of Pages  = 12940 pages
Number of Pages In Use   =  1459 pages (11.3%)

       +00000  +08000   +10000  +18000   +20000  +28000   +30000  +38000
81093000: ....C000000000000 0000000000000000 0000000000000000
          0000C000<=>00000
810d3000: 0000000000000000 0000000000000000 0000000000000000
          <=>0000000000000
81113000: 0000C000000000000 0000000000000000 0000000000000000
          0000000000000000
81153000: 0000000000000000 C000000000000000 0000000000000000
          00000000000000<=>
81193000: 0000000000000000 C000000000000000 0000000000000000
          0000000000000000
811d3000: 0000000000<=>000 C000<>0000000000 0000000000000000
          0000000000000000
81213000: 000000000000<=== ><============== ===============>
          <=======-===><--=
81253000: ================ ================ ================
```

```
          =====><=========
81293000: ================ ================ ================
          ================
812d3000: ==><=========== 0000000000000000 0000000000000000
          00000000000C00<==
81313000: ================ ====><======>0<> 000000<>000<=>00
          00<======>0<====
81353000: ==>0000000000000 <================ ==><==>000<><===
          ===>0<=>00000000
81393000: 0<===>00000000<= ========>00C0000 0000000000000000
          000000000<=><===
813d3000: ====>00<>0C000<= =====><=>0000000 000000000000<===
          ==========-><-==
81413000: ==========>0000 00000<========== ====>00000000000
          0000000000<=====
81453000: =========>000000 00C0<=>00000000< ==============>0
          0000<==========>
81493000: 0000000000000000 <======>
fcd6c000: ................ ................ ................
          ................
fcdac000: ................ ................ ................
          ...... ........
fcdec000: ................ ................ ................
          ................
!

kd> !xpool -map 1

Status Map of Pool Area Pages
-----------------------------
  '0': one page in use                                ('P': paged out)
  '<': start page of contiguous pages in use          ('{': paged out)
  '>': last page of contiguous pages in use           ('}': paged out)
  '=': intermediate page of contiguous pages in use ('-': paged out)
  '.': one page not used

Paged Pool Area Summary
-----------------------
Maximum Number of Pages  = 26624 pages
Number of Pages In Use   =  8685 pages (32.6%)
       +00000  +08000   +10000  +18000   +20000  +28000   +30000  +38000
e1000000: 0C000<>00<><>P00 0000000PPPPPPP0 PPPPPPPPPPPPP0P0
          0PPPPPPPPPPPP000
e1040000: 00PP0PPPPPPP0000 00000P000PPPPPPP P0000000000000000
          P0{}P0PP0P0PPPPP
e1080000: 0PPP00PPPPPPPP00 {>PPPPPPP0PPPPPP PPPPPPPPPPPPPP00
          0P{}PPPPPPPPPPP
e10c0000: PPPPPPPPPPPPPPP PPPPPPPPPPPPPPP PPPPPPPPP00PPPPP
          000000P0000000
e1100000: P000000000000{>0 0000000000000000C 000000000000P000
          0000000000000000
e1140000: 00PP000000000000 0PP0C0000<>00PPC 0000PPPPPP0PPPPP
          PPPPPPPPP0PP0PPP
```

```
e1180000: PPPPPPPPPPOOPPPP PPPPPPPPPPPPP?PP PPPPOPPPPPPPPPPP
          PPPPPPPPPPPPPOP{
e11c0000: >OPPOPPPPPPPPPPP POPPPPPPPPPPP{}P PPPPPPPPPPPPPPPP
          PPPPPPPPPPPP{}P
e1200000: PPPPPPPPPPPPPPPP PPPPPPOPPPOPPOPP PPPPPPPPPPPPPPPP
          PPPPPPPPPPPPPPPP
e1240000: PPPPOPPPPPPPOOPP PPPPPPPPP{}PPPPP PPPPPPPPPPOPOPPP
          OPOOPPPPPPPPPPOP
e1280000: PPPPPPOPPPPPOOPP POPOPPPPPPPPP<>O PPPOPO{>PP?POPOO
          POPP{}POPPOPPPPP
e12c0000: PPPOOPOOOPOPPPPP PPPPJOOPPPPPPPPP PPPPPPOOPPOPPPOO
          POOPOOOOOOPOPPOO
:
```

7.3.1 后备列表

Windows 2000 还提供一种称为“后备列表”（look - aside list）的快速内存分配机制。交换区和后备列表的基本区别在于：一般交换区分配可以改变大小，而后备列表只包含固定大小的块。因此，尽管从一般交换区可以提供的大小方面来说，它比较灵活。但是由于不必搜寻适合不同分配大小的空闲内存和不必使用自旋锁，后备列表将执行得更快。

执行程序组件和设备驱动程序能够利用 ExInitializeNPagedLookasideList 和 ExInitializePagedLookasideList 函数（其文档在 DDK 中）创建大小与经常分配的数据结构匹配的后备列表。为了最小化多处理系统的同步负担，几个执行程序子系统（如 I/O 管理器、高速缓存管理器和对象管理器等）为每个处理器经常访问的数据结构创建单独的后备列表。该执行程序还为小的分配（256 字节或更小）创建各个处理器通用的页式后备列表和非页式后备列表。

如果后备列表为空（首次创建时，它是空的），那么系统必须从页交换区或非页交换区分配空间。但是如果它包含一个空闲的结构，那么该分配可以快速满足（列表随着结构的返回而增长）。交换区分配例程按照设备驱动程序或执行子系统从后备列表中分配空间的频率自动调节存储在该表中的空闲缓冲区的大小——分配的频率越高，列表中的缓冲区就越多。如果它们不从该列表中分配，后备列表的大小将自动减少（当平衡集管理器系统被唤醒并调用 KiAdjustLookasideDepth 函数时该检查每秒钟进行一次）。

实验：观察系统后备列表

使用内核调试器命令 ! lookaside 可以显示各种系统后备列表的大小和内容。以下摘录来自该命令的输出：

```
kd> !lookaside

Lookaside "nt!IopSmallIrpLookasideList" @ 804758a0 "Irps"
    Type     =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      148   Max Alloc  =      592
    AllocateMisses =       32   FreeMisses =        9
    TotalAllocates =       52   TotalFrees =       32
    Hit Rate       =       38%  Hit Rate   =       71%
Lookaside "nt!IopLargeIrpLookasideList" @ 804756a0 "Irpl"
```

```
    Type      =     0000 NonPagedPool
    Current Depth  =        4   Max Depth  =        4
    Size           =      436   Max Alloc  =     1744
    AllocateMisses =     2623   FreeMisses =     2443
    TotalAllocates =     7039   TotalFrees =     6863
    Hit Rate       =      62%   Hit Rate   =      64%

Lookaside "nt!IopMdlLookasideList" @ 80475740 "Mdl "
    Type      =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      120   Max Alloc  =      480
    AllocateMisses =     7017   FreeMisses =     1824
    TotalAllocates =    10901   TotalFrees =     5711
    Hit Rate       =      35%   Hit Rate   =      68%

Total NonPaged currently allocated for above lists = 4200
Total NonPaged potential for above lists           = 6144
Total Paged currently allocated for above lists    = 5136
Total Paged potential for above lists              = 12032
```

7.3.2 驱动程序检验器

驱动程序检验器（Driver Verifier）是一种用来协助查找并隔离设备驱动程序或内核模式系统代码中经常发现的错误的机制。Microsoft 使用驱动程序检验器检测所有产商提交的供 Hardware Compatibility List（HCL）测试的设备驱动程序。这样做确保了 HCL 中的驱动程序与 Windows 2000 兼容并且没有一般的驱动程序错误。

驱动程序检验器由几个系统支持组件组成：内存管理器、I/O 管理器、HAL 以及 Win32k.sys，每个都具有可以打开的驱动程序检验选项。本节介绍与驱动程序检验器提供的检验选项有关的内存管理。

驱动程序检验器的配置与初始化

请运行 Driver Verifier Manager（\Winnt\System32\Verifier.exe），配置驱动程序检验器并查看有关该操作的统计信息。如图 7-9 所示，当你运行驱动程序检验器时，它将显示几个标签页

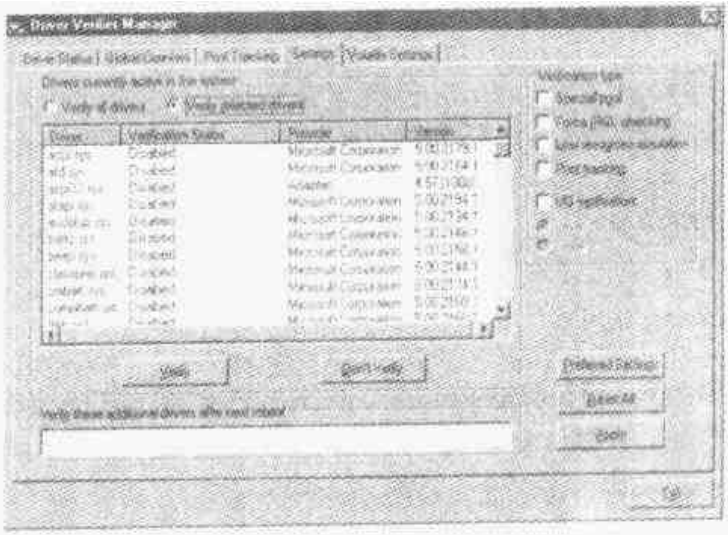


图 7-9 驱动程序检验器

面。你可以利用 Setting 标签指定所要检验的设备驱动程序（有一个 Verify All Drivers 的选项）以及所要进行的检验类别。

设置存储在注册表 HKLM \ SYSTEM \ CurrentContrilSet \ Control \ Session Manager \ Memory Management 下。VerifyDriverLevel 键值包含一个表示所允许的检验类别的位掩码。VerifyDrivers 键值包含需要确认的驱动程序的名字（这些键值直到你在 Driver Verifier Manager 中选择驱动程序进行检验时才会出现在该注册表中）。如果选择检验所有的驱动程序，VerifyDrives 将被设置成星号（*），在输入或改变了驱动程序检验器的设置之后，你可能需要重新启动计算机以便进行检验。

在启动过程的早期，内存管理器读取注册表键值 Driver Verifier 的值，决定哪些驱动程序需要检验以及驱动程序检验器的哪些检验选项被打开。随后，如果你选择了至少一个驱动程序进行检验，那么内核将检查你选所择进行检验的驱动程序列表中加载到内存中的每个设备驱动程序的名字。对于每个出现在这两个地方的设备驱动程序，内核将调用 MiApplyDriverVerifier 函数。该函数用对驱动程序检验器的引用代替大约对 40 个内核函数的引用，驱动程序检验器是这些函数的等价版本。例如，用对 VerifierAllocatePool 的调用代替 ExAllocatePool。窗口系统驱动程序也利用 Driver Verifier 函数进行类似的修改。

既然已经回顾了如何设置驱动程序检验器，现在让我们来检查一下四个与内存相关的可以用于设备驱动程序的检验选项：Special Pool、Pool Tracking 、Force IRQL Checking 以及 Low Resources Simulation。

Special Pool Special Pool 选项将使交换区分配例程把交换区分配处理为无效页面，这样分配之前或分配之后的引用都将导致内核模式的访问违例，从而使带有指向有缺陷的驱动程序的指针的系统崩溃。Special Pool 还将产生一些在驱动程序分配或释放内存时进行的确认检查。

当专用交换区打开时，交换区分配例程将分配一块内核内存区域供驱动程序检验器使用。驱动程序检验器将在检验的情况下的驱动程序的内存分配请求重定向到专用交换区，而不是标准的内核模式内存交换区。当设备驱动程序从专用交换区分配内存时，驱动程序检验器将地址分配集拢到偶数页边界。因为驱动程序检验器将分配的页面处理为无效页面，如果某个设备驱动程序试图读写超过缓冲区边界的部分，它将访问到一个无效的页面，从而内存管理器将产生一个内核模式访问违例。

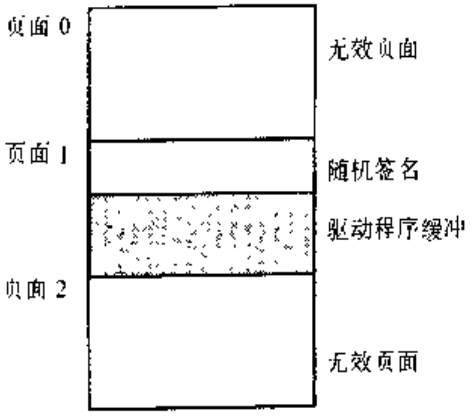


图 7-10 专用交换区分配格局

图 7-10 显示了一个专用交换区缓冲区的例子，驱动程序检验器在检查超限错误时将该缓冲区分配给设备驱动程序。

在默认情况下，驱动程序检验器执行超限（overrun）检测。它将设备驱动程序使用的缓冲区放在所分配的页面尾端，并用随机的模式填充该页的起始部分，从而实现这种检测。虽然 Driver Verifier Manager 不允许你指定 underrun 检测，但是你可以手工设置这种类型检测，方法是添加 DWORD 类型的注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \

MemoryManagement \ PoolTagOverruns 并将其置为 1（或者是运行 Gflags 工具并选择 Verify Start 选项取代默认的 Verify End 选项）。当 Windows 2000 强制进行 underrun 检测时，驱动程序检验器将在页面的起始部分分配驱动程序的缓冲区，而不是在页的尾端。

超限（overrun）检测配置还包括一些 underrun 检测措施。当驱动程序释放缓冲区并将内存返回给驱动程序检验器，驱动程序检验器将确认缓冲区之前的模式没有被改变。如果该模式改变了，设备驱动程序将就已经越过了缓冲区并写了缓冲区之外的内存。

专用交换区分配还将检查以确保分配与释放时处理器 IRQL 是合法的。该检查能够捕获某些设备驱动程序产生的错误：从 DPC/调度级或更高级别的 IRQL 分配可调页的内存。

你还可以通过添加 DWORD 类型的注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ MemoryManagement \ PoolTag 而手工配置专用交换区，该注册表键值代表系统用于专用交换区的分配标签。这样，即使驱动程序检验器没有配置成检验某个特定的设备驱动程序，如果与驱动程序分配的内存关联的标签与注册表键值 PoolTag 的值匹配的话，交换区分配例程将从专用交换区分配内存。如果你将 PoolTag 的值设置成 0x0000002a 或者是通配符（*），那么只要有足够的虚拟和物理内存，所有驱动程序分配的内存都将在专用交换区中。（如果没有足够的空闲页面——存在约束，但每次分配都使用两个页面，那么驱动程序将转换为从常规交换区分配）。

Pool Tracking 在设备驱动程序分配内存时，可以在它们的分配请求中指定一个可选的四字符标签。当禁止交换区跟踪时，Windows 2000 将忽略该标签。然而，如果允许交换区跟踪，交换区分配例程将把该标签与驱动程序分配的内存关联起来。使用 Poolmon（Windows 2000 Support Tool 的一部分），开发人员可以查看 Windows 2000 赋给每个标签的内存数量。监视内存的使用情况让开发人员能够检测内存泄露。内存泄露是指驱动程序释放不再需要的内存时发生的错误。驱动程序检验器还在 Driver Verifier Manager 的 Pool Tracking 标签中显示一般的交换区统计数据。你还可以使用内核调试器命令！verifier。该命令显示的信息比驱动程序检验器多，对驱动程序编写人员非常有用。

如果允许交换区跟踪，内存管理器将在驱动程序卸载的时刻检查该驱动程序是否释放了它所分配的所有内存。如果没有，它将使系统崩溃，表明该驱动程序有缺陷。

Force IRQL Checking 一个最常见的设备驱动程序缺陷发生在驱动程序访问可调页的数据或代码的时候，此时执行该驱动程序的处理器处于提高的 IRQL 级别。就象第 3 章中解释的那样，当 IRQL 是 DPC/调度级别或更高时，内存管理器不能处理页错误。当处理器在高级别的 IRQL 上执行时，系统通常不会检查访问可调页数据的设备驱动程序实例，因为此时被访问的可调页数据恰好常驻在物理内存。然而，其他时刻数据可能被页面调出，结果导致系统崩溃，终止代码为 IRQL-NOT-LESS-OR-EQUAL（也就是说，该 IRQL 级别高于在这种情况下访问可调页内存操作所需的级别）。

尽管测试设备驱动程序的这种错误通常很困难，但是驱动程序检验器简化了该工作。如果你选择了 Force IRQL Checking 选项，不管检验的驱动程序在什么时候提高 IRQL，驱动程序检验器都将迫使所有内核模式的可调页的数据和代码调出系统工作集。执行该项工作的内部函数是 MmTrimAllSystemPagedbleMemory。在允许该设置时，不管被检验的驱动程序在什么时候在 IRQL

被提高的情况下访问可调页的内存，系统都将立即检测到违例，结果系统崩溃从而确认该驱动程序有故障。

Low Resources Simulation 允许 Low Resources Simulation 选项将导致驱动程序检验器随机地使检验的设备驱动程序执行的内存分配失败。以往，开发人员写的许多设备驱动程序都假定内核内存总是可以获得的，即使内存耗光，设备驱动程序也不用为此担心，因为系统将莫名其妙的崩溃。然而，由于可能暂时出现内存不足的情况，因此设备驱动程序适当地处理分配失败错误并指出内核内存耗尽是很重要的。

在系统启动之后的 7 秒钟内——该时间足以通过关键的初始化时期，低内存条件可能阻碍设备驱动程序的加载——驱动程序检验器将开始任意地使正在检验的设备驱动程序分配调用失败。如果驱动程序没有正确地处理分配失败错误，这将出现类似系统崩溃的情况。

驱动程序检验器对于程序编写人员可以获得的检验和调试工具来说是很有价值的。许多第一次随同驱动程序检验器一起运行的设备驱动程序都有驱动程序检验器能够暴露的缺陷。这样，驱动程序检验器就使得所有运行在 Windows 2000 中的内核模式代码的质量在整体上得到提高。

7.4 地址空间布局

默认情况下，32 位 Windows 2000 上每个用户进程都可以拥有最大到 2GB 的专用地址空间，操作系统占用剩下的 2GB 空间。Windows 2000 Advanced Server 和 Windows 2000 Datacenter Server 支持启动中允许 3GB 用户空间的选项。这两种可能的地址空间布局在图 7-11 中表示。

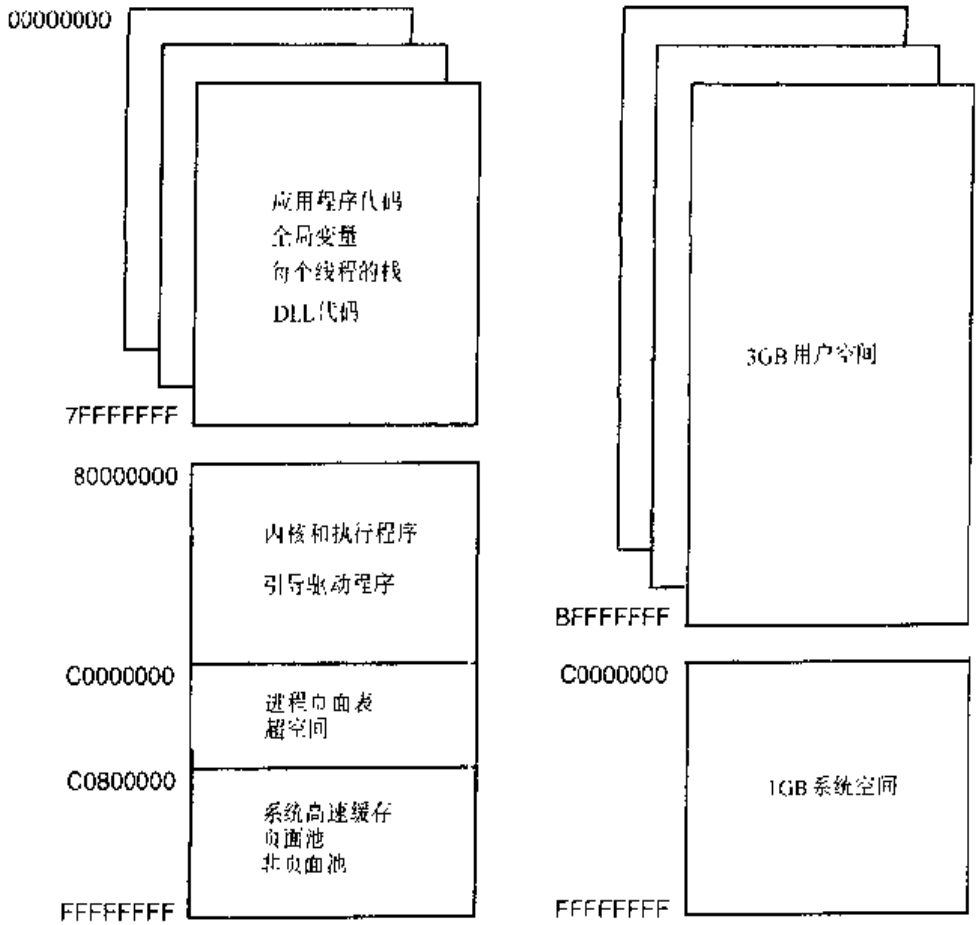


图 7-11 x86 虚拟内存空间布局

3GB 地址空间选项（通过 Boot.ini 中的/3GB 标志设置）给进程一个 3GB 的地址空间（给系统空间留下 1GB）。这个特性是作为临时解决方案而添加的，用它来满足一些应用程序的需要。例如，数据库服务程序要在内存中保存比使用 2GB 地址空间更多的数据。本章曾经表述的 AWE 函数提供了一个更好的解决方案，可以访问超过 2GB（或 3GB）限制的进程空间的数据。

对于一个要访问整个 3GB 地址空间的进程来说，映像文件在映像头中必需设置 IMAGE-FILE-LARGE-ADDRESS- AWARE 标志。否则 Windows 2000 会保留这 3GB 的地址空间，使应用程序不会查看大于 0x7FFFFFFF 的虚拟内存。你在创建可执行程序时通过指定新的链接器标志/LARGEADDRESS-SAWARE 来指定这个标志。在一个带有 2GB 用户地址空间的系统上运行应用程序时，这个标志不起任何作用。如果你启动带/3GB 开关的 Windows 2000 Professional 或 Windows 2000 Server，系统空间减少到 1GB，但是用户空间仍然限制到 2GB，即使设置了运行映像的 large-address-space-aware 标志。

Consumer Windows 的虚拟地址空间

Windows 95、Windows 98 和 Windows Millenium Edition 的虚拟地址空间的组织与 Windows 2000 不同。它还提供 4GB 的虚拟 32 位地址空间，为每个进程分配 2GB 的专用空间；但是它将剩下的 2GB 空间分割成系统空间（1GB）和作为所有共享内存区域的单一共享用户空间（1GB）。如图 7-12 所示：

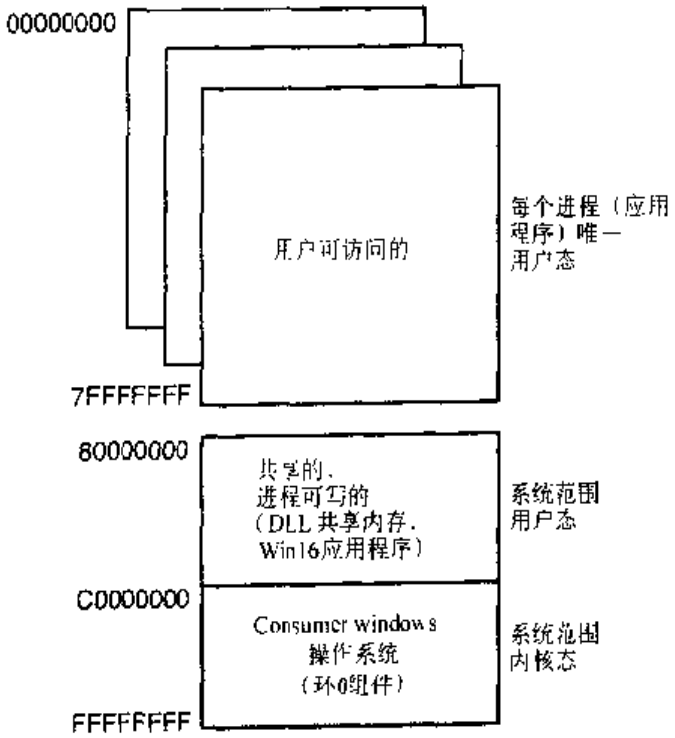


图 7-12

1-GB 的共享区域是用户模式可写的，因此任何 Win32 进程都能够写共享内存或映射文件，即使它们没有区域对象（用 Win32 术语而言是文件映射对象）。另外，Windows 2000 将共享的内存区域放在专用的进程地址空间中，这样避免了安全漏洞。另外，因为所有 MS－DOS 和 Win16 应用程序都在该同一共享的 1GB 区域，Win32 进程可能破坏在 Consumer Windows 中的 Win16 和 MS－DOS 应用程序的地址空间。

7.4.1 用户地址空间布局

2GB 的 Windows 2000 用户地址空间详细布局列在表 7-6 中。

表 7-6 Windows 2000 用户地址空间详细布局

范 围	大 小	功 能
0x0 ~ 0xFFFF	64KB	不可访问区域，用来帮助程序员避免不正确的指针引用；访问该范围内的地址将导致访问违例（注意该范围的地址可能被使用——这只是一种协助查找缺陷的约定。）
0x10000 ~ 0x7FFFFFFF	2GB 减去至少 192KB	进程专用地址空间
0x7FFDE000 ~ 0x7FFDEFFF	4kB	第一个线程的线程环境块（TEB）（参见第 6 章。）其他 TEB 在该页之前的页中创建（起始地址为 0x7FFDD000，并向后延伸）
0x7FFDF000 ~ 0x7FFDFFFF	4KB	进程环境块（PEB）（参见第 6 章。）
0x7FFE0000 ~ 0x7FFE0FFF	4KB	共享用户数据页，该只读的页面被映射到系统空间中包含系统时间、时钟的滴答记数以及版本号等信息的页面。该页的存在使得可以直接从用户模式读取这些数据而不需要转换为内核模式
0x7FFF1000 ~ 0x7FFEFFFF	60KB	不可访问的区域（紧随共享用户数据页的 64KB 区域之后的剩余区域。）
0x7FFF0000 ~ 0x7FFFFFFF	64KB	不可访问区域，用来防止线程传递跨越用户/系统空间边界的缓冲区。MmUserProbeAddress 包含该页的起始地址

表 7-7 中显示的系统变量定义了用户地址空间的范围。

表 7-7 Windows 2000 用户地址空间系统变量

系统变量	说明	x86 2GB 用户空间	x86 3GB 用户空间
MmHighestUserAddress	最高的用户地址（由于 TEB 和 PEB 的缘故，该可用的最高地址实际上要小一些。）	0x7FFEFFFF	0xBFFEFFFF
MmUserProbeAddress	最高的用户地址 + 1（用于试探用户缓冲区的可访问能力）	0x7FFF0000	0xBFFF0000

列举在表 7-8 中的性能计数器提供有关整个系统使用虚拟内存的情况。

表 7-8 Windows 2000 虚拟内存使用的性能计数器

性能计数器	系统变量	说 明
Memory: Committed Bytes	MmTotalCommittedPages	提交的专用地址空间的数量（其中可能有些在物理内存中，而另外一些在调页文件中）
Memory: Commit Limit	MmTotalCommitLimit	可以被提交而又不会增加调页文件大小（页面文件是可扩展的）的内存数量（以字节计数）
Memory: % Committed Bytes In use	MmTotalCommittedPages/ MmTotalCommitLimit	提交的字节与提交极限的比率

你可以通过表 7-9 中的进程性能计数器获取单个进程地址空间的利用情况。

表 7-9 单个进程使用的 Windows 2000 地址空间的性能计数器

性能计数器	说 明
Process: Virtual Bytes	整个进程地址空间的大小（包括共享和专用页面）
Process: Private Bytes	提交的专用（非共享）地址空间的大小（与 Process: Page File Bytes 相同）
Process: Page File Bytes	提交的专用（非共享）地址空间的大小（与 Process: Page Private Bytes 相同）
Process: Page File Bytes Peak	Process: Page File Bytes 的峰值

有一个 Performance 工具不能显示的称为“Process Address Space”的性能对象，在系统中，有 32 个计数器用于识别所选进程对地址空间的使用情况。对于四类进程地址空间（映像、映射、保留和未分配）中的每一种地址空间，存在 8 个单独的计数器（不能访问、只读、读/写、写复制、可执行、执行只读、执行读/写、执行写复制）。另外，还有一些计数器用于计算保留和空闲的全部进程地址空间。关于用户地址空间布局的详细信息，可以查询 Image 性能对象来报告每个映像（例如 DLL）的内存使用情况。

实验：观察进程内存的使用

试着使用性能工具检查在表 7-8 和 7-9 中列出的不同进程内存的性能计数器。同样你可以使用一些其他的实用程序来检查进程的物理内存和虚拟内存的使用情况。

例如，启动 Task Manger（按 ctrl + Shilft + Esc）并单击 Processes 标签，然后从 View 菜单中选择 Select Column。选择 Memory Usage 和 Virutal Memory Size，然后单击 OK，此时在屏幕上的显示如图 7-13 所示：

请记住，VM Size 栏不是进程虚拟内存的大小，而是进程的专用虚拟内存的大小（与表 7-9 中的 Process: Private Bytes 性能计数器相同）。

同样，正如你在工作集章节中发现的那样，Mem Usage 栏在每个进程的内存使用总和中计入了共享的页面。

Process Viewer 实用程序（在 Windows 2000 资源工具包中的 Pviewer.exe；Platforem SDK 中的 Piewer.exe）能显示每个进程地址空间的详细情况（这个实用程序的源代码在 MSDN 上的 Win32 范例程序中）。单击 Start 按钮，选择 Programs/Windows 2000 Support Tools/ Tools/Process Viewer，选择一个进程并单击 Memory Detail 按钮，这时会看到如图 7-14 的屏幕显示，试着单击 User Address Space For 列表框——你能选择由单独的映像或加载的 DLL 使用的地址空间。

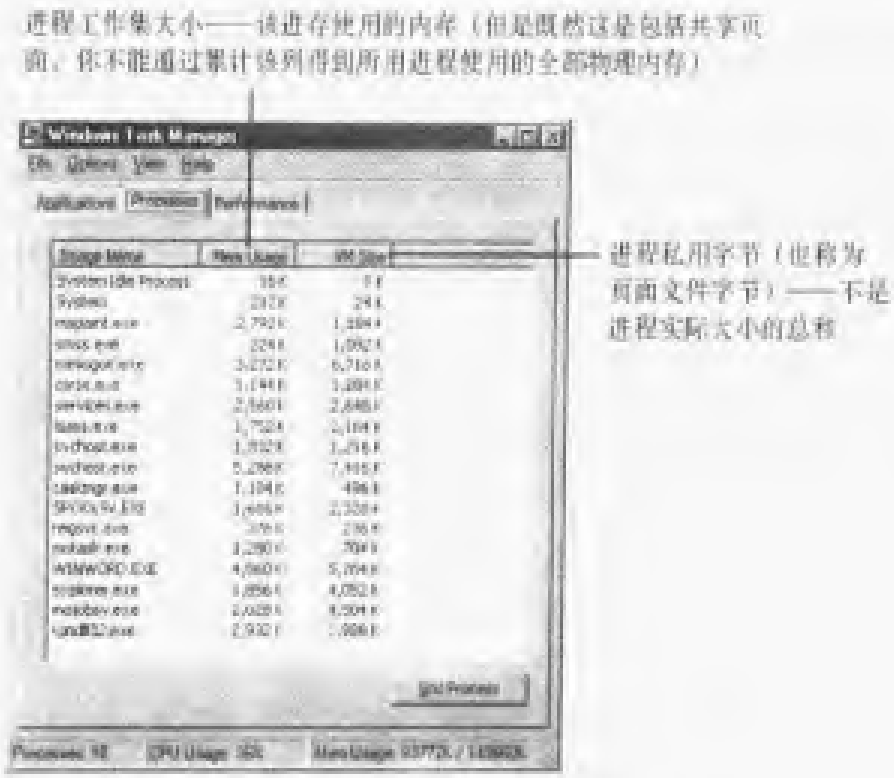


图 7-13 观察进程内存的使用方法一

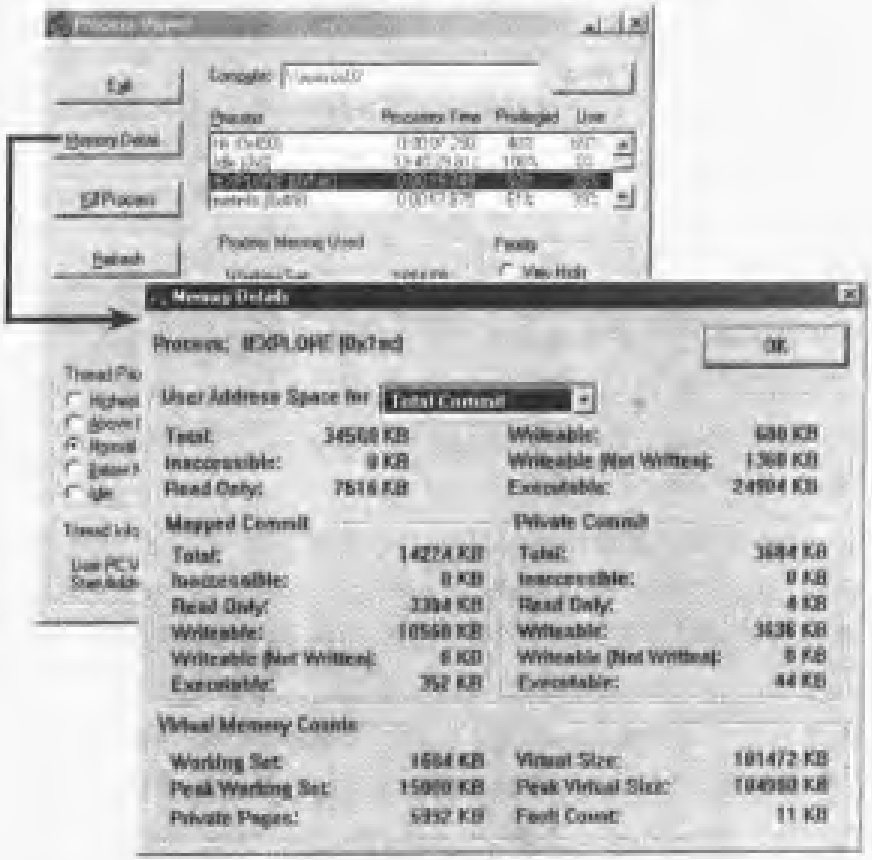


图 7-14 观察进程内存的使用方法二

7.4.2 系统地址空间布局

本节将介绍系统空间的详细布局和它包含的内容。图 7-15 列出了带有 2GB 系统空间的 x86 系统的整体布局（带有 1GB 系统空间的 x86 系统的详细情况将在本章后面讲述）。

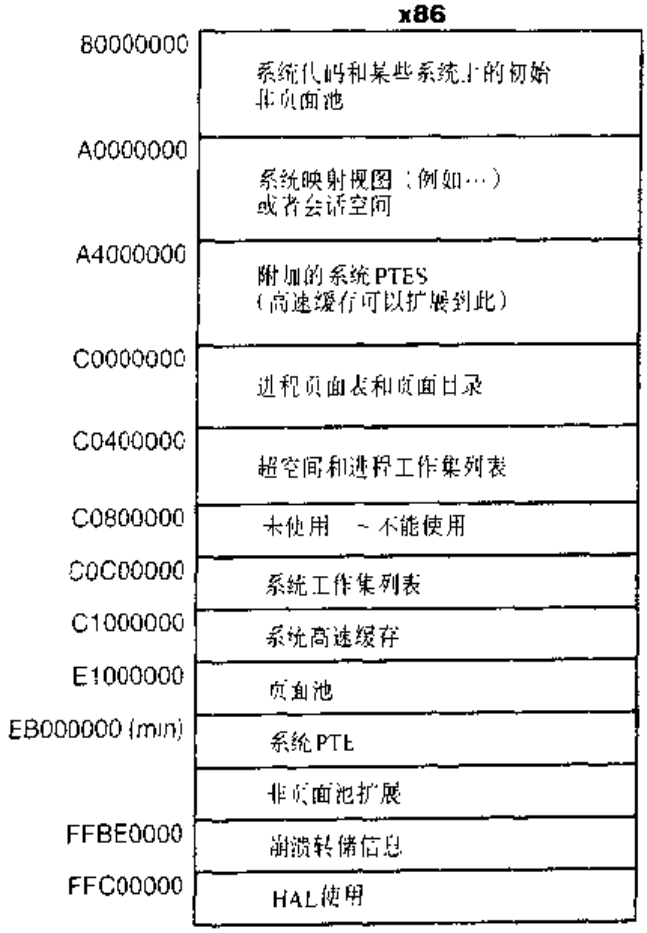


图 7-15 x86 系统空间布局

x86 体系结构在系统空间中有以下组件：

■ 系统代码 它包含了用来引导系统的操作系统映像、HAL 和设备驱动程序。

■ 系统映射视图 它用于映射 Win32k.sys——Win32 子系统的可加载内核模式部分以及它使用的内核模式图形驱动程序（有关 Win32k.sys 的更多信息参见第 2 章）。

■ 会话空间 用于映射用户会话相关的信息。（在 Terminal Services 安装后，Windows 2000 支持多用户会话），会话工作集列表描述了驻留和正使用的区域空间部分。

■ 进程页面表和页面目录 描述虚拟地址映射的结构。

■ 超空间 一个特殊的区域，被用来映射进程工作集列表和为下列操作临时映射物理页面：在自由列表中将页面清零（当清零列表为空并且需要清零页时）、使其他页面表中的页面表项无效（例如当从备用列表中删除一个页面时）和在进程创建时设置新进程的地址空间。

■ 系统工作集列表 描述系统工作集的系统工作集列表数据结构。[⊖]

⊖ 在内部，系统工作集被称为系统高速缓存工作集。该术语具有误导作用，因为它不仅包括系统高速缓存，也包括页式交换区、可调页系统代码和数据以及可调页的驱动程序代码和数据。

■ 系统高速缓存 用于映射在系统高速缓存中打开的文件的页面（有关高速缓存管理器的详细内容，请参阅第 11 章）。

■ 页交换区 可调页的系统内存堆。

■ 系统页面表项（PTE） 系统 PTE 的交换区，用于映射系统页面，例如 I/O 空间、内核堆栈和内存描述符列表。通过观察性能工具中的 Memory: Free System Page Table Entris 计数器的值，可以查看有多少系统 PTE 可以使用。

■ 非页交换区 不可调页的系统内存堆，通常以两部分存在；一部分在系统空间较低端，一部分在系统空间较高端。

■ 崩溃转储信息 保留用来记录有关系统故障状态的信息。

■ HAL 使用 为 HAL 相关的结构保留的系统内存。

本节的剩余部分由两个表组成，这两个表列出了系统空间的详细结构。在表 7-10 中列出了内核变量，这些内核变量包含了不同系统空间区域的起始地址和结束地址。一些系统空间区域是固定的；一些系统空间区是在系统启动时，根据内存大小和系统是在运行 Windows 2000 Professional 还是 Windows 2000 Server 而计算出来的。表 7-11 列出了 x86 系统上系统的空间结构，记住这些表反映了非 PAE 系统。运行 PAE - enalbed 内核映像的系统在系统地址空间布局上有轻微的不同。

表 7-10 描述系统空间区域的系统变量

系统变量	说 明	x86 2GB 系统空间（非 PAE 方式）	x86 1GB 系统空间（非 PAE 方式）
MmSystemRangeStart	系统空间的起始地址	0x80000000	0xC0000000
MmSystemCacheWorkingSetList	系统工作集列表	0xC0C00000	0xC0C00000
MmSystemCacheStart	系统高速缓存的开始	0xC1000000	0xC1000000
MmSystemCacheEnd	系统高速缓存的结束	计算的	计算的
MiSystemCacheStartExtra	系统高速缓存或系统 PTE 扩展的开始	0xA40000000	0
MiSystemCacheEndExtra	系统高速缓存或系统 PTE 扩展的结束	0xC0000000	0
MmPagedPoolStart	页交换区的开始	0xE1000000	0xE1000000
MmPagedPoolEnd	页交换区的结束	计算的（最大 482MB）	计算的（最大 160MB）
MmNonPagedSystemStart	系统 PTE 的开始	计算的（最低值为 0xEB000000）	计算的
MmNonPagedPoolStart	非页交换区的开始	计算的	计算的
MmNonPagedPoolExpansionStart	非页交换区扩展的开始	计算的	计算的
MmNonPagedPoolEnd	非页交换区的结束	0xFFBE0000	0xFFBE0000

表 7-11 x86 系统空间（非 PAE 模式）

范 围	大 小	功 能
0x80000000 ~ 0x9FFFFFFF	512MB	用来引导系统的代码（Ntoskrnl.exe 和 Hal.dll）和非页交换区的初始部分在有 2 - GB 系统空间且 RAM 大于 128MB 的 x86 系统中，前 512MB 是使用 x86 大页面 PDE 映射的
0xA0000000 ~ 0xA2FFFFFF	48MB	如果没有安装 Terminal Services，则是系统映射视图，否则为会话空间（见表 7-12）
0xA3000000 ~ 0xA3FFFFFF	16MB	运行 Terminal Services 系统的映射视图
0xA4000000 ~ 0xBFFFFFFF	448MB	系统额外的 PTE（用于内核堆栈、映射 I/O 空间等）或者是额外的系统高速缓存（适合于允许大的高速缓存的系统）
0xC0000000 ~ 0xC03FFFFF	4MB	进程页面表（页面目录位于 0xC0300000，大小为 4KB）这是映射在系统空间中的进程数据
0xC0400000 ~ 0xC07FFFFF	4MB	工作集列表和超级空间。这是映射在系统空间的进程数据
0xC0800000 ~ 0xC0BFFFFF	4MB	未使用
0xC0C00000 ~ 0xC0FFFFFF	4MB	系统工作集列表
0xC1000000 ~ 0xE0FFFFFF	512MB（最大）	系统高速缓存（在引导时计算大小）
0xE1000000 ~ 0xEAFFFFFF[1]	160MB（最大）	页交换区（在引导时计算大小）
0xEB000000 ~ 0xFFBDFFFF	331，875MB（339，840KB）	系统的 PTE 和非页交换区（在引导时计算大小）。如果注册表 PagedPoolSize 的键值为 - 1，那么系统 PTE 从 0xEB000000 移到 0xA4000000，这允许页交换区使用该地址空间
0xFFBE0000 ~ 0xFFFFFFFF	4.125MB（4224KB）	崩溃转储结构和专用的 HAL 数据结构

注：① 因为页交换区受包含非页交换区和系统 PTE 的区域的起始地址的限制，所以它只能在这些地址不能使用的情况下超出范围 0xE8000000。

会话空间

会话是由进程和其他表示单个用户的工作站登录会话的系统对象（如窗口站、桌面和窗口）组成的。每个会话都有一个会话相关的页交换区，Win32 子系统的内核模式部分使用该交换区分配会话专用的 GUI 数据结构。此外，每个会话都有自己的 Win32 子系统进程（Csrss.exe）和登录进程（Winlogon.exe）的副本。会话管理器进程（Smss.exe）负责创建新的会话，其中包括加载会话专用的 Win32k.sys 副本、创建该会话专用的对象管理器名字空间以及 Csrss 和 Winlogon 进程的实例。

为了虚拟化会话，所有会话范围的数据结构都被映射到系统空间中一个称为会话空间

(session space) 的区域。该地址的起始地址为 0xA0000000，并延伸到 0xA2FFFFFF。创建一个进程时，该地址范围将被映射到该进程所属会话的适当页面中。表 7-12 列举了安装了 Terminal Services 系统的会话空间的布局。

表 7-12 会话空间布局

范 围	大 小	功 能
0xA0000000 ~ 0xA07FFFFF	8MB	Win32k.sys 和基于 Windows NT 4 的打印驱动程序
0xA0800000 ~ 0xA0BFFFFF	4MB	MM-SESSION-SPACE 结构和会话工作集列表
0xA0C00000 ~ 0xA1FFFFFF	20MB	会话的映射视图
0xA2000000 ~ 0xA2FFFFFF	16MB	会话的页交换区

7.5 地址转换

前面你已经看到了 Windows 2000 是怎样构造 32 位虚拟地址空间的，让我们再来看 Windows 2000 怎样把这些地址空间映射到真实的物理页面。本节将描述在这种转换不能解析一个物理内存地址（页面调度）时会发生什么，并将解释 Windows 2000 如何通过工作集和页面帧号数据库来管理物理内存。

用户应用程序引用 32 位虚拟地址，CPU 利用内存管理器创建和维护的数据结构把虚拟地址转换为物理地址。例如，图 7-16 显示了如何将三个连续的虚拟页面映射到三个不连续的物

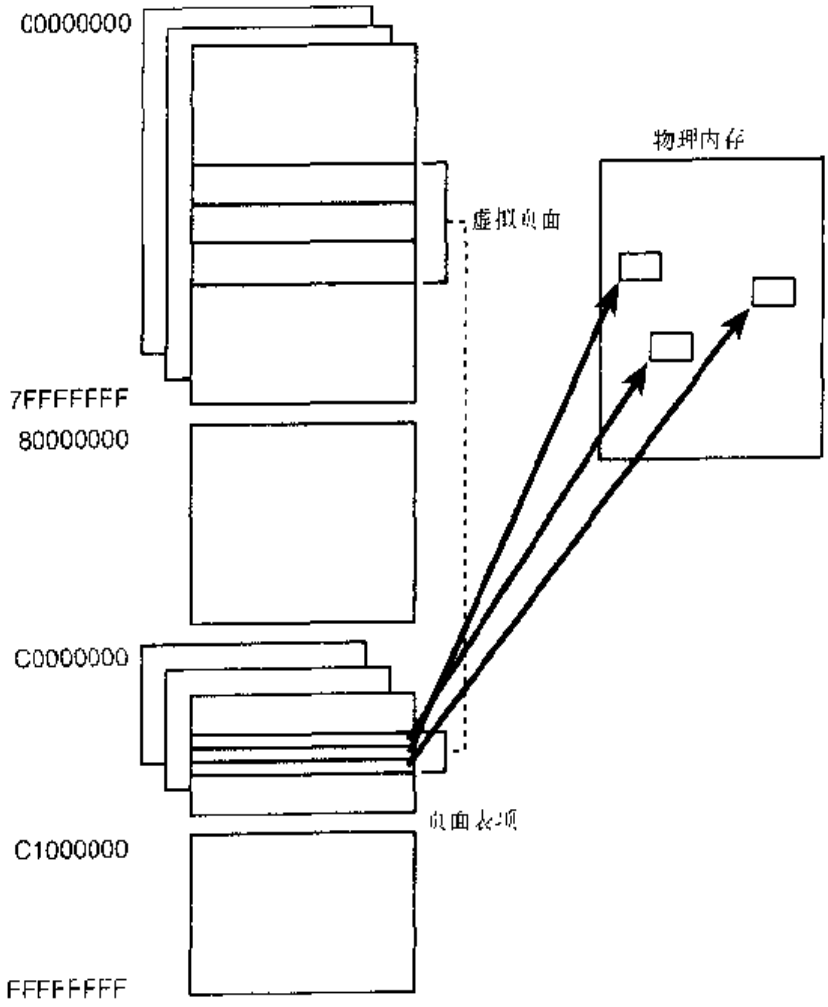


图 7-16 把虚拟地址映射为物理地址

理页面上。

图 7-16 中连接虚拟页面和 PTE 的虚线表示虚拟页面和物理内存之间的间接关系。虚拟地址不直接映射到物理地址。然而，正如在本节将要发现的那样，每个虚拟地址都和一个称为页面表项（page table entry，PTE）的系统空间的结构相关，PTE 包括映射到虚拟地址的物理地址。

注意 通过把物理内存地址映射到虚拟地址，可以使内核模式代码（例如设备驱动进程）引用物理内存地址。详细信息请参阅在 DDK 文档中描述的内存描述符表（MDL）支持例程。

在本节的剩余部分将详细解释 Windows 2000 是怎样实现这种映射的。

7.5.1 转换虚拟地址

默认情况，Windows 2000 使用二级页面表结构来转换物理地址和虚拟地址（运行 PAE 内核的系统使用三级页面表——本节假设系统为非 PAE 系统）。32 位的虚拟地址被解释为三个单独的组件：页面目录索引、页面表索引和字节索引 — 它们被当作描述页面映射的结构的索引，如图 7-17 所示。页大小和 PTE 宽度表明页面目录和页面表索引字段的宽度。例如，在 x86 系统上，因为页面为 4096 字节（$2^{12} = 4096$），字节索引则为 12 位。

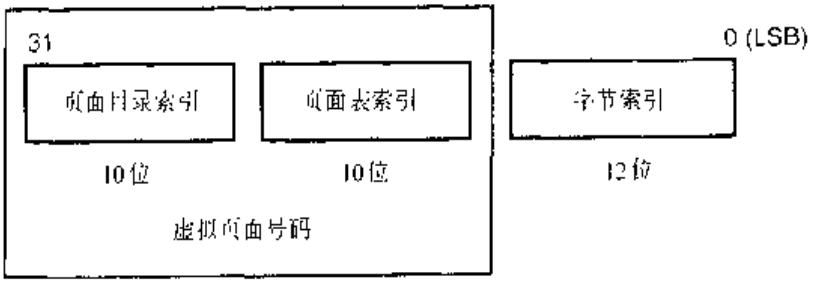


图 7-17 x86 系统上 32 位虚拟地址的组件

页面目录索引被用来查找页面表，在页面表中虚拟地址的 PTE 已经被查找。页面表索引被用来查找 PTE，正如前面提到的，PTE 包括了被映射到虚拟地址的物理地址。字节索引在该物理页面内寻找正确的地址。图 7-18 显示了这三个值之间的关系和如何把虚拟地址映射到物理地址。

以下是转换一个虚拟地址的基本步骤：

1）内存管理硬件为当前进程查找页面目录。在每个进程的环境切换发生时，硬件被告知新进程页面目录的地址，一般是通过操作系统设置专用的 CPU 寄存器来实现。

2）页面目录索引被用来作为在页面目录中查找页面目录项（PDE）的索引，页面目录项描述了映射虚拟地址时需要的页面表位置。PDE 包括页面表（如果它是驻留的——页面表不能调度出去）的页面帧号（PFN）。

3）页面表索引被用来作为在页面表中查找 PTE 的索引，PTE 描述了正在被讨论的虚拟页面的位置。

4）PTE 被用来查找页面。如果页面有效，则这个页面将包含物理内存中包含虚拟页面的页面的 PFN。如果页面无效，则内存管理错误处理程序将查找页面并试图使页面有效（参见页面错误处理部分）。如果不能使页面有效（例如由于保护错误），错误处理程序产生一个访问违

例或错误检查。

5）当 PTE 被指向一个有效的页面时，字节索引被用来查找在物理页面内所需数据的地址。

既然已经有了一个总体的印象，接下来看一看页面目录、页面表和 PTE 的详细结构。

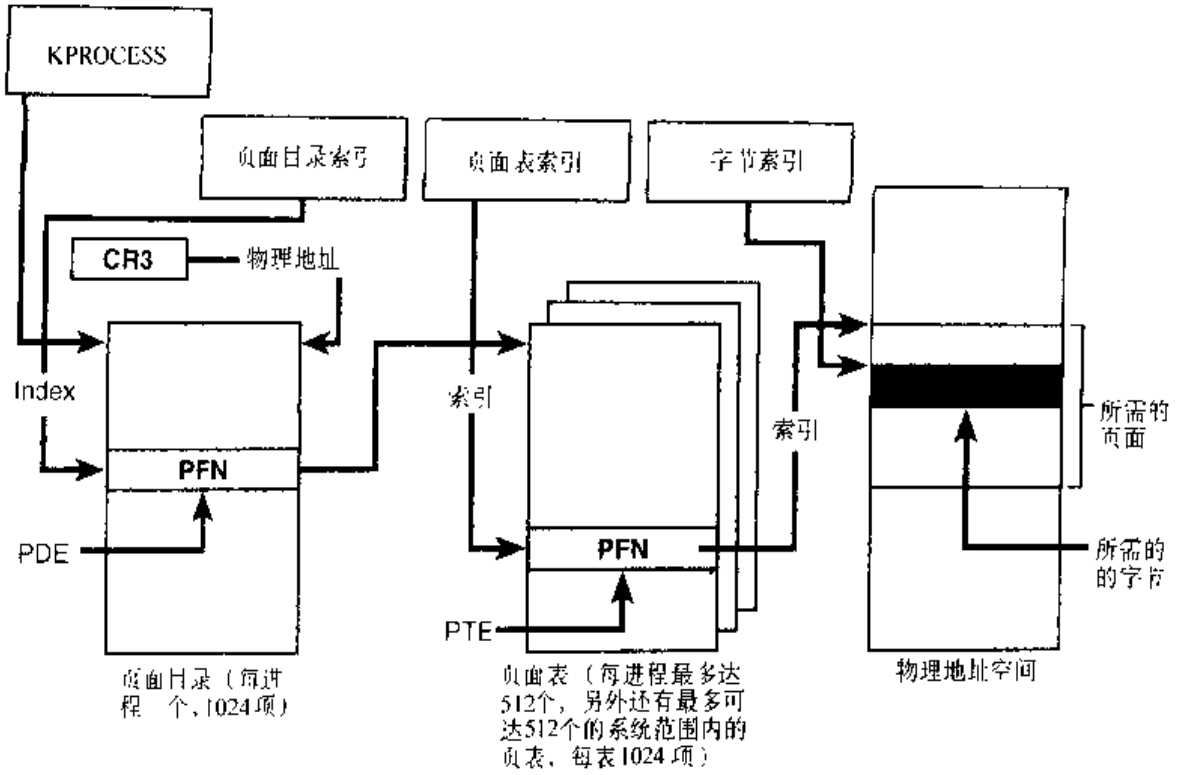


图 7-18　映射一个有效的虚拟地址（x86 相关的）

7.5.2　页面目录

每个进程都有一个单独的“页面目录”（page directory），它是内存管理器创建的页面，用于映射该进程的所有页面表的位置。进程页面目录的物理地址存储在内核进程（KPROCESS）块中，同样，它也被虚拟地映射到 x86 系统的 0xC0300000 地址上（运行 PAE 内核映像的系统在 0xC0600000）。所有运行在内核模式的代码引用虚拟地址，而不是物理地址。（关于 KPROCESS 和其他进程数据结构的详细情况请参阅第 6 章）。

CPU 之所以知道页面目录页面的位置是因为在 CPU 中有一个由操作系统加载的特殊的寄存器（x86 系统上的 CR3）包含页面目录物理地址。每次当环境切换到另外一个进程中的线程（不是当前正在执行的线程）时，这个寄存器通过在内核中的环境切换例程从 PROCESS 加载。因为在同一进程的全部线程可以共享同一个进程的地址空间，所以在同一进程的线程切换不会导致重新加载页面目录的物理地址。

页面目录由页面目录项（page directory entry（PDE））构成，目前每一个 PDE 页面目录是 4 个字节（在运行 PAE 内核映像的系统上为 8 字节），并且页面目录描述了进程的所有可能的页面表的状态和位置（如同在本章稍后将要描述的，页面表是根据需要创建的，所以，对于大多数进程来说，页面目录只能指向页面表的一个小集合）。在此对 PDE 的格式不再重述，因为它

的格式和硬件 PTE 几乎是相同的。

在 x86 系统上，描述全部 4GB 的虚拟地址空间需要 1024 个页面表（PAE 系统上为 2048）。映射这些页面表的进程页面目录包括 1024 个 PDE。这样一来，页面目录索引就要达到 10 位宽（$2^{10} = 1024$）。

实验：检查页面目录和 PDE

可以通过检查内核调试程序！process 的输出的 DirBase 字段来查看当前正在运行的进程的页面目录的物理地址，如图 7-19 所示。

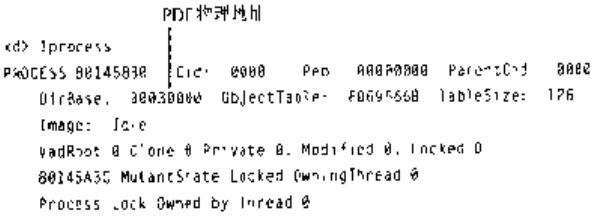


图 7-19 检查页面目录的物理地址

可以通过检查内核调试程序为一个特定的虚拟地址输出的 PTE 来了解页面目录的虚拟地址。输出如图 7-20 所示。

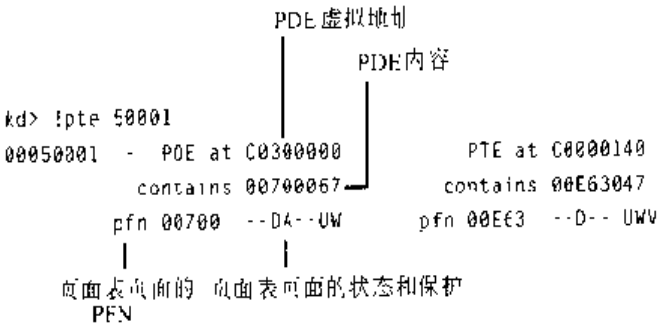


图 7-20 检查页面目录的虚拟地址

内核调试器输出的 PTE 部分见“页面表项”一节中定义。

7.5.3 进程和系统页面表

在页面内用字节偏移量来引用一个字节之前，CPU 首先需要能够查找包含所要求数据字节的页面。要做到这一点，操作系统就要构造包含映射信息的另一个内存页面，此页面包含查找含有所需数据的页面所需要的映射信息。这个含有映射信息的页面叫作“页面表”（page table）。因为 Windows 2000 为每一个进程都提供一个专用的地址空间，每一进程都有其自身的进程页面表集以映射其专用的地址空间，因此每一个进程的映射情况都是不相同的。

然而，页面表描述的全部进程都可以共享系统空间。当创建一个进程时，系统空间页面目录项被初始化为指向现有的系统页面表。但是如图 7-21 所示，并不是所有的进程都有相同的

系统空间视图。例如，如果页交换区扩展请求分配一个新的系统页面表，内存管理器将不能返回并更新所有的进程页面目录以指向新的系统页面表。与此相反，当进程引用了新的虚拟地址时，内存管理器将更新进程页面目录。

这样一来，当引用一个实际上是驻留在物理内存中的页交换区时，进程将获得一个页面错误，因为它的进程页面目录仍没有指向描述交换区新区域的新系统页面表。即使非页交换区也能扩展，在访问非页交换区时页面错误也不会发生，这是因为 Windows 2000 在系统初始化时为了能够描述它的最大值而建立了足够多的系统页面表。

系统 PTE 并不是一个无限的资源。Windows 2000 根据内存大小可以计算出要分配多少系统 PTE。你可以通过检查在性能工具中的 Memory：Free System Page Table Entry 计数器来了解有多少系统 PTE 可用。你也可以通过把 HKLM \ SYSTEM \ CurrentControl Set \ Control \ Session Manager \ Memory Manager \ System Pages 注册表键值设置为你所希望的 PTE 数量来重载在启动时计算的值。然而，在 x86 系统上 Windows 2000 将要分配的 PTE 的最大值是 128 000。

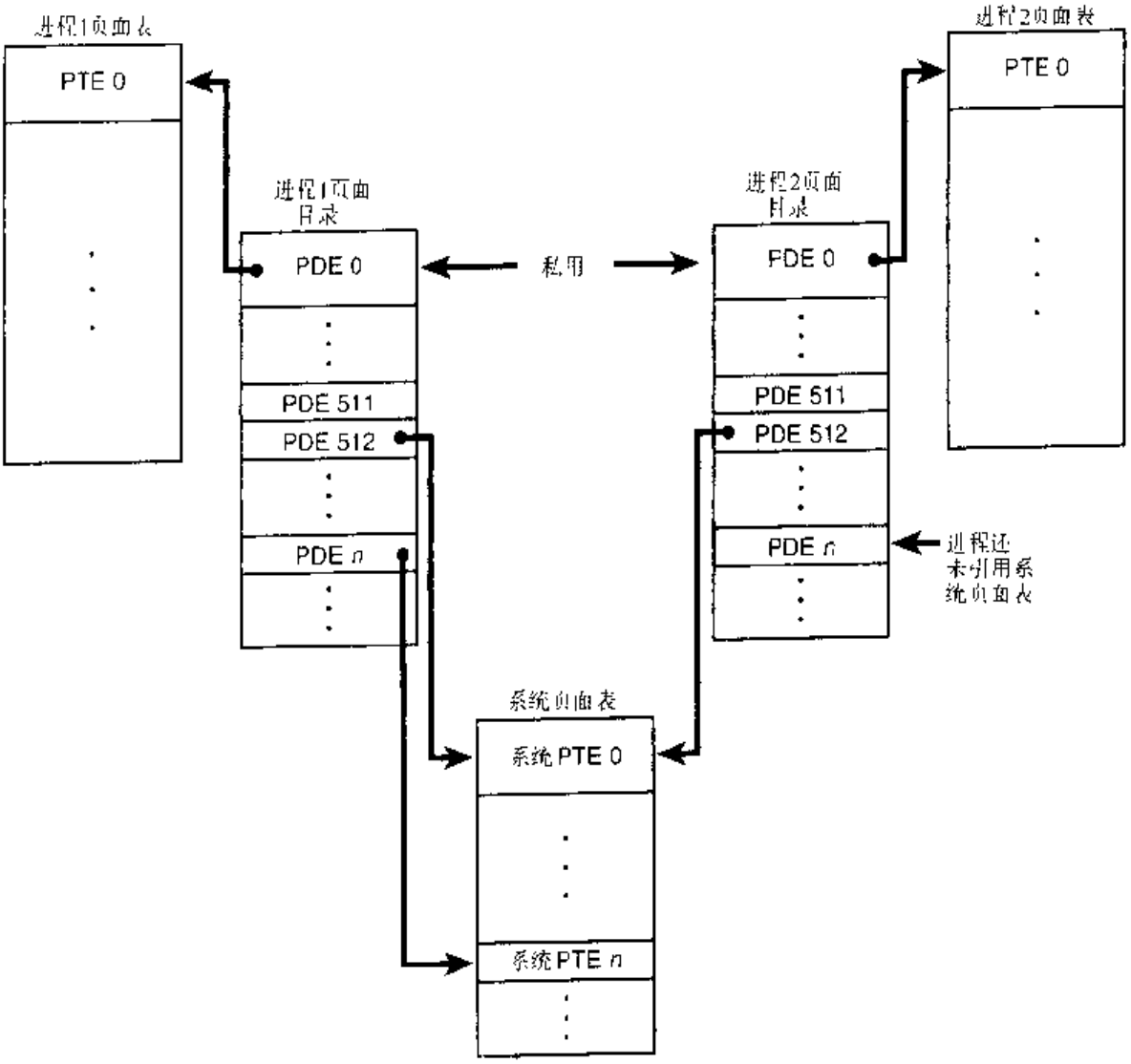


图 7-21 系统和进程专用页面表

7.5.4 页面表项

正如前面提到的，页面表是由一个页面表项（PTE）数组构成的。可以在内核调试程序中用！pte命令来检查PTE。（请参阅实验“变换地址”）。有效的PTE中（这是我们将要讨论的情况；而无效的PTE将在以后的部分中涉及）有两个主要的字段：包含数据的物理页面的页面帧号（PFN）或在内存中的某个页面的物理地址的页面帧号以及一些描述页面状态和保护的标志，如图7-22所示。

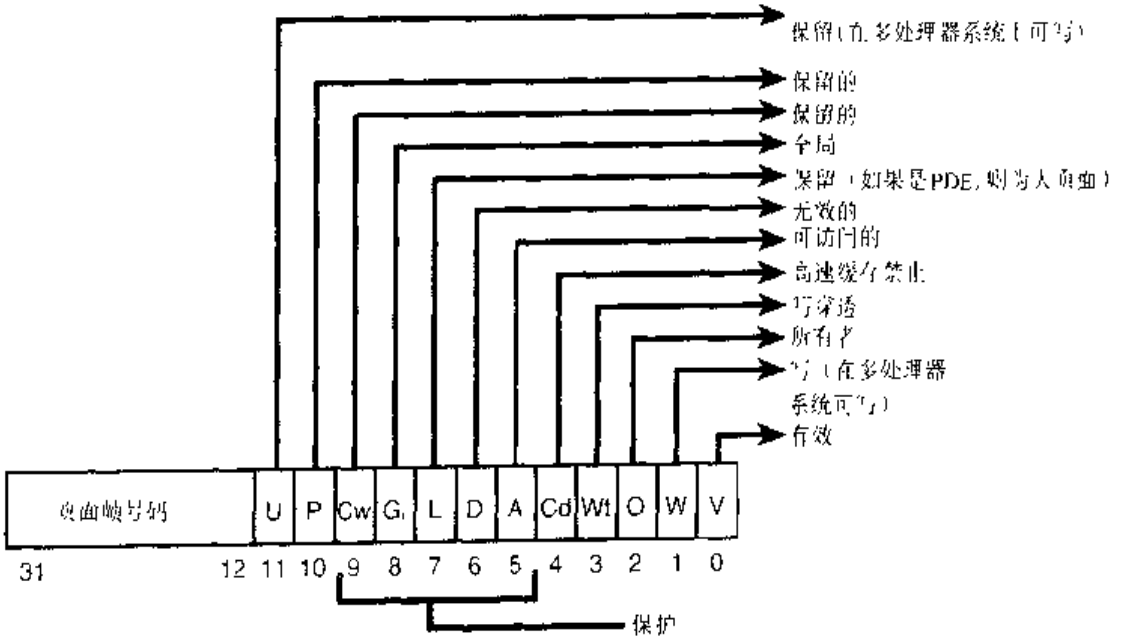


图7-22 有效的x86硬件PTE

正如你将要看到的一样，图7-22中被标记为Reserved的位只在PTE无效时使用（这些位由软件解释）。表7-13中主要描述了有效PTE中由硬件定义的位。

表7-13 PTE状态位和保护位

位 名	含 义
已访问	页面已经被读
禁用高速缓存	禁用页面的高速缓存
脏	页面已经被写入
全局	应用于所有进程的转换（例如，转换缓冲区的刷新不会影响此PTE）
大页面	在128M或更多内存的系统上，表明PDE映射了4MB页面（用于映射Ntoskrnl和HAL，初始的非页交换区等等）
所有者	表明用户模式代码是否可以访问页面或是否页面限制在内核模式下访问
有效	表明是否转换映射到物理内存中的页面
写穿透	禁用此页面的写入高速缓存，使页面立即刷新到磁盘
写入	在单处理器系统中，表明页面是读/写或只读；在多处理器系统中，表明页面是否可写。（写位存储在PTE中的保留位）

在 x86 系统中，硬件 PTE 含有“脏”和“已访问”位。如果 PTE 所代表的物理页面尚未读出或写入，“已访问”位将被清除；如果页面是初次读入或写入的，处理器就会设置该位。只有页面被初次写入，处理器才设置“脏”位。除了这两种位以外，x86 体系结构中还有提供页面保护的“写入”位。当清除此位时，页面处于“只读”状态；置该位后，页面就是可读/写的。如果线程试图在“写入”位清除时写页面，内存管理异常就将发生，此时内存管理器的访问错误处理程序（在下一节描述）就必须决定线程是否可以写这个页面（例如，如果页面真的被标明写时复制）或者是否产生一个访问违例。

多处理器 x86 系统中的硬件 PTE 有一个在软件中实现的附加“写入”位，这是因为当通过处理器刷新 PTE 高速缓存（称为转换后备缓冲区）时避免延迟。此位说明一个页面已经被正在另一个处理器上运行的线程写入了。

在 x86 硬件平台中，PTE 的大小总是 4 个字节（32 位），（在运行允许 PAE 的系统上为 8 字节），所以每一个页面表包含 1024 个 PTE（PAE 系统 512 个）（每 PTE 有 4 个字节，每页有 4096 字节），所以可以映射 1024 个页面（PAE 系统 512 个），一共 4MB（PAE 系统 2MB）的数据页面。

虚拟地址的页面表索引字段指出在页面表映射的数据页面中的哪一个 PTE 映射到了所需的数据页。在 x86 系统中，页面表索引的宽度是 10 位（PAE 是 9 位），允许你最多引用 1024 个 PTE（PAE 512 个）。然而，因为 Windows 2000 提供了 4GB 的专用虚拟地址空间，所以需要多个页面表来映射全部地址空间。要计算映射全部 4GB 的进程的虚拟地址空间所需页面表的数量，可以用 4GB 减去已经被单个页面表映射的虚拟内存的大小。在 x86 系统中，每一个页面表可映射 4MB（PAE 映射 2MB）的数据页面。因此，映射全部 4GB 的地址空间需要 1024 个页面表（4GB/4MB）或 2048 页面表，对于 PAE 是 4GB/2MB。

7.5.5 页面内字节

一旦内存管理器找到了所需要的物理页面，内存管理器就必定会在此页面内查找所需要的数据。这里就是引入字节索引字段的地方。字节索引字段告诉 CPU 你希望引用页面中哪一个字节的数据。在 x86 系统中，字节索引的宽度是 12 位，允许你最多引用 4096 个数据字节（即一个页面的大小）。

7.5.6 转换后备缓冲区

到目前为止，我们已经学习了每次地址转换都要进行的两项查找：一个是查找正确的页面表是否在页面目录中，另一个是查找正确的项是否在页面表中。因为在每次引用虚拟地址时都要做两项附加的内存查找将会导致系统性能下降，所以大多数的 CPU 都会高速缓存地址转换，因此重复访问同一个地址就不需要重新转换。x86 系统中的处理器都能以一种联系内存数组（array of associative memory）的形式提供一种称为转换后备缓冲区（translation look - aside buffer (*TLB*)）的高速缓存。联系内存，例如 TLB，是矢量，它的单元可以被同步地读出并和目标值相比较。在使用 TLB 时，矢量中包含大多数最近已使用页面的从虚拟地址到物理页面的映射和应用于每一个页面的页面保护类型，如图 7-23 所示。TLB 中的每一项都像一个高速缓存项，

它的标志包含了虚拟地址的一部分，并且它的数据部分保存了一个物理页面号码、保护字段、有效位和通常情况下都有一个显示与高速缓存 PTE 相符的页面条件的“脏”位。如果设置了一个 PTE 的“全局”位（用于对全部进程可见的全局系统空间页面），那么在进程环境切换中的 TLB 项就是有效的。

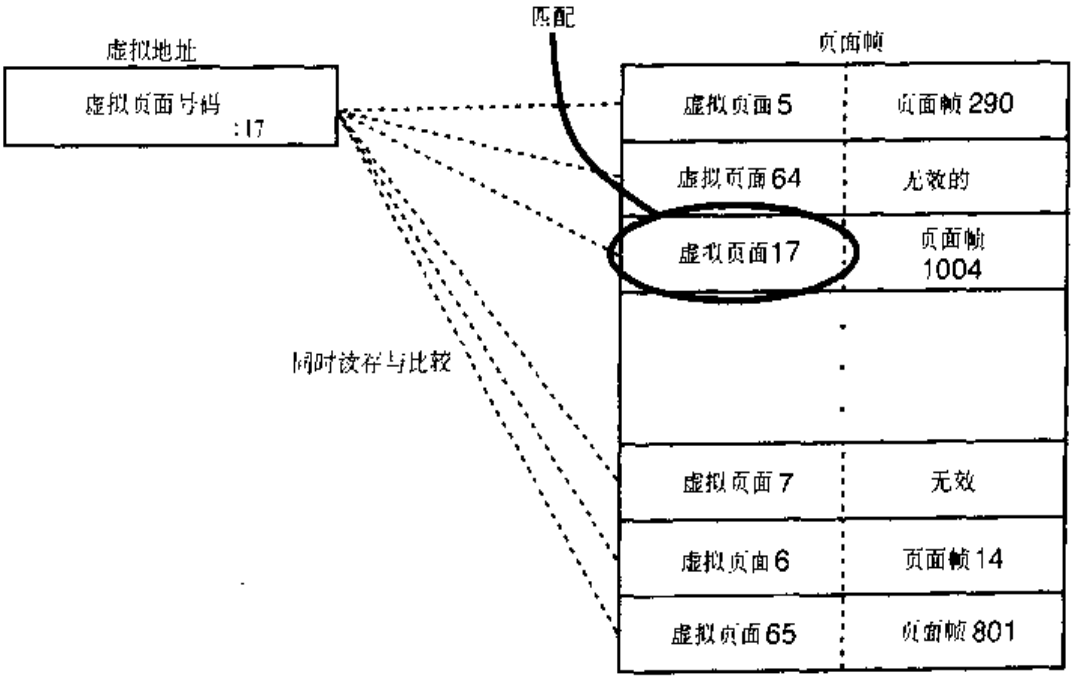


图 7-23 访问转换后备缓冲区

经常使用的虚拟地址很可能有项在 TLB 中，这就提供了非常快的虚拟地址到物理地址的转换和快速内存访问。如果虚拟地址不在 TLB 中，它可能仍然在内存中，但是需要多次内存访问来查找它，这就使得访问时间稍微有些减慢。如果虚拟页面被页面调出内存或者内存管理器改变了 PTE，那么内存管理器就必须明确地使 TLB 项无效。如果某个进程再次访问 TLB，就将发生页面错误，并且内存管理器会把页面调回内存，然后在 TLB 中为该页面重新创建项。

为了最大限度地利用通用代码，无论什么时候，只要有可能，内存管理器就会把所有的 PTE 视为相同的 PTE，而不管它们是由硬件维持的还是由软件维持的。例如，当某个 PTE 由无效变为有效时，内存管理器就会调用一个内核例程。此例程的工作是以硬件相关的体系结构需要的方式将新的 PTE 加载到 TLB 上。在 x86 系统上，代码是 NOP，因为在处理器加载到 TLB 时，不受软件的任何干预。

实验：地址转换

要弄清地址转换是如何工作的，就让我们先看看 x86 非 PAE 系统中转换虚拟地址的实例。这里要使用在内核调试程序中的可用工具来测试页面目录、页面表和 PTE，在此实例中，将使用当前虚拟地址为 0x50001 映射到有效物理地址的进程。在稍后的实例中，将看到如何使用内核调试程序查看无效地址的地址转换。

首先将 0x50001 转换为二进制，然后将它分隔成用于转换一个地址的 3 个字段。在二进制状态下，0x50001 是 101.0000.0000.0000.0001，分隔这些字段之后如图 7-24 所示。

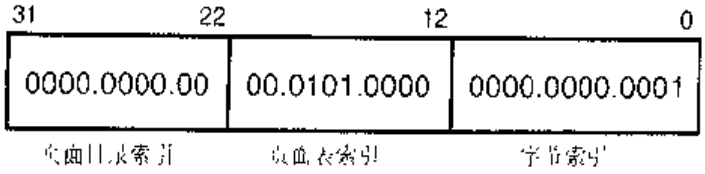


图 7-24 分隔字段

要启动转换进程，当某个线程在此进程中正在运行时，CPU 要求有进程页面目录的物理地址，它存储在 CR3 寄存器中。既可以通过检测 CR3 寄存器本身来显示该地址，也可以使用！process 命令为正在被讨论的进程转储 KPROCESS 块来显示该地址，显示如图 7-25 所示。

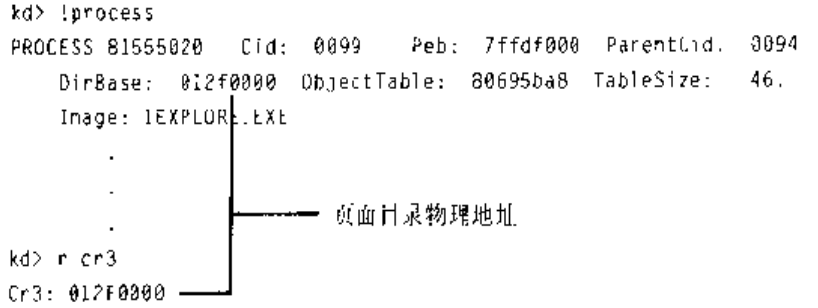


图 7-25 显示进程页面目录的物理地址

在这种情况下，页面目录被存储在物理地址 0x12F0000 中。正如前面的例子所显示的，页面目录索引字段在此例中是 0。因此 PDE 在物理地址 0x12F0000 中，内核调试程序中的！pte 命令显示了描述虚拟地址的 PDE 和 PTE，如下图 7-26 所示。

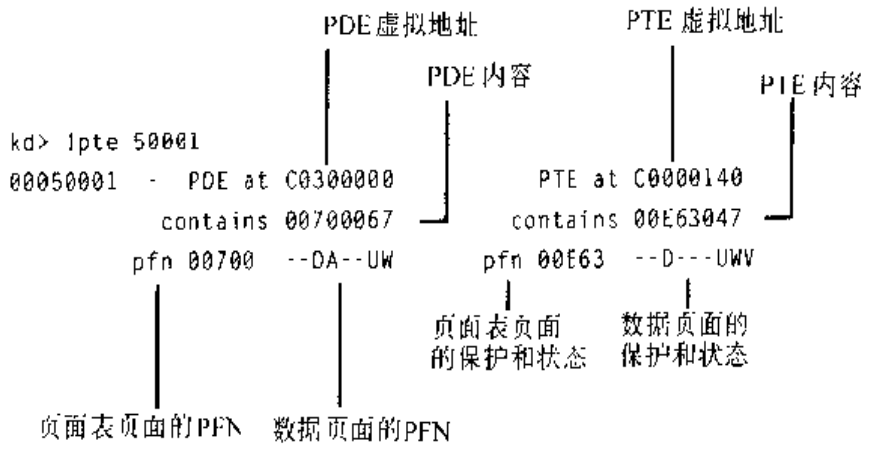


图 7-26 显示描述虚拟地址的 PDE 和 PTE

在第一列中，内核调试程序显示了 PDE，在第二列中显示了 PTE。注意，这里 PDE 地址用虚拟地址而不是物理地址表示。正如前面提到的，在 x86 系统中进程页面目录是在 0xC0300000 虚拟地址上启动的。因为我们是在页面目录中查看第一个 PDE，所以 PDE 地址和页面目录地址是相同的。

PTE 的虚拟地址为 0xC0000140。你可以使用页面表索引（在此例中为 0x50）乘以 PTE 的大小来计算该地址：40 * 0x50 = 0x140，因为内存管理器从 0xC0000000 后开始映射页面表，加上 140 后就可以得到由内核调试程序输出的虚拟地址：0xC0000140。页面表页面在 PFN 0x700

中，而数据页面在 PFN 0xe63 中

PTE 标志显示在 PFN 号码的右边。例如，描述被引用页面的 PTE 有 D - - - UWV 标志。D 在这里代表“脏”（页面已经被修改）、U 代表用户模式页面（与内核式页面相对）、W 代表可写页面（而不是只读页面）、V 表示有效（该 PTE 代表了物理内存中的有效页面）。

7.5.7 物理地址扩展

自从 Pentium Pro 以来的 Intel x86 处理器系列都包括一种称为“物理地址扩展”（Physical Address Extension，PAE）的内存映射模式。利用适当的芯片集，PAE 模式允许访问高达 64GB 的物理内存。当 X86 以 PAE 模式运行时，内存管理单元（MMU）将虚拟地址空间分割成如图 7-27所示的 4 个字段。

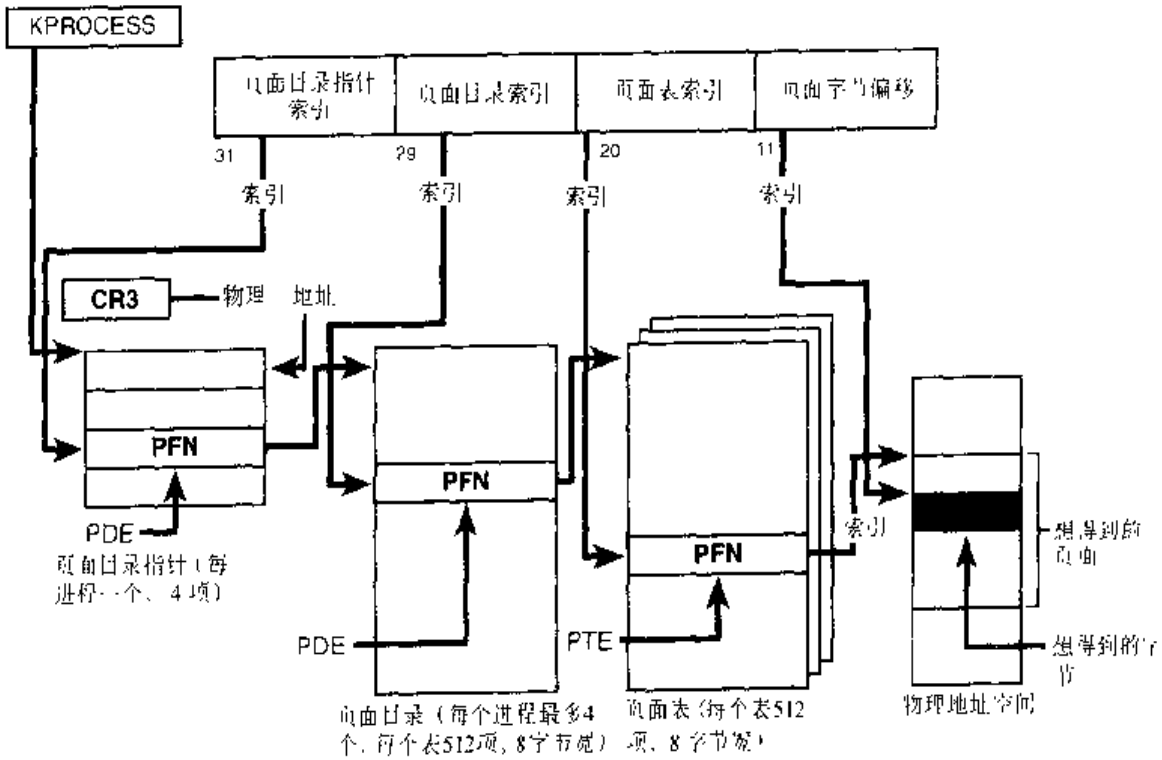


图 7-27 带 PAE 的页面映射

MMU 仍然实现页面目录和页表，但是在它们的上面存在第三级页面目录指针表。PAE 模式能够访问比标准转换模式更多的内存并不是因为额外的转换级别，而是因为 PDE 和 PTE 是 64 位宽度的而不是 32 位。系统内部使用 24 位表示物理地址。这使得 X86 支持的内存高达 2^{24+12}字节，或 64GB。

就象在第 2 章中解释的那样，内核模式映像（Ntoskrnl.exe）有一个称为 Ntkrnlpa.exe 而支持 PAE 的特别版本（多处理器版为 Ntkrpamp.exe）。为了选择该允许 PAE 的内核，你必须在 boot.ini 文件中用/PAE 开关引导。

该内核映像的特别版本安装在所有的 Windows 2000 系统中，甚至是小内存的 Windows 2000

Professional 系统。这样做的目的是为了测试的便利。因为 PAE 内核向设备驱动程序和其他系统代码呈现 64 位的地址，因此即使在小内存系统中用/PAE 引导也允许设备驱动程序开发人员测试它们的驱动程序的大地址部分。另一个相关的 Boot.ini 开关是/NOLOWMEM。该开关放弃 4GB 以下的内存，将设备驱动程序重新分配在该范围之上，这样保证了能够用大于 32 位的物理地址来表示这些驱动程序。

只有 Windows 2000 Advanced Server 和 Windows 2000 Datacenter Server 需要支持 4GB 以上的物理内存（请参见表 2-2）。利用 AWE Win32 函数（描述在"地址窗口扩展"），32 位的用户进程可以在这些系统中分配并控制大量的物理内存。

7.6 页面错误处理

在前面的章节中，你看到了当 PTE 有效时，如何解决地址转换。如果清除了 PTE 就表明所需的页面因某种原因对此进程（当前）是不可访问的。本节将描述无效的 PTE 类型以及如何解决对它的引用。

对无效页面的引用称为页错误（page fault）。内核陷阱处理程序（第 3 章中介绍过）调度这种错误到内存管理器错误处理程序（MmAccessFault）中解决，这个例程运行在引起错误的线程的环境中，并且负责尝试解决这个错误（如果可能）或产生适当的异常。这些错误可以由列在表 7-14 中的各种情况引起。

表 7-14 引起访问错误的原因

错误的原因	结 果
访问一个没有驻留在内存中但在磁盘上的页面文件或映射文件中的页面	分配一个物理页面，并将所需的页面从磁盘上读取到工作集中
访问一个备用或修改列表中的页面	将页面变换到进程或系统工作集中
访问一个没有提交的页面（例如，保留的地址空间或未分配的地址空间）	访问违例
从用户模式访问一个只能在内核模式下被访问的页面	访问违例
写入一个只读页面	访问违例
访问一个请求清零页面	在进程工作集中添加一个零填充的页面
写入保护页	保护页违例（如果引用用户模式堆栈，自动完成堆栈扩展）
写入一个写时复制页面	制作进程专用（或会话专用）页面副本，代替在进程、会话或系统工作集中的原始页面
引用一个在系统空间中有效但不在进程页面目录中的页面（例如在创建进程页面目录后页文换区被扩展）	从主系统目录结构复制页面目录项面放弃异常
在多处理器系统上写入一个有效但尚未写入的页面	在 PTE 中设置"脏"位

在下面的部分中描述了四种基本类型的被访问错误处理程序处理的无效 PTE。在这之后是

对无效 PTE 特殊情况的解释——原型页面表项（PTE）—— 它被用来实现可共享的页面。

7.6.1 无效的 PTE

下面列出了四种无效 PTE 的细节和它们的结构，一些标志与表 7-13 中描述的硬件 PTE 一样。

■ 页面文件 所需的页面驻留在调页文件中。初始化页面调入操作，如图 7-28 所示：

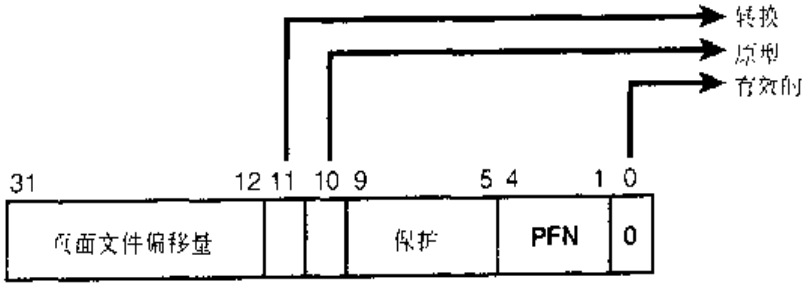


图 7-28 初始化页面调入操作

■ 请求清零 必须用零页面满足所需的页面，页面调度程序将查看零页面列表。如果表是空的，页面调度程序就从备用列表中取出一个页面并且把它置零。此 PTE 格式与上面所示的页面文件的 PTE 相同，但页面文件数和偏移量为零。

■ 转换 所需的页面在内存中的备用、更改或更改不写入列表上，从列表中删除此页并添加到工作集中，显示如下：

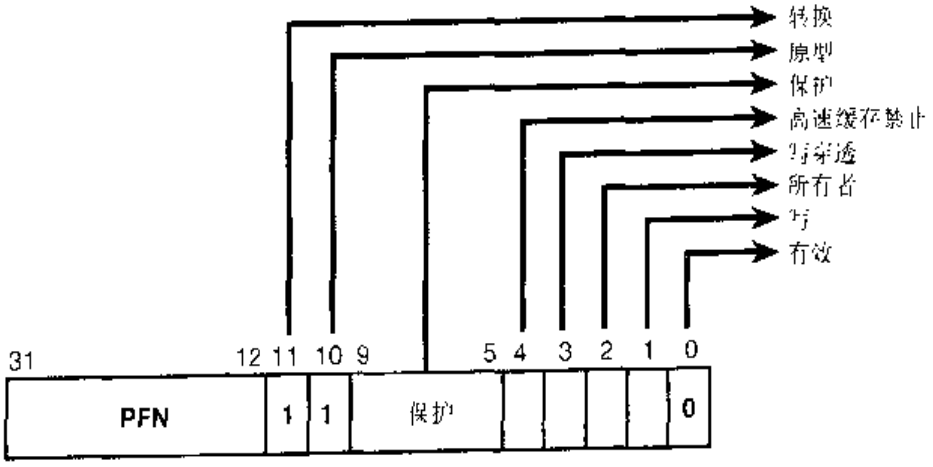


图 7-29 从列表中删除此页

■ 未知 PTE 为零或者页面表不存在。出现这两种情况，就意味着应该检查虚拟地址描述符（VAD）来决定是否此虚拟地址已经被提交。如果是，建立页面表来代表新提交的地址空间。（请参阅 7.7 节“虚拟地址描述符”对 VAD 的讨论。）

7.6.2 原型 PTE

如果一个页面可以在两个进程之间共享，内存管理器将依靠被称为“原型页面表项”（prototype page table entry（prototype PTE））的软件结构来映射这些潜在共享的页面。当首次创建区域对象时将创建一组原型 PTE。这些原型 PTE 是“段”（segment）结构的一部分，将在本章

最后描述。

当进程首次引用一个映射到区域对象视图的页面时（只有当映射视图时才创建的 VAD），内存管理器使用原型 PTE 中的信息填入在进程页面表中用于地址转换的实际 PTE。当共享页面为有效时，进程 PTE 和原型 PTE 都指向包含数据的物理页面。为了跟踪引用有效共享页面的进程 PTE 的数量，递增在“PFN 数据库项”（PFN database entry）中的一个计数器。这样，内存管理器可以决定何时一个共享页面不再被任何页面表引用，这样就可以把它置为无效并移动到转换列表中或写出到磁盘上。

当使一个页面无效时，用一个特殊的 PTE 填充进程页面表中的 PTE，这个特殊的 PTE 指向描述此页面的原型 PTE 项，如图 7-30 所示。

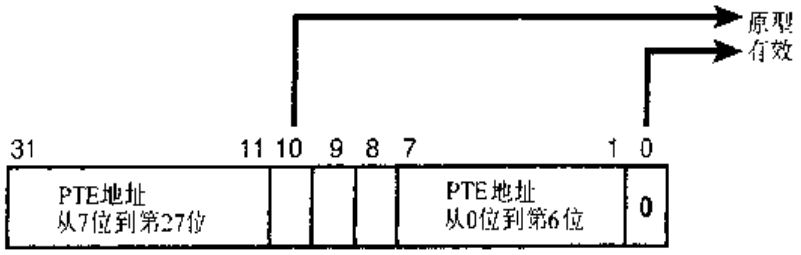


图 7-30 指向原型 PTE 的无效 PTE 的结构

这样，在以后访问此页面时，内存管理器就能够使用在这个 PTE 中编码的信息来查找原型 PTE，这个 PTE 反过来描述被引用的页面。共享页面可以是由原型 PTE 项所描述的六种不同状态中的一种：

- 活动/有效　此页面在物理内存中，做为另一个进程访问它的结果。
- 转换　所需的页面在内存中的备用或修改列表中。
- 修改不写入所需页面在内存并且在修改不写入列表中。（参见表 7-22）
- 请求零　所需的页面应该是零页面。
- 页面文件　所需的页面驻留在页面文件中。
- 映射文件　所需的页面驻留在映射文件中。

虽然这些原型 PTE 项的格式和本章前面描述的真实的 PTE 项相同，但是这些原型 PTE 不用于地址转换——它们是页面表和页面帧号数据库之间的一层，并且从不直接在页面表中出现。

通过把一个潜在共享页面的所有访问程序指向一个原型 PTE 来解决错误，内存管理器可以管理共享页面面不需要更新每个共享此页面进程的页面表。例如，一个共享的代码或数据页面在某点会被页面调出并存放到磁盘。当内存管理器从磁盘获取这个页面时，它只需更新原型 PTE 来指向页面的新的物理位置——每个共享此页面的进程的 PTE 都保持原样（清除有效位并仍指向原型 PTE）。然后，当进程引用页面时，真正的 PTE 将更新。

图 7-31 说明了在一个映射视图中的两个虚拟页面。一个是有效页面，另外一个是无效页面。如图所示，第一个页面是有效的，通过进程 PTE 和原型 PTE 指向它；第二个页面在调页文件中——原型 PTE 包含它的确切位置。进程 PTE（和任何其他的映射到那个页面的进程）指向这个原型 PTE。

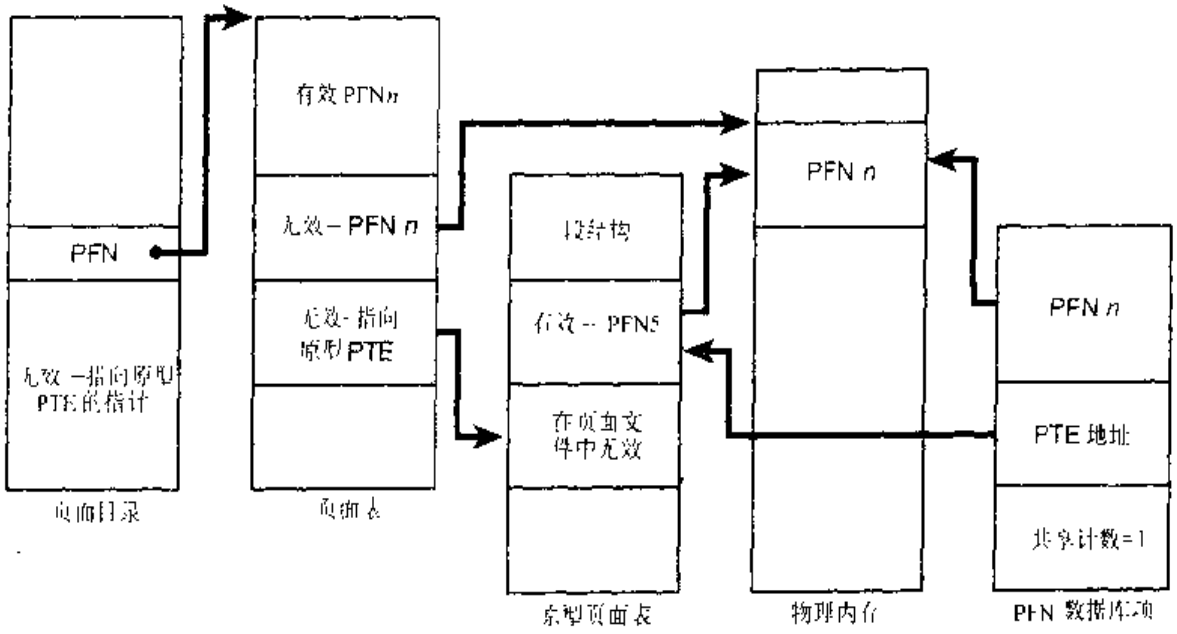


图 7-31 原型页面表项

7.6.3 页面调入 I/O

当必须对一个文件（页面调度或映射）发出读取操作来解决页错误时，将产生页面调入 I/O。同样，由于页面表是可调页的，当系统正在加载包含 PTE 或描述被引用的原型 PTE 的页面表页时，处理页错误可能会导致其他的页错误。

页面调入 I/O 操作是同步的——即线程等待一个事件直到 I/O 完成，并且不能被异步过程调度（APC）传送中断。页面调度程序使用 I/O 请求功能中的一个特殊的修改工具指示页面调度 I/O。完成页面调度 I/O 后，I/O 系统设置一个事件以唤醒页面调度程序并允许它继续页面调入处理。

当进行页面调度 I/O 操作时，出错的线程不拥有任何重要的内存管理同步对象。当页面调度 I/O 发生时，这将允许在进程中的其他线程发布虚拟内存函数和处理页错。但这也暴露出一些当 I/O 结束时页面调度程序必须识别的有趣情况：

- 在相同或不同进程中的另一个线程可能使同一个页面出错（这叫做页面冲突错误，将在下一节描述）。
- 页面可能已经从虚拟地址空间中删除（并被重新映射）。
- 页面保护可能已更改。
- 错误可能是关于原型 PTE 的，映射此原型 PTE 的页面可能被调出了工作集。

页面调度程序通过在页面调度 I/O 请求之前在线程的内核堆栈保存足够的状态来处理这样的情况。当请求结束时，它可以检测这些情况并且（如果有必要）不把页面设置为有效来消除页错误。当错误指令再次发出时，页面调度程序又被激活，PTE 在它的新状态下被再一次评估。

7.6.4 页面冲突错误

当另一个线程或进程使一个目前正在页面调入的页面出错时，这种情况就叫做页面冲突错

误。由于它们通常发生在多线程系统中，页面调度程序能检测并很好地处理页面冲突错误。如果另一个线程或进程使同一个页面出错，页面调度程序就检测页面冲突错误，通告页面正在转换中并且正在读取页（这个信息在 PFN 数据库项中）。在这种情况下，页面调度程序对在 PFN 数据库项中的指定事件发出等待操作。首先发送 I/O 的线程需要解决此错误以初始化这个事件。

当完成 I/O 操作时，所有等待该事件的线程的等待已经被满足。首先获得 PFN 数据库锁的线程负责执行页面调入完成操作。这些操作包括检查 I/O 状态以确定 I/O 操作成功完成、清除 PFN 数据库中的“读取在进行中”位并更新 PTE 元素。

当后来的线程获得 PFN 数据库锁来完成页面冲突错误时，页面调度程序将确认当清除“读取在进行中”位时初始化更新已经被执行，并且检查 PFN 数据库元素中的页面调入错误标志以确保页面调入 I/O 成功完成。如果设置了页面调入错误标志，PTE 将不能被更新并且在出错的线程中产生页面调入错误异常。

7.6.5 页面文件

页面文件被用来存贮那些仍然由某些进程使用却又不得不写到磁盘（因为修改页面的写操作）的修改页面。页面文件空间不会保留直到页面被写出磁盘而不是在提交的时候。然而，就象它们的创建一样，系统对专用页面的提交有限制。这样，Process: Page File Bytes 的性能计数器实际上是整个进程的私有提交内存，其中的部分或者全部都可能在页面文件中。（实际上，Process: Private Bytes 的性能计数器也是如此。）

内存管理系统在全局基础上从两个方面跟踪专用提交内存的使用，即提交（commitment）和在每个进程基础上的页面文件配额（另外，内存使用不同于页面文件使用——它表示专有提交内存使用）。每当需要新专有物理页的虚拟地址被提交时，提交和页面文件配额就被征用。一旦达到全局提交限制（物理内存和页面文件满），分配虚拟内存将失败直到进程释放提交的内存（例如，一个进程退出）。

Windows 2000 支持多达 16 个调页文件。当系统启动时，会话管理器进程（在第 2 章中描述）通过检查注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ Memory Manager \ PagingFiles 读取页面文件列表并打开页面文件。如果没有指定调页文件，默认的 20MB 的页面文件在启动分区上创建。（嵌入版本，例如 Windows NT 4 嵌入版本，没有默认页面文件。）一旦打开了页面文件，在系统运行期间页面不能被删除，因为“System”进程（也在第 2 章中描述）维持为每个页面文件的打开句柄。

实验：查看系统页面文件

要查看页面文件列表，查找注册表中的 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ Memory Management \ PagingFiles。不要试图改变注册表设置而增加或减少页面文件。使用控制面板中的系统应用程序增加或减少页面文件。单击在 Advanced 标签的 Performance Option 按钮，然后单击 Change 按钮。

要添加一个新的页面文件，“控制面板”将使用在 Ntdll.dll 中定义的 NtCreatePagingFile 系统服务（仅仅在内部）。页面文件总是以非压缩文件被创建，即使它所在的目录是压缩的。要保持新文件不被删除，一个句柄会被复制到“系统”进程，这样当创建进程关闭新的页面文件的句柄时，页面文件仍被另外的进程打开着。表 7-15 中列出的性能计数器允许在系统范围或每个页面文件基础上检查专用提交内存使用。没有办法可以用来确定一个进程的专有提交内存多少是驻留的和多少被调度到调页文件。

表 7-15 提交内存和页面文件性能计数器

性能计数器	说 明
Memory: Committed Bytes	已提交的虚拟内存的字节数（没有被保留）。由于包括在物理内存中没有调度出去的专有提交页，这个数据并不必然代表着页面使用。相反，它代表了如果进程成为完全非驻留时的页面文件空间使用的数量
Memory: Commit Limit	不需要扩充调页文件就可以被提交的虚拟内存字节数；如果页面文件可以被扩充，这个限制就是可变化的
Paging File: %Usage	已经提交的调页文件的百分数
Paging File: %Usage Peak	提交的调页文件的最高百分数

实验：使用 Task Manager 查看页面文件的使用

可以通过单击 Task Manger 的 Performance 标签查看系统页面文件和内存使用性能计数器。看到如图 7-32 所示的与页面文件相关的计数器：

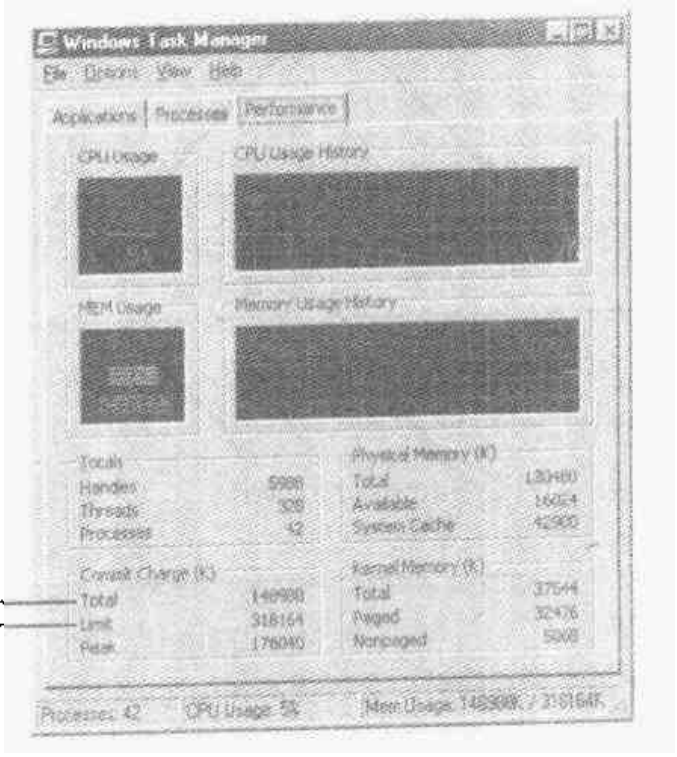


图 7-32 查看页面文件的使用

7.7 虚拟地址描述符

内存管理器使用请求页面调度算法来了解何时将页面调入内存，并等待直到某个线程引用一个地址并导致了一个页错误时才从磁盘中读回页面。像写时复制一样，请求页面调度是一种惰性求值形式——直到需要时它才去执行任务。

内存管理器使用惰性求值不仅把页面调进内存，而且构造描述新页面所需的页面表。例如，当一个线程使用 VirtualAlloc 提交了大量的虚拟内存时，内存管理器可以立即构造访问整个分配的内存区域所需的页面表。但是如果这个区域的某一部分从来不被访问又会发生什么样的情况呢？创建用于整个区域的页面表也许是白费力气。相反，内存管理器等待直到一个线程导致了页错误，才为那个页面创建页面表。这种方式极好地提高了那些保留和（或）提交了许多内存但很少访问它们的进程的性能。

用惰性求值算法，甚至分配大块的内存也会是一个快速的操作。然而，这种性能的提高是有代价的，当线程分配内存时，内存管理器必须作出反应提供线程使用的地址范围。因为内存管理器直到线程在事实上访问内存时才建立页面表，所以它不能确定哪些虚拟地址是空闲的。为了解决这个问题，内存管理器维持另一组数据结构来保持跟踪哪些虚拟地址已经在进程的地址空间中保留以及哪些没有在进程的地址空间中保留。这些数据结构被称为虚拟地址描述符（virtual address descriptor，VAD）。对每个进程，内存管理器都保留一组描述进程地址空间状态的 VAD。VAD 被构造成一个自平衡的二叉树来使查阅更有效。在图 7-33 中显示了 VAD 树的示意图。

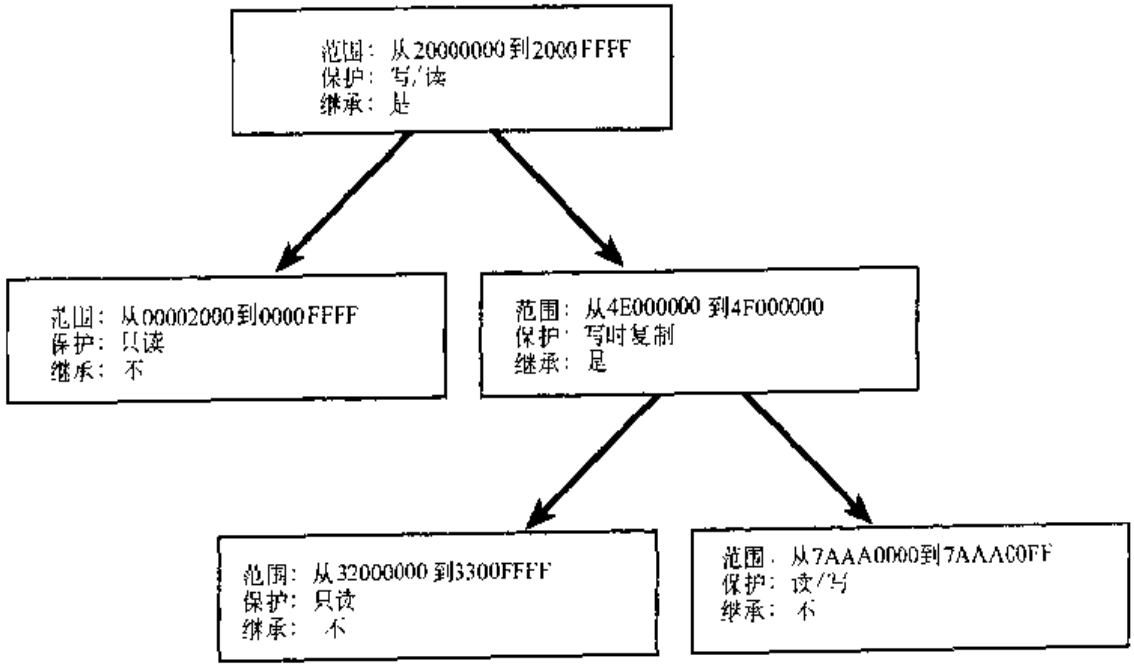


图 7-33 虚拟地址描述符

当进程保留地址空间或映射一个区域的视图时，内存管理器将创建一个 VAD 来保存分配请求提供的任何信息，例如保留的地址范围、地址范围是共享的还是专用的、子进程是否能继承地址范围的内容以及应用于范围内的页的页保护。

当线程第一次访问一个地址时，内存管理器必须为包含此地址的页面创建一个 PTE。要做到这些，它找到其地址范围包含所访问地址的 VAD，并且用它找到的信息填入 PTE。如果这个地址落在 VAD 覆盖的区域以外或在保留但没有提交的地址范围内，在试图使用它之前，内存管理器会知道线程没有分配内存并因此产生一个访问违例。

实验：查看虚拟地址描述符

可以使用内核调试程序的！vad 命令来查看指定进程的 VAD。首先用！process 命令找到 VAD 树的根地址。然后给！vad 命令指定地址，运行 Notepad.exe 的进程的 VAD 树的例子如下所示：

```
kd> !process 2a0 1
Searching for Process with Cid == 2a0
PROCESS 8614d030 SessionId: 0 Cid: 02a0 Peb: 7ffd7000 ParentCid: 0554
    DirBase: 00d93000  ObjectTable: 81bc47c8  TableSize:  41.
    Image: notepad.exe
    VadRoot 8118d868 Clone 0 Private 252. Modified 0. Locked 0.
    :

kd> !vad 8118d868
VAD     level      start      end    commit
84df4148 ( 2)        10       10         1 Private      READWRITE
850cdbe8 ( 3)        20       20         1 Private      READWRITE
810b0ee8 ( 1)        30       6f         7 Private      READWRITE
8109d308 ( 3)        70       16f       32 Private      READWRITE
810e9a28 ( 2)       170       17f        0 Mapped       READWRITE
84aedfc8 ( 3)       180       195        0 Mapped       READONLY
8118d868 ( 0)       1a0       1ce        0 Mapped       READONLY
81190a08 ( 4)       1d0       210        0 Mapped       READONLY
85c7b928 ( 3)       220       223        0 Mapped       READONLY
86253a08 ( 4)       230       2f7        0 Mapped       EXECUTE_READ
810aab48 ( 2)       300       342        0 Mapped       READONLY
80db5448 ( 5)       350       64f        0 Mapped       EXECUTE_READ

Total VADs:    49  average level:    6  maximum depth: 13
```

7.8 工作集

在以上几节中，集中描述了 Windows 2000 进程中的虚拟视图——页面表、PTE 和 VAD。在本章的剩余部分，将解释 Windows 2000 怎样在物理内存中保持一个虚拟地址子集。

正如你所记忆的，用来描述驻留在物理内存中的虚拟页面子集的术语叫作“工作集”(working set)。有两种工作集——进程工作集和系统工作集。

注意 支持 Windows 2000 的 Terminal Service（在单个 Windows 2000 服务器系统上支持多个单独的交互用户会话）的内核扩展将添加第三种类型的工作集：会话工作集。会话由一组进程和一个会话工作集组成，这个工作集用于通过 Win32 子系统(WIN32K.SYS) 的内核模式部分分配与内核模式会话相关的数据结构、会话工作集代

码和数据、会话页面调度交换区、会话映射视图和其他会话空间设备驱动程序。

在分析每一种工作集类型的细节之前，先看一看决定哪些页面将被调入物理内存并保留多长时间的整体策略。这以后，将研究两种类型的工作集。

7.8.1 页面调度策略

虚拟内存系统通常定义三种策略来规定如何（或何时）执行页面调度操作：取页策略、置页策略和替换策略。

“取页策略”（fetch policy）决定什么时候页面调度程序把一个页从磁盘调入内存。有一种取页策略试图把进程需要的页面在使用之前装入内存。另一种取页策略，称为“请求页面调度策略”（demand - paging policy），仅当发生页面错误时才将所需页面调入物理内存。在请求页面调度系统中，当进程的线程第一次开始执行时，进程会引发许多页错误，因为线程需要引用一些初始页面才能运行。一旦这些页面被调入内存，该进程的页面调度活动就会减少。

注意 为了优化映像的启动时间，Platform SDK 提供了一个名为 Working Set Tuner 的实用程序。这个实用程序记录了在可执行映像中的页面，按照它们在映像启动期间被引用的顺序放置它们，这样就减少了加载时间。

Windows 2000 内存管理器使用请求页面调度算法及“聚簇（clustering）”方法把页面装入内存。当线程收到页错误时，内存管理器把出错页面和它后面的一些页面装入内存。这个策略试图把线程引发的页面调度 I/O 的数量减到最小。因为有一些程序，特别是一些大的程序，在任何给定的时间内都只在它们的地址空间的小区域内执行，因此加载几簇虚拟页面会减少读取磁盘的次数。决定默认页面读取的簇数依赖于物理内存的大小，列在表 7-16 中。注意，可执行映像的页面与其他页面的值不同。

表 7-16 页错误读簇值

内存大小	映像中代码页的簇大小	映像中数据页的簇大小	其他页的簇大小
< 12MB	3	2	5
12 ~ 19MB	3	2	5
> 19MB	8	4	8

注：注意 Windows 2000 支持的最小内存为 32MB。然而，将来的嵌入式版本可能支持更小的内存

当线程收到页错误时，内存管理器也必须确定在物理内存中放置虚拟页面的地方，为此要使用的一组规则被称为“替换策略”。当选择页帧要考虑 CPU 内存高速缓存的大小来使不必要的高速缓存震荡最小。

如果当页错误发生时物理内存满了，使用“替换策略”来决定哪一个虚拟页面必须从内存中删除来为新的页面腾出空间。常用的替换策略包括“最近最少使用”（LUR）、“先进先出”（FIFO）。LRU 算法要求虚拟内存系统跟踪内存中的页面何时被使用。当需要新的页帧时，最长时间没有使用过的页面被调度到磁盘上，它的页帧被释放以满足页错误。FIFO 算法稍简单一些，它把在物理内存中驻留时间最长的页面删除而不管它使用得多么频繁。

替换策略可以进一步按特性分为全局的和局部的。全局的替换策略允许页错误被任何页帧

满足，而不管那些页帧是否属于另一个进程。例如，全局的替换策略使用 FIFO 算法在内存中寻找驻留时间最长的页面并且释放它来满足页错误；局部替换策略把对最老的搜索限定在引发页错误的进程拥有的页面集中。全局的放置策略使进程易受其他进程行为的侵犯——一个具有有害行为的应用程序能够通过在所有进程中引起过量的页面调度活动而破坏整个操作系统。在多处理器系统系统上，Windows 2000 实现局部 FIFO 替换策略的变化形式。在单处理器系统上，Windows 2000 实现更接近于最近最少访问策略的方法 LRU（称作“时钟算法”（clock algorithm），用于大部分的 UNIX 版本中）。Windows 2000 分配给每个进程一定数量的（动态调节的）页帧，称为“进程工作集”（process working set）（或在可调页的系统代码和数据的情况下，为“系统工作集”（system working set））。由于来自其他进程对物理内存的请求，当工作集达到它的界限或工作集需要被修整时，内存管理器将从工作集中删除页面直到它确认有足够的空闲页面。如何管理工作集将在下一节解释。

7.8.2 工作集管理

每个进程都以同样的默认的工作集的最大值和最小值开始。这些数值列在表 7-17 中，在系统初始化时计算它们并且严格地依赖物理内存的大小（对小、中、大内存系统的解释，请参阅 7.1.1 节“配置内存管理器”。

表 7-17 默认的工作集的最大值和最小值

内存大小	默认的最小工作集大小（页面）	默认的最大工作集大小（页面）
小	20	45
中	30	145
大	50	345

你可以用 Win32 的 SetProcessWorkingsetSize 函数为每个进程改变这些默认值，但是你必须有“增加调度优先级”的用户权力。最大工作集大小不能超过系统范围内的最大值，这个数值在系统初始化时计算并存储在全局变量 MmMaximumWorkingSetSize 中。这个数值被设置为在计算时有效页面的数量（置零、空闲和备用列表的大小）再减去 512 个页面。然而，这个计算值有固定的 1984MB 的限制，运行带有 3GB 用户空间的系统有 3008MB 的限制。

当页错误发生时，检查进程的工作集限制和系统上的空闲内存数。如果条件允许，内存管理器允许进程把它的工作集增加到最大值（或超过这个最大值——如果有足够的空闲页面就可以超过最大值）。然而，如果内存匮乏，Windows 2000 在页面错误时不增加页面而是替换页面。

尽管 Windows 2000 试图依靠把修改的页面写到磁盘来保持可用的内存，当修改的页面频繁产生时，需要更多的内存满足内存需求。因此，当物理内存很低时（MmAvailablePages 少于 MmMinimumFreePages），调用工作集管理器，运行在平衡集管理器环境中的一个例程（下一节描述）来初始化自动工作集修整（automatic working set trimming）来增加系统中有效空闲内存的数量。（通过提到过的 Win32 SetProcessWrokingSetSize 函数，可以初始化自动工作集修整自己的进程，例如，在应用程序初始化之后。）

如果存在工作集需要修整，工作集管理器检查可用内存并且决定哪个工作集需要修整。如果有足够的内存，工作集管理器计算需要多少页面可以从工作集中删除。如果需要修整，它查

找超过最小设置的工作集。同时也动态调整检查工作集的频率和要修整到优化顺序的进程列表的范围。例如，空闲长的大进程比频繁运行的小进程更先考虑；运行在前台的应用程序最后考虑；等等。

表 7-18 列出了一些影响工作集扩展和修整的系统变量，这些值是固定的或是系统设定的，不能通过注册表改变。

表 7-18 工作集相关的系统控制变量

变量	值	说明
MmWorkingSetSizeIncrement	6	如果有足够的可用页面并且工作集大小小于它的最大值，添加到工作集的页面数
MmWorkingSetSizeExpansion	20	如果工作集的大小达到了它的最大值并且系统中有足够的可用页面，扩展的最大工作集页面数
MmWsExpandThreshold	90	把工作集扩展到大于它的最大值时必须可用的页面数
MmPagesAboveWsMinimum	动态	如果每个工作集都达到最小值，从工作集中删除的页面数
MmPagesAboveWsThreshold	37	如果内存变小，并且 MmPagesAboveWsMinimm 大于该值，就修整工作集
MmWsAdjustThreshold	45	在试图缩小工作集之前由工作集缩小而释放的所需页面数
MmWsTrimReductuonGoal	29	通过工作集修整缩小的总页面数

当它发现进程的使用大于最小值时，将从进程工作集中删除一些页面，使这些页面能被其他的进程使用。如果空闲的内存还是太少，内存管理继续从进程的工作集中删除页面直到它达到系统中空闲页面的最小值。

如果由于上次修整后引起了很多页错误，此进程将免于修整，这个原理是，如果内存管理器犯了一个错误并且修整的页面是正被使用的页面，它会停止修整直到下一个修整周期（6 秒钟之后）。

在单处理器系统上和在多处理器系统上决定哪些页从工作集中删除的算法是不同的。在单处理器系统上，内存管理器试图删除最近不被访问的页面。它做到这点是通过检查硬件 PTE 中的“已访问位”来查看此页是否被访问。如果此位清除，页面被增加年龄，即计数器加 1 表明自从上次工作集修整扫描后页面没有被引用过。以后，页面年龄用于查找从工作集中删除的候选页面。

如果硬件 PTE 已访问位已被设置，内存管理器将清除并继续检查工作集中的下一页。以这种方式，如果下一次工作集管理器检查这个页面时“已访问位”是清空的，它就能知道这个页面自从上次检查后未被访问。这种为了页面删除的工作集列表扫描操作一直进行，直到删除了所需的页面数量或扫描回到开始点（下一次修整工作集时，扫描将从它上次离开的地点重新开始）。

在多处理器系统上，工作集管理器不检查已访问位；清除“已访问位”需要使其他处理器的 TLB 入口无效，这将导致运行在其他处理器上的在同一进程的线程的 TLB 高速缓存的不必要的丢失。这样，在多处理器系统上，页面将从工作集中被删除而不管“已访问位”的状态。

实验：查看进程工作集的大小

使用 Performance tool 中的以下性能计数器检查工作集大小。

计数器	说明
Process: Working Set	选择的进程工作集当前大小，单位字节
Process: Working Set Peak	选择的进程工作集峰值大小，单位字节
Process: Page Faults/Sec	进程每秒发生的页面错误数量

其他几个进程工作集检查工具（如 Task Manager、Pview 和 Pviewer）也可以显示进程工作集大小。

通过选择 Performance tools 实例框中的-Total process 得到所有的工作集。这个进程不是真实的，它是当前在系统运行的所有进程的特定计数器的简单的总和。然而，你看到的总数可能引起误解，由于每个进程的工作集包括与其他进程共享的页面 这样，如果两个或更多进程共享一个页面，此页面在每个工作集中都计算进去了。

实验：查看工作集列表

使用内核命令！wsle 查看工作集中的单个表项，以下例子是 LiveKd 的工作集列表的实际输出（此命令只能在 LiveKd 进程中使用）。

```
kd> !wsle /

Working Set @ c0502000
    Quota:      9f  FirstFree:  40  FirstDynamic:      5
    LastEntry  1fe  NextSlot:    3  LastInitialized  257
    NonDirect   5c  HashTable:   0  HashTableSize:     0

Virtual Address  Age  Locked  ReferenceCount
c0300203           0       1       1
c0301203           0       1       1
c0502203           0       1       1
c01df201           0       0       1
c01ff201           0       0       1
c0005201           0       0       1
c0001201           0       0       1
c0002201           0       0       1
c0000201           0       0       1
c0006201           0       0       1
77e87119           0       0       1
00402319           0       0       1
77e01201           0       0       1
7ffdf201           0       0       1
00130201           0       0       1
```

```
77e9e119        0       0       1
78033201        0       0       1
00230221        0       0       1
00131201        0       0       1
77d50119        0       0       1
00132201        0       0       1
c01e0201        0       0       1
00411309        0       0       1
0040d201        0       0       1
77edf201        0       0       1
77ee0201        0       0       1
77fcd201        0       0       1
0040e201        0       0       1
7ffc1009        0       0       1
00401319        0       0       1
```

注意一些工作集中列表的表项是页面表的页面（地址大于0xC0000000的那些），一些来自于系统DLL（在0x7nnnnnnn范围的那些）和一些来自于LiveKd.exe自身的代码（在0x004nnnnn范围中的那些）。

7.8.3 平衡集管理器和交换程序

工作集扩充和修整发生在叫作平衡集管理器（balance set manager）（KeBalanceSetManager例程）的系统线程环境中。平衡集管理器是在系统初始化时创建的。然而从技术的角度来看，平衡集管理器是内核的一部分，它通过调用内存管理器中的工作集管理器来执行工作集的分析和调整。

平衡集管理器等待两种不同的事件对象：一种是当设置为每秒激发一次的计时器到期后被设置为有信号态的事件；另一种是内部工作集管理器事件，管理器确定需要调整工作集时，它可以在不同的时候发出信号。例如，如果系统遇到高的页面出错率或空闲列表太少时，工作集管理器唤醒平衡集管理器以便能够调用内存管理器开始修整工作集。如果内存很多时，工作集管理器通过把出错页返回内存的方法允许出错进程逐渐增加它的工作集大小，但是工作集仅根据需要增加。

如果平衡集管理器由于其自身的1秒计时器到期而被唤醒，就会执行以下四个步骤：

1）因为1秒钟计时器已经到期，平衡集管理器每第四次唤醒时就会将一个事件设置为有信号状态，以唤醒称为“交换程序”（swapper）的系统线程（KeSwapProcessOrStack例程）。

2）检查“后备”列表，如果有必要就调整它们的深度（为了改善访问时间和减少交换区使用和交换区碎片）。

3）寻找由于处于CPU饥饿状态而需要确保其优先级被提高的线程（参见6.5.13节“提高优先级”一节）。

4）调用内存管理器的工作集管理器（工作集管理器有自身的内部计数器用来调整何时执行工作集修整和如何迅速修整。）

如果需要运行的线程把它的内核堆栈交换出内存或如果包含线程的进程把它的工作集交换出内存，交换程序也可以由在内核中的调度代码唤醒。交换程序寻找在一段时间内一直处于等待状态的线程（小内存系统中为3秒，中等内存或大内存系统中为7秒）。如果查找到了一个，交换程序就会让线程的内核堆栈处于变换状态（把页面移动到修改列表或后备列表）以便回收它的物理内存。基于这样的原理进行操作，即如果线程已经等待了很长时间，它将会等待更长的时间。当进程中的最后一个线程把它的内核堆栈从内存中删除时，进程工作集就被标记为全部交换出。例如，这也是长时间空闲的进程（例如登录后的 Winlogon）具有零工作集大小的原因。

7.8.4 系统工作集

如同进程中有工作集一样，可以由单一的系统工作集（system working set）管理操作系统中可分页的代码和数据。系统工作集中可以驻留五种不同的页面：

- 系统高速缓存页面。
- 页交换区。
- Ntoskrnl.exe 中可调页的代码。
- 设备驱动程序中可调页的代码。
- 系统映射视图（在 0xA0000000 映射的区域，例如 Win32k.sys）。

可以使用在表 7-19 中所示的性能计数器或系统变量来检查系统工作集的大小或影响工作集的五种组件的大小。需要记住的是性能计数器的值以字节为单位，而系统变量则以页面为单位计算。

可以通过检查 Memory：Cache Faults/Sec 性能计数器来检查系统工作集中的调页活动。这个计数器描述了在系统工作集中（软件和硬件两者）产生的页错误。包含此计数器值的系统变量是 MmSystemCacheWs.PageFaultCount。

表 7-19 系统工作集的性能计数器

性能计数器（字节）	系统变量（页面）	说明
Memory：Cache Bytes[1]	MmSystemCacheWs WorkingSetSize	系统工作集大小的总和（包括高速缓存、页交换区、可调页的 Ntoskrnl 和驱动程序代码以及系统映射的视图）；虽然其名称表明它是系统高速缓存的大小，但它并不只是系统高速缓存单独的大小
Memory：Cache Bytes Peak	MmSystemCacheWs.Peak	系统工作集大小的峰值
Memory：System Cache Resident Bytes	MmSystemCachePage	系统高速缓存消耗的物理内存
Memory：System Code Resident Bytes	MmSystemCodePage	Ntoskrnl.exe 中可调页代码消耗的物理内存

（续）

性能计数器（字节）	系统变量（页面）	说　明
Memory：System Driver Resident Bytes	MmSystemDriverPage	可调页的设备驱动程序代码消耗的物理内存
Memory：Pool paged Resident Bytes	MmPagedPoolPage	页交换区消耗的物理内存

注：　在内部，这个工作集被称为系统“高速缓存”（cache）工作集，尽管系统高速缓存只是五种不同组件中的一种，因此，当它们显示系统工作集的总的大小时，有些实用程序认为所显示的是文件高速缓存的大小。

系统工作集大小的最大值和最小值是在系统初始化时基于计算机的物理内存的数量和系统使运行 Windows2000 Professional 还是 Windows 2000 Server 来计算的。表 7-20 列出了系统依靠内存大小为基础的初始值选择。

表 7-20　系统工作集的最大值和最小值

内存大小	系统工作集的最小值（页面）	系统工作集的最大值（页面）
小	388	500
中	688	1150
大	1188	2050

如果注册表键值 HKLM \ SYSTEM \ CurrentContolSet \ Conrtol \ Session Manager \ Memory Management \ LargeSystemCache 设置为 1（Windows 2000 Server 中默认值为 0），并且可用页面的数量（表 7-25 描述的 MmAvailablePages）大于 350 + 6MB（在 x86 系统中总共有 1886 页），这些数字将被改变。在这种情况下，系统工作集的最大值设置为可用页面减去 4MB。如果该值大于 Windows 2000 支持的工作集大小（普通 x86 系统的最大值为 1984MB，或在带有 3GB 用户空间的系统中为 3008MB），系统工作集的最大值就要缩小为此最大值减去 5 个页面。

然后，Windows 2000 检查新的系统工作集的最大值是否大于系统高速缓存的虚拟大小。如果大于它，工作集就要缩小为系统高速缓存的虚拟大小。换句话说，系统集能够潜在地扩展以使用系统高速缓存保留的全部虚拟内存（关于系统高速缓存虚拟大小的详细信息，请参阅第 11 章）。

最后将做一个检查以确定系统工作集的最大值和最小值之差是否小于 500 个页面。如果小于，工作集的最小值就要缩小为工作集的最大值减去 500 个页面。然后把最后计算出的工作集的最大值和最小值存储在表 7-21 中的系统变量中（任何性能计数器都不能得到这些变量）。

表 7-21　存储工作集最小值或最大值的系统变量

变　量	类　型	说　明
MmSystemCacheWsMinimum 或 MmSystemCacheWs.MinimumWorkingSetSize	ULONG	工作集大小的最小值
MmSystemCacheWsMaximum 或 MmSystemCacheWs.MaximumWorkingSetSize	ULONG	工作集大小的最大值

7.9 页面帧号数据库

虽然工作集描述了进程或系统所拥有的驻留页面，但是页面帧号数据库（page frame number（PFN）database）描述了物理内存中每个页面的状态。页面将具有如表7-22显示的八种状态中的一种。

表7-22 页面状态

状 态	说 明
活动（也称为有效）	页面是工作集的一部分（进程工作集或系统工作集）或者它不在任何工作集中（例如下页式内核页面），并且有一个可用的PTE指向它
过渡	页面的临时状态，工作集不拥有此页面并且它也不存在于任何调页列表上。当到该页面的I/O正在进行中时，该页面就处于这种状态。把PTE编码以便能够正确地识别和处理页面冲突错误
备用	以前属于工作集但已经被删除的页面。自从页面最后驻留以来，它就没有被修改过。PTE仍然指物理页面，但是该页面被标记为无效并处于过渡状态
更改	以前属于工作集但已经被删除的页面。然而，页面在使用期间被修改并且它的内容尚未写入磁盘。PTE仍然指物理页面，但是该页面标记为无效并处于过渡状态。在物理页面使用前它必须先写到磁盘
更改不写入	除了做出标记以外，它和更改页面一样。这样内存管理器的修改页面写入程序不把它写入磁盘。高速缓存管理器在文件系统驱动程序发出请求时把页面标记为“更改不写入”。例如，NTFS为包含文件系统的元数据的页使用这个状态以便在受保护的页面被写入磁盘之前能够首先保证事务日志项刷新到磁盘（在第12章中对NTFS的事务日志做出解释。）
空闲	页面处于空闲状态但其中含有“脏”数据（由于安全的原因，在没有用零初始化页面的情况下，不能把“脏”页面作为用户页面在用户进程中使用）
置零	页面处于空闲状态并且已被清零页面线程初始化为零页面
坏页	产生奇偶检验错误或其他硬件错误的不能使用的页面

PFN数据库由结构数组组成，该数组代表系统中的内存每个物理页面。图7-34列出了页面帧号数据库和它与其他页面表的关系。正如图所示，有效的PTE指向页面帧号数据库中的项，并且页面帧号数据库中的项（对于非原型PFN）指回正在使用它们的页面表。对于原型PFNs，它们则指回原型PTE。

表7-22列出了页面状态，这六种状态被组织为链表以便内存管理器能够迅速查找特定类型的页面（活动/有效页面和过渡页面不在任何系统范围内的页面列表中）。图7-35显示了这些项是如何链接的例子。

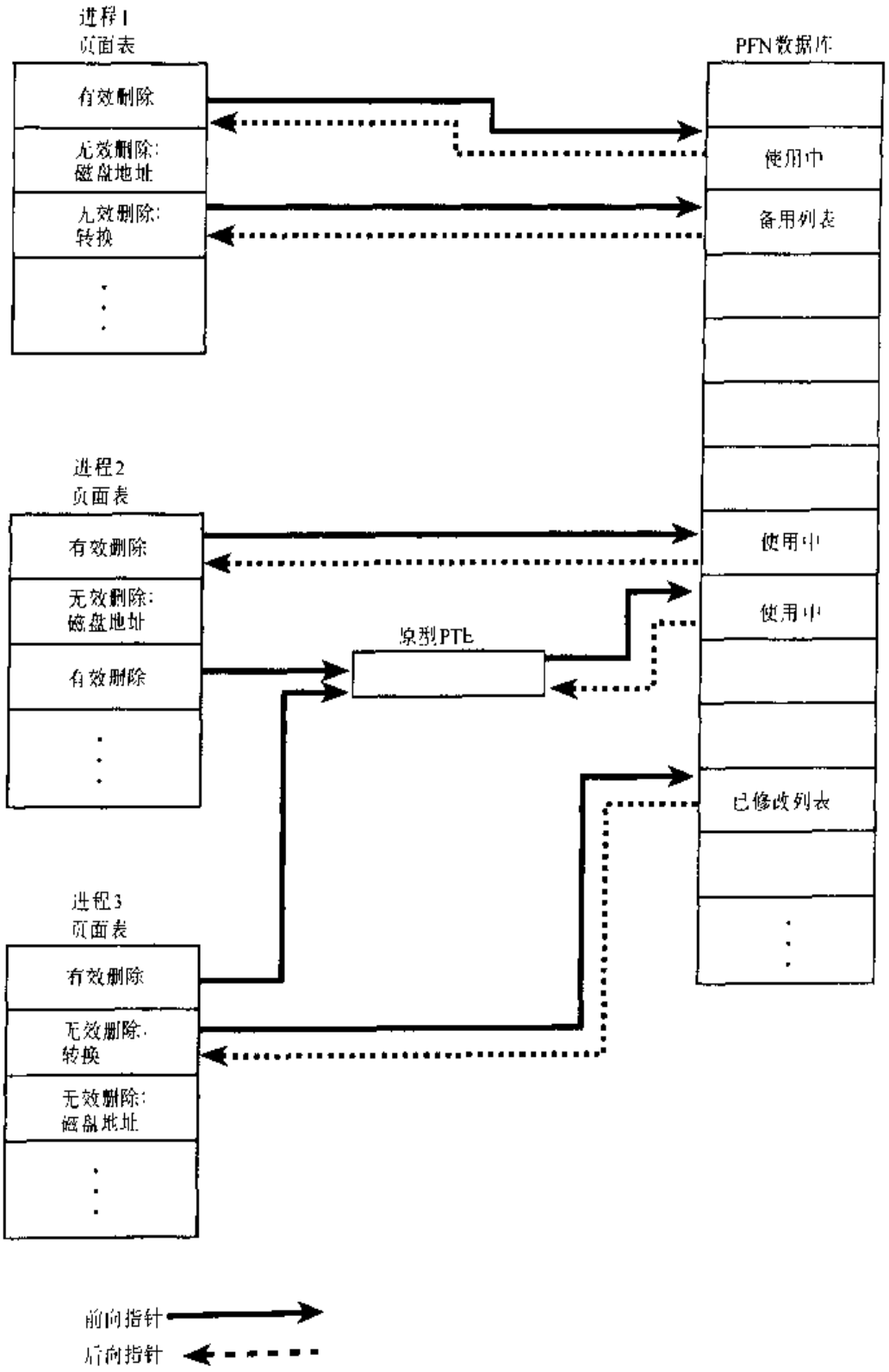


图 7-34 页面表和页面帧号数据库

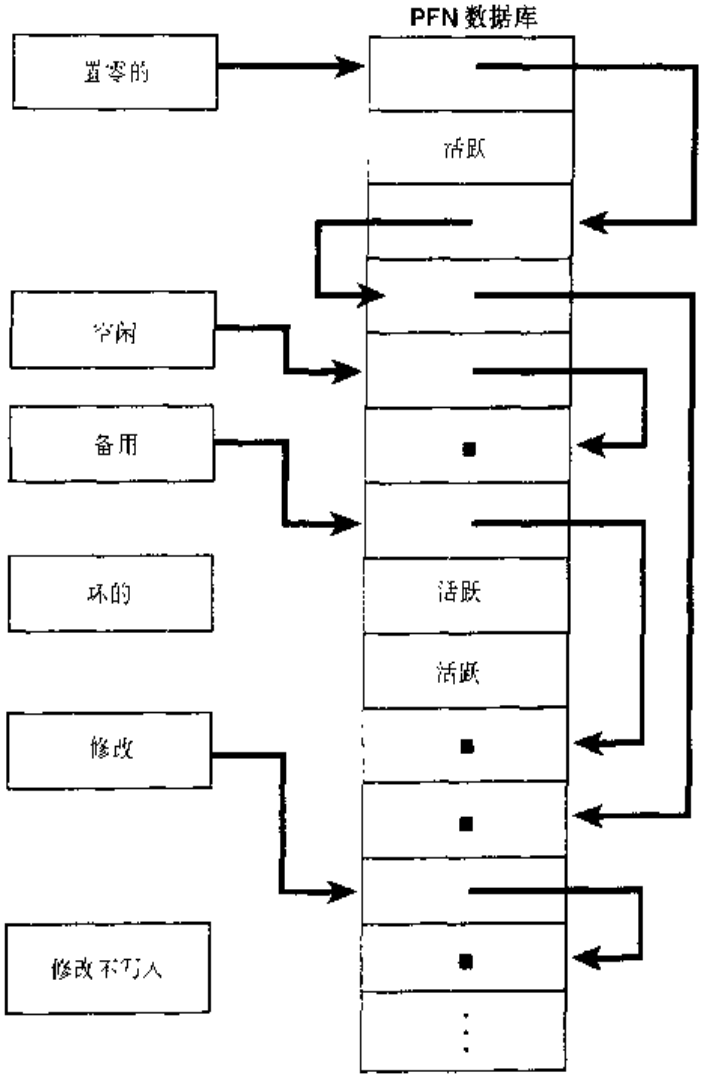


图 7-35 PFN 数据库中的页面列表

下一节将发现这些列表如何满足页面错误和如何从各种列表添加和删除页面。

实验：查看 PFN 数据库

使用内核命令！memusage，转储各种页面列表的大小，以下是此命令的输出：

```
kd> !memusage
 loading PFN database
loading (99% complete)
               Zeroed:      8 (    32 kb)
                 Free:      0 (     0 kb)
              Standby:   2809 ( 11236 kb)
             Modified:    756 (  3024 kb)
      ModifiedNoWrite:      1 (     4 kb)
        Active/Valid:  29150 (116600 kb)
           Transition:     10 (    40 kb)
              Unknown:      0 (     0 kb)
                TOTAL:  32734 (130936 kb)
Building kernel map
:
```

7.9.1 页面列表动态

图 7-36 显示了页帧过渡状态图。为了简明起见，在图中没有列出“更改不写入”列表。页帧以下列方式在调页链表之间移动：

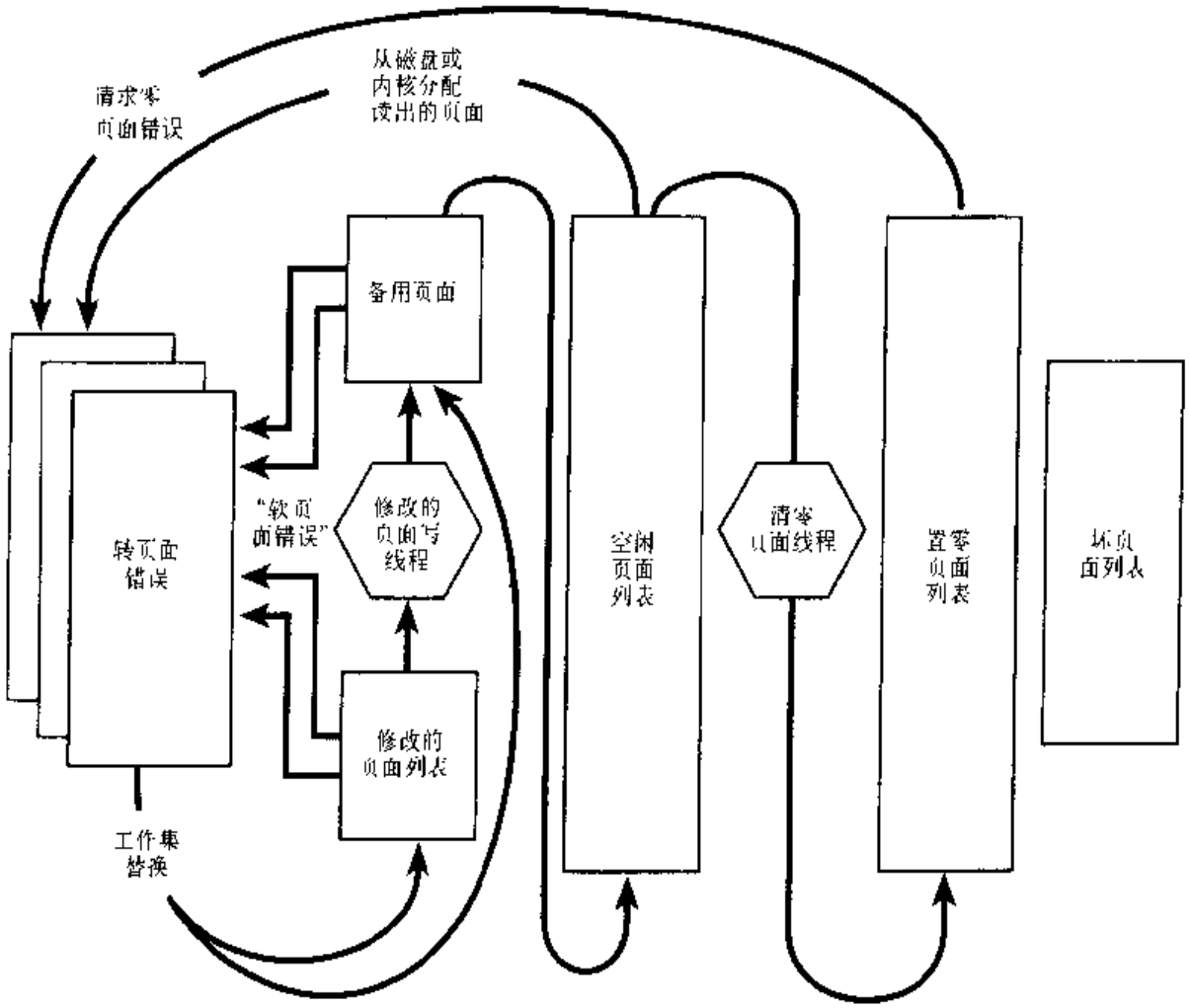


图 7-36 页面帧状态图

■ 当内存管理器需要一个零初始化的页面来为请求清零页面错误服务时（一个对定义为全为零的页面的引用，或者对从未被访问过的用户模式提交的专用页面的引用），它首先试图从零页面列表中得到一个页面；如果该表为空，它就从空闲页面列表得到一个页面，并用零初始化该页面。如果空闲列表为空，它就回到备用列表，并用零初始化那个页面。

需要把页面零初始化的原因之一是要满足 C2 安全性的要求。C2 指定用户态的进程必须得到初始化过的页帧，以免读出先前进程在内存中的内容。因此，内存管理器给用户模式进程零初始化的页帧，除非页面正从映射文件读入。如果是这种情况，内存管理器就要使用非零初始化的页帧，用磁盘中的数据初始化这些页帧。

零初始化页面列表是由一个被称为“清零页面线程”（zero page thread）（系统进程中的线程 0）的系统线程从空闲列表中组装的。清零页面线程等待一个事件对象给它发信号以便去运行。当空闲列表有八个或更多的页面时，该事件将被置为有信号状态。然而，只有在其他所有

的线程都不运行时，清零页面线程才能运行，这是因为清零页面线程在优先级为0的情况下运行，而用户线程可以被设置的最低优先级为1。

■ 当内存管理器不需要零初始化页面时，它将首先转向空闲列表；如果空闲列表为空，它就转到置零列表。如果置零列表为空，它就转到备用列表。在内存管理器能够在备用或更改列表中使用页帧之前，它必须首先从仍然指向页帧的无效 PTE（或原型 PTE）中返回并删除引用。因为在页面帧号数据库中的项含有指向先前用户的页面表（或用于共享页面的原型 PTE）的指针，内存管理器能够迅速地查找 PTE 并做出适当的更改。

■ 当进程不得不从它的工作集中放弃一页（因为它引用新的页面并且它的工作集已满、内存管理器修整了它的工作集）时，如果页面是“干净的”（没有更改），页面就进入备用列表，或者如果页面在驻留期间被更改了，它就进入更改列表。当进程退出时，所有专有页面进入空闲列表。同样，当最后对支持页面文件的区域的引用关闭时，这些页面进入空闲列表。

实验：查看页错误行为

使用 Windows 2000 的资源工具包中的 PFMON，可以在页面错误时观察页错误行为。软错误指由过渡列表中的一个可以满足的页错误。硬错误指磁盘读。下面的例子使用 Pfmon 启动 Notepad，然后退出。你将看到所输出的一部分。一定注意结尾处的页错误摘要信息。

```
C:\> pfmon notepad
SOFT: KiUserApcDispatcher : KiUserApcDispatcher
SOFT: LdrInitializeThunk : LdrInitializeThunk
SOFT: 0x77f61016 :  : 0x77f61016
SOFT: 0x77f6105b :  : fltused+0xe00
HARD: 0x77f6105b :  : fltused+0xe00
SOFT: LdrQueryImageFileExecutionOptions :
                          LdrQueryImageFileExecutionOptions
SOFT: RtlAppendUnicodeToString : RtlAppendUnicodeToString
SOFT: RtlInitUnicodeString : RtlInitUnicodeString
:
 notepad Caused   8 faults had   9 Soft  5 Hard faulted VA's
   ntdll Caused  94 faults had  42 Soft  8 Hard faulted VA's
comdlg32 Caused   3 faults had   0 Soft  3 Hard faulted VA's
 shlwapi Caused   2 faults had   2 Soft  2 Hard faulted VA's
   gdi32 Caused  18 faults had  10 Soft  2 Hard faulted VA's
kernel32 Caused  48 faults had  36 Soft  3 Hard faulted VA's
  user32 Caused  38 faults had  26 Soft  6 Hard faulted VA's
advapi32 Caused   7 faults had   6 Soft  3 Hard faulted VA's
  rpcrt4 Caused   6 faults had   4 Soft  2 Hard faulted VA's
comctl32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
 shell32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
         Caused  10 faults had   9 Soft  5 Hard faulted VA's
winspool Caused   4 faults had   2 Soft  2 Hard faulted VA's

PFMON: Total Faults 250
      (KM 74 UM 250 Soft 204, Hard 46, Code 121, Data 129)
```

当更改过的列表变得过大，或者如果置零列表和备用列表的大小低于最小阈值时（如同在系统引导时计算出的内核变量 MmMinimumFreePage 所指定的那样），就要激活称为“修改页面

的书写器”（modified page writer）的系统线程将页面写回磁盘，并将页面移入备用列表。

7.9.2 修改页面的书写器

修改页面的书写器负责当更改过的列表变得太大时，通过将页面写回磁盘来限制它的大小，它由两个系统线程组成：一种是把更改过的页面写到调页文件（MiModifiedPageWriter）。另一种将更改过的页面写到映射文件（MiMappedPageWriter）。需要这两种线程从而避免产生死锁，当没有可用的空闲页面时，如果写入映射文件页面引起页错误并因此请求空闲页面（从而请求修改页面的书写器创建更多的空闲页面），就会出现死锁。通过让修改页面的书写器以第二个系统线程执行映射文件页面调度 I/O，线程就可以等待而不阻塞正常的页面文件 I/O。

这两种线程以优先级 17 运行，并且在初始化之后，等待单独的事件对象来触发它们的操作。修改页面写入器事件由以下两个原因引起：

■ 当更改过的页面数超过了系统初始化过程中计算出的最大值时（MmModified PageMaximum）。

■ 当可用页面数（MmAvailablePages）小于 MmMinimumFreePages 时。

表 7-23 显示了触发修改页面的书写器唤醒来减少更改列表大小的页面数量和留在列表中的页面的数量。此值和其他的内存管理变量一起是在系统引导时计算出来的，并且依赖于物理内存的数量。

表 7-23 修改页面的书写器的值

内存大小	修改页面的阈值	保留的修改页面数
< 12MB	100	40
12 ~ 19MB	150	80
19 ~ 33MB	300	150
> 33MB（特例）	400	800

修改页面的书写器等待一个在预定的秒钟时间段后设置的事件（MiMappedPagesTooOldEvent）来显示映射文件（而不是修改页面）应该写到磁盘。缺省地，该值为 300 秒（5 分钟）。（你也能通过添加 DWORD 注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ Memory Management \ ModifiedPageLife 来重载该值）。附加事件的原因是在系统崩溃或者电源失败的情况下通过把修改页面写出以减少数据损失，即使修改列表还没有达到表 7-23 所列出的阈值

在被调用时，修改页面的书写器使用单一的 I/O 请求试图把尽可能多的页面写到磁盘。它通过检查在更改页面列表上用于页面的 PFN 数据库元素的原始 PTE 字段，查找在磁盘中连续位置上的页面来完成此操作。一旦创建了一个列表，页面就从更改列表中删除，发送 I/O 请求并且在成功完成 I/O 请求后，页面被置于备用列表的末尾。

正在被写入磁盘的页面能够被其他进程引用。当发生这种情况时，表示物理页面的 PFN 项中代表物理页面的引用计数和共享计数就会增加，以指出此页面正在被另一个进程使用。当 I/O 操作完成时，修改页面的书写器就会注意共享计数不再为 0，并且不会将此页面置于备用

列表中。

7.9.3 PFN数据结构

虽然PFN项有固定的长度，但是它们可以有几种不同的状态，而这要依赖于页面。所以，不同字段针对不同状态有不同的含义。图7-37显示了PFN项的各种状态。

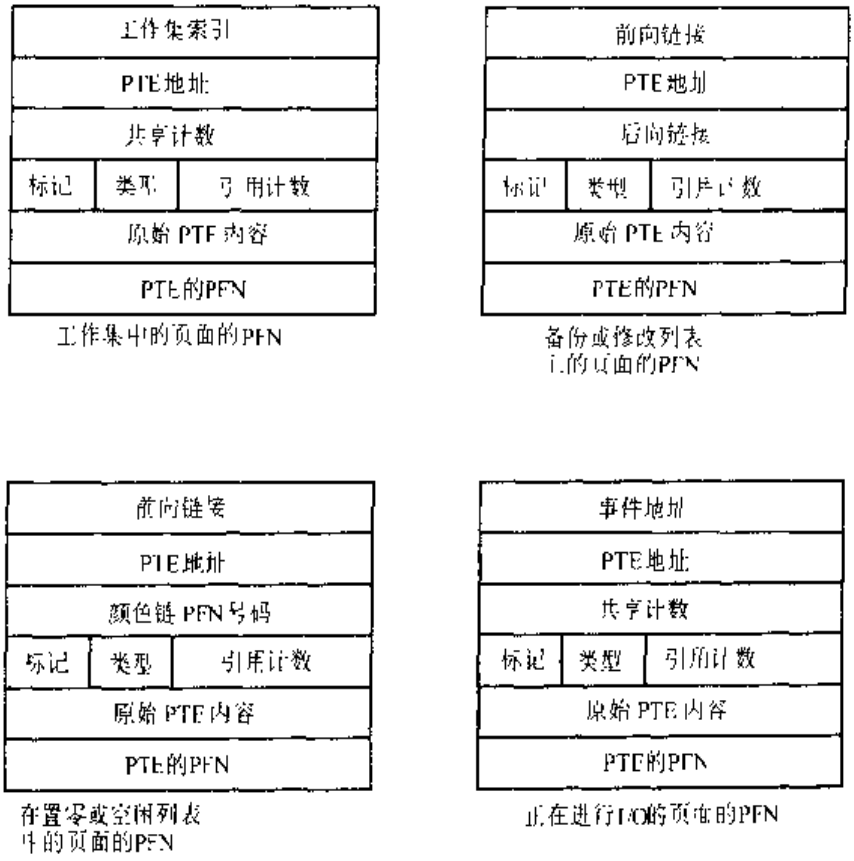


图7-37 PFN数据库项中的状态

几个字段在几个PFN类型中是相同的，但其他字段则特定于PNF的给定类型。下列字段可以出现在不只一种PFN类型中。

■ PTE地址 指向该页面的PTE虚拟地址。

■ 引用计数 该页面的引用数量。当一个页面被首次添加到工作集中和/或当用于I/O的页面被锁定在内存中时（例如，被设备驱动程序锁定），引用计数就递增。当共享计数变为0或页面从内存中解锁时，引用计数将减少。当共享计数变为0时，工作集就不再拥有页面。然后，根据它的引用计数，更新描述此页面的PFN项以把页面添加到空闲、备用或更改列表中。

■ 类型 由这个PFN代表的页面的类型（活动/有效、过渡、备用、更改、更改不写入、空闲、置零、坏页和过渡）。

■ 标志 在表7-24中列出的包含在标志字段中的信息。

■ 原始的PTE内容 所有的PFN项中都包含指向该页面的PTE的原始内容（可以是原型PTE）。保存PTE的内容允许当物理页面不再驻留时恢复它。

■ PTE的PFN包含指向这个页面PTE的页面表页的物理页面号码。

表 7-24 PFN 数据库项的标志

标志	含义
修改状态（Modified State）	显示页面是否被更改（如果页面被修改了，则从内存删除之前，页面的内容必须被保存到磁盘上）
原型 PTE（Prototype PTE）	显示 PFN 项引用的 PTE 是原型 PTE（例如，此页面是共享的）
奇偶错误（Parity error）	显示包含奇偶检验或错误更正控制错误的物理页面
正在读（Read in Progress）	显示对此页面的页面调入操作正在进行中，第一个 DWORD 包含当 I/O 完成时将被置为有信号的事件对象的地址；也用来显示用于非页交换区分配的第一个 PFN
正在写（Write in Progress）	显示页面的写入操作正在进行中，第一个 DWORD 包含当 I/O 完成时将被置为有信号态的事件对象的地址；也用来显示用于非页交换区分配的最后一个 PFN
非页交换区的启始地址（Start of nonpaged pool）	用于非页交换区，显示这是给非页交换区分配的第一个 PFN
非页交换区的尾端（End of nonpaged pool）	用于非页交换区，显示这是给非页交换区分配的最后一个 PFN
页面读入错误（Inpage error）	显示在此页面上进行页面调入操作时出现的 I/O 错误（在这种情况下，PFN 中的第一个字段包含错误代码）

剩下的字段是特定于 PFN 的类型的。例如，表 7-22 中的第一种 PFN 表示页面是活动的，并且是工作集的一部分。共享计数字段表示指向页面的 PTE 的数量（标记为只读、写时复制、或共享读/写的页面可由多个进程共享）。对于页面表页，这个字段是在页面表中的有效 PTE 的数量。只要共享计数大于 0，从内存中删除该页面就是非法的。

工作集索引字段就是一个进入进程（系统或会话的）工作集列表，如果在工作集中没有，就是零）工作集列表的索引，在工作集列表中驻留着映射这个物理页面的虚拟地址。如果该页面是专用页面，工作集索引字段就直接指向在工作集列表中的项，这是因为页面只映射单个虚拟地址。在共享页面的情况下，工作集索引是一种暗示（hint），它只保证对于使页面有效的第一个进程是正确的。（其他进程总是在需要时尽可能地使用相同的索引）最初设置字段的进程保证引用正确的索引，并且不需要把虚拟地址引用的工作集列表散列项添加到工作集散列树中。这种保证可以缩小工作集散列树的大小，并使得寻找那些特殊的直接项的过程变得比较快。

图 7-31 中的第二种 PFN 用于备用或更改列表中的页面。在这种情况下，向前和向后的链接字段就把列表中的元素在列表内链接在一起。这种链接允许容易地操纵页面以满足页错误。当页面为列表之一时，共享计数可以使用 0 定义（因为没有任何工作集使用此页面），因此可以使用向后链接覆盖。然而引用计数器可能不是零，因为用于这个页面的 I/O 可能正在进行中（例如，页面正被写到磁盘）。

图 7-37 中第三种 PFN 用于在空闲或置零列表上的页面。除了在这两种列表内的链接以外，这些 PFN 项还通过“颜色”（它们在处理器内存高速缓存中的位置）的字段链接这些物理页面。Windows 2000 通过使用 CPU 高速缓存中的不同物理页面使得 CPU 内存高速缓存不必要的振荡减少到最小。通过避免使用用于两个不同页面的相同的高速缓存项来完成这个优化。对于

带有直接映射高速缓存的处理器，利用硬件的能力进行优化就可以导致性能显著增加。

图 7-37 中第四种 PFN 则用于有 I/O 正在进行中的页面（例如，页面读取）。在进行 I/O 处理期间，第一个字段指向当 I/O 完成时将被置为有信号状态的事件对象。如果出现页错误，该字段就含有表示 I/O 错误的 Windows 2000 错误状态代码。这种 PFN 类型用于解决页面冲突错误。

实验：查看 PFN 项

可以使用内核调试器！pfn 命令查看单个 PFN 项，首先需要把 PFN 作为参数来提供（例如，！pfn 0 显示第一项，！pfn 1 显示第二项，以此类推）。在下面的例子种，显示了用于 PTE 的虚拟地址 0x50000，后面是包含页面目录的 PFN，然后是实际页面：

```
kd> !pte 50000
00050000  - PDE at C0300000        PTE at C0000140
          contains 00700067      contains 00DAA047
        pfn 00700 --DA--UWV    pfn 00DAA --D-  UWV
kd> !pfn 700
  PFN 00000700 at address 827CD800
  flink  00000004 blink / share count 00000010 pteaddress C0300000
  reference count 0001                                  color 0
  restore pte 00000080  containing page        00030  Active     M
  Modified

kd> !pfn daa
  PFN 00000DAA at address 827D77F0
  flink  00000077 blink / share count 00000001 pteaddress C0000140
  reference count 0001                                  color 0
  restore pte 00000080  containing page        00700  Active     M
  Modified
```

除了 PFN 数据库以外，表 7-25 中的系统变量还描述了物理内存的全部状态。

表 7-25 表述物理内存的系统变量

变 量	说 明
MmNumberofPhysicalPages	系统中可用物理页面的总数
MmAvailablePages	系统中可用页面的总数——置零、空闲和备用列表页面的总和
MmResidentAvailablePages	如果每个进程都处于其最小的工作集大小时可用的物理页面的总数

7.10 区域对象

正如你所记得的，在本章前面的关于共享内存的小节中提到的“区域对象”（Win32 子系统中叫作“文件映射对象”）表示两个或更多的进程可以共享的内存块，区域对象可以被映射到调页文件或磁盘上的其他文件中。

执行程序使用区域把可执行的映像加载到内存中，并且高速缓存管理器使用它们访问高速缓存文件中的数据（关于高速缓存管理器如何使用区域对象的详细信息参阅第 11 章）。也可以使用区域对象把文件映射到进程的地址空间中。通过映射区域对象的不同视图并直接读取或写入内存而不是文件（此活动称为映射文件 I/O（mapped file I/O），该文件可以被作为一个大型数组来访问。当程序访问无效页面（不是物理内存中的页面）时，就会出现页错误，并且内存管

理器会自动从映射文件中把页面调入内存。如果应用程序更改了页面，在内存管理器正常的调页操作期间，它就会把更改的内容写回文件（或者应用程序可以使用 Win32 FlushViewOfFile 函数刷新视图）。

同其他的对象一样，可以使用对象管理器分配和释放区域对象。对象管理器创建和初始化一个用于管理对象的对象头；内存管理器定义了区域对象体。同样，内存管理器实现用户模式线程可以调用的服务程序来检索并更改存储在区域对象体中的属性。图 7-38 显示了区域对象的结构。

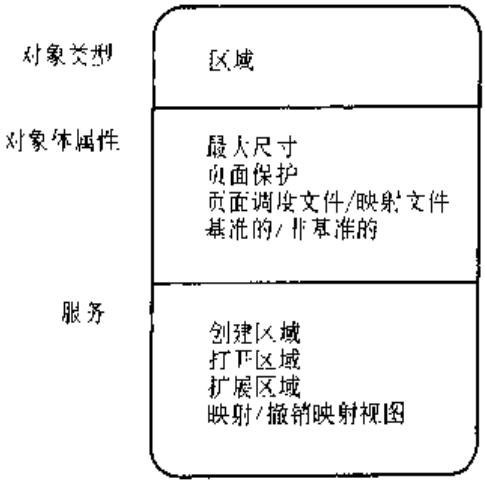


图 7-38 区域对象

表 7-26 存储在区域对象中的特有属性的总结。

表 7-26 区域对象体的属性

属性	目的
最大尺寸	区域可以增长的以字节为单位的最大值；如果映射一个文件，最大值是文件的大小
页面保护	当区域被创建时，分配给该区域的所有页面的基本页面的内存保护
调页文件/映射文件	显示被创建的区域是空的（由调页文件支持——如前面解释的，仅当页面需要写出磁盘时，支持页面文件的区域使用页面文件资源）或者用文件加载（由映射文件支持）
基准/非基准	显示区域是基准区域（为所有共享它的进程呈现相同的虚拟地址）还是非基准区域（为不同的进程呈现不同的虚拟地址）

实验：查看区域对象

使用 Object Viewer（书附带的 CD 中 \ Sysint \ Winobj 或平台 SDK 的 Winobj.exe），可以看到有全局名的区域列表。正如第三章解释的那样，这些名称存储在对象管理器的目录 \ BaseNamedObjects 下。

使用 Windows 2000 的资源工具包 OH（Open Handles）工具，可以列出为区域对象打开的句柄。下列命令显示对象类型为区域的全部打开的句柄，而不考虑区域是否有名称。（仅当其他进程需要通过名称来打开时，区域才必须由名称）。

```
c:\> oh  t section -a
00000008 System         Section  0070
000000BC smss.exe       Section  0004
000000A4 csrss.exe      Section  0004
000000A4 csrss.exe      Section  0024
000000A4 csrss.exe      Section  0038
000000A4 csrss.exe      Section  0040 \NLS\NlsSectionUnicode
000000A4 csrss.exe      Section  0044 \NLS\NlsSectionLocale
000000A4 csrss.exe      Section  0048 \NLS\NlsSectionCType
000000A4 csrss.exe      Section  004c \NLS\NlsSectionSortkey
000000A4 csrss.exe      Section  0050 \NLS\NlsSectionSortTbls
000000A0 winlogon.exe   Section  0004
000000A0 winlogon.exe   Section  0034
000000A0 winlogon.exe   Section  0168 \BaseNamedObjects\mmGlobalPnpInfo
:
```

你可以使用 HandleEx（书附带的 CD 中 \ Sysint \ Handleex）查看映射文件。选择 View 菜单的 View DLL 项。在 MM 列的带有星号的文件是映射文件（而不是 DLLs 和映像加载器作为模块调入的其他文件）。这里有一个例子，如图 7-39 所示：

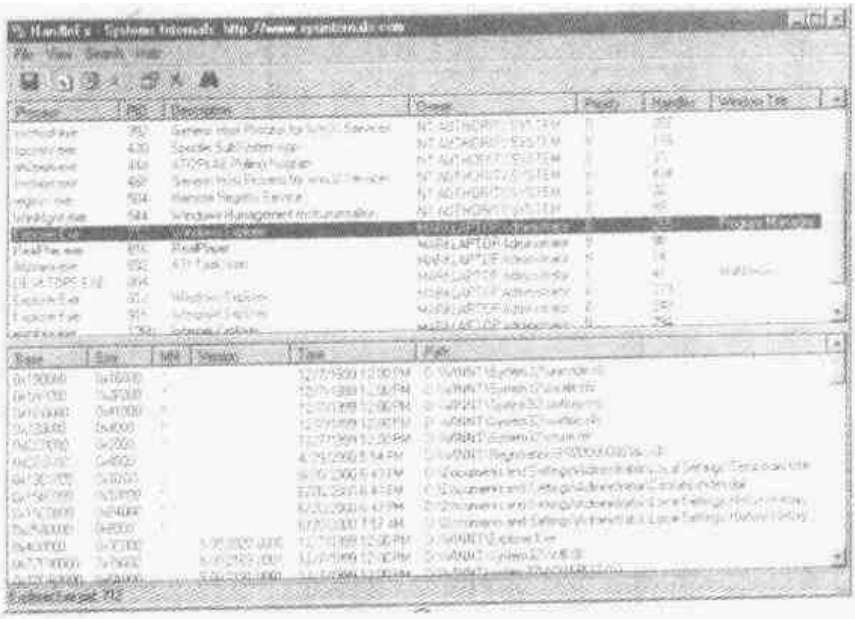


图 7-39 查看区域对象

图 7-40 显示了内存管理器维持的用于描述映射区域的数据结构。这些结构可以保证从映射文件中读出的数据是一致的，而忽略访问类型（打开文件、映射文件等等）。

对于每个打开的文件（由文件对象代表），这里都有一个单独的“区域对象指针”（secton object pointers）结构（这种结构将在第 11 章中讨论）。这种结构是为所有类型的文件访问维持数据一致性以及为文件提供高速缓存的关键。区域对象指针结构指向一个或两个“控制区域”（control areas）。当文件作为数据文件访问时，一个控制区域被用来映射文件，而当文件作为可执行的映像来运行时，另一个控制区被用来映射文件。控制区域反过来指向为文件的每个区域描述映射信息（只读、读写、复制写入等）的“子区域”（subsection）结构。控制区域也指向在页交换区中分配的“段”结构，“段”结构反过来指向原型 PTE，而这些原型 PTE 通常被区

域对象映射到实际页面上。正如本章前面描述的，进程页面表指向这些原型 PTE，而原型 PTE 反过来映射这些被引用的页面。

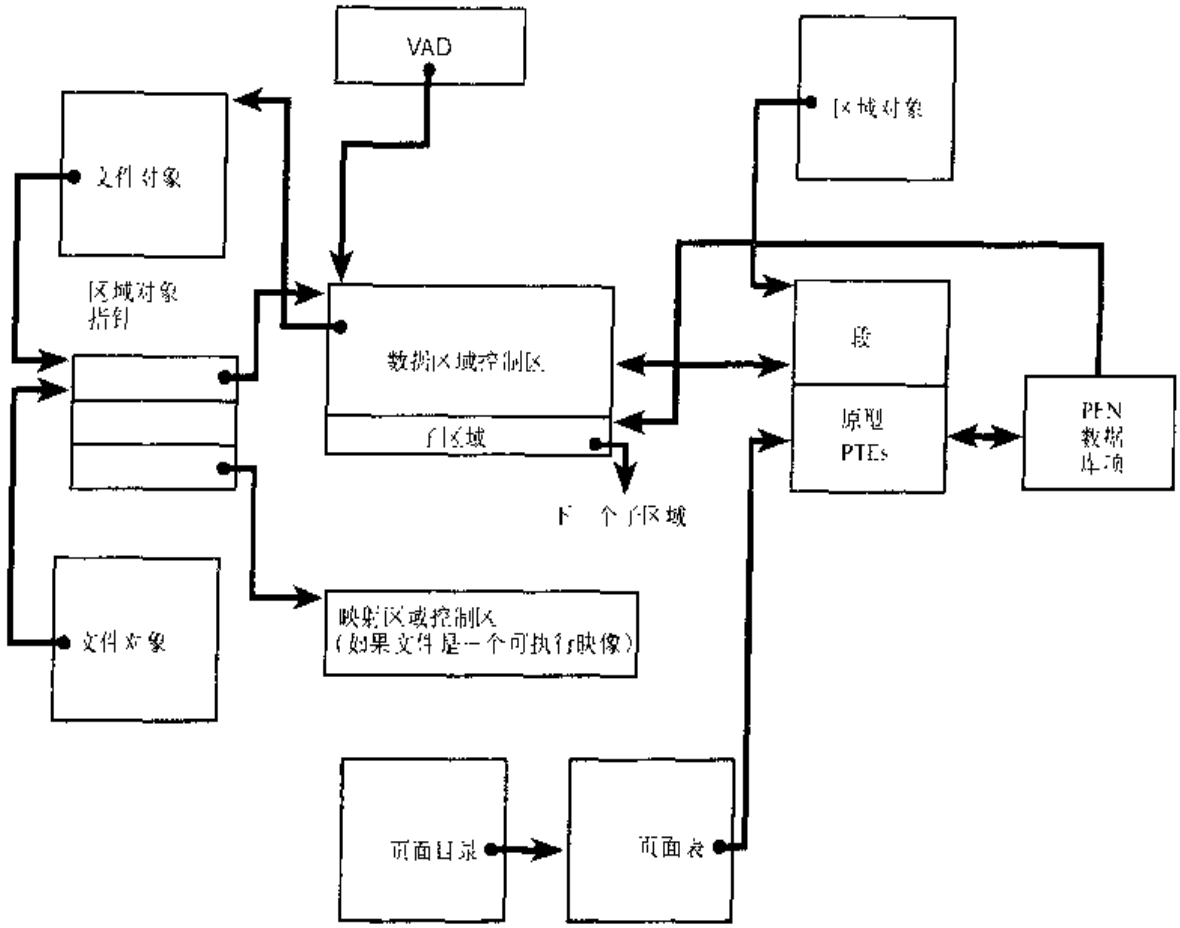


图 7-40 内部区域结构

虽然 Windows 2000 保证任何访问（读或写）文件的进程总是得到相同的、一致的数据，但存在一种情况文件页面有两个副本驻留在物理内存中（即使此情况下，所有的访问程序仍会得到最新的副本和保持一致的数据）。当文件作为数据文件已经被访问（已经被读取或写入），并且接着作为可执行的映像运行时，发生这种情况（例如，当一个映像被链接然后运行时——“链接程序”打开文件以便访问数据，然后当映像运行时，映像加载程序就把映像作为可执行的映像来映射）。

在内部，将发生下列操作：

1）当映像文件被创建时，将创建数据控制区域来表示在映像文件中被读取或写入的数据页面。

2）当运行映像并且创建区域对象来把映像映射为可执行程序（executable），内存管理器找到用于映像文件的区域对象指针指向的数据控制区域并刷新此区域。要保证在通过映像控制区域访问映像之前把修改的页面写入到磁盘中，这个步骤时必须的。

3）然后，内存管理器创建用于映像文件的控制区域。

4）当映像开始执行时，它的（只读）页面将从映像文件中产生错误。

因为由数据控制区映射的页面可能仍在驻留（在备用列表上），这是相同数据的两个副本

在内存中的不同页面上的情况。然而，这种复制并未导致数据一致性问题，如同已经提到过的，因为数据控制区域已经被刷新到磁盘上，所以从映像中读出的页面是最新的。

实验：查看控制区域

要想查找文件的控制区域结构的地址，你必须首先得到文件对象的地址。你可以通过使用！handle命令转储进程句柄表并且记录文件对象的对象地址来获得该地址，虽然内核调试器的！file命令显示了在文件对象中的基本信息，但是它不能显示指向区域对象结构的指针。然而，因为文件对象定义在公共的DDK头文件Ntddk.h中，所以可以查找偏移量（在Windows 2000中是0x14）。因此，在文件对象中简单地检查指向偏移量0x14处的指针，你将得到区域对象指针结构。该结构同时定义在Ntddk.h中。它由三个32位的指针组成：指向数据控制区域的指针、指向共享的高速缓存映射的指针（将在第11章中解释）和指向映像控制区域的指针。从区域对象指针结构中，你可以获得文件的控制区域的地址（如果文件存在），并可以把该地址送入！ca命令中。

另一种技术是使用！memusage命令显示全部控制区域的列表。下面的内容就是该命令输出的摘录

```
kd> !memusage
 loading PFN database
loading (99% complete)
            Zeroed:      9 (    36 kb)
              Free:      0 (     0 kb)
           Standby:   2103 (  8412 kb)
          Modified:    300 (  1200 kb)
     ModifiedNoWrite:      1 (     4 kb)
       Active/Valid:  30318 (121272 kb)
        Transition:      3 (    12 kb)
           Unknown:      0 (     0 kb)
             TOTAL:  32734 (130936 kb)
  Building kernel map
  Finished building kernel map
  Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables  name
8119b608  3000   528     0      0      0      0  mapped_file( WINWORD.EXE )
849e7c68    96     0     0      0      0      0    No Name for File
8109c388    24     0     0      0      0      0  mapped_file( OEVDTG.PKG )
81402488   236     0     0      0      0      0    No Name for File
80fba8a8   268     0     0      0      0      0  mapped_file( kernel32.pdb
810ab168  1504   380     0      0      0      0  mapped_file( OUTLLIB.DLL )
81126a08     0   276     0      0      0      0  mapped_file( H )
81112d28   656     0     0      0      0      0    No Name for File
:
```

“控制”列指向描述映射文件的控制区域结构。可以使用内核调试器中的！ca命令来显示控制区域、段和子区域。例如，要转储本例中的映射文件Winword.Exe的控制区域，在控制号后敲入！ca命令，以下是命令的输出：

```
kd> !ca 8119b608

ControlArea @8119b608
  Segment:    e2c28000  Flink             0  Blink:           0
  Section Ref        1  Pfn Ref         372  Mapped Views:    1
  User Ref           2  Subsections       6  Flush Count:     0
  File Object 8213fb98  ModWriteCount     0  System Views:    0
  WaitForDel         0  Paged Usage    3000  NonPaged Usage  100
  Flags (90000a0) Image File HadUserReference Accessed

   File: \Program Files\Microsoft Office\Office\WINWORD.EXE

Segment @ e2c28000:
   Base address        0  Total Ptes       86a  NonExtendPtes:      86a
   Image commit       d7  ControlArea 8119b608  SizeOfSegment: 86a000
   Image Base          0  Committed          0  PTE Template:  919b6c38
   Based Addr   30000000  ProtoPtes   e2c28038  Image Info:    e2c2a1e4

Subsection 1. @ 8119b640
   ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 8
   Base Pte     e2c28038  Ptes In subsect        1 Unused Ptes          0
   Flags              15  Sector Offset          0 Protection           1
    ReadOnly CopyOnWrite

Subsection 2. @ 8119b660
   ControlArea: 8119b608  Starting Sector 10 Number Of Sectors 3c00
   Base Pte     e2c2803c  Ptes In subsect      780 Unused Ptes          0
   Flags              35  Sector Offset          0 Protection           3
    ReadOnly CopyOnWrite

Subsection 3. @ 8119b680
   ControlArea: 8119b608  Starting Sector 3C10 Number Of Sectors 5d8

   Base Pte     e2c29e3c  Ptes In subsect       c1 Unused Ptes          0
   Flags              55  Sector Offset          0 Protection           5
    ReadOnly CopyOnWrite

Subsection 4. @ 8119b6a0
   ControlArea: 8119b608  Starting Sector 41E8 Number Of Sectors a8
   Base Pte     e2c2a140  Ptes In subsect       15 Unused Ptes          0
   Flags              55  Sector Offset          0 Protection           5
    ReadOnly CopyOnWrite

Subsection 5. @ 8119b6c0
   ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 0
   Base Pte     e2c2a194  Ptes In subsect        1 Unused Ptes          0
   Flags              55  Sector Offset          0 Protection           5
    ReadOnly CopyOnWrite

Subsection 6  @ 8119b6e0
   ControlArea: 8119b608  Starting Sector 4290 Number Of Sectors 90
   Base Pte     e2c2a198  Ptes In subsect       12 Unused Ptes          0
   Flags              15  Sector Offset          0 Protection           1
    ReadOnly CopyOnWrite
```

7.11 小结

在这一章中，我们讨论了 Windows 内存管理器实现虚拟内存管理的方法。象大多数 32 位操作系统一样，每个进程都允许访问专用的 32 位地址空间来保护一个进程的内存不被其他进程访问，但是允许进程安全和有效地共享内存。一些高级的性能同样可以利用，例如对映射文件的支持和稀疏分配内存的能力。Win32 环境子系统使得应用程序可以通过 Win32 API 获得内存管理器大多数的能力。

内存管理器的实现尽可能地依赖于惰性求值技术以避免执行耗时和不必要的操作，除非这些操作是需要的。它同样具有自调功能，既适合大型的多处理器服务器，也适合单处理器的桌面工作站。

在本章中没有描述的内存管理器的一个方面是它同高速缓存管理器的紧密集成，第 11 章中将谈及，在谈及这方面的内容之前，让我们进一步看看 Windows 2000 的安全机制。

第8章 安 全

在任何多用户访问相同的物理或网络资源环境下，防止未授权访问敏感数据是非常重要的。一个操作系统以及单个的用户必须能够防止文件、内存和配置设置受到不希望的查看和修改。操作系统安全包括明显的机制例如账号、口令和文件保护。同时也包括不太明显的机制例如防止操作系统被破坏，阻止非特权用户的行为（例如，重新启动计算机）和不允许用户程序恶意影响其他用户的程序或操作系统。

在本章，将解释 Microsoft Windows 2000 设计和实现的每一个方面将如何受到提供健壮性安全的严格要求的影响。

8.1 安全级别

美国国家安全中心（NCSC，www.radium.ncsc.mil）建立于 1981 年，作为美国国防部（DoD）国家安全机构（NSA）的分支机构，帮助政府、公司和家庭用户保护存储在计算机的个人和私有的数据。作为此目的的一部分，NCSC 建立了安全级别的范围，列在表 8-1 中，用于表明商用操作系统、网络组件和可信任程序提供的保护程度。1983 年，在美国国防部的 Trusted Computer System Evaluation Criteria（TCSEC）基础上制定了这些安全级别，并且通常被称为"橘皮书"。

表 8-1 TCSEC 级别水平

级别	说 明
A1	验证设计
B3	安全域
B2	结构化保护
B1	标定的访问保护
C2	受控制的访问保护
C1	自由访问保护（已遭废弃）
D	最小保护

TCSEC 标准由"信任等级"级别构成，高级别建立在低级别上并增加了更多的严格的保护和有效性要求。没有操作系统满足 A1，或"验证设计"级别。尽管少数操作系统取得了 B 级别中的某个级别，对普通的操作系统来说，C2 被认为是足够的并且是最高的实用级别。

1995 年 7 月，Windows Nt 的第一版带服务包 3 的 Microsoft Windows NT 3.5（工作站和服务器）取得了 C2 的级别。1999 年 3 月，带服务包 3 的 Microsoft Windows NT 4 获得了英国政府的 Information Technology Security（ITSEC）组织的 E3 级别，等同于美国的 C2 级别。1999 年 11 月，带服务包 6a 的单机和带网络配置的 Microsoft Windows NT 4 都达到了 C2 级别。

评级过程需要几年时间，尽管 Windows 2000 已经提交给了国际安全验证组织，但是仍需要一些时间才能评测完成。然而，Windows 2000 的基础安全体系结构发展比 Windows NT 4 更加健壮，正如 Windows NT 3.5 在 Windows NT 4 的基础上演变。Windows 2000 几乎一定会获得 Windows NT 4 所达到的级别。

要获得 C2 安全级别需要包括些什么呢？以下是关键的要求：

■ 安全登录机制，要求用户能被唯一确定并且仅在以某种方式验证后才能允许访问计算机。

■ 自由访问控制，允许资源的所有者决定谁可以访问资源和可以做什么。所有者能授权允许一个用户或一组用户的各种类型访问。

■ 安全审计，提供了检测和记录安全相关的事件或任何创建、访问和删除系统资源的能力。登录标识符记录了所有用户的标识，使得很容易跟踪执行未被授权行为的用户。

■ 对象重用保护，阻止用户看到另一个用户已经删除的数据，或访问另一个用户曾使用并释放的内存。例如，在一些操作系统中，有可能创建一定长度的新文件并检查文件内容可以看到数据，这些数据恰好放在文件分配的磁盘位置上。这些数据可能是其他用户已经删除的敏感信息。对象重用保护通过在分配给用户之前初始化所有对象，包括文件和内存，从而防止这种潜在的安全漏洞。

Windows NT 也满足 B 级别的两项要求。

■ 信任路径功能，防止用户登录时，特洛木伊马（Trojan horse）程序截获用户名和密码。Windows NT 信任路径功能以 Ctrl + Alt + Delete 登录提示序列形式出现。这种击键序列，众所周知称为“安全提示序列”（secure attention sequence），总是弹出一个登录对话框。所以特洛木伊马程序很容易被识别出来，当 SAS 进入时，特洛木伊马程序的假对话框将被忽略。

■ 信任机制管理，要求支持管理功能的单独帐号。例如，给管理（管理员）的分离帐号、可以用于备份计算机的用户帐号和标准用户。

Windows 2000 通过它的安全子系统和相关组件满足所有这些要求。

通用标准

1996 年 1 月，美国、英国、德国、法国、加拿大和荷兰发表了对于 Information Technology Security Evaluation（CCITSE）的联合发展通用标准。CCITSE 通常称为“Common Criteria”（CC），成为了产品安全评估的公认国际标准。

CC 比 TCSEC 信任级别更灵活并且在结构上更接近 ITSEC，而不是 TCSEC。CC 包含保护配置（Protection Profile，PP）概念用来把安全需求收集在更容易指定和比较的设置里，还包括一组能被 PP 引用的安全要求的安全目标（ST）概念。

Windows 2000 使用 CC 级别而不是 TCSEC 是由于美国政府不再使用 TCSEC 评估产品。在 www.radium.ncsc.mil /tpep /library /ccitse 可以得到更多信息。

8.2 安全系统组件

这里有一些实现 Windows 2000 安全的组件和数据库：

■ 安全引用监视器（SRM）Windows 2000 执行程序（\ Winnt \ System32 \ Ntoskrnl.exe）中

的组件，负责执行对象的安全访问检查操作特权（用户权力），如产生有效的安全审计消息。

■ 本机安全权限子系统（Lsass）运行在 \ Winnt \ System32 \ lsass.exe 映像下的用户模式进程，负责本机系统安全策略（例如：哪个用户允许登录此机、口令策略、授予用户和组的特权、系统安全审计设置、用户鉴别和发送安全审计信息到事件日志。本机安全授权服务（lsasrv - \ Winnt \ System32 \ lsasrv.dll），Lsass 加载的库实现了大部分的功能。

■ Lsass 策略数据库：包括本机安全策略设置的数据库，它存储在注册表 HKLM \ SECURITY 下。它包括的信息有：什么域可以信任用来确认验证登录企图、谁被允许访问系统以及访问方式（交互式、网络和服务登录）、谁被分配什么特权和完成什么样的安全审计。Lsass 策略数据库也存储“密匙”，包括用于高速缓存域登录的登录信息和 Win32 服务的用户帐号登录（关于 Win32 服务更多信息参见第 5 章）。

■ 安全帐号管理器（SAM）服务，负责管理包括本机上定义的用户名和组的数据库的一组子例程。SAM 服务由 \ Winnt \ System32 \ samsrv.dll）实现，运行在 Lsass 进程中。

■ Active Directory：一种目录服务，包含存储域中对象信息的数据库，域由一组计算机和它们相关的安全组组成，做为单独的项管理。Active Directory 存储域中对象的信息，包括用户、组和计算机。口令信息和域用户和组的特权信息存储在 Active Directory 中，它在被指定为域的域控制器的计算机上复制。Active Directory 服务器实现为 \ Winnt \ System32 \ ntdsa.dll，运行在 Lsass 进程中。

■ 验证包运行在 Lsass 进程环境中并实现了 Windows 2000 验证策略的 DLL。验证 DLL 负责检查给定用户与口令是否匹配，如果匹配，向 Lsass 返回有用户安全标识的详细信息的 Lsass 信息。

■ 登录过程（Winlogon）运行 \ Winnt \ System32 \ winlogon.exe 的用户模式进程，负责响应 SAS 和管理交互登录会话。例如，当用户登录时，Winlogon 建立一个用户 shell（用户界面）进程。

■ 图形标识和验证（GINA）运行在 Winlogon 进程中的一个用户模式 DLL，Winlogon 使用它获得用户名和口令或者智能卡 PIN。标准的 GINA 是 \ Winnt \ System32 \ Msgina.dll。

■ 网络登录服务（Netlogon）运行在 Lsass 中的 Win32 服务（\ Winnt \ System32 \ Netlogon.dll），负责响应 Microsoft LAN Manager 2 Windows NT（前 Windows 2000）的网络登录请求。验证象本机登录一样处理，发送给 lsass 验证。Netlogon 中有一个内建的查找服务用于查找域控制器。

■ 内核安全设备驱动程序（KsecDD）实现本机过程调用（LPC）接口的内核模式函数库。其他内核模式安全组件，包括加密文件系统（EFS），使用这个接口与用户模式 Lsass 通信。KSecDD 位于 \ Winnt \ System32 \ Drivers \ Ksecdd.sys。

图 8-1 显示了这些组件中的一些和所管理的数据库之间的关系。

运行在内核模式的 SRM 和运行在用户模式的 Lsass 使用 LPC 机制通信，LPC 在第 3 章已经描述。在系统初始化过程中，SRM 创建一个端口，称为 SeRmCommmandProt，用于 Lsass 连接。当 Lsass 进程开始启动时，它创建一个称为 SeLsaCommmandPort 的 LPC 端口。SRM 连接到这个端口，导致创建了专用通信端口。SRM 建立了一个超过 256 字节信息长的共享内存区域，在连接调用时传送句柄。一旦 SRM 和 Lsass 在系统初始化期间互连，它们不再监听各自的连接端口，因此，后面的用户进程没有办法恶意地成功连接到这些端口中的任何一个——请求永远不会完成。

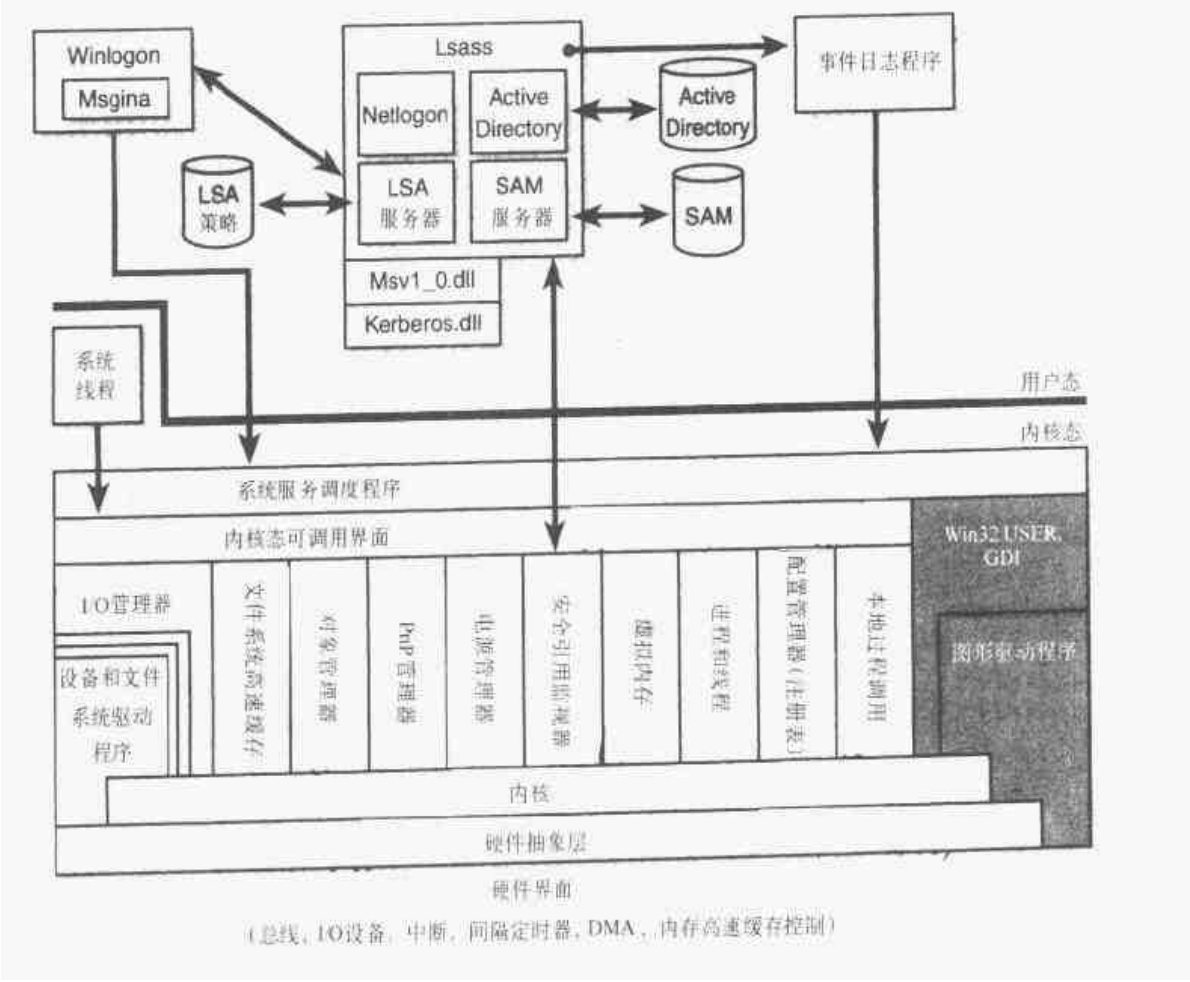


图 8-1 Windows 2000 安全组件

图 8-2 显示了在系统初始化后它们退出时的通信路径。

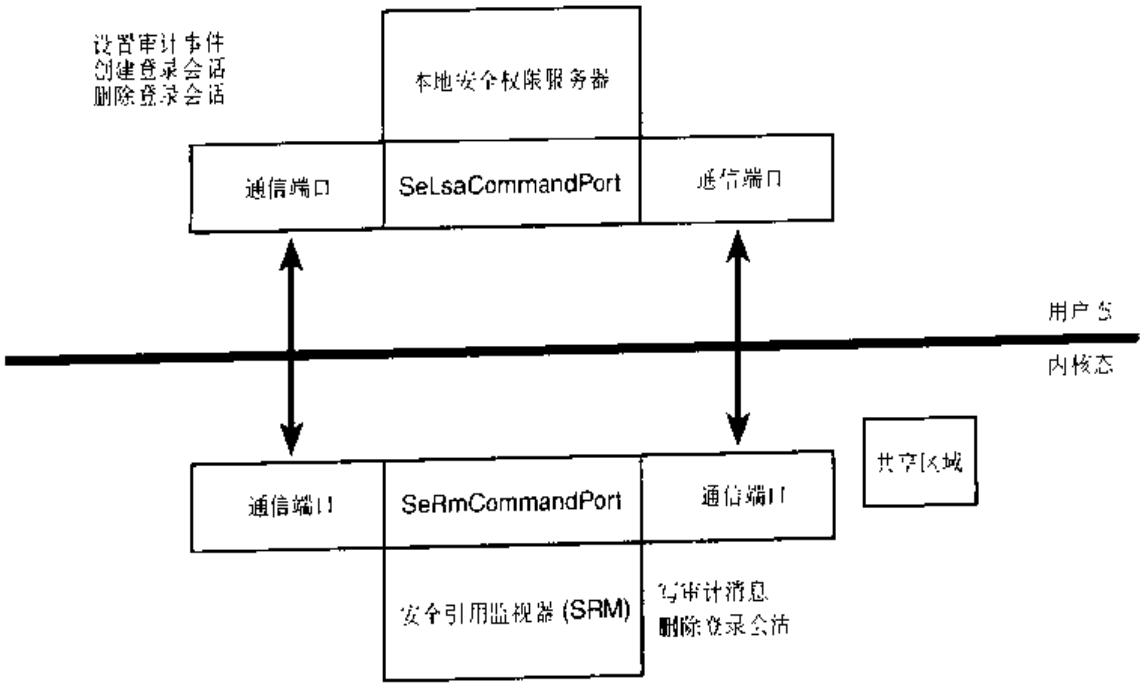


图 8-2 SRM 和 Lsass 间的通信

8.3 保护对象

对象保护和访问统计是自由访问控制和审计的核心。Windows 2000 可以保护的对象包括：文件、设备、邮箱（mailslot）、管道（命名的匿名的）、作业、进程、线程、事件、互斥、信号量、共享内存区域、I/O 完成端口、LPC 端口、可等待计时器、访问令牌、窗口站、桌面、网络共享、服务、注册表键和打印机。

因为向用户模式导出的（因此需要安全验证）系统资源在内核模式以对象的形式实现，Windows 2000 对象管理器在体现对象安全方面起到十分重要的作用（更多关于对象管理器信息参见第 3 章）。为了控制谁能操作对象，安全系统必须首先确认用户标识符。需要保证用户标识符就是 Windows 2000 在访问任何系统资源前要求验证登录的原因。当进程请求一个对象的名柄时，对象管理器和安全系统使用调用程序的安全标识符决定是否给调用程序分配它所请求的对象的名柄，获得句柄进程就可以访问对象。

正如本章后面讨论的，线程可以假设与进程不同的安全环境，这种机制称为“模拟”（impersonation）。当线程模拟时，安全验证机制使用线程的安全环境代替线程所属进程的安全环境。当线程不模拟时，安全验证机制依赖于线程所属进程的安全环境。记住所有同一个进程中的线程共享相同的句柄表非常重要，所以当线程打开一个对象，即使在模拟，进程中的所有线程也可以访问对象。

8.3.1 访问检查

在打开对象时，Windows 2000 安全模块要求线程预先指明想要对对象执行什么样类型的行为。系统根据线程期望的访问（thread’s desired access）完成访问检查，如果是授权访问，分配给线程的进程一个句柄，线程（或进程中其他线程）可以通过句柄对对象执行进一步操作。正如第 3 章说明的，对象管理器记录了在进程句柄表中名柄的访问许可。

当进程使用名字打开一个存在的对象时，这个事件导致对象管理器执行安全访问验证。当通过名字打开对象时，对象管理器在对象管理器名字空间查找指定的对象。如果对象位置不在辅助名字空间中，例如配置管理器的注册表名字空间或文件系统驱动程序的文件系统名字空间，一旦它查找对象，对象管理器就调用内部函数 ObpCreateHandle。正如名字暗示的一样，ObpCreateHandle 在进程句柄表中建立与对象相关联的项。但是，仅当另一个对象管理器函数 ObpIncrementHandleCount 指明线程有访问对象的权限时，ObpCreateHandle 调用执行程序函数 ExCreateHandle 创建句柄。另一个对象管理器函数 ObCheckObjectAccess 实际上执行安全访问检查并返回结果给 ObpIncrementHandleCount。

ObpIncrementHandleCount 传给 ObCheckObjectAccess 的参数有线程打开对象的安全凭证、线程请求对象的访问类型（读、写、删除和其他）和对象指针。ObCheckObjectAccess 首先锁住对象的安全和线程的安全环境。对象安全锁防止系统中其他线程在访问检查中改变对象的安全。线程安全环境锁防止安全验证过程中进程中其他线程或不同进程改变线程的安全标识符。ObCheckObjectAccess 随后调用对象安全方法获得对象的安全设置（关于对象方法描述参见第 3 章）。调用安全方法可能调用不同执行程序组件的函数。但是，许多执行程序对象依赖系统默

认的安全管理支持。

当定义对象的执行程序组件不想覆盖 SRM 的默认安全策略时，它将对象类型标明为有默认安全。任何时候 SRM 调用对象安全方法，首先检查对象是否有默认的安全，有默认安全的对象存储安全信息在对象头（object header），并且它的安全方法是 SetDefaultObjectMethod，一个不依靠默认安全的对象必须管理自己的安全信息和提供指定的安全方法。依赖缺省安全的对象包括互斥、事件和信号量。文件对象是一个覆盖默认安全对象的例子。定义文件对象类型的 I/O 管理器有文件系统驱动程序管理，驻留在它上面的文件为它的文件管理安全（或者选择不实现）。这样，当系统查询代表在 NTFS 卷上文件的文件对象的安全时，I/O 管理器文件对象安全方法使用 NTFS 文件系统驱动程序获取文件的安全。注意，当文件打开时 ObCheckObjectAccess 没有执行，这是由于文件驻留在辅助名字空间；仅当线程显式地查询或设置文件上的安全时，系统才调用文件对象的安全方法（例如，通过 Win32 SetFileSecurity 或者 GetFileSecurity 函数）

在得到对象安全信息后，ObCheckObjectAccess 调用 SRM 函数 SeAccessCheck。SeAccessCheck 是 Windows 2000 安全模型核心的函数之一。SeAccessCheck 接受的输入参数有对象安全信息、由 ObCheckObjectAccess 捕获的线程安全标识符和线程请求的访问。依据是否允许线程访问所请求的对象，SeAccessCheck 返回 True 或 False 。

另一个引起对象管理器执行访问验证的事件是进程使用一个存在的句柄引用对象。这样的引用常常间接发生，如当进程调用 Win32 API 操作对象并传递对象句柄时。例如，打开文件的线程能请求访问对象，此对象允许进程从文件中读。如果线程由它的安全环境及文件的安全设置指明允许以这种方式访问对象，那么对象管理器在进程句柄表中创建句柄代表文件。对象管理器把进程通过句柄授予的访问和句柄存储在一起。

随后，线程可以试着使用 Win32 函数 WriteFile 写入文件，把文件句柄做为参数通过 Ntdll.dll 的 WriteFile 调用系统服务 NtWriteFile，使用对象管理器函数 ObReferenceObjectByHandle 从句柄获得文件对象指针。ObReferenceObjectByHandle 接受调用程序从对象作为参数想获得的访问。在发现进程句柄表中的句柄项后，ObReferenceObjectByHandle 比较请求的访问和文件打开时允许的访问。这种情况下 ObReferenceObjectByHandle 将显示写操作失败，这是因为调用程序在文件打开时没有获得写访问。

Windows 2000 安全函数也能允许应用程序定义自己专用对象和调用 SRM 服务来体现 Windows 2000 的这些对象的安全模型。许多被对象管理器和其他执行程序组件用来保护它们对象的内核模式函数作为 Win32 用户模式 API 表现出来。例如 SeAccessCheck 的用户模式等价体是 AccessCheck。因此，Win32 应用程序可以利用这些安全模型的灵活性并且透明地集成 Windows 2000 中存在的验证和管理接口。

SRM 的安全模型本质是一个等式，包括三个输入：线程的安全标识符、线程想要的对对象的访问和对象的安全设置。输出是“Yes”或“No”并表明是否安全模型同意线程期望的访问，以下部分描述这些输入的更多细节并接着记录模型的访问验证算法。

8.3.2 安全标识符

Windows 2000 使用安全标识符（SID）代替使用名字（它可能唯一也可能不唯一）来标识

在系统中能执行动作的实体。用户有 SID，本地组和域组、本地机、域以及域成员也有。SID 是一个变长数值，由 SID 结构修改号、48 位标识符权限值和 32 位子权限变量号或相关标识符（RID）值组成。权限值标识发布 SID 的代理，这个代理一般是 Windows 2000 本机系统和域。子权限值标识了相对于发布权限的信任者，并且 RID 是 Windows 2000 在通用基础 SID 上创建独一无二的 SID 的简单方法，由于 SID 较长并且 Windows 2000 注意在每个 SID 内产生真正的随机数，实际上在世界任何地方 Windows 2000 都不可能在机器和域上产生两次相同的 SID。

当以文本显示时，每个 SID 有 S 前缀，并且破折号分开了变化的各部分：S—1—5—21—1463437245—1124812800—863842198－1128。

在这个 SID 中，修改号是 1，标识符权限值是 5（Windows 2000 安全权限）和 4 个子权限值加 1 个 RID（1128）组成剩下的 SID。这个 SID 是随机 SID，但域上的本地计算机可能有相同修改号，标识符权限值和子权限值号的 SID。

实验：使用 GetSID 检看帐号 SID

运行 GetSID 应用程序可以很容易看到代表你正使用帐号的 SID，位置在 Windows 2000 资源包中，以下是接口：

```
C:\>getsid
Usage: getsid \\server1 account \\server2 account
```

GetSID 是作为检测两个域控制器的帐号不一致性的工具。它接收两个用户名，每一个相对于一个服务器，并且从为它们指定的服务器中获取各自的 SID，然后比较 SID 并显示是否匹配，同时显示文本表示，尽管 GetID 事实上要求指定两个帐号名，仍然可以通过指定两个名字为相同帐号和服务器的方法简单得到一个帐号的 SID。这里有一个得到单—SID 的例子。

```
C:\>getsid \\w2kpro administrator \\w2kpro administrator
The SID for account W2KPRO\administrator matches account
  W2KPRO\administrator
The SID for account W2KPRO\administrator is
S-1-5-21-1123561945-484763869-1957994488-500
The SID for account W2KPRO\administrator is
S-1-5-21-1123561945-484763869 1957994488-500
```

当安装 Windows 2000 时，Windows 2000 安装程序发布一个计算机 SID。Windows 2000 将 SID 分配给计算机本机帐号。每个本机帐号 SID 都基于源计算机的 SID 并且结尾有 RID。用户和组帐号的 RIDS 从 1000 开始并且对每个新用户和组加 1。同样地，Windows 2000 发布一个 SID 给每一个新建的 Winhdows 2000 域。Windows 2000 发布的新域账号 SID 也基于域 SID，并且附加有 RID（从 1000 开始并且每个新用户和组增加 1）。1028 的 RID 显示 SID 是域发布的第 29 个 SID。

Windows 2000 发布的 SID 由计算机或域 SID 组成，SID 带有为预定义帐号和组而预定义的 RID。例如，管理员帐号的 RID 是 500，客人帐号 RID 是 501。再如，计算机本机管理者帐号有一个计算机 SID，用 500 的 RID 附加在后面：

S－1—5—13124455—12541255—6235125—500

Windows 2000 也定义了代表组的内建的本机和域 SID 号，例如，代表每个和任意帐号的

SID 是 Everyone 或 World，SIP：S—1—1—0，另一个 SID 能代表组的例子是网络组，它代表可以从网络登录机器的用户组。网络组 SID 是 S—1—5—2。表 8-2 是从 Platform SDK 文档中复制下来的，显示众所周知的基本 SID 的数值和使用。

表 8-2 广为人知的 SID

SID	Group	使 用
S—1—1—0	Everyone	包括所有用户的组
S—1—2—0	Local	登录本机（物理）连接系统终端的用户
S—1—3—0	Creator Owner ID	由创建新对象的用户安全标识符代替的安全标识符，这个 SID 用于可继承访问控制项（ACE）
S—1—3—1	Creator Group ID	由创建新对象的用户基本组 SID 代替的安全标识符，使用可继承 ACE 中的 SID

8.3.3 令牌

SRM 使用称为“令牌”（或访问令牌）的对象标识进程或线程的安全环境。安全环境由描述特权、账号和与进程或线程相关的组信息组成。在登录过程中（在本章结尾描述）中，Winlogon 创建一个代表登录用户的初始令牌并附接到用户登录 shell 进程上，所有用户执行的程序继承初始令牌的一个拷贝。也可以使用 Win32 LogonUser 函数产生令牌，可以通过传送令牌给 Win32 CreateProcessAsUser 函数用此令牌创建进程，此进程运行在通过 LogonUser 函数登录的用户安全环境中。由于不同用户帐号有不同特权集合和相关的组账号，令牌大小是变化的。但是，所有令牌包括的相同信息在图 8-3 中显示。

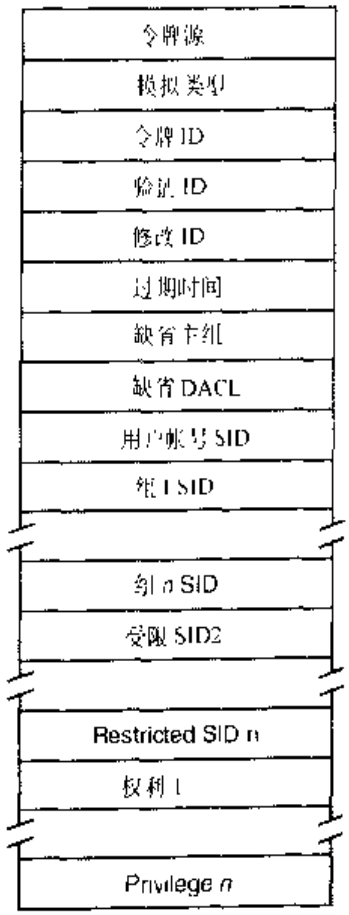


图 8-3 访问令牌

Windows 2000 的安全机制使用两个令牌组件决定令牌的线程或进程能做什么，一个组件由令牌用户帐号 SID 和组 SID 字段组成。SRM 用 SID 决定进程或线程是否可以获得对安全对象所请求的访问，例如 NTFS 文件令牌中组 SID 指明用户账号属于哪个组。当服务器应用程序执行客户请求的行为时，服务器应用程序禁止特定组以限制令牌的信任。禁止组几乎产生组不在令牌中的相同的效果。（禁止 SID 做为安全访问检查的一部分，本章后面描述）。

令牌中决定令牌的线程或进程可以做什么的第 2 个组件是权限数组。令牌权限数组是与令牌关联的权限列表。一个权限例子是关联令牌的进程或线程关机的权利。大概有两打令牌权限，最常用的几个列在表 8-3 中。

当进程或线程使用令牌创建对象时，令牌的默认基本组字段和默认自由访问控制列表

(DACL) 字段是 Windows 2000 应用于对象的安全属性。通过在令牌中包括安全信息，Windows 2000 使进程或线程能很方便地建立带标准安全属性的对象，因为进程或线程不必给创建的每个对象都要求自由安全信息。

表 8-3 一些常用权限

权限名	权限使用
SeBackup	备份时忽略安全检查
SeDebug	要求调试进程
SeShutdown	要求关闭本机系统
SeTakeOwnership	要求获得对象的所有权而不被允许自由访问

每个令牌类型区分基本令牌（标识进程安全环境的令牌）和模拟令牌（令牌线程临时采用不同的安全环境，通常是其他用户的）。模拟令牌带有模拟水平指明在令牌中什么类型的模拟是活动的。稍后将详细描述模拟。

令牌中剩下的字段提供多方面的信息。令牌源字段包括创建令牌的实体的简短文字描述，程序想知道令牌产生于哪里就使用令牌源区分源头，例如 Windows 2000 会话管理器、网络文件服务器、远程过程调用（RPC）服务器。令牌标识符是 SRM 建立令牌时分配给它的本机唯一标识符（LUID）。Windows 2000 执行程序维护执行程序 LUID，它是用于给每个令牌分配唯一数字标识符的计数器。

令牌验证 ID（authentication ID）是另一种 LUID。令牌创建者分配令牌的验证 ID。Lsass 是系统中典型的唯一令牌创建者，并且 Lsass 从执行程序 LUID 获得 LUID。然后，Lsass 给所有从初始登录令牌继承的令牌拷贝验证 ID。程序可以获得令牌的验证 ID 查看令牌是否与程序检查的其他令牌属于同一个登录会话。

每次令牌特征改变时，执行程序 LUID 刷新修改的 ID。应用程序检查修改的 ID 可以发现自上次使用以来安全环境所发生的改变。

令牌包含期限时间字段，自 Windows NT 3.1 以后的 Windows NT 技术提供但没有使用这个字段，Windows 2000 未来的版本考虑在过期之前一段时间有效的令牌。考虑到系统管理员设置用户的帐号期限，如果当前用户登录并且登录超过帐号期限，系统仍允许用户继续访问资源。仅有强制用户退出机器才能阻止用户访问资源。如果 Windows 2000 支持令牌期限，系统将阻止用户的令牌过期后打开资源。

实验：使用内核调试器观察访问令牌

内核调试器命令！tokenfields 显示内部令牌对象的格式。尽管这个结构与 Win32 API 函数返回的用户令牌结构不同，但字段是相似的。关于令牌更多信息，参见平台 SDK 文档的描述。

以下输出是从内核调试器命令！tokenfields 得到的：

可以使用命令！token 查看进程令牌，可以在！process 命令的输出中找到令牌地址，如下所示：

```
kd> !tokenfields
!tokenfields
 TOKEN structure offsets:
    TokenSource:          0x0
    AuthenticationId:     0x18
    ExpirationTime:       0x28
    ModifiedId:           0x30
    UserAndGroupCount:    0x3c
    PrivilegeCount:       0x44
    VariableLength:       0x48
    DynamicCharged:       0x4c
    DynamicAvailable:     0x50
    DefaultOwnerIndex:    0x54
    DefaultDacl:          0x6c
    TokenType:            0x70
    ImpersonationLevel:   0x74
    TokenFlags:           0x78
    TokenInUse:           0x79
    ProxyData:            0x7c
    AuditData:            0x80
    VariablePart:         0x84
```

```
kd> !process 380 1
!process 380 1
Searching for Process with Cid == 380
PROCESS ff8027a0 SessionId: 0  Cid: 0380  Peb: 7ffdf000  ParentCid: 0124
  DirBase: 06433000  ObjectTable: ff7e0b68  TableSize:  23.
  Image: cmd.exe
  VadRoot 84c30568 Clone 0 Private 77. Modified 0. Locked 0.
  DeviceMap 818a3368
  Token                             e22bc730
  ElapsedTime                       14:22:56.0536
  UserTime                          0:00:00.0040
  KernelTime                        0:00:00.0100
  QuotaPoolUsage[PagedPool]         13628
  QuotaPoolUsage[NonPagedPool]      1616
  Working Set Sizes (now,min,max)  (261, 50, 345)(1044KB, 200KB, 1380KB)
  PeakWorkingSetSize                262
  VirtualSize                       11 Mb
  PeakVirtualSize                   11 Mb
  PageFaultCount                    313
  MemoryPriority                    FOREGROUND
  BasePriority                      8
  CommitCharge                      86
```

```
kd> !token e22bc730
!token e22bc730
TOKEN e22bc730  Flags: 9  Source User32  b"  AuthentId (0, ae6d)
    Type:                 Primary (IN USE)
    Token ID:             1803a
    ParentToken ID:       0
```

```
Modified ID:            (0, 12bf5)
TokenFlags:             0x9
SidCount:               9
Sids:                   e22bc880
RestrictedSidCount:     0
RestrictedSids:         0
PrivilegeCount:         17
Privileges:             e22bc7b4
```

8.3.4 模拟

模拟是 Windows 2000 常用于安全模型的有力的特征。Windows 2000 也在客户/服务器程序模式中使用模拟。例如，服务器应用程序可以输出像文件、打印和数据库这样的资源。等待访问资源的客户向服务器发出请求。当服务器接到请求时，必须确信允许客户机可以在资源上执行想要的操作。例如，远程机器上的用户试图删除 NTFS 共享上的文件，服务器输出共享时必须确定是否同意用户删除文件。确定用户是否有权限的过程是让服务器查询用户的帐号、组 SID 和扫描文件安全属性。这个过程对程序来说是繁锁的、易出错的和不能透明地支持允许新的安全特性。因此，Windows 2000 提供模拟服务来简化服务器的工作。

模拟让服务器通知 SRM，服务器临时采用发送资源请求的客户的安全配置（profile）。服务器代表客户访问资源，并且 SRM 执行访问确认。通常，服务器比客户访问更多的资源并且在模拟时失去了一些安全凭证。但是，相反也正确，服务器在模拟时获得了安全凭证。

服务器仅在发出模拟请求的线程中模拟客户。线程控制数据结构包含对于模拟令牌的可选项。但是，代表线程实际安全凭证的线程基本令牌总是在进程控制结构中可访问的。

Windows 2000 通过几种机制使模拟可用。如果服务器通过名字管道与客户通信，服务器使用 ImpersonateNamedPipeClient Win32API 函数告诉 SRM 它想模拟管道另一端的用户。如果服务器通过动态数据交换（DDE）或 RPC 与客户通信，可以用 DdeImpersonateClient 和 RpcImpersonateClient 完成相似的模拟，线程可以使用 ImpersonateSelf 函数简单拷贝它的进程令牌从而创建模拟令牌。线程可以改变它的模拟令牌，例如来禁止 SID 和权限。最后，Security Support Provider Interface（SSPI）包能使用 ImpersonateSecurityContext 模拟它的客户。SSPI 实现网络安全模型诸如 LAN Manger2 或 Kerberos。

在服务器线程完成它的任务后，恢复它的基本安全配置。这种模拟形式对于执行客户指定行为的要求是方便的，它不利的地方是不能在客户环境中执行整个程序，另外，除非文件或打印机共享支持空会话（空会话是匿名登录的结果之一），模拟令牌不能访问网络共享的文件或打印机。

如果整个程序必须在客户安全环境中执行和访问网络资源，客户必须登录系统。LogonUser Win32API 函数能够做到。LogonUser 使用帐号名、口令、域或计算机名、登录类型（例如交互式、批处理或服务）和登录提供者作为输入，返回基本令牌。服务器线程采用令牌作为模拟令牌或服务器开始一个将客户凭证做为基本令牌的程序。从安全观点来看，LogonUser 创建的在交互式登录会话中运行程序的进程看起来象用户通过交互登录到机器上的程序。

Windows 2000 提供客户安全环境模拟的第 2 种方法与使用 logonUser 相似，通过获得客户的访问令牌，复制它并且使用它作为传送给 CreateProcessAsUser 命令的基本令牌。使用 LogonUser 和 CreateProcessAsUser 的缺点是服务器必须获得用户账号和口令。如果服务器在网络上传送此信息，服务器必须安全地加密它，防止监测网络的恶意用户捕获它。

为了防止模拟的误用，Windows 2000 不允许在没有客户同意的情况下服务器执行模拟。当连接服务器时，客户进程可以通过指定安全服务质量（Security Quality of Service，SQOS）限制服务器进程能执行的模拟级别。进程可以指定 SECURITY-ANONYMOUS、SECURITY-IDENTIFICATION 和 SECURITY-IMPERSONATION 和 SECURITY-DELEGATION 做为 Win32 CreateFile 函数的标志。根据客户安全环境每个级别可以使服务器执行不同操作类型：

■ SecurityAnonymous 是最严格的模拟级别——服务器不能模拟或标识客户。

■ SecurityIdentifiation 级别让服务器获得客户的标识符（SID）和客户权限，但服务器不能模拟客户，

■ Security Impersonation 级别让服务器标识和模拟本机系统的客户。

■ Security Delegation 是允许的最自由模拟级别，它让服务器模拟本机和远程的客户，Windows NT 4 和更早版本不完全支持 Security Delegation 水平的模拟。

如果客户没有设置模拟级别，Windows 2000 默认选择 Security Impersonation 级别。CreateFile 函数也接受 SEC-EFFECTIVE-ONLY 和 SECURITY-CONTEXT-TRACKING 作为模拟设置的限定词：

■ 当服务器模拟时，SECURITY-EFFECTIVE-ONLY 防止服务器访问客户机的权限或组。

■ SECURITY-CONTEXT-TRACKING 指定客户机对安全环境做的改变都反应在模拟它的服务器中，如果没有指定这选项，服务器在模拟时采用客户环境并不做任何改变，仅当客户机与服务器进程在同一系统上时，这个选项被授予。

8.3.5 受限令牌

Windows 2000 介绍了新的令牌类型，称为“受限令牌”（restricted tokens）。受限令牌使用 CreateRestrictedToken 函数从基本或模拟令牌建立。受限令牌是派生出它的令牌的拷贝，并有以下可能的修改：

■ 权限从令牌权限数组中删除。

■ 令牌中 SID 标记为 deny－only。

■ 令牌中 SID 标记为 restricted。

简要描述一下 deny－only 和 restricted SID 的行为，当应用程序想模拟降低了的安全水平的客户机时受限令牌十分有用，这主要出于运行不信任的代码的安全考虑，例如，受限令牌删除重启系统权限防止在受限令牌安全环境中执行的代码重新启动系统。

8.3.6 安全描述符和访问控制

确定用户凭证的令牌仅仅是对象安全等式的一部分，等式另一个部分是与对象关联的安全信息，指明谁可以在对象上执行什么行为。这个信息的结构称为“安全描述符”，安全描述符由以下属性组成：

■ 修改号 用于创建描述符的 SRM 安全模型版本。

■ 标志 定义描述符行为或特征的可选限定词，例子有 SE-DACL-PROTECTED 标志，它防止描述符从另一个对象继承安全设置。

■ 所有者 SID 所有者安全 SID。

■ 组 SID 对象的基本组安全 ID（仅由 POSIX）使用。

■ 自由访问控制表（DACL）指明谁可以对对象有什么访问。

■ 系统访问控制表（SACL）指明哪个用户的哪个操作应记录在安全审计日志中。

访问控制表（ACL）由头和零项或更多的访问安全项（ACE）结构组成。有两种类型的 ACL：DACL 和 SACL。在 DACL 中，每个 ACE 包括一个 SID 和一个访问掩码（可以简单解释为标志集合）。四种类型的 ACE 可能出现在 DACL 中：允许访问、拒绝访问、允许的对象和拒绝的对象。正如你期望的一样，允许访问的 ACE 授予用户访问权，拒绝访问的 ACE 否认在访问掩码中指定的访问权限。

允许对象和允许访问之间和拒绝对象和拒绝访问之间的不同之处在于对象类型仅用于 Active Directory 中。这些类型的 ACE 有一个 GUID（全局唯一标识符）字段，指明 ACE 仅仅应用于特定的对象或子对象（有 GUID 标识符的那些）。除此以外，当子对象在应用 ACE 的 Active Directory 容器中创建时，另一个可选的 GUID 显示什么样类型的子对象会继承 ACE。（GUID 是一个保证完全唯一的 128 位标识符）。

单个 ACE 允许的访问权限累计形成 ACL 允许的访问权限集合。如果在安全描述符中没有 DACL（空 DALL），那么每个用户都拥有对对象有完整访问。如果 DACL 空（0 个 ACEs），没有用户访问对象。

用在 DACL 中的 ACE 有一组标志控制和指定继承相关的 ACE 特征，一些对象命字空间有容器对象和叶对象（或只是对象）。容器可以包容其他容器对象和叶对象，这些都是它的子对象。容器的例子有文件系统名字空间的目录和注册表名字空间的键。ACE 中的某些标志控制 ACE 如何传播给与 ACE 关联的容器的子对象。表 8-4 复制了 Platform SDK 的一部分，列出了 ACE 标志的继承准则。

表 8-4 ACE 标志的继承准则

标 志	继承准则
CONTAINER-INHERIT-ACE	子对象是容器，例如目录将 ACE 继承为有效 ACE，继承的 ACE 是可继承的，除非 NO-PROPAGATE-INHERIT-ACE 位标志也被设置
INHERIT-ONLY-ACE	表明一个不控制对附接对象的访问的只继承 ACE
INHERIT-ACE	表明 ACE 被继承，当它传播一个可继承 ACE 给子对象时，系统设置此位
NO-RROPAGATE-INHERIT-ACE	如果子对象继承 ACE，系统清除在继承 ACE 中的 OBJECT-INHERIT-ACE 和 CONTAENER- INHERIT-ACE 标志。这个行为防止对象后续代继承 ACE
OBJECT-INHERIT-ACE	非容器子对象将 ACE 继承为有效 ACE，对于是容器的子对象、ACE 作为只继承的 ACE 继承，除非也设置了 NO－PROPAGATE－INHERIT-ACE 标志

SACL 包括两种类型的 ACE：系统审计 ACE 和系统审计对象 ACE。这些 ACE 指定哪些操作

在特定的用户和组应该审计的对象上进行。审计信息保存在系统审计日志中。成功和不成功的企图都被审计。象 DACL 对象相关的 ACE 的表兄一样，系统审计对象 ACE 指定显示 ACE 应用于对象和子对象类型的 GUID 以及可选的控制 ACE 传播到特定子对象类型的 GUID。如果 SACL 为空，对象没有审计发生。（本章后面描述安全审计）应用于 DACL ACE 的继承标志也应用于系统审计和系统审计对象 ACE。

图 8-4 是一个文件对象和它的 DACL 的简图。

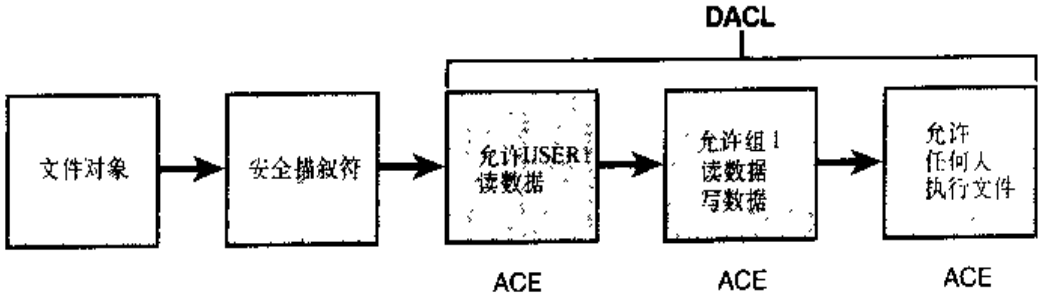


图 8-4 自由访问控制表（DACL）

如图 8-4 所示，第一个 ACE 允许 USER1 读文件，第二个 ACE 允许 TEAM1 组的成员读和写访问文件、第三个 ACE 允许所有其他用户（Everyone）执行访问。

1. ACL 分配

为了确定给新对象分配什么样的 DACL，安全系统使用以下四个分配规则的第一个可应用的规则：

1）如果在创建对象时，调用程序显式地提供安全描述符，安全系统将把它应用于对象。如果对象有名字并且驻留在一个容器对象中（例如，在 \ BaseNamedSecurity 对象管理器名字空间目录中的命名事件对象），系统把任何可继承的 ACE（可能从对象容器传播的 ACE）合并到 DACL，除非安全描述符具有防止继承的 SE - DACL - PROTECTED 标志。

2）如果调用程序没有提供安全描述符并且对象有名字，安全系统在新对象名字存储的容器中寻找安全描述符。一些对象目录的 ACE 可能标记为可继承，这意味着这些 ACE 可以应用于在对象目录中创建的新对象。如果存在任何可继承的 ACE，安全系统把它们全部合并到 ACL，这个 ACL 附接到新对象（Separate 标志显示 ACE 应该仅仅被容器对象继承而不是非容器对象）。

3）如果没有指定安全描述符并且对象没有继承任何 ACE，安全系统从调用程序访问令牌中获取默认 DACL 并且应用于新对象 Windows 2000 的几个子系统将 DACL 硬编码到对象创建过程中（例如服务、LSA 和 SAM 对象）。

4）如果没有指定的描述符、继承的 ACE 和默认的 DACL，安全系统创建没有 DACL 的对象，允许每个用户（所有用户和组）完全访问对象，当令牌包含空的默认 DACL 时，这个规则与第 3 个规则一样。

当给对象分配 SACL 时，系统使用的规则与用于 DACL 的规则相似，但有两个例外。第一个不同是继承的系统审计 ACE 不传播给标记为 SE - SACL - PROTECTED 的安全描述符对象（与 SE - DACL - PROTECTED 标志相似，它保护 DACL）。第二个不同是，如果没有指明安全审计

ACE 并且没有继承的 SACL，没有 SACL 应用于对象。这个行为不同于使用默认的 DACL，这是由于令牌没有默认 SACL。

当包含可继承 ACE 的安全描述符应用于容器时，系统自动传播可继承的 ACES 给子对象的安全描述符（注意，如果描述符 SE－DACL－PROTECTED 标志被设置，安全描述符的 DACL 不接收继承的 DACL ACE，并且如果描述符 SE－SACL－PROTECTED 标志被设置，它的 SACL 不继承 SACL ACE）。可继承 ACE 合并到已存在的子对象的安全描述符的顺序是任何显式地应用于 ACL 的 ACE 保留在对象继承的 ACE 之前。系统使用以下规则传播可继承的ACE。

■ 如果没有 DACL 的子对象继承 ACE，那么带有 DACL 的子对象仅包含继承 ACE。

■ 如果 DACL 为空的子对象继承 ACE，那么子对象的 DACL 包含一个继承的 ACE。

■ 对于仅在 Active Directory 中存在的对象，如果从父对象中删除一个可继承的 ACE，自动的继承机制将删除子对象继承的所有 ACE 副本。

■ 对于仅在 Active Directory 中存在的对象，如果自动继承导致从子对象中删除了所有 ACE，子对象将带有一个空的 DACL 而不是没有 DACL。

稍后你将发现，ACL 中 ACE 的顺序是 Windows 2000 安全模型的一个重要特征。

注意 因为文件系统的目录和注册表键不支持继承，所以 Windows Explorer 和 Regedt32 手工传播你为目录及其内容，或者是整个注册表子键指定的安全设置。

2. 决定访问

有两种算法用来决定对一个对象的访问：

■ 一是确定该对象所允许的最大访问量，一种使用 Win32GetEffectiveRightFromAcl 输出到用户模式的形式。

■ 一个是确定所期望的特定访问是否被允许，这可以利用 Win32AccessCheck 或 AccessCheckByType 函数实现。

第一个算法按以下步骤检查 DACL 中的项：

1）如果对象没有 DACL（一个空的 DACL），那么该对象将不会被保护，安全系统将授予所有用户访问权限。

2）如果调用程序具有 take－ownership 权限，安全系统将在检查 DACL 之前授予 write－owner 访问权限（稍后解释 take－ownership 和 write－owner 访问权限）。

3）如果调用程序是对象的所有者，那么将被授予 read－control 和 write－DACL 的访问权限。

4）对于每个被拒绝访问的 ACE，如果其中包含与调用程序的访问令牌相匹配的 SID，则 ACE 的访问掩码将从授予访问掩码中删除。

5）对于每个允许访问的 ACE，如果其中包含与调用程序的访问令牌相匹配的 SID，则 ACE 的访问掩码将添加到正在计算的授予访问掩码中，除非该访问已经被拒绝。

在检查完 DACL 中的所有项之后，经计算的授予访问掩码将返回给调用程序，以此作为允许对该对象访问的最大权限。该掩码代表总的访问类型集，调用程序在打开该对象时将能够成功地请求这些访问类型。

上面的描述只适应于该算法的内核模式形式。由 GetEffectiveRightsFromAcl 实现的 Win32 版

的差别在于：它不执行步骤2，并且包含的是单个用户或组SID，而不是访问令牌。

第二个算法依据调用程序的访问令牌来确定一个特定的访问请求是否可以被授予。Win32 API中每个处理安全对象的打开函数都有一个指定期望的访问掩码的参数。该参数是安全方程中最后的组成部分。要确定调用程序是否具有访问权限，请执行下列步骤：

1）如果对象没有DACL（空DACL），对象将不会被保护，安全系统会授予所希望的访问权限。

2）如果调用程序具有take-ownership权限，安全系统会授予其write-owner访问权限，然后检查DACL。然而如果write-owner访问是具有take-ownership调用程序所请求的唯一访问的话，那么安全系统授予它这种访问而不再检查DACL。

3）如果调用程序是对象的所有者，就将被授予read-control和write-DACL的访问权限。如果调用程序仅请求了这些访问权限，则这些权限将被授予而不检查DACL。

4）DACL中的每个ACE都会被从头至尾检查一遍。ACE如果满足以下条件之一将被处理：

a.ACE中的SID与调用程序访问令牌中的“enabled”SID匹配（不管那是主要SID，还是组SID）。

b.ACE是一个允许访问的ACE，并且ACE中的SID与调用程序访问令牌中的SID匹配，而该SID不是唯一拒绝的类型。

c.它是第二个通过受限SID检验的描述体，并且ACE中的SID与调用程序访问令牌中的受限SID匹配。

如果是一个access-allowed的ACE，那么在该ACE中的访问掩码中所请求的权限将被授予；如果已经授予了所有请求的权力，那么该访问检验就成功通过。如果它是一个被拒绝的访问的ACE，且所请求的所有访问权力都在拒绝访问权力范围内，那么对对象的访问会被拒绝。

5）如果已经到了DACL的尽头而某些请求的权力仍然未被授予，那么该访问将被拒绝。

6）如果所有访问都被授予，但调用程序的访问令牌至少有一个受限的SID，系统将重新扫描DACL的ACE，查找访问掩码与用户请求的访问相匹配的ACE，以及该ACE的与调用程序有限SID相匹配的SID。只有两次扫描都授予的请求访问权限才是授予给用户访问该对象的权力。

两种访问确认算法都依赖于允许和拒绝ACE的相对顺序。考虑一个对象具有两个ACE，其中一个ACE指定某个用户可以完全访问该对象，而另一个ACE拒绝用户访问。如果允许的ACE在拒绝的ACE之前，用户仍然能够获得对该对象的完全访问，但是如果顺序反过来，用户就不能获得任何访问该对象的权力。

较老的Win32函数，如AddAccessAllowedAce，将ACE添加到DACL的尾部，这通常不是所期望的行为，因此大多数在Windwos 2000之前的Win32应用程序被强制手工构造DACL，且在列表的前端放置拒绝ACE。新的Windows 2000函数，如SetSecurityInfo和SetNamedSecurityInfo，将拒绝的ACE以优先的顺序放置在允许的ACE之前。注意你可以利用安全编辑器对话框编辑NTFS文件的授权与注册表键，例如将所有拒绝的ACE放到列表的前面构造安全描述符。SetSecurityInfo和SetNamedSecurityInfo还将ACE继承规则应用到安全描述符。

图 8-5 显示了访问确认的例子，说明了 ACE 顺序的重要性。在该例子中，希望打开文件的用户的访问被拒绝，虽然该对象的 DACL 中的 ACE 授予了该访问权力（通过用户在 Writer 组中的成员关系），因为拒绝用户访问的 ACE 在授权访问的 ACE 之前。

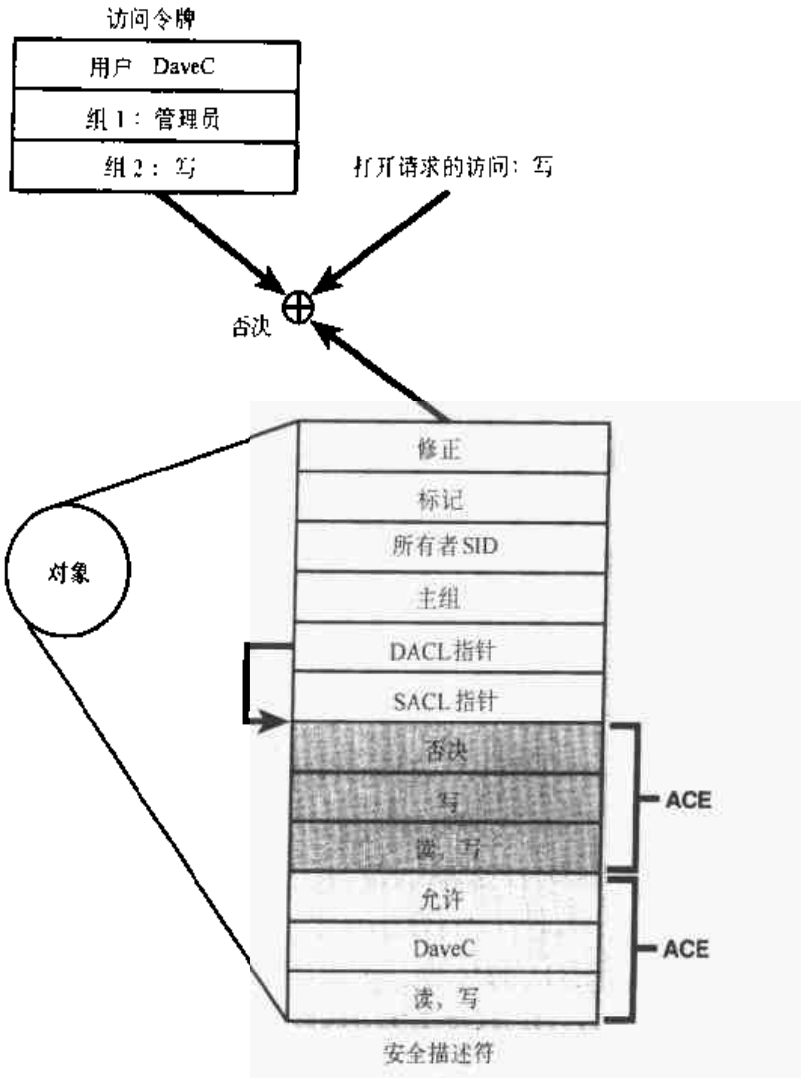


图 8-5 访问确认例子

就象在前面所陈述的那样，由于在进程每次使用句柄时都处理 DACL 可能使安全系统缺乏效率，所以 SRM 仅在打开句柄时进行这种处理，而不是在每次使用句柄的时候。这样，一旦进程成功地打开某个句柄，安全系统就不能取消所授予的访问权限，即使对象的 DACL 改变了。而且需要记住的是由于内核模式代码使用指针而不是句柄访问对象，在操作系统使用对象时不会进行访问检查。换句话说，在安全性方面，Windows 2000 执行程序相信它自己。

对象的所有者总是被授予该对象的写 DACL 访问权限的事实意味着用户访问它们自己拥有的对象从来不会受阻。由于某些原因，如果对象带有空的 DACL（无访问），所有者将仍能够使用写 DACL 权限打开该对象，然后将利用所期望的权限申请新的 DACL。

对于帐号，take－ownership 权限是一个相当强大的工具，例如给 Administrator 管理员帐号分配的该权限。利用该权限可以访问系统中的所有对象。假定一个对象被另一个用户所拥有，

并显式拒绝 Administrator 帐户的所有访问。利用 take - ownership 权限，管理器能够使用 write - owner 权限打开该对象，并将所有者改为 Adminstrator。然后管理器能够关闭该对象，并使用写 DACL 权限再次打开它，修改 DACL，授予 Adminstrator 帐户完全访问权限。

8.4 安全审计

作为访问检查的结果对象管理器可以生成审计事件，而用户应用程序可以使用 Win32 函数直接生成这些审计事件。内核模式代码通常只允许生成一个审计事件，有两个权限，SeSecurityPrivilege 和 SeAuditPrivilege，与审计有关。一个进程必须具有 SeSecurityPrivilege 权限才能够管理安全 Event Log，并查看或设置对象的 SACL。然而，调用审计系统服务的进程必须具有 SeAuditPrivilege 权限才能成功地生成审计记录。

本机系统的审计规则控制对审计一个特殊类型安全事件的决定。审计规则，也称为“本机安全策略”，是维护在本机系统上的 Lsass 安全规则的一部分。Lsass 在系统初始化时和更改规则时，向 SRM 发送消息通知其审计策略。Lsass 负责接收来自 SRM 在审计事件的基础上产生的审计记录，对它们进行编辑并发送到 Event Logger。Lsass（不是 SRM）发送这些记录是因为它添加了有关的细节，例如更完全地识别被审计的进程所需的信息。

SRM 通过到 Lsass 的 LPC 连接发送这些审计记录。Event Logger 随后将审计记录写入安全 Event Log。除了由 SRM 传递的审计记录之外，Lsass 和 SAM 二者都产生由 Lsass 直接发送到 Event Logger 的审计记录。图 8-6 描述了整个流程。

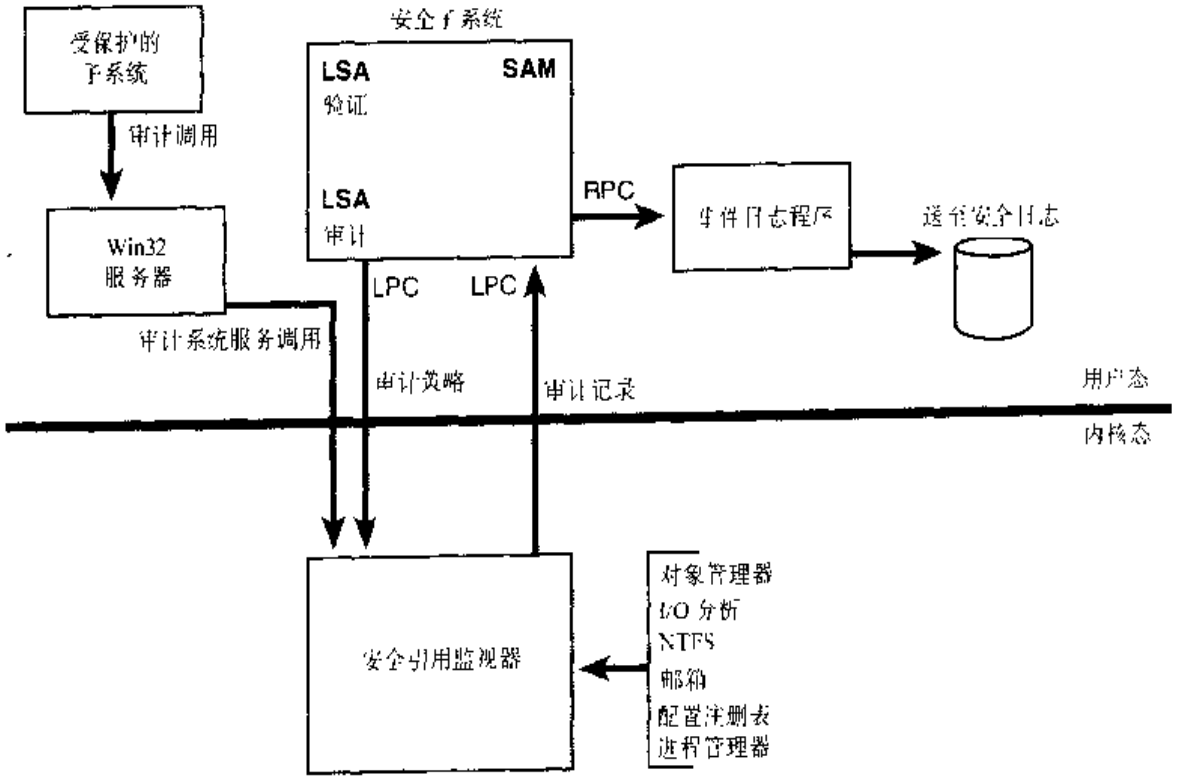


图 8-6 安全审计记录流程

当接收到审计记录后，它们被放到队列中以被发送到 LSA——它们不会被成批提交。可以

使用两种方式中的一个将审计记录从 SRM 移到安全子系统。如果审计较小（小于最大的 LPC 消息），那么它就被作为 LPC 消息发送。审计记录将从 SRM 的地址空间复制到 Lsass 进程的地址空间。如果审计记录较大，SRM 使用共享内存从而 Lsass 可以使用该消息，并在 LPC 消息中简单地传送一个指针。

图 8-7 将本章迄今所涉及的概念结合起来说明了基本的进程和线程的安全结构。请注意，图中进程对象和线程对象有 ACL，这像令牌访问对象一样。此外，图中线程 1 和线程 2 都有一个模拟的令牌，而线程 1 默认为进程访问令牌。

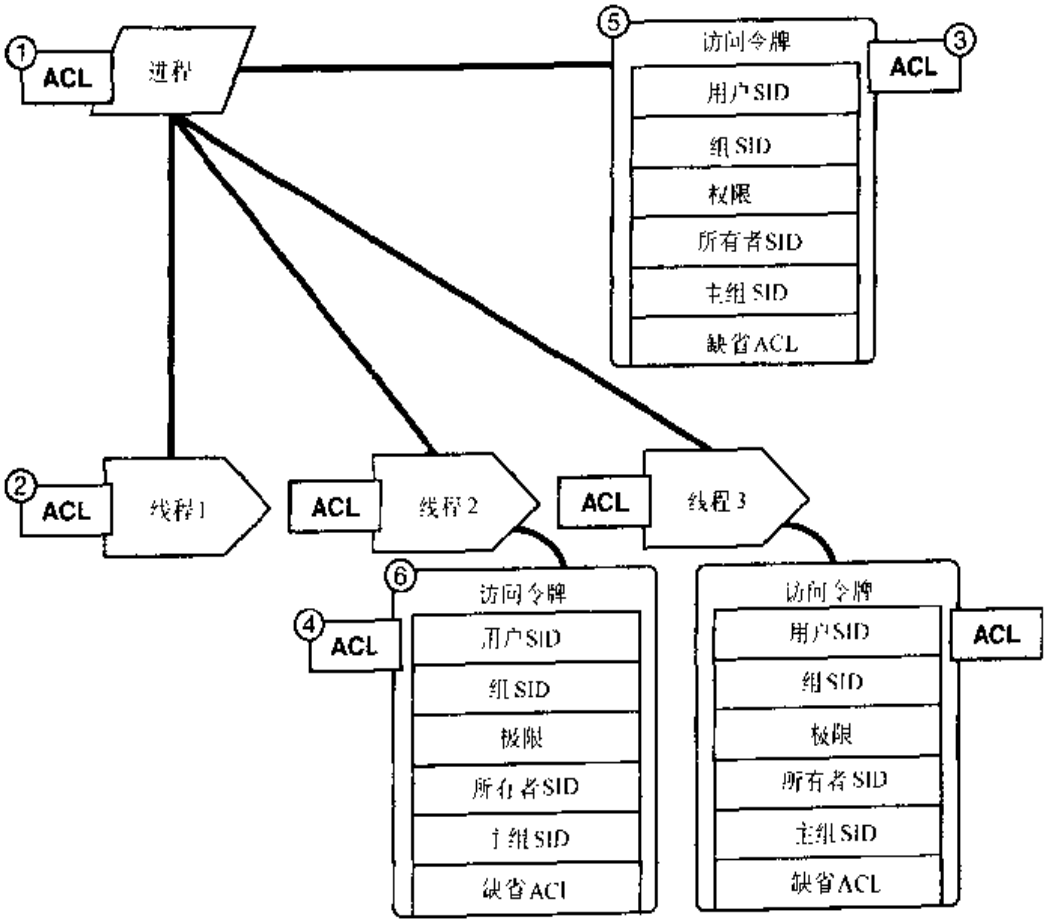


图 8-7 进程和线程的安全结构

实验：查看进程和线程的安全信息

利用工具 Process Explode（Pview.exe）可以查看进程和线程的安全描述符，并访问令牌。（该工具是 Windows NT 4 资源工具箱的一部分，但它不包括在 Windows 2000 资源工具箱中。然而，你可以从 www.reskit.com 中下载该工具。）Precoss Explode 工具中的 Security and Token 部分的六个数字按钮与图 8-7 中的进程和线程结构相匹配，如图 8-8。

在本例中，按钮 4（线程令牌 ACL）和按钮 6（线程访问令牌）是灰色的（被禁止），因为当前所选择的线程（1700 号）没有线程相关的访问令牌。

请在交互会话中试着查看一下某个进程的 ACL。你将看到 ACL 中你的用户 ID 和 SYSTEM 具有完全控制许可。

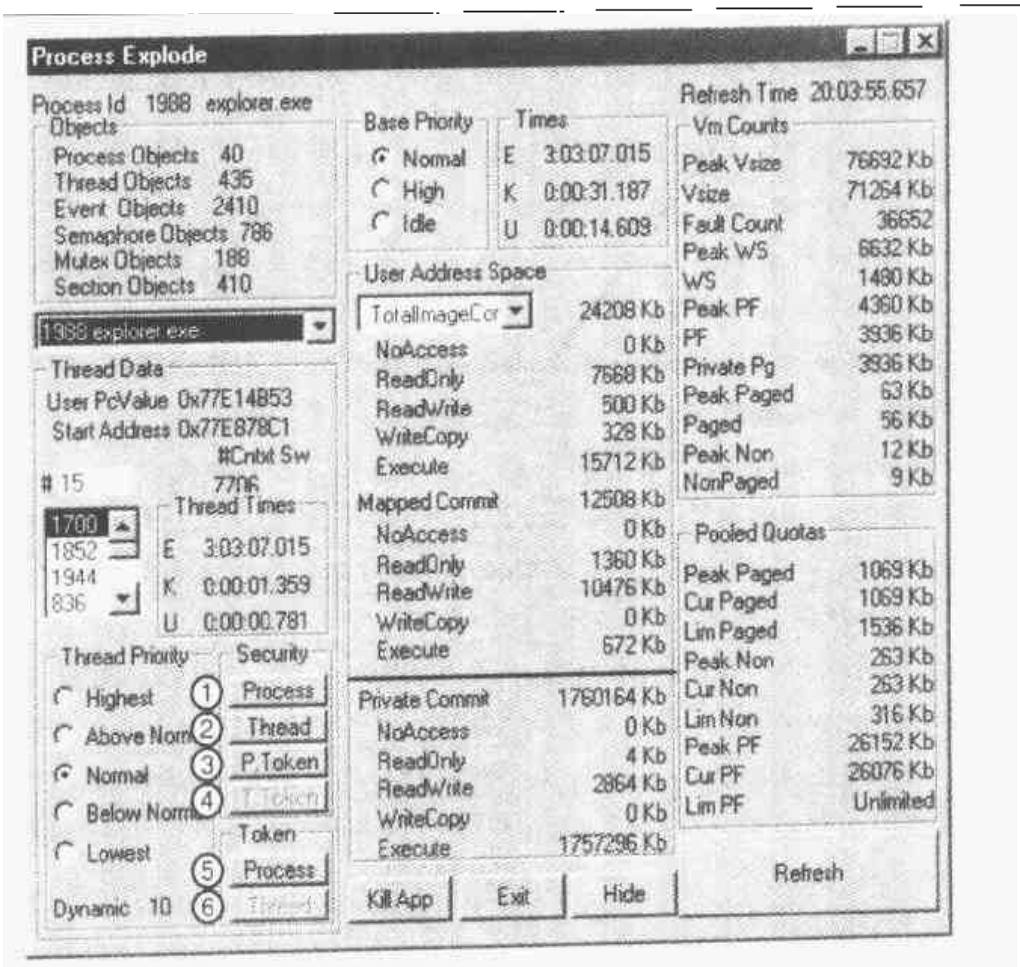


图 8-8 查看进程和线程的结构

为了观察运行中的进程安全结构，请试着进行以下操作：

1）创建一个命名为“test”的本机用户名。启动 Computer Management，添加一个用户（在 Control Panel 中打开 Administrator Tool，然后单击 Computer Management；或者是从 Start 菜单中选择 Run，并输入 compmgmt、msc）。展开 Local User And Group（在 System Tools 下），右击 Users，选择 New User。输入用户名 test（没有密码），清除 User Must Change Password At Next Logon 标志，并按 Create。（注意：你的域或组策略可能要求你输入密码。）

2）打开 Windows 2000 Command Prompt 窗口。（从 Start 菜单中选择 Programs/Accessories/Command Prompt。）

3）键入 runas/user: test cmd，创建一个运行在刚才添加的 test 用户名下的命令提示符。（注意：你可能需要在 test 的前面加上机器名或域名，如 runs/user: machine \ test cmd。）

4）在该提示符下运行 Windows 2000 资源工具包中的 Pulist。注意：除了在 test 用户名的环境中运行 Cmd.exe 和 Pulist.exe 外，你不能查看系统的进程的安全 ID。不能查看该 ID 的原因是 test 用户名在该进程访问令牌的 ACL 中，但是不在其他进程中。因此，你没有查看这些进程的许可。

5）现在退回到第 2 步中启动的命令提示符（运行在你为本实验所登录的帐户下），并再次运行 Pulist。你将看到在你的交互会话中所有进程的 ID（以及系统进程的，如果你的帐户是本机 Adminstrator 组中的成员的话）。然而，你不能查看到在 test 用户名（在第 3 步创建的）

下运行 Cmd.exe 进程的安全 ID，原因是相同的，在 test 用户名下，你不能查看没有运行在 test 下的进程。

6) 现在再次运行 Process Explode，在你的交互会话中选取一个进程（如 Explorer.exe），并修改该进程及其访问令牌的 ACL，以允许 test 用户具有读取访问权力。按下 Precess 按钮，调出 Process Permission 对话框，可以修改进程。按 Add 按钮在进程的 ACL 中添加一项。当组列表出现时，按下 Show Users 按钮，向下滚动 Names 列表框，选择 test 用户名，按 Add 按钮，并单击 OK 按钮。你将在 ACL 列表中看到 test 用户名。对于进程访问令牌，遵循同样的步骤，不过这次是按 P.Token 按钮，而不是 Process 按钮。

7) 现在返回到以 test 用户运行 Cmd 的进程，并再次运行 Pulist。这次，你将能够查看进程 Explorer.exe 的 ID，因为你已经被授予了该进程访问令牌的读取访问权限。

以下步骤在安装了 TerminalServices 的 Windows 2000 服务器系统中不起作用。原因是它们不能在步骤 10 之后所解释的环境中工作。

8) 作为最后的测试，启动 Task Manager，单击 Applications 标签，右击 cmd（running as test），并选取 Go To Process。这将使你进入 Process 标签，其中 cmd.exe 进程被高亮度显示。单击该进程，然后按下 End Process 按钮（在所出现的警告对话框中按 Yes）。你将得到访问被拒绝的错误，因为你不在该进程的 ACL 中。

9) 这是一个添加的练习，从运行 test 的命令提示符再次运行 Pview.exe，并改变 Cmd.exe 进程的进程访问权限，授予你当前使用的用户名完全访问权限。（在 Pview 中，按 Process 按钮，单击 Add，然后将你的用户名添加到 ACL 中，记得选择完全控制访问。）

10) 重试步骤 8——这次，你将能够在 test 用户名下终止命令行提示符，因为你授予了自己对该进程的完全访问权限。

步骤 8 到步骤 10 在安装了 Terminal Services 的系统中不起作用的原因是，这些步骤依赖于以下事实：在没有 Terminal Services 的系统中，Task Manager 使用 TerminateProcess 函数结束进程。然而，在运行着 Terminal Services 环境的系统中，当你指示 Task Manager 结束某个进程时，它将调用 Termsrv.exe，终端服务的服务来执行进程的终止。因为 Termsrv.exe 是运行在 System 帐户下的服务进程，它具有调试权限，使用该权限，它能够打开一个进程，而且它的终止不考虑进程的令牌或进程的安全设置。

8.5 登录

交互登录（相对于网络登录）是通过登录进程（Winlogon）、Lsass、一个或多个验证包和 SAM 或 Active Directory 的相互作用发生的。验证包是执行验证检查的 DLL。Kerberos 是 Windows 2000 中交互登录到域的验证包。MSV1-0 是 Windows 2000 交互登录到本地计算机的验证包，该包用于被信任的每个 Windows 2000 域的域登录以及没有可访问的域控制器情况。

Winlogon 是一个被信任的进程，负责管理与安全性相关的用户交互。它协调登录、启动用户 shell、处理注销和管理其他各种与安全性相关的操作，包括输入登录密码、更改密码以及锁定和解锁工作站。Winlogon 进程必须确保与安全性相关的操作对其他任何活动进程都是不可见

的。例如，Winlogon保证非信任的进程在进行这些操作期间不能控制桌面并由此获得对密码的访问。

Winlogon依靠一个图形标识与验证（GINA）的DLL来获取用户的帐户名和密码。默认的GINA是Msgina（\winnt\System32\Msgina.dll）。Msgina显示标准的Windows 2000登录对话框。允许使用其他的GINA替换Msgina使得Windows 2000能够使用不同的用户标识机制。例如，第三方可能提供使用指纹识别装置的GINA来确认用户，并从加密的数据库中提取它们的密码。

Winlogon是从键盘截取登录请求的唯一进程。从GINA获取用户名和密码后，Winlogon将调用Lsass来确认用户试图的登录。如果用户被确认，登录进程将激活一个代表该用户的登录shell。登录中涉及到的组件之间的交互显示在图8-9中。

除了提供可替换的GINA外，Winlogon还能够加载额外的网络供应商提供的DLL，用来进行第二次验证。该性能允许多个网络供应商在正常的登录期间一次收集所有的认证和验证信息。一个登录到Windows 2000的用户可能同时还通过某个UNIX服务器的验证。然后，那个用户就能够从Windows 2000的机器上访问UNIX服务器的资源，而不需要另外的验证。这种能力就是称之为单一注册（single sign-on）的一种形式。

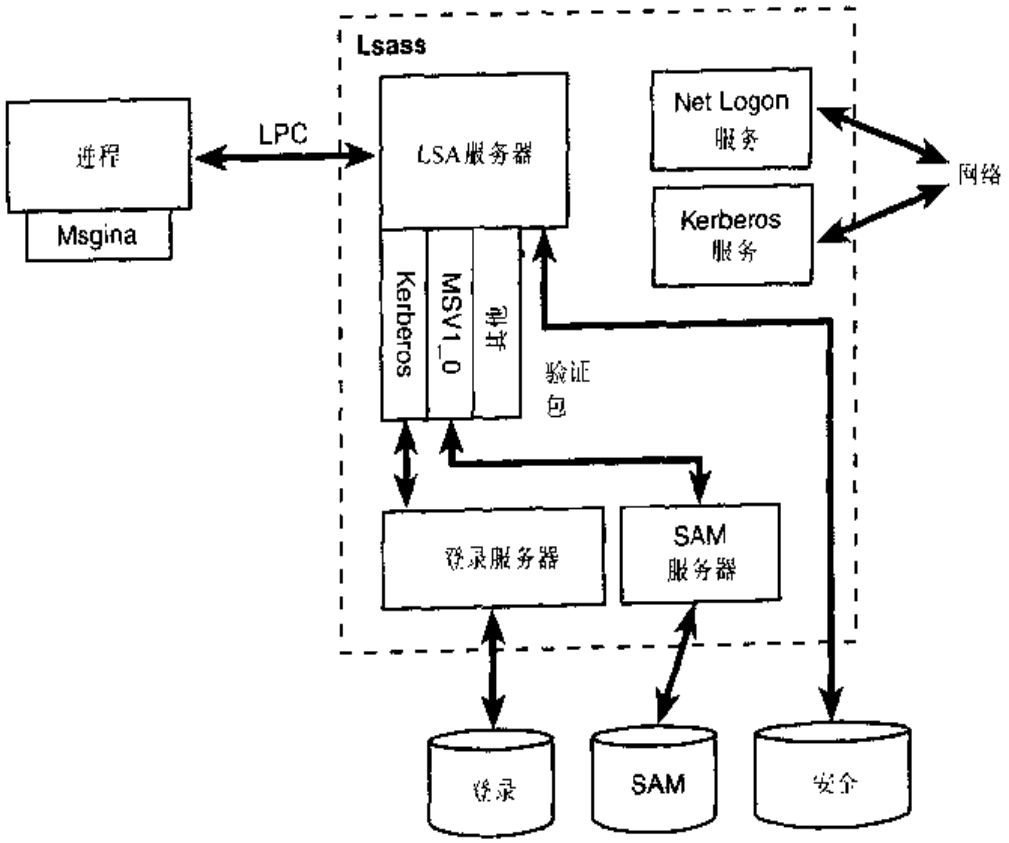


图8-9 登录中涉及的组件

8.5.1 Winlogon初始化

系统初始化过程中，在激活任何用户应用程序之前，Winlogon将执行以下步骤以确保它在系统为用户做好交互准备的情况下能够控制工作站：

1）创建并打开一个代表键盘、鼠标和监视器的窗口站，\Windows\WinSta0。Winlogon为

该窗口站创建一个安全描述符，该站有且只有一个只包含 Winlogon SID 的 ACE。这个唯一的安全描述符确保没有其他进程可以访问该工作站，除非得到 Winlogon 的明确许可。

2）创建并打开三个桌面：应用程序桌面（\ Windows \ WinSta0 \ Default）、Winlogon 桌面（\ Windows \ Winst0 \ Winlogon）和屏幕保护桌面（\ Windows \ WinSt0 \ ScreenSaver）。在 Winlogon 桌面上创建安全性以便只有 Winlogon 可以访问该桌面。其他两个桌面允许 Winlogon 和用户访问。这种安排意味着任何情况下 Winlogon 桌面都是被激活的，其他进程不能访问与该桌面相关的活动代码和数据。Windows 2000 利用该特性来保护包括密码、锁定和解锁桌面的安全操作。

在登录到计算机之前，可见的桌面是 Winlogon 的。登录之后，按下 Ctrl + Alt + Delete 可以将桌面从缺省状态切换到 Winlogon。（这说明了在按 Ctrl + Alt + Delete 然后取消 Windows Security 对话框返回时交互桌面上的所有窗口好象都消失了的原因）。这样，SAS 总是调用出 Winlogon 控制的安全桌面。

3）使用 LsaAuthenticationPort 建立与 Lsass 的 LPC 连接。该连接将用于在登录、注销和密码操作期间交换信息。该操作是通过调用 LsaRegisterLogonProcess 来完成的。

然后 Winlogon 执行以下 Windows 操作，建立窗口环境：

4）初始化并注册一个窗口类数据结构。它将 Winlogon 过程与随后将创建的窗口关联起来。

5）注册 SAS 并与刚才创建的窗口相联系，保证在用户输入 SAS 时，Winlogon 的窗口程序能够被调用。该措施阻止了特洛木伊马木马程序在输入 SAS 的时候获取屏幕的控制。

6）注册该窗口以便在用户注销或屏幕保护程序超时的时候能调用与该窗口相关的程序。Win 子系统检查以验证请求通知的进程是 Winlogon 进程。

一旦在初始化过程中创建了 Winlogon 桌面，那么该桌面就成为活动桌面。Winlogon 桌面在活动时，总是锁定的。Winlogon 只有在切换到应用程序桌面或屏幕保护桌面时才解锁（只有 Winlogon 进程能够锁定或解锁一个桌面）。

8.5.2 用户登录步骤

当用户按 SAS（或按下 Ctrl + Alt + Delete）时，登录就开始了。在按了 SAS 以后，Winlogon 调用 GINA 获取用户名和密码。Winlogon 还为该用户创建一个唯一的本机组，并将桌面的这个实例（键盘、屏幕和鼠标）分配给该用户。Winlogon 将该组作为 LsaLogonUser 调用的一部分传送至 Lsass。如果用户成功地登录，该组将被包含在登录进程令牌中——这保护了对桌面的访问。例如，另一个用户登录到同样的帐号但在不同的系统中，将不能够写第一个用户的桌面，因为第二个用户不在第一个用户的组中。

在输入用户名和密码之后，Winlogon 将相继调用每个注册的验证包。验证包列举在注册表 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Lsa 中。Winlogon 调用 Lsass 函数 LsaLogonUser 获取一个包句柄。Winlogon 通过 LsaLogonUser 传递包登录信息。一旦某个包验证了某个用户，Winlogon 将继续该用户的登录进程。如果没有任何包显示成功的登录，登录进程将终止。

Windows 2000 使用两个标准的验证包：Kerberos 和 MSV1 - 0。在单机 Windows 2000 系统中默认的验证包是 MSV1-0（\ Winnt \ System32 \ Msv1 - 0.dll），该包实现 LAN Manager 2 协议。

Lsass还使用域中成员计算机上的MSV1-0来验证在Windows 2000之前的不能查找用于验证的域控制器的域和计算机（膝上机在断开网络连接时将归入后一种情况）。Kerberos验证包，\Winnt\System32\Kerberos.dll，是在Windows 2000域成员计算机上使用的。Windows 2000 Kerberos包，在域控制器的Kerberos服务的协作下，支持第5版、第6版修订版Kerberos协议。该协议建立在Internet RFC 1510的基础上（关于Kerberos的细节信息，请访问Internet Engineering Task Force [IETF]的WEB站点www.ietf.org）。

MSV1-0验证包获取用户名称和散列的密码，并向SAM发送请求来获取账号信息，其中包括密码、用户所属的组和所有账号限制。MSV1-0首先检查账号限制，如允许访问的类别和时间等。如果用户由于SAM数据库中的限制而不能登录，那么登录调用将失败，MSV1-0返回一个失败状态给LSA。

然后，MSV1-0对比存储在SAM中的散列密码和用户名。在高速缓存域登录的情况下，MSV1-0使用Lsass函数访问高速缓存的信息，从LSA数据库中获取“密匙”。如果信息匹配，MSV1-0为该登录会话生成LUID，并调用Lsass创建登录会话，将该唯一的标识符与该会话关联起来，传递为该用户最终创建访问令牌锁所需的信息（请回忆，一个访问令牌包含用户SID、组SID以及用户配置文件信息，如宿主目录）。

如果MSV1-0需要使用远程系统进行验证，就象用户登录到受信任的Windows 2000域中一样，它将使用NetLogon服务与远程系统中的Netlogon实例进行通信，远程系统的Netlogon与其所在系统的MSV1-0验证包交互作用，将标识符验证结果返回给正在进行登录的系统。

Kerberos标识符验证的基本控制流程与MSV1-0相同。然而，大多数情况，域登录是在成员工作站和服务器上进行的（而不是域控制器），因此，作为标识符验证的一部分，验证包必须通过网络进行通信。该包是使用域控制器上的Kerberos服务通过TCP/IP端口（端口88）进行通信的。实现Kerberos验证协议的Kerberos服务（\Winnt\System32\Kdcsvc.dll）运行在域控制器的Lsass进程中。

在验证了Active Directory的用户帐号对象的散列用户名和密码信息之后（使用Active Directory服务器——\Winnt\System32\Ntdsa.dll），Kdcsvc将域凭证返回给Lsass。Lsass通过网络返回验证的结果和用户域登录凭证给正进行登录的系统（如果登录成功）。

注意 Kerberos验证的描述是很简单的，但是强调了所涉及的各个组件的主要作用。虽然Kerberos验证协议在Windows 2000的分布式域安全中扮演了重要作用，但是它的细节超出了本书的范围。

在验证登录之后，Lsass在本机策略数据库中查找允许该用户做的访问——交互式、网络或服务进程。如果请求的登录与允许的访问不匹配，那么登录企图将被终止。Lsass通过清除它的所有数据结构来删除最近创建的登录会话，然后向Winlogon返回失败信息，接着Winlogon向用户显示相应的消息。如果请求的访问被允许，Lsass会附接某些其他安全ID（例如“Everyone”、“Interactive”等等），然后它检查它的策略数据库来了解这个用户所拥有的所有被授予的权限，并将这些权限添加到该用户的访问令牌中。

当Lsass已经得到所有必要的信息后，它将调用执行程序来创建访问令牌。执行程序为交互式登录或服务登录创建一个主要访问令牌，为网络登录创建一个模拟令牌。在成功地创建了

访问令牌以后，Lsass 将复制令牌，创建一个可以被传送到 Winlogon 的句柄，然后关闭它自己的句柄。如果需要的话，将审计该登录操作。此时，Lsass 把成功信息连同验证包返回的访问令牌句柄、登录会话的 LUID 和配置文件信息（如果有的话）返回给 Winlogon。

然后，Winlogon 在注册表中查找 HKLM \ SOFTWARE \ Microsoft \ Windows NT \ Current Version \ Winlogon \ Userinit 的键值，并创建一个进程来运行该字符串所包含的值（该值可能是由逗号分隔的几个 .exe 文件）。缺省值是 Userinit.exe，它加载用户配置信息，然后创建一个运行 HKLM \ SOFTWARE \ Microsoft \ Windows NT \ Current Version \ Winlogon \ Shell 键值（缺省值为 Explorer.exe）的进程。然后 Userinit 退出（这就是用 tlist /t 之类的命令检查时 Explorer.exe 没有父进程的原因）。

8.6 小结

Windows 2000 提供了一个扩展的安全函数集合，该集合满足政府机关和商业安装的关键需求。本章，我们已经简要地介绍了组成这些安全特性的基本的内部组件。

在下一章中，我们将考察本书涉及的最后一个主要执行组件：I/O 系统。

第9章 I/O系统

Microsoft Windows 2000的I/O系统由几个执行程序组件组成，它们一起管理硬件设备，并为应用程序和系统提供硬件设备接口。本章将首先列举I/O系统的设计目标，这些设计目标影响着I/O系统的实现。接着将介绍I/O系统的组成组件，包括I/O管理器、即插即用（Plug and Play，PnP）管理器以及电源管理器。然后将分析I/O系统的结构和组件，以及各种不同的设备驱动程序。我们将考察描述设备、设备驱动程序以及I/O请求的主要数据结构。之后，我们将描述设备的检测方法和驱动程序的安装工作。最后，我们将回顾在整个系统中完成I/O请求的必要步骤。

9.1 设计目标

Windows 2000 I/O系统的设计目标为：

■ 加快单处理器和多处理器系统的I/O处理。

■ 利用标准的Windows 2000安全机制（在第8章介绍）保护共享资源。

■ 满足Microsoft Win32、OS/2以及POSIX子系统规定的I/O服务需求。

■ 提供服务以便使设备驱动程序的开发尽可能的容易，并允许使用高级语言编写驱动程序。

■ 允许在系统中动态地添加或删除驱动程序。这种添加或删除是基于用户的指示或者是系统中硬件设备的添加或删除而导致的自动配置。

■ 允许添加能够透明地修改其他驱动程序或设备的驱动程序，而不需要对所修改的驱动程序作任何更改。

■ 支持多个可安装的文件系统，其中包括FAT、CD-ROM文件系统（CDFS）、通用磁盘格式（Universal Disk Format，UDF）文件系统以及Windows 2000文件系统（NTFS）（关于文件系统的类型与结构的具体信息请参阅第12章）。

■ 允许系统以及个别硬件设备进入或离开低能耗状态以延长电交换区的寿命并节约能源。

在随后的章节中，我们将着眼于I/O系统是如何实现这些目标的。

9.2 I/O系统组件

Windows 2000的I/O系统如图9-1所示，由几个可执行模块和设备驱动程序组成。

■ I/O管理器将应用程序和系统组件连接到虚拟的、逻辑的以及物理的设备上，并定义支持这些设备的基础体系结构。

■ 一个设备驱动程序通常为某种特定类型的设备提供I/O接口。设备驱动程序接收I/O管理器发来的命令。这些命令被导向I/O管理器所管理的设备。当那些命令完成后驱动程序通知I/O管理器。设备驱动程序经常利用I/O管理器向前传递I/O指令给其他设备驱动程序。在某

种设备接口或控制的实现中，这些设备驱动程序是共享的。

■ PnP 管理器与 I/O 管理器和一种叫做“总线驱动程序（bus driver）”的设备驱动程序紧密合作，指导硬件资源的分配，并检测和响应硬件设备的到达和删除。当检测到设备时，PnP 管理器与总线驱动程序负责加载设备驱动程序。当一个设备被添加到系统而不具备合适的设备驱动程序时，可执行的即插即用组件调用某个用户模式的 PnP 管理器的设备安装服务。

■ 电源管理器也与 I/O 管理器紧密合作，指导系统以及个别设备驱动器实现电源状态的迁移。

■ Windows 管理装置（WMI）支持称为“WDM WMI 提供程序”的例程。在用户模式利用 WDM WMI 提供者作为中介与 WMI 通信，允许设备驱动程序作为间接的提供者（关于 WMI 的详细情况请参阅 5-3 节“Windows 管理装置”）。

■ 注册表充当数据库，存储附属于系统的基本设备的描述以及驱动器的初始化和配置。

■ INF 文件是驱动程序安装文件，用后缀 .inf 表示。INF 文件是一个特定硬件设备与控制该设备的驱动程序之间的联系纽带。它们由类似于脚本的指令组成，描述相应的设备、驱动程序文件的源目标位置、安装驱动程序所需要的注册表修改，以及驱动程序的依赖信息。Windows 2000 用来检验设备驱动文件是否通过 WHQL 测试的数字签名存储在 .cat 文件中。

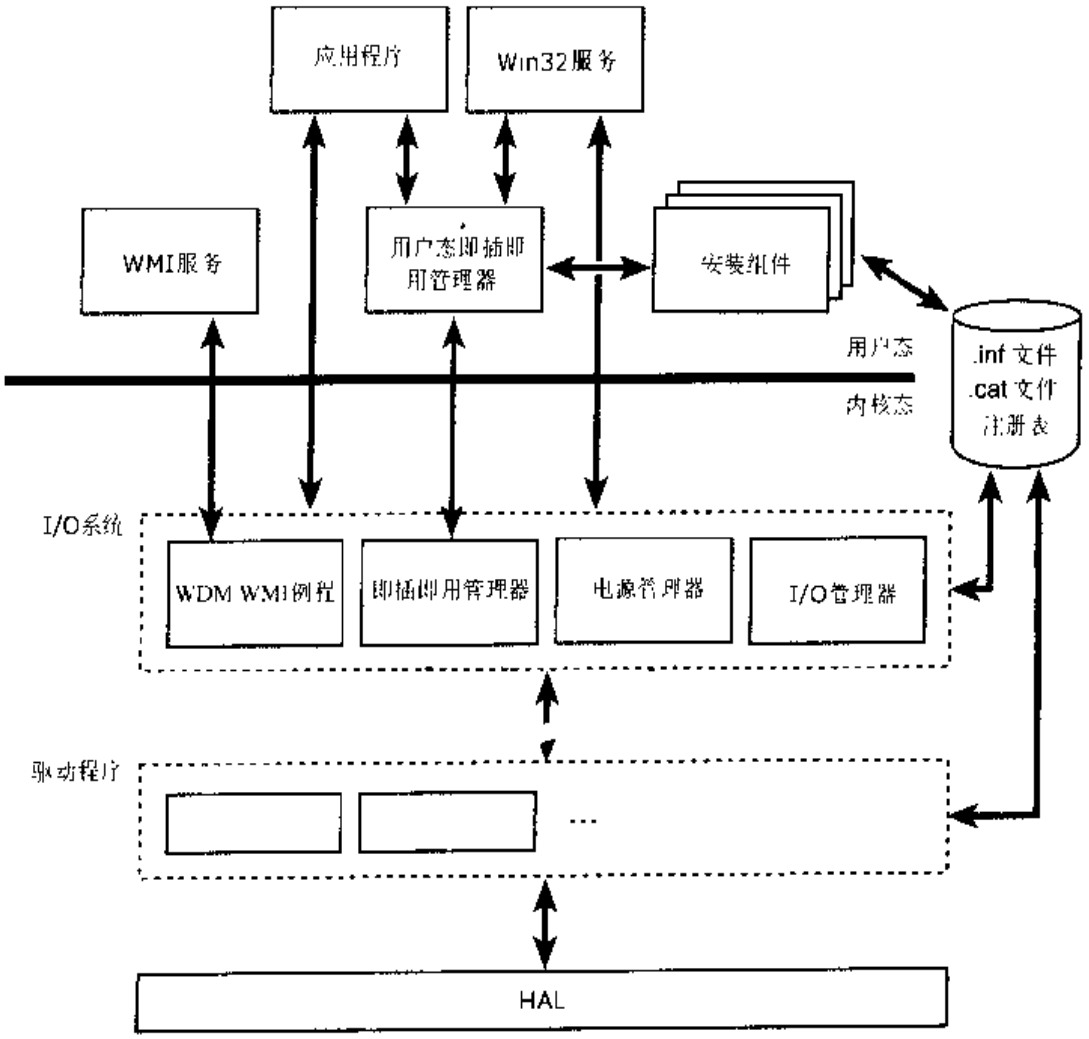


图 9-1 I/O 系统组件

■ 硬件抽象层（HAL）通过提供隐蔽平台之间的差别的API将驱动程序从处理器和中断控制器的规范中隔离开来。实际上，HAL是计算机主板上所有设备的总线驱动程序，不受其他驱动程序的控制。

绝大多数的I/O操作不会涉及刚才描述的所有组件。一个典型的I/O请求源于应用程序执行一个与I/O相关的函数（例如从设备读数据）。该函数由I/O管理器、一个或多个设备驱动程序以及HAL处理。

在Windows 2000中，线程在虚拟的文件上执行I/O操作。操作系统将所有的I/O请求抽象为对虚拟文件的操作，从而隐藏了I/O操作的目标可能不是一个文件结构型设备的事实。这种抽象将应用程序到设备的接口一般化。虚拟文件是指某种被当作文件对待的I/O资源或目的地，例如文件、目录、管道和邮件插槽等。所有读写的数据都被看作是指向相应虚拟文件的简单的字节流。用户模式应用程序（不管是Win32、POSIX还是OS/2）调用文档函数，后者再调用内部的I/O系统函数，进行文件的读写和其他操作。I/O管理器将对这些虚拟文件的请求动态地导向适当的设备驱动程序。图9-2说明了典型I/O请求的基本结构。

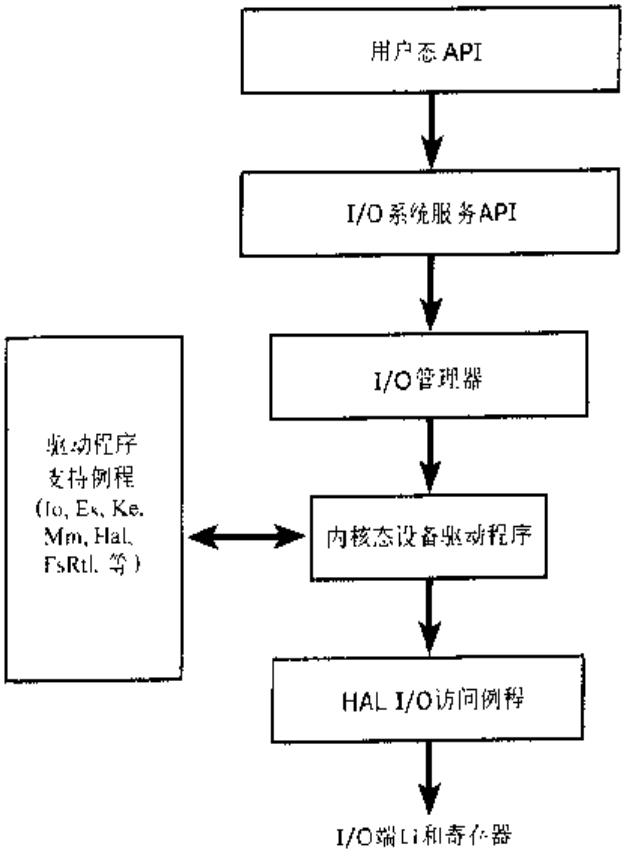


图9-2 典型的I/O请求流程

下面的章节，将进一步考察这些组件，更详细地分析I/O管理器，其中覆盖了各种设备驱动程序和关键I/O系统数据结构。然后我们将介绍PnP管理器和电源管理器的操作与任务。

9.2.1 I/O管理器

I/O管理器定义有序的工作框架或模型，在该框架内I/O请求被发送给设备驱动程序。I/O系统是包驱动的。大多数的I/O请求都是用I/O请求包（IRP）表示的。I/O请求包可以从一个I/O系统组件转移到另一组件（快速I/O是个例外，它不使用IRP）。这种设计允许单个应用程序线程并行地管理多个I/O请求。IRP是一种数据结构，包含描述一个I/O请求的完整信息（在“I/O请求包”一节，你将找到有关IRP的更多信息）。

I/O管理器创建代表I/O操作的IRP，将该IRP指针传递给正确的驱动程序，并在I/O操作完成后释放该包。相反，驱动程序接受IRP，执行IRP指定的操作，并在操作完成后将IRP传回给I/O管理器，或者是将其传递给另一个驱动程序进一步处理。

除了创建与释放IRP之外，I/O管理器还为不同的驱动程序提供公共代码。驱动程序调用

这些代码执行各自的 I/O 处理。通过在 I/O 管理器中合并共同任务，使单个的驱动程序变得更简单更紧凑。例如，I/O 管理器提供一个允许驱动程序调用其他驱动程序的函数。I/O 管理器还管理 I/O 请求缓冲区，为驱动程序提供超时支持，并记录哪些可安装的文件系统被加载到操作系统中。在 I/O 管理中有大约一百个不同的例程可被设备驱动程序调用。

I/O 管理器还提供灵活的 I/O 服务，允许环境子系统，如 Win32 和 POSIX，实现它们相应的 I/O 功能。这些服务包括用于异步 I/O 的高级服务，它们允许开发者建立可升级的高性能的服务器应用程序。

驱动程序提供的统一的、模块化的接口，使 I/O 管理器可以调用任何驱动程序而不需要与它的内部结构或细节有关的任何特殊知识。就如我们前面所陈述的那样，操作系统将所有的 I/O 请求当作指向文件对待；驱动程序将对虚拟文件的请求转换成对具体硬件的请求。驱动程序还能（利用 I/O 管理器）互相调用，以实现分层独立的 I/O 请求处理。

除了正常的打开、关闭、读、写功能外，Windows 2000 I/O 系统还提供几种高级的特性，如异步的、直接的、缓冲的、分散/集中 I/O 等，这些将在本章后面的“I/O 类型”一节介绍。

9.2.2 设备驱动程序

为了集成 I/O 管理器和其他 I/O 系统组件，设备驱动程序必须确保实现了它所管理的设备类型的指导准则以及管理设备时所起的作用。本节，我们将考察 Windows 2000 支持的设备驱动程序类型以及设备驱动程序的内部结构。

1. 设备驱动程序的类型

Windows 2000 支持许多不同类型的设备驱动程序和编程环境。即使在同一设备驱动程序类型内，编程环境也可能不同。这依赖于驱动程序所针对的具体的设备类型。本章的焦点在于内核模式设备驱动程序。有许多不同类型的内核模式驱动程序，它们可以广义地分成以下几类：

■“文件系统驱动程序”接收对文件的 I/O 请求，并通过启动其自身的更明确的海量存贮或网络设备驱动程序来满足该请求。

■“Windows 2000 驱动程序”是设备驱动程序，当需要时可与 Windows 2000 电源管理器和 PnP 管理器集成在一起。它们包括海量存贮设备、协议栈，以及网络适配器等驱动程序。

■“遗留驱动程序”是为 Microsoft Windows NT 而写的设备驱动程序，但是它们能够不加更改地在 Windows 2000 上运行。与其他 Windows 2000 驱动程序不同的是，它们不支持电源管理，不能与 Windows 2000 的 PnP 管理器一起工作，如果该驱动程序控制了某个硬件设备，那么该驱动程序可能限制电源管理以及系统的即插即用能力。

■ Win32 子系统的“显示驱动程序”将独立于设备的图形请求翻译成具体的设备请求。然后具体设备的请求与内核模式的视频小端口驱动程序组成对，一起完成对视频显示的支持。显示驱动程序通过直接写帧缓存，或者是与控制器中的图形加速芯片通信，实现绘图操作。小端口驱动程序负责显示控制器全局状态的变迁，如模式设置（屏幕分辨率，刷新速率，象素深度等）、光标（指示器）的定位以及颜色查找表的加载等。

■ WDM 驱动程序附属于 WDM 的设备驱动程序。WDM 包括对 Windows 2000 电源管理、即插即用和 WMI 的支持。WDM 是在 Windows 2000、Windows 98 和 Windows Millennium Edition 中实

现的，因此 WDM 驱动程序与这些操作系统是源代码兼容的，在许多情况下也是二进制兼容的。有三种 WDM 驱动程序：

■“总线驱动程序”管理逻辑或物理总线。作为例子的总线包括 PCMCIA、PCI、USB、IEEE 1394 和 ISA。总线驱动程序负责检测附加在其上的设备，并通知 PnP 管理器，除此之外它还能管理总线的电源设置。

■“功能驱动程序”管理某种特定类型的设备。总线驱动程序通过 PnP 管理器将设备呈现给功能驱动程序。功能驱动程序将设备的操作接口导出给操作系统。通常它是具有大量设备操作知识的驱动程序。

■“过滤器驱动程序”在逻辑层上位于功能驱动程序之上或之下。它增加或更改某个设备或者另外的驱动程序的行为。例如，键盘捕获工具可通过一个位于键盘功能驱动程序之上的键盘过滤器驱动程序来实现。

在 WDM 中，没有哪个驱动程序是负责控制一个设备的所有方面的。总线驱动程序负责检测总线成员关系的更改（设备的添加或删除），协助 PnP 管理器枚举总线上的设备，访问具体的总线配置寄存器，在某些情况下控制总线设备的电源。功能驱动程序一般是指那些只访问硬件设备的驱动程序。

注意：Windows 2000 中的 HAL 的作用不同于 Windows NT。在 Windows 2000 之前，第三方硬件零售商想要增加硬件总线本身所不能提供的支持将不得不实现一个定制的 HAL。而 Windows 2000 允许第三方实现某个总线驱动程序以提供硬件总线自身没有提供的支持。

除了以上几种设备驱动程序类型之外，Windows 2000 还支持几种用户模式驱动程序：

■“虚拟设备驱动程序”（VDD）用于模拟 16 位的 MS－DOS 应用程序。它们捕获 MS－DOS 应用程序所引用的 I/O 端口，将它们转换成 Win32 I/O 函数，然后将这些 I/O 函数传递给真正的设备驱动程序。因为 Windows 2000 是一个完全受保护的操作系统，MS－DOS 应用程序不能直接访问硬件，因此必须经过一个真正的内核模式的设备驱动程序。

■Win32 子系统的“打印机驱动程序”将独立于设备的图形请求翻译成具体的打印机指令。通常这些指令被向前传递给内核模式端口驱动程序，如并行端口驱动程序（Parport.sys）或通用并行总线（USB）打印机端口驱动程序（Usbprint.sys）。

对单个硬件的支持通常被分成几个驱动程序，每个提供该设备正常工作所需的一部分功能。除了 WDM 总线驱动程序、功能驱动程序和过滤器驱动程序之外，硬件支持还可能在以下组件之间分割：

■“类驱动程序”实现某特定类型设备的 I/O 处理，如磁盘、磁带或 CD－ROM。

■“端口驱动程序”实现针对于某种 I/O 端口的 I/O 处理，如 SCSI；这是被当作内核模式下的功能库而不是真正的设备驱动程序实现的。

■“小端口驱动程序”将对某种端口的普通的 I/O 请求映射成某种适配器，如具体 SCSI 适配器。小端口驱动程序是真正的设备驱动程序，它引入端口驱动程序提供的功能。

实例将有助于阐述设备驱动程序是如何工作的。文件系统驱动程序接收对某个文件某一位置的写数据请求。它将该请求转换成对磁盘某个特定的逻辑位置写一定数量的字节的请求。然后，它将该请求（通过 I/O 管理器）传递给简单的磁盘驱动程序。接着，磁盘驱动程序将该请求转换

磁盘上的物理位置（柱面/磁道/扇区）并操纵磁头写数据。该层次关系如图 9-3 所示。

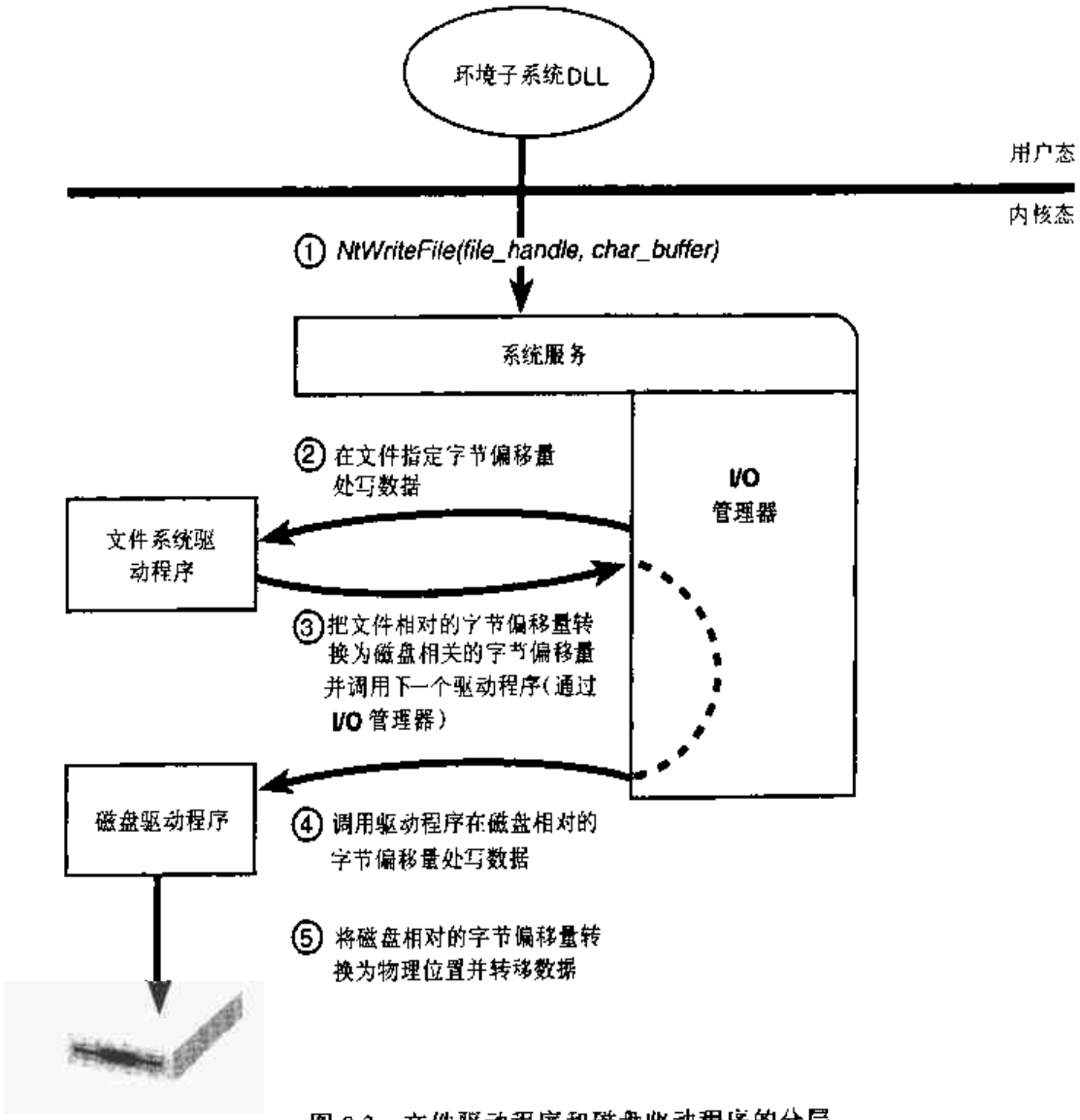


图 9-3 文件驱动程序和磁盘驱动程序的分层

该图说明了两个分层的驱动程序之间的任务划分。I/O 管理器接收写请求，该请求是相对于特定文件的起始位置的。I/O 管理器将请求传递给文件系统驱动程序。文件系统驱动程序将相对于文件的写操作转换成起始位置（磁盘上的扇区边界）和所要读写的字节数。文件系统驱动程序调用 I/O 管理器将请求传递给磁盘驱动程序，后者再将请求转换成物理磁盘位置并传输数据。

因为所有驱动程序——设备驱动程序和文件系统驱动程序——对操作系统都呈现同样的工作框架，所以可以很容易地将另外的驱动程序插入层次结构中，而不需改变现有的驱动程序或者是 I/O 系统。例如，可以通过添加驱动程序使几个磁盘看起来象一个非常大的磁盘。在 Windows 2000 中，存在这样的驱动程序以提供磁盘容错支持（虽然该驱动程序出现在所有的 Windows 2000 版本中，但只有 Windows 2000 服务器版才有磁盘容错支持）。如图 9-4 所示，这种逻辑上的卷管理器驱动程序位于文件系统与磁盘驱动器之间。

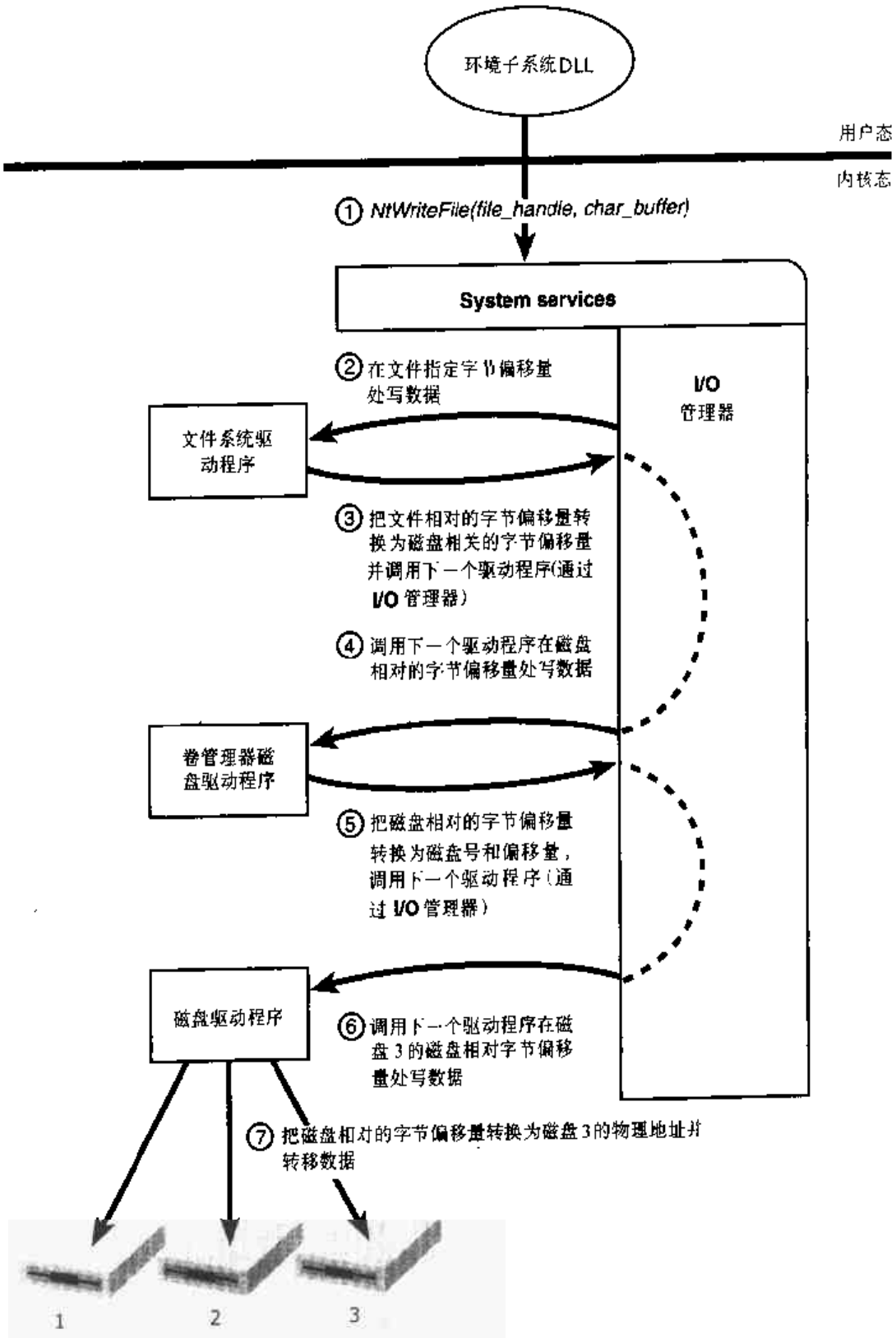


图 9-4 增加一个分层的驱动程序

实验：查看加载的驱动程序列表

进入到 Computer Management Microsoft Management Console（MMC）的 Drivers 部分或者是在桌面上右击 My Computer 并从环境菜单中选择 Manage，可以看到注册的驱动程序列表。

(Computer Management 插件在 Start 菜单的 Programs/Admimstrative Tools 文件夹中)。如图 9-5 所示，展开 System Tools、System Information、Software Environment 并选择 Drivers 可以查看 Drivers 部分。

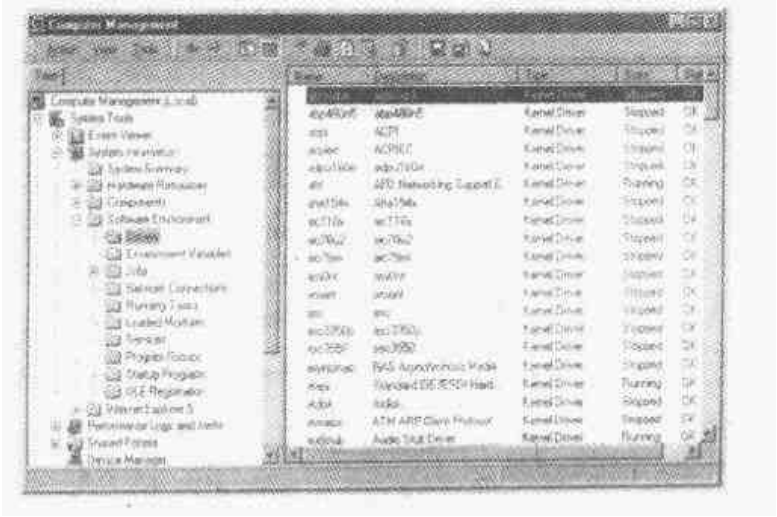


图 9-5 查看加载的驱动程序列表

利用 Windows 2000 资源工具箱中的 Drivers 工具或者是 Pstat 工具还可以获得加载的内核模式驱动程序列表。(Pstat 移植在平台软件开发工具中，可以从 Windows 2000 资源工具 Web 站点 www.microsoft.com/2000/library/resources/reskit 下载)。Pstat 在其显示的尾部列举驱动程序。(它首先列举系统中所有的进程和线程。) 这两个工具输出的唯一区别是 Pstat 显示驱动程序在系统地址空间中的加载地址。以下输出是 Pstat 显示的部分驱动程序信息。

```
C:\>pstat
  :
  ModuleName Load Addr   Code    Data    Paged       LinkDate
  ------------------------------------------------------------------------
ntoskrnl.exe 80400000  429184   96896  775360 Tue Dec 07 18:41:11 1999
     hal.dll 80062000   25856    6016   16160 Tue Nov 02 20:14:22 1999
 BOOTVID.DLL EE010000    5664    2464       0 Wed Nov 03 20:24:33 1999
    ACPI.sys BFFD8000   92096    8960   43488 Wed Nov 10 20:06:04 1999
  WMILIB.SYS EE1C8000     512       0    1152 Sat Sep 25 14:36:47 1999
     pci.sys EDC00000   12704    1536   31264 Wed Oct 27 19:11:08 1999
  isapnp.sys EDC10000   14368     832   22944 Sat Oct 02 16:00:35 1999
compbatt.sys EE014000    2496       0    2880 Fri Oct 22 18:32:49 1999
   BATTC.SYS EE100000     800       0    2976 Sun Oct 10 19:45:37 1999
intelide.sys EE1C9000    1760      32       0 Thu Oct 28 19:20:03 1999
 PCIIDEX.SYS EDE80000    4544     480   10944 Wed Oct 27 19:02:19 1999
  pcmcia.sys BFFBD000   32800    8864   23680 Fri Oct 29 19:20:08 1999
  ftdisk.sys BFFA0000    4640      32   95072 Mon Nov 22 14:36:23 1999
Diskperf.sys EE102000    1728      32    2016 Thu Sep 30 20:30:40 1999
    dmio.sys BFF7E000  104672   15168       0 Tue Nov 30 14:47:49 1999
  :
```

如果你利用内核调试器查看故障转储 (或活动的系统)，可以使用内核调试器的!drivers 命令得到类似的显示。

2. 驱动程序的结构

I/O系统驱动设备驱动程序的执行。设备驱动程序由一组例程组成，这些例程被用来处理不同阶段的I/O请求。图9-6说明了下面将描述的关键的功能驱动程序例程。

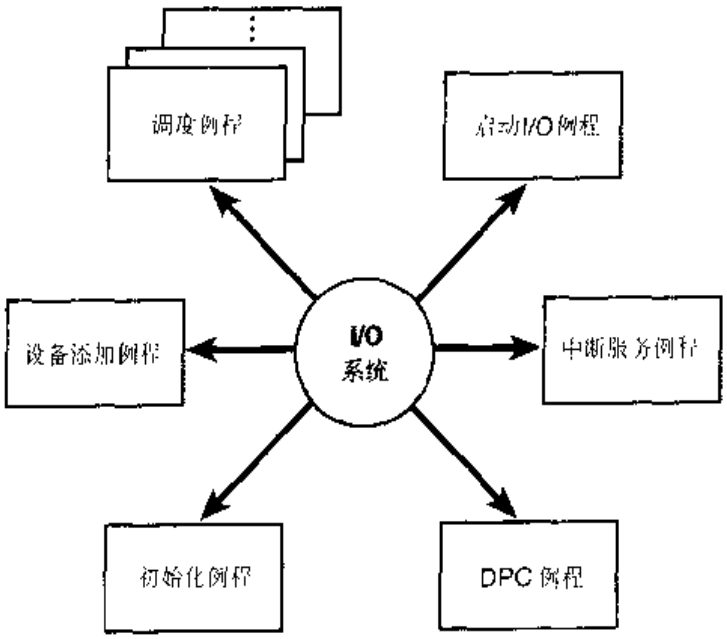


图9-6 主要的设备驱动程序例程

■ **初始化例程** 当I/O管理器将驱动程序载入操作系统时，它将执行驱动程序的初始化例程，该例程一般命令为DriverEntry。该例程将填充系统数据结构，以便使用I/O管理器注册的驱动程序的其他例程，同时，该例程还将执行所有必须的全局的驱动程序初始化工作。

■ **设备添加例程** 支持即插即用的驱动程序将实现一个设备添加例程。当检测到某个驱动程序所负责的设备时，PnP管理器将通过该例程向该驱动程序发送通知。在该例程中，设备驱动程序将分配一个设备对象（在本章的后部分介绍）表示此设备。

■ **一组调度例程** 调度例程是设备驱动程序提供的主要功能。一些可能的例子是打开、关闭、读写以及设备、文件系统或网络所支持的其他任何功能。当为执行某个I/O操作而被调用时，I/O管理器将生成一个IRP并通过调度例程之一调用驱动程序。

■ **启动I/O例程** 驱动程序利用启动I/O例程初始化设备的数据传输。该例程只在依赖于I/O管理器的IRP串行化的驱动程序中才有意义。I/O管理器通过确保一次只处理一个IRP来达到串行化驱动程序的IRP的目的。大多数驱动程序并行地处理多个IRP，但是串行化对某些驱动程序是有必要的，如键盘驱动程序。

■ **中断服务例程** 当设备中断时，内核中的中断调度程序将控制传送给该例程。在Windows 2000的I/O模型中，ISR运行在设备中断请求级上（DIRQL），因此它们执行尽可能少的工以避免不必要的低级中断阻塞（有关IRQC的详细情况，请参阅第3章）。ISR为了执行剩余的中断处理，将延迟的进程调用（DPC）排成队列（只有中断驱动的设备的驱动程序才具有ISR。例如文件系统驱动程序就不具备ISR）。

■ **服务于中断的DPC例程** 执行ISR之后，DPC例程将执行与设备中断处理有关的大部

分工作。DPC 例程执行比设备级的 ISR 更低的 IRQC（DPC/调度级别）以避免其他不必要的中断阻塞，DPC 例程初始化 I/O 完成例程，并启动下一个排队等待该设备的 I/O 操作。

尽管下面的例程没有出现在图 9-6 中，但是在多种设备驱动程序中可以找到它们：

■ **一个或多个 I/O 完成例程** 分层的驱动程序可能具有 I/O 完成例程。当更低级别的驱动程序完成 IRP 处理时，I/O 完成例程通知驱动程序。例如 I/O 管理器在设备驱动程序完成文件数据传输之后将调用文件系统驱动程序的 I/O 完成例程。该完成例程将告诉文件系统驱动程序有关操作的成败或取消等信息，并允许文件系统驱动程序执行清除操作。

■ **取消 I/O 例程** 如果一个 I/O 操作可以被取消，那么驱动程序将定义一个或多个取消 I/O 例程。当驱动程序收到一个可以被取消的 I/O 请求的 IRP 时，它将分配一个取消 I/O 例程给该 IRP。如果在请求完成或取消该操作（例如利用 Win32 的 CancelIO 函数）之前，存在发起该 I/O 操作线程，那么 I/O 管理器将执行 IRP 的取消例程，如果分配了的话。取消例程负责执行释放 IRP 处理中所获取的所有资源所必须的步骤，并结束该 IRP 使其处于被取消的状态。

■ **卸载例程** 卸载例程释放驱动程序使用的系统资源，以便 I/O 管理器能够从内存中删除它们。在初始化例程中获取的所有资源通常都是在卸载例程中释放的，驱动程序可以在系统运行时被加载或卸载。

■ **系统关闭通知例程** 该例程允许驱动程序在系统关闭时进行清除工作。

■ **错误登录例程** 当发生非预期的错误时（如磁盘块损坏），驱动程序的错误登录例程将记录该错误并通知 I/O 管理器。I/O 管理器将该信息写入错误日志文件。

9.2.3 即插即用管理器

即插即用（Plug and Play，PnP）管理器是支持 Windows 2000 识别并自适应地更改硬件配置能力的主要组件。用户不需要为安装或删除设备而去理解硬件或手工配置的复杂性。例如，正是 PnP 管理器使置于接坞站（Docking Station）上的运行着 Windows 2000 的膝上机自动地检测接坞站中的附加设备，并使用户能够利用它们。

即插即用功能需要硬件级、驱动程序级和操作系统级的协作。附加在总线上的设备枚举与识别的工业标准是 Windows 2000 即插即用功能的基础。例如，USB 标准规定了识别 USB 总线上的设备的方法。利用该基础，Windows 2000 即插即用提供以下功能：

■ PnP 管理器自动识别所安装的设备。该过程包括在启动期内枚举附属于系统的设备，以及检测在系统执行时添加与删除的设备。

■ 硬件资源的分配是 PnP 管理器的任务，它在一个称为“资源判优”（resource arbitration）的进程中收集隶属于系统的设备的硬件资源需求（中断、I/O 内存、I/O 寄存器或者是专用总线资源），并优化资源分配，从而满足每一个设备所必须的工作条件。因为硬件资源可在启动时的资源分配之后再添加到系统中，所以 PnP 管理器还必须能够重新分配资源以便动态地满足添加的设备的需求。

■ 加载适当的驱动程序是 PnP 管理器的另一项职责。PnP 管理器在设备识别的基础上判断能够管理该设备的驱动程序是否安装在系统中。如果是的话，它将指示 I/O 管理器加载它。如

果没有安装合适的驱动程序，那么内核模式的PnP管理器将与用户模式的PnP管理器进行交流以安装该设备，有可能还会请求用户协助查找合适的驱动程序集合。

■ PnP管理器还实现应用程序和驱动程序检测硬件配置的更改的机制。应用程序或驱动程序有时可能会需要专用的设备，因此Windows 2000为它们提供了一种请求告知设备的出现、添加或删除的方法。

1. 即插即用支持的级别

Windows 2000的目标是提供完全的即插即用，但是可能支持的级别依赖所属的设备和安装的驱动程序。如果某个设备或驱动程序不支持即插即用，那么系统的即插即用支持将打折扣。此外，不支持即插即用的驱动程序还可能阻止系统使用其他设备。表9-1列举了能够支持和不能够支持即插即用的设备和驱动程序的不同组合结果。

表9-1 设备和驱动程序的即插即用能力

	驱动程序的类型	
设备类型	即插即用	非即插即用
即插即用	完全即插即用	不能即插即用
非即插即用	可能部分即插即用	不能即插即用

非即插即用兼容的设备是指那些不支持自动检测的设备，如老式的ISA声卡。由于操作系统不知硬件的物理位置，某些操作如膝上机出坞、休眠、冬眠等，将被禁止。然而，如果即插即用驱动程序是为某个设备手工安装的，那么该驱动程序至少可以为该设备实现面向资源分配的PnP管理器。

与即插即用不兼容的驱动程序包括遗留的驱动程序在内，如那些运行在Windows NT 4上的驱动程序。虽然这些驱动程序可以在Windows 2000中继续使用，但是当为了容纳新添加的设备而必须重新分配资源时，PnP管理器不能重新配置分给这些设备的资源。例如某个设备能够使用I/O内存范围A和B，在启动期间PnP管理器将范围A分给它。如果后来某个只能使用范围A的设备加入到系统，那么PnP管理器将不能指示第一个设备驱动程序重新配置它自己以使用范围B。这样就阻止了第二个设备获取所需要的资源，其结果是系统不能使用该设备。遗留驱动程序还将损害机器的休眠或冬眠能力（详细情况请参阅9.2.4节“电源管理器”）。

2. 驱动程序对即插即用的支持

为了支持即插即用，驱动程序必须实现即插即用调度例程以及设备添加例程。总线驱动程序必须支持此功能驱动程序或过滤驱动程序更多种类的即插即用请求。例如，当PnP管理器在系统启动期间指导设备枚举的时候（本章后部分将详细介绍），它会向总线驱动程序要求在相应总线上找到的设备的描述信息。这种描述包括唯一区别每个设备的数据以及设备所需求的资源。PnP管理器获取该信息并加载所有为该设备安装的功能或过滤驱动程序。然后它将调用各个负责所安装的设备的驱动程序的设备添加例程。

功能和过滤器驱动程序是为在设备添加例程开始时管理设备而准备的，但是它们并不真正

与硬件设备打交道。相反，它们等待 PnP 管理器发送该设备的 start - device 命令给它们的即插即用调度例程。start - device 命令包括 PnP 管理器在资源仲裁期间决定的资源分配。当驱动程序收到 start - device 命令后，它就能够将它的设备配置成使用指定的资源。

当设备启动以后，PnP 管理器就能够向驱动程序发送其他即插即用命令，包括与设备的删除或资源的再分配等有关的命令。例如，当用户如图 9-7 所示的那样唤醒删除/弹出设备工具（在任务栏的 PC 卡图标上右击并选择 Unplug or Eject Hardware 可以访问它），要求 Windows 2000 弹出 PCMCIA 卡时，PnP 管理器将向所有注册了该设备的即插即用通知的应用程序发送 query - remove 通知。应用程序一般在它们的句柄中注册通知。在 query - remove 通知期间，它们将关闭句柄。如果没有应用程序否决 query - remove 请求，那么 PnP 管理器将发送 query - remove 命令给拥有将要弹出的设备的驱动程序。此刻，驱动程序有机会抵制删除或者是确保挂起的涉及该设备的 I/O 操作执行完，并开始拒绝对该设备的下一步 I/O 操作。如果驱动程序同意删除请求且没有其他打开该设备的句柄，那么 PnP 管理器下一步将发送 remove 命令给驱动程序，请求驱动程序中断对该设备的访问，并释放驱动程序为该设备分配的所有资源。

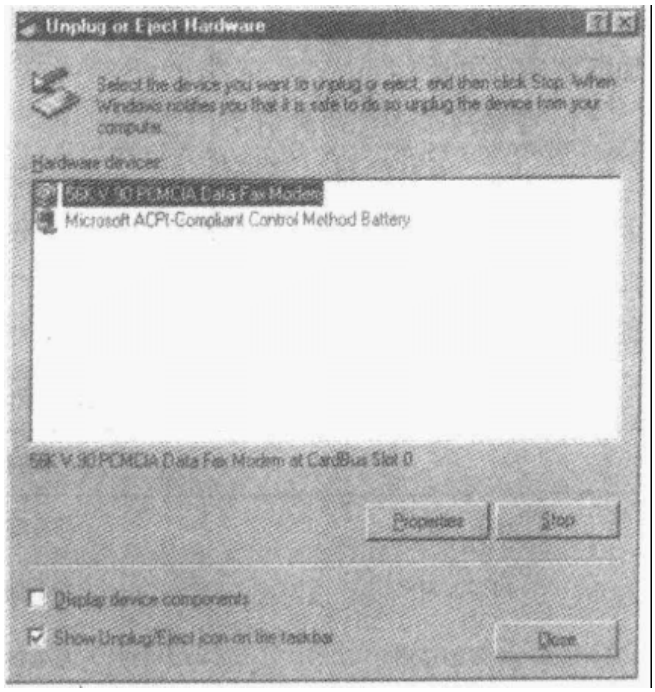


图 9-7 PC 卡删除/弹出工具

当 PnP 管理器需要重新分配设备资源时，它通过发送 query - stop 命令询问驱动程序是否可以暂时挂起设备上更进一步的活动。驱动程序或者同意该请求，或者拒绝该请求。如果同意的话将不会造成数据的损失或崩溃。就象 query - remove 命令一样，如果同意该请求，驱动程序将完成挂起 I/O 操作，不再初始化对该设备的 I/O 请求。这些 I/O 请求是指那些不能被中断而随后重新启动的 I/O 请求。驱动程序一般对新的 I/O 请求排队，因此资源的再分配对当前访问设备的应用程序而言是透明的。然后 PnP 管理器给驱动程序发送一条 stop 命令。此刻，PnP 管理器可以指示驱动程序给设备分配不同的资源，并再次发送该设备的 start - device 命令给驱动程序。

各种不同的即插即用命令基本上都是通过各种操作状态形成的状态转移表来指导设备的。简化形式的状态转移表显示在图 9-8（为简单起见，几种可能的转移和即插即用命令被删节了。另外，所画的状态图显示的是由功能驱动程序实现的。总线驱动程序实现的状态图更为复杂）。图中显示的而我们还没有讨论的一个状态是源于 PnP 管理器的 surprise - remove 命令。当用户删除某个设备而没有警告或者设备出错时将导致该命令。这就象用户弹出一块 PCMCIA 卡而不使用删除/弹出工具的情况一样。surprise - remove 命令要求驱动程序立即停止与设备的所有交互，并取消所有挂起的 I/O 请求，因为该设备不再隶属于系统。

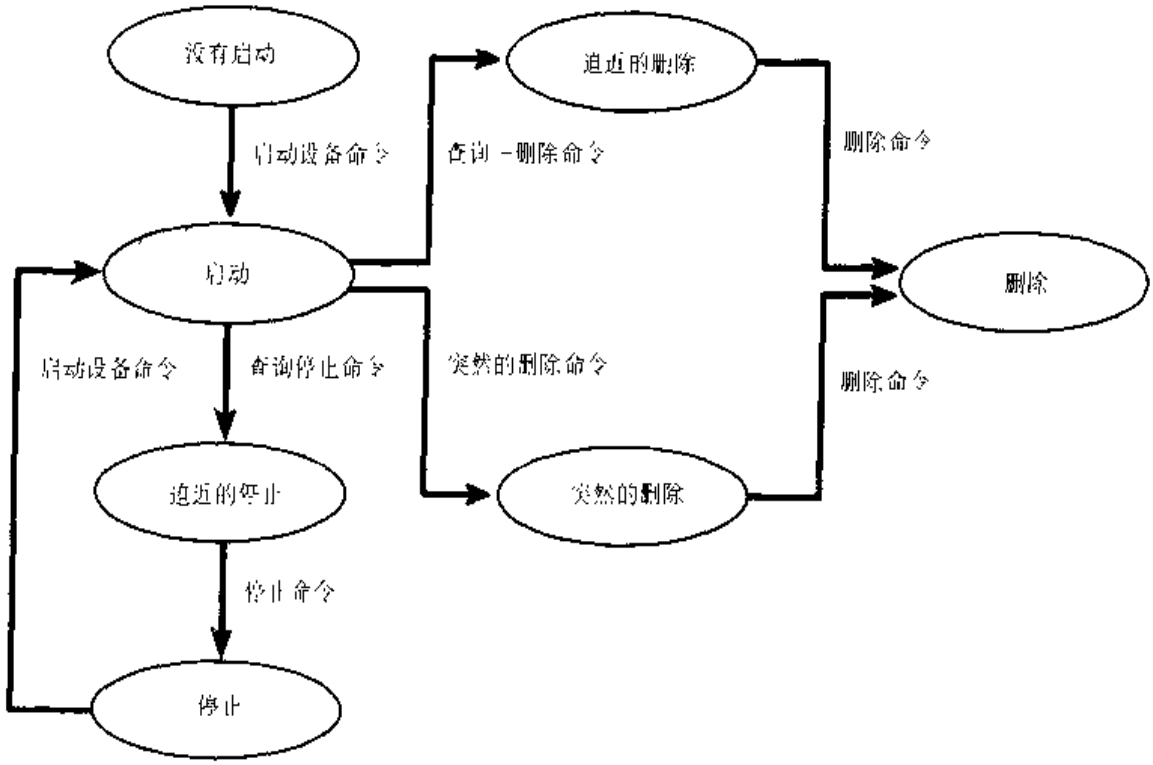


图9-8 简化形式的状态转移表

9.2.4 电源管理器

就象Windows 2000的即插即用功能需要系统的硬件支持那样，它的电源管理性能也需要硬件支持。这些硬件必须遵从高级配置与电源接口（Advanced Configuration and Power Interface，ACPC）规范（从www.teleport.com/~acpi/spec.htm可以得到）。作为这种需求的一个结果是，计算机的BIOS（Basic Input Output System）以及开机时运行的代码都必须遵从ACPI标准。大多数在1998年底以后制造的X86计算机都是ACPI兼容的。

注意 有些计算机特别是比较旧的计算机，不遵循ACPI标准。相反，它们遵从比较旧的高级电源管理（Advanced Power Management，APM）标准。该标准授权的电源管理能力比ACPI少。Windows 2000为APM系统提供有限的电源管理，但我们在此不讨论该主题的细节。本节主要集中Windows 2000在ACPI计算机上的表现。

ACPI标准为系统和设备定义了各种不同的电源级别。表9-2描述了系统的六种电源状态。分别以S0（完全打开或工作态）到S5（完全关闭）表示它们。每个状态具有以下特性：

- **电源消耗** 计算机耗费的电源总量。
- **软件恢复** 当计算机转移到下一个状态时，它从中所恢复的软件状态。
- **硬件延时** 计算返回到完全打开状态所花费的时间。

状态S1到S4是休眠状态，由于减少了电源消耗，在这些状态中计算机看起来好象关闭了。然而，为了转移到S0状态，计算机在内存或磁盘中仍保留了足够的信息。对于状态S1到S3需要足够的电源以保存计算机内存中的内容，以便转移到状态S0时（当用户或设备唤醒计算机时）电源管理器能从它挂起时离开的地方继续执行。尽管系统可以处于6种电源状态，当

表 9-2 系统电源状态定义

状 态	电源消耗	软件恢复	硬件延时
S0（完全打开）	最大	不可用	无
S1（休眠）	小于 S0，大于 S2	系统回到离开时的地方（返回 S0）	小于 2 秒
S2（休眠）	小于 S1，大于 S3	系统回到离开时的地方（返回 S0）	2 秒或更多
S3（休眠）	小于 S2；关闭处理器	系统回到离开时的地方（返回 S0）	约 2 秒
S4（冬眠）	对电源按钮和唤醒电路淌流电流	系统从保存的冬眠文件中重新启动并回到冬眠前离开的地方（返回 S0）	长且无定义
S5（完全关闭）	对电源按钮淌流电流	系统启动	长且无定义

系统移到状态 S4 时，电源管理器将压缩的内存内容保存到命名为 Hiberfile.sys 的睡眠文件中。该文件在引导卷的根目录下，其大小足以容纳所有未压缩的内存内容（压缩是为了减少磁盘 I/O 并改善睡眠和从睡眠中恢复的性能）。在完成内存的保存之后，计算机将关闭计算机。当用户随后打开计算机时，除了 Ntldr 检查外将出现正常的启动过程并检测存贮在睡眠文件中的内存映像的有效性。如果睡眠文件包含保存的系统状态，Ntldr 将该文件的内容读入内存，然后从睡眠文件记录的内存点处继续执行。

计算机从来不会直接在状态 S1 和 S4 之间转换。相反，它必须先转移到 S0，就象图 9-9 所示的那样。系统从 S1 到 S5 的任何一个状态转移到 S0 的过程称为唤醒，而从 S0 转移到 S1 至 S5 的任一个状态的过程被称为休眠。

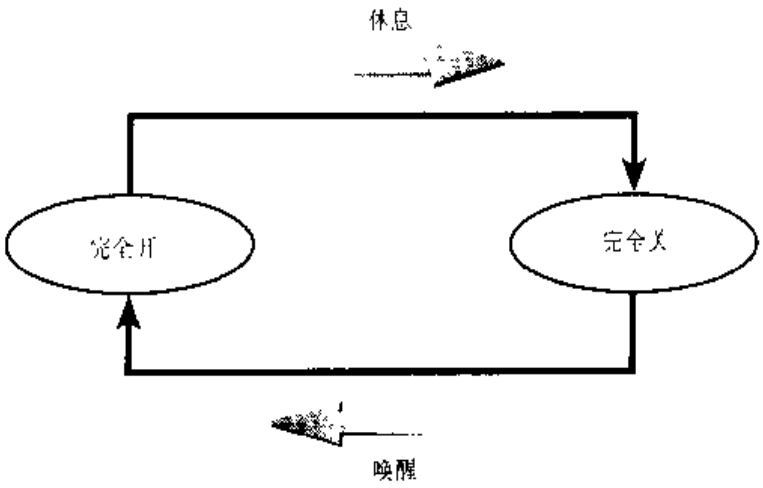


图 9-9 系统电源状态的迁移

尽管系统可以处于 6 种电源状态，但是 ACPI 规定设备只能有 4 种电源状态，从 D0 到 D3。D0 是完全打开的状态，D3 是完全关闭的状态。ACPI 标准将 D1 和 D2 的含义留给各个驱动程

序和设备去定义。只是状态D1消耗的能量必须小于或等于D0；而当设备处于状态D2时，它消耗的能量必须小于或等于D1。Microsoft联合主要的硬件OEM厂商，制定了一系列电源管理参考规范（可从Microsoft的网站www.microsoft.com/hwdev/specs/pmref获得）。这些规范指定某类设备所必须的电源状态（主要的设备类别有：显示、网络、SCSI等等）。对于某些设备，在完全打开和关闭之间不存在中间状态，这将导致这些状态没有定义。

1. 电源管理器的操作

Windows 2000的电源管理策略被分成两部分，分别在电源管理器和单独的设备驱动程序中实现。电源管理器是系统电源策略的拥有者。这种拥有意味着由电源管理器决定在任一给定的时刻哪种系统电源状态是合适的，何时需要睡眠、冬眠或关闭，还意味着由电源管理器指示系统中的设备进行合适的电源状态迁移。电源管理器考虑以下因素以决定何时需要进行系统电源状态的迁移：

- 系统活跃级别。
- 系统电池级别。
- 应用程序的关闭、冬眠或睡眠请求。
- 用户行为，如按下电源按钮。
- Control Panel的电源设置。

当PnP管理器执行设备枚举时，它收集有关设备的信息中有一部分是设备的电源管理能力。驱动程序将汇报它的设备是否支持设备的D1和D2状态。有的驱动程序还汇报从状态D1至D3迁移至状态D0的等待时间或需要的次数。为了帮助电源管理器决定何时进行系统电源状态的迁移，总线驱动程序还将返回一个表，该表实现了每个系统电源状态（从S0到S5）到设备支持的电源状态的映射。该表为每个系统状态列举了可能的最低的设备电源状态。在机器睡眠或休眠时，该表将直接反应各种电源级别的状态。例如，支持四种设备电源状态的总线可能返回如表9-3所示的映射表。当机器没有使用、为减少电源消耗而离开状态S0时，大多数设备驱动程序完全关闭它们的设备。然而，有些设备，如网卡，具有从睡眠状态唤醒系统的功能。该功能以及在该功能下表现的最低的设备电源状态也将在设备枚举期间汇报。

表9-3 系统－设备电源映射的例子

系统电源状态	设备电源状态
S0（完全关闭）	D0（完全打开）
S1（休眠）	D2
S2（休眠）	D2
S3（休眠）	D2
S4（冬眠）	D3（完全关闭）
S5（完全关闭）	D3（完全关闭）

2. 驱动程序电源操作

当电源管理器决定进行系统电源状态迁移时，它将给驱动程序的调度例程发送电源命令。可以有多个驱动程序负责管理一个设备，但只有一个驱动程序被指定为该设备的电源策略的拥

有者。该驱动程序在系统状态的基础上决定设备的电源状态。例如，系统从状态 S0 迁移到 S1，此时驱动程序可能将设备的电源状态从 D0 转移到 D1。为了避免直接通知共享设备管理的其他驱动程序，拥有设备电源管理策略的驱动程序通过 PoRequestPowerIrp 函数请求电源管理器告知其他驱动程序。电源管理器将发送设备电源命令给其他电源调度例程。这样允许电源管理器控制某个时刻系统中活跃的电源命令个数。例如，系统中的某些设备可能要求一定的启动电流。电源管理器确保这些设备不同时上电。

许多电源命令具有相应的查询命令。例如，当系统移向睡眠状态时，电源管理器将首先询问系统中的设备是否可以接受该迁移。如果设备正忙着执行紧急操作或者与其他硬件设备交互就可能拒绝该设备。结果系统将维持当前的系统电源设置。

实验：查看系统电源性能与策略

利用内核调试器命令！pocaps，可以查看计算机系统的电源性能。在运行 Windows 2000 Professional 的与 APCI 兼容的膝上机执行该命令的输出如下：

```
kd> !pocaps
PopCapabilities @ 0x8046edc0
  Misc Supported Features:  PwrButton SlpButton Lid S1 S3 S4 S5
                            Hiberfile FullWake
  Processor Features:       Thermal Throttle (MinThrottle = 03, Scale = 08)
  Disk Features:            SpinDown
  Battery Features:         BatteriesPresent
    Battery 0 - Capacity: 00000000  Granularity: 00000000
    Battery 1 - Capacity: 00000000  Granularity: 00000000
    Battery 2 - Capacity: 00000000  Granularity: 00000000
  Wake Caps
    Ac OnLine Wake:         Sx
    Soft Lid Wake:          Sx
    RTC Wake:               S3
    Min Device Wake:        Sx
    Default Wake:           Sx
```

Misc Supported Features 行报告除 S0（完全打开）之外的系统支持的电源状态 S1、S3、S4 和 S5（它没有实现 S2）。该行还列举一个有效的冬眠文件。冬眠的时候系统将内存保存在该文件中。

下页显示的 Power Options Properties 对话框（在 Control Panel 中选取 Power Optons 可得到该对话框）允许你配置系统电源策略的各个方面。所配置的具体属性依赖于我们刚才查看的系统电源性能。

在 APCI 兼容的膝上机上运行的 Windows 2000 Professional（类似如图 9-10 抓取如下屏幕快照的系统）一般提供最大的电源管理功能。在这样的系统中，你可以设置空闲超时检测，以控制何时关闭监视器、关闭硬盘，进入旁路模式（转移到系统电源状态 S1）并冬眠。此外，Power Options 中的 Advanced 标签允许你指定在按下电源按钮或冬眠按钮或者是关闭膝上机的盖子时系统中与电源有关的行为。

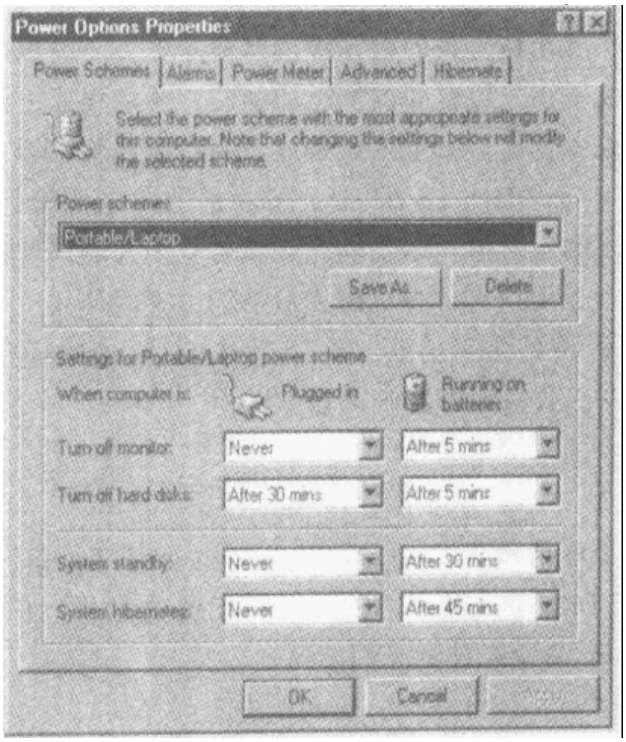


图 9-10 查看系统电源性能与策略

在 Power Options 中配置的设置直接影响系统电源策略中的值。利用调试器的！popolicy 命令可以显示这些值。下面是同一系统中该命令的输出结果：

```
kd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x80469180
  PowerButton:         Off  Flags: 00000003   Event: 00000000   Query UI
  SleepButton:       Sleep  Flags: 00000003   Event: 00000000   Query UI
  LidClose:      Hibernate  Flags: 00000001   Event: 00000000   Query
  Idle:               None  Flags: 00000001   Event: 00000000   Query
  OverThrottled:     Sleep  Flags: c0000004   Event: 00000000   Override
                   NoWakes Critical
  IdleTimeout:    00000000  IdleSensitivity:       32
  MinSleep:             S1  MaxSleep:             S3
  LidOpenWake:          S0  FastSleep:            S1
  WinLogonFlags:  00000000  S4Timeout:        00000000
  VideoTimeout:   00000000  VideoDim:             6e
  SpinTimeout:    00000708  OptForPower:          01
  FanTolerance:         64  ForcedThrottle:      64
  MinThrottle:          19
```

显示的第一行对应于 PowerOptions 中的 Advanced 标签中指定的按钮行为。在该系统中，电源按钮被解释为关闭开关；睡眠按钮则将系统转移到睡眠状态；膝上机盖子的关闭将导致系统冬眠。

输出的最后一行显示的是超时值。这些值是按秒计算的并以十六进制的形式显示。在此列举的值直接对应于你在 Power Options 屏幕快照中可以看到的设置值（膝上机被接上电源）。如果视频超时为零意味着从不关闭监视器；硬盘关闭超时为 0x708 则相当于 1800 秒或者是 30 分钟。

3. 驱动程序对设备电源的控制

除了响应与系统电源状态迁移有关的电源管理器命令外，驱动程序还能够控制设备的电源状态。在某些情况下，当设备在一段时间内都不活动时，驱动程序可能需要减少它所控制的设备的电源消耗。驱动程序或者自己检测空闲设备或者利用电源管理器提供的措施。如果使用电源管理器，那么它调用 PoRegisterDeviceForIdleDetection 函数向电源管理器注册。该函数通知电源管理器用于检测设备空闲的超时值，以及检测到设备空闲时应该使用的设备电源状态。驱动程序指定两个超时值：一个在用户将计算机配置成节能方式时使用；另一个在用户将计算机配置成性能优化时使用。不管设备是否活跃，在调用 PoRegisterDeviceForIdleDetection 后，驱动程序必须调用 PoSetDeviceBusy 函数通知电源管理器。

9.3 I/O 数据结构

有四种主要的数据与 I/O 请求相关：文件对象、驱动程序对象、设备对象和 I/O 请求包（IRP）。这些结构都定义在 DDK 的头文件 Ntddk.h 和 DDK 文档中。你可以利用内核调试器的！file、！drvobj、！devobj 和！irp 命令来显示它们。

9.3.1 文件对象

文件对象是处理文件或设备的内核模式结构。文件对象明显符合 Windows 2000 的对象标准：它们是两个或两个以上用户模式进程的线程可以共享的系统资源；它们可以有名称；它们由基于对象的安全机制所保护；并且它们支持同步。虽然在 Windows 2000 中的大多数共享资源是基于内存的资源，但是 I/O 系统管理的大多数的资源位于物理设备中或者就是实际的物理设备。尽管这有些差异，但在 I/O 系统中的共享资源，像在 Windows 2000 执行程序的其他组件中的一样，都当作对象操作（关于对象管理器的描述，请参阅第 3 章；关于对象安全性的信息，请参阅第 8 章）。

文件对象提供了基于内存的资源的表示法。该表示法遵从以 I/O 为中心的接口，可以对它们进行读写。表 9-4 列举了一些文件对象属性。关于具体字段的声明和大小，请参阅 Ntddk.h 中的 FILE_OBJECT 的结构定义。

表 9-4 文件对象属性

属性	目的
文件名	标识文件对象引用的物理文件
当前字节偏移	标识文件的当前位置（仅对同步 I/O 有效）
共享模式	表示当当前的调用程序正在使用文件时，其他调用程序是否可以打开此文件进行读、写或删除操作
打开模式标记	表示 I/O 是同步的或异步的、高速缓存的或非高速缓存的、顺序的或随机的等等
指向设备对象的指针	表示文件驻留的设备类型
指向卷参数块（VPB）的指针	表示文件驻留的卷或者分区
指向区域对象指针的指针	表示描述映射文件的根结构
指向私用高速缓存映射的指针	标识文件的哪一部分被高速缓存管理器缓存以及它驻留在高速缓存的哪一部分

当调用程序打开一个文件或简单设备时，I/O 管理器将返回一个文件对象句柄。图 9-11 说明了打开一个文件时所发生的情况。

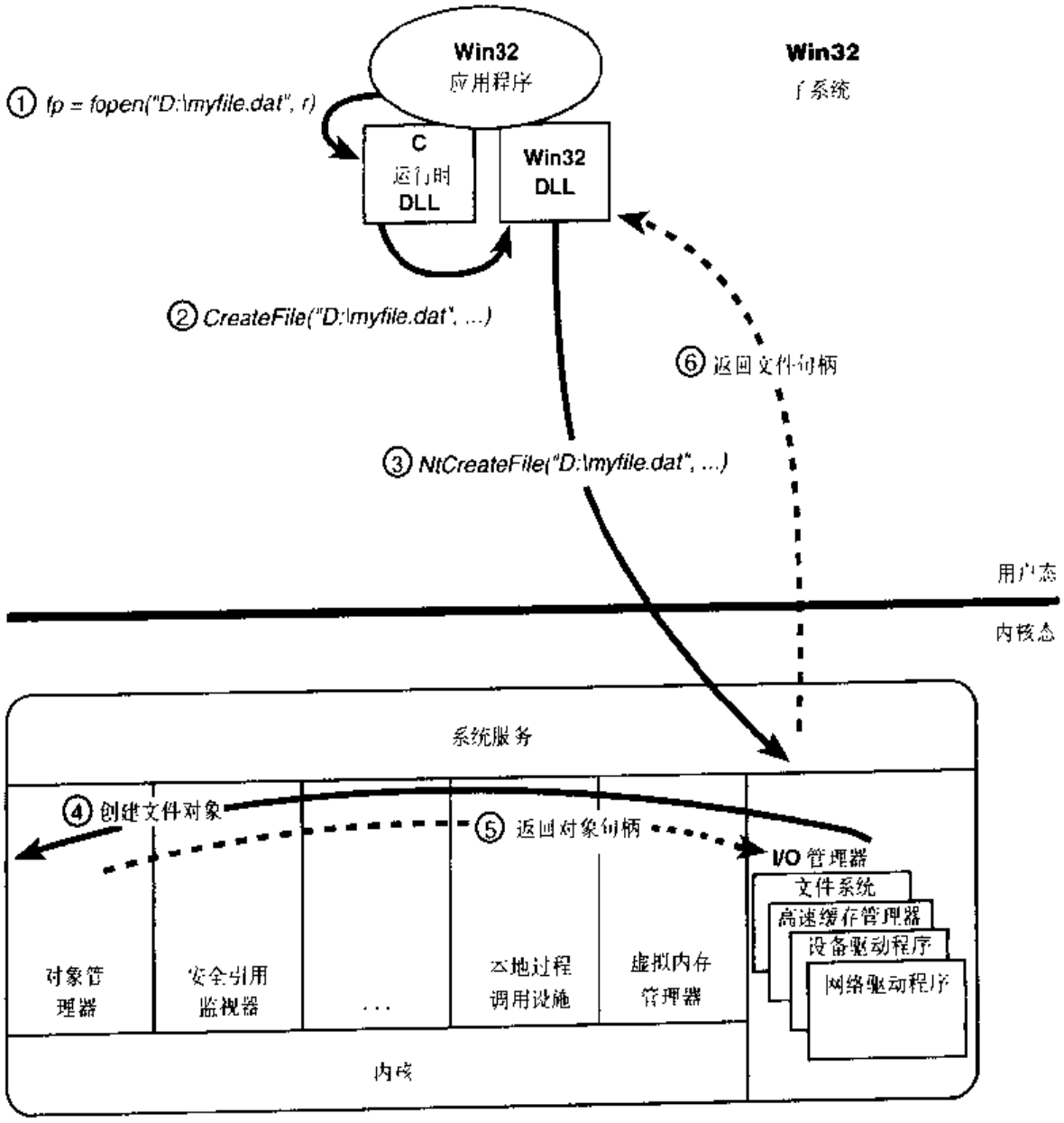


图 9-11 打开一个文件对象

在这个例子中，1）一个 C 程序调用运行时库函数 fopen。2）该函数接着调用 Win32 的 CreateFile 函数。3）Win32 子系统 DLL（在这里是 Kernel32.dll）然后调用 Ntdll.dll 中的本机 NtCreateFile 函数。在 Ntdll.dll 中的例程包含引发到内核模式系统服务调度程序的转换的适当指令。4）然后系统服务调度程序在 Ntoskrnl.exe 中调用真正的 NtCreateFile 例程（关于系统服务调度的详细信息，请参阅第 3 章）。

象其他的可执行对象一样，文件对象由包含访问控制表（ACL）的安全描述符保护。I/O 管理器查询安全子系统来决定文件的 ACL 是否允许进程去访问它的线程正在请求的文件。如果允许，(5, 6) 对象管理器将准予访问，并把它返回的文件句柄和授予的访问权联系起来。如果这个线程或在进程中的另一个线程需要执行额外的最初请求所没有指定的操作，那么线程就必须打

开另一个句柄。这将导致另一次安全检查（有关对象保护的详细信息，请参阅第 8 章）。

因为文件对象是一个基于内存的可共享资源的表示，而不是资源本身，所以它有别于其他的可执行对象。一个文件对象只包括对于对象句柄来说是唯一的数据，而文件本身将包括可被共享的数据或文本。每当一个线程打开一个文件句柄时，带有一组新的具体句柄属性的一个新的文件对象就被创建。例如，属性字节偏移量指的是在文件中下一次将要使用那个句柄做读取或写入操作的位置。每一个为文件打开的句柄都有专用的字节偏移量，即使该文件是被共享的。除了当一个进程复制一个文件句柄给另一个进程（使用 Win32 函数 DuplicateHandle）或当一个子进程从它的父进程那里继承一个文件句柄外，文件对象对于进程来说是唯一的。在这些情况下，两个进程有单独的指向同一文件对象的句柄。

尽管一个文件句柄对一个进程可能是唯一的，但在底层的物理资源却不是这样。因此，当使用任何共享的资源时，线程必须同步它们对共享文件、文件目录和设备的访问。例如，如果一个线程正在写入一个文件，当它打开文件句柄时，它应该指定独占的写访问去防止其他的线程在同一时间对该文件写入。作为一种选择，线程可以写文件期间通过 Win32 LockFile 函数锁定文件的某些部分。

9.3.2 驱动程序对象和设备对象

当线程打开一个文件对象句柄时，I/O 管理器必须从文件对象的名称决定它应该调用哪个（或哪些）驱动程序来处理请求。而且，I/O 管理器必须在线程下一次使用同一个文件句柄时可以查找这个信息。以下系统对象满足这些要求：

■“驱动程序对象”在系统中代表一个独立的驱动程序。I/O 管理器从驱动程序对象获取驱动程序的调度例程（入口点）的地址。

■“设备对象”在系统中代表一个物理或逻辑设备并描述了它的特征，例如它所需要的缓冲区对齐方式以及用来保存到来的 I/O 请求包的设备队列的位置。

当驱动程序被加载到系统中时，I/O 管理器将创建一个驱动程序对象，然后它调用驱动程序的初始化例程（如，Driver - Entry），该例程把驱动程序的入口点填入对象属性中。

加载之后，驱动程序可以任何时候调用 IoCreateDevice 创建表示设备的设备对象，甚至是驱动程序接口。然而大多数 Windows 2000 和 WDM 驱动程序，当 PnP 管理器通知它们呈现所管理的设备时，利用设备添加例程创建设备。另外，对于遗留驱动程序，当 I/O 管理器调用它们的初始化例程时，它们通常创建设备对象。当最后一个设备对象被删除，不再有剩余的设备引用时，I/O 管理器将卸载驱动程序。

当驱动程序创建设备对象时，驱动程序可以给设备指定一个名称。名称将设备对象置于设备对象管理器的名字空间中。驱动程序可以显式地指定一个名字，也可以让 I/O 管理器自动产生一个（对象管理器的名字空间在第 3 章介绍）。一般约定设备对象置于名字空间的 \ Device 目录中。应用程序不可使用 Win32 API 函数访问该目录。

注意 有些驱动程序将设备对象放到不同于 \ Device 的目录中。例如，Windows 2000 Logical Disk Manager 的卷管理器在 \ Device \ HarddiskDmVolumes 目录中创建代表磁盘分区的设备对象。关于存储体系结构的描述以及存储驱动程序对设备对象的使用，请参阅第 10 章。

如果驱动程序需要让应用程序能够打开设备对象，那么它必须针对 \ Device 目录中的设备对象的名字在 \ ?? 目录中创建一个符号链接。遗留驱动程序以及非面向硬件的驱动程序（如文件系统驱动程序）一般用众所周知的名字创建符号链接（如，\ Device \ Hardware2）。由于众所周知的名字在硬件动态地出现或消失的环境中工作得并不好，因此即插即用的驱动程序通过调用 IoRegisterDeviceInterface 函数而暴露一个或多个接口，指定一个 GUID（全局唯一标识符）表示暴露的功能类型。GUID 是一个 128 位的值。你可以利用 DDK 和 Platform SDK 中一个称为 Guidgen 的工具产生该值。在 128 位表示的数值范围内，Guidgen 产生的 GUID 从统计上讲将永远并且是全局唯一的。

IoRegisterDeviceInterface 确定与设备实例相关的符号链接。但是驱动程序必须在 I/O 管理器真正创建该链接之前调用 IoRegisterDeviceInterface 打开设备接口。驱动程序通常在 PnP 管理器向它发送 start - device 命令打开设备时做该项工作。

需要打开用 GUID 表示的设备对象的应用程序可以在用户空间中调用即插即用设置函数，如 SetupDiEnumDeviceInterfaces，枚举特定的 GUID 所表示的设备接口，并获得用来打开该设备对象的符号链接。对于由 SetupDiEnumDeviceInterfaces 所报告的设备，应用程序执行 SetupDiGetDeviceInterfaceDetail 获取有关该设备的其他信息，如自动生成的名字。从 SetupDiGetDeviceInterfaceDetail 中获取了设备名字后，应用程序可以执行 Win32 函数 CreateFile，打开设备并获得句柄。

实验：查看 \ Device 目录

你可以利用 Winobj 工具，该工具包含在所附 CD 的 \ Sysint \ Winobj. exe 中，或者是内核调试器命令 ! object 查看对象管理器名字空间中的 \ Device 下的设备名称。如图 9-12 显示的是一个 I/O 管理器分配的符号链接。该链接指向 \ Device 中一个具有自动生成的名字的设备对象。

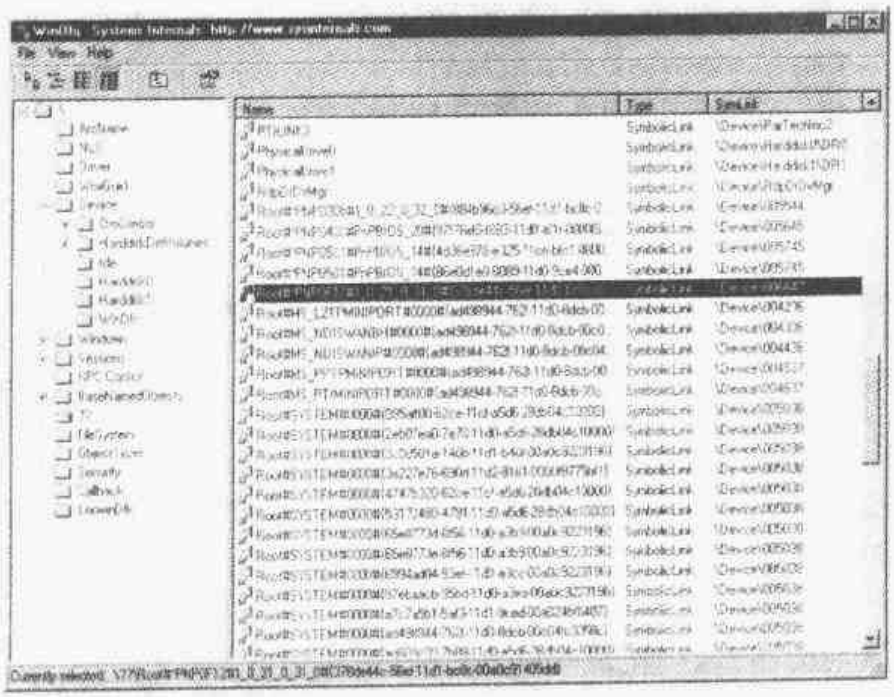


图 9-12 查看 \ Device 目录

当运行！object 内核调试器命令并指定 \ Device 目录时，可以得到类似于下面的输出：

```
kd> !object \device
Object: 81a8e170  Type: (81ab6120) Directory
    ObjectHeader: 81a8e158
    HandleCount: 0  PointerCount: 198
    Directory Object: 81a914d0  Name: Device
    9 symbolic links snapped through this directory
    HashBucket[ 00 ]: 81a6de10 Device 'KsecDD'
                      819c43f0 Device 'Beep'
                      81a6d6d0 Device 'Ndis'
                      81a61030 Device '00000019'
                      81aa7830 Device '003018'
                      81aa7a30 Device '002918'
    HashBucket[ 01 ]: 817c3c70 Device '00000026'
                      819c3d90 Device 'Netbios'
    HashBucket[ 02 ]: 818d1850 Device 'KSENUM#00000001'
                      81966890 Device 'Ip'
    HashBucket[ 03 ]: 818c5b70 Device 'KSENUM#00000002'
                      81a31038 Device 'Video0'
                      81a4fc70 Device 'KeyboardClass0'
    HashBucket[ 04 ]: 819d4410 Device 'NDProxy'
                      819c0040 Device 'Video1'
    HashBucket[ 05 ]: 817e7650 Device 'PcCard0-0'
                      819c70d0 Device 'Eawdmfd'
                      81a2aa50 Device
                      '{13199AF4-86FA-48C2-8074-468CA06AF86C}'
                      819c0ce0 Device 'Video2'
                      81a52040 Device 'Serial0'
                      81a6cbb0 Device 'PointerClass0'
                      81a215f0 Device '0000000a'
    HashBucket[ 06 ]: 817cb8c0 Device 'Serial1'
                      81900570 Device 'DebugMessageDevice'
                      81a277d0 Device 'USBPDO-0'
                      81a5e030 Device 'CompositeBattery'
```

当执行！object 命令并指定一个对象管理器目录对象对，内核调试器将按照对象管理器内部的组织方式转储该目录的内容。为了快速查找，目录将对象存储在基于对象名的散列表中，因此输出显示存储在该散列表中每个桶中的对象。

当文件被打开时，文件名中包含了该文件所驻留的设备对象名。例如，名称 \ Device \ Floppy0 \ Myfile.dat 是指软盘驱动器 A 上的文件 myfile.dat。子字符串 \ Device \ Floppy0 是 Windows 2000 内部设备对象的名称，代表那个软盘驱动器。当打开 Myfile.dat 时，I/O 管理器创建一个文件对象，并在文件对象中存储一个 Floppy0 设备的指针，然后给调用程序返回一个文件句柄。此后，当调用程序使用文件句柄时，I/O 管理器能够直接找到 Floppy0 设备对象。请记住，在 Win32 应用程序中不能使用 Windows 2000 内部设备名称。相反，设备名称必须出现在对

象管理器的名称空间中一个特定的目录 \ ?? 中（在 Windows NT 4 之前的名称是 \ DosDevices）。该目录包括到真实的 Windows 2000 内部设备名称的符号链接。设备驱动程序负责创建该目录中的链接以使 Win32 应用程序可以访问它们的设备。使用 Win32 的 QueryDosDevice 和 DefineDosDevice 函数，你可以以编程的方式来检查甚至改变这些链接。

实验：查看 Win32 设备名称到 Windows 2000 设备名称的映射

你可以利用包含在配套 CD（Sysint \ Winobj.exe）中的 Winobj 工具来查看定义 Win32 设备名称空间的符号链接。运行 Winobj 并单击 \ ?? 目录，可以得到如图 9-13 的显示：

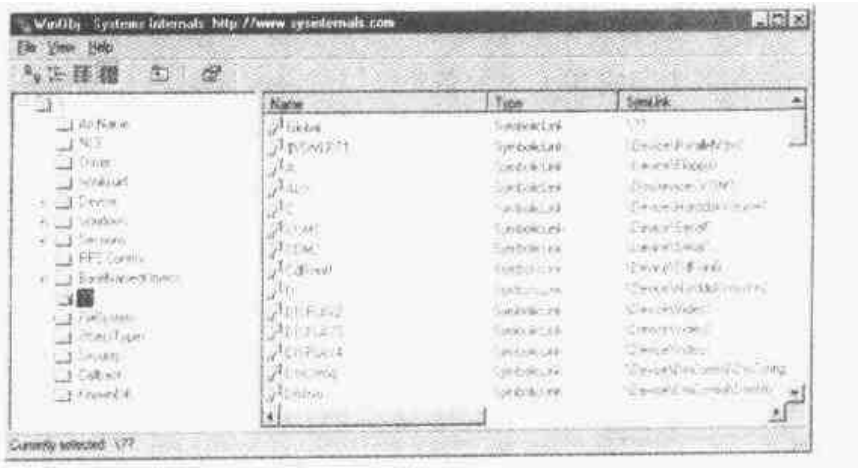


图 9-13 查看定义 Win32 设备名称空间的符号链接

注意符号链接在右边。试着双击设备 C:，你将看到类似如图 9-14 的显示：

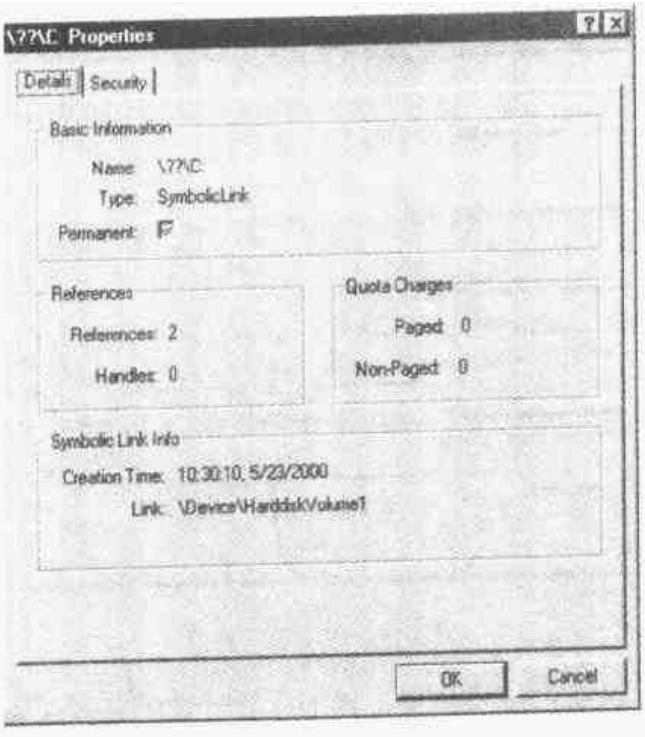


图 9-14 查看链接属性

C：是一个到内部设备 \ Device \ HarddiskVolume1，或者是系统中第一个硬盘的第一个卷的符号链接。Winobj 中的 COM1 项是到 \ Device \ Serial0 的符号链接，等等。请试着在命令提示符下用 subst 命令创建自己的链接。

如图 9-15 所示，设备对象反过来指向它自己的驱动程序对象，这样 I/O 管理器就知道在接收一个 I/O 请求时应该调用哪个驱动程序。它使用设备对象找到代表服务于该设备驱动程序的驱动程序对象，然后利用在初始请求中提供的功能代码来索引驱动程序对象。每个功能代码都对应于一个驱动程序的入口点（显示在图 9-15 中的功能代码将在本章后面的“IRP 堆栈查找”中介绍）。

驱动程序对象通常有多个与它相关的设备对象。设备对象列表代表驱动程序可以控制的物理设备、逻辑设备和虚拟设备。例如，硬盘的每个分区都有一个独立的包含具体分区信息的设备对象。然而，相同的硬盘驱动程序被用于访问所有的分区。当一个驱动程序从系统中被卸载时，I/O 管理器就会使用设备对象队列来确定哪个设备由于删除了驱动程序而受到影响。

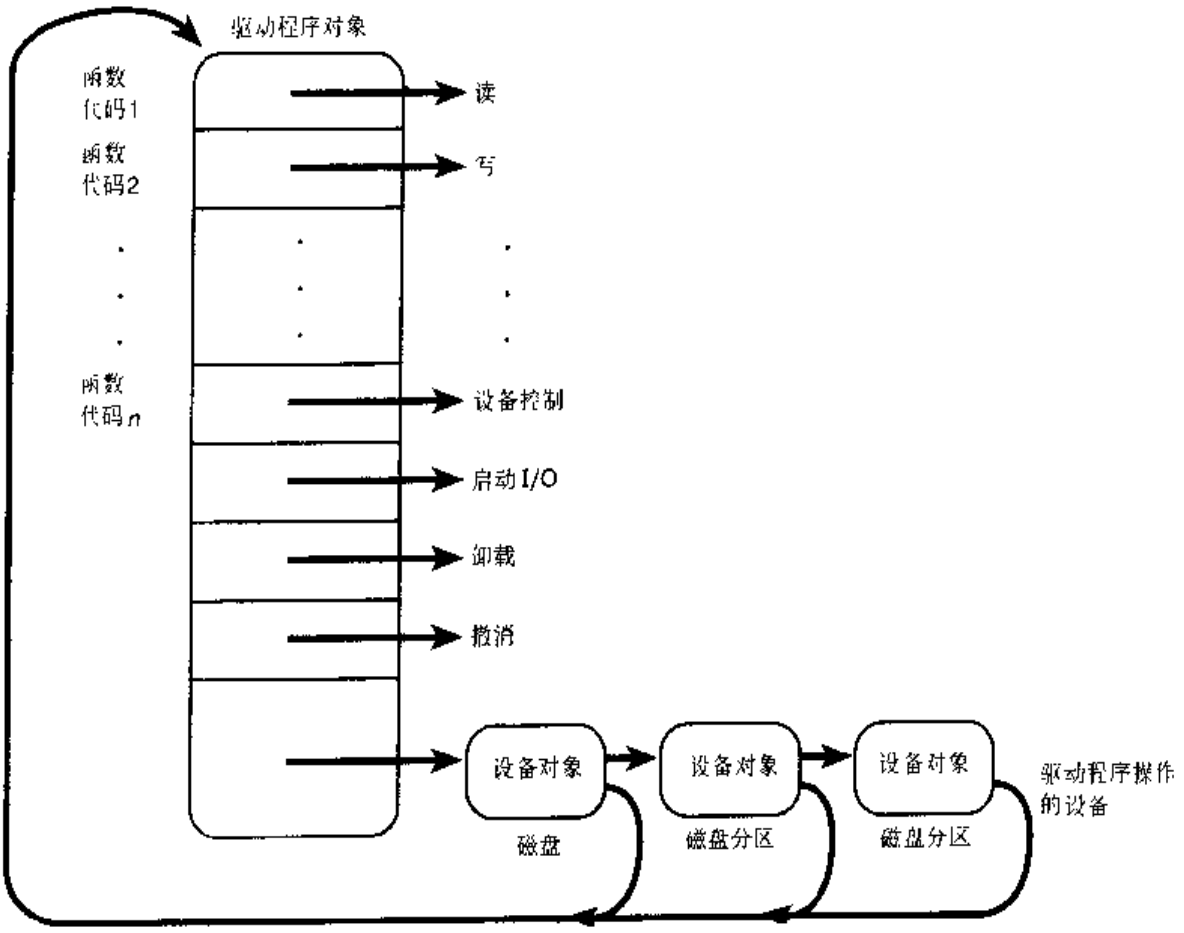


图 9-15 驱动程序对象

使用对象来记录关于驱动程序的信息，使 I/O 管理器不需要了解单个驱动程序的详细资料。I/O 管理器只需依靠指针来查找驱动程序。这样就提供分层的能力，并允许很容易地加载新的驱动程序。如果系统配置改变，用不同的对象代表设备和驱动程序也使得 I/O 系统可以很

容易地指派驱动程序来控制附加的或不同的设备。

9.3.3 I/O请求包

I/O请求包（IRP）是I/O系统用来存储处理I/O请求所需信息的地方。当线程调用I/O服务时，I/O管理器就构造一个IRP来表示在整个I/O系统进展中要进行的操作。如有可能，I/O管理器从每个处理器的两个非页式备份表（look - aside list）之一分配IRP：小的IRP备份表用一个堆栈位置存储（IRP堆栈被简短描述），大的IRP备份表用8个堆栈位置包含IRP。如果IRP需要的堆栈多于8个，I/O管理器将从非页交换区中分配IRP。在分配并初始化IRP之后，I/O管理器在IRP中存储一个调用程序文件对象的指针。

实验：显示驱动程序与设备对象

可以利用内核调试器命令！drvobj和！devobj显示相应的驱动程序和设备对象。在下面的例子中，查看的是键盘类驱动程序的驱动程序对象，以及它的孤立的设备对象。

```
kd> !drvobj kbdclass
Driver object (81869cb0) is for:
 \Driver\Kbdclass
Driver Extension List: (id , addr)

Device Object list:
81869310

kd> !devobj 81869310
Device object (81869310) is for:
 KeyboardClass0 \Driver\Kbdclass DriverObject 81869cb0
Current Irp a57a0e90 RefCount 0 Type 0000000b Flags 00002044
DevExt 818693c8 DevObjExt 818694b8
ExtensionFlags (0000000000)
AttachedDevice (Upper) 81869ie0 \Driver\Ctrl2cap
AttachedTo (Lower) 81869500 \Driver\i8042prt
Device queue is busy -- Queue empty.
```

注意，！devobj命令还显示你所查看的任何设备对象的地址和名称，这些设备对象位于你所查看的对象之上（行AttachedTo）以及所指定的对象之上（行Attched Device）。

图9-16显示的是一个I/O请求的例子。该例子说明了IRP与前面几节描述的文件、设备和驱动程序对象之间的关系。尽管这个例子显示了对一个单层设备驱动程序的I/O请求，但大多数I/O操作都不是这样直接进行的；它们往往要涉及一个或多个分层的驱动程序。（这种情况将在这一节稍后说明）。

9.3.4 IRP堆栈位置

IRP由两部分组成：固定头（通常称作IRP的体）和一个或多个堆栈位置。固定部分信息包括：请求的类型和大小、请求是同步还是异步、用于缓冲I/O的指向缓冲区的指针以及随请求的进展而变化的状态信息。IRP的堆栈位置包括一个功能码（由一个主码和次码组成）、功能

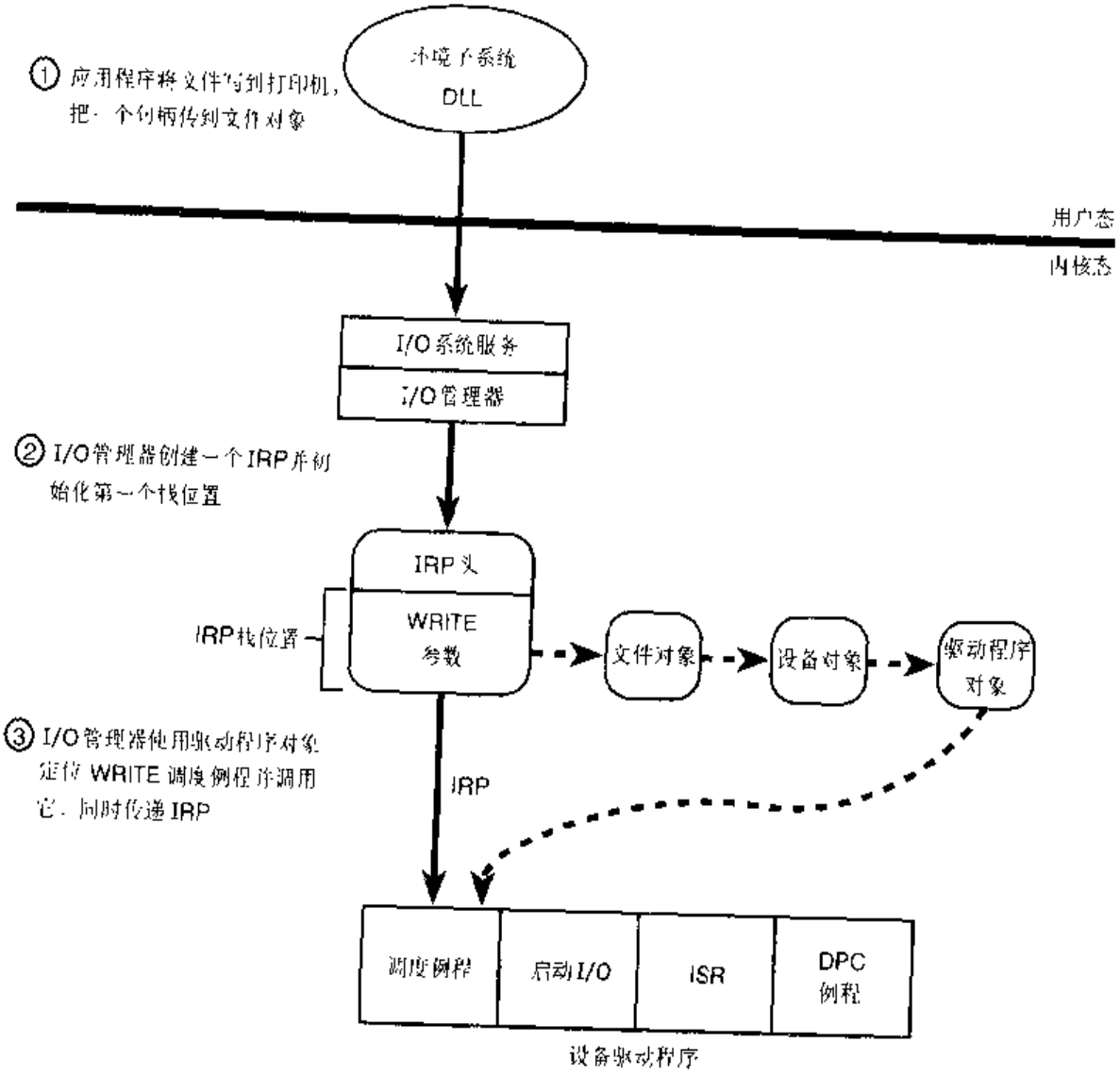


图 9-16 涉及单层驱动程序 I/O 请求的数据结构

特定的参数和一个指向调用程序文件对象的指针。主功能代码（major function code）指定传递 IRP 给驱动程序时 I/P 管理器唤醒的驱动程序的调度例程的名称。可选的次功能代码有时是作为主功能代码的修改者。电源和即插即用命令总是带有次功能代码。

大多数驱动程序都指定调度例程处理可能的主功能代码的一个子集，其中包括创建（打开）、读、写、设备 I/O 控制、电源、即插即用、系统（对于 WMI 命令）以及关闭（关于完整的主功能代码列表请参阅下面的实验）。文件系统驱动程序某类驱动程序的一个例子，该类驱动程序经常用函数填充几乎所有的调度入口点。I/O 管理器将所有驱动程序没有填充的调度入口点设置成它自己的 IopInvalidDeviceRequest，该函数将返回一错误代码给调用程序，指明对该设备指定的功能无效。

实验：驱动程序调度例程

在内核调试器命令！drvobj 中，在驱动程序对象名称的后面输入 2，可以查看该驱动程序为它的调度例程定义的功能列表：

```
kd> !drvobj kbdclass 2
Driver object (81869cb0) is for:
 \Driver\Kbdclass
Dispatch routines:
[00] IRP_MJ_CREATE                      edf0866e
     kbdclass!KeyboardClassCreate
[01] IRP_MJ_CREATE_NAMED_PIPE           80425354
     ntoskrnl!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       edf088ec
     kbdclass!KeyboardClassClose
[03] IRP_MJ_READ                        edf08b1c
     kbdclass!KeyboardClassRead
[04] IRP_MJ_WRITE                       80425354
     ntoskrnl!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           80425354
     ntoskrnl!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             80425354
     ntoskrnl!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    80425354
     ntoskrnl!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      80425354
     ntoskrnl!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               edf085d4
     kbdclass!KeyboardClassFlush
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    80425354
     ntoskrnl!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      80425354
     ntoskrnl!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           80425354
     ntoskrnl!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         80425354
     ntoskrnl!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              edf0a8ec
     kbdclass!KeyboardClassDeviceControl
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     edf0a380
     kbdclass!KeyboardClassPassThrough
[10] IRP_MJ_SHUTDOWN                    80425354
     ntoskrnl!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                80425354
     ntoskrnl!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     edf084b6
     kbdclass!KeyboardClassCleanup
[13] IRP_MJ_CREATE_MAILSLOT             80425354
     ntoskrnl!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              80425354
     ntoskrnl!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                80425354
     ntoskrnl!IopInvalidDeviceRequest
```

```
[16] IRP_MJ_POWER                       ecf0b5e2
     kbdclass!KeyboardClassPower
[17] IRP_MJ_SYSTEM_CONTROL              ecf0bbfe
     kbdclass!KeyboardClassSystemControl
[18] IRP_MJ_DEVICE_CHANGE               80425354
     ntoskrnl!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 80425354
     ntoskrnl!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   80425354
     ntoskrnl!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         edf09168
     kbdclass!KeyboardPnP
```

在处于活动状态时，每个 IPR 都存储在与请求该 I/O 的线程相关的 IRP 列表中。如果一个线程结束了或是终止了未完成的（outstanding）I/O 请求，这种安排就允许 I/O 系统查找并取消所有未完成的 IRP。

9.3.5 IRP 缓冲区管理

当应用程序或设备驱动程序使用系统服务 NtReadFile、NtWriteFile 和 NtDeviceIoControlFile（相应的 Win32 API 函数为 ReadFile 、WriteFile 与 DeviceIoControl）间接创建 IRP 时，I/O 管理器将决定它是否需要参与调用程序的输入或输出缓冲区管理。I/O 管理器执行三种类型的缓冲区管理：

■ **缓冲 I/O** I/O 管理器在非页交换区分配与调用程序同样大小的缓冲区。对于写操作，I/O 管理器在创建 IRP 时将调用程序的缓冲区数据复制到所分配的缓冲区中。对于读操作，I/O 管理器在 IRP 完成的时候将数据从所分配的缓冲区复制到用户的缓冲区，并释放所分配的缓冲区。

■ **直接 I/O** 当 I/O 管理器创建 IRP 时，它将锁定内存中的用户缓冲区（使其变为非页式）。当 I/O 管理器使用完 IRP 时，它将该缓冲区解锁。I/O 管理器在内存描述符列表（MDL）中存储该内存的描述。MDL 指定缓冲区占用的物理内存（关于 MDL 的详细情况请参阅 Windows 2000 DDK）。执行直接内存访问的设备只需要内存的物理描述，因此 MDL 对这样的设备的操作是足够的（支持 DMA 的设备直接在设备和计算机内存之间传输数据，不需要使用 CPU）。然而，如果驱动程序必须访问内存中的内容，它可以将缓冲区映射到系统的地址空间。

■ **无关 I/O** I/O 管理器不执行任何缓冲区管理。相反，将缓冲区管理留给设备的驱动程序去考虑。驱动程序可以手工选择执行 I/O 管理器在其他缓冲区管理中执行的步骤。

对于每种缓冲区管理，I/O 管理器将 IRP 中可使用的引用指向 I/O 缓冲区的位置。I/O 管理器执行的缓冲区管理类型依赖与驱动程序为每种操作所要求的缓冲区管理类型。设备 I/O 控制操作（由 NtDeviceIoControl 执行的操作）是用驱动程序定义的 I/O 控制码指定的。当启动包含控制码的 IRP 时，控制码中包括 I/O 管理器应该执行的缓冲区管理描述。

当调用程序的传输请求小于一页（4K）时，驱动程序通常使用缓冲的 I/O，而对于大的请求则使用直接 I/O。一页大约为该缓冲区的大小，在这个大小值点上，缓冲 I/O 的拷贝操作之

间的权衡大致与直接I/O执行的内存锁定管理负载相匹配。文件驱动程序通常使用I/O，因为不存在缓冲区管理问题，数据可以从文件系统的高速缓存中拷贝到调用程序的原始缓冲区中。大多数驱动程序不使用无关I/O的主要原因是当调用程序的线程正在执行时，只有调用程序的缓冲区指针有效。如果驱动程序必须从（向）ISR或DPC例程的设备传输数据，那么它必须确保调用程序的数据从进程的任何地方都可访问。这意味着缓冲区必须具有系统虚拟地址。

实验：查看IRP和线程IRP队列

你可以使用内核调试器命令！thread检查线程挂起的IRP。一个几乎一直具有挂起IRP的线程是Win32子系统的键盘输入线程。查找该线程，请执行内核调试器命令！stacks，并在Csrss中查找该线程。该线程列举在Win32k RawInputThread函数的前面。

```
kd> !stacks
Proc.Thread  Thread   ThreadState  Blocker
                                   [System]
   8.000004  fe504a60  BLOCKED      ntoskrnl!MmZeroPageThread+0x5f
   8.00000c  fe503ce0  BLOCKED      ntoskrnl!ExpWorkerThread+0x73
   8.000010  fe503a60  BLOCKED      ntoskrnl!ExpWorkerThread+0x72
   8.000014  fe5037e0  BLOCKED      ntoskrnl!ExpWorkerThread+0x73
   8.000018  fe503560  BLOCKED      ntoskrnl!ExpWorkerThread+0x73
   8.00001c  fe5032e0  BLOCKED      ntoskrnl!ExpWorkerThread+0x73
   8.000020  fe502020  BLOCKED      ntoskrnl!ExpWorkerThread+0x73
   8.000024  fe502da0  BLOCKED      ntoskrnl!ExpWorkerThread+0x73
:
                                   [csrss.exe]
  c0.0000c4  ff2d5020  BLOCKED      ?? Kernel stack not resident ??
  c0.0000c8  ff2d5d80  BLOCKED      ?? Kernel stack not resident ??
  c0.0000cc  ff2d4820  BLOCKED      ntdll+0xaaa7
  c0.0000d0  ff2d4460  BLOCKED      ?? Kernel stack not resident ??
  c0.0000d4  ff2d4120  BLOCKED      ?? Kernel stack not resident ??
  c0.0000dc  ff2cfda0  BLOCKED      ntdll+0xaaa7
  c0.00007c  ff2cbc40  BLOCKED      win32k!RawInputThread+0x3c2
  c0.0000e0  ff2cb480  BLOCKED      win32k!xxxMsgWaitForMultipleObjects+0x92
:
```

然后利用该程的地址（第2列）执行！thread命令：

```
kd> !thread ff2cbc40
THREAD ff2cbc40  Cid c0.7c  Teb: 00000000  Win32Thread: a20836e8
WAIT: (WrUserRequest) KernelMode Alertable
    ff2cc1a0  SynchronizationEvent
    ff2cbb28  SynchronizationEvent
    ff2cbae8  NotificationTimer
    ff2cbb68  SynchronizationEvent
IRP List:
    fee25388: (0006,0100) Flags: 00000970  Mdl: 00000000
Not impersonating
Owning Process ff2d9020
WaitTime (seconds)      15420441
Context Switch Count    328592
```

```
UserTime                  0:00:00.0000
KernelTime                0:00:00.0721
Start Address win32k!RawInputThread (0xa000c0b0)
Stack Init f20d0000 Current f20cfaf0 Base f20d0000 Limit f20cd000 Call 0
Priority 19 BasePriority 13 PriorityDecrement 0 DecrementCount 0

ChildEBP RetAddr  Args to Child
f20cfb08 8042d33d 80400b46 00000001 00000000 ntoskrnl!KiSwapThread+0xc5
f20cfb3c a000c3f3 00000004 ff2cbc08 00000001
 ntoskrnl!KeWaitForMultipleObjects+0x266
f20cfda8 804524f6 00000002 00000000 00000000 win32k!RawInputThread+0x3c2
f20cfddc 80465b62 a000c0b0 f1caf7d0 00000000
 ntoskrnl!PspSystemThreadStartup+0x69
00000000 00000000 00000000 00000000 00000000 ntoskrnl!KiThreadStartup+0x16
```

输出的样本显示该线程 IRP 列表中包含一个悬挂的 IRP。如果对该 IRP 使用！irp 命令，你可能看到类似于下面的输出：

```
kd> !irp fee25388
Irp is active with 4 stacks 4 is current (= 0xfee25440)
 No Mdl System Buffer = ff0acc48 Thread ff2cbc40:  Irp stack trace.
     cmd  flg cl Device   File     Completion-Context
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

            Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000

            Args: 00000000 00000000 00000000 00000000
 [  0, 0]   0  0 00000000 00000000 00000000-00000000
           Args: 00000000 00000000 00000000 00000000
>[  4, 0]   0 e1 ff43b390 ff2cb928 00000000-00000000
          \Driver\Kbdclass
           Args: 00000078 00000000 00000000 00000000
```

该输出显示该 IRP 具有 4 个堆栈位置，当前正被键盘类驱动程序所拥有，在 IRP 完成之前，键盘类驱动程序等待键盘输入。

另一个与 IRP 相关的调试器命令，！irpfind，允许你查看系统中所有挂起的 IRP。

```
kd> !irpfind
Scanning large pool allocation table for Tag: Irp?

Searching NonPaged pool (fe314000 : fe52c000) for Tag: Irp?

  Irp    [ Thread ] irpStack: (Mj,Mn)   DevObj  [Driver]
fe4f0568 [00000000] irpStack: ( 0, 0)  ff453790 [ \Driver\Cdrom]
fe4f22e8 [fe5028a0] irpStack: ( e, 0)  fe4f13b0 [ \Driver\Ftdisk]
fe4f3b28 [fe5028a0] irpStack: ( e, 0)  fe4f33f0 [ \Driver\Ftdisk]
fe4fdf68 [ff2ae8c0] irpStack: ( e, 0)  ff3f45f0 [ \Driver\Net8T]
  0xff2c9780.
fe50b6a8 [00000000] Irp is complete (CurrentLocation 3 > StackCount 2)
fe50d648 [00000000] irpStack: ( f, 0)  ff4526f0 [ \Driver\openhci]
```

```
fe513e68 [fe5028a0] irpStack: ( e, 0)  fe4f3690 [ \Driver\Ftdisk]
fe515e68 [fe5028a0] irpStack: ( e, 0)  fe4f2570 [ \Driver\Ftdisk]

Searching NonPaged pool (fe52c000 : ffb7f000) for Tag: Irp?

fefa4848 [ff1124e0] irpStack: ( e, 9)  ff2b3490 [ \Driver\AFD]
fefcc2e8 [ff0ecda0] irpStack: ( 3, 0)  ff3f3e70 [ \FileSystem\Npfs]
:
```

9.3.6 I/O完成端口

编写高性能的服务器应用程序要求实现一个高效的线程模型。如果处理客户请求的线程太多或太少都会导致性能问题。例如，如果服务器创建单个线程处理所有请求，那么用户可能变得饥饿，因为一次处理一个请求服务器将会阻塞。单个线程可以同时处理多个请求，当I/O操作启动时，从一个切换到另一个，但是这种结构将带来明显的复杂性，并且不能利用多处理器的优势。另一个极端是服务器可以处理一个大的线程交换区，以致每个用户请求都由一个具体的线程处理，这将导致线程失效（thread thrashing）。在这种情况下许多线程被唤醒，执行CPU处理，阻塞等待I/O，然后在请求完成之后又一次阻塞等待新的请求。如果没有别的情况，太多的线程将导致过多的环境切换，因为调度程序不得不将处理器时间在多个活动线程间分割。

服务器的目标就是使线程避免不必要的阻塞，尽可能减少环境切换，与此同时利用多线程最大化并行机制。理想的情况是在每个处理器上一个线程处理一个客户请求，如果有额外的请求等待时，它们完成请求后并不阻塞。然而，为了让这种最佳处理正确工作，应用程序必须有方法在处理客户请求的线程阻塞在I/O上时激活另一个线程（如将读文件当作处理的一部分）。

9.3.7 IoCompletion对象

应用程序将IoCompletion（以completion port的形式导出给Win32）执行程序对象当作与多文件句柄相关的I/O完成的焦点。一旦文件与完成端口关联起来了，任何异步的I/O操作的完成将导致一个完成包加入到完成端口队列。通过完成包排队等待完成端口描写程可以等待任何未完程的I/O执行完。WIN32 API WaitForMultipleObjects函数提供了类似的功能。但完成端口的优势在于并行性或者说应用程序活动的服务于客户请求的线程数量是在系统的帮助下控制的。

创建完成端口时，应用程序指定并行值。该值指定某一时刻可运行的与端口相关的线程的最大数目。如前所述，理想的情况是每个时刻系统中每个处理器都有一个活动的线程。Windows 2000利用端口的并行值控制应用程序中活动线程的数量。如果与端口关联的活动线程值等于该并行值，等待该完成端口的线程将不允许运行。相反，它将等待某个活动线程处理完当前操作并检查是否有别的包等待该端口。如果有一个，该线程将简单地抓取该包然后处理。发生这种情况时，没有环境切换，CPU几乎被完全利用了。

9.3.8 使用完成端口

图9-17演示了一个高级别的完成端口操作。完成端口是调用Win32 API函数CreateIoComple-

tionPort 创建的。阻塞在完成端口上的线程与该端口关联起来，并按照先进先出的方式被唤醒，因此最近阻塞的线程就是获得下一个包的线程。阻塞很长时间的线程可以把堆栈换出到磁盘。因此如果端口关联的线程多于需要处理的线程，那么阻塞最长的线程在内存中的脚印将减少。

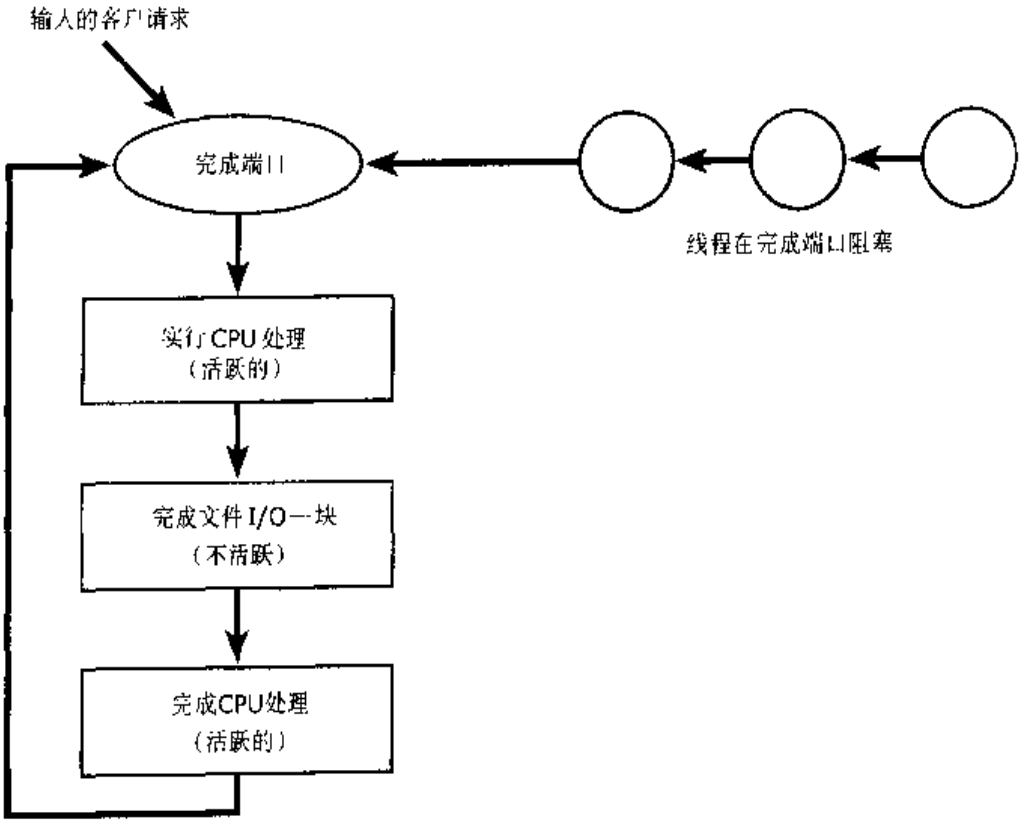


图 9-17 I/O 完成端口操作

服务器应用程序通常通过网络端点接受客户请求，这些网络端点是以文件句柄的形式表示的。这样的例子包括 Windows Sockets 2（Winsock2）套接字与命名管道。当服务器创建通信端点时，服务器将它们与一个完成端口关联起来。服务器的线程通过调用 GetQueuedCompletionStatus 在端口等待达到的请求。当线程从完成端口获得一个包时，它将离开并开始处理该请求，成为一个活动线程。在处理期间线程将阻塞多次，如从磁盘文件读写数据时或者与其他线程同步时。Windows 2000 检测这些活动并识别完成端口有不少于一个的活动线程。因此，当线程因为阻塞而变得不活跃时，如果队列中存在一个包的话，等待该完成端口的线程将被唤醒。

Microsoft 的方针是将并行值粗略地设置成系统中的处理器数目。请记住完成端口的活动线程数可能超过并行值的限制。请考虑该限制被指定为 1 的情况。一个客户请求到达，某个线程被调度来处理该请求，成为活动的。第 2 个请求到达，但是等待该端口的线程不允许处理，因为已经达到并行值。然后第一个线程等待 I/O 而阻塞，停止活动。然后第二个线程释放，当它仍活动时，第一个线程完成 I/O，使第一个线程活动起来。此刻——直到某个线程阻塞——并行值是 2，大于限制值 1。大多数时刻活动数目都保持在或刚好高于并行值。

完成端口 API 还使服务器应用程序能够利用 PostQueuedCompletionStatus 函数将私自定义的完成包在完成端口排队。服务器通常利用该函数通知线程外部事件，如优雅的关闭（shut down gracefully）需求。

9.4 驱动程序的加载、初始化与安装

在Windows 2000中，驱动程序的加载和初始化由两部分组成：显式加载和基于枚举的加载。显式加载是由注册表分支HKLM\SYSTEM\CurrentControlSet\Services指导的。该分支在第5章的“服务应用”一节中作了介绍。当PnP管理器动态加载总线驱动程序在总线枚举期间报告的设备的驱动程序时，将导致基于枚举的驱动程序加载。

9.4.1 启始值

在第5章，我们解释了每个驱动程序和Win32服务在当前控制集中的Service分支下都有一个注册表键值。该键值包含指定映像类型（如Win32服务、驱动程序和文件系统）的键值、驱动程序或映像文件的路径以及控制驱动程序或服务映像文件加载顺序的值。在明确的设备驱动程序的加载与Win32服务的加载之间有两个主要差别：

■ 只有设备驱动程序可以指定Start值为引导－启动（0）或者系统－启动（1）。

■ 设备驱动程序可以利用Group和Tag值控制在启动阶段的加载顺序。但是与服务不同的是，它们不能指定DependOnGroup值与DependOnService值。

第4章介绍了启动过程的各个阶段，并说明了驱动程序的Start值为0的意思是由操作系统的加载器加载该驱动程序。Start值为1表示I/O管理器在执行程序子系统完成初始化之后加载该驱动程序。I/O管理器按照启动阶段加载驱动程序的顺序调用驱动程序的初始化例程。与Win32服务一样，驱动程序利用注册表键值中的Group值指定它们所属的组。注册表键值HKLM\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder\List的值决定了在启动阶段组的加载顺序。

通过包含一个控制在组中的加载顺序的Tag值，驱动程序可以进一步明确它的加载顺序。I/O管理器按照驱动程序注册表键值中定义的Tag值排序每个组中的驱动程序。没有标志的驱动程序排在所在组的最后。你可能认为I/O管理器是先初始化具有小标志的驱动程序，然后初始化具有大标志的驱动程序。但是这并不是绝对的。注册表键值HKLM\SYSTEM\CurrentControlSet\Control\GroupOrderList决定标志在组中的优先权。利用该键值，Microsoft和设备驱动程序开发者可以使用权限重新定义系统的整数值。

下面是驱动程序设置Start值的指导原则：

■ 遗留驱动程序设置Start值以反映它们要在哪个启动阶段加载。

■ 必须通过启动加载器在系统启动期间加载的驱动程序，包含遗留驱动程序和Windows 2000驱动程序，指定Start值为boot－set（0）。例如系统总线驱动程序和引导文件系统驱动程序。

■ 不是启动系统所必须的驱动程序以及能检测到系统总线驱动程序不能枚举的设备的驱动程序，将Start值指定为系统－启动（1）。如串行端口驱动程序，能通知PnP管理器标准串行PC端口的到达，而这些端口是Setup检测到的并记录在注册表中。

■ 不需要支持即插即用的Windows 2000驱动程序或者是不需要出现在系统启动期间的遗留驱动程序，将Start值指定为自动－启动（2）。例如Multiple Universal Naming Convention（UNC）

Provider（MUP）驱动程序，该驱动程序支持基于 UNC 的远程资源路径命名（如，\\ REMOTE-COMPUTERNAME \ SHARE）。

■ 启动系统所不需要的即插即用驱动程序将 Start 值指定为 demand - start（3），如网络适配器驱动程序。

即插即用驱动程序的 Start 值以及可枚举设备的 Start 值的唯一目的是为了确保操作系统加载器加载该驱动程序—如果该驱动程序是成功启动系统所必须的话。除此之外，PnP 管理器的设备枚举进程，将在下面描述，决定即插即用驱动程序的加载顺序。

9.4.2 设备枚举

PnP 管理器利用一个称为“Root”的虚拟总线驱动程序开始设备枚举。该驱动程序代表整个计算机系统，同时还作为遗留驱动程序和 HAL 的总线驱动程序。HAL 象总线驱动程序那样工作。总线驱动程序直接枚举隶属于主板以及系统组件（如电交换区）的设备。与真正枚举相反的是，HAL 凭借 Setup 进程记录在注册表中的硬件描述检测主要总线（一般为 PCI 总线）和设备（如电池和风扇等）。

主要总线驱动程序枚举该总线上的设备，以及可能找到的其他总线上的设备。PnP 管理器初始化这些设备的驱动程序。接着这些驱动程序能够检测其他设备以及其他附属的总线。这种递归的枚举、加载驱动程序（如果驱动程序程序没有加载的话）、进一步枚举的过程一直进行，直到系统中的所有设备都被检测到并配置好。

当总线驱动程序向 PnP 管理器报告检测到的设备时，PnP 管理器创建一个表示设备之间关系的内部树结构，称为“设备树”（device tree）。树中的结点称为“设备结点”（devnode）。设备结点包含代表该设备的设备对象信息，以及通过 PnP 管理器存储在结点中的与即插即用有关的信息。图 9-18 显示了一个简化的设备树例子。该系统是 ACPI 兼容的，因此 ACPI 兼容的 HAL 作为主要总线枚举器。PCI 总线作为系统的主要总线，USB、ISA 和 SCSI 总线连接在其上。

Device Manager 工具显示了一个出现在默认配置的系统中的简单设备列表。从 Start 菜单的 Programs/Adminstrative Tools 文件夹中的 Computer Management 中可以获取该工具（也可以从 Control Panel 中的 System 工具的 Hardware 标签中得到）。你可以在 Device Manager's View 菜单中选择 Devices By Connection 选项，查看与该设备树相关的设备。图 9-19 显示的是 Device Manager's Devices By Connection 视图的例子。

考虑到设备枚举，驱动程序的加载和初始化顺序为：

1）I/O 管理器唤醒每个引导 - 启动驱动程序的登录例程。如果引导驱动程序具有子设备，那么 I/O 管理器将枚举那些设备，并报告给 PnP 管理器。如果它们的驱动程序是引导 - 启动驱动程序，将配置并启动这些子设备。如果某个设备的驱动程序不是引导 - 启动驱动程序，那么 PnP 管理器将为该设备创建一个设备结点，但是不会启动该设备或者是加载其驱动程序。

2）在引导 - 启动设备初始化之后，PnP 管理器将遍历设备树，为在步骤 1 中没有加载的设备结点加载驱动程序并启动该设备。在启动设备的时候，PnP 管理器枚举该设备的子设备，如果有子设备的话，启动那些设备的驱动程序并执行所必须的其子设备的枚举。在该步骤中 PnP

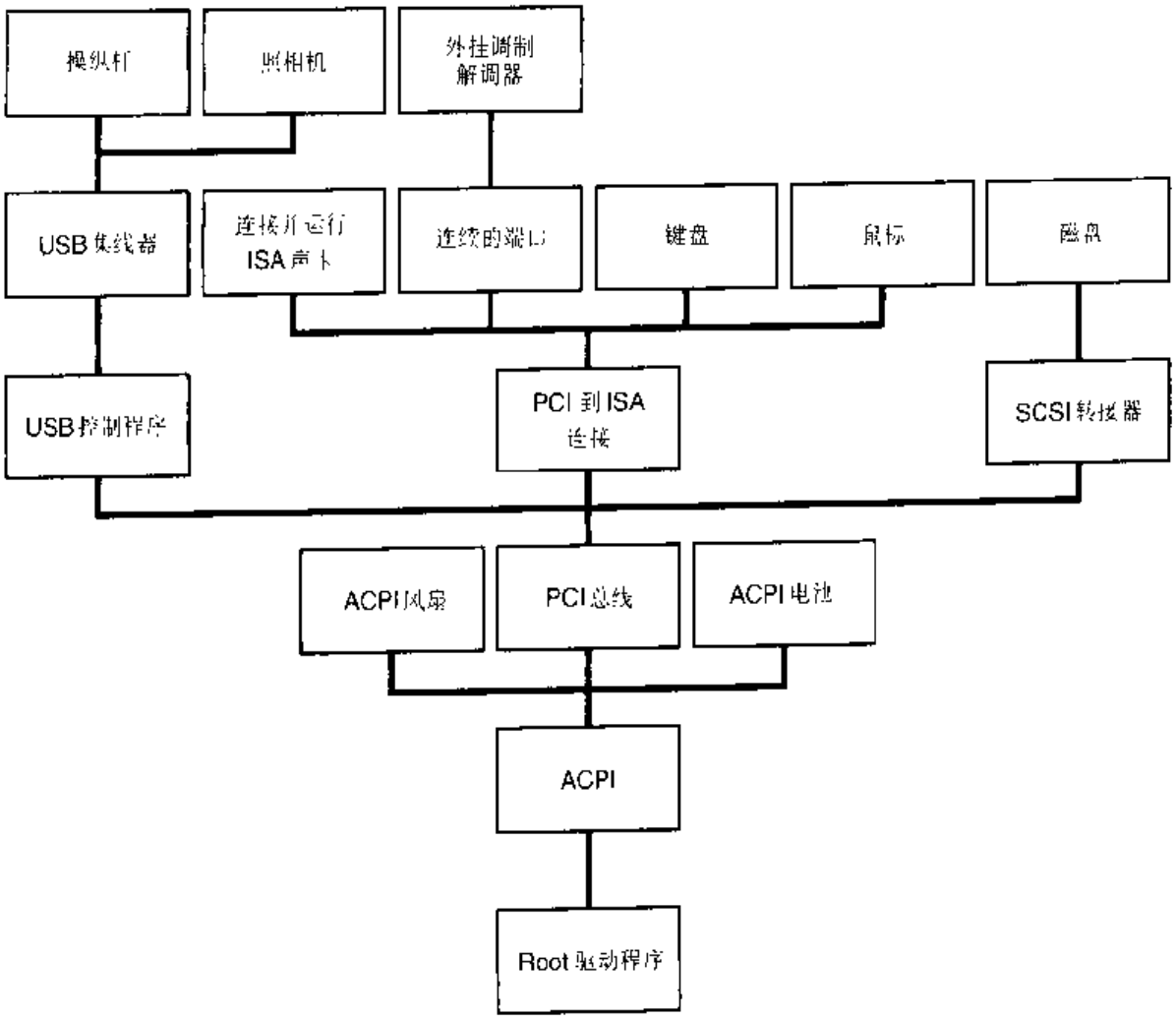


图 9-18 设备树例子

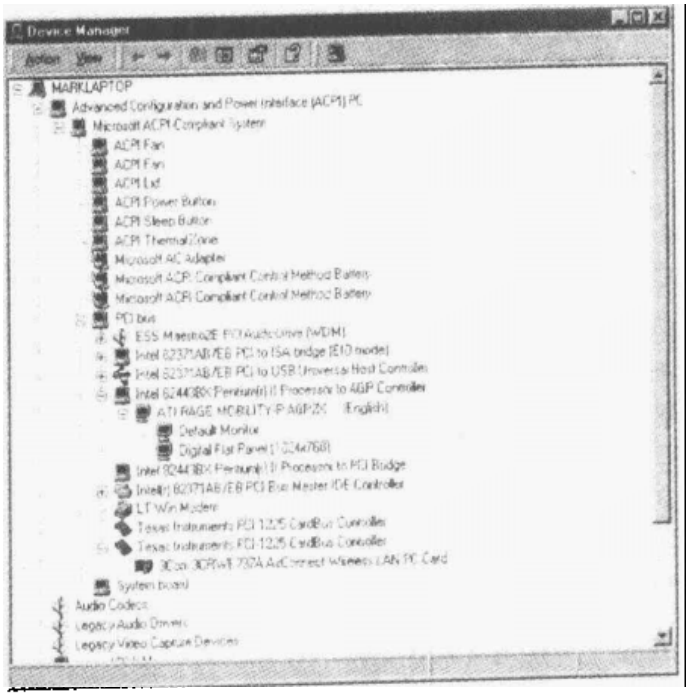


图 9-19 显示设备树的设备管理器

管理器将为检测到的设备加载驱动程序而不管驱动程序的启动值（一个例外是启动值被设置成禁止）。在该步骤的最后，所有即插即用设备都具有它们的驱动程序并被启动。但是那些不可枚举的设备及其子设备例外。

3）PnP 管理器加载所有 Start 值为系统 - 启动而还没有加载的驱动程序。那些驱动程序检测并报告不可枚举的设备。PnP 管理器为那些设备加载驱动程序，直到所有枚举的设备都被配置并启动了。

4）Service Control Manager 加载标明为自动 - 启动的驱动程序。

设备树的作用是指导 PnP 管理器和电源管理器向设备发布即插即用与电源 IRP。一般地，IRP 从设备结点的顶部流向底部。在某些情况下，设备结点中的驱动程序将创建新的 IRP 传送给其他结点，并且一直向上移动到根。即插即用流程和电源 IRP 将在本章后部进一步介绍。

实验：转储设备树

比使用 Devcie Manager 查看设备树更为详细的方法是使用内核调试器命令！devnode。指定命令选项 0 1 将转储如下所示的内部设备树结构，缩进条目显示了它们的层次关系：

```
kd> !devnode 0 1
Dumping IopRootDeviceNode (= 0x818a78e8)
DevNode 0x818a78e8 for PDO 0x818a79e0
  Parent 0000000000   Sibling 0000000000   Child 0x818a74c8
  InstancePath is "HTREE\ROOT\0"
  Flags (0x00040459)  DNF_MADEUP, DNF_PROCESSED,
                      DNF_ENUMERATED, DNF_ADDED,
                      DNF_NO_RESOURCE_REQUIRED, DNF_STARTED
  DisableableDepends = 10 (from children)
  DevNode 0x818a74c8 for PDO 0x818a75d0
    Parent 0x818a78e8   Sibling 0x818a7248   Child 0x81883228
    InstancePath is "Root\ACPI_HAL\0000"
    Flags (0x000405dd)  DNF_MADEUP, DNF_HAL_NODE,
                        DNF_PROCESSED, DNF_ENUMERATED,
                        DNF_ADDED, DNF_HAS_BOOT_CONFIG,
                        DNF_BOOT_CONFIG_RESERVED, DNF_NO_RESOURCE_REQUIRED,
                        DNF_STARTED
    DisableableDepends = 1 (from children)
    DevNode 0x81883228 for PDO 0x818838d0
      Parent 0x818a74c8   Sibling 0000000000   Child 0x81819408
      InstancePath is "ACPI_HAL\PNP0C08\0"
      ServiceName is "ACPI"
      Flags (0x000421d8)  DNF_PROCESSED, DNF_ENUMERATED,
                          DNF_ADDED, DNF_HAS_BOOT_CONFIG,
                          DNF_BOOT_CONFIG_RESERVED, DNF_RESOURCE_ASSIGNED,
                          DNF_STARTED
      CapabilityFlags (0x000000c0)  UniqueID, SilentInstall
      DisableableDepends = 10 (from children)
      DevNode 0x81819408 for PDO 0x81891530
        Parent 0x81883228   Sibling 0x818916c8   Child 0000000000
```

```
InstancePath is "ACPI\PNP0C08\1"
Flags (0x00040458)  DNF_PROCESSED, DNF_ENUMERATED,
                    DNF_ADDED, DNF_NO_RESOURCE_REQUIRED,
                    DNF_STARTED
UserFlags (0x00000008)  DNUF_NOT_DISABLEABLE
CapabilityFlags (0x00000100)  UniqueID, SilentInstall,
                              RawDeviceOK
DisableableDepends = 1 (including self)
```

为每个设备结点显示的信息包括 InstancePath，该设备的枚举注册表键值名称（存储在 HKLM \ SYSTEM \ CurrentControlSet \ Enum 中）以及 ServiceName，它对应于设备的驱动程序注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Service。若要查看分配给每个设备结点的资源如中断、端口和内存等，请在命令！devnode 中指定 0 3 命令选项。

自从系统安装以来所检测到的所有设备都记录在注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Enum 中。子键值的形式为〈Enumerator〉\〈Device ID〉\〈Instance ID〉，其中 Enumerator 是总线驱动程序，Device ID 是唯一的设备类型标识符，Instance ID 唯一指定同种设备的不同实例。

9.4.3 设备结点

图 9-19 显示一个设备结点至少由两个设备对象组成，有时更多：

■ 物理设备对象（PDO）。在枚举期间当总线驱动程序报告某个设备出现在总线上时，PnP 管理器指示总线驱动程序创建物理设备对象。PDO 表示设备的物理接口。

■ 一个或多个可选的位于 PDO 和 FDO 之间（下面介绍）、由总线过滤器驱动程序创建的过滤器设备对象（FiDO）。

■ 一个或多个可选的位于 PDO 和 FDO 之间（以及位于总线过滤器驱动程序创建的 FiDo 之上的）、由低级过滤器驱动程序创建的 FiDO。

■ 一个由功能驱动程序创建的、PnP 管理器加载以管理检测到的设备的功能设备对象（FDO）。FDO 表示设备的逻辑接口。功能驱动程序还可以作为总线驱动程序使用，如果设备附属于该 FDO 所表示的设备的话。功能驱动程序经常创建 FDO 对应的 PDO 接口（前面描述了），因此应用程序和其他驱动程序可以打开设备并与它进行交互操作。有时功能驱动程序被分割成单独的类/端口驱动程序和小端口驱动程序，它们一起管理 FDO 的 I/O。

■ 一个或多个位于 FDO 之上、由高级过滤器驱动程序创建的 FiDO。

设备结点是自低向上建立的，依赖于 I/O 管理器的层次化能力。因此 IRP 是从设备结点的顶部向底部流动的。然而，设备结点任何级别都可以选择完成一个 IRP。例如，功能驱动程序可以处理一个读请求，而不需将该 IRP 传递给总线驱动程序。只有在功能驱动程序需要某个总线驱动程序帮助执行特定总线的处理时，IRP 才流经到达底部的路径，然后进入包含该总线驱动程序的设备结点。

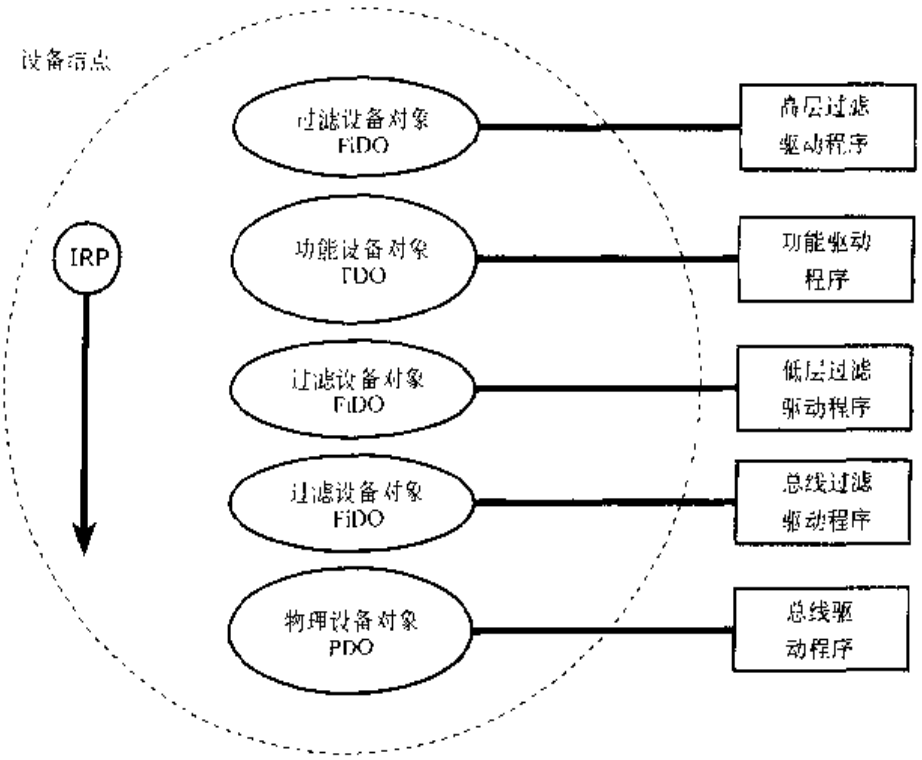


图 9-20　设备结点内部

9.4.4　设备结点驱动程序的加载

迄今为止，我们一直避免回答两个重要的问题：PnP 管理器是如何决定为特定设备加载什么样功能的驱动程序的呢？以及过滤器驱动程序是如何注册它们的出现以致于它们能够在创建设备结点的适当时刻被加载。

这两个问题的答案在于注册表。当总线驱动程序进行设备枚举的时候，它将所检测到的设备的设备标识符报告给 PnP 管理器。标识符是总线相关的。对于 USB 总线，标识符由 vendor ID（VID）和 product ID（PID）组成的，VID 表示制造该设备的硬件制造商，PID 是由制造商赋予该设备的（有关详细设备 ID 格式信息请参阅 DDK 文档）。这些 ID 一起形成了即插即用所称为的“设备 ID”。PnP 管理器还查询总线驱动程序的实例 ID，以帮助它区别同种硬件设备的不同实例。实例 ID 既可以描述一个相对于总线的位置（如 USB 端口），也可以是一个全局唯一的描述符（如串行号）。设备 ID 与实例 ID 联合形成设备实例 ID（DIID），PnP 管理器利用它查找该设备在注册表中的枚举分支（（HKLM \ SYSTEM | CurrentControlSet \ Enum）中的注册表键值。图 9-21 显示了一个键盘的枚举子键的例子。设备键值中包含描述性的数据以及命名的 Service 和 ClassGUID 值（从驱动程序的 INF 文件中可以得到），这些帮助 PnP 管理器查找设备的驱动程序。

利用 ClassGUID 值，PnP 管理器可以在 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Class 中查找设备的类键。键盘类键显示在图 9-21 中。枚举键和类键为 PnP 管理器提供加载该设备的设备结点的设备驱动程序所必须的信息。驱动程序按一下顺序加载：

1）设备枚举键中 LowerFilters 值指定的所有低级过滤器驱动程序。

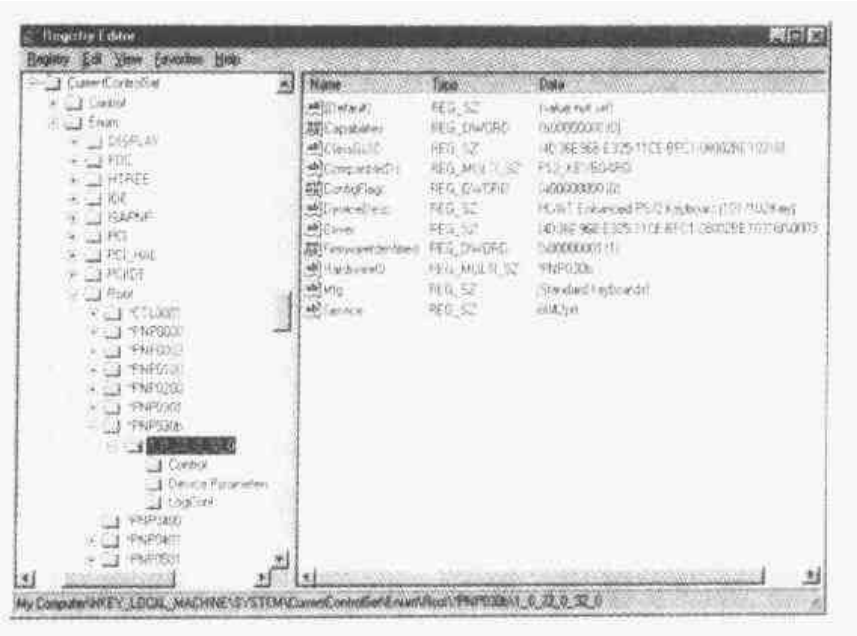


图 9-21 键盘枚举键

2）设备类键中的 LowerFilters 值指定的所有低级过滤器驱动程序。

3）设备枚举键中的 Service 值指定的功能驱动程序。该值解释为 HKLM \ SYSTEM \ CurrentControlSet \ Services 下的驱动程序键。

4）设备枚举键中的 UpperFilters 值指定的所有高级过滤器驱动程序；

5）设备类键中的 UpperFilters 值指定的所有高级过滤器驱动程序。

在所有情况下，驱动程序都是通过它们在 HKLM \ SYSTEM \ CurrentControlSet \ Service 中的键名引用的。

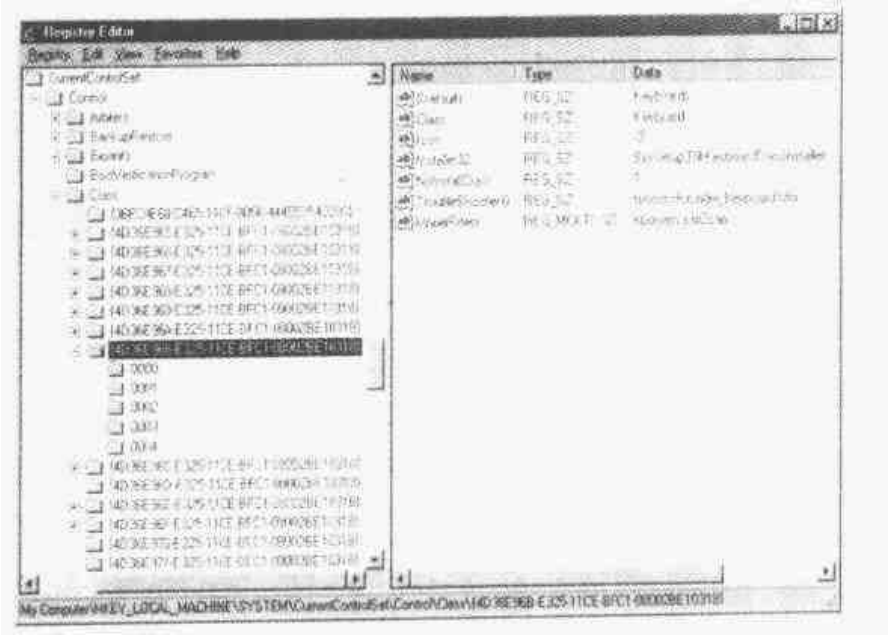


图 9-22 键盘类键

注意：DDK 称设备的枚举键为它的“硬件键”，类键为“软件键”。

图 9-21 和 9-22 中显示的键盘设备没有低级过滤器驱动程序。功能驱动程序为 i8042prt 驱动

程序。在键盘类键中有两个高级过滤器驱动程序：kbdclass 和 ctrl2cap。

9.4.5 驱动程序的安装

如果 PnP 管理器遇到没有安装驱动程序的设备，它将依赖用户模式的 PnP 管理器指导安装过程。如果设备是在系统启动期间检测到的，将为该设备指定一个设备结点，但是加载过程要推迟到用户模式的 PnP 管理器启动之后（用户模式的 PnP 管理器是在 \ winnt \ System32 \ Umpnpmgr.dll 中实现的，运行时作为 Service.exe 进程的一个服务）。

驱动程序的安装过程中涉及到的组件显示在图 9-23。图中的阴影对象表示通常有系统提供的组件，而没有加阴影的对象则包含在驱动程序安装文件中。首先，总线驱动程序通知 PnP 管理器某个使用 DIID 枚举的设备。1）PnP 管理器检查注册表中相应的功能驱动程序。2）如果没有找到，它将通过 DIID 通知用户模式 PnP 管理器新的设备。用户模式的 PnP 管理器首先试探着执行不需要用户干预的自动安装。如果安装进程涉及到需要用户交互的对话框，并且当前登录的用户具有管理权限，3）那么用户模式的 PnP 管理器将启动应用程序 Rundll32.exe（提供 Control Panel 实用程序的相同应用程序），执行 Hardware Installation Wizard（ \ Winnt \ System32 \ Newdev.dll）。如果当前登录的用户不具有管理权限（或者没有用户登录），并且设备的安装需要用户交互，那么 PnP 管理器将推迟安装指导授权用户登录。Hardware Installation Wizard 使用 Setup 和 CfgMgr（配置管理器）API 函数查找与所检测到的设备兼容的驱动程序的 INF 文件。该过程可能涉及需要用户插入包含制造商的 INF 文件的安装媒介，或者是在 \ Winnt \ Driver Cache \ i386 \ Driver.cab 文件中查找合适的 INF 文件，该文件包含与 Windows 2000 一起发送的驱动程序。

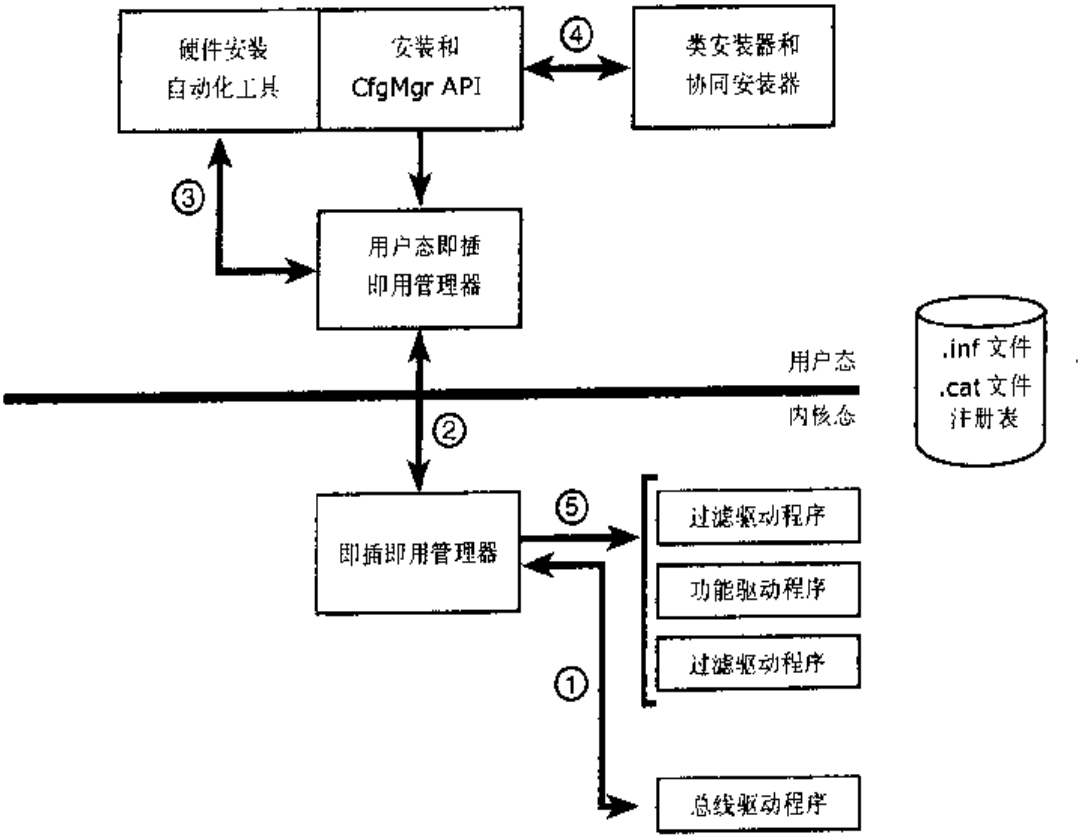


图 9-23　驱动程序的安装组件

为了查找新设备的驱动程序，安装进程从总线驱动程序那里得到一个 hardware ID 和 compatible ID 列表。这些 ID 描述了确认驱动程序安装文件（.inf）中的硬件设备的各种方法。该列表是排序的，以便首先列举最具体的硬件描述。如果在多个 INF 文件中找到匹配结果，精确匹配优先于不精确的匹配，带数字标记的 INF 优于无标记的，新标记的 INF 优于旧标记的。如果在兼容的 ID 中找到匹配，Hardware Installation Wizard 可以选择提示要求新的媒介，以防出现更新的硬件驱动程序。

INF 文件查找功能驱动程序文件，并包含填充驱动程序枚举键和类键的命令。同时 INF 文件还能够指示 Hardware Installation Wizard (4) 启动类或设备联合安装程序来执行类或设备相关的安装步骤，如显示配置对话框让用户指定设备的设置。

在真正安装设备驱动程序前，用户模式的 PnP 管理器将检查系统的驱动程序登录策略。该策略存储在注册表键 HKLM \ SYSTWEM \ Microsoft \ Driver Signing \ Policy 中，如果管理员指定了系统范围的策略的话；如果只有单个用户的策略，则存储在注册表键 HKCU \ Software \ Microsoft \ Driver Signing \ Policy 中。使用 Driver Signing Option 对话框可以配置该策略。该对话框显示在图 9-24，从 System Control Panel 工具中的 Hardware 标签可以访问该它。如果设置指定系统阻塞或警告未签名的驱动程序的安装，那么用户模式的 PnP 管理器将检查驱动程序的 INF 文件中包含该驱动程序数字签名的目录（以扩展名 .cat 结尾的文件）位置的项目。

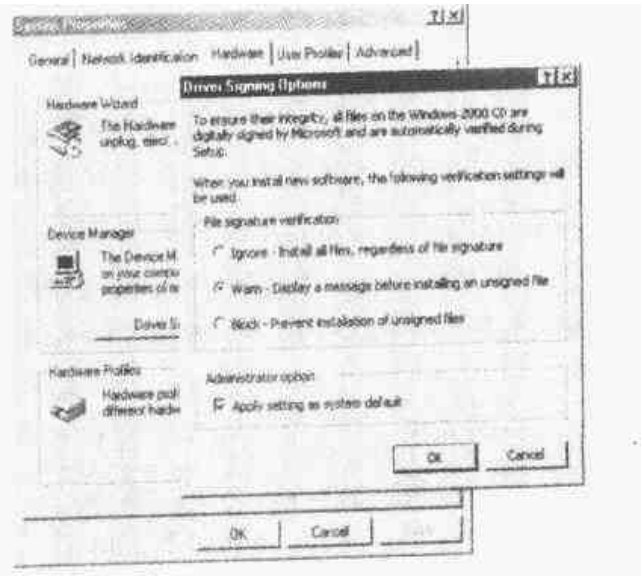


图 9-24 驱动程序签名策略选项

Microsoft 的 WHQL 测试包含在 Windows 2000 中的以及由硬件产商提供的驱动程序。当驱动程序通过 WHQL 测试时，它就被 Microsoft 签名。这意味着 WHQL 获取了一个散列值或者是唯一代表该驱动程序映射文件的值。然后利用 Microsoft 的私有驱动程序签名键对它进行加密签名。签名的散列存储在一个目录文件中，同时包含在 Windows 2000 的安装媒介中或者为了包含该驱动程序返回到提交该驱动程序的产商。

安装驱动程序时，用户模式的 PnP 管理器从它的目录文件中提取该驱动程序的签名，并利用 Microsoft 的驱动程序签名私用/公用密匙对的公用部分解密该签名，然后将结构的散列值与将要安装的驱动程序文件的散列值进行比较。如果散列值匹配，驱动程序被确认为通过 WHQL 测试。如果驱动程序不能通过签名验证，用户模式的 PnP 管理器将按照系统驱动程序签名策略的设置行事，或者安装或者警告用户驱动程序未签名，或者是悄悄地安装该驱动程序。

注意 利用安装程序手工配置注册表并将驱动程序文件复制到系统，以及由应用程序动态加载到系统的驱动程序，不会检测签名。只有那些利用 INF 文件安装的驱动程序需要按照系统的驱动程序签名策略进行验证。

在驱动程序程序安装之后，内核模式的 PnP 管理器（图 9-23 中的步骤 5）启动该驱动程

序，并调用设备添加例程，通知该驱动程序其所要加载的设备的出现。然后按照前面所描述的构造设备结点。

9.5 I/O处理

既然我们已经涉及了驱动程序的结构和类型以及支持它们的数据结构，那么让我们来考察一下I/O请求是如何在系统中流动的。I/O请求要经历几个可预见的处理阶段。阶段的变化依赖于该请求是指定一个由单层驱动程序操纵的设备，还是一个通过一个多层驱动程序到达的设备。更进一步的处理变化依赖于调用程序是指定异步还是同步I/O。因此我们将首先讨论这两种I/O类型，然后继续其他内容。

9.5.1 I/O类型

应用程序对它们发出的I/O请求有多种选择。例如，它们可以指定异步或同步I/O；可以指定I/O将设备数据映射到应用程序的地址空间以便应用程序通过虚拟地址空间访问而不是I/O API函数；可以指定I/O在单个请求中在设备与非连续的应用程序缓冲区之间传输数据。另外，I/O管理器给了驱动程序实现一个快捷I/O接口的选择，该接口可以经常为I/O处理减轻IRP分配。在本节，我们将解释各种不同的I/O。

1. 同步I/O与异步I/O

应用程序发出的大多数I/O操作都是同步的；也就是说，应用程序一直等待直到设备执行数据传输并在I/O完成时返回一个状态代码。然后，程序继续执行并立即访问所传输的数据。它们的最简单的形式，Win32 Readfile和Writefile函数，是同步执行的。在把控制返回给调用程序之前，它们完成一个I/O操作。

异步I/O允许应用程序发送I/O请求，然后当设备传输数据的同时，应用程序继续执行。这类I/O能够提高应用程序的吞吐率，因为它允许在I/O操作进行期间，应用程序继续其他的工作。要使用异步I/O，必须在Win32 CreateFile函数中指定FILE_FLAG_OVERLAPPED标志。当然，在发出异步I/O操作之后，线程必须小心地不访问任何来自I/O操作的数据，直到设备驱动程序完成数据传输。线程必须通过监控一个同步对象（无论是事件对象、I/O完成端口或文件对象本身）的句柄，使它的执行与I/O请求的完成同步。当I/O完成时，这些同步对象将会变成有信号态。

不管与I/O请求的类型如何，由IRP代表的内部I/O操作都被异步执行；也就是说，一旦一个I/O请求已经被启动，设备驱动程序就返回I/O系统。I/O系统是否返回调用程序取决于是否文件是为同步或异步I/O打开的。图9-25说明了当读取操作启动后的控制流程。注意，等待是由NtReadFile函数在内核模式下完成的，是否等待取决于文件对象中的重叠标志。

你可以使用Win32的HasOverlappedIoCompleted函数去测试挂起的异步I/O的状态。如果你正在使用I/O完成端口，则可用GetQueuedCompletionStatus函数。

2. 快速I/O

快速I/O是一个特殊的机制，该机制允许I/O系统不产生IRP而直接转到文件系统驱动程序或高速缓存管理器去完成I/O请求（在第11章和12章详细描述了快速I/O）。驱动程序注册

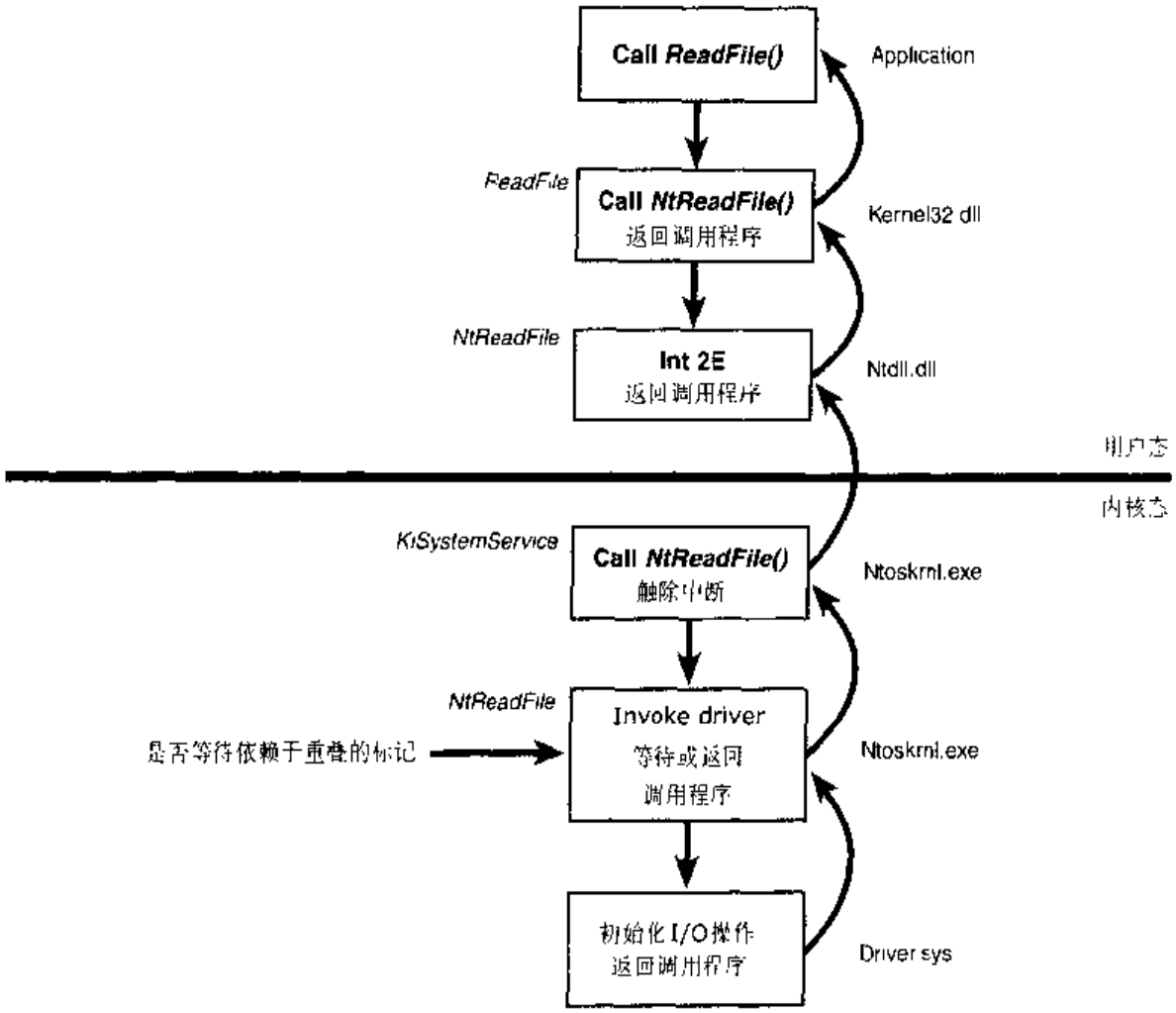


图 9-25 I/O 操作的控制流程

它的快速 I/O 的入口点，把它们输入到 PFAST_IO_DISPATCH 指针所指的驱动程序中的一个结构中。

实验：查看驱动程序注册的快速 I/O 例程

内核调试器命令 !drvobj 可以列举驱动程序注册在其驱动程序对象中的快速 I/O 例程。然而，一般只有文件系统驱动程序对于快速 I/O 有些作用。下面的输出显示了 NTFS 文件系统驱动程序对象的快速 I/O 表：

```
kd> !drvobj \filesystem\rtfs 2
Driver object (ff432670) is for:
 \FileSystem\Ntfs
Dispatch routines:
[00] IRP_MJ_CREATE
:

Fast I/O routines:
FastIoCheckIfPossible                   be3263ef
Ntfs!NtfsPostUsnChange+0xd8c
FastIoRead                              be31869e
Ntfs!NtfsCreateInternalStreamCommon+0x1b43
FastIoWrite                             be318df9
```

```
Ntfs!NtfsCreateInternalStreamCommon+0x229e
FastIoQueryBasicInfo                    be3020fa
Ntfs!NtfsRaiseStatus+0x105a8
FastIoQueryStandardInfo                 be317d1e
Ntfs!NtfsCreateInternalStreamCommon+0x11c3
FastIoLock                              be32622d
Ntfs!NtfsPostUsnChange+0xbca
FastIoUnlockSingle                      be326139
Ntfs!NtfsPostUsnChange+0xad6
:
```

结果显示 NTFS 已经把它的 NtfsPostUsnChange 例程注册为快速 I/O 表的 FastIoCheckIfPossible 项。正如快速 I/O 项的名字所意味的，I/O 管理器有时候调用该函数来发送快速 I/O 请求，给驱动程序一个机会指出什么时候在文件上的快速 I/O 操作不可能。

3. 映射文件 I/O 和文件高速缓存

映射文件 I/O 是 I/O 系统的一个重要特性，是 I/O 系统和内存管理器共同产生的（关于如何实现映射文件的详细信息，请参阅第 7 章）。“映射文件 I/O”是指把磁盘中的文件视为进程的虚拟内存的一部分的能力。程序可以把文件作为一个大的数组来访问，而无需做缓冲数据或执行磁盘 I/O 的工作。程序访问内存，同时内存管理器利用它的调页机制从磁盘文件中加载正确的页面。如果应用程序向它的虚拟地址空间写入数据，内存管理器就把更改作为正常页面调度的一部分写回到文件中。

在用户模式可以通过 Win32 CreateFileMapping 和 MapViewOfFile 函数使用映射文件 I/O。在操作系统中，映射文件 I/O 被用于重要的操作中，如文件高速缓存和映像活动（加载并运行可执行程序）。其他主要的使用映射文件 I/O 的程序有高速缓存管理器。文件系统使用高速缓存管理器将文件数据映射到虚拟内存中，从而为 I/O 受限的程序提供了更快的响应时间。当调用程序使用该文件时，内存管理器将把被访问的页面调入内存。尽管多数高速缓存系统在内存中分配固定数量的字节给高速缓存文件，但 Windows 2000 高速缓存的增大或缩小取决于可以获得的内存有多少。这种大小的变化是可能的，因为高速缓存管理器依赖于内存管理器来自动地扩充（或缩小）高速缓存的数量。通过使用在第 7 章中解释的正常工作集机制来实现这一功能。通过利用内存管理器的页面调度系统的优势，高速缓存管理器避免了重复内存管理器已经执行了的工作（高速缓存管理器的工作将在第 11 章中详细介绍）。

4. 分散/集中 I/O

Windows 2000 还支持一种特殊种类的高性能 I/O，称为“分散/集中”（scatter/gather）。可以通过 Win32 ReadFileScatter 和 WriteFileScatter 函数使用它。这些函数允许应用程序从虚拟内存中的一个以上的缓冲区中发出单一的读取或写入到磁盘文件的一块连续区域。为了使用分散/集中 I/O，文件必须以非高速缓存 I/O 方式打开，被使用的用户缓冲区必须是页对齐的，并且 I/O 必须是异步执行的（覆盖）。此外，如果 I/O 是直接的块存储设备，那么 I/O 必须是在设备扇区的边界对齐，且长度是扇区大小的倍数。

9.5.2 对单层驱动程序的I/O请求

这一节将跟踪对单层内核模式设备驱动程序的同步I/O请求。处理对单层驱动程序的同步I/O包括以下7步：

1）I/O请求经过子系统DLL。

2）子系统DLL调用I/O管理器的NtWriteFile服务。

3）I/O管理器分配一个描述该请求的IRP并通过调用其拥有的IoCallDriver函数将该IRP发送给驱动程序（这里指设备驱动程序）。

4）驱动程序将IRP中的数据传输到设备并启动I/O操作。

5）驱动程序通过中断CPU，发信号给I/O完成例程。

6）在设备完成了操作并且中断CPU时，设备驱动程序服务于中断。

7）驱动程序调用I/O管理器的IoCompleteRequest函数通知其自身已经完成IRP请求的处理，然后I/O管理器完成该I/O请求。

这7个步骤如图9-26所示。

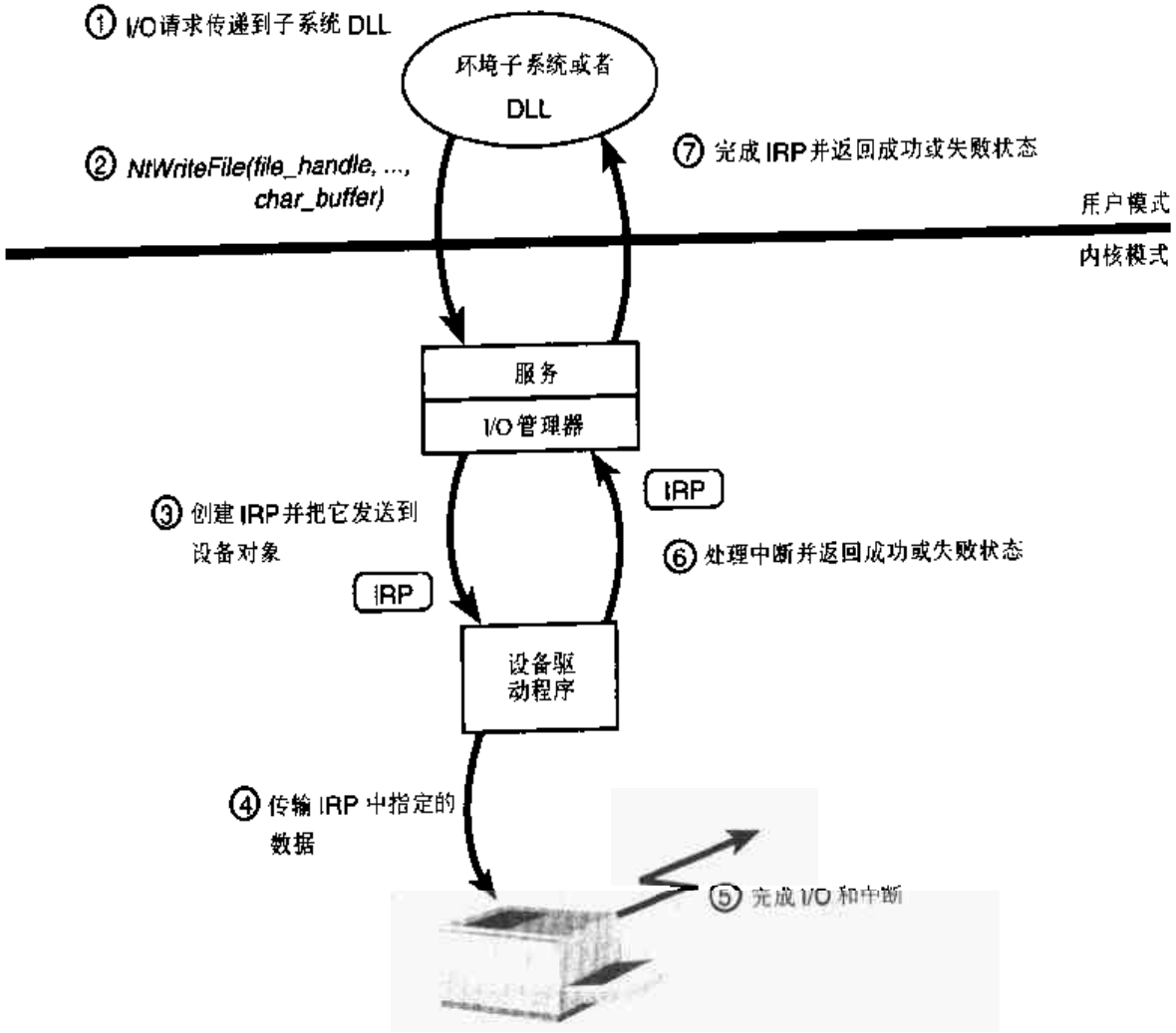


图9-26 排队并完成一个同步请求

既然，我们已经看过了怎样初始化 I/O，现在让我们更深入地看一下中断处理和 I/O 完成。

1. 中断服务

在 I/O 设备完成数据传输之后，它将中断并请求服务，并且 Windows 2000 内核、I/O 管理器和设备驱动程序将被调用而开始运作。图 9-27 说明了过程的第一阶段。（在第 3 章中描述了中断调度机制，包括 DPC 在内。在此，我们扼要地重述一下，因为 DPC 是 I/O 处理的关键）。

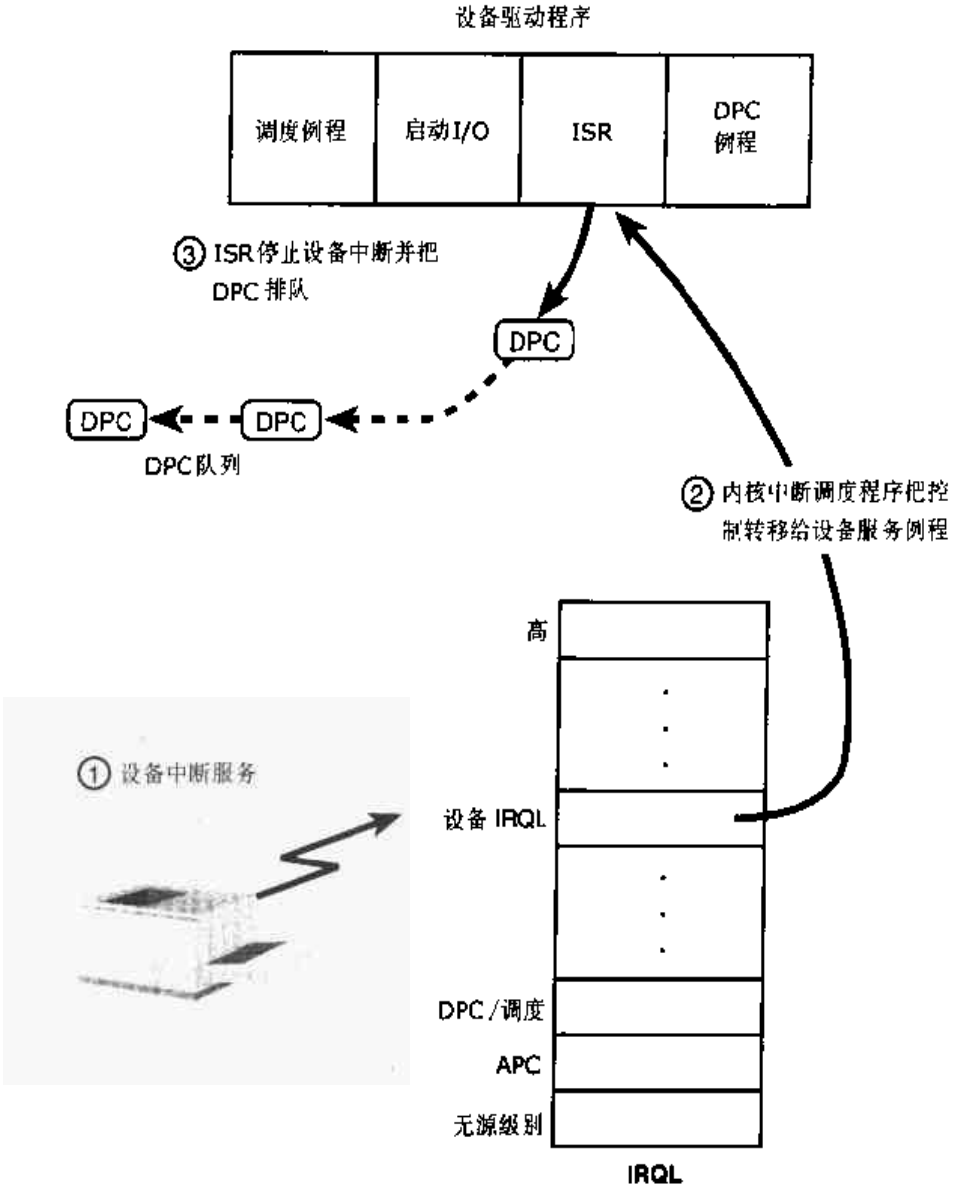


图 9-27 设备中断服务（第一阶段）

当设备中断发生时，处理器将控制转交给内核陷阱处理程序，内核陷阱处理程序将索引到其中断调度表来查找该设备的 ISR。Windows 2000 中的 ISR 一般用两个步骤来处理设备中断。当 ISR 被首次调用时，它通常只在设备 IRQL 上停留足够长的时间以获得设备状态，然后停止该设备的中断。接着它将一个 DPC 排入队列、清除中断并退出。过一段时间，在 DPC 例程被调用时，设备完成对该中断的处理。完成之后，设备将调用 I/O 管理器来完成 I/O 并处理 IRP。

它也可能启动下一个正在设备队列中等待的I/O请求。

使用DPC来执行大多数设备的服务的优点是任何优先级位于设备IRQL和Dispatch/DPC IRQL之间的被阻塞的中断都被允许发生在低优先级的DPC处理发生之前。这样中间优先级的中断就可以比其他中断更快地得到服务。I/O处理（DPC处理）的第二阶段如图9-28所示。

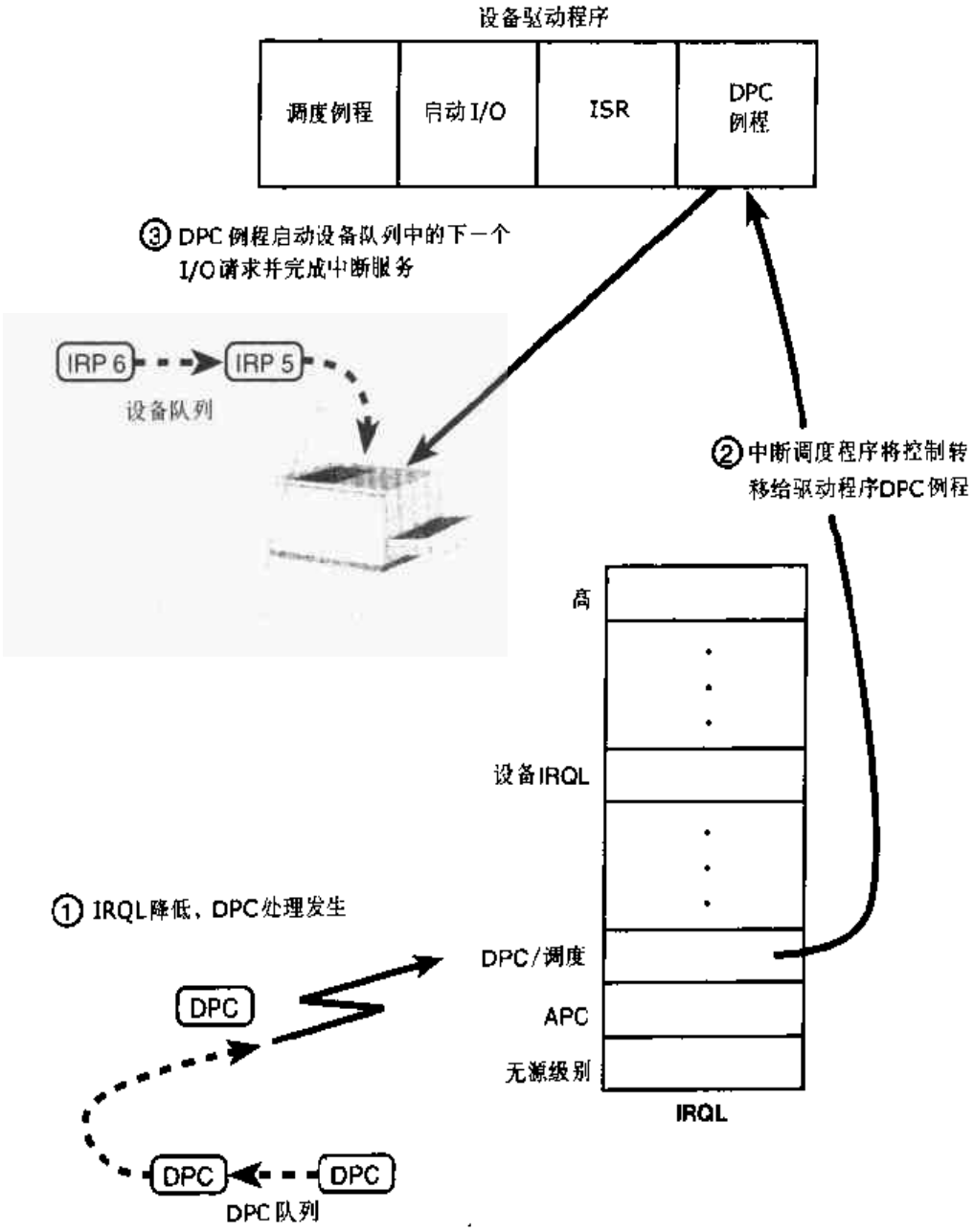


图9-28 服务一个设备中断（阶段2）

2. 完成I/O请求

设备驱动程序的DPC例程执行完以后，在考虑结束I/O请求之前可能还有一些工作要做。I/O处理的第三阶段称作I/O完成（I/O completion）。当驱动程序调用IoCompleteRequest通知I/O管理器它正在处理IRP中指定的请求（以及它所拥有的栈位置）时，将初始化该阶段。该步骤因I/O操作的不同而不同。例如，全部的I/O服务都把操作的结果记录在由调用程序提供的数据结构I/O状态块中。与此类似，一些执行缓冲I/O的服务要求I/O系统返回数据给调用线程。

在上述两种情况中，I/O 系统都必须把一些存储在系统内存中的数据复制到调用程序的虚拟地址空间中。如果 IRP 是同步完成的，那么调用程序的地址空间就是当前可直接访问的；但是如果 IRP 是异步完成的，则 I/O 管理器必须延迟 IRP 的完成直到能够访问调用程序的地址空间为止。为了获得调用程序的虚拟地址，I/O 管理器必须在调用程序的线程环境中进行数据传输——也就是说调用程序线程正在执行（这意味着调用程序的进程是当前进程并具有在处理器上活动的地址空间）。它通过把一个内核模式的异步过程调用（APC）排队到线程中来完成这个操作的。该处理如图 9-29 所示。

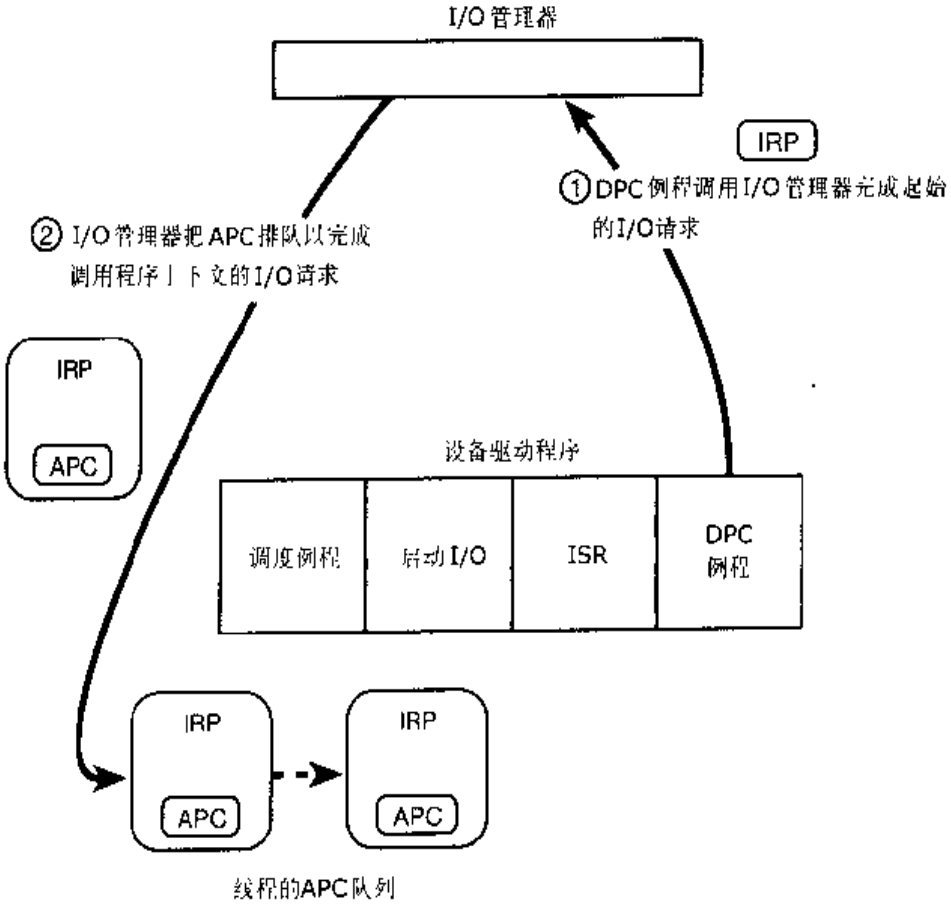


图 9-29 完成 I/O 请求（阶段 1）

正如第 3 章中所解释的，APC 在特定线程的环境中执行，而 DPC 在任意线程的环境中执行。这就意味着 DPC 例程不能触及用户模式进程的地址空间。同时请记住，DPC 具有比 APC 更高的软件中断优先级。

接下来，线程开始在较低的 IRQL 上执行；挂起的 APC 被发送。内核把控制权转交给 I/O 管理器的 APC 例程。该例程将把数据（如果有）和返回的状态复制到初始调用程序的地址空间，释放代表 I/O 操作的 IRP，并将调用程序的文件句柄（或调用程序提供的任何事件或 I/O 完成端口）设置为有信号状态。现在才可以认为 I/O 完成了。正在等待文件（或其他对象）句柄的最初调用程序或其他线程将从等待状态中释放出来并准备执行。

图 9-30 说明了 I/O 完成的第二阶段。

关于 I/O 完成最后要注意的是：异步 I/O 函数 ReadFileEx 和 WriteFileEx 允许调用程序提供

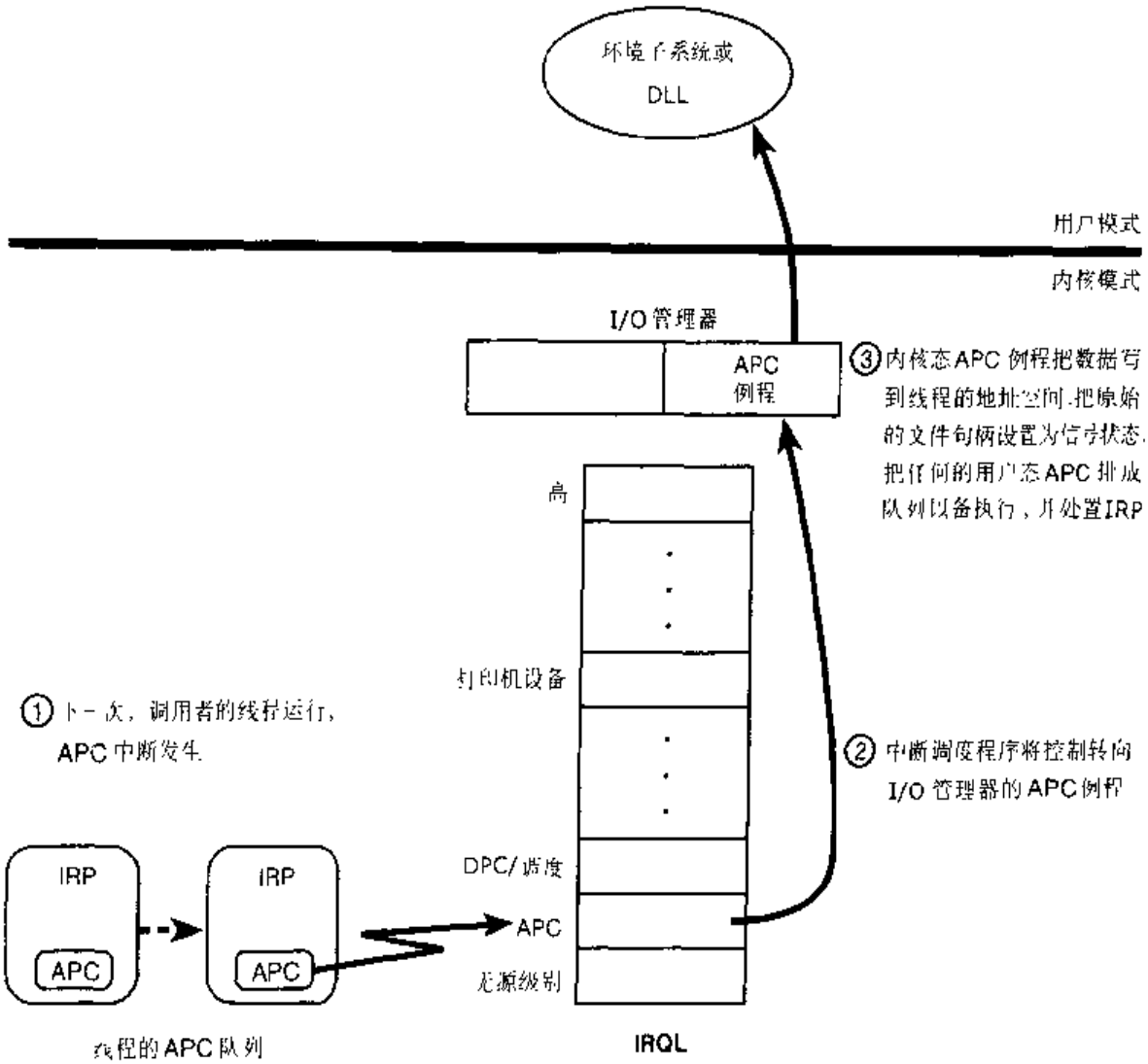


图 9-30 完成 I/O（阶段 2）

用户模式的 APC 作为参数。如果调用程序这样做了，I/O 管理器将把该 APC 插入到调用程序线程的 APC 队列中，以此作为 I/O 完成的最后一步。这个功能允许调用程序指定在 I/O 请求完成或被取消时将被调用的子程序。用户模式的 APC 例程在请求线程的环境中执行，并且只有在该线程进入可报警的等待状态（如调用 Win32 的函数 SleepEx、WaitForSingleObjectEx、WaitForMultipleObjectsEx）时才被发送。

9.5.3 对分层驱动程序的 I/O 请求

前面一节说明了一个指向由单一设备驱动程序控制的简单设备的 I/O 请求是如何被处理的。基于文件设备的 I/O 处理或对其他分层驱动程序的请求的 I/O 处理在很大程度上是相同的。很明显，主要区别在于一个或多个附加的处理层被加入到模型中。

图 9-31 说明了一个异步 I/O 请求是如何通过分层驱动程序的。该图采用一个由文件系统控制的磁盘作为例子。

I/O 管理器再次接到请求并创建 I/O 请求包来代表它。然而，这一次它把请求包发送给文件系统驱动程序。在那里，文件系统驱动程序能够对 I/O 操作行使很强的控制。根据调用程序

发出的请求类型，文件系统驱动程序可以把同一个 IRP 发送给磁盘驱动程序或者生成另外的 IRP 并把它们分别发送给磁盘驱动程序。

如果文件系统收到的请求将一个简单的直接请求转换为设备的话，那么它很有可能重用一个 IRP。例如，如果应用程序发布了读取请求，请求读取存储在软盘上文件中前 512 个字节，FAT 文件系统只是简单地调用磁盘驱动程序，要求它在文件的起始位置开始，从软盘读取一个扇区。

为了在分层驱动程序的请求中容纳多个驱动程序的重用，IRP 必须包含一系列 IRP 堆栈位置（不要与线程用来存储函数参数和返回地址的堆栈混淆）。每一个将被调用的驱动程序都有一个这样的数据区域，包含了每个驱动程序为了执行它自己的那部分请求所需要的信息——例如，功能码、参数和驱动程序的环境信息。如图 9-31 所示，当 IRP 从一个驱动程序传送到另

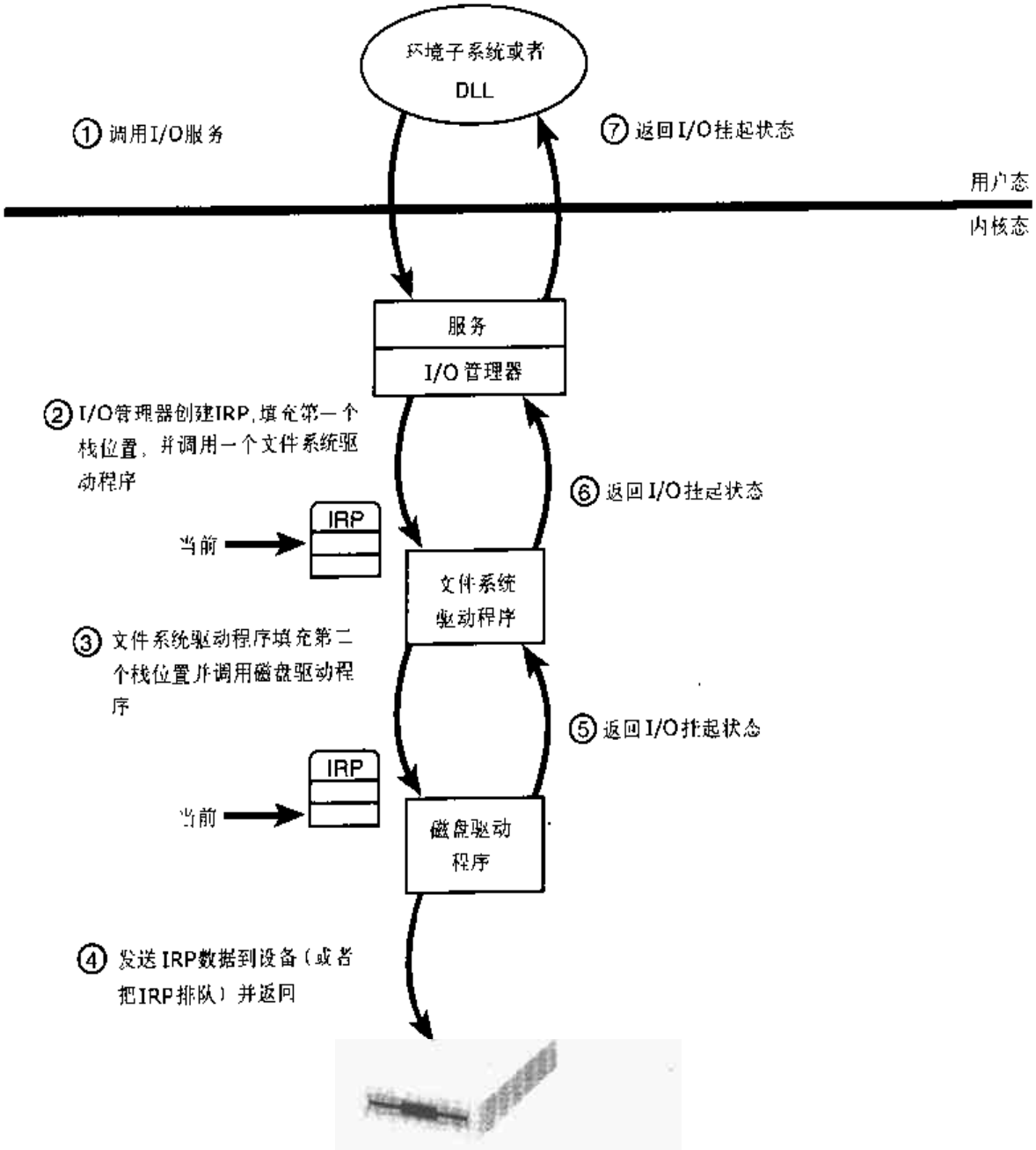


图 9-31 将一个异步请求插入到分层的驱动程序中

一个驱动程序时会填写附加的堆栈位置。你可以这样认为，在IRP的生存期内，从给它添加和从它当中删除数据的方式来看，IRP类似于一个堆栈。然而，IRP并不与任何特殊的进程相关，并且给它所分配的大小并不增加也不会缩小。I/O管理器是在I/O操作开始时，从非页式系统内存中给IRP分配内存。

在磁盘驱动程序完成数据传输之后，磁盘中断并且I/O完成。这如图9-32所示。

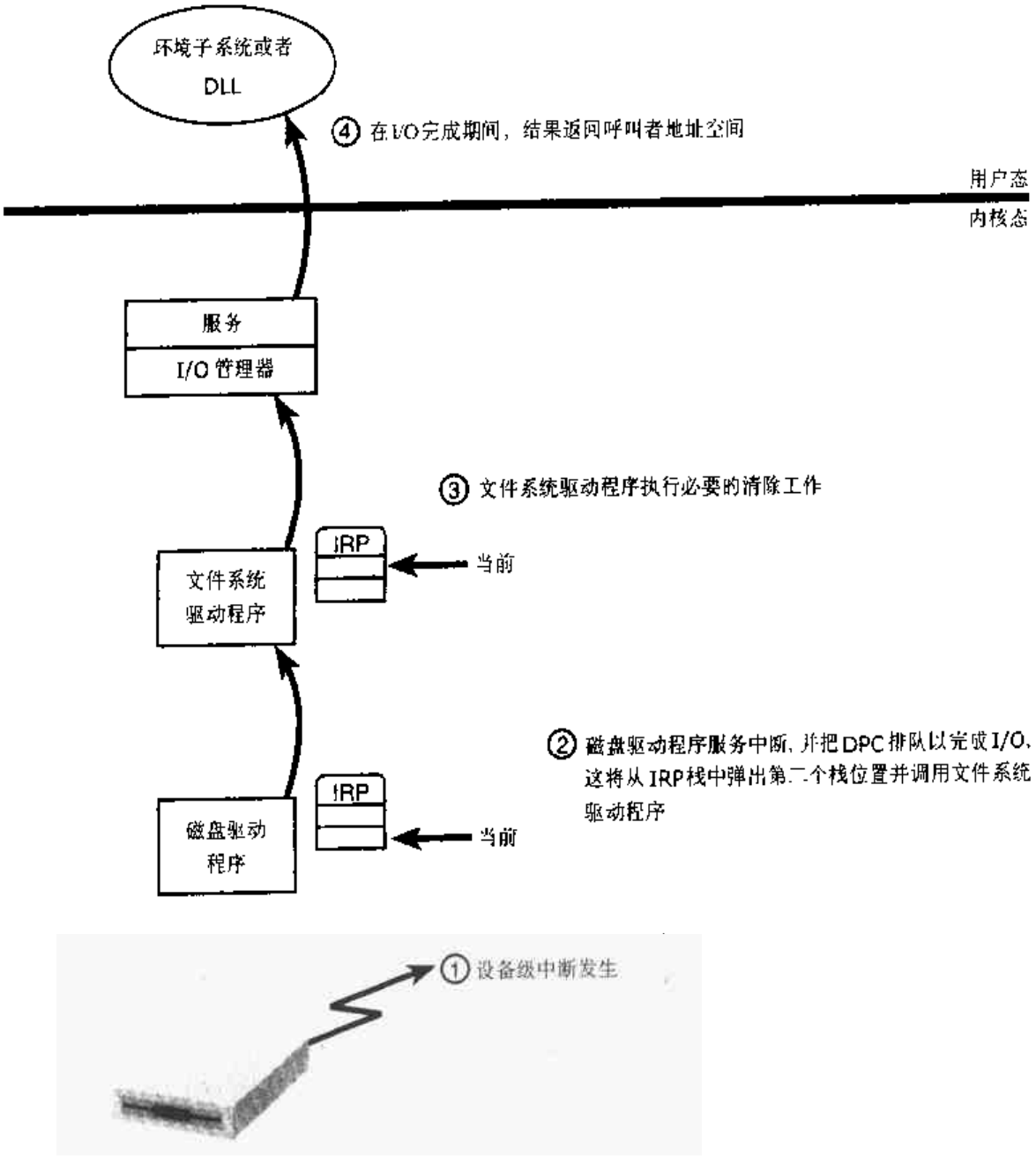


图9-32 完成分层的I/O请求

作为对IRP重用的替代方式，文件系统可以建立一组关联的IRP。这些IRP在单个I/O请求中并行工作。例如，如果将从文件中读取的数据分散在磁盘中，文件系统驱动程序就可以创建几个IRP，每个IRP从不同的扇区读取所请求数据的一部分。图9-33说明了该排队过程。

文件系统驱动程序将关联IRP发送给设备驱动程序。设备驱动程序把它们排入设备队列中。

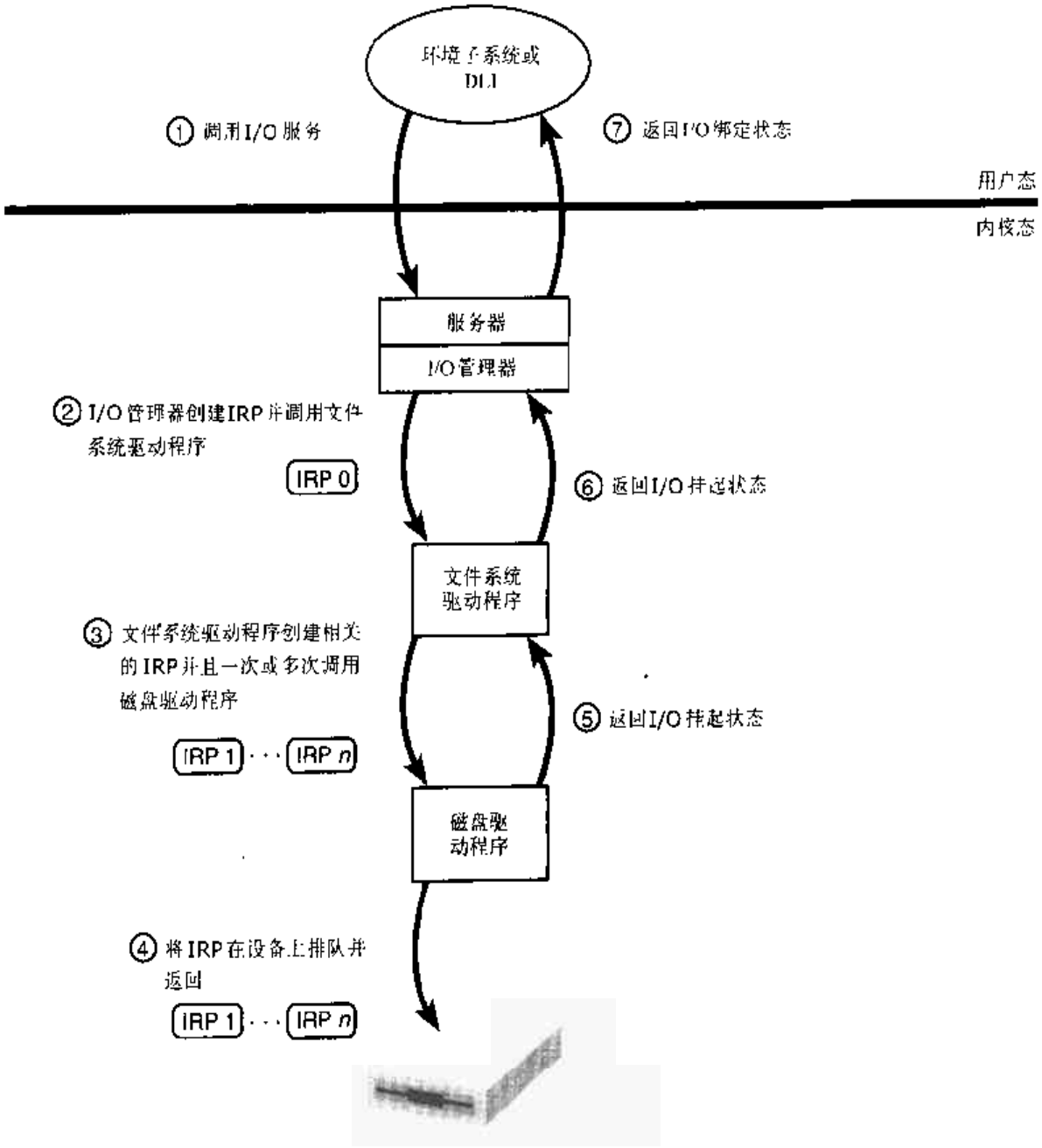


图 9-33 排队关联的 IRP

每次处理一个 IRP，文件系统驱动程序跟踪返回的数据。当所有关联的 IRP 都完成后，I/O 系统就完成最初的 IRP 并返回到调用程序。该过程显示在图 9-34 中。

注意 所有管理基于磁盘的文件系统的文件系统驱动程序都是驱动程序堆栈的一部分。这些堆栈至少有三层深度：文件系统驱动程序在顶层，卷管理器在中间，磁盘驱动程序在底层。此外，任何过滤器驱动程序都可以插入到这些驱动程序的上面或下面。为清晰起见，前面分层 I/O 请求的例子只包含了一个文件系统驱动程序和一个磁盘设备驱动程序。关于详细信息，请参阅第 10 章，存储管理。

9.5.4 I/O 完成端口操作

Win32 应用程序调用 Win32 API 函数 CreateIoCompletionPort 并指定一个空的完成端口句柄创

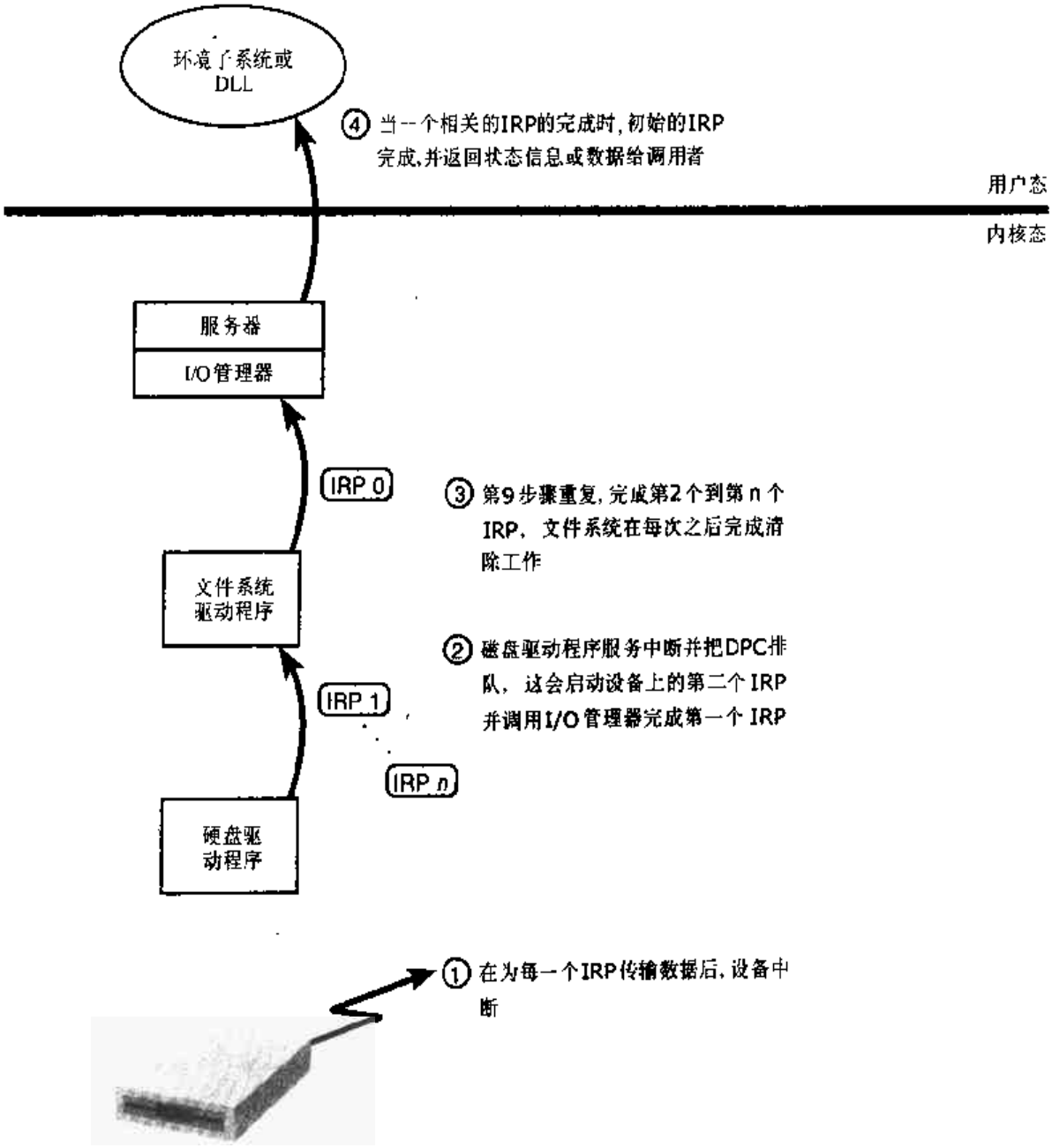


图 9-34 完成关联的 IRP

建完成端口。这将导致系统服务 NtCreateIoCompletion 的执行。该可执行的 IoCompletion 对象是建立在一个称为队列（queue）的内核同步对象的基础上的。这样，系统服务创建一个完成端口对象并在分配给端口的内存中初始化一个队列对象（还有一个端口指针指向队列对象，因为队列是在端口内存的起始位置）。队列对象有一个并行值。该值是在线程启动的时候指定的。在这种情况下使用的值就是传递给 CreateIoCompletionPort 的值。KeInitializeQueue 是 NtCreateIoCompletion 用来初始化端口队列对象的函数。

当应用程序调用 CreateIoCompletionPort 将文件句柄与端口关联起来时，系统服务 NtSetInformationFile 利用该句柄作为主要参数而执行。所设置的信息类别是 FileCompletionInformation，完成端口的句柄以及来自于 CreateIoCompletionPort 的 CompletionKey 参数是数据值。NtSetInformationFile 废弃获取文件对象的文件句柄，并分配一个完成环境数据结构。

最后 NtSetInfomationFile 设置文件对象中的 CompletionContext 字段，使其指向该环境结构。当文件对象的异步 I/O 操作完成时，I/O 管理器将检查文件对象的 CompletionContext 字段是否为非空。如果是，I/O 管理器将分配一个完成包，并调用 KeInsertQueue 将该包插入到完成端口队列中（注意完成端口对象和队列对象是同义的）。

当服务器线程唤醒 GetQueuedcompletionStatus，系统服务 NtRemoveIoCompletion 将执行。在验证参数并将完成端口句柄转换成端口指针后，NtRemoveIoCompletion 将调用 KeRemoveQueue。

你将看到，KeRemoveQueue 和 KeInsertQueue 是完成端口背后的引擎。它们决定等待 I/O 完成包的线程是否被激活。在内部，队列对象维护当前活动线程的数量以及活动线程的最大数目。如果某线程调用 KeRemoveQueue 时当前数值等于或者是超过该最大值，该线程将被（按照 LIFO 的顺序）插入一个等待完成包的处理的线程列表中。该线程列表挂起该队列对象。线程的控制块数据结构中有一个指针，该指针引用与其相关的队列对象；如果该指针为空，则表示该线程没有与某个队列关联起来。

Windows 2000 凭借线程控制块中的队列指针保持对非活动状态线程的跟踪，因为它们由于完成端口之外的原因而阻塞的。可能导致线程阻塞的调度例程（如 KeWaitForSingleObject、KeDelayExecutionThread 等）将检查线程的队列指针。如果指针不是 NULL，将调用 KiActiveWaiterQueue。该函数是一个与队列相关的函数，将减少与队列关联的活动线程的数量。如果结果数值小于最大值并且至少有一个完成端口在队列中，那么在队列线程列表前面的线程将被唤醒，并分配一个最老的包。反之，一个与队列相关的线程在阻塞之后无论什么时候唤醒，调度例程都将执行函数 KiUnwaitThread，增加队列的活动线程数。

最后，Win32 API 函数 PostQueuedCompletionStatus 将导致系统服务 NtSetIoCompletion 的执行。该函数只是简单地利用 KeInsertQueue 将指定的包插入到完成端口队列中。

图 9-35 显示一个操作中的完成端口例子。即使有两个线程准备处理完成包，但是并行值 1 只允许一个与该端口关联的线程活动，结果这两个线程就阻塞在该完成端口上。

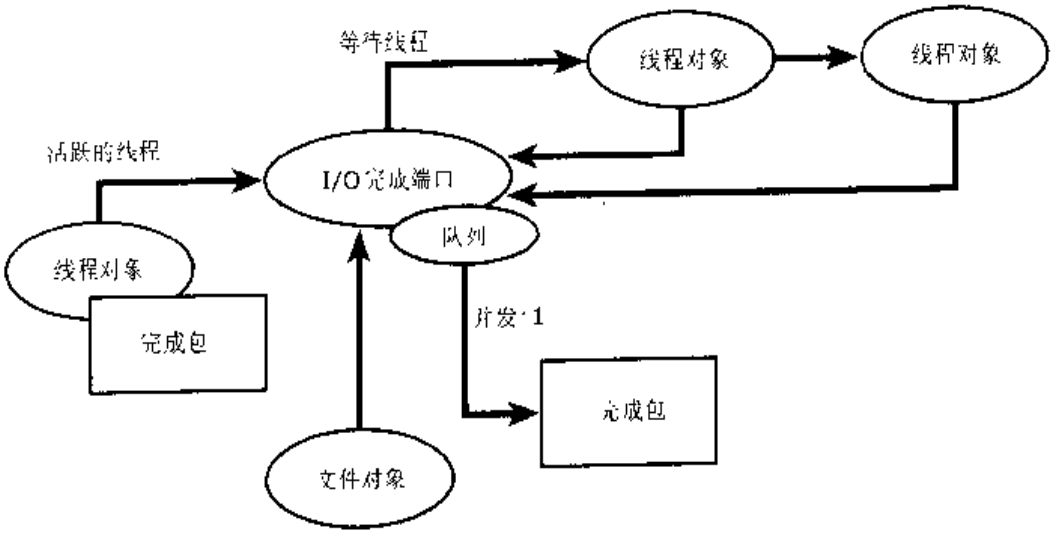


图 9-35 I/O 完成端口操作

9.5.5 同步

驱动程序必须同步执行它们对全局驱动程序数据和硬件寄存器的访问。这有两个原因：

■ 驱动程序的执行可以被高优先级的线程和时间片（或时间段）过期而抢先，或者是被中断所中断。

■ 在多处理器系统中，Windows 2000能够同时在多个处理器上运行驱动程序代码。

如果没有同步，将会发生崩溃。例如，因为设备驱动程序代码运行在无源的IRQL上，所以当调用程序初始化一个I/O操作时，它就可能被设备中断请求所中断，从而导致设备驱动程序的ISR执行而此时它所拥有的设备驱动程序已经运行。如果设备驱动程序正在修改其ISR也要修改的数据，如设备寄存器、堆存储器或静态数据，则在ISR执行时数据可能被破坏。图9-36说明了这个问题。

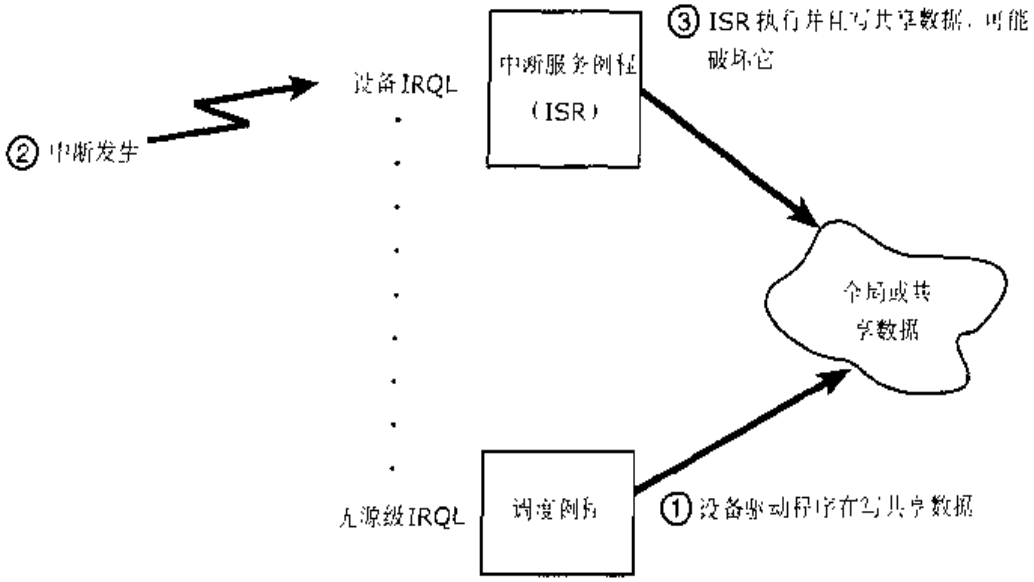


图9-36 将一个异步请求插入到分层的驱动程序

要避免这种情况，为Windows 2000编写的设备驱动程序就必须同步它对任何由设备驱动程序和ISR共享的数据的访问。在尝试更新共享数据之前，设备驱动程序必须锁定所有其他的线程（或CPU，在多处理器系统的情况下），以防止它们修改同一数据结构。

Windows 2000的内核提供了设备驱动程序访问其ISR也要访问的数据时必须调用的特殊的同步例程。这些内核同步例程当共享数据被访问时将禁止ISR的执行。在单CPU系统中，这些例程在更新一个结构之前把IRQL提高到一个指定的级别。然而，在多处理器系统中，由于一个驱动程序能同时在两个或两个以上的处理器上执行，所以这种技术就不足以阻止其他访问者。因此，另一种被称为“自旋锁”的机制被用来锁定来自特定CPU的独占访问的结构。（“自旋锁”在3.3.1节的“内核同步”中有解释。）

到目前为止，你应该意识到尽管ISR需要特殊的注意，但设备驱动程序使用的所有数据都将面临运行于另一个处理器上的同样设备驱动程序的访问。因此，对于设备驱动程序代码，同步它对所有全局的或共享数据使用（或其自身物理寄存器的访问）是很关键的。如果数据被ISR使用，设备驱动程序就必须使用内核同步例程；否则设备驱动程序可以使用内核自旋锁。

9.6 小结

I/O系统定义了Windows 2000中上的I/O处理模型，并且执行公用的或被多个驱动程序请

求的功能。它的主要职责是创建代表 I/O 请求的 IRP，并引导数据包通过不同驱动程序，在完成 I/O 时将结果返回给调用程序。I/O 管理器利用 I/O 系统对象来查找不同的驱动程序和设备，其中包括驱动程序对象和设备对象。在内部，Windows 2000 I/O 系统以异步操作方式获得高性能，并且向用户模式应用程序提供同步和异步 I/O 功能。

设备驱动程序不仅包括传统的硬件设备驱动程序，还包括文件系统、网络和分层过滤器驱动程序。所有驱动程序都具有相同的结构，并通过使用公用机制在彼此之间和 I/O 管理器之间进行通信。I/O 系统接口允许使用高级语言编写驱动程序以节省开发时间并增强它们的可移植性。因为驱动程序对操作系统呈现共同的结构，所以它们可以被分层，即把一层放在另一层上来达到模块化，并可以减少在驱动程序之间的复制。同样，所有的 Windows 2000 设备驱动程序都应被设计成能够在多处理器系统下正确工作。

最后，PnP 管理器的作用是与设备驱动程序一起工作，动态地检测硬件设备，建立内部设备树，指导设备的枚举以及驱动程序的安装。电源管理器与设备驱动程序协作，在可用的情况下将设备转移到低能耗状态，以节省能量并延长电池寿命。

最后四章涉及传统的与 I/O 系统相关的题：存储管理、文件系统（包括 NTFS 文件系统的细节）、高速缓存管理以及网络。

第10章 存储管理

存储管理定义了操作系统和非易失性设备及介质的接口方式。术语“存储”包含许多不同的设备，包括磁带设备、光媒介、CD光盘机、软盘和硬盘，微软的Windows 2000提供了对这些不同存储设备的特殊支持。因为本书关注的重点是Windows 2000的核心组件，在本章我们将主要关注Windows 2000的硬盘存储子系统的基础。Windows 2000中重要的对可移动介质和远程存储（离线存储）的支持是在用户模式下实现的。

在本章，将定义什么是基本磁盘和动态磁盘、并且解释它们是如何被分区的、系统的内核模式设备驱动程序如何作为文件系统和磁盘介质之间的接口、win2000中关于多分区磁盘管理特征的实现——包括为了提高可靠性和改善性能，如何实现系统数据的复制和在物理磁盘上分割文件系统数据。在最后我们通过实际观看Windows 2000分配驱动符的过程来总结本章的内容，同时讨论文件系统驱动程序如何装配它们负责管理的卷。

10.1 Windows 2000存储的演化历史

Windows 2000的存储管理的演化起源于微软的第一个操作系统MS-DOS，当硬盘变得很大时，MS-DOS需要实现对大容量硬盘的支持能力，为了实现这个目的，微软的第一步是让MS-DOS允许在一个物理盘上建立多个分区和逻辑盘。MS-DOS可以使用不同的文件系统类型如FAT12或FAT16格式化每个分区，并赋予每个分区一个不同的驱动符。MS-DOS版本3和版本4严格地限制它们创建的分区的大小和个数。，但是在MS-DOS 5的版本上分区模式变得很成熟了，在MS-DOS 5中，可以将一个盘分成任意大小和数量的区。

Windows NT借鉴了MS-DOS的分区模式，提供了同MS-DOS和Windows3.x分区格式的兼容性，同时可以让开发组用可靠的软件工具实现硬盘的管理。微软扩展了Windows NT中MS-DOS硬盘分区的基本概念，以支持企业级用户对硬盘管理的特殊要求：如硬盘跨越（disk spanning）和容错能力。从Windows NT的第一个版本Windows NT 3.1开始，系统管理员可以创建包含多个分区的卷，即允许一个大的卷包含多个不同物理硬盘的分区。同时通过实现基于软件的数据冗余实现容错能力。

虽然早于Windows 2000的在Windows NT中的MS-DOS风格的分区已经提供了足够的灵活性支持绝大多数的存储管理任务，但是它仍然有许多缺陷。一个缺陷是绝大多数的硬盘配置变化在生效前要求重新启动机器。在今天的社会里，服务器必须维持几个月的甚至几年的在线服务，任何重启，甚至是有计划的重启，都是很大的不方便。另一个缺陷是Windows NT 4中将用于MS-DOS分区风格的多分区硬盘配置信息存放在注册表中，这种安排意味着当你在系统间移动系统时，移动配置信息变得十分的费力，也就意味着当你需要重新安装操作系统时，很容易丢失配置信息。最后，Windows 2000的早期Windows版的系统必须为每个可能创建的本机卷和远程卷分配一个唯一的驱动符，驱动符的范围被限制在从字符A到Z。

为了更好的理解本章的其余部分，你必须熟悉以下基本术语。

■ 磁盘是物理存储设备，例如硬盘、3.5 英寸软盘、CD－ROM 等。

■ 磁盘被分为扇区、固定大小的编址块，扇区的大小是由硬件决定的。目前所有的 X－86 处理器硬盘扇区大小都是 512 个字节，典型 CD－ROM 扇区大小是 2048 个字节（未来的 X－86 处理器可能会支持更大的硬盘扇区大小）。

■ 分区是磁盘上连续扇区的集合。分区表或其他磁盘管理数据库中存放有分区的开始扇区、大小和其他分区特征。

■ 简单卷代表的是文件系统驱动器可以当作单个单元管理的单个分区的扇区对象。

■ 多分区卷表示的是多个分区的扇区，并且可以被文件系统驱动管理当作单一单元来管理。多分区卷提供单一卷不能提供的性能、可靠性和大小特性。

10.2 分区

在 Windows 2000 中引入了基本磁盘和动态磁盘的概念，Windows 2000 将依靠 MS-DOS 风格的分区方案的盘称为基本磁盘。换而言之，基本磁盘是 Windows 2000 的遗留盘。动态磁盘是 Windows 2000 新的概念，通过动态磁盘可以实现比基本磁盘更灵活的分区机制。基本磁盘和动态磁盘最基本的不同在于动态磁盘支持动态地创建新的多分区卷（基本磁盘仅仅支持向前兼容升级的 Windows NT 4 的安装）。回顾前一章所列的术语我们可以看出多分区卷可以提供简单卷不能提供的性能、大小以及可靠性方面的支持。简单磁盘的多分区卷配置信息存储在注册表中，而动态磁盘的多分区卷配置信息存储在磁盘上，将多分区卷配置信息存储在磁盘上而不是在存储在注册表中使动态磁盘与它描述的介质紧密相关，这样不太可能丢失数据并且在系统之间移动数据变得简单。

如果你不是手工地创建动态磁盘或将现存的基本磁盘转换为动态磁盘（具有足够的自由空间），Windows 2000 将所有的盘当作基本磁盘来管理。为了鼓励管理员使用动态磁盘，微软已经对基本磁盘的使用场合做了限制。例如，你仅可以在动态磁盘上创建一个新的多分区卷，（如果你更新一个 Windows NT 4 的系统，Windows 2000 将支持现存的多分区卷）另一个限制是 Windows 2000 仅让你在动态磁盘上动态地扩展卷的容量。动态磁盘的一个不利之处是它们使用的分区格式是专有的同其他的操作系统不兼容，包括其他版本的 Windows。因此，你不能在一个多引导环境中访问动态磁盘。

注意 由于很多原因，包括膝上机往往只有一个磁盘，并且膝上机的磁盘往往不容易在计算机间移动等。Windows 2000 在膝上机上仅使用基本磁盘。另外，仅有固定的磁盘可以是动态磁盘，其他位于 IEEE 1394 或 USB 总线上和共享的集群服务器上的磁盘总是基本磁盘。

10.2.1 基本分区

当你在计算机安装 Windows 2000 的时候，第一步要做的事情是在系统的主物理磁盘上创建一个分区。Windows 在这个分区上定义系统卷用于存储引导系统启动过程的代码。另外，Windows 2000 安装程序要求你创建一个分区用于引导卷的家（home）。Windows 2000 安装程序把系

统文件放在家中，同时创建一个系统目录（\WinNT）。系统卷和引导卷可以在同一个卷上，你没有必要新建一个分区用于引导卷。微软定义的系统卷和引导卷的术语有时候令人迷惑。系统卷是 Windows 2000 用于存放引导文件包括引导加载程序（NTldr）和 NTdetect，而引导卷是 Windows 2000 用于存放系统文件如 Ntoskrnl.exe 和其他内核代码文件的地方。

X86 硬件使用的标准 BIOS 实现要求采用 Windows 2000 的分区格式。即主磁盘的第一扇区包含有主引导记录（MBR）。当 x86 处理器引导的时候，计算机的基本输入输出设备（BIOS）读取主引导记录内容，并将主引导记录内容当作可执行的代码执行。当 BIOS 执行了必要的计算机硬件配置后，BIOS 调用 MBR 代码来初始化操作系统引导进程。在微软的操作系统包括 Windows 2000，主引导记录同样包含一个分区表，一个分区表有四个项，每个项定义了最多四个主分区的位置，分区表同样也记录了分区的类型，有些预定义的分区类型存在，同时每个分区的类型表示了在这个分区类型上可以存在什么样的文件系统类型。例如，FAT32 和 NTFS 分区类型，一个特殊的分区类型——可扩展分区类型——包括带有自己的分区表的其他 MBR。通过使用可扩展分区类型，微软的操作系统可以克服每个磁盘只有四个分区的限制。总之，关于可扩展分区的概念可以无限的扩大，也就是意味着在一个磁盘上可以存在无限多个分区而不受限制，在图 4-1 显示了一个磁盘分区的方案，Windows 2000 的引导进程明显地区分主分区和可扩展分区。系统必须将主磁盘的一个主分区标识成活动的，在 MBR 中的 Windows 2000 代码加载存储在系统卷上的活动分区的第一个扇区的代码到内存中，然后将控制交给这段代码。因为主分区的第一扇区承担着引导进程的角色，Windows 2000 将任何分区的第一扇区设计为引导扇区，回忆第 4 章所说的，每一个被文件系统格式化的分区都有一个引导扇区，在引导扇区中存放有关于该分区上的文件系统的结构。

注意 因为 Windows 2000 不支持在基本磁盘上建立多分区卷，一个新的基本磁盘分区（同从 Windows NT 4 的升级版迁移来的多分区卷相反）等同于一个卷，因为这个原因，当你在基本磁盘上创建卷时，磁盘管理 MMC 插件使用的是术语分区。

10.2.2 动态分区

正如我们所叙述的，动态磁盘在 Windows 2000 上是一种受欢迎的磁盘格式，是创建多分区卷的基础。Windows 2000 的 Logical Disk Manager（LDM）子系统由用户模式和设备驱动组件组成，用于监视动态磁盘。微软授权以前为 UNIX 开发 LDM 的 VERITAS 软件公司开发微软的 LDM。同微软密切合作，VERITAS 将它的 LDM 移植到了 Windows 2000 上，为操作系统提供更灵活的分区和多分区卷的能力。LDM 分区同 MS-DOS 风格下的分区的一个主要不同是 LDM 维护的是一个统一的数据库，在这个数据库中存放有系统所有动态磁盘包括多分区卷配置的分区信息。UNIX 版本的 LDM 包含有磁盘组的概念，也就是系统将所有的动态磁盘以磁盘组的方式共享通用数据库。VERITAS 为 Windows 2000 开发的商业卷管理软件同样也包含磁盘组的概念，但是 Windows 的 LDM 仅包含有一个磁盘组。

LDM 数据库位于每一个动态磁盘的最后 1M 保留空间。需要这块空间的原因是 Windows 2000 在你将一个基本磁盘转换为动态磁盘前，要求在基本磁盘的最后有自由空间。虽然动态磁盘的分区信息存放在 LDM 数据库中，但是 LDM 实现了一个 MS-DOS 风格的磁盘分区，可以

让遗留磁盘管理工具——包括运行在 Windows 2000 下的和其他工作在多重引导环境下的工具——不会误认为动态磁盘没有分区。因为 LDM 分区没有用 MS-DOS 风格描述，所以它们被称为“软分区”，MS-DOS 分格的分区则称为“硬分区”。

其他 LDM 创建 MS-DOS 风格分区表的原因是 Windows 2000 的启动代码可以在即使卷位于动态磁盘的情况下，找到系统卷和启动卷。（例如，NTldr 对 LDM 的分区一无所知）如果一个磁盘含有系统卷和引导卷，在 MS-DOS 分区表的硬分区描述了这些卷的位置。否则的话，硬分区起始于磁盘的第一柱面（典型的磁盘上的 63 个扇区），并且扩展到 LDM 数据库的开始位置。LDM 将这个分区标识为 LDM 类型，一种对 Windows 2000 来说是新的 MS-DOS 风格的分区类型。存放 MS-DOS 分区的地方是 LDM 创建软分区的地方，并且由 LDM 数据库组织。图 10-1 显示了这种动态磁盘布局。

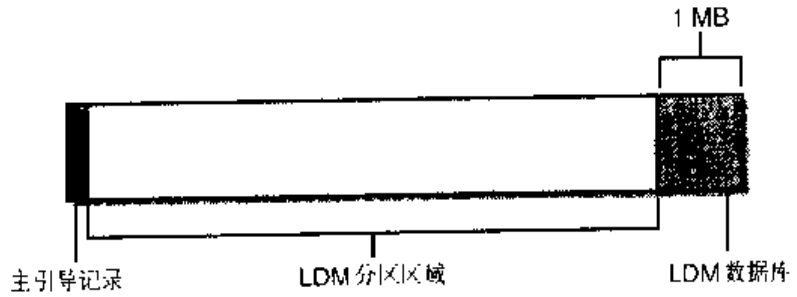


图 10-1 内部动态磁盘组织

LDM 由四个区组成，见图 10-2 所示：LDM 称为“专用头”（private header）的头扇区（header sector）、内容区域表、数据库记录区和事务日志区（图 10-2 中显示的第五个区域仅仅是专用头的简单拷贝）。专用头扇区位于动态磁盘最后 1M 空间的位置，当你漫游 Windows 2000 的时候，你会注意到 Windows 2000 使用 GUID 来标识每件事物，磁盘也不例外。GUID（全局唯一标识）是一个 128 位的值用于唯一标识 Windows 2000 中的对象。LDM 赋予系统中每一个动态磁盘一个 GUID，因此专用头的信息对于磁盘来说是专用的。专用头同样存有磁盘组的名字，名字由计算机的名字和“Dg0”连接而成，（例如，如果计算机的名字是 Susan，则磁盘组的名字就是 SusanDg0）和指向内容区域表开始位置的指针（类似早先提到的，LDM 的 Windows 2000 版实现仅包括一个磁盘组，因此磁盘组的名字将总是以 Dg0 结尾）。为了可靠性，LDM 在磁盘的最后扇区保留了专用头的拷贝。

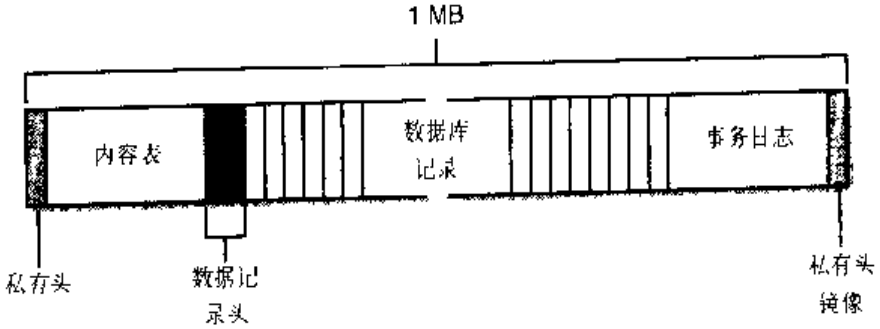


图 10-2 LDM 数据层

数据库表内容有16个扇区大小，并且包含有数据库布局的信息。LDM开始于数据库记录区域，随后紧接着作为数据库记录头的一个扇区的记录表。在这个扇区中存放有关于数据库记录区域的信息，包括它所包含的记录的个数，同数据库相关的磁盘组的名字和GUID，LDM用来在数据库中创建下一个项的序列号标识符。紧接着数据库记录头的扇区包含有128字节固定大小的记录，用来存储描述磁盘组的分区和卷信息。

一个数据库项可以是四种类型中的一种：分区、磁盘、组件和卷，LDM用数据库项的类型判断所描述的卷的三种层次。LDM把带有内部对象标识符的项连在一起。在最低层，分区项描述的是磁盘上集中的区域即软分区；存储在分区项的标识符把项连接成组件和磁盘项，磁盘项标识的是磁盘组的一部分，也就是动态磁盘和磁盘的GUID。组件项的作用是在一个或者多个分区项和每一个分区项相关的卷项间建立连接关系。卷项存储着一个卷的GUID、卷的全部大小、状态和驱动符暗示。比数据库记录大得多的磁盘项跨越多个记录；分区、组件和卷项很少跨越多个记录。

LDM要求三种项来描述简单的卷：分区，组件和卷项。下面的列表显示了一个简单的LDM数据库的内容，数据库描述了一个有200M空间卷的分区。

```
Disk Entry      Volume Entry    Component Entry    Partition Entry
Name: Disk1     Name: Volume1   Name: Volume1-01   Name: Disk1-01
GUID: XXX-XX... ID: 0x408       ID: 0x409          ID: 0x407
Disk ID: 0x404  State: ACTIVE   Parent ID: 0x408   Parent ID: 0x409
                Size: 200MB                        Disk ID: 0x404
                GUID: XXX-XX...                    Start: 300MB
                Drive Hint: H                      Size: 200MB
```

分区项描述了在磁盘上系统用来分配给卷的一块区域，组件项连接了分区项和卷项，卷项包含有Windows 2000用于内部标识卷的GUID。多分区卷需要三个以上的项。例如，一个带区卷（带区卷将在以后的章节介绍）包含有至少两个分区项、一个组件项和一个卷项。有不止一个组件项的唯一卷类型是一个镜像卷。LDM使用两个组件项用于镜像的目的是当你破坏了一个镜像的时候，系统可以在组件层分离它，同时创建两个卷，每个卷包含一个组件项。因为一个简单卷需要三个项和用来存放大约8000个项的1M大小的数据库，所以你可以在Windows 2000上建立的卷的个数的有效上限是2500个。

LDM数据库的最后区域是事务日志区域，用来备份当信息修改时存储备份的数据库信息。这种安装可以在系统崩溃或系统掉电的情况下保护数据库，LDM可以利用日志恢复数据库的一致状态。

实验：使用dmDiag工具观察LDM数据库

在Windows 2000的工具箱中包含有一个叫做DmDiag的工具，你可以用它查看系统物理磁盘、LDM数据库内容的详细信息、MS-DOS分区风格的磁盘分区的布局。除了同磁盘相关的数据信息外，LDM同时可以转储系统定义的装配点（关于装配点的介绍在本章的后面介绍）和设备对象以及在对象管理空间中的符号链接。

DmDiag 使用非命令行的方式，它的输出经常不止一屏幕，因此你可以使用重定向的方法将输出变换为文本文件以便可以在文本编辑器中查看，例如，使用命令 dmdiag > disk.txt，图 10-3 显示了 DmDiag 输出的摘录。首先是对系统磁盘类型的一个描述，然后是 LDM 数据库记录的列表，本图中描述了一个 1GB 大小的简单卷的情况。卷的数据库项列为 Volume1。

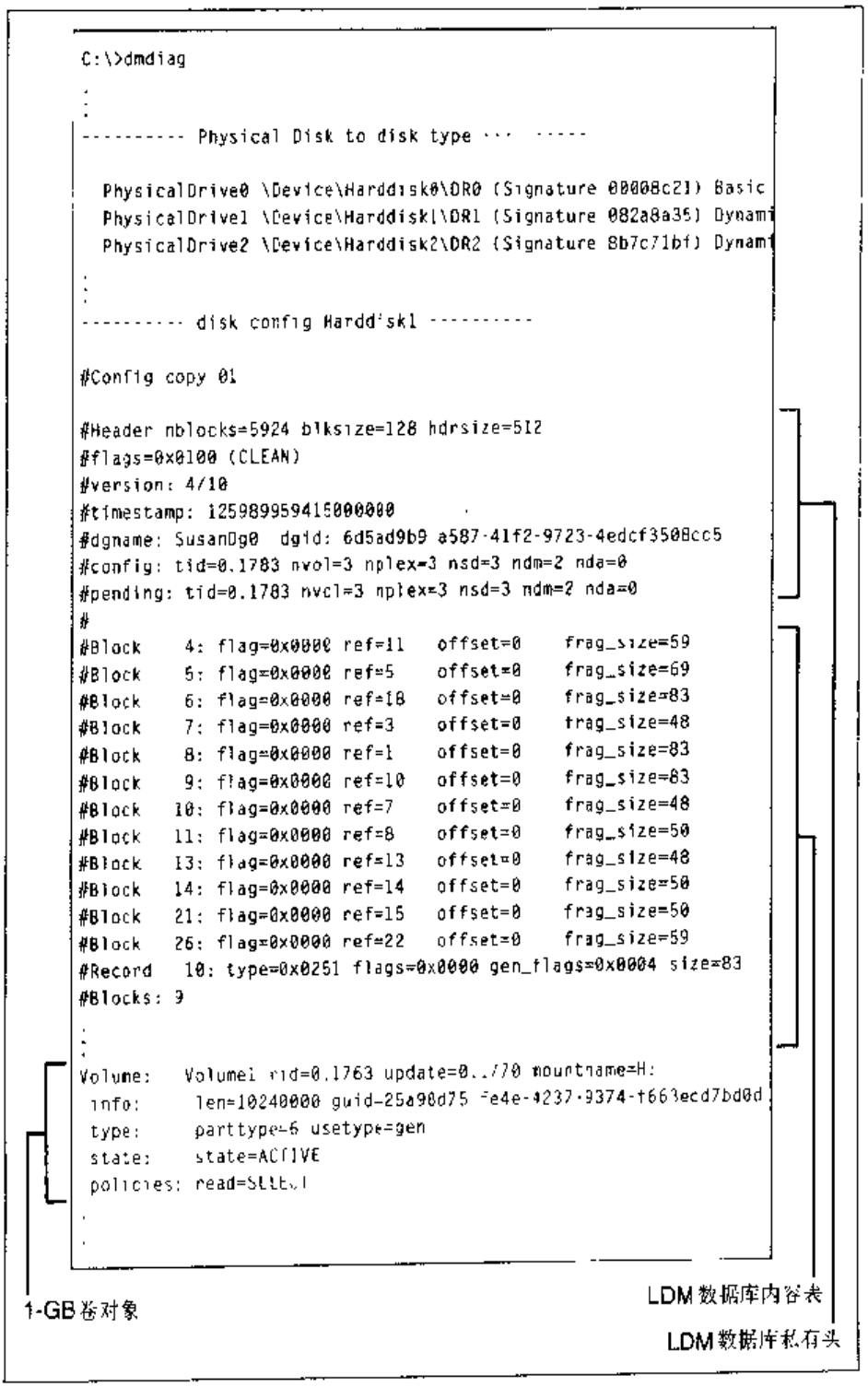


图 10-3 使用 dmDiag 工具观察 LDM 数据库

GUID分区表分区

作为提供标准或扩展的firmware平台用于操作系统在引导时使用的动机的一部分，Intel设计了可扩展的firmware平台接口（EFI）规范。EFI包括用firmware（如ROM）实现的微型操作系统环境，操作系统在系统引导时加载系统诊断代码和引导代码。EFI第一个目标是IA－64，这样Windows 2000的64位版本将使用EFI。你可以在developer.intel.com/technology/efi找到有关EFI的详细描述。

EFI定义了新的分区模式，称为GUID分区表（GPT），可以解决MS-DOS分格分区的一些弊端。例如，GPT分区结构的扇区地址使用的是64位而不是32位，一个32位的扇区地址空间访问2TB的存储空间是足够的，今天你不曾看见提供如此存储能力的设备。但是GPT标准的设计是为了以后的考虑。GPT模式的另一个考虑是GPT使用循环检验码（CRC）保证分区表的一致性，并且维护分区表的一个备份拷贝。GPT名字源于以下事实：除了用来为每个分区存储36字节的Unicode分区名外，还赋予了每个分区一个GUID。

图10-4显示了GPT分区布局的一个例子。同Windows 2000中基本磁盘和动态磁盘一样，GPT第一个扇区是MBR，然而，磁盘第二扇区和最后一个扇区存储GPT分区表。由于分区的可扩展能力，GPT分区不需要像MS-DOS一样的网状分区（nested partition）。

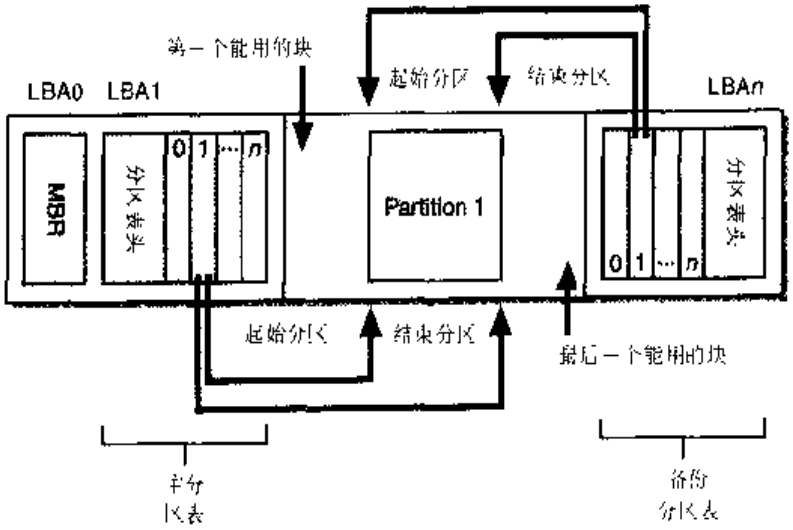


图10-4 GUID分区表分区

注意：LBA——逻辑块地址

虽然GPT分区方案是对MS-DOS风格的分区的改善，但它不是对Windows 2000 LDM分区的替代。就像Windows 2000在动态引导和系统磁盘上集成了MS-DOS风格的分区和LDM分区区域，这样操作系统加载程序能够查找Windows 2000引导和系统文件，在64位Windows 2000中LDM也集成了GPT分区方案。

10.3 存储驱动程序

正如你在第4章所看见的，NTldr是Windows 2000操作系统文件，它构成了Windows 2000

引导进程的第一部分。NTldr 驻留在系统卷上，系统卷上的引导扇区负责执行 NTldr。NTldr 从系统卷上读取 boot.ini 文件，然后将计算机的引导选择交给用户。Boot.ini 文件中命名分区信息是采用 multi（0）disk（0）rdisk（0）partition（1）的方式。这些名字是先进的 RISC 计算（ARC）名字，因为它们是被 alpha firmware 平台和其他 RISC 处理器使用的标准分区命名方式。NTldr 解释 boot.ini 文件中引导项中的名字，使得用户可以选择合适的引导分区，然后加载 Windows 2000 文件系统（开始处理注册表、NToskrnl.exe 和引导驱动程序）到内存以继续引导过程。

10.3.1 磁盘驱动程序

在初始化阶段，Windows 2000 的 I/O 管理者开始加载硬盘存储驱动程序，硬盘存储驱动程序在 Windows 2000 中遵循类/端口/小端口（class/port/miniport）的体系结构，微软提供对所有的存储设备实现了相同功能的类驱动程序，为特殊总线结构——如小型计算机系统接口总线（SCSI）或集成设备电路（IDE）系统——实现了相同功能的端口驱动程序，并且 OEM 提供小型端口驱动程序插入端口驱动程序作为 Windows 2000 到特殊实现的接口。

在磁盘存储驱动程序体系结构中，只有类驱动程序符合 Windows 2000 的设备驱动接口。小型端口驱动程序使用端口驱动程序接口而不是设备驱动程序接口。端口驱动程序简单地实现设备驱动程序支持例程集，它作为小型端口驱动程序与 Windows 2000 的接口。这种方法可以简化小型端口驱动程序开发者的工作，因为微软提供操作系统相关的端口驱动程序模块，给小型端口驱动程序模块提供在微软的 Window 98、Windows millennium Edition 和 Windows 2000 之间的二进制可移植能力。

Windows 2000 包括磁盘（\winNT\system32\drivers\disk.sys），实现了磁盘相同功能的类驱动程序。Windows 2000 同样提供了大量的磁盘端口驱动程序。例如，Scsiport.sys 是用于 SCSI 总线上磁盘的端口驱动程序，Pciidex.sys 是用于基于 IDE 系统的端口驱动程序。Windows 2000 也提供了一些小端口的驱动程序，其中包括一个用于 Adaptec1540 的 scsi 控制器系列的 Ahal54x.sys 驱动程序。在那些至少有一个基于 ATAPIIDE 设备的系统上，驱动程序 Atapi.sys 结合了端口和小型端口的功能。绝大多数 Windows 2000 的安装程序包含一个或者更多上面提到的驱动程序。

10.3.2 设备命名

Windows 2000 的磁盘类驱动程序创建代表磁盘和分区的设备对象。表示磁盘的设备对象用\device\hardDiskX\DR X 的方式标识，使用标识磁盘的数值取代其中的两个 X。磁盘类驱动程序使用 I/O 管理器的 IOReadPartionTable 函数来标识代表分区的设备对象。像小型端口驱动程序模块在引导过程的早期将磁盘标识为类驱动程序，磁盘类驱动程序对每个磁盘调用 IoReadPartionTable 函数，IoReadPartionTable 函数再调用类驱动程序，端口驱动程序和小型端口驱动程序提供的扇区级的 I/O 函数读取 MS-DOS 风格磁盘分区表并构造一个磁盘分区的内部表示。磁盘类驱动程序新建一个设备对象表示每一个由函数 IoReadPartitionTable 返回的主分区（包括可扩展分区内的主分区），下面是分区对象名的一个例子：

\Device\HardDisk0\DP（1）ox7e000 - 0x7ff50c00 + 2

这个名字标识了系统第一个磁盘上的第一个分区。名字的最前两个十六进制数（ox7e000 和 0x7ff50c00）代表了分区的开始和长度。最后一个数是由类驱动程序分配的内部标识符。

为了维持使用 Windows NT 4 的命名规则的应用程序的兼容性。磁盘类驱动程序使用 Windows NT 4 格式的名字建立一个符号链接来引用驱动程序创建的设备对象。例如，磁盘类驱动程序建立连接 \ device \ hardDisk0 \ partition0 来指向 \ device \ hardDisk0 \ DR0 \ device \ hardDisk0 \ partition1 来指向 \ device \ hardDisk0 \ DR1 。类驱动程序同样建立一个相同的 Win32 符号链接代表 Windows 2000 中那些用 Windows NT 系统建立的物理设备。同样，例如，链接 \ ?? \ physicalDriver0 指向 \ Device \ hardDisk0 \ DR0。图 10-5 显示 windObj 工具（在配套的光盘 \ Sysint \ Winobj.exe）显示一个磁盘目录用作基本磁盘的内容。你可以在右手的区域看见物理磁盘和分区设备对象。

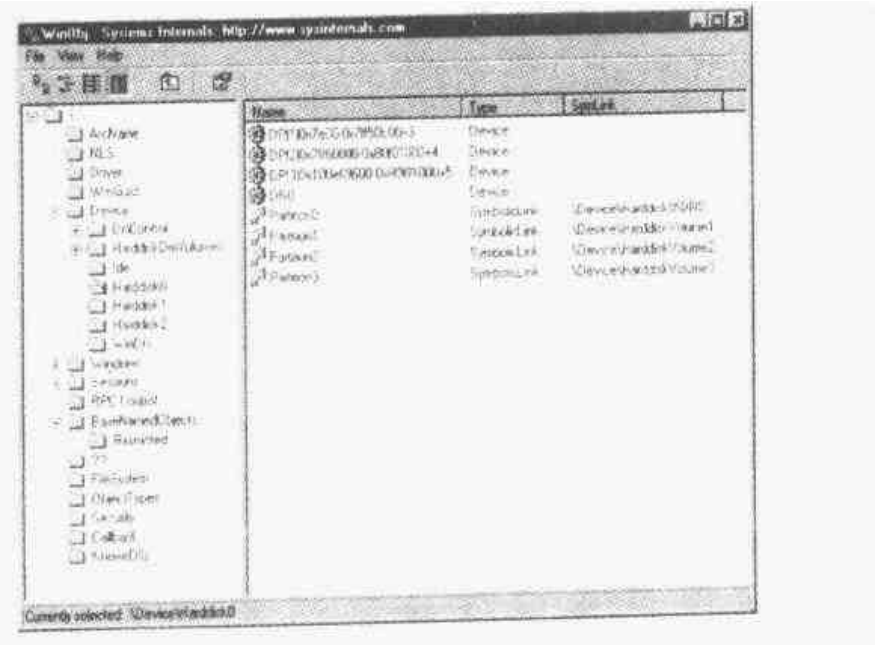


图 10-5 Winobj 显示了基本磁盘的硬盘目录

像你在第 3 章所看见的，Window32 的 API 不知道 Windows 2000 的对象管理名字空间。Windows 2000 保留了两个不同的名字空间子目录供 Win32 使用，其中一个是 \ ?? 子目录，（另一个是 \ BaseNamedObjects 子目录，在第 3 章有叙述）。在这个目录中，Windows 2000 使得 Win32 应用程序交互的设备对象可以获得，包括 COM 和并行端口及磁盘。因为磁盘对象实际驻留在其他子目录中，Windows 2000 使用符号链接把在 \ ?? 下和其他名字空间的对象连接起来。对系统上的每个物理磁盘，I/O 管理器建立一个 \ ?? \ PhysicalDrive X 链接指向 \ device \ HardDiskX \ partition。（用起始 0 的数字代替 x）直接同磁盘扇区交互的 Win32 应用程序使用 Win32 函数 createFile 打开磁盘，其中使用 \ \ . \ PhysicalDriveX（X 是磁盘的编号）为参数。Win32 程序在将名字提交 Windows 2000 对象管理器前，将名字转换为 \ ?? \ PhysicalDrive x 的形式。

10.3.3 基本磁盘管理

FtDisk 驱动程序（ \ winNT \ system32 \ drivers \ FtDisk.sys）创建磁盘设备对象代表基本磁盘上的卷。在 Windows NT 4.0 至少存在一个多分区卷的时候，FtDisk 才存在，在 Windows 2000，

FtDisk 在管理所有的基本磁盘卷 - 包括简单卷——上扮演着重要的角色。对每一个卷，FtDisk 建立一个形式为 \ device \ hardDiskVolume X 的设备对象，在这里 x 是从 1 开始的数字用来标识卷。FtDisk 使用注册表中的键 \ HKLM \ SYSTEM \ Disk 中存储的磁盘配置信息来决定系统包含什么样的基本卷、多分区卷或其他类型的卷。注意到 Windows 2000 中的 Disk 键是唯一从 Windows NT 4 或 Windows 98 上升级而来的，并且由于以前的磁盘管理活动（如将一个驱动符分配给一个卷或建立一个新的多分区卷）已经存在 Disk 键。为了避免对注册表的依赖性，FtDisk 将配置信息从 Disk 键中移到磁盘上隐藏的扇区上。

FtDisk 实际上是个总线驱动程序。因为它负责枚举基本磁盘以检测存在的基本卷，并报告给 Windows 2000 的即插即用（PnP）管理器。在分区管理器（partmgr.sys）驱动程序的帮助下，决定存在什么样的基本磁盘分区。分区管理器注册 PnP 管理器，这样 Windows 2000 可以在磁盘类驱动程序建立一个新的分区设备对象时通知分区管理器。分区管理器通过私用接口通知 FtDisk 新分区设备对象，同时创建分区管理器附接到分区对象的过滤器设备对象。无论什么时候分区设备对象被删除，过滤器对象的存在会促使 Windows 2000 通知分区管理器，这样分区管理器能够更新 FtDisk。当磁盘管理控制台（MMC）插件中的一个分区被删除时，磁盘类驱动程序删除一个分区设备对象。当 FtDisk 了解到分区信息后，它使用基本的磁盘配置信息决定卷相应的分区，并且当它了解到所有用卷描述的区后，就创建了一个卷设备对象。

Windows 2000 卷驱动符分配，一个已经有简短描述的过程，在 FtDisk 创建的卷设备对象所指的 \ ?? 对象管理器目录下创建一个符号链接。当系统或应用程序第一次访问卷时，Windows 2000 执行装配操作，让文件系统驱动程序有机会识别卷和它们管理的文件系统格式化卷。（装配卷的操作在本章的“卷装配”中介绍）。

10.3.4 动态磁盘管理

磁盘管理 MCC 插件 DLL（DMDiskManger，位于 \ winNT \ System32 \ dmdskmgr.dll），如图 10-6 所示，使用 LDM 磁盘管理员服务 DMAdmin（位于 winNT \ System32 \ Dmadmin.exe）来创建和改变 LDM 数据库的内容。当你启动磁盘管理 MMC 插件，如果它还没有运行，DMDiskManger 加载到内存中并启动 DMAdmin。DMAdmin 从每个磁盘中读取 LDM 数据库并将信息返回给 DMDiskManger。如果 DMAdmin 从其他计算机的磁盘组检查到数据库，它会意识到磁盘上的卷是异地的，如果你希望使用它时，你可以将它们导入目前的数据库中。当你改变动态磁盘的配置，DMDiskManger 通知 DMAdmin 这种变化，DMAdmin 将更新内存中的备份。当 DMAdmin 提交变化时，它将更新后的数据库传给 DMIO，dmio.sys 设备驱动程序。DMIO 等价于动态的 FtDisk，因此它控制对磁盘上数据库的访问和建立表示动态磁盘上卷的设备对象。当你退出磁盘管理时，DMDiskManger 停止执行并卸载 DMAdmin 服务。

DMIO 不知道如何解释它监视的数据库。DMAdmin 加载的动态连接库 DMConfig（在 winNT \ system32 \ dmconfig.dll 下）和另一个设备驱动程序 DMBoot（DMBoot.sys）负责解释数据库。DMConfig 知道如何读和更新数据库；DMBoot 仅知道如何读数据库。如果另一个 LDM 驱动程序 DMLoad（Dmload.sys）判断到目前在系统中至少有一个动态磁盘时，在引导过程 DMBoot 将被加载。DMLoad 通过询问 DMIO 做决定，如果至少存在一个动态磁盘，DMLoad 启动 DMBoot，DMBoot

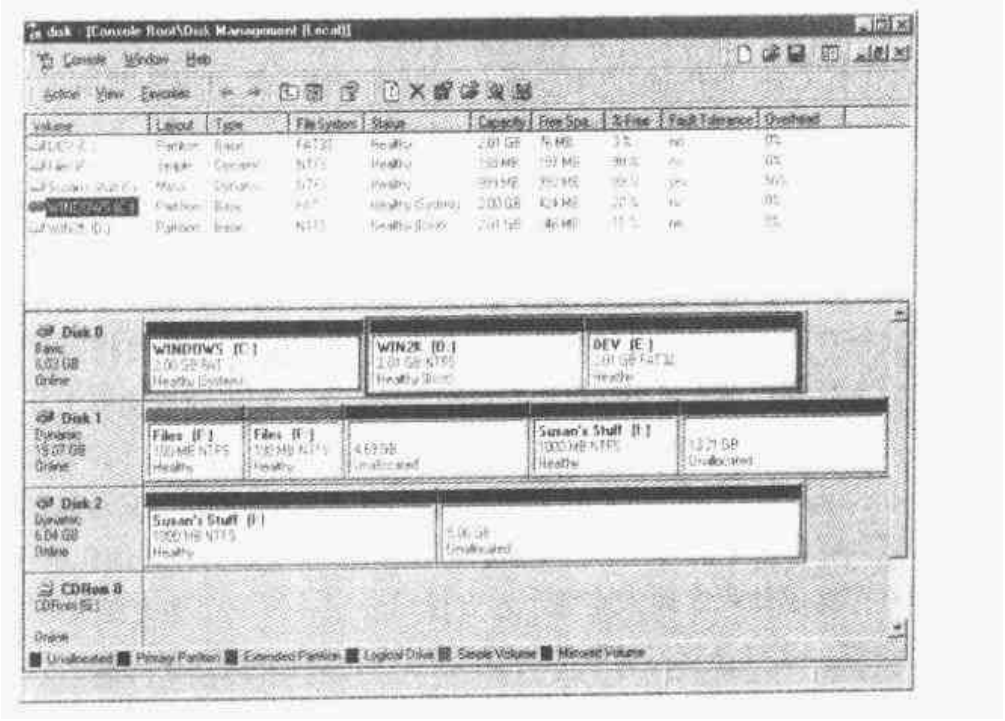


图 10-6 Disk Management MMC 插件

扫描 LDM 数据库，并通知 DMIO 它所遇到的每个卷的组成，这样 DMIO 可以创建表示卷的设备对象。一旦结束扫描，DMBoot 立即从内存中卸载，因为 DMIO 不包含任何的数据库解释逻辑，所以相对来说 DMBoot 较小。因为 DMIO 总是被加载，文件小是有利的。

像 FtDisk 一样，DMIO 是一个总线驱动程序，采用 \ device \ hardDiskDmVolumes \ physicalDmVolumes \ BlockVolumeX 的命名方式为每个动态卷创建一个设备对象，其中 X 是 DMIO 赋予卷的标识符。另外，DMIO 创建另一个设备对象表示卷中的原始 I/O（raw I/O），设备对象的名字为 \ device \ harddiskdmvolumes \ physicalDmVolumes \ RawVolumeX。图 10-7 显示了由 DMIO 创建的包含两个动态磁盘卷的系统中设备对象。DMIO 也在对象管理器名字空间创建许多符号链接，每

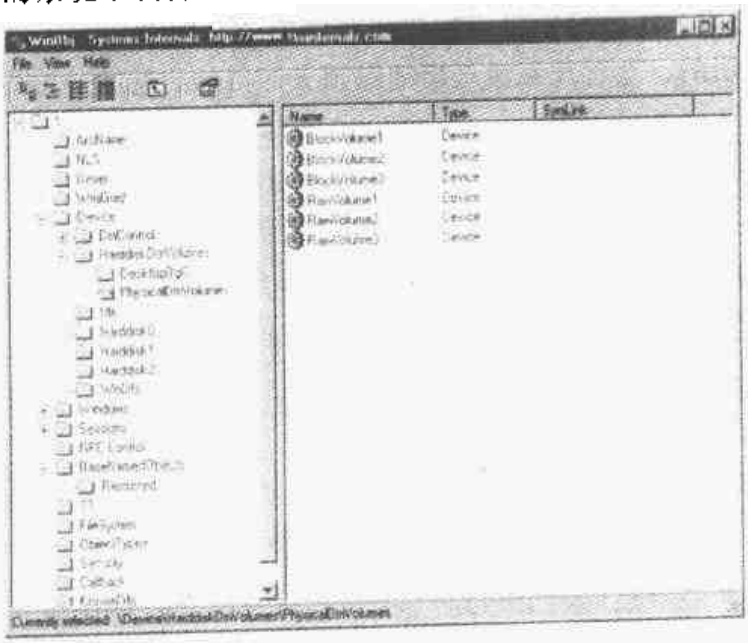


图 10-7 DMIO 驱动程序设备对象

个卷的第一个链接采用 \ Device \ HarddiskDmVolumes \ ComputerNameDg0 \ VolumesY 的形式，DMIO 用计算机名代替 ComputerName，用卷标（不同于 DMIO 赋予设备对象的内部标识符）代替 Y。在 PhysicalDmVolumes 目录下这些连接指向块设备对象。

10.3.5 磁盘性能监视

Windows 2000 的 I/O 体系结构允许设备对象的动态分层，第 9 章已经详细介绍。一个设备驱动程序可以创建设备对象，并将它附接到目标设备对象。I/O 管理器路由指向目标设备对象的请求到对象附接的设备对象。设备驱动程序使用这种机制监视、扩展和改变属于其他设备驱动程序的设备对象的行为。依靠分层的驱动程序是过滤器驱动程序，当一个过滤器驱动程序接收到一个目标设备的 I/O 请求包（IRP）时，过滤驱动程序对请求有充分的控制。它可以使请求失败，创建一个新的子请求或则将请求不加修改的传递给目标设备。（第 9 章已经详细介绍过 IRP）。

Windows 2000 存储驱动程序通常在两个地方使用分层。第一个地方是文件系统过滤器驱动程序。在更高的层次，文件系统过滤器驱动程序同文件系统驱动程序创建的表示装配的卷的目标设备对象附接。文件系统过滤器驱动程序拦截对装配的卷的请求，这样驱动程序可以实现自己的功能如监视、加密或在线病毒检查。文件监视器（在配套光盘的 \ Sysint \ Filemon.exe）是一个文件系统过滤驱动器程序的一个例子。

存储驱动程序通常使用分层的第二个地方是实现监视。Windows 2000 包含了一个名叫 DiskPerf 的分层存储驱动程序（磁盘性能驱动程序：\ Winnt \ System32 \ Drivers \ Diskperf.sys）。DiskPerf 附接代表物理磁盘的设备对象。（例如，\ Device \ Harddisk0 \ Partion0）这样它可以为性能工具提供性能相关的统计数据。统计数据包括每分钟读/写的字节数以及磁盘输入和输出的总时间。

10.4 多分区卷管理

FtDisk 和 DMIO 负责表示文件系统驱动程序管理的卷和将卷的 I/O 映射到低层。对于简单的卷，这个过程是简明的，卷管理器通过增加卷相关的偏移到卷的开始磁盘偏移来确保相关卷的偏移被转换到同相关的磁盘偏移。

多分区卷的复杂性是由于组成卷的分区可以位于不连续的分区甚至可以在不同的磁盘上。一些类型的多分区卷使用数据冗余的方式，因此要求更多的卷到磁盘偏移的转换。所以，FtDisk 和 DMIO 必须通过决定 I/O 最终影响的分区来处理所有多分区卷的 I/O 请求。

以下是 Windows 2000 可以使用的多分区卷的类型：

- 跨越卷
- 镜像卷
- 带区卷
- RAID－5 卷

在描述了多分区卷的分区配置和每一种类型的多分区卷的操作后，我们将讨论 FtDisk 和 DMIO 驱动程序对文件系统驱动程序发送给多分区卷的 IRP 的处理。因为 FtDisk 和 DMIO 支持同样的多分区卷类型，术语“卷管理器”可以表示 FtDisk 和 DMIO。

10.4.1 跨越卷

跨越卷是由在一个或多个磁盘上最大有 32 个自由分区构成的逻辑卷。Windows 2000 磁盘管理 MMC 插件可以将分区组合成一个跨越卷，跨越卷可以被格式化为任何 Windows 2000 文件系统支持的类型。图 10-8 显示了驱动符为 D 的一个 100Mb 的跨越卷：在第一个磁盘的倒数第三个扇区和第二个磁盘的第三个扇区之间创建。在 Windows NT 4 中跨越卷又叫卷集。

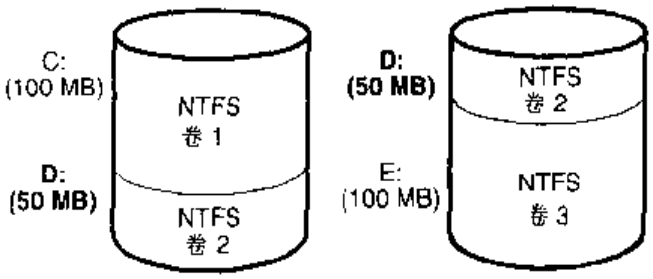


图 10-8 跨越卷

当将小的自由磁盘空间收集为一个更大的卷时，或从两个或多个小的磁盘创建一个单一的大的卷时，跨越卷十分的有用。如果跨越卷已经被格式化成 NTFS 的格式，它可以被扩展为包含更多的自由空间或额外磁盘空间的卷，而不影响已经存储在卷上的数据。这种扩展能力的最大好处之一是可以将整个在 NTFS 上的所有数据看作一个文件。NTFS 可以动态的增加逻辑卷的大小，因为用来记录卷的分配状态的位图是另一种文件，即位图文件。位图文件可以扩展包括为卷增加的任何的空间。另一方面，动态地扩充一个 FAT 卷需要 FAT 自己可以是可扩展的，这会使磁盘上的其他数据变得混乱。

卷管理器隐藏了安装在 Windows 2000 上文件系统的实际的磁盘物理配置。例如，将图 10-8 中的卷 D 看做一个普通的 100M 卷，NTFS 通过计算它的位图决定卷中可以用来分配的自由空间。然后，NTFS 调用卷管理器在卷上一个特殊字节偏移处读或写数据。卷管理器从第一个磁盘上的第一个自由扇区到最后一个磁盘的最后一个自由扇区对跨越卷上的物理扇区进行连续编号。它在给定字节偏移量的情况下确定哪个物理扇区在哪个磁盘上。

10.4.2 带区卷

带区卷最多由 32 个分区构成，每个磁盘一个分区，它们组成了一个逻辑卷。带区卷也可以叫作“RAID－0 卷”，图 10-9 显示了一个由三个分区组成，每个分区在一个磁盘上的带区卷

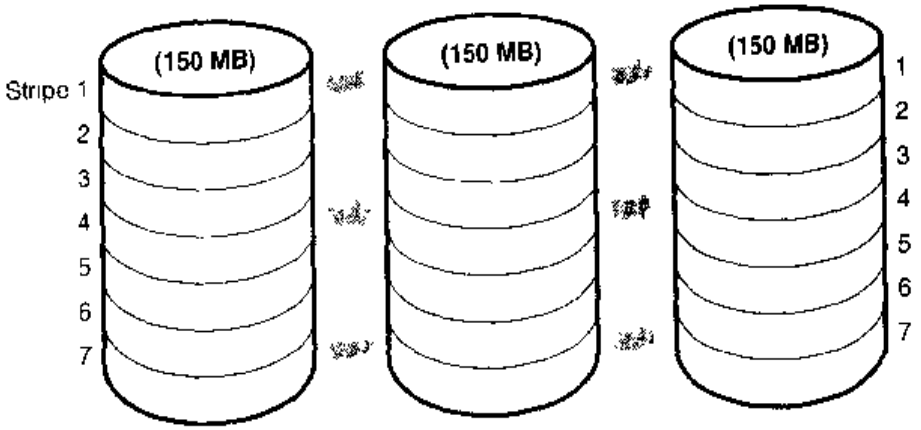


图 10-9 带区卷

(在带区卷上的分区不必要跨越整个磁盘，唯一的限制是在每个磁盘上的分区大小必须是一样大小的)。

对一个文件系统而言，带区卷看上去类似一个450MB的卷。但是卷管理器将通过在物理磁盘上分布卷数据优化数据的存储和获取时间。卷管理器访问磁盘上的物理扇区就好像它们是按顺序编号的，如图10-10所示：

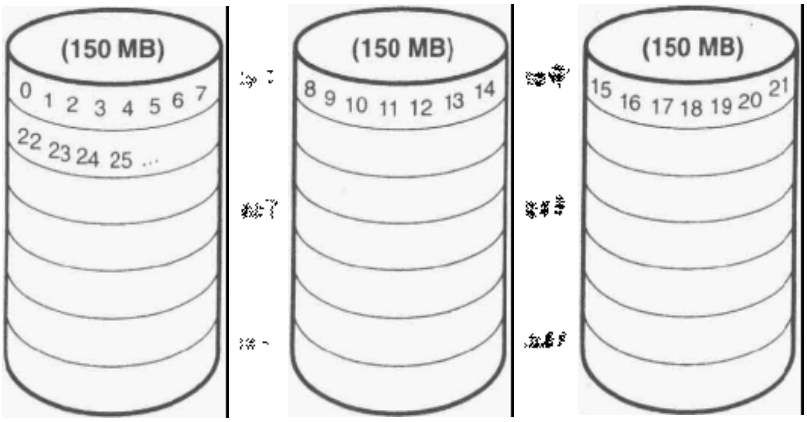


图10-10 带区卷上的物理扇区的逻辑号

因为每个带区被限制在64KB（一个选择用来防止对两个磁盘单独读写的值），数据在磁盘上的分布趋向于均匀的。带区增加了把多个悬而未决的读写操作限制在不同磁盘上的可能性。同时因为所有二个磁盘上的数据可以被并发的访问，对I/O的磁盘访问延迟时间可以减少，尤其在负载大系统上。

带区卷使得管理卷更加的方便，同时将输出输入负载分配到多个磁盘上，但是如果磁盘出现故障时，这两个卷管理性能不提供恢复数据的能力。然而，对于数据恢复卷管理器实现三种冗余存储模式：镜像卷，RAID－5卷和保留扇区（保留扇区和NTFS对它的支持的描述见第12章）这些性能可以通过Windows 2000的磁盘管理工具实现）

10.4.3 镜像卷

在镜像卷中，磁盘上分区的内容被在另一磁盘相同大小的分区上做了拷贝，镜像卷有时指RAID级别1，（PAID－1），图10-11显示了一个镜像卷。

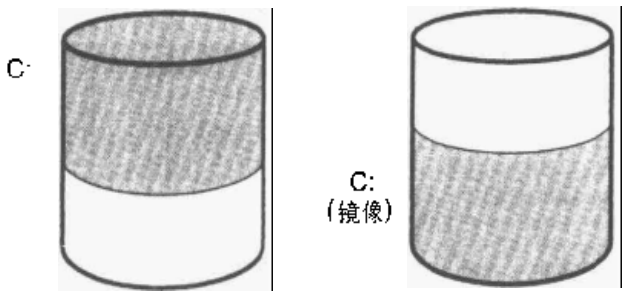


图10-11 镜像卷

当一个程序往驱动器C上写时，卷管理器将把数据写到镜像分区中相同的位置。如果第一个磁盘或C分区上的任何数据因为硬件或软件的原因而变得不可读的时候，卷管理器自动从镜像卷中访问数据。镜像卷可以格式化为Windows 2000支持的文件系统。文件系统驱动程序保持独立并不受卷管理器镜像活动的影响。

镜像卷可以在负载大的系统中有助于I/O的吞吐量，当I/O活动十分频繁的时候，卷管理器可以在主分区和镜像分区（由于大量的对每个磁盘连续不断的I/O操作）间平衡读操作。两个读操作可以同时进行，理论上可以在一半的时间完成。当一个文件被修改，两个镜像集的分区必须重写，但是磁盘写操作是异步进行的，这样用户的程序不会因为磁盘更新而受到影响。

镜像卷是被系统卷和引导卷支持的唯一多分区卷类型。原因是Windows 2000的引导代码包括MBR代码和NTldr都不具备对多分区卷复杂的理解的能力，镜像卷是个例外，因为引导代码将它们看做简单卷，从标记为引导或MS-DOS风格分区的系统驱动读代码。可以忽略另一半的镜像卷是因为引导不要求修改磁盘。

实验：观察镜像卷的I/O操作

使用Windows 2000的Performance Tool，你可以验证在写操作时，镜像卷将在两个磁盘上做拷贝操作以及读操作，如果相对不频繁时，操作只发生在卷的一半。这种实验要求三个硬盘和Windows 2000 Server、Windows 2000 Advanced Server或Windows 2000D Datacenter Server。如果你没有三个磁盘或一个服务器系统，你可以跳过试验安装指导，直接观看本次试验的性能工具屏幕上演示的结果。

使用磁盘管理MMC插件建立一个镜像卷。为了完成这一步，你可以执行以下步骤：

1）通过启动Computer Management运行Disk Management，展开Storage树，选择Disk Management（或在MMC控制台插入Disk Management）

2）右击驱动器没有分配的空间，选择Create Volume。

3）遵循Create Volume向导的指示创建一个简单卷。（确保你在另一个磁盘上有同你创建的卷一样大小的足够的空间）

4）右击新卷，从菜单中选择Add Mirror。

一旦你有了一个镜像卷，运行Performance Tool和为每个物理磁盘性能对象（即属于镜像的分区的磁盘实例）增加计数器，为每个实例选择disk reads/sec和disk writes/sec计数器，从第三个磁盘（不是镜像卷的磁盘）选择一个大的目录，拷贝其中的内容到镜像卷中，性能工具输出窗口看上去像下面的拷贝操作过程。

在整个时间区域相互重叠的上面两行是每一个磁盘的Disk Writes/Sec数，底下两行是Disk Reads/Sec行。图10-12显示了卷管理器（在这种情况下DMIO）正在将拷贝的文件数据写到卷的两个部分，但主要从一个部分中读。这种读行为的出现是因为在拷贝中的突出的I/O操作数并不能保证卷管理器执行更为积极的读操作负载平衡。

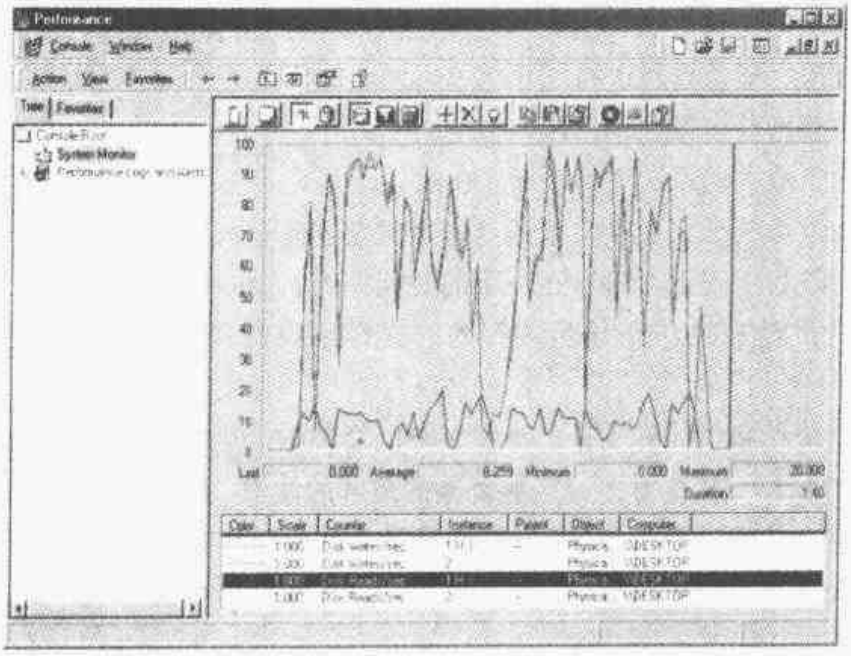


图 10-12 观察镜像卷的 I/O 操作

10.4.4 RAID－5 卷

RAID－5 卷是一个常规带区卷的容错变种。RAID－5 卷实现了 RAID 的第 5 层。它们也可以叫做带有奇偶检验的带区卷。因为它们建立在带区卷所使用的带区方法上。为每一个带区存储奇偶从而保留一个磁盘的等价物可以实现容错。图 10-13 是一个 RAID－5 卷的可视化表示。

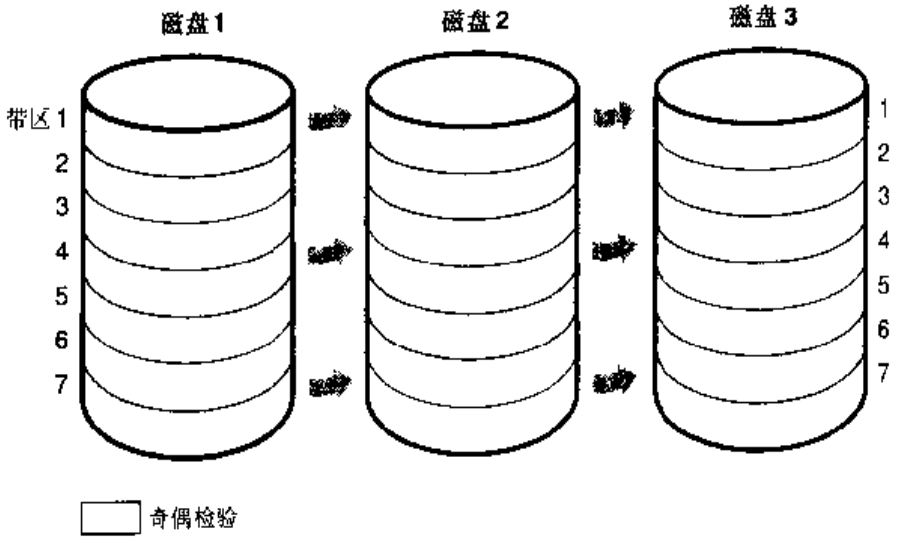


图 10-13 RAID－5 卷

图 10-13，带区 1 的奇偶检验被存储在磁盘 1 上。它包含有在第二个和第三个磁盘上第一个带区的字节和字节的逻辑异或。带区 2 的奇偶检验被存储在磁盘 2 上，带区 3 的奇偶检验被存储在磁盘 3 上。以这种方法在磁盘上旋转奇偶检验是一个 I/O 优化技术。每次数据被写到磁

盘上时，同修改字节相对应的奇偶检验要重新计算和重写。如果奇偶检验总是写到相同的磁盘上，那个磁盘将会不停的繁忙，变成I/O的瓶颈。

用RAID-5卷恢复故障磁盘依靠一个简单的数学原理：在一个n个变量的方程式中，如果你知道n-1个变量的值，你可以通过减法得到最后一个变量的值。例如，在方程式x+y=z，这里z表示奇偶带区，卷管理器计算z-y以判断x的值，为了找到y的值，计算z-x的值。卷管理器使用相似的原理恢复丢失的数据。如果在RAID-5卷中的磁盘出现故障或在一个磁盘上的数据不可读，卷管理器可以使用异或操作重新构造丢失的数据。

假定图10-13中的磁盘1发生故障，它的带区2和5上的内容可以使用相应的磁盘3上的带区与磁盘2上的带区做异或操作。磁盘1上带区3和6上的内容同样可以使用相似的方法恢复。至少三个磁盘（或在三个磁盘上三个相同大小分区）需要建立RAID-5卷。

10.4.5 卷的I/O操作

文件系统驱动程序管理存储在卷上的数据，但它依靠卷管理器与存储驱动程序交互从卷所驻留的磁盘上传输数据。通过装配过程，文件系统驱动程序获得一个卷管理器的卷对象指针（在本章后面部分再介绍），然后通过卷对象向卷管理器发送请求。当应用程序想直接操纵卷数据的时候，也可以绕过文件系统驱动程序向卷管理器发送请求。File-undelete程序是采用这种方式的一个很好例子，同样在Windows 2000的资源工具箱中也包含一个类似的叫DiskProbe的工具。

当一个文件系统驱动程序或应用程序向代表一个卷的设备对象发送I/O请求时，Windows 2000的I/O管理器向创建目标设备对象的卷管理器路由这些请求（请求来自一个IRP，一个自包含的包）。这样如果一个应用程序想读系统上第二个卷（在本例子中是个简单卷）的引导扇区，它打开设备对象\Device\HarddiskVolume2，然后向对象发送一个请求从设备上读偏移从0开始的512个字节。I/O管理器采用IRP的形式将请求发送给拥有设备对象的卷管理器，通知它IRP的目标是HarddiskVolume2设备。

因为Windows 2000为了逻辑上的便利，使用分区来表示硬盘上连续空间。卷管理器必须可以转换分区的偏移到磁盘相关的偏移。如果分区在磁盘上有3449个扇区，卷管理器在将请求传递给磁盘类驱动程序前，将调整IRP的参数并用一个合适的值表示偏移。磁盘类驱动程序使用小型端口驱动程序完成物理磁盘I/O操作，读请求的数据到IRP中指定的应用程序缓存区中。

一些卷管理器操作的例子可以帮组分清在处理多分区卷请求时卷管理器的角色。如果一个带区卷包含两个分区，分区1和分区2，用设备对象\Device\HarddiskDmVolumes\PhisicalDmVolumes\BlockVolumes3表示，如图10-14所示，I/O管理器定义了\??\D:指向\Device\HarddiskDmVolumes\PhisicalDmVolumes\BlockVolumes3，在这里computerName是计算机的名字，回想早些时候所说的连接可以是个符号链接，它指向相应的PhisicalDmVolumes目录（这里是Volumes3）中的卷设备对象。DMIO设备对象拦截发给\Device\HarddiskDmVolumes\PhisicalDmVolumes\BlockVolumes3的请求，同时在传递给磁盘类驱动程序前调整请求，DMIO对请求包的调整是保证分区1或分区2上的请求的目标带区偏移量正确。如果一个I/O跨越了卷的两个分区，DMIO必须发送两个辅助的I/O请求，每个磁盘一个。

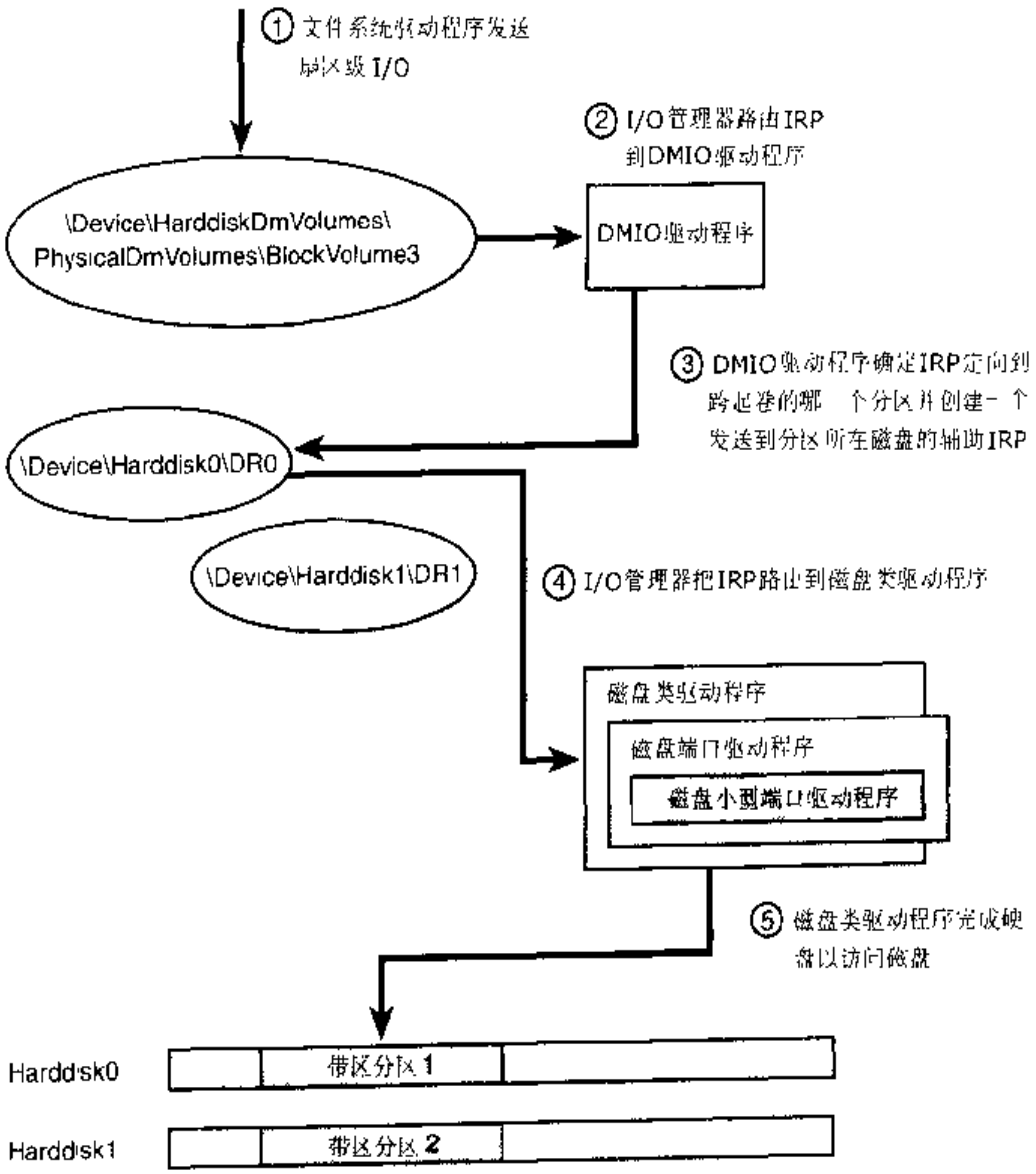


图 10-14 DMIO I/O 操作

当向一个镜像卷写时，DMIO 将分离每个请求，这样镜像每一半都可以收到写操作。对镜像的读操作，当对镜像的另一半读操作失败时，DMIO 将对镜像的另一半执行读操作。

10.5 卷名字空间

驱动器符分配是存储管理从 Windows NT 4 到 Windows 2000 重要转变的一个方面。即使如此，Windows 2000 包括对升级到 Windows 2000 的 Windows NT 4 安装程序转换磁盘驱动符的支持，Windows NT 4 的驱动符的赋值被存储在键 HKLM \ SYSTEM \ Disk 中，当更新过程读和写存储在 Windows 2000 位置上的信息时，系统不再引用 Disk 键。

10.5.1 装配管理器

Windows 2000 中的新的驱动器，装配管理器（Mountmgr.sys）为动态磁盘卷和在 Windows 2000 安装后的创建基本磁盘卷赋予磁盘符。Windows 2000 在 \ HKLM \ SYSTEM \ MountedDevices

下存储所有已经分配的磁盘驱动符。如果你察看注册表中的键内容，你会看见诸如 \ \?? \ Volume{x}(这里 x 是一个 GUID)和像?? \ C: 样的键值。每个卷有一个卷名，但是每个卷没有必要都有一个赋予的驱动符。图 10-15 显示了一个装配管理器注册表键内容的例子。注意到安装设备的键，像 Windows NT 4 的 Disk 键，没有被包含在控制集中，因此它不被所知最近的正确引导选项支持（参见 5.2.6 节的“接受引导和所知最近的正确”配置节有关控制集和最新引导选项的介绍）。

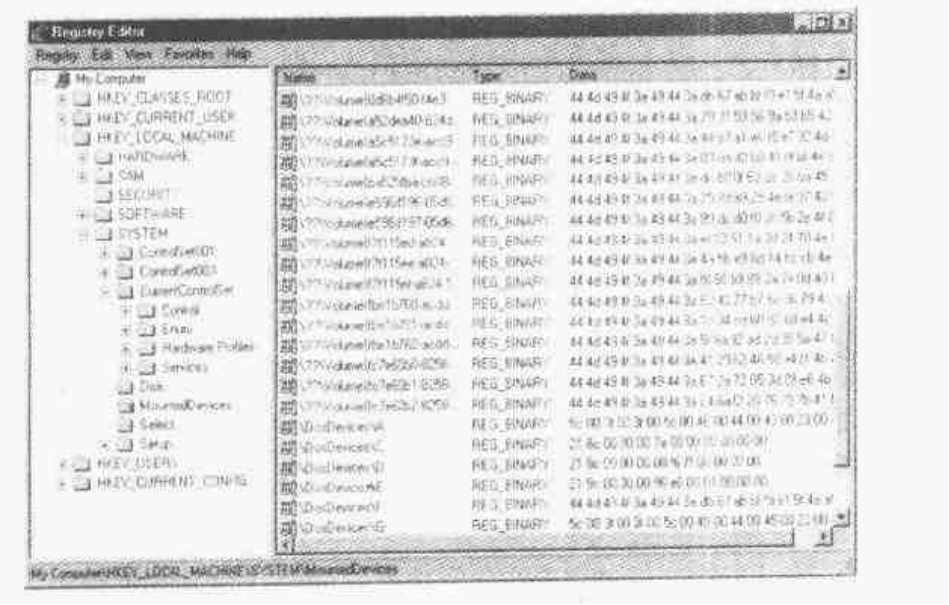


图 10-15 列在装配管理器的注册表键中的装配的设备

注册表用来存储基本磁盘卷驱动符和卷名的值是用 Windows NT 4 风格的磁盘签名，并开始于相关的卷的第一个分区。存储在注册表的动态磁盘卷的值包括卷的 DMIO 内部 GUID。当装配管理器在引导进程初始化的时候，它用 Windows 2000 注册即插即用子系统，这样当 FtDisk 或 DMIO 创建新的卷的时候它可以接受通知。当装配管理器接到这样的通知后，它可以决定新的卷的 GUID 或者磁盘签名，这样可以询问 FtDisk 或 DMIO（任何创建的卷）建议的驱动符。FtDisk 不返回这种建议（通过查询 Windows NT 4 的 \ HKLM \ SYSTEM \ Disk 键，在从 Windows NT 4 升级过程中，它可以返回建议）。而 DMIO 在卷数据库项中查询磁盘符暗示。

如果没有为卷建议的驱动符，装配管理器在卷的 GUID 或签名的引导下在内部的反映注册表内容的数据库中查询，装配管理器然后决定是否它的内部数据库包含有分配的驱动符。如果不存在可分配的驱动符，装配管理器使用第一个可以分配的驱动符（如果存在的话），定义一个新的分配，创建一个分配连接（如 \ ?? \ D:），更新装载设备的注册表的键值。如果没有可以利用的驱动符，磁盘驱动符将不创建。同时，装配管理器新建一个卷符号链接（即如果卷以前没有 GUID，则用 \ ?? \ Voulme{X}描述一个新的卷 GUID。这里的 GUID 不同于 DMIO 内部使用的卷 GUID）。

10.5.2 装配点

装配点，是 Windows 2000 中的新机制，可以让你通过 NTFS 上的目录连接卷，使得不通过

驱动符就可以访问卷。例如，你已经命名为C:\Projects的NTFS目录可以装配到任意个包含有你项目目录和文件的NTFS或FAT的卷上，如果你的项目卷有一个你命名的\CurrentProject\Description.txt文件，你可以通过路径C:\Projects\CurrentProject\Description.txt访问文件，使装配点成为可能的是一种叫做“重分析点”（reparse point）的技术（在第2章会详细讨论重分析点技术）。

一个重分析点是有固定头数据的任意一块数据，其中头数据被Windows 2000用来同NTFS文件或目录相关。一个应用程序或系统定义了一个重分析点的格式和行为，包括用来表示属于应用程序或系统的重分析点的唯一重分析点标识的值，同时也描述了一个重分析点数据部分的大小和含义（重分析点甚至可以是16KB的大小）。重分析点用一个固定的段存储它们唯一的标签。任何实现重分析点的应用程序必须提供一个文件系统过滤器驱动程序监视相关的重分析点返回代码，用于NTFS卷上的文件操作。当它检查到返回代码时，驱动程序必须采取相应的行为。任何时候，当NTFS处理文件操作并遇到与重分析点相关的文件或目录时会返回一个重分析状态代码。

Windows 2000的NTFS文件系统驱动程序、I/O管理器和对象管理器都部分实现重分析点的功能，对象管理器通过使用I/O管理器同系统文件驱动程序交互初始化路径名解析操作。因此，对象管理器必须为I/O管理器返回的重分析状态代码重新执行操作。I/O管理器实现装配点和其他重分析点需要的路径名修改操作。最后NTFS文件系统驱动必须将重分析点数据和文件或路径相关起来和标识它们。你可以将I/O管理器看做微软定义的重分析点的重分析点文件系统过滤驱动程序。

重分析点应用程序的一个例子是分层存储管理系统（HSM），它使用重分析点表示管理员移到离线磁带存储设备的文件。当用户试图访问离线磁带设备文件时，HSM过滤器驱动程序检查NTFS返回的重分析状态代码，同用户模式服务通信以从离线存储获取文件，从文件删除重分析点。在服务获取文件以后重新执行文件操作。这就是Windows 2000远程存储服务（RSS）过滤驱动程序rsfilter.sys使用重分析点的方式。

如果I/O管理器从NTFS接受一个重分析代码，这些重分析代码不同Windows 2000内建的重分析代码相关，没有过滤驱动程序要求重分析点。I/O管理器返回一个错误代码给对象管理器，对象管理器返回“文件不能够被系统访问”错误给要求访问文件或目录的应用程序。

装配点是一个将卷名(\??\Volume{X})当作重分析点数据的重分析点。当你使用磁盘管理MMC插件指派或删除卷的路径分配时，你可以创建装配点。你同样可以使用内建的命令行工具Mountvol.exe（\Winnt\System32\Mountvol.exe）创建和显示装配点。

装配管理器维护在每个NTFS卷上的装配管理器远程数据库，在数据库中存放有为卷定义的装配点记录。数据库文件，$MountMgrRemoteDataBase驻留在NTFS引导目录中。当磁盘从一个系统移到另一个系统或在多引导环境中，也就是在多个Windows 2000安装程序引导时（因为装配管理远程数据库的存在）时，装配点发生移动。NTFS同样跟踪NTFS元文件\$Extend\$Reparse中的装配点。（NTFS不允许任何应用程序察看元文件）。NTFS将装配点存储在元文件中，当Win32应用程序如磁盘管理器需要装配点定义时，Windows 2000可以容易地枚举装配点。

实验：递归装配点

本试验使用 Filemon（可以在配套 CD 的 \ Sysint \ Filemon.exe 下找到）显示递归装配点引发的有趣行为。递归装配点是一个用来连接所在的相同卷的装配点。执行递归装配点上的递归目录列表产生文件访问踪迹，清晰的显示 NTFS 创建装配点的方式。

为了创建和观察装配点。执行下列操作：

1）打开 Command Promtp 或 Window 浏览窗口，在名为 \ Recuse 目录下建立一个目录。

2）使用磁盘管理 MMC 插件，右击卷，选择 Change Drive Letter And Path。

3）当 Add/Remove 对话框出现时，输入你创建的目录路径如 I: \ Recurse。

4）启动 Filemon，在驱动器菜单，除了你刚才创建的装配点，不选其他所有卷。

现在你准备产生一个递归装配点踪迹。打开 Command Prompt，执行命令 dir/s I: \ Recurse。如果你观察所有在 Filemon's 中后来文件操作引用到递归的文件访问，你会注意到命令行首先访问 I: \ recurse，然后是 I: \ recurse \ recurse，依次类推。递归越来越深。

应用程序首先在递归的每一层执行目录列表操作，但当它遇到一个装配卷时，就深入装配点试图执行另一个目录列表操作。NTFS 返回重分析状态代码，它告诉对象管理器返回上一层，然后继续。最后当它返回到根目录时候，应用程序程序检查它在装配点递归时找到的文件和目录。应用程序从来不接收重分析代码是因为对象管理器有目的的重复执行。对象管理器处理从 NTFS 执行目录查找接收到的重分析码。

Filemon 表示的是同它们本机 IRP 类型一样的请求类型。这样打开目录和文件作为 IRP _ MJ _ CREATE 请求。文件或目录关闭请求是 IRP _ MJ _ CLOSE，目录查询是和 Filemon 的 Other 栏中的 FileBothDirectoryInfo 一块的 IRP _ MJ _ DIRECTORY _ CONTROL 请求。

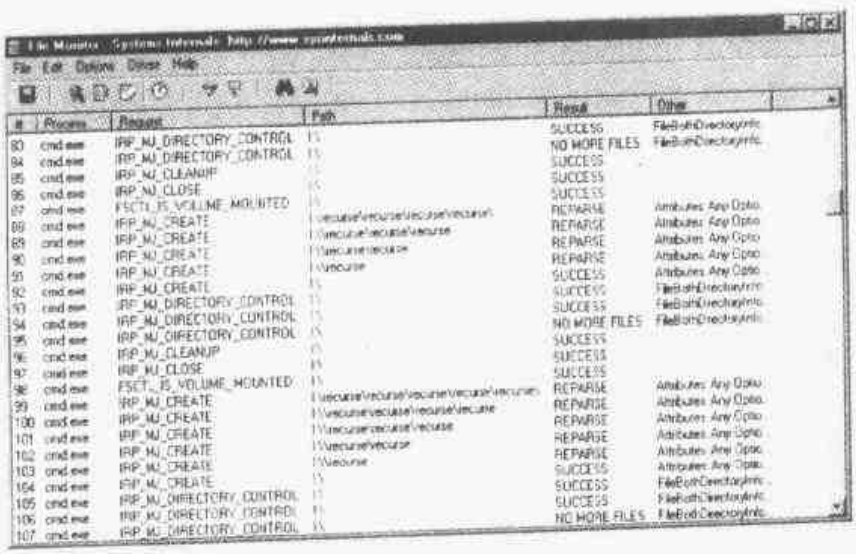


图 10-16 打开目录和文件

为了防止缓冲区溢出或无限循环，当目录深度达到 32 或者路径名超过 256 字符时，Command Prompt 和 Windows Explorer 停止它的递归。

10.5.3 卷装配

Windows 2000 给每个分区赋值并不表示分区包含有 Windows 2000 可以识别的文件系统格式。卷识别进程包括声明分区所有权的文件系统。这个进程首先在系统内核、设备驱动程序或应用程序访问分区上的文件或目录。文件系统驱动程序表明它对分区负责时，I/O 管理器将所有目标是分区的 IRP 导向拥有的驱动程序。Windows 2000 的装配操作包括三个组件：文件系统驱动程序注册、卷参数块（VPB）和装配请求。

I/O 管理器监视装配进程，并且知道可用的文件系统驱动程序，因为所有的文件系统在它们初始化的时候都会在 I/O 管理器注册。I/O 管理器提供函数 IoRegisterFileSystem 用于本机磁盘文件系统的注册。当一个文件系统驱动程序注册的时候，I/O 管理器在列表中存储指向驱动程序的引用

每个设备对象包含一个 VPB 数据结构，但是 I/O 管理器将 VPB 看作有意义的卷设备对象。一个 VPB 作为卷设备对象和文件系统驱动程序创建用来表示装配的文件系统实例的链接。如果一个 VPB 的文件系统引用是空的，将没有文件系统装配在卷上。I/O 管理器当设备对象执行具体的文件或目录打开 API 时，会检查卷设备对象的 VPB

例如，如果装配程序将驱动符 D 赋给系统的第二个卷时，它创建一个 \ ?? \ D: 符号链接映射到设备对象 \ Device \ HarddiskVolume2。试图打开驱动符 D 上 \ Temp \ Test.txt 的 Win32 应用程序在调用内核文件打开例程 NTCreateFile 前把它转换成 \ ?? \ D: \ Temp \ Test.txt 格式。NTCreateFile 使用对象管理器解析名字，对象管理器在路径 \ Temp \ Test.txt 没有映射的情况下遭遇 Device \ HarddiskVolumes 设备对象。在这里，I/O 管理器检查是否 \ Device \ HarddiskVolumes 的 VPB 引用一个文件系统。如果没有，I/O 管理器通过装配请求询问每个已经注册的文件驱动程序是否识别驱动符所拥有的格式。

实验：察看 VPB

你可以使用！vpb 核心调试命令察看 VPB 的内容。因为 VPB 被设备对象用来指向卷，你必须首先查找一个卷设备对象。为了做到这一点，你必须跳过卷管理驱动程序对象，查找代表卷的设备对象、显示设备对象，就可以显示 VPB 字段的内容。

如果你的系统有一个动态磁盘，你可以使用 DMIO 上的！drvojb 驱动程序对象察看命令，否则你需要定义 FtDisk 驱动程序。以下是个例子：

```
kd> !drvobj ftdisk
Driver object (818aec50) is for:
 \Driver\Ftdisk
Driver Extension List: (id , addr)

Device Object list:
818a5290  817e96f0  817e98b0  817e9030
818a73b0  818a7810  8182d030
```

！drvobj命令显示了驱动程序拥有的设备对象的地址，在这个例子里，有七个设备对象，其中一个代表了设备驱动程序的可编程接口，其他的则是卷设备对象。因为对象按照它们创建的逆序排列，设备接口对象总是第一位的，你可以看到第一个所列的设备对象是卷。现在在卷设备对象地址上执行！devobj内核调试命令。

```
kd> !devobj 818a5290
Device object (818a5290) is for:
 HarddiskVolume6 \Driver\Ftdisk DriverObject 818aec50
Current Irp 00000000 RefCount 3 Type 00000007 Flags 00001050
Vpb 818a5da8 DevExt 818a5348 DevObjExt 818a53f8 Dope 818a50a8
  DevNode 818a5ae8
ExtensionFlags (0xa0000000)
                              Unknown flags 0xa0000000
```

！devobj命令显示用于卷设备对象的VPB字段（显示的设备对象名叫HarddiskVolume6），现在你准备执行命令！vpb。

```
kd> !vpb 818a5da8
Vpb at 0x818a5da8
Flags: 0x1 mounted
DeviceObject: 0xB50dcac0
RealDevice:   0x818a5290
RefCount: 3
Volume Label:      GAMES
```

命令显示卷设备对象是由赋给卷名字GAMES的文件系统驱动程序装配的。VPB字段的RealDevice指回卷设备对象，DeviceObject字段指向装配的文件系统设备对象。

文件系统驱动程序识别装配卷的格式遵循的惯例是检查卷的引导记录，引导记录存储在卷的第一个扇区上。微软文件系统的引导记录包含一个存储文件系统格式类型的字段。文件系统驱动程序通常检查这个字段，如果显示是一个可以被管理的文件格式，它查询存储在引导记录中的其他信息。这些信息通常包括文件系统名字，足够的数据用于在文件系统驱动程序查找卷上的元数据文件，例如，NTFS将识别NTFS类型的卷当且仅当类型字段为“NTFS”，并且引导记录描述的元文件要一致。

如果文件系统驱动程序发出肯定的信号，I/O管理器填充VPB中的数据，然后传递带有剩下路径（即\Test）的打开请求给文件系统驱动程序。文件系统驱动使用文件系统格式解释存储在分区中的数据从而完成请求。当一个装配填充分区设备对象的VPB后，I/O管理器将以后的分区打开请求交给装配的文件系统设备驱动程序。如果没有文件驱动程序声明对分区的处理，一个叫Raw的Windows 2000内建的文件系统驱动程序将声明处理分区，并对所有对此分区的文件打开请求标为失败。图10-17显示了一个简单的例子，指向装配分区的I/O遵循的路径（图中省略了文件系统驱动程序同Windows 2000的缓存管理器间的交互）。

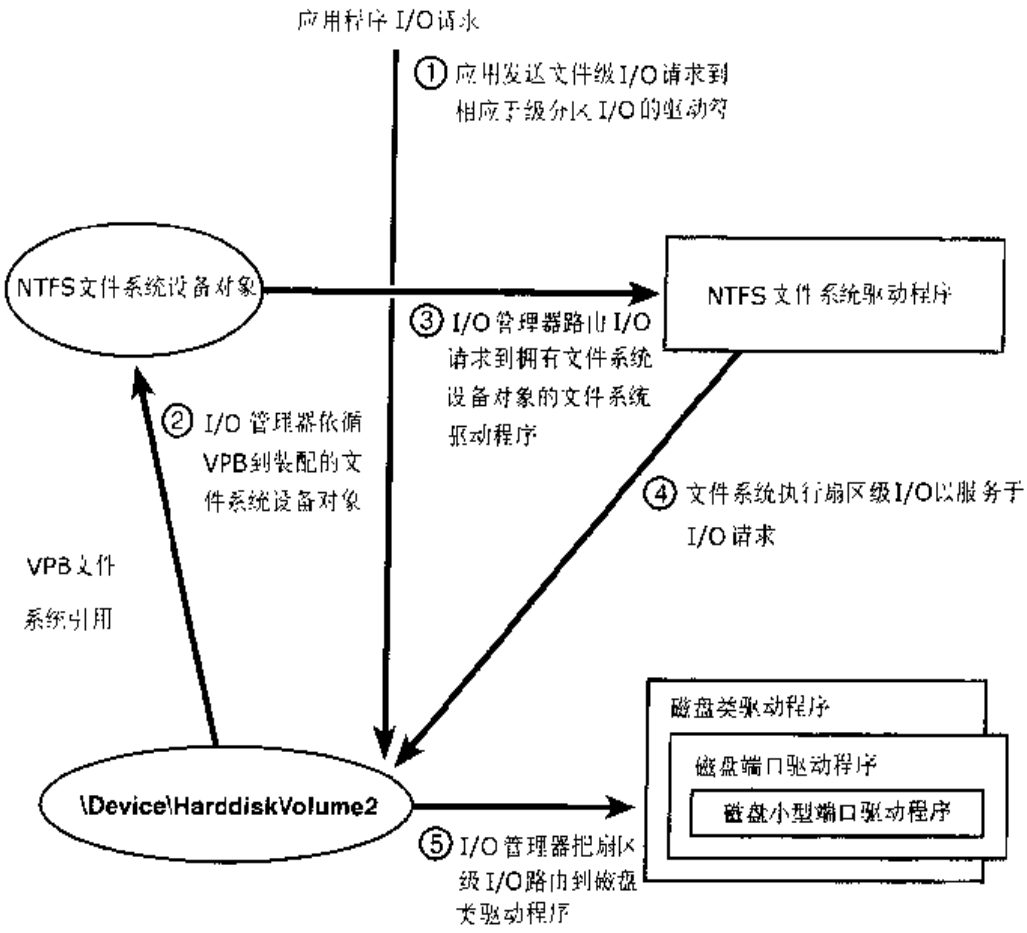


图 10-17 装配卷的 I/O 流程

代替加载每个文件系统驱动程序，Windows 2000 不考虑是否有没有可管理的卷，它尽量通过使用一个名字叫文件系统识别者（\ Winnt \ System32 \ Drivers \ Fs _ rec.sys 的代理驱动程序执行预备的文件系统识别操作从而有效的减少内存的使用率。文件系统识别者对每一个文件系统格式有足够的了解，这样 Windows 2000 可以检查一个引导记录并决定它是否同 Windows 2000 的文件驱动程序相关。当系统引导的时候，文件系统识别者可以当作一个文件系统驱动程序被加载，在 I/O 管理器在为一个新的卷执行文件系统装配操作时调用文件系统驱动程序。如果引导记录相应的文件系统驱动程序没有被加载，文件系统识别者将加载合适的文件系统启动模块：文件系统驱动程序加载完毕以后，文件系统识别者将建立的 IRP 传递给驱动程序，让文件系统驱动程序声明拥有这个卷。

除了引导卷在系统核心初始化时被装配，当 chkdsk 文件系统一致性检查程序在引导时运行时，文件系统会装配绝大多数的卷。引导时运行的 chkdsk 的程序是一个本机应用程序叫 autochk.exe（同 Win32 程序相比较），会话管理器（\ Winnt \ System32 \ Smss.exe）运行它，因为它在 \ HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ BootExecute 中被标识为引导时运行程序。Chkdsk 访问每个驱动器检查是否同驱动符相关的卷要执行一致性检查。

对于同一个磁盘执行多次装配过程的地方是可移动介质。Windows 2000 文件系统驱动程序通过查询磁盘的卷标识符号负责媒体的改变。如果它们发现卷标识符改变了，驱动程序将卸载磁盘并试图重新装配它。

10.6 小结

在本章，我们已经回顾了磁盘的组织结构、组件和 Windows 2000 执行磁盘存储管理所执行的操作。在下一章，我们将研究缓存管理器——装配卷类型的文件系统驱动程序操作必不可少的执行程序组件。

第 11 章　高速缓存管理器

Microsoft Windows 2000 高速缓存管理器是一组内核模式函数和系统线程，它们与内存管理器合作为所有 Windows 2000 文件系统驱动程序（本机的和网络的）提供数据高速缓存。本章将解释 Windows 2000 高速缓存管理器是如何工作的，其中包括关键内部数据结构和函数、在系统初始化时如何调整它的大小、如何与操作系统的其他组件相互影响以及如何通过性能计数器来观察它的活动，此外，还将描述影响文件高速缓存的 Win32 CreateFile 函数中的 5 个标志。

注意　所有超出解释高速缓存管理器如何工作的所需深度的内部函数都没有在本章列出。高速缓存管理器的编程接口包含在 Windows 2000 Instabllable File System（IFS）工具中。关于 IFS 工具的详细情况，请参阅 microsoft.com/ddk/ifskit/。

11.1　Windows 2000 高速缓存管理器的主要特性

Windows 2000 高速缓存管理器具有以下几个主要特性：

■ 支持所有文件系统类型（本机的和网络的）从而不需要为每个文件系统实现其自身的高速缓存管理代码。

■ 使用内存管理器控制哪些文件的哪些部分在物理内存中（在用户进程和操作系统之间权衡它们对物理内存的需求）。

■ 在一个虚拟块的基础上（文件内的偏移量）高速缓存数据——与在一个逻辑块的基础上（磁盘分区内的偏移量）高速缓存数据的大多数高速缓存系统相比，允许智能预读和高速访问高速缓存而不涉及文件系统驱动程序（这种高速缓存方法叫作“快速 I/O”，将在本章的稍后部分对它进行描述）。

■ 支持应用程序在打开文件时给出“暗示（hint）”（例如随机访问与顺序访问、临时文件的创建等等）。

■ 支持可恢复的文件系统（例如，那些使用事务处理日志的文件系统）以便在系统崩溃后恢复数据。

尽管将在这一章通篇讲述这些特性是如何在高速缓存管理器中使用的，但这一节将向你介绍在这些特性背后的有关概念。

11.1.1　单个、集中的系统高速缓存

有些操作系统依赖于各个独立的文件系统来高速缓存数据，这在实践中会导致在操作系统中重复高速缓存和内存管理代码，或者限制可以被高速缓存的数据种类。相比之下，Windows 2000 提供了一种集中的高速缓存措施来高速缓存所有在外部存储的数据，而不管这些数据是在本机硬盘、软盘、网络文件服务器、还是在 CD－ROM 上。所有数据都可以被高速缓存，无

论它是用户数据流（文件的内容以及对该文件正进行的读和写操作）或是文件系统的“元数据”（metadata）（例如目录和文件头）。正如你将在本章中所看到的那样，Windows 2000 访问高速缓存的方法取决于被高速缓存的数据类型

11.1.2　内存管理器

Windows 2000 高速缓存管理器的一个不同寻常的方面是它永远不知道物理内存中实际上有多少被高速缓存的数据。这听起来可能有些奇怪，因为高速缓存的目的就是将一个频繁访问的数据子集保存在物理内存中，以此作为提高 I/O 性能的一种方法。Windows 2000 高速缓存管理器不知道有多少数据在物理内存中的原因是它使用了标准的“区域对象”（section objects）（在 Win32 在术语中叫作“文件映射对象”（file mapping objects），通过把文件视图映射到系统虚拟地址空间来访问数据（区域对象是内存管理器的基本单元，第 7 章对此有详细介绍）。当这些被映射的视图中的地址被访问时，内存管理器将不在物理内存中的块页面调入，而当需要内存时，内存管理器将数据页面调出并写回到在高速缓存中打开（被映射入高速缓存中）的文件里。

通过使用映射文件在虚拟地址空间的基础上进行高速缓存，高速缓存管理器避免生成读或写 I/O 请求包来访问正被高速缓存的文件的数据。相反，它只是简单地向（或从）映射高速缓存文件部分的虚拟地址复制数据，并根据需要依赖内存管理器使数据出错从而将数据调入或调出内存。该处理允许内存管理器在系统高速缓存和用户进程的内存数量之间作出全局权衡（高速缓存管理器还初始化 I/O，例如延迟写（lazy writing），这将在本章的稍后部分进行介绍；然而，它调用内存管理器来写页面）。正如你将在下一节要学到的那样，这种设计使得打开被高速缓存文件的进程与那些将同一文件映射到用户地址空间的进程一样看到相同的数据

11.1.3　高速缓存的一致性

高速缓存管理器的一个重要功能是确保所有访问高速缓存数据的进程都得到那些数据的最新版本。当一个进程打开一个文件（该文件因此而被高速缓存）而另一个进程直接将该文件映射到它的地址空间（使用 Win32 MapViewOffile 函数）时就将出现问题。这个潜在问题在 Windows 2000 中不会发生，因为高速缓存管理器和将文件映射到其地址空间的用户应用程序都使用相同的内存管理文件映射服务。如图 11-1 所示，内存管理器保证对每个唯一的映射文件只有一个表示（不考虑区域对象或映射视图的数目），它将一个文件的所有视图（即使它们互相重叠）映射到物理内存的单个页面组中（关于内存管理器如何中使用映射文件的详细信息，请参阅第 7 章）。

因此，例如：如果进程 1（View 1）有一个文件视图映射到它的用户地址空间，进程 2 通过系统高速缓存访问该视图，那么进程 2 将会看到进程 1 对该视图所做的任何修改，而不必等到它们被刷新以后。内存管理器不会刷新所有用户映射的页面——只刷新那些已知已经被写入的页面（因为它们有修改设置位）。这样，在 Windows 2000 下任何访问文件进程总能得到该文件的最新版本，即使某些进程通过 I/O 系统打开该文件，而另一些进程使用 Win32 文件映射函数将该文件映射到它们地址空间。

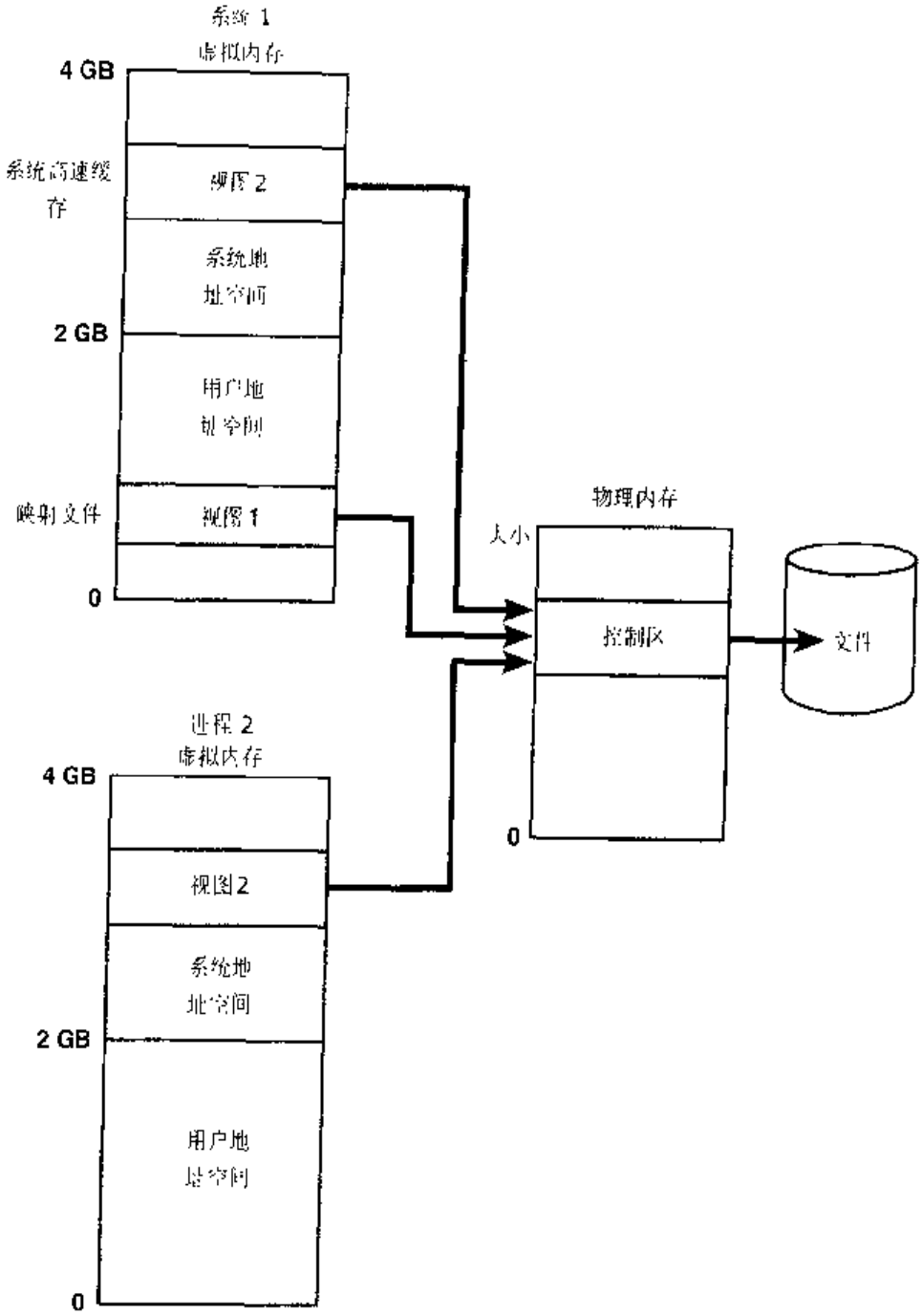


图 11-1 一致的高速缓存方案

注意 网络重定向器的高速缓存一致性比本机文件系统要稍微复杂些，因为访问网络数据时，网络重定向器必须实现额外的刷新与清除操作以确保高速缓存的一致性。关于机会主义的锁定描述，Windows 2000 高速缓存一致性机制，请参阅第 13 章。

11.1.4 虚拟块高速缓存

许多操作系统高速缓存管理器（包括 Novell NetWare 、OpenVMS、OS/2 以及大多数旧的 UNIX 操作系统）在逻辑块（logical block）的基础上高速缓存数据。采用这种方法，高速缓存管理器可以跟踪磁盘分区的哪些块在高速缓存中。相比之下，Windows 2000 高速缓管理使用一

种称为“虚拟块高速缓存”（virtual block caching）的方法，它可以跟踪哪些文件的哪些部分在高速缓存中。使用位于内存管理器中的特殊的系统高速缓存例程，高速缓存管理器能够通过将文件的 256KB 视图映射到系统虚拟地址空间来监视那些文件的部分。这一方法的主要优点如下：

■ 它为智能预读提供了可能性；因为高速缓存跟踪哪些文件的哪些部分在高速缓存中，因而可以预测调用程序下一步将访问的地方。

■ 对于已经在高速缓存中的数据的请求，它允许 I/O 系统绕过文件系统（快速 I/O）。因为高速缓存管理器知道哪些文件的哪些部分在高速缓存中，所以它可以返回被高速缓存的数据的地址来满足 I/O 请求，而不必调用文件系统。

关于智能预读和快速 I/O 工作的细节将会在本章的稍后部分介绍。

11.1.5　基于流的高速缓存

相对于文件高速缓存，Windows 2000 的高速缓存管理器也被设计用来进行“流高速缓存”（stream caching）。“流”（stream）是文件中的字节序列。有些文件系统，如 NTFS，允许文件包含多个流。高速缓存管理器通过单独地高速缓存每个流来适应这些文件系统。NTFS 通过将它的主文件表（将在第 12 章中介绍）组织成流并且同时高速缓存这些流来利用这一特性。事实上，虽然 Windows 2000 高速缓存管理器可能被称为高速缓存文件，但实际上却是高速缓存流（所有的文件都至少有一个数据流）。这些流通过文件名标识。如果一个文件中有在多个流，则通过流名称来标识。

11.1.6　可恢复文件系统支持

可恢复文件系统，例如 NTFS，被设计成在系统失败后能够重构磁盘卷结构。该功能意味着，系统失败时正在进行中的 I/O 操作必须全部完成，或者在系统重新启动时从磁盘完全退回。完成一半的 I/O 操作可能损坏磁盘卷，甚至导致整个卷都不可访问。为了避免出现这种问题，可恢复文件系统会维护一个日志文件，将更改写入卷之前，在该文件中记录着对文件系统结构（文件系统的元数据）将要做的每一个更新。如果因系统失败而中断了正在进行的卷修改，可恢复文件系统利用存贮在日志文件中的信息重新开始卷的更新。

注意　术语“元数据”只适用于文件系统结构的更改：文件和目录的创建、重命名以及删除。

为了保证成功的卷恢复，每个记录了卷更新的日志文件记录必须在卷更新之前完全写入磁盘。因为磁盘写是高速缓存的，所在高速缓存管理器和文件系统必须共同工作以确保下述操作按顺序执行：

1）文件系统写一个日志文件记录，记录将要进行的卷更新。

2）文件系统调用高速缓存管理器将日志文件记录刷新到磁盘上。

3）文件系统将卷更新写入高速缓存，也就是说，修改其高速缓存的元数据。

4）高速缓存管理器将改变的元数据刷新到磁盘上，更新卷结构。（实际上，与卷修改一样，日志文件记录在被刷的磁盘上之前批处理的。）

当文件系统将数据写入高速缓存时，它可以提供一个逻辑序列号（LSN）来标识日志文件中的记录。该逻辑序列号对应于高速缓存的更新。高速缓存管理器跟踪这些号码，记录高速缓存中

与每个页面相关的最低和最高的 LSN（代表最旧和最新的日志文件记录）。此外，由事务日志记录保护的数据流被 NTFS 标志为“不可写”，因此在相应的日志记录被写之前，修改页面的书写器不会意外地写这些页面（当修改页面书写器查看到一个页面被这种方法标志时，它便把该页面移到专用列表中，高速缓存管理器会在适当的时候，如当延迟书写器工作时，刷新该列表）。

当高速缓存管理器准备将一组脏页刷新到磁盘时，它将确定与要刷新页相关的最高的 LSN 并将该数字报告给文件系统。文件系统然后回调高速缓存管理器，指挥其将日志文件刷新到由所报告的 LSN 指定的位置。当高速缓存管理器将日志文件刷新到 LSN 指定的位置以后便将相应的卷结构更新刷新到磁盘，这样就确保了在实际执行操作之前记录了它将要进行的操作。文件系统和高速缓存管理器之间的这些操作保证了系统失败后磁盘卷的可修复性

11.2 高速缓存结构

因为 Windows 2000 系统高速缓存管理器在虚拟基础上高速缓存数据，所以它所管理的是系统虚拟地址空间的区域（而不是物理内存区域）。高速缓存管理器然后把每个地址空间区分成一系列 256KB 大小的槽，这些槽被称为“视图”，如图 11-2 所示（关于系统空间布局的详

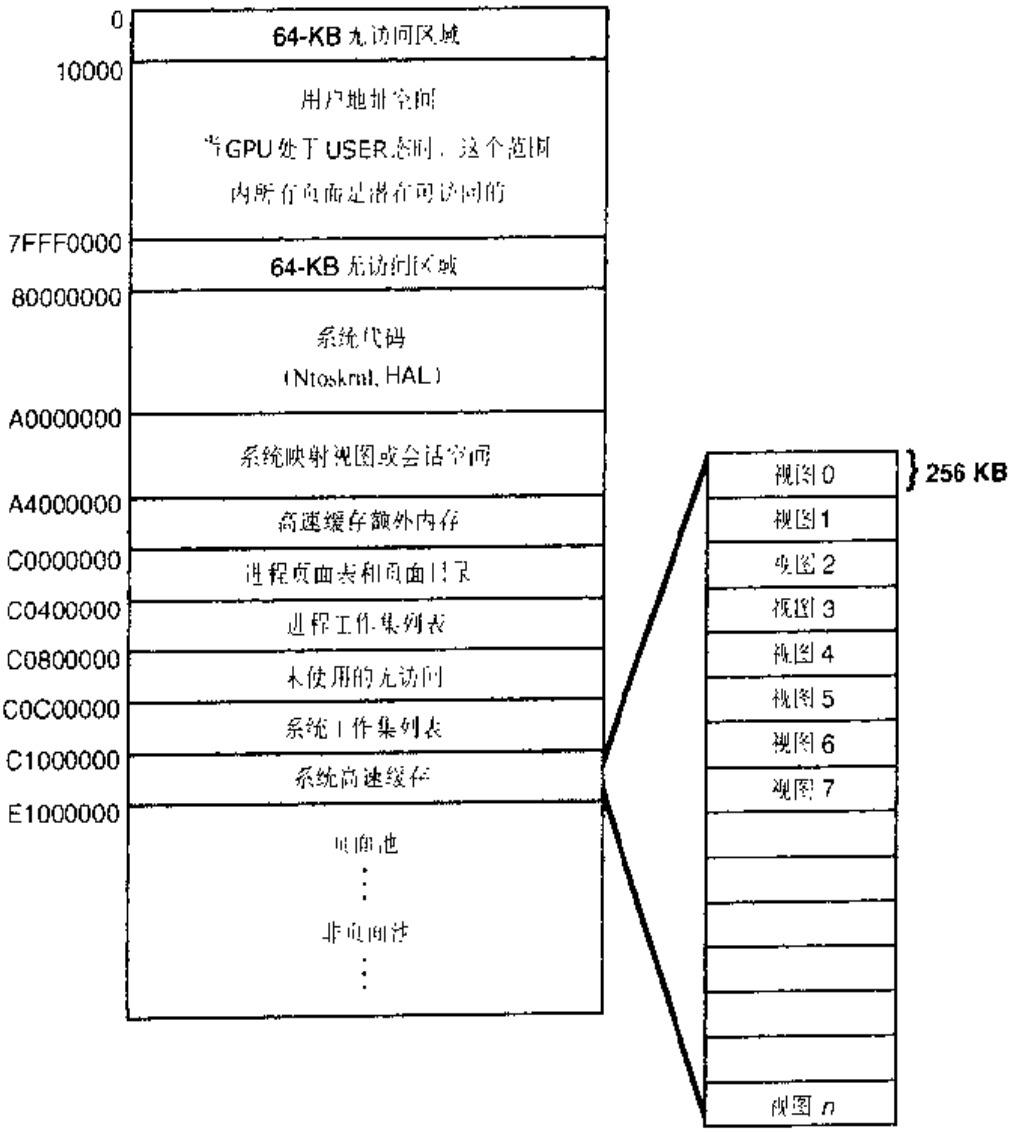


图 11-2 系统高速缓存地址空间

细描述，请参阅第 7 章）

在文件的首次 I/O（读或写）操作时，高速缓存管理器把包含有所请求的数据的文件的 256KB 对齐区中的一个 256KB 视图映射到系统高速缓存的空闲槽中。例如，如果把从偏移量 300，000 开始的 10 个字节读入一个文件，被映射的视图将从偏移量 262144（该文件的第二个 256KB 区域）开始并延伸 256KB。

高速缓存管理器在循环的基础上（round－robin basis）将文件视图映射到高速缓存的槽中，将所请求的第一个视图映射到第一个 256KB 的槽中，将第二个视图映射到第二个 256 KB 的槽中，依此类推，如图 11-3 所示。在该例子中，文件 B 被首先映射，文件 A 第二，文件 C 第三，因此文件 B 的映射块占用高速缓存的第一个槽。注意只有文件 B 的第一个 256KB 部分被映射，这是由于只有该文件的该部分被访问。另外虽然文件 C 只有 100KB（小于系统高速缓存中的视图），但它也需要在高速缓存中占用一个 256KB 的槽。

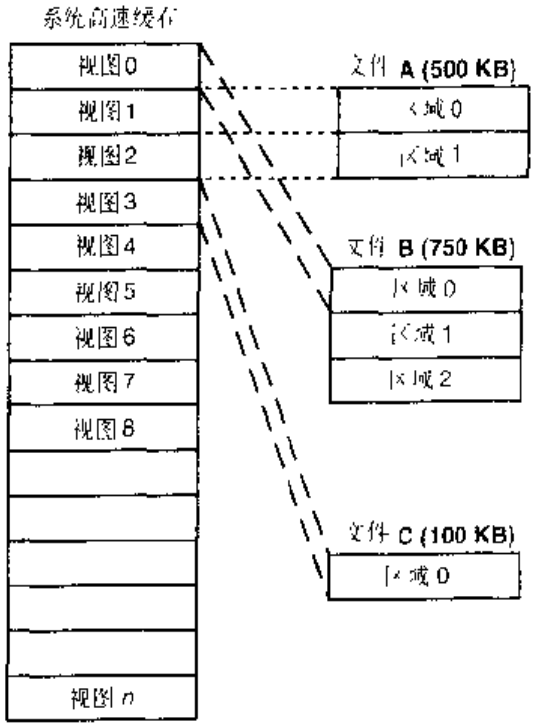


图 11-3　被映射到系统高速缓存的不同大小的文件

高速缓存管理器保证视图只要活动就被映射。然而，一个视图只有在从文件读或写期间才被标记为活动。除非进程在调用 CreateFile 打开文件时指定了 FILE_FLAG_RANDOM_ACCESS，否则高速缓存管理器在映射新的文件视图时，将不会映射不活动的文件映射。未映射的页面被发送到备份或修改的链表中（依赖于它们是否被修改）。因为内存管理器向高速缓存管理器输出一个专用接口，因此高速缓存管理器能够指引页面被放置到这些链表的头或尾。对应于用 FILE_FLAG_SEQUENTIAL_SCAN 标志打开的文件视图的页面被移到链表的前端，而其他则被移到末端。这种方案鼓励重用属于顺序访问文件的页面并防止大的文件复制操作影响到小部分的物理内存。

如果高速缓存管理器需要映射一个文件视图，并且在高速缓存中没有空闲的槽时，它就会取消最近非活动的映射视图的映射并使用该槽。如果没有可用的视图，将返回 I/O 错误，指出没有足够的可用系统资源来执行该操作。假定视图只有在读/写操作期间才被标志为活动，然而，这种情况极不可能发生，因为要发生这种情况就要有成千上万的文件被同时访问。

11.3　高速缓存的大小

以下将解释 Windows 2000 如何计算系统高速缓存（虚拟高速缓存和物理高速缓存）的大小。因为大多数的计算都涉及内存管理，系统高速缓存的大小将取决于多种因素，包括内存的大小和正在运行的 Windows 2000 的版本。

11.3.1　高速缓存的虚拟大小

系统高速缓存的虚拟大小是已经安装的物理内存的函数。缺省的大小是 64MB。如果系统有多于 4032 个（16MB）物理内存页面，高速缓存的大小被设置为 128MB，并且每增加 4MB 物理内存，高速缓存的大小则增加 64MB。使用该算法，一个具有 64MB 的物理内存的计算机的系统高速缓存的虚拟大小为：

128MB + (64MB - 16MB)/4MB × 64MB = 896MB

表 11-1 显示了系统高速缓存虚拟大小的最小值和最大值，以及起始和结束地址。如果系统计算的高速缓存的虚拟大小大于 512MB，那么将从一个称为高速缓存额外内存（cache extra memory）的地址空间中分配虚拟内存给高速缓存。

表 11-1　系统数据缓存的大小和位置

平　台	地址范围	最小/最大虚拟大小
x86 2GB 系统空间	0xC1000000 ~ E0FFFFFF， 0xA4000000 ~ BFFFFFFF	64 ~ 960MB
x86 1GB 系统空间	0xC1000000 ~ DBFFFFFF	64 ~ 432MB
x86 1GB 系统空间，带 Terminal Service	0xC1000000 ~ DCFFFFFF	64 ~ 448MB

表 11-2 列举了包含系统高速缓存的虚拟大小和位置的系统变量。

表 11-2　用于系统高速缓存虚拟大小和地址的系统变量

系统变量	描　述
MmSystemCacheStart	高速缓存的起始虚拟地址
MmSystemCacheEnd	高速缓存的结束虚拟地址
MiSystemCacheStartExtra	高速缓存特别内存的起始虚拟地址，如果高速缓存的大小 > 512MB
MiSystemCacheEndExtra	高速缓存特别内存的结束虚拟地址，如果高速缓存的大小 > 512MB
MmSizeOfSystemCacheInPages	页面中高速缓存的最大尺寸

11.3.2　高速缓存的物理大小

正如在前面所提到的那样，Windows 2000 高速缓存管理器与其他操作系统在设计上的主要差别是物理内存管理向全局内存管理器的授权。因此，处理工作集扩充和修整以及管理更改和备用链表的现有代码也可以用于控制系统缓存的大小、动态地平衡进程和操作系统之间对物理内存的需求。

系统高速缓存没有自己的工作集，而是共享一个包括高速缓存数据、页交换区、可调页的 Ntoskrnl 代码、可调页的驱动程序代码在内的系统集。正如在第 7 章“系统工作集”所解释的那样，这个单一的工作集在内部被叫作“系统高速缓存工作集（system cache working set）”，尽管

系统高速缓存仅是组成工作集的组件之一。从本书目的出发，将把工作集简单地称为系统工作集。

通过查看列举在表 11-3 中的性能计数器或系统变量，你可以对比系统高速缓存的物理大小和整个系统工作集大小以及系统工作集中的页错误信息。

表 11-3　用于系统高速缓存和页面错误信息物理大小的系统变量

性能计数器（字节）	系统变量（页面）	描　述
Memory：System Cache resident Bytes	MmSystemCachePage	系统高速缓存消耗的物理内存
Memory：Cache Bytes	MmSystemcacheWs.WorkingSetSize	系统工作集的大小（包括高速缓存、页交换区、可调页的代码和系统映射的视图）这不是高速缓存的大小（就象名字所暗示的那样）
Memory：Cache Bytes Peak	MmSystemcacheWs.Peak	系统工作集的峰值
Memory：cache Faults/Sec	MmSystemCacheWs.PageFaultCount	系统工作集中（不仅是高速缓存）的页错误

大多数声称显示系统高速缓存大小的实用程序（例如 Task Manager、Pview、Pstat、Pmon、Perfmtr 等等），事实上显示的是总的系统工作集的大小，而不仅仅是高速缓存大小。造成这种不准确的原因是性能计数器 Memory：Cache Bytes（请参见表 11-3）返回的是总的系统工作集大小，其中包括了系统高速缓存、页交换区和可调页系统代码等，而名称和说明文本暗示它只代表高速缓存大小。例如，如果你启动 Task Manager（通过按 Ctrl + Shift + Esc）并单击 Performance 标签，名为 System Cache 的字段实际上是系统工作集大小，就如你在图 11-4 中所见的那样。

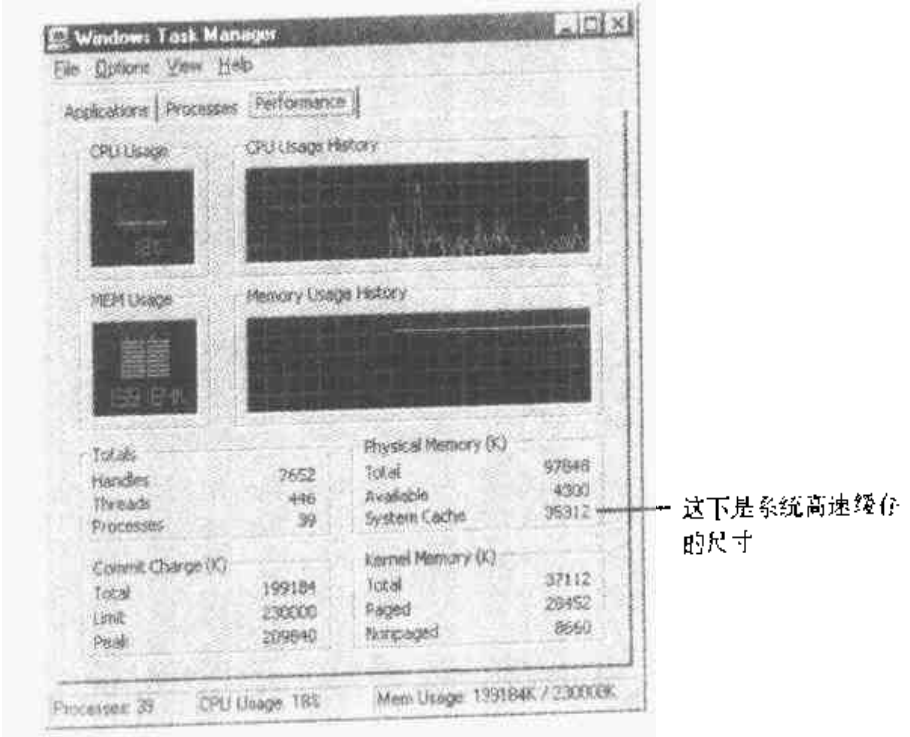


图 11-4　Windows 2000 Task Manager 报告的不是系统高速缓存的大小

有许多内部系统变量控制工作集的扩充和修整，例如 MmWorkingSetReductionMaxCacheWs，MmworkingSetReductionMinCacheWs，MmWorkingSetVolReductionMaxCacheWs 以及 MmPeriodocAgressiveCacheWsMin。虽然在本书中没有详细介绍这些变量，但在第 7 章中，我们描述了内存管理器管理工作集的一般规则

实验：查看高速缓存的工作集

调试器命令！filecache 可以输出有关高速缓存使用的物理内存的信息，还包括映射到虚拟地址控制块（VACB）中的文件名，如果可以的话，你可以看到如下输出（文件系统驱动程序利用无名的文件流高速缓存元数据）.

```
kd> !filecache 3
***** Dump file cache******
File Cache Information
  Current size 6344 kb
  Peak size    14440 kb
  Loading file cache database (131072 PTEs)...
  File cache PTEs loaded, loading PFNs...
  File cache has 278 valid pages
  File cache PFN data extracted
  Building kernel map
  Finished building kernel map

  Usage Summary (in kb):
Control Valid Standby Dirty Shared Locked PageTables  name
ff3f5908    20       0     0      0      0          0    No Name for File
ff3a0328     8       0     0      0      0          0    No Name for File
fe50ba68    72       0     0      0      0          0    No Name for File
fe4edd68     4       0     0      0      0          0    No Name for File
fe4edcc8    88       0     0      0      0          0    No Name for File
fe4ed7e8    16       0     0      0      0          0    No Name for File
ff457968     4       0     0      0      0          0    No Name for File
ff427de8     4       0     0      0      0          0    No Name for File
ff142708     4       0     0      0      0          0  mapped_file( ias.mdb )
ff163708     4       0     0      0      0          0    No Name for File
ff1314c8     4       0     0      0      0          0    No Name for File
ff0ec448     4       0     0      0      0          0  mapped_file( dnary.mdb )
ff0f23a8     4       0     0      0      0          0    No Name for File
fe50bda8     4       0     0      0      0          0    No Name for File
ff0691a8     4       0     0      0      0          0    No Name for File
ff071b08     4       0     0      0      0          0    No Name for File
ff068b28     4       0     0      0      0          0    No Name for File
ff0564c8     4       0     0      0      0          0    No Name for File
ff070c48     4       0     0      0      0          0    No Name for File
ff018d08     4       0     0      0      0          0    No Name for File
ff013d48     4       0     0      0      0          0    No Name for File
ff0e5f08     4       0     0      0      0          0    No Name for File
ff02a628     4       0     0      0      0          0    No Name for File
```

```
ff0f0ba8     4       0     0     0     0     0    No Name for File
ff0e8b28     4       0     0     0     0     0    No Name for File
ff05a628     4       0     0     0     0     0    No Name for File
ff104e08     8       0     0     0     0     0    No Name for File
ff016328     4       0     0     0     0     0    No Name for File
ff06da88     4       0     0     0     0     0    No Name for File
ff2d2e28   692       0     3     0     0     0  mapped_file( ntoskrnl.pdb )
ff0afce8     4       0     0     0     0     0    No Name for File
ff001ae8    12       0     0     0     0     0    No Name for File
feffe308     4       0     0     0     0     0    No Name for File
ff175508    12       0     0     0     0     0    No Name for File
feffd328    12       0     0     0     0     0    No Name for File
feffc408     4       0     0     0     0     0  mapped_file( mshtml.tlb )
fefed788    12       0     0     0     0     0    No Name for File
feff9d28     4       0     3     0     0     0  mapped_file
( relationship[1].gif )
ff03dd08     4       0     0     0     0     0  mapped_file
( emotional[1].gif )
ff059828     4       0     3     0     0     3  mapped_file
( ADSAdClient31[1].htm )
fefe72e8     4       0     0     0     0     0  mapped_file
( comp_medicine[1].gif )
fefe9868    12       0     0     0     0     0  mapped_file
( ADSAdClient31[1].htm )
unable to get subsection fefe4d08
feffd8a8     8       0     3     0     0     0  mapped_file
( ADSAdClient31[1].htm )
fefef008     4       0     3     0     0     0  mapped_file
( 002924Z0001_LG[1].gif )
ff2f64e8     4       0     0     0     0     0  mapped_file( SOFTWARE .OG \
```

11.4　高速缓存数据结构

高速缓存管理器使用如下数据结构跟踪被高速缓存的文件：

■ 系统高速缓存中的每个 256KB 的槽都由一个 VACB 描述。

■ 每个单独打开的高速缓存文件都有一个专有的高速缓存映射，其中包含了用于控制预读（在本章的后部分讨论）的信息。

■ 每个被高速缓存的文件都有一个单独的共享的高速缓存映射结构，该结构指向系统高速缓存中包含映射文件视图的槽。

这些结构和它们的关系将在下一节描述。

11.4.1　系统范围的高速缓存数据结构

高速缓存管理器使用称作“虚拟地址控制块（VACB）”的数组结构跟踪系统高速缓存中视图的状态。在系统初始化期间，高速缓存管理器分配一个单一的非页交换区的块来包含描述系统高速缓存所需的 VACB。它将 VACB 数组的地址存储在变量 CcVacbs 中。每个 VACB 都代表系统高速缓存中的一个 256KB 的视图，如图 11-5 所示。VACB 的结构显示图 11-6 中。

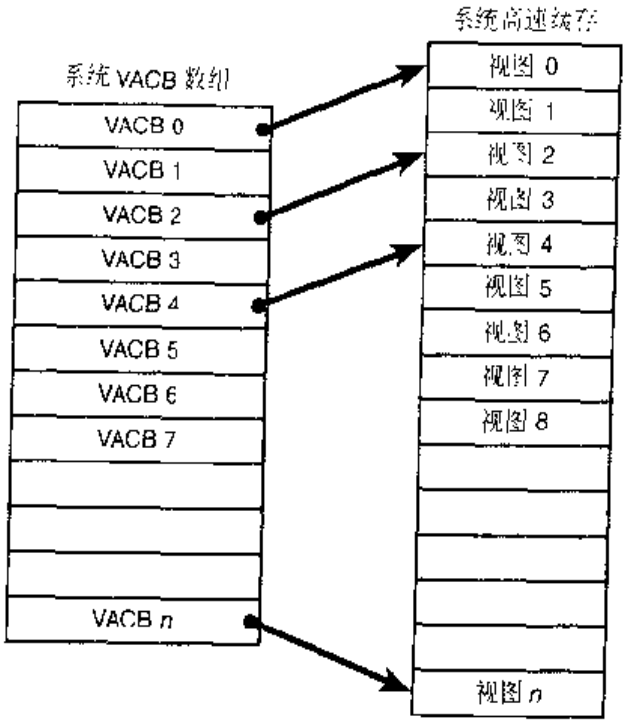


图 11-5 系统 VACB 数组

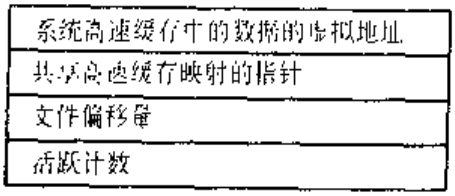


图 11-6 VACB 结构

正如你在图 11-6 中所看到的那样，VACB 的第一个字段是系统高速缓存中数据的虚拟地址。第二个字段是一个指向共享高速缓存映射结构的指针，该指针标识哪一个文件被高速缓存了。第三个字段标识视图在文件内部（总是基于 256KB 的间隔尺寸）的偏移量。最后，VACB 包含了对该视图的引用数量，即现有多少活动的读/写操作正访问该视图。在文件的 I/O 操作期间，文件的 VACB 引用计数增加。然后，在文件的 I/O 操作结束时，文件的引用计数减少。对于文件系统元数据的访问，活动计数表示有多少文件系统驱动程序只有视图锁定在内存中的页面。

11.4.2 每个文件的高速缓存数据结构

每次打开文件句柄都有一个相应的文件对象（在第 9 章中有文件对象详细介绍）。如果文件被高速缓存，文件对象就指向一个包含最后两次读取位置的专用高速缓存映射（private cache map）结构，这样高速缓存管理器就可以执行智能预读（描述在 11.5.2 节“智能预读”一节）。此外，所有文件对象的专用高速缓存映射都被链接在一起。

每个高速缓存文件（与文件对象相反）都有一个共享高速缓存映射（shared cache map）结构。该结构描述高速缓存文件的状态，包括其大小（基于安全考虑）和有效数据长度（有效数据长度字段的功能将在“写回高速缓存和延迟写”一节中进行解释）。共享高速缓存映射还指向“区域对象”（由内存管理器维护，它描述文件到虚拟内存的映射）与该文件相关的专用高速缓存映射链表以及所有在系统高速缓存中描述文件当前映射视图的 VACB。（关于区域对象指针的详细信息，请参阅第 7 章）。图 11-7 说明了每个文件的高速缓存数据结构间的关系。

当请求从一个特定的文件进行读取时，高速缓存管理器必须回答两个问题：

1）文件是否在高速缓存中？

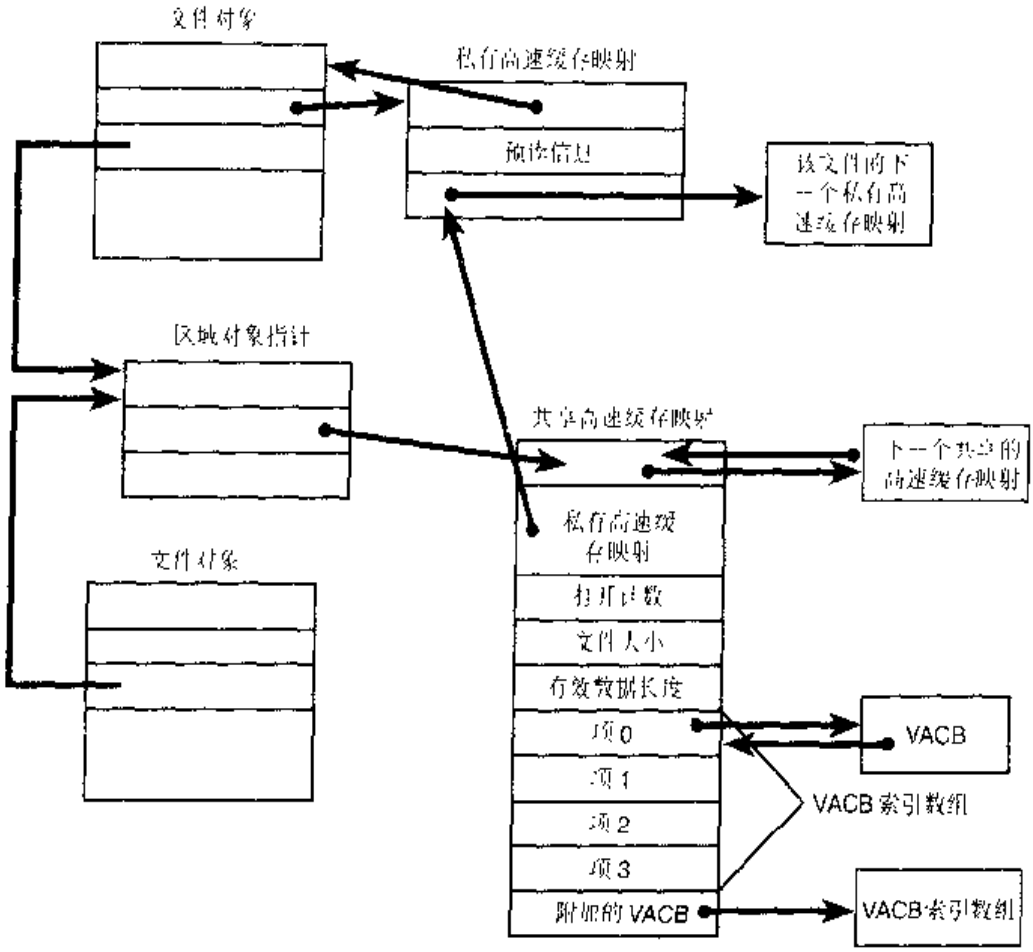


图 11-7　每个文件的高速缓存数据结构

2）如果在，哪个 VACB（如果有许多 VACB）涉及到了所请求的位置？

换句话说，高速缓存管理器必须确定所请求的地址的文件的视图是否被映射到了系统高速缓存。如果没有 VACB 包含所需文件的偏移量，那么被请求的数据在当前就没有被映射到系统高速缓存。

为了跟踪给定文件的哪些视图被映射到了系统高速缓存，高速缓存管理器将维护一个指向 VACB 的指针数组，称为“VACB 索引数组”（VACB index array）。VACB 索引数组的第一项指向文件的第一个 256KB，第二项指向该文件的第二个 256KB，依此类推。图 11-8 显示了当前映射到系统高速缓存中的三个不同文件的四个不同区域。

当一个进程在给定位置访问一个特定的文件时，高速缓存管理器在文件的 VACB 索引数组中查找适当的项，判断所请求的数据是否被映射到系统高速缓存中。如果该数组项非零（因此它包含一个指向 VACB 的指针），就说明被引用的文件区域在高速缓存中。接着，VACB 指向系统高速缓存中该文件视图被映射的位置。如果该项为零，高速缓存管理器就必须在系统高速缓存中寻找一个闲置的槽（也就是闲置的 VACB）来映射所请求的视图。

作为对大小的优化，共享高速缓存映射包含一个有四个项的 VACB 索引数组。因为每个 VACB 描述 256KB，所以这个固定大小的小索引数组总共可以描述的文件高达 1MB。如果文件大于 1MB，将在非页交换区分配单独的 VACB 索引数组。将文件的大小除以 256KB，并在有余

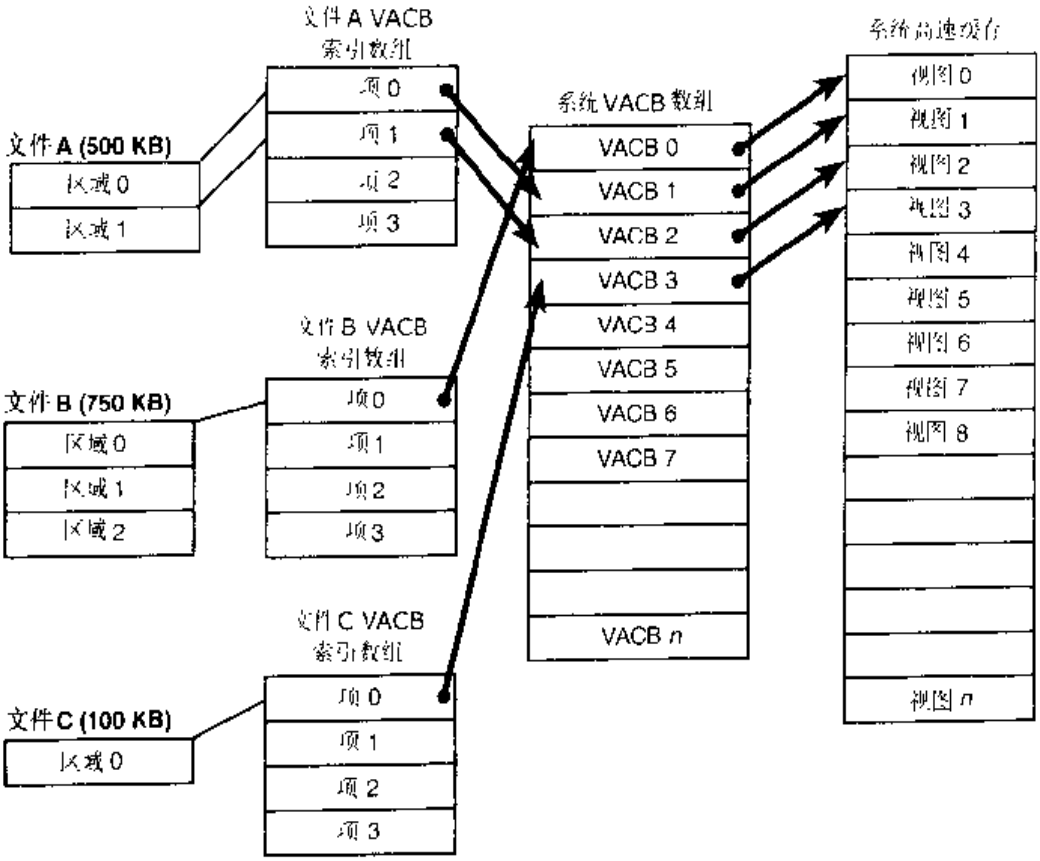


图 11-8 VACB 索引数组

数的情况下进行取整加 1，即为该索引数组的大小。然后，共享高速缓存映射指向这个单独的结构

更进一步的优化是，如果文件超过 32GB，从非页交换区分配的 VACB 索引数组将变成稀疏的多级索引数组，其中每个数组包含 128 项。可以利用以下公式计算所需要的索引级数：

（表示文件大小所需要的位数 - 18）/7

取不小于该方程结果的最小整数为最终结果。方程中 18 来源于以下事实：一个 VACB 代表 256KB，而 256KB 是 2^{18}。7 是因为该数组中每级有 128 项，2^7 等于 128。这样，一个最大的需要 63 位（高速缓存管理器所支持的最大尺寸）来描述的文件仅仅只需要 7 级。该数组是稀疏的，因为只有高速缓存管理器分配的分支在最低级的索引数组中具有活动视图。图 11-9 显示了一个稀疏文件多级 VACB 数组例子。该文件大到需要三级才能表示。

为了高效处理特别大而又只有一小部分有效数据的稀疏文件，必须采用该方案，因为只有分配了足够的数组才能处理文件并行映射的视图。例如，一个 32GB 的文件，只有 256KB 被映射到高速缓存管理器虚拟地址空间，将需要一个分配了 3 个索引数组的 VACB 数组，因为该数组的每个分支只有一个映射，一个 32GB（2^{35} 字节）的文件需要一个三级数组。如果高速缓存管理器不为该文件使用多级 VACB 数组优化，那么将不得不分配一个包含 128000 项 VACB 数组，或者是等价的 1000 个索引数组。

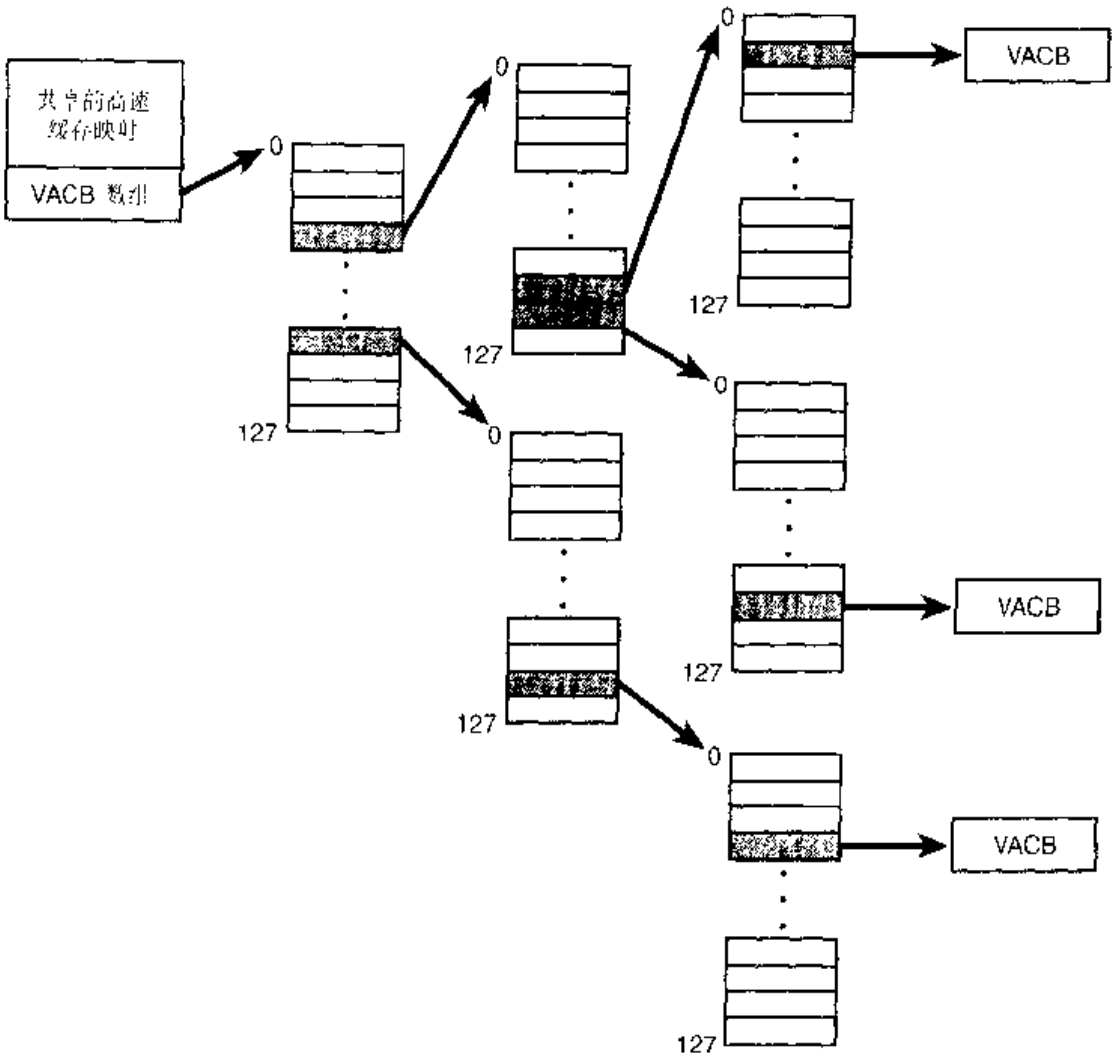


图 11-9　多级 VACB 数组

11.5　高速缓存操作

在这一节，你将看到高速缓存管理器如何代表文件系统驱动程序对文件数据进行读写操作。记住，只有文件打开（例如，使用 Win32 的 CreateFile 函数）后，在文件的 I/O 操作中才涉及高速缓存管理器。被映射的文件不经过高速缓存管理器，用 FILE_FLAG_NO_BUFFERING 标志打开的文件也不经过高速缓存管理器。

11.5.1　写回高速缓存和延迟写

Windows 2000 高速缓存管理器实现了带有延迟写的写回高速缓存，这意味着写入文件的数据首先存储在内存的高速缓存页面中，稍后再写入磁盘。这样，便允许写操作进行短时间的积累，然后将所有数据一次性刷新到磁盘，从而降低了总的磁盘 I/O 操作数。

高速缓存管理器必须明确地调用内存管理器以刷新高速缓存页面，因为只有当对物理内存的需求超出所提供的内存时，内存管理器才能将内存的内容写入磁盘，对易失性数据也是如此。然而，高速缓存的文件数据是代表非易失性磁盘数据。如果一个进程修改了高速缓存的数

据，用户将期望修改的内容及时地在磁盘上反映出来

确定高速缓存的刷新频率是非常重要的。如果高速缓存刷新过于频繁，系统性能将会由于不必要的 I/O 操作而降低。如果高速缓存刷新过少，将面临在系统失败时丢失所修改的文件数据（当用户已经申请应用程序保存所做的修改而仍出现了数据丢失时，会使用户更加愤怒）或将物理内存用尽的危险（因为它正被过量修改的页面占用着）。

要平衡这种利害关系，由高速缓存管理器每秒一次建立一个系统线程——延迟书写器——排列将写到磁盘的系统高速缓存中脏页的八分之一，将其写入磁盘。如果脏页面的速率大于已经确定的应该进行写的数目，那么延迟书写器将计算并且写一定数量的额外脏页面以匹配该速率。来自于系统范围的关键工作者线程交换区的系统工作者线程真正执行 I/O 操作。

注意 对于安全级别为 C2 的文件系统（如 NTFS），高速缓存管理器为文件系统提供了一种跟踪方式来跟踪有多少数据在什么时候被写入到文件中。在延迟书写器将脏页面刷新到磁盘后，高速缓存管理器将通知文件系统，指示其更新文件有效数据长度的视图。

通过检查列在表 11-4 中的高速缓存性能计数器或系统变量，你可以查看延迟书写器的活动。

表 11-4 用于检查延迟书写器的系统变量

性能计数器（概率）	系统变量（计数）	描述
Cache: Lazy Write Flushes/Sec	CcLazyWriteIos	延迟书写器刷新的数目
Cache: Lazy Write Pages/Sec	CcLazyWritePages	由延迟书写器所写的页面数

1. 计算脏页阈值

"脏页阈值"（dirty page threshold）是指在唤醒延迟书写器将页面写回磁盘之前，高速缓存管理器保持在内存中的页面数量。该数值在系统初始化时计算的，且依赖于物理内存大小和注册表键值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ SessionManager \ Memory Management \ LargeSystemCache 的值。在 Windows 2000 Professional 和 Windows 2000 Server 中，该值默认为 0。你可以通过 Windows 2000 Server 系统中的 GUI 调整该值。这是通过修改文件服务器的服务属性来实现的。（调出网络连接的属性，双击 File And Printer Sharing For Microsoft Networks）。虽然在 Windows 2000 Professional 中也有该服务，但是不能调整它的参数。图 11-10 显示了一个对话框，通过它可以修改在 Server 网络服务中分配给本机和网络应用程序的内存数量。

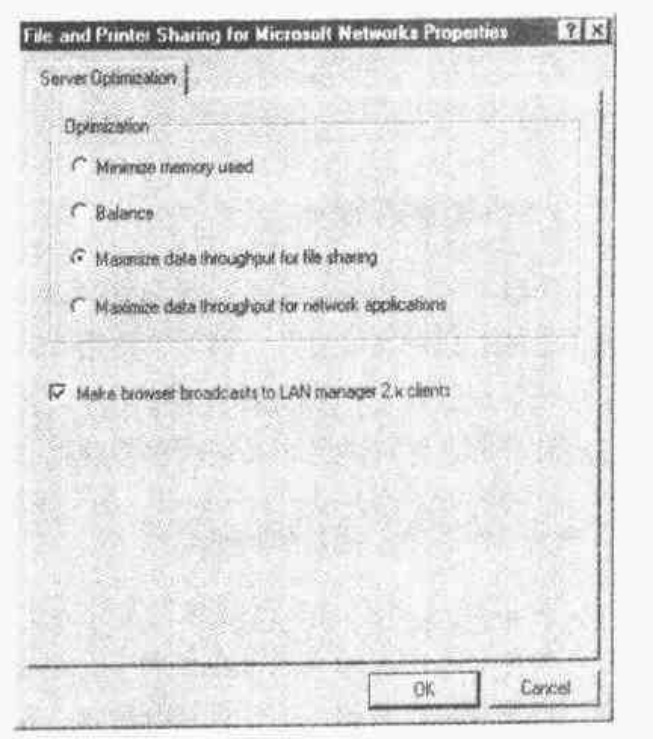


图 11-10 File And Printer Sharing For Microsoft Networks Properties 对话框，可以用来修改 Windwos 2000 Server 网络服务的属性

图 11-10 所示的设置，Maximize Data Throughput For File Sharing，是运行着 Terminal Services 的 Server 系统的默认值，即 LargeSystemCache 的值为 1。选择其他设置会将 LargeSystemCache 值设置为 0。虽然在 File And Printer Sharing For Microsoft Networks Properties 对话框的 Optimization 中的四个设置都影响系统高速缓存的行为，通常它还会影响文件服务器服务的行为。

表 11-5 包含了用于计算脏页阈值的算法。如果系统最大工作集大小大于 4MB ---这是经常发生的，表 11-5 中的计算将被忽略（关于内存管理器是如何选择系统工作集大小的，即它是如何确定工作集是小、中、还是大的信息，请参阅第 7 章）。当最大工作集超过 4MB 时，脏页阈值被设置为系统最大工作集大小减去 2MB。

表 11-5　计算脏页面阈值的算法

系统内存大小	脏页面阈值
小	物理页面/8
中	物理页面/4
大	以上两值之和

2. 禁用对文件的延迟写

如果调用 Win32 CreateFile 函数，指定 FILE _ ATTRIBUTE _ TEMPORARY 标志，建立一个临时文件，除非出现严重的物理内存不足或者是文件被关闭，否则延迟书写器不会把脏页写入磁盘。延迟书写器的这种特性提高了系统性能——延迟书写器不会立即把最终可能被丢弃的数据写入磁盘。应用程序通常在关闭之前删除临时文件。

3. 强制高速缓存写穿透（Write through）至磁盘

由于一些应用程序不允许在向磁盘写文件和查看磁盘更新之间出现即使很短的延迟，所以高速缓存管理器还支持基于每个文件的写穿透高速缓存：修改一经产生便被写入磁盘。为了开启写穿透高速缓存，需在调用 CreateFile 函数时指定 FILE _ FLAG _ WRITE _ THROUGH 标志。作为另一种选择，当一个线程到达需要把它的数据写入磁盘的时候，可以使用 Win32 FlushFileBuffers 函数显示地刷新一个打开的文件。通过列在表 11-6 中的性能计数器或者系统变量，可以观察作为写穿透 I/O 请求或者是明确调用 FlushfileBuffers 的结果的高速缓存刷新操作。

表 11-6　查看高速缓存刷新操作的系统变量

性能计数器（频率）	系统变量（计数）	描　　述
Cache: Data Flushes/Sec	CcDataFlushes	高速缓存页面被显示刷新的次数或者是由于写穿透（Write through）而产生的刷新次数
Cache: Data Flushes Pages/Sec	CcDataPages	显示地刷新的页面数或者是由于写穿透（Write through）造成的页面刷新数

4. 刷新映射文件

如果延迟书写器必须将数据从视图写到磁盘，而该视图还被映射到其他进程的地址空间，

此时情况将稍微复杂些。因为高速缓存管理器只知道它更改过的页面（被其他进程修改的页面只有修改的进程才知道，因为页表项中被修改页面的修改位保存在进程的专用页表中）。为处理这种情况，内存管理器在用户映射文件时通知高速缓存管理器。当在高速缓存内刷新这样一个文件时（例如作为调用 Win32 FlushFileBuffer 函数的结果），高速缓存管理器会写在高速缓存中的脏页面，然后将检查该文件是否还被其他进程映射。当高速缓存管理器查看到这样的文件，它将刷新该节的整个视图，以便将第二个进程可能修改的页面写出。如果用户映射一个也在高速缓存中打开的文件的视图，当视图没有被映射时，被修改过的页面就会标志为脏页。当延迟书写器线程在以后刷新该视图时，那些脏页将被写入磁盘。只要按下列顺序进行操作这个过程就会起作用：

1）用户取消视图映射。

2）进程刷新文件缓冲区。

如果不按照该顺序进行操作，你将无法预测哪些页面将被写入磁盘。

11.5.2 智能预读

Windows 2000 高速缓存管理器采用空间局部性原则，在当前正在读取的数据的基础上，通过预测调用进程下一步可能读取的数据来执行“智能预读”。因为系统高速缓存是建立在虚拟地址的基础上，对于一个特定的文件来说该文件是连续的，所以它们在物理内存中是否连续并不重要。对于逻辑块高速缓存的文件预读更为复杂，需要文件系统驱动程序和块高速缓存的密切协同，因为高速缓存系统是建立在被访问的磁盘数据的相对位置上，而文件并不一定是连续地存储在磁盘上。

预读有两种类型：“虚拟地址预读”（virtual address read - ahead）和“带历史记录的异步预读”（asynchronous read - ahead with history）这两种类型的“预读”将在下面的两节中解释。你可以通过 Cache: Read Aheads/Sec 性能计数器或 CcReadAheadIos 系统变量来检查预读活动。

1. 虚拟地址预读

回忆第 7 章的情况，当内存管理器解决页错误时，它就会把靠近一个明显已经访问过的页的几页读入内存，这种方法称作“聚类”（clustering）。对于那些按顺序读取的应用程序，这种“虚拟地址预读”（Virtual address read - ahead）操作减少了检索数据时所必需的磁盘读取次数。内存管理器的这种方法的唯一不足是：由于这种预读是在解决页面错误的情况下进行的，所以必须同步进行。而此时等待被调页的数据返回内存的线程处于等待状态。

2. 带有历史记录的异步预读

由内存管理器执行的虚拟地址预读改善了 I/O 性能，但是这仅限于按顺序访问的数据。为了将预读的好处扩展到随机访问数据的情况下，高速缓存管理器将为正被访问的文件句柄在专用高速缓存映射中维护最近两次读取请求的历史记录，这种方法被称为“带有历史记录的异步预读”。如果可以从调用程序显示地随机读取中确定某种模式，那么高速缓存管理器就会把这种模式延伸。例如，如果调用程序先读取第 4000 页，然后是第 3000 页，那么高速缓存管理器就会假定调用程序需要读的下一页是第 2000 页，并预读该页。

注意 尽管调用程序建立预测序列必须启动最少三个读操作，但是只有两个存储在专

用高速缓存映射中。

为了提高预读的效率，Win32 CreateFile 函数提供了一个表示顺序文件访问的标志：FILE_FLAG_SEQUENTIAL_SCAN。如果设置该标志，高速缓存管理器就不会为调用程序的预测保存历史记录，而是执行顺序预读。然而，当文件被读入高速缓存的工作集中时，高速缓存管理器将取消不再活动的文件视图，并指示内存管理器将属于该视图的页面放到备用列表或修改列表（如果页面被修改的话）的前端，从而使它们能够尽快重用。通过每次读取时的单独 I/O 操作，预读数据将是原来的三倍（如，192KB 而不是 64KB）。在调用程序继续读取的同时，高速缓存管理器就会预读附加的数据块，一直保持有一次先于调用程序的预读（当前读的大小）。

高速缓存管理器的预读是异步的，因为它是在不同于调用程序线程的线程中进行的，并与调用程序的执行并行进行。当被调用来检索被高速缓存的数据，高速缓存管理器首先访问被请求的虚拟页面来满足该请求，然后将一个附加的 I/O 请求插入到系统工作者线程的队列来获取附加数据。工作者线程然后在后台执行，读取调用程序的下一次读请求中参与的附加数据。预读的页面被载入到内存而程序继续执行，因此当调用程序需要该数据时，该数据已经在内存中。

尽管带有历史记录的异步预读技术占用的内存要比标准的预读多，但是却使磁盘 I/O 减至最小，并且进一步改善了应用程序大量读取高速缓存顺序数据的性能。Cache: Read Aheads/Sec 性能计数器显示顺序访问预读操作。

对于那些没有可预测读取模式的应用程序来说，可以在调用 CrateFile 函数时指定 FILE_FLAG_RANDOM_ACCESS 标志。该标志命令高速缓存管理器不要尝试预测应用程序下一步要读取的位置，这样就禁止了预读。该标志还将阻止高速缓存管理器在访问文件时强制性取消文件视图的映射，从而降低在应用程序在此访问该文件的部分时文件的映射/取消映射活动。

11.5.3　系统线程

正如在前面所提到的那样，高速缓存管理器通过为公共关键系统工作者线程交换区提交请求来执行延迟写和预读 I/O 操作。然而，对于内存小的和内存中等的系统，可供使用的线程的数目比关键工作者系统线程的总数少一个（对于内存大的系统，比总数少两个）。

在内部，高速缓存管理器把它的工作请求组织到两个列表中（尽管这是由可执行的工作者线程来为它们服务的）：

- “快速队列”（express queue）用于预读操作。
- “常规队列”用于延迟书写扫描（用于刷新脏数据）、延误写和延迟关闭。

为了跟踪工作者线程需要执行的工作项（work item），高速缓存管理器创建其自身的内部每个处理器的备份列表（look - aside list），这是一个固定长度的工作者队列项目结构表。（备份列表在第 7 章中讨论）。工作者队列项目的数目取决于系统的大小：对于小内存系统是 32、对于中等内存系统是 64、对于大内存的 Windows 2000 Professional 系统是 128，而对于大内存系统 Windows 2000 Server 是 256。

11.5.4　快速 I/O

任何时候只要有可能，就可通过使用被称作快速 I/O 的高速机制来处理高速缓存文件的读取

和写入。快速 I/O 是一种读取或写入高速缓存文件的方法，通过此方法不会产生一个 I/O 请求包（IRP），这描述在第 7 章中。利用快速 I/O，I/O 管理器就可以调用文件系统驱动程序的快速 I/O 例程来查看是否可以直接从高速缓存管理器进行 I/O 而不产生 IRP。

由于 Windows 2000 高速缓存管理器跟踪哪些文件的哪些块在高速缓存中，因此文件系统驱动程序可以简单地通过复制或从页面映射到被引用的实际文件，不经过产生 IRP 的开销，就可使用高速缓存管理器访问文件数据。

快速 I/O 并不总是发生。例如，在对文件进行第一次读取或写入时需要设置该文件以用于高速缓存（将文件映射到高速缓存中然后设置高速缓存数据结构，等内容在前面的“高速缓存数据结构”一节中做了解释）。如果调用程序指定了异步读入或写入，快速 I/O 将无法使用。原因是在满足将缓冲区复制到系统高速缓存或从系统高速缓存复制所需的调页 I/O 操作期间，调用程序可能会被延迟（stall），这样就不能真正提供所请求的异步 I/O 操作。甚至在同步 I/O 时，文件系统驱动程序可能会判定不能使用快速 I/O 机制处理 I/O 操作，例如，如果正在操作的文件有被锁定的字节范围（作为调用 Win32 LockFile 和 UnlockFile 函数的结果）。由于高速缓存管理器不知道锁定的是哪个文件的哪一部分，就需要文件系统驱动程序检查读取或写入的有效性，这就会产生一个 IRP。快速 I/O 的决策树显示在图 11-11。

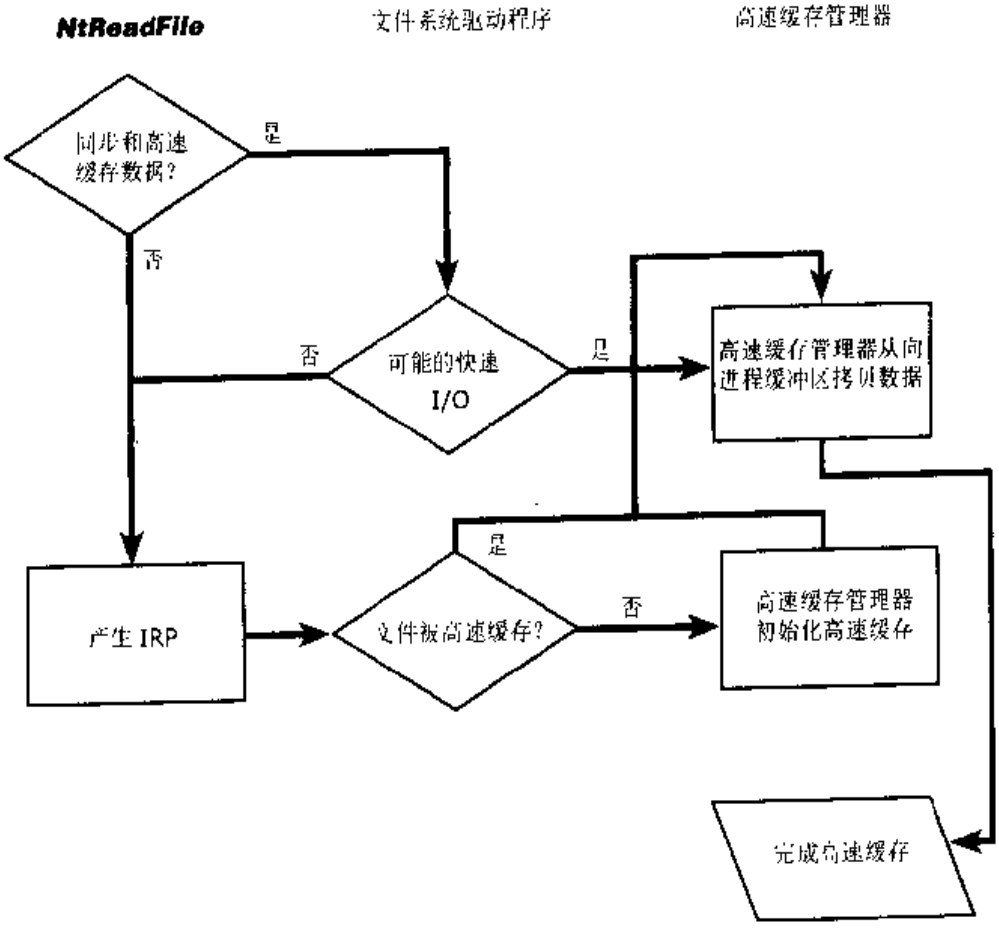


图 11-11 快速 I/O 决策树

下面是使用快速 I/O 进行读写或写入时涉及的几个步骤：

1）线程执行一个读取或写入操作。

2）如果文件被高速缓存并且 I/O 是同步的，该请求会传送到文件系统驱动程序的快速 I/O 入口点。如果文件没有被高速缓存，文件系统驱动程序会设置该文件以用于高速缓存，这样下一次就可以用快速 I/O 来满足读取或写入请求了。

3）如果文件系统驱动程序的快速 I/O 例程判定可以使用快速 I/O，那么它将调用高速缓存管理器的读取或写入例程直接访问高速缓存中的文件数据（如果不能使用快速 I/O，文件系统驱动程序将返回到 I/O 系统，然后 I/O 系统为该 I/O 产生一个 IRP，最终调用文件系统的常规读取例程）。

4）高速缓存管理器将所提供的文件偏移量转换为在高速缓存中虚拟地址。

5）对于读取，高速缓存管理器把数据从高速缓存复制到提出请求的进程的缓冲区中；对于写入，它把数据从缓冲区中复制到高速缓存中。

6）发生以下操作之一：

a）对于读取，更新调用程序专用高速缓存映射中的预读信息。

b）对于写入，设置在高速缓存中所有被修改页面的脏位以便延迟书写器能够将其刷新到磁盘中。

c）对于写穿透（Write through）文件，所有的修改都被刷新到磁盘中。

注意　快速 I/O 路径不限制所请求的数据已经驻留在物理内存中的情况。就象在前面列表中的第 5 步和第 6 步那样，高速缓存管理器简单地访问已经打开的期望从中读取数据的文件的虚拟地址空间。如果高速缓存失败，内存管理器将动态地将数据分页载入物理内存。

列举在表 11-7 中的性能计数器或系统变量可以用来确定系统的快速 I/O 活动。

表 11-7　用于确定快速 I/O 活动的系统变量

性能计数器（频率）	系统变量（计数）	描　　述
Cache: Sync Fas Reads/Sec	CcFastReadWait	作为快速 I/O 请求处理的同步读取
Cache: Async Fast Reads/Sec	CcFastReadNoWait	作为快速 I/O 请求处理的异步读取（该值总为零因为在 Windows 2000 中没有异步快速读取）
Cache: Fast Read Resource Misses/sec	CcFastReadresourceMiss	由于资源冲突而不能满足的快速 I/O 操作（对于 FAT 可能出现该情况，NTFS 则不会）
Cache: Fast Read Not Possibles/Sec	CcFastreadNotPossible	不能满足的快速 I/O 操作（由文件系统驱动程序决定；如带字节范围锁定的文件不能使用快速 I/O）

11.6　高速缓存支持例程

当一个文件用于读写操作而被首次访问时，文件系统驱动程序负责确定文件的某些部分是否被映射到系统高速缓存中。如果没有，文件系统驱动程序必须调用 CcInitializeCacheMap 函数，建立在前一节中所描述的每个文件的数据结构。

一旦文件被设置成高速访问，文件系统驱动程序将调用几个函数中的一个访问该文件中的数据。访问高速缓存数据的基本方法有三种，每一种适用于一个特定情况。

- 复制读取法在系统空间的高速缓存缓冲区和用户空间的进程缓冲区之间复制用户数据。
- 映射和牵制法使用虚拟地址直接将数据读取和写入到高速缓存缓冲区。
- 物理内存访问法使用物理地址直接将数据读取和写入到高速缓存冲区。

当内存管理器处理页错误时，文件系统驱动程序必须提供两种版本的文件读取操作（高速缓存和非高速缓存）以避免出现无穷循环。当内存管理器通过调用文件系统从文件（经过设备驱动程序）中获取数据来解决页错误时，它必须设置 IRP 中的“非高速缓存”标志，指定其为非高速缓存读取操作。

下面三节将解释这些高速缓存的访问机制、它们的目的以及使用方法。

11.6.1 复制到高速缓存和从高速缓存复制

因为系统高速缓存位于系统空间，所以被映射到每个进程的地址空间。然而，正如所有的系统空间页面一样，高速缓存页面不接受来自用户模式的访问，因为那样有可能造成潜在的安全漏洞（例如，一个进程可能无权读取某个文件，而该文件的数据当前正包含在系统高速缓存中）。所以，用户应用程序对高速缓存文件的读写必须由内核模式例程来服务。内核模式例程将在系统空间高速缓存缓冲区与驻留在进程地址空间的应用程序缓冲区之间复制数据。文件系统驱动程序用来执行这个操作的函数列举在表 11-8 中。

表 11-8 读写高速缓存的内核模式函数

函 数	说 明
CcCopyRead	从系统高速缓存复制指定范围的字节到用户缓冲区
CcFastCopyRead	更快的 CcCopyRead 变体，但只限于 32 位的偏移量与同步读取（NTFS 采用，而不是 FAT）
CcFastWrite	从用户缓冲区复制指定范围的字节到系统高速缓存
CcFastcopyWrite	更快的 CcFastWrite 变体，但只限于 32 位的偏移量与同步、非写穿透（Write through）写（NTFS 采用，而不是 FAT）

通过列举在表 11-9 中的性能计数器或系统变量，可以检查对高速缓存的读取活动。

表 11-9 检查对高速缓存读取活动的系统变量

性能计数器（频率）	系统变量（计数器）	说 明
Cache:Copy Read Hits %	(CcCopyReadWait + CcCopyReadNowait)/ (CcCopyReadWait + (CcCopyReadWaitMiss + CcCopyReadNoWait) + CcCopyReadNoWaitMiss)	对高速缓存中文件的复制读取百分比（复制读取可能仍产生调页 I/O ——计数器 Memory: Cache Faults/Sec 报告系统工作集的页面出错活动，但是包括硬件和软件页面错误，所以该计数器仍然不能指示真正的由高速缓存错误引起的调页 I/O）
Cache:Copy Reads/Sec	CcCopyReadWait + CcCopyReadNoWait	总的对高速缓存的复制读取

（续）

性能计数器(频率)	系统变量(计数器)	说　　明
Cache:Sync Copy Reads/Sec	CcCopyReadWait	对高速缓存的同步读取
Cache: Async Copy Reads/Sec	CcCopyReadNoWait	对高速缓存的异步读取

11.6.2　带映射和牵制接口的高速缓存

就象用户应用程序读写磁盘文件的数据时一样，文件系统驱动程序需要读取和写入描述文件自身的数据(元数据或卷结构数据)。然而，由于文件系统驱动程序是在内核模式运行的，所以如果高速缓存管理器得到适当的通知，它们将在系统高速缓存中直接修改数据。为进行这种优化，高速缓存管理器提供了表 11-10 所示的函数。这些函数允许文件系统驱驱动程序在虚拟内存中查找文件系统元数据驻留在内存中的位置，这允许在不借助中间缓冲区的情况下直接修改元数据。

表 11-10　查找元数据位置的函数

函　　数	说　　明
CcMapData	映射读取的字节范围
CcPinRead	映射读写的字节范围，并牵制它
CcPrepinPinWrite	映射并牵制写的字节范围
CcPinMappedData	牵制先前映射的缓冲区
CcSetDirtyPinnedData	通知高速缓存管理器数据已经修改完毕
CcUnpinData	释放页面以使从内存中删除它们

如果文件系统驱动程序需要读取高速缓存中的文件系统元数据，那么它可以调用高速缓存管理器的映射接口来获取该数据的虚拟地址。高速缓冲管理器将触及的所有被请求的页面，并把它们调入内存，然后将控制权返回给文件系统驱动程序。然后，文件系统驱动程序就可以直接访问这些数据。

如果文件系统驱动程序需要修改高速缓存页面，它可以调用高速缓存管理器的牵制服务，该服务可以保护在内存中修改的页面。这些页面并不是真正锁定在内存中（这类似于设备驱动程序为直接内存访问传输而锁定页面）。相反，内存管理器的映射页面书写器（在第 7 章有解释）发现页面被牵制之后，不会将这些页面写入磁盘，直到它们被文件系统驱动程序释放为止。页面被释放后，高速缓存管理器刷新对磁盘的所有修改，并释放元数据占用的高速缓存视图。

映射与牵制（Pinning）接口解决了实现文件系统时遇到的一个棘手的问题：缓冲区管理。如果不能直接操作高速缓存元数据，当更新卷结构时，文件系统必须预测所需要的最大缓冲区数目。通过让文件系统在高速缓存中直接访问和修改元数据，高速缓存管理器对缓冲区的需求，只要在内存管理器提供的虚拟内存中的简单地更新卷结构。文件系统所遇到唯一限制就是可用内存的数量。

你可以通过列在表 11-11 中的性能计数器或系统变量来检查高速缓存中的牵制和映射活动。

表 11-11 检查牵制和映射活动的系统变量

性能计数器（频率）	系统变量（频率）	说明
Cache: data Map Hits %	(CcMapDataWait + CcMapDataNowait) / (CcMapDataWait + CcMapDataNoWait) + (CcMapDataWaitMiss + CcCopyReadNoWaitMiss)	对高速缓存中文件数据的映射百分比（复制读取可能仍在产生调页 I/O）
Cache: data Maps/Sec	CcMapDataWait + CcMapDataNoWait	总的高速缓存数据映射
Cache: Sync Data Maps/Sec	CcMapDataWait	同步高速缓存数据映射
Cache: Async Data Maps/Sec	CcMappDataNoWait	异步高速缓存数据映射
Cache: Data Map Pins/Sec	CcPinMappedDataCount	牵制映射数据的请求数
Cache: Pin Read Hits %	(CcPinReadWait + CcPinReadNowait) / (CcPinReadWait + CcPinReadNoWait) + (CcPinReadWaitMiss + CcPinReadNoWaitMiss)	受牵制的读取高速缓存中文件的百分比（复制读取可能仍在产生调页 I/O）
Cache: Pin Reads/Sec	CcPinReadWait + CcPinReadNoWait	总的对高速缓存的牵制读取
Cache: Sync Pin Reads/Sec	CcPinReadWait	对高速缓存的同步牵制读取
Cache: Async Pin Reads/Sec	CcPinReadNoWait	对高速缓存的异步牵制读取

11.6.3 带直接存储器存取接口的高速缓存

除了映射和牵制接口可以用来直接访问高速缓存中元数据之外，高速缓存管理器提供了第三个高速缓存数据接口：直接存储器存取（DMA）。DMA 函数用来读取或写入高速缓存页面而不许打扰缓冲区，如网络文件系统通过网络进行传输。

DMA 接口把高速缓存用户数据的物理地址返回给文件系统（而映射和牵制接口返回的是虚拟地址），这些物理地址可以直接将数据从物理内存传输到网络设备。尽管少量的数据（1KB－2KB）可以用一般的基于缓冲区的复制接口，但是对于较大规模的数据传输，DMA 接口可以显著提高网络服务器的性能，处理来自远程系统的文件请求。

为了描述这些物理内存的引用，需要使用“内存描述符列表”（MDL 在第 7 章中介绍）。表 11-12 中描述的四个独立的函数创建高速缓存管理器的 DMA 接口。

表 11-12 创建 DMA 接口的函数

函数	说明
CcMdlRead	返回一个描述指定字节范围的 MDL
CcMdlreadComplete	释放 MDL

（续）

函　数	说　明
CcMdlWrite	返回一个描述指定字节范围的 MDL（可能包含 0）
CcMdleWriteComplete	释放 MDL 并标记写的范围

通过列举在表 11-13 中性能计数器或系统变量，可以查看高速缓存 MDL 的活动情况。

表 11-13　查看高速缓存 MDL 的活动的系统变量

性能计数器(频率)	系统变量(频率)	说　明
Cache:MDL Read Hits %	(Cc MDL ReadWait + Cc MDL ReadNowait)/ (Cc MDL ReadWait + (Cc MDL ReadWaitMiss + Cc MDL readNoWait) + Cc MDL ReadNoWaitMiss)	对高速缓存中文件的 MDL 读取百分比（引用由 MDL 读取所满足的页面仍可能产生分页 I/O）
Cache: MDL Reads/Sec	Cc MDL ReadWait + Cc MDL ReadNoWait	高速缓存总的 MDL 读取
Cache: Sync MDL Reads/Sec	Cc MDL ReadWait	对高速缓存的同步 MDL 读取
Cache: Async MDL Reads/Sec	Cc MDL ReadNoWait	对高速缓存的异步 MDL 读取

11.6.4　写入调整

Windows 2000 必须确认调度的写入操作是否会影响系统的性能，然后再安排各项延迟写入操作。首先，Windows 2000 要询问立即写入的一定数量的字节是否会损害系统性能，并在必要时阻止该项写入操作。然后，它设置回调，当允许再次写入时会自动写入这些字节。一旦获悉将要进行一项写入操作，高速缓存管理器就要确定目前高速缓存中有多少脏页以及有多少可用的物理内存。如果空闲的物理页面太少，高速缓存管理器将立即暂停请求向高速缓存中写入数据的文件系统线程。高速缓存管理器的延迟书写器将一些脏页刷新到磁盘上，然后允许被暂停的文件系统线程继续执行。这种“写入调整”（write throttling）可以在文件系统或网络服务器进行大规模写入操作时避免因缺少内存而导致系统性能的下降。

对于低速通信线路的网络重定向数据传输，写入调整同样很有用。例如，假定一个本机进程要通过一条 9200 波特的线路向远程文件系统写入大量的数据。这些数据在高速缓存的延迟书写器刷新高速缓存之前不会被写入远程磁盘。如果网络重定向程序已经积累了许多将要刷新到磁盘的脏页，那么接收者就有可能在传送完成前收到网络超时。借助 CcSetDirtyThreshold 函数，高速缓存管理器允许网络重定向程序设置一个可以容忍的高速缓存脏页数量界限，以防止出现上述情况。通过限制脏页的数量，重定向程序确保了高速缓存刷新操作导致网络超时。

实验：查看写入调整参数

内核调试器命令 !defwrites 可以输出高速缓存管理器使用的内核变量的值，其中包括在决定是否进行写入调整时文件高速缓存中的脏页面数量：

```
kd> !defwrites
*** Cache Write Throttle Analysis ***

        CcTotalDirtyPages:                 758 (    3032 Kb)
        CcDirtyPageThreshold:              770 (    3080 Kb)
        MmAvailablePages:                42255 (  169020 Kb)
        MmThrottleTop:                     250 (    1000 Kb)
        MmThrottleBottom:                   30 (     120 Kb)
        MmModifiedPageListHead.Total:      689 (    2756 Kb)

CcTotalDirtyPages within 64 (max charge) pages of the threshold, writes
  may be throttled

Check these thread(s): CcWriteBehind(LazyWriter)
Check critical workqueue for the lazy writer, !exqueue 15
```

该输出显示脏页面的数量接近触发写入调整的阈值（CcDirtyPageThrehold）。因此如果某个进程在该实验的时刻试图写入超过 12 页（48KB）页面数，那么该写入可能回被推迟直到书写器降低了脏页面的数值。

11.7 小结

Windows 2000 高速缓存管理器为降低磁盘 I/O、提高整体系统的吞吐量提供了一个的高速的、智能化的机制。通过以虚拟块为基础的高速缓存，Windows 2000 高速缓存管理能够执行智能预读、依靠全局内存管理器的基本映射文件来访问文件数据，高速缓存管理器能够提供专用的快速 I/O 操作机制以减少读取和写入操作所占用 CPU 的时间；同时，它将所有与物理内存管理有关的问题留给单独的 Windows 2000 全局内存管理器，从而减少了代码复制，提高了效率。

第12章 文件系统

在本章，我们将展示由 Microsoft Windows 2000 所支持的文件系统格式的全貌。然后，我们将描述文件系统驱动程序的类型和它们的基本操作，包括它们是如何同其他系统组件如内存理器和高速缓冲存储器相互作用的。Windows 2000 包含一个称为“NTFS 文件系统”的固有的文件系统格式。在本章我们将集中讨论关于 NTFS 的卷磁盘布局和 NTFS 的高级特征，如压缩、可恢复性、配额和加密。

为了充分地理解本章，必须要熟悉第 10 章介绍的术语，包括卷（volume）和分区（partition）。同时，还要熟悉以下这些附加的术语：

■ 扇区（sector）是存储介质上的可寻址硬件块。X86 系统的硬盘几乎总定义 512 字节大小的扇区。因此，如果操作系统想要修改硬盘上的第 632 个字节，它必须将一个 512 字节数据块写入到磁盘上的第二个扇区。

■ 文件系统格式（file system format）定义了文件数据存储在存储介质上的方式，并且影响文件系统的特征。例如：不允许用户权限同文件和目录相关联的文件格式不支持安全功能。文件系统格式还可以影响文件大小和文件系统支持的存储设备的限制。最后，一些高效的文件系统格式既支持大文件也支持小文件，或者既支持大磁盘也支持小磁盘。

■ 簇是许多文件系统格式使用的寻址块。簇的大小通常为扇区大小的许多倍，如图 12-1 所示。文件系统格式利用簇能更有效地管理硬盘空间，比扇区大的簇把硬盘分成许多更易管理的块。一个较大的簇的潜在弊端是浪费磁盘空间或内部的碎片。产生这样的结果是因为文件的大小不能恰好为簇的大小的倍数。

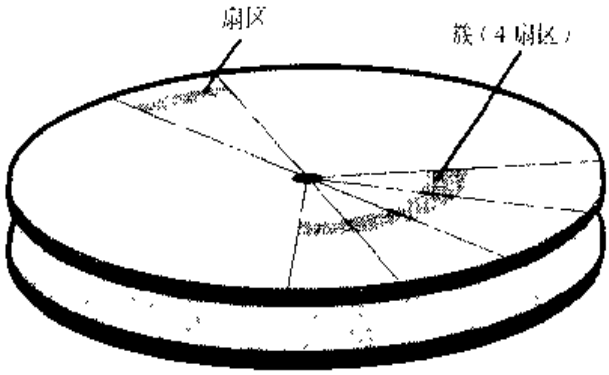


图 12-1 磁盘上的扇区和一个簇

■ 元数据（Metadata）是存储在卷上支持文件系统格式管理的数据。它不能被用来访问应用程序。例如元数据包括定义文件所在位置的数据和卷上的目录。

12.1 Windows 2000 文件系统格式

Windows 2000 支持下列文件系统格式：

■ CDFS

■ UDF

■ FAT12，FAT26，FAT32

■ NTFS

每种格式都适合各自特定的环境，你可以从下面章节中看出。

12.1.1 CDFS

CDFS 或 CD－ROM 文件系统，在 1998 年是作为 CD－ROM 介质的只读标准格式定义的，它是一个相对简单的格式。Windows 2000 在 *\Winnt\Sytem32\Drivers\Cdfs.sys* 中实现了遵从 ISO 9660 标准水平的 CDFS，并且支持 ISO 9660 标准 level 2 所定义的长文件名。由于 CDFS 的简单性，使得 CDFS 格式有许多局限性。

■ 目录和文件名长度必须小于 32 个字符。

■ 目录树的层次不能超过 8 层。

由于工业已经采用通用磁盘格式（UDF）作为只读介质标准，所以 CDFS 被认为是一个遗留格式。

12.1.2 UDF

Windows 2000 UDF 文件系统的实现遵从 ISO 13346 标准，并且支持 UDF 1.03 和 1.5 版本。OSTA（光存储器技术学会）规定：对光磁存储介质，主要是 DVD－ROM，用 1995 年定义的 UDF 格式替代 CDFS，UDF 包含在 DVD 规格说明中，并且比 CDFS 灵活得多。UDF 文件系统具备下列的优点：

■ 文件名可以长达 255 个字符。

■ 最大路径长度为 1023 个字符。

■ 文件名可以是大写和小写字母。

尽管按理说 UDF 格式是为可重写的介质设计的，但 Windows 2000 UDF 驱动程序（*\Winint\System32\Drivers\Udfs.sys*）只支持只读

12.1.3 FAT12、FAT16 和 FAT32

Windows 2000 支持 FAT 文件系统，主要是便于其他版本的 Microsoft Windows 的升级，在多引导系统下保证同其他操作系统的兼容，并且作为一种软盘格式。Windows 2000 FAT 文件系统驱动程序在 *\Winnt\System32\Drivers\Fastfat.sys* 中实现。

每个 FAT 格式包括一个号码，它指定了用于识别磁盘上簇的格式的位数。FAT12 的 12 位簇标识符限定了一个分区最多存储 2^{12}（4096）个簇。Windows 2000 使用的簇大小在容量上为从 512 字节到 8KB，它限制了一个 FAT12 卷大小为 32MB。因此，对所有的可以存储 1.44MB 数据的 $5_{1/4}$ 寸软盘和 3.5 寸软盘，Windows 2000 用 FAT12 作为文件格式。

FAT16 具有 16 位簇标识符，可以访问 2^{15}（65，536）个簇。对于 Windows 2000，FAT16 簇的大小从 512 字节（扇区容量）变化到 64KB，限定 FAT16 卷大小为 4GB。Windows 2000 簇的大小取决于卷的大小。不同的大小列于表 12-1。如果用 format 命令或磁盘管理插件格式化一个小于 16MB 的卷为 FAT 格式，Windows 2000 会使用 FAT12 而不是 FAT16 格式。

表 12-1　Windows 2000 默认的 FAT16 簇大小

卷大小	簇大小
0 – 32 MB	512 字节
33 – 64MB	1KB
65 – 128MB	2KB
129 – 256MB	4KB
257 – 511MB	8KB
512 – 1023MB	16KB
1024 – 2047MB	32KB
2048 – 4095MB	64KB

FAT 卷被分成几个区，见图 12-2。文件分配表给出了 FAT 文件系统格式的名字，对卷里的每一簇都有一个项。要成功地解释卷的内容，文件分配表是非常关键的，因此，FAT 格式包含文件分配表的两个拷贝，这样，如果一个文件系统驱动程序或一致性检验程序（例如，chkdsk）不能访问其中的一个文件分配表（由于坏扇区导致），它可以访问另一个文件分配表。

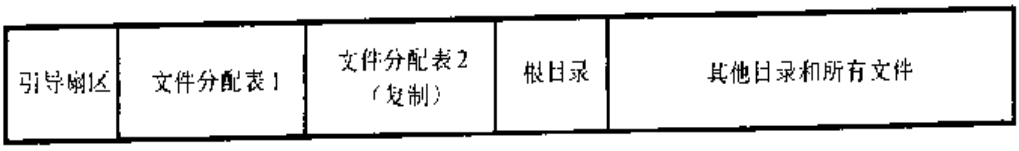


图 12-2　FAT 格式组织结构

文件分配表中的项定义了文件和目录的文件分配链（见图 12-3），链中的链接指示了文件数据的下一个簇。一个文件目录项存储了文件的起始簇，文件分配链的最后一项为保留值，对 FAT16 来说，为 0xFFFF，对 FAT12 来说，为 0xFFF。对没有使用的簇，FAT 项的值为 0。你可以从图 12-3 看到，FILE1 被分配到簇 2、3 和 4。FILE2 使用簇 5、6 和 8。FILE3 使用簇 7。

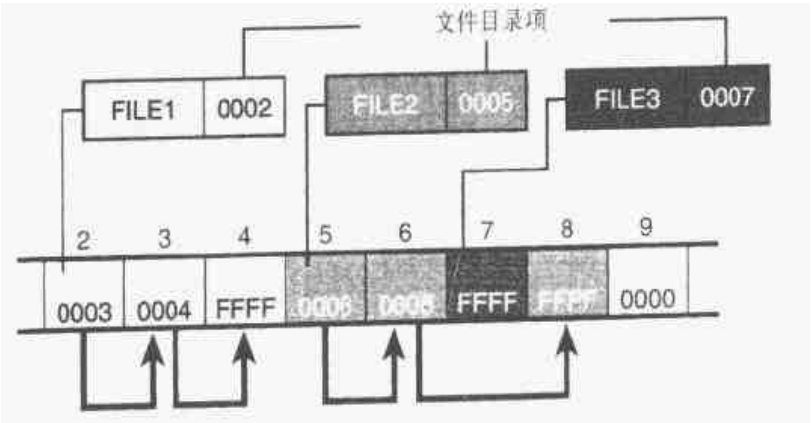


图 12-3　FAT 文件分配链表

FAT12 和 FAT16 卷的根目录在卷的开始预先分配了足够的空间来存储 256 个目录项，它指定了根目录可以存储的文件和目录数的上限（在 FAT32 根目录中，没有预先分配的空间或容量限制）。FAT 目录项是 32 位的，它存储一个文件的名字、大小、起始簇和时间戳（最后访

问、创建等等）信息。如果一个文件名遵循 Unicode 或者不遵循 MS－DOS 8.3 命名规则，那么就分配附加的目录项存储长文件名。附加的目录项先于文件的主项。图 12-4 显示了一个名为“The quick brown fox”的文件的目录项实例。系统已经创建 THEQUI~1.FOX8.3 表示名字（你在目录项中看不到“.”，因为假定它出现在 8 个字符之后）和两个更多的目录项存储 Unicode 长文件名。图中的每行由 16 字节组成。

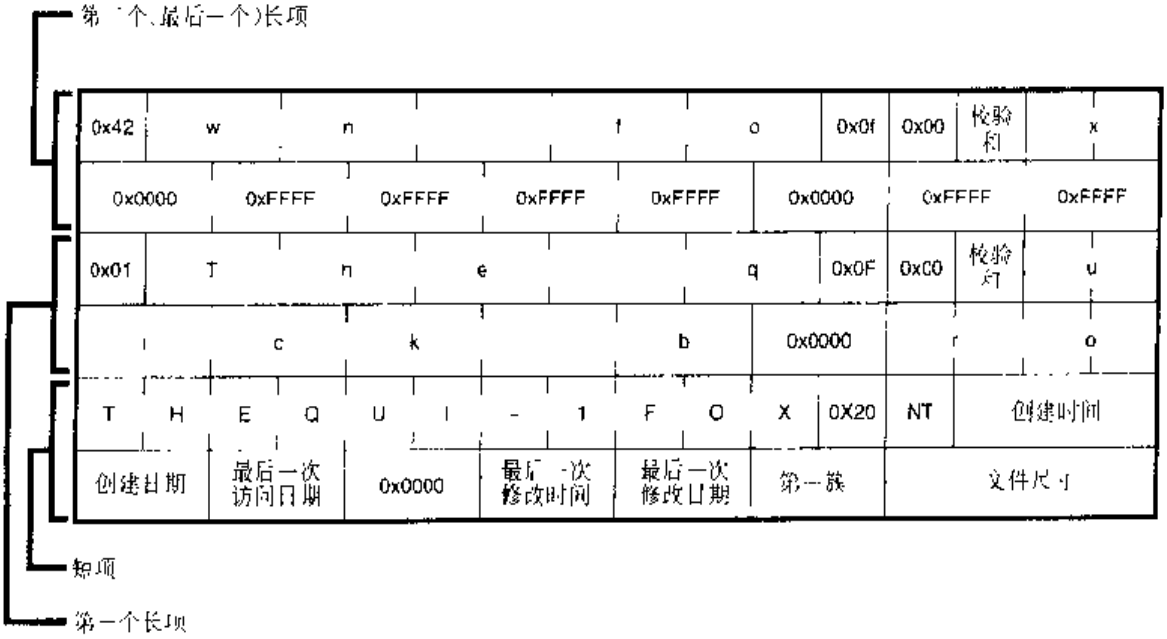


图 12-4 FAT 目录项

FAT32 是基于 FAT 的最新定义的文件系统格式，它用于 Windows 95 OSR2、Windows 98 和 Windows Millennium Edition。FAT32 尽管使用 32 位簇标识符但保留高 4 位，因此，实际上它只有 28 位簇标识符。由于 FAT32 簇大小最大可达 32KB，从理论上讲，FAT32 有能力访问 8 兆兆位（TB）卷。尽管 Windows 2000 支持已存在的更大容量的 FAT32 卷（在其他操作系统中创建），但它限制新的 FAT32 卷最大为 32GB。FAT32 潜在的簇数量更大，这使它可以比 FAT16 更高效地管理磁盘。它用 512 字节簇最多处理 128MB 卷。表 12-2 显示了 FAT32 卷的默认簇大小。

表 12-2 FAT32 卷的缺省簇大小

分区大小	簇大小
32MB 到 8GB	4KB
8GB 到 16GB	8KB
16GB 到 32GB	16KB
32GB	32KB

除了对簇数目有更高限制以外，FAT32 还有超过 FAT12 和 FAT16 的其他优点，包括：不在卷上预先定义的位置存储 FAT32 根目录、根目录在大小上不具上限，而且为安全起见 FAT32 保存了引导扇区的一个拷贝。同 FAT16 一样，FAT32 限制文件大小最大为 4GB，因为目录以 32 位值存储文件大小。

12.1.4 NTFS

正如在文章开头所说的，NTFS 文件系统是 Windows 2000 本机的文件系统格式。NTFS 使用 64 位簇进行索引，这使得 NTFS 有能力寻址达 16 exabytes（160 亿 GB），Windows 2000 限制了用 32 位簇即 128TB（用 64KB 簇）寻址 NTFS 卷的大小。表 12-3 显示了 NTFS 卷默认簇的大小（当格式化 NTFS 卷时，可以超过默认簇的大小）。

表 12-3 NTFS 卷默认簇大小

卷大小	缺省簇大小
512MB 或更少	512 字节
512MB ~ 1024MB（1GB）	1kB
1025MB ~ 2048MB（2GB）	2kB
大于 2048MB	4kB

NTFS 具有许多高级的特征，例如文件和目录的安全性、配额、文件压缩、基于目录的符号链接和加密。其中最具意义的特征是可恢复性。如果一个系统突然中断，FAT 卷中的元数据（*metedata*）就会处于一种不一致的状态，导致大量文件和目录数据的毁损。NTFS 记录以事务方式被转换为元数据，这样文件系统结构就可以修补成一个一致的状态，而不会损失文件或目录结构信息。（然而会丢失文件数据）。

在今后的章节中，将会详细地描述 NTFS 数据结构和高级特征。

12.2 文件系统驱动程序体系结构

文件系统驱动程序（FSD）管理文件系统格式。尽管 NTFS 以内核模式运行，它们在许多方面不同于标准内核模式驱动程序。也许，最有意义的是，它们必须用 I/O 管理器注册成 FSD，而且更加广泛地与内存管理器和高速缓冲管理器进行交互。因此，它们用的是标准驱动程序使用的输出 Ntoskrnl 功能的超集。鉴于为建立标准内核模式驱动程序，需要 Windows 2000 DDK，你必须使用 Windows 2000 安装文件系统（IFS）工具创建文件系统驱动程序（关于 DDK 的更多信息，见第一章，关于 IFS 工具的更多信息，见 www.Microsoft.com/ddk/ifdkit）。

Windows 2000 有两种不同类型的文件系统驱动程序：

- 本机 FSD 管理直接同计算机相连的卷；
- 网络 FSD 允许用户访问同远程计算机相连的数据卷。

12.2.1 本机 FSD

本机 FSD 包括 Ntfs.sys、Fastfat.sys、Udfs.sys、Cdfs.sys 和 RAW FSD（在 Ntoskrnl.exe 中集成）。图 12-5 显示了本机 FSD 如何同 I/O 管理器和存储设备驱动程序进行交互作用的一个简单图。正如我们在 10.5.3 节“卷装配”中所描述的一样，本机 FSD 负责用 I/O 管理器注册，一旦 FSD 注册后，当应用程序或系统开始访问卷时，I/O 管理器就可以调用 FSD 执行卷识别。卷识别涉及到一个卷引导扇区的检测和文件系统元数据的一致性检测。

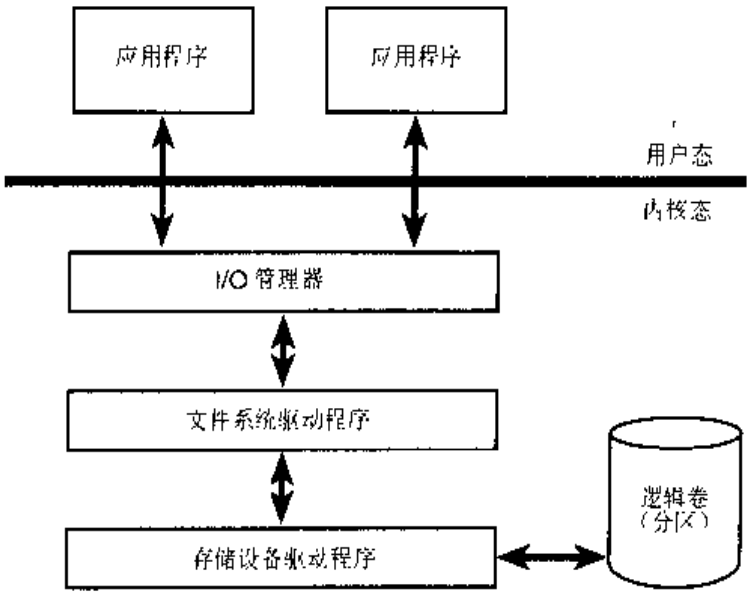


图 12-5 本机 FSD

每个支持 Windows 2000 文件系统格式的第一个扇区被保留作为卷的引导扇区。由于引导扇区包含足够的信息，本机 FSD 既可以识别扇区所在卷上包含 FSD 管理的格式，又可以确定其他对于用来识别卷上元数据的位置来说必要的元数据。

当本机 FSD 识别一个卷时，它创建一个代表已建立的文件系统格式的设备对象。I/O 管理器通过卷参数块（VPB）链接卷设备对象（它由存储设备创建）和 FSD 创建的设备对象。VPB 的链接导致了 I/O 管理器把对卷设备对象的 I/O 重定向为对 FSD 设备对象的 I/O 请求（关于 VPB 的更多信息见第 10 章）。

为提高性能，本机 FSD 通常使用高速缓存管理器高速缓存文件系统数据（包括元文件）。FSD 还和内存管理器结合起来，以使映射文件能被正确地实现。例如，当应用程序试图截断一个文件时，为了检验没有超出截断点映射文件必须查询内存管理器，Windows 2000 不允许通过截断或文件删除命令来删除被应用程序映射的文件数据。

本机 FSD 也支持文件系统卸载（dismount）操作，它允许系统断开 FSD 和卷对象的链接。当应用程序对磁盘卷上的内容进行原始访问（raw access）或者和卷相联系的介质改变时，卸载就会发生。卸载后，应用程序第一次访问介质，I/O 管理器重新初始化介质的卷的装配操作。

12.2.2 远程 FSD

远程 FSD 包括两个组件：客户机和服务器。客户机端远程 FSD 允许应用程序访问远程的文件和目录。客户机远程 FSD 从应用程序接受 I/O 请求，并把它们翻译成网络文件系统协议命令，FSD 通过网络将协议命令发送到服务器端远程 FSD。服务器端 FSD 倾听来自网络连接的命令，并通过向本机 FSD 发送 I/O 请求来实现，而该本机 FSD 管理这些命令的目标文件和目录驻留的卷。图 12-6 显示了同远程 FSD 交互作用的客户机端和服务器端之间的关系。

Windows 2000 包含命名为“LANMan 重定向器”的客户机端 FSD 和命名为“LANMan 服务器”的服务器端远程 FSD 服务器。重定向器作为端口/小型端口组合实现，端口驱动程序

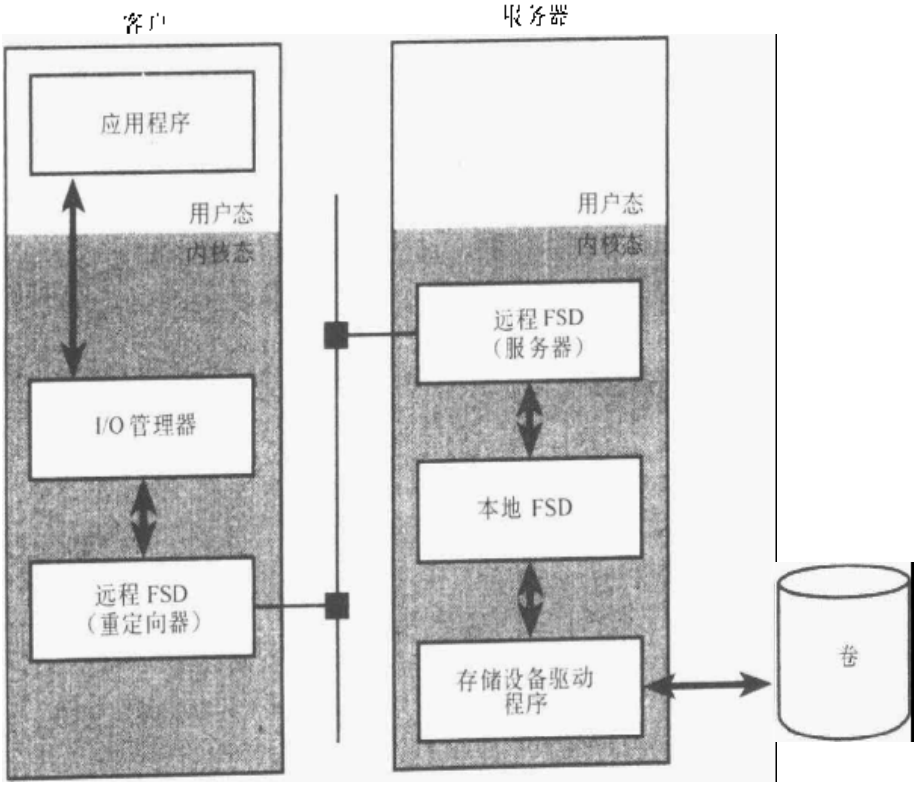


图 12-6 远程 FSD 操作

(*\Winnt\System32\Drivers\Rdbss.sys*) 作为驱动程序子例程库实现，小型端口 (*\Winnt\System32\Drivers\Mrxsmb.sys*) 通过端口驱动程序使用服务。端口/小型端口模式简化了重定向器的发展，由于端口驱动程序被所有的远程小型端口驱动程序共享，它能处理许多涉及到 Windows2000 I/O 管理器的客户机端远程 FSD 接口的细节问题。除了 FSD 组件，LANMan 重定向器和 LANMan 服务器各自包括命名为 "Workstation" 和 "Server" 的 Win32 服务程序。

Windows 2000 依赖于 Common Internet File System (CIFS) 协议格式化重定向器和服务器之间的交换信息。CIFS 是 Microsoft 服务器信息块 (SMB) 协议的改进版本（关于 CFIS 更多信息，见网站 www.cifs.com.）。

象本机 FSD 一样，客户机端远程 FSD 对属于远程文件和目录的本机高速缓存文件数据使用高速缓存服务。然而，客户机端远程 FSD 必须实现分布式高速缓存一致协议，称为 Oplocks (机会主义锁定)，这样，当 FSD 访问远程文件时应用程序所见的数据和访问相同文件的应用程序在其他计算机上运行时所见的数据相同。尽管服务器端 FSD 参与维护客户机之间的高速缓冲一致，由于本机 FSD 高速缓存它们自己的数据，服务器端 FSD 不高速缓存本机 FSD 的数据 (Oplocks，重定向器和服务器将在第 13 章 "通用互联网文件系统 (CIFS)" 进行更详细地描述)。

注意 置于文件系统驱动程序上的过滤器驱动程序称为文件系统过滤器驱动程序。它具有可以看见所有文件系统请求并有选择地修改或完成这些请求的能力。这种能力使得一批应用程序得以执行，包括正在访问的病毒扫描程序和远程文件复制服务程序。在配套 CD 盘上\Sysint\Filemon 上的 Filemon，是通过 (pass-through) 过滤器的文件系统过滤器驱动程序的实例。Filemon 显示了文件系统的实时活动，它没有更改它所见到的请求。

实验：显示文件系统注册表列表

当 I/O 管理器向内存加载设备驱动程序时，设备驱动程序一般命名它创建的驱动程序对象来代表驱动程序，这样，设备驱动程序就位于设备对象管理器的目录下。I/O 管理器加载的任意驱动程序的设备对象都有一个位于*FileSystem* 目录下的 Type 属性值 SERVICE – FILE – SYSTEM – DRIVER（2）。因此，使用象 Winobj（CD）这样的工具，就可以看到注册到系统的文件系统，正如图 12-7 所示。（注意一些文件系统也将设备对象置于*File System* 目录下）。

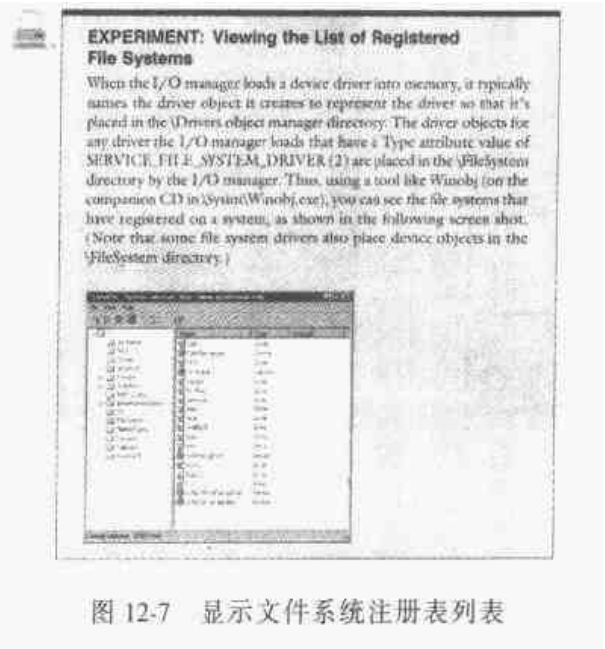


图 12-7　显示文件系统注册表列表

12.2.3　文件系统操作

应用程序和系统以两种方式访问文件：直接地通过文件 I/O 功能实现（如 ReadFile 和 WriteFile）和间接地通过读或写代表映射文件区域的地址空间的一部分。（关于映射文件的更多信息见第 7 章）。图 12-8 为一个简化的图表，它显示了涉及到这些文件系统操作的组件和这些组件相互作用的方式。正如你所看到的，FSD 可以通过下面路径调用：

- 从用户或系统线程执行显式文件 I/O 调用。
- 从内存管理器的修改页面的书写器（modified page writer）调用。
- 间接地从高速缓冲管理器的延迟书写器（lazy writer）调用。
- 间接地从高速缓冲管理器的预读线程调用。
- 从内存管理器的页面错误处理程序调用。

下面部分将描述每种情况下周围的环境和 FSD 对每种情况的响应所采取的步骤。

1. 显式文件 I/O

应用程序访问文件最普遍的方式是通过调用 Win32 I/O 函数例如 CreatFile、Readfile 和 Writefile。应用程序用 CreateFile 打开一个文件，然后通过将从 CreateFile 返回的句柄传递给其他 Win32函数来读、写和删除文件。CreatFile 函数在 Kernel32.dll 和 Win32 客户机端 DLL 中实现。它调用本机函数 NtCreatFile 对应用程序传送的路径形成一个相对根目录的完整路径名。

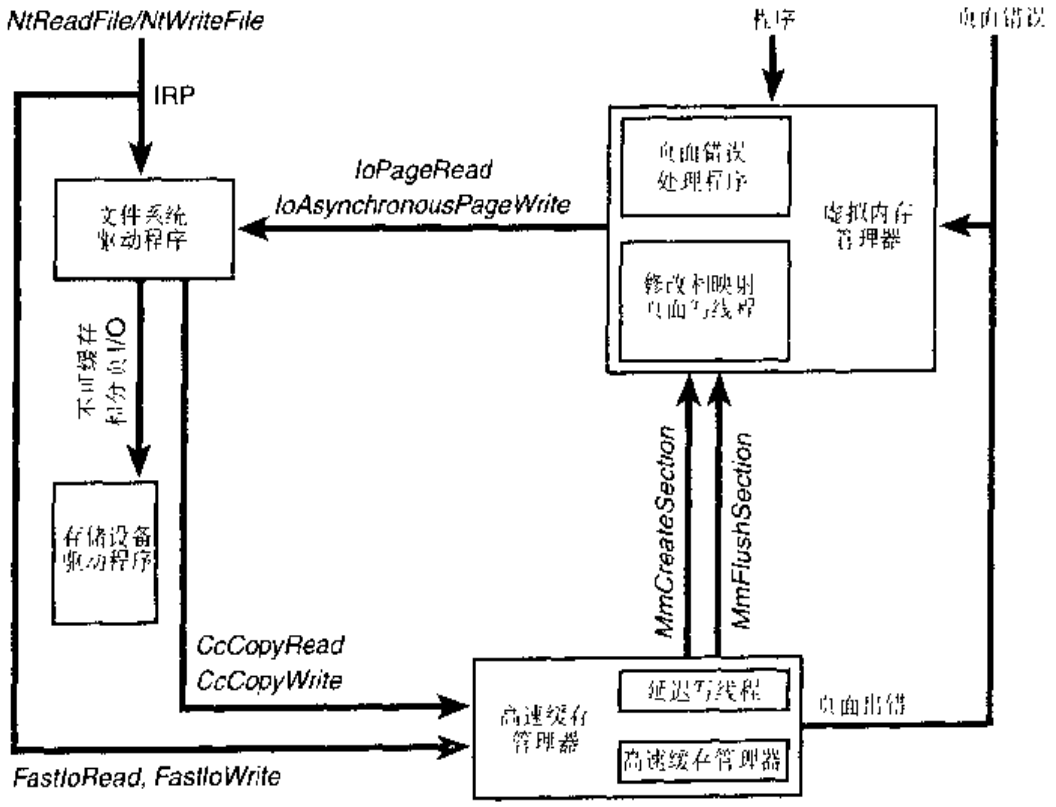


图 12-8 文件系统 I/O 涉及的组件

（“.”和“..”指路径名的符号）在路径前面加上“\??”（例如，\?? \C:\Susan\Todo.txt）。

NtCreatFile 系统服务用 ObOpenByName 打开文件，文件从对象管理器根目录和路径名的第一部分（“??”）开始解析文件名。\?? 是包含着代表被分配驱动器符的卷的符号链接的子目录（以及串行口和其他 Win32 应用程序直接访问的符号链接）。这样名字的“C:”部分被映射到 \?? \C: 符号链接。?? 符号链接指向 \Device 下的卷设备对象，所以当对象管理器遇到卷对象时，对象管理器把其他路径名交给 I/O 管理器为设备对象注册的解析函数，IopParseDevice。（在动态磁盘卷上，符号链接指向一个中间的符号链接，中间的符号链接指向卷的设备对象）。图 12-9 显示了卷对象是如何通过对象管理器名字空间被访问的，它显示了符号链接 \?? \C: 如何指向 \Device\HarddiskVolume1 的卷设备对象。

当锁定调用程序的安全环境和从调用程序令牌获得安全信息以后，IopParseDevice 建立了一个 IRP_MJ_CREATE 类型的 I/O 请求包（IRP），它还建立了一个文件对象，用于存储已打开文件的名字，它跟踪卷设备对象的 VPB 去查找卷的装配文件系统设备对象，并利用 IoCallDriver 将 IRP 传递给拥有文件系统设备对象的文件系统驱动程序。

当 FSD 接收到 IRP_MJ_CREATEIRP，它查询指定的文件，执行安全确认，如果文件存在并且允许用户以它请求的方式访问文件，FSD 就会返回成功的代码。对象管理器在进程的句柄表中为文件对象建立一个句柄，这个句柄通过调用链再传递回来，最后它作为一个 CreatFile 的返回参数到达应用程序。如果文件系统不能创建句柄，I/O 管理器就删除为它创建的文件对象。

我们跳过了关于 FSD 如何定位在卷上打开的文件的细节，然而 ReadFile 函数调用操作共享了

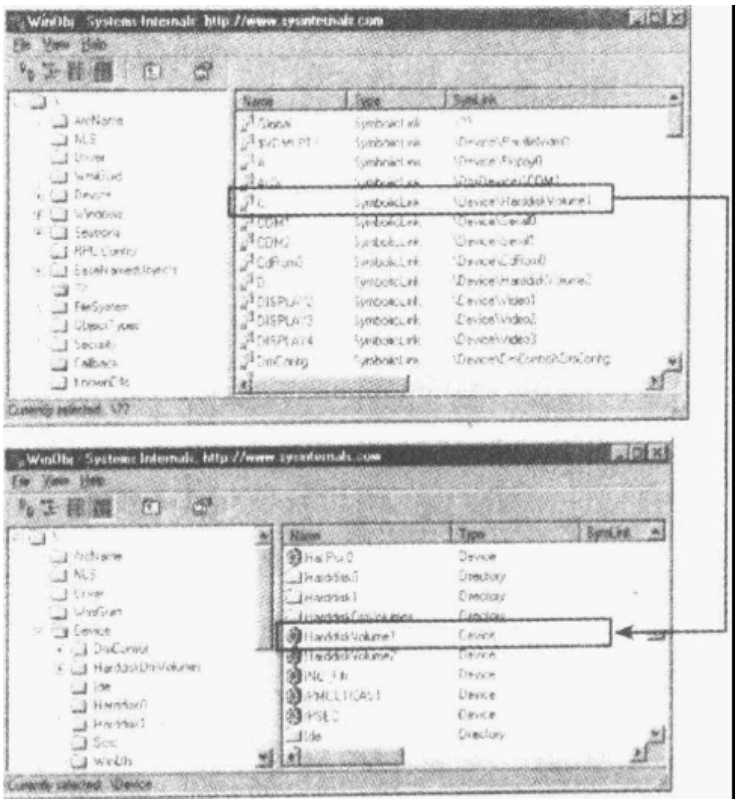


图 12-9 驱动器符名字映射

许多 FSD 同高速缓冲管理器和存储管理器的交互。尽管调用 ReadFile 进入内核的路径同调用 CreatFile 产生的结果是一样的，NtReadfile 系统服务不需要执行名字查询——它调用对象管理器将从 ReadFile 传递过来的句柄转变成文件对象的指针。如果句柄指示，当文件被打开时允许调用程序读文件，NtReadFile 就继续建立 IRP－MJ－READ 类型 IRP，并把它发送到文件所在的 FSD。NtReadfile 获得存储在文件对象的 FSD 的设备对象，调用 IoCallDriver，I/O 管理器根据设备对象确定 FSD 的位置，并把 IRP 传送给 FSD。

如果已读的文件能被高速缓存（当文件打开时，FILE－FLAG－BUFFERING 标记没有传递给 CreatFile），FSD 会检验文件对象的高速缓存是否已经初始化。如果文件对象的高速缓存被初始化，文件对象中的 PrivateCacheMap 字段就会指向专用的高速缓存映射数据结构（在第 11 章描述的），如果 FSD 没有初始化文件对象的高速缓存，PrivateCacheMap 字段将为空。如果引用相同文件的另一个文件对象没有初始化高速缓存，FSD 将调用高速缓存管理器 CcInitialize-CacheMap 函数来初始化高速缓存，这涉及到创建专用高速缓存映射的高速缓存管理器。如果另一个文件对象引用还没有初始化高速缓存的相同文件，还会涉及到共享高速缓存的映射和区域对象。

如果经验证，文件允许高速缓存，FSD 就从高速缓存管理器的虚拟内存拷贝请求的文件数据到 ReadFile 函数传递给线程的缓冲区。文件系统在一个 try/except 块内执行拷贝，这样它就可以捕捉到由于无效应用缓冲区带来的错误。文件系统用来执行拷贝的函数是高速缓存管理器的 CcCopyRead 函数。CcCopyRead 采用文件对象，文件偏移量和长度作为参数。

当高速缓存管理器执行 CcCopyRead 时，CcCopyRead 获取存储在文件对象中指向共享高速

缓存映射的指针。回顾第11章讲过，共享高速缓存映射存储指向虚拟地址控制块（VACB）的指针，文件的每256KB块用一个VACB项。如果已读部分文件的指针是空的，CcCopyRead就分配一个VACB，在高速缓存管理器的虚拟地址空间保留256KB视图，将文件指定部分映射到视图，然后，CcCopyRead简单地拷贝映射视图的文件数据到缓冲区（缓冲区最初传递到readfile）。如果文件数据不保存在物理内存，复制操作会产生页面出错，由MmAccessFault服务。

当发生页面出错时，MmAccessFault检查引起出错的虚拟地址和确定虚拟地址描述符在引起出错的进程VAD树中的位置（关于VAD树的更多信息见第7章）。在这种方案中，VAD描述了高速缓存管理器的正在读文件的映射视图，于是，MmAccessFault调用MiDispatchFault处理有效的虚拟内存地址上的页面错误。MmAccessFault确定控制区的位置（VAD指向的）并且通过控制区寻找代表打开文件的文件对象（如果文件打开不只一次，在专用的高速缓存映射中一定存在通过指针链接的文件对象的列表）。

利用在手头的文件对象，MiDispatchFault调用I/O管理器IoPageRead函数建立IRP（IRP－MJ－READ类型），并将IRP发送到FSD，FSD拥有文件对象指向的设备对象。这样，文件系统可以重入读取通过CcCopyRead请求的数据。但是，这时IRP被标志为非高速缓存的和调页I/O。这些标志指示FSD应该直接从磁盘获取文件数据，FSD这样做是通过确定在磁盘上的哪些簇包含请求数据并将IRP发送到拥有文件驻留的卷设备对象的卷管理器。在FSD的设备对象的卷参数块（VPB）字段指向卷设备对象。

虚拟内存管理器等待FSD完成IRP的读入，然后将控制权返回到高速缓存管理器，高速缓存管理器继续拷贝被页面出错中断的操作。当CcCopyRead完成时，FSD在高速缓存管理器和虚拟内存管理器的帮助下，已经将请求文件数据拷贝到线程缓冲区，并将控制权返回到称为"NtReadFile"的线程。

除了NtWriteFile文件服务系统产生一个IRP－MJ－WRITE类型的IRP和FSD调用CcCopyWrite而不是CcCopyRead以外，WriteFile的路径是相似的。像CcCopyRead一样，CcCopyWrite保证正在被写入的文件的一部分被映射到高速缓存，然后把缓冲区传递到WriteFile的内容拷贝到高速缓存。

如果文件数据已经保存在系统的工作集中，那么，刚刚描述的情况中有许多变化。如果文件数据已经保存在高速缓存，CcCopyRead就不会导致页面出错。在一定条件下，NtWriteFile和NtReadFile调用一个FSD的快速I/O入口点而不是立即建立和发送IRP到FSD。下面是遵循的一些条件：读入的文件部分必须驻留在文件的第一个4GB、文件可以不锁定、读入或写入的文件部分必须在当前分配的文件大小范围内。

对于大多数FSD，快速I/O读写入口点调用高速缓存管理器的CcFastCopyRead和CcFastCopyWrite函数。这些标准拷贝例程的变种保证在执行拷贝操作之前文件数据被映射到文件系统高速缓存管理器。如果不满足这些条件，CcFastCopyRead和CcFastCopyWrite显示执行快速I/O是不可能的。当不可能执行快速I/O时，NtWriteFile和NtReadFile就会求助于创建一个IRP（关于快速I/O更多完整的描述见第11章）。

2. 内存管理器的修改和映射页面的书写器

内存管理器的修改和映射页面的书写器周期性地刷新修改的页面。线程调用IoASynchro-

nousPageWrite 创建 IRP - MJ - WRITE 类型的 IRP，并且把页面写入一个调页文件或一个映射后被修改的文件。象 MiDispatchFault 创建的 IRP 一样，这些 IRP 被标记为非高速缓存的和调页 I/O 这样 FSD 忽略文件系统高速缓存管理器并且直接向存储驱动程序发送 IRP 以把内存写入磁盘。

3. 高速缓存管理器的延迟书写器

高速缓存管理器延迟书写器在写入修改页时也发挥作用，因为它定期刷新映射到高速缓存它知道是肮文件扇区的视图。高速缓存管理器通过调用 MmFlushSection 执行的刷新操作，触发内存管理器写入正在刷新到磁盘上的区域部分的任何修改页。像修改和映射页面书写器一样，MmFlushSection 使用 IoAsynchronousPageWrite 函数将数据发送到 FSD。

4. 高速缓存管理器预读线程

高速缓存管理器包括一个负责在应用程序、驱动程序或系统线程显示地发出请求前试图从文件读入数据的线程。预读线程利用在文件中被执行的读操作的历史决定要读多少数据，历史被保存在文件对象的专用高速缓存映射中。当线程执行预读时，它简单地映射了它想要读入高速缓存的文件的一部分（必要时分配 VAB）和处理映射的数据。内存访问导致的页面错误调用页面错误处理程序将页面读入系统的工作集。

5. 内存管理器的页面错误处理程序

我们在显式文件 I/O 和内存管理器预读部分描述了如何使用页面出错处理程序，当任意应用程序访问映射文件视图的虚拟内存和遇到代表文件部分的页面不属于应用程序工作集时，应用程序也会调用页面出错处理程序。当内存管理器从 CcCopyRead 或 CcCopyWrite 产生一个页面出错时，内存管理器的 MmAccessFault 处理程序遵循同样的步骤，并通过 IoPageRead 将 IRP 发送到文件被保存的文件系统。

12.3 NTFS 设计目标和特性

在下面的部分，将看看驱动 NTFS 设计的需求。然后将检查 NTFS 的高级特性。

12.3.1 高端文件系统需求

从开始，NTFS 就被定义为包含要求有企业类文件系统的特征。为了减小突发的系统断电或崩溃带来的数据损失，文件系统必须保证始终保护文件系统的元数据的完整性，并且保护未经许可访问的敏感数据，同时，文件系统必须有一个集成安全模式。最后，为保护用户数据，文件系统必须允许以基于软件的数据冗余作为硬件冗余的解决方案的一种低成本选择。在这部分，你会弄清楚 NTFS 是如何实现每种能力的。

1. 可恢复性

为满足可靠数据存储和数据存取的需求，NTFS 提供了基于原子事务（atomic transaction）概念的文件系统恢复。原子事务是一种处理数据库修改的技术，这样系统故障不会影响数据库的正确性和完整性。原子事务的基本原则是一些被称为“事务”的数据库操作是要么全有要么全无的命题。（一个事务被定义为改变文件系统数据或改变卷的目录结构的 I/O 操作）。组成事务的各自的磁盘更新必须被原子地执行，也就是说，一旦开始执行事务，所有的磁盘更新必须

被完成。如果系统故障中断了事务，已经完成的部分必须撤消，或返回。返回操作将数据库返回到先前已知的一致性状态，好象事务动作从未发生一样。

NTFS利用原子事务执行文件系统的可恢复特性，如果一个程序执行了改变NTFS驱动程序结构的I/O操作，如改变目录结构、扩展一个文件、为一个新文件分配空间等等，NTFS视为这样的操作为原子事务。它保证事务要么被完成，要么在执行事务出现系统故障时候被返回。关于NTFS如何操作的细节在"NTFS可恢复支持"部分有说明。

另外，NTFS对关键的文件系统信息使用冗余的存储。这样，如果磁盘上的扇区坏了，NTFS仍能访问卷的关键的文件系统数据。文件系统数据的冗余同磁盘结构的FAT文件系统和HPFS文件系统（OS/2本机文件系统格式）完全不同，后者用单个扇区存储关键的文件系统数据，在这样文件系统下，如果扇区中的一个发生读错误，整个卷就会丢失。

2. 安全性

NTFS的安全直接源于Windows 2000的对象模型。文件和目录防止未经授权的用户访问（关于Windows 2000安全性的更多信息，见第8章）。一个打开的文件作为一个具有安全描述符的文件对象实现。安全描述符作为的文件一部分保存在磁盘上。在进程打开任意一个对象的句柄前，包括文件对象，Windows 2000安全系统要检测进程是否有这样做的权限。安全描述符与用户登录到系统并提供识别密码的请求相结合来确保没有进程可以访问文件，除非由系统管理员或文件的拥有者给予特定许可（关于安全描述符的更多信息，见第8章"安全描述符和访问控制"部分，关于文件对象的更多细节，见第9章"文件对象"部分）。

3. 数据冗余度和容错性

除了文件系统数据的可恢复性，一些用户要求他们自己的数据在断电或磁盘发生灾难性故障时不会遭到破坏。NTFS的可恢复能力确实可以保证卷上的文件系统仍旧可以访问，但它们不保证用户文件的完全恢复。对不能冒险丢失文件数据的应用程序的保护是通过数据冗余度来实现的。

用户文件的数据冗余度是通过Windows 2000分层驱动模式（在第9章说明）实现的。它提供了磁盘容错支持。NTFS同卷管理器进行通信，卷管理器又同硬盘驱动程序进行通信并将数据写入磁盘。卷管理器可以镜像（mirror），或者从一个磁盘复制数据到另一个磁盘，这样总能获得冗余的复制。这种支持通常被称为RAID level 1。卷管理器也允许数据以带区形式跨过三个或更多的磁盘写入，相当于用一个磁盘来维护奇偶检验信息。如果一个磁盘的数据丢失或变成不能被访问，驱动程序可以通过五斥OR操作来重建磁盘的内容。这种支持被称为RAID level 5（关于带区卷、镜像卷和RAD－5卷的更多信息见第10章）。

12.3.2 NTFS的高级特性

NTFS除了具有可恢复性、安全性、可靠性和对任务关键系统的有效性以外，它还包括下面的高级特征，这些高级特征允许NTFS支持更广范围的应用程序。其中的一些特征被做为API应用程序利用，另一些特征是内部特征。

- 多重数据流
- 基于Unicode命名

- 通用检索功能
- 动态坏簇重映射
- 硬链接和 junction
- 压缩和稀疏文件
- 改变日志
- 每个用户卷配额
- 链接跟踪
- 加密
- 支持 POSIX
- 碎片整理

下面部分将提供这些特征的全貌。

1. 多重数据流

在 NTFS，同文件相关联的信息的每个单元包括文件的名字、所有者、时间戳（time stamp）、内容等等，被作为文件属性（NTFS 文件对象属性）来实现。每个属性包括一个单一流，也就是说，一个简单的字节序列。这类操作使得容易向文件添加更多的属性（更多流）。因为文件的数据仅仅是文件的“另一个属性”并且因为新属性可以被添加，所以 NTFS 文件（和文件目录）可以包含许多数据流。

NTFS 文件有一个默认数据流，它没有名字。应用程序能创建另外的命名数据流，并且通过引用它们的名字来访问它们。为了避免改变 Microsoft Win32 API，数据流的名字通过在文件名后添加冒号来指定。Microsoft Win32 API 采用字符串作为文件名参数。因为冒号是保留字符，它可以作为文件名和数据流名之间的分隔符，举例说明如下：

myfile.dat：stream2

每个流有各自的分配大小（为它保留了多少磁盘空间），实际的大小（调用程序使用了多少字节）和有效的数据长度（多少数据流被初始化）。另外，每个流被指定一个单独的文件锁定，用来锁定字节范围，并且每个流允许并行访问。

Windows 2000 使用多重数据流的一个组件是 Apple Macintosh 文件服务器支持，这个支持来自 Windows 2000 服务器。Macintosh 系统的每个文件使用两个流——个用来存储数据，另一个用来存储资源信息，如文件类型和用来表示文件的图标。因为 NTFS 允许多重数据流，一个 Macintosh 用户可以将整个 Macintosh 文件夹复制到 Windows 2000 服务器，另一个 Macintosh 用户可以从这个服务器复制文件夹而不会丢失资源信息。

Windows Explorer 是另一个使用流的应用程序。当你右击一个 NTFS 文件，选择 Propertites，结果对话框的 Summary 项使你将信息同文件关联，例如标题、对象、作者和关键字。Windows Explorer 在它添加的文件的可替换流里保存信息，取名为“汇总信息（Summary Information）。”

其他应用程序也可以用多重数据流特征。例如 backup 功能，可以使用额外的数据流存储文件上的备份相关的时间戳（time stamp）；或者存档实用程序可以实现分级存储，因超过一定时间的或者在指定的一段时期内没被访问的文件都被转移到磁带。这种功能可以将文件拷贝到磁带、设置文件默认数据流为 0，并且添加一个指定了文件的名字以及文件保存在磁带上位置

的数据流，

实验：观察流

大多数 Windows 2000 应用程序被设计得无法和可替换的命名数据流工作，但是 echo 和 more 命令却可以，这样，一个简单的查看流的活动的方法是用 echo 创建一个命名流，然后用 more 显示。下面的命令系列用一个命名流的流创建了一个名为 text 的文件

```
C:\>echo hello > test:stream
C:\>more < test:stream
hello
C:\>
```

如果你执行一个目录列表，test 的文件大小不反映保存在交替流的数据因为 NTFS 只返回文件查询操作的无名的数据流大小，包括目录列表。

```
C:\>dir test
 Volume in drive C is WINDOWS
 Volume Serial Number is 3991-3040

 Directory of C:\

08/01/00  02:37p                 0 test
               1 File(s)              0 bytes
                            112,559,080 bytes free
```

2. 基于 Unicode 命名

就像 Windows 2000 作为一个整体一样，NTFS 完全允许统一的字符编码标准，利用统一的字符编码标准的字符存储文件名、目录和卷。Unicode 是 16 位字符编码格式，它允许世界上主要语言的每种字符被唯一表示，这样，有助于数据从一个国家向另一个国家的转移。Unicode 是对传统的国际字符表示的改进——使用双字节编码格式，一些字符用 8 位，其他的用 16 位，这种技术需要加载不同的代码页去建立可以获得的字符。因为 Unicode 对每一个字符的表示是唯一的，它不依赖加载的代码页。路径上的每个目录和文件名字符最多可达 255 个字符，可包含 Unicode 字符、嵌入空格和多个点

3. 通用索引功能

NTFS 结构被设计为允许在磁盘卷上索引文件属性。这种结构使得文件系统能有效地确定匹配某种标准的文件的位置——如，在特定的目录下的所有的文件。FAT 文件系统索引文件，但不对它们进行分类，在大的目录下查询较慢

一些 NTFS 特征利用通用索引，包括固结的安全描述符，删除了复制的文件。在这种情况下，卷的文件和目录的安全描述符被存储在一个单一的内部流里，并且一些 NTFS 特征通过 NTFS 定义的内部安全标识符索引。

4. 坏簇动态重映射

通常，如果一个程序设法从一个坏的磁盘扇区读取数据，读操作不能被执行，而且被分配

的簇的数据变成不能访问的。如果磁盘被格式化为一个容错 NTFS 卷格式，Windows 2000 容错驱动程序动态索引存储在坏扇区的数据的正确拷贝，然后给 NTFS 发出扇区是坏的警告。NTFS 分配一个新簇代替坏扇区所在的簇，并将数据拷贝到新簇。NTFS 标记出坏簇以后不再使用它。这种数据可恢复性和坏簇动态重映射对于文件服务器和容错系统或任何不能承担丢失数据的应用程序来说是尤其有用的特征。当扇区变坏时，如果没有加载卷管理器，NTFS 仍旧替换坏簇并且不再使用它，但 NTFS 不能恢复在坏扇区里的数据。

5. 硬链接和 junction

硬链接允许指向同一个文件和目录的多重路径。如果你创建一个硬链接命名为 *C:\Users\Documents\Spec.doc*，它指向已存在的文件 *C:\My Documents\Spec.doc*，两个路径链接同一个磁盘文件，你可以用任何一个路径改变文件。进程利用 Win32 的 CreateHardLink 函数或者 ln POSIX 函数创建硬链接。

实验：创建一个硬链接

尽管应用程序能用 Win32 CreateHardLink 函数创建一个硬链接，却没有工具使用这个函数。但是你可以用 Windows 2000 资源软件包中的 POSIX ln 实用工具创建硬链接。因为 POSIX ln 工具不能通过资源软件包的安装程序安装，所以你要从资源软件包 CD 中 *\Apps\Posix* 目录手工拷贝它们。

除了硬链接，NTFS 还支持另一种称为“junction”的重定向类型，又称为符号链接，它允许目录重定向文件或目录路径名变换到可替换的目录。例如，路径 *C:\Drivers* 是一个重定向到 *C:\Winnt\System32\Drivers* 的 junction，一个读取 *C:\Drivers\Ntfs.sys* 的应用程序实际上读取的是 *C:\Winnt\System32\Drivers\Ntfs.sys*。*junction* 对于提升位于目录树深层的目录到一个更便利的深度而又不影响原始树的结构来说，是很有用方法。当 Ntfs.sys 通过 Junction 访问时，刚才引用的例子提升驱动程序目录到卷的根目录，并将 Ntfs.sys 目录层深度从 3 层降为 1 层。你不能使用 Junctions 链接远程的目录—只能链接本机卷的目录。

Junctions 基于被称为重分析点（reparse points）的 NFTS 机制（重分析点将在本章后面“重分析点”部分进一步讨论）。重分析点是有着称为“重分析数据”（reparse data）的一块数据的文件或目录。重分析数据是用户定义的关于文件或目录的数据，例如文件或目录的状态或位置，它们可以通过创建数据的应用程序、文件系统过滤器驱动程序或 I/O 管理器从重分析点读取。在文件或目录查询期间，当 NTFS 遇到一个重分析点时，它返回重分析状态代码（reparse status code），它向附接到卷文件系统过滤驱动程序和 I/O 管理器发送信号，以检查重分析数据。每类重分析点有唯一的重分析标记（reparse tag）。重分析标记允许负责解释重分析点的重分析数据的组件在不检查重分析数据的情况下去识别重分析点。重分析标记的所有者，文件系统过滤驱动程序或 I/O 管理器，在识别重分析数据时，可以选择下面的选项之一。

- 重分析标记所有者能处理跨过重分析点的在文件 I/O 操作中指定的路径名，并让 I/O 操作用改变了的路径名进行重发送。例如，junctions 用这种方法重定向一个目录查询。
- 重分析标记所有者能从文件删除重分析点，用某种方法更改文件，然后重新发送文件 I/O

操作。Windows 2000 分级存储管理（HSM）系统以这种方法使用重分析点。HSM 通过将文件的内容转移到磁带存储，而将重分析点留在文件的位置。当进程访问一个已经被存档的文件时，HSM 过滤驱动程序（*\Winnt\System32\Drivers\Rsfilter.sys*）从文件删除重分析点，从存档介质读取文件数据，并重新访问。这样，对于进程访问存档文件来说，脱机数据的检索是透明的。

因为不存在建立重分析点的 Win32 函数，所以，进程必须用 Win32 DeviceIoControl 函数来使用 FSCTL - SET - REPARSE - POINT 文件系统控制代码。一种方法可以用 FSCTL - GET - REPARSE - POINT 文件系统控制代码来查询一个重分析点的内容。FILE - ATTRIBUTE - REPARSE - POINT 标志被设置在重分析点的文件属性里，这样，应用程序可以通过用 GetFileAttributes 函数检验重分析点

实验：创建一个 junction

Windows 2000 不包括创建 junctions 的任何工具，但是你既可用配套 CD（*\Sysint\Junction.exe*）上的 junction 工具也可用 Windows 2000 资源软件包工具链接来创建一个 junction。链接工具允许你查看已存在的 junctions 的定义，而且 junction 允许你查看关于 junction 的信息和其他重分析点标记。

6. 压缩和稀疏文件

NTFS 支持文件数据的压缩。因为 NTFS 执行压缩和解压缩过程是透明的，所以在利用这个特性时无须修改应用。目录也能被压缩，这意味着后来在目录下创建的任何文件都被压缩了

应用程序通过传递 FSCTL - SET - COMPRESSION 文件系统控制代码到 DeviceIoControl 来对文件进行压缩和解压缩。用 FSCTL - GET - COMPRESSION 文件系统控制代码查询文件或目录的压缩状态。被压缩的文件或目录在它属性设置中有标志，这样应用程序也可以用 GetFileAttribute 确定文件或目录的压缩状态。

第二种压缩类型被认为是稀疏文件（sparse file）。如果一个文件被标记为稀疏，NTFS 在卷上不为应用程序指定为空的文件的那部分分配空间。当应用程序读取稀疏文件的空区时，NTFS 返回充满 0 的缓冲区。这种类型压缩对于实现循环（缓冲登录（circular - buffer logging）的客户机/服务器应用程序是很有利的。循环 - 缓冲登录时，服务器记录文件的信息而客户机异步地读信息。因为服务器写入的信息在客户机读取后不再有用，所以没必要将信息存储在文件上。通过把文件变成稀疏的，客户机可以指定读取的文件的一部分为空，释放卷上的空间。服务器可继续向文件添加新的信息，而不必担心文件会大得占用卷上所有可用的空间。

对于压缩文件，NTFS 透明地管理稀疏文件。应用程序通过将 FSCTL - SET - COMPRESSION 文件系统控制代码传递到 DeviceIoControl 来指定一个文件的稀疏状态。应用程序用 FSCTL - SET - ZERO - DATA 代码设置文件为空的范围。应用程序可以通过控制代码 FSCTL - QUERY - ALLOCATED - RANGES 向 NTFS 请求文件哪部分为稀疏的描述。稀疏文件的一个应用范例是 NTFS 更改日志（change journal），将在下面描述。

7. 更改日志（change journal）

许多类型的应用程序需要监控文件和目录的改变。例如，一个自动备份程序可以执行一个

初始的完全备份，然后基于文件的改变增加备份。对一个应用程序监控卷的变化的一个简单的方式就是扫描卷、记录文件和目录的状态和随后的扫描探测差别。但是，尤其在有几千或几万个文件的计算机中，这个过程反而会影响系统的性能。

对应用程序来说，一个可替换的方法是用 Win32 函数 FindFirstChangeNotification 或 ReadDirectoryChanges 注册目录通知。作为输入参数，应用程序指定它想要监控的目录名，一旦目录内容改变时，就返回函数。尽管这种方法比卷扫描更有效，但它要求应用程序一直在运行，且这些函数也需要应用程序扫描目录，因为 FindFirstChangeNotification 不显示什么发生改变——只是目录中的某些事物发生改变。然而，如果缓冲区溢出，应用程序就必须求助于扫描目录。

NTFS 提供的第三个方法克服了前两个方法的缺点：一个应用程序可以用 *DeviceIoControl* 函数 FSCTL – CREATE – USN – JOURNAL 文件系统代码配置 NTFS 更改日志功能，让关于文件和目录的 NTFS 记录信息转移到一个称为更改日志（*change journal*）的内部文件。一个更改历程通常足够大，实际上可以保证应用程序获得处理改变的机会而不丢失任何改变。应用程序用 FSCTL – QUERY – USN – JOURNAL 文件系统控制从更改历程读取记录，并且它们可以指定直到获得新记录时，才完成 *DeviceIoControl* 函数。

8. 每个用户卷配额

系统管理员通常需要跟踪或限制共享存储器卷的用户磁盘空间的使用，这样 NTFS 就包括配额管理支持。NTFS 配额管理支持允许每个用户体现配额，它对于使用跟踪和跟踪用户达到警戒和极限阀值时是很有用的。当用户超过他的警戒极限时，NTFS 可以被配置成记录事件，向系统事件日志（Event Log）显示发生。类似地，如果用户试图使用超出他的配额极限许可的卷存储，NTFS 可以在系统事件日志记录一个事件而且不执行应用程序文件 I/O，这个应用程序文件 I/O 将导致“磁盘已满的错误代码”的配额违例。

NTFS 跟踪用户卷使用情况，是依据如下事实：它用创建文件和目录的用户的安全 ID（SID）号标记文件和目录（见第 8 章 SID 的定义）。用户所有的文件和目录的逻辑大小根据用户的管理员定义的配额极限记数。这样，用户不能通过创建一个比配额允许还大的空的稀疏文件，再用非零数据填充文件的方式来超出他的或她的配额极限。类似地，50KB 文件可能压缩到 10KB，而 50KB 用于配额统计。

在默认情况下，卷不激活配额跟踪。你需要使用卷的属性（Properties）对话框的配额（Quota）标记去激活配额，如图 12-10 所示，指定默认警戒和极限阀值，并配置当用户达到默认警告和极限阀值时 NTFS 的发生行为。你可以从这个对话框启动配额项（Quota Entries）工具，它使得管理员能指定每个用户不同的极限

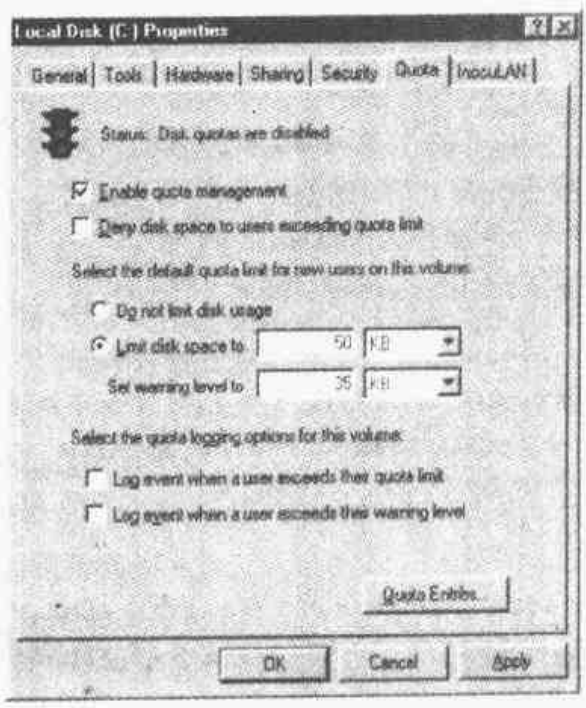


图 12-10 卷属性对话框

和行为。想和 NTFS 配额管理器发生交互作用的应用程序使用 COM 配额接口，它包括 IDiskQuotaControl，IDiskQuotaUser，和 IDiskQuotaEvents。

9. 链接跟踪

shell 快捷方式允许用户将文件放置在 shell 名字空间（例如，在桌面上），它用来链接位于文件系统名字空间的文件。Windows 2000 开始菜单广泛使用 shell 快捷方式。类似地，对象链接和嵌入（OLE）链接允许文档从一个应用程序透明地嵌入到其他应用程序的文档。Microsoft Office 2000 一套产品，包括 PowerPoint、Excel 和 Word 都使用 OLE 链接。

尽管 shell 和 OLE 链接提供了一个很简单的方法用 shell 名字空间将文件同另一个文件相关联，但在过去，它们一直难于管理。如果用户在 Windows NT 4、Windows 95 和 Windows 98 移动一个 shell 的源或 OLE 链接，链接就会被破坏并且系统必须用试探法确定链接源的位置。Windows 2000 的 NTFS 包括称为“分布链接跟踪”的服务应用程序，当链接目标移动时，分布链接跟踪维护 shell 和 OLE 链接的完整性。如果位于一个 NTFS 卷的链接源移动到起始卷的区域内的其他 NTFS 卷，使用 NTFS 链接跟踪支持，链接跟踪服务就能透明地跟踪它的移动并且更新反映改变的链接。

NTFS 链接跟踪支持基于被认为是对象 ID（object ID）的可选文件属性。一个应用程序可以用 FSCTL - CREATE - OR - GET - OBJECT - ID（如果一个文件没有被分配 ID 才分配）和 FSCTL - SET - OBJECT - ID 文件系统控制代码向一个文件分配对象 ID。对象 ID 可以用 FSCTL - CREATE - OR - GET - OBJECT - ID 和 FSCTL - GET - OBJECT - ID 文件系统控制代码来查询。应用程序用 FSCTL - DELETE - OBJECT - ID 文件系统控制代码从文件删除对象 ID。

10. 加密

企业用户经常在计算机上存储敏感数据。尽管存储在公司服务器上的数据通常通过特有的网络安全设置和物理访问控制被安全地保护，但是存储在便携式电脑的数据，如果便携式电脑丢失或被盗窃的话，就会被暴露。NTFS 文件权限不提供保护，因为可以不必运行 Windows 2000 而使用 NTFS 文件读取软件，NTFS 卷可以被完全访问而不需要考虑安全性。此外，当一个可选的 Windows 2000 安装程序从管理员帐户访问文件时，也使得 NTFS 文件权限成为无用的。回顾第 8 章，管理员帐户有取得所有权和备份两个特权，两个特权都允许管理员帐户通过覆盖对象的安全设置来访问任意被标志为安全的对象。

NTFS 包括一个称为加密文件系统（EFS）的设备，用户可以用它来加密敏感数据。EFS 的操作，正如文件压缩，对应用程序来说完全是透明的，这意味着当应用程序授权用户查看数据的帐户下运行时，文件数据被自动解密，当授权应用程序改变数据时，文件数据被自动加密。

注意 NTFS 不允许位于系统卷的根目录或 \Winnt 目录下的文件被加密，因为在这些目录下的许多文件在引导过程期间被访问，而且 EFS 在引导过程期间不被使用。

EFS 是依赖于 Windows 2000 用户模式提供的加密服务，因此它由与 NTFS 紧密结合的内核模式设备驱动程序和与本机安全权限子系统（Lsass）通信的用户模式 DLL 以及加密 DLL 组成。

被加密的文件只能用帐户的 EFS 私有/公用密匙对的密匙来访问，私有密匙用帐户的密码锁定。这样，丢失或被窃的便携式电脑的 EFS - 加密文件在没有许可查看数据的帐户密码的情况下，不能用任何方法访问（除了强制加密的攻击）。

应用程序可以用 Win32 API 函数 *EncryptFile* 和 *DecryptFile* 去加密和解密文件，用 *FileEncryptionStatus* 去获取文件或目录的 EFS 相关的属性，判断文件或目录是否被加密。

11. 支持 POSIX

在第 2 章解释过，Windows 2000 其中的一个要求是完全支持 POSIX 1003.1 标准，在文件系统区域，支持 POSIX 标准要求区分大小写的文件和目录名、遍历权限（当确定用户是否访问文件或目录时，使用一个路径中每个目录的安全性）、"文件更改时间"时间戳（它不同于 MS-DOS "最后一次修改"时间戳和硬链接（指向同一文件的多重目录项）。NTFS 实现上述的每一个特征。

12. 碎片整理

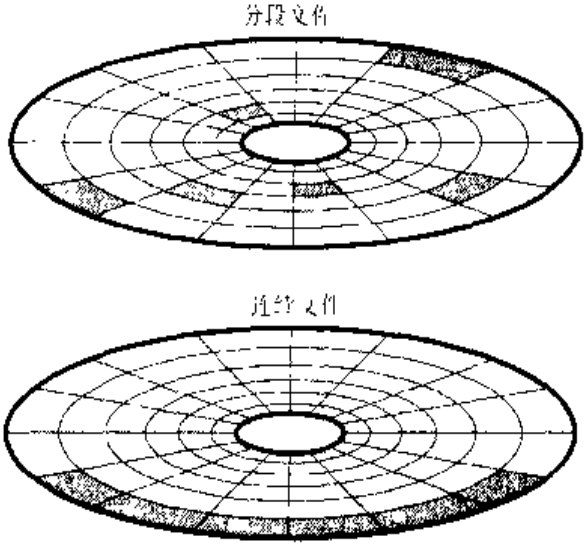


图 12-11　碎片和连续文件

很多人普遍的想法是 NTFS 自动优化磁盘上的文件位置不会出现碎片文件。如果一个文件的数据占据不连续的簇，它就被分割。例如图 12-11 展示一个由 3 个碎片组成的碎片文件。然而，像大多数文件系统（包括 Windows 2000 的 FAT 版本），NTFS 不很努力地去保持文件的连续，除了保留一个被认为是 MFT 的主文件表（MFT）区的磁盘空间区。（当卷的自由空间变得较少时，NTFS 让其他的文件从 MFT 区分配）为 MFT 区保留一个空闲区能帮助 MFT 保持连续，但是，MFT 同样也能变成碎片（关于 MFT 的更多信息见这章后面的"主文件表（MFT）"）。

为了易于第三方磁盘碎片整理工具的开发，Windows 2000 包括碎片整理 API，这种工具用于移动文件数据，这样文件就可以占据连续簇。API 包括文件系统控件，它让文件系统获得卷的没有使用和正在使用的簇的映射（FSCTL-GET-VOLUME-BITMAP）、获得文件簇使用情况的映射（FSCTL-GET-RETRIEVAL-POINTERS）和移动文件（FSCTL-MOVE-FILE）

Windows 2000 包括一个内建的碎片整理工具，可通过磁盘碎片整理功能（*\Winnt\System32\Dfrg.msc*）访问它。内置的碎片整理工具有许多局限，例如不能在命令提示符下运行，不能被自动调度。第三方磁盘碎片整理的产品特色之处在于提供了一个丰富的特征集。

12.4　NTFS 文件系统驱动程序

正如第 9 章所描述的，在 Windows 2000 I/O 系统框架中，NTFS 和其他的文件系统是可加载的以内核模式运行的设备驱动程序。它们间接地被那些使用 Win32 或其他 I/O API（例如 POSIX）的应用程序调用。如图 12-12 所示，Windows 2000 环境子系统调用 Windows 2000 系统服务，Windows 2000 系统服务依次查找适当的加载的驱动程序并调用它们（关于系统服务调度的描述见第 3 章"*系统服务调度*"部分）

分层的驱动程序通过调用 Windows 2000 执行程序的 I/O 管理器将 I/O 请求传递给另一个驱动程序。依赖 I/O 管理器作为中间介质，分层的驱动程序允许每个驱动程序保持它的独立性，这

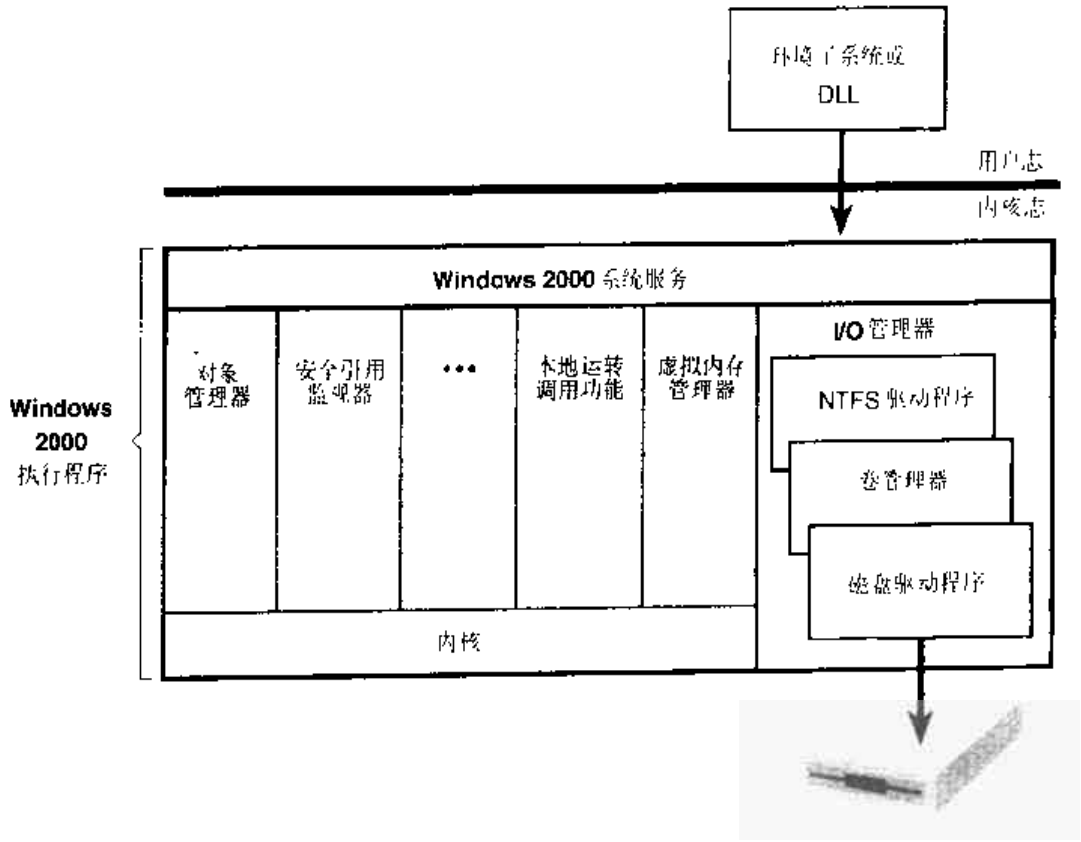


图 12-12 Windows 2000 I/O 系统组件

样它可以被加载或卸载而不影响其他的驱动程序。另外，NTFS 驱动程序同三个其他的 Windows 2000 执行程序组件进行交互，如图 12-13 左边部分所示，这三个组件同文件系统密切相连。

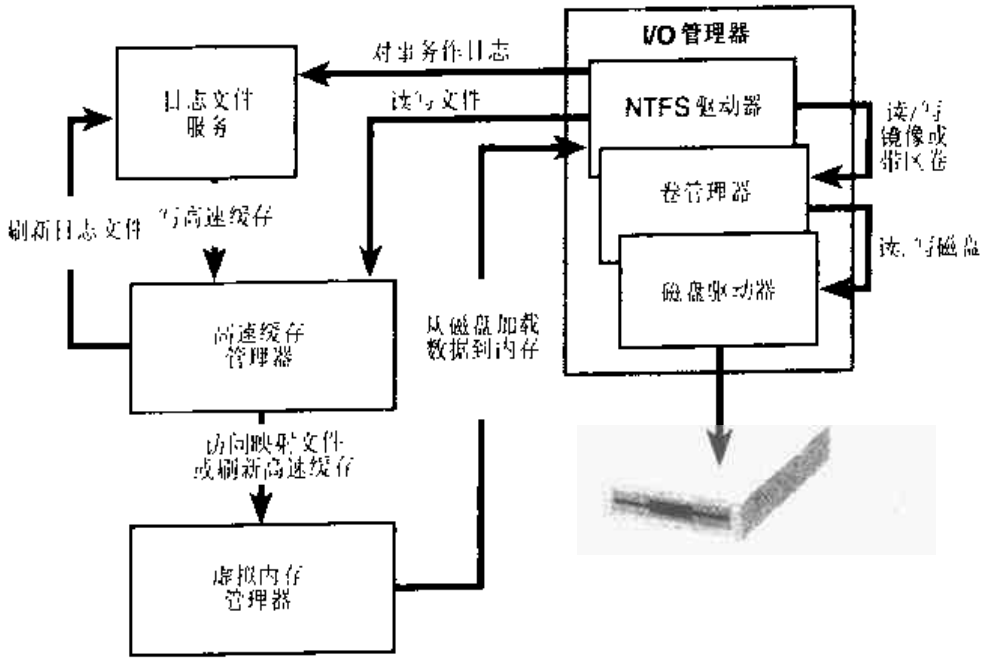


图 12-13 NTFS 和相关组件

日志文件服务（LFS）是 NTFS 为保证磁盘写操作的日志所提供服务的一部分。LFS 写的日志文件在系统发生崩溃时用来恢复 NTFS 格式卷的。关于 LFS 的更多信息见“日志文件服务

(LFS)”一节。

高速缓存管理器是 Windows 2000 执行程序组件，它提供 NTFS 系统范围的高速缓冲服务和其他的文件系统驱动程序，包括网络文件系统驱动程序（服务器和重定向器）。为 Windows 2000 实现的所有文件系统通过将高速缓冲文件映射到系统地址空间来访问它们，然后访问虚拟内存。为此，高速缓冲管理器为 Windows 2000 内存管理器提供一个特殊的文件系统接口。当程序试图访问没有被装入高速缓冲区（在高速缓冲区中未见到的）文件的一部分时，内存管理器调用 NTFS 访问磁盘驱动程序并从磁盘获得文件内容。高速缓存管理器通过使用延迟书写器调用内存管理器将高速缓冲区的内容刷新到磁盘作为后台活动来优化磁盘 I/O（异步磁盘写入）。（内存管理器的详细内容，在 11 章。）

NTFS 是通过将文件实现为对象来体现 Windows 2000 对象模型。这种操作允许文件共享和被对象管理器保护。对象管理器是 Windows 2000 用来管理所有执行程序一层次对象的组件。（对象管理器在第 3 章“对象管理器”一节描述。）

应用程序创建和访问文件正如创建和访问其他 Windows 2000 对象一样：通过对象句柄。当 I/O 请求到达 NTFS 时，Windows 2000 对象管理器和安全系统已经验证了调用进程具备试图访问文件对象的权利。安全系统已经比较了调用程序的访问令牌和文件对象的访问控制表中的项（关于访问控制表的更多信息见第 8 章）。I/O 管理器已经将文件句柄转换为指向文件对象的指针。NTFS 利用文件对象的信息访问磁盘上的文件。

图 12-14 显示了链接文件句柄和文件系统的磁盘结构的数据结构

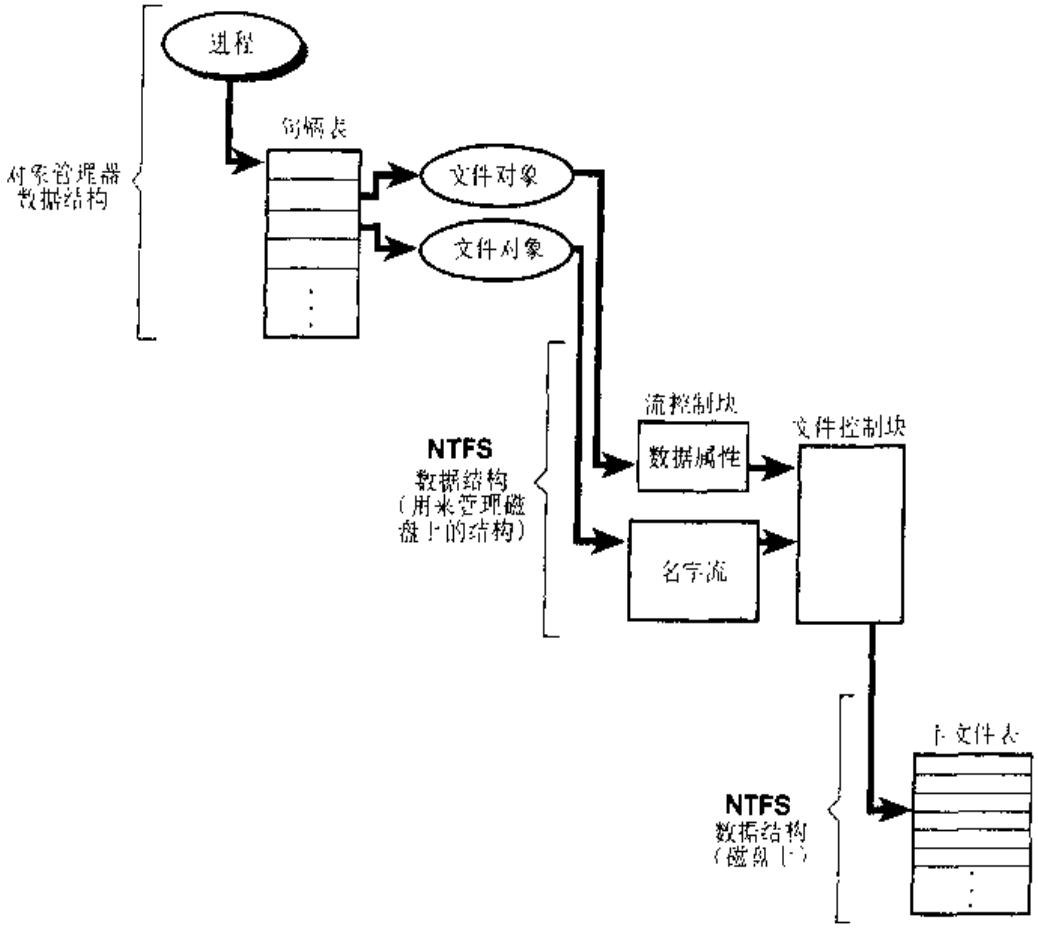


图 12-14 NTFS 数据结构

NTFS 通过跟踪一些指针从文件对象获得磁盘上文件的位置。如图 12-14 所示，一个文件对象，代表对打开文件系统服务的单一调用，它指向调用程序试图去读或写的文件属性的流控制块（SCB）。在图 12-13 中，进程已经既打开了文件无名的数据属性又打开了文件已命名的流（可替换的数据属性）。SCB 代表单个文件属性，并包含关于在文件中如何找到具体属性的信息。一个文件的所有 SCB 指向一个称为文件控制块的通用数据结构（FCB）。FCB 包含一个指向基于磁盘的主文件表（MFT）的文件的记录的指针（实际上，是文件引用，它将在本章后面“文件引用数”部分予以解释）。MFT 将在下面部分详细描述。

12.5 NTFS 磁盘结构

这部分将描述 NTFS 卷的磁盘结构，包括磁盘空间如何被划分和组织成簇，文件如何被组织成目录、实际文件数据和属性信息如何被存储到磁盘，最后，NTFS 如何进行数据压缩工作。

12.5.1 卷

NTFS 结构开始于卷。一个卷对应着磁盘上的一个逻辑分区，当你格式化磁盘或磁盘的一部分为 NTFS 卷时，卷就被创建。你还可以用 Windows 2000 磁盘管理程序 MMC 插件创建跨越多个磁盘的 RAID 卷。

磁盘可以有一个或多个卷。NTFS 独立地处理每个卷。三个硬盘为 150MB 的磁盘配置的示例如图 12-15。

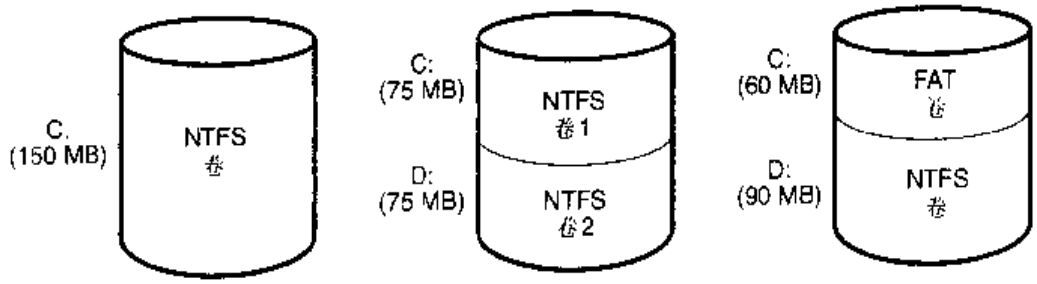


图 12-15 磁盘配置示例

卷由一系列文件加上任意额外的磁盘分区上未分配的空间组成。在 FAT 文件系统中，一个卷还包括专门为文件系统使用而格式化的区。但是，NTFS 卷存储所有的文件系统数据，例如位图和目录，甚至作为常规文件的引导程序。

12.5.2 簇

在 NTFS 卷上簇的大小或者簇因子（cluster factor），是当用户用格式化命令或磁盘管理程序 MMC 插件格式化卷时建立的。默认簇的因子随着卷的大小不同而改变，但它是物理扇区的整数倍，总是 2 的幂次（1 扇区、2 扇区、4 扇区、8 扇区等等）。簇因子可以写为簇的字节数，例如 512 字节、1KB 或 2KB。

在内部，NTFS 只涉及到簇（然而，NTFS 形成了低层次卷 I/O 操作，这样它是扇区对齐的，并且它的长度是扇区大小的倍数）。NTFS 使用簇作为它的分配单元保证它与物理扇区大小无关。这种独立性允许 NTFS 通过使用一个较大的簇因子来高效支持非常大的扇区或支持不是

512 字节扇区大小的非标准磁盘。在一个较大的卷，使用较大的簇因子可以减少碎片和加速分配以较少浪费磁盘空间。使用 Windows 2000 命令提示符下获得的 format 命令和 Disk Management MMC 插件中 Action 菜单上 All Tasks 选项下 Format 菜单选项可以基于卷的大小选择默认簇因子，但是你可以覆盖这个大小。

NTFS 通过逻辑簇号（LCN）指定磁盘上的物理位置。LCN 是所有的簇从卷的开始到结尾的简单的编号。当磁盘驱动程序接口需要时，为了将 LCN 转换为一个物理磁盘地址，NTFS 用簇因子乘以 LCN 获得卷上的物理字节偏移量。NTFS 用虚拟簇号（VCN）引用文件数据。VCN 对属于特定文件的簇从 0 到 m 进行编码。VCN 不必在物理上是连续的，然而它们可以被映射到卷上的任意 LCN 编码。

12.5.3 主文件表

在 NTFS，所有存储在卷上的数据都包含在文件中，包括用来定位和获取文件的数据结构，引导程序数据和记录这个卷（NTFS 元数据）的分配状态的记录的位图（bitmap）。在文件中存储一切使得文件系统很容易地定位和维护数据，每个单独的文件可以被一个安全描述符保护。除此以外，如果磁盘一个特定部分坏了，NTFS 能重分配元数据防止磁盘不能被访问。

MFT 是 NTFS 卷结构的核心，MFT 作为一个文件记录数组实现。不管簇的大小是多少，每个文件记录的大小固定在 1KB（文件记录结构在“文件记录”一节中描述）。从逻辑上讲，MFT 包含卷上每个文件的一个记录，包括 MFT 本身的一个记录。除了 MFT，每个 NTFS 卷包含元数据文件集，元数据文件集包含用来实现文件系统结构的信息。这些 NTFS 元数据文件的每一个都具有以 4 个美元符号（$）开始的名字，但符号是被隐藏的。例如，MFT 的文件名是 $ Mft。NTFS 卷上其他的文件是正常的用户文件和目录，如图 12-16 所示。

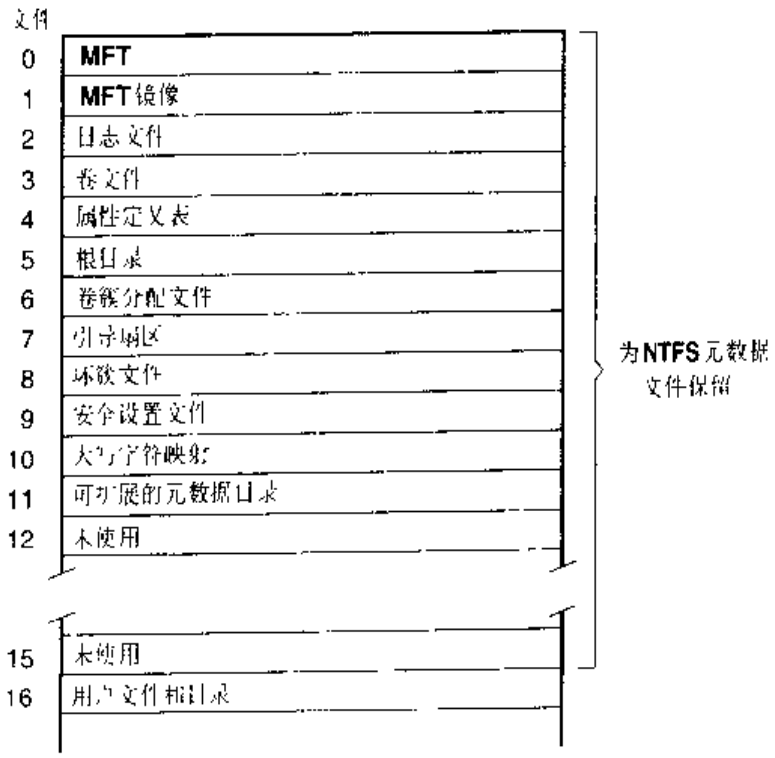


图 12-16 在 MFT 中 NTFS 元数据文件记录

通常，每个MFT记录对应着一个不同的文件。如果文件具有大量属性或者文件被分成大量的碎片，但对单独一个文件来说，可能不止需要一个记录。在上面情况下，MFT的第一个记录存储了其他文件的位置，被称为基文件记录（base file record）。

实验：查看MFT

包含在OEM支持工具（OEM Support Tool）（Windows 2000调试工具的一部分，并可在 *support.microsoft.com/support/kb/articles/Q253/0/66.asp* 中下载）中的Nfi功能允许你转储NTFS卷的MFT的内容，也允许你转换一个卷簇号或物理磁盘扇区数（仅在非RAID卷）到包含它的文件，如果它是文件的一部分。MFT的前16项是为元数据文件保留的，但是可替换的元数据文件（只有卷使用相关特征时才出现）不属于这个区的范围：\ $ *Exxtend*\ $ *Quota*、\ $ *Extend*\ $ *ObjId*、\ $ *Extend*\ $ *UsnJrnl* 和 $ *Extend*\ $ *Reparse*。下面对卷执行的转储使用了重分析点（$ Reparse point）、配额（$ Quota）和对象ID（$ ObjId）：

```
C:\>nfi G:\
NTFS File Sector Information Utility.
Copyright (C) Microsoft Corporation 1999. All rights reserved.

File 0
Master File Table ($Mft)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 32-52447 (0x20-0xccdf)
    $BITMAP (nonresident)
        logical sectors 16-23 (0x10-0x17)

File 1
Master File Table Mirror ($MftMirr)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048728-2048735 (0x1f42d8-0x1f42df)

File 2
Log File ($LogFile)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2048736-2073343 (0x1f42e0-0x1fa2ff)

File 3
DASD ($Volume)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $OBJECT_ID (resident)
    $SECURITY_DESCRIPTOR (resident)
    $VOLUME_NAME (resident)
    $VOLUME_INFORMATION (resident)
    $DATA (resident)
```

```
File 4
Attribute Definition Table ($AttrDef)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 512256-512263 (0x7d100-0x7d107)

File 5
Root Directory
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $INDEX_ROOT $I30 (resident)
    $INDEX_ALLOCATION $I30 (nonresident)
        logical sectors 2073416-2073423 (0x1fa348-0x1fa34f)
    $BITMAP $I30 (resident)

File 6
Volume Bitmap ($Bitmap)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073424-2073679 (0x1fa350-0x1fa44f)

File 7
Boot Sectors ($Boot)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (nonresident)
        logical sectors 0-15 (0x0-0xf)

File 8
Bad Cluster List ($BadClus)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (resident)
    $DATA $Bad (nonresident)

File 9
Security ($Secure)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA $SDS (nonresident)
        logical sectors 2073932-2074447 (0x1fa54c-0x1fa74f)
        logical sectors 523160-523163 (0x7fb98-0x7fb9b)
    $INDEX_ROOT $SDH (resident)
    $INDEX_ROOT $SII (resident)
    $INDEX_ALLOCATION $SDH (nonresident)
        logical sectors 1876152-1876159 (0x1ca0b8-0x1ca0bf)
    $INDEX_ALLOCATION $SII (nonresident)
        logical sectors 24-31 (0x18-0x1f)
```

```
    $BITMAP $SDH (resident)
    $BITMAP $SII (resident)

File 10
Upcase table ($UpCase)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $DATA (nonresident)
        logical sectors 2073676-2073911 (0x1fa44c-0x1fa547)

File 11
Extend Table ($Extend)
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $I30 (resident)

File 12
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 13
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 14
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 15
(unknown/unnamed)
    $STANDARD_INFORMATION (resident)
    $SECURITY_DESCRIPTOR (resident)
    $DATA (resident)

File 24
\$Extend\$Quota
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)
    $INDEX_ROOT $Q (resident)

File 25
\$Extend\$ObjId
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $O (resident)
```

```
File 26
\$Extend\$Reparse
    $STANDARD_INFORMATION (resident)
    $FILE_NAME (resident)
    $INDEX_ROOT $R (resident)
```

当 NTFS 第一次访问卷时，它必须装配卷——也就是，NTFS 从磁盘读取元数据和构造内部数据结构，这样它能处理应用程序文件系统访问。为了装配卷，NTFS 要在引导扇区查看 MFT 的物理磁盘地址。MTF 自己的文件记录是表中的第一项；第二个文件记录指向位于磁盘中部的位置称为“MFT 镜像（mirror）”的文件（文件名 $ MftMirr）。MFT 镜像包含 MFT 的最初几行的拷贝。如果因为某种原因，MFT 文件的一部分不能被读入，MFT 的这些部分拷贝被用来定位元数据文件。

一旦 NTFS 发现了 MFT 的文件记录，它就获得了文件记录中的数据属性的 VCN 到 LCN 的映射信息，并把它存储在内存中。每一个 run 有一个从 VCN 到 LCN 的映射以及 run 的长度，它们是用任意 VCN 定位 LCN 所有必须的信息。这个映射信息告诉 NTFS 组成 MFT 的 run 位于磁盘的位置（run 将在本章后面部分“驻留和非驻留属性”予以解释）。NTFS 然后处理一些更多的元数据文件，并打开文件。接着，NTFS 执行它的文件系统恢复操作（在“可恢复性”一节进行了描述），最后，NTFS 打开了它保留的元数据文件。此时的卷用户可以访问了。

当系统运行时，NTFS 写入另一个重要的元数据文件，日志文件（文件名 $ LOGFile）。NTFS 利用日志文件记录所有影响 NTFS 卷结构的操作，包括文件创建或任何命令，例如 copy，它改变了目录结构。当系统崩溃时，日志文件用于恢复 NTFS 卷。

MFT 的另一项是为根目录保留的（即“\”）。它的文件记录包括存储在 NTFS 目录结构的根下的文件和目录的索引。当 NTFS 第一次被要求打开一个文件时，NTFS 在根目录的文件记录下开始它的搜索。打开文件以后，NTFS 存储文件的 MFT 文件索引，这样当 NTFS 以后读和写文件时，它能直接地访问文件的 MFT 记录。

NTFS 在位图文件（文件名 $ Bitmap）记录了卷的分配状态。位图文件的数据属性包含了一个位图，它的每个位代表了卷上的一个簇，用来识别一个簇是否是空闲的，还是已经被分配给了文件。

安全文件（文件名 $ Secure）存储卷范围的的安全描述符数据库。NTFS 文件和目录有各自可设置的安全描述符，但为了保存空间，NTFS 在普通文件中存储设置，它允许具有相同的安全设置的文件和目录引用相同的安全描述符。在大多数环境下，整个目录树具有相同的安全设置，于是这种优化提供了有意义的补救。

另一个文件系统，引导文件（文件名 $ BOOT），存储了 Windows 2000 引导程序代码。为了引导系统，引导程序代码必须位于一个指定的磁盘地址。但是，格式化期间，格式化命令通过为文件创建一个文件记录将这个区域定义为一个文件。创建引导文件允许 NTFS 遵循它的原则：使磁盘上的所有一切成为文件。引导文件同 NTFS 源文件一样可以通过应用于所有 Windows 2000 对象的安全描述符被单独地保护。利用“磁盘上的任何事物都为文件”模式也意味着引导程序可以通过标准文件 I/O 进行修改，尽管引导文件不允许被编辑。

为记录磁盘卷上的任意损坏位置和被认为是卷文件（文件名 $ VOLUME）的文件，NTFS 也保留一个坏簇文件（文件名 $ Bad Clus）。卷文件包括卷名、卷被格式化的 NTFS 版本和设置的一个位，它指示当一个磁盘已经有毁损时，必须用 Chkdsk 功能进行修复。Chkdsk 功能在本章后面将详细介绍。大写字母文件（uppercase file）（文件名 $ UpCase）包括一个小写和大写字符的转换表。NTFS 维护一个包含属性定义表（文件名 AttDef）的文件。属性定义表定义了支持卷的属性类型和指示了在系统恢复操作期间它们是否能被索引、恢复等等。

NTFS 在扩展元数据目录（文件名 $ Extend）存储一些元数据文件，包括对象标识符文件（文件名 $ Ovbjld）、配额文件（文件名 $ Quota）、更改日志文件（文件名 $ UsnJrnl）和重分析文件（文件名 $ Reparse）。这些文件保存同 NTFS 选择特征相关的信息。对象标识符文件存储文件对象 ID、配额文件存储被分配配额的卷上的配额极限和行为信息。更改日志文件记录文件和目录的改变，重分析点文件存储关于卷上的哪个文件和目录包含重分析点数据的信息。

12.5.4 文件引用数

在 NTFS 卷上的文件是通过被称为“文件引用”的 64 位值来识别的。文件引用包括一个文件号和一个系列号。文件号对应着 MFT 中的文件的文件记录位置减 1（如果文件超过一个文件记录，就对应着基文件记录的位置减 1）。当每次 MFT 文件记录位置被再次使用时，文件引用序列号就递增 1，这使得 NTFS 执行内部一致性检查。文件引用说明见图 12-17。

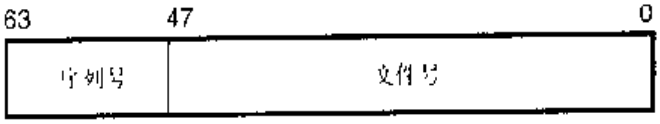


图 12-17 文件引用

12.5.5 文件记录

NTFS 做为属性/值对的集合存储文件，而不是作为文本或二进制数据的存储库查看文件，属性/值对的其中的一个是 NTFS 所包含的数据（称为无名数据属性）。组成一个文件的其他属性包括文件名、时间戳（time stamp）信息和可能的附加命名数据属性。图 12-18 说明了一个小

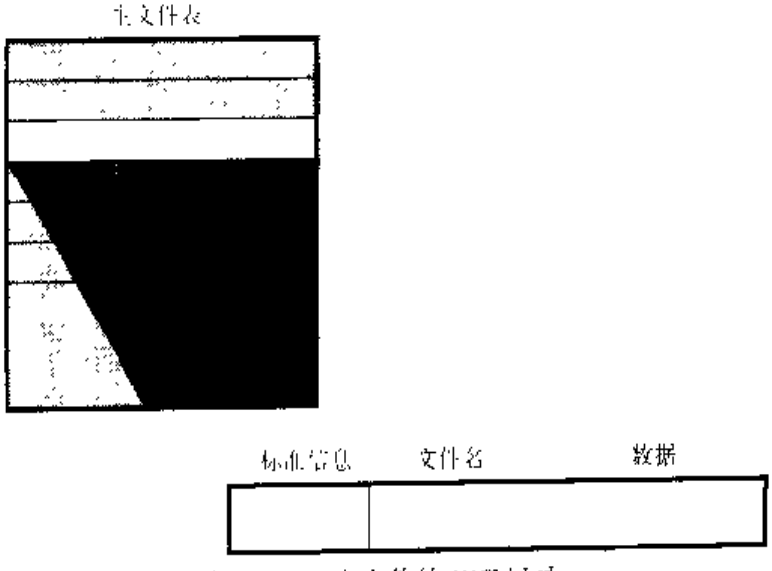


图 12-18 小文件的 MFT 记录

文件的 MFT 记录。

每个文件属性以单独的字节流存储在文件中。严格地说，NTFS 不读和写文件—它读和写属性流。NTFS 支持这些属性操作：创建、删除、读（字节范围）和写（字节范围）。读和写服务通常在文件的无名的数据属性上操作。然而，调用程序通过使用命名数据流格式可以指定不同的数据属性。

表 12-4 列出了 NTFS 卷上的文件属性（并不是每个文件都拥有所有的属性）。

表 12-4 NTFS 文件属性

属性	属性名	描述
卷信息	$ VOLUME - INFORMATION, $ VOLUME - NAME	这些属性仅存在于 $ VOLUME 元数据文件中。它们存贮卷版本沙标信息
标准信息	$ STANDRAD - INFORMATION	文件属性例如只读、存档等等 时间戳（time stamp）包括文件被创建和最后修改的时间；有多少目录指向文件（它的硬链接计数）
文件名	$ FILE - NAME	Unicode 字符文件名。一个文件可以有许多文件名属性。实际上当一个文件硬链接存在时或当一个具有长文件名的文件被 MS - DOS 和 16 位 Microsoft Windows 应用程序访问时，有一个自动生成的“短名”
安全描述符	$ SECURITU - DESCRIPTOR	这个属性与先前版本的 NTFS 向后兼容。Windows 2000 版本的 NTFS 在 $ Secure 元数据文件存储所有安全描述符，在具有相同设置的文件和目录中共享描述符。先前版本的 NTFS 用每个文件和目录存储专用描述信息
数据	$ DATA	文件的内容。在 NTFS 中，一个文件有一个默认的无名数据属性和附加的命名数据属性；也就是一个文件可以有多重数据流。一个目录没有默认数据属性但可以有可替换的命名数据属性
索引根目录 索引分配 和索引位图	$ INDEX - ROOT - $ INDEX - $ ALLOCATION, $ BITMAP	用来实现文件名分配的三个属性和大目录的位图索引（限于目录）
属性列表	$ ATTRIBUTE - LIST	组成文件的属性列表和每个属性所在位置的文件引用。当文件需要不止一个 MFT 文件记录时，就出现这个很少使用的属性
对象 ID	$ OBJECT - ID	文件或目录的 64 字节标识符，使用对于卷唯一的低 16 字节（128 位），链接跟踪服务向 shell 快捷 键和 OLE 链接源文件分配对象 ID。NTFS 提供 API，这样可以用文件和目录的对象 ID 而不是它们的文件名打开文件和目录
重分析信息	$ PEPARSE - POINT	这个属性存储文件重分析点数据。NTFS junction 和包括这个属性的装配点
扩展属性	$ EA, $ EA - INFORMATION	扩展属性实际上不被使用，但它提供对 OS/2 应用程序的向后兼容
EFS 信息	$ LOGGED - UTILITY - STREAM	EFS 存储用来管理加密文件的属性中的数据 例如，用来解密文件加密版本的密匙和授权访问文件的用户列表。在属性名中使用了“logged”是因为这个属性的变更被记录在卷的日志文件中以使文件可恢复（在以后的章节描述）

表 12-4 显示了属性名；然而属性实际上对应着数字类型代码，NTFS 用数字类型代码对文件记录的属性进行排序。MFT 记录中的文件属性是根据这种类型代码被排序的（按升序排列），如果一个文件有多重数据属性或多重文件名，属性类型就会出现不止一次。

文件记录的每个属性是用它的属性类型代码标识的，并且有一个值和一个可替换的名字。属性值是组成属性的位流。例如，$ FIEL－NAME 属性值是文件的名字；$ DATA 的属性值是用户存储在文件里的无论什么字节。

尽管与索引相关的属性和 $ DATA 属性通常有名字，但大多数属性没有名字。名字用来区分一个文件所包含的相同类型的多重属性。例如，具有一个命名的数据流的文件有两个 $ DATA 属性：存储缺省的无名数据流的无名的 $ DATA 属性和一个具有可替换流的名字并对命名流的数据进行分类的命名 $ DATA 属性。

12.5.6 文件名

NTFS 和 FAT 都允许路径中的每个文件名字符长达 255 个。文件名可以包含 Unicode 字符也可以包含多个点和嵌入空格。然而，支持 MS－DOS 的 FAT 文件系统文件名限于 8 个（非 Unicode）字符，跟随在点后的是一个 3 个字符的扩展名。图 12-19 提供了一个支持 Windows 2000 的不同文件名空间的可视表示，图 12-19 还显示了不同类型的文件名空间的交叉。

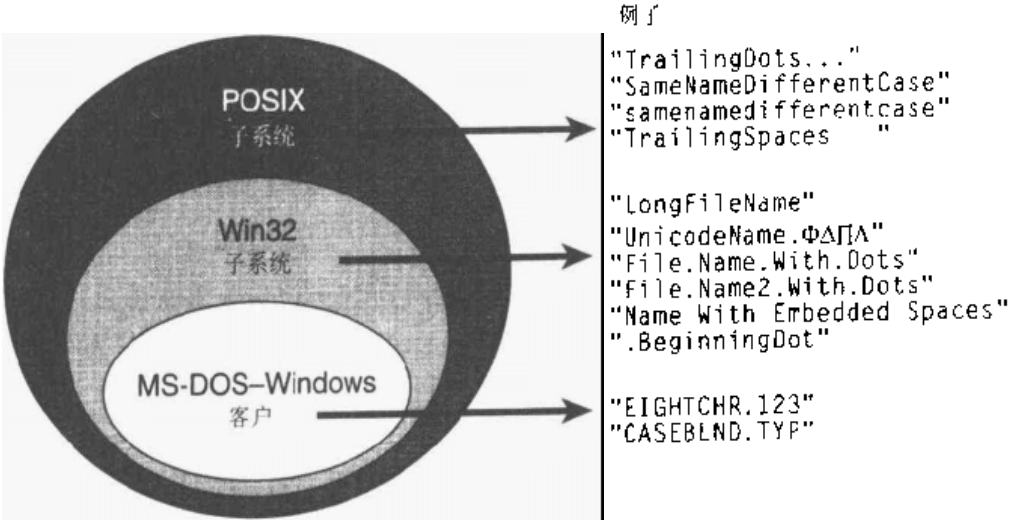


图 12-19 Windows 2000 文件名字空间

POSIX 子系统要求 Windows 2000 支持的所有应用程序执行环境中最大的名字空间，这样，NTFS 名字空间就等价于 POSIX 名字空间。POSIX 子系统可以创建对 Win32 和 MS－DOS 应用程序来说不可视的名字，包括使用后缀点和后缀空格的名字。通常，用大的 POSIX 名字空间创建文件不是问题，因为只要你想要 POSIX 子系统或 POSIX 客户机系统使用那个文件你就可以这样做。

32 位 Windows（Win32）应用程序和 MS－DOS Windows 应用程序关系十分密切，但是，图 12-18 表示 Win32 子系统能在 NTFS 卷上创建 MS－DOS 和 16 位 Windows 应用程序不能见到的文件名。这组应用程序包含比 MS－DOS 8.3 格式更长的文件名，应用程序文件名包含那些 Unicode（国际的）字符，那些有多点或以一个点开始的和那些有嵌入空格的文件名。当一个文件以这种名字创建时，NTFS 自动生成一个备用的文件的 MS－DOS 类型文件名。当你用 dir 命令

使用/x 选项时，Windows 2000 就显示这些短文件名。

MS－DOS 文件名是 NTFS 文件的全部功能的别名，并被以长文件名存储在相同的目录下。具有自动生成的 MS－DOS 文件名的 MFT 记录如图 12-20 所示。

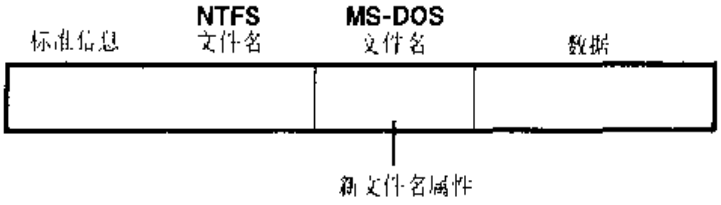


图 12-20　具有 MS－DOS 文件名属性的 MFT 文件记录

NTFS 名和生成的 MS－DOS 名都被存储在相同的文件记录中，这样它们指向相同的文件。MS－DOS 名可以被用来打开、读取、写入或复制文件。如果一个用户用长文件名或短文件名对文件重命名，新名字将取代已经存在的名字。如果新名字不是有效的 MS－DOS 名，NTFS 就为这个文件生成另一个 MS－DOS 名。

注意　POSIX 硬链接以相似的方式实现。当 POSIX 文件的硬链接被创建时，NTFS 向文件的 MFT 文件记录添加另一个文件名属性。尽管在这方面，两种情形是不同的。当一个用户删除一个具有多重名（硬链接）POSIX 文件，文件记录和文件仍在原来的位置。只有当最后一个文件名（硬链接）被删除后，文件和它的目录才被删除。尽管，一个文件既有 NTFS 名又有自动生成的 MS－DOS 名，用户可以用两个中的任何一个来删除文件。

这里的 NTFS 算法用来将一个长文件名生成一个 MS－DOS 名。

1）从长文件名的任意字符删除不合法的 MS－DOS 名，包括空格和 Unicode 字符。删除前缀和后缀的点。删除所有其他的嵌入点，除了最后一个。

2）将点前的字符串截短为 6 个字符，和添加字符串“~n”（n 是从 1 开始的号，用来区别截断后的相同的名字的不同文件）。将点后的字符串——如果存在的话——截短为 3 个字符。

3）将结果表示为大写字母。MS－DOS 是区分大小写的，它保证 NTFS 不会只因大小写不同产生一个新的名字。

4）如果生成的名字与目录下已存在的名字相同，~n 串增 1。

表 12-5 显示了图 12-19 中的长 Win32 文件名和它们的 NTFS 生成的 MS－DOS 版本。目前的算法和图 12-19 的实例应该给你一个像 NTFS 生成 MS－DOS 类型文件名的概念。但是，应用程序开发者不应该依赖算法，因为它在将来可能要改变。

表 12-5　NTFS 生成的文件名

Win32 Long Name	NTFS－Generated Short Name
LongFileName	LONGFI~1
UnicodeName.ΦΔΠΛ	UNICOD~1
File.Name With.Dots	FILENA~1.DOT
File.Name2.With.Dots	FILENA~2.DOT
Name With Embedded Spaces	NAMEWI~1
Beginning Dot	BDGINN~1

12.5.7 驻留和非驻留属性

如果一个文件很小，它的所有属性和属性值（例如它的数据）就放在文件记录里。当属性值直接存储在 MFT 时，属性被称为驻留属性（例如图 12-18 中，所有的属性都是驻留的）。一些属性总是被定义为驻留的，这样 NTFS 可以定位非驻留的属性。例如，标准信息和索引根目录总是驻留的。

每种属性以一个标准头开始，在头中包含有关属性信息和 NTFS 用一般方法管理属性所需的信息。这个头总是驻留的。它记录了属性值为驻留的还是非驻留的。对于驻留属性，头也包含从头到属性值的偏移量和属性值的长度，图 12-21 说明了文件名的属性。

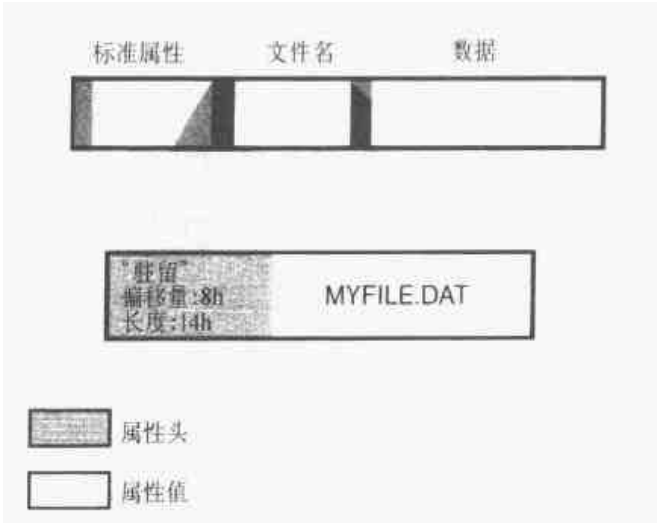


图 12-21 驻留属性头和值

当属性值直接被存储在 MFT 时，NTFS 访问属性值需要的时间被大大降低。NTFS 只访问磁盘一次，并且立即索引数据，而不是在表中查找文件然后通过读取一个连续分配单元去找到文件的数据（例如，正如 FAT 文件系统所做的）。

小目录的属性和小文件一样能驻留在 MFT 中。如图 12-22 所示。对小目录来说，索引根目录属性包括文件的文件引用和目录的子目录索引。

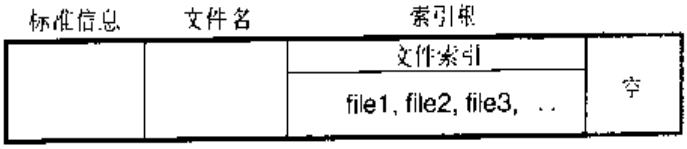


图 12-22 小目录的 MFT 文件记录

当然，许多文件和目录不能被挤入一个 1KB 固定大小的 MFT 记录。如果一个特定的属性，如文件的数据属性，包含在一个 MFT 文件记录中太大，NTFS 将在磁盘上为属性数据分配一个与 MFT 分开的簇。这个区被称为 run（或 extent）。如果属性值后来增长了（例如用户向文件添加数据），NTFS 为添加的数据分配另一个 run。属性值存储在 run 而不是 MFT 中的属性被称为"非驻留属性"。文件系统决定了一个特定的属性是驻留的还是非驻留的；数据的位置对访问数据的进程来说是透明的。

当属性是非驻留时，而一个大文件的数据属性也可能是，它的头包含 NTFS 需要用来查找磁盘上的属性值的信息。图 12-23 显示了存储在两个 run 中的非驻留属性。

在标准属性中，只有那些能增长的属性才能是非驻留的。对文件来说，能增长的属性是数据和属性表。（在图 12-23 没显示）。标准信息和文件名属性总是驻留的。

一个大的目录也能有非驻留属性（或属性的部分），如图 12-24 所示。在这个例子中，MFT 文件记录没有足够的空间存储组成大目录的文件索引。索引的一部分存储在索引根目录属性

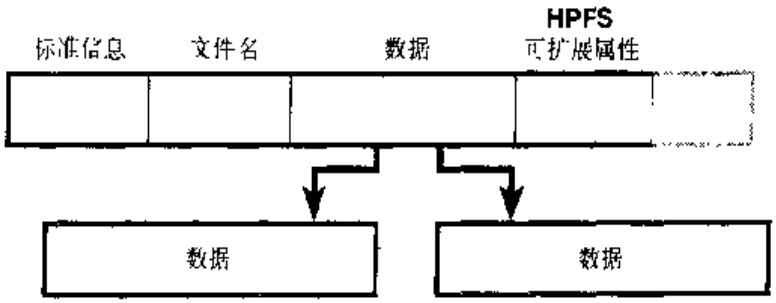


图 12-23 用两个数据 run 的大文件的 MFT 文件记录

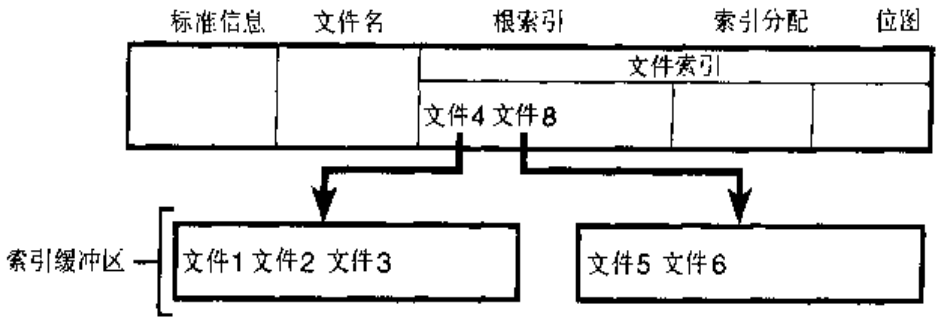


图 12-24 带非驻留文件名索引的大目录的 MFT 文件记录

中，其他的索引被存储在称为索引缓冲区的非驻留 run 中。索引根目录、索引分配和位图属性在这里以一种简化的形式显示。在下部分它们将被详细描述。标准信息和文件名属性总是驻留的。头和至少是索引根目录属性值的一部分对目录来说也是驻留的。

当一个文件的（或目录的）属性不适合 MFT 记录并且需要另外的分配时，NTFS 通过 VCN - LCN 映射对跟踪 run。LCN 代表整个卷从 0 到 n 的簇序列。NTFS 对属于一个特定文件的簇从 0 到 m 进行编号。例如，位于一个非驻留数据属性的簇按如图 12-25 所示进行编号。

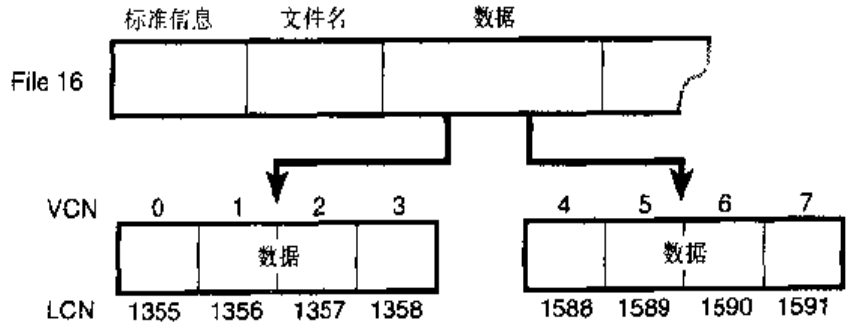


图 12-25 非驻留数据属性的 VCN

如果这个文件有超过两个的 run，第三个 run 号将从 VCN8 开始。如图 12-26 所示，数据属性头包括这两个 run 的 VCN - LCN 映射，它使得 NTFS 很容易找到磁盘上的分配。

尽管图 12-26 只显示了数据 run，如果 MFT 文件记录中没有足够的空间包含其他的属性，其他的属性可以被存储在 run 上。如果一个特定文件有许多属性满足 MFT 记录，第二个 MFT 记录用来包含附加属性（或非驻留属性的属性头）。如果这样，称为“属性表”的属性就被添

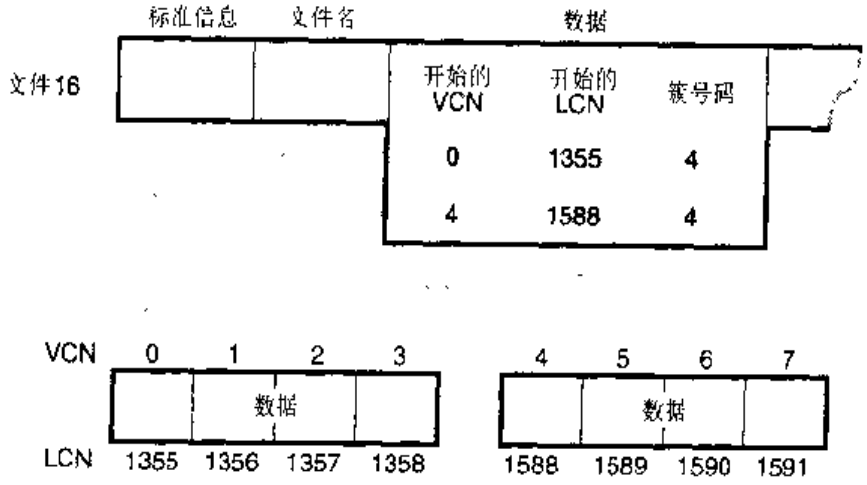


图 12-26 对非驻留数据属性来说的 VCN - to - LCN 映射

加。属性表属性包括名字和每个文件的属性的类型代码以及属性所在的 MFT 记录的文件引用。当文件变得很大或文件被分割很厉害以至于单个的 MFT 记录不能包含需要找到它所有 run 的 VCN - LCN 映射时，就要使用属性表属性。超过 2000 个 run 的文件尤其需要一个属性表。

12.5.8 索引

在 NTFS 中，文件目录是文件名的简单索引——也就是，为了便于快速访问而以一种特定的方式组织的文件名集合（伴随它们的文件引用）。为创建一个目录，NTFS 索引目录文件下的文件名属性。卷的根目录下的 MFT 记录如图 12-27 所示。

从概念上讲，目录的 MFT 项包含在它的目录文件分类表的索引根目录属性中。尽管对大的目录，文件名实际上存储在 4KB 固定大小的索引缓冲区中，索引缓冲区包含和组织文件名。索引缓冲区实现的是一个 b + tree 数据结构，它可以减少查找特定文件所需的访问磁盘的次数。索引根目录属性包含 b + tree（根子目录）的第一层并指向包含下层的索引缓冲区（或许是更多的子目录或文件）。

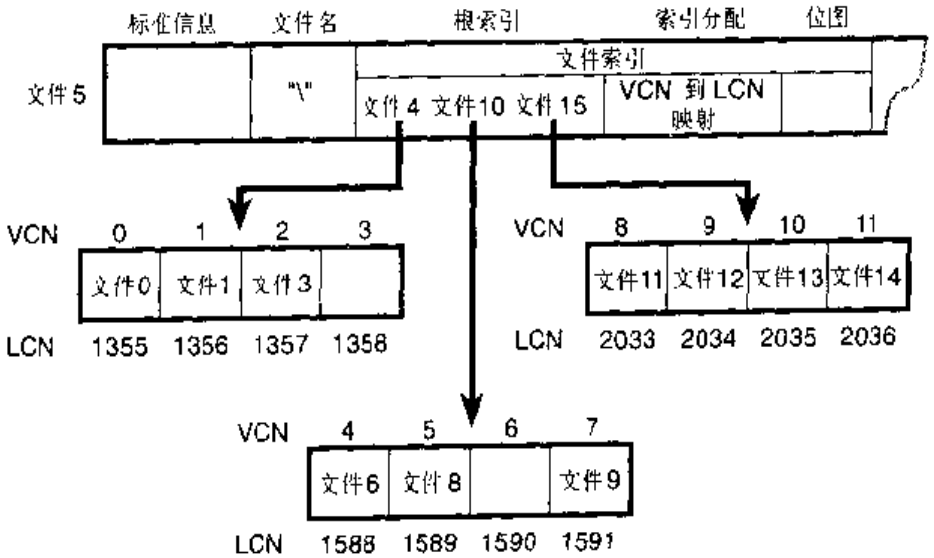


图 12-27 卷根目录的文件名索引

图 12-27 显示了具有索引根目录属性的文件名和索引缓冲区（例如，file6）。但索引中每个项也包括描述文件的 MFT 中的文件引用和文件的时间戳（time stamp）以及文件大小的信息。NTFS 从文件的 MFT 记录复制时间戳和文件大小信息。被 FAT 和 NTFS 使用的这种技术需要在两个位置更新要写入的信息。虽然如此，它可以显著加速目录浏览的优化，因为它允许文件系统无需打开目录下的每个文件去显示每个文件的时间戳和大小。

索引分配属性包含将索引缓冲区 run 的 VCN 映射到显示索引缓冲区在磁盘上位置的 LCN，位图属性跟踪索引缓冲区中的哪个 VCN 正被使用和哪个 VCN 没被使用。图 12-27 显示了每个 VCN（也就是，每个簇）的文件项，但是文件名项实际上被压缩到每个簇。每 4KB 索引缓冲区可以包含 20～30 个文件名项。

b + tree 数据结构是平衡树类型，它对于组织存储在磁盘上的分类数据是很理想的，因为它减少了查找一个项所需要的磁盘访问次数。在 MFT 中，目录的索引根目录属性包括许多文件名，它们作为 b + tree 的第二层索引。索引根目录属性里的每个文件名都有一个与它相关的指向索引缓冲区的选择指针。索引缓冲区指向包含具有小于它自己的词典编辑值（lexicographic value）的文件名。例如，图 12-27，file4 是 b + + 的第一层目录。它指向包含文件名小于它本身的索引缓冲区——文件名 file0、file1 和 file3 的索引缓冲区。注意在这个例子中使用的 file1、file2 等的名字不是字面上的文件名而是想显示文件的相对位置的名字，文件是按显示的顺序进行词典排序的。

在 b + tree 中保存文件有很多好处。因为文件名按分类次序进行存储，所以目录查询很快。当高层次软件在目录中列文件时，NTFS 返回已排序的名字。最后，因为 b + tree 趋向变得更宽而不是更深，所以 NTFS 的快速查询时间不会随目录增长降低。

NTFS 也提供包括索引文件名在内的索引数据的通用支持。正如我们在前面强调的，一个文件可以有一个 NTFS 分配给它的对象 ID，对象 ID 存储在文件的 $ OBJECT - ID 属性中。NTFS 提供 API 允许应用程序使用文件的对象 ID 而不是文件名来打开一个文件。NTFS 因此必须使转换一个对象 ID 到文件的文件号的过程有效。为了做到这一点，NTFS 将卷的所有对象 ID 的映射存储在 \ $ Extend \ $ ObjId 下元数据文件对象 ID 的文件引用号中。NTFS 在 $ ObjId 的 $ O 索引中对对象 ID 进行排序。就象文件名在文件名索引中一样，对象 ID 索引也以 b + tree 形式存储。

12.5.9 数据压缩和稀疏文件

NTFS 支持以每个文件、每个目录或每个卷为基础的压缩（NTFS 只对用户数据，而不对文件系统元数据进行压缩）。你可以用 Win32 *GetVolumeInformation* 函数区别一个卷是否被压缩。用 Win32 *GetCompressedFileSize* 函数去获取文件实际压缩的尺寸。最后，用 Win32 *DeviceControl* 函数去检查和改变文件或目录的压缩设置（见 FSCTL - GET - COMPRESSION 和 FSCTL - SET - COMPRESSION 文件系统控制代码）。记住，尽管设置文件的压缩状态为立即压缩（解压缩）文件，设置目录或卷的压缩状态不会导致任何立即压缩或解压缩。取而代之的是在目录或卷的压缩状态的设置中，对目录和卷内所有新创建的文件和子目录设置了一个默认的压缩状态。

下面的部分通过分析一个压缩稀疏数据的简单例子介绍了 NTFS 压缩。后面部分继续讨论

了常规文件和稀疏文件的压缩。

1. 压缩稀疏数据

稀疏数据通常很大但只包含相对于它的大小来说一小部分的非零数据。稀疏矩阵是稀疏数据的一个例子。正如以前所描述的，NTFS 使用 VCN，从 0 到 m 列举一个文件的簇。每个 VCN 映射到相应的 LCN，LCN 标识簇的磁盘位置。图 12-28 说明了正常（磁盘分配）的非压缩文件的 run，包括 VCN 和 LCN 之间的映射

图 12-28 非压缩文件的 run

这个文件用3 个 run 存储，每个 run 有 4 个簇，整个文件有 12 簇。图 12-29 显示了这个文件的 MFT 记录。正如以前描述的，为节省空间，包含 VCN-LCN 映射的 MFT 记录的数据属性只记录每个 run 的一个映射，而不记录每个簇的一个映射。注意，尽管这样，从 0 到 11 的每个 VCN 有一个和它相应的 LCN 相联系。第一个项从 VCN 0 开始，覆盖 4 个簇，第二个项从 VCN 4 开始覆盖 4 个簇等等。这个项格式对非压缩文件来说是典型的。

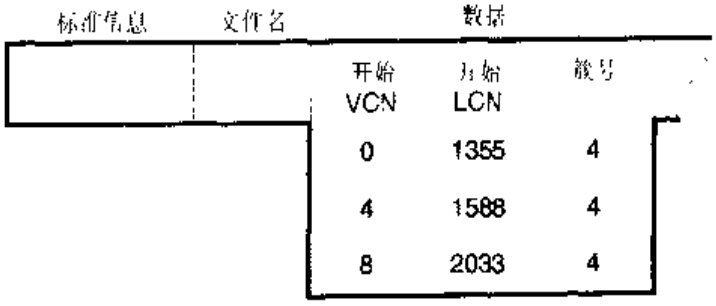


图 12-29 非压缩文件的 MFT 记录

当用户在 NTFS 卷上选择一个文件进行压缩时，NTFS 压缩的一个方法是从文件删除为零值的长串。如果文件数据是稀疏的，文件数据尤其可以收缩到占据它应要求占据的磁盘空间的一部分。在以后的对文件的写入中，NTFS 只分配包含非零数据的 run。

图 12-30 描述了包含稀疏数据的压缩文件的 run。注意文件的 VCN 在一定范围（16～31 和 64～127）没有磁盘分配。

压缩文件的 MFT 记录遗漏了包含零的 VCN 块，这样就不给它们分配物理存储。例如，图 12-31 的第一个数据项开始于 VCN 0，覆盖 16 个簇。第二个项跳到了 VCN 32，覆盖 16 个簇。

当程序从一个压缩文件读取数据时，NTFS 通过校验 MFT 记录确定从 VCN-LCN 的映射是否覆盖已读的位置。如果程序从文件未分配的“孔”读取，这意味着数据在由零组成的文件的一部分中。如果一个程序向一个“孔”写入非零数据，NTFS 分配磁盘空间然后写入数据。这个方法对包含许多零数据的稀疏文件数据非常有效。

2. 压缩非稀疏数据

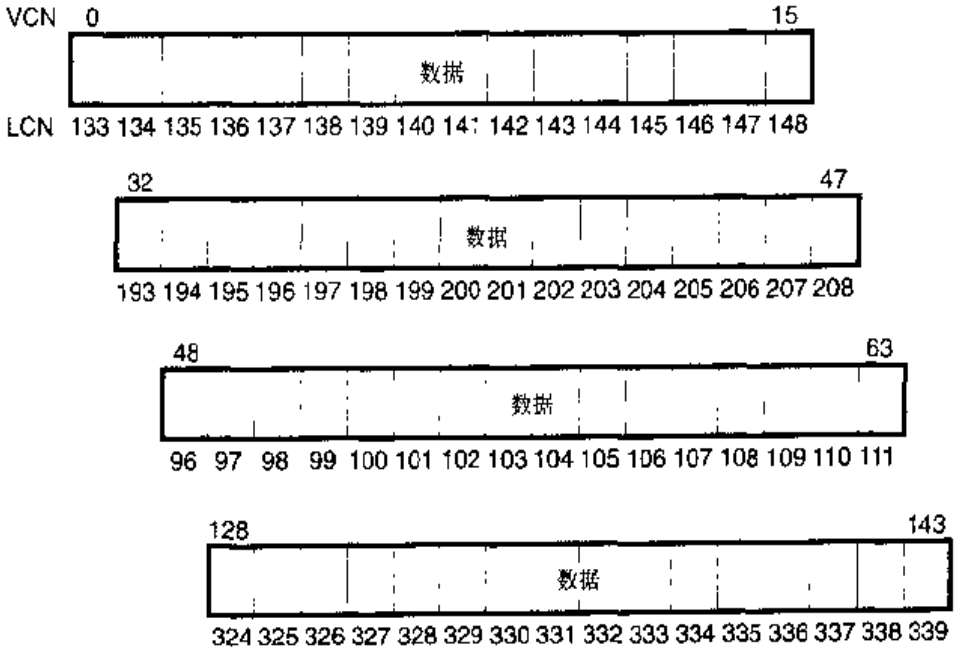


图 12-30 包含稀疏数据的压缩文件的 run

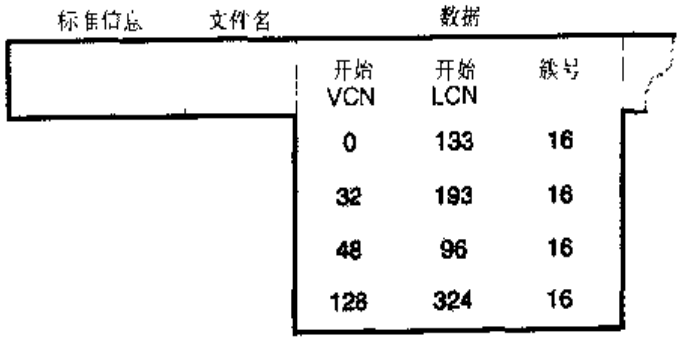


图 12-31 包含稀疏数据的压缩文件的 MFT 记录

先前的压缩稀疏文件的例子有点象是专门设计的。它针对一个文件的整个部分允满零但文件的其余数据不受压缩影响的情况描述了“压缩”。大多数文件的数据不是稀疏的，但它仍能通过压缩算法应用程序被压缩。

在 NFS，用户可以指定单个文件或一个目录下的所有文件的压缩（在目录下创建的标记为压缩的新文件是自动被压缩的——存在的文件必须被单独地进行压缩）。当 NTFS 压缩一个文件时，它将文件未处理的数据分成 16 簇的压缩单元（例如，对于 512 字节簇等于 8KB）。如果完全压缩，文件中的某一数据系列可能压缩不多；这样对每个文件中的压缩单元，NTFS 确定压缩单元是否可以节约至少 1 簇存储。如果压缩单元不释放至少一个簇，NTFS 就分配一个 16 簇 run 并将在那个单元的数据写入没有被压缩的磁盘。如果在 16 簇单元的数据要压缩到 15 或更少的簇，NTFS 只分配需要包含压缩数据的簇号，然后将簇号写入磁盘。图 12-32 说明了有四个 run 的文件的压缩。图中没有阴影的区域代表压缩后文件所占据的实际存储位置。第一个、第二个和第四个 run 被压缩了；第三个 run 没被压缩。甚至用一个非压缩的 run 压缩这个文件，能节省磁盘空间的 26 个簇或磁盘空间的 41%。

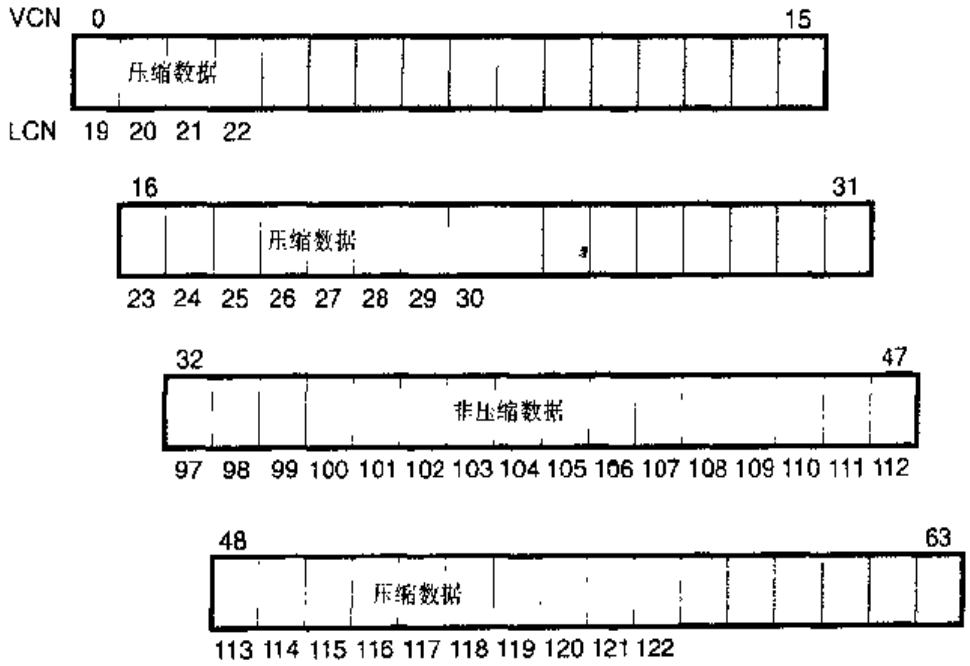


图 12-32 压缩文件的数据 run

注意 尽管本章的图表显示的是连续的 LCN，但压缩单元不需要以物理空间的连续簇存储。占据非连续簇的 run 生成了比图 12-32 显示的稍微有点复杂的 MFT 记录。

当向一个压缩文件写入数据时，NTFS 保证每个 run 在虚拟的 16 簇边界上开始。这样每个 run 的起始 VCN 是 16 的倍数，并且 run 不多于 16 个簇。NTFS 访问压缩文件时，每次至少读和写一个压缩单元。当 NTFS 写入压缩数据时，NTFS 尽量在物理空间连续的位置上存储压缩单元，这样它能以单个的 I/O 操作读取全部数据。选择 NTFS 16 簇压缩单元大小去减少内部的碎片：文件压缩单元越大，需要存储数据的整个磁盘空间越小。16 簇压缩单元表示在生成较小的压缩文件和减缓随机存取文件的应用程序的读操作之间的权衡。必须为每个高速缓存的丢失解压缩 16 簇的等效值（高速缓存丢失在随机文件存取时很容易发生）。图 12-33 显示了图 12-32 所示的压缩文件的 MFT 记录。

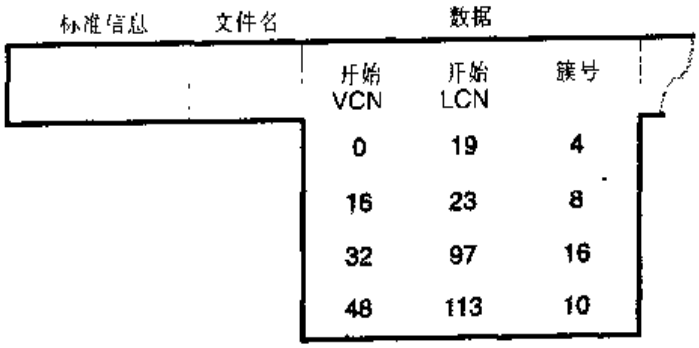


图 12-33 压缩文件的 MFT 记录

这个压缩文件和前面包含稀疏数据的例子的区别在于这个文件的压缩 run 中的三个不超过 16 簇。从文件的 MFT 文件记录中读取信息使得 NTFS 知道文件中的数据是否被压缩了。任意少于 16 簇的 run 包含当 NTFS 第一次读取数据到高速缓存时，NTFS 必须解压缩的压缩数据。一个恰好有 16 个簇的 run 不包含压缩数据，因此也不需要解压缩。

如果 run 的数据已经被压缩，NTFS 就把数据解压缩到临时缓冲区（Scratch buffer），然后把它拷贝到调用程序的缓冲区。NTFS 也加载压缩数据到高速缓存区，这使得以后从相同的 run 读取和从任何其他高速缓存的读取一样快。NTFS 将把文件的任何更新写入到高速缓冲区，让延迟书写器进行压缩，将修改的数据异步地写入到磁盘。这种方法保证写入一个压缩的文件不会比写入一个非压缩的文件延迟很多。

只要可能，NTFS 就保持压缩文件的磁盘分配的连续。正如 LCN 表明，图 12-32 所示的压缩文件的前两个 run 以及最后两个是物理连续的。当两个或更多的 run 是连续的，NTFS 执行磁盘预读，实际上它用的是其他文件的数据。因为在程序请求数据前，连续文件数据的读取和解压缩不是同步发生的，后来的读操作直接从高速缓冲区获得数据，这样大大加强了读的性能。

3. 稀疏文件

稀疏文件（NTFS 文件类型，与在前面描述的由稀疏数据组成的文件正相反）对于 NTFS 不执行压缩的文件的非稀疏数据来说实质上是压缩文件。但是，NTFS 管理稀疏文件的 MFT 记录 run 数据的方式同它管理由稀疏数据和非稀疏数据组成的压缩文件的方式是一样的。

12.5.10 重分析点

正如前面章节描述的，重分析点是应用程序定义的达 16KB 的重分析数据块和一个存储在文件或目录的 $ REPARSE - POINT 属性中的 32 位重分析标记。只要应用程序创建或删除一个重分析点时，NTFS 就更新 \ $ Extend\ $ Reparse 的元数据文件，在元数据文件中，NTFS 存储识别文件的文件记录号的项目和包含重分析点的目录。在中心位置存储记录使得 NTFS 提供了列举所有卷的重分析点或仅是特殊类型的重分析点的应用程序的接口，例如装配点（关于装配点的更多信息见第 10 章）。\ $ Extend\ $ Reparse 文件用 NTFS 的一般索引功能通过重分析点标记比较文件项目（按命名为 $ R 的索引）。

12.5.11 更改日志文件

更改日志文件，\ $ Extend\ $ UsnJrnl，是仅当应用程序允许更改日志时，NTFS 才创建的稀疏文件。日志在 $ J 数据流存储更改项。项包含下列关于文件或目录的更改信息。

- 时间的更改。
- 类型的更改（删除、重命名、大小扩展等等）。
- 文件或目录的属性。
- 文件或目录的名字。
- 文件或目录的文件引用号。
- 文件的父目录的文件引用号。

日志是稀疏的，这样它从不会溢出；当磁盘上的日志大小超过文件定义的最大值时，NTFS 仅仅开始以零值表示文件数据，这些文件数据先于具有大小等于最大日志的更改信息的窗口，如图 12-34 所示。当应用程序连续地超过日志文件的大小时，为了防止连续不断的调整大小，NTFS 只有当它的大小是应用程序定义的值的 2 倍，超过最大配置大小时，才紧缩日志。

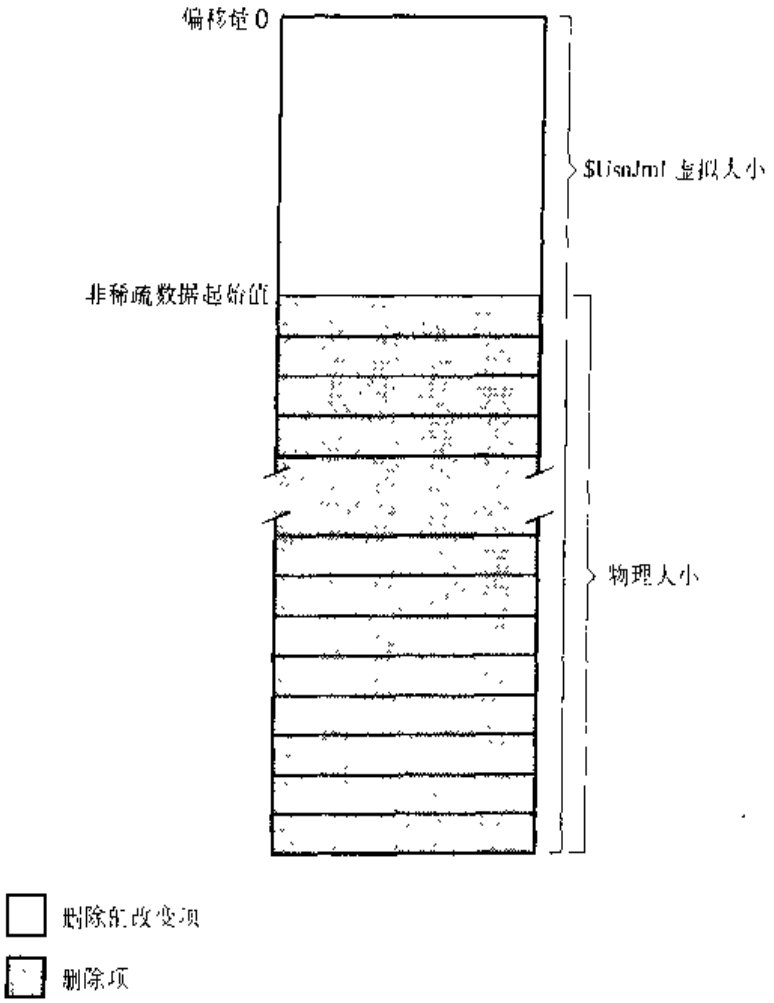


图 12-34　更改历程（$ UsnJrnl）空间分配

12.5.12　对象 ID

除了将分配给文件或目录的对象 ID 存储在 MFT 记录的 $ OBJECT - ID 属性中对分配一个文件或目录的对象 ID 进行分类，NTFS 也保证对象 ID 和对象 ID 在 $ *Extend* \ $ *ObjId* 元数据文件的 $ O 索引中的文件引用号相对应。索引通过对象 ID 检验项目，使得 NTFS 容易快速定位一个基于它的 ID 的文件。这种特性允许应用程序借助非文档化的本机 API 功能使用对象 ID 打开一个文件或目录。

12.5.13　配额跟踪

NTFS 配额跟踪功能将 ID 所有者同创建文件和存储用户所有者的 ID 的每个用户相联。用户所有者的 ID 是用户创建的每个文件或目录的 ID。为确定一个用户是否已被分配一个 ID，NTFS 用户的 SID 作为密匙索引 $ Extend \ $ Quota 元数据文件的 $ O 索引。如果一个 ID 没有被定位，NTFS 为用户分配一个唯一的 ID 并记录 $ O 索引的关联。

$ Extend \ $ Quota 也包含命名为 $ Q 的索引，NTFS 用它存储每个用户配额信息项，通过 ID 所有者检验目录。当用户试图在卷上分配空间时，NTFS 用所有者 ID 查询用户的配额项和确定用户的配额是否留下足够的空间允许分配。

12.5.14 统一的安全

一般索引的另一个例子在 \ $ Secure 元数据文件中可见，元数据文件存储卷上所有文件和目录的安全描述符。NTFS 分配每个安全描述符一个 NTFS 安全 ID（这些描述符同第 8 章描述的 SID 是不同的）。

当一个进程对一个文件或目录应用安全描述符时，NTFS 获得一个描述符的 32 位散列，并查询相应的存储在 \ $ Secure 文件中名字为 $ SDH 的索引中的安全 ID。多个安全描述符能散列到同一个值，这样，NTFS 比较被应用于任意具有同样的散列的值的安全描述符以验证确切的匹配。如果 NTFS 查找在 $ SDH 索引中应用的安全描述符的位置，NTFS 向文件分配相关的安全 ID。否则，它分配一个新的安全 ID，更新 $ SDH 索引，向 $ SH 索引添加安全描述符。$ SH 索引是通过安全 ID 检验的，这样当用户试图打开一个文件或目录时，NTFS 能用文件或目录的安全 ID 迅速定位文件或目录的安全描述符。

12.6 支持 NTFS 恢复

如果发生停电或系统崩溃，文件系统操作（事务）会被完整地保留，并且卷的磁盘结构也会保持完整，无须运行磁盘修复程序。NTFS Chkdsk 程序用来修复由于 I/O 错误（例如：坏磁盘扇区、电流异常、磁盘故障）或软件故障引起的灾难性磁盘毁损。但是，随着 NTFS 恢复功能的使用，Chkdsk 很少被使用。

正如前面提到的（在“可恢复性”小节），NTFS 使用事务处理方案实现可恢复功能。这种策略确保了极迅速的完整磁盘恢复。NTFS 限制它对文件系统数据的恢复过程，保证用户至少决不因为文件系统的毁损而丢失卷；然而，除非一个应用程序采取特殊的措施（例如将高速缓存的文件刷新到磁盘），当系统崩溃时，NTFS 不保证用户数据被完全更新。在大多数 Windows 2000 的数据库产品可以获得基于事务的用户数据的保护，例如，Microsoft SQL 服务器。在文件系统中不实现用户数据恢复的决策代表着在完全的容错文件系统和其他对所有文件操作提供优化的系统之间的权衡。

12.6.1 文件系统设计演变

可恢复文件系统的发展是文件系统设计进展的一个迈进。在过去，创建文件系统的 I/O 和支持高速缓存的有两种常用的方法：认真写（careful write）和延迟写（lazy write）。为数字设备公司（现在是 COMPAQ）的 VAX/VMS 和一些其他的专用的操作系统发展的文件系统使用的是认真写算法，而 OS/2HPFS 和大多数老的 UNIX 文件系统使用的是延迟写文件系统模式。

下面的两小节简单地回顾了这两种类型的文件系统和它们在安全和性能之间的内部权衡。第三小节描述了 NTFS 的恢复方法并解释了它与其他两个策略的不同之处。

1. 认真写文件系统

当一个文件系统崩溃或不起作用时，正在进行的 I/O 操作被立即或过早中断。这种突然中断将引起文件系统的不一致性，这取决于正在进行的是什么 I/O 操作或操作和它们向前进行的

程度。在本语境下的不一致性是指文件系统的毁损——例如，文件名出现在目录列表，但是文件系统不知道文件在哪里或者不能访问文件。最严重的文件毁损可以导致整个卷不能被访问。

一个认真写文件系统不能防止文件系统出现不一致性，但是认真写文件系统对文件系统的写操作进行排序，这样，即使在最坏的情况下，系统崩溃导致的文件的不一致是可预测、非关键的。文件系统可以在它空闲的时候整理文件的不一致。

当任何类型的文件系统接受到更新磁盘的请求时，在更新完成之前它必须执行一些子操作。使用认真写策略的文件系统，子操作总是序列化地写入磁盘。例如，当为一个文件分配磁盘空间时，文件系统首先在它的位图中设置一些位，然后再给文件分配空间。当位被设置后如果突然停电，认真写文件系统不能访问一些磁盘空间——由设置的位表示的空间，但存在的数据不会被毁损。

对写操作进行序列化意味着 I/O 请求按它们被接受的次序填充。如果一个进程分配磁盘空间不久，另一个进程就创建了一个文件，认真写文件系统在它开始创建文件时就完成了磁盘分配，因为交叉存取两个 I/O 请求的子操作会导致不一致的状态。

注意 FAT 文件系统使用写穿透（write - through）算法，它使文件的修改被立即写入到磁盘。不像认真写方法，写穿透方法不需要文件系统为防止不一致性而对写进行排序。

认真写文件系统的主要优点是万一发生故障，卷会保持一致并可用，无须立即运行速度很慢的修复程序。修复程序需要修改由于系统故障造成的可以预测的、非破坏性的磁盘的不一致，但是修复程序可以在方便的时候运行，特别在系统重启动时。

2. 延迟写文件系统

认真写文件系统牺牲了速度来换取安全。延迟写文件系统通过使用写回（write - back）高速缓存策略来提高性能；也就是，它将文件修改写入高速缓存，然后通常作为一种后台活动以优化的方式将高速缓存的内容刷新到磁盘。

可以采取许多形式进行同延迟写高速缓冲技术相关的性能的改善。首先，磁盘写的总数减少，因为磁盘写是序列化的，所以不要求立即写磁盘，缓冲区的内容在被写入磁盘之前，可以被修改许多次。其次，因为文件系统无需磁盘写的完成就可以返回对调用程序的控制，所以服务应用程序请求速度被大大地提高了。最后，延迟写策略忽略了当交叉执行两个或更多 I/O 请求时导致的文件卷上的不一致的中间状态，这样使文件系统变为多线程容易得多，允许在同一时间执行不只一个 I/O 操作。

延迟写技术缺点是在一个卷处于非常不一致的状态以致于它不能被文件系统修改期间，延迟写技术要产生时间间隔。所以，延迟写文件系统必须一直跟踪卷的状态。总而言之，延迟写文件系统在性能上优于认真写文件系统。但当系统出现故障时会以更大的冒险和用户的不便利作为代价。

3. 可恢复的文件系统

可恢复的文件系统例如 NTFS，在保留延迟写文件系统性能时，设法超过认真写文件系统的安全性能。可恢复的文件系统通过使用最初为事务处理发展的日志技术（有时称为 journaling）来保证卷的一致性。如果操作系统崩溃，可恢复的文件系统通过访问已经保存在日

志文件的信息的恢复过程来恢复一致性。因为文件系统已经记录了磁盘写，所以不管卷有多大，恢复过程只需要几秒的时间

可恢复文件系统的恢复过程是精确的，它保证卷以一致的状态被存储。在 NTFS 中同延迟写文件系统相关的不充分的恢复不会发生。

可恢复文件为它所提供的安全要付出一些代价。每个改变卷结构的事务要求为每个事务的子操作写入一个记录到日志文件。日志记录开销通过文件系统日志记录的批处理被改善——用单一的 I/O 操作向日志文件写入多个记录。另外，可恢复文件可以采用延迟写文件系统的优化技术，它甚至可以在高速存刷新之间增加间隔的长度，因为在高速缓存的更改刷新到磁盘之前，当文件系统崩溃时，文件系统可以被恢复。这样就获得了延迟写文件系统弥补的高速缓存性能，并经常是超过了恢复文件的日志活动的开销。

认真写和延迟写文件系统都不保证对用户文件的数据保护。当应用程序正在写文件时，如果系统崩溃，文件就可能丢失或毁损。更糟糕的，崩溃可以毁损延迟写文件系统、破坏已存在的文件或甚至使整个的卷不能被访问。

NTFS 可恢复文件系统实现了一些能改进传统文件系统的可靠性的策略。首先，NTFS 的可恢复性保证卷的结构不会被毁损。这样当文件系统出现故障时，所有的文件都可以被访问。

其次，尽管 NTFS 在系统崩溃的情况下不保证对用户数据的保护—— 高速缓存区的一些改变会丢失，但应用程序可以利用 NTFS 的写穿透（write - through）和高速缓存刷新（cache flushing）能力来保证磁盘上的文件在适当的间隔修改。高速缓存写穿透（Cache write through）强制磁盘上的写操作立即执行，而高速缓存刷新（cache flushing）强制高速缓存的内容写入磁盘。这些都是有效的操作。NTFS 不必要执行额外的磁盘 I/O 操作去刷新对很多不同的文件系统数据结构的更改，因为数据结构的改变以单一的写操作被记录到日志文件中；如果发生故障并且高速缓存区的内容丢失，文件系统的更改可以从日志文件恢复。而且，不同于 FAT 文件系统，NTFS 保证用户数据是一致的，并当执行写穿透或高速缓存刷新操作时可以立即获取用户数据，即便在后来系统出现故障也可以获取。

12.6.2　日志

NTFS 通过称做“日志”的事务处理技术提供文件系统的可恢复性。在 NTFS 日志中，任何改变重要的文件系统数据结构的事务的子操作在它们被传送到磁盘之前就被记录到日志文件。这样，如果文件系统崩溃了，部分完成的事务当系统重新工作时可以被重执行（redo）或撤消（undo）。在事务处理中，这种技术被认为是“预写日志”（write ahead logging）。在 NTFS 中，事务包括向磁盘写文件和删除一个文件并且可以由一些子操作组成。

1. 日志文件服务（LFS）

日志文件服务（LFS）是 NTFS 驱动程序中的一系列内核模式子程序，NTFS 利用驱动程序访问日志文件。尽管最初的设计为不只一个客户机提供日志和恢复服务，LFS 只被 NTFS 使用。假若这样的话，调用程序 NTFS 将一个指向打开文件对象的指针传递给 LFS，这就指定了将要访问的日志文件。LFS 或者初始化一个新的日志文件或者调用 Windows 2000 高速缓存管理器访问已存在的日志文件，如图 12-35 所示。

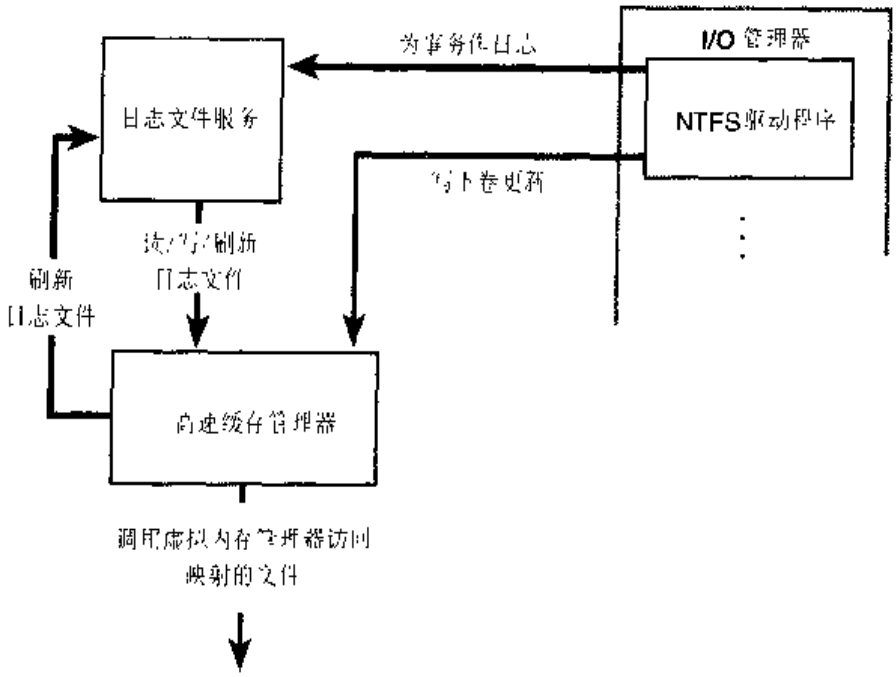


图 12-35 日志文件服务（LFS）

LFS 将日志文件分成两个区：重启动区域（restart area）和“无限的”日志区域（logging area），如图 12-36 所示。

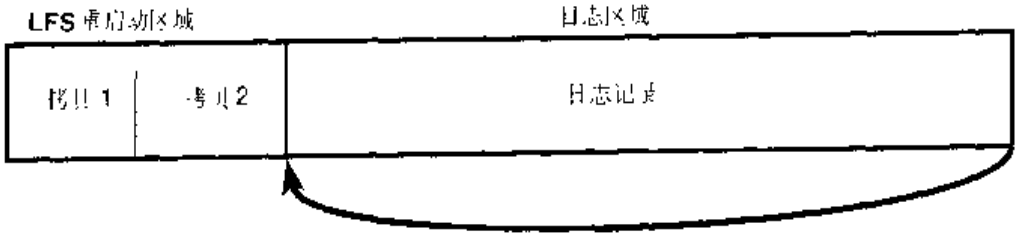


图 12-36 日志文件区

NTFS 调用 LFS 读写重启动区域，NTFS 用重启动区域存储环境信息——例如在文件系统出现故障后的恢复期间，NTFS 开始读的日志区的位置。为防止第一个被毁损或不能被访问，NTFS 保留了重启动数据（restart data）的第二个拷贝。日志文件剩余的部分是日志区域，它包含从一系统出现故障。NTFS 为了恢复一个卷所写的事务记录。LFS 通过循环地复用日志文件使日志文件看起来象是无穷的（在保证不会重写它需要的信息的同时）。LFS 使用逻辑序列号（LSN）识别写入日志的记录。当 LFS 循环整个文件时，LFS 增加了 LSN 的值。NTFS 用 64 位表示 LSN，这样可能的 LSN 数如此大以至于几乎是无穷的。

NTFS 从不直接从日志文件读事务或向日志文件写事务。LFS 向 NTFS 提供了服务，NTFS 调用这些服务来打开日志文件、写日志记录、以正向或反向次序读日志文件、刷新日志记录到一个特定的 LSN 或设置日志文件的开始为较高的 LSN 的服务。在恢复期间，NTFS 调用 LFS 完成下面的操作：正向读日志记录以重执行任何被记录在日志文件中但在系统出现故障时没有被刷新到磁盘上的事务；回读日志记录撤消事务或重新运行任何在系统崩溃前没有完全被记为日志的事务；并且当 NTFS 不再需要旧的日志文件中的事务记录时，将日志文件的开始设置为一

个有较高的 LSN 的记录。

下面说明了系统如何保证卷被恢复：

1）NTFS 首先调用 LFS 去记录日志文件中将要修改卷结构的任意事务。

2）NTFS 修改卷（也在高速缓存中）。

3）高速缓冲区管理器促使 LFS 将日志文件刷新到磁盘。（LFS 通过回调高速缓冲管理器实现刷新，告诉它哪个内存页需要刷新。回头查阅图 12-34 显示的调用序列）。

4）当高速缓存管理器将日志文件刷新到磁盘以后，它刷新卷的变化（元数据操作本身）到磁盘。

如果文件系统的修改最终是不成功的，这些步骤保证相应的事务可以在日志文件中获取，并且作为文件系统恢复过程的一部分可以被重执行或撤消。

当系统被重新引导后，卷第一次被使用时，文件系统开始恢复　NTFS 检查在系统崩溃前记录在日志文件中的事务是否被应用到卷，如果不是，它重执行这些事务。NTFS 也保证在系统崩溃前没被完全记录到日志的事务被撤消，这样它们不会在卷上出现。

2. 日志记录类型

LFS 允许它的客户向日志文件写任何类型的记录。NTFS 可以写许多类型的记录。在这里描述两种类型，更新记录（update record）和检查点记录（checkpoint record）。

更新记录　更新记录是 NTFS 写到日志文件的最通用的记录类型。每种更新记录包含两种信息：

■ 重做信息　在事务从高速缓存被刷新之前，如果系统发生故障，如何对卷再应用一个完全日志（“提交”）的事务。

■ 撤消信息　当系统发生故障时，如何反向一个只有部分日志（“不提交”）的事务的子操作。

图 12-37 显示了日志文件的三个更新记录。每个记录代表了一个创建一个新文件的事务的子操作。每个更新记录的重做项告诉 NTFS 如何对卷再应用子操作，并且撤消项告诉 NTFS 如何重新运行（撤消）子操作。

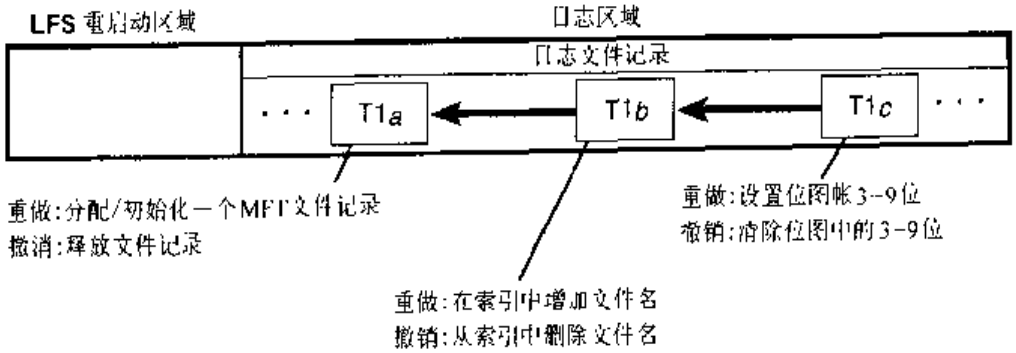


图 12-37　日志文件中的更新记录

当一个事务被记为日志之后（在此例中，通过调用 LFS 将一个更新的记录写到日志文件），NTFS 在高速缓存执行卷本身的子操作。当 NTFS 已经完成高速缓冲区的更新时，它将另一个完全记录了整个事务的记录写到日志文件——被认为是“提交（committing）”的事务的子操作。一旦事务被提交，即使后来操作系统发生故障，NTFS 也将保证整个的事务会在卷上出现。

当系统崩溃后进行恢复时，NTFS 遍历日志文件并重执行每个提交的事务。尽管在系统崩溃前，NTFS 完成了提交的事务，但它并不知道高速缓存管理器是否及时地将卷的修改刷新到磁盘上。当系统出现故障时，高速缓存的更新可能被丢失。因此，NTFS 仅仅为确保磁盘是最新的而再次执行提交的事务。

在文件系统恢复期间，当重新执行提交的事务时，NTFS 找到当出现故障时不被提交的日志文件的所有事务，并重新运行（撤消）已经记录为日志的每个子操作。在图 12-37，NTFS 将首先撤消 $T1_1$ 子操作，然后沿着逆向指针到 $T1_0$ 并撤消这个子操作。NTFS 将继续跟踪逆向指针撤消子操作，直到它达到事务的第一个子操作。通过跟踪指针，NTFS 知道有多少以及哪个更新记录必须撤消，然后重新运行事务。

重执行和撤消信息既可物理地描述也可逻辑地描述。物理描述根据不能被更改、删除等的磁盘上的特定字节的范围指定卷的更新。逻辑描述根据诸如“删除文件 A.dat”的操作表示卷的更新。由于软件的最底层保存文件系统结构，NTFS 用物理描述写更新记录。然而因为逻辑描述的更新记录比物理描述的更新记录更紧凑，所以事务处理或其他的应用级软件以逻辑术语写更新记录可能会获益。逻辑描述必须依赖 NTFS 去理解诸如删除一个文件涉及的操作。

NTFS 为下面的每个事务写更新记录（通常很多）：

- 创建一个文件。
- 删除一个文件。
- 扩展一个文件。
- 截断一个文件。
- 设置文件信息。
- 重命名一个文件。
- 改变应用到文件的安全性。

必须认真地设计更新记录的重执行和撤消信息，因为即使 NTFS 撤消事务、恢复系统故障、或者甚至是正常操作，它都有可能试图重执行一个已经被执行的事务，或正相反，去撤消从未发生的或已经被撤消的事务。类似的，NTFS 可能试图重执行或撤消由一些更新记录组成的事务，这些更新记录中只有一些在磁盘上是完全的。更新记录的格式必须保证执行冗余的重执行或撤消操作是“幂等的”（idem potent），也就是说，有一个中性的影响。例如，设置一个已设置的位没有什么影响，但切换一个已经切换的位则会有影响。文件系统也必须正确处理中间卷的状态。

检查点记录 除了更新记录，NTFS 定期向日志文件写检查点记录，如图 12-38 所示。

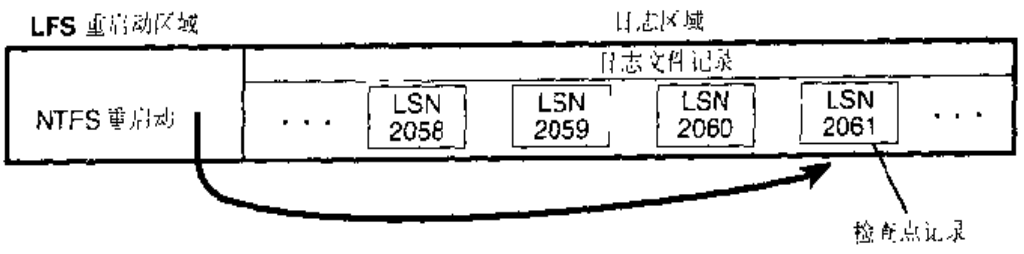


图 12-38 日志文件的检查点记录

如果立即发生系统崩溃，检查点记录帮助 NTFS 决定恢复一个卷所需要的过程。利用保存在检查点记录的信息，例如，NTFS 知道，日志文件要退回多久才能开始它的恢复。当写入一个检查点记录后，NTFS 在重启动区域保存记录的 LSN。这样，发生系统崩溃后，NTFS 开始文件系统恢复时能迅速找到最新写的检查点记录。

尽管 LFS 向 NTFS 展示的日志文件好象是无穷大的，其实并非如此。日志文件充足的大小和检验点频繁的写（通常在日志文件中释放空间的操作）使日志文件填满一个远程的 LFS 成为可能。但是，LFS 通过跟踪一些数字解释这种可能：

- 可用日志空间。
- 需要写一个进来的日志记录和取消写所需的空间数。
- 需要重新运行所有激活的（非提交的）事务所需的空间数。

如果日志文件不包含足够的可用空间满足最后两项，LFS 就返回一个“日志文件已满”错误，同时，NTFS 引发异常。NTFS 异常处理程序重新运行当前事务，并把它放在以后要重启动的队列中。

为释放日志文件所占空间，NTFS 必须即刻阻止对文件的进一步事务。为做到这点，NTFS 阻止文件的创建和删除，然后请求对所有的系统文件的专用访问和对所有用户文件的共享访问。逐步地，激活的事务要么完全完成，要么收到“log file full”异常。NTFS 重新运行并把收到异常的事务排队。

一旦 NTFS 已经阻塞以上描述的文件的事务活动，NTFS 就调用高速缓存管理器将未写的数据刷新到磁盘，包括未写的日志文件数据。当所有未写数据被安全刷新到磁盘后，NTFS 不再需要日志文件中的数据。NTFS 重新设置日志文件的开始到当前位置，使日志文件变为“空”，然后重新启动排队的事务。除了 I/O 过程的短暂暂停外，“日志文件已满”错误对正在执行的程序没有影响

这种情况是 NTFS 如何使用日志文件不仅进行文件系统恢复，而且在正常操作期间进行错误恢复的一个例子。在下面的部分你可以看到更多的关于错误恢复的细节。

12.6.3 恢复

NTFS 在系统被重新引导后程序第一次访问 NFTS 卷的时候自动执行磁盘恢复。（如果不需要恢复，这个过程是微不足道的）。恢复依赖于 NTFS 在内存中维护的两个表：

- 事务表记下已经开始但还没有提交的事务。这些激活的事务的子操作必须在恢复期间从磁盘删除。
- 脏页面表记录了在高速缓冲区的哪个页面包含没有被写入到磁盘的文件系统结构的修改。这个数据在恢复期间必须被刷新到磁盘。

NTFS 每 5 秒向日志文件写一个检查点记录。就在 NTFS 写之前，它调用 LFS 保存当前的事务表的复制和日志文件中脏页面表的复制。然后 NTFS 在检查点记录中记录包含复制表的日志记录的 LSN。当系统崩溃后开始恢复时，NTFS 调用 LFS 查找包含最新的检查点记录、最新的事务和脏页面表的复制的日志记录，然后将表复制到内存

日志文件通常在上一个检查点记录后包含更多的更新记录。这些更新记录表示当上一个检

查点记录被写入后卷的修改。NTFS必须更新事务和脏页面表以包含这些操作。更新这些表后，NTFS用这些表和日志文件的内容更新卷本身。

为了使卷的恢复有效，NTFS扫描3次日志文件，在首次最小化磁盘I/O期间将文件加载到内存，每次过程都有一个特定的目的：

1）分析。

2）重做事务

3）撤消事务。

1. 分析过程

在分析过程（analysis pass）期间，如图12-39所示，NTFS在日志文件中从上一个检查点操作的起始正向扫描找到更新记录，并用它们更新拷贝到内存的事务和脏页面表。注意图中检查点操作保存日志文件的三个记录，更新记录可能散布在这些记录中。因此NTFS必须在检查点操作的起始开始它的扫描

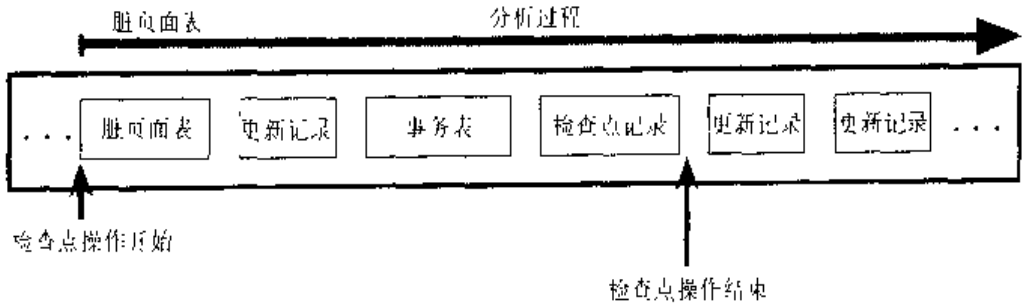


图 12-39 分析过程

当检查点操作开始后，出现在日志文件中的大多数更新记录要么表示对事务表的修改要么是对脏页面表的修改。例如，如果更新记录是一个“提交的事务”记录，记录表示的事务必须从事务表中删除。类似的，如果更新记录是修改文件系统数据结构的“页面更新”的记录，为反映这个变化，脏页面表必须被更新。

一旦这些表在内存被更新，NTFS就扫描这些表确定记录了磁盘上没有完成的操作的最老的更新记录的LSN。事务表包含未提交的（未完成的）事务的LSN和脏页面表，脏页面表包含没有被刷新到磁盘的高速缓存的记录的LSN。NTFS在这两个表中发现的最老的更新记录确定了在哪儿开始重做过程 如果最后一个检查点记录是老的，NTFS就会在那开始重执行过程。

2. 重做过程

在重做过程期间，如图12-40，NTFS在日志文件中从最老的更新记录的LSN开始正向扫描，

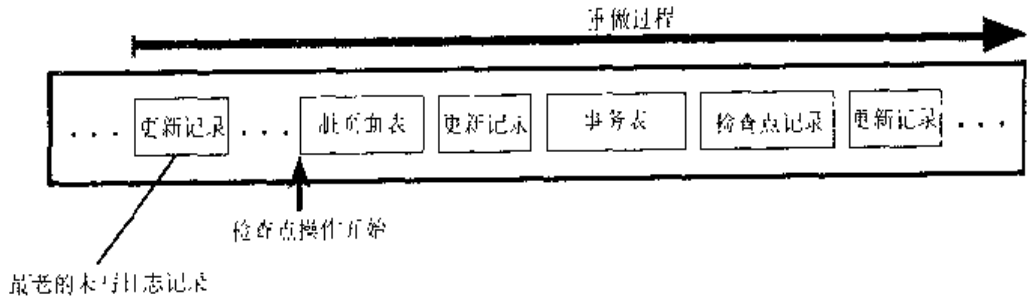


图 12-40 重做过程

最老的更新记录是 NTFS 在分析过程期间发现的。NTFS 寻找“页面更新”记录，它包含在系统出现故障前被写入的卷的修改，但是可能还没有被刷新到磁盘。NTFS 在高速缓存重新执行这些更新。

当 NTFS 到达日志文件的结尾时，它已经用必要的卷修改更新了高速缓存。高速缓存管理器的延迟书写器可以在后台开始将高速缓存的内容写到磁盘。

3. 撤销过程

NTFS 完成重做过程后，NTFS 就开始它的撤销过程操作，在撤销操作中，它重新运行当系统出现故障后任意一个没有被提交的事务。图 12-41 显示了日志文件的两个事务；事务 1 在电源出现故障前被提交，但事务 2 没有被提交。NTFS 必须撤销事务 2。

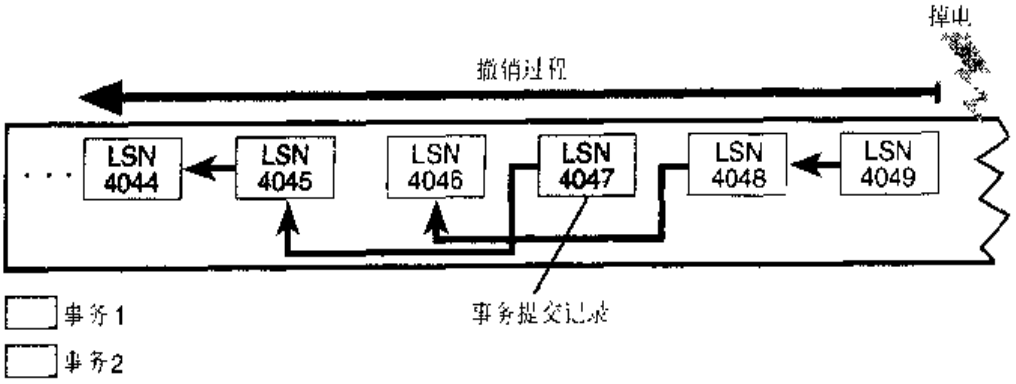


图 12-41　撤销过程

假定事务 2 创建一个文件，一个包括 3 个子操作的操作，每个子操作有它自己的更新记录。事务的更新记录是用日志文件的逆向指针链接的，因为更新记录通常是不相连的。

NTFS 事务表为每个未提交的事务列出了最后记为日志的更新记录的 LSN。在此例中，事务表将 LSN 4049 标记为为事务 2 最后记为日志的更新记录。如图 12-42 从左到右所示，NTFS 重新运行事务 2。

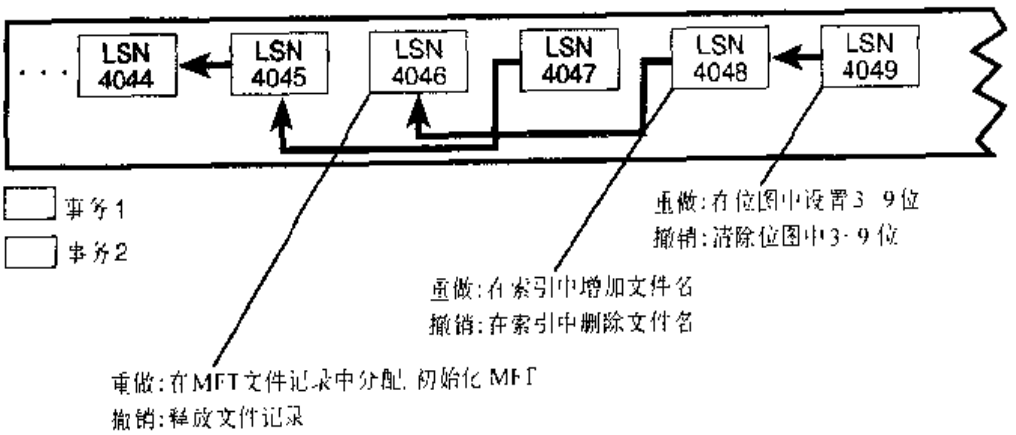


图 12-42　撤销一个事务

找到 LSN 4049 以后，NTFS 找到撤销信息并执行它，清除分配位图的第 3 到 9 位。NTFS 然后跟踪指向 LSN 4048 的逆向指针，它指导 NTFS 从合适的文件名索引中删除新的文件名。最后，当更新记录用 LSN 4046 指定时，NTFS 顺着逆向指针并释放为文件保留的 MFT 的文件记

录。现在重新运行事务2。如果存在要撤销其他的未提交的事务，NTFS遵循同样的方法重新运行它们。因为撤销事务影响卷的文件系统结构，NTFS必须在日志文件记录撤销操作。毕竟，在恢复期间，电源可能会再次出现故障，NTFS将不得不重新执行撤销操作。

当卷完成恢复的撤销过程时，它已经被重新保存为一致的状态。在这点上，NTFS通过刷新高速缓冲区的变化到磁盘来保证卷是更新的。然后，NTFS通过将一个“空”LFS写入重启动区域来指示卷是一致的。而且，如果系统立刻再次出现故障，无须执行恢复。恢复是完整的。

NTFS保证恢复会使卷返回到一些预先存在的一致状态，但是不必要返回到恰好为系统崩溃前存在的状态。NTFS不能保证这点是因为它用“延迟提交”算法提高性能，这意味着日志文件在每次“提交的事务”记录被写时，日志文件不立即被刷新到磁盘。取而代之的是，当高速缓存管理器调用LFS将日志文件刷新到磁盘或当LFS将检查点（每5秒一次）写到日志文件时，大量的“提交的事务”记录被成批执行和写。已恢复的卷可能没被完全更新的另一个原因是当系统崩溃时，一些并行的事务可能是激活的，并行事务中的一些“提交的事务”记录可能及时到达到磁盘而其他的不能。由恢复生成的一致的卷包括所有“提交的事务”记录已经写到磁盘的卷更新记录，但不包括“提交的事务”记录没写到磁盘的更新。

当系统出现故障后，NTFS用日志文件恢复一个卷，但也利用从日志事务得到的重要的“freebie”。文件系统必须包含许多用于恢复在正常文件I/O过程中发生的文件系统错误的代码。因为NTFS记录了每个修改卷结构的事务，当文件系统出现错误时，NTFS可用日志文件恢复，这样就大大简化了NTFS的错误处理代码。先前描述的“日志文件已满”错误是用日志文件进行错误恢复的一个例子。

程序收到的大多数I/O错误不是文件系统错误，因此不能完全被NTFS解决。例如，当调用NTFS创建文件时，NTFS可能开始在MFT创建一个文件记录，然后在目录索引中输入一个文件名。然而当NTFS设法在位图中为文件分配空间时，它可能发现磁盘是满的并且不能完成创建请求。在这种情况下，NTFS用日志文件的信息撤消它已经完成的那部分操作，并且释放它为文件保留的数据结构。然后，NTFS返回调用程序一个“磁盘满”的错误，调用程序反过来必须对错误作出适当的反应。

12.7 NTFS坏簇恢复

包含于Windows 2000的卷管理器FtDisk（对于基本磁盘）和逻辑磁盘管理器（LDM，对于动态磁盘）可以恢复在默认容错卷上的坏扇区的数据，但如果一个硬盘不用SCSI协议或没有备用的扇区，卷管理器不能完成扇区备份去替换坏扇区（关于卷管理器的更多信息见第10章）。当文件系统从扇区读取数据时，卷管理器不是恢复数据，而是向文件系统返回数据只有一个拷贝的警告。

FAT文件系统对卷管理器的警告不作出反应。并且，这些文件系统和卷管理器都不能记住坏扇区，因此一个用户必须运行Chkdsk或Format命令防止卷管理器对文件系统的数据重复恢复。用Chkdsk或Format删除不用的坏扇区都不理想。Chkdsk要用很长的时间才能发现和删除坏扇区，Format清除它格式化的分区中所有的数据。

在文件系统中，同卷管理器的扇区备份等价的是 NTFS 动态替代包含坏扇区的簇和跟踪坏簇以免重新使用。（回顾 NTFS 通过逻辑簇寻址而不是物理扇区寻址来维护可移植性）当卷管理器不能执行扇区备份时，NTFS 执行这些功能。当卷管理器返回一个坏扇区警告或当硬盘驱动程序返回一个坏扇区错误时，NTFS 就分配一个新簇替换包含坏扇区的簇。NTFS 通过复制卷管理器已经恢复到新簇的数据重建数据冗余。

图 12-43 显示在簇变坏之前，当用户文件退出时，在数据 run 中有坏簇的用户文件的 MFT 记录。当 MFT 记录接收到一个坏扇区错误时，NTFS 将包含扇区的簇重分配给它的坏扇区文件，这就防止了坏扇区被分配到另一个文件。然后 NTFS 为文件分配一个新簇，并改变文件的 VCN 到 LCN 映射为指向新簇的映射。这种坏扇区的重映射（在本章前面介绍的）如图 12-44 所示。包含坏扇区的簇号为 1357 的簇被簇号为 1049 的新簇替代。

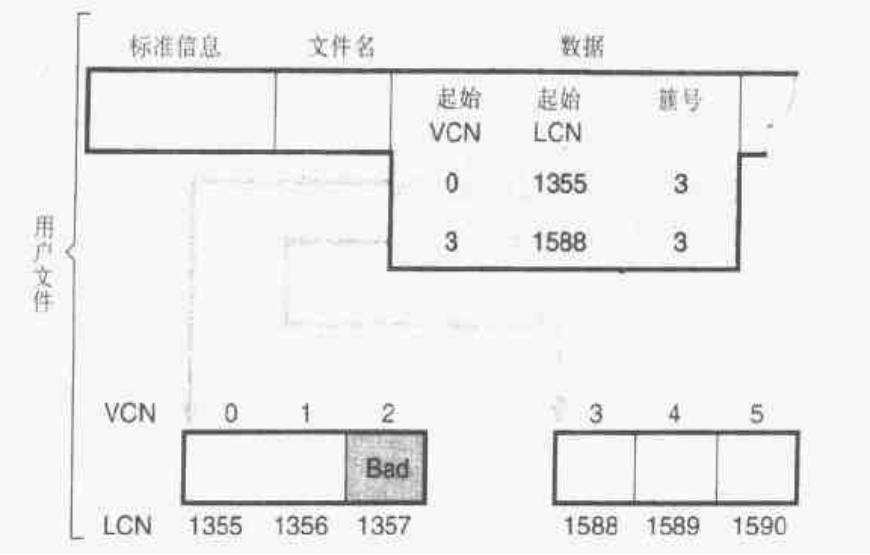


图 12-43 有坏簇的用户文件的 MFT 记录

坏扇区错误是不希望出现的，但当它们确实出现时，NTFS 同卷管理器相结合提供了最好的可能解决办法。如果坏扇区在一个冗余的卷上，卷管理器就恢复数据，如果可以的话就替换扇区。如果卷管理器不能替换扇区，就返回给 NTFS 一个警告，并且 NTFS 替换包含坏扇区的簇。

如果卷没有被配置为冗余的卷，那么坏扇区的数据不能被恢复。当卷被格式化为一个 FAT 卷，卷管理器不能恢复数据时，从坏扇区读取数据将导致不确定的结果。如果文件系统的一些控制结构保留在坏扇区，那么整个文件或成组文件可能会丢失（或者可能，整个磁盘会丢失）。充其量是受影响的文件数据被丢失（除了坏扇区以外的文件的全部数据）。而且，FAT 文件系统可能向卷上的相同文件或另一个文件重分配坏扇区，导致问题重新出现。

像其他的文件系统一样，没有卷管理器的帮助，NTFS 不能恢复坏扇区的数据。然而，NTFS 在很大程度上包含了坏扇区能引起的损坏。如果 NTFS 在进行读取操作期间发现了坏扇区，它重映射扇区所在的簇，如图 12-44 所示。如果卷没有被设置为一个冗余的卷，NTFS 返回调用程序一个“数据读取”错误。尽管在那个簇的数据丢失，文件的其余部分和文件系统仍旧是完整的；调用程序可以对数据丢失作出恰当的反应，坏簇在以后的分配中就不会被再使用。如果

NTFS 在写操作而不是读操作发现了坏簇，NFTS 在写之前重映射簇，这样既不会丢失数据也不会产生错误。

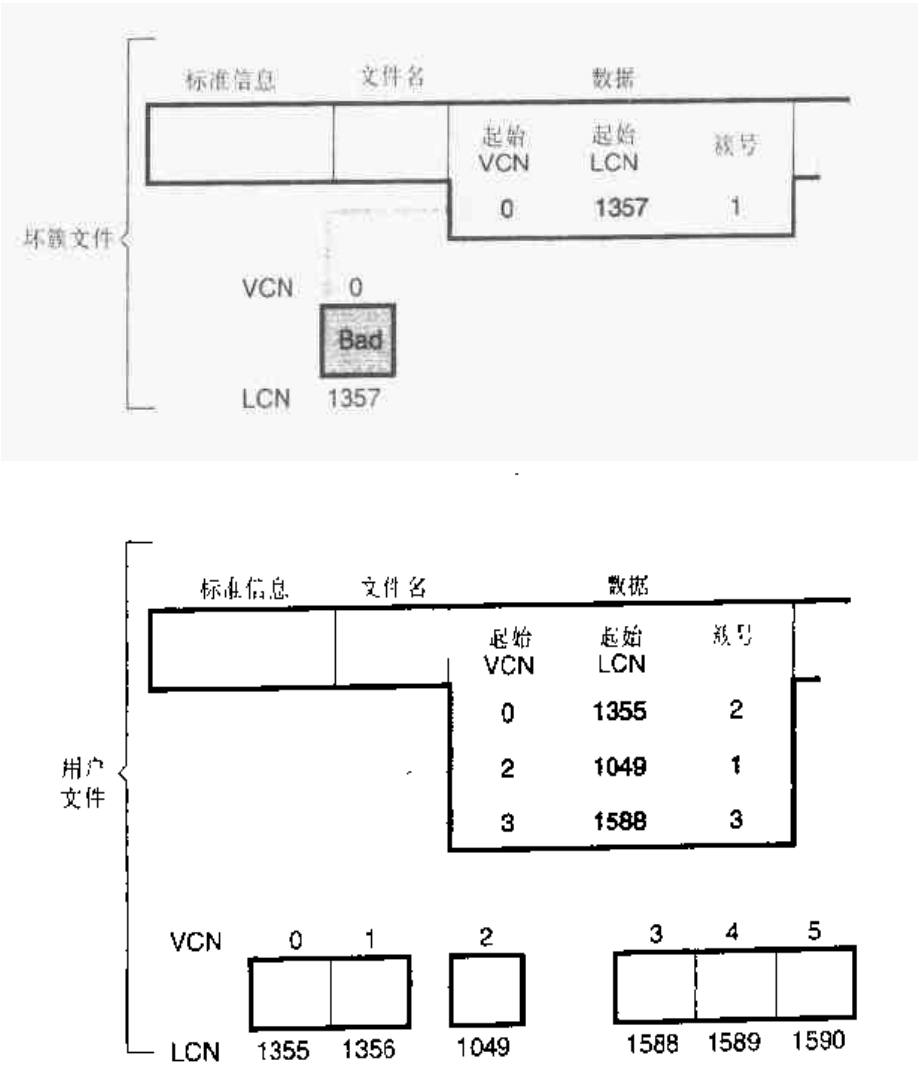


图 12-44 坏簇重映射

如果文件系统数据保存在一个坏扇区中，采取相同的恢复过程。如果坏扇区在一个冗余的卷上，NTFS 使用卷管理器恢复的数据动态地替换簇。如果卷不是冗余的，数据不能被恢复，NTFS 在卷文件上设置一个位显示卷上的毁损。当系统接下来被重引导时，NTFS Chkdsk 工具检验这个设置的位，如果位被设置，就执行 Chkdsk，并通过重构造 NTFS 元数据修补文件系统的毁损。

在少数情况下，甚至在容错配置下发生文件系统毁损。两个错误可以同时损坏文件系统数据和重构造数据的方法。当 NTFS 正写入 MFT 文件记录、文件名索引或日志文件的镜像拷贝时，如果系统崩溃，这时文件系统数据的镜像拷贝可能没被完全更新。如果系统被重引导并且在主磁盘上，同在磁盘镜像上未完全写入的相同位置出现坏扇区错误，NTFS 就不能恢复磁盘镜像的正确数据。NTFS 实现特定的方案检测这种文件系统数据毁损。一旦 NTFS 发现了不一致，它就在卷文件上设置毁损位，这样当系统下次被重引导时，Chkdsk 就会重新构造 NTFS 元数据。因为在容错磁盘配置下，文件系统毁损很少发生，所以很少需要用到 Chkdsk。Chkdsk

支持的是安全预防措施，而不是首选的（first-line）数据恢复策略。

在 NTFS 上，Chkdsk 的使用大大不同于它在 FAT 文件系统的使用。在向磁盘写任何事物前，FAT 设置卷的脏位，然后当修改完成后重设置位。如果当系统崩溃时，正在进行 I/O 操作，无效位是被留下的设置，当系统被重引导时，运行 Chkdsk。在 NTFS 上，只有当意外的或不能读取的文件系统数据被发现并且 NTFS 在单独的卷上不能恢复冗余的卷或冗余的文件系统结构上的数据时，才运行 Chkdsk（当 MFT 的部分需要引导系统和运行 NTFS 恢复过程时，系统的引导扇区才被复制。这样的冗余保证 NTFS 总能引导和恢复它本身）。

表 12-6 总结了根据在这部分已经描述的不同的条件，在 Windows 2000 所支持的文件系统中的格式化的磁盘卷上，当扇区变坏时所发生的情况。

表 12-6 NTFS 数据恢复情况总结

情况	具有备用扇区的 SCSI 磁盘	非 SCSI 磁盘或无备用扇区的磁盘[1]
容错卷[2]	1 卷管理器恢复数据	1 卷管理器恢复数据
	2 卷管理器执行扇区备份（替代坏扇区）	2 卷管理器向文件系统发送数据和坏扇区错误
	3 文件系统未意识到错误	3 NTFS 执行簇重映射
非容错卷	1 卷管理器不能恢复数据	1 卷管理器不能恢复数据
	2 卷管理器向文件系统发送坏扇区错误	2 卷管理器向文件系统发送坏扇区错误
	3 NTFS 执行簇重映射，数据丢失[3]	3 NTFS 执行簇重映射数据丢失

注：①在上述这些情况，卷管理器都不能执行扇区备份：1）不使用 SCSI 协议的硬盘不提供扇区备份用的标准接口；2）一些硬盘不提供扇区备份的硬件支持，如果许多扇区变坏，提供扇区备份的 SCSI 硬盘最终会用完备用扇区。

②容错卷是下面中的一个：镜像集或 RAID－5 集。

③在写操作中，没有数据被丢失；NTFS 在写之前重映射簇。

如果坏扇区出现的卷是容错卷（镜像映射或 RAID－5 卷）并且如果硬盘是支持扇区备份（没有用完备用扇区）的，那么你正在用的文件系统是 FAT 还是 NTFS 并不重要，卷管理器无须用户或文件系统的干预就替换坏扇区。

如果坏扇区位于不支持扇区备份的硬盘上，文件系统负责替换（重映射）坏扇区或（在 NTFS 的情况下）坏扇区所驻留的簇。FAT 文件系统不提供扇区或簇的重映射。NTFS 簇重映射有益于修补文件的坏点（bad spot）而不会损坏文件（或损坏文件系统），而且坏簇不会重分配到相同的或另一个文件。

12.8 文件系统安全加密

EFS 安全性依赖于 Microsoft 在 WindowsNT 4 引进的 Windows 2000 密码支持。文件第一次被加密时，EFS 给执行加密的用户帐号分配私用/公共密匙对供文件加密。用户通过 Windows Explorer 给文件加密。打开文件属性对话框，按 Adanced，然后选择 Encrypt Contents To Secure Data 选项，如图 12-45

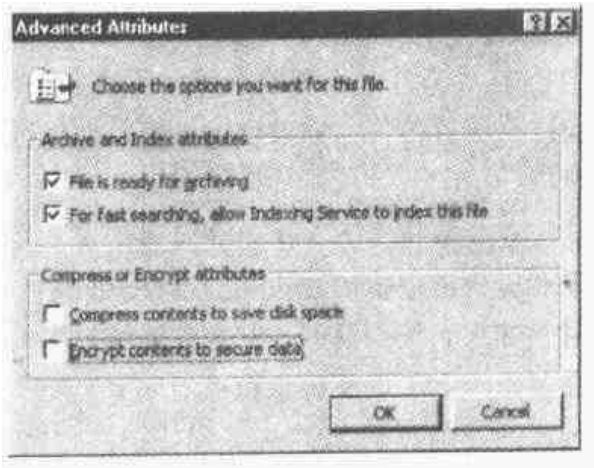


图 12-45 用 Adanced Attribute 对话框加密文件

所示。用户也可以通过称为“cipher”的命令行实用程序加密文件。Windows 2000 自动加密保留在被指定为加密目录下的文件。当文件被加密时，EFS 为 EFS 称之为“文件加密密匙”（FEK）的文件生成一个随机数。EFS 用 DESX 算法使用 FEK 加密文件的内容，DESX 算法是一个更强的数据加密标准（DES）算法的变种。EFS 把文件和文件的 FEK 存储在一起，但使用基于 RSA 公共密匙的加密算法用用户的 EFS 公用密匙来加密文件。当 EFS 完成这些步骤后，文件就是安全的了：其他用户若没有文件的 FEK 就不能解密数据，并且没有私用密匙不能解密 FEK。

EFS 用私用/公共密匙算法加密 FEK。EFS 用 DESX 来加密文件数据是因为 DESX 是对称的加密算法，这意味着它使用相同的密匙去加密和解密数据。对称加密算法速度尤其快，这使得它很适合大量数据的加密，例如对文件数据的加密。然而，对称加密算法也有一个弱点：如果获得密匙，你可以绕过安全保护。如果多个用户想要共享一个仅被 DESX 保护的加密文件，每个用户要请求访问文件的 FEK。显然，FEK 不被加密将成为一个安全问题，但是加密 FEK 一次就要求所有的用户共享同一个 FEK 加密密匙——它是另一个潜在的安全问题。

保证 FEK 的安全是一个很难的问题，EFS 通过基于它一半的加密体系结构公共密匙来解决该问题。为单个用户加密文件的 FEK 可以使多个用户共享一个加密文件。EFS 可以用每个用户的公共密匙加密文件的 FEK 并用文件保存每个用户的加密 FEK。任何人可以访问一个用户的公共密匙，但是没有人能用公共密匙解密用公共密匙加密的数据。用户能解密一个文件的唯一方法就是用他们的私用密匙。操作系统必须在一个安全的位置访问并保存私用密匙。用户的私用密匙解密文件的 FEK 的用户加密拷贝。Windows 2000 在计算机硬盘（不是非常安全的）上保存私用密匙，但是随后发布的操作系统使用户可以在便携式介质（例如智能卡）上保存他们的私用密匙。基于算法的公共密匙通常是很慢的，但 EFS 只用这些算法加密 FEK。分开公共可用密匙和私用密匙的使得密匙管理比对称加密算法要容易些，并解决了保证 FEK 安全的难题。

很多组件的共同工作使 EFS 得以工作，如图 12-46 所示的 EFS 体系结构。正如你所看见的，EFS 是作为运行在内核模式中的设备驱动程序实现的，并且和 NTFS 文件系统驱动程序有紧密联系。只要 NTFS 遇到一个加密文件，NTFS 就执行当 EFS 初始化时，EFS 驱动程序用 NTFS 注册的在 EFS 驱动程序中的函数。当应用程序访问加密文件时，EFS 函数加密和解密文件数据。尽管 EFS 用文件数据保存 FEK，用户的公共密匙可以加密 FEK。为了加密或解密文件数据，EFS 必须借助于保留在用户模式的密码服务来解密文件的 FEK。

本机安全权限子系统（*Lsass*—*\Winnt\System32\Lsass.exe*）管理登录会话，但也处理 EFS 密匙管理零碎工作。例如，为解密用户想要访问的文件数据，EFS 驱动程序需要解密一个 FEK，EFS 向 Lsass 发送一个请求。EFS 通过本机的调用过程（LPC）发送请求。KsecDD（*\\Winnt\System32\Drivers\Ksecdd.sys*）设备驱动程序向其他需要向 Lsass 发送 LPC 信息的驱动程序输出函数。

倾听远程过程调用（PRC）的 Lsass 的本机安全权限服务器（*Lsass*—*\Winnt\System32\Lsasrv.dll*）组件要求传递解密一个 FEK 的请求到合适的 EFS 相关的解密函数，解密函数也驻留在 Lsasrv 中。Lsasrv 用 Mircrosoft CryptoCAPI（也被引用为 CAPI）的函数解密 FEK，EFS 驱动程序以加密的形式将 FEK 发送到 Lsass。

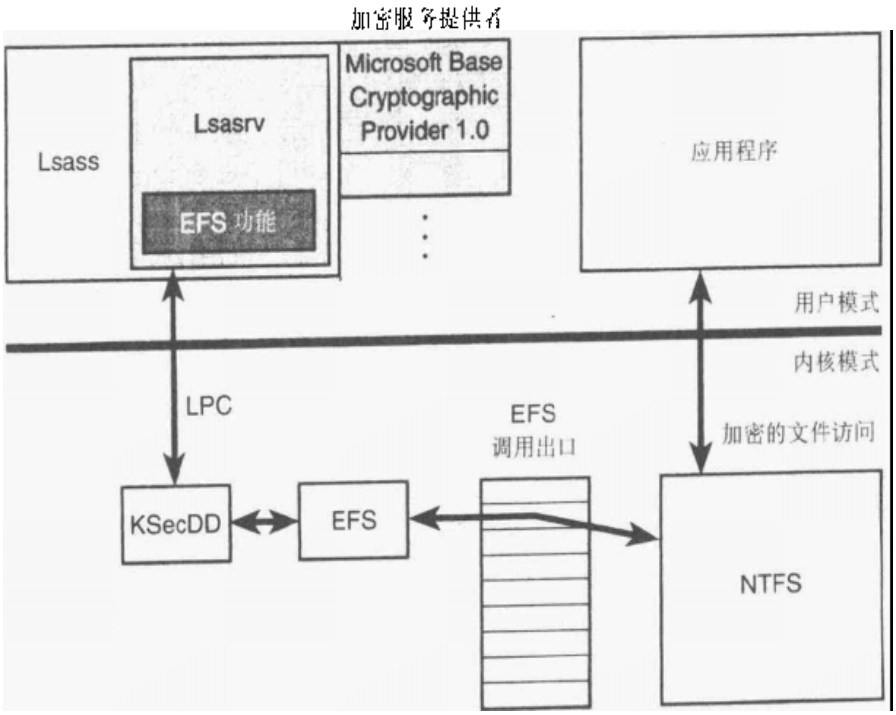


图 12-46 EFS 体系结构

CryptoCAPI 由加密服务提供者（CSP）DLL 组成，CSP DLL 使得应用程序可使用不同的密码服务（例如加密/解密和散列法）。例如，CSP DLL 管理用户私用和公共密匙的获取，这样，Lsasrv 无须关心密匙如何被保护的细节或甚至于加密算法的细节。当 Lsasrv 解密 FEK 后，Lsasrv 通过 LPC 回复信息向 EFS 驱动程序返回 FEK。当 EFS 接收到解密的 FEK 时，EFS 可以用 DESX 解密 NTFS 文件数据。让我们看一下 EFS 是如何与 NTFS 相结合的以及 Lsasrv 如何使用 CryptoAPI 管理密匙。

12.8.1 注册回调

NTFS 的执行并不要求 EFS 驱动程序的存在（*\Winnt\System32\Drivers\Efs.sys*），但如果 EFS 驱动程序不存在，加密文件不能被访问。NTFS 有一个 EFS 驱动程序的插件接口，这样，当 EFS 驱动程序初始化时，它可以附接于 NTFS。NTFS 驱动程序输出一些为 EFS 驱动程序使用的函数。其中包括这样一个函数，EFS 调用它通知 NTFS EFS 的存在和与 EFS 相关的 API EFS 是可用的。

12.8.2 第一次加密文件

NTFS 驱动程序只调用当 NTFS 遇到一个加密文件时注册的 EFS 函数。文件的属性记录了同文件记录被压缩（在本章前面讨论过）类似的加密方式。NTFS 和 EFS 有一些特定的接口用来把非加密文件转换为加密文件，但用户模式组件主要驱动这个进程。Windows 2000 允许你用两种方式加密文件：通过用 cipher 命令行实用程序或在 Windows Explorer 文件的 Advanced Attributes 对话框复选框中选取 Encrypt Contents To Secure Data。Windows Explorer 和 cipher 都依赖于 Adva-

pi32.dll（高级 Win32 API DLL）输出的 Win32 API 加密文件（EncryptFile）函数。Advapi32 加载另一个 DLL Feclient.dll（文件加密客户 DLL）获得 API，在 Lsasrv Advapi32 使用 API 通过 LPC 激活 EFS 接口。

当 Lsasrv 从 Feclient 接收加密文件的 LPC 信息时，Lsasrv 用 Windows 2000 模拟功能来模拟运行正在加密文件的应用程序（cipher 或 Windows Explorer）的用户。这个过程使得 Windows 2000 将 Lsasrv 执行的文件操作处理成好象是要加密文件的用户正在执行它们。Lsasrv 通常在系统帐号下运行。（系统帐号在第八章已描述）实际上，如果不模拟用户，Lsasrv 通常不允许访问所讨论的文件。

Lsasrv 接下来在卷的 System Volume Information 目录下生成一个日志文件，Lsasrv 把加密过程的进展记录在日志文件中。日志文件通常命名为 Efs0.log，但如果正在进行加密，日志文件就用递增的号替换 0 直到生成当前加密的唯一的日志文件名。

CryptoAPI 依赖用户的注册表的配置文件（profile）存贮的信息，这样 Lsasrv 下一步用 Userenv.dll（用户环境 DLL）的 *LoadUserProfile* API 函数加载配置文件（profile）到正在模拟的用户的注册表中。通常，用户配置文件已经被加载，因为当用户登录时，Winlogon 加载一个用户的配置文件。然而，如果用户用 Windows 2000 RunAs 命令登录一个不同的帐号，当你设法用这个帐号访问加密文件时，帐号的配置文件可能不会被加载。

然后 Lsasrv 用 Microsoft Base Cryptographic Provider 1.0 CSP 的 RSA 加密程序产生文件的 FEK。

1. 构造密匙环

在这点上，Lsasrv 有一个 FEK，并且能构造用文件保存的 EFS 信息，包括 FEK 加密的版本。Lsasrv 读取执行加密的用户的（*HEKY – CURRENT – USER \ Software \ Microsoft \ WindowsNT \ CurrentVersion \ EFS \ CurrentKeys \ CertificateHash*）键值去获得用户的公共密匙签名（注意这个密匙在注册中不会出现直到一个文件或文件夹被加密）。Lsasrv 使用这个签名访问用户的公共密匙和加密的 FEK。

Lsasrv 现在可以构造 EFS 用文件保存的信息。EFS 只保存加密文件的一块信息，这块信息包含共享文件的每个用户的项。这些项被称为密匙项（key entry），EFS 在文件的 EFS 数据的数据解密字段（DDF）部分保存它们。多个密匙项的集合被称为密匙环（key ring），因为，在前面提过 EFS 让多个用户共享加密的文件。

图 12-47 显示了文件的 EFS 信息格式和密匙项格式。EFS 在密匙项的第一部分保存足够的信息来精确描述用户的公共密匙。数据包括用户的安全 ID（SID）、密匙存储所在的容器名、加密提供者名和私用/公共密匙对的证书散列。密匙项的第二部分包含 FEK 的加密版本。Lsasrv 使用 RSA 算法和用户的公共密匙用 CryptoAPI 加密 FEK。

下一步，Lsasrv 创建了另一个包含恢复密匙项的密匙环。EFS 在文件的数据恢复字段（data recovery field，DRF）保存关于恢复密匙项的信息。DRF 项的格式同 DDF 项的格式是一致的。使用 DRF 的目的是当管理权限必须使用用户数据时，DRF 让指定的帐号或恢复代理（recovery agent）解密用户的文件。例如，假定一个公司雇员使用一个可以让他在智能卡保存私人密匙的 CryptoAPI，然后，他丢失了卡。如果没有恢复代理，就没有人能恢复他的加密数据。

恢复代理是根据本地计算机或域的加密数据恢复代理安全策略定义的。这个策略可以从

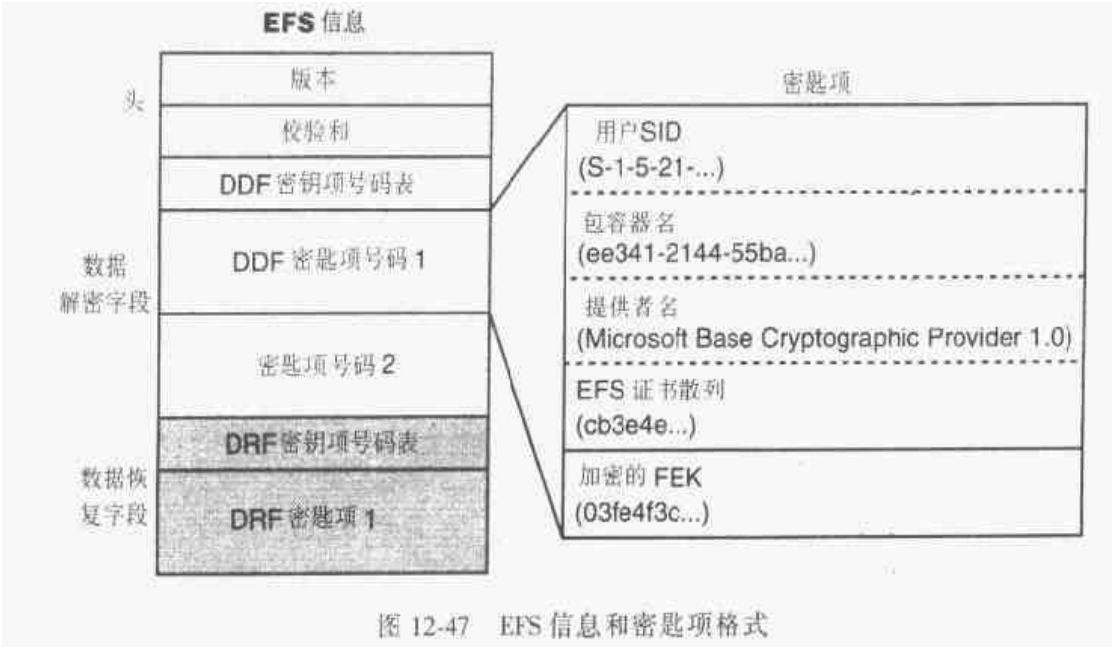


图 12-47　EFS 信息和密匙项格式

Group Policy MMC 插件获得，如图 12-48 所示。当你使用恢复代理向导（通过右击加密数据恢复代理并从 New 选项选择加密恢复代理）时，你可以添加恢复代理并指定为恢复 EFS 的恢复代理使用的是哪个私用/公共密匙对（由证书指定）。Lsasrv 在初始化并且接收到恢复策略已经改变的通知时解释恢复策略。EFS 通过使用为 EFS 恢复注册的加密提供者，为每个恢复代理创建一个 DRF 密匙项。默认恢复代理提供者是 Base Cryptographic Provider 1.0 RSA 加密程序——同 Lsasrv 对用户密匙使用的提供者一样。

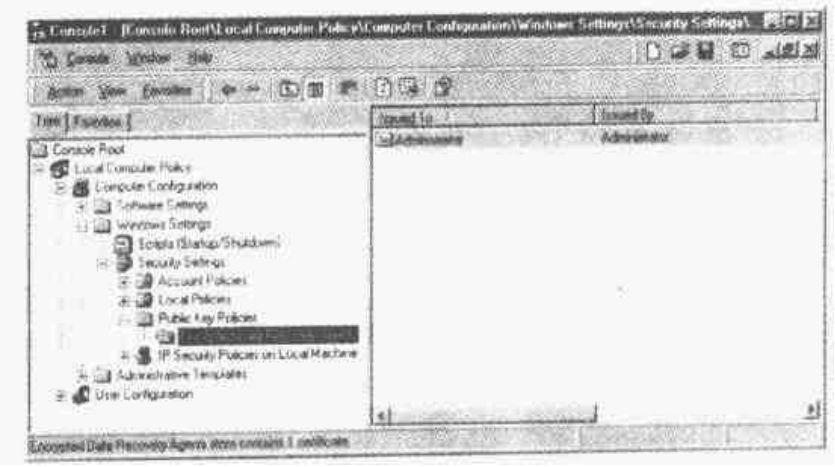


图 12-48　加密的数据恢复代理组策略

在为文件创建 EFS 信息的最后一步，Lsasrv 用 Base Cryptographic Provider 1.0 MD5 散列工具计算了 DDK 和 DRF 的校验和。Lsasrv 在 EFS 信息头保存了校验和的结果。在解密期间，EFS 引用这个校验和以确保文件的 EFS 信息内容没有被毁损或篡改。

2. 加密文件数据

图 12-49 说明了加密过程流程。Lsasrv 为一个用户想要加密的文件构造了必要的信息后，才能开始加密文件。Lsasrv 为正在进行加密的文件创建了一个备份文件 Efs0.tmp（如果其他备份文件存在，Lsasrv 用更高的号）。Lsasrv 在包含正在进行加密的文件目录下创建备份文件。

Lsasrv 对备份文件应用受限安全描述符，这样只有系统帐号才能访问文件的内容，Lsasrv 接下来初始化在加密过程第一阶段创建的日志文件。最后，Lsasrv 在日志文件记录了备份文件已经被创建。Lsasrv 只有当文件完全被备份后才加密初始文件。

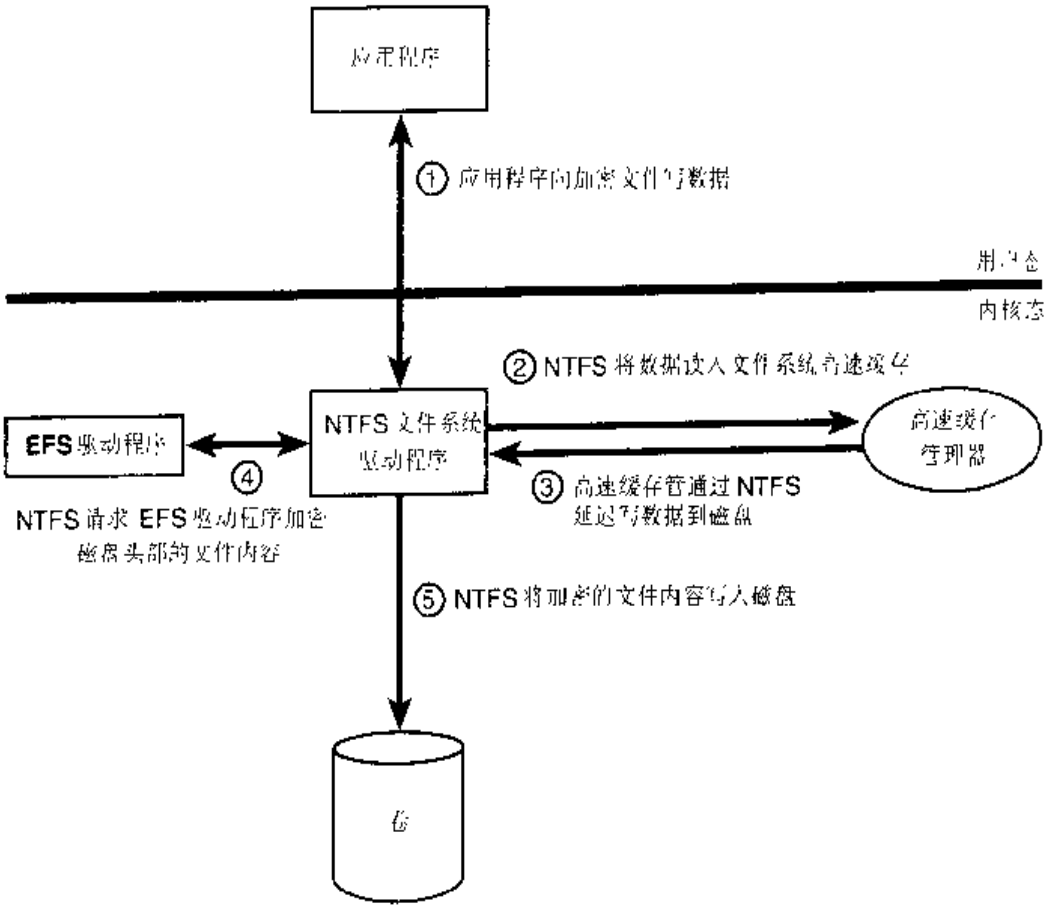


图 12-49 EFS 流程

Lsasrv 随后开始通过 NTFS 向 EFS 设备驱动程序发送一个命令，这个命令向初始文件添加 Lsasrv 刚创建的 EFS 信息。EFS 接收这个命令，但因为 NTFS 不理解 EFS 命令，所以 NTFE 调用 EFS 驱动程序。EFS 驱动程序接受 Lsasrv 发送的信息并使用 NTFS 输出函数将信息应用到文件输出的 NTFS 函数，NTFS 函数使 EFS 向 NTFS 文件添加了 $ LOGGED - UTILITY - STREAM 属性。执行过程返回到 Lsasrv，Lsasrv 将正在进行加密的文件内容复制到备份文件。当备份复制（包括所有可替换数据流的备份）完成后，Lsasrv 就在日志文件记录备份文件是更新的。然后 Lsasrv 向 NTFS 发送另一个命令让 NTFS 加密初始文件的内容。

当 NTFS 接收到加密 EFS 命令时，NTFS 删除初始文件的内容并将备份的数据复制到文件。当 NTFS 复制了文件每个扇区后，NTFS 刷新文件系统高速缓存扇区的数据，这促使高速缓存管理器通知 NTFS 将文件数据写入磁盘。因为文件被标记为加密的，在写文件过程中，在 NTFS 写数据到磁盘前，NTFS 调用 EFS 加密数据。EFS 用 NTFS 传递给它的未加密的 FEK 去执行文件的 DESX 加密，每次一个扇区（512 字节）。

在输出到美国以外的范围的 Windows 2000 版本中，EFS 驱动程序实现 56 位密匙 DESX 加密。在仅限于 U.S. 的 Windows 2000 版本中，密匙长达 128 位。

当 EFS 加密文件时，Lsasrv 在日志文件记录加密成功并删除文件的备份拷贝。最后，Lsasrv 删除日志文件并将控制返回到请求文件加密的应用程序。

3. 加密过程总结

下面列出了 EFS 执行加密一个文件的步骤的总结：

1）必要时，加载用户的配置文件。

2）日志文件在系统卷信息目录下被创建，并命名为 Efsx.log，这里 x 是唯一的数字（例如，Efs0.log）。当随后的步骤被执行后，记录被写入日志，这样，在加密过程中万一系统出现故障，文件仍可以被恢复。

3）Base Cryptographic Provider 1.0 为文件生成了一个随机的 128 位 FEK。

4）用户 EFS 私用/公共密匙对是被生成或获得的。*HEKY - CURRENT - USER\Software\Microsoft\WindowsNT\CurrentVersion\EFS\CurrentKeys\CertificateHash* 标别用户的密匙对。

5）DDK 密匙环是为有用户项的文件创建的。项包含用用户的 EFS 公共密匙加密的 FEK 的一个拷贝。

6）DRF 密匙环是为文件创建的。它对系统的每个恢复代理都有一个项，每个项包含恢复代理的 EFS 公共密匙加密的 FEK 一个拷贝。

7）当文件将要被加密时，以 Efs0.tmp 形式命名的备份文件在相同的目录下被创建。

8）DDF 和 DRF 密匙环被添加到头，并且作为 EFS 属性增加到文件中。

9）备份文件被标志为加密的，并且初始文件被拷贝到备份文件。

10）初始文件的内容被损坏，备份文件会拷贝到初始文件。这个拷贝操作导致了初始文件的数据被加密，因为文件现在已被标志为加密的。

11）备份文件被删除。

12）日志文件被删除。

13）（如果用户配置文件在第一步被加载）用户配置文件被卸载。

在系统加密过程中，如果系统崩溃，初始文件不会保持完整，备份文件也不会包含一致的拷贝。当系统崩溃后，Lsasrv 开始初始化，Lsasrv 在系统的每个 NTFS 卷的系统卷信息子目录下寻找日志文件。如果发现一个或更多的日志文件，便检查日志文件的内容并确定如何进行恢复。如果在系统崩溃的时候，初始文件没被修改，Lsasrv 就删除日志文件和相对应的备份文件；否则，Lsasrv 将备份文件复制到初始的部分加密的文件上，然后删除日志和备份。当 Lsasrv 处理日志文件后，就加密而言，文件系统将处于一个一致的状态，没有用户数据的丢失。

12.8.3 解密过程

当用户打开一个加密的文件时，就开始了解密过程。当 NTFS 打开文件时，它检查文件的属性然后执行 EFS 驱动程序的可调函数。EFS 驱动程序读取同加密文件相关联的 $ LOGGED-UTILITY-STREAM 属性。为读取这个属性，驱动程序调用 EFS 支持的函数，这个函数是 NTFS 输出的为 EFS 所用。NTFS 完成必要的步骤后打开文件。EFS 驱动程序保证打开文件的用户有权访问文件的加密数据（也就是说，在 DDK 和 DRF 密匙环中的加密的 FEK 对应于同用户相关联的私用/公共密匙对）。当 EFS 执行这个确认时，EFS 获得用户可能在文件上执行后来的数据

操作的文件加密的 FEK。

EFS不能解密 EFK，并依赖于 Lsasrv（Lsasrv 可以用 CryptoAPI）执行 FEK 加密。EFS 通过 Ksecdd.sys 驱动程序向 Lsasrv 发送一条 LPC 信息，让 Lsasrv 获得在 $ LOGGED - UTILITY - STREAM 属性数据（EFS 数据）中对应于正打开文件的用户的加密的 FEK 解密形式。

当 Lsasrv 接收到 LPC 信息后，如果配置文件没有被加载，Lsasrv 执行 Userenv.dll（用户环境 DLL）API 函数 *LoadUserProfile* 将用户的配置文件加载到注册表。Lsasrv 使用用户的私用密匙设法解密每个 FEK，并通过 EFS 的数据的每个密匙字段继续向下进行。对于每一个密匙，Lsasrv 试图解密一个 DDF 或 DRF 密匙项的 FEK。如果在密匙字段中的证书散列不引用用户所用的密匙，Lsasrv 就移动到下一个密匙字段。如果 Lsasrv 不能解密任何 DDK 或 DRF 密匙字段的 FEK，用户不能获得文件的 FEK。因此 EFS 拒绝打开文件的应用程序的访问。然而，如果 Lsasrv 识别出散列对应于用户所用者的密匙，Lsasrv 通过用 CryptoAPI 的用户私用密匙解密 FEK。

因为当 Lsasrv 解密 FEK 时，Lsasrv 既处理 DDK 密匙环又处理 DRF 密匙环，Lsasrv 自动执行文件的恢复操作。如果没有被注册访问加密的文件（也就是说，在 DDF 密匙环，恢复代理没有相应的字段）的恢复代理要设法访问一个文件，EFS 会让恢复代理获得访问，因为代理在 DRF 密匙圈已获得了密匙字段的密匙对。

1. 解密 FEK 高速缓存

从 EFS 驱动程序到 Lsasrv 再从 Lsasrv 回到 EFS 驱动程序的行程需要花相对来说很长的时间——在一个典型的系统上，解密一个 FEK 的过程中，CryptoAPI 的使用导致超过 2000 个注册表 API 的调用和 400 个文件系统的访问。EFS 驱动程序在 NTFS 的帮助下，用高速缓存设法避免这种花费。

2. 解密文件数据

当应用程序打开一个加密的文件后，应用程序可以读文件也可写文件。在 NTFS 将数据放置在文件系统高速缓存之前，当 NTFS 从磁盘读取数据时，NTFS 调用 EFS 驱动程序解密文件数据。类似的，当应用程序向一个文件写数据时，在文件系统的高速缓冲区的数据仍旧是未加密的形式直到应用程序或高速缓存管理器用 NTFS 将数据刷新回磁盘。当一个加密的文件的数据从高速缓存写回到磁盘，NTFS 调用 EFS 驱动程序加密数据。

正如前面强调的，EFS 驱动程序以 512 字节为单位执行加密和解密。512 字节的大小对驱动程序来说是最有利的，因为磁盘读写是在 512 字节倍数的扇区发生的。

12.8.4 备份加密的文件

任何文件加密功能设计的一个重要方面是不能获得未加密形式的文件数据，除了通过加密功能访问文件的应用程序。这个限制尤其影响了备份程序，在这里文档介质存储文件。EFS 通过为备份程序提供一个功能处理这个问题，这样程序可以以文件的加密状态备份和保存文件。因此，备份程序无须能够解密文件数据，也无须在文件数据的备份过程中解密文件数据。

备份程序用 Windows 2000 的新 EFS API 函数 *OpenEncryptedFileRaw*、*ReadEncryptedFileRaw*、*WriteEncryptedFileRaw* 和 *CloseEncryptedFileRaw* 访问文件的加密内容。Advapi32.dll 库提供这些 API 函数，这些 API 函数都用 LPC 激活 Lsasrv 中的相应函数。例如，在备份操作过程中，在备份程

序打开文件进行原始访问后，程序调用 *OpenEncryptedFileRaw* 获得文件数据。Lsasrv 函数 *EfsReadFileRaw* 向 NTFS 驱动程序发送控制命令（用 DESX 进行 EFS 会话密匙解密），先读取文件的 EFS 属性然后加密内容。

EfsReadFileRaw 要读取一个大文件，可能必须执行多个读操作。当 EfsReadFileRaw 读取大文件的每部分时，Lsasrv 向执行回调函数的 Advapi32.dll 发送 RPC 信息。当 Lsasrv 发送 *ReadEncryptedFileRaw* 时，由备份程序指定回调函数。*EfsReadFileRaw* 将它刚读的加密数据交给回调函数。回调函数可以向备份介质写数据。备份程序以类似的方式保存加密的文件。当 Lsasrv *EfsWriteFileRaw* 的函数正恢复文件的内容时，程序调用 API 函数 *WriteEncryptedFileRaw*，这就激活了备份程序中的回调函数从备份介质获得未加密的数据。

实验：查看 EFS 信息

EFS 有一些其他的 API 函数，应用程序能用它们处理加密文件。例如，应用程序用 API 函数 *AddUserToEncryptedFile* 使额外的用户访问一个加密文件，用函数 *RemoveUserFromEncrytedFile* 取消用户对加密文件的访问。应用程序用 *QueryUsersOnEncrytedFile* 函数获得与 DDK 和 DRF 密匙环字段相关联的文件的信息。*QueryUsersOnEncrytedFile* 返回 SID、证书散列值并显示每个 DDK 和 DRF 密匙字段包含的信息。当一个加密文件被指定为一个命令行参数时，下面是包括在配套 CD 下的*Sysint**Efsdump.exe* 的 EFDDump 程序的输出结果：

```
C:\>efsdump test.txt
EFS Information Dumper v1.02
Copyright (C) 1999 Mark Russinovich
Systems Internals - http://www.sysinternals.com

test.txt:
DDF Entry:
    SUSANCOMP\Joe:
        CN=Joe,L=EFS,OU=EFS File Encryption Certificate
DRF Entry:
    SUSANCOMP\Administrator
        OU=EFS File Encryption Certificate, L=EFS, CN=Administrator
```

你可以从中看出文件 text.txt 有用户 Joe 的一个 DDF 项和管理员的一个 DRF 项，后者是恢复代理在系统当前注册的唯一项。

12.9 小结

正如你在本章序言所见，NTFS 的首要目标是提供一个不仅可靠而且速度快的文件系统。Windows 2000 的磁盘 I/O 性能不只依赖于 NTFS 的实现。然而，在很大程度上依赖于 NTFS 和 Windows 2000 的高速缓存管理器的最佳协同作用。当为工作站和服务器系统提供史无前例的可靠性水平和高端数据存储特征时，NTFS 和高速缓存管理器实现了可观的 I/O 性能。

第13章 连网机制

微软 Windows 2000 的设计内嵌了网络的功能，它包含了与 I/O 系统和 Win 32 API 集成在一起的广泛的网络支持。网络软件的四种基本类型是服务程序、API、协议和网络适配器驱动程序，它们各自成层并叠置在相邻的层上共同组成一个网络栈。Windows 2000 为每层设计了良好的接口，因此除了应用 Windows 2000 自身携带的多种不同的 API、协议和适配器驱动程序外，第三方还可以通过开发自己的应用程序，扩展操作系统的网络功能。

本章将从顶端开始逐渐向底部推进，向你介绍 Windows 2000 网络栈。首先，我们首先向你介绍 Windows 2000 网络软件组件和开放系统互联（Open Systems Interconnection，OSI）参考模型的映射关系。然后，简单地介绍 Windows 2000 中应用的网络 API 和它们的实现方式。你将了解到网络资源名字解决方案的工作方式以及协议驱动程序如何实现。在掌握了网络适配器设备驱动程序后，我们将向你介绍绑定（binding），它是连接协议和网络适配器的纽带。最后，将要简要描述 Windows 2000 里包含的层状网络服务，例如 Active Directory 目录服务和文件复制服务（FRS）。

13.1 OSI 参考模型

网络软件的目的是从计算机的应用程序中接收到一个请求（通常是一个 I/O 请求），传递到另一台计算机，在远程计算机上执行请求，然后将结果返回到第一台计算机。在这个过程中，请求需要在沿途中被转换多次。一个高层次的请求，比如“从计算机 z 的文件 y 中读取 x 个字节”，需要软件来确定如何到达计算机 z 以及 z 能识别什么样的通信软件，然后为了在网络中传输，必须对请求进行变换——如，分割成短的信息包。当请求到达另一端时，必须对其进行完整性检查，解码，并发送到正确的操作系统组件中执行。最后，应答被编码后通过网络传送回来。

为了帮助不同的计算机制造商规范和集成它们的网络软件，1974 年，国际标准化组织（ISO）定义了一个计算机之间传递信息的软件模型，即 OSI。该模型分 7 层，如图 13-1 所示

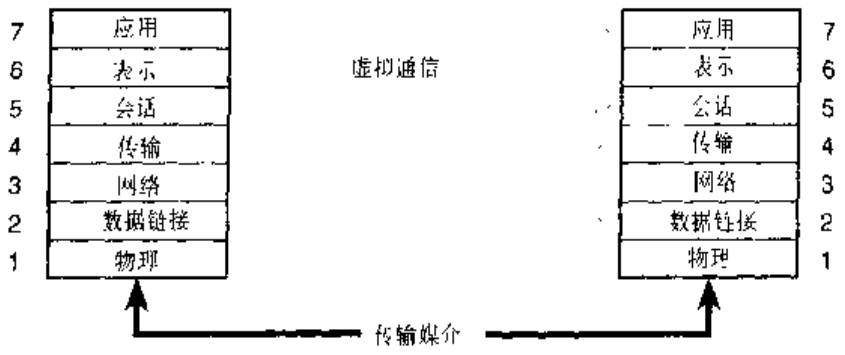


图 13-1 OSI 参考模型

OSI 参考模型是一个理想化的方案，很少有系统精确地实现了它，但是它常常作为网络原

理讨论的框架。一台计算机上的每一层都假设与另一台计算机上的同一层进行“对话”，两台计算机都在同一层次上“讲”同样的语言或协议。然而事实上，网络传输在客户机上必须向下传输到每一层，经过网络传递，然后再向上传输到目标机的各层，直到到达能够解释并执行请求的那一层。

13.1.1 OSI 层

OSI 模型中每一层的目的向比其更高的层提供服务，并且将在低一层完成的服务抽象化。详细地描述每层的功能不是本书所涉及的范围，但是在这里我们对不同的层进行简要的描述。

- 应用层：处理两个网络应用程序之间的信息传输，包括安全检查、参与的计算机识别和数据交换的初始化等功能；
- 表示层：处理数据格式化，包括象一行是以回车/换行符结束还是反以回车结束、数据是否被压缩、编码等等这类问题；
- 会话层：管理协作应用程序之间的连接，其中包括高层次的同步和对应用程序“讲话”、“接听”内容的监视；
- 传输层：在客户机一方，传输层负责将信息分成包，并分配给它们序列号，以保证所有的信息包都按正确的顺序被接收。在目标机上，传输层负责整理组装收到的信息包。传输层也用于屏蔽硬件变化对会话层的影响；
- 网络层：建立包头、处理路由、阻塞控制以及网际互连。它是理解网络拓扑的最高层，网络拓扑即网络中机器的物理设置、带宽限制等等；
- 数据链路层：传递低层次的数据帧，等待对它们已被接收的确认，并且负责将在不可靠的线路中丢失的数据帧再次发送；
- 物理层：将比特流传递至网络电缆或其他物理传输介质。

图 13-1 中←→代表将一个请求发送到远程计算机时使用的协议。如前所述，7 层中每一层都假设它与另一台计算机上的同一层对话，并且使用共同的协议。协议栈是指协议的集合，即通过协议栈，请求向下传输并且向上返回到网络中的各层。

13.1.2 Windows 2000 连网组件

图 13-2 是关于 Windows 2000 连网各组件的简图，图中表明每一个组件如何与 OSI 参考模型相匹配，以及各层之间使用什么样的网络协议。OSI 层与连网组件之间的映射并不精确，因此图中一些组件穿越层与层之间。组件包括以下几种：

- 网络 API 为网络通信应用程序提供一种独立于协议的途径，网络 API 能够在用户模式或同时在用户模式和内核模式下实现，有时网络 API 是另外一个实现了特定编程模型或提供附加服务的网络 API 的包装器（注意，网络 API 一词也指由与网络相关的软件提供的任何程序接口）。
- Transport Driver Interface（TDI）客户机（传输驱动程序接口客户机）是内核模式的设备驱动程序，通常用来完成网络 API 实现的内核模式部分。TDI 客户依据这样一个事实获取它们的名字，即它们发送到协议驱动程序的 I/O 请求包（IRP）按照 Windows 2000 传输驱

动程序接口标准（在 DDK 中有文档）被格式化。该标准为内核模式设备驱动程序提供一种统一的编程接口（参见第 9 章，了解更多关于 IRP 的信息）。

- TDI 传输（也被称作传输）网络驱动程序接口规范（NDIS）协议驱动程序，以及协议驱动程序，它们是内核模式协议驱动程序。它们从 TDI 客户接收 IRP，并且处理这些 IRP 代表的请求。这一过程可能需要与对等体进行网络通信，促使 TDI 传输为 IRP 中传输的数据添加协议特定的报头（如 TCP，NDP，IPX），并且利用 NDIS 函数与适配器驱动程序通信（在 DDK 中也有文档）。一般来说，TDI 传输透明地执行信息操作如分段、重新组装、序列化、确认接收以及重新传递等以实现应用程序的网络通信。
- NDIS 库（Ndis.sys）为适配器驱动程序提供封装性，从而隐藏 Windows 2000 内核环境的特殊性。NDIS 库输出供 TDI 传输使用的函数以及适配器驱动程序的支持函数
- NDIS 小端口驱动程序是内核模式驱动程序，负责 TDI 传输与特定网络适配器的接口。NDIS 小端口驱动程序包装在 windows 2000 NDIS 库中，这样封装提供了与 Microsoft Consumer Windows 相兼容的平台无关性。NDIS 小端口驱动程序不处理 IRP；但它们注册了一个 NDIS 库的调用表的接口，该接口包含指向对应于 NDIS 库输出到 TDI 传输的函数的函数的指针。NDIS 小端口驱动程序通过应用映射为 HAL 函数的 NDIS 库函数与网络适配器程序通信。

如图所示，OSI 层与实际的软件不对应。例如，TDI 传输经常跨越好几层。事实上，软件底部的四层常常合称为传输层，位于上面三层中的软件组件被称作“传输用户”。

在本章后面的部分，我们将检查图 13-2 中所示的各个连网组件（和图中未显示的其他组件），看它们是如何互相协调的，又是如何作为一个整体与 windows 2000 相关的。

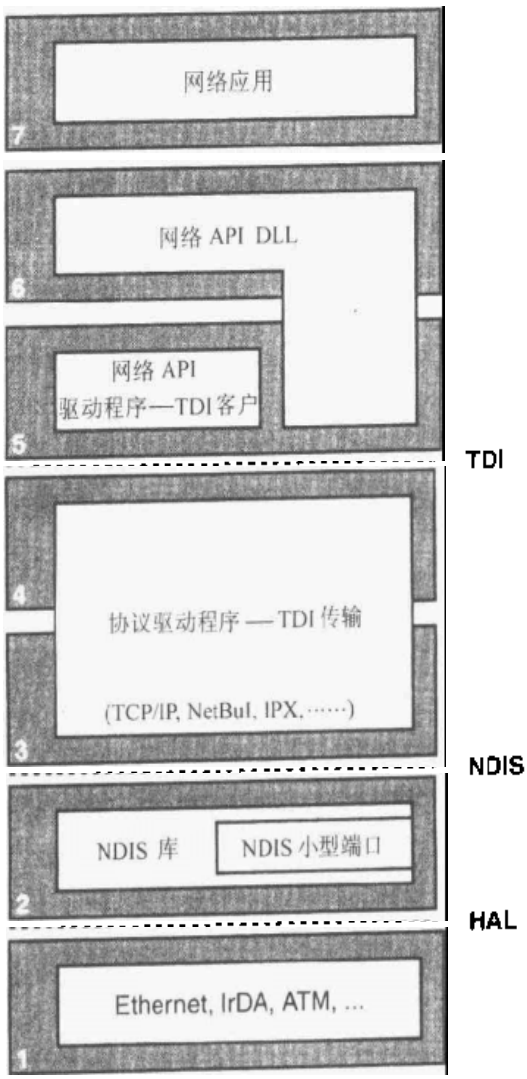


图 13-2 OSI 模型和 Windows 2000 连网组件

13.2 网络 API

Windows 2000 含有多种网络 API，既为以前的应用程序提供支持，又与工业标准兼容。在本节，将简单地浏览一下网络 API，描述应用程序如何使用它们。必须牢记：决定应用程序使用哪一个 API 取决于 API 的特性，比如该 API 能够使用哪一种协议，API 是否支持可靠的或双向通信以及 API 对应用程序可能运行于其中的

其他 Windows 平台的可移植能力。我们将着重讨论下面的网络 API：

- 命名管道和邮箱（mailslots）；
- Windows Socket（windock）；
- 远程过程调用（RPC）；
- 公共因特网文件系统（CIFS）；
- NetBIOS。

此外，还将简单地介绍几种建立在这些 API 之上并且在典型的 Windows 2000 系统上广泛应用的 API。

13.2.1 命名管道和邮箱

命名管道和邮箱是微软原来为 OS/2 LAN 管理器开发的编程 API，后来移植到 Windows NT 上。命名管道提供可靠的双向通信，而邮箱提供不可靠的单向数据传递。邮箱的一个优点是它们提供广播能力。Windows 2000 中，这两种 API 都充分利用了它的安全性，使服务器准确地控制哪一个客户机可以与之连接。

名字服务器分配给命名管道和客户机的名字遵从 Windows 2000 统一命名规则（UNC），它是一种在 Windows 网络上标识资源的协议无关的方法。UNC 名字的实现将在后面叙述。

1. 命名管道操作

命名管道通信由一个命名管道服务器和一个命名管道客户组成。命名管道服务器是一个创建客户可以与之相连的命名管道的应用程序，命名管道名字的格式为 \\Server\Pipe\PipeName。

其中名字的 Server 部分指定命名管道服务器正在运行的计算机（命名管道服务器不能在远程系统中创建命名管道），这个名字可能是一个 DNS 名字（如 mspress.microsoft.com）、NetBIOS 名字（mspres）或是 IP 地址（255.0.0.0），名字的 Pipe 部分必须是字符串“Pipe”，管道名是分配给命名管道的唯一名字。命名管道名字的这一独有部分可包含子路径，例如 \\MyComputer\Pipe\MyServerAPP\ComectionPipe。

命名管道服务器用 CreateNamedPipe Win32 函数创建命名管道。该函数的一个输入参数是指向命名管道的指针，形式为 \\.\Pipe\PipeName。“\\.\”是 Win32 为“这台计算机”定义的别名。函数接收的其他参数包括一个可选的安全描述符，它保护对命名管道的访问；一个标志，确定管道是双向或单向的；一个值参数表示管道支持的并发连接的最大数量，以及另外一个标志指示管道应该以字节模式还是消息模式运行。

多数网络 API 仅以字节模式运行，即随同一个发送函数发送的一条消息可能要求接收器接收多次，从碎片中建立完整的消息。用消息模式运行的命名管道应用程序使接收器的运行简单化，因为接收与发送之间是一对一的。一个接收器每次可以获得一条完整的消息，不必关心跟踪消息碎片。

对于一个特定的名字首次调用 CreateNamedPipe 函数即创建了该名字的第一个实例，并且建立了具有这个名字的所有命名管道实例的行为。服务器再次调用 CreateNamedPipe 函数就再次创建实例，直到实例数量达到首次调用时规定的最大数值。在至少创建了一个命名管道实例后，服务器运行 ConnectNamedPipe Win32 函数，使其创建的命名管道能够与客户相连。ConnectNamedPipe 可以同步或异步运行，直到客户与实例建立连接（或一个错误产生）函数才算运行完。

命名管道客户使用 Win32 CreatFile 或 CallNamedPipe 函数与服务器相连，它们确定服务器创建的管道的名字。如果服务器已经调用了 ConnectNamedPipe，客户的安全配置和它对管道的访问请求（读、写）对命名管道的安全描述符就是有效的（参见第 8 章关于 Windows 2000 应用的安全检查算法）。如果客户被授权访问命名管道，它就收到一个表示命名管道连接的客户方的句柄，服务器对 ConnectNamedPipe 的调用也就完成

在建立了命名管道连接后，客户与服务器可以利用 ReadFile 和 WriteFile Win32 函数从管道中读/或向管道中写数据。命名管道对信息传输支持同步和异步操作。图 13-3 表明客户与服务器通过命名管道实例通信的过程

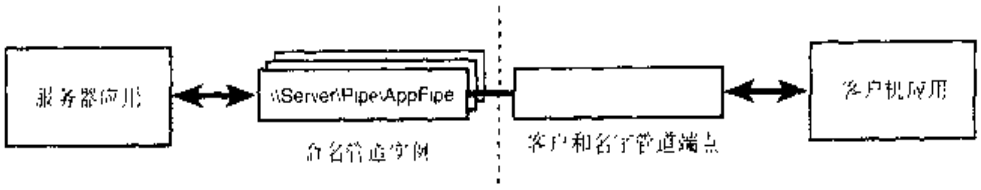


图 13-3 命名管道通信

命名管道网络 API 的独特之处在于它允许服务器通过使用 ImpersonateNamedPipeClient 函数模拟客户。详见第 8 章“模拟”一节，关于客户/服务器应用程序中如何使用模拟的讨论。

2. 邮箱操作

邮箱提供不可靠的单向广播机制 利用这种通信形式的应用程序实例之一是时间（time）同步服务程序，它可以每几秒钟在域中传播一个源时间。对网络中的每台计算机来说，接收源时间并不紧要，因此利用邮箱是一个很好的选择。

像命名管道一样，邮箱与 win32 API 集成在一起。邮箱服务器通过 CreatMailslot 函数创建一个邮箱。CreateMailslot 接收一个形如“\ \ . \ Mailslot \ MailslotName”的名字作为输入参数。同样与命名管道相同，邮箱服务器只能在正在运行的计算机上创建邮箱，而且它分配给 邮箱的名字中可以包含有子路径。CreateMailslot 也用一个安全描述符控制客户对邮箱的访问。CreateMailslot 返回的句柄是重叠的，这意味着在句柄上执行的操作如发送和接收消息是异步的。

由于邮箱是单向和不可靠的，CreateMailslot 函数不象 CreateNamedPipe 函数那样需要详多参数。CreateMailslot 函数创建邮箱后，服务器通过在表示邮箱的句柄上运行 ReadFile 函数很方便地倾听来访客户消息。

邮箱客户使用与命名管道客户相似的名字格式，但有所不同，这种差异使 邮箱客户可以在客户域或指定的域中将消息传递给指定名字的所有邮箱。为了将一条消息发送给一个特定的邮箱实例，客户机需要调用 CreateFile 函数，指定特定的计算机名字。例如“\ \ Server \ Mailslot \ MailslotName”（客户可指定 \ \ . \ 表示本地计算机）。如果客户想在它所属的域中获得表示给定名字的所有邮箱的句柄，它必须用“\ \ * \ Mailslot \ MailslotName”这种形式指定名字，如果客户想在不同的域中把句柄传递给指定名字的所有 邮箱，则使用的格式为“\ \ DomainName \ Mailslot \ MailslotName”。

获得表示邮箱客户端的句柄后，客户通过调用“WriteFile”发送消息，由于邮箱实现的方式，只能广播长度小于 425 字节的消息。如果消息长度大于 425 字节，邮箱实现就采用一种可靠的通信机制，它需要一对一的客户/服务器连接，这样就降低了广播能力。奇怪的是邮箱运行可使长度为 425 或 426 字节的消息截短到 424 字节。这些限制使邮箱通常不适合发送长度大

于424字节的消息。图13-4表示在域中一个客户机向多个邮箱服务器广播的过程。

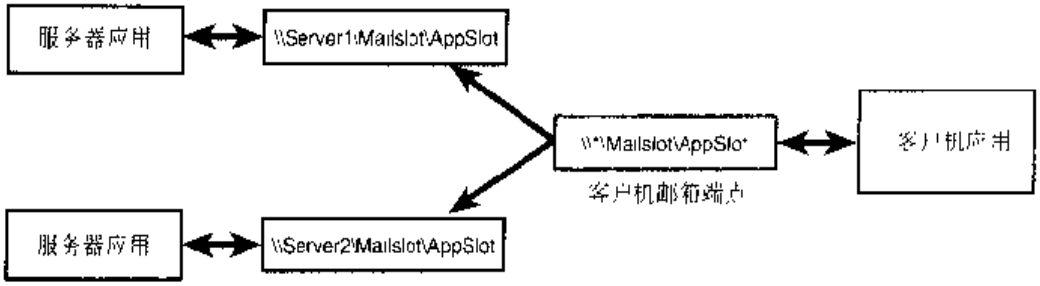


图13-4 邮箱通信

3. 命名管道与邮箱实现

命名管道与邮箱都与Win32集成在一起，因此这两种函数都在Kernel32.dll Win32客户端DLL中实现。ReadFile和WriteFile是应用程序利用命名管道和邮箱发送和接收消息的函数，它们是标准的Win32 I/O例程。CreateFile函数也是一个标准的win32 I/O例程，客户用它来打开命名管道或邮箱。然而，命名管道和邮箱应用程序指定的名字确定由命名管道文件系统驱动程序（\Winnt\System32\Drivers\Npfs.sys）和邮箱文件系统驱动程序（\Winnt\System32\Drivers\Msfs.sys）管理的文件系统名字空间，如图13-5所示。命名管道文件系统驱动程序建立一个名为\Device\NamedPipe的设备对象和名为\?\Pipe的符号链接，邮箱文件系统驱动程序创建一个名为\Device\Mailslot的设备对象和指向这个对象的名为\??\邮箱的符合链接（参见第3章有关\??对象管理器目录的解释）。传递给CreateFile的形如\\.\Pipe…和\\.\Mailslot…的名字的前缀\\.\被译成\\??\，这样名字通过符号连接解析为设备对象。特殊函数CreateNamedPipe与CreateMailslot使用对应的本机函数NtCreateNamedPipeFile和NtCreateMailslotFie。

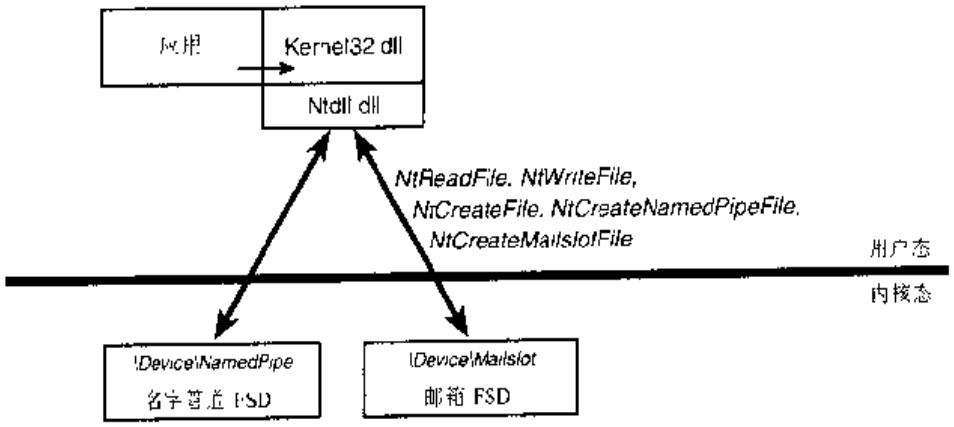


图13-5 命名管道与邮箱实现

本章后面，将讨论当一个指定远程命名管道或邮箱的名字映射到远程系统时，重定向器文件系统驱动程序是如何介入的。然而，当服务器创建或客户打开一个命名管道/邮箱时，这台服务器或客户机上合适的文件系统驱动程序（FSD）最终被激活。在内核模式FSD中实现命名管道和邮箱有许多原因，最主要的一个原因即是FSD与对象管理器名字空间集成在一起，并且能使用文件对象表示打开的命名管道和邮箱，这样集成有几个优点：

- FSD使用内核模式安全函数为命名管道和邮箱实现标准的Windows 2000安全性。
- 由于FSD与对象管理器名字空间集成在一起，应用程序可以利用Creat File函数打开命

名管道或邮箱。

- 应用程序能够利用 Win32 函数如 ReadFile、writeFile 等与命名管道和邮箱相互作用。
- FSD 依赖于对象管理器对表示命名管道和邮箱的文件对象进行句柄跟踪和引用计数。
- FSD 可以实现它们自己的命名管道和邮箱名字空间，用于目录完成。

由于命名管道和邮箱名字解析使用重定向器 FSD 在网络中进行通信，因此它们间接地依赖于 CIFS 协议（后面介绍）。CIFS 通过使用 IPX、TCP/IP 以及 Net BEUI 协议工作，因此运行在至少含有其中一个协议的系统中的应用程序可以使用命名管道和邮箱。

实验：列举命名管道名字空间，并且查看命名管道活动

使用 Win32 API 打开命名管道 FSD 的根目录并且显示目录是不可能的，但是你可以通过使用本机 API 服务程序来做到。Pipelist 工具（配套 CD 盘上 \ Sysint \ Pipelpst.exe）可以显示定义在计算机上的 命名管道的名字、为名字建立的实例数量以及服务器调用 CreateNamedPipe 定义的实例最大数量。下面是一个 Pipelist 输出的例子：

```
C:\>pipelist
PipeList v1.01
by Mark Russinovich
http://www.sysinternals.com

Pipe Name                                  Instances      Max Inst
---- ----  --------
InitShutdown                                   2             -1
lsass                                          3             -1
ntsvcs                                        32             -1
scerpc                                         2             -1
net\NtControlPipe1                             1              1
DhcpClient                                     1             -1
net\NtControlPipe2                             1              1
Winsock2\CatalogChangeListener-184-0           1              1
net\NtControlPipe3                             1              1
spoolss                                        2              1
net\NtControlPipe4                             1              1
net\NtControlPipe5                             1              1
net\NtControlPipe0                             1              1
net\NtControlPipe6                             1              1
net\NtControlPipe7                             1              1
winreg                                         2             -1
Winsock2\CatalogChangeListener-208-0           1              1
atsvc                                          2             -1
net\NtControlPipe8                             1              1
SecondaryLogon                                 1             10
SfcApi                                         2             -1
ProfMapApi                                     2             -1
winlogonrpc                                    2             -1
Winsock2\CatalogChangeListener-d8-0            1              1
epmapper                                       2             -1
POLICYAGENT                                    2             -1
WMIEP_e4                                       2             -1
WMIEP_1d4                                      2             -1
tapsrv                                         2             -1
ROUTER                                         9             -1
WMIEP_3c4                                      2             -1
WMIEP_3c0                                      2             -1
WMIEP_3ec                                      2             -1
```

从输出可看出，一些系统组件使用命名管道作为它们的通信机制。例如，Winlogon 创建的 InitShutdown 管道用来接收远程关闭命令，Runas 服务程序创建的 SecondaryLogon 管道代表 Runas 实用工具执行登录操作。通过使用 HandleEx（配套 CD 上 \ Sysint \ Handleex.exe）中的对象搜索工具，你可以确定什么进程有管道打开。注意，Max Instance 的值为 -1 意味着对于给定的名字，其 Instance 的数量没有上限。

Filemon 文件系统过滤器驱动程序（配套 CD 上的 \ Sysint \ Filemon.exe）能够附接到 Npfs.sys 或 Msfs.sys 文件系统驱动程序上，因此可以观看系统中发生的所有命名管道或邮箱活动。从 Filemon 驱动程序菜单中选择 NamedPipe 或 Mailslot 菜单项，将 Filemon 附接到相应的驱动程序上。图 13-6 显示当打开桌面上 My Network Places 图标时，Filemon 捕获产生的命名管道活动的情况。注意这些消息是通过 Lsass 或工作站服务命名管道传输的。

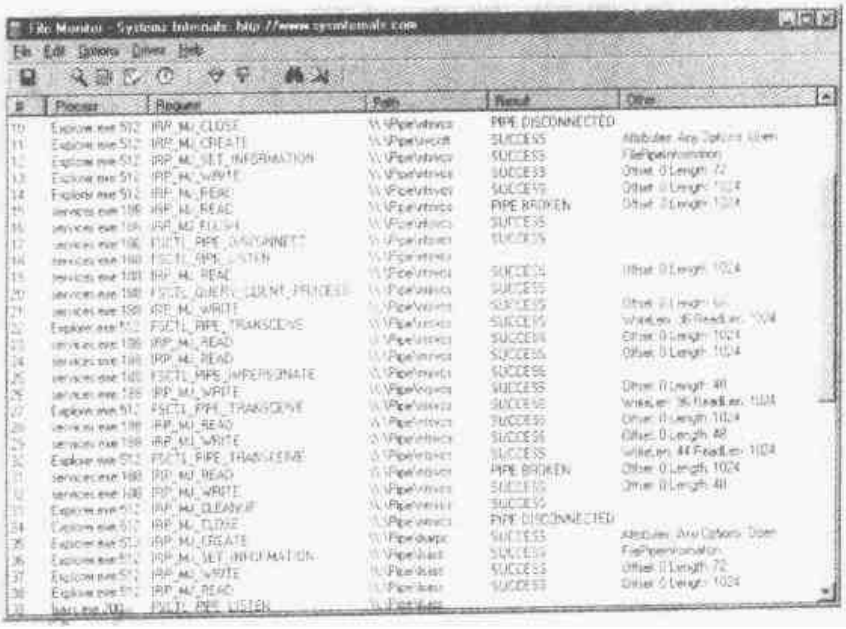


图 13-6

13.2.2 Windows Socket

Windows Socket（Winsock）是微软对 BSD（Berkeley Software Distribution）Socket 的实现，自从 20 世纪 80 年代 UNIX 系统使用在因特网上的网络通信以来，BSD Sockets 就成为了一个标准化的编程 API。Windows 2000 对套接字的支持使得 UNIX 网络应用程序向 Windows 2000 的移植变得相对直观。Winsock 不仅包含 BSD Sockets 的大部分功能，而且含有 Microsoft 特有的并且仍在继续发展的增强功能。Winsock 支持可靠的面向连接的通信以及不可靠的无连接通信。Windows 2000 提供 Winsock2.2，同样地它或者与所有版本的 Consumer Windows 包括在一起，或者作为附加产品使用。

Winsock 包括如下的特点：

- 支持分散/集中和异步应用程序 I/O。
- 遵循服务质量（quality of Service，QoS）约定，以至于当底层网络支持 QoS 时应用程序可

以协商延迟益和带宽要求

- 除了 Windows 2000 要求它支持的扩展性外，Winsock 具有的扩展性使它可以与协议一起使用。
- 除应用程序与 Winsock 正在使用的协议定义的名字空间外，Winsock 支持集成化名字空间。例如服务器可以在 Active Directory 中公开它的名字，并且利用名字空间扩展，客户可以在 Active Directory 中查找服务器地址。
- 支持同时传送到多个接收器的多点消息。

我们将观察典型地 Winsock 操作，并描述 Winsock 扩展的方式。

1. Winsock 操作

在调用一个初始化函数对 Winsock API 实行初始化后，Winsock 应用程序采取的第一步是创建一个套接字，它将表示通信端点。套接字必须绑定到本地机上的一个地址，因此应用程序执行的第二步是绑定，Winsock 是一个协议独立的 API，因此一个地址可以指定给 Winsock 运行的系统中安装的任何协议（NetBEUI，TCP/IP，IPX）。绑定完成后，服务器和客户执行的步骤分开，与面向连接和无连接套接字操作执行的步骤一样。

面向连接的 Winsock 服务器在套接字上执行倾听（listen）操作，显示它为套接字支持的连接量。然后它执行接收操作，允许客户与套接字相连。如果有一个暂停连接请求，则接收调用立即结束；否则当连接请求到达时它才结束。连接建立后，accept 函数返回一个新的套接字，它表示连接的服务端。服务器通过使用函数 recv 和 send 可以执行接收和发送动作。

面向连接的客户通过指定远程地址的 Winsock connect 函数与服务器相连。连接建立后，客户可以通过它的套接字发送和接收信息。图 13-7 表示 Winsock 客户与服务器之间的面向连接通信。

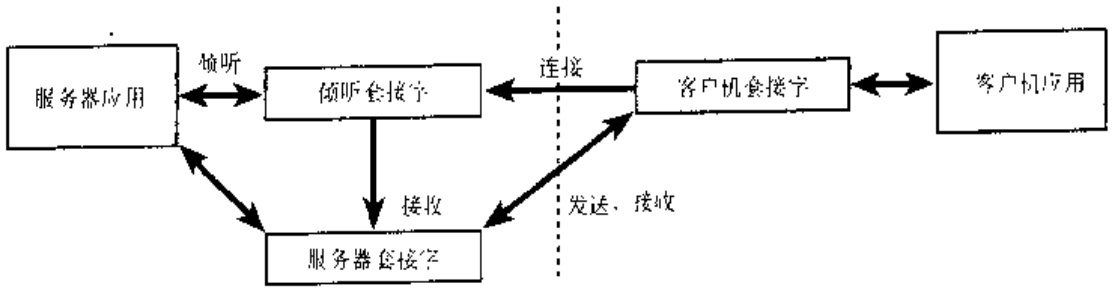


图 13-7 面向连接 Winsock 运行

绑定一个地址后，无连接服务器与无连接客户没有区别：它只要用每条消息指定远程地址，就可以通过套接字发送和接收消息。当使用无连接消息，也称作数据报文时，如果发送方获得一个错误代码，它就知道消息没有接收到，则执行下一次接收操作

2. Winsock Extensions

从 Windows 编程角度来看，Winsock API 突出的特点是它集成了 Windows 消息。Winsock 应用程序可利用这个特征实现异步套接字操作，并且通过标准 Windows 消息函数或执行回调函数来接收操作完成的通知。这一功能简化了 Windows 应用程序的设计，因为这样一来，应用程序不必是多线程的或管理同步对象，以至于既执行网络 I/O，又要响应来自窗口管理器的用户输入或请求来更新应用程序窗口。BSD 风格的 Winsock 函数基于消息的版本名字通常以 WSA 作为名字前缀 —例如 WSAAccept。

除了支持与 BSD Sockets 中实现的函数相对应的支持函数外，微软也增加了一部分不属于 Winsock 标准函数的函数。其中有两个函数是值得介绍的，即 AcceptEx 和 TransmitFile，因为 Windows 2000 上许多 Web 服务器都使用它们获得较高性能。AcceptEx 函数是 accept 函数的一个版本，在与客户建立连接的过程中，它返回客户地址和客户的第一条消息。有了这个函数，Web 服务器避免执行多个 Winsock 函数，否则将是必须的。

建立与客户的连接后，，Web 服务器通常发送一个文件给客户，如一个网页。TransmitFile 函数的实现是与 Windows 2000 高速缓存管理器集成在一起的，这样客户可以直接从文件系统高速缓存器中发送文件，用这种方式发送数据称作零拷贝，因为服务器不必接触文件数据就发送它；它只需简单地确定一个文件句柄和发送文件的范围，就可以发送。除此之外，TransmitFile 允许服务器预先考虑或添接数据到文件数据，这样服务器可以发送文件头信息，其中可能包含有 Web 服务器的名字和表示服务器发送的信息大小的字段。Internet Information Service（IIS）5.0 使用 AcceptEx 和 TramsmitFile 函数，它与 Windows 2000 捆绑在一起。

3. 扩展 Winsock

Windows 2000 中 Winsock 是一个可扩展的 API，因为第三方增加一个传输服务提供程序（transport service provider）使 Winsock 与其他协议建立接口以及增加一个名字空间服务提供程序（namespace service provider）来增强 Winsock 的名字解析性能。服务提供程序利用 Winsock 服务供应接口（SPI）插入 Winsock 中。当一个传输服务提供程序用 Winsock 注册时，Winsock 利用传输服务提供程序来实现套接字函数，如 connect 和 accept 函数。关于传输服务提供程序如何运行函数没有任何限制，但是实现过程常常涉及到与内核模式中传输驱动程序的通信。

任何 Winsock 客户/服务器应用程序都有一个要求，即对于服务器来说，要使它的地址对客户是可用的，这样客户才能与服务器建立连接。运行在 TCP/IP 协议下的标准服务程序使用“众所周知的地址”来实现地址有效，只要浏览器知道 Web 服务器正在运行的计算机的名字，它就可以通过指定这个服务器地址连接到 Web 服务器（服务器的 IP 地址与用于 HTTP 的端口号：80 相连）。名字空间服务提供程序使服务器可以以其他方式注册。例如，一个名字空间服务提供程序可能在服务器方的 Active Directory 中注册服务器地址，在客户方的 Active Directory 中查找服务器地址。名字空间服务提供程序通过实现标准的 Winsock 名字解析函数，如 gethostbyaddr、getservbyname 以及 getservbyport，为 Winsock 提供这一性能。

4. Winsock 实现

Winsock 的实现过程如图 13-7 所示。它的应用程序接口由一个 API DLL，Ws2-32.dll（\Winnt\Systerm32\Ws2-32.dll）组成，它提供应用程序访问 Winsock 函数的接口。Ws2-32.dll 调用名字空间服务程序和传输服务提供程序来执行名字和消息操作。对于微软提供的支持 Winsock 的协议来说，Msafd.dll 库函数就像一个传输服务提供程序，而且 Msafd.dll 使用 Winsock Helper 库，它是专门与内核模式协议驱动程序通信的协议。例如，Wshtcpip.dll 是 TCP/IP 协议的帮助程序，Wshnetbs.dll 是 NetBEUI 协议的帮助程序。Mswsock.dll（\Winnt\System32\Mswsock.dll）实现 Microsoft Winsock 扩展函数，如 TransmitFile、AcceptEx 和 WSARecvEx。Windows 2000 载有 *TCP/IP*、*NetBEUI*、AppleTalk、IPX/SPX、ATM 以及 IrDA（远红外数据协会）的帮助程序 DLL 和 DNS（TCP/IP）、Active Directory 以及 IPX/SPX 名字空间服务提供程序。

实验：查看 Winsock Service Provider

Platform SDK 包含的 Windows Sockets Configuration 实用程序（Sporder.exe）显示了注册的 Winsock 传输和名字空间提供程序，并且允许改变枚举传输服务提供程序的顺序。例如，如果有两个 TCP/IP 传输服务提供程序，第一个列出的是使用 TCP/IP 协议的 Winsock 应用程序的缺省提供程序。如图 13-8 是一个来自于 Sporder 的屏幕快照，表示了注册的传输服务提供程序。

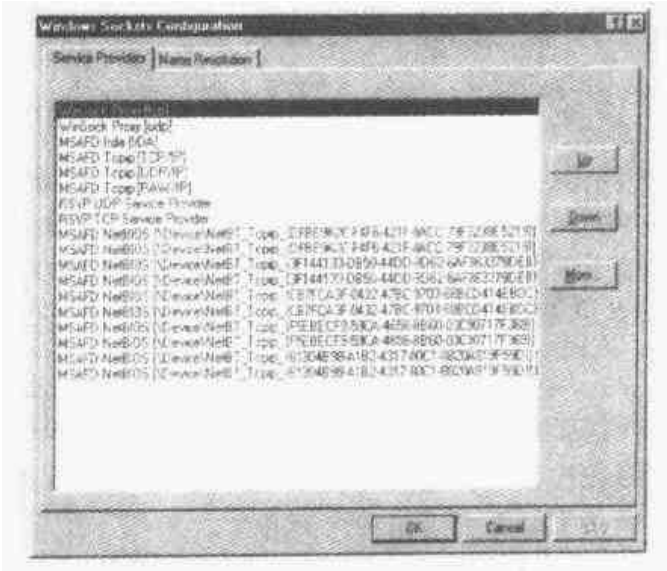


图 13-8　查看 Winsock Service Provider

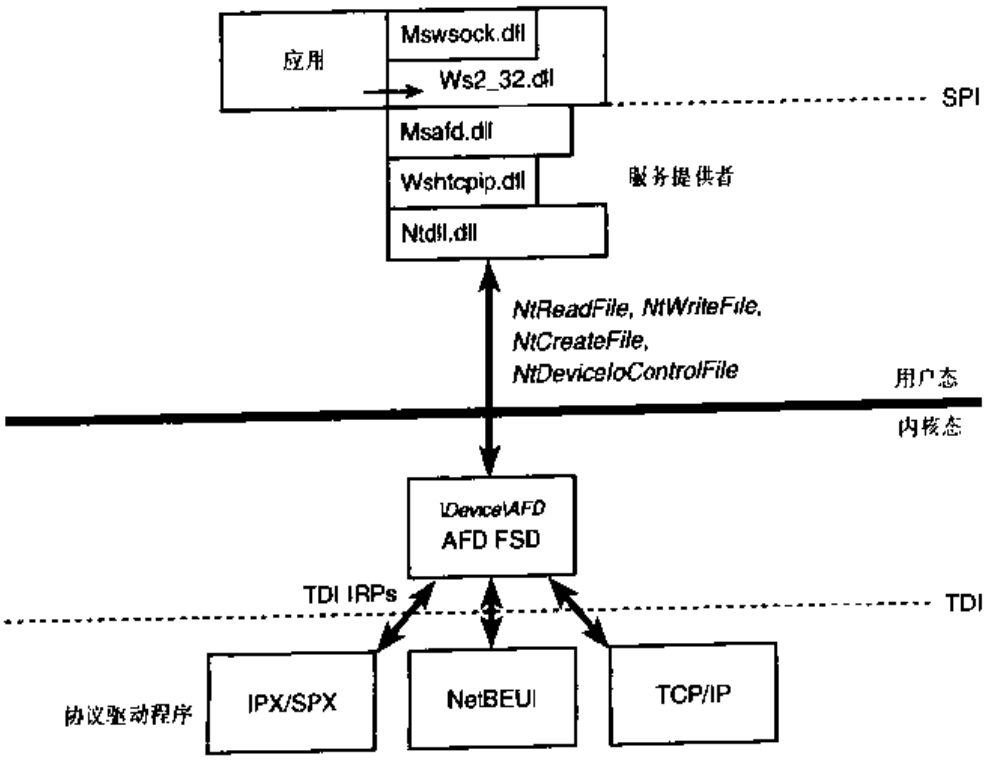


图 13-9　Winsock 实现

与命名管道和邮箱 API 一样，Winsock 集成了 Win32 I/O 模型，并且使用文件句柄表示套接字，这一支持需要内核模式文件系统驱动程序的帮助。因此 Msafd.dll 利用辅助函数驱动程序（AFD-\ Winnt \ System32 \ Drivers \ Afd.sys）的服务来实现基于套接字的函数。AFD 是一个 TDI 客户程序，实现网络套接字操作，如通过给协议驱动程序发送 TDI IRP 来发送和接收消息。AFD 没有被编码而用于专门的协议驱动节程序；相反，Msafd.dll 通知 AFD 用于每个套接字的协议的名字，以使 AFD 能打开表示协议的设备对象。

13.2.3 远程过程调用

远程过程调用（RPC）是 80 年代早期发展的网络编程标准。The Open Software Foundation（现为 The Open Group）将分布式计算环境（DCE）的 RPC 部分作为分布式计算标准。尽管有第二个 RPC 标准 SunRPC，但 Microsoft RPC 的实现仍与 OSF/DCE 标准兼容。RPC 建立在其他网络 API，如命名管道和 Winsock 上，提供一个可替换的编程模式，这在某种程度上对应用程序开发商隐藏了网络编程的细节。

1.RPC 操作

RPC 的性能之一是允许程序员创建包含任何数量过程的应用程序，它们中的一些在本机上运行，另外一些则通过网络在远程机器上运行。RPC 提供网络操作的过程视图，而不是以传输为中心的视图，这样大大简化了分布式应用程序的开发。

传统的网络软件围绕处理的 I/O 模型建立结构。例如在 Windows 2000 中，当应用程序发布一个远程 I/O 请求时，网络操作就被启动。操作系统将它向前传送给重定向器（redirector）来处理。重定向器通过建立客户机与其不可见的远程文件系统的交互作用，作为一个远程文件系统运行。重定向器将操作传递给远程文件系统，在远程系统响应请求并且返回结果后，本机网卡中断。内核处理中断，原始 I/O 操作完成，将结果返回给调用程序。

RPC 也使用一种不同的方法。RPC 应用程序与其他结构应用程序一样，都用一个主程序调用过程或过程库完成专门的任务。然而 RPC 应用程序与常规应用程序的区别在于 RPC 应用程序中的一些过程在远程计算机上执行，如图 13-10 所示，而其他的在本机上执行。

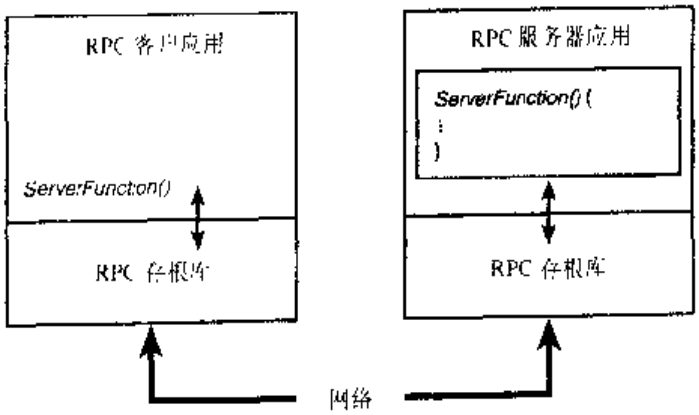


图 13-10 RPC 操作

对 RPC 应用程序来说，所有的过程似乎都在本机上运行。换言之，即并不需要程序员主动地编写代码来实现在网络中传输计算机的或 I/O 相关的请求处理网络协议和网络错误以及等

待结果等操作，RPC软件会自动地处理这些任务。Windows 2000 RPC实用程序可以处理输入到系统中的任何有效的传输。

要编写RPC应用程序，程序员必须确定哪些过程在本机上运行，哪些过程将在远程机运行。例如，假设一台普通的工作站与一台Cray超级计算机或一台专门为高速向量操作设计的计算机具有网络连接。如果程序员正在编写一个处理大型矩阵的应用程序，从性能的角度来考虑，也就意味着他在编写一个RPC应用程序，将数学运算加载到远程机上运行。

RPC应用程序是这样工作的：首先，当应用程序运行时，它会调用本机上的过程，也会调用远程机上的过程。为了处理后一种情况，应用程序被链接到一个本机静态链接库或DLL上，它们包含有许多存根过程，每个远程过程对应一个。对于简单的应用程序，存根过程与应用程序静态链接，但对于较大的应用程序组件，存根过程则包含在单独的DLL中。在本章后面涉及的DCOM中，尤其使用后一种方法。存根过程与远程程序同名，并且使用同样的接口，但是存根过程不执行请求操作，而是携带传递给它的参数并且对它们进行序列化（marshal）以便在网络中传递。序列化参数意味着用特殊的方法对参数进行排序并打包，以适应网络连接，如解析引用和建立指针指向的任何数据结构的拷贝。

然后存根调用RPC运行时过程定位远程过程驻留的计算机，确定该计算机使用的传输机制，并且利用本机传输软件将请求发送给它。当远程服务器接收到RPC请求后，它就对参数进行反序列化（unmarshal）操作（即marshal的逆向操作），再次构造原过程调用，并且调用过程。当服务器执行完毕，它就按反顺序将结果返回到调用过程。

除了这里描述的同步函数调用为基础的接口以外，Windows 2000 RPC也支持异步RPC。异步RPC允许RPC应用程序执行一个函数，但是不等到函执行完成就可以继续处理。也就是说，应用程序可以执行其他代码，然后，当来自于服务器的反应到达时，RPC运行时程序就通知事件对象客户机与异步调用相关联。客户机可以使用标准的Win32函数，如WaitForSingleObject，了解函数的完成。

除了运行时RPC，微软RPC实用程序还包含一个编译器，即微软接口定义语言（MIDL）编译器。MIDL编译器简化了RPC应用程序的建立。程序员编写一系列的函数源代码（假设为C或C++应用程序），它们描述远程例程，然后把这些例程放在一个文件中。然后程序员给这些源代码增加一些额外的信息，诸如例程包的网络唯一标识符、版本号，加上指明参数是输入、输出或两者都是的属性信息。这些加工润色后的源代码形成开发商的IDL文件。

一旦IDL文件建立，程序员用MIDL编译器编译它，这样就产生前面提到的客户方和服务器方的存根例程与应用程序中包含的头文件。当客户方的应用程序连接到存根例程时，所有的远程程序引用就被调用。然后这些远程程序就用相似的方式被安装在服务器计算机上。这样想调用已有的RPC应用程序的程序员只需编写客户方软件，并把应用程序连接到本机RPC运行时程序上即可。

RPC运行时程序利用通用RPC传输提供程序接口与传输协议通信。提供程序接口就象RPC实用程序与传输协议之间的一个薄层，将RPC操作映射在传输协议提供的函数上。Windows 2000 RPC实用程序为命名管道、NetBIOS以及TCP/IP协议实现传输提供程序DLL。你可以编写新的提供程序DLL支持其他的传输协议。用类似的方式，RPC实用程序设计成可与不同

的网络安全实用程序共同工作。

大多数 Windows 2000 网络服务程序是 RPC 应用程序，这就意味着本机过程和远程计算机上的过程都可以调用它们。这样远程客户机可以调用服务器服务程序列举共享资源、打开文件、向打印队列写数据、激活服务器上的用户或者调用消息服务程序把消息直接传递给你（当然，所有这些行为都有安全约束）。

服务器名字公布（Server name publishing）位于 RPC 中，并且与 Active Directory 集成在一起，它是服务器在客户机可访问的位置注册自己名字的能力。如果没有安装 Active Directory，RPC 名字定位器服务程序依赖 NetBIOS 广播。这一行为确保与 Windows NT 4 系统的相互可操作性，并且允许 RPC 在单独服务器和工作站上运行。

2.RPC 安全性

Windows 2000 RPC 包含与安全支持提供程序（SSP）的集成，因此 RPC 客户机和服务器可以使用验证或加密通信。当 RPC 服务器想安全通信时，它必须用一个 SSP 注册它的 SSP 专用主名（principal name）。当客户机与这样的服务器绑定时，客户机注册自己的安全凭证，指明服务器的主名，绑定时，客户机也指定它想要的验证等级。不同的验证等级的存在是为了一方面保证只有授权的客户机与服务器相连，证实服务器收到的每条消息都源自授权客户机，另一方面检查 RPC 消息的完整性以便检测操作，甚至加密 RPC 消息数据。很明显，验证的等级越高，要求的处理过程越多。

一个 SSP 不仅为 RPC，而且为 Winsock 处理网络通信的验证和加密的细节。Windows 2000 包含大量的内建 SSP，包括实现 Kerberos 5 身份验证的 Kerberos SSP 和 Secure Channel（SChannel）。SChannel 实现 Secure Sockets Layer（SSL）、Transport Layer Security（TLS）协议以及私用通信技术（PCT）。在缺乏专用 SSP 时，RPC 使用内置的命名管道安全。

RPC 安全程序的另一特征是服务器用 RpcImpersonateClient 函数模拟客户安全身份的能力。服务器代表客户机完成模拟操作后，通过调用 RpcRevertoToSelf 或者 RpcRevertToSelfEx 函数返回自己的安全身份（见第 8 章关于模拟（impersonation）的更多信息）。

3.RPC 实现

RPC 实现过程如图 13-11，它表明了基于 RPC 的应用程序与 RPC 运行时（run-time）DLL（Winnt \ System32 \ Rpcrt4.dll）的连接。RPC 运行时 DLL 通过应用程序的 RPC 函数的存根以及发送和接收被序列化的数据的函数来提供序列化和反序列化函数。RPC 运行时 DLL 支持处理网络中 RPC 的例程以及一种称为本机 RPC 形式的 RPC。本机 RPC 可用于同一系统中两个过程之间的通信，而且 RPC 运行时 DLL 使用内核模式中本机程序调用（LPC）实用程序作为本机网络 API。（详见第 3 章关于 LPC 的信息）。当 RPC 基于非本机通信机制时，RPC 运行时 DLL 使用 Winsock，命名管道或 Message Queuing（稍后介绍）API。

对于名字注册和查找，RPC 应用程序连接 RPC 名字服务 DLL（\ Winnt \ System32 \ Rpcns4.dll）。DLL 与 RPC 子系统（RPCSS – \ Winnt \ System32 \ Rpcss.dll）通信，RPCSS 作为一个 Win32 服务程序实现。RPCSS 本身是一个 RPC 应用程序，它与其他系统上自身的实例通信以完成名字查询和注册（为清晰起见，图 13-11 没有显示出 RPCSS 与 RPC 运行时 DLL 的连接）。

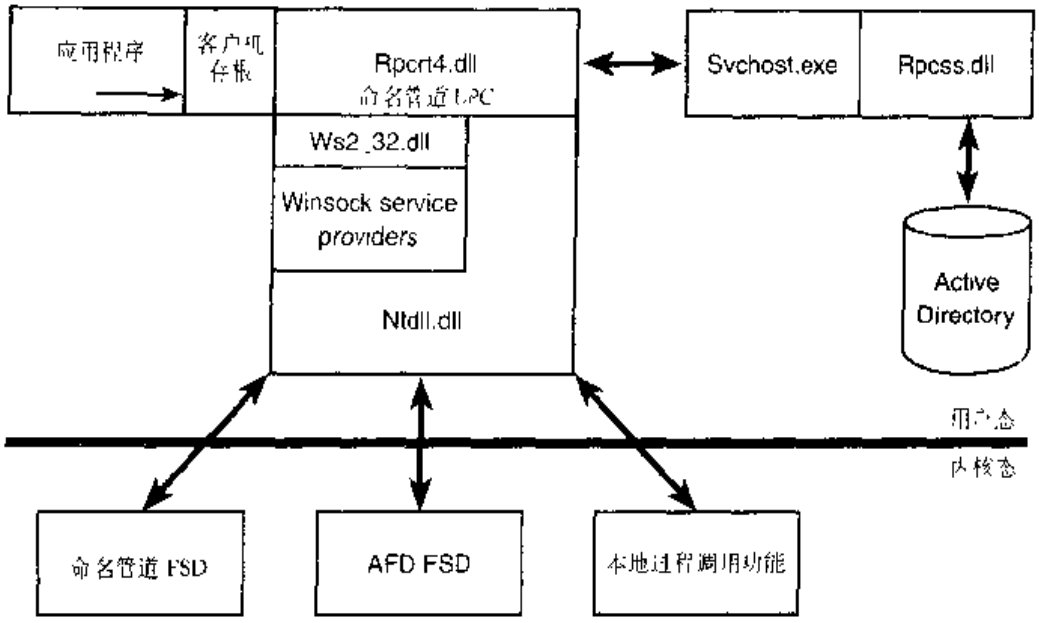


图 13-11　RPC 实现

13.2.4　通用网络文件系统

Common Internet File System（CIFS）是 Server Message Block（SMB）协议的增强形式，是 Windows 2000 用于实现文件共享使用的协议。由于应用程序使用标准的 Win 32 文件 I/O 函数访问远程文件，因此应用程序不直接使用 CIFS 协议，但该协议用来处理 I/O 请求。CIFS 定义打印机共享规则，因此 Windows 2000 也使用 CIFS 定义它。尽管 CIFS 本身不是一个 API，但是由于文件和打印机共享建立在 CIFS 基础上，而且通过 Win32 API 指向应用程序，所以我们在这里介绍它。

CIFS 是一个公开发布的微软标准（见 Platform SDK 文档），它允许第三方与 Windows 2000 文件服务器和 Windows 2000 文件共享客户互相操作。例如，Samba 共享软件套件允许 UNIX 系统为客户提供服务，又使 UNIX 应用程序可以访问由 Windows 2000 系统提供的文件。其他支持 CIFS 的平台有 DEC VMS 和 Apple Macintosh。

Windows 2000 文件共享基于运行在客户机上的重定向器 FSD（简称重定向器），它与运行在服务器上的服务器 FSD 通信。重定向器 FSD 拦截指向驻留在服务器上的文件的 Win32 文件 I/O，将 CIFS 信息传递给服务器文件系统来执行客户机请求。服务器接收到 CIFS 信息并把它们翻译成它发送到本机 FSD 的 I/O 操作指令，如 NTFS。图 13-12 显示重定向器与服务器相互通信的情况。

由于与 Windows 2000 I/O 集成，重定向器和服务器 FSD 相对于文件服务器在可替换用户空间上的实现有几个优点：

- 它们可以直接与 TDI 传输和本机 FSD 交互。
- 它们可以与高速缓存管理器集成到客户机系统上的无缝高速缓存服务器文件数据（稍后将介绍 Windows 2000 使用的高速缓存协议）。
- 应用程序可以使用标准的 Win32 文件 I/O 函数，如 CreatFile、ReadFile 以及 WriteFile 访问远程文件。

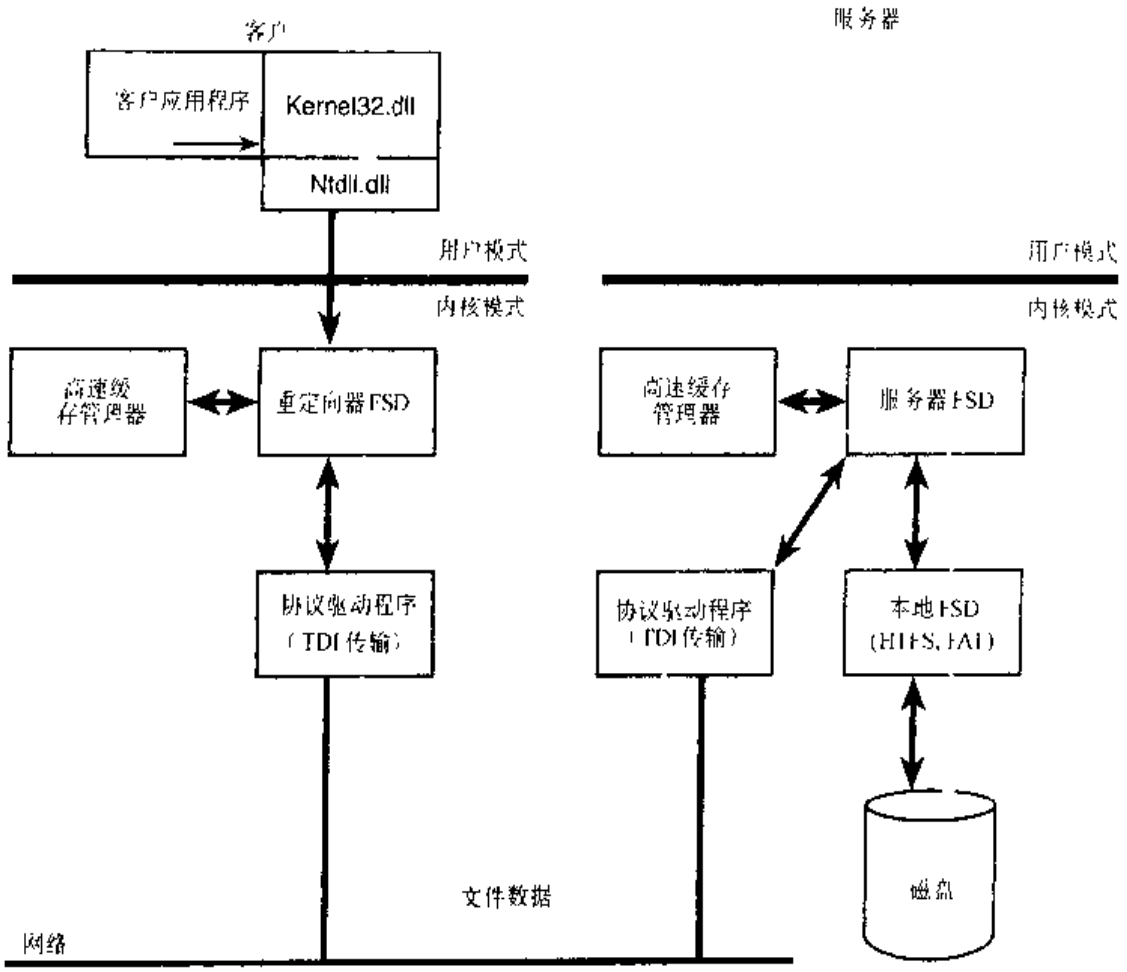


图 13-12 CIFS 文件共享

Windows 2000 重定向器和服务器 FSD 依赖于所有内核模式文件服务器和客户软件使用的标准网络资源命名惯例。如果远程文件共享使用驱动符连接，则网络文件名与本机名字使用相同的方式指定。然而，重定向器也支持 UNC 名字。关于网络资源名字解析将在本章后面的“网络资源名解析”一节中讨论。

1. CIFS 实现

Windows 2000 上的重定向器 FSD 在端口/小端口模型中实现，这与许多其他的设备驱动程序类型相同（见第 9 章关于设备驱动程序）。微软公司提供了一个名为 \ Winnt \ System32 \ Drivers \ Rdbss.sys 的重定向器库，开发商可以为它写一个重定向器小端口程序。重定向器库隐藏了重定向器实现的许多细节，如与高速缓存管理器、内存管理器以及 TDI 传输的集成。CIFS 小端口驱动程序为 \ Winnt \ System32 \ Drivers \ Mrxsmb.sys。

Rdbss 利用高速缓存管理器服务程序高速缓存文件数据并充分发挥智能预读（intelligent read-ahead），本书的第 11 章、12 章讨论高速缓存管理器服务程序。CIFS 小端口驱动程序通过 TDI API 向远程服务器发送 CIFS 指令。它可以使用任何支持 TDI 接口的传输协议，如 NetBEUI、NetBT（使用 TCP/IP 的 NetBIOS）和 TCP/IP。

在作为文件服务器的系统中，服务器（\ Winnt \ System32 \ Drivers \ Srv.sys）FSD 倾听来自于客户机的 CIFS 命令，它的作用相等于被访问的文件驻留的本机 FSD 的代理接口。服务器

FSD 利用固有的 Windows 2000 FSD 实现的文件系统接口实现零拷贝发送功能。接口允许服务器 FSD 获取驻留在服务器文件系统高速缓存中的文件数据的内存描述符列表（MDL）的描述，并且使用网络适配器驱动程序将 MDL 传递给 TDL 传输程序以便在网络中传输（见第 7 章关于 MDL 的详细信息）。如果没有本机 FSD 和高速缓存管理器的支持，服务器 FSD 就不得不先拷贝文件数据到自己的缓冲区，然后再传递给 TDI 传输。

服务器 FSD 和重定向器 FSD 都有对应的 Win32 服务程序，服务器以及工作站，它们运行在服务控制管理器（SCM）进程中为驱动程序提供管理员管理接口。

2. 分布式文件高速缓存

如果单个客户访问服务器上的文件，显然客户可以安全地高速缓存客户系统中的文件数据。然而当两个客户机访问同一文件时，必须采取措施在两个客户和服务器之间提供文件的一致视图。Windows 2000 解决这个问题，即分布式高速缓存一致性，是通过机会主义锁定（opportunistic lock，oplock）机制实现的。当客户想访问服务器文件时，它必须首先申请一个 oplock。服务器授予客户的 oplock 类型决定客户可以运行的高速缓存类型。

有三种主要的 oplock 类型：

- 当客户对文件具有专有访问权时，它就被授予 Level I oplock。拥有这种 oplock 的客户可以在客户系统上高速缓存读和写的数据。
- Level II oplock 代表一个共享的文件锁。拥有 Level II oplock 的客户可以高速缓存读数据，但向文件写数据使 Level II oplock 无效。
- Batch oplock 是最开放的 oplock 形式。客户可以高速缓存向文件读和写的数据，也可以不申请额外的 oplock 就打开或关闭文件。Batch oplock 特别用于仅仅支持批处理文件的运行，批处理文件运行时可以重复地打开或关闭一个文件。

如果一个客户没有 oplock，就不能在本机高速缓存读和写数据，相反必须从服务器返回数据，而且将所有改变的信息直接发送给服务器。

图 13-13 的例子将有助于说明 oplock 的操作。服务器自动地授予第一个客户 Level I oplock，使它可以打开服务器文件访问。客户重定向器高速缓存文件数据，以便在客户机的文件高速缓存器中读和写。如果第二个客户打开文件，它也申请一个 Level I oplock。然而由于两个客户访问同一文件，服务器必须采取步骤向两个客户显示文件数据的一致视图。如果第一个客户已经向文件中写入数据，如图 13-13 所示，则服务器取消它的 oplock，并且不授予任何一个客户 oplock。当第一个客户的 oplock 被取消或中断时，客户把它已经高速缓存的任何数据刷新返回给服务器。

如果第一个客户没有向文件写数据，则它的 oplock 被中断并授予 Level II oplock，与第二个客户的 oplock 类型相同。这样两个客户都可以高速缓存读数据，但如果任何一方向文件写数据，服务器就会取消它们的 oplock 以使非高速缓存操作开始运行。一旦 oplock 被中断，对于文件的相同打开实例不再授予 oplock。然而，如果一个客户关闭文件后又再次打开，服务器为它分配何种 oplock 则取决于其他客户具有什么样的文件打开方式以及它们中是否至少有一个已经向文件写入数据。

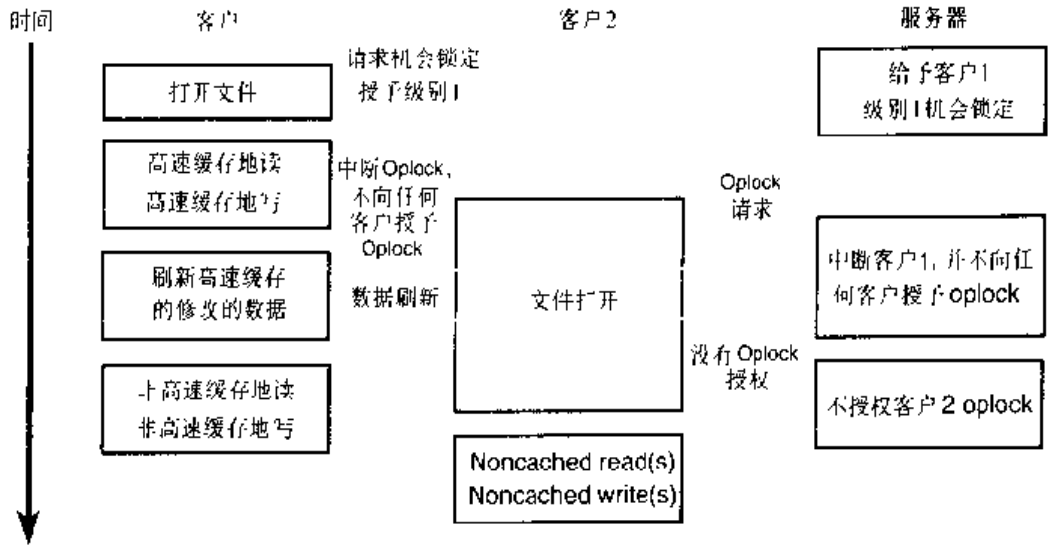


图 13-13 Oplock 举例

13.2.5 NetBIOS

到本世纪90年代，网络基本输入/输出（NetBIOS）编程API已经成为PC机上应用最为广泛的编程API。NetBIOS允许可靠的面向连接和不可靠的无连接通信。Windows 2000为它的遗留（legacy）应用程序支持NetBIOS。微软不鼓励应用程序开发商使用NetBIOS，因为其他的API如命名管道和Winsock的灵活性和可移植性更强。NetBIOS由Windows 2000上的TCP/IP、NetBEUI及IPX/SPX协议支持。

1.NetBIOS名字

NetBIOS依赖于命名约定，借此计算机和网络服务程序被分配一个16字节的名字，称为"NetBIOS名字"。NetBIOS名字的第16个字节被看成修饰词，它可以指定一个名字是唯一的或是组的一部分。只有唯一NetBIOS名字的一个实例才可以分配给网络。但是多个应用程序可以分配相同的组名。客户可以通过将消息发送给组名广播消息。

为了支持与Windows NT系统和Consumer Windows的互操作性，Windows 2000自动地为域定义了一个NetBIOS名字，即管理器分配给域的域名系统（DNS）名字的前15个字节。例如，如果域名为mspress.microsoft.com，则域的NetBIOS名字为mspress。同样地，Windows 2000安装时要求管理器为每个计算机分配一个NetBIOS名字。

NetBIOS常用到的另一个概念是LAN适配器（LANA）号。每一个成层置于网络适配器上的NetBIOS兼容协议都分得一个LANA号。例如，如果一台计算机有两个网络适配器，并且TCP/IP和NetBEUI协议可使用其中的任何一个，那么就会有4个LANA号。LANA号是很重要的，因为NetBIOS应用程序必须明确地把它的服务程序名字分配给每个LANA，这样它就会接收客户连接。如果应用程序为客户连接倾听到一个特定的名字，客户只有通过该名字注册的网络适配器上的协议才能访问它。

一个称为"Windows Internet Name Service"（WINS）的网络服务程序维护NetBIOS与TCP/IP协议地址之间的映射。如果没有安装WINS，则在Windows网络中用名字广播传播名字。注意：NetBIOS名字次要于DNS名字；计算机名字首先通过DNS名字注册和解析，只有DNS名字解析

失败时，Windows 2000 才返回到 NetBIOS 名字（本章后面“网络资源名字解析”一节中介绍 DNS 名字解析）。

2. NetBIOS 操作

NetBIOS 服务器应用程序使用 NetBIOS 枚举系统中出现的 LANA，并且为每个 LANA 分配一个代表应用程序服务的 NetBIOS 名字。如果服务器是面向连接的，它就完成一个 NetBIOS 倾听命令等待客户连接企图。客户机连接后，服务器执行 NetBIOS 函数发送和接收数据。无连接通信与此相似，但服务器不建立连接，只简单地读信息。

面向连接的客户机使用 NetBIOS 函数建立与 NetBIOS 服务器的连接，然后进一步执行 NetBIOS 函数发送和接收数据。一个已建立的 NetBIOS 连接也称作“会话”（session）。如果客户机想发送无连接消息，它只需用发送函数指定服务器 NetBIOS 名字即可。

NetBIOS 由大量函数组成，但这些函数都通过同一接口 Netbios 路由。这一路由方案是 NetBIOS 作为 MS-DOS 中断服务程序在 MS－DOS 上实现时的遗留结果。一个 NetBIOS 应用程序将会执行一个 MS－DOS 中断，并且为 NetBIOS 运行程序传递一个数据结构。NetBIOS 运行程序指定正在执行的命令的各个方面。因此，Windows 2000 中的 Netbios 函数带有唯一参数，它是包含应用程序请求的服务程序专用参数的数据结构。

实验：用 Nbtstat 查看 NetBIOS 名字

你可以利用 Windows 2000 中包含的 Nbtstat 命令显示系统中的活动会话、计算机上高速缓存的 NetBIOS－TCP/IP 名字映射以及计算机定义的 NetBIOS 名字。这里是用 Nbtstat －n 命令显示计算机上定义的 NetBIOS 名字的例子

```
C:\>nbtstat -n
Local Area Connection:
Node IpAddress: [10.0.0.5] Scope Id: []

                NetBIOS Local Name Table

      Name               Type         Status
-----------------------------------------------
    MARKLAP        <00>  UNIQUE       Registered
    MARKLAP        <20>  UNIQUE       Registered
    MARK           <00>  GROUP        Registered
    MARK           <1E>  GROUP        Registered


\Device\NetBT_Tcpip_{0A546AD9-3AD1-4A89-8588-1ADBF9AC5620}:
Node IpAddress: [32.100.225.5] Scope Id: []

                NetBIOS Local Name Table

      Name               Type         Status
-----------------------------------------------
    MARKLAP        <00>  UNIQUE       Registered
    MARK           <00>  GROUP        Registered
    MARK           <1E>  GROUP        Registered
```

3. NetBIOS API 实现

NetBIOSAPI 实现的组件如图 13-14 所示。Netbios 函数通过 \ Winnt \ System32 \ Netapi32.dll 输出到应用程序。Netapi32.dll 为称为 "NetBIOS 仿真器" (emulator) (\ Winnt \ System32 \ Drivers \ Netbios.sys) 的内核模式驱动程序打开一个句柄，并且代表应用程序发出 Win32 DeviceIoControl 文件命令。NetBIOS 仿真器将应用程序发出的 NetBIOS 命令翻译成 TDI 命令，并且发送给协议驱动程序。

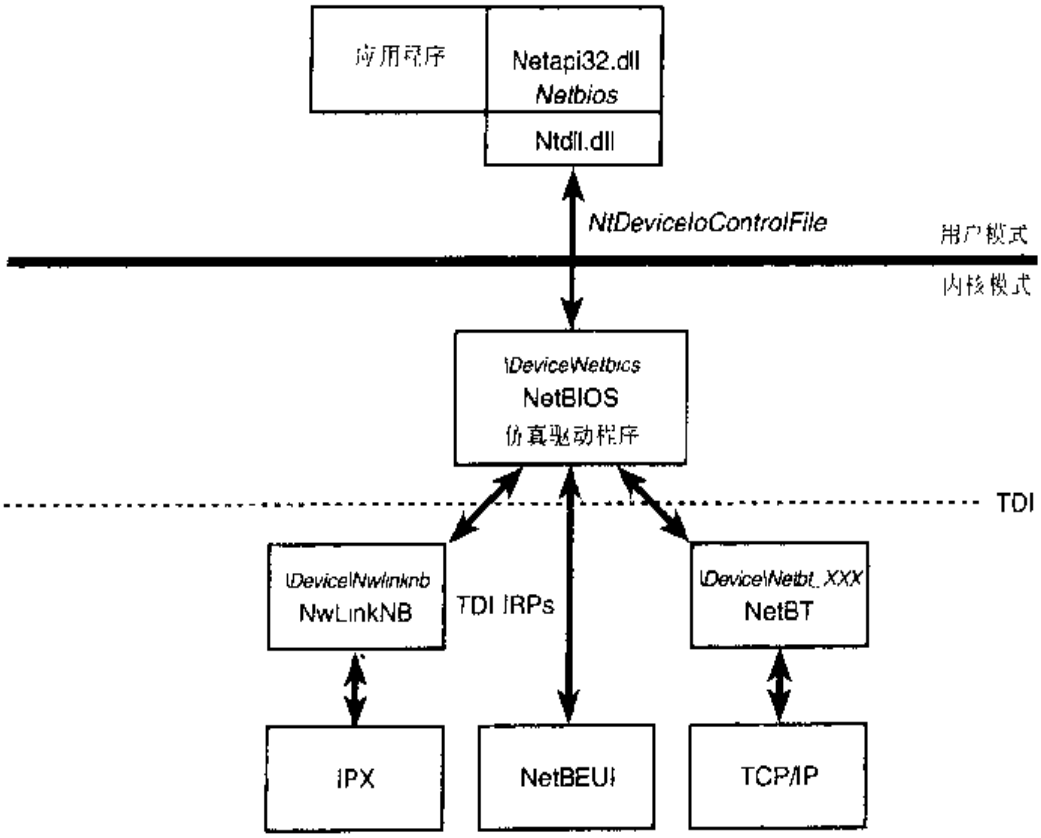


图 13-14 NetBIOS API 实现

如果一个应用程序想在 TCP/IP 协议上使用 NetBIOS，NetBIOS 仿真器需要 NetBT 驱动程序 (\ Winnt \ System32 \ Drivers \ Netbt.sys)。TCP/IP 驱动程序上的 NetBIOS 即是 NetBT，它负责支持 NetBEUI 协议固有的而 TCP/IP 不具有的 NetBIOS 语义 (稍后讨论)。例如，NetBIOS 依赖于 NetBEUI 的消息模式传输和 NetBIOS 名字解析功能，因此 NetBT 驱动程序在 TCP/IP 协议顶层实现它们。与此相似，NwLinkNB 驱动程序在 IPX/SPX 协议上实现 NetBIOS 语义。

13.2.6 其他网络 API

Windows 2000 包含其他的 API，它们不常使用，或者层状置于已经讨论过的 API 上面 (超出本书范围)。然而其中的三个对 2000 系统和许多应用程序的运行是相当重要的，值得简要叙述。它们是 Telephony API (TAPI)，Distributed Component Object Model (DCOM) 和 Message Queuing。

1. Telephony API

电话应用程序接口集成了计算机与通信设备如电话和调制解调器。Windows 2000 中，电话

应用程序也包括声音网络协议（VoIP）、群播多介质会议以及实时协作（RTC）等应用程序。Windows 2000包含的Telephony API是为那些想通过支持电话的设备进行通信的应用程序而设计的。TAPI抽象化了设备管理的具体细节，因此TAPI应用程序可以不作修改而用于各种不同的设备。Windows 2000为C应用程序装载TAPI、TAPI 2.2两个版本，为COM应用程序装载TAPI 3.0。

TAPI可以分解成用于设备、会话和介质控制的API子集。设备控制接口允许TAPI传递设备特征，改变设备特征和为TAPI应用程序查询设备特征。设备特征包括线路、地址标识符、设备事件以及介质类型。会话控制接口使应用程序在两个或多个地址间建立连接。会话控制操作与成熟的电话支持的操作相似，即包括会话初始化、应答、接收、向前、暂停、传输、挂断。最后，介质控制接口允许TAPI应用程序实现声调探测和拨号等操作。

微软电话应用程序接口的结构如图13-15所示，以TAPI Service（\winnt\System32\Tapisrv.dll）为中心，它在Service Hose进程中作为Win32 Service服务实现。（见第5章关于Win32 Service的信息）。TAPI应用程序利用TAPI客户端的DLL（\winnt\System32\Tapi32.dll）通过RPC与TAPI服务程序通信。

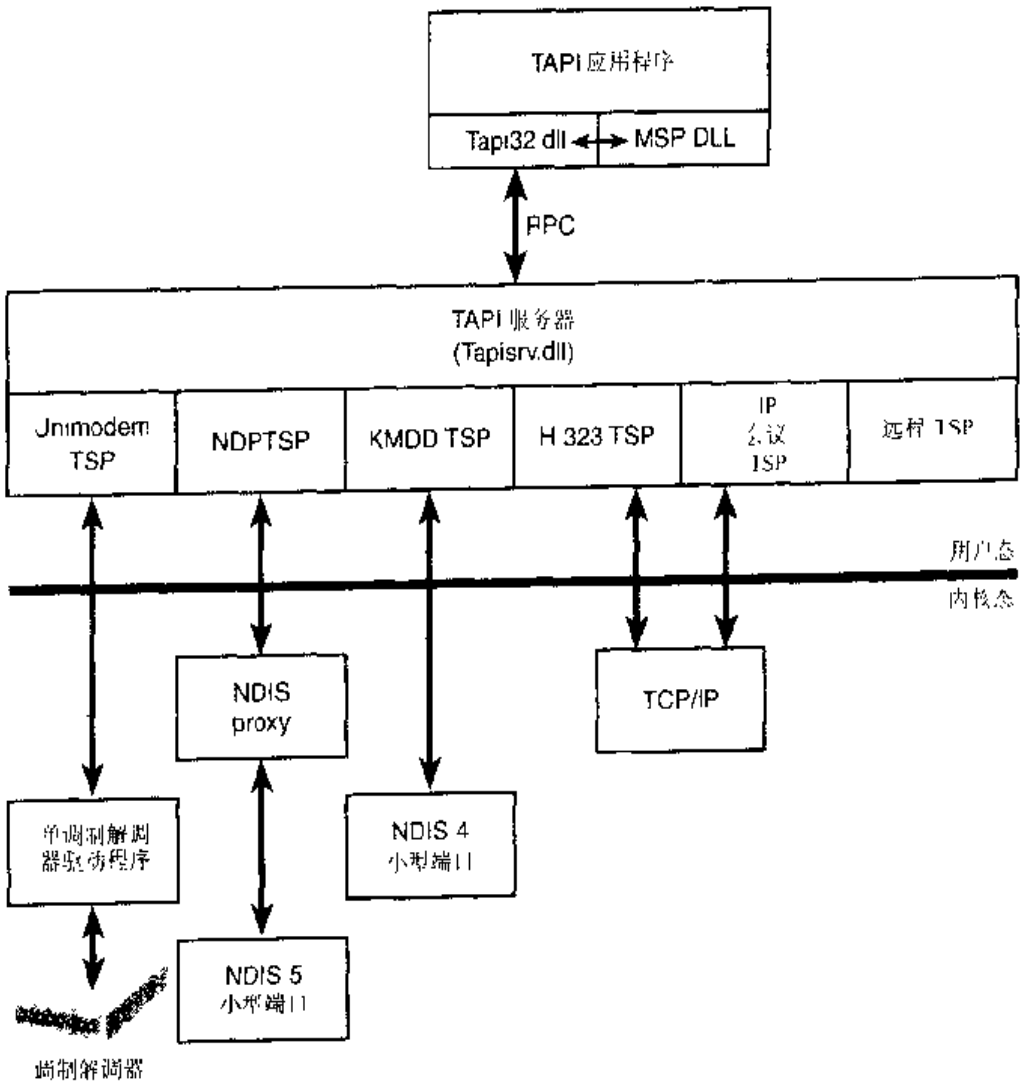


图13-15 TAPI结构

为了支持设备抽象化（abstraction），TAPI 服务程序加载 TAPI 服务提供程序（TSP），它为特定的设备或设备类提供接口。例如，Unimodem TSP 提供 TAPI 服务程序与大多数类型调制解调器的接口。TAPI 也包含介质服务提供程序（MSP），它允许 TAPI 应用程序增强对介质的介质专有控制。应用程序加载的 MSP DLL 都有相应的 TAPI 服务器中的 TSP 。与 Windows 2000 捆在一起的其他 TSP 包括下面几种：

- 远程 TSP 提供对远程通信资源的访问，如调制解调器交换区。
- H.323 TSP 允许 TAPI 应用程序遵从 H.323 电话通信协议建立并且接收图像和声音调用。
- NDIS Proxy TSP（NDPTSP）与 TAPI 线一样提供 NDIS 5 小型端口（本章后面讨论）。
- 内核模式设备驱动程序（KMDD）TSP 与 TAPI 线一样为 WAN 设备提供 NDIS 4 小端口。
- IP Conference TSP 支持 TCP/IP 连接上的会议。

TAPI 体系结构和文档使得第三方有可能开发自己的 TAPI TSP。

2. DCOM

微软的 COM API 使应用程序由不同的组件组成，每一组件是一个可替换的自包含模块。一个 COM 对象输出一个面向对象的方法接口以实现在对象内部操纵数据。由于 COM 对象提供良好的接口，开发商可以运行新的对象来扩展已有接口，并且可以用新的支持动态地更新应用程序。

DCOM 通过让应用程序组件驻留在不同的计算上来实现 COM 扩展，也就是说应用程序不必担心一个 COM 对象在本机上，而另一个可能跨越 LAN 。DCOM 提供位置透明性，这样简化了分布式应用程序的开发。DCOM 不是一个自包含的 API，但是它依赖于 RPC 来完成它的工作。

3. Message Queuing

微软最新的网络服务程序 Message Queuing 在 Windows NT 4 企业版中引进。Message Queuing 是为开发分布式应用程序提供的大众化平台，分布式应用程序利用松散的耦合消息。因此 Message Queuing 既是一个 API，也是一个消息基本结构。它的灵活性在于它的队列就像一个消息存储器，在这里发送者可以为接收者排列消息，接收者可以根据具体情况解散消息队列。发送者与接收者不必为使用 Message Queuing 建立连接，它们甚至不必同时执行，它允许无连接异步信息交换。

Message Queuing 的一个突出特点是它集成了 Microsoft Transaction Server（MTS）和 SQL Server，因此它能够参与 Microsoft Distributed Transaction Coordinator（MS DTC）的协调事务，使用带 Message Queuing 的 MS DTC，允许你开发三层应用程序可靠的事务功能。

13.3 网络资源名字解析

应用程序可用两种方式检验或访问远程系统上的资源。一种是通过使用带 Win32 函数的 UNC 标准直接访问远程资源；另一种是通过使用 Windows 网络（WNet）API 枚举计算机以及这些计算机输出的共享资源。两种方法都利用了重定向器的功能查找它们到达网络的途径。如前所述，为了从客户机访问 CIFS 服务器，微软提供了一个 CIFS 重定向器，它具有一个称为“重

定向器 FDS”的内核模式组件以及一个称为“Workstation Service”的用户模式组件。微软公司也提供了能访问 Novell Netware 服务器共享资源的重定向器，而且第三方可以向 Windows 2000 添加自己的重定向器。这一节，我们将来研究远程 I/O 请求发出后决定激活哪个重定向器的软件。主要组件如下：

- Multiple provider router（MPR）是一个 DLL，它决定当应用程序用 Win32 WNet API 浏览远程文件系统时去访问哪一个网络。
- Multiple UNC provider（MUP）是一个驱动程序，它决定当应用程序用 win32 I/O API 打开远程文件时去访问哪个网络。

我们将通过讨论域名系统（(DNS）来总结本节，DNS 是 Windows 2000 中计算机名字解析的核心。

13.3.1 MPR

Win32 WNet 函数允许应用程序（包括 Windows Explorer My Network Places）与网络资源（如文件服务器和打印机）建立连接，并且浏览任何形式的远程文件系统内容。因为 WNet API 可被调用在使用不同传输协议的不同的网络上工作，因此必须提供在网络中可以正确发送请求并且正确理解远程服务器返回结果的软件。图 13－16 显示了负责这些任务的重定向器软件。

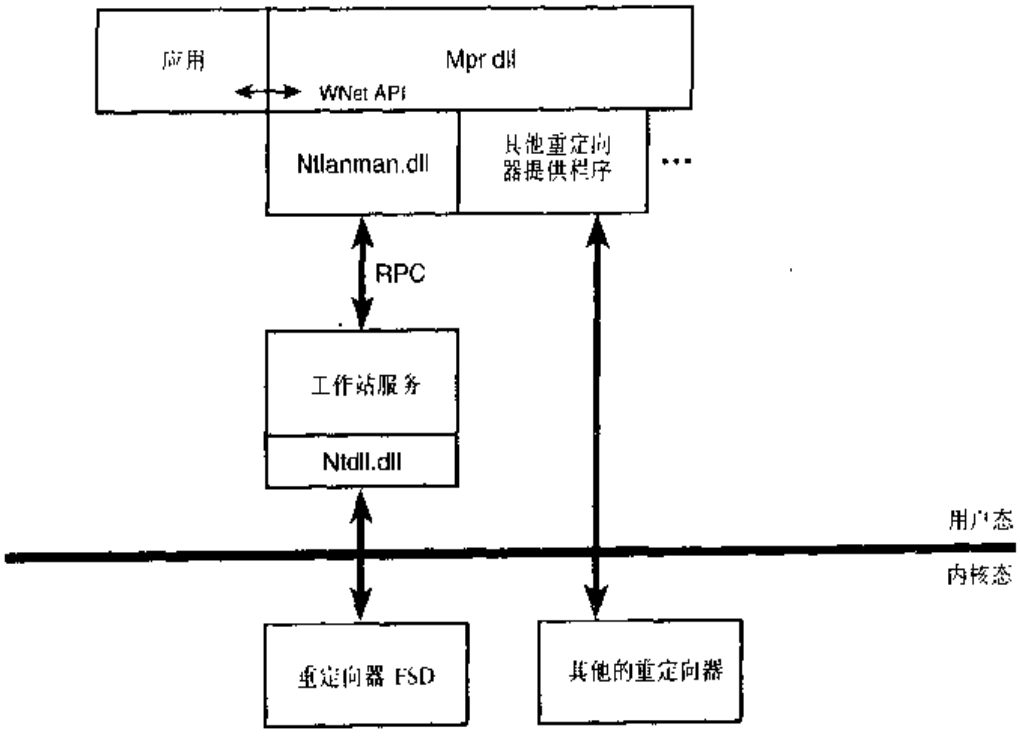


图 13-16 MPR 组件

一个提供程序（Provider）是把 Windows 2000 建立为远程网络服务器的客户的软件。一个 WNet 提供程序执行的操作包括建立和中断网络连接，远程打印以及交换数据。内建的 WNet 提供程序包含一个 DLL、Workstation 服务程序以及重定向器。其他的网络提供程序仅需提供一个 DLL 和一个重定向器。

当应用程序调用 WNet 例程时，调用直接传递给 MPR DLL。MRP（Multiple Provide Router，

多提供程序路由器）接受调用并且决定哪个 WNet 提供程序识别被访问的资源。MPR 下的每一个提供程序 DLL 提供一系列标准函数，它们集中起来称为“提供程序接口”。这个接口允许 MPR 决定应用程序正试图访问哪个网络和直接将请求发送给合适的 WNet 提供程序软件。重定向器的提供程序是 \ Winnt \ System32 \ Ntlanman.dll，与 HKLM \ SYSTEM \ Current Controlset \ Services \ lanmanworkstation \ Network provider 注册键下的 ProviderPath 键值指示的一样。

当被 WNetAddConnection API 函数调用与远程网络资源连接时，MPR 检查 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Networkprovider \ Order \ ProviderOrder 注册表键值来决定提供程序加载到哪个网络上。MPR 按它们在注册表中的排列顺序一次查询一个程序，直到一个重定向器识别出资源或者所有可用的提供程序都被轮询过为止。你可以用 Advanced Settings 对话框改变 ProviderOrder，如图 13-17 所示（图示屏幕快照的系统只安装了一个提供程序）。从 Network And Dial - Up Connections 应用程序的 Advanced 菜单中可以打开该对话框。你可以在桌面上右击 My-Network Place 图标并且从下拉菜单中选择属性 Properties 按纽访问 Network And Dial - Up Connections 应用程序，也可以从开始菜单的 Settings 选项中选择它。

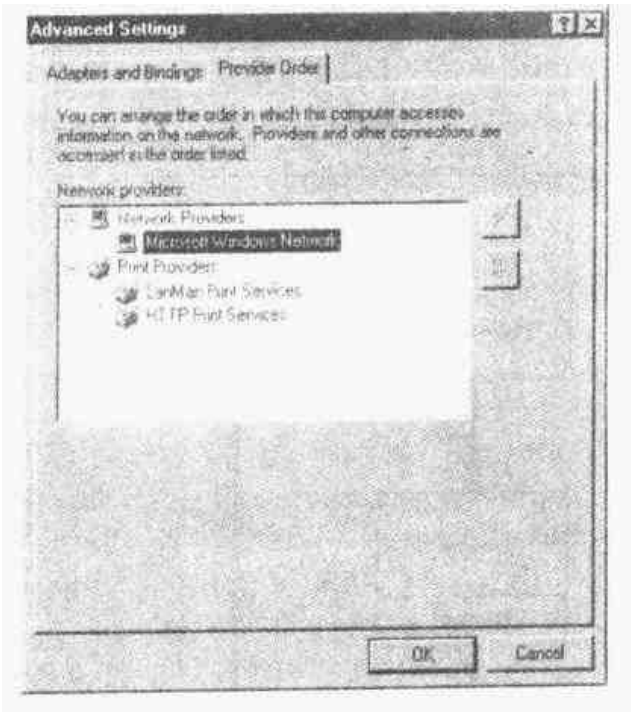


图 13-17 Provider 命令编辑器

WNetAddConnection 也可以为远程资源分配一个驱动器符或设备名字。此时，WNetAddConnection 将调用路由到相应的网络提供程序。反过来，提供程序在对象管理器名字空间建立一个符号链接对象为网络映射定义给重定向器（即远程 FSD）的驱动器符。

图 13 - 18 表示 \ ?? 目录，在这里你可以查看几种表示远程文件共享链接的驱动器符。本图表明重定向器创建了一个名为 \ Device \ LanmanRedirector 的设备对象，符号链接值中的其他文本字符表示驱动符对应哪个远程资源。当用户打开 X: \ Book \ Chap 13.doc 文件时，通过符号链接解析的路径的未解析部分传递给重定向器，即“X: 0 \ dual \ e \ Book \ Chap13.doc.”。这样重定向器就注意到正在被访问的资源位于 dual 服务器的 E 共享部位。

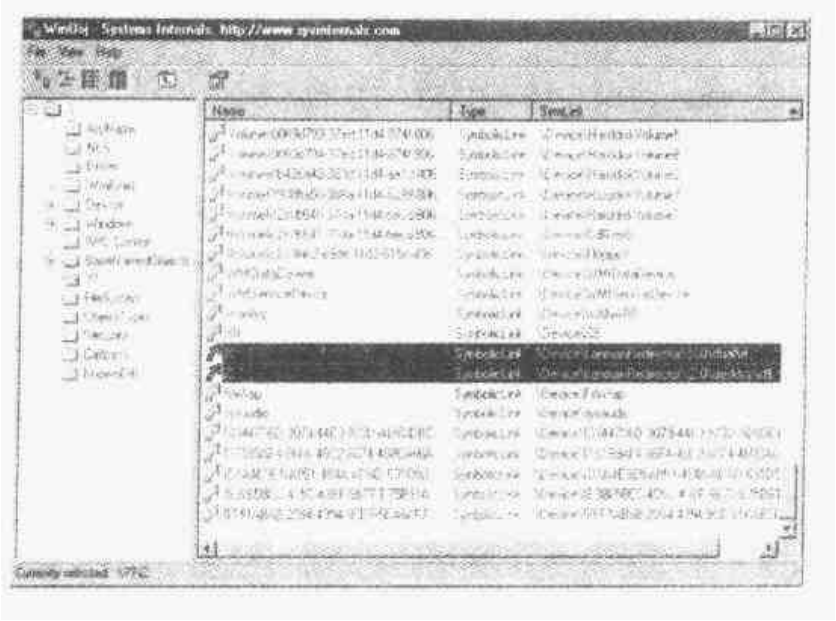


图 13-18 网络资源名字解析

像内建重定向器一样，当重定向器被加载到系统并且初始化时，其他重定向器在对象管理器名字空间中创建一个设备对象。然后，当 WNet 或其他的 API 调用对象管理器以打开不同网络上的资源时，对象管理器就会利用设备对象作为一个跳转点进入远程文件系统。远程文件系统调用与设备对象相关的 I/O 管理器分析方法查找能处理请求的重定向器 FSD（见第 12 章关于文件系统驱动程序的信息）。

13.3.2 MUP

MUP（Multiple UNC Provider，多 UNC 提供程序）是与 MRP 相似的一个网络组件。它屏蔽目标为有 UNC 名字的文件或设备（名字以字符 \ \ 开头，表示是网络上的资源）的请求。MRP 像 MUP 一样，处理这些请求，确定哪个本机重定向器识别远程资源。与 MRP 不同，MUP 是一个设备驱动程序（系统引导时加载），发出 I/O 请求到低层次的驱动程序，在这种情况下即是到达重定向器，如图 13-19 所示。

当应用程序通过指定一个 UNC 名字（不是如前所述的重定向的驱动符）首次试图打开一个远程文件或设备时，MUP 驱动程序就被激活。当 Win32 客户端 DLL Kernel32.dll（它是输出文件 - I/O 相关的 API 的 DLL）接收到这样的请求时，子系统将 UNC 名字添加到字符串 \ ?? \ UNC 后面，然后调用 NtCreateFile 系统服务程序打开文件。这个对象名字是解析到代表着 MUP 提供程序的设备对象 \ Device \ MUP 的符号连接的名字。

MUP 驱动程序接收请求并且发送一个异步 IRP 给每一个注册的 重定向器，等待它们中的一个识别资源名字并且应答。当重定向器识别出名字时，它显示了名字的唯一性如何。例如，如果名字是 \ \ WIN2KSERVER \ PUBLIC \ insidew2k \ Chap13.doc，则重定向器识别它并且将字符串 \ \ WIN2KSERVE \ PUBLIC 声明为自己的。MUP 驱动程序高速缓存这个信，然后跳过定时查询操作，直接将以此字符串开始的请求发送给重定向器。MUP 驱动程序高速缓存器有一个超时设定，因此静止一段时间后，字符串与特定的重定向器的连接就会过期。如果不止一个重定向器声明特定资源，则 MUP 驱动程序就利用加载的重定向器的注册表的 ProviderOrder

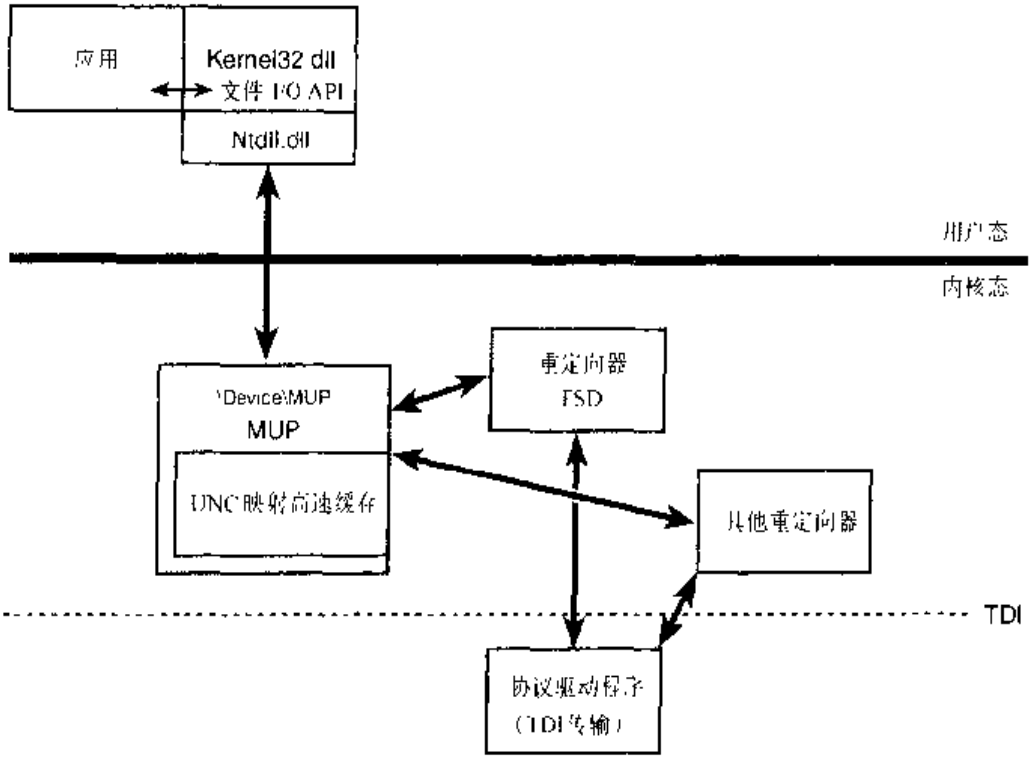


图 13-19 Multiple UNC Provider (MUP)

值列表来确定哪个重定向器优先。

13.3.3 域名系统

域名系统（Domain Name System，DNS）是因特网名（如 WWW.microsoft.com）被翻译成它们对应的 IP 地址遵循的标准。如果网络应用程序想要将一个 DNS 名字解析为 IP 地址，它就利用 TCP/IP 协议向 DNS 服务器发送一个 DNS 查找请求。DNS 服务器实现用于执行翻译的名字/IP 地址对的分布式数据库，每个服务器对特定的区进行翻译。详细描述 DNS 超出本书的范围，但 DNS 是 Windows 2000 中命名的基础，因此它是主要的 Windows 2000 名字解析协议。

Windows 2000 DNS 服务器像包含在 Windows 2000 服务器版中的 Win32 服务程序（\Winnt\System32\Dns.exe）一样运行。标准的 DNS 服务器实现依赖于作为翻译数据库的文本文件，但 Windows 2000 DNS 服务器在 Active Directory 中可以设置成存储区信息。

13.4 协议驱动程序

网络 API 必然携带 API 请求并且将它们翻译成低层次网络协议请求，以便在网络中传输。API 驱动程序依靠内核模式中的传输协议驱动程序完成实际的翻译工作。将 API 从下面的协议中分离出来可以赋予网络结构灵活性，使每一个 API 可以利用大量不同的协议。尽管其他的协议是有选择性的安装，如 AppleTalk 协议在 Windows 2000 服务器上与 ServicesFor Macintosh 一起安装，但 Windows 2000 包含的协议驱动程序有 Data Link Control（DLL）、NetBEUI、TCP/IP 和 NWLink。这里对每个协议进行简单的描述：

- DLC 是一个相对低级的协议，一些 IBM 主机和一些 Hewlett - Packard 网络打印机使用。在某种意义上它是一个“原始的”协议，没有网络 API 可以使用它；想要使用 DLC 协

议的应用程序必须直接与 DLC 传输协议设备驱动程序接口。

- 1985 年 IBM 和微软公司都引入 NetBEUI 协议，而且微软公司把它作为 LAN Manager 和 NetBIOS API 的缺省协议。虽然微软增强了 NetBEUI，但由于它不具有路由能力并且在 WAN 上表现很糟糕，NetBEUI 协议受到限制。NetBEUI（NetBIOS Extended User Interface）这样命名是因为它与 NetBIOS API 紧密集成，但是微软的 NetBEUI 协议驱动程序实现的协议是 NetBIOS Frame（NBF）格式。Windows 2000 包括 NetBEUI 协议主要是为了与遗留 Windows 系统（Windows NT 4 和 Consumer Windows）互操作。
- 网络的快速发展和其对 TCP/IP 协议的依赖使得 TCP/IP 成为 Windows 2000 中主要的协议，DARPA（Defense Advanced Research Projects Agency）在 1969 年开发了 TCP/IP，专门作为 Internet 基础；因此 TCP/IP 具有友好的 WAN 特征如路由能力和良好的 WAN 性能。TCP/IP 是优选的 Windows 2000 协议，也是唯一缺省安装的协议。
- Nwlink 由 Novell 的 IPX 和 SPX 协议组成。Windows 2000 包含 NWlink 是为了实现与 Novell Netware 服务器的互操作性。

一般来说在 Windows 2000 中，TDI 传输实现与它们的主要协议相关的所有协议，例如，TCP/IP 驱动程序（\Winnt\System32\Drivers\Tcpip.sys）实现 TCP、UDP、IP、ARP、ICMP 以及 IGMP。通常 TDI 传输创建表示指定协议的设备对象，使客户可以获得表示协议的一个文件对象，并且通过利用 IRP 为协议发布网络 I/O。TCP/IP 驱动程序创建 3 个设备对象，代表 3 个不同的 TDI 客户可访问协议：\Device\Tcp，\Device\Udp 和 \Device\Ip。

为了使网络 API 驱动程序对它们想使用的每个传输协议不必采用不同的接口，微软建立了 TDI（传输驱动程序接口）标准。如前所述，TDI 接口实质上是将网络请求格式化为 IRP 的方式和网络地址与通信分配方式的一个约定，附在 TDI 标准上的传输协议为它们的客户输出 TDI 接口，包括网络 API 驱动程序如 AFD 和重定向器。作为一个 Windows 2000 设备驱动程序实现的传输协议称之为 TDI 传输。由于 TDI 传输是设备驱动程序，因此它们负责把从客户接口接收的请求格式化为 IRP。

\Winnt\System32\Drivers\Tdi.sys 库中的支持函数和开发商在它们的驱动程序中包含的定义组成 TDI 接口。TDI 编程模型与 Winsock 编程模型相似。为了建立与远程服务器的连接，TDI 客户需要执行下列步骤：

1）客户分配并且格式化一个 address open TDI IRP 以分配一个地址，TDI 传输返回一个表示地址的文件对象，称之为“地址对象”（address object），这一步相当于调用 Winsock 函数的“bind”。

2）然后客户机分配并且格式化一个 connection open TDI IRP，TDI 传输返回一个表示连接的文件对象，称之为“连接对象”。这一步相当于调用 Winsock 的 socket 函数。

3）客户机用相关地址（associate address）TDI IRP 将连接对象与地址对象相关（Winsock 中没有对应的步骤）。

4）接收远程连接的 TDI 客户发布一个 listenTDI IRP 指定连接对象支持的连接数量，然后发布一个 accept TDI IRP，这样当远程系统建立连接（或产生错误）时，TDI 传输完成。这些操作相等于使用 Winsock 的 listen 和 accept 函数。

5）要想与远程服务器建立连接的 TDI 客户发布一个 Connect TDI IRP，指定这个连接对象，当连接建立（或发生错误）时 TDI 传输就结束。发布 Connect TDI IRP 相当于调用 Connect Win-

sock 函数。

对于无连接协议如 UDP，TDI 也支持无连接通信。此外，TDI 提供给 TDI 客户一个用 TDI 传输注册事件回调（event callback，即函数直接被调用）的方法。例如当它接收到来自于网络上的数据时，TDI 传输可以激活注册的客户接收回调。TDI 这一基于事件的回调特征允许 TDI 传输向它的客户通知网络事件，而且依赖于事件回调的客户接收到网络数据时，也不必提前分配资源如缓冲区等，因为它们可以查看 TDI 协议驱动程序提供的缓冲区内容。

实验：观察 TDI 活动

TDImon 是包含在配套 CD 中的一个名为 \ Sysint \ Tdimon.Exe 的实用程序，它是附接在 TCP/IP 驱动程序创建的 \ Device \ TCP 和 \ Device \ Udp 设备对象上的过滤器驱动程序。附接上后，TDImon 查看 TDI 客户发送给协议的每个 IRP。通过截获 TDI 客户事件回调注册，TDImon 还可监视事件回调。TDImon 驱动程序发送 TDI 活动的信息并在它的 GUI 中显示，在那里你可以看到操作时间、发生的 TDI 活动类型、TCP 连接的本机和远程地址或 UDP 端点的本机地址、IRP 或事件回调的结果状态码以及发送或接收的字节数等其他信息。如图 13-20 是 TDImon 监视 Microsoft Internet Explorer 浏览网页时产生的 TDI 活动的屏幕快照。

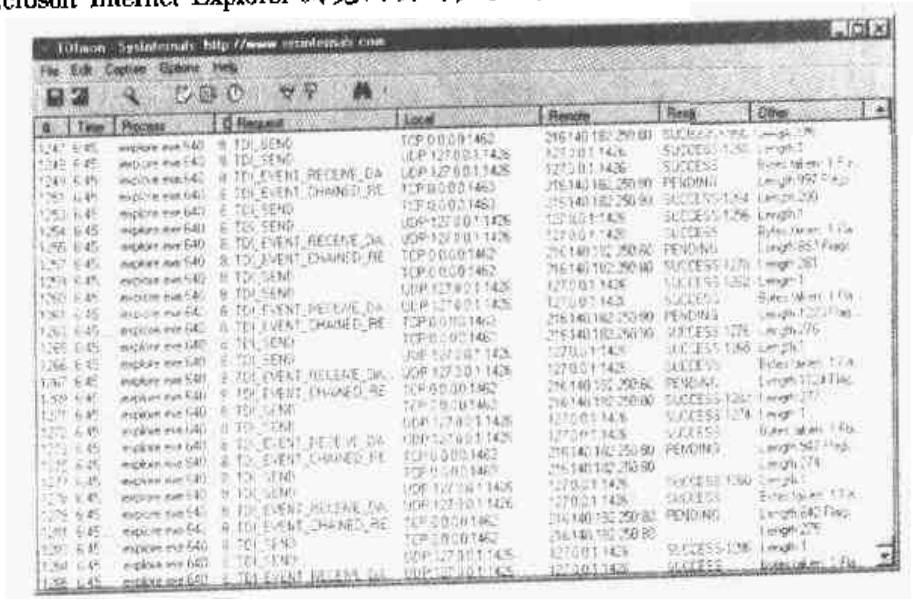


图 13-20 观察 TDI 活动

作为 TDI 操作本质上是异步的证据，Result 列中的 PENDING 代码表示操作已经初始化，但定义操作的 IRP 还未结束。考虑到其他操作的初始化，为了正确显示完成的顺序，每个 IRP 或事件回调的发布都标记一个序列号。如果在一个 IRP 完成之前其他的 IRP 发送或结束，则 IRP 的结束也标记一个序列号，如 Result 栏中显示的那样。例如，屏幕快照中标为 1278 的 IRP 在标号为 1279 的 IRP 发布后，你在 Result 列中可以看到它的标号为 1280。

13.5 NDIS 驱动程序

当一个协议驱动程序想从网络读或向网络写用它的协议格式格式化了的信息时，这个驱动

程序必须使用网络适配器。因为指望协议驱动程序识别市场上（专用网络适配器数以千计）每个网络适配器的细微差别是行不通的，网络适配器开发商提供设备驱动程序，它可以携带网络消息，并且通过开发商提供的专用硬件传输信息。1989 年，微软与 3Com 联合开发了网络驱动程序接口规范（Network Driver Interface Specification）（NDIS），它可以以设备无关的方式实现协议驱动程序与网络适配器驱动程序的通信。与 NDIS 对应的网络适配器驱动程序称之为 NDIS 驱动程序或 NDIS 小型端口驱动程序。载有 Windows 2000 的 NDIS 版本是 NDIS 5 。

Windows 2000 中，NDIS 库（ \ Winnt \ System32 \ Drivers \ Ndis.sys ）实现 TDI 传输（特别地）与 NDIS 驱动程序之间的 NDIS 边界。与 Tdi.sys 一样，NDIS 库是一个帮助文件库，NDIS 驱动程序客户用它格式化发送给 NDIS 驱动程序的命令。NDIS 驱动程序与 NDIS 库接口，实现接收请求和返回应答。图 13 – 21 表示了不同的 NDIS 相关组件之间的关系。

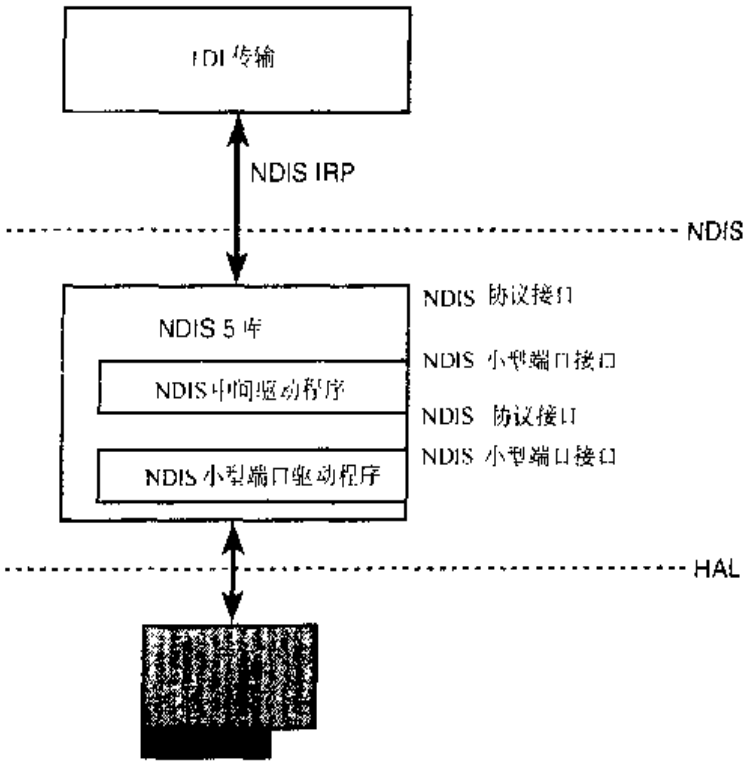


图 13-21 NDIS 组件

微软公司网络体系结构设计的目的之一就是让网络适配器开发商轻松地开发 NDIS 驱动程序，携带驱动代码以及在 Consumer Windows 和 Windows 2000 中移植驱动代码。因此，不仅仅是提供 NDIS 边界帮助文件程序，NDIS 库为 NDIS 驱动程序提供一个完整的运行环境。NDIS 驱动程序不是真正的 Windows 2000 驱动程序，因为如果没有 NDIS 库提供给它们的封装性，NDIS 驱动程序不能运行。这一保护层完全包装 NDIS 驱动程序，使 NDIS 驱动程序不接收和处理 IRP。然而，NDIS 库从 TDI 服务器接收 IRP，并且将 IRP 翻译成 NDIS 驱动程序中的调用。NDIS 驱动程序也不必担心重入（reentrancy），重入时 NDIS 库在 NDIS 驱动程序完成前一个请求的服务之前，用新的请求激活 NDIS 驱动程序。避免重入意味着 NDIS 驱动程序编写者不必担心复杂的同步。由于多处理器上可能发生并行运行，同步操作更不安全。

注意 NDIS 库从 TDI 传输和 NDIS 小端口驱动程序隐藏了这样一个事实，即 NDIS 库

利用 IRP 表示网络请求。事实确实如此，NDIS 库请求 TDI 传输通过调用 NdisAllocatePacket 分配一个 NDIS 包，并且通过调用 NDIS 库函数（如 NdisSend）将 NDIS 包传递给 NDIS 小端口。在 Windows 2000 中，NDIS 库用 IRP 实现 NDIS 包，但在 Consumer Windows 中不是这样。

虽然 NDIS 库的 NDIS 驱动程序序列化简化了开发，但序列化阻碍了多处理器的可伸缩性（Scalability）。标准 NDIS 4 驱动程序（NDIS 库的 Windows NT 4 版本）对多处理器的一些操作不能适应。微软公司在 NDIS 5 中为开发商提供了反序列化操作。NDIS 5 驱动程序可以向 NDIS 库表示它们不想被序列化；然后 NDIS 库一旦接收到描述该请求的 IRP，立即将请求转发给驱动程序。NDIS 驱动程序负责多个同时发生的请求的排队和管理，但是反序列化有利于较高层次多处理器性能的发挥。

NDIS 5 具备以下特征：

- NDIS 驱动程序可以报告它们的网络介质是否处于活动状态的信息，这样使 Windows 2000 可以在任务栏显示网络连接/非连接图标。这一特征同样允许协议和其他的应用程序意识到这一状态并作出相应的反应。例如，TCP/IP 传输将利用这一信息确定它应该何时再次评价从 DHCP 接收到的地址信息。
- TCP/IP 任务卸载允许小端口利用网络适配器的高级特性执行诸如包检校和（packet checksum）和网络协议安全（IPSec）等操作。任务卸载可以通过减轻 CPU 负担改善系统性能。
- 包快速发送允许网络适配器硬件将目标位置不对的的包路由给远程系统，而不必交给 CPU 处理。
- Wake - on - LAN 允许一个 Wake - on - LAN - capable 网络适配器使 Windows 2000 脱离电源挂起状态。能够激发网络适配器向系统发送信号的事件有介质连接（如向网络适配器中插入网络电缆）、协议注册的协议专有的模式接收（TCP/IP 传输对于地址解析协议［ARP］请求要求被唤醒）以及对于 Ethernet 网络适配器 magic 包（一个包含网络适配器的 Ethernet 地址的 16 个连续拷贝）的接收。
- 面向连接 NDIS 允许 NDIS 驱动程序管理面向连接介质如 ATM（异步转换模式）设备（面向连接 NDIS 稍后将详细讨论）。

NDIS 库为 NDIS 驱动程序提供的与网络适配器硬件的接口通过直接翻译为 HAL 中对应函数的函数是可用的。

实验：列出加载的 NDIS 小端口

Ndiskd 内核调试器扩展库包括！miniports 和！miniport 命令，它可以让你用内核调试器列出已加载的小端口，给出小端口块（Windows 2000 用于跟踪小端口的数据结构）的地址，查看小端口驱动程序的详细信息。下面的例子表示用！miniports 和！miniport 命令显示所有小端口的情况，然后指明了作为系统与 PCI Ethernet 网络适配器接口的小端口（注意 WAN 小端口驱动程序与拨号连接一起运行）。

```
kd> .load ndiskd
Loaded ndiskd extension DLL

kd> !miniports
Driver verifier level: 0
Failed allocations: 0
Miniport Driver Block: 817aa610
  Miniport: 817b1130 RAS Async Adapter
Miniport Driver Block: 81a1ef30
  Miniport: 81a1ea70 Direct Parallel
Miniport Driver Block: 81a21cd0
  Miniport: 81a217f0 WAN Miniport (PPTP)
Miniport Driver Block: 81a22650
  Miniport: 816545f0 WAN Miniport (NetBEUI, Dial Out)
  Miniport: 81a21130 WAN Miniport (IP)
Miniport Driver Block: 81a23290
  Miniport: 81a22130 WAN Miniport (L2TP)
Miniport Driver Block: 81a275f0
  Miniport: 81a25130 Intel 8255x-based PCI Ethernet Adapter (10/100)

kd> !miniport 81a25130
 Miniport 81a25130 : Intel 8255x-based PCI Ethernet Adapter (10/100)
    Flags          : 20413208
                     BUS_MASTER, INDICATES_PACKETS, IGNORE_REQUEST_QUEUE
                     IGNORE_TOKEN_RING_ERRORS, NDIS_5_0,
                     RESOURCES_AVAILABLE, DESERIALIZED, MEDIA_CONNECTED,
                     NOT_SUPPORTS_MEDIA_SENSE,
    PnPFlags       : 00010021
                     PM_SUPPORTED, DEVICE_POWER_ENABLED, RECEIVED_START
    CheckforHang interval: 2 seconds
        CurrentTick    : 0001
        IntervalTicks : 0001
   InternalResetCount : 0000
   MiniportResetCount : 0000

   References: 3
   UserModeOpenReferences: 0

   PnPDeviceState : PNP_DEVICE_STARTED
   CurrentDevicePowerState : PowerDeviceD0
   Bus PM capabilities
   DeviceD1:                1
   DeviceD2:                1
   WakeFromD0:              0
   WakeFromD1:              1
   WakeFromD2:              0
   WakeFromD3:              0

   SystemState              DeviceState
   PowerSystemUnspecified   PowerDeviceUnspecified
```

```
        S0                          D0
        S1                          D1
        S2                          PowerDeviceUnspecified
        S3                          PowerDeviceUnspecified
        S4                          D3
        S5                          D3
        SystemWake: S1
        DeviceWake: D1

    WakeupMethodes Enablec 6:
        WAKE_UP_PATTERN_MATCH  WAKE_UP_LINK_CHANGE
    WakeUpCapabilities of the miniport
        MinMagicPacketWakeUp: 4
        MinPatternWakeUp: 4
        MinLinkChangeWakeUp: 4

        Current PnP and PM Settings:          : 00000030
                        DISABLE_WAKE_UP, DISABLE_WAKE_ON_RECONNECT,

        Allocated Resources:
            Memory: f4100000, Length: 1000
            IO Port: 1440, Length: 40
            Memory: f4000000, Length: 100000
            Interrupt Level: 9, Vector: 9
        Translated Allocated Resources:
            Memory: f4100000, Length: 1000
            IO Port: 1440, Length: 40
         Memory: f4000000, Length: 100000
         Interrupt Level: 12, Vector: 39
    MediaType      : 802.3
    DeviceObject   : 81a25030, PhysDO : 81a93cd0  Next DO: 81a63030
    MapRegisters   : 819fc000
    FirstPendingPkt: 0
    SingleWorkItems:
      [0]: 81a254e8 [1]: 81a254f4 [2]: 81a25500 [3]: 81a2550c
      [4]: 81a25518 [5]: 81a25524
    DriverVerifyFlags       : 00000000
    Miniport Open Block Queue:
      8164b888: Protocol 816524a8 = NBF, ProtocolContext 81649030
      8191f628: Protocol 81928d88 = TCPIP, ProtocolContext 8191f728
     Miniport Interrupt 81a00970
```

对于被检测的小端口来说，Flags 字段表示小端口支持反序列化操作（DESERIALIZED）、介质现在处于活动状态（MEDIA - CONNECTED）以及它是一个 NDIS 5 小端口驱动程序（NDIS-5-0）。同时列出的还有网络适配器系统对设备电源状态的映射以及即插即用管理器分配给网络适配器的总线资源（关于电源状态映射的信息参见9.2.4节“电源管理器”）。

13.5.1 NDIS 小端口特征

NDIS 模型也支持混和 TDI 传输—NDIS 驱动程序，称为“NDIS 中介设备驱动程序”。这些程序位于 TDI 传输与 NDIS 驱动程序之间。对于 NDIS 驱动程序来说，NDIS 中介驱动程序就像一个 TDI 传输；而对于 TDI 传输，NDIS 中介驱动程序好像一个 NDIS 驱动程序。由于 NDIS 中介驱动程序位于协议驱动程序和网络驱动程序之间，所以它们能够监视系统中发生的所有网络阻塞现象。为网络适配器提供容错和网络负载平衡选择的软件，如微软的 Networt Load Balancing Provider 就是以 NDIS 中介驱动程序为基础的。作为微软公司的服务质量 Quality of Service（QoS）实现的一部分，包调度程序也是中介驱动程序的一个例子。

13.5.2 面向连接的 NDIS

NDIS 5 引入了一种新的 NDIS 驱动程序类型，即面向连接 NDIS 小端口驱动程序。因此，对面向连接网络硬件（如 ATM）的支持是 Windows 2000 固有的，它产生 Windows 2000 网络体系结构中连接管理和建立的标准。面向连接 NDIS 驱动程序使用许多与标准 NDIS 驱动程序相同的 API，然而，面向连接 NDIS 通过建立网络连接发送数据包，而不是将数据包放在网络介质上。

除了对面向连接介质的小端口支持外，NDIS 5 包含支持面向连接小端口驱动程序的驱动程序定义：

- 调用管理器（call manager）是为面向连接的客户（稍后讨论）提供调用安装和拆卸服务的 NDIS 驱动程序。调用管理器用面向连接小端口与其他网络实体如网络开关或其他调用管理器交换信号消息。调用管理器支持一个或多个信号协议，如 ATM User - Network Interface（UNI）3.1。
- 集成化小端口调用管理器（MCM）也是为面向连接客户提供调用管理器服务的面向连接小端口驱动程序。MCM 实质上是一个具有内建调用管理器的 NDIS 小端口驱动程序。
- 面向连接的客户使用调用管理器或 MCM 的调用安装和拆卸服务以及面向连接 NDIS 的小端口驱动程序的发送和接收服务。面向连接客户可以为网络椎栈中的高层次提供自己的协议服务，或者运行一个仿真层，该层是无连接的遗留协议与面向连接介质的接口。LAN 仿真（LANM）是面向连接客户实现仿真层的一个例子，它隐藏了 ATM 的面向连接特征，向它上面协议显示了一个无连接的介质（如 Ethernet）。

图 13-22 表示这些组件之间的关系。

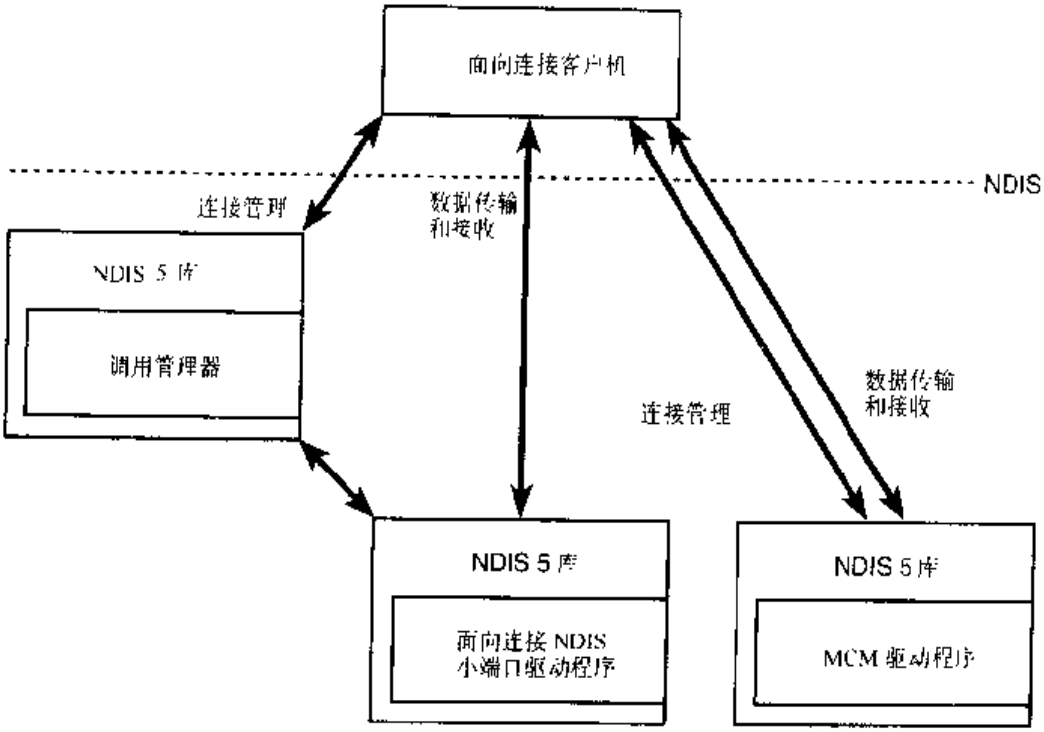


图 13-22 面向连接的 NDIS 驱动程序

实验：利用网络监视器获取网络包

Windows 2000 服务器带有一个称为网络监视器的工具。网络监视器可以让你通过安装 NDIS 中介设备驱动程序，捕获流过系统中一个或多个 NDIS 小端口驱动程序的信息包。在使用网络监视器（Network Monitor）之前，你需要在你的系统中安装 Windows 2000 Network Monitor Tools。为了安装这些工具，你要在 Control Pannel 中打开 Add / Romove Programs，选择 Add / Remove Windows Component。选择 Management And Monitoring Tools，按 Details，选择 Network Monitor Tools，然后按 OK

安装完 Network Monitor Tools 后，你可通过下列步骤安装 Network Monitor：

1）右击桌面上 My Network Places 图标，激活 Network And Dial－up Connection 应用程序，然后选择 Properties；或者从开始菜单中的 Seetings 中选择 Network And Dial－up Connections。

2）右击本机网络适配器，选择 Properties，然后在 Local Area Connection Properties 对话框中单击 Install 按钮。

3）选择 Protocol，单击 Add 按钮。

4）选择 Network Monitor Drivers，然后单击 OK。

安装完毕，你可以通过从开始菜单中的管理器文件夹程序组中选择 Network Monitor 启动它。

Network Monitor 可能问你想监视哪个连接。选择要监视的连接后，在工具栏中按下 Start Capture 按钮开始监视。在监视的连接上执行产生网络活动的操作，在你看到 Network Monitor 已经捕获到的信息包时，点击 Stop And View 按钮（与“眼镜”图标相邻的停止按钮）停止监视。Network Monitor 会显示图 13-23 捕获数据。

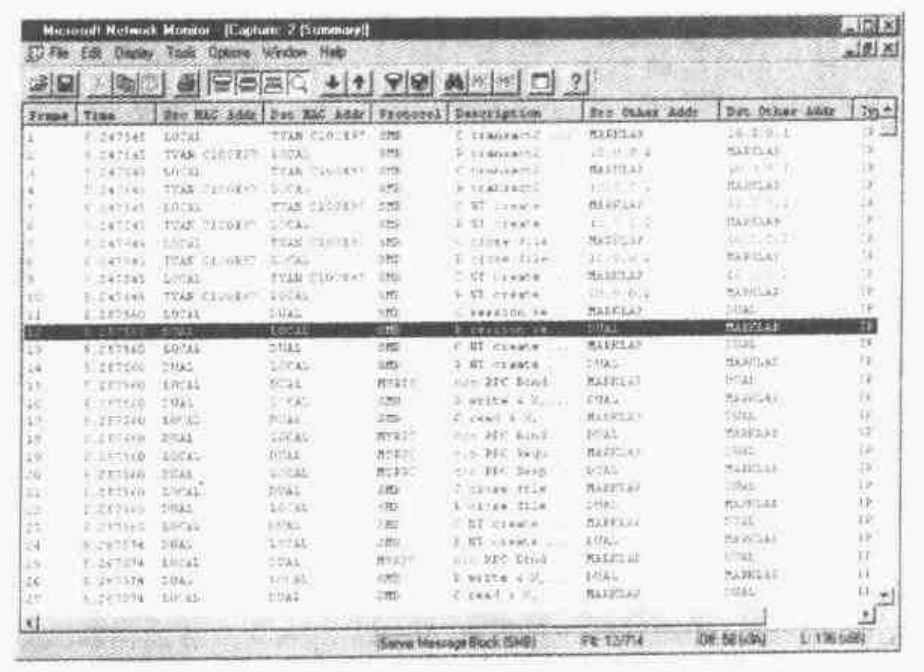


图 13-23 Net Work Monitor 显示捕获的数据

图 18-23 显示了当远程文件被访问时 Network Monitor 捕获到的 SMB（CIFS）包。如果双击一行，Network Monitor 会切换到显示不同分层应用程序和协议头信息的窗口，如图 13-24 所示。

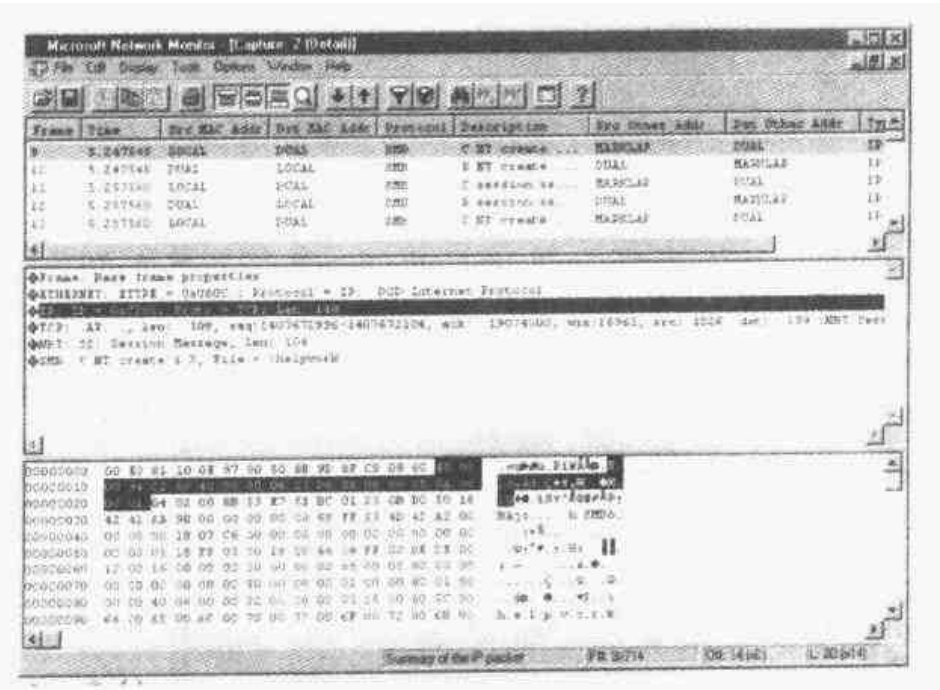


图 13-24 显示 SMB 包

Network Monitor 也具有许多其他特征，如捕获触发器和过滤器等，这些特征使 Network Monitor 成为一个强有力的网络故障排除工具。

13.6 绑定

Windows 2000 网络体系结构的最后一个令人困惑的地方是不同层的组件互相叠放的方式——网络 API 层、TDI 传输驱动层、NDIS 驱动层。连接不同层的过程称为 binding，即绑定。如果你曾经利用 Network Properties 添加或删除过网络组件从而改变你的网络设置，那么你可能已经看到过绑定过程。

当你安装一个网络组件时，必须为它提供一个 INF 文件（第 9 章中讨论过 INF 文件。）。该文件指出安装 API 例程必须在安装和设置组件之后，包括绑定相关性和绑定关系。开发商可以为专用组件指定绑定依赖性以使 SCM 不仅按正确的顺序安装组件，而且只要专用组件所依赖的其他组件在系统中，它就可以安装该组件。绑定关系建立不同层组件之间的连接，它由绑定引擎在组件 INF 文件其他信息的帮助下确定。这些连接指定每层的网络组件可以使用其下面那层的哪些组件。

例如，Workstation 服务（重定向器）会自动绑定出现在系统中的 NBF（NetBEUI）和 TCP/IP 协议。绑定顺序决定绑定的优先权，你可以在 Advanced Settings 对话框的 AdaptersAnd Bindings 标签栏中检验绑定顺序，如图 13-25 所示（参见 9.3 节“网络资源名字解析”中关于如何启动 Advanced Settings 对话框的介绍。）当重定向器接收到一个访问远程文件的请求时，它就将这个请求同时递交给两个协议。当应答产生时，重定向器等待直到它接收到来自于任何高级优先协议驱动程序的应答。只在这时重定向器才将结果返回给调用程序。因此，重新组织绑定，使高优先权的绑定对网络中的多数计算机最有效或最适用是十分有益的。你可以用 Advanced Settings 对话框删除绑定。

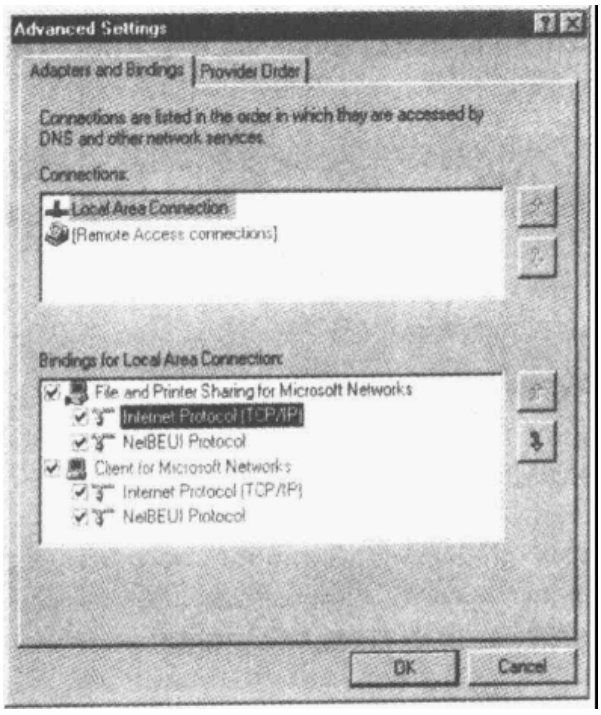


图 13-25 用 Advanced Settings 对话框编辑绑定

Bind值存储组件的绑定信息，它位于网络组件注册表配置键的Linkage子键中。例如，如果你检验HKLM \ SYSTEM \ Current Control Set \ Service \ LanmanWorkstation \ Linkage \ Bind，就可了解Workstation服务程序的绑定信息。

13.7 分层网络服务

Windows 2000包含建立在API和本章讨论的组件基础上的网络服务。描述这些服务的功能和详细的内部实现不是本书涉及的内容，但在本节中，我们将对远程访问、Active Directory、Network Load Balancing（网络装载平衡）、File Replication Service（FRS，文件复制服务）以及DFS（分布式文件系统）进行简略叙述。此外，Windows 2000支持一些基于TCP/IP协议扩展的服务，它们包括网络地址翻译（NAT）、网络协议安全（IPSec）和QoS，我们将一一介绍。

13.7.1 远程访问

远程访问在Windows 2000 Server中可用，允许远程访问客户机与远程访问服务器连接并且访问远程网络资源，如文件、打印机和网络服务程序。就好像客户机与远程访问服务器的网络具有物理连接一样。

Windows 2000提供下列两种远程访问：

- 拨号远程访问用于通过电话和其他通信设备与远程服务器连接的客户机。通信介质用来创建客户机与服务器之间暂时的物理或虚拟连接。
- 虚拟专用网（Virtual Private Network，VPN）远程访问允许VPN客户机通过IP网络如Internet与服务器建立虚拟的点对点的连接。

远程访问不同于远程控制方案，因为远程访问作为Windows 2000网络的连接代理，而远程控制方案在服务器上运行应用程序，为客户提供一个用户界面。

13.7.2 活动目录

活动目录（Active Directory）是轻型目录访问协议（Lightweight Directory Access Protocol，LDAP）目录服务的Windows 2000实现。活动目录基于存储表示Windows 2000网络中应用程序定义的资源的对象的数据库。例如，Windows 2000域的结构和成员，包括用户帐号和密码信息，被存储在Active Directory中。

架构（schema）指定对象类和定义对象属性的属性。活动目录架构中的对象是按层次排列的，与注册表的逻辑组织特别相似，在这里容器对象可以存储其他的对象，包括其他的容器对象。（见第8章关于容器对象的更多信息。）

活动目录支持大量的API，客户可以用它访问活动目录数据库中的对象：

- LDAP C API是一个使用LDAP网络协议的C语言API。用C或C++写的应用程序可以直接使用该API，其他语言写的应用程序可以通过传输层访问它。
- Active Directory Service Interfaces（ADSI）是一个活动目录的COM接口，它抽象了LDAP编程的细节。ADSI支持多种语言，包括Microsoft Visual Basic、C和Microsoft Visual C++。Microsoft Windows Script Host（WSH）应用程序可使用ADSI。

- Messaging API（MAPI）支持与 Microsoft Exchange 客户和 Outlook Address Book 客户应用程序的兼容。
- Security Account Manager（SAM）API 建立于活动目录的顶层，为登录身份验证包如 MSVI-0（\Winnt\System32\MSVI-0.dll），用于遗传的 NT LAN Manager 验证），和 Kerberos（\Winnt\System32\Kdcsvc.dll）提供验证。
- Windows NT 4 网络 API（Net API）被 Windows NT 4 客户用来通过 SAM 访问 Active Directory。

活动目录作为一个 Win32 数据库文件 \Winnt\Ntds\Ntds.dit 实现，可以在一个域中的多个域控制器上复制。活动目录服务程序是运行在本机安全权限子系统（Local Security Authority SubSystem LASASS）进程中的一个 Win32 服务程序，它使用 DLL 管理数据库。为保护数据库的完整性，这些 DLL 实现了数据库的磁盘上的结构，并且提供基于事务的更新。活动目录数据库存储以 Extensible Storage Engine（ESE）数据库为基础，ESE 由 Microsoft Exchange Server 5.5 客户/服务器报文和群件使用。图 13-26 为活动目录的系统结构。

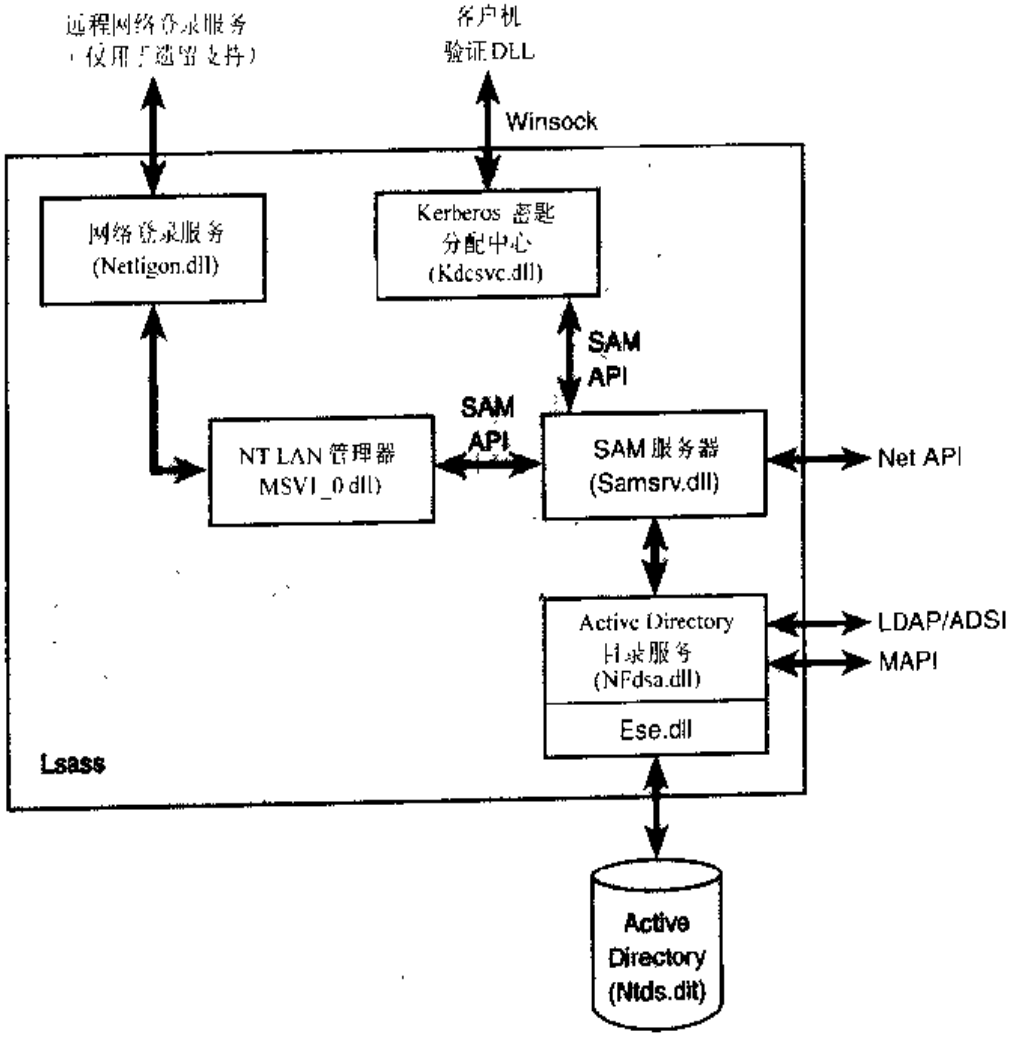


图 13-26 活动目录体系结构

13.7.3 网络负载平衡

正如我们本章前面所述的，网络负载平衡是以 NDIS 中介驱动程序技术为基础的，它包括

在 Windows 2000 Advanced Server 中，网络负载平衡允许建立包含 32 台计算机的集群，在网络负载平衡中称为集群主机（cluster hosts）。集群维护唯一的虚拟 IP 地址，它是为客户访问公开的，客户请求到达集群中的所有计算机，然而只有集群主机响应请求。网络负载平衡 NDIS 驱动程序用分布式方式在可用的集群主机之间进行有效的客户空间分割。这样，每台主机负责处理自己的那部分来访客户请求，而且总是有且仅有一台主机处理每个请求。决定处理客户机请求的那台集群主机允许请求传送到 TCP/IP 协议驱动程序并最终到达服务器应用程序；而其他的集群主机不是这样。如果一台集群主机失败，则其他的集群主机意识到该主机不能再处理请求，就会把来访的客户请求重新分配给剩余的集群主机。新的请求不再发送给失败的主机。另一台集群主机添加到集群中取代它，然后无缝地开始处理客户请求。

网络负载平衡不是一个通用的集群方案，因为与客户机通信的服务器应用程序必须具有一定的特性：首先，它必须基于 TCP/IP 协议；其次，它必须能够在网络负载平衡集群中处理任何系统中的客户请求。后者意味着为了服务于客户请求必须访问共享状态的应用程序必须自己管理共享状态——网络负载平衡不包含在集群中自动分布共享状态的服务程序。理想的适合于网络负载集群的应用程序包含一个 Web 服务器，它服务于静态内容、Windows Media Server 和 Terminal Services。图 13-27 是一个网络负载平衡操作的例子。

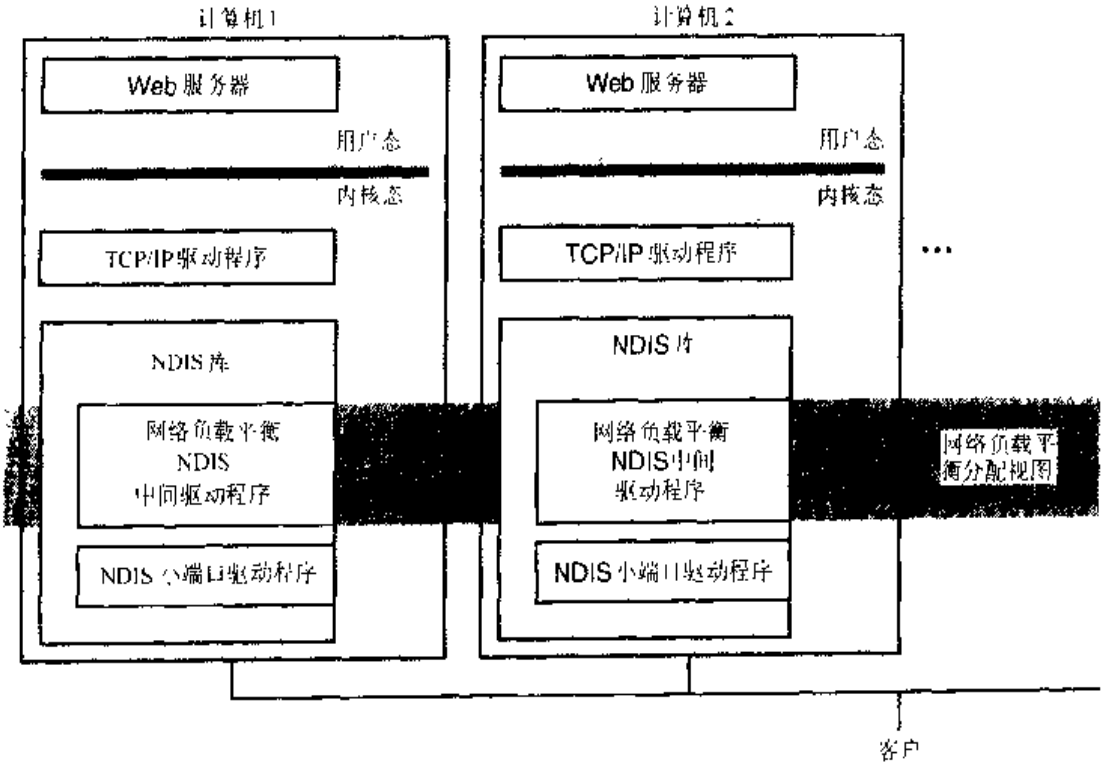


图 13-27 网络负载平衡操作

13.7.4 文件复制服务程序

文件复制服务（FRS）包含在 Windows 2000 Server 中。它的主要目的是复制域控制器的 \SYSVOL 目录下的内容。该目录下 Windows 2000 域控制器存储了登录脚本和组策略（组策略允许管理员为域中的计算机定义使用和安全策略。）。除此之外，FRS 可以用于复制系统之间共享

的分布式文件系统（DFS）。FRS 允许分布式多管理者复制（distributed multimaster replication），使任何服务器都能执行复制活动，当改变一个复制目录或文件时，这种变化会被传播到其他的域控制器。

FRS 中的一个基本概念是复制组（replica set），它由两个或多个系统组成，系统按照管理员定义好的进度表和拓扑关系在系统之间复制目录树内容。只有 NTFS 卷上的目录可以被复制，因为 FRS 依赖 NTFS 更改日志（change journal）探测复制组目录中文件的变化。由于 FRS 基于多管理者复制，所以理论上它可以支持几百个甚至上千个系统组成一个复制组，而且复制组中的计算机可以与任意的网络拓扑结构连接（如环形、星形或网格形），计算机也可以是多个复制组的成员。

FRS 作为一个 Win32 服务实现，它利用加密的身份验证 RPC 进行运行在不同计算机上的自身实例程序之间的通信。另外，由于 Active Directory 含有它自己的复制能力，FRS 利用 Active Directory API 从域 Active Directory 中获取 FRS 的设置信息。

13.7.5 分布式文件系统

分布式文件系统（DFS）是置于 Workstation 服务顶层的服务，它将所有的文件共享资源连接成一个单独的名字空间。文件共享资源可以驻留在相同或不同的计算机上，DFS 用位置透明的方式为客户机提供访问这些资源的途径。DFS 名字空间的根必须是一个定义于 Windows 2000 服务器上的文件共享资源。

除了发布统一的网络资源名字空间外，DFS 还通过 DFS 复制组提供其他的优质服务，DFS 复制组以 FRS 复制组为基础。管理器可以从两个或更多个共享资源文件建立 DFS 复制组，因此 FRS 在复制组的共享资源之间拷贝数据以保持它们内容的同步性。DFS 通过随机地选择复制组中的成员，提供有限形式的负载平衡，完成客户对复制组中数据的请求。此外，当一个成员不可用时，DFS 可以将请求路由到复制组的一个或多个工作成员，获得高实用性。

组成 DFS 系统结构的组件如图 13-28。DFS 服务器端的实现包括一个 Win32 服务（\Winnt

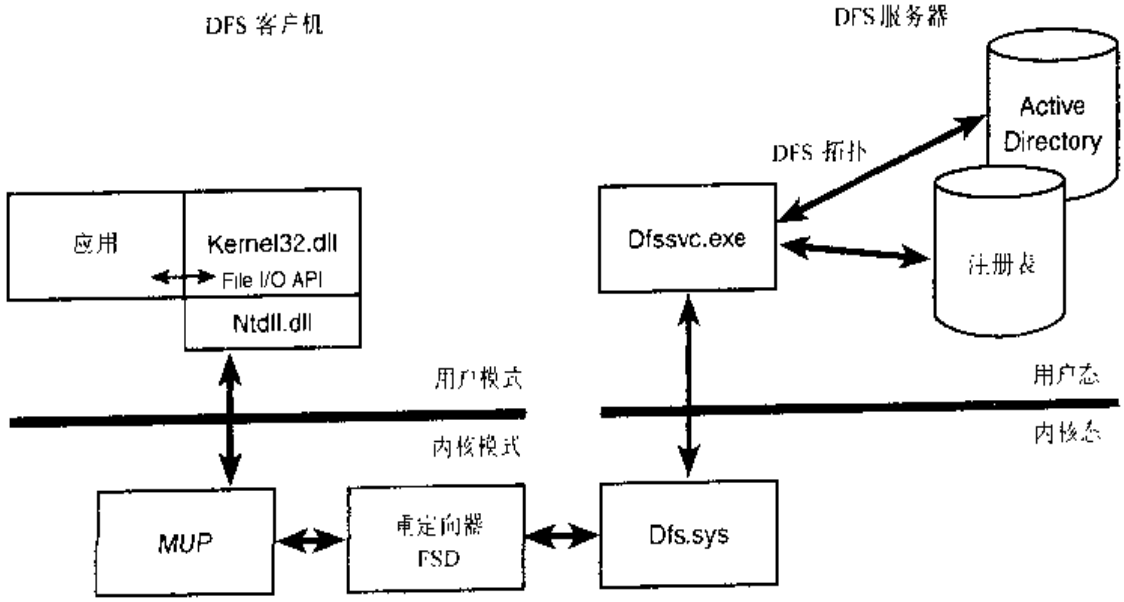


图 13-28 DFS 组件

\System32\Dfssvc.exe）和设备驱动程序（\Winnt\System32\Drivers\Dfs.sys）。DFS服务程序负责输出DFS拓扑管理接口和维护注册表（非Active Directory系统）或Active Directory中的DFS拓扑结构。DFS驱动程序接收到客户请求时就执行拓扑查询工作，将客户机指向其请求的文件驻留系统。

在客户端，DFS的实现依赖于MUP驱动程序、NetWare与CIFS重定向器的支持。当客户机发出一个文件I/O请求在DFS名字空间指定一个文件时，客户端的MUP驱动程序利用适当的重定向器与DFS服务器通信。

13.7.6 TCP/IP扩展

其他的Windows 2000网络服务程序依赖于使用专用接口与TCP/IP协议驱动程序集成在一起的添加的驱动程序扩展TCP/IP协议驱动程序的基本网络特性。它们包括网络地址翻译（NAT）、网络协议安全（Internet Protocol Security）以及服务质量（Quality of Service）。

1. 网络地址翻译

NAT是一个路由选择服务，它允许多个本机IP地址映射到一个单一IP地址。如果没有NAT，必须分配给LAN的每台计算机一个公共IP地址以实现网络通信。NAT允许分配给LAN的计算机一个IP地址，而其他的计算机通过它连接到网络上。NAT在LAN地址和公共IP地址之间翻译，把包从因特网路由到适当的LAN计算机。

Windows 2000上NAT组件包括一个与TCP/IP栈接口的NAT设备驱动程序和管理器用于定义地址翻译的编辑器。NAT可作为一个协议用Routing And Remote Access MMC插件安装，或者通过使用Network And Dial-up Connection工具设置Internet连接安装。（尽管用Routing And Remote Access MMC插件安装时NAT更易设置）。

2. 因特网协议安全（Internet Protocol Security）

Internet Protocol Security（IPSec）与Windows 2000 TCP/IP栈集成在一起，它为IP数据提供保护以防止探听（spoofing）和操纵，并且防止基于IP的进攻。这两个目的均通过基于密码技术的保护服务程序、安全协议以及动态键管理程序实现。基于IPSec的通信具有下列特性：

- 身份证验证实IP消息的来源和完整性。
- 完整性保护，防止IP数据在传递过程在没有探测到的情况下被修改。
- 保密性利用加密措施确保只有消息的合法接收者才能解密消息的内容。
- 反重放（antireplay）确保每一个消息包是独一无二的，不能重复利用。这些特性防止探听器应答捕获到的消息以建立会话层或对数据进行非法访问。

Windows 2000中IPSec依赖于存储于Active Directory中的组策略进行配置，它利用Active Directory的Kerberos 5身份验证程序对参与IPSec消息交换的计算机进行身份验证。IPSec使用基于Windows 2000 Cryto API认证服务程序的私用有/公用密匙对加密和解密IPSec消息数据，并且作为它身份验证过程的一部分设置口令。（见第12章“加密文件系统安全”一节关于Crypto API的详情）。

IPSec的实现由IPSec设备驱动程序（\Winnt\System32\Drivers\IPSec.sys）组成，它集成了TCP/IP协议驱动程序。在用户空间，策略代理从Active Directory获得IPSec配置信息，并且将IPSec过滤信息（使用IP通信的IPSec地址过滤器）传递给IPSec驱动程序，将安全设置传递

给 Internet Key Exchange（IKE）模块，IKE 模块等待来自 IPSec 驱动程序的安全连接请求，并协议请求，将结果返回给 IPSec 驱动程序以在身份验证和加密期间使用。

3. 服务质量（QoS）

如果不采取特殊措施，IP 通信量按照先到先服务原则在网络中传递。应用程序对它们消息的优先权没有控制能力，它们可能表现出突发性的网络特性，即偶尔应用程序获得高吞吐量和低延迟，但在另外一些情况下网络的性能极低。尽管大多数情况下这种服务可以接受，但越来越多的网络应用程序要求更加一致的服务水平或者叫服务质量（QoS）保证。视频会议、介质流以及企业资源规划（ERP）就是要求良好的网络性能的应用例子。QoS 允许应用程序设定最小带宽和最大延迟，这一要求只有在发送器和接收器之间的每个网络软硬件组件支持 QoS 标准，如 IEEE8020 IP 标准时才能满足。IEEE 8020 IP 是一个工业标准，它设定 QoS 包的格式以及 OSI 的第二层设备（开关和网络适配器）如何对它们作出反应。

Windows 2000 QoS 支持一部分以微软定义的 Winsock ATI 为基础的 API，这些 API 允许应用程序请求 QoS 在它们的 Winsock 套接字上通信。例如，一个应用于程序用 WSCInstall QoSTemplate 安装一个 QoS 模板指定预定的带宽和存储量。（只有具有管理特权的应用程序才可以使用 QoS）。另一个 API——流量控制（TC）API——允许管理应用程序更精确地控制计算机连接的网络中的通信流量。

Windows 2000 QoS 实现的核心是 Resource Reservation Setup Protocol（RSVP）Win32 服务（\ Winnt \ System32 \ Rsvp.exe），如图 13-29 所示。RSVP Winsock 服务提供程序（\ Winnt \ System32

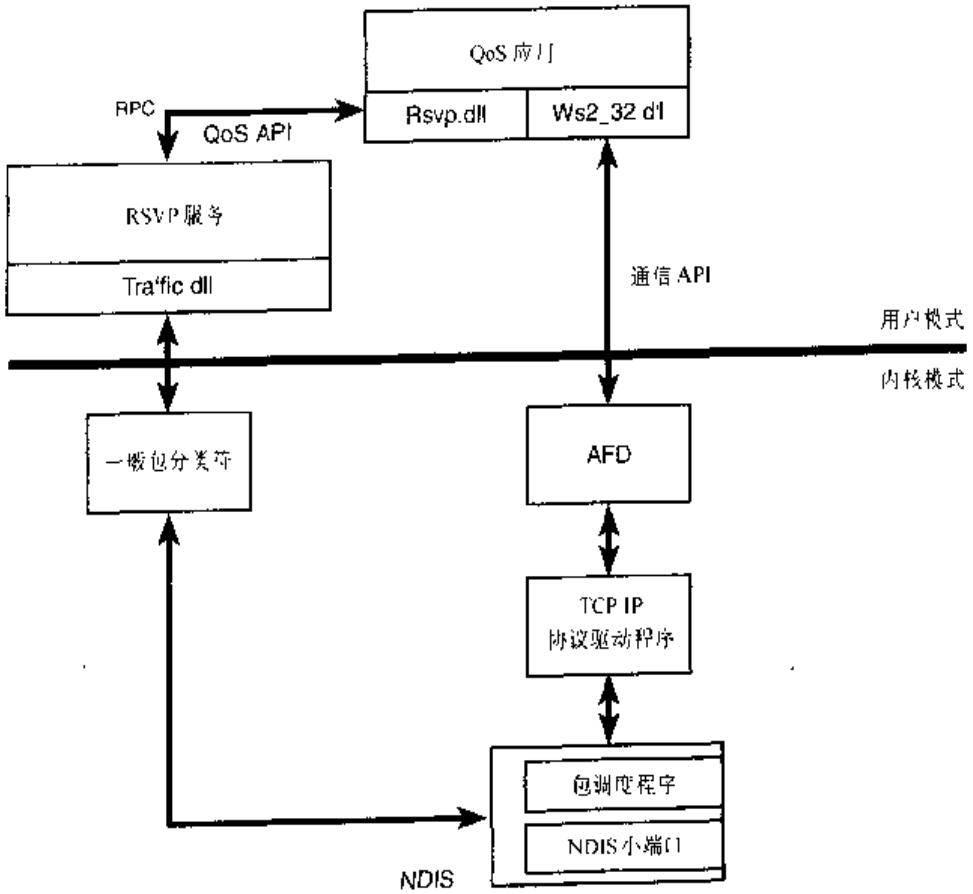


图 13-29 QoS 体系结构

\ Rsvp.dll ）通过 RPC 使应用程序 QoS 请求与 RSVP 服务程序通信。反过来，RSVP 服务程序用 TC API 控制通信流量。TCP API 在 \ Winnt \ System32 \ Traffic.dll 中实现，它向 Generic Packet Classifier（GRC）驱动程序（ \ Winnt \ System32 \ Drivers \ Msgpc.sys）发送 I/O 控制命令。GPC 驱动程序与 QoS 包调度程序 NDIS 中介驱动程序（ \ Winnt \ System32 \ Drivers \ psched.sys）紧密连系，控制从计算机到网络的包流量以至于承诺给特定应用程序的 QoS 水准可以得到满足，并且保证适当的 QoS 报头信息添加到期望的包上。

13.8 小结

Windows 2000 网络体系结构为网络 API、网络协议驱动程序以及网络适配器驱动程序提供了一个灵活的基础结构。Windows 2000 体系结构充分利用 I/O 分层性，支持网络扩展性随计算机网络的发展而发展。当新的协议出现时，开发商可以编写 TDI 传输在 Windows 2000 上实现协议。同样地，新的 API 可以与已有的 Windows 2000 协议驱动程序通信。最后，Windows 2000 上运行的大量网络 API 给予网络应用程序开发商充分的实现方法空间，每一种都具有不同编程模型和协议支持。

术 语 表

access - control list（访问控制表，ACL）—— 枚举谁拥有对对象的访问权限的安全描述符的一部分。对象的拥有者能够改变访问控制表，允许或拒绝其他对对象的访问。ACL 由 ACL 头和空项或更多的的访问控制项（ACE）结构构成。带有空的访问控制项的 ACL 被称为空 ACL，表明没有用户拥有对对象的访问权限。

access token（访问令牌）—— 一个包含有进程或线程的安全标识符的数据结构，它包含有安全 ID（SID）、用户所属的用户组表以及允许和禁止的权限表。每一个进程都有一个主要的从父进程缺省继承来的访问令牌。

add - device routine（添加设备例程）——由驱动程序实现的支持即插即用的例程。当驱动程序检测到由它负责的设备时，即插即用管理器通过这个例程发出一个驱动程序通知。在这个例程中，驱动程序典型地分配一个设备对象用来代表设备。

Address Windowing Extensions（地址窗口扩展，AWE）—— Window 2000 中一种允许 32 位应用程序分配 64GB 物理内存并把视图或窗口映射到 2GB 虚拟内存的机制。使用 AWE，把管理虚拟内存向物理内存映射的任务落在了程序员身上，但它解决了直接访问，比在 32 位进程地址空间中任何时候都能够映射更多的物理内存。

affinity mask（相似性掩码）—— 一种指定允许线程在其上处理的处理器的位掩码。初始线程相似性掩码从进程相似性掩码继承。

Aging（年龄）—— 一个对页面计数的进程，如果自上次对工作集进行修整扫描以来，页面没有被访问，则年龄增加 1。在单处理器系统上，如果页面近来没有被访问，工作集管理器将删除页面。该过程首先清除硬件页面表项（PTE）的访问位，然后再检查该位以确定页面是否被访问。如果访问位仍处于清零状态，这表明在扫描之间页面没有被访问，这样便增加年龄。最后，页面的年龄被用来确定候选页面，并将它从工作集中删除。

alterable wait state（可报警的等待状态）——指一种线程状态，线程或者在对象句柄上等待并指明它的等待状态是可报警的（使用 Win32 WaitforMultipleObjectsEx 函数）或者直接测试看它是否有挂起的异步过程调用（APC）（使用 SleepEx）。在两种情况下，如果用户模式 APC 是挂起的，内核中断（报警）线程，并把控制交给 APC 例程，当 APC 例程完成后恢复线程的执行。仅仅当线程处于可报警的等待状态时，用户模式 APC 方可交付给线程。

allocation granularity（分配粒度）—— 虚拟内存分配的粒度。Windows 2000 按照系统分配粒度值定义的基值对齐保留的进程地址空间的每个区域，该值可以从 Win32 GetSystemInfo 函数中获得。目前，它为 64KB。之所以选择这个数字，是考虑到对未来大尺寸页面的处理器支持时（例如，到 64KB），改变关于分配对齐的假设的风险会因此得到降低。（Windows 2000 内核模式代码不受这种限制；它可以以单一页面粒度保留内存）。

alternative hive（可替换的 hive）—— 作为关键的 SYSTEM hive 的备份。可替换的 hive 以 Sys-

tem.alt 的方式存储在 \ Winnt \ System32 \ Config 中。当 hive 同步地刷新系统 hive 的脏扇区时，hive 也同步地更新 System.alt hive。如果配置管理器在系统引导时候检测到 SYSTEM hive 已遭到破坏时，配置管理器将加载可替换的 hive。如果该 hive 是可用的，它使用可替换的 hive 更新原来的 SYSTEM hive。

APC queue（异步进程调用队列）——一个异步过程调用在其中等待执行的队列。APC 队列（一个用于用户模式，一个用于内核模式）是线程专用的——每一个线程都有自己的 APC 队列（不像 DPC 队列，它适用于整个处理器）。

asymmetric mulitprocessing（非对称式多处理，ASMP）——运行在多处理器系统上，通常选择一个处理器用来运行操作系统代码，而其他处理器用来执行用户代码。

asynchronous I/O（异步 I/O）——一种 I/O 模型，它允许应用程序发出一个 I/O 请求，在请求被完成期间继续执行。这种 I/O 类型可以提高应用程序的吞吐量，因为它允许应用程序在进行 I/O 操作的过程中继续其他工作。

asynchronous procedure call（APC，异步过程调用）——为用户程序和系统代码提供在特定用户线程（进而特定进程地址空间）环境中执行代码的方法的一种功能。一个 APC 既可以是内核模式的也可以是用户模式的（内核模式 APC 不需要目标线程的允许以便在该线程的环境中运行，而用户模式 APC 则需要。）

atomic transaction（原子事务）——一种处理数据库修改的技术，这样系统的失败不会影响数据库的正确性和完整性。原子事务的基本原则是一些称为事务的数据库的操作，具有要么全有要么全无的属性。构成事务的单独磁盘更新必须原子地执行。这也就是说，一旦事务开始执行，磁盘更新必须完成。如果系统失败中断了事务，完成的那部分必须撤消，或者重新执行。重新执行操作把数据库返回已知的一致状态，就象事务没有发生一样。参见 transaction。

attribute list（属性列表）——在 NTFS 文件头中的特殊文件属性，它包含有附加的属性。属性列表主要在当一个文件的主文件表（MFT）中有太多属性项的情况下创建。属性列表属性包含有每个文件属性的名字和类型代码以及记录属性所位于的 MFT 记录的文件引用。

authentication package（验证包）——运行在 Lsass 进程的环境中，并实现 Windows 2000 验证策略的动态链接库。验证动态链接库负责检查给定的用户名与密码是否匹配，如果匹配，返回至给出用户的安全标识的 Lsass 信息。Windows 2000 验证包包含有 Kerberos 和 MSV1-0。

automatic working set trimming（自动工作集修整）——当物理内存变少时，内存管理器用于增加系统中可用内存的技术。

bad-cluster file（坏簇文件）——记录磁盘中损坏位置（bad spot）的系统文件（文件名 $ BadClus）。

balance set manager（平衡集管理器）——每秒钟唤醒一次以检查和可能初始化多种调度和内存管理相关事件的系统线程。

basic disk（基本磁盘）——依赖于 MS-DOS 风格分区方案的磁盘。参见 dynamic disk。

bitmap file（位图文件）——被 NTFS 用于记录卷分配状态的系统文件（文件名 $ Bitmap）。位图文件的数据属性包含有位图，每一个数据位代表卷的一个簇，标志着每一簇是否已经分配。

boot code（引导代码）——系统引导时候执行的代码。

boot device driver（引导设备驱动程序）——用于引导系统的设备驱动程序。

boot file（引导文件）——存储有 Windows 2000 引导程序代码的系统文件（文件名为$Boot）。

boot partition（引导分区）——包含有内核操作系统文件的分区。启动时，系统识别引导分区 主引导记录（MBR）代码扫描主要的分区表直到它找到标志为可引导的分区。当 MBR 至少发现一个那样的标志时，它从被标记的分区里读入第一个扇区到内存，并把控制转交给分区里的代码。

bus driver（总线驱动程序）——服务于总线控制器、适配器、桥或其他有子设备的设备的驱动程序。总线驱动程序是必要的驱动程序，微软经常提供这些。系统上每一种类型的总线（如 PCI、PCMCIA 和 USB）都有一个总线驱动程序。

cache manager（高速缓存管理器）——Windows 2000 执行程序的一个组件，它为 NTFS 和其他文件系统驱动程序提供系统范围内的高速缓存服务，包括网络文件系统驱动程序（服务器和重定位器）。

Careful write（认真写）——一种创建文件系统 I/O 和缓存支持的技术。参见 write - through。

change journal（更改日志）——一种被 NTFS 文件系统记录信息允许应用有效的监视文件和目录改变的内部文件。更改日志 通常很大，几乎保证每一个应用程序都有机会处理这些改变而不丢失任何信息。

checked build（检查链接编译）——仅仅作为 MSDN Professional（或者 Universal）预定的一部分可用的 Windows2000Professional 的特殊的调试版本（在 Windows 2000 Server 上不可用）。检查链接编译可以通过在编译 Windows 2000 资源时把编译时刻的 DEBUG 标志设置为 TRUE 来创建。

checkpoint record（检查点记录）——当突然发生崩溃时候，帮助 NTFS 确定需要什么样的处理来恢复卷的记录。这个记录也包括重做和撤消的信息。

class driver（类驱动程序）——为特定类型的设备如磁盘、磁带或 CD-ROM 实现 I/O 处理的一种内核模式设备驱动程序。

clock algorithm（时钟算法）——在单处理机系统上实现的工作集页面替换策略，与最近使用策略相似（在大多数的 UNIX 版本上实现）。

clock interrupt handler（时钟中断处理程序）——更新系统时间，并递减跟踪现存线程运行时间的计数器的系统例程。

cluster factor（簇因子）——卷上的簇大小，当用户用 FORMAT 命令或 Disk Management Microsoft Console（MMC）插件格式化卷时创建。

cluster remapping（簇重映射）——NTFS 动态地从坏扇区的簇里获取好的数据，并分配新的簇，把数据拷贝到新的簇的过程。

clustering（聚类）——一种内存管理器用来解决页面错误的方法，该方法把明确访问过的页面的邻近页面都读至内存。

cluster（簇）——卷被划分的相同大小的分配单元。每一卷应该用 16 位来唯一标识。

collided page fault（页面冲突错误）——当另外的进程或线程导致正在调入的页面发生冲突时产生的错误。

commitment（提交）——内存管理器在统一的基础上用以跟踪专用提交内存使用情况的进程。

common model（通用模型）——通用信息模型（Common Information Model，CIM）中一组代表与系统管理领域相关但与具体实现无关的对象的类。这些类被认为是 CIM 内核模型的扩展。参见 core model。

complete information dump（完整信息转储）——在系统崩溃时，包括所有物理内存全部信息的内存转储。这种类型的转储要求页面文件的尺寸至少不小于物理内存。因为对大的内存系统要求非常大的页面文件，这种类型的转储文件最不普遍。Windows NT 仅仅支持这种类型的崩溃转储文件。

completion port（完成端口）——向线程发送 I/O 完成通知的机制。一旦一个文件与完成端口相联系，一个文件上的任何异步 I/O 操作一旦完成就会导致一个完成包添加到完成端口的队列中。通过简单地等待完成包（completion packet）添加到完成端口队列，线程可以等待任何未完成的 I/O 在多个文件上完成。通过完成端口，并发应用程序服务于客户机请求的活动线程数即可在系统的帮助下得到控制。

configuration manager（配置管理器）——负责实现和管理系统注册表的执行程序的主要组件。

container object（容器对象）——可以容纳其他对象、包括其他容器对象的名字空间对象。容器对象的例子有文件系统名字空间的目录和注册表名字空间的键。

context switch（环境切换）——保存正在运行的线程的易失机器状态或者加载另一线程的易失状态以及启动新线程执行的过程。

control object（控制对象）——为控制多种操作系统功能而建立语义的一系列内核对象。这包括内核进程对象，异步过程调用（APC）对象，延迟过程调用（DPC）对象以及很多 I/O 系统使用的对象，例如中断对象。

core model（内核模型）——在通用信息模型（CIM）中作为 WBEM 标准的一部分的一系列类。这些类是 CIM 的基本语言并代表着可以应用于管理所有领域的对象。参见 commom model。

deferred procedure call（DPC，延迟过程调用）——在中断服务例程（ISR）执行后完成大部分涉及到处理设备中断任务的例程。延迟过程调用在低于 ISR 级别的中断请求级别（IRQL）执行，这样避免不必要地阻塞其他中断。DPC 例程初始化 I/O 完成并启动设备上下一加入队列的 I/O 操作。

deferred procedure call object（延迟过程调用对象）——描述延迟中断处理到 DPC/调度级别的请求的内核控制对象。（参见中断请求级别（IRQR））。该对象对用户模式程序是不可见的，但对设备驱动程序和其他系统代码是可见的。DPC 对象包含的最重要信息是当内核处理 DPC 中断时候需要调用的系统函数地址。

demand - paging policy（请求页面策略）——当页面错误发生时，请求向物理内存加载一个页面的策略。在请求页面系统中，当进程的线程在第一次开始执行时，因为线程得到继续时，

需要访问初始的页面组，进程会导致很多页面错误。一旦这些页面组加载到内存，进程会减少调页的活动。

device driver（设备驱动程序） 作为 I/O 系统和相关硬件接口的可加载内核模块（典型的以 .sys 结束）。Windows 2000 上的设备驱动程序并不直接处理硬件设备。相反它调用硬件应用层（HAL）与硬件接口。

device ID（设备 ID） 向即插即用管理器报告的设备标识符。该标识符是总线专有的。对 USB 总线来说，标识符由制造硬件的产商 ID（VID）和产商赋给产品的产品 ID（PID）组成。

device instance ID（设备实例 ID、DID）——由设备 ID 和即插即用管理器用于在注册表中枚举目录中定位设备的键的实例 ID 组成的标识符（HKLM \ SYSTEM \ CurrentControlSet \ Enum）。

device object（设备对象）——在系统中代表物理、逻辑、虚拟设备并用来描述特征的数据结构，例如在缓冲区中需要的定位以及设备队列中用来保存输入的 I/O 请求包的位置。

device tree（设备树） 一即插即用管理器创建用于代表设备之间关系的内部树。树中的结点称之为设备结点。参见 devnode。

devnode（设备结点）——设备树中的结点。设备结点包含表明设备的设备对象相关信息以及其他一些即插即用管理器保存的即插即用相关信息。参见 device tree。

dirty page threshold（脏页面阈值） 一在唤醒缓存管理器延迟写（Lazy writer）系统线程把页面写回磁盘之前系统缓存保留在内存中的页面数。该值在系统初始化时计算并依赖于物理内存的大小和注册表值 HKLM \ SYSTEM \ CurrentControlSet \ Control \ Session Manager \ MemoryManagement \ LargeSystemCache 的值。

disk group（磁盘组）——共享公用数据库的动态磁盘。为 Windows 2000 设计的 VERITAS 的商用卷管理软件包含有磁盘组，但 Windows 2000 的 Logical Disk Manager（LDM）实现仅仅包含一个磁盘组。

dispatch code（调度代码）——当初始化时存储在中断对象内的汇编语言代码指令。当中断发生时，执行这些指令。

dispatch routine（调度例程）——设备驱动程序提供的主要功能。调度例程的主要例子有 open、close、read 和 write 以及其他一些设备和文件系统或网络支持的功能。当被调用来完成 I/O操作时，I/O 管理器产生一个 IRP 并通过驱动程序的调度例程来调用驱动程序。

dispatcher（调度程序）— 内核中实现 Windows 2000 调度的一系列例程。Windows 2000 没有一个单独的调度程序模块或例程——这些代码分散在内核中，在内核中与调度相关的事件发生。

dispatcher database（调度程序数据库） ——内核维持的实行线程调度决策的一系列数据结构。调度数据库跟踪等待执行的线程以及哪个处理器在执行哪个线程。参见 dispatcher ready queue。

dispatcher header（调度程序头）— 包含有对象类型、发送信号状态以及在该对象上等待的线程列表的数据结构。

dispatcher objects（调度程序对象） 一结合了同步能力能改变或影响线程调度的一系列内

核对象。调度对象包含有内核进程、互斥（内部称为 mutant）、事件、内核事件对、信号量、计时器以及能等待的计时器（waitable timer）。

dispatcher ready queue（调度程序就绪队列）——调度数据库中最重要的结构（位于 KiDispatcherReady List Head）。调度就绪队列实际上是一系列队列，一个队列用于一个调度优先权。这些队列包含有处于就绪状态的线程，这些线程等待被调度执行。

display driver（显示驱动程序）——把设备无关的图形请求翻译为设备指定请求的驱动程序。设备指的请求接着与内核模式视频小端口驱动程序相匹配完成视频显示支持。一个显示驱动程序负责实现绘制操作，要么通过直接写入帧缓冲区要么通过与控制器中的图形加速芯片通信。

driver object（驱动程序对象）——代表系统中个体驱动程序的数据结构为 I/O 管理器记录每个调度例程的地址（入口点）。

driver support routine（驱动程序支持例程）——设备驱动程序调用以完成 I/O 请求的例程。

dynamic disk（动态磁盘）——支持多分区卷的磁盘，与基本磁盘相比提供更为灵活的分区方案。参见 basic disk。

dynamic - link library（动态链接库，DLL）——一系列作为二进制映像连接起来的可调用子程序。可以被使用它们的应用程序动态加载。

environment subsystems（环境子系统）——向用户应用程序显示本机操作系统服务的用户进程。这样提供了操作系统环境或 personality。Windows 2000 有三个环境子系统：Win32、Posix 和 OS/2 1.2。

event（事件）——可以用于同步的具有持久状态的对象（可以发送信号或者不发送信号）；可以触发行为的系统发生。

exception（异常）——执行特定指令产生的同步情况。在同样的条件下用同样的数据运行同样的程序可以复制异常。

exception dispatcher（异常调度程序）——服务于除了那些足够简单的能被陷阱处理程序（trap handler）解决的异常之外的所有异常的内核模块。异常调度器的任务是寻找能够处理异常的异常处理程序。

executive（执行程序）—— Ntoskrnl.exe 的上层。（内核是下层）。执行程序包括基本的操作系统服务，如进程和线程管理器、虚拟内存管理器、内存管理器、安全访问监视器、I/O 系统和高速缓存管理器。参见 kernel。

executive object（执行程序对象）——由各种执行程序的组件实现的对象（如进程管理器、内存管理器、I/O 子系统等等）。执行程序对象和对象服务是环境子系统用于创建自己版本的对象和资源的原语。因为执行程序对象典型地由环境子系统代表用户应用创建或者作为规范操作的部分由操作系统的多个组件创建，很多执行程序对象包括（封装）一个或多个内核对象。参见 kernel object。

executive resource（执行程序资源）——提供专有访问（如互斥）以及共享读访问（多个读共享对结构进行的 read - only 访问）。因为执行程序资源仅仅对内核模式代码可用，它们不能通过 Win32 API 访问。

extended partition（扩展分区）——包含带有自己的分区表的主引导记录（MBR）的特殊分区类型。通过使用扩展分区，微软的操作系统克服了每个硬盘只能有四个分区的明显限制。一般来说，扩展分区允许无定限的递归。这就意味着一个硬盘的分区数量不存在明显的上限。参见 partition。

fast I/O（快速 I/O）——直接读和写高速缓冲的文件而不产生 I/O 请求包（IRP）的方法。

fast LPC（快速 LPC）——用于在线程之间发送消息的特殊进程内通信功能。

file mapping object（文件映射对象）——用于实现共享内存的内存管理器中 Win32 API 的底层基本要素（在内部称为 section object）。参见 section object。

file reference（文件引用）——在 NTFS 卷中标识文件的 64 位值。文件引用由文件号和顺序号组成。文件号相应于文件的文件记录在文件表中的位置减一（如果文件有不止一个文件记录，则相当于基本文件记录的位置减一）。

file system driver（文件系统驱动程序，FSD）——接收到文件的 I/O 请求并通过发送自己的更为显式的请求到物理设备驱动程序来满足请求的一种内核模式的设备驱动程序。参见本机文件系统驱动程序（FSD），网络文件系统驱动程序（FSD）。

file system filter driver（文件系统过滤器驱动程序）——截获 I/O 请求完成附加操作，并把它们向更低层驱动程序传送的一种内核模式设备驱动程序。

file system format（文件系统格式）——定义在存储介质上文件数据的存储方式以及对文件系统特性的影响。例如，用户的访问权限没有与文件和目录相联系的格式是不安全的。文件系统格式也可以设置文件大小的限制以及文件系统支持的存储设备。

filter device object（过滤器设备对象，FiDO）——作为设备结点一部分的设备对象。一个或更多的可选的过滤器设备对象位于物理设备对象（PDO）和功能设备对象（FDO）之间，或者位于功能设备对象之上。参见 devnode，function device object（FDO），physical device object（PDO）。

filter driver（过滤器驱动程序）——见 file system filter driver。

foreground application（前台应用程序）——拥有聚焦窗口线程的进程。

free bulid（自由链接编译）——可以作为零售商品购买的 Windows 2000 系统，在编译和链接时支持完全的编译器优化，并有从映像中获取的内部符号表信息。

function driver（功能驱动程序）——为设备提供操作接口的主要设备驱动程序。除非原始地使用设备（一种 I/O 通过总线驱动程序和总线过滤器驱动程序完成的实现如 SCSI PassThru）否则需要驱动程序。功能驱动程序是对特定的设备了解得最多的驱动程序，通常是访问设备指定寄存器的仅有驱动程序。

functional device object（FDO，功能设备对象）——功能设备对象是设备结点必要的一部分。即插即用管理器加载用来管理检测到的设备的功能驱动程序创建功能设备对象。一个功能设备对象代表设备的逻辑接口。参见 devnode，filter device object（FiDO），physical device object（PDO）。

graphical identification and authentication（图形标识和验证，GINA）——在 Winlogon 进程中运行的用户模式动态链接库，Winlogon 用来获取用户的名字和密码以及智能卡 PIN。标准的 GINA

是 \ Winnt \ System32 \ Msgina.dll。

handle（句柄）——对象的标识符。当进程使用名字创建或打开一个对象时，进程为该对象获得句柄。通过句柄访问对象比使用名字要快，因为对象管理器忽略了名字查找而直接查找对象。

handle table（句柄表）——包含进程打开句柄的所有对象指针的表。句柄表按照三级方案实现，与 x86 内存管理单元实现虚拟到物理地址转换的方法相似。

hardware abstraction layer（硬件抽象层，HAL）——提供了 Windows 2000 运行的硬件平台的低层接口的可加载内核模式模块（Hal.dll）。HAL 隐藏了硬件相关的细节如 I/O 接口，中断控制器和多处理器通信机制——任何体系结构专有的和机器相关的功能。

hash（散列）——使用密码算法从一块数据（例如，一个文件）产生的统计学意义上的唯一值。因为不同的数据产生不同的散列，散列可以用来检测数据的毁坏和窜改。Windows 2000 驱动程序信号功能使用散列。

heap（堆栈）——可以通过堆栈管理器提供的一系列函数来划分和以小块分配的一或多个页面区域。

heap manager（堆栈管理器）——可以分配和回收可变大小内存的一系列函数（并不基于页面粒度）。堆栈管理器函数存在两个地方：Ntdll.dll 和 Ntoskrnl.exe。子系统 API（如 Win32 堆栈 API）使用 Ntdll.dll 中的拷贝，各种执行程序的组件和设备驱动程序使用 Ntoskrnl 中的拷贝。

hive——存储在磁盘中包含有注册表信息的一些文件中的一个。每个 hive 包含有一个注册表树，它有一个作为树的起始点或根的键。

hyperspace（超级空间）——用来映射进程工作集列表或为一些操作临时映射其他物理页面的特殊区域。这些操作如：进行自由列表上页面清零操作（当置零列表是空的并且需要置零页面的时候），使其他页面列表中的页面表项无效（如当页面从备用列表中删除移除时），在进程创建时建立新进程地址空间。

I/O completion routine（I/O 完成例程）——当低级别驱动程序完成 I/O 请求包（IRP）处理时通过由分层驱动程序通知驱动程序来实现的例程。例如 I/O 管理器在设备驱动程序完成文件数据的转移时调用文件系统驱动程序的 I/O 完成例程。完成例程通知文件系统操作成功、失败或撤消以及允许文件系统驱动程序完成清除操作。

I/O request packet（I/O 请求包，IRP）——控制每个阶段 I/O 操作如何处理的数据结构。大多数 I/O 请求由 IRP 表示，它从一个 I/O 系统的组件转移到另一个组件。

I/O subsystem API（I/O 子系统 API）——子系统动态链接库调用以实现子系统文档化的 I/O 功能的内部执行程序系统服务（如 NtReadFile 和 NtWriteFile）。

I/O system（I/O 系统）——接收 I/O 请求（从用户模式和内核模式调用程序）并以不同形式发送它们到 I/O 设备的 Windows 2000 执行程序组件。

ideal processor（理想处理器）——特定线程应该优先考虑运行的处理器

idle summary（空闲汇总）——每一个设置位代表着一个空闲处理器的位屏蔽（KiIdleSummary）。

impersonation（模拟）——除了进程的令牌外允许线程有不同访问令牌的能力。

initialization routine（初始化例程）——I/O 管理器在加载驱动程序到操作系统上时执行的驱动程序例程。初始化例程创建 I/O 管理器用于识别和访问驱动程序的系统对象。

in - paging I/O（页面调入 I/O）——当为了解决页面错误而必须对文件发出读操作而出现的情况。页面调入 I/O 操作是同步的—线程等待事件直到 I/O 完成，并且不能被异步过程调用（APC）中断。

instancing（实例化）——表示为名字空间的相同部分做单独拷贝的术语。实例化 \ DosDevices 使得每个用户具有不同驱动符和 Win32 对象如串行端口成为可能。

intelligent file read - ahead（智能文件预读）——一种基于调用线程现在正在读的数据而预测下一步可能读的数据的技术。

inter - process interrupt（处理器间中断，IPI）——内核发送请求其他处理器完成某种操作的中断，如调度执行特定线程或更新它的转换后备缓冲区（translation look - aside buffer）高速缓存。

interrupt（中断）——与处理机正在执行的内容无关的异步事件（可以在任何时候发生）。中断主要由 I/O 设备、处理器时钟或者定时器产生，它们可以被允许或禁止。

interrupt dispatch table（中断调度表，IDT）——Windows 2000 用于定位处理特定中断的例程的数据结构。中断源的中断请求级（IRQL）作为表的索引，表项指向中断处理例程。

interrupt dispatcher（中断调度器）——响应中断的内核陷阱处理程序的子模块。

interrupt object（中断对象）——允许设备驱动程序为它们的设备注册中断服务例程的内核控制对象。一个中断对象包括内核需要把设备 ISR 与特定的中断级联系起来的所有信息，包括 ISR 的地址、设备中断的中断请求级以及通过它把 ISR 联系起来的内核调度表的入口。

interrupt request（中断请求，IRQ）——标识中断的值。在 x86 系统上，外部 I/O 中断进入到中断控制器的某条线上。控制器反过来在一条线上中断处理器。一旦处理器被中断，它查询控制器获取中断请求（IRQ）。中断控制器把 IRQ 翻译为中断号，这个号可以作为进入中断调度表的索引，并把控制转向合适的中断调度例程。在系统启动时，Windows 2000 用指向处理中断和异常的内核例程的指针来填充中断调度表。

interrupt request level（中断请求级，IRQL）——Windows 2000 设置的中断优先级方案。内核在内部用从 0 到 31 的数字代表 IRQL，数字越高代表中断级别越高。虽然内核为软件中断定义了 IRQL 的标准集，但硬件抽象层把硬件中断号映射为 IRQL。

interrupt service routine（中断服务例程，ISR）——当设备发送中断时，内核调度器把控制转向此处的设备驱动程序例程。在 Windows 2000I/O 模型中，ISRs 运行在高的设备中断请求级别上，这样它们尽可能少的工作以避免不必要的阻塞低级别的中断。ISR 把运行在低的级别上的延迟的过程调用排成队列，并执行中断处理的剩余部分。只有中断驱动的驱动程序有 ISR。例如文件系统没有中断服务例程。

job object（作业对象）——Windows 2000 中的可命名、安全而可共享的控制与作业相关的进程属性的对象。作业对象的基本功能是允许进程组可以作为单元管理和处理。作业对象也记录了与作业相关的所有进程以及与作业相关但以前终止的进程的基本日志信息。

journaling（日志）——起初为事务处理开发的日志技术，而像 NTFS 这样的可以恢复文件

系统使用它保证卷的一致性。

kernel（内核）—— Ntoskrnl.exe 的最底层。内核作为执行程序的组件，确定操作如何使用处理器以及确保它们被谨慎地使用。内核提供了线程调度、陷阱处理和异常调度、中断处理和调度以及多处理器同步。参见 executive。

kernel debugger（内核调试程序）——用于调试驱动程序、查错挂起系统并检查崩溃系统的转储的工具。因为能够显示标准工具不可见的 Windows 2000 内部系统信息，它也是研究 Windows 2000 内部的有用工具。（与本书一块发行的 CD 上的 LiveKD 允许在正在运行的系统上使用标准内核调试程序）。

kernel handle table（内核句柄表）——该表中的句柄只能在进程环境中从内核模式访问（在内部以 ObpKernelHandleTable 名访问）。内核模式函数能在任何进程环境中引用该表的句柄而无任何性能影响。

kernel memory dump（内核内存转储）——在系统崩溃时，在物理内存中仅包含内核模式读/写页面的内存转储类型（Windows 2000 服务器系统的缺省模式）。内核内存转储不包含属于用户进程的页面。因为只有内核模式代码能直接导致 Windows 2000 崩溃，用户模式页面在调试崩溃时是没有必要的。没有方法可以预测内核内存转储的大小，因为该大小仅依赖于操作系统和机器上的驱动程序分配的内核模式内存。

kernel mode（内核模式）—— 在处理器上执行的代码的特权模式，在这种情况下可以访问所有内存和使用所有 CPU 指令。操作系统代码只能在内核模式下运行（如系统服务和设备驱动程序）。参见 user mode。

kernel object（内核对象）—— Windows 2000 内核实现的原语对象集。这些对象对用户模式代码是不可见的，但仅能在执行程序中创建和使用。内核对象提供了基本的能力，如同步，执行程序对象建立在它的基础上。参见 executive object。

kernel streaming filter driver（内核流过滤器驱动程序）——链在一块用于对数据流进行信号处理的内核模式驱动程序，如记录和显示音频和视频。

kernel - mode device driver（内核模式设备驱动程序）——唯一能够直接控制和访问硬件设备的驱动程序。

kernel - mode graphics driver（内核模式图形驱动程序）——把设备无关的图形（GDI）请求翻译为设备指定请求的 Win32 子系统显示或者打印设备驱动程序。

key（键）——用于引用注册表中数据的机制。虽然键出现在对象管理名字空间，注册表以一种类似于管理文件对象的方式管理它们。零个或更多的键值与键对象关联；键值包含键的数据。

key control block（键控制块）——存储注册表键的完整路径名的结构，包括控制块引用的键结点的单元索引，以及包含当键的最后句柄关闭时，用以显示配置管理器是否需要删除控制块引用的键单元的标志（flag）。在 Windows 2000 里，所有的键控制块是以字母排列的二分树，这样可以通过名字对现存的键控制块进行快速搜索。键对象指向相应的键控制块，这样如果两个应用程序打开相同的注册表键，每一个应用程序都接收到一个键对象，并且每一个键对象指向相同的键控制块。

key object（键对象）——配置管理器定义的集成注册表的名字空间和内核的一般名字空间的一种对象类型。

last known good control set（所知最近的正确控制集）——当用户成功登录时在注册表键 HKLM \ SYSTEM \ CurrentControlSet 下制作的一个关键的启动时信息的拷贝。在注册表的配置改变导致系统不能成功启动时，所知最近的正确控制集可以在启动时选择。

lazy IRQL（延迟 IRQL）——避免访问可编程的中断控制器（PIC）的性能优化。当中断请求级别升高时，硬件抽象层在内部记录下新的中断请求级别而不是改变中断屏蔽。如果较低的优先级别中断随后发生，硬件抽象层会把第一个中断的中断屏蔽设置为适当的配置并延迟较低的优先级别直到中断请求级别降低。因此，如果当中断请求级别升高时没有低级别的中断发生，硬件抽象层不需要修改 PIC。

lazy writer（延迟书写器）——当有后台程序活动时（异步磁盘写），一系列调用内存管理器来刷新磁盘高速缓存内容的系统线程。高速缓存管理器使用延迟写来优化磁盘 I/O。

legacy driver（遗留驱动程序）——为微软 Windows NT 编写的设备驱动程序，但在 Windows 2000 不加修改就可运行。它们不同于其他一些 Windows 2000 驱动程序，不支持电源管理，在 Windows 2000 即插即用管理器上不能工作。

local file system driver（本机文件系统驱动程序，FSD）——管理直接与计算机相连的卷的驱动程序。参见文件系统驱动程序。

local procedure call（本机过程调用，LPC）——用于高速消息传递的进程间通信机制（不能通过 Win32 API 获得，而只能通过仅由 Windows 2000 操作系统组件使用的内部机制）。本机过程调用典型地用于服务进程以及那个服务器进程的一个或更多的客户进程之间的通信。本机过程调用可以在两个用户模式进程之间或者在一个内核模式组件及一个用户模式进程之间建立。

local security authority（LSA）server（本机安全权限服务器）——运行在映像 \ Winnt \ System32 \ Lsass.exe 下的用户模式进程，它负责本机系统安全策略（如哪些用户允许登录机器、密码策略、授予用户和用户组的权限以及系统安全审计设置）、用户验证和发送安全审计消息到事件日志。Lsass 加载的库，也就是 LSA 服务（Lsasrv - \ Winnt \ System32 \ Lsasrv.dll），实现了这些功能中的大多数。

local security authority（LSA）server policy database（本机安全权限（LSA）服务器策略数据库）——包含有系统安全策略设置的数据库。该数据库包含有什么域名可以信任用于认证登录企图，谁拥有访问系统权限以及如何完成（交互、网络和服务登录），谁被赋予了什么样的权利，需要完成什么样的安全审计等等诸如此类的信息。

log file（日志文件）——NTFS 用来记录影响 NTFS 卷结构的所有操作的元数据文件（文件名$ LogFile），包括文件创建或者象 copy 这样影响目录结构的任何命令。日志文件用于在系统崩溃后恢复 NTFS 卷。

log hive（日志 hive）——配置管理器使用的注册表 hive，用于确保非瞬间注册表 hive 始终处于可恢复状态。每一个非瞬间 hive 有一个相应的日志 hive，它是一个有着与 hive 相同的文件名和 .log 扩展名的隐藏文件。参见 hive。

logging（日志）——在系统崩溃和其他错误发生时，NTFS 用于维护文件系统完整性的事务

处理技术。在 NTFS 日志中，任何可以改变重要文件系统数据结构的事务子操作在它们进行磁盘操作之前都记录在日志文件中，这样如果系统崩溃，部分完成的事务在系统恢复时可以重做或者撤消。

logical cluster number（逻辑簇号，LCN）——NTFS 用于定位磁盘物理位置从卷起始到结束的簇号。当磁盘驱动程序需要把逻辑簇号转换为实际的磁盘位置时，NTFS 用簇因子乘以逻辑簇号以获得卷的实际字节偏移。

logical sequence number（逻辑序列号，LSN）——NTFS 用于标识写到日志文件的记录的数。

logon process（登录进程）——运行 Winlogon.exe 的用户模式进程，负责截取用户名和密码并把它们发送到本机安全权限服务器用于认证，并在用户会话中创建初始进程。

look - aside list（备份表）——包含固定大小块的快速内存分配机制。备份表或者可分页或者不可分页，这样它们可以从可分页交换区和不可分页交换区中分配。

mapped file I/O（映射文件 I/O）——浏览驻留在作为进程虚拟内存一部分的磁盘上的文件的能力。一个程序可以把文件当作大的数组访问而不缓冲数据或者执行磁盘 I/O。程序访问内存，而内存管理器使用它的分页机制从磁盘文件加载正确页面。如果应用向它的虚拟地址空间写，内存管理器把这些改变作为正常的调页写回文件。

mask（屏蔽）——在中断执行之前中断通过该进程等待正在执行的线程降低 IRQL。从 IRQL 高于现有级别的中断源产生的中断中断处理器，而 IRQL 等于或低于现有级别的中断源产生的中断将被屏蔽掉直到正在执行的线程降低 IRQL。

master file table（主文件表，MFT）——NTFS 卷结构的核心。MFT 实现为文件记录数组。每一个文件记录的大小固定为 1KB，而不考虑簇大小。

memory manager（内存管理器）——实现了请求页面虚拟内存的 Windows 2000 执行程序组件，并看起来好像让每一个进程拥有了 4GB 32 位地址空间（而把此地址空间的子集映射为物理内存）。

metadata（元数据）——描述磁盘上的文件的数据。也称为 volume structure data。

metadata file（元数据文件）——在每一个 NTFS 卷中，包含了用于实现文件系统结构的信息的一些文件。

MFT mirror（MFT 镜像）——位于包含有一些主文件表前几行的拷贝的硬盘中间的 NTFS 元数据文件（文件名$ MFTMirr）。

miniport driver（小端口驱动程序）——把一般 I/O 请求映射到适配器类型端口的内核模式设备驱动程序类型，如特定的 SCSI 适配器。

mirrored set（镜像组）——一种把磁盘的分区内容复制到另一个磁盘上的等大小分区的技术。

modified page writer（修改页面的书写器）——当修改页面表太大时，通过把页面写回后端的存储位置的方式负责限制修改页面表的大小的虚拟内存管理器中的线程。修改页面的书写器包含有两个系统线程。一个把修改页写回调度文件（MiModifiedPageWriter），第二个把修改页写回映射的文件（MiMappedPageWritter）。

mount（装配）——当第一次访问卷时 NTFS 使用的技术。在这种情况下，装配意味着为卷

的使用做准备。为了装配卷，NTFS 查找引导文件寻找主文件表的物理磁盘地址。

mount point（装配点）——通过 NTFS 卷上的目录允许卷联接的机制，这使得卷不需要分配驱动符即可以访问。NTFS 上的再分析点（reparse point）使得装配点变得可能。

MSDN —— Microsoft Developer Network，微软对开发者的支持程序。MSDN 提供三个 CD-ROM 预订程序。MSDN Library、Professional 以及 Universal。更多的信息参见 msdn.microsoft.com。

multipartition volume（多分区卷）——代表着来自多个分区的扇区并且文件系统驱动程序作为单一单元管理的对象。多分区卷提供了单一卷不能提供的性能、可靠性以及大小特性。

mutant ——互斥（mutex）的内部名。

mutex（互斥）——用于对资源的访问进行序列化的同步机制。

name retention（名字保留）——对象管理器所实现的对象保留的第一阶段。名字保留由存在的对象打开的句柄数所控制，每次一个进程打开一个对象的句柄，对象管理器就在对象的头递增打开的句柄记数器。当进程完成对象的使用并且关闭句柄时，对象管理器递减打开的句柄计数器。当计算器减至为零时，对象管理器从整体名字空间中删除对象名。这种删除防止了新的进程打开该对象的句柄。

native application（本机应用程序）——仅仅使用 Ntdll 提供的系统服务并且不是 Win32 子系统的客户机的应用程序。Smss（会话管理器）是一个本机应用程序的例子。

network file system driver（网络文件系统驱动程序，FSD）——允许用户访问连接到远程计算机的数据卷的驱动程序。参见 file system driver。

network logon service（网络登录服务）——位于 Service.exe 进程中响应网络登录请求的用户模式服务。认证按照本机登录处理，通过发送它们到 Lsass 进程以便认证。

network redirectors and server（网络重定向器和服务器）——分别传送远程 I/O 请求到网络上的机器并接收那样的请求的文件系统驱动程序。

nonpaged pool（非页交换区）——包括一定范围的系统虚拟地址的内存交换区，可以保证在任何时候驻留在物理内存中，这样可以导致在没有 I/O 调页的情况下从任何地址空间访问。非页交换区可以在系统初始化时创建，并且被内核模式组件用来分配系统内存。

Ntdll.dll ——一个特别的系统支持库，主要用于使用子系统 DLL。子系统 DLL 包含有到 Windows 2000 执行程序系统服务的调度存根和被子系统、子系统动态链接库以及其他的本机映像使用的内部支持函数。

Ntkrnlmp.exe ——多处理器系统的执行程序和内核。

Ntoskrnl.exe ——单处理器系统的执行程序和内核。

object（对象）——在 Windows 2000 执行程序中，一个静态定义的对象类型的单一运行时事例。

object attribute（对象属性）——对象中的数据字段，部分定义对象的状态。

object directory（对象目录）——其他对象的容器对象。对象目录用于实现分层名字空间，而其他对象存储在它里面。

object handle（对象句柄）——通过执行进程（ERPOCESS）块指向进程指定句柄表的索引。

object manager（对象管理器）——负责创建、删除、保护并跟踪对象的 Windows 2000 执行

程序组件。对象管理器集中了否则将分散在整个操作系统的资源控制操作。

object method（对象方法）——处理对象的方法，通常读或改变对象的属性。例如，一个进程的 open 方法会接收一个进程的标识符作为输入，并返回指向对象的指针为输出。

object reuse protection（对象复用保护）——防止用户浏览其他用户已经删除的数据或者访问别的用户先前使用而后释放的内存的方法。对象复用保护通过在分配给用户之前初始化所有的对象包括文件和内存的方式来防止潜在的安全漏洞。

object type（对象类型）——系统定义的数据类型，包括操作数据类型实例以及一系列对象属性的服务，有时候称为“对象类”。

Page directory（页面目录）——内存管理器创建用来为进程映射所有页面表位置的页面。每一个进程有一个单独的页面目录。

page directory entries（页面目录项，PDE）——页面目录由页面目录项组成，每一个项有四字节长并为该进程描述了所有可能页面表的状态和位置。

page fault（页面错误）——对无效页面的引用。内核陷阱处理程序调度这种错误给内存管理器的错误处理器去处理。

page file backed section（支持页面文件的区域）——映射到指定内存的区域对象。

page file quota（页面文件配额）——对一个进程可以消耗的指定页面的数目的限制——并不一定就是页面文件空间。

page frame database（页面帧数据库）——描述物理内存中每一个页面状态的数据库。页面处于八种状态之一：活动的（也称为有效的）、过渡的、备份的、修改的、modified - no - write、自由的、清零的以及坏的。

page frame number（PFN）database（页面帧号数据库）——描述物理内存中每一个页面中的状态。

page table（页面表）——操作系统创建用来描述进程地址空间中虚拟页面位置的一页映射信息（由页面表项的数组构成），由于 Windows 2000 为每一个进程提供私用地址空间，因为对每一个进程来说映射都不同，所以每一个进程都有自己的进程页面表来映射私用地址空间。描述系统空间的页面表在所有的进程之间共享。

page table entry（页面表项，PTE）——进程页面表中包含有虚拟地址映射的地址的项。页面可能在物理内存中也可能在磁盘上。

paged pool（页交换区）——系统空间中可以调入或调出系统进程工作集的虚拟内存区域。页交换区在系统初始化时创建，并被内核模式组件用来分配系统内存。单处理器系统有两个页交换区，多处理器系统有四个。具有不止一个页交换区可以减少系统代码块同时调用交换区例程导致阻塞的频率。

paging（调页）——把内存的内容写入硬盘以及释放物理内存这样它能用于其他进程或者操作系统本身的过程。因为大多数系统有比正在运行的进程使用的所有虚拟内存少得多的物理内存（每一个进程有 2GB 或 3GB），内存管理器转移或把一些内容调入到磁盘上。

partition（分区）——微软操作系统硬盘不连续区域。文件系统（如 FAT 和 NTFS）把每一个分区格式化为一个卷。一个硬盘可以包含最多四个主分区。参见 extended partition。

partition table（分区表）——包含定义了硬盘上四个主分区的位置的四个项目的主引导记录（MBR）的一部分。分区表也记录了分区的类型。有很多预定义的分区表类型，并且一个分区的类型指定了分区包含哪种文件系统。

physical address extension（物理地址扩展，PAE） 包含于自 Pentium Pro 以来所有 Intel x86 处理器的内存映射模式。对于适当的芯片组，PAE 模式允许访问到 64GB 物理内存。当 x86 以 PAE 模式执行时，内存管理单元（MMU）把虚拟内存分割为四个区域。

physical device object（物理设备对象、PDA）——设备结点必要的一部分。物理设备对象代表一个设备的物理接口。参见 devnode。

plug and play（PnP）manager（即插即用管理器）——执行程序用来决定哪一个驱动程序需要用来支持特定的设备并加载那些驱动程序的主要组件。PnP 管理器在枚举时获取每一个设备的硬件资源需求。基于每一个设备的资源需求，PnP 管理器分配适当的硬件资源如 I/O 端口、IRQ、DMA 频道以及内存位置。在系统上的设备改变（设备添加或删除）时它也负责发送适当的事件通知。

port driver（端口驱动程序）——一种实现了处理一种 I/O 端口类型——如 SCSI——的 I/O 请求的内核模式设备驱动程序。

port object（端口对象）——本机过程调用（LPC）输出用来维护通信需要的状态的单个执行程序对象。

power manager（电源管理器）——执行程序协调能源事件以及向设备驱动程序产生电源管理 I/O 通知的主要组件。当系统空闲时，电源管理器通过把 CPU 设置为空闲这样的配置来减少能源消耗。单个设备的电源消耗的改变通过设备驱动程序来处理，但由电源管理器来协调。

printer driver（打印机驱动程序） 驱动程序把设备无关的图形请求翻译为打印机指定的命令。这些命令接着传送到内核模式端口驱动程序，如并行端口驱动程序（Parport.sys），或者 USB 打印机端口驱动程序（Usbprint.sys）。

private cache map（专用高速缓存映射）——包含最后两次读位置的结构，这样高速缓存管理器可以进行智能预读。

Process（进程）——执行一系列线程对象所必须的虚拟地址空间和控制信息。

process ID（进程 ID）——唯一的进程标识符（内部称为“客户机 ID”）。

process working set（进程工作集）——进程虚拟地址空间的子集，它被运行的进程常驻和拥有。参见 system working set。

processor affinity（处理器相似性）——线程允许运行的处理器集。

processor control register（处理器控制寄存器，PCR）——包含有系统中每一个处理器的信息的数据结构以及它的扩展处理器控制块（processor control block（PRCB）），处理器的信息如现在的中断请求级别（IRQL）、指向硬件中断调度表（IDT）的指针、现在运行的线程、下一个选择运行的线程。内核模式以及硬件抽象层使用这些信息来完成体系结构指定和机器指定的动作。PCR 的部分和 PRCB 结构在 Windows 2000 设备驱动程序工具包（DDK）头文件 Ntddk.h 中公开定义。

protected-mode（保护模式）——引导过程中的一种状态，这时没有虚拟内存向物理内存

的映射，但可以访问完全的 32 位内存。在系统处于保护状态时，Ntldr 可以访问所有的物理内存。

protocol driver（协议驱动程序） — 实现了网络协议如 TCP/IP、NetBEUI 或 IPX/SPX 的驱动程序。

protocol page table entries（协议页面表项，prototype PTE）—— 当页面可以在两个进程中共享时，内存管理器赖以映射潜在的共享页面的软件结构。当一个区域对象第一次创建时，一个协议 PTE 数组也同时创建。

quality of servce（服务质量，Qos）——确保关键的网络应用程序获得最高优先级别的网络技术。

quatum（时间片）——在 Windows 2000 中断某个线程而寻找是否另外一个处于同一优先级的线程等待运行，或者该优先级别需要降低时，线程允许运行的时间的长度。

quatum unit（时间片单元）——一个代表着线程能够运行直到它的时间片过期的时间的值。该值是一个整数，而不是一段时间。

queue（队列）——线程把 I/O 完成的通知加入队列和退出队列的方法（在 Win32API 中称为 I/O 完成端口。

queued spinlock（队列自旋锁）——一种仅能被内核使用而不能为执行程序组件或设备驱动程序输出的特殊类型自旋锁。队列自旋锁在多处理器上比标准自旋锁可伸缩性更好。参见 spinlock。

quota charges（配额管理）——在 Windows 2000 对象管理器中，当进程中的线程打开对象中的句柄时，对象管理器从进程分配的分页交换区和/或非分页交换区配额减去的对象数的记录。

RAID－5 volume（RAID－5 卷）——常规带区卷的容错变种。容错通过为每一个带区存储奇偶检验来维护磁盘的等价体来实现。也称为“stripe volume withparity”。

ready summary（就绪汇总）——Windows 2000 维护用来加速选择运行或抢占的线程的 32 位掩码（KiReadySummary）。

real mode（实模式）——一种虚拟地址到物理地址的转换不会发生的操作模式，这就意味着使用内存地址的程序把它们解释为物理地址并且只有计算机的物理内存的第一个 1MB 是可以访问的。简单的 MS－DOS 程序在实模式的环境下执行。

recoverability（可恢复性）——NTFS 允许系统从未预料到的系统停机状态恢复的高级特性。如果系统以不可预料的方式停机，FAT 卷的元数据会处于不一致的状态，导致大量的目录和文件数据破坏。NTFS 通过以事务方式对元数据的改变进行日志记录来实现可恢复性，这样文件系统结构可以修补为一致的状态，而不会丢失文件和目录结构信息。（然而，文件数据会丢失）。

redo information（重做信息）——包含在 NTFS 检查点记录中的信息，解释了如果系统在事务信息从缓冲区中清除之前失败，如何在卷上重新应用完全日志（提交）事务的子操作。

reference count（引用计数）——对象管理器关于已经为操作系统进程分配了多少对象指针数目的记录。当每次给对象分发指针时，对象管理器为对象的引用计数器加一；当内核模式组件完成使用指针后，它们调用对象管理器来递减对象的引用计数。

reparse data（重分析数据）——用户定义的有关文件或目录的数据，如它的状态或位置，它可以被创建数据的应用程序、文件系统过滤器驱动程序或 I/O 管理器从重分析点（reparse point）中读取。

reparse point（重分析点）——有一个与之相关的称为“重分析数据”的一块数据的一个 NTFS 文件或目录。

resident attribute（驻留属性）——直接存储在主文件表中的属性。（如果一个文件很小，所有的属性和值（例如它的数据）填充在文件记录里。）

resource arbitration（资源判优）——一个被即插即用管理器用来优化分配硬件资源的进程，这样使每一个设备符合它的操作所必须的要求。因为硬件设备可以在引导时的资源分配之后添加，即插即用管理器必须也能够重新分配资源以便适应动态添加的设备。

restricted token（受限令牌）——使用 CreateRestrictedToken 函数从主要的或模拟（impersonation）的令牌中创建的令牌。限制令牌是衍生令牌可能有些修改的拷贝：权限可能从令牌的权限数组中删除。令牌中的 SID 可以标记为 deny - only。令牌中的 SIDS 可以标记为 restricted。

ring（环）——定义在 Intel x86 处理器体系结构中用以保护系统代码或数据以免被权限低的代码无意或恶意覆盖的权限级别。Windows 2000 中使用权限级别 0 来表示内核模式（或者 0 环）和权限级别 3 为用户模式（或者 3 环）。

safe mode（安全模式）——包含最少的设备驱动程序和服务集的引导配置。通过仅仅依赖引导时必要的驱动程序和服务，Windows 2000 避免加载可能导致崩溃的第三方和其他不必要的驱动程序。

SAM database（SAM 数据库）——包含有定义的用户和用户组以及其他密码和属性的数据库（存储在注册表 HKLM \ SAM 下）。

scatter/gather I/O（分散/集中 I/O）——Windows 2000 支持的、通过 ReadFileScatter 和 WriteFileGather 函数可以获得的高性能 I/O。这些函数允许应用程序从虚拟内存的不止一个缓冲区中发起一个读或写到硬盘的一个文件的连续区域。使用分散/集中 I/O，文件必须为非高速缓存 I/O 打开，使用的用户缓冲区必须页对齐，I/O 必须异步（重叠）。

section object（区域对象）——代表两个或更多进程可以共享的一块内存的对象。区域对象可以映射为调页文件或硬盘上的另一个文件。执行程序使用区域对象来加载可执行的映像到内存，并且高速缓存管理器使用它们来访问高速缓存文件中的数据。在 Win32 子系统中，区域对象称为“文件映射对象”（filemapping object）。

section object pointer（区域对象指针）——每一个打开文件（由文件对象所代表）的结构，它是为所有类型的文件访问以及为文件提供高速缓存而维护数据一致性的关键。区域对象指针结构指向一个或两个控制区域。一个控制区域在访问数据文件时用于映射文件，而另外一个在作为执行映像运行时用于映射文件。

sector（扇区）——物理硬盘的硬件可编址部分。一个 IBM 兼容机上的硬盘扇区典型地为 512 字节。为硬盘准备定义逻辑驱动器的工具包括 MS - DOS Fdisk 或者 Windows 2000 安装程序。被称为主引导记录（MBR）的数据扇区写在硬盘的第一个扇区上。MBR 包括包含有可执行指令（称为引导代码）的一些固定空间和带有定义硬盘上的主分区位置的四个项的分区表。参见

boot code，partition table 。

secure attention sequence（安全提示序列，SAS）——输入用于通知 Winlogon 用户登录请求的按键组合。

security Accounts Manager（SAM）service（安全账户管理服务）——负责管理包含有定义在本地机器上或为域定义的（如果系统是一个域控制器）用户名或用户组的数据库的一些了例程。安全账户管理器在 Lsass 进程的环境中运行。

security auditing（安全审计）—— Windows 2000 检测和记录重要的安全相关的事件或任何创建、访问或删除系统资源的企图的方法。登录标识符记录所有用户的标识，这使得跟踪任何未授权的行为变得容易。

security identifier（安全标识符，SID）——唯一标识系统中进行操作的实体的方法。SID 是一个可变长度的数值，包含有 SID 结构修正数，48 位的标识符授权值，32 位的子授权或相对标识符（*RID*）值的可变数。

security reference monitor（安全引用监视器，SRM）——在本地机器上体现安全策略的 Windows 2000 可执行程序的组件。它监视操作系统资源，完成运行时对象保护和审计。

semaphore（信号量）——通过允许最多的线程数目来访问信号量保护的资源的方式提供资源门的计数器。

server process（服务器进程）—— 作为 Windows 2000 服务的用户进程，诸如 Event log 和 Schedule 服务。很多添加的服务器应用程序，诸如 Microsoft SQL Server 和 Microsoft Exchange Server，也包括作为 Windows 2000 服务的组件。

session（会话）——由进程和其他代表着单个用户工作站登录会话的系统对象构成（诸如 Windows 站、桌面以及窗口）。每一个会话有一个会话指定的分页交换区，它被 Win32 子系统的内核模式部分（Win32k.sys）用来分配会话私用的 GUI 数据结构。除此以外，每一个会话都有 Win32 子系统进程（Csrss.exe）和登录进程（Winlogon.exe）的自备拷贝。

session space（会话空间）——系统空间用于映射指定到用户会话的信息的组件。（当安装 terminal Services 时，Windows 2000 支持多个用户会话）。会话工作集表描述常驻和使用的会话空间的部分。

shared cache map（共享的高速缓存映像）——描述一个高速缓存文件状态的结构，包括它的大小（为安全原因）和它的有效数据长度。

shared memory（共享内存）——不止一个进程可见的或者不止一个虚拟地址空间中出现的内存。

signal state（信号状态）——同步对象的状态。

simple volume（简单卷）——代表文件系统驱动程序作为单一单元管理的单一分区的扇区中的一系列对象。

single sign - on（单一注册）——登录信息可以同时在不止一个系统上验证的能力。例如，一个登录 Windows 2000 系统的用户可能同时被一个 UNIX 服务器验证。这个用户能够在运行 Windows 2000 的机器上不需要附加的验证来访问 UNIX 服务器上的资源。

small memory dump（小内存转储）——小内存转储（Windows 2000 Professional，缺省 64KB 大

小）包含有停止代码和参数、加载的设备驱动程序列表、描述当前的进程和线程的数据结构（称为 EPROCESS 和 ETHREAD）以及导致崩溃的线程的内核栈。

spanned volume（跨越卷）——由一个或更多的磁盘上最多 32 个自由分区构成的单一逻辑卷。Windows 2000 Disk management Microsoft Management Console（MMC）插件组合分区为一个跨越卷，它们可以被 Windows 2000 支持的任何一个文件系统格式化。

sparse file（稀疏文件）——经常很大的文件，包含相对于它们的大小来说很少的非零数据。

spinlock（自旋锁）——内核用于实现多处理器互斥的锁机制。自旋锁得名来自于以下事实：内核（因此，处理器）一直在受约束，“在自旋”，直到得到锁。自旋锁象它们保护的数据结构一样，常驻在公用内存中。参见 queued spinlock。

stack frame（栈帧）——当过程调用时，代表着过程激活的信息被推入帧。栈帧有着与之相关的一个或更多的异常处理程序。它们当中的每一个保护着源程序中的特定代码块。

stream（流）——文件中顺序的字节码。

striped volume（带区卷）——多至 32 个分区的系列，每个硬盘一个分区，它们组合成一个逻辑卷。带区卷也称为 RAID 0 级别卷（RAID－0）。带区卷中的每一个分区不需要跨越整个硬盘。仅有的限制是每一个硬盘上的分区大小一样。

strucutured exception handling（结构化异常处理）——异常发生时候允许应用程序获得控制的异常处理类型。应用程序可以纠正情况并返回异常发生的位置，展开栈（这样结束产生异常的子程序的执行），或者返回不能识别异常的系统，并继续搜索能够处理异常的异常处理程序。

subsection（子区）——为文件每个区描述映射信息的结构（read－only、read－write、copy－on－write，等等）。

subsystem dynamic－link library（子系统动态链接库，DLL）——把一个文档化的函数映射为适当的非文档化的 Windows 2000 系统服务调用的动态链接库。这种映射可能涉及也可能不涉及到向用户应用程序提供服务的环境子系统进程发送消息。

symbolic link（符号链接）——不直接引用对象名的机制。

symmetric encryption algorithm（对称式加密算法）——使用同一个密匙加密或解密数据的算法。对称式加密算法非常快，这使得它适合加密大量的数据，诸如文件数据。然而，它们也有缺点，如果获取密匙后，可以绕过安全机制。

symmetric multi－processing（对称式多处理，SMP）——不存在主处理器的多处理操作系统——操作系统以及用户线程可以调度到任何一个处理器上。所有的处理器共享共同的内存空间。

Synchronization（同步）——线程通过等待对象从一个状态转换到另一个状态来同步它的执行的能力。线程可以与执行程序进程、线程、文件、事件、信号量、互斥以及定时器同步。区域、端口、访问令牌、对象目录、符号链接、配置（profile）以及关键对象都不支持同步。

synchronous I/O（同步 I/O）——一种 I/O 模型，在这种情况下设备进行数据传输，当 I/O 完成时返回状态代码。程序接着立刻访问传输的数据。当用最简单的形式使用时，Win32 ReadFile 和 WriteFile 函数能够同步执行。在将控制交给调用程序之前，它们完成 I/O 操作。

system access - control list（系统访问控制列表，SACL）——指定哪些用户的哪些操作应该记载在安全审计日志里。

system audit ACE（系统审计 ACE）——被系统访问控制列表（SACL）包含的访问控制项（access - control entry）以及系统审计对象 ACE，它指定了特定的用户和用户组在对象上的哪些操作应该被审计。系统审计对象 ACE 指定了一个 GUID 显示 ACE 适用的对象和子对象类型以及一个可选的 GUID，它控制 ACE 到特定的子对象类型的传播。如果一个系统审计访问控制列表是空的，则对象没有被审计。

system cache（系统高速缓存）——用于映射在系统高速缓存中打开的文件的页面。

system page table entry（系统页面表项，PTE）——用于映射系统页面诸如 I/O 空间、内核栈以及内存描述符列表的系统 PTE 交换区。

system service dispatch table（系统服务调度表）——每一项都包含指向系统服务的指针而不是中断处理例程的指针的表。

system service（or executive system service）（系统服务（或者执行程序系统服务））——Windows 2000 操作系统中用户模式可以调用的本机函数。例如，NtCreateProcess 是 Win32 CreateProcess 函数调用来创建新进程的内部系统服务。

system support process（系统支持进程）——不属于 Windows 2000 服务的用户进程，诸如登录进程和会话管理器。（也就是说，不被服务控制器启动）。

system thread（系统线程）——仅在内核模式运行的线程。系统线程始终驻留在系统进程中（始终是进程 ID 2）。这些线程具有常规用户模式线程的所有属性和环境（诸如硬件环境、优先级等等），但不管是在 Ntoskrnl.exe 中还是在其他被加载的设备驱动程序中，它们仅能运行在系统空间代码加载的内核模式执行代码中。系统线程不拥有用户进程地址空间，因此必须从操作系统内存堆栈诸如调页或非页交换区中分配动态存储。

system worker thread（系统工作者线程）——在系统初始化时在 System 进程中创建的仅为代表其他线程工作而存在的线程。

system working set（系统工作集）——正在被系统高速缓存使用的物理内存、页交换区、Ntoskrnl.exe 中可调页的代码和设备驱动程序中可调页的代码。参见 process working set。

thread（线程）——在进程中 Windows 2000 调度执行的实体。线程包括代表处理器状态的一些易失寄存器的内容；两个栈，一个在线程处于内核模式时使用，而另外一个在线程处于用户模式时使用；为子系统、运行库以及动态链接库使用的私用存储区域；称为“线程 ID”的唯一标识符（内部也称为“客户机 ID”）。

thread context（线程环境）——线程的易失寄存器、栈和私用存储区域。因为对于 Windows 2000 运行的每一种机器体系结构这些信息都不同，这些结构是体系结构指定的。实际上，Win32 GetThreadContext 函数返回的 CONTEXT 结构是 Win32 API 中机器相关的唯一公开数据结构。

timer（定时器）——当一定的时间消逝时通知线程的机制。

Transaction（事务）——修改文件系统数据或改变卷目录结构的 I/O 操作。构成事务的单独硬盘更新必须以原子属性的方式执行。也就是说，一旦事务开始执行，所有的硬盘更新必须完

成。如果系统失败中断了事务，已经完成的那部分必须撤消或重新运行。重新运行操作使数据库恢复前面已知的一致的状态，仿佛事务从未发生。

transaction table（事务表）——跟踪已经开始还未完成的事务的表。在恢复期间这些活动的事务的子操作必须删除。

transition（过渡）——一种无效页面表项（PTE），这种情况下，可得到的表在内存中要么是备用（standby）、修改的（modified）或者 modified - no - write 表。页面从列表中删除并添加在工作集中。

translation look - aside buffer（变换监视缓冲区）——最近变换的虚拟页面数的 CPU 高速缓存。

trap（陷阱）——当异常或中断发生时，处理器捕获执行线程的机制，执行线程从用户模式转向内核模式，并把控制转向操作系统的固定位置。在 Windows 20000 中，处理器把控制转向内核陷阱处理程序。

trap frame（陷阱帧）——中断线程执行状态存储在其中的数据结构。在处理完中断或异常后，这些信息允许内核恢复线程的执行。陷阱帧通常是线程完成环境的子集。

trap handler（陷阱处理程序）——内核中作为转换板的模块，监视处理器检测到的异常和中断，并把控制转向处理这些情况的代码的模块。

type object（类型对象）——包含对于对象的每个实例都相同的信息的内部系统对象。

Unicode——为世界上大多数众所周知的字符集定义的唯一的 16 位值的国际字符集标准。

update record（更新记录）——NTFS 写到日志文件中最通用类型的记录。每一个更新记录包含重做更新文件系统结构的操作所需要的信息。

user mode（用户模式）——应用程序运行于其中的非特权处理器模式。一些有限的接口在这种模式下可用，并且系统数据的访问也是受限制的。参见 kernel mode。

view（视图）——进程所需要的区域对象的一部分。区域对象可以引用比进程地址空间更大的文件。（如果调页文件支持区域对象，调页文件中必须存在充足的空间以包含它）。为了访问一个非常大的区域对象，进程可以通过调用 MapViewOfFile 函数并指定映射的范围来映射区域的视图。因为只有区域对象的视图在需要时必须映射到内存，所以映射视图允许进程保存地址空间。参见 section object。

virtual address descriptor（虚拟地址描述符，VAD）——内存管理器维护用来跟踪哪些虚拟地址已经在进程地址空间中保留的数据结构。为了使查找有效，WAD 的结构被设置为自平衡的二分树。

virtual address space（虚拟地址空间）——进程能使用的一组虚拟内存地址。

virtual block caching（虚拟块高速缓存）——Windows 2000 高速缓存管理器用来跟踪哪些文件的哪些部分在高速缓存之中的方法。

virtual cluster number（虚拟簇号，VCN）——虚拟簇号给属于特定文件的簇从 0 到 m 编号。VCN 在物理上并不要求必须是连续的，但它们可以映射为卷上逻辑簇号的任意数目。

virtual device driver（虚拟设备驱动程序，VDD）——用于仿真 16 位 MS - DOS 应用程序的驱动程序。它们捕获 MS - DOS 应用对 I/O 端口的访问，并把它们翻译为本机 Win32 I/O 函数。

因为 Windows 2000 是一个完全被保护的操作系统，因此用户模式 MS - DOS 应用程序不直接访问硬件，而通过一个实际的内核模式设备驱动程序。

virtual memory manager（虚拟内存管理器）—— 实现虚拟内存，内存管理器方案能够为每一个进程提供可以超过实际物理内存的大的私用地址空间。

volume（卷） ——被当作一个整体单元的一个或更多逻辑磁盘分区。

volume file（卷文件）——包含有卷名、卷按照其格式化的 NTFS 版本号的系统文件。（文件名为$ Volume）当它的一个位（bit）被设置时表示硬盘已经发生破坏并且必须被 Chkdsk 实用工具修复。

volume manager（卷管理器）——用来表示 FtDisk 和 DMIO 的术语，因为 FtDisk 和 DMIO 都支持同样的多分区卷类型。

volume set（卷组）——由一个或更多的硬盘上的最多 32 个自由空间组成的单一逻辑卷。

wait block（等待块）——代表等待对象的线程的数据结构。每一个处于等待状态的线程都有一个代表线程正在其上等待的对象的等待块列表。每一个调度对象都有代表着哪一个线程正在对象上等待的等待块列表。

wait hint（等待暗示）——在通知系统关机已经完成之前一个表明服务应该等待多久的值。

WDM driver（WDM 驱动程序）——遵从 Windows 驱动程序模型（WDM）的设备驱动程序。WDM 包含对 Windows 2000 电源管理、即插即用以及 Windows 管理装置（WMI）的支持。WDM 在 Windows 2000，Windows 98 以及 Windows Millennium Edition 上实现，这样 WDM 驱动程序在这些操作系统之间是源代码兼容，并且在很多情况下是二进制代码兼容的。有三种 WDM 驱动程序的类型：总线驱动程序、功能驱动程序以及过滤器驱动程序。

Win32 application programming interface（Win32 应用程序编程接口，API）—— Micorsoft Windows 操作系统系列主要编程接口，这些操作系统包括 Windows NT 4、Windows 2000、Microsoft Windows 95、Windows 98、Windows Millennium edition 以及 Microsoft Windows CE。

Win32 service（Win32 服务）——在系统启动时，启动提供与交互用户关系不紧密的服务的进程的机制。服务类似于 UNIX 守护进程并经常实现客户机/服务器应用程序的服务端。

Windows station（窗口站）—— 一个窗口站包含桌面，桌面包含窗口。在一个终端上，仅有一个窗口站可见并接受用户鼠标和键盘输入。在一个 Terminal Services 环境里，每个会话有一个窗口是可见的，但所有的服务都作为终端会话的一部分运行。

Windows 2000 driver（Windows 2000 驱动程序）——当必要时，与 Windows 2000 电源管理和即插即用管理器集成的设备驱动程序。它们包括海量存储设备、协议栈以及网络适配器的驱动程序。

Windows Management Instrumentation（WMI）manager（Windows 管理装置（WMI）管理器）——使得设备驱动程序可以公开性能和配置信息，并从用户模式 WMI 服务中接收命令的执行程序的组件。WMI 信息既可以在本地机器，也可以通过网络在远程机器上获取。

work item（工作项）—— 当设备驱动程序或者执行程序组件通过调用执行程序函数 ExQueueWorkItem 或者 IoQueueWorkItem 来请求系统工作者线程的服务时，放在队列调度器对象上的工作单元。工作项包括指向例程的指针以及当例程处理工作项时线程传递给例程的参数。例

程由设备驱动程序或者请求被动级别执行的执行程序的组件实现。

working set（工作集）——驻留在物理内存中的虚拟页的子集。有两种工作集——进程工作集和系统工作集。

working set manager（工作集管理器）——运行在平衡组管理器系统线程（balance set manager system thread）的环境中初始化工作集的自动修整以增加系统中可获得的自由内存的例程。虽然 Windows 2000 企图通过把修改的页面写回硬盘来保留可获得的内存，当以非常高的速率产生修改的页面时，需要更多的内存以满足内存需求，当物理内存变少时（当 MmAvailablePage 少于 MmMinimumFreePage 时），调用工作集管理器。

write - back（写回）——延迟写文件系统使用以提高性能的高速缓存策略。在写回时，文件系统把文件修改写到高速缓存中，并以优化的方式刷新高速缓存的内容到硬盘，通常这种过程以后台活动存在。

write - through（写穿透）——FAT 文件系统立刻把硬盘的修改写到硬盘的算法。不象认真写（careful - write）方法，写穿透技术不要求文件系统有序的写以避免不一致性。参见 careful write。

zero page thread（清零页面线程）——一个内核模式系统线程（System 进程中的线程 0）。清零页面线程把自由列表上的页面都置为零，这样清零页面的高速缓存可以满足将要置零请求页面错误的需要。

Inside Microsoft Windows 2000, Third Edition

Windows 2000 内部揭密

《Windows 2000注册表管理》

ISBN7-111-08149-8/TP·1539
页数：262 页　定价：28.00 元

《Windows 2000系统编程》

ISBN7-111-08618-3/TP·1736
页数：458 页　定价：55.00 元

《Windows 2000 DNS技术指南》

ISBN7-111-07571-4/TP·1186
页数：208 页　定价：32.00 元

ISBN 7-111-09100-0

9 787111 091004

在多处理器支持　线程调度　中断处理　内存管理　安全　I/O处理　文件系统驱动程序　以及文件高速缓存等专题上包含极有价值的细节。

查询注册表机制揭秘　进程句柄　注册表精简　Microsoft Win32服务精简　Windows Management Instrumentation(WMI)　地址窗口扩展(AWE)　即插即用　电源管理　Windows驱动程序模型(WDM)　存储体系结构　NTFS加强以及网络等主题上提供最新信息。

展示如何使用最新实用工具观察Windows 2000内部行为的动手实验。配套光盘包含本书电子版　以及展示Windows 2000内核的工具。

- 《Windows 核心编程》
- 《Visual C++ 6 学习指南》
- 《Visual Basic 6 学习指南》
- 《Windows 网络编程技术》
- 《SQL Server 7 资源指南》
- 《Visual Basic 6 编程技术大全》
- 《Windows DNA可扩展设计》
- 《Microsoft SQL Server 7性能优化》
- 《XML学习指南》
- 《Windows 2000安全Web应用程序设计》
- 《微软XML解决方案》
- 《Windows 2000 Server资源大全》（共6卷）
- 《Win32开发人员参考库》（共5卷）
- 《活动目录开发人员参考库》（共5卷）
- 《COM+与Visual Basic 6分布式应用程序设计》（第2版）
- 《Windows 2000性能优化技术参考》
- 《SQL Server 2000 Analysis Services学习指南》
- 《COM+技术解决方案设计》
- 《Windows 2000内部揭密》

适用水平：中、高级

封面设计 曲春希
www.china-pub.com
北京市西城区百万庄南街1号　100037
购书热线：(010)68995265, 8006100280(北京地区)

ISBN 7-111-09100-0/TP·2018
定价：69.00 元(附光盘)

微软公司核心技术书库

Windows 2000 内部揭密

（美） David A. Solomon
Mark E. Russinovich 著

詹剑锋 张文耀 黄 艳 等译

机械工业出版社
China Machine Press

本书深入揭示 Windows 2000 内部结构和运行机制，涉及 Windows 2000 最基础的系统组件和基本概念。主要内容包括系统体系结构、系统机制、管理机制、内存管理、安全机制、I/O 系统、文件系统、网络体系等。本书用大量实验展示了 Windows 2000 的内核，有效地使读者深刻地理解 Windows 2000 系统，充分利用该系统进行应用开发。配套光盘包含本书电子版，以及展示 Windows 2000 内核的工具。

本书版权登记号：图字：01－2000－3118

图书在版编目（CIP）数据

Windows 2000 内部揭密/（美）所罗门（Solomon, D.A.），（美）罗森欧维斯（Russinovich, M.E.）著；詹剑锋等译. －北京：机械工业出版社，2001.10

（微软公司核心技术书库）

书名原文：Inside Microsoft Windows 2000, Third Edition

ISBN 7－111－09100－0

Ⅰ.W... Ⅱ.①所...②罗...③詹... Ⅲ.窗口软件，Windows 2000 Ⅳ.TP316.7

中国版本图书馆 CIP 数据核字（2001）第 045663 号

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）

责任编辑：宋 宏 张鸿斌

北京昌平奔腾印刷厂印刷 ·新华书店北京发行所发行

2001 年 10 月第 1 版第 1 次印刷

787mm × 1092mm1/16·36.5 印张

印数：0 001-5 000 册

定价：69.00 元（附光盘）

序　言

我很感谢作者给我这个机会为这么重要的书写序言。

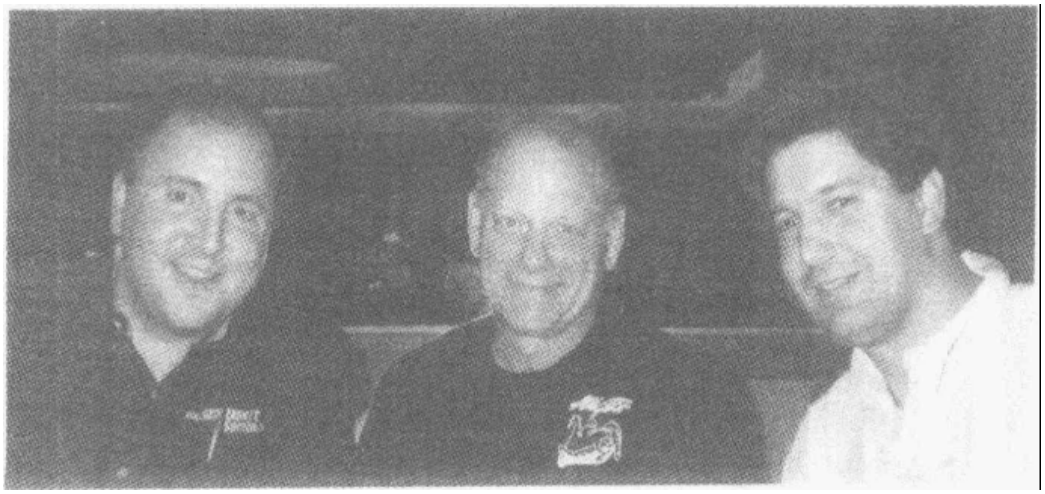
当我在 Digital Equipment Corporation（数字设备公司）工作时，第一次结识当时只有 16 岁的 David Solomon，当时我在为 VAX 开发 VMS 操作系统。从那时开始，他涉及操作系统的开发和操作系统内部的讲解工作。我与 Mark Russinovich 经常接触是最近的事，但是我很早就知道他对操作系统有很深的见解。他做了很多惊人的工作，例如在 Microsoft Windows 98 上运行的 NTFS 文件系统和动态 Microsoft Windows 2000 内核调试器，该调试器可以用来研究正在运行的 Windows 2000 系统。

Microsoft Windows NT 系统开发于 1988 年 10 月，目的是创建一种解决 OS/2 兼容性、安全、POSIX、多处理、网络集成和可靠的可移植系统　随着 Windows 3.0 的出现和巨大成功，开发这个系统的目的变为与 Windows 直接兼容，并且把 OS/2 兼容系统变为子系统。

我们最初认为可以在两年内开发出 Windows NT 系统，但结果直到四年半后的 1993 年夏天 Windows NT 才发布。这一版本支持 Intel i386、ntel i486 和 MIPS R4000 处理器。六个星期后我们又推出了支持 Digital Alpha 处理器的版本。

Windows NT 第一版比预期的规模大，时间晚，所以下一个推出的计划叫 Daytona，这是以佛罗里达州的高速公路命名的。这个版本的目的是减小系统的占用空间，提高系统速度，当然也使它更加可靠。在 1994 年秋天推出 Windows NT 3.5 的六个月之后，我们又推出了 Windows NT 3.51，这是最新的可以支持 IBM PowerPC 处理器的版本。

Windows NT 下一个版本的推出是为了更新用户界面以便与 Windows 95 相一致，并且集成在微软已开发了几年的 Cairo 技术。这个版本用了两年多的时间，于 1996 年夏推出，这个版本就是 Windows NT 4.0。



左起：David Solomon、David Cutler　Mark Russinovich

接着推出了 Windows 2000 系统。同以前的版本一样，Windows 2000 也是基于 Windows NT 技

术建立的，并且增加了像活动目录（Active Directory）这样的新的重要特性。开发 Windows 2000 用了三年半时间，它是经过多次测试和改版的 Windows NT 的最新版。Windows 2000 是对四个体系结构经过 11 年开发所达到的顶峰。它的代码库正在向新的 Intel IA－64 体系结构移植。Windows 2000 是目前我们开发的 Windows NT 技术的最好版本，我们还将继续努力开发新的版本。

本书是介绍 Windows 2000 的内部结构和运行机制的权威著作。作者在 Windows NT 的代码细节上曾经做过大量工作，在本书中使用了很多工具并例举了大量实验以帮助读者理解该系统的工作原理。本书应成为系统开发人员的必备参考书。

微软公司高级突出贡献工程师

David N. Cutler

艾 奇 序

1996 年 8 月我们开始开发 Microsoft Windows 2000。大约三年半之后即 1999 年 12 月 15 日，我们开发了 Windows 2000 Professional、Windows 2000 Server 和 Windows 2000 Advanced Server，并投入生产。在 5000 名工作人员的共同努力下，Windows 2000 成为微软乃至全电脑业中最大的操作系统，它也是我们所生产的最可靠、最综合的系统。

现在，世界上许多互联网站和大型企业都在使用 Windows 2000，Windows 2000 开始成为商业乃至家庭中的标准服务操作系统。Windows 2000 包括许多使我们惊奇的新技术，它可以用于桌面机或笔记本电脑，并且有许多服务功能，包括文件、打印、Web、数据库、事务处理、拨号、路由、流媒体、业务流程应用等等。了解所有这些功能需要花费很多精力，但如果你从这个系统的内核概念开始理解，则更容易掌握这些功能。

如果你像我一样想知道系统究竟是怎样工作的，那么仅仅阅读指导手册或帮助文件是远远不够的。本书将介绍 Windows 2000 内部工作机制，介绍如何更好地利用它，如何最大限度地确保运行的可靠性和安全性，如何判断错误的根源等等。

本书在展示 Windows 2000 的内部技术上做了很多工作。本书包含了直接用于实验和进行诊断的工具，这是本书重要的资源。读完本书后，你会更深入地理解这个系统是如何协调的，新版本是如何改进的，并且会更充分地利用它。

虽然我对 Windows 2000 很熟悉，但读了本书后，我知道了许多以前不了解的东西。打开了本书便如同揭开了有史以来最重要的操作系统的面纱。

微软公司平台部副总监

Jim Allchin

前　言

本书是供计算机专业人员使用的，主要面向开发人员和系统管理人员。它介绍了 Microsoft Windows 2000 操作系统内部的工作原理。本书可帮助开发人员在 Windows 2000 平台上开发特殊功能时更好地理解设计方案的基本原理，也可以帮助开发人员解决复杂的难题，还可以帮助系统管理员理解系统运行的规律并解决系统故障。阅读本书可以深入了解 Windows 2000 的工作原理和运行规律。

本书的结构

本书前两章介绍了全书所用的基本术语和概念。接下来的三章讲述了系统的基本工作机制，即分别介绍了系统机制、启动与关机、系统管理机制。后面章节分别介绍了进程、线程和作业，内存管理，安全机制，输入输出系统，存储管理，高速缓存，文件系统和网络。几乎涉及了所有 Windows 2000 操作系统的内核概念。

第三版的特征

本书是第三版，包括了许多《Inside Windows NT》第二版没有的内容，例如启动与关机、内部服务、内部注册、文件系统驱动和网络。包括 Windows 2000 改进和增强，例如 Windows Driver Model（WDM）、即插即用、电源管理、Windows 管理装置、加密、作业对象和终端服务。

本书也是第一次附带一张光盘，包括了浏览 Windows 2000 系统内部的工具。利用光盘也可以查阅本书的电子版。另外书中增加了许多实验，例如讲述利用内核调试器观察 Windows 2000 系统内部状态。

实验

当涉及到某个工具可以用来揭示或演示 Windows 2000 内部运行的特征时，便在“实验”框里介绍其步骤。本书中有很多这样的例子。你一边阅读就可以一边看到 Windows 2000 内部工作的过程。这样比只是看书印象要深得多。很多实验都要用到内核调试器。本书配套光盘上的动态内核调试器（LiveKD）使这些实验运行起来容易且安全。

未涉及到的内容

Windows 2000 是一个庞大而复杂的操作系统。本书并没有涉及到 Windows 2000 内部的每一个方面，只专注介绍基础系统组件。例如本书没有涉及 COM +，它是 Windows 分布式面向对象程序设计设施的基础。

本书只介绍系统内部工作机制，不是针对用户、程序员或系统管理员的，没有讲述如何使用、编程和配置 Windows 2000。

警示和告诫

本书讲述的是 Windows 2000 的内部结构和操作，但很多信息在出版过程中已有所改变（虽然外部界面例如 Win32 API 没有改变）。例如，我们提到了 Windows 2000 系统内部例程、数据结构、内核变量和内部用来作出资源大小和性能相关决定的算法和数值。这些定义的要点在发布过程中可能改变。

并不是说本书中描述的要点都会在出版过程中改变，但我们不能保证它们没有改变。任何使用了没存档的接口的软件都有可能在未来发布的 Windows 2000 上无法工作。更糟的是，使用了没存档的接口的基于内核模式的软件（例如设备驱动器）在 Windows 2000 新的发布版本中可能会导致死机。

配套光盘的使用

这套光盘包括 Sysinternals Web 站点（www.sysinternals.com）的全部内容和其他一些有用的工具。那个网站是由 Mark Russinovich（本书作者之一）和 Bryce Cogswell 负责的。这套光盘也包括本书的电子版以及调试工具和符号（使用调试工具和符号的方法见光盘上的 Readme.txt 文件）。

要查看光盘上的内容，只需把光盘插到光驱里。如果系统有自动运行功能，则画面显示选择项以供查看。如果没有自动运行功能，就从光驱的目录中运行光盘。

系统内部

为了便于用户使用，网站 www.sysinternals.com 的内容已经包括在光盘中，你可以在该网站上找到本书中的实验工具，也可以从光盘上运行这些工具或通过从自动显示画面选择运行安装程序，并按照安装说明把这些工具安装在硬盘上。

可以通过从自动显示画面选择 Browse CD Sysinternals（浏览光盘的 Sysinternals）选项，或者从 Sysinternals - WebSite 目录打开 Ntinternals.htm 文件来查看整个网站内容。可以通过从自动显示画面选择 Run Setup（安装），并根据提示将整个网站的内容拷贝到硬盘中。

要得到最新的 Sysinternals 网站的内容和工具，请访问 www.sysinternals.com 网站，也可以从光盘的自动显示画面选择 Browse Online Sysinternals（浏览在线的 Sysinternals）选项进入该网站。

工具

光盘提供了许多工具，位于 \ Tools 文件夹。这些工具中有性能监视扩展 DLL（KVarPerf），允许你从性能工具中监视 Windows 2000 内核变量。另外还有 LiveKd 工具，它允许标准微软内核调试工具不需要特殊调试选项的情况下运行，微软内核调试工具有 Kd.exe、Windbg.exe、I386Kd.exe 等等。

从自动显示画面中选择 Run Setup（运行安装）选项（或在 \ Setup 文件夹中运行 Setup.exe 文件），并根据提示来安装这些工具，也可以直接从光盘中运行这些工具，然而 LiveKd 必须在光盘上的 Debuggers 目录而不是 Tools 目录下运行。关于光盘的 LiveKd 和系统内核调试的更多信

息请参见光盘根目录下的 Readme.txt 文件)。

系统需求

下列是运行光盘所需要的系统配置:

■ 任何支持的 Microsoft Windows 2000 Professional、Server、Advanced Server 或者 Datacenter Server 配置。

■ 装载实验用的工具 (Tools 和 Sysint 文件夹),需要 5MB 硬盘空间。如果安装 www.sysinternals.com 网站,则需要 30MB 空间。装载 Debuggers 目录下的内容需要 20MB 空间,装载全部 Symbols 内容需要 20MB 空间。

■ 本书中有些实验要用到 Windows 2000 的支持工具、调试工具和源组件 (Professional 版或 Server 版)。第 1 章介绍了这些工具和它们的位置。

E - book

光盘包含本书的电子版,可以在屏幕上阅读本书,查找内容。安装和使用电子版本参见 Ebook 目录下的 Readme.txt 文件。

技术支持

我们做了很大努力以确保书和配套光盘内容的正确性。如果遇到问题,请参考下面的资料。

作者联系方式

本书不是完美的,有可能存在一些不当之处。我们删去了一些本应保留的内容。如果发现错误或提出建议,请给我们发 E - mail 到 insidew2k@sysinternals.com。本书的更新和修改内容会在 www.sysinternals.com/insidew2k 网页上刊出。

微软出版社联系方式

微软公司在下列网站也提供书中错误的正确解答:

http://mspress.microsoft.com/support/

如果你对本书或配套光盘有什么意见和建议,可与微软出版社联系:

通信地址:

Microsoft Press

Attn: Inside Microsoft Windows 2000 Editor

One Microsoft Way

Redmond, WA 98052 - 6399

E - mail 地址:

mspinput@microsoft.com

上述地址中没有产品支持的信息,访问 www.microsoft.com/windows 2000 网址就可了解微软

公司的产品，也可以在工作日太平洋时间上午 6 点到下午 6 点打电话（425）635－7011 与公司联系，或者访问微软的在线支持网站：*suppport.microsoft.com/support*。

参加本书翻译的有詹剑锋、张文耀、黄艳、魏名、廖彬、巩慧、黄千云、韩华、程小鹏、周武、陈文海、詹向前、刘宏友、魏晓东、袁峰、马健忠、戴思全、叶迪盛、戴亚希、叶海男、陈炳阳、蔡良彬、刘奕、吕可为、王立彬和李小燕。