Win32 多线程程序设计

Multithreading Applications in Win32

The Complete Guide to Threads

线程完全手册

Jim Beveridge & Robert Wiener 著
侯捷 译

Pearson Education

函数索引

_beginthread 249
_beginthreadex 224
AfxBeginThread 279,288
CloseHandle 38
CoInitializeEx 435
CreateEvent 121
CreateFile 152
CreateFileMapping 346
CreateIoCompletionPort 176
CreateMutex 108
CreateSemaphore 117
CreateThread 26
DeleteCriticalSection 96
DisableThreadLibraryCalls 374
DllMain 369
_endthread 250
_endthreadex 227
EnterCriticalSection 97
ExitThread 45
FileIOCompletionRoutine 164
GetExitCodeThread 41
GetQueuedCompletionStatus 179
GetStdHandle 238
GetThreadPriority 141
InitializeCriticalSection 96
InterlockedDecrement 126
InterlockedExchange 127
InterlockedIncrement 126
LeaveCriticalSection 97
MapViewOfFile 348
MsgWaitForMultipleObjects 86
MsgWaitForMultipleObjectsEx 164
OpenFileMapping 351
PostThreadMessage 315
ReadFile 153
ReadFileEx 163
ReleaseMutex 111
ReleaseSemaphore 119
ReplyMessage 309
ResumeThread 145
SetThreadPriority 140
SleepEx 163
SuspendThread 146
TerminateThread 132
TlsAlloc 386
TlsFree 388
TlsGetValue 387
TlsSetValue 386
UnmapViewOfFile 353
GetOverlappedResult 155
WaitForMultipleObjects 81
WaitForMultipleObjectsEx 164
WaitForSingleObject 72
WaitForSingleObjectEx 163
WriteFile 153
WriteFileEx 163

Win32 多线程程序设计

线程完全手册

Multithreading Applications in Win32

The Complete Guide to Threads

Jim Beveridge & Robert Wiener 著

侯 捷 译

译　序

侯　捷

thread 就是“线”。台湾计算机术语采用“绪”这个译词，“绪”就是“线”的雅称，multithread 就是“多绪”。大陆计算机术语采用“线程”一词，multithread 就是“多线程”。

Threads（线程）是比 processes（进程）更小的执行单元，CPU 的调度与时间分配皆以 threads 为对象。

计算机领域中早就存在 threads 的观念和技术，但是早期个人电脑操作系统（主要是 DOS），别说 multithread，连 multitask，multiuser 亦不可得。因此，从当时，乃至延伸至今，threads 的概念和功能对许多非计算机专业科班出身者而言，属于一种“崇高而难以亲近”的位阶，对许多计算机专业科班出身者而言，却又只是“操作系统”这门课里高高在上的一个名词。

本书第一章第一句话值得玩味：“计算机工业界每有新的技术问世，人们总是不遗余力地去担忧它是不是够重要。公司行号虎视眈眈地注意其竞争对手，直到对方采用并宣扬这技术有多么重要，才开始急急赶上。不论这技术是不是真的很重要，每一个人都想尽办法让终端用户感觉真的很重要。终端用户终于真的觉得需要它了——即使他们完全不了解那是什么东西。”

threads 大约就是这么一种东西吧。OS/2、Windows NT、Windows 95 这类“新一代 PC 操作系统”初上市时，便一再强调其抢先式多任务（preemptive multitasking）的多线程（multithreaded）环境。拜强势行销之赐，霎时间线头到处飞舞，高深的

计算机术语在街巷里弄之间传播了开来，颇有点 “Neural Fuzzy”洗衣机的味道。

这倒也算是好事！

搞不清楚 threads 是什么，对终端用户而言或许没有关系，对技术人员可就不妙。Threads 绝对可以缩短程序的执行时间吗？应该尽量多产生 threads 来帮助程序工作吗？任何种类的程序都可以获得 multithreads 的好处吗？错！错！错！似是而非的观念可能会把你的程序带往更坏（而非更好）的境界。

Threads 不是新东西，但它借着 Windows 的庞大装机量初次广泛进入个人电脑世界，带给个人电脑巨大的冲击。产生 threads 毫无困难，要让它们分工容易，而要让它们合作，那可就得花相当多的心思。Threads 不一定带来好处，运用不当的话，它会在执行效率上惩罚你。

Threads 是 Win32 操作系统和 Win32 程序设计不可或缺的重要环节，每一本重量级 Win32 程序设计书籍都不会忽略这个题目（请参考附录 B）。但是这些书多半仅以一章（甚至只是一节）来介绍这个题目。不够，真的不够，我们缺乏一本兼具理论并重实际的 threads 专著。《Multithreading Applications in Win32》的内容兼具理论和实际，轻薄短小的身形则在大部头书当道的今天让我们心情轻松。这是一本导人性书籍，在 threads 专著里算是比较容易入门的。但是你必须知道，threads 不可能让你轻松学习！同步控制、多线程通讯、数据一致性……样样耗费你的心神，考验你专心致志的程度。读这本书，还请你武装一下自己的精神。

对于中译本，我有以下两点说明：

1. 译本内的程序实例直接取自书附光盘。如果与英文版书面代码稍有出入，恐怕是因为作者直接在实际程序上做了点小变动而未能及时反应到书面。如果出现这种差异，我会在程序代码列表之后以译注的方式告诉你。

2. 译本保留了相当多的原文技术术语，主要是考虑本书的潜在读者层。如果不采用原文术语，可能各位反而要倒译回去半看半猜，那么译本的价值就

适得其反了。许多地方我不厌其烦地在中文术语后面加上原文术语，为的也是同样的原因。

Multithreading 非常重要。当支持多处理器（multiprocessor）的操作系统逐渐普及时，具备多处理器的个人计算机也逐渐普及。我相信，多线程程序设计是每一位技术人员都必须面对的技术。即便现在，多线程能够提高多人、多任务程序的使用者接口（UI）反应度，同样也是高阶技术人员应该追求的目标。

侯捷　新竹　1997.05.31
jjhou@jjhou.com
http://www.jjhou.com（繁体中文）
http://jjhou.csdn.net（简体中文）

目　录

函数索引（Function Index）……封面里
常见问答集（Frequently Asked Questions）……vii

第一篇　上路吧，线程

第 1 章　为什么要“千头万绪”……3
一条曲折的路……4
与线程共枕……7
为什么最终用户也需要多线程多任务……8
Win32 基础……10
Context Switching……14
Race Conditions（竞争条件）……16
Atomic Operations（原子操作）……19
线程之间如何通讯……22
好消息与坏消息……22

第 2 章　线程的第一次接触……25
产生一个线程……26
使用多个线程的结果……31
核心对象（Kernel Objects）……36
线程结束代码（Exit Code）……40
结束一个线程……45
错误处理……48
后台打印（Background Printing）……50
成功的秘诀……59

第 3 章 快跑与等待.....61
看似闲暇却忙碌（Busy Waiting）.....62
性能监视器（Performance Monitor）.....66
等待一个线程的结束.....72
叮咚：被激发的对象（Signaled Objects）.....74
等待多个对象.....77
在一个 GUI 程序中等待.....85
提要.....91

第 4 章 同步控制（Synchronization）.....93
Critical Sections（关键区域、临界区域）.....95
死锁（Deadlock）.....102
哲学家进餐问题（The Dining Philosophers）.....103
互斥器（Mutexes）.....107
信号量（Semaphores）.....115
事件（Event Objects）.....120
从 Worker 线程中显示输出.....124
Interlocked Variables.....125
同步机制摘要.....128

第 5 章 不要让线程成为脱缰野马.....131
干净地终止一个线程.....132
线程优先权（Thread Priority）.....138
初始化一个线程.....144
提要.....146

第 6 章 Overlapped I/O，在你身后变戏法.....149
Win32 文件操作函数.....151
被激发的 File Handles.....155
被激发的 Event 对象.....159
异步过程调用(Asynchronous Procedure Calls，APCs).....163
对文件进行 Overlapped I/O 的缺点.....171
I/O Completion Ports.....172

对 Sockets 使用 Overlapped I/O.....182
提要.....190

第二篇 多线程程序设计的工具与手法

第 7 章 数据一致性（Data Consistency）.....195
认识 volatile 关键字.....196
Referential Integrity.....200
The Readers/Writers Lock.....205
我需要锁定吗?.....214
Lock Granularity（锁定粒度）.....215
提要.....216

第 8 章 使用 C Run-time Library.....219
什么是 C Runtime Library 多线程版本.....220
选择一个多线程版本的 C Runtime Library.....221
以 C Runtime Library 启动线程.....224
哪一个好：CreateThread()抑或 _beginthreadex()?.....227
避免 stdio.h.....237
一个安全的多线程程序.....240
结束进程（Process）.....248
为什么你应该避免 _beginthread().....248
提要.....251

第 9 章 使用 C++.....253
处理有问题的 _beginthreadex()函数原型.....253
以一个 C++ 对象启动一个线程.....256
建立比较安全的 Critical Sections.....265
建立比较安全的 Locks.....268
建立可互换（Interchangeable）的 locks.....270
异常情况（Exceptions）的处理.....274
提要.....274

第 10 章 MFC 中的线程.....277
在 MFC 中启动一个 Worker 线程.....278
安全地使用 AfxBeginThread()的传回值.....282
在 MFC 中启动一个 UI 线程.....288
与 MFC 对象共处.....293
MFC 的同步控制.....296
MFC 对于 MsgWaitForMultipleObjects()的支持.....300
提要.....301

第 11 章 GDI 与窗口管理.....303
线程的消息队列.....304
消息如何周游列国.....306
GUI 效率问题.....311
以 Worker 线程完成多线程版 MDI 程序.....311
多个上层窗口（Top Level Windows）如何是好？.....313
线程之间的通讯.....314
NT 的影子线程（shadow thread）.....316
关于“Cancel”对话框.....316
锁住 GDI 对象.....319
提要.....319

第 12 章 调试.....321
使用 Windows NT.....322
有计划地对付错误.....322
Bench Testing.....323
线程对话框.....324
运转记录（Logging）.....325
内存记号（Memory Trails）.....327
硬件调试寄存器(Hardware Debug Registers).....328
科学方法.....330
提要.....333

第 13 章 进程之间的通讯（Interprocess Communication）.....335
以消息队列权充数据转运中心.....336
使用共享内存（Shared Memory）.....345
使用指针指向共享内存（Shared Memory）.....354
较高层次的进程通讯（IPC）.....362
提要.....364

第 14 章 建造 DLLs.....367
DLL 的通告消息（Notifications）.....369
通告消息（Notifications）的问题.....375
DLL 进入点的依序执行（Serialization）特性.....378
MFC 中的 DLL 通告消息（Notifications）.....379
喂食给 Worker 线程.....380
线程局部存储（Thread Local Storage，TLS）.....384
_declspec(thread).....390
数据的一致性.....392
提要.....393

第三篇 真实世界中的多线程应用程序

第 15 章 规划一个应用程序.....397
多线程的理由.....398
要线程还是要进程？.....403
多线程程序的架构.....404
评估既有程序代码的适用性.....406
对 ODBC 做规划.....411
第三方的函数库（Third-Party Libraries）.....413
提要.....413

第 16 章 ISAPI.....415
Web 服务器及其工作原理.....416
ISAPI.....417

IS2ODBC 范例程序 420
提要 427

第 17 章 OLE，ActiveX，COM 429
COM 的线程模型（COM Threading Models） 431
AUTOINCR 范例程序 437
提要 443

附录 A MTVERIFY 宏 445
附录 B 更多的信息 451

常见问答集

Frequently Asked Questions

FAQ 01：合作型(cooperative)多任务与抢先式(preemptive)多任务有何不同？ 5

FAQ 02：我可以在 Win32s 中使用多个线程吗？ 6

FAQ 03：线程和进程有何不同？ 10

FAQ 04：线程在操作系统中携带多少“行李”？ 11

FAQ 05：Context Switch 是怎么发生的？ 14

FAQ 06：为什么我应该调用 CloseHandle()？ 38

FAQ 07：为什么可以在不结束线程的情况下关闭其 handle？ 40

FAQ 08：如果线程还在运行而我的程序结束了，会怎样？ 47

FAQ 09：什么是 MTVERIFY？ 48

FAQ 10：我如何得知一个核心对象是否处于激发状态？ 74

FAQ 11：什么是一个被激发的对象？ 75

FAQ 12：“激发”对于不同的核心对象有什么不同的意义？ 76

FAQ 13：我如何在主线程中等待一个 handle？ 85

FAQ 14：如果线程在 critical sections 中停很久，会怎样？ 101

FAQ 15：如果线程在 critical sections 中结束，会怎样？ 101

FAQ 16：我如何避免死锁？ 103

FAQ 17：我能够等待一个以上的 critical sections 吗？ 106
FAQ 18：谁才拥有 semaphore？ 118
FAQ 19：Event object 有什么用途？ 120
FAQ 20：如果我对着一个 event 对象调用 PulseEvent()
并且没有线程正在等待，会怎样？ 124
FAQ 21：什么是 overlapped I/O？ 150
FAQ 22：Overlapped I/O 在 Windows 95 上有什么限制？ 150
FAQ 23：我能够以 C runtime library 使用 overlapped I/O 吗？ 152
FAQ 24：Overlapped I/O 总是异步地(asynchronously)执行吗？ 158
FAQ 25：我应该如何为 overlapped I/O 产生一个 event 对象？ 159
FAQ 26：ReadFileEx()和 WriteFileEx()的优点是什么？ 163
FAQ 27：一个 I/O completion routine 何时被调用？ 163
FAQ 28：我如何把一个用户自定义数据传递给 I/O completion routine？ 165
FAQ 29：我如何把 C++ 成员函数当做一个 I/O completion routine？ 170
FAQ 30：在一个高效率服务器(server)上我应该怎么进行 I/O？ 172
FAQ 31：为什么一个 I/O completion ports 是如此特殊？ 175
FAQ 32：一个 I/O completion port 上应该安排多少个线程等待？ 179
FAQ 33：为什么我不应该使用 select()？ 183
FAQ 34：volatile 如何影响编译器的最优化操作？ 198
FAQ 35：什么是 Readers/Writers lock？ 206
FAQ 36：一次应该锁住多少数据？ 215
FAQ 37：我应该使用多线程版本的 C run-time library 吗？ 220
FAQ 38：我如何选择一套适当的 C run-time library？ 221
FAQ 39：我如何使用 _beginthreadex()和 _endthreadex()？ 224
FAQ 40：什么时候我应该使用 _beginthreadex()而非 CreateThread()？ 227
FAQ 41：我如何使用 Console API 取代 stdio.h？ 240

FAQ 42：为什么我不应该使用 _beginthread()? 248
FAQ 43：我如何以一个 C++ 成员函数当做线程起始函数？ 256
FAQ 44：我如何以一个成员函数当做线程起始函数？ 261
FAQ 45：我如何能够阻止一个线程杀掉它自己？ 282
FAQ 46：CWinApp 和主线程之间有什么关系？ 290
FAQ 47：我如何设定 AfxBeginThread()中的 pThreadClass 参数？ 293
FAQ 48：我如何对一个特定的线程调试？ 324
FAQ 49：如果一个新的线程使用了我的 DLL，我如何被告知？ 370
FAQ 50：为什么我在写 DLL 时需要小心所谓的动态链接？ 376
FAQ 51：为什么我在 DllMain 中启动一个线程时必须特别小心？ 379
FAQ 52：我如何在 DLL 中设定一个 thread local storage(TLS)？ 389
FAQ 53：_declspec(thread)的限制是什么？ 392
FAQ 54：我应该在什么时候使用多线程？ 398
FAQ 55：我能够对既有程序代码进行多线程操作吗？ 406
FAQ 56：我可以在我的数据库应用程序中使用多线程吗？ 411

作者序

1992 年，我参加了第二届 Microsoft Professional Developers Conference，当时，Win32 首次对大量观众展示。这是一个重大的事件，其中有许多盅魅的新技术第一次亮相。观众不断地对展示的东西喝彩。当我看到这些新技术时，我身上 Unix 的那一部分说“时候到了”，而 Windows 3.1 的那一部分则是高呼“哈利路亚”。

五年过去了，好几版 Windows NT 问世了。我愈是深入此书，愈是了解 1992 年的那场盛宴之中我的了解是多么贫乏。我听过许多“睿智”的话，比如“为 MDI 程序的每一个窗口准备一个线程”之类，事实上根本是错误的。有些技术（诸如 overlapped I/O）很明显地被曲解了。

当我钻研此书时，我发现许多其他书籍和文章从头到尾描述了各式各样的 Win32 函数，却很少着墨在如何使用它们，或如何搭配其他函数使用。某些函数，例如 MsgWaitForMultipleObjects()，是 Win32 程序的运转中心，却几乎没有任何范例程序正确使用过它。在我所发现的文件给了我太多的挫败之后，我决定尽可能以实务导向的方式完成此书，如此一来，你就能看到那些 Win32 函数一起合作的情景。

这本书的读者群很广，Win16 和 Unix 上的程序开发者欲转移到 Win32，或是有经验的 Win32 程序开发者，都是我的对象。你会发现一些不曾看过的问答，例如“为什么_beginthreadex()真的很重要”等等。程序范例从基本层面到同步机制，到多线程 COM 和 ISAPI 应用程序，都有。

这是一本可以让你上手实验“Win32线程”的书。我描述了基础观念，如果你需要更理论性的知识，你还得多看点其他资料。线程的大部分问题都已经被Dijkstra和Courtois那样的人在25年前就解决掉了，他们的论文直到今天还适用。

如果你对那些只挑软柿子吃而故意忽略困难部分的书籍感到绝望，我希望本书能够让你绝地逢生。某些章节是经过了长时间的实验、多篇相关文章（来自杂志、期刊、Microsoft Knowledge Base）的阅读、众多源代码的剖析之后，萃炼而得。像“线程与C runtime函数库以及MFC的关系”这种主题，我保证你会从中获得许多闪亮的宝石。

许多程序员对于线程是既期待又怕受伤害。Unix的人嘲笑它，因为进程（process）看起来就已经不错了。Win16的人也嘲笑它，因为PeekMessage()也运作得很好嘛！我写这本书时首先认识的一个关键问题就是：如果有任何一个人在乎所谓的线程，他为的是什么？

如果你开发的是服务器（server）产品，你就应该对线程深深在乎，因为I/O completion ports使用它。I/O completion ports是唯一能够搭配Win32 sockets或named pipes完成高效率I/O的方法。请你“跑”到第6章看个明白（用“走”字显得太慢了）。

如果你开发的是Web产品，那么IIS（Internet Information Server）的扩充软件也是靠多线程DLLs完成的。这种技术的背景观念散布于整本书，而第16章则告诉你如何施行那些观念。

本书第一篇的程序是以C完成的，C++的分量很少。从第二篇开始我们就往C++移动了。一如我在第9章所说，C++是大势所趋，不论你是否使用MFC。如果你的C++根基不稳，我希望你特别注意一下第9章。

谁应该读这本书

凡是 C/C++ 程序开发人员，并有 Windows 程序设计经验（不论是 Win16 或 Win32），企图对线程、核心对象、overlapped I/O in Win32 获得更坚实之认识者，本书就是针对你们而写的。本书谈的是 API 函数的使用、你会遭遇的问题，以及 Windows 架构对其用途的影响。

读过本书之后，你将有能力分析哪里是线程可以发挥效用的地方，哪里是你应该躲闪它们的地方。几乎整本书的重点都放在产生真正可用的程序上。神秘手法以及不安全的设计已经被我排除了。

Unix 程序员将会发现，Win32 和 Unix 之间有着基础观念上的差异。

本书架构

本书的第一篇——“上路吧，线程”，为你建立必要的基础，包括线程的启动和结束、核心对象、激发和未激发状态的意义、同步机制及其用途。有经验的 Win32 程序员或许可以跳过这一部分。不过第 6 章所讨论的主题（诸如 I/O completion ports），是非常重要的题目，而且总的来说，其文件非常贫乏。

本书的第二篇，“多线程程序设计的工具与策略”，介绍 C runtime 函数库和 MFC 对线程的支持、如何在 USER 和 GDI 的限制之下施行多线程、如何产生一个 DLL、如何对多线程程序调试。这一部分的许多信息来自于我们的研究和实际经验。

本书的第三篇，“真实世界中的多线程应用程序”，谈论如何组织一个程序，使它有效支持多线程。本篇示范两个真实世界中的应用软件，第一个是个 freethreaded OLE automation server，第二个是 ISAPI 程序，是个 IIS（Internet Information Server）扩充软件，示范如何和 JET 数据库交谈。

关于范例程序

整本书中我用了许多文本模式程序来示范观念。有些人不喜欢这种选择，但我相信一个 50~100 行的程序绝对比一个有着消息循环、资源文件、窗口函数……并且超过 750 行的 Windows 程序更能够集中读者的目光。读这本书你不会看到太多与用户界面相关的东西。我相信本书的范例程序以其目前的形态对你会比较有帮助。

面对这些范例程序，我使用了多方的错误检验。虽然这些检验导致程序代码看起来有些杂乱，但是错误检验对于生产一个真正的应用软件很重要，对于一本书也很重要。

相关阅读

有两样东西是任意时候你尝试写任意 Win32 程序时应该要准备的。第一样东西是 Microsoft Developer Network。其光盘内含不可置信的巨量技术资料，包括 Microsoft Knowledge Base、编译器和 Win32 的完整手册，以及 Microsoft Systems Journal。第二样东西是 Jeffrey Richter 的一本卓越书籍：Advanced Windows NT: The Developers Guide to the Win32 API for Windows NT 3.5 and Windows 95（Microsoft Press，1995）。虽然其中遗漏了某些 NT 3.51 的新东西，但其他方面非常有价值。

译注：Jeffrey 的书已出至第 3 版，名为 Advanced Windows Third Edition。它的小标题写着：for Windows 95 & Windows NT 4.0。

第一篇

上路吧，线程

Threads in Action

第 1 章　为什么要“千头万绪” 3
第 2 章　线程的第一次接触 25
第 3 章　快跑与等待 61
第 4 章　同步控制（Synchronization） 93
第 5 章　不要让线程成为脱缰野马 131
第 6 章　Overlapped I/O，在你身后变戏法 149

为什么要“千头万绪”

Why You Need Multithreading

这一章解释为什么“多线程多任务”是程序开发者与用户都需要的一个重要资产。本章描述重要术语，如 thread 和 context switch，并讨论了 race condition——多线程多任务的祸乱根源。

电脑工业界每有新的技术问世，人们总是不遗余力地去担忧“它是不是够重要”。公司行号虎视眈眈地注意其竞争对手，直到对方采用并宣扬这技术有多么重要，才开始急急赶上。不论这技术是不是真的很重要，每一个人都想尽办法让最终用户感觉“真的很重要”。好啦，于是最终用户真的觉得需要它了——即使他们完全不了解那是什么东西。

“线程”程序设计正处在这个循环的起点。虽然线程在各式各样的操作系统上已经存在了不只十年，但它毕竟还是藉着无孔不入的 Windows 95 和 Windows NT，才能够打进家庭软件和商务应用软件中。

不久的将来，多线程多任务软件将广泛地蔓延开来。线程将成为每一个软件开发者必须使用的标准程序工具。并不是每一个程序都必须使用线程，然而多线程多任务——一如多媒体软件或 Internet 软件所支持的——将使程序的效

率得以高度发挥。线程可以改善用户对于软件操作的感受，简化程序的开发，在同一时间的一台服务器上提供对成百上千用户的支持。用户通常只知晓其结果，他们不知道背后是什么力量促成了这伟大的改良。

单线程程序就像超级市场中唯一的一位出纳员。这个出纳员对于小量采购可以快速结账，但如果有人采买了一大车货品，结账就需要点时间了，其他每一个人都必须等待。

多线程程序像是有一群出纳员，每人负责一条线。某些线专门用来为大买家服务，其他线处理小市民的采买。一条线瘫痪了，并不会影响其他线。

根据这样的宏观印象，下面是一个简单的定义：

多线程，使程序得以将其工作分开，独立运作，不互相影响。

线程并不总是被要求达到这样的目标，不过它们的确使这个目标更容易达成。为了了解线程在什么地方进入程序设计的大版图中，我们最好稍稍知道，自从 MS-DOS 问世到现在，程序员的需求有了些什么样的改变。

一条曲折的路

过去 15 年来，在微软操作系统上工作的程序开发者，花费在程序与程序的合作上的精力愈来愈少。由于用户的需要以及程序体积的增长，操作系统必须负担愈来愈多的任务在“多任务”上头，并且让一切顺利。

MS-DOS

最初是 MS-DOS，其 1.0 版应该几乎已被所有人遗忘。它没有支持磁盘子目录，没有批处理语言（batch language），甚至没有 CONFIG.SYS 和 AUTOEXEC.BAT。它不支持 task 或 process（进程）的观念，程序执行起来便占据了整部机器的控制权。如果你运行 Lotus 1-2-3，你就不能够再做任何其他

事情。既然连进程的概念都没有，它当然不可能是一个多任务（multitasking）甚至多线程（multithreading）的操作系统。

2.x 版有了一些突破，允许所谓的常驻程序（TSR， Terminated and Stay Resident）存在。Sidekick 就是个 TSR，可以在其他程序还在运行时就运行起来。然而，MS-DOS 把 TSR 视为系统的一部分，不是一个应用程序或是一个进程，所以也就没有它应有的权利。可以这么说，TSR 并入应用程序中，但必须和操作系统一起呼吸。也许你还记得那一段为了 TSR 正常运作而艰苦奋战的日子。

但也有连 TSR 都动弹不得的时刻。如果 DOS 正在格式化一张磁盘，任何其他事情，包括 TSR，都必须停止动作。简单地说，即使在今天，MS-DOS 也谈不上多任务，更别提多线程了。

Microsoft Windows

? FAQ 01：合作型 (cooperative) 多任务与抢先式 (preemptive) 多任务有何不同？

Microsoft Windows 的前三个版本（1、2、3）都允许同时执行多个程序，但分享 CPU 是程序（而非操作系统）的责任。如果有一个程序决定咬住 CPU 不放，其他程序就停摆了。因此，我们说 Windows 是“合作型多任务”。在 2.0 和 3.0 那个时代，还是有许多程序拒绝与别人共享资源，慢慢地大家都进步了，也学习到如何写一个“举止良好”的程序。但这还是花费了大家许多宝贵的精力在诸如“调试”这样的工作上。

由于 Windows 的底层依赖 DOS，当格式化一张磁盘或拷贝一个文件到软盘上时，依然让任何其他人都动弹不得。

当此之时，Unix、VMS、AmigaDOS 等操作系统都已经支持一种名为“抢先式多任务”（preemptive multitasking）的模式，意思是操作系统能够强迫应用程序把 CPU 分享给其他人，程序员不需要什么额外的努力。

你知道吗…

Windows 3.x 的多任务

Windows 3.x 的确有支持抢先式多任务，但程序员却不可得之。由于 DOS 程序毫无共享观念，抢先式多任务是让它们不得擅专整部机器的唯一方法，而所有 Windows 程序则被“放在一起，视为单一的 DOS 程序”，所以 Windows 3.x 对 DOS 程序的确是抢先式多任务，但对于 Windows 程序则不是。

OS/2，一个最初由 Microsoft 和 IBM 携手开发，准备用来取代 Windows 的操作系统，拥有许多其他操作系统所拥有的特性。OS/2 是一个 16 位系统，支持内存保护与抢先式多任务。OS/2 的 API（Application Programming Interface）相当程度地与 Windows API 不同，也因此走上了命运坎坷的移植之路。微软终于在 1990 年终止对 OS/2 的拥有权。

Windows NT 的救赎

Windows NT 完全支持 32 位程序的抢先式多任务，把我们带到今天的艺术境界。Windows NT 可以将费时太多的程序接管过去，将执行权给予另一个程序。程序员再不必为了释放控制权而伤脑筋去写 PeekMessage()循环了。 NT 有一组全新的 API，名为 Win32，是 Windows 3.1 API 的超集。

重要！

Win32 API 被移植到 Windows 3.1 上，称为 Win32s。不幸的是许多功能无法移植。Win32s 既不支持抢先式多任务也不支持多线程。本书之中没有任何一个程序可以在 Win32s 上执行，所以我不会再提起它。

? FAQ 02：我可以在 Win32s 中使用多个线程吗？

虽然 16 位程序仍然回溯相容地以合作型多任务方式共同存在，但它们与 32 位程序间的确是抢先式多任务。在 Windows NT 中，一个 16 位程序或甚至一个 DOS 程序都没有办法绝对控制 CPU。

Windows NT 提供了一个与早期 MS-DOS 之间的明显对比。我们已经从一个甚至连进程概念都没有的系统，进入了一个多线程多任务、具有高度安全防护性

能的系统。更令人惊讶的是，你可以对软盘进行任何操作（包括文件拷贝）而不会冲击其他程序。如果你正在执行 Word，你甚至不会知道软盘正在操作。这比任何其他现象都能够解释抢先式多任务的威力。甚至操作系统本身的行为都不能够妨碍程序的执行。

抢先式多任务、多线程，以及 Win32 API 是如此重要，以至于它们也加入了 Windows 4.0——也就是 Windows 95 之中。

你知道吗…

微软曾经卖过 Unix

在 20 世纪 80 年代中期，微软曾经卖过一个 Unix 版本，名为 XENIX。在 OS/2 和 Windows NT 推出的前几年，XENIX 就有了抢先式多任务，但它与 DOS 十分不相容，也不容易学习和维护，所以 XENIX 早已淡出市场。

与线程共枕

从 DOS 到 Windows NT 的演化之路，与应用软件的设计息息相关。例如，旧版的 Microsoft Word 可以同时编辑文字、重编页码，这并不复杂。新版的 Word 还能够同时打印。而 Microsoft Word 95 能够打印、拼写检查、重编页码一起来，当然，此时用户可以继续他的编辑工作。

由于并未使用线程，所以 Word 必须在所有它负责的工作之间大演 CPU 戏法。这和合作型多任务差不多，只不过每件事情都发生在同一程序中罢了。和 Windows 3.x 中的程序一样，Word 必须确保每一个操作都能够顺利执行；也和 DOS 中的程序一样，Word 必须处理它自己的工作切换（task switching）。为了把打印工作切换为重编页码的工作，Word 必须储存打印操作的当时状态，并把重编页码原先的状态恢复回来。储存“操作中的状态”是很困难的，需要很小心地设计。

理想上，我们希望操作系统为我们分担上述这种“切换工作时所需的状态储存与恢复”的责任。这和 Windows NT 解决抢先式多任务时所面临的问题是一样的，只是现在我们必须在程序中来解决它。

线程来了！

为什么最终用户也需要多线程多任务

虽然让程序多线程多任务不失为一个酷点子，但是让管理阶层了解多线程会带来什么好处，恐怕才是更重要的事。下面是一些若非线程则无以解决的例子。

如果你曾经尝试使用 Windows NT 3.x 的 File Manager 连接一个忙碌的服务器，我想你会知道，File Manager 基本上会悬在那儿一动也不动，直到服务器完成所有操作。程序员绝对没办法改善这种局面，因为操作停滞在操作系统里面。如果它停滞在你的程序中，你可以放一个 PeekMessage()循环。但是对于操作系统，你没有太多的控制能力。因此，如果文件操作需要一段长时间的话，控制权不可能立刻传回到用户手上。

如果 File Manager 有多个线程，那么当一个线程必须停下来等待与服务器连接时，其他线程还是可以继续动作的，你的 UI（User Interface）也因此持续有反应。一个永远有反应的 UI 是很重要的。

这项技术已经实现于 Windows 95 的桌面（desktop）。如果你从某文件夹拷贝文件到另一文件夹，这中间你还可以再打开并关闭桌面上的文件夹和文件。事实上，你可以同时进行多项拷贝操作，于是你会获得多个窗口，每个窗口呈现拷贝操作的进度（图 1-1）。由于大部分工作由操作系统完成，桌面（desktop）不可能在没有多线程的情况下完成这样的绝技。

Windows 95 桌面 (desktop) 的多线程多任务还不是很完美。一旦你开始拷贝文件到某个文件夹中，那么在拷贝结束之前，你没有办法对该文件夹进行任何

操作。这说明线程并不是万灵丹。程序还是应该“适当地”使用线程，而只有那些被程序员特别青睐并设计的东西，才能够做并行处理。

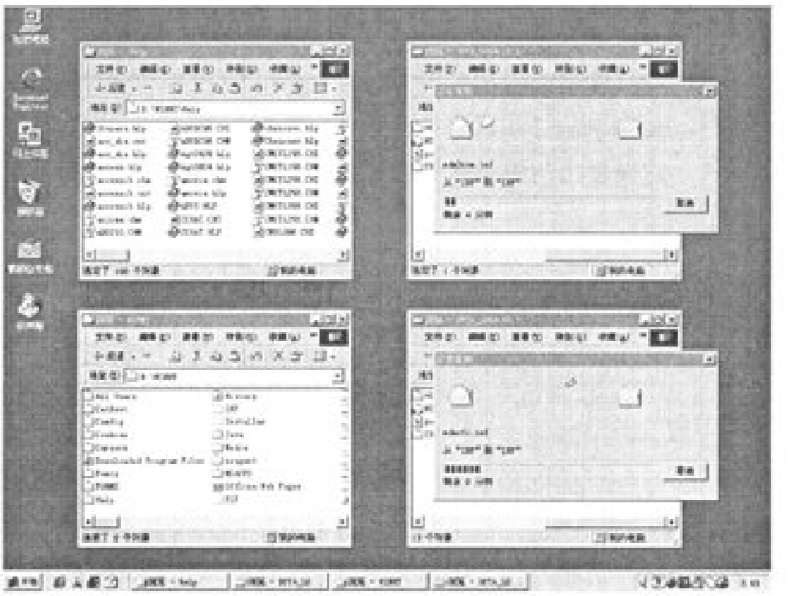


图 1-1 Explorer（资源管理器）执行多项拷贝操作

任何使用光驱的程序也会感受到多线程多任务的好处。光驱驱动程序的搜寻时间大约是 200 毫秒（ms）左右。Pentium CPU 大约每秒执行一千万个指令，200 毫秒的等待表示有许多能量都浪费掉了。这一段 CPU 时间可以用来更新屏幕、将 wave 文件混音、准备缓冲区或进行任何其他活动。使用多个线程可以去除许多长久以来存在于光驱和网络上的“瓶颈”。第 6 章我们就可以看到该怎么做。

Win32 基础

? FAQ 03：线程和进程有何不同？

观察线程如何运作之前，让我们先确定进程（process）和线程（thread）到底在 Win32 中代表什么意义。我将给你一个非常高阶的概观，观照低阶的行为。这里提出的解释非常地概括化，而其实现……呃，当然，你晓得，要更复杂得多。

进程（Processes）

从 Win32 的角度来看，进程含有内存和资源。被进程拥有的内存，理论上可以高达 2GB。资源则包括核心对象（如 file handles 和线程）、USER 资源（如对话框和字符串）、GDI 资源（如 Device Context 和 brushes）。

进程就像一本活页笔记夹，你可以在其中的活页上写东西，也可以擦掉内容或甚至整页撕掉，活页笔记夹只是持有那些东西而已。同理，进程本身并不能够执行，它只是提供一个安置内存和线程的地方（译注）。

译注 Matt Pietrek 在其 Windows 95 System Programming SECRETS（Windows 95 系统程序设计大奥秘/侯俊杰译/旗标出版）一书中的解释是："进程就是一大堆对象的拥有权的集合。也就是说，进程拥有对象。进程可以拥有内存（更精确地说是拥有 memory context），可以拥有 file handles，可以拥有线程，可以拥有一大串 DLL 模块（被载入这一进程的地址空间中）。"请见原著#102 页，译本#103 页。

UNIX 上述定义与其他操作系统如 Unix 的定义有些不同。在 Unix 中，进程与其主线程是相同的东西。在许多支持线程（或是所谓 lightweight processes）的 Unix 版本中，线程的幻觉其实是由 runtime library 产生的，操作系统毫无所悉。

内存

每一个进程都关系到内存。内存就像是前面所说的活页笔记夹中的活页纸，它代表的意义完全得看纸面上写些什么而定。内存可以大致分为三种类型：

- Code
- Data
- Stack

Code 是程序的可执行部分，一定是只读（read only）性质。这是 CPU 唯一允许执行的内存。可执行 Windows NT 的两种芯片：Intel 芯片和 RISC 芯片都有这项限制。

Data 是你的程序中的所有变量（不包括函数中的局部变量），可以区分为全局变量和静态变量两种。当然线程也可以使用 malloc()或 new 动态配置内存。

Stack 是你调用函数时所用的堆栈空间，其中有局部变量。每个线程产生时配有一个堆栈。如果不需要，操作系统会将它动态扩充。

所有这些内存对进程中的所有线程都是可用的。这在多线程程序中虽然带来很大的方便，却也带来很大的灾难。

线程（Threads）

进程和内存并没有真正“做”什么事情。一旦 CPU 开始执行程序代码，你就有了一个“线程”。在同一时间同一个进程，你就可以拥有一大把线程，执行同一段代码。

? FAQ 04：线程在操作系统中携带多少“行李”？

定义一个线程，需要的数据并不多。线程在“任何时刻下的状态”被定义在进程的某块内存中，以及 CPU 中的可涂改拍纸片上。其他的重要数据，如变量以及应用程序的调用堆栈，储存在进程中那些可被其他线程共享的内存内。

我所谓“CPU的可涂改拍纸片”指的是CPU寄存器。在大部分CPU中这些寄存器既不是内存的一部分，也不可能使用一个指针指向它。Intel Pentium内含通用（general-purpose）寄存器（EAX、EBX、ECX、EDX）、堆栈指针（ESP），以及指令指针（EIP）。只要检验这些寄存器，操作系统就能够知道线程在任何时刻的工作状态。

你知道吗…

Win32操作系统

没有所谓的“纯”Win32操作系统。对Windows NT而言，Win32 API只是可以同时存在于Windows NT kernel中的“多重人格”的一重（译注）。Windows 95内部则坚实依赖16位技术，其Win32 API只是16位技术的一层包装。

译注 我想作者所谓的“多重人格”，指的是NT的受保护子系统（protected subsystem），其中包括Win32子系统、POSIX子系统、OS/2子系统。

为什么不使用多个进程？

人们一旦接触到线程，他们最常问的问题就是：有什么是线程能够给我而多进程（multi-processes）所不能给我的？言下之意就是，线程又怎么样！最重要的答案便是：线程价廉。线程启动比较快，退出比较快，对系统资源的冲击也比较小。而且，线程彼此分享了大部分核心对象（如file handles）的拥有权。

如果使用多重进程，最困难的问题大概是如何把窗口的handle交给另一个进程。在Win32中，handle只在其诞生地（进程中）才有意义。这是一种安全警戒，避免某个进程危及另一个进程的资源。为了分享窗口handle，你必须明明白白地产生该handle的一个副本，并且可以被其他进程使用。在一个多线程程序中，所有线程都可以使用这个窗口handle，因为handle和线程生活在同一个进程之中。

即使完全不需要窗口，也有另外全然不同的问题。例如，服务器软件似乎

应该是多进程的爱好者。毕竟在同一个服务器上每一个用户做的事情不尽相同，不是吗？想一想 WWW 服务器吧。一个 Web 服务器可能每天得为百万人次服务，同一时间可能涌进数百人。然而 Web 服务器是“stateless”，用户不需要登录到 Web 服务器，只需发出特别的请求（requests）。除了少数例外，进来的每一个请求完全与其他的请求无关联。而且，每一个请求通常只带有小量数据。

虽然，为每一个请求产生一个新的进程也很容易，但其额外负担（overhead）非常惊人：必须载入服务器软件的一个全新副本，配置大量内存并初始化。新的这个副本必须将状态调整到与原先状态相同。数秒钟可能就这样过去了——在这么一个忙碌的服务器上。新的进程现在准备好为这个请求的 8KB 数据服务，而任务完成后每一样东西又必须拉下马来。为了搬移这 8KB 数据，有太多工作要做。操作系统内部可能必须操作好几 MB 的内存才能为此请求服务。

如果你以单一线程为所有的请求服务，你就无法获得硬件最佳使用率。如果有许多个线程，就可以令某些线程等待系统资源，如网络和硬盘。操作系统可以很有效率地处理这些请求，因为它可以在任何时刻知道谁正在等待什么，并决定怎样调整最好。如果只有单一线程，要让硬件充分发挥性能就很困难了。如果线程必须做任意大面积的计算，所有其他的请求都必须等待！

在客户/服务器（client/server）环境中的另一个极端，是所谓的工作组数据库服务器（workgroup database server）。可能会有一些用户对它产生连接（connection）并持续数小时之久。这种情况下产生个别的进程以服务个别的用户，其额外负担（overhead）无足道哉。不幸的是，每个人还是使用同一个数据库，所以进程之间还是必须商议，看谁可以读一笔数据或写一笔数据，或改变一个索引。这和我们面对线程时遭遇的问题一样，只不过现在更复杂，因为要共享的是跨越进程的数据结构。如果使用线程，则程序的所有内存全由其线程共享。

UNIX Unix 产生一个进程所需的额外负担（overhead）比在 Win32 中低得多。然而，Win32 产生一个线程，代价又更低廉得多。一个单 CPU 的 Windows NT 机器上拥有 500 个线程并不是太过分的事，而大部分 Unix 如果有 500 个进程就会有严重的效率问题，甚至即使其中大部分没有活动（non-active）。

Context Switching

? FAQ 05: Context Switch 是怎样发生的?

在一个抢先式多任务系统中，操作系统小心地确保每一个线程都有机会执行。它依赖硬件的协助以及许多的簿记工作。当硬件计时器认为某个线程已经执行够久了，就会发出一个中断（interrupt），于是 CPU 取得目前这个线程的当前状态，也就是把所有寄存器内容拷贝到堆栈之中，再把它从堆栈拷贝到一个 CONTEXT 结构（这样便储存了线程的状态）中，以备以后再用。

要切换不同的线程，操作系统应先切换该线程所隶属之进程的内存（译注），然后恢复该线程放在 CONTEXT 结构中的寄存器值。这整个过程便称为 context switch。

译注 也就是换一套 memory context —— page directory 和 page tables。关于这一点，Matt Pietrek 在其 Windows 95 System Programming SECRETS（Windows 95 系统程序设计大奥秘/侯俊杰译）一书有极详细的说明。请见原著 #303 页，译本 #305 页。

如果第二个线程属于不同进程所有，则这两个进程没有办法共享任何内存。这种隔离策略可以保护进程免受其他人突如其来的伤害。最诡异的是两个进程以为它们在相同的地址上运行。两个进程的 0x1f000 指针事实上指向不同的实际内存。因此，不同进程间的线程如果要通讯，唯有依赖特别的设

计，使之拥有共享内存（shared memory）．如果两个线程属于同一进程，它们将共享所有的内存．

Context Switch 可能在一秒钟之内发生数百次．这样快速的切换将给你造成幻觉，以为电脑真的在同一时间做了许多事情．图 1-2 显示了在一部闲置的 Windows NT 上，每秒所发生的 context switches．此图显示大约在 296~1126 次之间．

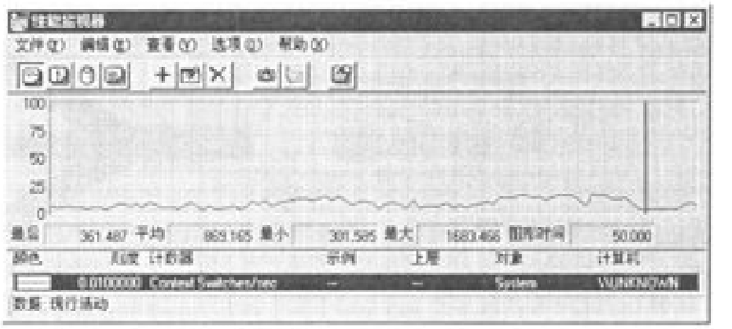


图 1-2 Windows NT workstation 4.0 上的 context switches

Context Switch 的效率

每次 context switch 都要缴点效率税金．如果你的程序有 500 个线程，你要缴的税金就比较多．

如果两个线程都在运行，并企图计算圆周率 PI 到小数点后百万位，它们将比单纯一个线程所跑速度的一半再慢一点．这是因为 CPU 花费有限的时间做 context switch．因此“两个线程同时计算 PI”所花费的时间比“一个线程接连做两次一样的工作”所花费的时间长一点．

从用户的观点来看，抢先式多任务使电脑看起来像是能够同时处理许多任务．而实际上 CPU 一次只能做一件事情．当然，你也可能买一部有多个 CPU 的电脑．CPU 愈多，就有愈多线程可以同时执行，不需要 context switch．

Windows NT 以其 Symmetric Multi-Processing（SMP）技术支持多 CPU 机器。每一个 CPU 执行一个不同的线程。任何线程可能在任何时间被任何一个 CPU 执行。一部双 CPU 系统可以同时执行同一进程（或不同进程）的两个线程。在一部 SMP 系统中，多个线程真的有可能被同时调用。图 1-3 显示了一个 SMP 系统将 #3 线程排给 #1 CPU 去执行。

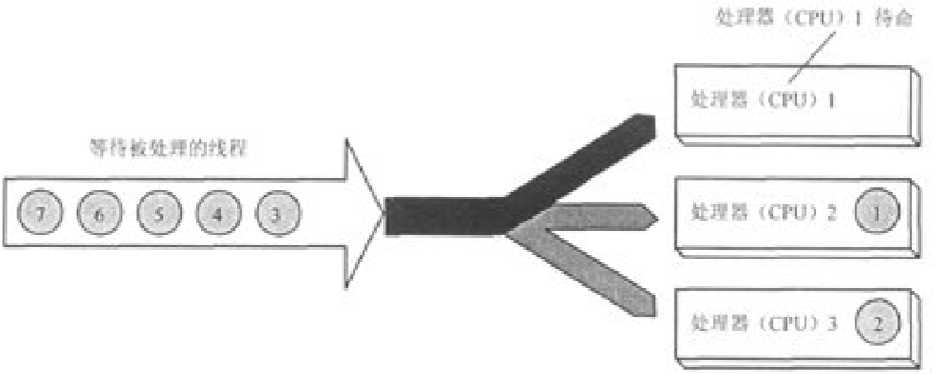


图 1-3 SMP（Symmetric Multi-Processing）

Race Conditions（竞争条件）

Context switching 是抢先式多任务的心脏。在一个合作型多任务系统中，操作系统必须得到程序的允许才能够改变线程（译注），但是在抢先式多任务系统中，控制权被强制移转，也因此两个线程之间的执行次序变得不可预期，这不可预期性便造成了所谓的 race conditions（竞争条件）。

译注 你可能会奇怪，在合作型多任务系统如 Windows 3.x 中，只有进程，哪有什么线程。作者在这里的意思是指该进程即为其主线程（而且也就这么一个线程）。

定义 race condition 的最好做法就是给个实例。假设你正在一个文件服务器中编辑一串电话号码，文件打开来后看起来像这样：

```
Joe       555-3765
Elaine    555-9300
Tom       555-2040
```

现在你为 Karl 加上一笔新数据。正当你输入 Karl 的电话号码的时候，另一个人也打开文件并输入 Ted 的数据。你们都储存了你们的输入。

问题是，谁的数据会留下来？答案当然是最后存档的那个人，而前一个人的输入会被覆盖掉。这两个人面临的就是 race condition。

以链表（Linked List）为例

相同的问题也发生在程序代码身上。假设你的函数加一笔数据到链表之中：

```
struct Node
{
   struct Node *next;
   int data;
};

struct List
{
   struct Node *head;
};

void AddHead(struct List *list, struct Node *node)
{
   node->next = list->head;
   list->head = node;
}
```

简单而直截了当，不必多说什么。现在让我们假设，链表一开始有一个节点，如图 1-4 所示。

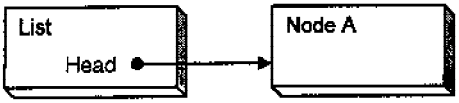


图 1-4 链表拥有一个节点

现在线程 #1 调用此函数，准备加一个节点 B，而 context switch 却发生于 AddHead()的某两行之间，我们看看会发生什么事。此时链表的状态将如图 1-5 所示。

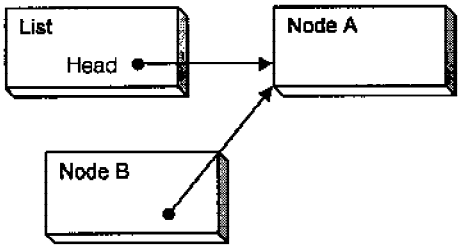


图 1-5 链表在 context switch 发生之后的状态

现在线程 #2 尝试加上节点 C，而且成功地完成了任务，如图 1-6 所示。

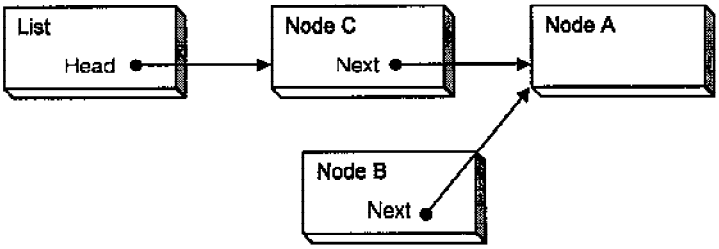


图 1-6 链表在线程 #2 结束后的状态

最后又轮到线程 #1，继续其未完成的工作，于是把链表头设到节点 B 去了。节点 C 于是被切断，或许就此成了一个内存泄漏（memory leak）。由于抢先式多任务之故，线程 #1 没有办法了解某些状况在线程回返之前已经改变了，因而留下图 1-7 所示的迷乱。

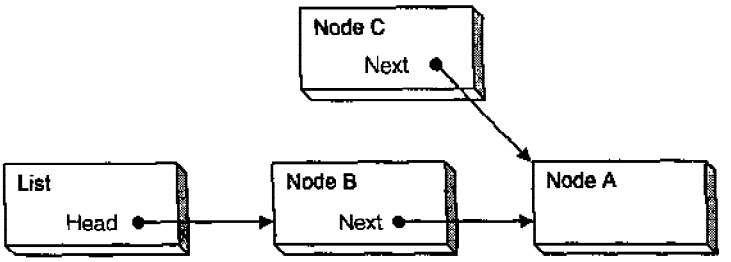


图 1-7 线程 #1 完成后的受损链表

或许这种情况的发生只有百万分之一的机率，或许不值得大惊小怪。然而别忘了你的 CPU 是以每秒两千万到五千万个指令的速度在运转，对于一个常常被使用的链表而言，race condition 的发生可能是一天多起。

Atomic Operations（原子操作）

严密计划以阻止 race condition 的发生是有必要的。困难点在于鉴定出 race condition 的发生地点。长时间在高级语言如 C/C++ 之中打滚的你，或许已经忘了 CPU 是在非常低阶中操作。当你思考 race condition 时，如果忽略“低阶”这档子事，可能会生成难以去除的“臭虫”喔！

以上题为例，最简单的解决办法似乎就是在程序代码中放一个标记，以便观察是否能够安全地继续下去。

```
int flag;

AddHead(struct List *list, struct Node *node)
{
    while(flag != 0)
        ;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

现在，我们看看产生出来的汇编语言代码（assembly code），如表 1-1 所列。

列表 1-1 AddHead()所产生的汇编语言代码（assembly code）

```
#0001  ;       : {
#0002
#0003                xor  eax, eax
#0004  $L86:
#0005
#0006  ;       :     while(flag != 0)
#0007
#0008          cmp   DWORD PTR _flag, eax
#0009          jne   SHORT $L86
#0010
#0011  ;       :         ;
#0012  ;       :     flag = 1;
#0013
#0014          mov   eax, DWORD PTR _list$[esp-4]
#0015          mov   ecx, DWORD PTR _node$[esp-4]
#0016          mov   DWORD PTR _flag, 1
#0017
#0018  ;       :     node->next = list->head;
#0019
#0020          mov   edx, DWORD PTR [eax]
#0021          mov   DWORD PTR [ecx], edx
#0022
#0023  ;       :     list->head = node;
#0024
```

```
#0025          mov    DWORD PTR [eax], ecx
#0026
#0027  ;       :      flag = 0;
#0028
#0029          mov    DWORD PTR _flag, 0
#0030
#0031  ;       : }
#0032
#0033          ret    0
```

重要！

> 简简单单的几行 C 指令竟然扩张为那么多的机器指令！所以千万不要认为 C 的一个指令一定可以安全执行完毕，而不在乎 context switch 是否发生。

上述函数有数种可能会导致失败。如果 context switch 发生于 #8 和 #9 两行之间，另一个线程即可进入该函数——即使该函数设立了标记。此外，如果 context switch 发生于 #20 或 #21 行之后，这个函数也可能会失败。

一个操作（operation）如果能够不受中断地完成，我们称之为 atomic operation。上例中检查标记和设立标记的动作必须是一个 atomic operation，如果它被中断，就会产生一个 race condition。

检查标记和设立标记的动作是如此地平常，所以被实现于许多 CPU 硬件中，名为 Test 和 Set 指令。实际上只有操作系统会使用 Test 和 Set 指令。操作系统提供比较高阶的机制，让你的程序用以取代 Test 和 Set。

本书第一篇的焦点放在 race conditions 可能发生之事态的管理。第二篇及第三篇中我将讨论其他技术。

线程之间如何通讯

一个和 race conditions 很有关系的问题就是：线程通讯问题。一个线程若要有大用途，你必须告诉它做什么事情。Win32 提供了一个简单的方法，供应线程的启动（start-up）信息。我们将在第 2 章看到这一点。但是两个线程间的交谈就十分棘手了。线程怎么知道数据要放在哪里？

最明显的答案就是放在全局变量中，因为所有线程都可以读它写它。但如果线程 A 写下一个全局变量，线程 B 如何知道数据在那里？当线程 B 读了数据，线程 A 如何知道线程 B 已经取走？如果线程 A 必须在线程 B 取走数据之前写入更多数据，怎么办？

如果线程 B 结束了而线程 A 正等着它将数据取走，又怎么办？这个问题更复杂，因为线程可能不断地被产生和被摧毁。

简短地说，线程之间的通讯问题可能非常棘手。本书第二篇我们将看到如何经由各式各样的系统机制解决这些问题。

好消息与坏消息

使用线程并非没有代价。你该不会认为天下有白吃的午餐吧。使用线程可能会引发数个潜在性的严重问题。除非经过小心设计，多线程程序有着不可预期、测试困难的倾向。

如果你曾经在厨房和别人共事过，你知道，有时候额外的协助反而带来更多的麻烦。如果你们都做同一件事情，你就必须不断和另一个说话，找出什么已经做了什么还没做。调味料要不要搅一搅？锅子该不该上炉了？如果两个人做完全不同的事情，情况会比较好一些，例如一个人准备蔬菜而另一个人拌沙拉。即使这样还是会有问题发生，因为刀子有限，碗有限，连冰箱的空间都有

限．你们必须先有计划，了解谁用哪个器具哪个碗．有时候你必须等待，如果你要的东西正被使用．

写多线程程序有点像那个样子．如果你有许多个线程都处理同一件事情，那么将很容易出错，除非那些工作被很小心地安排过．如果你的线程分别处理不同的事情，它们还是需要共享一些数据结构，或对话框，或文件，或其他什么东西．成功的秘诀是小心地做计划：谁要什么？何时要？怎么要？

线程的第一次接触

Getting a Feel for Threads

本章介绍一些用以产生、监视、退出线程的 Win32 函数调用，同时也展示一些线程运转过程给你看，并示范一个后台打印程序。这一章也介绍了 MTVERIFY，一个用来帮助对本书范例程序进行调试的宏。

如果你问我，在 Win16 环境中实现一个后台打印程序的感想，我只有一句话回答：真受不了！我想你会发现，同样的问题在 Win32 中有了比较容易的解决方式。

要在 Win32 环境中实现出后台打印，你必须学习如何产生一个线程，如何结束一个线程。我会示范给你看，让你知道多线程程序的行为，因为多线程程序有着让人大吃一惊的行为倾向。

Console Applications

在这一章以及本书的许多地方，我将使用 console 程序为载体，介绍 SDK 函数的用法，最后再把每一样技术合并到一个 Windows 程序中。

Console 程序是一种文字模式的程序，看起来像一个 DOS 窗口。与 Windows 3.x 和 16 位 Windows 程序不同的一点是，console 程序充分支持 printf()。很少有人知道 32 位 Windows 程序也可以有文字窗口！我们将利用这一个性质帮助说明程序的行为。Console 程序可以从 File Manager（文件管理器）或 Explorer（资源管理器）或 DOS 提示符下启动。如有必要，立刻就会产生一个文字窗口。

产生一个线程

产生一个线程（并因而成就一个多线程程序），是以 CreateThread()作为一切行动的开始。此函数的原型如下。（译注：可从 WINBASE.H 获得）

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    DWORD dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

参数

lpThreadAttributes	描述施行于这一新线程的 security 属性。NULL 表示使用缺省值。此参数在 Windows 95 中被忽略。本书并不论及 security。
dwStackSize	新线程拥有自己的堆栈。0 表示使用缺省大小：1MB。

lpStartAddress	新线程将开始的起始地址。这是一个函数指针。(译注：在 C 语言中函数名称即代表函数指针，所以这里可放一个函数名称)
lpParameter	此值将被传送到上述所指定之新线程函数去，作为参数。
dwCreationFlags	允许你产生一个暂时挂起的线程。默认情况是“立即开始执行”。
lpThreadId	新线程的 ID 会被传回到这里。

返回值

如果 CreateThread( )成功，传回一个 handle，代表新线程。否则传回一个 FALSE。如果失败，你可以调用 GetLastError( )获知原因。

重要！

> 如果你不需要线程 ID，lpThreadId 参数可以被设为 NULL。但这只在 Windows NT 中才行得通，Windows 95 中不行。如果你要写一个程序可以在 Windows 95 中运行，不要把 lpThreadId 参数设为 NULL。

这个函数的运作方式可能会让你在领悟过程中遇到一些棘手的地方，因为它不像任何其他传统程序。为了提供一个良好的起点，我将列出两种对比情况，一种使用 CreateThread( )，另一种不使用 CreateThread( )。列表 2-1 显示“单纯的函数调用”和“启动线程”两种情况的执行结果，其中的号码标示出函数的调用次序。

列表 2-1 “单纯的函数调用”与“使用多个线程”之间的对比

标准函数调用	启动线程

#include <stdio.h>
 	
int square(int n);
 	
void main( )
{
  int result; (1)
 	
  result = square(5); (2)
 	
  printf("%d\n", result); (5)
}
 	
int square(int n)
{
  int product = n * n; (3)
  return product; (4)
}
	#include <windows.h>
	
	DWORD WINAPI ThreadFunc(LPVOID);
	
	void main( )
	{
	  HANDLE hThread; (1)
	  DWORD threadId;
	
	  hThread = CreateThread(NULL,
	  0,
	  ThreadFunc,
	  0,
	  0,
	  &threadId); (2)
	
	  printf("Thread running"); (4)
	}
	
	DWORD WINAPI ThreadFunc(LPVOID p)
	{
	  // ... (4)
	}

面对一个函数调用操作，控制权会转移到被调用函数中，执行完毕后再返回原调用处。为了印出平方值，square( )必须回到 main( )。

产生一个线程，情况与上述十分类似，但是有点曲折。前例（列表 2-1）左手边我们直接调用 square( )，而右手边的多线程范例中，我们不直接调用 ThreadFunc( )，我们调用的是 CreateThread( )并交给它 ThreadFunc( )地址。CreateThread( )开启一个新的线程，该线程调用 ThreadFunc( )。而原来的线

程继续前进。换句话说，ThreadFunc( )被异步地（asynchronously）执行了，意思是 ThreadFunc( )不需要在 main( )继续前进之前完成。所以，返回值也就不可能像传统方式那样。

并不是 ThreadFunc( )这个函数有什么魔力。你的线程起始函数可以任意命名，甚至命名为 Fred( )亦无不可。其间的关系完全靠你交给 CreateThread( )的函数地址而建立。

请注意列表 2-1 中有两行标记为(4)。它们大体上在同一时间被执行，但真正的顺序无法预期。一旦 ThreadFunc( )启动，它就完全独立于原调用端了。

为了尝试对于底层工作更有感觉，我们来看一个真正做事情的程序。列表 2-2 启动 5 个线程并且分别交给它们参数 0~4。每一个线程打印其参数 10 次，然后结束。

列表 2-2 NUMBERS.C——一个真正运转的多线程程序

```
#0001  /*
#0002   * Numbers.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 2, Listing 2-1
#0006   *
#0007   * Starts five threads and gives visible feedback
#0008   * of these threads running by printing a number
#0009   * passed in from the primary thread.
#0010   *
#0011   */
#0012
#0013  #define WIN32_LEAN_AND_MEAN
#0014  #include <stdio.h>
#0015  #include <stdlib.h>
#0016  #include <windows.h>
#0017
#0018  DWORD WINAPI ThreadFunc(LPVOID);
#0019
#0020  int main( )
```

```
#0021 {
#0022     HANDLE hThrd;
#0023     DWORD threadId;
#0024     int i;
#0025
#0026     for (i=0; i<5; i++)
#0027     {
#0028         hThrd = CreateThread(NULL,
#0029             0,
#0030             ThreadFunc,
#0031             (LPVOID)i,
#0032             0,
#0033             &threadId );
#0034         if (hThrd)
#0035         {
#0036             printf("Thread launched %d\n", i);
#0037         }
#0038     }
#0039     // Wait for the threads to complete.
#0040     // We'll see a better way of doing this later.
#0041     Sleep(2000);
#0042
#0043     return EXIT_SUCCESS;
#0044 }
#0045
#0046 DWORD WINAPI ThreadFunc(LPVOID n)
#0047 {
#0048     int i;
#0049     for (i=0;i<10;i++)
#0050         printf("%d%d%d%d%d%d%d%d%d%d\n",n,n,n,n,n,n,n,n,n,n);
#0051     return 0;
#0052 }
```

请你特别注意 ThreadFunc( )的函数定义：

```
DWORD WINAPI ThreadFunc(LPVOID n)
```

CreateThread( )的函数声明中期望第三个参数是个函数指针，指向某种特定类型的函数：返回值为 DWORD，调用约定是 WINAPI（译注），有一个

LPVOID 参数。换句话说，如果你的线程函数不符合这些要求，编译时便会出现错误信息。一开始就精准地声明正确的函数类型，比日后才强制类型转换（cast）好。不要强制类型转换，编译器就会进行适当的类型检验工作，并让你知道你的声明是否正确。举个例子，如果你忘了在函数声明中写“WINAPI”，而又强制转换类型，线程一启动就会坠毁，因为出现“调用约定（calling convention）”错误。

译注 除了 c、pascal、stdcall 之外，什么时候又来一个 WINAPI 调用约定（calling convention）呢？噢，WINAPI 在 WINDEF.H 中被定义为：

```
#define WINAPI    __stdcall
```

我想你会注意到，程序列表一开始就有一个 ThreadFunc( )函数原型，这么一来你就可以使用它作为 CreateThread( )的参数了。

使用多个线程的结果

图 2-1 显示这一程序在历经一些时间后的执行结果。左边栏位“正常的函数调用”所显示的是：如果我们连续五次直接调用 ThreadFunc( )所可能期望的结果。当然这样的调用操作并没有出现在列表 2-2 中。

另两栏显示的是真正的执行结果。第一次执行时，结果显示于屏幕上。第二次执行时，我把结果导引到文件。图 2-1 涵盖了多线程程序的一些最重要的观念。

正常的函数调用	执行结果一(输出到屏幕)	执行结果二(输出到文件)
Calling Function	Thread launched	Thread launched 0
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000
00000000	00000000	00000000

正常的函数调用	执行结果一(输出到屏幕)	执行结果二(输出到文件)
00000000	00000000	00000000
00000000	Thread launched	00000000
00000000	11111111	00000000
00000000	11111111	00000000
00000000	11111111	00000000
00000000	11111111	00000000
Calling Function	11111111	Thread launched 1
11111111	22222222	11111111
11111111	22222222	11111111
11111111	22222222	11111111
11111111	22222222	11111111
11111111	Thread launched	11111111
11111111	Thread launched	11111111
11111111	00000000	11111111
11111111	00000000	11111111
11111111	00000000	11111111
11111111	00000000	11111111
Calling Function	11111111	Thread launched 2
22222222	11111111	Thread launched 3
22222222	11111111	Thread launched 4
22222222	11111111	22222222
22222222	11111111	22222222
22222222	22222222	22222222
22222222	22222222	22222222
22222222	22222222	22222222
22222222	22222222	22222222
22222222	33333333	22222222
22222222	33333333	22222222
Calling Function	33333333	22222222
33333333	33333333	22222222

正常的函数调用	执行结果一(输出到屏幕)	执行结果二(输出到文件)
33333333	33333333	44444444
33333333	Thread launched	44444444
33333333	44444444	44444444
33333333	44444444	44444444
33333333	00000000	44444444
33333333	22222222	44444444
33333333	22222222	44444444
33333333	33333333	44444444
33333333	333344444444	44444444
Calling Function	44444444	44444444
44444444	44444444	33333333
44444444	44444444	33333333
44444444	3333	33333333
44444444	33333333	33333333
44444444	33333333	33333333
44444444	33333333	33333333
44444444	44444444	33333333
44444444	44444444	33333333
44444444	44444444	33333333
44444444	44444444	33333333

图 2-1 NUMBERS 的执行结果

多线程程序无法预期

第一点，多线程程序是无法预测其行为的。你看到了，第二栏显示的结果不同于第三栏。每一次执行 NUMBERS，你会获得不同的结果。

许多程序员耗费他们的青春，期望电脑运算的结果能够精准地重复呈现。但是在多线程程序中，其结果视 CPU 的速度、线程的工作、CPU 的忙碌程度等等因素……非常非常多的因素……而不同。

这种无可预期性导致一个很特别的结果，那就是，如果程序完蛋了，很难重复其过程。本书的重要目标就是教你如何从一个多线程程序中获得可预期的结果。

执行次序无法保证

第二个重点就是，线程的执行顺序无法保证。在第三栏中，你看到线程 #4 结束于线程 #3 启动之前——甚至即使线程 #3 较早诞生。是的，线程彼此之间的执行顺序应该视之为随机。

Task Switches 可能在任何时刻任何地点发生

如果你仔细观察第二栏，你会看到这一行：

```
333344444444
```

这一行之所以有不同的数字杂处，是因为 context switch 发生于线程正在显示其结果之时。这种行为就是典型的 race condition。Race condition 的结果端视哪一个线程先完成其操作。

在上述那行之下，你可以看到线程 #3 被允许继续执行，完成其未竟事业：

```
3333
```

像这样的问题是可以解决的，只要你在编译时使用 /MT 或 /MD 选项，表示要使用多线程版本的 C runtime library（详见第 8 章）。多线程版函数库确保你的输出不会像此例这样被中断并混合。本例是为了突显多线程多任务的许多潜在性问题，所以没有使用多线程版 C runtime 函数库。

UNIX

Unix Line Mode

这个问题绝不会发生在 line mode（cbreak）的 Unix 程序中。由于 Windows 并不是被设计为在终端机上运行，Windows 程序的输出不会先被放到一个以一行一行为根据的缓冲区中。如果你曾经在 Unix 环境下尝试过这样的命令：

```
ls & ; ls &
```

则这两个命令的输出会以一行一行为根据而混合，但是一行之中绝不会内含其他程序的输出。

线程对于小的改变有高度的敏感

或许你已注意到了，本例各个线程的输出，第三栏结果比第二栏更容易分组归类。这是因为第三栏重导到文件，而第二栏显示于屏幕。重导至文件的那些输出远比屏幕上的输出快得多。

这样的差异告诉我们，程序可能只因为一点小小的不同——甚至不是程序本身的因素——而结果互异。在抢先式多任务调度中，每个线程被许以一个固定量的时间片（称为 timeslice），然后控制权就会被转移。由于“屏幕显示”是个慢动作，线程在一份 timeslice 中所能处理的行数也就比较少。不过，虽然程序的执行次序有所改变，但每一份工作都还是能够圆满完成。

由于多线程程序如此敏感，程序代码的小改变，也可能引发戏剧性的变化。可谓差之毫厘，失之千里。如果你把下面这些代码放到 ThreadFunc()中的 printf()之后，输出结果会比第二栏和第三栏更加随机不定。

```
int j; for (j=0; j<100000; j++) ;
```

虽然你大概不会放置一个像这样的延迟循环于你的程序中，但其效果和任何长时间计算或代价高昂之函数调用是一样的。这样的敏感度可能引起你不曾

预期的问题．曾经有过光辉历史的 printf( )夹杀调试法，可能会完全改变多线程程序的行为．

线程并不总是立刻启动

在第二栏和第三栏中，线程 #0 和 #1 一被 main( )产生，就立刻开始打印其结果．这样的行为可是无法保证的唷．在第三栏中，线程 #2、#3、#4 被产生后并没有立刻打印．稍后我会详细讨论这个问题．

核心对象（Kernel Objects）

CreateThread( )传回两个值，用以识别一个新的线程．第一个值是个 HANDLE，这也是 CreateThread( )的返回值，大部分与线程有关的 API 函数都需要它．第二个值是由 lpThreadId 带回来的线程 ID．线程 ID 是一个全局变量，可以独一无二地表示系统任一进程中的某个线程．AttachThreadInput( )和 PostThreadMessage( )就需要用到线程 ID，这两个函数允许你影响其他人（线程）的消息队列．调试器和进程观察器也需要线程 ID．为了安全防护的缘故，你不可能根据一个线程的 ID 而获得其 handle．

CreateThread( )传回来的 handle 被称为一个核心对象（kernel object）．核心对象其实和所谓的 GDI 对象，如画笔、画刷或 DC 是差不多的，只不过它由 KERNEL32.DLL 管理，而非 GDI32.DLL 管理．两种对象之间有许多相似性．

GDI 对象是 Windows 的基础部分．在 Win16 或 Win32 中它们都是由操作系统管理．通常你不需要知道其数据格式．例如，你可能会调用 SelectObject( )或 ReleaseObject( )以处理 GDI 对象；Windows 隐藏了实现细节，只是给你一个 HDC 或一个 HBRUSH，那都是对象的 handle．

核心对象以 HANDLE 为使用时的参考依据。与 GDI 的 HBRUSH、HPEN、HPALETTE 以及其他种 handles 不同的是，只有一种 handle 可以代表核心对象。所谓 handle，其实是个指针，指向操作系统内存空间中的某样东西，那东西不允许你直接取得。你的程序不能够直接取用它，为的是维护系统的完整性与安全性。

下面是各种 Win32 核心对象的清单。本书涵盖 pipes 之外的每一种核心对象。

- 进程（processes）
- 线程（threads）
- 文件（files）
- 事件（events）
- 信号量（semaphores）
- 互斥器（mutexes）
- 管道（Pipes。分为 named 和 anonymous 两种）

我将在第 4 章谈到事件、信号量、互斥器。这些核心对象用来整合许多的线程或进程，使它们在大合奏中弹奏出美妙的音符。

GDI 对象和核心对象之间有一个主要的不同。GDI 对象有单一拥有者，不是进程就是线程。核心对象可以有一个以上的拥有者，甚至可以跨进程。为了保持对每一位主人（拥有者）的追踪，核心对象保持了一个引用计数（reference count），以记录有多少 handles 对应到此对象。对象中也记录了哪一个进程或线程是拥有者。如果你调用 CreateThread()或是其他会传回 handle 的函数，引用计数便累加 1。当你调用 CloseHandle()时，引用计数便递减 1。稍后你便会看到 CloseHandle 函数。一旦引用计数降至 0，这一核心对象即自动被摧毁。

面对一个打开的对象，区分其拥有者是进程或是线程，是件很重要的事情。因为这会决定系统何时做清除善后（cleans up）操作。所谓 cleanup 操作，包括将该进程或线程所拥有的每一个对象的引用计数减 1。若有必要，则对象会被摧毁掉。程序员不能选择由进程或线程拥有对象，一切得视对象类型而定。

由于引用计数的设计，对象有可能在“产生该对象之进程”结束之后还继续幸存。Win32 提供各种机制，让其他进程得以取得一个核心对象的 handle。如果某个进程握有某个核心对象的 handle，而该对象的原创者（进程）已经“作古”了，此核心对象并不会被摧毁。

CloseHandle 的重要性

当你完成你的工作后，应该调用 CloseHandle 释放核心对象。

```
BOOL CloseHandle (
    HANDLE hObject
);
```

参数

hObject	代表一个已打开之对象 handle

返回值

如果成功，CloseHandle( )传回 TRUE。如果失败则传回 FALSE，此时你可以调用 GetLastError( )获知失败原因。

? FAQ 06：为什么我应该调用 CloseHandle( )?

如果一个进程没有在结束之前针对它所打开的核心对象调用 CloseHandle( )，操作系统会自动把那些对象的引用计数下降 1。虽然你可以依赖系统做实体（physical）上的清除（cleanup）工作，然而逻辑上的清除工作又是完全不同的一回事，特别是如果你有许多个进程的话，因为系统并不知道对象实际代表什么意义，所以它不可能知道解构顺序是否重要。

如果一个进程常常产生“worker 线程”（译注）而老是不关闭线程的 handle，那么这个进程可能最终有数百甚至数千个开启的“线程核心对象”留给操作系统去清理。这样的资源泄漏（resource leaks）可能会对效率带来负面的影响。

译注 所谓 worker 线程，是指完全不牵扯到图形用户界面（GUI），纯粹做运算的线程。请看本章最后“后台打印”一节中的“The Microsoft Threading Model”。

还有一件事也很重要：你不可以依赖“因线程的结束而清理所有被这一线程产生的核心对象”。许多对象，例如文件，是被进程拥有，而非被线程拥有。在进程结束之前不能够清理它们。

为了更正确一些，需要时列表 2-2 中的 NUMBERS 程序做一点修改。下面是修改后的 main( )函数，这一次调用 CloseHandle( ):

```
#0001  int main( )
#0002  {
#0003      HANDLE hThrd;
#0004      DWORD threadId;
#0005      int i;
#0006
#0007      for (i=0; i<5; i++)
#0008      {
#0009          hThrd = CreateThread(NULL,
#0010              0,
#0011              ThreadFunc,
#0012              (LPVOID)i,
#0013              0,
#0014              &threadId );
#0015          if (hThrd)
#0016          {
#0017              printf("Thread launched %d\n", i);
#0018              CloseHandle(hThrd);
#0019          }
#0020      }
#0021      // Wait for the threads to complete.
#0022      // We'll see a better way of doing this later.
#0023      Sleep(2000);
#0024
#0025      return EXIT_SUCCESS;
#0026  }
```

线程对象与线程的不同

? FAQ 07：为什么可以在不结束线程的情况下关闭其handle？

线程的 handle 是指向“线程核心对象”，而不是指向线程本身。对大部分 API 而言，这项差异没什么影响。当你调用 CloseHandle( )并给予它一个线程 handle 时，你只不过是表示，你希望自己和此核心对象不再有任何瓜葛。CloseHandle( )唯一做的事情就是把引用计数减 1。如果该值变成 0，对象会自动被操作系统摧毁。

“线程核心对象”引用到的那个线程也会令核心对象开启。因此，线程对象的默认引用计数是 2。当你调用 CloseHandle( )时，引用计数下降 1，当线程结束时，引用计数再降 1。只有当两件事情都发生了（不管顺序如何）的时候，这个对象才会被真正清除。

“引用计数”机制保证新的线程有个地方可以写下其返回值。这样的机制也保证旧线程能够读取那个返回值——只要它没有调用 CloseHandle( )。

由于被 CreateThread( )传回的那个 handle 属进程所有，而非线程所有，所以很可能有一个新产生的线程调用 CloseHandle( )，取代原来的线程。Microsoft Visual C++ runtime library 中的 _beginthread( )就是这么做的。请看第 8 章的讨论。

线程结束代码（Exit Code）

我在列表 2-2 的 NUMBERS 程序中放了一个 Sleep( )于 main( )的最后，给予线程有从容的时间结束。理想上我们希望确定线程真的完成了，并且我们也检验了其结束代码，然后才让程序结束掉。如果 CPU 很忙碌，有可能 Sleep( )返回时那些线程尚未结束。

Windows NT 的安装程序就是一个例子。当主线程从光盘中拷贝文件到硬盘，第二线程也同时为你产生一个紧急开机磁片：把硬盘文件拷贝到软盘。不知道谁先完成，所以主线程必须确定第二线程完成后才能够继续下去。否则要不是软盘，要不就是硬盘，没有完成其拷贝操作。

线程结束代码可以藉着调用 GetExitCodeThread( )（并给予 CreateThread 所获得的线程 ID 作为参数）而得知：

```
BOOL GetExitCodeThread(
    HANDLE hThread,
    LPDWORD lpExitCode
);
```

参数

hThread	由 CreateThread( )传回的线程 handle
lpExitCode	指向一个 DWORD，用以接受结束代码（exit code）

返回值

如果成功，GetExitCodeThread( )传回 TRUE，否则传回 FALSE．如果失败，你可以调用 GetLastError( )找出原因．如果线程已结束，那么线程的结束代码会被放在 lpExitCode 参数中带回来．如果线程尚未结束，lpExitCode 带回来的值是 STILL_ACTIVE．

列表 2-3 的程序启动两个线程，然后要求你按任意键结束之．当你按下任意键，它会检查是否两个线程确实都结束了．如果都结束了，程序便会显示其结束代码，并结束主线程．否则你会被告知哪一个线程还在运转。

虽然这一范例程序有点不自然，但本章稍后的后台打印程序真的也有相同问题．当一个后台线程正在打印，如果用户选按【File/Exit】，会发生什么事？以本章的层次，将只是单纯地拒绝【File/Exit】，直到线程完成其任务为止。

列表 2-3 EXITCODE.C——示范 GetExitCodeThread( )的用法

```
#0001  /*
#0002   * ExitCode.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 2, Listing 2-2
#0006   *
#0007   * Start two threads and try to exit
#0008   * when the user presses a key.
#0009   */
#0010
#0011  #define WIN32_LEAN_AND_MEAN
#0012  #include <stdio.h>
#0013  #include <stdlib.h>
#0014  #include <windows.h>
#0015  #include <conio.h>
#0016
#0017  DWORD WINAPI ThreadFunc(LPVOID);
#0018
#0019  int main( )
#0020  {
#0021      HANDLE hThrd1;
#0022      HANDLE hThrd2;
#0023      DWORD exitCode1 = 0;
#0024      DWORD exitCode2 = 0;
#0025      DWORD threadId;
#0026
#0027      hThrd1 = CreateThread(NULL,
#0028          0,
#0029          ThreadFunc,
#0030          (LPVOID)1,
#0031          0,
#0032          &threadId );
#0033      if (hThrd1)
#0034          printf("Thread 1 launched\n");
#0035
#0036      hThrd2 = CreateThread(NULL,
#0037          0,
#0038          ThreadFunc,
#0039          (LPVOID)2,
```

```
#0040         0,
#0041         &threadId );
#0042     if (hThrd2)
#0043         printf("Thread 2 launched\n");
#0044
#0045
#0046     // Keep waiting until both calls to
#0047     // GetExitCodeThread succeed AND
#0048     // neither of them returns STILL_ACTIVE.
#0049     // This method is not optimal - we'll
#0050     // see the correct way in Chapter 3.
#0051     for (;;)
#0052     {
#0053         printf("Press any key to exit..\n");
#0054         getch( );
#0055
#0056         GetExitCodeThread(hThrd1, &exitCode1);
#0057         GetExitCodeThread(hThrd2, &exitCode2);
#0058         if ( exitCode1 == STILL_ACTIVE )
#0059             puts("Thread 1 is still running!");
#0060         if ( exitCode2 == STILL_ACTIVE )
#0061             puts("Thread 2 is still running!");
#0062         if ( exitCode1 != STILL_ACTIVE
#0063             && exitCode2 != STILL_ACTIVE )
#0064             break;
#0065     }
#0066
#0067     CloseHandle(hThrd1);
#0068     CloseHandle(hThrd2);
#0069
#0070     printf("Thread 1 returned %d\n", exitCode1);
#0071     printf("Thread 2 returned %d\n", exitCode2);
#0072
#0073     return EXIT_SUCCESS;
#0074 }
#0075
#0076
#0077 /*
#0078  * Take the startup value, do some simple math on it,
#0079  * and return the calculated value.
#0080  */
```

```
#0081  DWORD WINAPI ThreadFunc(LPVOID n)
#0082  {
#0083      Sleep((DWORD)n*1000*2);
#0084      return (DWORD)n * 10;
#0085  }
```

程序输出：

```
Thread 1 launched
Thread 2 launched
Press any key to exit..
Thread 1 is still running!
Thread 2 is still running!
Press any key to exit..
Thread 2 is still running!
Press any key to exit..
Thread 1 returned 10
Thread 2 returned 20
```

GetExitCodeThread()将传回线程函数（本例为 ThreadFunc）的返回值。然而，GetExitCodeThread()的一个糟糕行为是，当线程还在进行，尚未有所谓结束代码时，它会传回 TRUE 表示成功。如果这样，lpExitCode 指向的内存区域中应该放的是 STILL_ACTIVE。你必须小心这种行为，也就是说你不可能从其返回值中知道“到底是线程还在运行呢，还是它已结束，但返回值为 STILL_ACTIVE”。

重要！ 在这一章中，我将使用 GetExitCodeThread( )等待一个线程的结束，然而这并不是好方法。第 3 章有正确的方法。

结束一个线程

前两个程序，是靠着线程函数的结束而结束线程。有时候可能需要用更强制性的手法结束一个线程。你可以使用 ExitThread( )。

```
VOID ExitThread(
    DWORD dwExitCode
);
```

参数

dwExitCode	指定此线程之结束代码

返回值

没有，此函数从不返回。

这个函数有点像 C runtime library 中的 exit( )函数，因为它可以在任何时候被调用并且绝不会返回。任何代码若放在此行之下，保证不会被执行。

列表 2-4 示范了 ExitThread( )。其中的线程总是有一个返回值为 4，因为 AnotherFunc( )从不返回。

列表 2-4 EXITTHRD.C——示范 ExitThread( )

```
#0001  /*
#0002   * ExitThrd.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 2, Listing 2-3
#0006   *
#0007   * Demonstrate ExitThread
#0008   */
#0009
#0010  #define WIN32_LEAN_AND_MEAN
```

```
#0011 #include <stdio.h>
#0012 #include <stdlib.h>
#0013 #include <windows.h>
#0014
#0015 DWORD WINAPI ThreadFunc(LPVOID);
#0016 void AnotherFunc(void);
#0017
#0018 int main( )
#0019 {
#0020     HANDLE hThrd;
#0021     DWORD exitCode = 0;
#0022     DWORD threadId;
#0023
#0024     hThrd = CreateThread(NULL,
#0025         0,
#0026         ThreadFunc,
#0027         (LPVOID)1,
#0028         0,
#0029         &threadId );
#0030     if (hThrd)
#0031         printf("Thread launched\n");
#0032
#0033     for(;;)
#0034     {
#0035         BOOL rc;
#0036         rc = GetExitCodeThread(hThrd, &exitCode);
#0037         if (rc && exitCode != STILL_ACTIVE)
#0038             break;
#0039     }
#0040
#0041     CloseHandle(hThrd);
#0042
#0043     printf("Thread returned %d\n", exitCode);
#0044
#0045     return EXIT_SUCCESS;
#0046 }
#0047
#0048
#0049 /*
#0050  * Call a function to do something that terminates
#0051  * the thread with ExitThread instead of returning.
```

```
#0052  */
#0053  DWORD WINAPI ThreadFunc(LPVOID n)
#0054  {
#0055     printf("Thread running\n");
#0056     AnotherFunc( );
#0057     return 0;
#0058  }
#0059
#0060  void AnotherFunc( )
#0061  {
#0062     printf("About to exit thread\n");
#0063     ExitThread(4);
#0064     // It is impossible to get here, this line
#0065     // will never be printed
#0066     printf("This will never print\n");
#0067  }
```

程序输出：

```
Thread launched
Thread running
About to exit thread
Thread returned 4
```

结束主线程

? FAQ 08：如果线程还在运行而我的程序结束了，会怎样？

程序启动后就执行的那个线程称为主线程（primary thread）。主线程有两个特点。第一，它必须负责 GUI（Graphic User Interface）程序中的主消息循环。第二，这一线程的结束（不论是因为返回或因为调用了 ExitThread()）会使得程序中的所有线程都被强迫结束，程序也因此而结束。其他线程没有机会做清理工作。

如果你没有在列表 2-2 的 NUMBERS 程序的最后放一个 Sleep()，程序会只印出一部分的数据，然后就停止。C runtime library 造成这样的结果，那是由于“当 main() 返回因而结束主线程”时，它们当时的行为所致。这个行为将在第 8 章有比较详细的讨论。此刻请记住一点，在 main( )或

WinMain( )结束（返回，return）之前，总是先等待所有的线程都结束。第3章会告诉你如何适当地完成这一任务。

在主线程中调用ExitThread( )也可以，那会导致主线程结束而“worker线程”继续存在。然而这么做会跳过runtime library中的清理（cleanup）函数，因而没有将已开启的文件清理掉。我不建议你这么做。

错误处理

在跳到后台打印之前，值得花点时间看看Win32如何处理错误。特别是在Windows NT，几乎任何事情一旦出错，都可以调用GetLastError而获得描述。经验显示，线程的各种相关函数是错误高危险群，而适当的错误处理可以阻止挫败并产生一个比较可信赖的程序。

? FAQ 09：什么是MTVERIFY?

我将在本书使用一个名为MTVERIFY的宏，既适用于GUI程序也适用于console程序。这个宏内部其实是记录并解释Win32 GetLastError( )的结果。如果Win32函数失败，MTVERIFY( )会打印出一段简短的文字说明。附录A有MTVERIFY宏的源代码与解说。

本书这些范例程序都已经被仔细检查过了，但是当你尝试修改这些程序时，你还是会发现错误检验的无上价值。

列表2-5示范如何使用MTVERIFY，以及它的效果。这个程序类似上一个程序，但是对CloseHandle的调用操作已经被移到GetExitCodeThread的上方，为的是导致GetExitCodeThread的失败。

列表 2-5 · ERROR.C——示范错误处理函数

```
#0001  /*
#0002   * Error.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 2, Listing 2-4
#0006   *
#0007   * Demonstrate ExitThread
#0008   */
#0009
#0010  #define WIN32_LEAN_AND_MEAN
#0011  #include <stdio.h>
#0012  #include <stdlib.h>
#0013  #include <windows.h>
#0014  #include "MtVerify.h"
#0015
#0016  DWORD WINAPI ThreadFunc(LPVOID);
#0017
#0018  int main( )
#0019  {
#0020      HANDLE hThrd;
#0021      DWORD exitCode = 0;
#0022      DWORD threadId;
#0023
#0024      MTVERIFY( hThrd = CreateThread(NULL,
#0025          0,
#0026          ThreadFunc,
#0027          (LPVOID)1,
#0028          0,
#0029          &threadId )
#0030      );
#0031      if (hThrd)
#0032          printf("Thread launched\n");
#0033
#0034      MTVERIFY( CloseHandle(hThrd) );
#0035
#0036      for(;;)
#0037      {
#0038          BOOL rc;
```

```
#0039         MTVERIFY( rc = GetExitCodeThread(hThrd, &exitCode) );
#0040         if ( rc && exitCode != STILL_ACTIVE )
#0041            break;
#0042      }
#0043
#0044      printf("Thread returned %d\n", exitCode);
#0045
#0046      return EXIT_SUCCESS;
#0047   }
#0048
#0049
#0050   DWORD WINAPI ThreadFunc(LPVOID n)
#0051   {
#0052      printf("Thread running\n");
#0053      return 0;
#0054   }
```

程序输出：

```
Thread launched
Thread running
The following call failed at line 38 in Error.c:
    rc = GetExitCodeThread(hThread, &exitCode)

Reason: The handle is invalid.
```

后台打印（Background Printing）

在所有外围设备中，打印机大概是速度最慢的。喷墨打印机或点阵打印机大约要花数分钟才能印好一页。高速激光打印机则大约每分钟可打印 24 页。但是离每分钟可以搬移将近 300 MB 数据的硬盘速度，则有天壤之别。

后台打印是如此地诱惑人心，于是刺激了 MS-DOS 2.0 版的第一个多任务常驻程序(译注：我想作者是指 PRINT.COM)。Windows 的 Print Manager（打印

管理器）负责在程序完成其对打印数据的处理后，接手打印机事务。但程序光是准备那些打印用的数据，也着实需要好几分钟呢！如果使用多个线程，我们就可以让一个线程负责处理打印数据，另一个线程负责控制用户界面。

译注 关于 Windows 打印过程，与打印管理器之间的搭配，以及打印管理器与打印机的搭配，请参考 Charles Petzold 的 Programming Windows 95 第 15 章："Using the Printer"，尤其是原著 #786 页和 #787 页的两张图。

根据三个已经学过的线程 API 函数，现在你已经有足够的能力写出一个小型的后台打印程序。

The Microsoft Threading Model（微软的多线程模型）

Win32 说明文件一再强调线程分为 GUI 线程和 worker 线程两种。GUI 线程负责建造窗口以及处理主消息循环。worker 负责执行纯粹运算工作，如重新计算或重新编页等等，它们会导致主线程的消息队列失去反应。一般而言，GUI 线程绝不会去做那些不能够马上完成的工作。

GUI 线程的定义是：拥有消息队列的线程。任何一个特定窗口的消息总是被产生这一窗口的线程抓到并处理。所有对此窗口的改变也都应该由该线程完成。

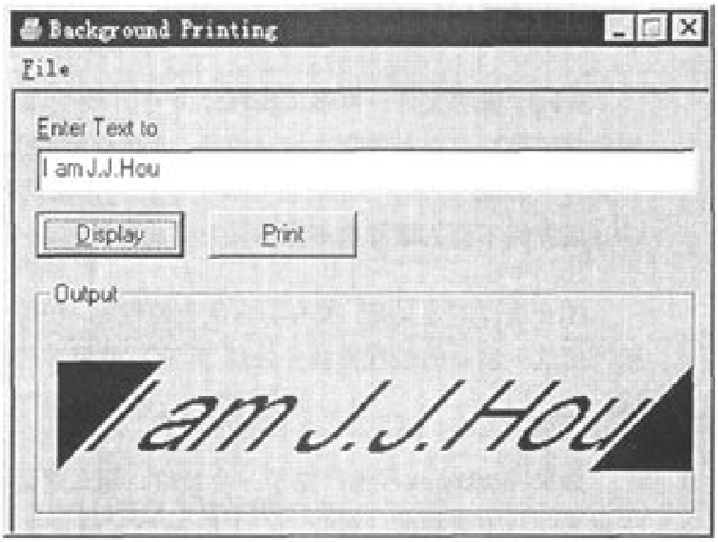
如果 worker 线程也产生了一个窗口，那么就会有一个消息队列随之被产生出来并且附着到此线程身上，于是 worker 线程摇身一变成了 GUI 线程。这里的意思是，worker 线程不能够产生窗口、对话框、消息框，或任何其他与 UI 有关的东西。

如果一个 worker 线程需要输入或输出错误信息，它应该授权给 UI 线程来做，并且将结果通知给 worker 线程。我们将在第 11 章看到相关的详细讨论。

“打印”范例程序说明

你可以在光盘中找到一个程序，名为 BACKPRNT．当程序启动时，你会看到一个 edit 控件和一个 menu．在 edit 控件中输入一些文字，然后按下【Display】或【Print】．文字会被转化为斜体图形，然后重新显示于窗口中或输出到打印机，视你按下哪一个按钮而定．如果你选的是【Print】，你将总是会获得一个“打印对话框”．

译注 以下是 BACKPRNT 的执行画面。



只要 worker 线程被启动，这个程序就可以随时准备让你输入新的信息．你输入多少字符串，程序就产生多少 worker 线程．

“打印”范例程序架构

你或许已经从这一章中嗅出多线程程序设计困难重重的味道。这个例子可以带给你一个与本书其他较大程序一致的设计目标：

简单和安全，更甚于复杂和速度！

在整本书中，我企图告诉你如何在线程之间以最低表面积来设计程序。所谓表面积，意指必须被线程共享的数据结构。要知道，程序愈是密切地与线程有关系，愈是容易产生错误并发生 race condition。

为了符合“最低密度”这个目标，BACKPRNT 的主线程负责把 worker 线程需要的所有信息捆绑在一起，然后产生 worker 线程，起始函数是 PrintText( )，显示于列表 2-6 中。

列表 2-6 节录自 BACKPRNT 的 PrintText( )函数内容

```
#0001  //
#0002  // Asks user which printer to use, then creates
#0003  // background printing thread.
#0004  //
#0005  HANDLE PrintText(HWND hwndParent, char *pszText)
#0006  {
#0007      ThreadPrintInfo *pInfo;
#0008      HANDLE hThread;
#0009      DWORD dwThreadId;
#0010      int result;
#0011      DOCINFO docInfo;
#0012
#0013      PRINTDLG dlgPrint;
#0014
#0015      // Put up Common Dialog for Printing and get hDC.
#0016      memset(&dlgPrint, 0, sizeof(PRINTDLG));
#0017      dlgPrint.lStructSize = sizeof(PRINTDLG);
#0018      dlgPrint.hwndOwner = hwndParent;
#0019      dlgPrint.Flags = PD_ALLPAGES | PD_USEDEVMODECOPIES
#0020            | PD_NOPAGENUMS | PD_NOSELECTION | PD_RETURNDC;
```

```
#0021     dlgPrint.hInstance = hInst;
#0022     if (!PrintDlg(&dlgPrint))
#0023        return;
#0024
#0025     // Initialize Printer device
#0026     docInfo.cbSize = sizeof(DOCINFO);
#0027     docInfo.lpszDocName = "Background Printing Example";
#0028     docInfo.lpszOutput = NULL;
#0029     docInfo.lpszDatatype = NULL;
#0030     docInfo.fwType = 0;
#0031     result = StartDoc(dlgPrint.hDC, &docInfo);
#0032     result = StartPage(dlgPrint.hDC);
#0033
#0034     pInfo = HeapAlloc(GetProcessHeap( ),
#0035                  HEAP_ZERO_MEMORY,
#0036                  sizeof(ThreadPrintInfo));
#0037     pInfo->hDlg = hwndParent;
#0038     pInfo->hWndParent = hwndParent;
#0039     pInfo->hDc = dlgPrint.hDC;
#0040     pInfo->bPrint = TRUE;
#0041     strcpy(pInfo->szText, pszText);
#0042
#0043     MTVERIFY( hThread = CreateThread(NULL, 0,
#0044        BackgroundPrintThread, (LPVOID)pInfo,
#0045        0, &dwThreadId ));
#0046
#0047     // keep track of all background printing threads
#0048     gPrintJobs[gNumPrinting++] = hThread;
#0049
#0050     return hThread;
#0051  }
```

译注 在光盘的源代码中，列表 2-6 的 #0005 是：

```
#0005  void PrintText(HWND hwndParent, char *pszText)
```

而且没有这一行：

```
#0050     return hThread;
```

以下是这个函数的控制流程：

1. 显示“打印对话框”。务必在主线程中进行此事，否则如果用户快速地选按【Print】数次，每一个 worker 线程会在几乎同一时间显示自己的“打印对话框”。这可不是什么好事情。

2. 在 heap 中产生一个数据块。如果使用全局变量，容易被改写；如果使用堆栈变量，又容易超过生存范围（out of scope）。唯一能够把数据安全交给线程的地方，就是 heap。

3. 将数据块初始化，并复制一份，输入字符串到其中。如果我们让 worker 线程直接从 edit 控件中读取数据，很有可能用户在 worker 线程尚未完成其启动动作前，又改变了字符串的内容。这是一种 race condition。

4. 启动线程。

5. 储存线程的 handle。我们需要这个 handle，才能确保线程尚在运转时用户不得结束程序。

为了让主线程取得原本准备给 worker 线程的数据，我们必须确保每件事按正确的次序发生，而 worker 线程也因此收到最新的输入字符串。当 worker 线程启动后，它并不需要任何来自对话框的信息，或全局变量。

BACKPRNT 保持对每一个 worker 线程的追踪。如果用户企图离开程序，GUI 线程会先等待所有的 worker 线程都结束。下面的代码完成这个等待操作：

```
#0001  for (index = 0; index < gNumPrinting; index++)
#0002  {
#0003      DWORD status;
#0004      do
#0005      {   // Wait for thread to terminate
#0006                  MTVERIFY(GetExitCodeThread(gPrintJobs[index],
&status));
#0007          Sleep(10);
#0008      } while (status == STILL_ACTIVE);
#0009
#0010  } // end for
```

译注 上面 #0006 行在光盘源代码中为：

```
#0006          GetExitCodeThread(gPrintJobs[index], &status);
```

这并不是非常好的等待操作。你将在下一章看到其他几个比较好的做法。然而这里已经告诉你了，你必须对执行中的线程负起责任。如果你没有等待就离开，线程还是会结束，但你的打印结果恐怕不如预期的好。

后台线程

后台线程和主线程完全分开。它使用不同的数据结构、不同的 DC，并且是非交谈式的。任何东西如果要被画在屏幕上，都必须送出一个消息给主线程（GUI 线程），告诉它去做这件事情。打印机 DC（printer Device Context）可以被直接写入数据，因为它是特别为此线程而产生的。后台线程的程序代码显示于列表 2-7 中。

列表 2-7 节录自 BACKPRNT 的 BackgroundPrintThread( )函数内容

```
#0001  DWORD WINAPI BackgroundPrintThread(LPVOID pVoid)
#0002  {
#0003      ThreadPrintInfo *pInfo = (ThreadPrintInfo*) pVoid;
#0004      RECT rect;
#0005      RECT rectMem;
#0006      HDC hDcMem;
#0007      HBITMAP bmpMem;
#0008      HBITMAP bmpOld;
#0009      int x, y;
#0010      int counter = 0;
#0011      int nHeight;
#0012      HFONT hFont;
#0013      HFONT hFontOld;
#0014
#0015      // Get dimensions of paper into rect
#0016      rect.left = 0;
#0017      rect.top = 0;
#0018      rect.right =  GetDeviceCaps(pInfo->hDc, HORZRES);
#0019      rect.bottom = GetDeviceCaps(pInfo->hDc, VERTRES);
#0020
```

```
#0021    nHeight = -MulDiv(36, GetDeviceCaps(pInfo->hDc, LOGPIXELSY), 72);
#0022
#0023    // Create Font
#0024    hFont = CreateFont(nHeight, 0,
#0025       0, 0, FW_DONTCARE,
#0026       FALSE, FALSE, FALSE,
#0027       ANSI_CHARSET,
#0028       OUT_TT_PRECIS,
#0029       CLIP_DEFAULT_PRECIS,
#0030       PROOF_QUALITY,
#0031       VARIABLE_PITCH,
#0032       NULL);
#0033    MTASSERT( hFont != 0);
#0034
#0035    // Draw into memory device context
#0036    hDcMem = CreateCompatibleDC(pInfo->hDc);
#0037    hFontOld = SelectObject(hDcMem, hFont);
#0038    iHeight = DrawText(hDcMem, pInfo->szText, -1,  &rect,
           DT_LEFT | DT_NOPREFIX | DT_WORDBREAK | DT_CALCRECT);
#0039    rectMem = rect;
#0040    rectMem.left = rect.left + iHeight;
#0041    rectMem.right = rect.right + (iHeight*2);
#0042    bmpMem = CreateCompatibleBitmap(hDcMem,
#0043                                rectMem.right, rect.bottom);
#0044    bmpOld = SelectObject(hDcMem, bmpMem);
#0045    OffsetRect(&rect, iHeight, 0);
#0046    DrawText(hDcMem, pInfo->szText, -1,  &rect,
#0047           DT_LEFT | DT_NOPREFIX | DT_WORDBREAK);
#0048
#0049    // Italicize bitmap. We use GetPixel and
#0050    // SetPixel because they are horribly inefficient,
#0051    // thereby causing the thread to run for awhile.
#0052    for (y = 0; y < iHeight; y++)
#0053    {   // Italicize line y
#0054       for (x = rectMem.right; x > iHeight; x--)
#0055       {   // Move specified pixel to the right.
#0056          COLORREF color;
#0057          int offset;
#0058          offset = y - iHeight;
#0059          color = GetPixel(hDcMem, x + offset, y);
#0060          if (color != 0)
#0061             counter++;
```

```
#0062            SetPixel(hDcMem, x, y, color);
#0063        } // end for x
#0064    } // end for y
#0065    MTASSERT( counter > 0);
#0066
#0067    // Copy bitmap of italicized text from memory to device
#0068    if (pInfo->bPrint)
#0069    {
#0070        BitBlt(pInfo->hDc, 50, 50, rectMem.right-rect.left,
             rectMem.bottom-rect.top,
#0071            hDcMem, iHeight, 0, SRCCOPY);
#0072    }
#0073
#0074    SelectObject(hDcMem, hFontOld);
#0075    SelectObject(hDcMem, bmpOld);
#0076    DeleteDC(hDcMem);
#0077
#0078    if (!pInfo->bPrint)
#0079    {
#0080        // We can't just write to the global variable where the
#0081        // bitmap is kept or we might overwrite the work of
#0082        // another thread, thereby "losing" a bitmap
#0083
#0084        // Also, if we used PostMessage instead of SendMessage, then
#0085        // the rectangle could have been deleted (it's on the stack)
#0086        // by the time the main message loop is reached.
#0087        SendMessage(pInfo->hDlg, WM_SHOWBITMAP, (WPARAM)&rectMem,
             (LPARAM) bmpMem);
#0088    }
#0089
#0090    if (pInfo->bPrint)
#0091    {   // Finish printing
#0092        int result;
#0093
#0094        result = EndPage(pInfo->hDc);
#0095        MTASSERT (result != SP_ERROR);
#0096        result = EndDoc(pInfo->hDc);
#0097        MTASSERT (result != SP_ERROR);
#0098        DeleteDC(pInfo->hDc);
#0099        // If we are printing, we are done with the bitmap.
#0100        DeleteObject(bmpMem);
```

```
#0101      }
#0102      else
#0103      {
#0104         ReleaseDC(pInfo->hWndParent, pInfo->hDc);
#0105      }
#0106
#0107      // free data structure passed in.
#0108      HeapFree(GetProcessHeap( ), 0, pInfo);
#0109
#0110      return 0;
#0111   }
```

成功的秘诀

即使这么简单的程序，还是可以揭示出多线程程序设计的成功关键：

1. 各线程的数据要分离开来，避免使用全局变量。
2. 不要在线程之间共享 GDI 对象。
3. 确定你知道你的线程状态。不要径自结束程序而不等待它们的结束。
4. 让主线程处理用户界面（UI）。

你可以在第 11 章发现更多关于“线程共享 GDI 对象”的信息。你可以在第 3 章找到更多有关于“等待线程结束”的信息。

快跑与等待

Hurry Up and Wait

本章告诉你 CPU 如何共享，程序如何降低对系统资源的冲击。我介绍了所谓的性能监视器。本章还介绍受激发对象（signaled objects）和各式各样的 Wait...() 函数。

你已经在第 2 章中看到了，使用 GetExitCodeThread() 可以决定一个线程是否还在执行．只要持续不断地检查 GetExitCodeThread() 的返回值，就可以等待某个线程结束．如果你没有等待线程结束就莽撞地结束程序，线程会被系统强制结束掉——在它完成它的工作之前。

如果只有一或两个线程以这种方式等待，倒也还好．但如果你有两百个线程都是靠不断地调用 GetExitCodeThread() 来等待结束呢？突然之间你会发现，CPU 的用途似乎都没有花在刀口上．因此，让等待变得比较有效率，是很重要的一件事．这一章要告诉你如何避免让线程白白浪费 CPU 时间，并且介绍一些工具，它们可以帮助你找出问题。

“等待某个什么东西”是线程常常需要做的事．当你读取用户的输入，或是存取磁盘文件时，你的线程必须等待，因为磁盘存取速度和用户输入动作的速度是 CPU 速度的百万（甚至千万）分之一．等待是线程的“必

要之恶”。

看似闲暇却忙碌（Busy Waiting）

我在第 2 章中使用了两个等待技术。第一个技术是 Win32 Sleep() 函数。这个函数要求操作系统中止线程动作，直到渡过某个指定时间之后才恢复。虽然很简单，实际上你却不可能事先知道什么事情要等待多久。即使一个原本可以快速完成的工作，也可能需要数分钟——如果有另一个更高优先权的线程也正在执行的话。

我的第二个技术是使用所谓的 busy loop，不断调用 GetExitCodeThread()，直到其结果不再是 STILL_ACTIVE。Busy loop 有时候也被称为 busy waits。一个 busy loop 通常是可以依赖的，但是它有重大的缺点：浪费 CPU 时间。这一点是如此地重要，所以我必须再强调一次：

绝对不要在 Win32 中使用 busy loop

让我们看看一个 busy loop 对系统效率造成的冲击。列表 3-1 的程序计算圆周率 PI 值两次，并显示每次所花费的时间。第一次是直接调用计算函数，第二次则是产生一个负责计算的 worker 线程，主线程则进入 busy loop 等待之。

列表 3-1 BUSYWAIT.C——busy loop 对效率的影响

```
#0001  /*
#0002   * BusyWait.c
#0003   *
#0004   * Sample code for Multithreading Applications in Win32
#0005   * This is from Chapter 3, Listing 3-1
```

```
#0006  *
#0007  * Demonstrate the effect on performance
#0008  * of using a busy loop. First call the
#0009  * worker routine with just a function call
#0010  * to get a baseline performance reading,
#0011  * then create a second thread and a
#0012  * busy loop.
#0013  *
#0014  * Build command:  cl /MD busywait.c
#0015  */
#0016
#0017  #define WIN32_LEAN_AND_MEAN
#0018  #include <stdio.h>
#0019  #include <stdlib.h>
#0020  #include <windows.h>
#0021  #include <time.h>
#0022  #include "MtVerify.h"
#0023
#0024  DWORD WINAPI ThreadFunc(LPVOID);
#0025
#0026  int main()
#0027  {
#0028      HANDLE hThrd;
#0029      DWORD exitCode = 0;
#0030      DWORD threadId;
#0031      DWORD begin;
#0032      DWORD elapsed;
#0033
#0034      puts("Timing normal function call...");
#0035      begin = GetTickCount();
#0036      ThreadFunc(0);
#0037      elapsed = GetTickCount()-begin;
#0038      printf("Function call took:  %d.%.03d seconds\n\n",
#0039                  elapsed/1000, elapsed%1000);
#0040
#0041      puts("Timing thread + busy loop...");
#0042      begin = GetTickCount();
#0043
#0044      MTVERIFY( hThrd = CreateThread(NULL,
#0045          0,
#0046          ThreadFunc,
```

```
#0047         (LPVOID)1,
#0048         0,
#0049         &threadId )
#0050     );
#0051     /* This busy loop chews up lots of CPU time */
#0052     for (;;)
#0053     {
#0054         GetExitCodeThread(hThrd, &exitCode);
#0055         if ( exitCode != STILL_ACTIVE )
#0056             break;
#0057     }
#0058
#0059     elapsed = GetTickCount()-begin;
#0060     printf("Thread + busy loop took:  %d.%.03d seconds\n",
#0061               elapsed/1000, elapsed%1000);
#0062
#0063     MTVERIFY( CloseHandle(hThrd) );
#0064
#0065     return EXIT_SUCCESS;
#0066 }
#0067
#0068
#0069 /*
#0070  * Cute little busy work routine that computes the value
#0071  * of PI using probability.  Highly dependent on having
#0072  * a good random number generator (rand is iffy)
#0073  */
#0074 DWORD WINAPI ThreadFunc(LPVOID n)
#0075 {
#0076     int i;
#0077     int inside = 0;
#0078     double val;
#0079
#0080     UNREFERENCED_PARAMETER(n);
#0081
#0082     /* Seed the random-number generator */
#0083     srand( (unsigned)time( NULL ) );
#0084
#0085     for (i=0; i<1000000; i++)
#0086     {
#0087         double x = (double)(rand())/RAND_MAX;
```

```
#0088          double y = (double)(rand())/RAND_MAX;
#0089          if ( (x*x + y*y) <= 1.0 )
#0090              inside++;
#0091      }
#0092      val = (double)inside / i;
#0093      printf("PI = %.4g\n", val*4);
#0094      return 0;
#0095  }
```

程序输出：

```
Timing normal function call...
PI = 3.146
Function call took:  7.993 seconds

Timing thread + busy loop...
PI = 3.14
Thread + busy loop took:  15.946 seconds
```

惊讶吗？第二次计算使用了 busy loop，所花时间竟然几乎是一般函数调用的两倍．这是抢先式多任务造成的影响！操作系统没有能力分辨哪个线程的工作是有用的，哪个线程的工作是比较没有用的，所以每个线程获得一律平等的 CPU 时间．本例的主线程使用其所获得的 CPU 时间，疯狂地检查 GetExitCodeThread() 的传回值，每秒数百万次．吓，一点生产力都没有！

如果 worker 线程做的事情不需要太多的 CPU 时间，问题可能会更严重．书附盘片中有一个名为 BUSY3 的程序就是这样，它的 busy loop 花费所有可用的 CPU 时间来检查线程结束了没有．如果这是唯一执行起来的程序，那也罢了，但你却因此不能够使用 Explorer（资源管理器）等等，而你的 busy loop 从其他真正需要 CPU 时间的程序中取走宝贵的资源．Busy loop 对于那种需要面对数十个甚至数百个客户的服务器而言，真是灾情惨重．

尽管列表 3-1 中的 busy loop 缺点是如此明显，你还是很可能在无意识状态下在 Win16 环境中使用类似的算法做出 busy loop．Win16 并没有操作系统层级的设备帮助你等待（译注：作者指的是像 GetExitCodeThread() 之类的

函数），于是反而鼓励产生出更可怜的 busy loop。下面的程序片段显示 Win16 程序中一个常见的等待循环：

```
void Wait(DWORD ms)
{
    DWORD begin = GetTickCount();
    while (GetTickCount() - begin < ms)
        ;  // Do Nothing
}
```

这段代码用掉所有可用的 CPU 时间，纯粹只是等待（呆等），不做任何其他事情。虽然 Win16 提供有计时器（timer），程序流程还是必须回到主消息循环来才能使用。有时候上述做法比较简单（译注）。总的来说，busy loop 可能在你的程序代码中到处滋生——即使你没有使用线程。

译注 作者的意思是说，虽然你可以利用 WM_TIMER 漂亮地解决某些问题，但上述那种单纯的等待，在技术上简单得多。

性能监视器（Performance Monitor）

幸运的是，不论 Windows NT 或 Windows 95 都提供了工具，在你的程序过度使用 CPU 时间时给予警告。当我尝试写一个“animated selection bounding rectangle”（译注）时，我用上了这些工具。

译注 “animated selection bounding rectangle”就是那种有橡皮筋效果的四方盒，让你选择一块四方区域。

如果有适当的设计，一个小小的“animated rectangle”应该几乎不需要花费任何 CPU 时间。但当我使用性能监视器时，我发现它用掉了几乎所有的 CPU 时间，甚至即使程序都已经被我最小化了，还是一样。显然，其中有问题。

我的第一次修正导致 CPU 使用率下降到一个可以接受的程度。然后我执行另一个程序。此时“Animated rectangle”的 CPU 用量突然升高，甚至即使我的程序没做什么事。我做了相当的调整，捏捏拉拉，最后终于让“闲置时间的处理”使用最小量 CPU 时间——不管程序是否在前台。

让我们看看这些工具。它们对于检验 Win32 程序性能十分重要，对于开发多线程程序也有非常高的价值。

Windows NT 性能监视器（Performance Monitor）

译注 本小节有许多关于 Windows 用户界面(UI)的操作动作。我采用 Windows 95 中文版和 Windows NT 4.0 中文版上出现的中文名称。

在 Windows NT 中，性能监视器提供一张即时图形，显示许多发生在操作系统内部的事情。你可以从 NT 3.x 的文件管理器或从 NT 4.x 的【开始】菜单中选择【系统管理工具（公用）】，再打开性能监视器：

一旦性能监视器执行起来，请选按工具栏上标有 + 号的按钮（译注：那是【添加计数器】按钮）。你会在随后出现的对话框左边清单中看到“Processor”和“Processor Utilization”两个项目。如果选按【添加】按钮，性能监视器便开始产生一张足以表示 CPU 忙碌程度的“心电图”，如图 3-1 所示。上上下下的曲线反映出 CPU 的动作频繁度。

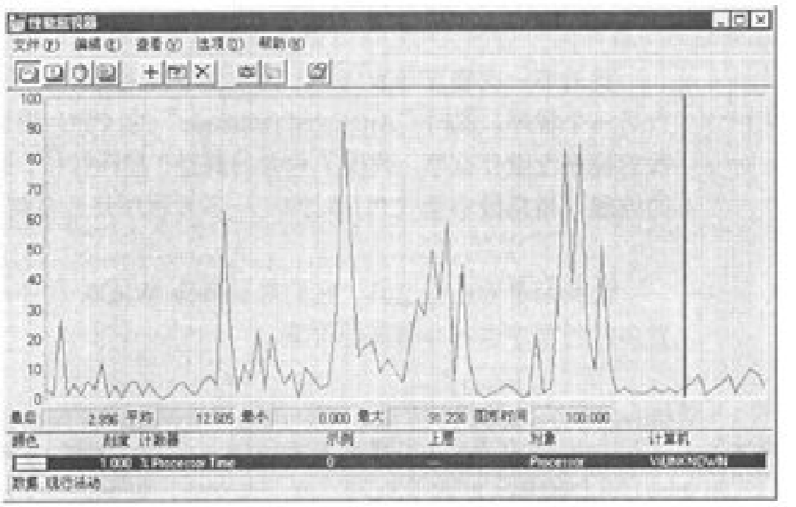


图 3-1 性能监视器中的 CPU 使用率

译注 请注意，我发现 Windows NT 4.0 中文版的图形界面或说明文字中把 thread 译为“线程”，把 instance 译为“示例”，把 Parent 译为“上层”。都和一般所使用的译词不相同，而且不对味。

当图 3-1 所示的线条尚未到达 100% 时，表示运算能力还有余裕可以发挥。如果线条直达 100%，CPU 就得开始分割时间给所有进程共享了，而其结果就是每件事情都慢了下来。这确是我们所看到的 BUSYWAIT 的行为。

由于性能监视器可以告诉我们 CPU 的忙碌程度，它于是成了一个重要的工具，用以确定我们所写的程序有预期的行为。当你启动一个程序时，如果 CPU 利用率到达 100% 并且在该处停留很久，这就是个很明显的警告：你的程序可能有一个 busy loop。

你可以使用性能监视器来观察 CPU 时间如何被切割给线程分享——看看 BUSYWAIT 就知道了。但第一个问题是，如何让 BUSYWAIT 维持足够长的时间以便我们观察其行为？在 BUSY2 程序中，我把 ThreadFunc() 末尾的循环从百万次改为 100 个百万次。这样的改变使程序维持够长的时间让我们看看它到底做了些什么。注意，你可以按下 Ctrl-C 结束这个程序。

现在，执行 BUSY2，然后切换到性能监视器．选按工具栏上的 + 按钮（【添加计数器】按钮），增加更多的项目到图片中．如果你选择对话框的【对象】清单中的“Thread”，右边的【示例】（译注：英文版为 Instance）清单内就会出现一系列目前正在运作的线程，如图 3-2．

BUSY2 的两个线程都会显示在这份清单上头．请选择第一个线程并按下【添加】，再选择第二个线程并按下【添加】，于是获得图 3-3．

你可以从此图看出，每一个线程获得的 CPU 时间都比可用时间的一半还少一点．另 10% 的系统时间因其他进程的瓜分或某些额外负担（overhead）而流失．两个线程启动于不同时刻，这是因为我得花点时间才能够把第二个线程加进来．

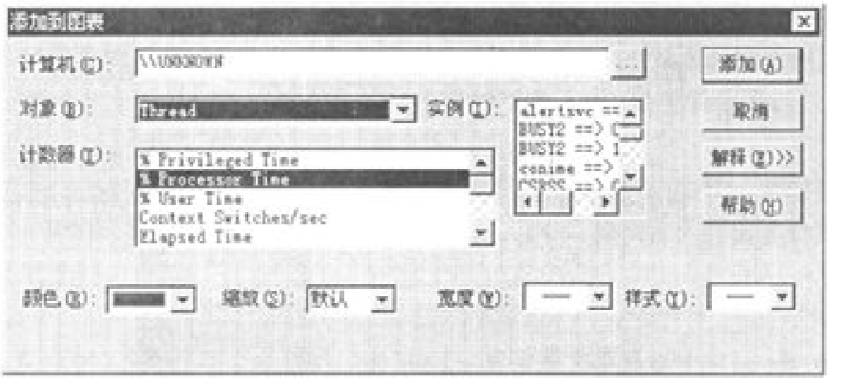


图 3-2　在性能监视器中监视线程

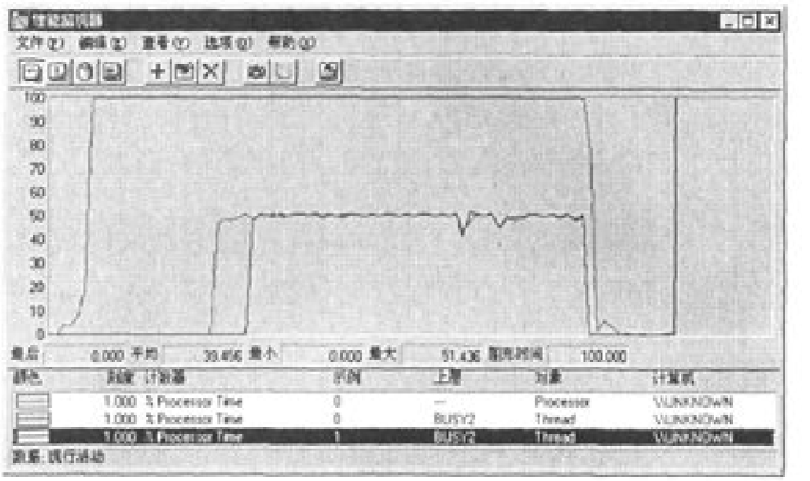


图 3-3 在性能监视器中观察 BUSY2 的线程

Windows 95 系统监视器（System Monitor）

Windows 95 有一个程序名曰系统监视器，允许你做一些类似 NT 性能监视器的事情．请按下【开始】按钮并选择【程序/附件/系统工具/系统监视器】：

如果此程序未安装，请打开“控制面板”，选择【添加/删除程序】，选择【Windows 安装程序】附页，从清单中选择【系统工具】，然后按下【详细资料】按钮．在接近清单尾端处，请确认“系统监视器”被选中，然后按【确定】按钮．

一旦你让系统监视器执行起来，请选按【编辑/添加项目】，然后在【添加项目】对话框中，选择左边的“核心”（“Kernel”）类别和右边的“处理器使用情况(%)”（“Processor Usage(%)”）项目，再按下【确定】按钮．现在你应该看到图 3-4．

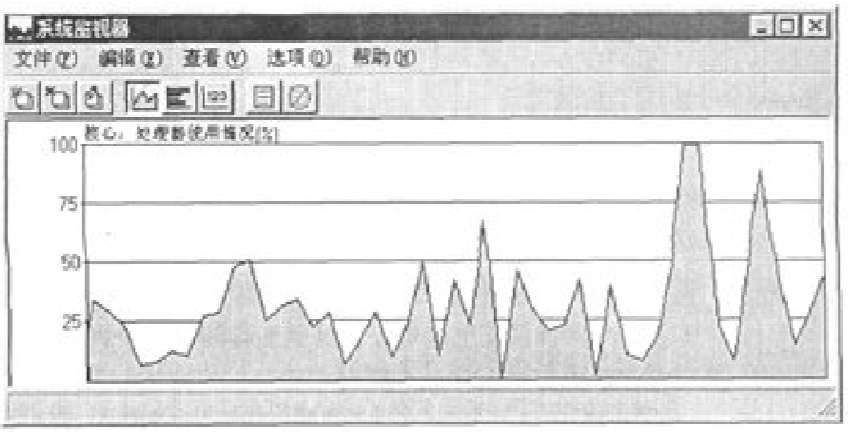


图 3-4 SYSMON.EXE 所显示的 CPU 使用情况

Windows 95 的系统监视器比 Windows NT 的性能监视器多了一些限制。你只能要求显示 CPU 的整体使用率，不能要求显示个别线程的 CPU 使用率。然而这个工具仍然能够在你的程序贪婪夺取 CPU 时给你警告。

如果你一执行系统监视器，其图形就一直稳定地停留在 100%，可能是你有一个 Win16 程序或一个 DOS 程序独占了 CPU。如果你能找出罪魁祸首并且把它杀掉，事情才有意义。

译注 关于 Windows 95 的系统监视器（SysMon），有两篇文章可以参考。一是 Matt Pietrek 发表于 Microsoft Systems Journal（MSJ）1995.09 的 Under The Hood 专栏，讲述 SysMon 所监视之系统内部数据与 Registry 之间的关系。另一篇文章是 Vitaly Vatnikov 发表于 Windows Developer's Journal（WDJ）1997.04 的 "A VxD for Custom SYSMON Statistics"，讲述如何让 ring3 程序使用 PREF.VXD 所开放的 ring0 界面，并藉着 SysMon 显示出自定义的统计数量。PREF.VXD 是存在于 VMM32.VXD 中的一个 VxD。

等待一个线程的结束

现在你已经知道为什么 busy loop 是个坏点子了，让我们看看 Win32 中的改善方法．我们需要一个新版的 Sleep()，它能够在某个线程结束时（而不是某段时间结束时）被调用．由于让线程停工是操作系统的责任，很合理地我们会认为操作系统也有责任让其他线程知道某个线程停工了．

Win32 提供的一个名为 WaitForSingleObject() 的函数就可以这么做．它的第一个参数是个核心对象（如线程）的 handle．为了方便讨论，我把即将等待的线程称为线程 #1，把正在执行的线程称为线程 #2．刚刚说的“线程核心对象”指的是线程 #2．

调用 WaitForSingleObject() 并放置一个“线程核心对象”作为参数，将使线程 #1 开始睡眠，直到线程 #2 结束为止．就像 Sleep() 函数一样，WaitForSingleObject() 也有一个参数用来指定最长的等待时间．

```
DWORD WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds
);
```

参数

hHandle	等待对象的 handle（代表一个核心对象）．在本例中，此为线程 handle．
dwMilliseconds	等待的最长时间．时间终了，即使 handle 尚未成为激发状态，此函数还是要返回．此值可以是 0（代表立刻返回），也可以是 INFINITE 代表无穷等待．

返回值

如果函数失败，则传回 WAIT_FAILED。这时候你可调用 GetLastError() 取得更多信息。此函数的成功有三个因素：

1. 等待的目标（核心对象）变成激发状态。这种情况下返回值将为 WAIT_OBJECT_0。

2. 核心对象变成激发状态之前，等待时间终了。这种情况下返回值将为 WAIT_TIMEOUT。

3. 如果一个拥有 mutex（互斥器）的线程结束前没有释放 mutex，则传回 WAIT_ABANDONED。Mutexes 将在第 4 章有更多讨论。

获得一个线程对象的 handle 之后，WaitForSingleObject() 要求操作系统让线程 #1 睡觉，直到以下任何一种情况发生：

- 线程 #2 结束
- dwMilliseconds 时间终了。该值系从函数调用后开始计算。

由于操作系统持续追踪线程 #2，即使线程 #2 失事或被强迫结束，WaitForSingleObject() 仍然能够正常运作。

在第 2 章的列表 2-3 中，我们使用下面的代码来等待 worker 线程的结束：

```
for (;;)
{
   int rc;
   rc = GetExitCodeThread(hThrd, &exitCode);
   if (!rc && exitCode != STILL_ACTIVE )
      break;
}
```

这个 busy loop 现在可以被下面这行代码取代了：

```
WaitForSingleObject( hThrd, INFINITE );
```

? FAQ 10：我如何得知一个核心对象是否处于激发状态？

关于 time-out，有一个特别重要的用途，但很少被人注意。设定 time-out 为 0，使你能够检查 handle 的状态并立刻返回，没有片刻停留。如果 handle 已经备妥，那么这个函数会成功并传回 WAIT_OBJECT_0。否则，这个函数立刻返回并传回 WAIT_TIMEOUT。

另有其他一些理由使你需要设定 time-out 参数。最简单的一个理由就是你不希望被粘住，特别是在调试时。如果你所等待的线程进入了一个无穷循环，你或许可以根据此函数的返回值以及 time-out 终了与否，获得一些警告。

在上述例子中，你可以利用 time-out 提供一个动画，表示你正在等待某个线程的结束。你可以每 500 毫秒就 time-out 一次，更新图示，然后再继续等待。

WaitForSingleObject() 可以面对许多种 handles 工作，不一定要是本例所使用的线程 handle。事实上，Win32 中大部分以 HANDLE 表示的对象都能够作为 WaitForSingleObject() 的等待目标。视你所拥有的对象不同，操作系统等待的事情也不一样。形式上来说，系统等待着这一对象“被激发”。

叮咚：被激发的对象（Signaled Objects）

第 2 章中我曾经提过 Win32 的各种核心对象，如文件、线程、互斥器（Mutexes）等等。这些对象的状态都可能是线程感兴趣的东西。信号量（semaphores）和互斥器（mutexes）可记录红灯绿灯状态，文件对象可告诉我们一个 I/O 操作何时完成，线程对象则一如所见，可以告诉我们它何时结束。

线程可以使用像 WaitForSingleObject() 这样的函数，有效地等待上述任何情况发生。问题是，我们真正等待的到底是什么？

? FAQ 11: 什么是一个被激发的对象?

可被 WaitForSingleObject() 使用的核心对象有两种状态：激发与未激发。WaitForSingleObject() 会在目标物变成激发状态时返回。事实上我们也几乎是以此作为对象激发与否的操作型定义。

当核心对象被激发时，会导致 WaitForSingleObject() 醒来。稍后你将看到的其他 Wait() 函数也是如此。

当线程正在执行时，线程对象处于未激发状态。当线程结束时，线程对象就被激发了。因此，任何线程如果等待的是一个线程对象，将会在等待对象结束时被调用，因为当时线程对象自动变成激发状态。

UNIX

Unix 中的信号（signals）和 Win32 中的被激发对象（signaled objects）是完全不同的两个观念。Unix 中的一个信号（signal）是一个异步事件（asynchronous event），通常对你的程序有立即的冲击。而在 Win32 中当一个对象变成激发，意思是说它的内部状态有所改变，这对程序没有冲击，除非有一个线程正在等待此对象的激发。

数个线程可以同时等待相同的线程 handle。当该线程 handle 变成激发状态时，所有等待中的线程都会被唤醒。然而，其他核心对象可能只唤醒一个等待中的线程。到底是哪一种行为，得视你等待什么样的对象而定。某些对象的激发状态只能够维持到一个等待中的线程被唤醒，其他对象的激发状态则或许可以维持到它又被明白地重置（reset）。下一章你就会看到一个名为 EVENTTST 的程序，让你实地体会一下线程如何应付激发对象。

表 3-1 显示在 Win32 的核心对象中，受激发和未激发的意义。

? FAQ 12：

“激发”对于不同的核心对象有什么不同的意义？

表 3-1 核心对象激发状态的意义

对 象	说 明
Thread*（线程）	当线程结束时，线程对象即被激发．当线程还在进行时，则对象处于未激发状态．线程对象系由 CreateThread() 或 CreateRemoteThread() 产生
Process*（进程）	当进程结束时，进程对象即被激发．当进程还在进行时，则对象处于未激发状态．CreateProcess() 或 OpenProcess() 会传回一个进程对象的 handle
Change Notification	当一个特定的磁盘子目录中发生一件特别的变化时，此对象即被激发。此对象系由 FindFirstChangeNotification() 产生（译注）
Console Input*	当 console 窗口的输入缓冲区中有数据可用时，此对象将处于激发状态．CreateFile() 和 GetStdFile() 两函数可以获得 console handle．详见第 8 章
Event*	Event 对象的状态直接受控于应用程序所使用的三个 Win32 函数：SetEvent()、PulseEvent()、ResetEvent()．CreateEvent() 和 OpenEvent() 都可以传回一个 event object handle．Event 对象的状态也可以被操作系统设定——如果使用于“overlapped”操作时（详见第 4 章）
Mutex*	如果 mutex 没有被任何线程拥有，它就是处于激发状态。一旦一个等待 mutex 的函数返回了，mutex 也就自动重置为未激发状态．CreateMutex() 和 OpenMutex() 都可以获得一个 mutex handle．详见第 4 章
Semaphore*	Semaphore 有点像 mutex，但它有个计数器，可以约束其拥有者（线程）的个数．当计数器内容大于 0 时，semaphore 处于激发状态，当计数器内容等于 0 时，semaphore 处于未激发状态．CreateSemaphore() 和 OpenSemaphore() 可以传回一个 semaphore handle．详见第 4 章

* 表示此种对象将涵盖于本书内容之中。

译注 上述各种对象，只有 Change Notification 未涵盖于本书范围内，所以我做一点补充。所谓“当一个特定的磁盘子目录中发生一个特别的变化”，指的是以下六种变化：

代 码	意 义
FILE_NOTIFY_CHANGE_FILE_NAME	产生、删除、重新命名一个文件
FILE_NOTIFY_CHANGE_DIR_NAME	产生或删除一个子目录
FILE_NOTIFY_CHANGE_ATTRIBUTES	目录及子目录中的任何属性改变
FILE_NOTIFY_CHANGE_SIZE	目录及子目录中的任何文件大小的改变
FILE_NOTIFY_CHANGE_LAST_WRITE	目录及子目录中的任何文件的最后写入时间的改变
FILE_NOTIFY_CHANGE_SECURITY	目录及子目录中的任何安全属性改变

等待多个对象

现在让我们写一个程序，使用最多三个线程来完成六项工作。列表 3-2 显示的是 TaskQueS 范例程序，其中有一个函数名为 ThreadFunc()，用以执行某些事情然后返回。工作的执行是靠调用 Sleep() 来模拟，时间长度则是随机给予。只要一个线程结束，就会有另一个线程被产生，做下一个工作。

列表 3-2 TASKQUES.C 分配工作并以 WaitForSingleObject()等待之

```
#0001  /*
#0002   * TaskQueS.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 3, Listing 3-2
#0006   *
#0007   * Call ThreadFunc NUM_TASKS times, using
#0008   * no more than THREAD_POOL_SIZE threads.
#0009   * This version uses WaitForSingleObject,
#0010   * which gives a very suboptimal solution.
```

```
#0011  *
#0012  * Build command:  cl /MD TaskQueS.c
#0013  */
#0014
#0015  #define WIN32_LEAN_AND_MEAN
#0016  #include <stdio.h>
#0017  #include <stdlib.h>
#0018  #include <windows.h>
#0019  #include "MtVerify.h"
#0020
#0021  DWORD WINAPI ThreadFunc(LPVOID);
#0022
#0023  #define THREAD_POOL_SIZE 3
#0024  #define MAX_THREAD_INDEX THREAD_POOL_SIZE-1
#0025  #define NUM_TASKS 6
#0026
#0027  int main()
#0028  {
#0029     HANDLE  hThrds[THREAD_POOL_SIZE];
#0030     int     slot = 0;
#0031     DWORD   threadId;
#0032     int     i;
#0033     DWORD   exitCode;
#0034
#0035     /*          i=   1 2 3 4 5 6 7 8 9
#0036      * Start Thread   X X X X X X
#0037      * Wait on thread      X X X X X X
#0038      */
#0039     for (i=1; i<=NUM_TASKS; i++)
#0040     {
#0041        if (i > THREAD_POOL_SIZE)
#0042        {
#0043           WaitForSingleObject(hThrds[slot], INFINITE);
#0044           MTVERIFY( GetExitCodeThread(hThrds[slot], &exitCode) );
#0045           printf("Slot %d terminated\n", exitCode );
#0046           MTVERIFY( CloseHandle(hThrds[slot]) );
#0047        }
#0048        MTVERIFY( hThrds[slot] = CreateThread(NULL,
#0049           0,
#0050           ThreadFunc,
#0051           (LPVOID)slot,
```

```
#0052            0,
#0053            &threadId ) );
#0054         printf("Launched thread #%d (slot %d)\n", i, slot);
#0055         if (++slot > MAX_THREAD_INDEX)
#0056            slot = 0;
#0057      }
#0058      for (slot=0; slot<THREAD_POOL_SIZE; slot++)
#0059      {
#0060         WaitForSingleObject(hThrds[slot], INFINITE);
#0061         MTVERIFY( CloseHandle(hThrds[slot]) );
#0062      }
#0063      printf("All slots terminated\n");
#0064
#0065      return EXIT_SUCCESS;
#0066   }
#0067
#0068   /*
#0069    * This function just calls Sleep for
#0070    * a random amount of time, thereby
#0071    * simulating some tasks that takes time.
#0072    *
#0073    * The param "n" is the index into
#0074    * the handle array, kept for informational
#0075    * purposes.
#0076    */
#0077   DWORD WINAPI ThreadFunc(LPVOID n)
#0078   {
#0079      srand( GetTickCount() );
#0080
#0081      Sleep((rand()%8)*500+500);
#0082      printf("Slot %d idle\n", n);
#0083      return ((DWORD)n);
#0084   }
```

这个程序有严重的问题：它的效率非常低，因为它假设线程结束的次序会和它们被产生的次序相同。下面就有一个典型的执行结果，立刻可以证明这种假设是错误的。

当一个新的线程被产生时，主线程（primary thread）会印出“Launched thread”字样。每一个线程会在自己即将结束前印出“Slot x idle”字样，而主线程则在发现某个线程结束后印出“Slot x terminated”字样。

程序输出：

```
Launched thread #1 (slot 0)
Launched thread #2 (slot 1)
Launched thread #3 (slot 2)
Slot 1 idle
Slot 2 idle
Slot 0 idle
Slot 0 terminated
Launched thread #4 (slot 0)
Slot 1 terminated
Launched thread #5 (slot 1)
Slot 2 terminated
Launched thread #6 (slot 2)
Slot 2 idle
Slot 0 idle
Slot 1 idle
All slots terminated
```

你可以在输出结果的最前面看到，当第一个 worker 线程（Slot 0）结束时，第二个和第三个 worker 线程也随之结束。虽然程序的目标是要时时保持三个线程的生存，但是显然在某一段时间里没有任何线程存在。我们需要某种方法，可以监视目前生存的任何线程的结束。

WaitForMultipleObjects()

Win32 函数中的 WaitForMultipleObjects() 允许你在同一时间等待一个以上的对象。你必须将一个由 handles 组成的数组交给此函数，并指定要等待其中一个对象或是全部的对象。下面就是这个函数的原型：

```
DWORD WaitForMultipleObjects(
    DWORD nCount,
    CONST HANDLE *lpHandles,
    BOOL bWaitAll,
    DWORD dwMilliseconds
);
```

参数

nCount	表示 lpHandles 所指之 handles 数组的元素个数。最大容量是 MAXIMUM_WAIT_OBJECTS。
lpHandles	指向一个由对象 handles 所组成的数组。这些 handles 不需要为相同的类型。
bWaitAll	如果此为 TRUE，表示所有的 handles 都必须激发，此函数才得以返回。否则此函数将在任何一个 handle 激发时就返回。
dwMilliseconds	当该时间长度终了时，即使没有任何 handles 激发，此函数也会返回。此值可为 0，以便测试。亦可指定为 INFINITE，表示无穷等待。

返回值

WaitForMultipleObjects() 的返回值有些复杂。

- 如果因时间终了而返回，则返回值是 WAIT_TIMEOUT，类似 WaitForSingleObject()。
- 如果 bWaitAll 是 TRUE，那么返回值将是 WAIT_OBJECT_0。
- 如果 bWaitAll 是 FALSE，那么将返回值减去 WAIT_OBJECT_0，就表示数组中的哪一个 handle 被激发了。
- 如果你等待的对象中有任何 mutexes，那么返回值可能从 WAIT_ABANDONED_0 到 WAIT_ABANDONED_0 + nCount - 1。
- 如果函数失败，它会传回 WAIT_FAILED。这时候你可以使用 GetLastError() 找出失败的原因。

注意，handles 数组中的元素个数有上限，绝对不能够超过 MAXIMUM_WAIT_OBJECTS。在 Windows NT 3.x 和 4.0 中，其值为 64（译注：在 Windows 95 中也一样）。

我们可以利用 WaitForMultipleObjects() 来重新改写 TaskQueS 程序，让它的动作比较有效率一些。我们以 WaitForMultipleObjects 取代 WaitForSingleObject()，并指定 bWaitAll 参数为 FALSE。现在，操作系统会在同一时间监视所有的 handles。主线程可以在任何一个 worker 线程结束时获得通知，而不再盲目地以为线程是依序结束。

下一个改变就是利用 WaitForMultipleObjects() 让程序尾端清爽一些。我们把 bWaitAll 参数设为 TRUE（列表 3-3）。这么一来，main() 就不会结束，直到所有的 worker 线程都完成。

列表 3-3 TASKQUEM.C 分配工作并以 WaitForMultipleObjects() 等待之

```
#0001  /*
#0002   * TaskQueM.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 3, Listing 3-3
#0006   *
#0007   * Call ThreadFunc NUM_TASKS times, using
#0008   * no more than THREAD_POOL_SIZE threads.
#0009   * This version uses WaitForMultipleObjects
#0010   * to provide a more optimal solution.
#0011   *
#0012   * Build command:  cl /MD TaskQueM.c
#0013   */
#0014
#0015  #define WIN32_LEAN_AND_MEAN
#0016  #include <stdio.h>
#0017  #include <stdlib.h>
#0018  #include <windows.h>
#0019  #include "MtVerify.h"
#0020
#0021  DWORD WINAPI ThreadFunc(LPVOID);
```

```
#0022
#0023  #define THREAD_POOL_SIZE 3
#0024  #define MAX_THREAD_INDEX THREAD_POOL_SIZE-1
#0025  #define NUM_TASKS 6
#0026
#0027  int main()
#0028  {
#0029      HANDLE  hThrds[THREAD_POOL_SIZE];
#0030      int     slot = 0;
#0031      DWORD   threadId;
#0032      int     i;
#0033      DWORD   rc;
#0034
#0035      for (i=1; i<=NUM_TASKS; i++)
#0036      {
#0037          /* Until we've used all threads in *
#0038           * the pool, do not need to wait   *
#0039           * for one to exit                 */
#0040          if (i > THREAD_POOL_SIZE)
#0041          {
#0042              /* Wait for one thread to terminate */
#0043              rc = WaitForMultipleObjects(
#0044                  THREAD_POOL_SIZE,
#0045                  hThrds,
#0046                  FALSE,
#0047                  INFINITE );
#0048              slot = rc - WAIT_OBJECT_0;
#0049              MTVERIFY( slot >= 0
#0050                  && slot < THREAD_POOL_SIZE );
#0051              printf("Slot %d terminated\n", slot );
#0052              MTVERIFY( CloseHandle(hThrds[slot]) );
#0053          }
#0054          /* Create a new thread in the given
#0055           * available slot */
#0056          MTVERIFY( hThrds[slot++] = CreateThread(NULL,
#0057              0,
#0058              ThreadFunc,
#0059              (LPVOID)slot,
#0060              0,
#0061              &threadId ) );
#0062          printf("Launched thread #%d (slot %d)\n", i, slot);
```

```
#0063     }
#0064
#0065     /* Now wait for all threads to terminate */
#0066     rc = WaitForMultipleObjects(
#0067        THREAD_POOL_SIZE,
#0068        hThrds,
#0069        TRUE,
#0070        INFINITE );
#0071     MTVERIFY( rc >= WAIT_OBJECT_0
#0072           && rc < WAIT_OBJECT_0+THREAD_POOL_SIZE);
#0073     for (slot=0; slot<THREAD_POOL_SIZE; slot++)
#0074        MTVERIFY( CloseHandle(hThrds[slot]) );
#0075     printf("All slots terminated\n");
#0076
#0077     return EXIT_SUCCESS;
#0078  }
#0079
#0080  /*
#0081   * This function just calls Sleep for
#0082   * a random amount of time, thereby
#0083   * simulating some task that takes time.
#0084   *
#0085   * The param "n" is the index into
#0086   * the handle array, kept for informational
#0087   * purposes.
#0088   */
#0089  DWORD WINAPI ThreadFunc(LPVOID n)
#0090  {
#0091     srand( GetTickCount() );
#0092
#0093     Sleep((rand()%10)*800+500);
#0094     printf("Slot %d idle\n", n);
#0095     return ((DWORD)n);
#0096  }
```

程序输出：

```
Launched thread #1 (slot 1)
Launched thread #2 (slot 2)
Launched thread #3 (slot 3)
```

```
Slot 1 idle
Slot 0 terminated
Launched thread #4 (slot 1)
Slot 2 idle
Slot 3 idle
Slot 1 terminated
Launched thread #5 (slot 2)
Slot 2 terminated
Launched thread #6 (slot 3)
Slot 1 idle
Slot 2 idle
Slot 3 idle
All slots terminated
```

在一个 GUI 程序中等待

“主消息循环”是 16 位 Windows 程序中得以不依靠 busy loop 而能够等待某些事物的一个地方。Windows 程序中的标准消息循环看起来像这个样子：

```
while (GetMessage(&msg, NULL, 0, 0,))
{
   TranslateMessage(&msg);
   DispatchMessage(&msg);
}
```

? FAQ 13：我如何在主线程中等待一个 handle?

GetMessage() 有点像是特殊版本的 WaitForSingleObject()，它等待消息而不是核心对象。一旦你调用 GetMessage()，除非有一个消息真正进入你的消息队列（message queue）之中，否则它不会返回。在此期间，Windows 就可以自由地将 CPU 时间给予其他程序。GetMessage() 是 Win16 合作型多任务的关键。

“常常回到主消息循环”是十分重要的一件事。如果你没这么做，你的窗口就会停止重绘，你的程序菜单就不再有作用，用户不喜欢的事情则慢慢开始

发生。问题是，如果你正使用 WaitForSingleObject()或 WaitForMultipleObjects()等待某个对象被激发，你根本没有办法回到主消息循环中去。

下面这种做法并不能真正解决问题：当主线程正在处理主消息循环时，不使用第二个线程来等待 handles。因为如果你这么做，只是把问题从某处移到另一处，你还是得决定到底要使用 "polling" 方法，或是使用一个 Win32 Wait...() 函数，来察知新线程的结束。

为了解决这个问题，主消息循环必须修改，使它得以同时等待消息或是核心对象被激发。你必须使用一个 MsgWaitForMultipleObjects() 函数。这个函数非常类似 WaitForMultipleObjects()，但它会在“对象被激发”或“消息到达队列”时被唤醒而返回。MsgWaitForMultipleObjects() 多接受一个参数，允许指定哪些消息是观察对象。

```
DWORD MsgWaitForMultipleObjects(
    DWORD nCount,
    LPHANDLE pHandles,
    BOOL fWaitAll,
    DWORD dwMilliseconds,
    DWORD dwWakeMask
);
```

参数

dwWakeMask	欲观察的用户输入消息，可以是： QS_ALLINPUT QS_HOTKEY QS_INPUT QS_KEY QS_MOUSE QS_MOUSEBUTTON QS_MOUSEMOVE QS_PAINT

QS_POSTMESSAGE
QS_SENDMESSAGE
QS_TIMER

返回值

和 WaitForMultipleObjects() 相比较，MsgWaitForMultipleObjects() 有一些额外的返回值意义。为了表示“消息到达队列”，返回值将是 WAIT_OBJECT_0 + nCount.

MsgWaitForMultipleObjects() 的正确使用方式是改写主消息循环，使得激发状态之 handles 得以像消息一样地被对待. 视其返回值而定，消息循环或许调用 GetMessage() 并处理下一个消息，或许调用你所提供的一个处理函数，以处理受激发的 handles. 列表 3-4 是一个骨干架构，显示在主消息循环中如何使用 MsgWaitForMultipleObjects().

列表 3-4 一个主消息循环，内含 MsgWaitForMultipleObjects()

```
#0001  DWORD   nWaitCount;
#0002  HANDLE  hWaitArray[4];
#0003  BOOL    quit;
#0004  int     exitCode;
#0005
#0006  while (!quit)
#0007  {
#0008      MSG msg;
#0009      int rc;
#0010
#0011      rc = MsgWaitForMultipleObjects(
#0012                              nWaitCount,
#0013                              hWaitArray,
#0014                              FALSE,
#0015                              INFINITE,
#0016                              QS_ALLINPUT);
#0017
#0018      if (rc == WAIT_OBJECT_0 + nWaitCount)
```

```
#0019     {
#0020        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
#0021        {   // Get Next message in queue
#0022           if (msg.message == WM_QUIT)
#0023           {
#0024              quit = TRUE;
#0025              exitCode = msg.wParam;
#0026              break;
#0027           } // end if
#0028           TranslateMessage(&msg);
#0029           DispatchMessage(&msg);
#0030        } // end while
#0031     }
#0032     else if (rc >= WAIT_OBJECT_0 && rc < WAIT_OBJECT_0 + nWaitCount)
#0033     {
#0034        int nIndex = rc - WAIT_OBJET_0;
#0035        // We now know that the handle at array position
#0036        // nIndex was signaled.
#0037        // We would have had to keep track of what those
#0038        // handle mean to decide what to do next.
#0039     }
#0040     else if (rc == WAIT_TIMEOUT)
#0041     {
#0042        // Timeout expired
#0043     }
#0044     else if (rc >= WAIT_ABANDONED_0 && rc < WAIT_ABANDONED_0 + nWaitCount)
#0045     {
#0046        int nIndex = rc - WAIT_ABANDONED_0;
#0047        // A thread died that owned a mutex
#0048        // More about this in Chapter 4
#0049     }
#0050     else
#0051     {
#0052        // Something went wrong
#0053     }
#0054  }
```

这一程序片段还有最后一个问题没有解决．或许有某种需求，在程序收到 WM_QUIT 之后，必须等待某些 handles 被激发。以第 2 章的 BACKPRNT 程序

为例，即使还有线程在运行，用户仍然可以选择【Exit】。那么，除非你等待所有的线程都被激发，否则当主循环结束时，所有的后台线程都将被贸然结束掉。

使用 MsgWaitForMultipleObjects()，我们就可以改写 BACKPRNT 的主循环，使得当 worker 线程还在运行时，【Exit】菜单会被除能（disabled）。下面是第 2 章的 BACKPRNT 程序的消息循环，非常典型的一个主消息循环：

```
while (GetMessage(&msg, NULL, 0, 0))
{   // Get Next message in queue
   if(hDlgMain == NULL || !IsDialogMessage(hDlgMain,&msg))
   {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
   }
} // end while
```

PRNTWAIT 范例程序

你可以在书附盘片中找到一个 PRNTWAIT 程序。这个程序是 BACKPRNT 的新版，允许用户在任何时刻选择【File/Exit】菜单项，并在程序结束之前等待所有的线程结束。PRNTWAIT 也会显示目前有多少线程正在运行。

列表 3-5 显示 PRNTWAIT 的主消息循环，其中用到 MsgWaitForMultipleObjects()。如你所见，整个结构变复杂了许多。这个范例只处理列表 3-4 中的某些情况而已。例如，PRNTWAIT 不使用 mutexes，所以消息循环不需要处理 WAIT_ABANDONED。

列表 3-5 主消息循环，节录自 PRNTWAIT

```
#0001  while (!quit || gNumPrinting > 0)
#0002  {   // Wait for next message or object being signaled
#0003      DWORD   dwWake;
#0004      dwWake = MsgWaitForMultipleObjects(
#0005                              gNumPrinting,
#0006                              gPrintJobs,
#0007                              FALSE,
```

```
#0008                           INFINITE,
#0009                           QS_ALLEVENTS);
#0010
#0011     if (dwWake >= WAIT_OBJECT_0 && dwWake < WAIT_OBJECT_0 + gNumPrinting)
#0012     {   // Object has been signaled
#0013        // Reorder the handle array so we do not leave
#0014        // empty slots. Take the handle at the end of
#0015        // the array and move it into the now-empty slot.
#0016        int index = dwWake - WAIT_OBJECT_0;
#0017        gPrintJobs[index] = gPrintJobs[gNumPrinting-1];
#0018        gPrintJobs[gNumPrinting-1] = 0;
#0019        gNumPrinting--;
#0020        SendMessage(hDlgMain, WM_THREADCOUNT, gNumPrinting, 0L);
#0021     } // end if
#0022     else if (dwWake == WAIT_OBJECT_0 + gNumPrinting)
#0023     {
#0024        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
#0025        {   // Get Next message in queue
#0026           if(hDlgMain == NULL || !IsDialogMessage(hDlgMain,&msg))
#0027           {
#0028              if (msg.message == WM_QUIT)
#0029              {
#0030                 quit = TRUE;
#0031                 exitCode = msg.wParam;
#0032                 break;
#0033              } // end if
#0034              TranslateMessage(&msg);
#0035              DispatchMessage(&msg);
#0036           }
#0037        } // end while
#0038     }
#0039  } // end while
```

有数种情况是这个循环必须处理而却可能在它第一次设计时容易被忽略的：

1. 在你收到 WM_QUIT 之后，Windows 仍然会传送消息给你。如果你要在收到 WM_QUIT 之后等待所有线程结束，你必须继续处理你的消息，否则窗口会变得反应迟钝，而且没有重绘能力。

2. MsgWaitForMultipleObjects() 不允许 handles 数组中有缝隙产生。所以当某个 handle 被激发了时，你应该在下一次调用 MsgWaitForMultipleObjects() 之前先把 handles 数组做个整理、紧压，不要只是把数组中的 handle 设为 NULL。如果你仔细观察上述程序代码，你会发现我把最尾端的 handle 移到腾空的位置，然后再把数组大小减 1。

3. 如果有另一个线程更改了对象数组，而那是你正在等待的，那么你需要一种方法，可以强迫 MsgWaitForMultipleObjects() 返回，并重新开始，以包含这个新的 handle。列表 3-5 的解决之道是利用 WM_THREADCOUNT 消息。

第三案需要更多的讨论。通常程序中只会有一个地方调用 MsgWaitForMultipleObjects()，而这个调用存在于消息循环之中。你用在 MsgWaitForMultipleObjects() 身上的 handles 数组，完全在你的控制之下。如果你需要等待更多核心对象，只要把它们加到数组之中即可。上述第三案所讨论的是，另一个线程需要修改 handles 数组的情况。很明显地，这时候不可能单纯地直接把 handle 写进数组之中，因为数组已经交给 MsgWaitForMultipleObjects() 了。所以你必须强迫 MsgWaitForMultipleObjects() 返回，使你有机会更新数组，然后再重新启动 MsgWaitForMultipleObjects()。

提要

在这一章中我们重温了 busy loops 的不良结果，并且学习如何使用 Windows NT 的性能监视器捕捉其中的问题。我们也认识了所谓的“激发状态的对象”，并且学习如何在一个 worker 线程或一个 GUI 线程中等待一个或多个这样的对象。最后，我们看到了如何重建一个主消息循环，俾能够适当地使用 MsgWaitForMultipleObjects()。

同步控制

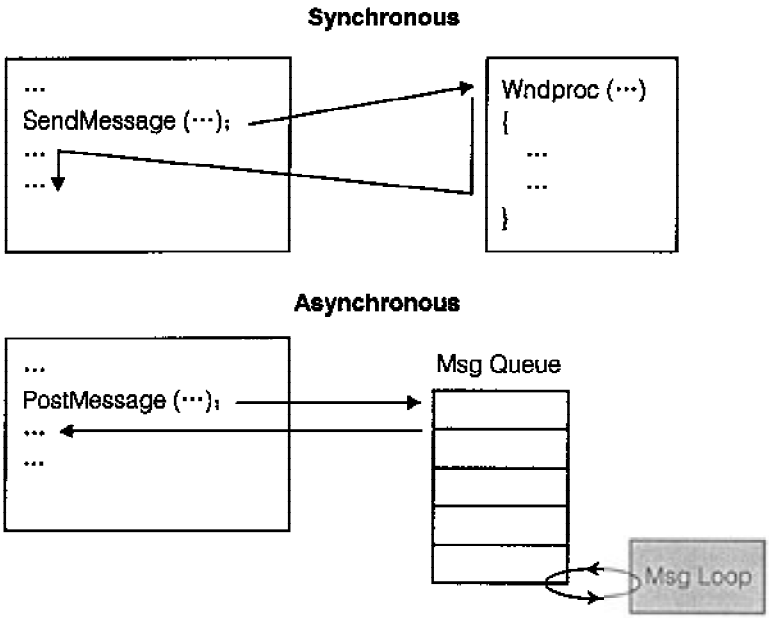
Synchronization

本章讨论 Win32 同步机制，并特别把重点放在多任务环境的效率上。

撰写多线程程序的一个最具挑战性的问题就是：如何让一个线程和另一个线程合作。除非你让它们同心协力，否则必然会出现如第 2 章所说的“race conditions”（竞争条件）和“data corruption”（数据被破坏）的情况。

在典型的办公室文化中，协调工作是由管理者来执行的。类似的解决方案，也就是“让某个线程成为大家的老板”。当然可以在软件中实现出来，但是每逢它们需要指挥时，就要它们排队等候，其实有着严重的缺点。通常那会使得队伍又长又慢。这对于一个高效率的电算系统而言，实在不是一个有用的解决方案。

译注 让我先对同步（synchronous）与异步（asynchronous）做个说明。当程序 1 调用程序 2 时，程序 1 停下不动，直到程序 2 完成回到程序 1 来，程序 1 才继续下去，这就是所谓的“synchronous”。如果程序 1 调用程序 2 后，径自继续自己的下一个动作，那么两者之间就是所谓的“asynchronous”。Win32 API 中的 SendMessage() 就是同步行为，而 PostMessage() 就是异步行为。如下图：



在 Windows 系统中，PostMessage() 是把消息放到对方的消息队列中，然后不管三七二十一，就回到原调用点继续执行，所以这是异步（asynchronous）行为。而 SendMessage() 根本就像是“直接调用窗口之窗口函数”，除非等该窗口函数结束，是不会回到原调用点的，所以它是同步（synchronous）行为。

Win32 中关于进程和线程的协调工作是由同步机制（synchronous mechanism）来完成的。同步机制相当于线程之间的红绿灯。你可以设计让一组线程使用同一个红绿灯系统。这个红绿灯系统负责给某个线程绿灯而给其他线程红灯。这一组红绿灯系统必须确保每一个线程都有机会获得绿灯。

有好多种同步机制可以运用。使用哪一种则完全视欲解决的问题而定。当我讨论每一种同步机制时，我会说明“何时”以及“为什么”应该使用它。

这些同步机制常常以各种方式组合在一起，以产生出更精密的机制。如果你把那些基本的同步机制视为建筑物的小件组块，你就能够设计出更适合你的特殊同步机制。

Critical Sections（关键区域、临界区域）

Win32 之中最容易使用的一个同步机制就是 critical sections．所谓 critical sections 意指一小块“用来处理一份被共享之资源”的程序代码．这里所谓的资源，并不是指来自 .RES（资源文件）的 Windows 资源，而是广义地指一块内存、一个数据结构、一个文件，或任何其他具有“使用之排他性”的东西．也就是说，“资源”每一次（同一时间内）只能够被一个线程处理．

你可能必须在程序的许多地方处理这一块可共享的资源．所有这些程序代码可以被同一个 critical section 保护起来．为了阻止问题发生，一次只能有一个线程获准进入 critical section 中（相对地也就是说资源受到了保护）．实施的方式是在程序中加上“进入”或“离开”critical section 的操作．如果有一个线程已经“进入”某个 critical section，另一个线程就绝对不能够进入同一个 critical section．

在 Win32 程序中你可以为每一个需要保护的资源声明一个 CRITICAL_SECTION 类型的变量．这个变量扮演红绿灯的角色，让同一时间内只有一个线程进入 critical section．

Critical section 并不是核心对象．因此，没有所谓 handle 这样的东西．它和核心对象不同，它存在于进程的内存空间中．你不需要使用像“Create”这样的 API 函数获得一个 critical section handle．你应该做的是将一个类型为 CRITICAL_SECTION 的局部变量初始化，方法是调用 InitializeCriticalSection()：

```
VOID InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);
```

参数

lpCriticalSection 一个指针，指向欲被初始化的 CRITICAL_SECTION 变量。这个变量应该在你的程序中定义。

返回值

此函数传回 void。

当你用毕 critical section 时，你必须调用 DeleteCriticalSection() 清除它。这个函数并没有“释放对象”的意义在里头，不要把它和 C++ 的 delete 运算符混淆了。

```
VOID DeleteCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);
```

参数

lpCriticalSection 指向一个不再需要的 CRITICAL_SECTION 变量。

返回值

此函数传回 void。

下面就是一个基本的调用程序，用来产生并摧毁一个 critical section。请注意：gCriticalSection 被声明在程序最上方，作为任一线程都可以使用的全局变量。

```
CRITICAL_SECTION gCriticalSection;

void CreateDeleteCriticalSection()
{
    InitializeCriticalSection(&gCriticalSection);
      /* Do something here */
    DeleteCriticalSection(&gCriticalSection);
}
```

一旦 critical section 被初始化，每一个线程就可以进入其中——只要它通过了 EnterCriticalSection() 这一关。

VOID EnterCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);

参数

lpCriticalSection 指向一个你即将锁定的 CRITICAL_SECTION 变量。

返回值

此函数传回 void。

当线程准备好要离开 critical section 时，它必须调用 LeaveCriticalSection()：

VOID LeaveCriticalSection(
LPCRITICAL_SECTION lpCriticalSection
);

参数

lpCriticalSection 指向一个你即将解除锁定的 CRITICAL_SECTION 变量。

返回值

此函数传回 void。

延续稍早我所举的例子，下面是使用我所产生之 critical section 的一个例子：

```
void UpdateData()
{
   EnterCriticalSection(&gCriticalSection);
      /* Update the resource */
   LeaveCriticalSection(&gCriticalSection);
}
```

你可能会发现，有好几个函数都需要进入同一个 critical section（以上例而言指的就是 gCriticalSection）中，它们都前后包夹着 Enter/Leave 函数，并使用相同的参数。你应该在每一个存取全局数据的地方使用 Enter/Leave 函数。有时候 Enter/Leave 甚至会在同一个函数中出现数次——如果这个函数需要很长的运行时间。

不知道你是否还记得第 1 章那个被破坏的链表（linked list）例子，问题在于，像 insert 和 add 这样的操作应该避免同时发生。让我们看看如何使用 critical sections 来阻止破坏的发生。每次处理链表，前后都必须包夹进入和离开 critical section 的操作。列表 4-1 是一个实例。

列表 4-1 链表，配合 critical section

```
#0001  typedef struct _Node
#0002  {
#0003     struct _Node *next;
#0004     int data;
#0005  } Node;
#0006
#0007  typedef struct _List
#0008  {
#0009     Node *head;
#0010     CRITICAL_SECTION critical_sec;
#0011  } List;
#0012
#0013  List *CreateList()
#0014  {
```

```
#0015     List *pList = (List *)malloc(sizeof(pist));
#0016     pList->head = NULL;
#0017     InitializeCriticalSection(&pList->critical_sec);
#0018     return pList;
#0019  }
#0020
#0021  void DeleteList(List *pList)
#0022  {
#0023     DeleteCriticalSection(&pList->critical_sec);
#0024     free(pList);
#0025  }
#0026
#0027  void AddHead(List *pList, Node *node)
#0028  {
#0029     EnterCriticalSection(&pList->critical_sec);
#0030     node->next = pList->head;
#0031     pList->head = node;
#0032     LeaveCriticalSection(&pList->critical_sec);
#0033  }
#0034
#0035  void Insert(List *pList, Node *afterNode, Node *newNode)
#0036  {
#0037     EnterCriticalSection(&pList->critical_sec);
#0038     if (afterNode == NULL) {
#0039        AddHead(pList, newNode);
#0040     }
#0041     else
#0042     {
#0043        newNode->next = afterNode->next;
#0044        afterNode->next = newNode;
#0045     }
#0046     LeaveCriticalSection(&pList->critical_sec);
#0047  }
#0048
#0049  Node *Next(List *pList, Node *node)
#0050  {
#0051     Node* next;
#0052     EnterCriticalSection(&pList->critical_sec);
#0053     next = node->next;
#0054     LeaveCriticalSection(&pList->critical_sec);
#0055     return next;
#0056  }
```

加上了额外的 critical section 操作之后，同一时间里最多就只有一个人能够读（或写）链表内容。请注意，我把 CRITICAL_SECTION 变量放在 List 结构之中。你也可以使用一个全局变量取代之，但我是希望每一个链表实体都能够独立地读写。如果只使用一个全局性 critical section，就表示一次只能读写一个链表，这会产生效率上的严重问题。

你或许纳闷，为什么 Next() 也需要环绕一个 critical section，毕竟它只是处理单一一个值而已。还记得吗，第 1 章曾经说过，return node->next 实际上被编译为数个机器指令，而不是一个“不可分割的操作”（所谓的 atomic operation）。如果我们在前后加上 critical section 的保护，就能够强迫该操作成为“不可分割的”。

上述程序代码存在着一个微妙点。在 Next() 离开 critical section 之后，但尚未 return 之前，没有什么东西能够保护这个 node 免受另一个线程的删除操作。这个问题可以靠更高阶的“readers/writers 锁定”解决之。我们将在第 7 章解释怎么做。

这个简短的例子也说明了 Win32 critical section 的另一个性质。一旦线程进入一个 critical section，它就能够一再地重复进入该 critical section。这也就是为什么 Insert() 可以调用 AddHead() 而不需先调用 LeaveCriticalSection() 的缘故。唯一的警告就是，每一个“进入”操作都必须有一个对应的“离开”操作。如果某个线程调用 EnterCriticalSection() 5 次，它也必须调用 LeaveCriticalSection() 5 次，该 critical section 才能够被释放。

最小锁定时间

在任何关于同步机制的讨论中，不论是在 Win32 或 Unix 或其他操作系统，你一定会一再地听到这样一条规则：

不要长时间锁住一份资源

如果你一直让资源被锁定，你就会阻止其他线程的执行，并把整个程序带到一个完全停止的状态。以 critical section 来说，当某个线程进入 critical section 时，该项资源即被锁定。

我们很难定义所谓“长时间”是多长。如果你在网络上进行操作，并且是在一个拨号网络上，长时间可能是指数分钟。如果你所处理的是应用程序的一项关键性资源，长时间可能是指数个毫秒（milliseconds）。

我能够给你的最牢靠而最立即的警告就是，千万不要在一个 critical section 之中调用 Sleep() 或任何 Wait...() API 函数。

当你以一个同步机制保护一份资源时，有一点必须常记在心，那就是：这项资源被使用的频率如何？线程必须多快释放这份资源，才能确保整个程序的运作很平顺？

? FAQ 14：如果线程在 critical sections 中停很久，会怎样？

某些人会关心这样的问题：如果我再也不释放资源（或不离开 critical section，或不释放 mutex……等等），会怎样？答案是：不会怎样！

操作系统不会当掉。用户不会获得任何错误信息。最坏的情况是，当主线程（一个 GUI 线程）需要使用这被锁定的资源时，程序会挂在那儿，动也不动。真的，同步机制并没有什么神奇魔法。

避免 Dangling Critical Sections

? FAQ 15：如果线程在 critical sections 中结束，会怎样？

Critical section 的一个缺点就是，没有办法获知进入 critical section 中的那个线程是生是死。从另一个角度看，由于 critical section 不是核心对象，如果进入 critical section 的那个线程结束了或当掉了，而没有调用 LeaveCriticalSection() 的话，系统没有办法将该 critical section 清除。如果你需要那样的机能，你应该使用 mutex（本章稍后将介绍 mutex）。

Jeffrey Richter 在他所主持的 Win32 Q&A 专栏（Microsoft Systems Journal，1996/07）中曾经提到过，Windows NT 和 Windows 95 在管理 dangling critical sections 时有极大的不同。在 Windows NT 之中，如果一个线程进入某个 critical section 而在未离开的情况下就结束，该 critical section 会被永远锁住。然而在 Windows 95 中，如果发生同样的事情，其他等着要进入该 critical section 的线程，将获准进入。这基本上是一个严重的问题，因为你竟然可以在你的程序处于不稳定状态时进入该 critical section。

死锁（Deadlock）

为每一个链表（linked list）准备一个 critical section 之后，我却开启了另一个问题。请看下面这个用来交换两个链表内容的函数：

```
void SwapLists(List *list, List *list2)
{
    List *tmp_list;
    EnterCriticalSection(list1->critical_sec);
    EnterCriticalSection(list2->critical_sec);
    tmp->list = list1->head;
    list1->head = list2->head;
    list2->head = temp->list;
    LeaveCriticalSection(list1->critical_sec);
    LeaveCriticalSection(list2->critical_sec);
}
```

看出问题了吗？假设下面两次调用发生在不同线程的同一个时间点：

线程 A　SwapLists(home_address_list, work_address_list);
线程 B　SwapLists(work_address_list, home_address_list);

而在线程 A 的 SwapLists() 的第一次 EnterCriticalSection() 之后，发生了 context switch（译注：也就是调度程序选换了一个线程），然后线程 B 执行

了它的 SwapLists()操作，两个线程于是会落入“我等你，你等我”的轮回。线程A需要 work_address_list，线程B需要 home_address_list，而双方都掌握有对方所要的东西。这种情况称为死锁（deadlock），或称为死亡拥抱（The Deadly Embrace）。

? FAQ 16：我如何避免死锁？

任何时候当一段代码需要两个（或更多）资源时，都有潜在性的死锁阴影。死锁的情况可能非常复杂，许多线程的独立性彼此纠缠在一起。虽然有一些算法可以侦测并仲裁死锁状态，基本上它们仍嫌过于复杂。对大部分程序而言，最好的政策就是找出一种方法以确保死锁不会发生。稍后你会看到，强迫将资源锁定，使它们成为 "all-or-nothing"（要不统统获得，要不统统没有），可以阻止死锁的发生。

哲学家进餐问题（The Dining Philosophers）

有一个很著名的死锁问题，就是所谓的“哲学家进餐问题”。虽然你可以在任何一本讲解操作系统或多线程的书籍中看到这个命题，但我想这里可以提供给你一个最好的讨论，因为我可以显示给你看这些哲学家真正的动作。你可以执行书附盘片中的 DINING 程序，看到哲学家们的行为。

哲学家进餐问题是这样子的：好几位哲学家围绕着餐桌坐，每一位哲学家要么思考，要么等待，要么就吃饭。为了吃饭，哲学家必须拿起两支筷子（分放于左右两端）。不幸的是，筷子的数量和哲学家相等，所以每支筷子必须由两位哲学家共享。图 4-1 显现出这种状态。

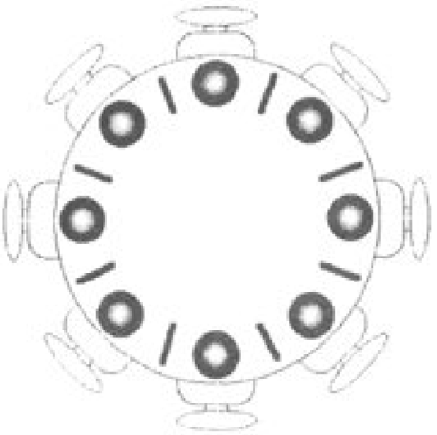
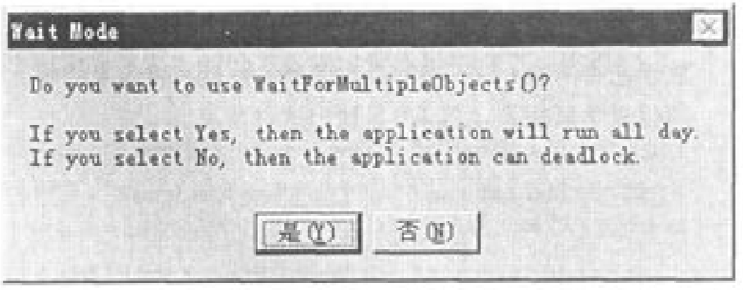


图 4-1　哲学家进餐问题的餐桌排列

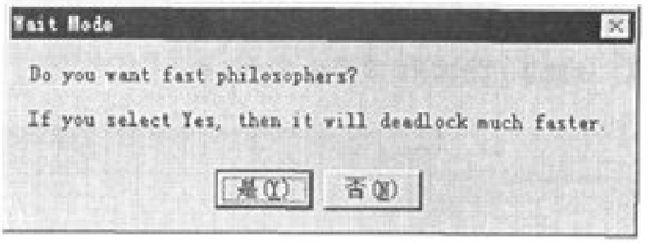
哲学家都是有点倔强的人，他们不愿意在吃完之前放下他们的筷子．因此，如果每位哲学家都抓住了左手边的筷子，他们就不可能抓到右手边的筷子，因为右边的哲学家正在使用那支筷子，而且拒绝出让．

书附盘片中的 DINING 程序，允许你指定该程序如何解决哲学家进餐问题（译注）．如果你允许死锁发生，哲学家会一次取得一支筷子，而当他手上只有一支筷子时，他就是处于等待状态．如果你不允许死锁发生，哲学家要么一次获得两支筷子，要么就什么都得不到．

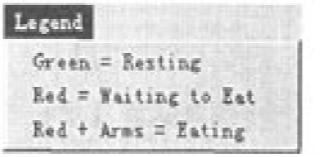
译注　DINING 程序一开始会问你这个问题：



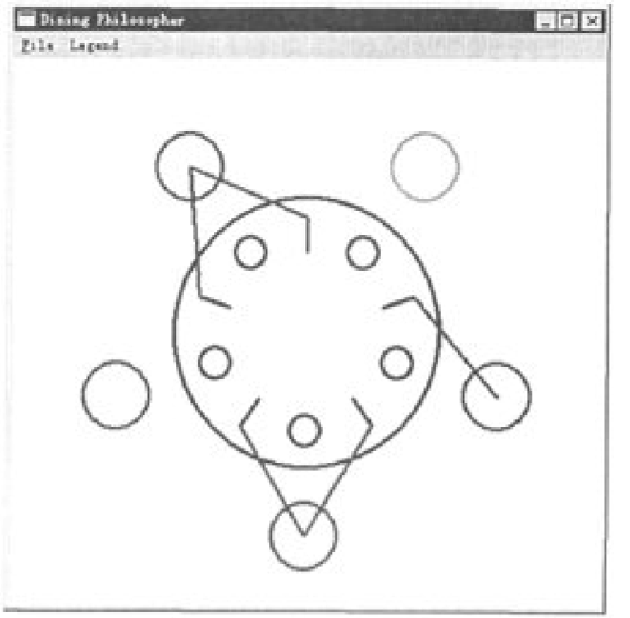
如果你回答 No，它会再问你一个问题：



然后才开始执行。拉下选单中的【Legend】，可以看到图例说明。

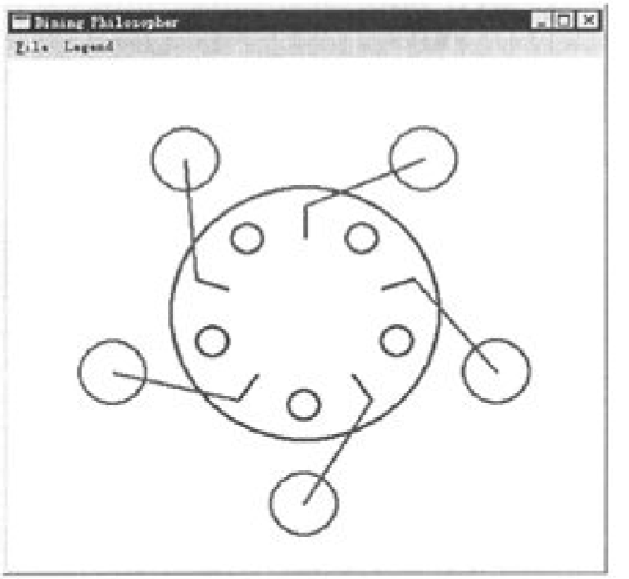


绿色表示休息，红色（拿到一支筷子）表示等着要吃东西了。红色加上两支筷子，表示开始吃东西。



执行 DINING 程序，你会看到哲学家通常都吃得到饭。但是当每一位哲学家手持一支筷子的情况发生时，死锁就出现了。

译注 这就是 DINING 程序的死锁状态：



? FAQ 17：我能够等待一个以上的 critical sections 吗？

在 SwapLists() 中所发生的死锁问题，是因为它必须等待两个 critical section．哲学家的死锁问题则是因为他们都必须等待两支筷子．在程序之中我们必须要求操作系统给我们一点帮助，如 critical sections 或其他什么东西．不幸的是，critical sections 没办法使用于这种情况．

顺带一提

这个问题也可以这样解决：使用一个 critical section 并允许一次只能够有一位哲学家拿起或放下筷子．这并不是最好的解决方法，因为它限制了哲学家的自由意志，同时也让他们做了非必要的等待．

还记得上一章最后提到的 WaitForMultipleObjects() 函数吗？它允许你对操作系统发出“等待”的要求，直到所有指定的对象都激发才返回．同一时

刻取得两支筷子而非一支筷子，是本案的关键。但是 WaitForMultipleObjects() 等待的只是核心对象，critical section 并不是核心对象（它没有 handle）。因此，我们必须寻求另一种 Win32 同步机制：mutex。

互斥器（Mutexes）

Win32 的 Mutex 用途和 critical section 非常类似，但是它牺牲速度以增加弹性。或许你已经猜到了，mutex 是 MUTual EXclusion 的缩写。一个时间内只能够有一个线程拥有 mutex，就好像同一时间内只能够有一个线程进入同一个 critical section 一样。

虽然 mutex 和 critical section 做相同的事情，但是它们的运作还是有差别的：

- 锁住一个未被拥有的 mutex，比锁住一个未被拥有的 critical section，需要花费几乎 100 倍的时间。因为 critical section 不需要进入操作系统核心，直接在“user mode”就可以进行操作。（译注：作者这里所谓的“user mode”，是相对于 Windows NT 的“kernel mode”而言。至于 Windows 95 底下，没有所谓“user mode”这个名词或观念，应该是指 ring3 层次。）
- Mutexes 可以跨进程使用。Critical section 则只能够在同一个进程中使用。
- 等待一个 mutex 时，你可以指定“结束等待”的时间长度。但对于 critical section 则不行。

以下是两种对象的相关函数比较：

CRITICAL_SECTION	Mutex 核心对象
InitializeCriticalSection()	CreateMutex()
	OpenMutex()
EnterCriticalSection()	WaitForSingleObject()
	WaitForMultipleObjects()
	MsgWaitForMultipleObjects()
LeaveCriticalSection()	ReleaseMutex()
DeleteCriticalSection()	CloseHandle()

为了能够跨进程使用同一个 mutex，你可以在产生 mutex 时指定其名称。如果你指定了名称，系统中的其他任何线程就可以使用这个名称来处理该 mutex。一定要使用名称，因为你没有办法把 handle 交给一个执行中的进程。

记住，其他程序也可能使用这个同步机制，所以 mutex 名称对整个系统而言是全局性的。不要把你的 mutex 对象命名为“Object”或“Mutex”之类，那太过普遍。请使用一些独一无二的名称，如公司名称或应用程序名称等等。

产生一个互斥器（Mutex）

与 critical sections 不同，当你产生一个 mutex 时，你有某些选择空间。Mutex 是一个核心对象，因此它被保持在系统核心之中，并且和其他核心对象一样，有所谓的引用计数（reference count）。虽然 mutex 的机能与 critical section 十分类似，但由于 Win32 术语带来的迷惑，mutex 可能不大容易了解。你可以利用 CreateMutex() 产生一个 mutex：

```
HANDLE CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitialOwner,
    LPCTSTR lpName
);
```

参数

lpMutexAttributes	安全属性。NULL 表示使用默认的属性。这一指定在 Windows 95 中无效。
bInitialOwner	如果你希望“调用 CreateMutex() 的这个线程”拥有产生出来的 mutex，就将此值设为 TRUE。

lpName	mutex 的名称（一个字符串）。任何进程或线程都可以根据此名称使用这一 mutex。名称可以是任意字符串，只要不含反斜线（backslash，\）即可。

返回值

如果成功，则传回一个 handle，否则传回 NULL。调用 GetLastError() 可以获得更进一步的信息。如果指定的 mutex 名称已经存在，GetLastError() 会传回 ERROR_ALREADY_EXISTS。

当你不再需要一个 mutex 时，你可以调用 CloseHandle() 将它关闭。和其他核心对象一样，mutex 有一个引用计数（reference count）。每次你调用 CloseHandle()，引用计数便减 1。当引用计数达到 0 时，mutex 便自动被系统清除掉。下面这个 CreateAndDeleteMutex() 函数会先产生一个 mutex 然后把它删除。这个 mutex 没有安全属性，不属于现行线程，名为“Demonstration Mutex”。

```
HANDLE hMutex;

void CreateAndDeleteMutex()
{
   hMutex = CreateMutex(NULL, FALSE, "Demonstration Mutex");
      /* Do something here */
   CloseHandle(hMutex);
}
```

打开一个互斥器（Mutex）

如果 mutex 已经被产生了，并有一个名称，那么任何其他的进程和线程便可以根据该名称打开那个 mutex（我这里并不考虑安全属性）。

如果你调用 CreateMutex() 并指定一个早已存在的 mutex 名称，Win32 会回给你一个 mutex handle，而不会为你产生一个新的 mutex。就像上面所说的，GetLastError() 会传回 ERROR_ALREADY_EXISTS。

你也可以使用 OpenMutex() 打开（而非产生）一个原已存在的 mutex。这种情况通常是因为，你写了一个 client 进程，并与同一台机器上的 server 进程交谈，而只有 server 进程才应该产生 mutex，因为它保护了 server 所定义的结构体。

关于 OpenMutex()，请参阅 Win32 Programmer's Reference。你也可以在 Visual C++ 的联机帮助文件中找到相关资料。

锁住一个互斥器（Mutex）

欲获得一个 mutex 的拥有权，请使用 Win32 的 Wait...() 函数。Wait...() 对 mutex 所做的事情和 EnterCriticalSection() 对 critical section 所做的事情差不多，倒是一大堆术语容易把你迷惑了。

一旦没有任何线程拥有 mutex，这个 mutex 便处于激发状态。因此，如果没有任何线程拥有那个 mutex，Wait...() 便会成功。反过来说，当线程拥有 mutex 时，它便不处于激发状态。如果有某个线程正在等待一个未被激发的 mutex，它便将进入“blocking”（阻塞）状态。也就是说，该线程会停止执行，直到 mutex 被其拥有者释放并处于激发状态。

下面是某种情节的发展：

1. 我们有一个 mutex，此时没有任何线程拥有它，也就是说，它处于非激发状态（译注）。
2. 某个线程调用 WaitForSingleObject()（或任何其他的 Wait...() 函数），并指定该 mutex handle 为参数。
3. Win32 于是将该 mutex 的拥有权给予这个线程，然后将此 mutex 的状态短暂地设为激发状态，于是 Wait...() 函数返回。
4. Mutex 立刻又被设定为非激发状态，使任何处于等待状态下的其他线程没有办法获得其拥有权。
5. 获得该 mutex 之线程调用 ReleaseMutex()，将 mutex 释放掉。于是循环回到第一场景，周而复始。

译注 我想你很容易被作者的上一段文字迷惑，因为它的第一点和更前一段文字中的“一旦没有任何线程拥有 mutex，这个 mutex 便处于激发状态”有点背道而驰。基本上，或许更精密地说，所谓的“mutex 激发状态”应该是：当没有任何线程拥有该 mutex 而且有一个线程正以 Wait...() 等待该 mutex，该 mutex 就会短暂地出现激发状态，使 Wait...() 得以返回。

ReleaseMutex() 的规格如下：

```
BOOL ReleaseMutex(
    HANDLE hMutex
);
```

参数

hMutex 欲释放之 mutex 的 handle。

返回值

如果成功，传回 TRUE。如果失败，传回 FALSE。

Mutex 的拥有权是第二个容易引人迷惑的地方。Mutex 的拥有权并非属于那个产生它的线程，而是那个最后对此 mutex 进行 Wait...() 操作并且尚未进行 ReleaseMutex() 操作的线程。线程拥有 mutex 就好像线程进入 critical section 一样。一次只能有一个线程拥有该 mutex。

Mutex 的被摧毁和其拥有权没有什么关系。和大部分其他的核心对象一样，mutex 是在其引用计数降为 0 时被操作系统摧毁的。每当线程对此 mutex 调用一次 CloseHandle()，或是当线程结束时，mutex 的引用计数即下降 1。

如果拥有某 mutex 之线程结束了，该 mutex 会被自动清除的唯一情况是：此线程是最后一个与该 mutex handle 有关联的线程。否则此核心对象的引用计数仍然是比 0 大，其他线程（以及进程）仍然可以拥有此 mutex 的合法 handle。然而，当线程结束而没有释放某个 mutex 时，有一种特殊的处理方式。

处理被舍弃的互斥器（Mutexes）

在一个适当的程序中，线程绝对不应该在它即将结束前还拥有一个mutex，因为这意味着线程没有能够适当地清除其资源。不幸地是，我们并不身处在一个完美的世界，有时候，因为某种理由，线程可能没有在结束前调用ReleaseMutex()。为了解决这个问题，mutex 有一个非常重要的特性。这性质在各种同步机制中是独一无二的。如果线程拥有一个 mutex 而在结束前没有调用 ReleaseMutex()，mutex 不会被摧毁。取而代之的是，该 mutex 会被视为“未被拥有”以及“未被激发”，而下一个等待中的线程会被以WAIT_ABANDONED_0 通知。不论线程是因为 ExitThread() 而结束，或是因当掉而结束，这种情况都存在。

如果其他线程正以 WaitForMultipleObjects() 等待此 mutex，该函数也会返回，传回值介于 WAIT_ABANDONED_0 和 (WAIT_ABANDONED_0_n+1) 之间，其中的 n 是指 handle 数组的元素个数。线程可以根据这个值了解到究竟哪一个 mutex 被放弃了。至于 WaitForSingleObject()，则只是传回WAIT_ABANDONED_0。

“知道一个 mutex 被舍弃”是一件简单的事情，但要知道如何应对可就比较困难了。毕竟 mutex 是用来确保某些操作能够自动被进行的，如果线程死于半途，很有可能被保护的那些数据会受到无法修复的伤害。

让哲学家们进餐

让我们回头重新看看哲学家进餐问题。在范例程序 DINING 中产生一组mutexes 用来表示那些筷子。产生 mutexes 的程序操作像这样：

```
for (i=0; i<PHILOSOPHERS; i++)
    gChopStick[i] = CreateMutex(NULL, FALSE, NULL);
```

CreateMutex() 的参数告诉我们，这些 mutexes 的安全属性采用缺省值，没有初始拥有者，也没有名称。每一支筷子有一个 mutex 对应之。我

们之所以使用未具名的 mutexes，为的是筷子数组是全局数据，每一个线程都能够存取它。

就像 critical sections 一样，mutexes 用来保护资源。在存取一个受保护的资源时，你的程序代码必须收到 mutex 的拥有权——藉由调用 Wait...() 函数获得。

哲学家们可以使用 WaitForSingleObject() 来等待吃饭，但那可就像 critical section 一样了（同时也带来相同的死锁问题）。他们也可以使用 WaitForMultipleObjects() 来等待，于是可以修正因 EnterCriticalSection() 和 WaitForSingleObject() 而造成的死锁问题。

实际上我们是使用 WaitForMultipleObjects()来等待两支筷子。如果只有一支筷子可用，不算是取得一“双”筷子（WaitForMultipleObjects() 也因此不会返回）。程序代码像这样：

```
WaitForMultipleObjects(2, myChopsticks, TRUE, INFINITE);
```

这个函数的参数告诉我们，myChopsticks 数组中有两个 handles 是等待目标。当其中每一个 handle 都处于激发状态时，该函数才会返回。它会无穷尽地等待下去，没有时间限制。

如果你以 WaitForMultipleObjects() 的方式执行 DINING 程序，你会发现哲学家们能够持续地吃，死锁永远不会发生。

修正 SwapLists

我们用于解决哲学家进餐问题的技术，也可以用来解决我们在 SwapLists() 所遭遇的问题。任何时候只要你想锁住超过一个以上的同步对象，你就有死锁的潜在病因。如果总是在相同时间把所有对象都锁住，问题可去矣。列表 4-2 显示新版的 SwapLists()。

列表 4-2 使用 WaitForMultipleObjects() 修正 SwapLists

```
#0001  struct Node
#0002  {
#0003      struct Node *next;
#0004      int data;
#0005  };
#0006
#0007  struct List
#0008  {
#0009      struct Node *head;
#0010      HANDLE hMutex;
#0011  };
#0012
#0013  struct List *CreateList()
#0014  {
#0015      List *list = (List *)malloc(sizeof(struct List));
#0016      list->head = NULL;
#0017      list->hMutex = CreateMutex(NULL, FALSE, NULL);
#0018      return list;
#0019  }
#0020
#0021  void DeleteList(struct List *list)
#0022  {
#0023      CloseHandle(list->hMutex);
#0024      free(list);
#0025  }
#0026
#0027  void SwapLists(struct List *list, struct List *list2)
#0028  {
#0029      struct List *tmp_list;
#0030      HANDLE arrhandles[2];
#0031
#0032      arrhandles[0] = list1->hMutex;
#0033      arrhandles[1] = list2->hMutex;
#0034      WaitForMultipleObjects(2, arrHandles, TRUE, INFINITE);
#0035      tmp_list = list1->head;
#0036      list1->head = list2->head;
#0037      list2->head = tmp_list;
#0038      ReleaseMutex(arrhandles[0]);
#0039      ReleaseMutex(arrhandles[1]);
#0040  }
```

为什么有一个最初拥有者？

CreateMutex() 的第二个参数 bInitialOwner，允许你指定现行线程（current thread）是否立刻拥有即将产生出来的 mutex。乍见之下这个参数或许只是提供一种方便性，但事实上它阻止了一种 race condition 的发生。

与 critical section 不同，mutexes 可以跨进程使用，以及跨线程使用。Mutex 可以根据其名称而被开启。所以，另一个进程可以完全不需要和产生 mutex 的进程打声招呼，就根据名称开启一个 mutex。如果没有 bInitialOwner，你就必须写下这样的代码：

```
HANDLE hMutex = CreateMutex(NULL, FALSE, "Sample Name");
int result = WaitForSingleObject(hMutex, INFINITE);
```

但是这样的安排可能会产生 race condition。如果在 CreateMutex 完成之后，发生了一个 context switch，执行权被切换到另一个线程，那么其他进程就有可能在 mutex 的产生者调用 WaitForSingleObject() 之前，锁住这个 mutex 对象。

信号量（Semaphores）

许多文件中都会提到 semaphores（信号量），因为在电脑科学中它是最具历史的同步机制。它可以让你陷入理论的泥淖之中，教授们则喜欢问你一些有关于信号量的疑难杂症。你可能不容易找到一些关于 semaphores 的有用例子，但是我告诉你，它是解决各种 producer/consumer 问题的关键要素。这种问题会存有一个缓冲区，可能在同一时间内被读出数据或被写入数据。

Win32 中的一个 semaphore 可以被锁住最多 n 次，其中 n 是 semaphore 被产生时指定的。n 常常被设计用来代表“可以锁住一份资源”的线程个数，不过并非单独一个线程就不能够拥有所有的锁定。这没有什么理由可言。

这里有一个例子，告诉你为什么你需要一个 semaphore。考虑一下某人（我称之为 Steve）的情况。他想在加州租一辆车。租车店柜台后面坐了好几位租车代理人。Steve 告诉租车代理人说他想要一部敞篷车，接待他的那位代理人往窗外一看，有三辆敞篷车可以用，于是开始写派车单。不幸的是，就那么巧，有另三个人也同时要一辆敞篷车，而他们的租车代理人也正在做 Steve 的代理人的相同动作。现在，有四个人想租三辆车，而必然有某个人要被淘汰出局。

让我们留下这小小的悬疑画面，并祈祷 Steve 租得到车。租车公司这边的问题是，他们不可能即时写下派车单并且马上给租车人钥匙。整个租车程序过长，长到足够让另一位代理人把同一辆车租给另一个人。这种情况我们已经在多线程的情况下一再地看到了。如果有许多个线程正在处理相同的资源，那么必须有某些机制被用来阻止线程干扰其他线程。

如果我们尝试写一个程序解决汽车出租问题，方法之一就是为每辆车加上一个 mutex 保护之。这虽然可行，但你可能得为一家大租车公司生产成百甚至成千个 mutexes。

另一个方法就是以单一的 mutex 为所有车辆服务，或说为所有的敞篷车服务，但这样的话一次就只能有一个店员出租敞篷车。这或许可以减少店员人数，但是面对一家忙碌的出租公司，客户可能因此转移到其竞争对手那里去。

解决之道是：首先，所有的敞篷车都被视为相同（是啊，什么时候你租车还选颜色的？），在钥匙被交到客户手上之前，唯一需要知道的就是现在有几辆车可以用。我们可以用 semaphore 来维护这个数字，并保证不论是增加或减少其值，都是在一个不可分割的动作内完成。当 semaphore 的数值降为 0 时，不论什么人要租车，就得等待了。

理论可以证明，mutex 是 semaphore 的一种退化。如果你产生一个 semaphore 并令最大值为 1，那就是一个 mutex。也因此，mutex 又常被称为 binary semaphore。如果某个线程拥有一个 binary semaphore，那么就没有其他线程能够获得其拥有权。在 Win32 中，这两种东西的拥有权（ownership）的意义完全不同，所以它们不能够交换使用。semaphores 不像 mutexes，它并没有所谓的“wait abandoned”状态可以被其他线程侦测到。

在许多系统中，semaphores 常被使用，因为 mutexes 可能并不存在。在 Win32 中 semaphores 被使用的情况就少得多，因为 mutex 存在的缘故。

产生信号量（Semaphore）

要在 Win32 环境中产生一个 semaphore，必须使用 CreateSemaphore() 函数调用：

```
HANDLE CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpAttributes,
    LONG lInitialCount,
    LONG lMaximumCount,
    LPCTSTR lpName
);
```

参数

lpAttributes	安全属性。如果是 NULL 就表示要使用默认属性。Windows 95 忽略这一参数。
lInitialCount	semaphore 的初值。必须大于或等于 0，并且小于或等于 lMaximumCount。
lMaximumCount	Semaphore 的最大值。这也就是在同一时间内能够锁住 semaphore 之线程的最多个数。
lpName	Semaphore 的名称（一个字符串）。任何线程（或进程）都可以根据这一名称引用到这个 semaphore。这个值可以是 NULL，意思是产生一个没有名字的 semaphore。

返回值

如果成功就传回一个 handle，否则传回 NULL。不论哪一种情况，GetLastError() 都会传回一个合理的结果。如果指定的 semaphore 名称已经存在，则该函数还是成功的，GetLastError() 会传回 ERROR_ALREADY_EXISTS。

获得锁定

Semaphore 的各个相关术语，其晦涩比起 mutexes 真是有过之而无不及。首先请你了解，semaphore 的现值代表的意义是目前可用的资源数。如果 semaphore 的现值为 1，表示还有一个锁定动作可以成功。如果现值为 5，就表示还有五个锁定动作可以成功。

每当一个锁定动作成功，semaphore 的现值就会减 1。你可以使用任何一种 Wait...() 函数（例如 WaitForSingleObject()）要求锁定一个 semaphore。因此，如果 semaphore 的现值不为 0，Wait...() 函数会立刻返回。这和 mutex 很像，如果没有任何线程拥有 mutex，Wait...() 会立刻返回。

? FAQ 18：谁才拥有 semaphore?

如果锁定成功，你也不会收到 semaphore 的拥有权。因为可以有一个以上的线程同时锁定一个 semaphore，所以谈 semaphore 的拥有权并没有太多帮助。在 semaphore 身上并没有所谓“独占锁定”这种事情。也因为没有拥有权的观念，一个线程可以反复调用 Wait...() 函数以产生新的锁定。这和 mutex 绝不相同：拥有 mutex 的线程不论再调用多少次 Wait...() 函数，也不会被阻塞住。

一旦 semaphore 的现值降到 0，就表示资源已经耗尽。此时，任何线程如果调用 Wait...() 函数，必然要等待，直到某个锁定被解除为止。

解除锁定（Releasing Locks）

为了解除锁定，你必须调用 ReleaseSemaphore()。这个函数将 semaphore 的现值增加一个定额，通常是 1，并传回 semaphore 的前一个现值。

ReleaseSemaphore() 和 ReleaseMutex() 旗鼓相当。当你调用 WaitForSingleObject() 并获得一个 semaphore 锁定之后，你就需要调用 ReleaseSemaphore()。Semaphore 常常被用来保护固定大小的环状缓冲区（ring buffer）。程序如果要读取环状缓冲区的内容，必须等待 semaphore。

线程将数据写入环状缓冲区，写入的数据可能不只一笔，在这种情况下解除锁定时的 semaphore 增额应该等于写入的数据笔数。

```
BOOL ReleaseSemaphore(
        HANDLE hSemaphore,
        LONG lReleaseCount,
        LPLONG lpPreviousCount
    );
```

参数

hSemaphore	Semaphore 的 handle。
lReleaseCount	Semaphore 现值的增额。该值不可以是负值或 0。
lpPreviousCount	藉此传回 semaphore 原来的现值。

返回值

如果成功，则传回 TRUE。否则传回 FALSE。失败时可调用 GetLastError() 获得原因。

ReleaseSemaphore() 对于 semaphore 所造成的现值的增加，绝对不会超过 CreateSemaphore() 时所指定的 lMaximumCount。

请记住，lpPreviousCount 所传回来的是一个瞬间值。你不可以把 lReleaseCount 加上 *lpPreviousCount，就当作是 semaphore 的现值，因为其他线程可能已经改变了 semaphore 的值。

与 mutex 不同的是，调用 ReleaseSemaphore() 的那个线程，并不一定就得是调用 Wait...() 的那个线程。任何线程都可以在任何时间调用 ReleaseSemaphore()，解除被任何线程锁定的 semaphore。

为什么 semaphore 要有一个初值

CreateSemaphore() 的第二个参数是 lInitialCount，它的存在理由和 CreateMutex() 的 bInitialOwner 参数的存在理由是一样的。如果你把初值设定为 0，你的线程就可以在产生 semaphore 之后进行所有必要的初始化工作。待初始化工作完成后，调用 ReleaseSemaphore() 就可以把现值增加到其最大可能值。

以环状缓冲区（ring buffer）为例，semaphore 通常被产生时是以 0 为初值，所以任何一个等待中的线程都会停下来。一旦有东西被加到环状缓冲区中，我们就以 ReleaseSemaphore() 增加 semaphore 的值，于是等待中的线程就可以继续进行。

如果“将数据写入环状缓冲区”的那个线程，在它（或任何其他线程）调用 Wait...() 函数之前，先调用 ReleaseSemaphore()，会出现想象不到的结果。就某种意义而言，这就完全退缩到 mutex 的运作情况了。

事件（Event Objects）

Win32 中最具弹性的同步机制就属 events 对象了。Event 对象是一种核心对象，它的唯一目的就是成为激发状态或未激发状态。这两种状态全由程序来控制，不会成为 Wait...() 函数的副作用。

? FAQ 19: Event object 有什么用途？

Event 对象之所以有大用途，正是因为它们的状态完全在你掌控之下。Mutexes 和 semaphores 就不一样了，它们的状态会因为诸如 WaitForSingleObject() 之类的函数调用而变化。所以，你可以精确告诉一个 event 对象做什么事，以及什么时候去做。

Event 对象被运用在多种类型的高级 I/O 操作中。你可以在第 6 章看到一些例子。Event 对象也可以用来设计你自己的同步对象。

为了产生一个 event 对象，你必须调用 CreateEvent()：

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    BOOL bManualReset,
    BOOL bInitialState,
    LPCTSTR lpName
);
```

参数

lpEventAttributes	安全属性。NULL 表示使用默认属性。该属性在 Windows 95 中会被忽略。
bManualReset	如为 FALSE，表示这个 event 将在变成激发状态（因而唤醒一个线程）之后，自动重置（reset）为非激发状态。如果是 TRUE，表示不会自动重置，必须靠程序操作（调用 ResetEvent()）才能将激发状态的 event 重置为非激发状态。
bInitialState	如为 TRUE，表示这个 event 一开始处于激发状态。如为 FALSE，则表示这个 event 一开始处于非激发状态。
lpName	Event 对象的名称。任何线程或进程都可以根据这个文字名称，使用这一 event 对象。

返回值

如果调用成功，会传回一个 event handle，GetLastError() 会传回 0。如果 lpName 所指定的 event 对象已经存在，CreateEvent() 传回的是该 event handle，而不会产生一个新的。这时候 GetLastError() 会传回 ERROR_ALREADY_EXISTS。如果 CreateEvent() 失败，传回的是 NULL，GetLastError() 可以获得更进一步的失败信息。

书附盘片中的范例程序名为 EVENTTST。它会产生三个线程，其任务就是等待一个 event 对象，并且在被唤醒（也就是 event 呈现激发状态）时告诉我们。至于 event 对象的状态，则完全由我们（用户）来控制。这个程序的大体结构是，先产生 event 对象，再产生三个线程，然后等你按下按钮。

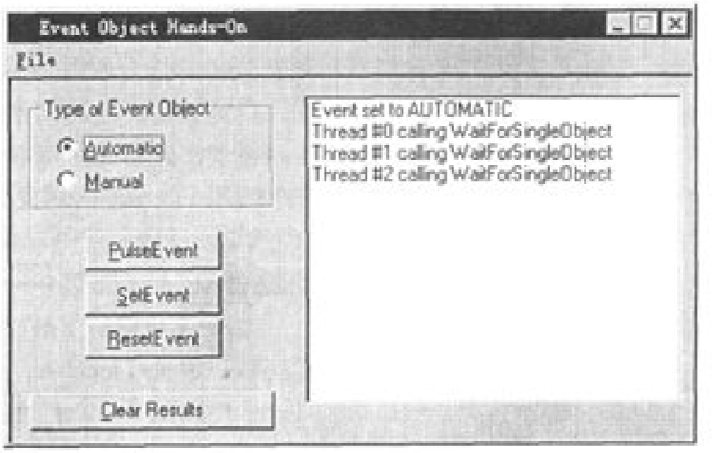
如果你按的是【SetEvent】按钮或【ResetEvent】按钮或【PulseEvent】按钮，EVENTTST 会调用对应的 API 函数。这些函数稍后会有比较详细的讨论，下面是一个简短的说明。

函数	说明
SetEvent()	把 event 对象设为激发状态
ResetEvent()	把 event 对象设为非激发状态（译注：在此我要提醒读者，"Reset" 的意思是“设定为非激发状态”，而非“重新设定为激发状态”。）
PulseEvent()	如果是一个 Manual Reset Event：把 event 对象设为激发状态，唤醒“所有”等待中的线程，然后 event 恢复为非激发状态。如果是一个 Auto Reset Event：把 event 对象设为激发状态，唤醒“一个”等待中的线程，然后 event 恢复为非激发状态

如果你选择一个新的 event 类型（若不是“Automatic”就是“Manual”），原有的 event 对象和线程统统会被摧毁，程序重新产生出新的 event 和新的线程。

使用各种设定所造成的程序行为，非常富于教育性。请你尝试针对“Automatic”和“Manual”两种 event，分别按下三个按钮，观察其行为。

译注 以下是 EVENTTST 的执行画面：



如果你选择的是“Automatic”，则 event 对象总是处于非激发状态，所以你按下【ResetEvent】不会产生什么效果。但按下【SetEvent】和【PulseEvent】会唤醒一个等待中的线程。

如果你选择的是“Manual”，event 对象的状态可能是激发，也可能是非激发，视上一次调用的是 SetEvent() 或 ResetEvent() 而定。按下【SetEvent】会使得每一个等待中的线程立刻苏醒，所以你会在执行画面上看到不断有等待、苏醒、等待、苏醒……的信息跑出来。按下【PulseEvent】会使得目前等待中的所有线程苏醒过来（随后立刻又进入等待状态）。

从这个程序的执行，我们可以发现另一个重点：操作系统会强迫让等待中的线程有轮番更替的机会。对于本章介绍的所有同步机制，这都是一个重要的行为。如果操作系统没有强迫实现某种层次的公平性，可能会有某个线程不断获得执行机会，而某个线程一直未能获得 CPU 的青睐。这种情况被称为 starvation（饥饿）。

? FAQ 20：如果我对着一个 event 对象调用 PulseEvent() 并且没有线程正在等待，会怎样？

如果你面对一个 AutoReset event 对象调用 SetEvent() 或 PulseEvent()，而彼时并没有任何线程正在等待，会怎样？EVENTTST 程序并没有实地论证这一点。这种情况下这个 event 会被遗失。换句话说，除非有线程正在等待，否则 event 不会被保存下来。这样的行为使得“要求苏醒”的请求颇容易被遗失掉。例如，线程 A 累加一个计数器，之后调用 WaitForSingleObject() 等待一个 event 对象。如果在这些动作之间发生 context switch，线程 B 起而执行，它检查计数器内容然后对着同一个 event 对象调用 PulseEvent()。这时候这个“要求苏醒”的请求会遗失掉，因为这个 pulse 不会被储存起来（译注：因为还没有线程处于等待状态嘛）。

另一种情况可能会引起死锁。假设“receiver”线程检查队列中是否有字符，这时候发生 context switch，切换到“sender”线程，它对一个 event 对象进行 pulse 操作，这时候又发生 context switch，回到 receiver 线程，调用 WaitForSingleObject()，等待 event 对象。由于这个动作发生在 sender 线程激发 event 之后，所以 event 会遗失，于是 receiver 永远不会醒来，程序进入死锁状态。这正是 semaphore 之所以被创造用以解决问题的地方。

从 Worker 线程中显示输出

此刻我想先打个岔，请各位看看 EVENTTST 如何让 worker 线程（那三个等待中的线程）把字符串放到列表框（listbox）中。列表框的消息循环总是被单一线程（也就是程序的主线程）掌管，虽然这并非绝对必要，但是让主线程负责所有的屏幕更新工作，是相当理想的。

我在程序中定义了一个消息，名为 WM_PLEASE_UPDATE。当 worker 线程认为需要把一笔新的项目放到列表框中时，就送这个消息给主线程。Worker 线程使用 SendMessage() 完成这件事情，以便制造出一种“函数调用”的效果。在主线程处理完毕该消息之前，SendMessage() 不会返回，所以我们可以保证所有的输出有条不紊，不至于乱了次序。

请注意，我一直仰赖一件事实：所有的数据可以被所有的线程取用。我使用 sprintf() 在线程的堆栈中产生一个字符串，然后将此字符串地址以 SendMessage() 送出。主线程在更新列表框的画面时，即使用到这个地址，一旦主线程完成这个消息的处理，SendMessage() 便返回，worker 线程于是继续进行下去。

想象一下，如果我以 PostMessage() 代替 SendMessage()，会发生什么情况？由于 PostMessage() 会立刻返回，所以当主线程抓取字符串内容要显示时，或许该字符串内容早已又被 worker 线程改写了。这就是多线程序设计中最常见的一种两难取舍：在最佳速度和最佳安全性之间取舍。在这里我宁愿选择比较慢但是比较安全的做法。

Interlocked Variables

同步机制的最简单类型是使用 interlocked 函数，对着标准的 32 位变量进行操作。这些函数并没有提供“等待”机能，它们只是保证对某个特定变量的存取操作是“一个一个接顺序来”。稍后我会把这些 interlocked 函数展示出来，因为唯有你自己亲身比较它们和其他同步机制的差异，才能够了解它们的用途。

考虑一下，如果你需要维护一个 32 位计数器的“排他性存取”性质，你该怎么做。你可能会想产生一个 critical section 或一个 mutex，拥有它，然后进行你的操作，然后再释放拥有权。一个 32 位变量的存取操作只需要 2~3 个机器指令，因此上述的准备动作实在是太多了些，几乎呈现两个 order 的倍数。

类似的 32 位计数器发生在所谓的引用计数（reference counting）身上，例如系统核心对于核心对象之 handle 的处理。基本上当一个核心对象的引用计数降为 0 时，这个对象就应该被摧毁。你可以“要么降低其引用计数值”，“要么判断它是否等于 0”，但是没办法两者并行。

InterlockedDecrement() 可以双效合一，它先将计数器内容减 1，再将其值与 0 做比较，并且传回比较结果。

所谓的 interlocked 函数，共有两个：

- InterlockedIncrement()
- InterlockedDecrement()

这两个函数都只能够和 0 做比较，不能和任何其他数值比较。

```
LONG InterlockedIncrement(
    LPLONG lpTarget
);

LONG InterlockedDecrement(
    LPLONG lpTarget
);
```

参数

lpTarget 32 位变量的地址。这个变量内容将被递增或递减，结果将与 0 作比较。这个地址必须指向 long word。

返回值

变量值经过运算（加 1 或减 1）后，如果等于 0，传回 0；如果大于 0，传回一个正值；如果小于 0，传回一个负值。

Interlocked...() 函数的传回值代表计数器和 0 的比较结果。这一点对于实现我们曾经提过的所谓“引用计数”（reference counting）非常重要，因为我们必须知道“引用计数”何时到达 0。如果没有这个比较，问题就回到了原点，你必须在增减操作之前先锁定该计数器，以使增减操作成为一个“不可切割”的操作。

对专家而言…

Windows 3.x 的确支持抢先式多任务，但程序员却不可得之。由于 DOS 程序毫无共享观念，多任务是让它们不得擅专整部机器的唯一方法。而所有 Windows 程序则被"放在一起，视为单一的 DOS 程序"。所以 Windows 3.x 对 DOS 程序的确是抢先式多任务，但对于 Windows 程序则不是。

如果你使用新版 OLE（搭配 apartment model 或 free threading model 的那种），你应该使用 Interlocked...() 函数来维护你的对象的引用计数。请在 AddRef() 之中调用 InterlockedIncrement() 并且在 Release() 之中调用 InterlockedDecrement()。

InterlockedExchange() 可以设定一个新值并传回旧值。就像 Increment/Decrement 函数一样，它提供了一个在多线程环境下的安全做法，用以完成一个很基础的运算操作。

```
LONG InterlockedExchange(
    LPLONG lpTarget,
    LONG lValue
);
```

参数

lpTarget 32 位变量的地址。这个指针必须指向 long word。

lValue 用以取代 lpTarget 所指内容之新值。

返回值

传回先前由 lpTarget 所指之内容。

同步机制摘要

Critical Section

Critical section（临界区）用来实现“排他性占有”。适用范围是单一进程的各线程之间。它是：

- 一个局部性对象，不是一个核心对象。
- 快速而有效率。
- 不能够同时有一个以上的 critical section 被等待。
- 无法侦测是否已被某个线程放弃。

Mutex

Mutex 是一个核心对象，可以在不同的线程之间实现“排他性占有”，甚至即使那些线程分属不同进程。它是：

- 一个核心对象。
- 如果拥有 mutex 的那个线程结束，则会产生一个 “abandoned” 错误信息。
- 可以使用 Wait...() 等待一个 mutex。
- 可以具名，因此可以被其他进程开启。
- 只能被拥有它的那个线程释放（released）。

Semaphore

Semaphore 被用来追踪有限的资源。它是：

- 一个核心对象。
- 没有拥有者。

- 可以具名，因此可以被其他进程开启。
- 可以被任何一个线程释放（released）。

Event Object

Event object 通常使用于 overlapped I/O（第 6 章），或用来设计某些自定义的同步对象。它是：

- 一个核心对象。
- 完全在程序掌控之下。
- 适用于设计新的同步对象。
- “要求苏醒”的请求并不会被储存起来，可能会遗失掉。
- 可以具名，因此可以被其他进程开启。

Interlocked Variable

如果 Interlocked...() 函数被使用于所谓的 spin-lock，那么它们只是一种同步机制。所谓 spin-lock 是一种 busy loop，被预期在极短时间内执行，所以有最小的额外负担（overhead）。系统核心偶尔会使用它们。除此之外，interlocked variables 主要用于引用计数。它们：

- 允许对 4 字节的数值有些基本的同步操作，不需动用到 critical section 或 mutex 之类。
- 在 SMP（Symmetric Multi-Processors）操作系统中亦可有效运作。

不要让线程成为脱缰野马

Keeping Your Threads on a Leash

这一章描述如何初始化一个新线程，如何停止一个执行中的线程，以及如何了解并调整线程优先权。

读过这一章之后，你将有能力回答一个 Win32 多线程程序设计的最基本问题。你一定曾经在 Usenet 的 Win32 论坛中一再地看过这个问题。当我开始在 Win32 上使用线程时，这个问题就一直在折磨我。我花了数天甚至数周的时间来寻找答案，并且希望找到的是一个好答案。

这个问题是：

我如何在某个线程内终止另一个正在运行的线程？

我们将在这一章看到答案，以及诸如“在一个线程中控制其他线程”等等类似问题的答案。

干净地终止一个线程

我曾经在第 2 章产生一个后台线程，用以输出一张屏幕外的 bitmap 图。我们必须解决的一个最复杂的问题就是，如果用户企图结束程序，而这张 bitmap 图尚未完成，怎么办？第 2 章的一个鸵鸟做法就是在任何 worker 线程还没完成其工作之前，不准用户结束程序。只要修改主消息循环，使消息循环不得在任何一个 worker 线程尚未结束之前结束，即可办到。这种做法的最大优点就是“简单”，但万一 bitmap 图十分复杂，需要很长的工作时间，那么程序有可能看起来像是“挂”了一样。

利用 TerminateThread() 放弃一个线程

这正是 Win32 程序设计的一般性问题。我如何能够安全地关闭任何执行中的线程呢？最明显的答案就是利用 TerminateThread()：

```
BOOL TerminateThread(
    HANDLE hThread,
    DWORD dwExitCode
);
```

参数

hThread	欲令其结束之线程的 handle。该线程就是我们的行动目标。
dwExitCode	该线程的结束代码。

返回值

如果函数成功，则传回 TRUE。如果失败，则传回 FALSE。GetLastError() 可以获知更多细节。

TerminateThread() 看起来不错，直到我读了一份文件，上面说："TerminateThread() 是一个危险的函数，应该在最不得已的情况下才使用"。这是一个非常明白的警告。

TerminateThread() 强迫其行动目标（一个线程）结束，手段激烈而有力，甚至不允许该线程有任何"挣扎"的机会。这带来的副作用便是，线程没有机会在结束前清理自己。对线程而言，这可能导致前功尽弃。这个函数不会在目标线程中丢出一个异常情况（exception），目标线程在核心层面就被根本抹杀了。目标线程没有机会捕捉所谓的"结束请求"，并从而获得清理自己的机会。

还有另一个令人不愉快的情况。目标线程的堆栈没有被释放掉，于是可能会引起一大块内存泄漏（memory leak）。而且，任何一个与此线程有附着关系的 DLLs 也都没有机会获得"线程解除附着"的通知。

此函数唯一可以预期并依恃的是，线程 handle 将变成激发状态（译注：因为线程结束了），并且传回 dwExitCode 所指定的结束代码。

这个函数所带来的隐伏危机还包括：如果线程正进入一个 critical section 之中，该 critical section 将因此永远处于锁定状态，因为 critical section 不像 mutex 那样有所谓的 "abandoned" 状态。如果目标线程正在更新一份数据结构，这份数据结构也将永远处于不稳定状态。没有任何方法可以阻止这些问题的发生。

我的结论是：离 TerminateThread() 远远地！

使用信号（Signals）

下一个似乎可行的想法是使用 signals。在 Unix 系统中，signals 是跨进程传送通告（notifications）的标准方法。在 Unix 系统中 SIGTERM 相当于"请你离开"的意思，SIGKILL 则是粗略相当于 TerminateThread()。

这个点子似乎不错，因为 C runtime library 支持标准的 signals，如 SIGABRT 和 SIGINT。各种 signals 的处理函数可以利用 C 函数 signal() 设立之。

但是我很快就进入了一个死胡同。C runtime 函数中没有一个名为 kill()，而那是 Unix 系统藉以送出 signal 的操作。是有一个 raise() 啦，但只能够传送 signal 给目前的线程。

? FAQ 02：我可以在 Win32s 中使用多个线程吗？

观察过 C runtime library 的源代码之后，我发现 signals 其实是利用 Win32 的异常情况（exceptions）模拟的，Win32 之中并没有真正的 signals，所以这个想法也行不通。

跨越线程，丢出异常情况（Exceptions）

我真正要做的就是在目标线程中引发一个异常情况（exception）。如果有必要在结束前清理某些东西，目标线程可以设法捕捉此一异常情况，否则它可以什么都不管地直接结束自己的生命。

经过数个小时的努力之后，我可以很确定地告诉各位，Win32 API 中没有什么标准方法可以把一个异常情况丢到另一个线程中。真是不幸，因为这样的行为正是我们所需要的。

对专家而言…

关于“模拟丢出一个异常情况到另一个线程”的做法，在 Usenet 上有广泛的讨论。技术之一是利用 debugging API 写一个不合法的指令到目标线程的目前地址上。另一种做法是改变一个常用的指针，使它指向一个不合法地址，因而强迫程序代码产生一个异常情况。这两种做法都有缺点，但当你别无它途时，不妨一试。

设立一个标记

当所有方法都失败时，不妨返朴归真，回到最简单最明白的路上。Win32 核准的做法是在你的程序代码中设立一个标记，利用其值来要求线程结束自己。

这个技术有十分明显的优点，可以保证目标线程在结束之前有安全而一致的状态。其缺点也十分明显：线程需要一个 polling 机制，时时检查标记值，以决定该不该结束自己。此刻的你听到“polling”会不会有毛骨悚然的感觉？不不，我们并不是要写一个 busy loop 来检验标记值，我们的做法是使用一个手动重置（manual-reset）的 event 对象。Worker 线程可以检查该 event 对象的状态或是等待它，视情况而定。

结束一个线程，听起来好容易，但是结束程序必须按次序进行，以避免发生 race conditions。让程序依次序进行是非常重要的，特别是在程序要结束之前。结束一个程序就好像拆除一栋建筑物一样，在你以推土机轧平它之前，你必须确定每一个人都安全离开了屋子。结束一个程序也是这样，每一个线程都被迫结束，不管它进行到哪里。

让我们看一个简单的例子。列表 5-1 的范例程序中，THRDTERM 产生两个线程，周期性地检查一个 event 对象，以决定要不要结束自己。程序代码非常类似第 3 章的 BUSY2。

列表 5-1 THRDTERM—干净地结束一个线程

```
#0001  /*
#0002   * ThrdTerm.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 5, Listing 5-1
#0006   *
#0007   * Demonstrates how to request threads to exit.
#0008   *
#0009   * Build command: cl /MD ThrdTerm.c
#0010   */
#0011
```

```
#0012  #define WIN32_LEAN_AND_MEAN
#0013  #include <stdio.h>
#0014  #include <stdlib.h>
#0015  #include <windows.h>
#0016  #include <time.h>
#0017  #include "MtVerify.h"
#0018
#0019  DWORD WINAPI ThreadFunc(LPVOID);
#0020
#0021  HANDLE hRequestExitEvent = FALSE;
#0022
#0023  int main()
#0024  {
#0025      HANDLE hThreads[2];
#0026      DWORD dwThreadId;
#0027      DWORD dwExitCode = 0;
#0028      int i;
#0029
#0030      hRequestExitEvent = CreateEvent(
#0031          NULL, TRUE, FALSE, NULL);
#0032
#0033      for (i=0; i<2; i++)
#0034          MTVERIFY( hThreads[i] = CreateThread(NULL,
#0035              0,
#0036              ThreadFunc,
#0037              (LPVOID)i,
#0038              0,
#0039              &dwThreadId )
#0040          );
#0041
#0042      // Wait around for awhile, make
#0043      // sure the thread is running.
#0044      Sleep(1000);
#0045
#0046      SetEvent(hRequestExitEvent);
#0047      WaitForMultipleObjects(2, hThreads, TRUE, INFINITE);
#0048
#0049      for (i=0; i<2; i++)
#0050          MTVERIFY( CloseHandle(hThreads[i]) );
#0051
#0052      return EXIT_SUCCESS;
```

```
#0053  }
#0054
#0055
#0056  DWORD WINAPI ThreadFunc(LPVOID p)
#0057  {
#0058      int i;
#0059      int inside = 0;
#0060
#0061      UNREFERENCED_PARAMETER(p);
#0062
#0063      /* Seed the random-number generator */
#0064      srand( (unsigned)time( NULL ) );
#0065
#0066      for (i=0; i<1000000; i++)
#0067      {
#0068          double x = (double)(rand())/RAND_MAX;
#0069          double y = (double)(rand())/RAND_MAX;
#0070          if ( (x*x + y*y) <= 1.0 )
#0071              inside++;
#0072          if(WaitForSingleObject(hRequestExitEvent, 0) != WAIT_TIMEOUT)
#0073          {
#0074              printf("Received request to terminate\n");
#0075              return (DWORD)-1;
#0076          }
#0077      }
#0078      printf("PI = %.4g\n", (double)inside / i * 4);
#0079      return 0;
#0080  }
```

首先请注意，event 对象已经被用来取代一个简单的全局变量了。这个例子并不一定需要 event 对象，但如果我们使用它，worker 线程就能够在必要时候等待之。例如 worker 线程可以利用 event 对象来等待一个 Internet socket 的连接成功，或是等待用户发出离开的请求。我们只要把上述 worker 线程中的 WaitForSingleObject() 改为 WaitForMultipleObjects() 即可。

第二个要注意的是，所有线程共用同一个 event 对象。如果程序尝试要结束，没有必要再为每一个线程产生一个各别通告机制。因为所有的线程都必须立刻被通告。其他情况下可能需要更好的控制。

最后一点，请注意 main() 所代表的主线程，有一个 WaitForMultipleObjects() 操作，等待所有的线程 handles。等待线程 handles 变成激发状态，可以保证线程都已经安全地离开了。虽然本例是无穷等待，但商业软件可能会设一个上限值，例如 15 秒或 30 秒，通常那已经足以表示线程失去了反应。

线程优先权（Thread Priority）

有没有过这样的经验？你坐在你的车子里，目的地还在好几公里之遥，而时间已经很晚了。你拼命想告诉那些挡住你去路的人们，今天这个约会对你是多么多么重要，能不能请他们统统……呃……滚到马路外？很不幸，道路系统并没有纳入所谓的优先权观念。如果有某条专用道是给“非常重要”的通行所用的，你就可以摆脱那些如潮水般在你四周的车辆和行人，岂不甚妙？

Win32 有所谓的优先权（priority）观念，用以决定下一个获得 CPU 时间的线程是谁。较高优先权的线程必然获得较多的 CPU 时间。关于优先权的完整讨论其实相当复杂。你可以无分轩轾地给予每一个线程相同的优先权，这可能会使你承担不少麻烦。你也可以明智地使用优先权，使自己能够调整程序的执行次序。例如你可以设定你的 GUI 线程有较高优先权，使它对于用户的反应能够比较平顺一些，或者你可以改变 worker 线程的优先权，使它们只在系统的闲置时间（idle time）里工作。

Win32 优先权是以数值表现的，并以进程的“优先权类别（priority class）”、线程的“优先权层级（priority level）”和操作系统当时采用的“动态提升（Dynamic Boost）”作为计算基准。所有因素放在一起，最后获得一个 0~31 的数值。拥有最高优先权之线程，即为下一个将执行起来的线程。如果你有一大把 worker 线程，其“优先权类别”和“优先权层级”都相同，那么就每一个轮流执行。这是所谓的 “round robin” 调度方式。如果你有一个线程总是拥有最高优先权，那么它就永远获得 CPU 时间，别人都别玩了。这就是为什么必须明智而谨慎地使用优先权的原因。

优先权类别（Priority Class）

“优先权类别”是进程的属性之一．这个属性可以表现出这一进程和其他进程比较之下的重要性。Win32 提供四种优先权类别，每一个类别对应一个基本的优先权层级．表格 5-1 展示了四个优先权类别．

表格 5-1 优先权类别（Priority Classes）

优先权类别（Priority Classes）	基础优先权值（base priority）
HIGH_PRIORITY_CLASS	13
IDLE_PRIORITY_CLASS	4
NORMAL_PRIORITY_CLASS	7 or 8（译注：有些资料上写 7 or 9）
REALTIME_PRIORITY_CLASS	24

大部分程序使用 NORMAL_PRIORITY_CLASS．少数情况下才会考虑使用其他类别，例如，Task Manager 就是使用 HIGH_PRIORITY_CLASS，所以即使其他程序处于非常忙碌的状态下，它也总是能够有所反应．

Windows NT 中有一个好例子，可以说明到底应不应该使用某些特定的优先权类别．以 OpenGL 完成的屏幕保护程序（screen saver）看似密集地使用了所有 CPU 时间．如果这个屏幕保护程序启动时你正在进行系统备份，备份操作会慢下来，像蜗牛一样．但如果屏幕保护程序使用 IDLE_PRIORITY_CLASS，它就只会在 CPU 绝对空闲的时候才执行．

最后一个类别是 REALTIME_PRIORITY_CLASS．这个类别用以协助解决一些和时间有密切关系的工作．举个例子，如果有个程序必须反应一个设备驱动程序的行为，而该驱动程序用来实时监控（real-time monitoring）真实世界中的一台仪器，那么将该进程设为这个优先权类别，就可以使它甚至优于核心进程和设备驱动程序．这个优先权类别不应该用于标准 GUI 程序或甚至于典型的服务器程序．

优先权类别适用于进程而非线程．你可以利用 SetPriorityClass() 和 GetPriorityClass() 来调整和验证其值。本书并未涵盖这两个函数的说明。

优先权层级（Priority Level）

线程的优先权层级（Priority Level）是对进程的优先权类别的一个修改，使你能够调整同一个进程内的各线程的相对重要性。一共有七种优先权层级，显示于表格 5-2 中。

表格 5-2 优先权层级（Priority Levels）

优先权层级（Priority Levels）	调整值
THREAD_PRIORITY_HIGHEST	+2
THREAD_PRIORITY_ABOVE_NORMAL	+1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	–1
THREAD_PRIORITY_LOWEST	–2
THREAD_PRIORITY_IDLE	Set to 1
THREAD_PRIORITY_TIME_CRITICAL	Set to 15

注意：对于 REALTIME_PRIORITY_CLASS 的调整值，有点不同于上表所列。

优先权层级可以利用 SetThreadPriority() 改变之。

```
BOOL SetThreadPriority(
    HANDLE hThread,
    int nPriority
);
```

参数

hThread	代表欲调整优先权的那个线程。
nPriority	表格 5-2 所显示的数值。

返回值

如果函数成功，就传回表格 5-2 所列的其中一个值。如果函数失败，就传回 FALSE。GetLastError() 可以获得更详细的信息。

线程目前的优先权层级可以利用 GetThreadPriority() 获知。

```
int GetThreadPriority(
    HANDLE hThread
);
```

参数

hThread	代表一个线程

返回值

如果函数成功，就传回 TRUE。如果函数失败，就传回 THREAD_PRIORITY_ERROR_RETURN。GetLastError() 可以获得更详细的信息。

KERNEL32.DLL 中的优先权

我使用 Windows 95 所提供的 PVIEW32，观察我的系统中的各个进程，结果如图 5-1 所示。我看到系统模块 KERNEL32.DLL 有八个线程，其优先权类别是 HIGH_PRIORITY_CLASS，所以其基础优先权值为 13。检查其线程，发现有四个线程的优先权层级是 THREAD_PRIORITY_LOWEST，所以其优先权为 11。三个线程的优先权层级是 THREAD_PRIORITY_NORMAL，所以其优先权为 13。一个线程的优先权层级是 THREAD_PRIORITY_TIME_CRITICAL，所以其优先权为 15。最后这个线程应该总是能够在任何其他“非实时线程”之前被调度程序选中执行。

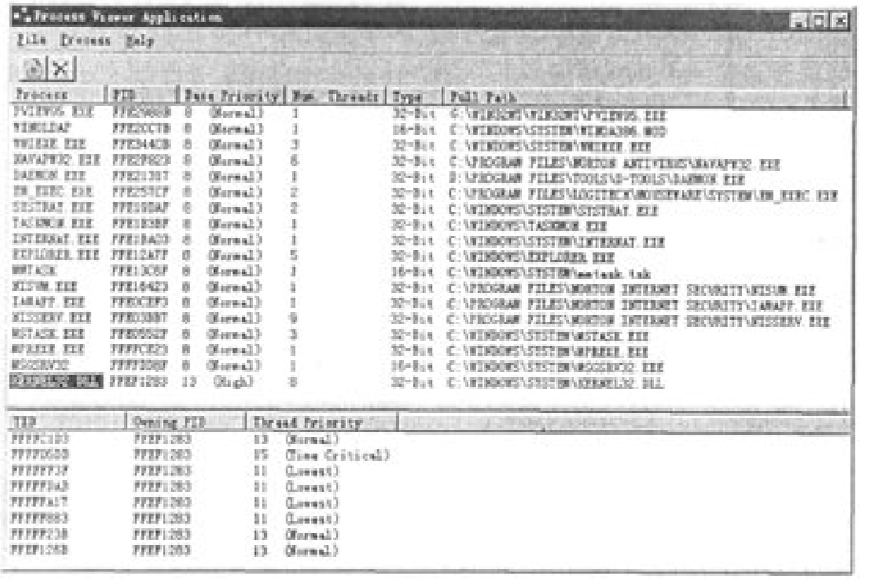


图 5-1 Windows 95 中的 PVIEW32

动态提升（Dynamic Boost）

决定线程真正优先权的最后一个因素是其目前的动态提升值（Dynamic Boost）。所谓动态提升是对优先权的一种调整，使系统能够机动对待线程，以强化程序的可用性。

最容易被我们观察的，便是 Windows NT 施行于所有前台程序的“线程动态提升”。图 5-2 的 系统属性 中的【性能】附页，允许用户指定前台程序应该对用户有怎样的回应。你可以在【我的电脑】中按下右键，并选择【属性】而获得这一画面。

默认情况下图 5-2 的“动态提升”被设定为最大，这使得拥有键盘焦点的程序（前台程序）的优先权得以提升 +2。这个设定使得前台程序比后台程序获得较多的 CPU 时间，因此即使系统忙碌，前台程序还是容易保持其 UI 敏感度。

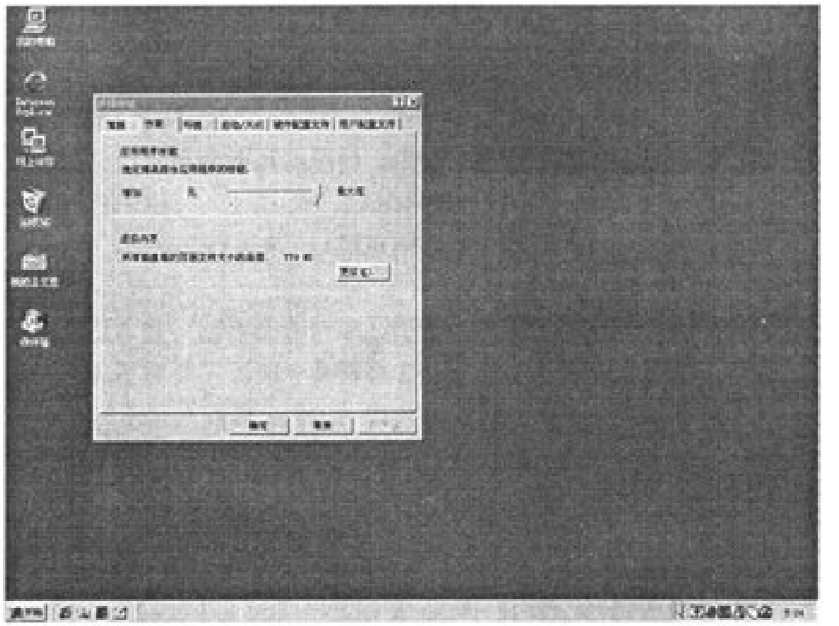


图 5-2 Windows NT 4.0 的系统属性

第二种优先权动态提升也适用于同属一个进程的线程，用以反应用户的输入或磁盘的输入。例如，只要线程获得键盘输入，该线程就得到一个 +5 的优先权调整值。这使得该线程有机会处理那个输入，并且提供立即的回应给用户。其他可能引起优先权动态提升的情况还包括鼠标消息、计时器消息等等。

最后一种优先权动态提升的情况可能发生在任何一个线程（不限属于哪一个进程）身上。那是在一个“等待状态”获得满足时发生的，例如有一个线程正在等待一个 mutex，当 Wait...() 返回时，该线程的优先权会获得动态提升。这样的提升意味着 critical sections 将尽可能地被快速处理，而等待时间将尽可能地缩短。

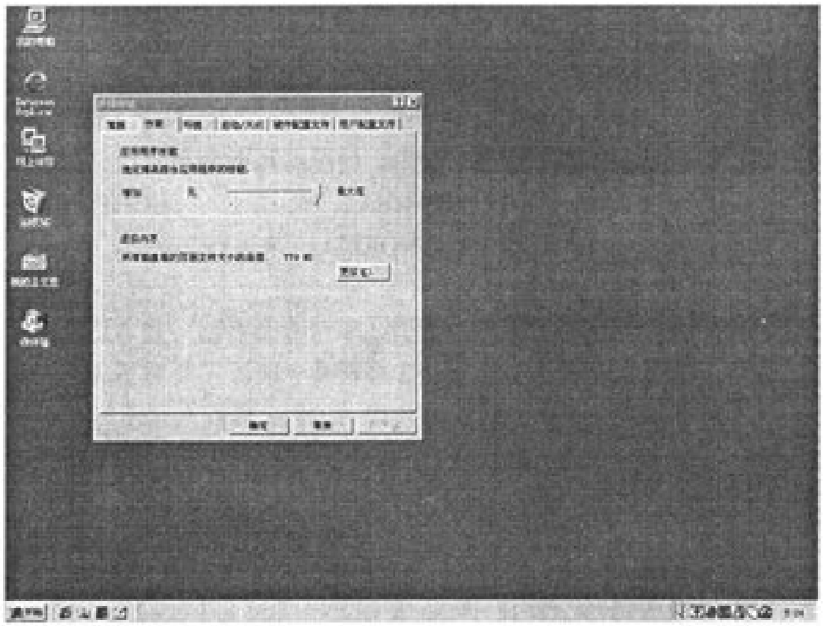


图 5-2 Windows NT 4.0 的系统属性

第二种优先权动态提升也适用于同属一个进程的线程，用以反应用户的输入或磁盘的输入。例如，只要线程获得键盘输入，该线程就得到一个 +5 的优先权调整值。这使得该线程有机会处理那个输入，并且提供立即的回应给用户。其他可能引起优先权动态提升的情况还包括鼠标消息、计时器消息等等。

最后一种优先权动态提升的情况可能发生在任何一个线程（不限属于哪一个进程）身上。那是在一个“等待状态”获得满足时发生的，例如有一个线程正在等待一个 mutex，当 Wait...() 返回时，该线程的优先权会获得动态提升。这样的提升意味着 critical sections 将尽可能地被快速处理，而等待时间将尽可能地缩短。

CreateThread() 说：产生一个新线程，传回其 handle，但不要马上开始执行之。

下面是一个例子：

```
HANDLE hThread;
DWORD threadId;

hThread = CreateThread(NULL,
                       0,
                       ThreadFunc,
                       0,
                       CREATE_SUSPENDED,
                       &threadId);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
```

一旦线程设定妥当，你可以调用 ResumeThread() 开始执行：

DWORD ResumeThread(
HANDLE hThread
);

参数

hThread	欲被再次执行的线程。

返回值

如果函数成功，则传回线程的前一个挂起次数。如果失败，则传回 0xFFFFFFFF。GetLastError() 可以获得更详细的信息。

挂起（suspending）一个线程

相对于 ResumeThread()，毫不令人惊讶地，有一个 SuspendThread() 函数。这个函数允许调用端指定一个线程睡眠（挂起）。直到又有人调

用了 ResumeThread()，线程才会醒来．因此，睡眠中的线程不可能唤醒自己，这个函数的规格如下：

```
DWORD SuspendThread(
    HANDLE hThread
);
```

参数

hThread	欲被挂起的线程．

返回值

如果函数成功，则传回线程目前的挂起次数．如果失败，则传回0xFFFFFFFF．GetLastError() 可以获得更详细的信息．

SuspendThread() 是另一个可能会潜在引发问题的函数．考虑一下这种情况：一个进程拥有三个线程A，B，C．线程C正在某个 critical section 内，而线程B正在等它出来．然后，线程A挂起了线程C．在这种情况下，线程C将永远不会离开 critical section，而线程B也就相当于进入了死锁状态．

SuspendThread() 的最大用途就是用来协助撰写调试器．调试器允许在程序员的控制之下，启动或停止任何一个线程．

提要

在这一章中你看到了“结束一个线程”的正确方法，你也看到了线程优先权的定义方式，及其改变所带来的影响．最后你还看到了如何利用CREATE_SUSPENDED 来初始化一个线程—在它开始执行之前．

Overlapped I/O，在你身后变戏法

这一章描述如何使用 overlapped I/O（也就是 asynchronous I/O）。某些时候 overlapped I/O 可以取代多线程的功用。然而，overlapped I/O 加上 completion ports，常被设计为多线程处理，以便在一个“受制于 I/O 的程序”（所谓 I/O bound 程序）中获得高效率。

译注 深层讨论 Win32 平台（WinNT 和 Win95）的 File Systems 和 Device I/O 的书籍极少，Advanced Windows 3rd edition（Jeffrey Richter/Microsoft Press）第 13 章和第 14 章有很多宝贵的内容，可供参考。

截至目前我已经花了数章篇幅告诉各位“如何”以及“为什么”要使用线程。我将以这一章介绍一个你可能不想使用多线程的场合。许多应用程序，例如终端机模拟程序，都需要在同一时间处理对一个以上的文件的读写操作。利用 Win32 所谓的 overlapped I/O 特性，你就可以让所有这些 I/O 操作并行处理，并且当任何一个 I/O 完成时，你的程序会收到一个通告。其他操作系统把这个特性称为 nonblocking I/O 或 asynchronous I/O。

回头看看第 2 章和第 3 章，那里曾经示范如何产生一个线程负责后台打印操作。事实上“后台打印”这种说法是错误的，我们在后台所进行的操作只不过是产生打印机所需的数据。Windows 操作系统负责以打印管理器（Printer

Manager）完成打印．打印管理器会 “spooled” 那些准备好的数据，并且以打印机所能接受的速度，慢慢地将数据喂给打印机．

关键在于 I/O 设备是个慢速设备，不论打印机、调制解调器，甚至硬盘，与 CPU 相比都奇慢无比．坐下来干等 I/O 的完成是一件不甚明智的事情．有时候数据的流动率非常惊人，把数据从你的文件服务器中以 Ethernet 速度搬走，其速度可能高达每秒一百万个字节．如果你尝试从文件服务器中读取 100KB ，在用户的眼光来看几乎是瞬间完成．但是，要知道，你的线程执行这个命令，已经浪费了 10 个一百万次 CPU 周期．

现在想象一下，同一个文件服务器，透过一条拨号线路，被 Remote Access Services（RAS）处理．于是，100KB 的数据量在 ISDN 上需要 15 秒，在 9600 bps 线路上需要几乎两分钟．即使从用户的眼光来看，一个窗口在这么长的时间中完全没有反应，也是相当可怕的．

这个问题的明显解决方案就是，使用另一个线程来进行 I/O．然而，这就产生出一些相关问题，包括如何在主线程中操控许多个 worker 线程、如何设定同步机制、如何处理错误情况、如何显示对话框．这些难题都将在本章出现并解决之．

? FAQ 21：什么是 overlapped I/O？

一个最简单的回答：overlapped I/O 是 Win32 的一项技术，你可以要求操作系统为你传送数据，并且在传送完毕时通知你．这项技术使你的程序在 I/O 进行过程中仍然能够继续处理事务．事实上，操作系统内部正是以线程来完成 overlapped I/O．你可以获得线程的所有利益，而不需付出什么痛苦代价．

? FAQ 22：Overlapped I/O 在 Windows 95 上有什么限制？

重要！

Windows 95 所支持的 overlapped I/O 有些限制，只适用于 named pipes、mailslots、serial I/O、以及 socket() 或 accept() 所传回来的 sockets，它并不支持磁盘或光盘中的文件操作，本章的所有例子只在 Windows NT 下才能够有效运作．

这一章对于 overlapped I/O 的讨论，将从最简单的应用开始，然后再演变到最高级的应用。

- 激发的文件 handles
- 激发的 event 对象
- 异步过程调用（Asynchronous Procedure Calls，APCs）
- I/O completion ports

其中以 I/O completion ports 特别显得重要，因为它们是唯一适用于高负载服务器（必须同时维护许多连接线路）的一个技术。Completion ports 利用一些线程，帮助平衡由“I/O 请求”所引起的负载。这样的架构特别适合用在 SMP 系统（译注：支持多个 CPU 的操作系统）中产生所谓的 “scalable” 服务器。

译注 所谓 scalable 系统，是指能够藉着增加 RAM 或磁盘空间或 CPU 个数而提升应用程序效能的一种系统。

Win32 文件操作函数

Win32 之中有三个基本的函数用来执行 I/O，它们是：

- CreateFile()
- ReadFile()
- WriteFile()

没有另外哪一个函数用来关闭文件，只要调用 CloseHandle() 即可。本章对于这些函数将只涵盖其与 overlapped I/O 有关的部分，至于其他和文件 I/O 有关的部分，请参考 Win32 Programmer's Reference。

CreateFile() 可以用来打开各式各样的资源，包括（但不限制于）：

- 文件（硬盘、软盘、光盘或其他）
- 串行口和并行口（serial and parallel ports）
- Named pipes

- Console（请看第 8 章）

CreateFile() 的函数原型看起来像这样：

```
HANDLE CreateFile(
    LPCTSTR lpFileName,          // 指向文件名称
    DWORD dwDesiredAccess,       // 存取模式（读或写）
    DWORD dwShareMode,           // 共享模式（share mode）
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // 指向安全属性结构
    DWORD dwCreationDisposition,  // 如何产生
    DWORD dwFlagsAndAttributes,   // 文件属性
    HANDLE hTemplateFile          // 一个临时文件，将拥有全部的属性拷贝
```

其中第 6 个参数 dwFlagsAndAttributes 是使用 overlapped I/O 的关键.这个参数可以藉由许多个数值组合在一起而完成,其中对于本处讨论最重要的一个数值便是 FILE_FLAG_OVERLAPPED. 你可以藉着这个参数，指定使用同步（传统的）调用，或是使用 overlapped（异步）调用，但不能够两个都指定. 换句话说，如果这个标记值设立，那么对该文件的每一个操作都将是 overlapped.

一个不常被讨论的 overlapped I/O 性质是，它可以在同一时间读（或写）文件的许多部分. 微妙处在于这些操作都使用相同的文件 handle. 因此，当你使用 overlapped I/O 时，没有所谓“目前的文件位置”这样的观念. 每一次读或写的操作都必须包含其文件位置.

如果你发出许多个 overlapped 请求，那么执行次序无法保证. 虽然你在单一磁盘中对文件进行操作时很少会有这样的行为，但如果面对多个磁盘，或不同种类的设备（如网络和磁盘），就常常会看到 I/O 请求完全失去次序.

? FAQ 23: 我能够以 C runtime library 使用 overlapped I/O 吗?

你将不可能藉由调用 C runtime library 中的 stdio.h 函数而使用 overlapped I/O. 因此，没有很方便的方法可以实现 overlapped text-based I/O. 例如，fgets() 允许你一次读取一行文字，但你不能够使用 fgets()、fprintf() 或任何其他类似的 C runtime 函数来进行 overlapped I/O.

Overlapped I/O 的基本型式是以 ReadFile() 和 WriteFile() 完成的. 这两

个函数的原型如下：

```
BOOL ReadFile(
    HANDLE hFile,                  // 欲读之文件
    LPVOID lpBuffer,               // 接收数据之缓冲区
    DWORD nNumberOfBytesToRead,    // 欲读取的字节个数
    LPDWORD lpNumberOfBytesRead,   // 实际读取的字节个数的地址
    LPOVERLAPPED lpOverlapped      // 指针，指向 overlapped info
);

BOOL WriteFile(
    HANDLE hFile,                  // 欲写之文件
    LPCVOID lpBuffer,              // 储存数据之缓冲区
    DWORD nNumberOfBytesToWrite,   // 欲写入的字节个数
    LPDWORD lpNumberOfBytesWritten, // 实际写入的字节个数的地址
    LPOVERLAPPED lpOverlapped      // 指针，指向 overlapped info
);
```

这两个函数很像 C runtime 函数中的 fread() 和 fwrite()，差别在于最后一个参数 lpOverlapped。如果 CreateFile() 的第 6 个参数被指定为 FILE_FLAG_OVERLAPPED，你就必须在上述的 lpOverlapped 参数中提供一个指针，指向一个 OVERLAPPED 结构。

OVERLAPPED 结构

OVERLAPPED 结构执行两个重要的功能。第一，它像一把钥匙，用以识别每一个目前正在进行的 overlapped 操作。第二，它在你和系统之间提供了一个共享区域，参数可以在该区域中双向传递。

OVERLAPPED 结构看起来像这样：

```
typedef struct _OVERLAPPED {
    DWORD   Internal;
    DWORD   InternalHigh;
    DWORD   Offset;
    DWORD   OffsetHigh;
    HANDLE  hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

OVERLAPPED 结构中的成员

表格 6-1 描述 OVERLAPPED 结构中的每一个成员。

表格 6-1 OVERLAPPED 结构中的成员（栏位）

成员名称	说 明
Internal	通常它被保留。然而当 GetOverlappedResult() 传回 FALSE 并且 GetLastError() 并非传回 ERROR_IO_PENDING 时，这个栏位将内含一个视系统而定的状态
InternalHigh	通常它被保留。然而当 GetOverlappedResult() 传回 TRUE 时，这个栏位将内含“被传输数据的长度”
Offset	文件之中开始被读或被写的偏移位置（以字节为单位）。该偏移位置从文件头开始起算。如果目标设备（例如 pipes）并没有支持文件位置，此栏位将被忽略
OffsetHigh	64 位的文件偏移位置中，较高的 32 位。如果目标设备（例如 pipes）并没有支持文件位置，此栏位将被忽略
hEvent	一个手动重置（manual-reset）的 event 对象，当 overlapped I/O 完成时即被激发。ReadFileEx() 和 WriteFileEx() 会忽略这个栏位，彼时它可能被用来传递一个用户自定义的指针

由于 OVERLAPPED 结构的生命期超越 ReadFile() 和 WriteFile() 函数，所以把这个结构放在一个安全的地方是很重要的事情。通常局部变量并不是一个安全的地方，因为它会很快就越过了生存范围（out of scope）。最安全的地方就是 heap。

现在我们有了所有的基础物质，让我们看看如何运用它们。

被激发的 File Handles

最简单的 overlapped I/O 类型，是使用它自己的文件 handle 作为同步机制。首先你以 FILE_FLAG_OVERLAPPED 告诉 Win32 说你不要使用默认的同步 I/O。然后，你设立一个 OVERLAPPED 结构，其中内含“I/O 请求”的所有必要参数，并以此识别这个“I/O 请求”，直到它完成为止。接下来，调用 ReadFile() 并以 OVERLAPPED 结构的地址作为最后一个参数。这时候，理论上，Win32 会在后台处理你的请求。你的程序可以放心地继续处理其他事情。

如果你需要等待 overlapped I/O 的执行结果，作为 WaitForMultipleObjects() 的一部分，请在 handle 数组中加上这个文件 handle。文件 handle 是一个核心对象，一旦操作完毕即被激发。当你完成操作之后，请调用 GetOverlappedResult() 以确定结果如何。

调用 GetOverlappedResult()，你获得的结果和“调用 ReadFile() 或 WriteFile() 而没有指定 overlapped I/O”所传回的东西一样。这个函数的价值在于，在文件操作真正完成之前，你不可能确实知道它是否成功。甚至在一个完美无瑕的环境下读一个已知的磁盘文件，也有可能发生硬件错误、服务器当掉，或任何未能预期的错误。因此，调用 GetOverlappedResult() 是很重要的。

GetOverlappedResult() 规格如下：

```
BOOL GetOverlappedResult(
    HANDLE hFile,
    LPOVERLAPPED lpOverlapped,
    LPDWORD lpNumberOfBytesTransferred,
    BOOL bWait
);
```

参数

hFile	文件或设备（device）的 handle。
lpOverlapped	一个指针，指向 OVERLAPPED 结构。
lpNumberOfBytesTransferred	一个指针，指向 DWORD，用以表示真正被传输的字节个数。
bWait	一个布尔值，用以表示是否要等待操作完成。TRUE 表示要等待。

返回值

如果 overlapped 操作成功，此函数传回 TRUE。失败则传回 FALSE。GetLastError() 可获得更详细的失败信息。如果 bWait 为 FALSE 而 overlapped 还是没有完成，GetLastError() 会传回 ERROR_IO_INCOMPLETE。

列表 6-1 所显示的程序代码从文件 C:\WINDOWS\WINFILE.EXE 的第 1500 位置处读入 300 个字节。这段代码放在书附盘片的 IOBYFILE 程序中。

列表 6-1 引用自 IOBYFILE.C—overlapped I/O with a signaled file handle

```
#0001  int ReadSomething()
#0002  {
#0003      BOOL rc;
#0004      HANDLE hFile;
#0005      DWORD numread;
#0006      OVERLAPPED overlap;
#0007      char buf[512];
#0008
#0009      // open the file for overlapped reads
#0010      hFile = CreateFile( "C:\\WINDOWS\\WINFILE.EXE",
#0011                  GENERIC_READ,
#0012                  FILE_SHARE_READ|FILE_SHARE_WRITE,
#0013                  NULL,
#0014                  OPEN_EXISTING,
#0015                  FILE_FLAG_OVERLAPPED,
#0016                  NULL
#0017              );
```

```
#0018     if (hFile == INVALID_HANDLE_VALUE)
#0019        return -1;
#0020
#0021     // Initialize the OVERLAPPED structure
#0022     memset(&overlap, 0, sizeof(overlap));
#0023     overlap.Offset = 1500;
#0024
#0025     // Request the data
#0026     rc = ReadFile(
#0027               hFile,
#0028               buf,
#0029               300,
#0030               &numread,
#0031               &overlap
#0032            );
#0033
#0034     if (rc)
#0035     {
#0036        // The data was read successfully
#0037     }
#0038     else
#0039     {
#0040        // Was the operation queued ?
#0041        if (GetLastError() == ERROR_IO_PENDING)
#0042        {
#0043           // We could do something else for awhile here...
#0044
#0045           WaitForSingleObject(hFile, INFINITE);
#0046           rc = GetOverlappedResult(
#0047                                hFile,
#0048                                &overlap,
#0049                                &numread,
#0050                                FALSE
#0051                              );
#0052        }
#0053        else
#0054        {
#0055           // Something went wrong
#0056        }
#0057     }
#0058
```

```
#0059      CloseHandle(hFile);
#0060
#0061      return TRUE;
#0062   }
```

? FAQ 24: Overlapped I/O 总是异步地 (asynchronously) 执行吗?

在这段程序代码中我们得特别注意几点。第一，虽然你要求一个 overlapped 操作，但它并不一定就是 overlapped! 如果数据已经被放进 cache 中，或如果操作系统认为它可以很快速地取得那份数据，那么文件操作就会在 ReadFile() 返回之前完成，而 ReadFile() 将传回 TRUE。这种情况下，文件 handle 处于激发状态，而对文件的操作可被视为就像 overlapped 一样。

下一个需要注意的是，如果你要求一个文件操作为 overlapped，而操作系统把这个“操作请求”放到队列中等待执行，那么 ReadFile() 和 WriteFile() 都会传回 FALSE 以示失败。这个行为并不是很直观，你必须调用 GetLastError() 并确定它传回 ERROR_IO_PENDING，那意味着“overlapped I/O 请求”被放进队列之中等待执行。GetLastError() 也可能传回其他的值，例如 ERROR_HANDLE_EOF，那就真正代表一个错误了。

请注意 overlapped I/O 如何解决“没有文件指针”这个问题。事实上 OVERLAPPED 结构本身就包含了“这个操作应该从哪里开始”的信息。那当然是值得注意的东西，但是在列表 6-1 中未显示出来。OVERLAPPED 结构有能力处理 64 位的偏移值，因此即使面对非常巨大的文件也不怕。

为了等待文件操作完成，我把文件的 handle 交给 WaitForSingleObject()。一旦 overlapped 操作完成，该文件 handle 就会成为激发状态。我所举的这个例子过于简单，真实世界里，可能会在程序的某个集中处，等待许多个 handles。

列表 6-1 调用 WaitForSingleObject() 其实是多余的，因为 GetOverlappedResult() 就可以用来等待 overlapped 操作的完成。不过由于实际上你常常会以一个 Wait...() 函数去等待 overlapped 操作完成，所以我才做这样的示范。

被激发的 Event 对象

以文件 handle 作为激发机制，有一个明显的限制，那就是没办法说出到底是哪一个 overlapped 操作完成了。如果每个文件 handle 只有一个操作等待决定，上述问题其实并不成为问题。但是如我稍早所说，系统有可能同时接受数个操作，而它们都使用同一个文件 handle。于是很明显地，为每一个可能正在进行中的 overlapped 操作调用 GetOverlappedResult()，并不是很有效率的做法。毫不令人惊讶，Win32 提供了一个比较好的做法，用以解决这样的问题。

OVERLAPPED 结构中的最后一个栏位，是一个 event handle。如果你使用文件 handle 作为激发对象，那么此栏位可为 NULL。当这个栏位被设定为一个 event 对象时，系统核心会在 overlapped 操作完成的时候，自动将此 event 对象给激发起来。由于每一个 overlapped 操作都有它自己独一无二的 OVERLAPPED 结构，所以每一个结构都有它自己独一无二的一个 event 对象，用以代表该操作。

? FAQ 25：我应该如何为 overlapped I/O 产生一个 event 对象？

有一件事很重要：你所使用的 event 对象必须是手动重置（manual-reset）而非自动重置（auto-reset，详见第 4 章）。如果你使用自动重置，就可能产生出 race condition，因为系统核心有可能在你有机会等待该 event 对象之前，先激发它，而你知道，event 对象的激发状态是不能够保留的（这一点和 semaphore 不同）。于是这个 event 状态将遗失，而你的 Wait...() 函数永不返回。但是一个手动重置的 event，一旦激发，就一直处于激发状态，直到你动手将它改变。

使用 event 对象搭配 overlapped I/O，你就可以对同一个文件发出多个读取操作和多个写入操作，每一个操作有自己的 event 对象；然后再调用 WaitForMultipleObjects() 来等待其中之一（或全部）完成。

来自 IOBYEVNT 的相关函数代码显示于列表 6-2。其中的函数能够将一个特定的请求（并指定文件偏移值和数据大小）记录起来等待执行。

列表 6-2 节录自 IOBYEVNT.C—overlapped I/O with signaled events

```
#0001 // Need to keep the events in their own array so we can wait on them.
#0002 HANDLE  ghEvents[MAX_REQUESTS];
#0003 // Keep track of each individual I/O operation
#0004 OVERLAPPED gOverlapped[MAX_REQUESTS];
#0005 // Handle to the file of interest.
#0006 HANDLE ghFile;
#0007 // Need a place to put all this data
#0008 char gBuffers[MAX_REQUESTS][READ_SIZE];
#0009
#0010 int QueueRequest(int nIndex, DWORD dwLocation, DWORD dwAmount)
#0011 {
#0012     int i;
#0013     BOOL rc;
#0014     DWORD dwNumread;
#0015     DWORD err;
#0016
#0017     MTVERIFY(
#0018         ghEvents[nIndex] = CreateEvent(
#0019                     NULL,    // No security
#0020                     TRUE,    // Manual reset - extremely important!
#0021                     FALSE,   // Initially set Event to non-signaled state
#0022                     NULL     // No name
#0023                    )
#0024     );
#0025     gOverlapped[nIndex].hEvent = ghEvents[nIndex];
#0026     gOverlapped[nIndex].Offset = dwLocation;
#0027
#0028     for (i=0; i<MAX_TRY_COUNT; i++)
#0029     {
#0030         rc = ReadFile(
#0031             ghFile,
#0032             gBuffers[nIndex],
#0033             dwAmount,
#0034             &dwNumread,
#0035             &gOverlapped[nIndex]
#0036         );
#0037
#0038         // Handle success
#0039         if (rc)
```

```
#0040        {
#0041           printf("Read #%d completed immediately.\n", nIndex);
#0042           return TRUE;
#0043        }
#0044
#0045        err = GetLastError();
#0046
#0047        // Handle the error that isn't an error. rc is zero here.
#0048        if (err == ERROR_IO_PENDING)
#0049        {
#0050           // asynchronous i/o is still in progress
#0051           printf("Read #%d queued for overlapped I/O.\n", nIndex);
#0052           return TRUE;
#0053        }
#0054
#0055        // Handle recoverable error
#0056        if ( err == ERROR_INVALID_USER_BUFFER ||
#0057             err == ERROR_NOT_ENOUGH_QUOTA ||
#0058             err == ERROR_NOT_ENOUGH_MEMORY )
#0059        {
#0060           Sleep(50);  // Wait around and try later
#0061           continue;
#0062        }
#0063
#0064        // Give up on fatal error.
#0065        break;
#0066     }
#0067
#0068     printf("ReadFile failed.\n");
#0069     return -1;
#0070  }
```

你应该注意，这段代码为每一个“overlapped I/O 请求”产生一个新的 event 对象，其 handle 存放在 OVERLAPPED 结构之中，也存放在一个全局数组中，以便给 WaitForMultipleObjects() 使用。

再一次我要告诉你，你可能会看到“I/O 请求”立刻完成的情况。下面就是一个实际的执行结果。

IOBYEVNT 的输出结果：

```
Read #0 queued for overlapped I/O.
Read #1 completed immediately.
Read #2 completed immediately.
Read #3 completed immediately.
Read #4 completed immediately.
QUEUED!!
Read #0 return 1. 512 bytes were read.
Read #1 return 1. 512 bytes were read.
Read #2 return 1. 512 bytes were read.
Read #3 return 1. 512 bytes were read.
Read #4 return 1. 512 bytes were read.
```

本例之中，Windows NT 在第一个“读人请求”之后，做了一个预先读取的操作（译注：也就是第一次读入时，多读一些数据），因此后续的“读人请求”立刻完成并返回。

你一定也看到了，ReadFile() 身处一个循环之中，由于操作系统的运作原理，你为 overlapped I/O 所指定的缓冲区必须在内存中锁定。如果系统（或同一个程序中）有太多缓冲区在同一时间被锁定，可能对效率是一个很大的伤害。因此，系统有时候必须为程序“降温”（降低速度啦）。但是当然不能把它们阻塞住（blocking），因为那又使得 overlapped I/O 失去意义。Win32 会传回一个像 ERROR_INVALID_USER_BUFFER 那样的错误信息，表示此刻没有足够的资源来处理这个“I/O 请求”。这种问题也会发生在 IOBYFILE 程序中，只不过，为了让程序代码清爽一些，我省略了对这些错误信息的处理。

本章一开始我曾提出这样的问题：以 Remote Access Service（RAS）连接到一个服务器，原本 1/10 秒可以完成的数据搬移，现在需要两分钟。这时候我们就可以把这个操作记录起来，并做成 overlapped I/O，然后在主消息循环中使用 MsgWaitForMultipleObjects() 以便在取得数据后有所反应。

异步过程调用（Asynchronous Procedure Calls，APCs）

? FAQ 26：ReadFileEx()和WriteFileEx()的优点是什么？

使用 overlapped I/O 并搭配 event 对象，会产生两个基础性问题。第一个问题是，使用 WaitForMultipleObjects()，你只能够等待最多达 MAXIMUM_WAIT_OBJECTS 个对象。在 Windows NT 3.x 和 4.0 所提供的 Win32 SDK 中，此最大值为 64。如果你要等待 64 个以上的对象，就会出问题。所以即使在一个客户/服务器环境（client-server）中，你也只能同时拥有 64 个连接点。第二个问题是，你必须不断根据“哪一个 handle 被激发”而计算如何反应。你必须有一个分派表格（dispatch table）和 WaitForMultipleObjects() 的 handles 数组结合起来。

这两个问题可以靠一个所谓的异步过程调用（Asynchronous Procedure Call，APC）解决。只要使用“Ex” 版的 ReadFile() 和 WriteFile()，你就可以调用这个机制。这两个函数允许你指定一个额外的参数，是一个 callback 函数地址。当一个 overlapped I/O 完成时，系统应该调用该 callback 函数。这个 callback 函数被称为 I/O completion routine，因为系统是在某一个特别的 overlapped I/O 操作完成之后调用它。

? FAQ 27：一个 I/O completion routine 何时被调用？

然而，Windows 不会贸然中断你的程序，然后调用你提供的这个 callback 函数。系统只有在线程说“好，现在是个安全时机”时才调用你的 callback 函数。以 Windows 的说法就是：你的线程必须在所谓的 “alertable” 状态之下才行。如果有一个 I/O 操作完成，而线程不处于 “alertable” 状态，那么对 I/O completion routine 的调用就会暂时被保留下来。因此，当一个线程终于进入 “alertable” 状态时，可能已经有一大堆储备的 APCs 等待被处理。

如果线程因为以下五个函数而处于等待状态，而其 “alertable” 标记被设为 TRUE，则该线程就是处于 “alertable” 状态：

- SleepEx()
- WaitForSingleObjectEx()

- WaitForMultipleObjectsEx()
- MsgWaitForMultipleObjectsEx()
- SignalObjectAndWait()

“只有当程序处于 “alertable” 状态下时，APCs 才会被调用”这个观念是很重要的。其结果就是，当你的程序正在进行精确至小数点后 10000 位的圆周率的计算时，I/O completion routine 不会被调用。如果原线程正忙于重绘屏幕，也不会有另一个线程被使用。

对专家而言…

用于 overlapped I/O 的 APCs 是一种所谓的 user mode APCs。Windows NT 另有一种所谓的 kernel mode APCs。Kernel mode APCs 也会像 user mode APCs 一样被保存起来，但一个 kernel mode APC 一定会在下一个 timeslice 被调用，不管线程当时正在做什么。Kernel mode APCs 用来处理系统机能，不在应用程序的控制之中。

你所提供的 I/O completion routine 应该有这样的型式：

```
VOID WINAPI FileIOCompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped
);
```

参数

dwErrorCode	这个参数内含以下的值：0 表示操作完成，ERROR_HANDLE_EOF 表示操作已经到了文件尾端。
dwNumberOfBytesTransferred	真正被传输的数据字节数。
lpOverlapped	指向 OVERLAPPED 结构。此结构由开启 overlapped I/O 操作的函数提供。

? FAQ 28：我如何把一个用户自定义数据传递给 I/O completion routine?

I/O completion routine 需要一些东西以了解其环境。如果它不知道 I/O 操作完成了什么，它也就很难决定对此数据要做些什么。使用 APCs 时，OVERLAPPED 结构中的 hEvent 栏位不需要用来放置一个 event handle。Win32 文件上说此时 hEvent 栏位可以由程序员自由运用。那么最大的用途就是：首先配置一个结构，描述数据来自哪里，或是要对数据进行一些什么操作，然后将 hEvent 栏位设定指向该结构。

列表 6-3 展示了 Overlapped I/O with APCs 的运用。这份列表包含 IOBYAPC 的完整源代码。

列表 6-3 IOBYAPC.C——Overlapped I/O with APCs

```
#0001  /*
#0002   * IoByAPC.c
#0003   *
#0004   * Sample code for Multithreading Applications in Win32
#0005   * This is from Chapter 6, Listing 6-3
#0006   *
#0007   * Demonstrates how to use APC's (asynchronous
#0008   * procedure calls) instead of signaled objects
#0009   * to service multiple outstanding overlapped
#0010   * operations on a file.
#0011   */
#0012
#0013  #define WIN32_LEAN_AND_MEAN
#0014  #include <stdio.h>
#0015  #include <stdlib.h>
#0016  #include <windows.h>
#0017  #include "MtVerify.h"
#0018
#0019  //
#0020  // Constants
#0021  //
#0022  #define MAX_REQUESTS    5
#0023  #define READ_SIZE       512
#0024  #define MAX_TRY_COUNT   5
#0025
#0026  //
#0027  // Function prototypes
```

```
#0028  //
#0029  void CheckOsVersion();
#0030    int  QueueRequest(int  nIndex,  DWORD  dwLocation,  DWORD
dwAmount);
#0031
#0032
#0033  //
#0034  // Global variables
#0035  //
#0036
#0037  // Need a single event object so we know when all I/O is finished
#0038  HANDLE  ghEvent;
#0039  // Keep track of each individual I/O operation
#0040  OVERLAPPED gOverlapped[MAX_REQUESTS];
#0041  // Handle to the file of interest.
#0042  HANDLE ghFile;
#0043  // Need a place to put all this data
#0044  char gBuffers[MAX_REQUESTS][READ_SIZE];
#0045  int nCompletionCount;
#0046
#0047  /*
#0048   * I/O Completion routine gets called
#0049   * when app is alertable (in WaitForSingleObjectEx)
#0050   * and an overlapped I/O operation has completed.
#0051   */
#0052  VOID WINAPI FileIOCompletionRoutine(
#0053      DWORD dwErrorCode,  // completion code
#0054      DWORD dwNumberOfBytesTransfered, //number of bytes transferred
#0055      LPOVERLAPPED lpOverlapped   // pointer to structure with
I/O information
#0056     )
#0057  {
#0058      // The event handle is really the user defined data
#0059      int nIndex = (int)(lpOverlapped->hEvent);
#0060      printf("Read #%d returned %d. %d bytes were read.\n",
#0061          nIndex,
#0062          dwErrorCode,
#0063          dwNumberOfBytesTransfered);
#0064
#0065      if (++nCompletionCount == MAX_REQUESTS)
#0066          SetEvent(ghEvent);  // Cause the wait to terminate
#0067  }
#0068
#0069
```

```
#0070  int main()
#0071  {
#0072      int i;
#0073      char szPath[MAX_PATH];
#0074
#0075      CheckOsVersion();
#0076
#0077      // Need to know when to stop
#0078      MTVERIFY(
#0079          ghEvent = CreateEvent(
#0080                      NULL,// No security
#0081                      TRUE,// Manual reset-extremely important!
#0082                      FALSE,// Initially set Event to non-signaled state
#0083                      NULL     // No name
#0084                     )
#0085      );
#0086
#0087      GetWindowsDirectory(szPath, sizeof(szPath));
#0088      strcat(szPath, "\\WINHLP32.EXE");
#0089      // Open the file for overlapped reads
#0090      ghFile = CreateFile( szPath,
#0091                   GENERIC_READ,
#0092                   FILE_SHARE_READ|FILE_SHARE_WRITE,
#0093                   NULL,
#0094                   OPEN_EXISTING,
#0095                   FILE_FLAG_OVERLAPPED,
#0096                   NULL
#0097                );
#0098      if (ghFile == INVALID_HANDLE_VALUE)
#0099      {
#0100          printf("Could not open %s\n", szPath);
#0101          return -1;
#0102      }
#0103
#0104      // Queue up a few requests
#0105      for (i=0; i<MAX_REQUESTS; i++)
#0106      {
#0107          // Read some bytes every few K
#0108          QueueRequest(i, i*16384, READ_SIZE);
#0109      }
#0110
#0111      printf("QUEUED!!\n");
```

```
#0112
#0113     // Wait for all the operations to complete.
#0114     for (;;)
#0115     {
#0116        DWORD rc;
#0117        rc = WaitForSingleObjectEx(ghEvent, INFINITE, TRUE );
#0118        if (rc == WAIT_OBJECT_0)
#0119           break;
#0120        MTVERIFY(rc == WAIT_IO_COMPLETION);
#0121     }
#0122
#0123     CloseHandle(ghFile);
#0124
#0125     return EXIT_SUCCESS;
#0126  }
#0127
#0128
#0129  /*
#0130   * Call ReadFileEx to start an overlapped request.
#0131   * Make sure we handle errors that are recoverable.
#0132   */
#0133  int QueueRequest(int nIndex, DWORD dwLocation, DWORD dwAmount)
#0134  {
#0135     int i;
#0136     BOOL rc;
#0137     DWORD err;
#0138
#0139     gOverlapped[nIndex].hEvent = (HANDLE)nIndex;
#0140     gOverlapped[nIndex].Offset = dwLocation;
#0141
#0142     for (i=0; i<MAX_TRY_COUNT; i++)
#0143     {
#0144        rc = ReadFileEx(
#0145           ghFile,
#0146           gBuffers[nIndex],
#0147           dwAmount,
#0148           &gOverlapped[nIndex],
#0149           FileIOCompletionRoutine
#0150        );
#0151
#0152        // Handle success
#0153        if (rc)
```

```
#0154         {
#0155            // asynchronous i/o is still in progress
#0156             printf("Read #%d queued for overlapped I/O.\n",
nIndex);
#0157            return TRUE;
#0158         }
#0159
#0160         err = GetLastError();
#0161
#0162         // Handle recoverable error
#0163         if ( err == ERROR_INVALID_USER_BUFFER ||
#0164              err == ERROR_NOT_ENOUGH_QUOTA ||
#0165              err == ERROR_NOT_ENOUGH_MEMORY )
#0166         {
#0167            Sleep(50);  // Wait around and try later
#0168            continue;
#0169         }
#0170
#0171         // Give up on fatal error.
#0172         break;
#0173      }
#0174
#0175      printf("ReadFileEx failed.\n");
#0176      return -1;
#0177   }
#0178
#0179   //
#0180   // Make sure we are running under an operating
#0181   // system that supports overlapped I/O to files.
#0182   //
#0183   void CheckOsVersion()
#0184   {
#0185      OSVERSIONINFO   ver;
#0186      BOOL           bResult;
#0187
#0188      ver.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
#0189
#0190      bResult = GetVersionEx((LPOSVERSIONINFO) &ver);
#0191
#0192      if ( (!bResult) ||
#0193           (ver.dwPlatformId != VER_PLATFORM_WIN32_NT) )
#0194      {
#0195        fprintf(stderr, "IoByAPC must be run under Windows NT.\n");
```

```
#0196          exit(EXIT_FAILURE);
#0197       }
#0198
#0199    }
```

本例的 QueueRequest() 函数非常类似 IOBYEVNT 中的同名函数。最大的差别在于并没有把 “returned immediately” 和 “not complete” 区分开来。你可以看到，这个函数也处理了“系统缺乏资源”的情况。

本例新增的函数是 FileIOCompletionRoutine()。这是一个 callback 函数，当 overlapped I/O 完成的时候，由系统调用之。

最后，请你注意，main() 调用 WaitForSingleObjectEx()，所以 APCs 会被处理。我们必须在一个循环中完成此事，因为在处理完 APCs 之后，WaitForSingleObjectEx() 会传回 WAIT_IO_COMPLETION。

在 C++ 中产生一个 I/O Completion Routines

? FAQ 29：我如何把 C++ 成员函数当做一个 I/O completion routine?

一个 C++ 成员函数不能拿来当做一个 I/O completion routine，除非它是一个 static 函数。如果它是 static，它就没有 this 指针，也就不能够直接调用那些需要 this 指针的成员函数。

解决之道是储存一个指针，指向用户自定义数据（一个对象），然后经由此指针调用一个 C++ 成员函数。由于 static 成员函数是类的一部分，你还是可以调用 private 成员函数。这个问题和“如何把 C++ 成员函数当做一个线程函数”非常类似。我会在第 9 章描述解决方法。

APC Wrap-up

尽管我们已经知道 APCs 的存在，我却很少看到它的应用——不管是用在等待 ReadFile() 和 WriteFile() 的文件 handle 或 event 对象身上。如果

你使用上述任何一种技术，你都必须理解被激发之 handle 的意义，并将它分派到你自己的处理函数中。总的来说，这是浪费时间，因为系统有能力为你处理所有令人发牢骚的琐事。

对文件进行 Overlapped I/O 的缺点

作为撰写本书的一部分工作，我多方面测试了 overlapped I/O 的使用效益。结果十分令人意外。似乎 Windows NT 是以“I/O 请求”的大小来决定要不要将此请求先记录下来。如果你要求的是 8KB 数据，这很小，Windows NT 会先将你的线程停住，立刻完成这 8KB 数据的操作，而不会先把 I/O 请求摆在一旁。

这会带来怎样的问题呢？对于比 64KB 小的磁盘传输量而言，用于移动磁盘读写头的平均时间，比用于搬移数据的平均时间还多。假设传输率是 5MB/秒（通常一个 PCI 总线加上一个 SCSI 控制器，再搭配一个 SCSI-II 外围设备，而且数据没有被 “cached” 在主机端或设备端，就拥有这种速度），通常 10 ms 用来移动磁盘读写头，1.8 ms 用来搬移 8KB 数据。这些时间并不包括 SCSI 总线的 setup time，以及其他一些驱动程序上的耗费时间。

由于操作系统在面对快速数据传输时，选择将你的进程先停下来，所以你的进程遗失至少（平均）12 ms 的时间。但如果你企图搬移的是 1MB 的数据，I/O 操作会被先记录起来，你的线程可以继续进行。

我的测试显示，对于一系列少于 32KB 的数据传输请求，使用 overlapped I/O 所花的时间比单纯调用 ReadFile() 所花时间要多 15%。在像 Web 服务器这样的环境中，几乎每一笔数据传输量都很小，使用 overlapped I/O 反而会降低整体效率，而不是让你的程序更自由自在地做后台工作。

一种合理的变化是，以少量的线程负责所有的硬盘 I/O，然后把对这些线程的 I/O 请求，保持在一个队列之中。这种模式接近 I/O completion ports

（稍后描述）。测试结果显示，这样的模式比 overlapped 操作或 non-overlapped 操作的效率都要高。

跳过虚拟内存管理器，直接进入文件系统也是可能的，只要在 CreateFile() 时使用 FILE_FLAG_NO_BUFFERING 标记值。然而，这么做也可能会跳过 cache 管理器，于是有引发剧烈的效能降低的潜在性。相关讨论已经逾越了本书范围。有一篇关于 Windows NT 文件系统的好文章，出现在 Microsoft Systems Journal，1995 年七月号："Design and Implementation of the Windows NT Virtual Block Cache Manager"，作者是 Helen Custer。

最后我还要说一点。有两种情况，overlapped I/O 总是同步执行，甚至即使 FILE_FLAG_NO_BUFFERING 已经指定。第一种情况是你进行一个写入操作而造成文件的扩展。第二种情况是你读写一个压缩文件。

I/O Completion Ports

虽然 APCs 是完成 overlapped I/O 的一个非常便捷的方法，但它们还是有它们的缺点。最大的问题就是，有好几个 I/O APIs 并不支持 APCs，如 listen() 和 WaitCommEvent() 便是两个例子。APCs 的另一个问题是，只有发出"overlapped 请求"的那个线程才能够提供 callback 函数。然而在一个"scalable"（译注）系统中，最好任何线程都能够服务 events。

译注 所谓 scalable 系统，是指藉着 RAM 或磁盘空间或 CPU 个数的增加而能够提升应用程序效能的一种系统。

? FAQ 30：在一个高效率服务器（server）上我应该怎么进行 I/O？

在 Windows NT 3.5 中有第四种 overlapped I/O 的存在，称为 I/O completion ports。靠着"一大堆线程服务一大堆 events"的性质，completion ports 比较容易建立起 "scalable" 服务器。

尽管名称类似，I/O completion ports 与 APCs 中所用的 I/O completion routines 没有任何关联

Completion ports 解决了我们截至目前看到的所有问题：

- 与 WaitForMultipleObjects() 不同，这里不限制 handles 的个数。
- I/O completion ports 允许一个线程将一个请求暂时保存下来，而由另一个线程为它做实际服务。
- I/O completion ports 默默支持 scalable 架构。

服务器的线程模型

有三个基本方法，可以决定一个服务器上需要多少个线程：

- 单独一个线程。在一个文件服务器或一个简单的 Web 服务器上，单一线程可以使用 overlapped I/O 来搬移数据。这个线程可以常保 CPU 和磁盘尽可能地忙碌。然而，如果线程还必须进行任何其他操作，则整个服务器会陷入泥淖之中。

- 每个 client 给予一个线程。如果你为每一个 client 产生一个线程，那么理论上每个人都可以分到相当不错的反应时间，因为 CPU 的能量被平均分配了。然而事实上系统资源是有限的，到了某种情况下，系统效率就会剧烈下降。如果有 2000 个 clients，这种做法就不切实际了。

- 每个 CPU 给予一个线程。这种做法让每一个 CPU 尽可能忙碌，不至于出现哪一个 CPU 有过度饱和的情况。如果你使用 event 对象或 APCs，此法将难以实现，因为它们都十分紧密地和某个线程绑在一起。这个问题的解决方案必须靠一种特殊的同步对象，称为 I/O completion ports。

I/O completion port 是一种非常特殊的核心对象，用来综合一堆线程，让它们为“overlapped 请求”服务。其所提供的功能甚至可以跨越多个 CPUs。Completion port 可以自动补偿成长中的服务器，适合应用于沉重的负担。

Completion Ports 做些什么

截至目前，从本章之中我们已经看到了，一个 overlapped I/O 操作的完成，可以藉由文件 handle 的激发、event 对象的激发，或是 APC，通知某个对此感兴趣的线程。I/O completion port 的运作方式截然不同。

我们搜寻的目标是任何一个能够为“completed overlapped I/O request”服务的线程。如果使用其他种类的 overlapped I/O 机制，你必须锁定，并且由一个特定的线程为一个特定的“I/O 请求”服务。I/O completion port 允许你将“启动 overlapped 请求的线程”和“提供服务的线程”拆伙。

为了使用 I/O completion port，你的程序应该产生一堆线程，统统在 I/O completion port 上等待着。这些线程都将成为“能够处理 completed overlapped I/O request”的线程之一——只要线程在 I/O completion port 上等待，它就自然而然成为那种线程。

每次有新的文件因为 overlapped I/O 而开启，你就可以让它的文件 handle 和 I/O completion port 产生关联。一旦这样的关系建立起来，任何文件操作如果成功完成，便会导致 I/O completion packet 被送到 completion port 去。这是发生于操作系统之内的操作，对应用程序而言是透明的。

为了回应 I/O completion packet，completion port 释放了一个等待中的线程。如果目前并没有线程正在等待，completion port 不会产生新线程。

释放出来的线程被授予足够的信息，使它能够辨识“completed overlapped I/O operation”的背景环境，然后这个线程就起而行，处理该操作请求。但它还是属于原来那一堆线程（指定给此 completion port）的一分子，差别在于这个线程现在成为一个作用中的（active）线程，而不是一个等待中的（waiting）线程。当这个线程将“overlapped I/O 请求”处理完毕时，它应该再次在这个 I/O completion port 上等待。

? FAQ 31：为什么一个 I/O completion ports 是如此特殊？

以此概观为基础，我大略可以这样描述一个 completion port：它是一个机制，用来管理一堆线程如何为 completed overlapped I/O requests 服务。然而，completion port 远比一个简单的分派器丰富得多，I/O completion port 也像一个活门（阀）一样，保持一个 CPU 或多个 CPUs 尽可能地忙碌，但也避免它们被太多的线程淹没。I/O completion port 企图保持并行处理的线程个数在某个数字左右。一般而言你希望让所有的 CPUs 都忙碌，所以默认情况下并行处理的线程个数就是 CPUs 的个数。

I/O completion port 运作过程中令人迷惑的部分就是，当一个线程被阻塞住时，它将发出通告，并提交另一个线程。假设在单一 CPU 的系统中，有两个线程都正在一个 I/O completion port 上等待，线程 1 被唤醒，并且从网络上获得一包数据。为了服务这个数据包，线程 1 必须从磁盘中读一个文件，所以它调用 CreateFile() 和 ReadFile()，但不在 overlapped 模式中。

Completion port 于是通告说线程 1 被磁盘 I/O 滞留了，并且提交线程 2 以使“目前作用中的线程个数”到达需求数量。

当线程 1 从磁盘操作中返回时，或许现在有两个线程同时在执行——甚至即使“作用中的线程个数”应该只是 1（因为 CPU 个数只是 1）。这一行为令人惊讶，但却是正确的。Completion port 不会再提交另一个线程，除非“作用中的线程个数”再次降到 1 以下。

操作概观

使用一个 completion port 的快速摘要列在下面。后续数小节中我将详细描述下面的每一点。

1. 产生一个 I/O completion port。
2. 让它和一个文件 handle 产生关联。
3. 产生一堆线程。
4. 让每一个线程都在 completion port 上等待。
5. 开始对着那个文件 handle 发出一些 overlapped I/O 请求。

当新文件被开启时，它们可以在任何时候与 I/O completion port 产生关联。在 completion port 上等待的线程不应该做“为 completion port 服务”以外的事情，因为这些线程将一直都是 completion port 所持续追踪的那一堆线程的一部分。

产生一个 I/O Completion Port

I/O completion port 是一个核心对象，你必须使用 CreateIoCompletionPort() 才能产生它：

```
HANDLE CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    DWORD CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

参数

FileHandle	文件或设备(device)的 handle。在 Windows NT 3.51 之后，此栏位可设定为 INVALID_HANDLE_VALUE，于是产生一个没有和任何文件 handle 有关系的 port。
ExistingCompletionPort	如果此栏位被指定，那么上一栏位 FileHandle 就会被加到此 port 之上，而不会产生新的 port。指定 NULL 可以产生一个新的 port。
CompletionKey	用户自定义的一个数值，将被交给提供服务的线程。此值和 FileHandle 有关联。
NumberOfConcurrentThreads	与此 I/O completion port 有关联的线程个数。

返回值

如果函数成功，则传回一个 I/O completion port 的 handle。如果函数失败，则传回 FALSE。GetLastError() 可以获得更详细的失败原因。

任何文件只要附着到一个 I/O completion port 身上，都必须先以 FILE_FLAG_OVERLAPPED 开启。如果已经附着上去，就不能够再以 ReadFileEx() 或 WriteFileEx() 操作它。你可以任意关闭这样的一个文件，没有什么安全上的顾虑。

请注意在 Windows NT 3.5 中你必须给予一个合法的 FileHandle。但是在 Windows NT 3.51 之后，你可以给予 INVALID_HANDLE_VALUE。其实问题也可以绕一个弯解决：你产生一个临时性文件，然后用它产生一个 completion port。

通常你会想要把 NumberOfConcurrentThreads 设定为 0，如此一来，在多 CPU 系统下，就可以有尽可能多的线程运行起来。这可以使每一个 CPU 都尽可能地忙碌，降低因过度的 context switch 而造成的额外浪费（overhead）。

与一个文件 handle 产生关联

CreateIoCompletionPort() 通常被调用两次。第一次先指定 FileHandle 为 INVALID_HANDLE_VALUE，并设定 ExistingCompletionPort 为 NULL，用以产生一个 port。然后再为每一个欲附着上去的文件 handle 调用一次 CreateIoCompletionPort()。后续的这些调用应该将 ExistingCompletionPort 设定为第一次调用所传回的 handle。例如：

```
HANDLE hPort;
HANDLE hFiles[MAX_FILES];
int index;

//Create the completion port
hPort = CreateIoCompletionPort(
```

```
                    INVALID_HANDLE_VALUE,
                    NULL,
                    0,    // key
                    0     // default # of threads
        );

// Now associate each file handle
for (index = 0; index < OPEN_FILES; index++)
{
    CreateIoCompletionPort(
            hFiles[index],
            hPort,
            0,      // key
            0       // default # of threads
    );
}
```

请注意所有操作都在一个线程中完成。通常，completion port 是在同一个线程中完成设立。真正用来提供服务的那些 worker 线程此时不需要做什么贡献。

产生一堆线程

一旦 completion port 产生出来，你就可以设立在该 port 上等待的那些线程了。I/O completion port 并不自己产生那些线程，它只是使用由你产生的线程。因此，你必须自己以 CreateThread() 或 _beginthreadex()（第 8 章）或 AfxBeginThread()（第 10 章）产生出线程。

当你一产生这些线程时，它们都应该在 completion port 上等待。当线程开始为各个“请求”服务时，池子里的线程的组织如下：

```
        目前正在执行的线程
+       被阻塞的线程
+       在 completion port 上等待的线程
----------------------------------------
=       池子里的所有线程的个数
```

因为如此，所以你应该产生比 CPU 个数还多的线程．如果你只有一个 CPU，而你也只产生了一个线程，那么当该线程阻塞（blocking）时，你的 CPU 也变成闲置（idle）的了．由于池子里没有其他线程，completion port 也就没有办法为任何数据包（packets）服务——甚至即使 CPU 的能量游刃有余．

? FAQ 32：一个 I/O completion port 上应该安排多少个线程等待？

合理的线程个数应该是 CPU 个数的两倍再加 2．没有理由说你不能够产生更多的线程，但记住，线程并不是免费的，100 个 worker 线程在 completion port 上等待，并不会让系统速度更快一些．

在一个 I/O Completion Port 上等待

Worker 线程初始化自己之后，它应该调用 GetQueuedCompletionStatus()．这个操作像是 WaitForSingleObject() 和 GetOverlappedResult() 的组合．函数规格如下：

```
BOOL GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytesTransferred,
    LPDWORD lpCompletionKey,
    LPOVERLAPPED *lpOverlapped,
    DWORD dwMilliseconds
);
```

参数

CompletionPort	将在其上等待的 completion port．
lpNumberOfBytesTransferred	一个指针，指向 DWORD．该 DWORD 将收到“被传输的数据字节数”．
lpCompletionKey	一个指针，指向 DWORD．该 DWORD 将收到由 CreateIoCompletionPort() 所定义的 key．
lpOverlapped	这个栏位的名称是个错误．它其实应该命名为 lpIpOverlapped，你应该把一个

	指针的地址放在上面。系统会填以一个 overlapped 结构的指针。该结构用以初始化 I/O 操作。
dwMilliseconds	等待的最长时间（毫秒）。如果时间终了，lpOverlapped 将被设为 NULL，而函数传回 FALSE。

返回值

如果函数成功地将一个 completion packet 从队列中取出，并完成一个成功的操作，函数将传回 TRUE，并填写由 lpNumberOfBytesTransferred、lpCompletionKey、lpOverlapped 所指向的变量内容。

如果操作失败，但 completion packet 已经从队列中取出，则函数传回 FALSE，lpOverlapped 指向失败之操作。调用 GetLastError() 可获知为什么 I/O 操作会失败。

如果函数失败，则传回 FALSE，并将 lpOverlapped 设为 NULL。调用 GetLastError() 可获知为什么函数会失败。

与其他的核心对象，如 semaphores 和 mutexes 不同，在 completion port 上等待的线程是以先进后出（first in last out，FILO）的次序提供服务。没有什么理由需要担忧次序的公平性，因为所有的线程都做完全相同的事情。使用先进后出（FILO）规则，一个进行中的线程调用 GetQueuedCompletionStatus() 就可以取得下一个请求（request），并保持执行状态，没有阻塞（blocking）。这是非常有效率的。线程如果等待太长的时间，就有可能会被置换出去（paged out）。最近执行过的线程则通常还在内存中，不需要先置换进来（paged in）才能执行。

发出“Overlapped I/O 请求”

下面这些调用可以启动“能够被一个 I/O completion port 掌握”的 I/O 操作：

- ConnectNamePipe()
- DeviceIoControl()
- LockFileEx()
- ReadFile()
- TransactNamePipe()
- WaitCommEvent()
- WriteFile()

为了使用 completion port，主线程（或任何其他线程）可以对着一个与此 completion port 有关联的文件，进行读、写、或其他任何操作。该线程不需要调用 WaitForMultipleObjects()，因为池子里的各个线程都曾经调用过 GetQueuedCompletionStatus()。一旦 I/O 操作完成，一个等待中的线程将会自动被释放，以服务该操作。

避免 Completion Packets

常常会有这种情况发生：你读（或写）一个文件，但是当操作完成时，你并不希望 I/O completion port 被通告。网络服务器就是一个例子，在那里，线程从一个文件中读进一个针对 named pipe 或 socket 的请求，然后将回应写到同一文件中。问题是文件已经以 overlapped I/O 状态打开了，所以写入操作将被 overlapped（译注：即异步）。而当写入操作完成时，I/O completion port 将收到一个 packet 中。如果写入操作的结果不是顶重要，那么服务器将花费许多时间去处理一些不怎么重要的 completion packets。解决之道是将“每个操作均会引发 completion port 通告”的开关关闭。

我们可以进行一个 I/O 操作，却不引发一个 I/O completion packet 被送往 completion port。我们可以以旧有的受激发的 event 对象机制取而代之。为了这么做，请你设定一个 OVERLAPPED 结构，内含一个合法的手动重置（manual-reset）event 对象，放在 hEvent 栏位。然后把该 handle 的最低位设为 1。虽然这听起来像是一种黑客（hack）行为，但文件中明明白白有交待。下面是个例子：

```
OVERLAPPED overlap;
HANDLE hFile;
char buffer[128];
DWORD dwBytesWritten;

memset(&overlap, 0, sizeof(OVERLAPPED));
overlap.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
overlap.hEvent = (HANDLE)((DWORD)overlap.hEvent | 0x1);

WriteFile(hFile, buffer, 128,& dwBytesWritten, &overlap);
```

对 Sockets 使用 Overlapped I/O

书附盘片中的 ECHO 范例程序，示范如何使用 I/O completion port 实现与标准的 TCP port 7 一样的行为，将每一个写往它的数据都摹仿一遍。这个程序包含两个可执行文件。ECHOSRV 是一个服务器，倾听 TCP port 5554（一个任意数值）并且将它从 socket 中读到的每一样东西写回相同的 socket。ECHOCLI 是一个客户端，负责取得你的输入，送往服务器，然后印出它从服务器收到的回应。

ECHOSRV 和 ECHOCLI 都是使用 TCP/IP 的 Winsock 应用程序。你的机器必须安装有 TCP/IP 才能够执行它们。由于 ECHOSRV 有 completion port，所以它只能够在 Windows NT 3.51（或更新版本）中执行。ECHOCLI 可以在 Windows 95 中执行，如果你更改其 IP 地址的话。两个程序目前都是和 127.0.0.1 交谈，那是本机（local host）的一个别名。

服务器可以处理同时发生的各种请求。由于在不同窗口上同时输入字符很困难，我在书附盘片中提供了一个批处理文件，名为 TESTME.BAT，它会开启数个窗口，每一个窗口都从相同的文本文件中读数据，而不再依赖键盘输入。

? FAQ 33：为什么我不应该使用 select()?

Completion port 是在 Windows NT 机器上与网络交谈时唯一能够获得高产能的方法。虽然 Windows NT 也支持 select()，而那是同时支持多路复用的标准方法，但是 select() 不能够因为多 CPUs 而对应用程序效能有所提升。

David Treadwell，Windows Socket 的一位开发者，曾经在他的一篇名为“Developing Transport-Independent Applications Using the Windows Sockets Interface” 的文章（可从 Microsoft Developer's Network CD-ROM 查得）中提到过，凡使用 select() 的应用程序，其效率可能受损，因为每一个网络 I/O call 都会经过 select()，因而招致严重的 CPU 额外负担。这种效率在 CPU 的使用率不是关键因素时，可以被接受，但是当需要高效率时，当然就会带来问题。

当你调用 CreateIoCompletionPort()，将一个文件和一个 completion port 产生关联时，你必须交出一个 key。这个 key 只不过是用户自定义的一个数值而已。在 ECHOSRV 中我配置了一个名为 ContextKey 的结构，用以追踪在目前这个 handle 上什么东西被读或什么东西被写。结构如下：

```
struct ContextKey
{
   SOCKET       sock;
   // Input
   char         InBuffer[4];
   OVERLAPPED    ovIn;
   // Output
   int          nOutBufIndex;
   char         OutBuffer[MAXLINE];
   OVERLAPPED    ovOut;
   DWORD        dwWritten;
};
```

InBuffer 被用于 overlapped 读取。OutBuffer 被用于 overlapped 写入。当字符一次一个被读进 InBuffer 时，它们也被累积到 OutBuffer 中，当完整的一行完成后，它们就被写回 socket。

ECHOSRV 的 main() 函数显示于列表 6-4。它开启 socket，产生 I/O completion port，启动 worker 线程，并进入循环之中等待连接。它正是之前曾经说过的五大步骤的实现。请注意一点：一个 SOCKET 可以被放到原本期望一个 HANDLE 的地方。

列表 6-4 ECHOSRV.C 中的 main()——设立一个 I/O completion port

```
#0001  int main(int argc, char *argv[])
#0002  {
#0003      SOCKET  listener;
#0004      SOCKET  newsocket;
#0005      WSADATA WsaData;
#0006      struct sockaddr_in serverAddress;
#0007      struct sockaddr_in clientAddress;
#0008      int     clientAddressLength;
#0009      int     err;
#0010
#0011      CheckOsVersion();
#0012
#0013      err = WSAStartup (0x0101, &WsaData);
#0014      if (err == SOCKET_ERROR)
#0015      {
#0016          FatalError("WSAStartup Failed");
#0017          return EXIT_FAILURE;
#0018      }
#0019
#0020      /*
#0021       * Open a TCP socket connection to the server
#0022       * By default, a socket is always opened
#0023       * for overlapped I/O.  Do NOT attach this
#0024       * socket (listener) to the I/O completion
#0025       * port!
#0026       */
#0027      listener = socket(AF_INET, SOCK_STREAM, 0);
#0028      if (listener < 0)
#0029      {
#0030          FatalError("socket() failed");
#0031          return EXIT_FAILURE;
#0032      }
#0033
```

```
#0034     /*
#0035      * Bind our local address
#0036      */
#0037     memset(&serverAddress, 0, sizeof(serverAddress));
#0038     serverAddress.sin_family      = AF_INET;
#0039     serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
#0040     serverAddress.sin_port        = htons(SERV_TCP_PORT);
#0041
#0042     err = bind(listener,
#0043            (struct sockaddr *)&serverAddress,
#0044            sizeof(serverAddress)
#0045          );
#0046     if (err < 0)
#0047        FatalError("bind() failed");
#0048
#0049     ghCompletionPort = CreateIoCompletionPort(
#0050            INVALID_HANDLE_VALUE,
#0051            NULL,   // No prior port
#0052            0,      // No key
#0053            0       // Use default  # of threads
#0054            );
#0055     if (ghCompletionPort == NULL)
#0056        FatalError("CreateIoCompletionPort() failed");
#0057
#0058     CreateWorkerThreads();
#0059
#0060     listen(listener, 5);
#0061
#0062     fprintf(stderr, "Echo Server with I/O Completion Ports\n");
#0063     fprintf(stderr, "Running on TCP port %d\n", SERV_TCP_PORT);
#0064     fprintf(stderr, "\nPress Ctrl+C to stop the server\n");
#0065
#0066     //
#0067     // Loop forever accepting requests new connections
#0068     // and starting reading from them.
#0069     //
#0070     for (;;)
#0071     {
#0072        struct ContextKey *pKey;
#0073
#0074        clientAddressLength = sizeof(clientAddress);
```

```
#0075         newsocket = accept(listener,
#0076                            (struct sockaddr *)&clientAddress,
#0077                            &clientAddressLength);
#0078         if (newsocket < 0)
#0079         {
#0080             FatalError("accept() Failed");
#0081             return EXIT_FAILURE;
#0082         }
#0083
#0084         // Create a context key and initialize it.
#0085         // calloc will zero the buffer
#0086         pKey = calloc(1, sizeof(struct ContextKey));
#0087         pKey->sock = newsocket;
#0088         pKey->ovOut.hEvent = CreateEvent(NULL,TRUE,FALSE, NULL);
#0089         // Set the event for writing so that packets
#0090         // will not be sent to the completion port when
#0091         // a write finishes.
#0092         pKey->ovOut.hEvent = (HANDLE)((DWORD)pKey->ovOut.hEvent| 0x1);
#0093
#0094         // Associate the socket with the completion port
#0095         CreateIoCompletionPort(
#0096                (HANDLE)newsocket,
#0097                ghCompletionPort,
#0098                (DWORD)pKey,   // Key
#0099                0              // Use default # of threads
#0100            );
#0101
#0102         // Kick off the first read
#0103         IssueRead(pKey);
#0104     }
#0105     return 0;
#0106 }
```

函数 main() 调用 CreateWorkerThreads()，后者启动数个线程。不需要给予线程启动参数，因为 completion port 是全局变量，每一个人都可以处理之。CreateWorkerThreads() 显示于列表 6-5。

列表 6-5 ECHOSRV.C 中的 CreateWorkerThreads()

```
#0001  void CreateWorkerThreads()
#0002  {
#0003      SYSTEM_INFO  sysinfo;
#0004      DWORD        dwThreadId;
#0005      DWORD        dwThreads;
#0006      DWORD        i;
#0007
#0008      GetSystemInfo(&sysinfo);
#0009      dwThreads = sysinfo.dwNumberOfProcessors * 2 + 2;
#0010      for (i=0; i<dwThreads; i++)
#0011      {
#0012          HANDLE hThread;
#0013          hThread = CreateThread(
#0014              NULL, 0, ThreadFunc, NULL, 0, &dwThreadId
#0015              );
#0016          CloseHandle(hThread);
#0017      }
#0018  }
```

最后一个有趣的函数是 ThreadFunc()，那是 worker 线程花掉所有时间的地方。一旦线程启动，它就进入永远的循环，在其中调用 GetQueueCompletionStatus()。当每一个 completion packet 被处理之后，我们使用 GetQueueCompletionStatus() 传回来的 key，当做指向 ContextKey 结构的指针，我们已经在该结构中放置了许多必要的信息。

回到 main() 函数，我们为 overlapped 结构产生一个 event 对象，并将其最低位设立，以表示当写入操作完成时，不需要任何 packet 被送往 completion port。范例程序并未检查完成后的“overlapped 写入操作”的结果。

ThreadFunc() 显示于列表 6-6。请注意 bResult 所代表的各种可能错误状态是如何地排列，以及 lpOverlapped 指针是如何地检验。

列表 6-6 ECHOSRV.C 中的 ThreadFunc()

```
#0001  DWORD WINAPI ThreadFunc(LPVOID pVoid)
#0002  {
#0003      BOOL    bResult;
#0004      DWORD   dwNumRead;
#0005      struct ContextKey *pCntx;
#0006      LPOVERLAPPED lpOverlapped;
#0007
#0008      UNREFERENCED_PARAMETER(pVoid);
#0009
#0010      // Loop forever on getting packets from
#0011      // the I/O completion port.
#0012      for (;;)
#0013      {
#0014          bResult = GetQueuedCompletionStatus(
#0015                ghCompletionPort,
#0016                &dwNumRead,
#0017                &(DWORD)pCntx,
#0018                &lpOverlapped,
#0019                INFINITE
#0020             );
#0021
#0022          if (bResult == FALSE
#0023             && lpOverlapped == NULL)
#0024          {
#0025             FatalError(
#0026          "ThreadFunc - Illegal call to GetQueuedCompletionStatus");
#0027          }
#0028
#0029          else if (bResult == FALSE
#0030             && lpOverlapped != NULL)
#0031          {
#0032             // This happens occasionally instead of
#0033             // end-of-file. Not sure why.
#0034             closesocket(pCntx->sock);
#0035             free(pCntx);
#0036             fprintf(stderr,
#0037                "ThreadFunc - I/O operation failed\n");
#0038          }
#0039
```

```
#0040         else if (dwNumRead == 0)
#0041         {
#0042            closesocket(pCntx->sock);
#0043            free(pCntx);
#0044            fprintf(stderr, "ThreadFunc - End of file.\n");
#0045         }
#0046
#0047         // Got a valid data block!
#0048         // Save the data to our buffer and write it
#0049         // all back out (echo it) if we have see a \n
#0050         else
#0051         {
#0052            // Figure out where in the buffer to save the character
#0053            char *pch = &pCntx->OutBuffer[pCntx->nOutBufIndex++];
#0054            *pch++ = pCntx->InBuffer[0];
#0055            *pch = '\0';    // For debugging, WriteFile doesn't care
#0056            if (pCntx->InBuffer[0] == '\n')
#0057            {
#0058               WriteFile(
#0059                      (HANDLE)(pCntx->sock),
#0060                      pCntx->OutBuffer,
#0061                      pCntx->nOutBufIndex,
#0062                      &pCntx->dwWritten,
#0063                      &pCntx->ovOut
#0064                   );
#0065               pCntx->nOutBufIndex = 0;
#0066               fprintf(stderr, "Echo on socket %x.\n", pCntx->sock);
#0067            }
#0068
#0069            // Start a new read
#0070            IssueRead(pCntx);
#0071         }
#0072      }
#0073
#0074      return 0;
#0075   }
```

虽然这个服务器没有最优化，因为它一次只读一个字节，但它却是最简单的例子，可以适当地示范 I/O completion ports。ECHOSRV 的另一些辅助函数并未列出于列表 6-6 中，client 方面则完全没有列出。你可以在书附盘片的

ECHO 子目录中找到完整的源代码。

提要

这一章带着你做了一趟旋风般的旅游，游览所谓的 overlapped I/O。那是一种实现异步 I/O 的技术，可以避免使用多线程。Overlapped I/O 可以藉着使用激发的文件 handles，或是激发的 event 对象，或是异步过程调用（APCs），或是 I/O completion ports 而完成。

I/O completion ports 非常重要，因为它们作为产生高效率的服务器很受欢迎，而且也具有 “scalable” 的性质。ECHO 范例程序即表现出此种类型的服务器，程序中实现出一个使用 I/O completion ports 的 Winsock 应用程序。

第二篇

多线程程序设计的工具与手法

Multithreading Tools and Tricks

第 7 章　数据一致性（Data Consistency）..... 195

第 8 章　使用 C Run-time Library..... 219

第 9 章　使用 C++..... 253

第 10 章　MFC 中的线程..... 277

第 11 章　GDI 与窗口管理..... 303

第 12 章　调试..... 321

第 13 章　进程之间的通讯（Interprocess Communication）..... 335

第 14 章　建造 DLLs..... 367

数据一致性

Data Consistency

这一章描述 volatile 关键字的使用，并讨论 Readers/Writers Lock 的设计和运用。

到目前为止我们已经看过了许多小程序，它们分别被小心地设计用来解释某些观念。在第 2 章和第 3 章中，我们尝试后台打印。但是我们忽略了在线程之间共享数据的问题。做法是将数据预先包装，使得只有一个线程必须读它。在第 4 章中我们看到了 EVENTTST 程序，展示了 event 对象如何运作，但没有介绍它们的应用。没有一个例子真正考虑到现实世界的问题，程序不能够被灵活地切割，以适用于最理想或最简单的模型。

这一章将带你看看，当多个线程正在读写同一个数组，并且线程又被动态“喂”以其他数据时，该如何处理真实世界中的数据。这种形势明显地复杂得多。

认识 volatile 关键字

甚至在你花了许多的努力，正确写出一个多线程程序之后，你会不会怀疑编译器是否会产生不安全的程序代码？答案是“Yes”。有时候你必须告诉编译器：数据被共享，然后适当的程序代码才能够被造出来。

我相信你一定遇到过这样的问题：你把某人的名字和电话号码写到你的通讯录中，数个月之后企图打电话给这个人，却发现资料已经过期了。同样的情况也可能发生在编译器为你产生的程序代码中。

编译器最优化的结果是，设法把常用到的数据放在 CPU 的内部寄存器中。这些寄存器就像你的通讯录一样。数据从寄存器中读出，远比从内存中读出快得多。就好像你从你的通讯录中读数据，远比从大电话簿中读数据要快得多。当然啦，如果另一个线程改变了内存中的变量值，那么此变量在寄存器中的拷贝值就算是“过期”了。

在一个单线程中这种情况不可能发生。编译器可以分析你的程序的每一个操作，然后确保数据在适当时候会被重新载入。然而在一个多线程程序中就不可能知道其他线程在做什么，所以编译器一定不能够允许让一个共享变量的值拷贝到寄存器中。

对专家而言…

可能你曾经注意过编译器选项 /Oa，它可以“Assume to aliasing”。这个最优化操作通常是被关闭的。当你产生一个指针指向一个数值，而不是直接处理该数值时，“aliasing”便会发生。例如下面这一行就会产生一个 alias（别名）：

```
int *pVal = &Contexts[j].buf[index];
```

Aliasing 很容易迷惑编译器，所以编译器需要一些操作来保护之，那些操作会带来轻微的效率报应（变差）。

Aliasing 事实上会引起和多线程所制造出来的一样的问题，因为当一个变量内容改变时，编译器并不总是能够知道。

C 和 C++ 有一个鲜为人知的关键字，教导编译器如何在一个 variable-by-variable 的基础上采取行动。这个关键字是 volatile（译注：中文意思是“有挥发性的、易变的”）。这个关键字告诉编译器不要持有变量的临时性拷贝。它可以适用于基础类型，如 int 或 long，也适用于一整个 C 结构或 C++ 类。后面这种情况下，结构或类的所有成员都会被视为 volatile。

使用 volatile 并不会否定 critical sections 或 mutexes 的需要。例如你说：

```
a = a + 3
```

还是会有一小段时间，a 会被放置在一个寄存器中，因为算术运算只能够在寄存器中进行。一般而言，volatile 关键字适用于行与行之间，而不是放在行内。

让我们看一个非常简单的函数，观察编译器制造出来的汇编语言码中的瑕疵，并看看 volatile 如何修正这个瑕疵。这个范例函数是一个 busy loop，虽然我早就耳提面命地告诉你不要写出一个 busy loop，但它却是解释这里的观念的一个最好的例子。本例之中，WaitForKey() 等待一个字符的到来：

```
void WaitForKey(char *pch)
{
   while (*pch == 0)
      ;
}
```

当你把最优化选项统统关闭后，编译这个程序，获得以下结果。进入点和退出点已经被我移除（为了让程序代码清爽一些）。粗体字代表 C 源代码。

```
;       while (*pch == 0)
$L27:
        ; Load the address stored in pch
        mov     eax, DWORD PTR _pch$[ebp]
        ; Load the character into the EAX register
        movsx   eax, BYTE PTR [eax]
        ; Compare the value to zero
        test    eax, eax
        ; If not zero, exit loop
        jne     $L28
;       ;
        jmp     $L27
$L28:
; }
```

这个未曾最优化的函数代码不断地载入适当的地址，载入地址中的内容，测试其结果。慢，但是准确。此版本在多线程程序上没有问题。

? FAQ 34: volatile 如何影响编译器的最优化操作?

现在我们看看最优化带来什么影响：

```
; {
        ; Load the address stored in pch
        mov     eax, DWORD PTR _pch$[esp-4]
        ; Load the character into the AL register
        movsx   al, BYTE PTR [eax]
;       while (*pch == 0)
$L84:
        ; Compare the value in the AL register to zero
        test    al, al
        ; If still zero, try again
        je      SHORT $L84
;       ;
; }
```

短多了。最优化果然有用。但是请注意编译器把 MOV 指令放到循环之外，这个操作称为 loop-invariant removal。这在单线程程序中应该是一个很好的最优化，但是在多线程程序中，如果另一个线程改变了数值，则循环永远不会结束。被测试的值永远被放在寄存器中，很明显那是一只“臭虫”。

解决办法是重写 WaitForKey()，把参数 pch 声明为 volatile：

```
void WaitForKey(volatile char *pch)
{
    while (*pch == 0)
        ;
}
```

这项改变对于非最优化的版本没有影响，但请你看看最优化后的结果：

```
; {
        ; Load the address stored in pch
        mov     eax, DWORD PTR _pch$[esp-4]
;       while (*pch == 0)
$L84:
        ; Directly compare the value to zero
        cmp     BYTE PTR [eax], 0
        ; If still zero, try again
        je      SHORT $L84
;       ;
; }
```

这个版本几乎完美，地址不会改变，所以地址声明被移到循环之外。地址内容是 volatile，所以每次循环之中它不断地被重新检查。

精细地说，把一个 const volatile 变量传给函数作为参数是合法的。如此的声明意味着函数不能够改变变量的值，但是变量的值却可以被另一个线程在任何时间改变掉。

const 和 volatile 都是 ANSI 的标准关键字，所有的 C/C++ 编译器都应该有支持。

Referential Integrity

看过如何处理某些在汇编语言层面所产生的问题之后，让我们移往另一个方向，看看因为 “logic data integrity” 对上 “physical data integrity” 而产生的问题。

SQL Transactions

在数据库的世界中，有所谓 transaction（事务）的观念，意思是“一组改变”，并且是一组“所有改变都发生之后，才具意义”的改变。例如一个销售数据库通常都会有个发票栏，以及用以登记采购项目的数个栏位。显然，这些栏位如果单独存在，没有意义可言。

一个软件开发者必须有某种方式，能够让人一次把发票栏和其他采购栏全部填好，否则这个数据库就有可能在某一时刻出现数据不一致的事情：客户如果在要命的时刻恰好进入数据库读该行数据，那么要不是采购栏漏失掉，就可能是发票栏漏失掉。数据库专用术语称此为 “referential integrity”。

问题出在，数据库其实只不过是个电脑程序，它们只能够一步一步进行。你必须填上发票栏位，然后再一笔一笔填上采购项目。如果客户的信用卡号码不清楚，你还得回过头去将采购项目一笔一笔删除。

SQL（Structured Query Language）数据库解决这个问题的做法便是，引入 transaction 的观念。当你开始一笔 transaction 后，数据库便将你的改变一一储存起来，直到 transaction 完成为止。然后，所有的改变再一次性地放进数据库中。当 transaction 正在进行时，数据库引擎确定不会有任何客户拿到 transaction 的一部分。

多线程程序也有同样的问题。我们曾经在第 4 章的 SwapLists() 中有过惊鸿一瞥。其中，单一改变却伴随着两个不同的数据结构。改变第一个数据结构而尚未改变第二个数据结构时，系统存在着短暂的“数据不一致”状态。

最简单的例子发生在为一个数组附加数据的操作上。为了在数组尾端加上一笔数据（假设数组够大），通常你必须把数据写入数组元素空间之中，然后改变计数器。如果你不能够让这两个操作不为外力插入，你就必须负担“数据不一致”的风险。

译注 其实在数据库术语中，consistency 和 transaction 和 integrity 有更细微的意义与区分：

某人有两个户头。当他从户头 A 转 100 元到户头 B 时，户头 A 必须减 100 元，户头 B 必须加 100 元。如果两个操作都完成，才称为一项事务（transaction）完成。只要有一个操作未完成，该项事务就作废，而所有已完成的操作都必须还原（roll back）。

更精细地说，事务（transaction）通常是指那种必须在“短时间”内完成的一组操作，因为有客户急着等待结果出来。相对于“短时间”的工作，则有长时间的批次（batch）作业。一个批次作业里可能蕴含许多许多笔事务。

Data consistency（数据一致性）一词通常是代表 transaction 的完成与否。如果 transaction 未能顺利完成，我们就说 consistency 出了问题，所有相关操作必须还原（roll back）。

至于 referential integrity（参照完整性），是关系数据库中的一个非常重要的观念。假设我的数据库中有两张表，一是部门表，内有三笔数据，代表三个部门 101、102、103。另一是职员表，其中有一栏为部门代号。任何一个职员的部门代号必须是 101、102、103 三者之一，这样的关系称为 referential integrity（又称为 referential constrain）。如果职员数据中出现 104 部门代号，就是 referential integrity 出了问题。

数据库管理员可以利用 SQL 做一些定义，使得 referential integrity 得以维护，使得职员资料中有错误部门代号时立刻给予错误信息，并拒绝接受。

以上是我询问我的朋友、IBM DB2 数据库专家庄济诚先生得到的心得。

删除形状的例子

甚至还有其他一些更隐晦的情况存在。考虑一下这种情况：一个绘图软件有一组形状要画。线程 1 负责用户界面，线程 2 负责画出形状。有一个数组，用来放置一些指针，指向一个个的形状物。数组以 mutex 保护起来以避免讹误。

1. 线程 2 调用 GetShapePointer() 取得第 5 个形状。
2. GetShapePointer() 等待数组的 mutex。
3. GetShapePointer() 获得 mutex 的拥有权。
4. GetShapePointer() 取得第 5 个形状的指针。
5. GetShapePointer() 释放 mutex 并传回形状指针。
6. 此时，线程 1 调用了 DeleteShape()，后者需要 mutex。
7. DeleteShape() 获得 mutex 的拥有权。
8. DeleteShape() 释放该形状所使用的内存。
9. 线程 2 的指针于是指向了一块不再适用的内存。
10. 线程 2 当掉。

因为程序的确是当掉了，我们可以放心地说这里面有问题。但，如何解决？

一个明显的答案就是，形状物应该有自己的 mutex。唔，听起来合理，但不切实际。如果一个复杂的工业设计，有 10000 个零组件，难道程序中要产生出 10000 个 mutexes 吗？因此而产生的那些额外费用未免也太高了些。把形状物设计为一群一群，或是有层次结构，怎么样？每一群应该有自己一个 mutex 吗？

为了安全起见，形状物能够被锁定的时刻就是当链表被锁定的时刻，否则就会为 race condition 带来一些机会。从上述的过程来看，GetShapePointer() 应该在传回形状物指针之前先锁住它。而调用端必须知道要把形状物解除锁定。

最后一个问题就是，上述过程也为“死锁”引入了一个巨大的机会。如果不论何时，当你得到一个指针你就把形状物锁定，那么当线程 A 锁定 5 而需要 3 时，线程 B 锁定 3 而需要 5 时，吓，死锁就出现了，程序立刻“挂”掉。

以 Exclusive Locks（排他锁）保护数据

问题可以被解决，只要把你的数据看成是一棵树，而直接从根部锁定。这么一来，从根部到顶端的所有数据都会被锁住。稍早所提的行动过程中，由形状物指针所构成的数组就是数据树的根，形状物或形状物群组，是数据树的枝和叶。如果某个线程在获得此树任一数据的指针时，就锁住整棵树，那么就不可能有其他线程可以做出任何危及此树的操作。

Win16 OpenFile() 支持的所谓共享模式（sharing mode）可用来解释这样的模型。刚才我所说的，大略相等于共享模式中的 OF_SHARE_EXCLUSIVE 模式。意思是所有其他程序对此文件的读写操作一律加以拒绝。此外，如果你调用这个函数时已经有另一个程序正在使用此文件，则你的调用会失败。如果你以 OF_SHARE_EXCLUSIVE 模式打开一个文件，你就能够保证你对此文件所进行的任何操作不会受到其他程序的干扰。

我们可以用相同的理论来看待我们的数据树。如果我们想在线程中改变数据树的某一部分，则同一时刻就不应该允许其他线程读写数据树的任何部分，因为那会导致数据的不安全。另一个线程欲读数据，会造成不安全，因为当时的指针已被更改，以致于数据不一致。另一个线程欲写数据，也会造成不安全，因为两个线程可能会彼此践踏破坏，就好像第 1 章的图 1-4~图 1-7 所描述的链表那样。

请注意，我所谓的“写入”操作，包括删除、改变、修改、重新排列……都算是。我所谓的“读取”操作则包括“对数据结构有所认识”的各种操作都算，包括“获得一个指针指向数据结构的内容”等等。因此，如果 DeleteShape() 欲删除一个形状物，而另一个线程却在当时有一个指针指向任何形状物，都算是不合法的。

在第 4 章，当我们需要改变一个数据结构时，我们在每一个会碰触该结构的函数中锁定该数据结构。例如：

```
void AddLineItems(List *pList)
{
    Node node;
    while ( /* There are more line items */ )
    {
        GetLineItem(&node);
        AddHead(pList, &node);
    }
}

void AddHead(List *pList, Node *pNode)
{
    EnterCriticalSection(&pList->critical_sec);
    pNode->next = pList->head;
    pList->head = pNode;
    LeaveCriticalSection(&pList->critical_sec);
}
```

AddLineItems() 更新链表中的 “line items”，做法是为每一条线调用一次 AddHead()。然而，注意，链表结构只有在 AddHead() 之中才被锁定，这导致另一个线程可以在这一次调用 AddHead() 和在下一次调用 AddHead() 之间堂而皇之地获得属于它自己的锁定。

把锁定搬到更高层次

如果我们把锁定操作搬到上层来，就可以安全地加上所有的项目，不必担心其他线程的干扰。例如：

```
void AddLineItems(List *pList)
{
    Node node;
    EnterCriticalSection(&pList->critical_sec);
    while ( /* There are more line items */ )
    {
        GetLineItem(&node);
        AddHead(pList, &node);
    }
```

```
    LeaveCriticalSection(&pList->critical_sec);
}

void AddHead(List *pList, Node *pNode)
{
    pNode->next = pList->head;
    pList->head = pNode;
}
```

这样的模型特别有助于 Windows 程序，因为其线程是事件驱动（event driven）模式，而且常常返回到主消息循环中。如此架构会提供明显的时间点以锁定数据结构，而结构也可以很轻易地被解除锁定，就像 call stack unwind（译注）那样。

译注 所谓 call stack unwind，是指当异常情况（exception）发生时，堆栈中的所有对象都被清除干净。有些编译器有这样的能力。

这样的模式也带来一些限制。其他数据结构将不能够直接指向这个数据结构，因为没有什么方法可以适当处理“对象已被删除而指针仍然存在”的情况。从前面的例子看来，这不算是什么太麻烦的限制，特别在 C++ 中。因为 C++ 对象间的区隔有助于更良好的定义。

The Readers/Writers Lock

“排他锁定”从纸面上看起来是不错，但是我听到你们许多人在低语抱怨三个字：

效率！　　效率！！　　效率！！！

如果每一个线程都必须排队才能读取一个中心数据结构，那对效率是大伤害。“排他锁定”会强迫所有的存取操作都按照先后次序来，而多线程的利益也就流失掉了。这种情况下你最好还是使用单一线程，因为其他线程将浪费绝大多数的时间在等待上面。

? FAQ 35: 什么是 Readers/Writers lock?

让我们回到先前对 OpenFile() 的分析之中。如果你看过它支持的所谓共享模式（sharing mode），或许你会注意到它还有另外两个模式：OF_SHARE_DENY_READ 和 OF_SHARE_DENY_WRITE。这些模式提供比“排他锁定”更好的控制。事实上程序中的许多个线程在同一时间读取数据并不会有什么问题，只要当时没有其他线程企图改变数据就好。任何线程如果要读取文件内容，应该以 OF_SHARE_DENY_WRITE 开启该文件。

不过，我们也已经决定，如果一个线程正在写入数据，其他线程不论读或写都应该被阻止。所以，任何一个线程如果企图对文件写入数据，它应该以 OF_SHARE_DENY_READ 和 OF_SHARE_DENY_WRITE 开启该文件。那么，直到写入操作完成之前，不会有其他线程可以对此文件进行任何操作。

多线程程序中当然可以完全精确地成就这样的行为，关键在于如何让它们适当地运作。我所描述的解决方案是从一些论文（注）中推导而来的。其伪码列出于下。这个设计是将优先权给予 reader，但是其他做法也可以将优先权给予 writer，或是公平分享。

注：请参考 Courtois, P.J., Heymans, F., and Parnas, D.L.: “Concurrent Control with Readers and Writers,” Communications of the ACM, vol.10, pp.667-668, Oct. 1971.

为 Reader 锁定：

```
Lock( ReaderMutex )
ReadCount = ReaderCount + 1
if (ReaderCount == 0)
   Unlock( DataSemaphore )
Unlock( ReaderMutex )
```

为 Reader 解除锁定：

```
Lock( ReaderMutex )
ReadCount = ReaderCount - 1
if (ReaderCount == 0)
   Unlock( DataSemaphore )
Unlock( ReaderMutex )
```

为 Writer 锁定：

```
Lock( DataSemaphore )
```

为 Writer 解除锁定：

```
UnLock( DataSemaphore )
```

这样的设计看起来简单得有点虚假，其实还有许多细节有待完成。一个常见的问题就是："为什么需要 ReaderCount？为什么不让 semaphore 做掉所有的工作？毕竟 semaphore 就是用来数数儿的不是吗？"答案是，semaphores 一开始从最大值开始，一直下降到 0。只要其值不为 0，"锁定"就还是需要。而此处的问题恰恰相反，我们需要从 0 开始计数，往上累加。

另一个常见的问题是："为什么需要 ReaderMutex？没有它难道就不能够工作了吗？"的确是，因为会发生 race condition。想象一下，如果线程 1 是第一个尝试获得 ReadLock 者，它先增加 ReadCounter 值，此时 ReadCounter 为 1。然后经由一次 context switch，线程 2 获得了执行权。线程 2 也尝试获得 ReadLock，它也增加 ReadCounter 值，于是 ReadCounter 现在变成 2。线程 2 获得执行后又是线程 1 。此时线程 1 检查 ReadCounter，发现它等于 2，于是条件失败，数据库不会再被锁住—即使两个 reader 都认为它们锁住了数据库。

设计一个像这样的 Reader/Writer 算法的解决方案时，最困难的部分在于，你要记住，没有哪一个线程拥有对 Readers/Writers locks 的中央控制权。你必须确定没有任何一个激发的 events 曾经因为 race condition 而遗失掉。在写这一章时我曾经调查过 Internet 上我所发现的数个其他种实现方法，大部分的它们都是错误的，要不是当掉就是进入死锁。问题通常出在作者们忽略了分析"获得"和"释放"之间的潜在性相互影响。如果没有应用正式的方法，执行这样的分析是非常困难的。

我认为唯一能够了解所有这些东西的办法就是：自己写代码．尽管算法很短很简单，却花了我数个小时才实现出来并让它顺利执行。其中的困难包括转化为适当的 Win32 对象，并放置适当的错误处理函数．

下面是我终于完成的一个足以反应 Readers/Writers lock 的数据结构：

```
typedef struct _RWLock
{
   // Handle to a mutex that allows
   // a single reader at a time access
   // to the reader counter.
   HANDLE hMutex;

   // Handle to a semaphore that keeps
   // the data locked for either the
   // readers or the writers.
   HANDLE hDataLock;

   // The count of the number of readers.
   // Can legally be zero or one while
   // a writer has the data locked.
   int  nReaderCount;
} RWLock;
```

很明显，不是吗？大重点就在 hDataLock．当我最初实现它时，我使用一个 mutex，因为我想数据要么被锁住，要么不被锁住，然后我一再出现错误．问题在于取得 lock 的第一个 reader 并不一定必须是最后一个离开的 reader。因此，可能有某个线程锁定 mutex，而另一个线程企图释放 mutex．然而在 Win32 mutexes 中这是不合理的，于是我又把 hDataLock 改为一个 semaphore，因为 semaphore 可以被任意线程锁定，被任意线程释放．这个 semaphore 有最大值 1，初始值 1，意味着它最初未被拥有，而最多仅一个线程可以锁定它．InitRWLock() 列于列表 7-1。

列表 7-1 节录自 READWRIT 的 InitRWLock()

```
#0001  BOOL InitRWLock(RWLock *pLock)
#0002  {
#0003      pLock->nReaderCount = 0;
#0004      pLock->hDataLock = CreateSemaphore(NULL, 1, 1, NULL);
#0005      if (pLock->hDataLock == NULL)
#0006          return FALSE;
#0007      pLock->hMutex = CreateMutex(NULL, FALSE, NULL);
#0008      if (pLock->hMutex == NULL)
#0009      {
#0010          CloseHandle(pLock->hDataLock);
#0011          return FALSE;
#0012      }
#0013      return TRUE;
#0014  }
```

我所面对的下一个问题是如何加上错误处理函数。我产生一个名为 MyWaitForSingleObject() 的函数，看起来像这样：

```
BOOL MyWaitForSingleObject(HANDLE hObject)
{
    int result;

    result = WaitForSingleObject(hObject, MAXIMUM_TIMEOUT);
    // Comment this out if you want this to be non-fatal
    if (result != WAIT_OBJECT_0)
        FatalError("MyWaitForSingleObject - "
            "Wait failed, you probably forgot to call release!");
    return (result == WAIT_OBJECT_0);
}
```

这个函数总是检查返回值，并且在出现错误时发出错误信息。本例中的 MAXIMUM_TIMEOUT 设计为两秒。在任何实际的应用软件中，如果一个结构持续被锁住两秒，那么几乎可以说需要对它做点分析，以确定是否有瓶颈存在。

你看到了，在我“因为没有成功锁定”而发出一个严重错误信息之后，我又传回一个真假值，表示是否已经成功锁定。这是一个容易引起误解的地方。在一般程序中我应该使用 MTVERIFY、MTASSERT，或 C 或 MFC 中对等的东西来检验错误。然而上述那些错误检验只在程序的 debug 模式中才有效，一旦改为 release，模式就全消失不见了。正如你在本章一开始所见，release 版本会把编译器的最优化选项打开，因而它们在多线程程序中失败的机会比在单线程程序中更高。因此，我必须使用一个在 release 模式中仍然有效的错误检验方法。当程序改用 debug 模式时，FatalError() 可以改变为只是传回 FALSE，不再“列出错误信息并结束程序”。

我还放了其他各式各样的错误检查，它们被每一个“企图改变受保护之数据”的函数调用之。它们被列出于列表 7-2。

列表 7-2 节录自 READWRIT 的 ReadOK() 和 WriteOK()

```
#0001  BOOL ReadOK(RWLock *pLock)
#0002  {
#0003      // This check is not perfect, because we
#0004      // do not know for sure if we are one of
#0005      // the readers.
#0006      return (pLock->nReaderCount > 0);
#0007  }
#0008
#0009  BOOL WriteOK(RWLock *pLock)
#0010  {
#0011      DWORD result;
#0012
#0013      // The first reader may be waiting in the mutex,
#0014      // but any more than that is an error.
#0015      if (pLock->nReaderCount > 1)
#0016          return FALSE;
#0017
#0018      // This check is not perfect, because we
#0019      // do not know for sure if this thread was
#0020      // the one that had the semaphore locked.
#0021      result = WaitForSingleObject(pLock->hDataLock, 0);
#0022      if (result == WAIT_TIMEOUT)
#0023          return TRUE;
```

```
#0024
#0025      // a count is kept, which was incremented in Wait.
#0026      result = ReleaseSemaphore(pLock->hDataLock, 1, NULL);
#0027      if (result == FALSE)
#0028         FatalError("WriteOK - ReleaseSemaphore failed");
#0029      return FALSE;
#0030   }
```

如果你识破其中的注解，你可能会看出我遇到的一些问题。例如，你会想，如果 read-count 不是 0，那么你将不被允许进行写入操作。错！一个 reader 处于 mutex 之内，并等待一个已经被 writer 锁住的 hDataLock semaphore，是有可能的。因此，合法性检验只有在“一个 reader 以上”时才会失败。

另一个问题是如何决定目前的 semaphore 值，以便知道它是否被锁定。但是没有什么办法可以读取一个 semaphore 的现值！如果你尝试调用 ReleaseSemaphore() 并给予第二个参数 lReleaseCount 为 0，企图从第三个参数 lpPreviousCount 获得 semaphore 的现值，那么这一调用操作会失败。这里所使用的解决方案是，首先确定“因 semaphore 而失败的锁定”应该已经被锁定了。如果锁定成功，就不能够写入；锁定必须被解除，才能够恢复其值。

理想上这两个函数应该有一个精确的列表，列出哪一个线程锁住了数据树，打算读或写。否则，当一个线程进行锁定操作，准备开始写数据时，另一个线程可能因为程序代码的臭虫，于是也尝试写数据，并且自以为是可行的。这份列表或许可以以目前的线程 ID 来维持。请注意，使用 GetCurrentThread() 所获得的线程 handle 并不提供一个有用的、独一无二 ID，因为 handle 所代表的只是一个特殊数值，可用以表示现行（current）线程，如此而已。

获得（Acquiring）和让予（Releasing）Locks

负责“获得和让予”locks 的函数，源代码列于列表 7-3。再一次请你注意，错误被小心地检查并传回。这些函数和你先前所看到的伪码基本上是一样

的东西。就如我在第 13 章将说明的，我想，“了解”的最大的障碍在于术语。调用 MyWaitForSingleObject() 会做出锁定操作，调用 ReleaseMutex() 或 ReleaseSemaphore() 则会做出解除锁定的操作。

列表 7-3 节录自 READWRIT，索求和释放“各个锁定”

```
#0001  BOOL AcquireReadLock(RWLock *pLock)
#0002  {
#0003      BOOL result = TRUE;
#0004
#0005      if (!MyWaitForSingleObject(pLock->hMutex))
#0006          return FALSE;
#0007
#0008      if(++pLock->nReaderCount == 1)
#0009          result = MyWaitForSingleObject(pLock->hDataLock);
#0010
#0011      ReleaseMutex(pLock->hMutex);
#0012      return result;
#0013  }
#0014
#0015  BOOL ReleaseReadLock(RWLock *pLock)
#0016  {
#0017      int result;
#0018      LONG lPrevCount;
#0019
#0020      if (!MyWaitForSingleObject(pLock->hMutex))
#0021          return FALSE;
#0022
#0023      if (--pLock->nReaderCount == 0)
#0024          result = ReleaseSemaphore(pLock->hDataLock, 1, &lPrevCount);
#0025
#0026      ReleaseMutex(pLock->hMutex);
#0027      return result;
#0028  }
#0029
#0030  BOOL AcquireWriteLock(RWLock *pLock)
#0031  {
#0032      return MyWaitForSingleObject(pLock->hDataLock);
#0033  }
#0034
#0035  BOOL ReleaseWriteLock(RWLock *pLock)
```

```
#0036  {
#0037      int result;
#0038      LONG lPrevCount;
#0039
#0040      result = ReleaseSemaphore(pLock->hDataLock, 1, &lPrevCount);
#0041      if (lPrevCount != 0)
#0042          FatalError("ReleaseWriteLock - Semaphore was not locked!");
#0043      return result;
#0044  }
```

执行范例程序

Readers/Writers 函数被内含并展示于范例程序 READWRIT 之中。READWRIT.C 和 READWRIT.H 源代码内含上述函数的完整源代码。LIST.C 内含一个最简单的、并使用 Readers/Writers lock 的链表（listed list）。它同时也是个测试程序，以四个线程同时载入、读取、拆除链表。

如果你观察像 DeleteHead() 这样的一个函数，你会发现它不再尝试锁定链表。其父函数被预期应该那么做，以符合逻辑上的一致性。DeleteHead() 调用 WriteOK() 以验证调用者已经做了正确的事情。例如：

```
BOOL DeleteHead(List *pList)
{
    Node *pNode;

    if (!WriteOK(&pList->lock))
        return FatalError("DeleteHead - not allowed to write!");

    if (pList->pHead == NULL)
        return FALSE;

    pNode = pList->pHead->pNext;
    GlobalFree(pList->pHead);
    pList->pHead = pNode;
    return TRUE;
}
```

在测试函数中，四个线程之一调用此函数，并以三个 DeleteHead 操作删除链表中的数据项。

```
DWORD WINAPI DeleteThreadFunc(LPVOID n)
{
    int i;

    for (i=0; i<100; i++)
    {
        Sleep(1);
        AcquireWriteLock(&gpList->lock);
        DeleteHead(gpList);
        DeleteHead(gpList);
        DeleteHead(gpList);
        ReleaseWriteLock(&gpList->lock);
    }

    return 0;
}
```

执行这个程序，会打印出两个搜寻线程的结果，以及许多可能的错误。或许，在获得和让予（acquire and release）的函数中设定中断点，并且观察当时发生什么事情，才是比较更富教育性的。

我需要锁定吗？

很多人不知道如何决定数据是否需要保护。使用同步机制会使程序效率降低，而且它们也不容易使用，但在某些情况下又非用不可。下面是一些指导方针：

- 如果你不确定，那么你或许需要一个锁定。
- 如果你在一个以上的线程中使用同一块数据，那么你必须保护它。我所谓的“使用”，一般而言包括读取、与之做比较、写入、更新、改变，或任何其他操作，只要会用到变量名称的都算。

- 如果一个基础的类型，32 位或更小（例如一个 DWORD 或 int），并且是单独数据（换句话说没有所谓逻辑一致性的问题），那么你可以读它或以值和它做比较。如果你需要改变它，请使用 Interlocked...() 函数。你应该把该变量声明为 volatile。
- 如果你有许多数据，请考虑使用一个已经考虑了多用户和多线程的数据库管理程序。这样的系统被设计用来处理复杂的锁定事态，包括管理逻辑一致性和所谓的 referential integrity（译注：该名词出现在本章稍早，我有一个颇大篇幅的译注用来解释它）。

另一个机制也可能有用，那就是限制数据只能够在某个线程中被处理，然后把该线程当做是一种迷你服务器——对于其他想要读写那份数据的线程而言。这样的机制可以去除一些 lock 的使用需求，因为要存取该数据一定得通过该特定的服务器线程。这个观念有点类似于 C++ 中的 private 成员变量。

Lock Granularity（锁定粒度）

? FAQ 36：一次应该锁住多少数据？

我曾经提过的，锁住 10000 个 CAD 形状，是太过极端的一个例子。它只是举例说明了所谓的 “fine granularity locking”。而另一个例子，也就是以 Readers/Writers lock 锁住一整棵数据树，则呈现一种所谓的“coarse granularity locking”。两种情况都十分极端。下面是一份整理：

Coarse Granularity	Fine Granularity
• 使用简单	• 很容易产生死锁
• 死锁的风险极低	• 效率瓶颈的情况尚可令人放心
• 容易产生效率瓶颈	• “锁定”所花的时间，可能招致高度浪费

很显然，一个程序不必一定得完全使用 fine locking 或 coarse locking，它可以视需要组合使用。我给你的唯一警告就是，fine granularity locking 可能会成为调试的午夜恶梦，因为错综复杂的依存关系

（dependencies）可能会出现．如果你决定使用 fine granularity locking，请把死锁的解决方案放进你的程序代码中，以便协助你找出问题的所在．你可以在许多操作系统教科书中找到这样的代码．如果你需要一个起点，不妨参考 Andrew S. Tanenbaum 所著的 “Operation Systems, Design and Implementation” 一书（Prentice-Hall 出版，1987），其中有 Banker 算法．

我建议你的程序一开始尽量少用 locks．然后，当你开始发现一些瓶颈时，再开始适量地使用 locks．你必须慢慢地、一步一步地进行，并且有完整的测试．这十分重要，否则死锁一旦出现，就不容易找出问题，因为你一次做了太多改变，不知哪一个是引发问题的关键．最坏的情况下，问题可能是因好几个改变的组合而引起，不是因为某个单独的改变而引起．

提要

你已经看到了如何使用 volatile 关键字来教导编译器不要对变量做出临时的拷贝．你也已经看到了有关于 Readers/Writers lock 的实现开发和应用的详细讨论．现在的你对于多线程所带来的复杂主题已经有了比较好的鉴赏能力．如果可能，我还是继续建议你避免在线程之间共享数据．

使用 C Run-time Library

这一章告诉你为什么应该以 _beginthreadex() 取代 CreateThread()，并讨论你不需要这么做的一些极少数情况。本章也告诉你如何避免使用 C runtime stdio 函数，改用 Win32 Console API。

重要！

如果你使用 MFC 来开发程序，则本章大部分内容可以被第 9 章取代。注意，不要在一个 MFC 程序中使用_beginthreadex() 或 CreateThread()。

在微软的 Programming Techniques 说明文件中有一句看似悲惨的警告：

警告：如果你在一个与 LIBCMT.LIB 链接的程序中调用 C runtime 函数，你的线程就必须以 _beginthread() 启动之。不要使用 Win32 的 ExitThread() 和 CreateThread()。

这个警告带出了数个问题。第一个问题是 _beginthread() 有一个 race condition，以致于不可完全信赖地使用它。第二个问题是，上述说明文字中并没有提到为什么。

我将以 _beginthreadex()（而不是 _beginthread()）的讨论作为本章序幕。_beginthreadex() 可以取代 _beginthread() 并解决后者所带来的一些问题。我将在本章最后面讨论 _beginthread()。

? FAQ 37：我应该使用多线程版本的 C run-time library 吗？

重要！

如果你写一个多线程程序，而且没有使用 MFC，那么你应该总是和多线程版本的 C Run-time Library 链接，并且总是以 _beginthreadex() 和 _endthreadex() 取代 CreateThread() 和 ExitThread()。_beginthreadex() 的参数和 CreateThread() 一样，并且承担适度的 C runtime library 初始化工作。

虽然 _beginthread() 有一些严重的问题，使微软上述的警告失色，不过这些问题都已经在 _beginthreadex() 中解决了。只要你以 _beginthreadex() 取代 CreateThread()，你就可以在任何线程中安全地调用任何 C runtime 函数。

我们马上来看看 C Run-time Library 多线程版本到底是什么，为什么你必须使用它，以及如何使用 _beginthreadex() 和 _endthreadex()。我们也会看到，在某些情况下，你可以写一个多线程程序，却使用单线程版本的 C runtime library。

什么是 C Runtime Library 多线程版本

原先的 C runtime library 有一些严重的问题，使它无法被多线程程序使用。当 C runtime library 于 20 世纪 70 年代产生出来时，内存容量还很小，多任务是个新奇观念，更别提什么多线程了。

C runtime library 使用数个全局变量和静态变量，这可能在多线程程序中彼此引起冲突。最为大众所知的就是 errno，当 runtime library 发生错误时（特别是文件相关函数），其值会被设定。请你想一想，如果两个线程都以 FILE*函数进行文件 I/O 操作，而两者都设定 errno，会发生什么事呢？

很明显这是一个 race condition，其中一个线程会得到错误的结果。

另一个例子是字符串函数 strtok()，它会维护一个指针，指向目前正被处理的字符串。C runtime library 之中“针对每一线程都有一份”的数据结构，将在第 15 章介绍。

C runtime library 中还有一些数据结构也必须被修改为“thread-safe”（在多线程情况下保证安全）。例如 fopen() 传回的一个 FILE*，基本上是一个指针指向 runtime library 中的一个描述表格（descriptor table）。这个描述表格必须以同步机制保护之，以避免冲突。另外，runtime library 也有可能进行它自己的内存配置操作，这时候内存处理函数也必须被保护。

使用先前数章所描述的同步机制，就有可能产生一个 runtime library，支持多线程。问题是加上那样的支持之后，在大小以及效率两方面都可能因为同步机制的执行而遭受不良波及——甚至即使你只启动了一个线程。

Visual C++ 的折衷方案是提供两个版本的 C runtime library。一个版本给单线程程序使用，一个版本给多线程程序使用。多线程版有两个很大的差别，第一，如 errno 之流的变量，现在变成每个线程各拥有一个。第二，多线程版中的数据结构以同步机制加以保护。

重要！ MFC 程序必须使用多线程版的 C runtime library，否则你就会在链接时获得“undefined function”的错误信息。

选择一个多线程版本的 C Runtime Library

? FAQ 38：我如何选择一套适当的 C run-time library？

C runtime library 有数个不同的版本，如果你选择错误，程序就不能够顺利链接。由于很多人不知道如何改变所使用的版本，所以下两节我会告诉你如何在 Visual C++的集成环境（IDE）中选择一个 C runtime library，以及如何在命令行（command line）模式下选择一个 C runtime library。

Visual C++ 4.x IDE

在 Visual C++ 4.x 中，只要遵循下列步骤，你就可以选择某个 runtime library 版本。

1. 选按【Build/Setting】。
2. 选按【C/C++】选项页。
3. 在【Category】列表中选择“Code Generation”。
4. 拉下【Use run-time library】组合框。

于是出现三种版本供你选择：

- Single-Threaded（static）
- Multithreaded（static）
- Multithreaded DLL

每种版本亦各有一个对应的调试版。

当你在【C/C++】选项页中做了上述更改，【Link】选项页便会自动选择适当的函数库链接，如图 8-1 所示。

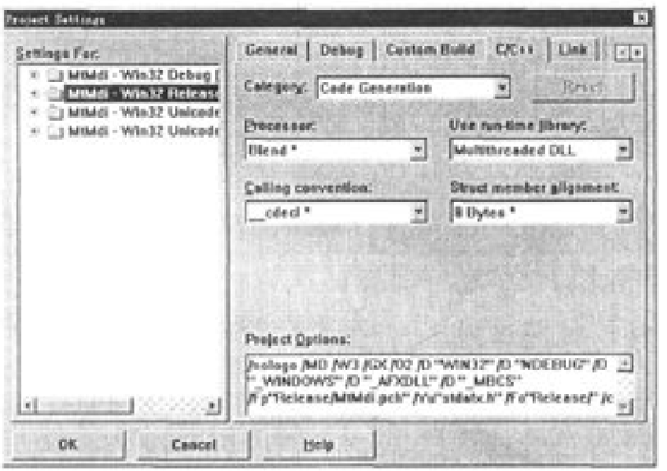


图 8-1 Visual C++ 4.x 环境设定

当你产生一个新的程序项目时，默认的 runtime library 是单线程版。而如果你使用 AppWizard 和 MFC，默认的 runtime library 是多线程版。针对 MFC 程序，runtime library 的动态链接版系在调试时使用，静态链接版则应该在出货时使用。

重要！ 如果你在链接过程中收到错误信息“_beginthreadex is undefined”，意思是你误用了单线程版的 runtime library。你必须改用多线程版。

命令行模式

如果你以命令行方式或是从一个 external makefile 中执行 Visual C++ 编译器，你可以根据下列选项决定使用哪一版的 C runtime library：

```
/ML     Single-Threaded
/MT     Multithreaded (static)
/MD     Multithreaded DLL
/MLd    Debug Single-Threaded
/MTd    Debug Multithreaded (static)
/MDd    Debug Multithreaded DLL
```

如果你要使用调试版、多线程版、动态链接版，你可以这样做：

```
cl /MDd srcfile.c
```

如果你要使用单线程版、静态链接版，你可以这样做：

```
cl /ML srcfile.c
```

上述情况也是默认情况，所以你也可以直接这样做：

```
cl srcfile.c
```

以 C Runtime Library 启动线程

? FAQ 39: 我如何使用 _beginthreadex() 和 _endthreadex()?

为了保证多线程情况下的安全，runtime library 必须为每一个由它所启动和结束的线程做一些簿记工夫。没有这些簿记工作，runtime library 就不知道要为每一个线程配置一块新的内存，作为线程的局部变量用。因此，CreateThread() 有一个名为 _beginthreadex() 的外包函数，负责额外的簿记工作。

重要!

_beginthreadex() 的参数和 CreateThread() 的参数其实完全相同，不过它已经把 Win32 数据类型“净化”过了（译注：意思是使用标准的 C 数据类型，而不再使用 Windows 自定义的数据类型）。这并不好，因为这会妨碍编译器对类型的检验工作。

参数类型被净化，目的是要使这个函数能够移植到其他操作系统。理论上净化了 Win32 数据类型之后，_beginthreadex() 就可以实现于其他平台，因为完全不需要 windows.h，不幸的是由于你还是需要调用 CloseHandle()，所以你还是需要含入 windows.h。整个情况似乎是，微软空有一个好主意，但是未能落实它。

以下对于 _beginthreadex() 的描述之中，面对每一个参数，我都涵盖了对应的 Win32 类型解释。经验告诉我，不够直接的说明，对众人而言，迷惑多于帮助。如果你把这个函数视为 CreateThread() 的一个看起来比较有趣的版本，那就容易了解得多。

```
unsigned long _beginthreadex(
    void *security,
    unsigned stack_size,
    unsigned (__stdcall *start_address)(void *),
    void *arglist,
```

```
    unsigned initflag,
    unsigned* thrdaddr
);
```

参数

security	相当于 CreateThread() 中的 Security 参数。NULL 表示使用默认的安全属性。Windows 95 会忽略此参数。对应的 Win32 数据类型是 LPSECURITY_ATTRIBUTES。
stack_size	新线程的堆栈大小，单位是字节（byte）。对应的 Win32 数据类型是 DWORD。
start_address	线程启动时所执行的函数。对应的 Win32 数据类型是 LPTHREAD_START_ROUTINE。
arglist	新线程将收到的一个指针。这个指针只是单纯地被传递过去，runtime library 并没有对它做拷贝操作。对应的 Win32 数据类型是 LPVOID。
initflag	启动时的状态标记。对应的 Win32 数据类型是 DWORD。
thrdaddr	新线程的 ID 将藉此参数传回。对应的 Win32 数据类型是 LPDWORD。

返回值

_beginthreadex() 传回线程的 handle。此值必须被强制转换类型为 Win32 的 HANDLE 后才能使用。如果函数失败，传回 0，而其原因将被设定在 errno 和 doserrno 全局变量中。

请注意函数名称中的下划线符号。它必须存在，因为这不是个标准的 ANSI C runtime library 函数。你不会在 Unix 或 OS/2 的编译器中找到这个函数。

虽然从 _beginthreadex() 的声明中不能明显看出，然而事实上它所传回的 unsigned long 是一个 Win32 HANDLE，指向新的线程，亦或者传回的是 0——如果函数失败的话。换句话说，传回值和 CreateThread() 相同，但 _beginthreadex() 另外还设立了 errno 和 doserrno。

下面是一个最简单的使用范例：

```
#0001  #include <windows.h>
#0002  #include <process.h>
#0003  unsigned __stdcall myfunc(void* p);
#0004
#0005  void main()
#0006  {
#0007      unsigned long thd;
#0008      unsigned tid;
#0009
#0010      thd = _beginthreadex(NULL,
#0011                           0,
#0012                           myfunc,
#0013                           0,
#0014                           0,
#0015                           &tid );
#0016      if (thd != NULL)
#0017      {
#0018          CloseHandle(thd);
#0019      }
#0020  }
#0021
#0022  unsigned __stdcall myfunc(void* p)
#0023  {
#0024      // ...
#0025  }
```

重要！ 由于 _beginthreadex() 调用 CreateThread()，所以你必须对着 _beginthreadex()的传回值调用 CloseHandle()。请参考第 3 章对于核心对象的讨论。

另外，还有一个 C runtime 函数，是与 ExitThread() 相对应的，名为 _endthreadex()，

规格如下：

```
void _endthreadex( unsigned );
```

和 ExitThread() 一样，_endthreadex() 可以被线程在任意时间调用，它需要一个表示“线程返回代码”的参数。事实上，当线程的 startup 函数返回时，_endthreadex() 会自动被 runtime library 调用。

重要！

> 绝对不要在一个“以 _beginthreadex() 启动的线程”中调用 ExitThread()，因为这么一来，C runtime library 就没有机会释放“为该线程而配置的资源”了。

哪一个好：CreateThread() 抑或 _beginthreadex()？

到目前为止，我想你已经知道了，要以 C runtime library 写一个多线程程序，你必须使用：

1. 多线程版本的 C runtime library。
2. _beginthreadex()/_endthreadex()

? FAQ 40：什么时候我应该使用 _beginthreadex() 而非 CreateThread()？

因此，一个程序如果使用多个线程，而不在任何 worker 线程中调用 runtime library，应该能够与单线程版的 runtime library 链接并以 CreateThread() 取代 _beginthreadex()。然而，“C 程序不调用任何 runtime library 函数”，几乎是不可能的。事实上，编译器也常常会代为调用一些位于 runtime library 中的辅助函数，而你当然没有能力阻止这件

事情发生。所以，你能够跳避多线程版的 C runtime library 并使用 CreateThread() 吗？

在讨论这件事情之前，我必须先说，你得很严肃地考虑是否要做这样的事情。C runtime library 提供了一些很重要的优点，而且你也很难避免你的项目中的其他人使用 runtime 函数。由于“startup code”和一些辅助函数都存在于 C runtime library 之中，因而我们不可能在链接器选项中把它们去除。

假设我有非常严密的同步条件需要遵守，而使用 C runtime library 却有可能因为其内部锁定机制，导致死锁（deadlock）问题发生。这大概是我唯一可能考虑不使用 C runtime library 的情况。

解决方案是，C runtime library 中有一组子集合，可以在这种情况下被安全地调用。这些函数事实上也有对等的 Win32 函数可以替代。不过也有些很棒的东西并没有对等的 Win32 函数，如 stream I/O 和 printf() 等等。

下面是一些一般性的规则。如果主线程以外的任何线程进行以下操作，你就应该使用多线程版的 C runtime library，并使用 _beginthreadex() 和 _endthreadex()：

- 在 C 程序中使用 malloc() 和 free()，或是在 C++ 程序中使用 new 和 delete。

- 调用 stdio.h 或 io.h 中声明的任何函数，包括像 fopen()、open()、getchar()、write()、printf() 等等。所有这些函数都用到共享的数据结构以及 errno。你可以使用 wsprintf() 将字符串格式化，如此就不需要 stdio.h 了。如果链接器抱怨说它找不到 wsprintf()，你得链接 USER32.LIB。

- 使用浮点变量或浮点运算函数。

- 调用任何一个使用了静态缓冲区的 runtime 函数，如 asctime()、strtok() 或 rand()。

也就是说，如果 worker 线程没有使用上述那些函数，那么单线程版的 runtime library 以及 CreateProcess() 都是安全的。

我准备了一些例子。在第一个例子（列表 8-1）中，worker 线程搜寻文件中的一个字符串。它所需要的内存缓冲区系在主线程中以 calloc() 配置，并在 worker 线程中以 free() 清除。worker 线程使用了 fopen() 以及其他一些文件处理函数。因此，这个例子必须使用多线程版 runtime library，以及 _beginthreadex()。

列表 8-1 SRCHCRT.C——使用 _beginthreadex()

```
#0001  /*
#0002   * SrchCrt.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 8, Listing 8-1
#0006   *
#0007   * Uses multiple threads to search the files
#0008   * "*.c" in the current directory for the string
#0009   * given on the command line.
#0010   *
#0011   * This example uses the multithreaded version of
#0012   * the C run-time library so as to be able to use
#0013   * the FILE functions as well as calloc and free.
#0014   *
#0015   * Build this file with the command line: cl /MD SrchCrt.c
#0016   *
#0017   */
#0018
#0019  #define WIN32_LEAN_AND_MEAN
#0020  #include <stdio.h>
#0021  #include <stdlib.h>
#0022  #include <windows.h>
#0023  #include <process.h>    /* _beginthreadex, _endthreadex */
#0024  #include <stddef.h>
#0025  #include "MtVerify.h"
#0026
#0027  DWORD WINAPI SearchProc( void *arg );
#0028
```

```
#0029  #define MAX_THREADS 3
#0030
#0031  HANDLE hThreadLimitSemaphore;
#0032  char szSearchFor[1024];
#0033
#0034  int main(int argc, char *argv[])
#0035  {
#0036     WIN32_FIND_DATA *lpFindData;
#0037     HANDLE hFindFile;
#0038     HANDLE hThread;
#0039     DWORD dummy;
#0040     int i;
#0041
#0042     if (argc != 2)
#0043     {
#0044        printf("Usage: %s <search-string>\n", argv[0]);
#0045        return EXIT_FAILURE;
#0046     }
#0047
#0048     /* Put search string where everyone can see it */
#0049     strcpy(szSearchFor, argv[1]);
#0050
#0051     /* Each thread will be given its own results buffer */
#0052     lpFindData = calloc( 1, sizeof(WIN32_FIND_DATA) );
#0053
#0054     /* Semaphore prevents too many threads from running */
#0055     MTVERIFY( hThreadLimitSemaphore = CreateSemaphore(
#0056           NULL,          /* Security */
#0057           MAX_THREADS,    /* Make all of them available */
#0058           MAX_THREADS,    /* No more than MAX_THREADS */
#0059           NULL )         /* Unnamed */
#0060        );
#0061
#0062     hFindFile = FindFirstFile( "*.c", lpFindData );
#0063
#0064     if (hFindFile == INVALID_HANDLE_VALUE)
#0065        return EXIT_FAILURE;
#0066
#0067     do {
#0068        WaitForSingleObject( hThreadLimitSemaphore,
#0069           INFINITE );
```

```
#0070
#0071         MTVERIFY(
#0072             hThread = (HANDLE)_beginthreadex(NULL,
#0073                       0,
#0074                       SearchProc,
#0075                       lpFindData,
#0076                       0,
#0077                       &dummy
#0078             )
#0079         );
#0080         MTVERIFY( CloseHandle( hThread ) );
#0081
#0082         lpFindData = calloc( 1, sizeof(WIN32_FIND_DATA) );
#0083
#0084     } while ( FindNextFile( hFindFile, lpFindData ));
#0085
#0086     FindClose( hFindFile );
#0087
#0088     for (i=0; i<MAX_THREADS; i++)
#0089         WaitForSingleObject(
#0090                   hThreadLimitSemaphore,
#0091                   INFINITE );
#0092     MTVERIFY( CloseHandle( hThreadLimitSemaphore ) );
#0093
#0094     return EXIT_SUCCESS;
#0095 }
#0096
#0097
#0098 DWORD __stdcall SearchProc( void *arg )
#0099 {
#0100     WIN32_FIND_DATA *lpFindData = (WIN32_FIND_DATA *)arg;
#0101     char buf[1024];
#0102     FILE* fp;
#0103
#0104     fp = fopen(lpFindData->cFileName, "r");
#0105     if (!fp)
#0106         return EXIT_FAILURE;
#0107
#0108     while (fgets(buf, sizeof(buf), fp))
#0109     {
#0110         /* Inefficient search strategy, but it's easy */
```

```
#0111         if (strstr(buf, szSearchFor))
#0112             printf("%s: %s", lpFindData->cFileName, buf);
#0113     }
#0114
#0115     fclose(fp);
#0116     free(lpFindData);
#0117
#0118     MTVERIFY( ReleaseSemaphore( hThreadLimitSemaphore,
#0119               1,          // Add one to the count
#0120               NULL ) );   // Do not need the old value
#0121 }
```

第二个例子显示于列表 8-2，大部分 C runtime 函数都已被移除。这个程序有两个显著的缺点。第一个缺点是我们必须自己进行锁定操作，以确保各个线程的输出不会混杂在一起。如果用的是多线程版 C runtime library，你只要轻轻松松地放置 mutex（互斥器）在任何“牵涉到文件 handle”的操作周围，一样可以阻止不同线程的输出混淆在一起（第 2 章就是如此）。

第二个缺点是，Win32 并未提供有缓冲能力的 stream I/O，所以我必须写一个简单函数，名为 GetLine()，从文件中一个 byte 一个 byte 地读取数据。GetLine() 的做法并非最佳，但如果你没有具缓冲能力的 C I/O 函数可用，这是必要的做法。显然，这一版程序比上一版复杂得多，因为它花了很多精力来开发 C runtime library 原本就有的功能。

列表 8-2 SRCHWIN.C——安全使用 CreateThread()

```
#0001 /*
#0002  * SrchWin.c
#0003  *
#0004  * Sample code for "Multithreading Applications in Win32"
#0005  * This is from Chapter 8, Listing 8-2
#0006  *
#0007  * Uses multiple threads to search the files
#0008  * "*.c" in the current directory for the string
#0009  * given on the command line.
#0010  *
#0011  * This example avoids most C run-time functions
```

```
#0012  * so that it can use the single-threaded
#0013  * C libraries.
#0014  *
#0015  * It is necessary to use a critical section to
#0016  * divvy up output to the screen or the various
#0017  * threads end up with their output intermingled.
#0018  * Normally the multithreaded C run-time does this
#0019  * automatically if you use printf.
#0020  *
#0021  */
#0022
#0023 #include <windows.h>
#0024 #include "MtVerify.h"
#0025
#0026 DWORD WINAPI SearchProc( void *arg );
#0027 BOOL GetLine( HANDLE hFile, LPSTR buf, DWORD size );
#0028
#0029 #define MAX_THREADS 3
#0030
#0031 HANDLE hThreadLimitSemaphore;    /* Counting semaphore */
#0032 HANDLE hConsoleOut;             /* Console output */
#0033 CRITICAL_SECTION ScreenCritical; /* Lock screen updates */
#0034
#0035 char szSearchFor[1024];
#0036
#0037 int main(int argc, char *argv[])
#0038 {
#0039     WIN32_FIND_DATA *lpFindData;
#0040     HANDLE hFindFile;
#0041     HANDLE hThread;
#0042     DWORD dummy;
#0043     int i;
#0044
#0045     hConsoleOut = GetStdHandle( STD_OUTPUT_HANDLE );
#0046
#0047     if (argc != 2)
#0048     {
#0049         char errbuf[512];
#0050         wsprintf(errbuf,
#0051             "Usage: %s <search-string>\n",
#0052             argv[0]);
```

```
#0053        WriteFile( hConsoleOut,
#0054            errbuf,
#0055            strlen(errbuf),
#0056            &dummy,
#0057            FALSE );
#0058        return EXIT_FAILURE;
#0059    }
#0060
#0061    /* Put search string where everyone can see it */
#0062    strcpy(szSearchFor, argv[1]);
#0063
#0064    /* Allocate a find buffer to be handed
#0065     * to the first thread */
#0066    lpFindData = HeapAlloc( GetProcessHeap(),
#0067           HEAP_ZERO_MEMORY,
#0068           sizeof(WIN32_FIND_DATA) );
#0069
#0070    /* Semaphore prevents too many threads from running */
#0071    MTVERIFY( hThreadLimitSemaphore = CreateSemaphore(
#0072           NULL,          /* Security */
#0073           MAX_THREADS,    /* Make all of them available */
#0074           MAX_THREADS,    /* Allow a total of MAX_THREADS */
#0075           NULL )         /* Unnamed */
#0076        );
#0077
#0078    InitializeCriticalSection(&ScreenCritical);
#0079
#0080    hFindFile = FindFirstFile( "*.c", lpFindData );
#0081
#0082    if (hFindFile == INVALID_HANDLE_VALUE)
#0083        return EXIT_FAILURE;
#0084
#0085    do {
#0086        WaitForSingleObject( hThreadLimitSemaphore,
#0087                    INFINITE );
#0088
#0089        MTVERIFY( hThread = CreateThread(NULL,
#0090                    0,
#0091                    SearchProc,
#0092                    lpFindData, // arglist
#0093                    0,
```

```
#0094                       &dummy )
#0095         );
#0096
#0097        MTVERIFY( CloseHandle( hThread ) );
#0098
#0099        lpFindData = HeapAlloc( GetProcessHeap(),
#0100                     HEAP_ZERO_MEMORY,
#0101                     sizeof(WIN32_FIND_DATA) );
#0102
#0103     } while ( FindNextFile( hFindFile, lpFindData ));
#0104
#0105     FindClose( hFindFile );
#0106     hFindFile = INVALID_HANDLE_VALUE;
#0107
#0108     for (i=0; i<MAX_THREADS; i++)
#0109        WaitForSingleObject( hThreadLimitSemaphore,
#0110                     INFINITE );
#0111
#0112     MTVERIFY( CloseHandle( hThreadLimitSemaphore ) );
#0113
#0114     return EXIT_SUCCESS;
#0115  }
#0116
#0117
#0118  DWORD WINAPI SearchProc( void *arg )
#0119  {
#0120     WIN32_FIND_DATA *lpFindData = (WIN32_FIND_DATA *)arg;
#0121     char buf[1024];
#0122     HANDLE hFile;
#0123     DWORD dummy;
#0124
#0125     hFile = CreateFile(lpFindData->cFileName,
#0126                        GENERIC_READ,
#0127                        FILE_SHARE_READ,
#0128                        NULL,
#0129                        OPEN_EXISTING,
#0130                        FILE_FLAG_SEQUENTIAL_SCAN,
#0131                        NULL
#0132                      );
#0133     if (!hFile)
#0134        return 1;   /* Silently ignore problem files */
```

```
#0135
#0136     while (GetLine(hFile, buf, sizeof(buf)))
#0137     {
#0138         /* Inefficient search strategy, but it's easy */
#0139         if (strstr(buf, szSearchFor))
#0140         {
#0141             /* Make sure that this thread is the
#0142              * only one writing to this handle */
#0143             EnterCriticalSection( &ScreenCritical );
#0144
#0145             WriteFile( hConsoleOut,
#0146                 lpFindData->cFileName,
#0147                 strlen(lpFindData->cFileName),
#0148                 &dummy,
#0149                 FALSE );
#0150             WriteFile( hConsoleOut,
#0151                 ": ", 2, &dummy, FALSE );
#0152             WriteFile( hConsoleOut,
#0153                 buf, strlen(buf), &dummy, FALSE );
#0154             WriteFile( hConsoleOut,
#0155                 "\r\n", 2, &dummy, FALSE );
#0156
#0157             LeaveCriticalSection( &ScreenCritical );
#0158         }
#0159     }
#0160
#0161     CloseHandle(hFile);
#0162     HeapFree( GetProcessHeap(), 0, lpFindData);
#0163
#0164     MTVERIFY( ReleaseSemaphore( hThreadLimitSemaphore,
#0165                                 1,
#0166                                 NULL ) );
#0167 }
#0168
#0169 /*
#0170  * Unlike fgets(), this routine throws away CR/LF
#0171  * automatically.  Calling ReadFile() one character
#0172  * at a time is slow, but this illustrates the
#0173  * advantages of using stdio under some conditions
#0174  * (because buffering the stream yourself is difficult)
#0175  */
```

```
#0176  BOOL GetLine(HANDLE hFile, LPSTR buf, DWORD size)
#0177  {
#0178      DWORD total = 0;
#0179      DWORD numread;
#0180      int state = 0;  /* 0 = Looking for non-newline */
#0181                      /* 1 = Stop after first newline */
#0182
#0183      for (;;)
#0184      {
#0185          if (total == size-1)
#0186          {
#0187              buf[size-1] = '\0';
#0188              return TRUE;
#0189          }
#0190          if (!ReadFile(hFile, buf+total, 1, &numread, 0)
#0191                  || numread == 0)
#0192          {
#0193              buf[total] = '\0';
#0194              return total != 0;
#0195          }
#0196          if (buf[total] == '\r' || buf[total] == '\n')
#0197          {
#0198              if (state == 0)
#0199                  continue;
#0200              buf[total] = '\0';
#0201              return TRUE;
#0202          }
#0203          state = 1;
#0204          total++;
#0205      }
#0206  }
```

避免 stdio.h

许多时候，舍弃 C runtime library 并不是那么困难，因为 Win32 提供了许多关于文件处理、内存管理等的函数。令大家担心的，其实是在没有 stdio.h 的情况下如何进行屏幕输出。

为了避免使用 stdio.h，有三个不同的问题必须提出来讨论。第一个问题就是字符串格式化.这可经由 sprintf() 的一个 Windows 兄弟,名为 wsprintf() 来解决。wsprintf() 其实还细分为 _wsprintfA() 和 _wsprintfW()，前者处理 ANSI 字符串，后者处理 Unicode 字符串。这个函数是系统核心的一部分，与 C runtime library 没有关系。它的操作与功能和 C runtime 的 sprintf() 大同小异，差别在于对浮点数的处理。

第二个问题是找到 stdin 和 stdout 的替代品，这是 C runtime library 中预先定义好的 handles，分别代表标准输入与标准输出。这些文件 handles 在默认情况下是被导人到屏幕，但当然它也可以被导到文件去，只要使用者进行了重定向的操作。例如：

```
sort < oldfile > newfile
```

一般而言，Win32 程序使用 CreateFile()、ReadFile()、WriteFile()、CloseHandle()作为其基本 I/O 操作。最后三个函数都需要一个 handle，此 handle 由 CreateFile() 传回。Win32 之中亦有完全对等于 stdin、stdout、stderr 的东西，可以使用 API 函数 GetStdHandle() 获得之：

```
HANDLE GetStdHandle(
    DWORD nStdHandle
);
```

参数

nStdHandle	设定传回之 handle 型态．必须是以下三者之一： STD_INPUT_HANDLE STD_OUTPUT_HANDLE STD_ERROR_HANDLE

你可以把这个函数所传回的 handle，用在 ReadFile() 和 WriteFile() 身上。你可以在列表 8-2 的 SRCHWIN 程序以及稍后即将出现的 BANNER 程序中看到这一函数的使用实例。

最后一个必须解决的问题是，如何直接控制屏幕。在 MS-DOS 之下你可以用下列四种方法完成：

1. 利用 BIOS 显示中断服务函数；
2. 利用对视频显示缓冲区（video buffer）的直接读写；
3. 利用 C runtime conio.h 中的函数；
4. 利用 C runtime stdio.h 中的函数。

虽然第三种方法和第四种方法在 Win32 中都还是可以的，但它们都属于 C runtime library 的一部分，不该在一个未使用多线程版 C runtime library 的多线程程序中调用。

不论是 Windows NT 或 Windows 95，最自然的做法就是使用 Win32 API 的一小支，称为 Console API 的那一部分。Console API 允许你直接读写屏幕，它提供对光标的控制，以及对字符属性、窗口标题、鼠标等等的控制。它甚至允许你把屏幕上的四方形整块搬动。

使用 Console API 并不困难。下一节的 BANNER 程序会示范许多一般性的用法。

UNIX

Unix 程序员将程序移植到 Windows NT 时，常会有的一个疑虑就是：没有 curses——一个终端机管理软件套件。虽然如今的 Windows NT 上已经有了 curses 实现品，但你应该把 Console API 想象为更适当的替代品。和 curses 不同的是，Console API 不需要顾虑终端机型态的多样性，也不必忧虑终端机的慢速连接和慢速更新。

一个安全的多线程程序

? FAQ 41：我如何使用 Console API 取代 stdio.h?

BANNER，列表 8-3 的例子，示范了数个重点。它使用 Win32 Console API 取代一般的 stdio 函数，包括光标的处理以及 gets() 函数。它同时还：

- 主线程以 HeapAlloc() 分配内存，worker 线程以 HeapFree() 释放内存。注意，除非是多线程版本，否则 C runtime library 中的内存管理函数在多线程程序中是不安全的。
- 所有随机变量都在主线程中产生。C runtime library 的 rand() 必须在各调用之间维持其状态。
- 当结束的时刻来临时，利用一个 event 对象同时激发所有线程对象。
- 这个范例程序既不链接多线程版的 C runtime library，也没这个必要。

列表 8-3 BANNER.C——Console I/O 加上 CreateThread()

```
#0001  /*
#0002   * Banner.c
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 8, Listing 8-3
#0006   *
#0007   * Demonstrates how to write a program that can use
#0008   * CreateThread instead of calling _beginthreadex.
#0009   * This program does not need the multithread library.
#0010   *
#0011   * This program could use ReadConsole and WriteConsole.
#0012   * There are minor but significant differences between
#0013   * these functions and ReadFile and WriteFile.
#0014   *
#0015   * This program is ANSI only, it will not compile
#0016   * for Unicode.
#0017   */
#0018
#0019  #define WIN32_LEAN_AND_MEAN
#0020  #include <stdio.h>
#0021  #include <stdlib.h>
```

```
#0022 #include <windows.h>
#0023 #include <time.h>      /* to init rand() */
#0024 #include "MtVerify.h"
#0025
#0026 /*********************************************
#0027  * Constants
#0028  */
#0029 #define MAX_THREADS  256
#0030
#0031 #define INPUT_BUF_SIZE     80
#0032 #define BANNER_SIZE        12
#0033 #define OUTPUT_TEXT_COLOR   BACKGROUND_BLUE | \
#0034         FOREGROUND_RED|FOREGROUND_GREEN|FOREGROUND_BLUE
#0035
#0036 /*********************************************
#0037  * Function Prototypes
#0038  */
#0039 void MainLoop( void  );
#0040 void ClearScreen( void );
#0041 void ShutDownThreads( void );
#0042 void Prompt( LPCSTR str );    /* Display title bar info */
#0043 int StripCr( LPSTR buf );
#0044
#0045 /* Thread startup function */
#0046 DWORD WINAPI BannerProc( LPVOID pParam );
#0047
#0048 /*********************************************
#0049  * Global Variables
#0050  */
#0051 HANDLE  hConsoleIn;       /* Console input  */
#0052 HANDLE  hConsoleOut;      /* Console output */
#0053 HANDLE  hRunObject;       /* "Keep Running" event object */
#0054 HANDLE  ThreadHandles[MAX_THREADS];
#0055 int     nThreads;         /* Number of threads started */
#0056
#0057 CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
#0058
#0059 /*********************************************
#0060  * Stucture passed to thread on startup
#0061  */
#0062 typedef struct {
```

```
#0063     TCHAR buf[INPUT_BUF_SIZE];
#0064     SHORT x;
#0065     SHORT y;
#0066 } DataBlock;
#0067
#0068
#0069 /*********************************************
#0070  * Primary thread enters here
#0071  */
#0072 int main()
#0073 {
#0074     /* Get display screen information & clear the screen.*/
#0075     hConsoleIn = GetStdHandle( STD_INPUT_HANDLE );
#0076     hConsoleOut = GetStdHandle( STD_OUTPUT_HANDLE );
#0077     GetConsoleScreenBufferInfo( hConsoleOut, &csbiInfo );
#0078
#0079     ClearScreen();
#0080
#0081     /* Create the event object that keeps threads running. */
#0082     MTVERIFY( hRunObject = CreateEvent(
#0083           NULL,          /* Security */
#0084           TRUE,          /* Manual event */
#0085           0,            /* Clear on creation */
#0086           NULL)          /* Name of object */
#0087     );
#0088
#0089     /* Start waiting for keyboard input to
#0090      * dispatch threads or exit. */
#0091     MainLoop();
#0092
#0093     ShutDownThreads();
#0094
#0095     ClearScreen();
#0096
#0097     CloseHandle( hRunObject );
#0098     CloseHandle( hConsoleIn );
#0099     CloseHandle( hConsoleOut );
#0100
#0101     return EXIT_SUCCESS;
#0102 }
#0103
```

```
#0104  void ShutDownThreads( void )
#0105  {
#0106      if (nThreads > 0)
#0107      {
#0108          /* Since this is a manual event, all
#0109           * threads will be woken up at once. */
#0110          MTVERIFY( SetEvent(hRunObject) );
#0111          MTVERIFY( WaitForMultipleObjects(
#0112                    nThreads,
#0113                    ThreadHandles,
#0114                    TRUE, INFINITE
#0115                ) != WAIT_FAILED
#0116          );
#0117          while (--nThreads)
#0118             MTVERIFY( CloseHandle(
#0119                    ThreadHandles[nThreads] ) );
#0120      }
#0121  }
#0122
#0123  /* Dispatch and count threads. */
#0124  void MainLoop( void )
#0125  {
#0126      TCHAR buf[INPUT_BUF_SIZE];
#0127      DWORD bytesRead;
#0128      DataBlock *data_block;
#0129      DWORD thread_id;
#0130
#0131      srand(time(NULL));
#0132      for (;;)
#0133      {
#0134          Prompt(
#0135              "Type string to display or ENTER to exit: "
#0136          );
#0137          MTVERIFY( ReadFile( hConsoleIn,
#0138                        buf,
#0139                        INPUT_BUF_SIZE-1,
#0140                        &bytesRead,
#0141                        NULL)
#0142          );
#0143          /* ReadFile is binary, not line oriented,
#0144           * so terminate the string. */
```

```
#0145          buf[bytesRead] = '\0';
#0146          MTVERIFY( FlushConsoleInputBuffer( hConsoleIn ) );
#0147          if (StripCr( buf ) == 0)
#0148             break;
#0149
#0150          if (nThreads < MAX_THREADS)
#0151          {
#0152             /*
#0153              * Use the Win32 HeapAlloc() instead of
#0154              * malloc() because we would need the
#0155              * multithread library if the worker
#0156              * thread had to call free().
#0157              */
#0158             data_block = HeapAlloc(
#0159                          GetProcessHeap(),
#0160                          HEAP_ZERO_MEMORY,
#0161                          sizeof(DataBlock) );
#0162             strcpy(data_block->buf, buf);
#0163
#0164             /*
#0165              * Pick a random place on the screen to put
#0166              * this banner. You may not call rand in the
#0167              * worker thread because it is one of the
#0168              * functions that must maintain state
#0169              * between calls.
#0170              */
#0171             data_block->x = rand()
#0172                * (csbiInfo.dwSize.X - BANNER_SIZE)
#0173                / RAND_MAX;
#0174             data_block->y = rand()
#0175                * (csbiInfo.dwSize.Y - 1)
#0176                / RAND_MAX + 1;
#0177
#0178             MTVERIFY(
#0179                ThreadHandles[nThreads++] = CreateThread(
#0180                      NULL,
#0181                      0,
#0182                      BannerProc,
#0183                      data_block,
#0184                      0,
#0185                      &thread_id )
```

```
#0186            );
#0187        }
#0188    }
#0189 }
#0190
#0191 int StripCr( LPSTR buf )
#0192 {
#0193    int len = strlen(buf);
#0194    for (;;)
#0195    {
#0196       if (len <= 0) return 0;
#0197       else if (buf[--len] == '\r' )
#0198          buf[len] = ' ';
#0199       else if (buf[len] == '\n' )
#0200          buf[len] = ' ';
#0201       else break;
#0202    }
#0203    return len;
#0204 }
#0205
#0206
#0207 void ClearScreen( void )
#0208 {
#0209    DWORD    dummy;
#0210    COORD    Home = { 0, 0 };
#0211    FillConsoleOutputAttribute( hConsoleOut,
#0212           csbiInfo.wAttributes,
#0213           csbiInfo.dwSize.X * csbiInfo.dwSize.Y,
#0214           Home,
#0215           &dummy );
#0216    FillConsoleOutputCharacter( hConsoleOut,
#0217           ' ',
#0218           csbiInfo.dwSize.X * csbiInfo.dwSize.Y,
#0219           Home,
#0220           &dummy );
#0221 }
#0222
#0223
#0224 void Prompt( LPCSTR str )
#0225 {
#0226    COORD   Home = { 0, 0 };
```

```
#0227     DWORD   dummy;
#0228     int len = strlen(str);
#0229
#0230     SetConsoleCursorPosition( hConsoleOut, Home );
#0231     WriteFile( hConsoleOut, str, len, &dummy, FALSE );
#0232     Home.X = len;
#0233     FillConsoleOutputCharacter( hConsoleOut,
#0234           ' ',
#0235           csbiInfo.dwSize.X-len,
#0236           Home,
#0237           &dummy );
#0238  }
#0239
#0240  /*************************************************************
#0241   * Routines from here down are used only by worker threads
#0242   */
#0243
#0244  DWORD WINAPI BannerProc( LPVOID pParam )
#0245  {
#0246     DataBlock *thread_data_block = pParam;
#0247     COORD     TopLeft = {0,0};
#0248     COORD     Size = {BANNER_SIZE ,1};
#0249     int       i, j;
#0250     int       len;
#0251     int       ScrollPosition = 0;
#0252     TCHAR     OutputBuf[INPUT_BUF_SIZE+BANNER_SIZE];
#0253     CHAR_INFO CharBuf[INPUT_BUF_SIZE+BANNER_SIZE];
#0254     SMALL_RECT rect;
#0255
#0256     rect.Left   = thread_data_block->x;
#0257     rect.Right  = rect.Left  + BANNER_SIZE;
#0258     rect.Top    = thread_data_block->y;
#0259     rect.Bottom = rect.Top;
#0260
#0261     /* Set up the string so the output routine
#0262      * does not have figure out wrapping. */
#0263     strcpy(OutputBuf, thread_data_block->buf);
#0264     len = strlen(OutputBuf);
#0265     for (i=len; i<BANNER_SIZE; i++)
#0266        OutputBuf[i] = ' ';
#0267     if (len<BANNER_SIZE) len = BANNER_SIZE;
```

```
#0268     strncpy(OutputBuf+len, OutputBuf, BANNER_SIZE);
#0269     OutputBuf[len+BANNER_SIZE-1] = '\0';
#0270
#0271     MTVERIFY( HeapFree( GetProcessHeap(), 0, pParam ) );
#0272
#0273     do
#0274     {
#0275        for (i=ScrollPosition++, j=0;
#0276              j<BANNER_SIZE;
#0277              i++, j++)
#0278        {
#0279           CharBuf[j].Char.AsciiChar = OutputBuf[i];
#0280           CharBuf[j].Attributes = OUTPUT_TEXT_COLOR;
#0281        }
#0282        if (ScrollPosition == len)
#0283           ScrollPosition = 0;
#0284
#0285        MTVERIFY( WriteConsoleOutput(
#0286                     hConsoleOut,
#0287                     CharBuf,
#0288                     Size,
#0289                     TopLeft,
#0290                     &rect)
#0291        );
#0292
#0293     /*
#0294      * This next statement has the dual purpose of
#0295      * being a choke on how often the banner is updated
#0296      * (because the timeout forces the thread to wait for
#0297      * awhile) as well as causing the thread to exit
#0298      * when the event object is signaled.
#0299      */
#0300     } while ( WaitForSingleObject(
#0301                          hRunObject,
#0302                          125L
#0303               ) == WAIT_TIMEOUT );
#0304
#0305     return 0;
#0306  }
```

结束进程（Process）

为了适当清除 C runtime library 中的结构，对于以 _beginthread() 或 _beginthreadex() 来产生新线程的程序，你应该使用以下两种技术之一以结束程序：

1. 调用 C runtime library 的 exit() 函数。
2. 从 main() 返回系统。

任何一种情况下，runtime library 都会自动进行清理操作（cleanup），最后调用 ExitProcess()。使用任一种技术都不会等待线程的结束。任何正在运行的线程都会被自动终止。如果你必须等待某个线程操作完毕，你必须使用第 4 章所讨论的那些同步机制。

在极端糟糕的情况下，你也可能在任何线程中调用 abort()。这个操作非不得已不要使用，因为其间既没有任何退出程序（exit handler）可以调用，文件缓冲区也没有清理。

为什么你应该避免 _beginthread()

? FAQ 42：为什么我不应该使用 _beginthread()?

对 Microsoft C runtime library 而言，_beginthreadex() 是一个很新的函数。长期以来，多线程程序唯一能够使用的只有 _beginthread()。两者之间有一些意义深长的差异。下面是 _beginthread() 的声明：

```
unsigned long  _beginthread (
    void (__cdecl *start_address) (void *),
    unsigned stack_size,
    void *arglist
);
```

参数

start_address	线程的起始函数。
stack_size,	堆栈大小，以字节（byte）为单位。与 CreateThread() 相同，此值若为 0，表示使用默认大小。
arglist	指向一块数据。新线程将收到此指针。runtime library 不会为该数据另外拷贝一份。

返回值

如果失败，传回 -1。否则传回一个 unsigned long，代表新线程的 handle。这个 handle 也许可用，也许不可用，调用端不见得可以安全地使用该 handle。

译注 读到这里你必然心中有疑惑。怎么 handle 不能用呢？下面第二点就提出解释。

_beginthread() 被认为是一个头脑简单的函数。它存在几个基本问题。第一，_beginthread() 并没有要求获得和 CreateThread() 完全一样的参数，因此有些事情它就办不到，如将线程产生于挂起状态，以便优先权可调整以及数据可被初始化等等。

第二，也是最重要的一点，被 _beginthread() 产生出来的线程所做的第一件事就是关闭自己的 handle。这样做是为了隐藏 Win32 的实现细节。因此，_beginthread() 传回的 handle 可能在当时是不可用（invalid）的。如果你尝试使用由 _beginthread() 传回的 handle，无可避免会导致一个

race condition。没有这个 handle，也就没有办法等待这个线程的结束，改变其参数、或甚至取得其结束代码。

还记得第 2 章吗？被 CreateThread() 调用的“线程起始函数”必须是 WINAPI 型态（实际上是 __stdcall）。而从另一角度看，_beginthread() 获得一个指针指向正常的 C 函数（调用约定为 __cdecl）并传回 void。下面是使用 _beginthread() 并指定一个“线程起始函数”的简单轮廓：

```
#0001  void MyFunc(LPVOID);
#0002
#0003  void main()
#0004  {
#0005      unsigned long htd;
#0006
#0007      htd = _beginthread(MyFunc,
#0008                         0,     // stack size
#0009                         0 );   // argument
#0010
#0011      if (htd == -1)
#0012      {
#0013          // thread creation failed
#0014      }
#0015  }
#0016
#0017  void MyFunc(LPVOID arg)
#0018  {
#0019      // ...
#0020  }
```

另有一个函数对应于 ExitThread()，名为 _endthread()，型态如下：

```
void _endthread(void);
```

你看到了，_endthread() 并没有指定结束代码。因此，对着一个以 _beginthread() 产生出来的线程调用 GetExitCodeThread() 是没有意义的。让任何一个以 _beginthread() 产生出来的线程传回一个结束代码同样地毫无意义。

如果你读过本章一开始列出的那个微软的警告，很明显当你要使用 C runtime library 时，你必须连带使用 _beginthread()。当然，如果你还想象希望能够使用 CreateThread()去改变安全属性或令线程挂起，可能你会一筹莫展。也因此才有后来的 _beginthreadex() 出现。你还是可以在 runtime library 中找到 _beginthread()，但我不鼓励用它。

提要

这一章之中你看到了 _beginthread() 和 _beginthreadex() 的差异。你知道了什么时候必须使用多线程版的 C runtime library，以及如何在某些情况下使用单线程版本。你也看到了如何以 Console API 取代 stdio。最后，你还看到了 _beginthread() 的问题以及为什么应该使用 _beginthreadex()。

使用 C++

这一章描述如何以 C++ 类产生多个线程。本章告诉你为什么 C++ 可以让多线程程序设计明显地比较容易，也比较安全。

到目前为止，我们所有的范例程序都以 C 完成．现在我要开始使用 C++。在多线程程序设计方面， C++ 提供了明显的优点，因为你可以保证对象如何地被处理，如何地同步化。

我假设你已经有基本的 C++ 能力，包括对类（classes）、虚函数（virtual functions）、构造函数（constructors）、析构函数（destructors）的了解．如果你对这些术语还不是很了解，市面上有一大堆的书讲这些东西。

处理有问题的 _beginthreadex() 函数原型

大步前进之前，我必须先指出，在 Visual C++ 4.x 的 C runtime library 之中，_beginthreadex() 的原型声明有个小“臭虫”。这个“臭虫”在 C 程

序中无伤大雅，但在 C++中不行，因为 C++ 有更严格的类型检验。问题起源自 _beginthreadex()：

```
unsigned long _beginthreadex(
                    void *security,
                    unsigned stack_size,
                    unsigned (* start_address)(void*),
                    void *arglist,
                    unsigned initflag,
                    unsigned *thrdaddr );
```

其中第三和第六个参数，start_address 和 thrdaddr，在定义之中都牵涉到 unsigned。更正式地说是 unsigned int。如果你观察 CreateThread 的定义，你会发现这两个参数都被定义为 DWORD，也就是 unsigned long。这两种类型在 32 位编译器中是没有区别的，所以 C 编译器忽略其间的差异。但 Visual C++ 4.x 中的 C++ 编译器有比较严格的类型检验，会注意到这个差异。如果你声明 thrdaddr 为 DWORD 却企图将“地址”交给它，你会获得这样的错误信息：

```
BadClass.cpp(36) : error C2664: '_beginthreadex' : cannot convert
parameter 6 from 'unsigned long *' to 'unsigned int *'(new behavior;
please see help)
```

最后一行所说的“new behavior”解释了为什么这个问题没有出现在 Visual C++ 2.x 中（_beginthreadex() 第一次出现的时候）。

这个问题有两个解决方案。第一是把你的变量声明为 unsigned，也就是 _beginthreadex() 所希望的类型。这个方法最简单，但如果函数原型有一天做了修正，你又得回头把所有的相关变量改回其类型。这样的改变基本上会激起阵阵涟漪，并影响程序代码的其他部分。

第二个解决方法是将变量声明为 CreateThread() 所希望的类型，然后，在丢给 _beginthreadex() 之前，再把它强制转换类型。我将使用这个方法，而强制转换类型的工作则交给一个 typedef 来完成，如此一来，万一函数类型有所修正，我们也可以很快、很方便地应对。

列表 9-1 示范以这种方法来调用 _beginthreadex()。PBEGINTHREADEX_THREADFUNC 类型提供了一种将线程起始函数强制转换类型的方法，而 PBEGINTHREADEX_THREADID 类型则提供了对于线程 ID 的一种强制转换类型方法。我将在整本书中使用这两个类型。

列表 9-1 CPPSKEL.CPP——最阳春的线程启动代码

```
#0001  /*
#0002   * CppSkel.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 9, Listing 9-1
#0006   *
#0007   * Show how to cast the parameters to _beginthreadex
#0008   * so that they will work in Visual C++ 4.x with
#0009   * the new stricter type checking.
#0010   *
#0011   * Build this file with the command line:
#0012   *
#0013   *    cl /MD CppSkel.cpp
#0014   *
#0015   */
#0016
#0017  #define WIN32_LEAN_AND_MEAN
#0018  #include <stdio.h>
#0019  #include <stdlib.h>
#0020  #include <windows.h>
#0021  #include <process.h>
#0022
#0023  typedef unsigned (WINAPI *PBEGINTHREADEX_THREADFUNC)(
#0024     LPVOID lpThreadParameter
#0025     );
#0026  typedef unsigned *PBEGINTHREADEX_THREADID;
#0027
#0028  DWORD WINAPI ThreadFunc(LPVOID);
#0029
#0030  int main()
#0031  {
#0032     HANDLE hThread;
#0033     DWORD dwThreadId;
```

```
#0034     int i = 0;
#0035
#0036     hThread = (HANDLE)_beginthreadex(NULL,
#0037            0,
#0038            (PBEGINTHREADEX_THREADFUNC)ThreadFunc,
#0039            (LPVOID)i,
#0040            0,
#0041            (PBEGINTHREADEX_THREADID)&dwThreadId
#0042         );
#0043     if (hThread) {
#0044        WaitForSingleObject(hThread, INFINITE);
#0045        CloseHandle(hThread);
#0046     }
#0047     return EXIT_SUCCESS;
#0048  }
#0049
#0050  DWORD WINAPI ThreadFunc(LPVOID n)
#0051  {
#0052     // Do something ...
#0053
#0054     return 0;
#0055  }
```

以一个 C++ 对象启动一个线程

C++ 中隐藏的 this 指针使得我们在企图启动一个线程时遭遇一些问题．下面是错误的做法和正确的做法．

? FAQ 43：我如何以一个 C++ 成员函数当做线程起始函数？

错误的做法

大部分人尝试写他们的第一个多线程程序时，常常萌发这样的想法，希望在产生一个对象的同时也产生一个线程．基本上这需要两个成员函数（member

（member functions），一个用来产生新线程，另一个则作为线程的起始函数。下面这个 ThreadObject 类就很典型：

```
#0001  class ThreadObject
#0002  {
#0003  public:
#0004      void StartThread();
#0005      virtual DWORD WINAPI ThreadFunc(LPVOID param);
#0006  private:
#0007      HANDLE m_hThread;
#0008      DWORD m_ThreadId;
#0009  };
```

ThreadObject 封装了线程起始函数，并记住线程起始时的相关信息。ThreadObject 是一个基类，任何需要线程的人都可以从它再派生出新的类，并设计自己的虚函数。派生类应该改写虚函数 ThreadFunc()，至于线程启动机能则可以免费继承得到。列表 9-2 是一个简短的例子，示范如何使用 ThreadObject。

列表 9-2 BADCLASS.CPP（一个错误示范）

```
#0001  /*
#0002   * BadClass.cpp
#0003   *
#0004   * Sample code for "Multitasking Applications in Win32"
#0005   * This is from Chapter 9, Listing 9-2
#0006   *
#0007   * Shows the wrong way to try and start a thread
#0008   * based on a class member function.
#0009   *
#0010   * THIS FILE IS NOT SUPPOSED TO COMPILE SUCCESSFULLY
#0011   * You should get an error on line 51.
#0012   *
#0013   * Build this file with the command line:
#0014   *
#0015   *    cl /MD BadClass.cpp
#0016   *
#0017   */
```

```
#0018
#0019 #include <windows.h>
#0020 #include <stdio.h>
#0021 #include <process.h>
#0022
#0023 typedef unsigned (WINAPI *PBEGINTHREADEX_THREADFUNC)(
#0024    LPVOID lpThreadParameter
#0025    );
#0026 typedef unsigned *PBEGINTHREADEX_THREADID;
#0027
#0028
#0029 class ThreadObject
#0030 {
#0031 public:
#0032    ThreadObject();
#0033    void StartThread();
#0034    virtual DWORD WINAPI ThreadFunc(LPVOID param);
#0035    void WaitForExit();
#0036 private:
#0037    HANDLE  m_hThread;
#0038    DWORD m_ThreadId;
#0039 };
#0040
#0041 ThreadObject::ThreadObject()
#0042 {
#0043    m_hThread = NULL;
#0044    m_ThreadId = 0;
#0045 }
#0046
#0047 void ThreadObject::StartThread()
#0048 {
#0049    m_hThread = (HANDLE)_beginthreadex(NULL,
#0050           0,
#0051           (PBEGINTHREADEX_THREADFUNC)ThreadFunc,
#0052           0,
#0053           0,
#0054           (PBEGINTHREADEX_THREADID)&m_ThreadId );
#0055    if (m_hThread) {
#0056       printf("Thread launched\n");
#0057    }
#0058 }
```

```
#0059
#0060  void ThreadObject::WaitForExit()
#0061  {
#0062      WaitForSingleObject(m_hThread, INFINITE);
#0063      CloseHandle(m_hThread);
#0064  }
#0065
#0066  DWORD WINAPI ThreadObject::ThreadFunc(LPVOID param)
#0067  {
#0068      // Do something useful ...
#0069
#0070      return 0;
#0071  }
#0072
#0073  void main()
#0074  {
#0075      ThreadObject obj;
#0076
#0077      obj.StartThread();
#0078      obj.WaitForExit();
#0079  }
```

列表 9-2 的程序有一个小问题：它不能够被顺利编译。如果你尝试编译它，你会在 Visual C++ 4.x 中获得这样的错误信息：

```
BadClass.cpp(51) : error C2643: illegal cast from pointer to member
```

如果你使用前面所说的，针对 _beginthreadex()“臭虫”的第一种解决方案，也就是把 ThreadFunc() 声明为 unsigned 而不是 DWORD，情况只怕更糟。这时候你不需要类型转换，而函数调用应该像这样：

```
m_hThread = (HANDLE)_beginthreadex(NULL,
                                   0,
                                   ThreadFunc,
                                   0,
                                   0,
                                   &m_ThreadId );
```

这一版程序的确可以编译并链接（事实上它应该不可以！这是编译器的一个“臭虫”），然而当程序执行起来时，马上当掉。

当你执行它时，屏幕上印出“Thread launched”，表示对 _beginthreadex() 的调用操作的确发生了，但是调试器上的“调用堆栈（call stack）”显示如下：

```
'vcall'() + 4 bytes BaseThreadStart@8 + 97 bytes
```

很明显地，线程在它一启动时就当掉了，但没有显示出为什么。

剖析 C++ 成员函数

为了了解为什么列表 9-1 的程序代码不能够有效运作，我们有必要先了解清楚成员函数实际上如何工作。一个非静态的类成员函数都有一个隐藏起来的参数被推入堆栈之中。当编译器需要处理类的成员变量时，它需要这个隐藏参数的帮忙。这就是“this”参数（一个指针）。因此，函数 ThreadObject::ThreadObject(LPVOID param) 事实上有两个参数，一个是 this 指针，另一个是 param。

当操作系统启动一个新线程时，它也为该线程产生一个专用的堆栈。操作系统必须在这一新堆栈中重新产生一个对你的线程函数的调用操作。这也就是为什么线程函数的类型一定要符合 __cdecl 或 WINAPI（也就是 __stacall）的原因。

BADCLASS 当掉的原因就是因为 ThreadObject::ThreadFunc 预期会有一个 this 指针，而操作系统只知道把 param 参数推入新堆栈中。毫不令人惊讶，当然当掉了！

译注 关于 this 指针，以及下一节将提到的解决方案“静态成员函数”，在《深入浅出 MFC》（侯俊杰著/华中科技大学出版社出版）第 2 章的“this 指针”一节以及“静态成员”一节有十分深入而详细的说明。该书第 6 章的“Callback 函数”一节曾举例说明为什么“callback 函数一定必须是静态成员函数”，而事

实上，线程函数是一个被操作系统调用的函数，正是一个 callback 函数！

正确的做法

? FAQ 44: 我如何以一个成员函数当做线程起始函数?

为了以一个成员函数启动一个线程，要么你就使用静态成员函数，要么你就得使用 C 函数（而非 C++ 成员函数）。基本上这两种技术都导入一个辅助函数，正确地建立“调用成员函数时所需的堆栈”。其实这两个技术的本质是相同的，但静态成员函数的一个优点就是，它能够处理类的 private 成员变量和 protected 成员变量（如果它们也都是 static 的话）。

列表 9-3 是一个使用静态成员函数的实例。

列表 9-3 MEMBER.CPP——藉由一个成员函数来启动一个线程

```
#0001  /*
#0002   * Member.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 9, Listing 9-3
#0006   *
#0007   * Shows how to start a thread based on a
#0008   * class member function using a static
#0009   * member function.
#0010   *
#0011   * Build this file with the command line:
#0012   *
#0013   *    cl /MD Member.cpp
#0014   *
#0015   */
#0016
#0017  #define WIN32_LEAN_AND_MEAN
#0018  #include <stdio.h>
#0019  #include <stdlib.h>
#0020  #include <windows.h>
#0021  #include <process.h>
```

```
#0022
#0023 typedef unsigned (WINAPI *PBEGINTHREADEX_THREADFUNC)(
#0024    LPVOID lpThreadParameter
#0025    );
#0026 typedef unsigned *PBEGINTHREADEX_THREADID;
#0027
#0028 //
#0029 // This ThreadObject is created by a thread
#0030 // that wants to start another thread. All
#0031 // public member functions except ThreadFunc()
#0032 // are called by that original thread. The
#0033 // virtual function ThreadMemberFunc() is
#0034 // the start of the new thread.
#0035 //
#0036 class ThreadObject
#0037 {
#0038 public:
#0039    ThreadObject();
#0040    void StartThread();
#0041    void WaitForExit();
#0042
#0043    static DWORD WINAPI ThreadFunc(LPVOID param);
#0044
#0045 protected:
#0046    virtual DWORD ThreadMemberFunc();
#0047
#0048    HANDLE  m_hThread;
#0049    DWORD   m_ThreadId;
#0050 };
#0051
#0052 ThreadObject::ThreadObject()
#0053 {
#0054    m_hThread = NULL;
#0055    m_ThreadId = 0;
#0056 }
#0057
#0058 void ThreadObject::StartThread()
#0059 {
#0060    m_hThread = (HANDLE)_beginthreadex(NULL,
#0061          0,
```

```
#0062          (PBEGINTHREADEX_THREADFUNC) ThreadObject::ThreadFunc,
#0063          (LPVOID)this,
#0064          0,
#0065          (PBEGINTHREADEX_THREADID) &m_ThreadId );
#0066      if (m_hThread) {
#0067         printf("Thread launched\n");
#0068      }
#0069  }
#0070
#0071  void ThreadObject::WaitForExit()
#0072  {
#0073      WaitForSingleObject(m_hThread, INFINITE);
#0074      CloseHandle(m_hThread);
#0075  }
#0076
#0077  //
#0078  // This is a static member function.  Unlike
#0079  // C static functions, you only place the static
#0080  // declaration on the function declaration in the
#0081  // class, not on its implementation.
#0082  //
#0083  // Static member functions have no "this" pointer,
#0084  // but do have access rights.
#0085  //
#0086  DWORD WINAPI ThreadObject::ThreadFunc(LPVOID param)
#0087  {
#0088      // Use the param as the address of the object
#0089      ThreadObject* pto = (ThreadObject*)param;
#0090      // Call the member function. Since we have a
#0091      // proper object pointer, even virtual functions
#0092      // will be called properly.
#0093      return pto->ThreadMemberFunc();
#0094  }
#0095
#0096
#0097  //
#0098  // This above function ThreadObject::ThreadFunc()
#0099  // calls this function after the thread starts up.
#0100  //
#0101  DWORD ThreadObject::ThreadMemberFunc()
#0102  {
#0103      // Do something useful ...
```

```
#0104     return 0;
#0105  }
#0106
#0107
#0108  void main()
#0109  {
#0110     ThreadObject obj;
#0111
#0112     obj.StartThread();
#0113     obj.WaitForExit();
#0114  }
```

如要改用 C-style 函数来取代静态成员函数，你只需要对程序代码做一点点极小的改变即可。你必须以 C 语言型式来声明 ThreadFunc()，你必须把线程的起始成员函数移到类以外的公共区域（public section），你必须在函数名称中拿掉类名称。完整的程序代码列于书附盘片中的 MEMBER2.CPP 内，其中唯一的改变，列出于下：

```
#0001  DWORD WINAPI ThreadFunc(LPVOID param);
#0002
#0003  class ThreadObject
#0004  {
#0005  public:
#0006     // ...
#0007
#0008     // Thread member function must be public
#0009     // or the C-style function will not have
#0010     // access rights.
#0011     virtual DWORD ThreadMemberFunc();
#0012
#0013  protected:
#0014     // ...
#0015  };
#0016
#0017  DWORD WINAPI ThreadFunc(LPVOID param)
#0018  {
#0019     ThreadObject* pto = (ThreadObject*)param;
#0020     return pto->ThreadMemberFunc();
#0021  }
```

建立比较安全的 Critical Sections

以 C++ 实现多线程程序设计，有这么多优点，以至于把它拿来与 C 做个比较时，真不知道从哪里说起。在接下来的数节中，我将告诉你如何利用构造函数（constructor）和析构函数（destructor）写一个比较安全的多线程程序，以及如何利用虚函数（virtual function）和多型（polymorphism）等特质做出一个可互换的锁定机制（locking mechanisms）。

就让我们从“产生一个类并内嵌一个 critical section”开始吧。我要使用两个最基本的 C++ 性质，构造函数和析构函数，提供有保障的初始化操作和清理（cleanup）操作。

还记得第 3 章你必须操心何时调用 InitializeCriticalSection() 和 DeleteCriticalSection() 吗？如果使用构造函数和析构函数，它们都会被适当地自动调用。

列表 9-4 显示了一个非常简单的范例。

列表 9-4 在 C++ 类中封装一个 critical section

```
#0001  class CriticalSection
#0002  {
#0003  public:
#0004         CriticalSection();
#0005        ~CriticalSection();
#0006         void Enter();
#0007         void Leave();
#0008  private:
#0009         CRITICAL_SECTION m_CritSect;
#0010  };
#0011
#0012  CriticalSection::CriticalSection()
#0013  {
#0014         InitializeCriticalSection(&m_CritSect);
#0015  }
```

```
#0016
#0017 CriticalSection::~CriticalSection()
#0018 {
#0019       DeleteCriticalSection(&m_CritSect);
#0020 }
#0021
#0022 CriticalSection::Enter()
#0023 {
#0024       EnterCriticalSection(&m_CritSect);
#0025 }
#0026
#0027 CriticalSection::Leave()
#0028 {
#0029       LeaveCriticalSection(&m_CritSect);
#0030 }
```

乍见之下这个类似乎非常无聊，但事实上，比起使用最直接的 API 函数，它提供了一些很棒的优点。为了保护一个字符串变量，我们可以在类中加上一个 CriticalSection 成员变量。由于 C++ 会自动调用构造函数和析构函数，所以程序员使用字符串变量时，不需要修改任何程序代码，就可以用上 critical section。

举个例子，列表 9-5 就显示了一个 String 类，使用 CriticalSection 类。

列表 9-5 在 String 类中使用 CriticalSection 类

```
#0001 class String
#0002 {
#0003 public:
#0004       String();
#0005       virtual ~String();
#0006       virtual void Set(char* str);
#0007       int GetLength();
#0008 private:
#0009       CriticalSection m_Sync;
#0010       Char*          m_pData;
#0011 };
```

```
#0012
#0013  String::String()
#0014  {
#0015          // The constructor for m_Sync will have
#0016          // already been called automatically because
#0017          // it is a member variable.
#0018          m_pData = NULL;
#0019  }
#0020
#0021  String::~String()
#0022  {
#0023          // Use the "array delete" operator
#0024          // Note: "delete" checks for NULL automatically
#0025          m_Sync.Enter();
#0026          delete [] m_pData;
#0027          m_Sync.Leave();
#0028          // The destructor for m_Sync will be
#0029          // called automatically.
#0030  }
#0031
#0032  void String::Set(char *str)
#0033  {
#0034          m_Sync.Enter();
#0035          delete [] m_pData;
#0036          m_pData = new char[::strlen(str)+1];
#0037          ::strcpy(m_pData, str);
#0038          m_Sync.Leave();
#0039  }
#0040
#0041  int String::GetLength()
#0042  {
#0043          if (m_pData == NULL)
#0044              return 0;
#0045
#0046          m_Sync.Enter();
#0047          int len = ::strlen(m_pData);
#0048          m_Sync.Leave();
#0049          return len;
#0050  }
```

你可以声明一个 String 变量，可以完全不需要了解 critical section，就获得同步（synchronization）效果。例如，下面这个函数将在多线程程序中运作良好：

```
void SomeFunction(String& str)
{
       str.Set("Multithreading");
}
```

建立比较安全的 Locks

现在我们能够自动产生一个 critical section 并自动清除之了。让我们设计另一个类并自动清除 lock。如果你企图写一个类似 Truncate() 的函数，并在其中使用 CriticalSection 类（如目前所定义），最终你必须做一些手动的清除工作。例如，为了让函数能够在进行到一半时结束并返回（return），你必须在每一个返回点上调用 Release()（很明显地，我可以做一点不同的结构化设计，而将此问题解决，但请容我以此为例）。

```
#0001  void String::Truncate(int length)
#0002  {
#0003          if (m_pData == NULL)
#0004              return ;
#0005          m_Sync.Enter();
#0006          if (length >= GetLength())
#0007          {
#0008              m_Sync.Leave();
#0009              return;
#0010          }
#0011          m_pData[length] = '\0';
#0012          m_Sync.Leave();
#0013  }
```

啊，我可以产生另一个类：Lock，其构造函数和析构函数负责“进入”和“离开”critical section（列表 9-6）。Lock 的构造函数需要将一个 CriticalSection 对象指针作为唯一参数。在其内部，Lock 持有一个指针，指向被锁定的 CriticalSection 对象。

列表 9-6 抽象的同步化对象（Abstracting synchronization objects）

```
#0001  class Lock
#0002  {
#0003  public:
#0004         Lock(CriticalSection* pCritSect);
#0005        ~Lock();
#0006  private:
#0007         CriticalSection* m_pCritical;
#0008  };
#0009
#0010  Lock::Lock(CriticalSection* pCritSect)
#0011  {
#0012         m_pCritical = pCritSect;
#0013         EnterCriticalSection(m_pCritical);
#0014  }
#0015
#0016  Lock::~Lock()
#0017  {
#0018         LeaveCriticalSection(m_pCritical);
#0019  }
```

就像 CrticalSection 类一样，Lock 类本身没有做太多事情，但你看看，现在可以多么容易地改写 Truncate()。析构函数会被自动调用，而 critical section 会被自动地解除锁定。绝不会再有忘记将 critical section 解除锁定的情况发生。

```
#0001  void String::Truncate(int length)
#0002  {
#0003         if (m_pData == NULL)
#0004             return ;
#0005         // Declaring a "Lock" variable will call
#0006         // the constructor automatically.
```

```
#0007         Lock lock(&m_Sync);
#0008         if (length >= GetLength())
#0009         {
#0010            // lock cleans itself up automatically
#0011            return;
#0012         }
#0013         m_pData[length] = '\0';
#0014         // lock cleans itself up automatically
#0015   }
```

在 C++ 中，当一个变量超过其存活范围（scope），它的析构函数会被自动调用。所以当此函数“中途离席”，lock 变量的析构函数就会被调用起来。

建立可互换（Interchangeable）的 Locks

现在我要使用 C++ 虚函数来建立可自由互换的同步机制。在 C++ 中，一个抽象基类（Abstract Base Class，ABC）可以透过一个标准界面来定义一个对象。我们将建立一个名为 LockableObject 的 ABC，再根据它来产生实际类，实现出某些同步机制出来。

下面就是 LockableObject。其设计和 CriticalSection 差不多，但是我们将把 Enter() 和 Leave() 改名为 Lock() 和 Unlock()，以反应出此类的特色。

```
#0001   class LockableObject
#0002   {
#0003   public:
#0004         LockableObject() { }
#0005         virtual ~LockableObject() { }
#0006         virtual void Lock() = 0;
#0007         virtual void Unlock() = 0;
#0008   };
```

现在我们可以根据 LockableObject 派生出第二版的 CriticalSection 了（列表 9-7），除了声明部分，看起来和前一版似乎没什么不同。但注意，除了构造函数，每一个成员函数都被声明为虚函数。

列表 9-7 从 LockableObject 派生而来的 CriticalSection

```
#0001 class CriticalSectionV2 : public LockableObject
#0002 {
#0003 public:
#0004       CriticalSectionV2();
#0005       virtual ~CriticalSectionV2();
#0006       virtual void Lock();
#0007       virtual void Unlock();
#0008
#0009 private:
#0010       CRITICAL_SECTION m_CritSect;
#0011 };
#0012
#0013 CriticalSectionV2::CriticalSectionV2()
#0014 {
#0015       InitializeCriticalSection(&m_CritSect);
#0016 }
#0017
#0018 CriticalSectionV2::~CriticalSectionV2()
#0019 {
#0020       DeleteCriticalSection(&m_CritSect);
#0021 }
#0022
#0023 CriticalSectionV2::Lock()
#0024 {
#0025       EnterCriticalSection(&m_CritSect);
#0026 }
#0027
#0028 CriticalSectionV2::Unlock()
#0029 {
#0030       LeaveCriticalSection(&m_CritSect);
#0031 }
```

现在我们可以写 Lock 类的第二个版本，使它接受一般性的 LockableObject，而不再是特定的某个同步控制对象。请注意虚函数是如何地用来锁定对象而根本不需要知道其真正对象类型是什么（列表 9-8）。

列表 9-8 一个极安全又简单无比的 Lock

```
#0001  class LockV2
#0002  {
#0003  public:
#0004         LockV2(LockableObject* pLockable);
#0005        ~LockV2();
#0006
#0007  private:
#0008         LockableObject* m_pLockable;
#0009  };
#0010
#0011  LockV2:LockV2(LockableObject* pLockable)
#0012  {
#0013         m_pLockable = pLockable;
#0014         m_pLockable->Lock();
#0015  }
#0016
#0017  LockV2:~LockV2()
#0018  {
#0019         m_pLockable->Unlock();
#0020  }
```

现在我们手上有了 CriticalSectionV2 和 LockV2，彼此间互有协定。以 C++ 类取代对 Win32 API 的直接调用，就可以保证初始化操作和清理操作总是能够被正确地执行。现在让我们使用这些新的同步控制类，写下 String 类的第二版（列表 9-9）。

列表 9-9 以前述“安全又简单无比的 Lock”重写 String 类

```
#0001  class StringV2
#0002  {
#0003  public:
#0004         StringV2();
```

```
#0005         virtual ~StringV2();
#0006         virtual void Set(char* str);
#0007         int GetLength();
#0008  private:
#0009         CriticalSectionV2 m_Lockable;
#0010         Char*             m_pData;
#0011  };
#0012
#0013  StringV2::StringV2()
#0014  {
#0015         m_pData = NULL;
#0016  }
#0017
#0018  StringV2::~StringV2()
#0019  {
#0020         // The program must ensure that
#0021         // it is safe to destroy the object.
#0022         delete [] m_pData;
#0023  }
#0024
#0025  void StringV2::Set(char *str)
#0026  {
#0027         LockV2 localLock(&m_Lockable);
#0028         delete [] m_pData;
#0029         m_pData = NULL; // In case new throws an exception
#0030         m_pData = new char[::strlen(str)+1];
#0031         ::strcpy(m_pData, str);
#0032  }
#0033
#0034  int String::GetLength()
#0035  {
#0036         LockV2 localLock(&m_Lockable);
#0037         if (m_pData == NULL)
#0038            return 0;
#0039         return ::strlen(m_pData);
#0040  }
```

异常情况（Exceptions）的处理

使用 C++ 类实现同步机制的最后一个巨大好处是，它们可以和异常情况处理机制（exception handling）一起运作。我假设你已经熟悉 C++ 异常情况和 Win32 异常情况的运作。

异常情况处理机制的好处是，它在标准的函数调用和返回（call/return）模式之外运作。当一个异常情况被丢出来时，Win32 就和 C++ runtime library 一起合作，“unwind”堆栈（译注）并清除所有的变量。以 C 语言撰写程序代码，并使之能够安全处理异常情况，是比较困难的，因为你必须自己做所有的杂务。而在 C++ 之中，只要有一些先见之明，你就可以让编译器做掉所有的工作。

译注 所谓“stack unwinding”就是：当一个异常情况发生时，将堆栈中的所有对象都析构掉。

使用本章所开发出来的 Lock 类，被锁定的对象会在“stack unwinding”发生时自动析构。是的，析构函数会自动针对对象解除锁定。其结果也就是 locks 被自动清除干净。

提要

就像直到目前你在本书中的所见所闻一样，多线程的一个成功关键就在于勤奋地处理你的数据。绝对不要去碰触那些你没有把握保持一致性的数据。

C++ 漂亮的地方就在于它可以利用类的定义绝对保证数据的合法性。通过类似幽禁的方式，把数据局限在类的 private 区域内，就可以约束类用户，必须借着行为良好的成员函数才能够存取那些数据。虽然你也可以争辩

说所有操作都可以使用 C 语言来完成，但是，让编译器施行你的设计需求，是很重要的一项利益。如果程序员想走捷径，抄短路，直接处理未经锁定的数据，可能会制造出一些很难挑出的“臭虫”。

说所有操作都可以使用 C 语言来完成，但是，让编译器施行你的设计需求，是很重要的一项利益。如果程序员想走捷径，抄短路，直接处理未经锁定的数据，可能会制造出一些很难挑出的“臭虫”。

MFC 中的线程

这一章介绍支持 worker 线程和 UI 线程的 MFC 类，以及那些封装了 Win32 同步对象的 MFC 类。

重要！ 如果要在 MFC 程序中产生一个线程，而该线程将调用 MFC 函数或使用 MFC 的任何数据，那么你必须以 AfxBeginThread() 或 CWinThread::CreateThread() 来产生这些线程。

Microsoft Foundation Classes，也就是一般人简称的 MFC，是微软公司对于“降低 Windows 程序设计之厌烦无聊及困难度”而做出的最大贡献。MFC 使得对话框的产生极为简单。它也实现出消息派送系统（message dispatching），处理 WPARAM 和 LPARAM 的易犯错误。MFC 甚至是引诱某些人进入 C++ 的原动力。

不久前，MFC 也加入了对多线程的支持。在一个典型的 MFC 程序中，多线程的支持隐藏在一大段非常惊人的工作之后。MFC 甚至企图强化某些与多线程有关的 Win32 观念。我想，虽然它的成功与否还有争论，但我们不妨来看看它的设计观念及其后续的改良。

到目前为止，面对 Win32 程序设计中的线程，你已经看到了两个层次。一个是使用 Win32 API 的 CreateThread() 和 EndThread()，一个是使用 C runtime library 所提供的 _beginthreadex() 和 _endthreadex()。这两种方式其实还蛮类似的，MFC 的线程类则提供了完全不同的故事。

在 MFC 中启动一个 Worker 线程

你已经看过了 worker 线程和 GUI 线程之间的一些理论上的差异。两者一般而言都是以 CreateThread() 或 _beginthreadex() 开始其生命。如果线程调用 GetMessage() 或 CreateWindow() 之类函数，消息队列便会产生（译注），而 worker 线程也就摇身一变成了 GUI 线程（或称为 UI 线程）。

译注 可能你一直把消息队列想象为一个数组（array），所以对于“消息队列便会产生”这句话感到迷惑。没错，消息队列是一个数组，但那是 Windows 3.1 时代的老故事。Windows 95 的消息队列是一个链表（linked-list），所以，必要的时候，才有元素产生出来。关于消息队列的数据结构，请参考 Matt Pietrek 所著的“Windows 95 System Programming SECRETS”第 3 章。

MFC 对这两种线程有重大的区别。虽然两种线程都是以 AfxBeginThread() 启动，但是 MFC 利用 C++ 函数的 overloading 性质，对该函数提供了两种不同的声明。编译器会根据你所提供的参数，自动选择正确的一个来用。

AfxBeginThread() 的第一种形式用来启动一个 worker 线程。请注意这一函数使用 C++ 的参数缺省能力。是的，你可以只用两个参数调用此函数，其他参数将使用函数原型中声明的缺省值。

```
CWinThread* AfxBeginThread(
    AFX_THREADPROC pfnThreadProc,
    LPVOID pParam,
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
```

参数

pfnThreadProc	函数名称，用来启动线程。
pParam	任意 4 字节数值，用来传给新线程。它可以是个整数，或指针，或单纯只是个 0。
nPriority	新线程的优先权。如果是 0（缺省值），表示新线程的优先权将与目前优先权相同。
nStackSize	新线程的堆栈大小（字节）。和 CreateThread() 一样，0 表示使用默认的堆栈大小。
dwCreateFlags	此值必须为 0 或 CREATE_SUSPENDED。如果你省略它，表示标记为 0，也就是立刻开始新线程的生命。
lpSecurityAttrs	CreateThread() 所需的安全属性。在 Windows 95 中无效。

返回值

如果失败，传回 NULL。否则传回一个指针，指向新产生出来的 CWinThread 对象。

第一件需要注意的事情就是，AfxBeginThread() 传回一个指向 CWinThread 对象的指针，而非一个 HANDLE。和其他许多 MFC 类一样，CWinThread 只是许多相关函数（此处为与线程有关的 API）的一层外包装而已，其中的

成员函数，如 ResumeThread() 和 SetThreadPriority() 都应该是你耳熟能详的名称，只不过它们现在成了 C++ 成员函数。

CWinThread 被极其小心地设计，解决了许多我们在 CreateThread() 和 _beginthreadex() 中所遇到的困难。CWinThread 甚至正确地处理了原本期望 _beginthread() 所做的清理（cleanup）工作。视你的需求，CWinThread 可以极有弹性。

列表 10-1 是最简单的一个使用实例。这个范例在机能上和第 2 章的 NUMBERS 一样，只不过是改用 C++ 和 MFC 来写。

列表 10-1 NUMBERS.CPP——以 AfxBeginThread() 产生 worker 线程

```
#0001  /*
#0002   * Numbers.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 10, Listing 10-1
#0006   *
#0007   * Demonstrate basic thread startup in MFC
#0008   * using AfxBeginThread.
#0009   *
#0010   * Compile with the IDE or: nmake -f numbers.mak
#0011   */
#0012
#0013  #include <afxwin.h>
#0014
#0015  CWinApp TheApp;
#0016
#0017  UINT ThreadFunc(LPVOID);
#0018
#0019  int main()
#0020  {
#0021      for (int i=0; i<5; i++)
#0022      {
#0023          if (AfxBeginThread( ThreadFunc, (LPVOID)i ))
#0024              printf("Thread launched %d\n", i);
#0025      }
#0026
```

```
#0027     // Wait for the threads to complete.
#0028     Sleep(2000);
#0029
#0030     return 0;
#0031  }
#0032
#0033
#0034  UINT ThreadFunc(LPVOID n)
#0035  {
#0036     for (int i=0;i<10;i++)
#0037        printf("%d%d%d%d%d%d%d%d%d%d\n",n,n,n,n,n,n,n,n,n,n);
#0038     return 0;
#0039  }
```

把这个程序和 C 语言版做一比较，可真是轻快多了。我们不需要为 AfxBeginThread() 提供任何暧昧不明的参数，我们也不需要声明变量用以储存像线程 ID 之类的东西。更重要的是，我们不必担心何时关闭 handle。让我们看看这些是如何办到的。

- 清除 CWinThread 对象。默认情况下，当线程结束时，CWinThread 对象会自动被删除，这是因为 MFC 安插了它自己的线程启动函数，并且更在你自己的线程函数之前，于是可以承担清理工作。好消息是，对于初上路的程序员而言，事情因此简化了许多。坏消息是，这使得我们需要做更多事情，才能让我们“等待”CWinThread 对象。我马上就会告诉你怎么做。
- 关闭线程的 handle。用以删除 CWinThread 对象的那个清理函数，同时也关闭了线程的 handle。当线程结束其生命时，它的 handle 也就被关闭了。
- 储存线程的 handle 和 ID。这些数据被储存在 CWinThread 对象的成员变量中，分别是 m_hThread 和 m_nThreadID。不过，这些数据并不是可以随意取用的。稍后我会说明这一点。

MFC 的另一个优点是它有 assert 操作。就像我们在先前数章中用过的 MTVERIFY 宏的道理一样，MFC 也有它自己的调试哲学，可以确保参数正确，调用成功。例如，假设你以 NULL 作为线程启动函数的地址，并建立 MFC 程序的调试版（debug build），MFC 便会对你提出警告。不过如果

你建造的是 MFC 程序的一般版（release build），就没办法获得这种好处。这是一种选择，你可以在调试版中获得安全，在一般版中获得速度。“assertion checking”是 MFC 里面最有用的一个性质。

安全地使用 AfxBeginThread() 的传回值

或许你已经注意到了，我在列表 10-1 中调用的是 Sleep() 而不是 WaitForMultipleObjects()，原因是 CWinThread 的缺省用途有着和 _beginthread() 一样的意义。如果从线程启动到结束所花费的时间很短，CWinThread 对象可能在 AfxBeginThread() 返回时就已经被砍了。在这种情况下，任何操作只要触及线程的 handle，就会让你的程序当掉。幸运的是，这个替代方案很简单，并且已经内建在 CWinThread 中了。你必须付出的代价是，对象的管理需要比较多的工作。

? FAQ 45：我如何能够阻止一个线程杀掉它自己？

CWinThread 中有一个成员变量 m_bAutoDelete，这个参数可以阻止 CWinThread 对象被自动删除。为了能够设定此变量而不产生一个 race condition，你必须先以挂起状态产生线程。列表 10-2 显示 NUMBERS2，它适当地完成了这种启动逻辑。

列表 10-2 NUMBERS2.CPP——使用 AfxBeginThread() 并将“自动删除”功能关闭

```
#0001  /*
#0002   * Numbers2.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 10, Listing 10-2
#0006   *
#0007   * Demonstrate thread startup in MFC
#0008   * using AfxBeginThread, but prevent
#0009   * CWinThread from auto-deletion so that
#0010   * we can wait on the thread.
```

你建造的是 MFC 程序的一般版（release build），就没办法获得这种好处。这是一种选择，你可以在调试版中获得安全，在一般版中获得速度。“assertion checking”是 MFC 里面最有用的一个性质。

安全地使用 AfxBeginThread() 的传回值

或许你已经注意到了，我在列表 10-1 中调用的是 Sleep() 而不是 WaitForMultipleObjects()，原因是 CWinThread 的缺省用途有着和 _beginthread() 一样的意义。如果从线程启动到结束所花费的时间很短，CWinThread 对象可能在 AfxBeginThread() 返回时就已经被砍了。在这种情况下，任何操作只要触及线程的 handle，就会让你的程序当掉。幸运的是，这个替代方案很简单，并且已经内建在 CWinThread 中了。你必须付出的代价是，对象的管理需要比较多的工作。

? FAQ 45：我如何能够阻止一个线程杀掉它自己？

CWinThread 中有一个成员变量 m_bAutoDelete，这个参数可以阻止 CWinThread 对象被自动删除。为了能够设定此变量而不产生一个 race condition，你必须先以挂起状态产生线程。列表 10-2 显示 NUMBERS2，它适当地完成了这种启动逻辑。

列表 10-2 NUMBERS2.CPP——使用 AfxBeginThread() 并将“自动删除”功能关闭

```
#0001  /*
#0002   * Numbers2.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 10, Listing 10-2
#0006   *
#0007   * Demonstrate thread startup in MFC
#0008   * using AfxBeginThread, but prevent
#0009   * CWinThread from auto-deletion so that
#0010   * we can wait on the thread.
```

```
#0053          printf("%d%d%d%d%d%d%d%d\n",n,n,n,n,n,n,n,n);
#0054      return 0;
#0055  }
```

以下所描述的启动逻辑，能够适当地初始化 CWinThread 对象中的数据。当我们从 CWinThread 中派生出我们自己的类，并且需要适当地初始化时，你会再次看到这样的逻辑。

1. 产生线程，并令其为 CREATE_SUSPENDED 状态。

2. 设定 CWinThread 对象中的成员变量（译注：指的就是 m_bAutoDelete）。

3. 调用 ResumeThread()。

很明显，如果你设定对象，使它不能够被自动删除，那么你就得自己删除之。在 NUMBERS2，对象是以这种方法删除的：

```
delete pThreads[i];
```

请注意 NUMBERS2 并没有调用 CloseHandle() 关闭线程的 handle。CWinThread 的析构函数会自动完成此事，不管 m_bAutoDelete 是否被设立。

同时也请你注意，我们正在使用的是 MFC 形式的 ResumeThread()，而非 raw Win32 形式。别忘了，MFC 调试版会先做某些“assertion checking”，然后才调用 raw Win32 形式的 ResumeThread()。由于这个调用是被实现成一个 inline 函数，所以使用它并不需要付出速度上的代价。

```
pThreads[i]->ResumeThread();
```

从头做起，谈“线程启动”

你可以自行使用 CWinThread 以获得比 AfxBeginThread() 更好的控制。AfxBeginThread() 只不过是一个辅助函数，将 CWinThread 的使用做一层外包装而已。AfxBeginThread() 内部实际上做了以下服务：

1. 在 heap 中配置一个新的 CWinThread 对象。
2. 调用 CWinThread::CreateThread() 并设定属性，使线程以挂起状态产生。
3. 设定线程优先权。
4. 调用 CWinThread::ResumeThread()。

很明显地，你自己也可以轻易完成这些事情。以下就是为什么需要自己动手的原因。CWinThread 结构是一个储存“线程间通讯”（Interthread communication）数据的一个理想地方，却不是储存线程启动信息的理想地方。如果使用 _beginthreadex() 或 CreateThread()，你可以配置一个启动信息的数据结构，并把此结构的指针当做线程函数的参数（LPVOID）。

为了能够有效运作，你必须首先对 CWinThread 进行 subclassing 操作（译注），并在其中加上自己的数据。为了获得额外的利益，我们现在可以设定这一线程的构造函数，使对象被产生的时候，“自动删除”能力不起作用。列表 10-3 就是一个例子，告诉你如何从 CWinThread 中产生出一个自己的类，我把它命名为 CUserThread。我们可以指定启动函数为一个静态成员函数，一如第 9 章所述。这么做的最大优点就是我们可以降低 namespace 的污染。我们现在可以拥有 20 个同样名为 ThreadFunc() 的函数，每一个函数都针对一个不同种类的线程。如何识别？就靠“它是哪一个类的成员”为凭了。本例的线程启动函数的全名是 CUserThread::ThreadFunc()。

译注 所谓“subclassing”操作，此处意指派生类（derived classes）的操作。而不是 raw API 中的 subclassing 操作（用到 SetWindowLong、GetWindowLong 那一套）。后者精神虽同，手法有异。

列表 10-3 NUMCLASS.CPP——不靠 AfxBeginThread() 产生 worker 线程

```
#0001  /*
#0002   * NumClass.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 10, Listing 10-3
#0006   *
#0007   * Demonstrate worker thread startup in MFC
#0008   * without AfxBeginThread.
```

```
#0009  *
#0010  * Compile with the IDE or: nmake -f NumClass.mak
#0011  */
#0012
#0013 #include <afxwin.h>
#0014
#0015 CWinApp TheApp;
#0016
#0017
#0018 class CUserThread : public CWinThread
#0019 {
#0020 public:  // Member functions
#0021     CUserThread(AFX_THREADPROC pfnThreadProc);
#0022
#0023     static UINT ThreadFunc(LPVOID param);
#0024
#0025 public:  // Member data
#0026     int m_nStartCounter;
#0027
#0028 private: // The "real" startup function
#0029     virtual void Go();
#0030 };
#0031
#0032 CUserThread::CUserThread(AFX_THREADPROC pfnThreadProc)
#0033 : CWinThread(pfnThreadProc, NULL)  // Undocumented constructor
#0034 {
#0035     m_bAutoDelete = FALSE;
#0036
#0037     // Set the pointer to the class to be the startup value.
#0038     // m_pThreadParams is undocumented,
#0039     // but there is no work-around.
#0040     m_pThreadParams = this;
#0041 }
#0042
#0043
#0044 int main()
#0045 {
#0046     CUserThread* pThreads[5];
#0047
#0048     for (int i=0; i<5; i++)
#0049     {
#0050         // Pass our static member as the startup function
```

```
#0051        pThreads[i] = new CUserThread( CUserThread::ThreadFunc );
#0052
#0053         // Set the appropriate member variable
#0054         pThreads[i]->m_nStartCounter = i;
#0055
#0056         // Start the thread in motion
#0057         VERIFY(
#0058           pThreads[i]->CreateThread() );
#0059         printf("Thread launched %d\n", i);
#0060      }
#0061
#0062      for (i=0; i<5; i++)
#0063      {
#0064        WaitForSingleObject(pThreads[i]->m_hThread, INFINITE);
#0065        delete pThreads[i];
#0066      }
#0067
#0068      return 0;
#0069   }
#0070
#0071
#0072   // static
#0073   UINT CUserThread::ThreadFunc(LPVOID n)
#0074   {
#0075      CUserThread* pThread = (CUserThread*)n;
#0076      pThread->Go();
#0077      return 0;
#0078   }
#0079
#0080   void CUserThread::Go()
#0081   {
#0082      int n = m_nStartCounter;
#0083      for (int i=0;i<10;i++)
#0084         printf("%d%d%d%d%d%d%d%d%d\n",n,n,n,n,n,n,n,n,n);
#0085   }
```

列表 10-3 的程序指出一个有趣的 MFC 问题。AfxBeginThread() 使用了一个未公开的 CWinThread 构造函数，以设定启动函数。这个疏忽很奇怪，因为如果没有这个构造函数，我们就不可能为了 non-GUI 线程而对 CWinThread 进行量身定制（customize）的操作。

这个未公开构造函数有两个参数，分别是启动函数和启动时的参数。CUserThread 以一个“initializer list”调用 CWinThread 的 nondefault 构造函数，这几乎给了我们所需要的行为模式。然而我们还需要设定一个未公开的成员变量 CWinThread::m_pThreadParams，它最终会变成线程函数的起始参数。

配置了 CUserThread 的结构之后，我们将线程所需的成员变量初始化，然后调用成员函数 CreateThread()。我们将使用缺省形式：无任何参数，这使得这一函数非常容易完成。

如果你要重复使用这段代码，你可以从 CUserThread 派生出自己的类，然后改写成员函数 Go()就可以了。

在 MFC 中启动一个 UI 线程

在 MFC 中启动一个 UI 线程和启动一个 worker 线程是全然不同的两回事。你还是需要用到 CWinThread 和 AfxBeginThread()，但它们的用法和前面所说的截然不同。可以说，MFC 已经为我们做了许多刚才所说的那些关于“将线程的产生过程抽象化”的工作。

AfxBeginThread() 第二版本

以下是 AfxBeginThread() 的第二种形式：

```
CWinThread* AfxBeginThread(
    CRuntimeClass* pThreadClass,
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
```

```
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

参数

pThreadClass	指向你所产生的一个类的 runtime class（译注）。该类派生自 CWinThread。 译注 所谓 runtime class，是 MFC 为了实现出 RTTI（Runtime Class Information）和 Dynamic Creation 等性质，而设计的一种架构，其具体内容则为 CRuntimeClass。整个 MFC 系统在你的程序开始执行之前一刻，以许多的 CRuntimeClass 对象（每一对象代表一个 MFC 类）组织成一个巨大绵密的网络（其实是个链表）。详情请看《深入浅出 MFC》（侯俊杰/松岗）第 3 章的“类别型录网与 CRuntimeClass”和第 8 章的“DYNAMIC/DYNCREATE/SERIAL 三宏”。
nPriority	新线程的优先权。如果是 0（缺省值），表示新线程将拥有和目前线程相同的优先权。
nStackSize	新线程的堆栈大小（字节）。和 CreateThread() 相同，此值若为 0，表示使用默认的堆栈大小。
dwCreateFlags	此标记值若不为 0 就是 CREATE_SUSPENDED。如果遗漏未写，默认为 0，于是线程在产生之后立刻开始执行。
lpSecurityAttrs	安全属性，就像 CreateThread() 所需的一样。在 Windows 95 中无效。

返回值

AfxBeginThread() 传回 NULL 表示失败。否则，它会传回一个指针，指向新的 CWinThread 对象。

当我第一次看到这个函数时，我想它是 CreateThread() 的一个轻微变种，于是我说“没问题啦”。但是当我尝试使用它，放进第一个参数时，却踢到一块铁板。我才猛然警觉，这个函数与我的认知是多么不同。

AfxBeginThread() 的 UI 线程版本，期望为你在 pThreadClass 参数中所指定的类配置一个对象。此类必须派生自 CWinThread。CWinThread 提供了一堆多样化的虚函数，你可以改写之以帮助消息的处理、线程的启动（startup）和清理（cleanup），以及异常情况的处理（exception handling）。这些虚函数列举如下：

- InitInstance()
- ExitInstance()
- OnIdle()
- PreTranslateMessage()
- IsIdleMessage()
- ProcessWndProcException()
- ProcessMessageFilter()
- Run()

你并不是非得改写它们不可。默认情况下只要改写 InitInstance()，MFC 就会开启一个消息循环。

? FAQ 46：CWinApp 和主线程之间有什么关系？

如果你已经写过一些 MFC 程序，上述这些函数名称有一些我想你并不陌生。InitInstance() 和 ExitInstance() 是 AppWizard 自动为每一个 MFC 程序产生的。至于 PreTranslateMessage() 则是每一个企图深入 MFC 消息绕行系统（message routing）的人所熟悉的。其实，每一个 MFC 程序所需要的那个所谓的 "application object"，就是派生自 CWinThread（译注：虽然表面上它是 CWinApp 对象，不过 CWinApp 又派生自 CWinThread）。因此，主线程毫无间隙地能够和整个程序以及其他线程整合在一起。

以 ClassWizard 产生一个 UI 线程

回到我们原来的主题上。如何能够设定你自己的一个 CWinThread 派生对象，并把它交给 AfxBeginThread() 呢？最简单的方法就是使用 Visual C++ 所提供的 wizards 工具。在 Visual C++ 4.x 中操作步骤如下：

1. 打开你的项目——它必须是一个 MFC 应用程序。
2. 在【View】菜单中选择 ClassWizard。
3. 选按【Add Class】按钮。
4. 从下拉菜单中选按【New...】项目。
5. 你会看到如图 10-1 所示的【Create New Class】对话框。
6. 输入类名称，例如 CDemoThread。
7. 接下来是关键操作：在【Base class】下拉列表框中选择 CWinThread。
8. 选按【Create】钮。

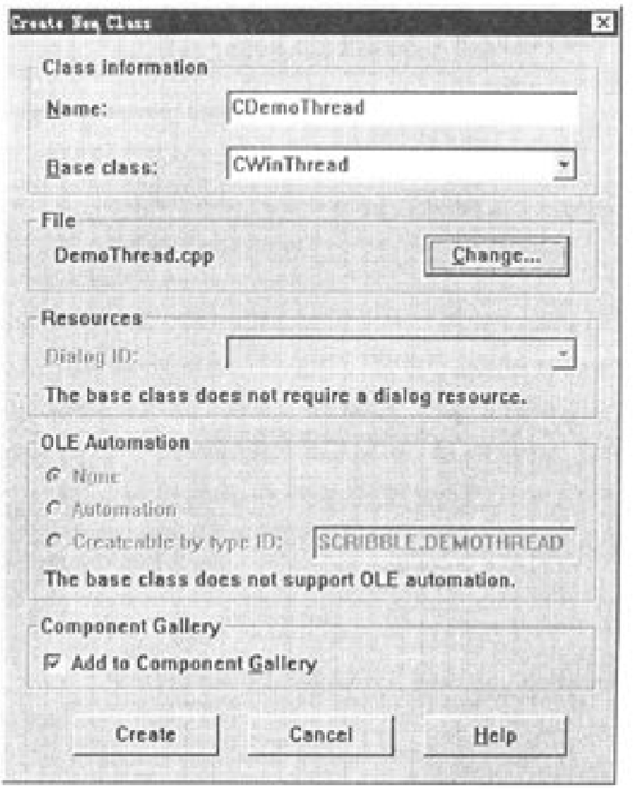


图 10-1 【Create New Class】对话框（译注：这是 Visual C++ 4.0 画面。Visual C++ 6.0 略有不同）

本例之中，ClassWizard 会为我们产生两个文件，分别是 DemoThread.cpp 和 DemoThread.h，并把它们加到项目之中。在 DemoThread.cpp 中，你会发现 ClassWizard 已经帮我们产生了 InitInstance() 和 ExitInstance()，并为该线程产生出最上层的消息映射表（message map）。还有比这更轻松的吗?

ClassWizard 所产生的 CDemoThread 类声明显示于列表 10-4。

列表 10-4 ClassWizard 所产生的 CDemoThread 类声明

```
#0001  class CDemoThread : public CWinThread
#0002  {
#0003      DECLARE_DYNCREATE(CDemoThread)
#0004  protected:
#0005      CDemoThread();  // protected constructor used by dynamic creation
#0006
#0007  // Attributes
#0008  public:
#0009
#0010  // Operations
#0011  public:
#0012
#0013  // Overrides
#0014      // ClassWizard generated virtual function overrides
#0015      //{{AFX_VIRTUAL(CDemoThread)
#0016      public:
#0017      virtual BOOL InitInstance();
#0018      virtual int ExitInstance();
#0019      //}}AFX_VIRTUAL
#0020
#0021  // Implementation
#0022  protected:
#0023      virtual ~CDemoThread();
#0024
#0025      // Generated message map functions
#0026      //{{AFX_MSG(CDemoThread)
#0027      // NOTE - the ClassWizard will add and remove member functions here.
#0028      //}}AFX_MSG
#0029
#0030      DECLARE_MESSAGE_MAP()
#0031  };
```

在类声明一开始处，你会看到 DECLARE_DYNCREATE() 宏的使用。这个宏实现出运行时类型识别（RTTI）和动态生成（Dynamic Creation），同时也是让 CDemoThread 与 AfxBeginThread() 合作的关键。在 DemoThread.cpp 中你还会看到 IMPLEMENT_DYNCREATE() 宏。DECLARE_xxx 和 IMPLEMENT_xxx 两宏总是成双成对地被使用。

启动 UI 线程

? FAQ 47：我如何设定 AfxBeginThread() 中的 pThreadClass 参数？

一旦 wizard 产生出线程类以及 DYNCREATE 宏，你就应该准备使用 AfxBeginThread() 了。下面是个范例：

```
CDemoThread* pThread = (CDemoThread*)AfxBeginThread(
                    RUNTIME_CLASS(CDemoThread)
                    );
```

其中作为参数的，是一个数据结构（译注：即 CRuntimeClass 对象），允许 MFC 在运行时产生该类的一个实体。就像 worker 线程的情况一样，当你以 AfxBeginThread() 产生一个 UI 线程时，你就让这个函数去配置数据结构并传回。你也可以在你的构造函数中设定 m_AutoDelete 为 FALSE。这一次会比较容易些，因为你已经派生了这一类，并且构造函数将被 AfxBeginThread() 调用。

一旦线程开始执行，MFC 将调用你的 InitInstance() 成员函数，并进入消息循环中。你的 CWinThread 派生类的消息映射表（message map）将被作为消息派送的指引。

与 MFC 对象共处

MFC 多线程程序有一个重大限制，会影响你所做的几乎每一件事情。MFC 各对象和 Win32 handles 之间的映射关系记录在线程局部存储（Thread Local

Storage，TLS）之中，因此，你没有办法把一个 MFC 对象从某线程手上交到另一线程手上，你也不能够在线程之间传递 MFC 对象指针。我所谓的 MFC 对象包括（但不限于）CWnd、CDC、CPen、CBrush、CFont、CBitmap、CPalette。这个限制的存在阻止了“为这些对象产生同步机制”的必要性，那会大大影响 MFC 的速度效率。

这个限制有几个分歧。如果两个线程都调用 CWnd::GetDlgItem() 以取得对话框中的一个控件（例如 edit），那么每个线程应该获得不同的指针——甚至即使两个线程的对象是同一个控件。如果面对一个指针，其所指对象并没有永久的 MFC 结构，那么当对此指针的一个索求行为出现时，MFC 往往会产生出一些临时性对象。例如，CWnd::GetDlgItem() 往往会在被索求一个指针（例如指向一个 CEdit* 或一个 CStatic*）时，产生一个临时性对象。这些对象会在下一次程序进入闲置循环（idle loop）时被清理掉。如果这些对象被许多线程共享，MFC 就没有能力预期它们的生命，也因此没有能力执行清理工作。因此，MFC 为每一个有需求的线程产生一个新对象。

这个限制（关于在线程之间交换对象）的意思是说，你不能够放一个指针（指向一个 CWnd）到结构之中，而该结构被一个 worker 线程使用。你也不能够把一个指向 CDialog 或 CView 的指针交给另一个线程。当你需要调用 view 或 document 中的一个成员函数，特别是像 UpdateAllViews() 这样的函数时，上述的限制很快便会恶化。

MFC 在许多地方检查“横跨线程之对象的使用情况”。任何时候，只要 MFC 对着对象调用 ASSERT_VALID，它便会检查对象是否保持在线程局部存储（TLS）中。如果你尝试调用一个像 CDocument::UpdateAllViews() 这样的函数，那么当程序运行时，CWnd::AssertValid 会产生一个 assertion。下面的注解来自于 assert 操作之下：

```
// Note: if either of the above asserts fire and you are
// writing a multithreaded application, it is likely that
// you have passed a C++ object from one thread to another
// and have used that object in a way that was not intended.
// (only simple inline wrapper functions should be used)
//
```

```
// In general, CWnd objects should be passed by HWND from
// one thread to another.  The receiving thread can wrap
// the HWND with a CWnd object by using CWnd::FromHandle.
//
// It is dangerous to pass C++ objects from one thread to
// another, unless the objects are designed to be used in
// such a manner.
```

译注 这一段说明出现在 MFC 源代码的 WINCORE.CPP 中的 CWnd::AssertValid() 成员函数内。

线程局部存储（TLS）的使用说明了以 AfxBeginThread() 在 MFC 程序中产生 UI 线程的重要性。如果你用的是 _beginthreadex() 或 CreateThread() 时，MFC 不会给你机会产生出用以维护其 handles 的必要结构。

在线程之间共享对象，这里倒是有一个不太方便的替代方案：不要放置 MFC 对象，改放对象的 handle。你可以利用 GetSafeHandle() 获得派生自 CGdiObject 的对象的 handle。这样的对象包括 CPen 和 CPalette 对象。你还可以利用 GetSafeHwnd() 获得派生自 CWnd 的对象的 handle，如 CDialog 对象。

当你把 handle 传递给新线程时，线程可以把该 handle 附着到一个新的 MFC 对象：使用 FromHandle() 可以产生一个临时对象，使用 Attach() 则可以产生一个永久对象。例如，你把一个 HDC 交给线程，你可以利用以下程序代码把这个 HDC 附着到一个永久的 CDC 对象上：

```
HDC hOriginalDC = //...
CDC dc;
dc.Attach(hOriginalDC);
```

而在退出之前，线程应该调用 Detach()：

```
dc.Detach();
```

如果线程只是想短暂地使用这个数值，它可以产生一个临时对象，像这样：

```
CDC *pDC = CDC::FromHandle(hOriginalDC);
```

并不是所有的 MFC 对象都很容易以该技术传递。CView 就是个例子。你可以轻易取得一个 view 的窗口 handle，并将它交给一个新线程，但最好这个新线程可以把此 handle 附着到一个 CWnd。因为并没有 CView::FromHandle() 函数可以产生一个临时性的 view，像镜子一样反映出原来的那一个。原来的 CView 结构不再可用，所以所有相关的 view 信息也都不再可用了。

回到 UpdateAllViews()，此函数运作时所使用的指针被埋藏在 document 中，无法改变。因此，其他线程没有任何方法可以调用此函数。唯一一个替代方案就是送出一个用户自定义消息，回到原线程中，告诉它更新它的 views。

Visual C++所附的 MFC 范例程序 MTGDI 中，示范了上述所讨论的技术。

MFC 的同步控制

在第 9 章我们已经看过了，如何利用 C++ 来简化对同步对象的锁定和解除锁定操作。MFC 有一些内建类，用来以相同方式处理同步对象。对于每一个标准的 Win32 同步对象（critical sections、events、mutexes、semaphores），MFC 分别提供了对应的类，将其机能封装起来：

对象名称	MFC 类名称
Critical section	CCriticalSection
Event	CEvent
Semaphore	CSemaphore
Mutex	CMutex

虽然每一种类型有不同的构造函数，能够善用其特定机能，但是另外也有一个一般的基础界面，通用于每一个类。上述每一个类都派生自 CSyncObject，后者为每一个类提供了一套最小量的一致性界面。换句话说，不论你所拥有的是哪一种对象，调用 Lock() 或 Unlock() 成员函数，都没问题。CSyncObject 也可以被交给任何一个期待“同步控制对象之 handle”的 Win32 函数。我想，比起尝试记忆“用来锁定和解除锁定每一种对象所需的各种 API 函数”，上面这样的机能会使这些对象比较容易使用。

注意：你必须写出 #include <afxmt.h> 才能够使用 MFC 同步控制对象。

利用 MFC 来写 String 类

让我们以 MFC 的同步控制类，重写第 9 章的 String 类，请看列表 10-5。我们称此类为 StringV3。其中唯一的改变就是以 CCriticalSection 取代 CriticalSection，并调用 Lock() 和 Unlock() 以取代 Enter() 和 Leave()。

警告：因为效率的考量，MFC 的字符串处理类 CString 并没有实现出可安全应用于线程的锁定操作。如果要让 CString 能够安全应用于多线程环境，你必须自己花一些功夫。

列表 10-5 以 MFC 的同步控制类，完成一个比较好的 String 类

```
#0001  class StringV3
#0002  {
#0003  public:
#0004      StringV3();
#0005      virtual ~StringV3();
#0006      virtual void Set(char *str);
#0007      int GetLength();
#0008  private:
#0009      CCriticalSection m_Sync;
#0010      char*            m_pData;
#0011  };
```

```
#0012
#0013 StringV3::StringV3()
#0014 {
#0015     // The constructor for m_Sync will have
#0016     // already been called automatically because
#0017     // it is a member variable.
#0018     m_pData = NULL;
#0019 }
#0020
#0021 StringV3::~StringV3()
#0022 {
#0023     // Use the "array delete" operator.
#0024     // Note: "delete" checks for NULL automatically.
#0025     m_Sync.Lock();
#0026     delete [] m_pData;
#0027     m_Sync.Unlock();
#0028     // the destructor for m_Sync will be
#0029     // called automatically.
#0030 }
#0031
#0032 void StringV3::Set(char *str)
#0033 {
#0034     m_Sync.Lock();
#0035     delete [] m_pData;
#0036     m_pData = new char[::strlen(str)+1];
#0037     ::strcpy(m_pData, str);
#0038     m_Sync.Unlock();
#0039 }
#0040
#0041 int StringV3::GetLength()
#0042 {
#0043     if (m_pData == NULL)
#0044         return 0;
#0045     m_Sync.Lock();
#0046     int len = ::strlen(m_pData);
#0047     m_Sync.Unlock();
#0048     return len;
#0049 }
```

MFC 封装了 Lock 操作

列表 10-5 的 StringV3 仍然蒙受我们在第 9 章的类中所遭遇的问题。我们必须非常小心，总是记得在返回之前调用 Unlock()。并且，记住，在 C++ 异常情况处理过程中，锁定操作将不会被清除。

MFC 提供两个类以修正这个问题。第一个是 CSingleLock，非常类似我们在第 9 章所写的 Lock 类。其用法显示于列表 10-6。第二个类是 CMultiLock，封装了 WaitForMultipleObjects() 机能。

列表 10-6 以 CSingleLock 完成一个比较好的 String 类

```
#0001  class StringV4
#0002  {
#0003  public:
#0004      StringV4();
#0005      virtual ~StringV4();
#0006      virtual void Set(char *str);
#0007      int GetLength();
#0008  private:
#0009      CSyncObject* m_pLockable;
#0010      char*        m_pData;
#0011  };
#0012
#0013  StringV4::StringV4()
#0014  {
#0015      m_pData = NULL;
#0016      m_pLockable = new CMutex;
#0017  }
#0018
#0019  StringV4::~StringV4()
#0020  {
#0021      delete [] m_pData;
#0022      delete m_pLockable;
#0023  }
#0024
#0025  void StringV4::Set(char *str)
#0026  {
```

```
#0027     CSingleLock localLock(m_pLockable, TRUE);
#0028     delete [] m_pData;
#0029     m_pData = NULL;  // In case new throws an exception
#0030     m_pData = new char[::strlen(str)+1];
#0031     ::strcpy(m_pData, str);
#0032  }
#0033
#0034  int StringV4::GetLength()
#0035  {
#0036     CSingleLock localLock(m_pLockable, TRUE);
#0037     if (m_pData == NULL)
#0038        return 0;
#0039     return ::strlen(m_pData);
#0040  }
```

MFC 同步控制的限制

虽然 MFC 的同步控制类对于基本结构的锁定很有帮助，但它们面对比较复杂的工作时却黯然失色。例如，它们没有支持“alertable”形式的 WaitForSingleObject() 和 WaitForMultipleObjects()，所以 overlapped I/O with APCs 就没有办法获得服务。另外，可能会持续锁定数秒钟之久的对象，通常会要求程序回到主消息循环，然后使用 MsgWaitForMultipleObjects()，然而 MFC 类却不允许这种行为。

MFC 对于 MsgWaitForMultipleObjects() 的支持

稍早在本书中我们曾经看过如何修改一个主消息循环，使它支持 MsgWaitForMultipleObjects()，如此一来程序可以有效率地等待消息或核心对象。这件事情在 MFC 程序中就有一点困难，因为是 MFC 自己在运行消息循环，不是你。

有两个可能的解决方案。第一个做法是产生另一个线程用来等待，当对象被激发时即送出一个消息给主线程。这个方法的优点是很容易实现，但是代价不小，因为 context switch 和消息的处理会常常发生，带来较高的额外负担（overhead）。

另一种做法就是改写 MFC 的消息循环。幸运的是消息循环驻停在虚函数之中，所以你可以改写之。它存在于 CWinThread::Run() 和 CWinThread::PumpMessage() 之中，源代码可从 MFC\SRC 的 THRDCORE.CPP 文件中获得。

这种做法的缺点就是，PumpMessage() 是个未公开函数，而 Run() 函数中用以处理闲置时间（idle time）的逻辑又有一点暧昧不明。如果这两个函数在未来的 MFC 版本中被改变了，你就得自己动手把你改变的那一部分移植到新版来。

那些函数的改写部分并没有显示于此，因为它非常依赖 MFC 版本，并且紧紧地和你的程序的数据结构绑在一块儿。

提要

这一章中你看到了如何产生 worker 线程和 UI 线程：要不就是单独调用 AfxBeginThread()，要不就使用更低阶的调用方式，也就是 CWinThread::CreateThread()。你也看到了如何从 CWinThread 中派生一个类，你可以为了产生 UI 线程而把它交给 AfxBeginThread()。最后，我们回到基础面，畅谈 MFC 如何在底层封装 Win32 同步对象。

GDI 与窗口管理

本章描述消息队列（message queue）如何和线程一起工作，并告诉你为什么你不应该为 MDI 程序中的每一个窗口准备一个线程。

你知道电视“脱口秀“中关于文字的心理测试节目有一段多么有趣的诗文模仿秀吗？医生说出一个字，病人就必须说出他心中想到的第一个联想字。试着对一个 Win16 程序员说出“Multithread”这个字看看，吓，你会发现十之八九的回答是 MDI（Multiple Document Interface）。

看来 MDI 似乎是个不错的起点，让你开始你的多线程程序设计之旅。每一个 MDI 窗口可视为一个迷你而独立的程序，于是你可以让某个 MDI 窗口的 GUI 线程打印其内容，而让另一个 MDI 窗口的 GUI 线程负责编辑工作。

在我们开始任何更深入的学习之前，关于“为每一个 MDI 窗口产生一个线程”这件事情，让我先给你一个警告：

不要这么做！

接下来的数节，我要解释 Windows 系统中的消息队列架构，为什么它会导出我的警告，以及如何设计你的程序，以获得预期的效果。

线程的消息队列

让我们先花一些时间来比较 Win16 和 Win32 两者的消息队列。在 Win16 中，所有窗口共享同一个消息队列。如果某个程序停止处理消息队列中的数据（也就是消息），那么所有的窗口就都会停止回应。这在 Windows 3.x 中是一个严重的问题，也是系统之所以会被锁死而不再做出反应的最大原因。

在 Win32 中，每一个线程有它自己专属的消息队列。这并不意味着每一个窗口有它自己的消息队列，因为一个线程可以产生许多窗口。如果一个线程停止回应，或是它忙于一段耗时的计算工作，那么由它所产生的窗口统统都会停止回应，但系统中的其他窗口还是继续正常运作。

以下是一个非常基本的规则，用来管理 Win32 中的线程、消息、窗口的互动：

所有传送给某一窗口之消息，将由产生该窗口之线程负责处理。

还记得吗，每一件发生于窗口上的事情都由“消息及其传递”来控制。这一事实在 MFC 程序以及使用 message cracker（译注：定义于 WINDOWSX.H 中的一组宏）的程序中不是很明显。举个例子，如果你利用 SetWindowText() 放一些新文字到 edit 控件中，控件之所以能够更新是因为送出一个 WM_SETTEXT 消息到 edit 控件的窗口函数中。

你对窗口所做的任何一件事情基本上都会被该窗口的窗口函数处理，并因此被产生该窗口的线程处理。这样的架构可能会在你的程序代码中产生一种反直觉的行为。举个例子，我打算使用一个独立的线程来填充数千笔数据到一个 listbox 中，我希望我能够在该线程中处理 listbox 的消息。

但是每当我加入一笔数据，消息却被送到我的主线程去。我的主线程产生这个 listbox 控件，所以它有义务处理这个控件的消息。每一个控件在对话框中都是个别的窗口，有自己的窗口函数。

从 Windows 的眼光来看，屏幕上到处充斥窗口，其数量远比你所看到的有着标题栏和下拉式菜单的所谓“标准窗口”还多得多。记住，屏幕上的每一个控件都是一个独立窗口，有着自己的窗口函数。这意味着单单一个对话框中就可能有许多个小窗口。Windows 95 标准的【File Open】对话框（图 11-1）就是这样。

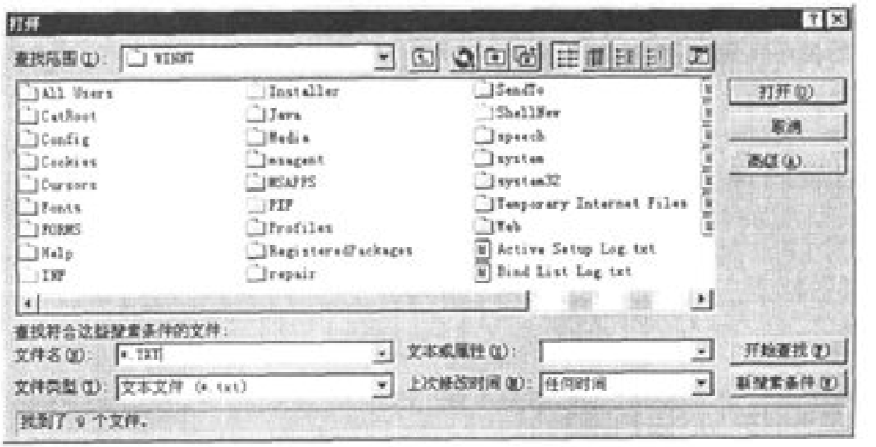


图 11-1 【File Open】对话框

利用 Spy++ 观察这个窗口，结果如图 11-2。的确，每一个控件都是一个窗口，每一个窗口有它自己的窗口函数，并且有一个线程用来处理所有的消息。

图 11-2 Spy++ 显示对话框的内容（各个控件）

消息如何周游列国

当你进入一个窗口函数时，如果观察 Windows NT 中的调用堆栈（call stack），通常总是可以追踪回到 WinMain()，那是你的程序的起始点（Windows 95 则因为 16-bit thunking（译注）的缘故，故事不是这样发展）。你会在调用堆栈(call stack)中看到某些函数位于 USER32.DLL 或 KERNEL32.DLL 之中。这些函数用来将消息派送（dispatch）到适当的窗口函数去，并调用该窗口函数。

译注 所谓 thunking，可以说是一种转换，把 16 位函数调用（包括其中的参数以及参数所代表或隐含的地址等等）转换为 32 位函数调用。或是反向转换。16 位转为 32 位称为 thunk up，32 位转为 16 位称为 thunk down，而其间的转换机制称为 thunking layer。

Windows 会自动计算出哪一个线程应该收到消息，以及线程应该如何被告知说有这么一个消息进来。一共有四种可能，列于下表。

	SendMessage()	PostMessage()
同一线程	直接调用窗口函数	把消息放在消息队列中然后立刻返回
不同的线程	切换到新线程中并调用窗口函数。在该窗口函数结束之前，SendMessage() 不会返回	PostMessage() 立刻返回，消息则被放到另一线程的消息队列中

译注 以下我将直接使用“send”和“post”一词，而不译为中文。“send”表示使用 SendMessage() 传送消息，“post”表示使用 PostMessage() 传送消息。

请注意，当你“send”一个消息给另一线程所掌握的窗口时，会发生什么事情？系统必须做一次 context switch，切换到另一线程去，调用该窗口函数，然后再做一次 context switch 切换回来。与一般的函数调用比起来，其间的额外负担毋宁说是太大了些。

这个巨大的额外负担正是为什么“每个 MDI 窗口不应该有一个线程”的头号原因。其间的牵连倒不是立刻就能浮现出来，毕竟一个 MDI 程序背后的观念是要让每一个窗口能够独立工作，所以似乎不会喜欢让各个窗口间常有沟通。问题出在主窗口！

时髦的 Windows 程序往往挂上一大堆琳琅满目的“勋章”：工具栏（toolbars）啦、状态栏（status bars）啦、调色板（palettes）啦。它们统

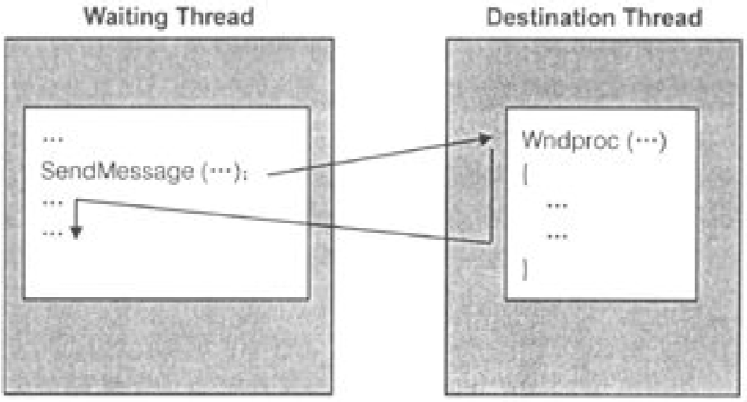
统必须被处理、被更新状态、被致能（enabled）或除能（disabled）。这些动作通常由目前作用中的子窗口负责，因为这些“勋章”的状态系由作用中的子窗口决定。在 MFC 中，这个行为已经被 OnUpdate() 完全承担下来。如果这些“勋章”存活于不同的线程中，那么更新其状态可能需要数百次 context switches。这样的结果将对效率带来严重的冲击。

睡眠之中还能工作的线程

有一件事情很重要，必须了解：当你“send”一个消息出去时，你的线程将暂时处于睡眠状态。这使得 SendMessage() 就像一个典型的函数调用操作。

虽然你的线程正在等待 SendMessage() 的返回，但它还是可以处理外界对其拥有之窗口的任何 SendMessage() 调用操作，甚至即使线程不处于主消息循环（其中有 GetMessage() 和 DispatchMessage() 操作）之中。如果 Windows 不这样设计，那么该线程所拥有的其他任何窗口就会停止反应，并且无法回答来自外界的任何 SendMessage() 操作。

译注 稍后会出现“waiting thread”和“destination thread”两个名词，所以我先以一张图解释这两个线程的身份。



虽然 waiting thread 可以处理来自 SendMessage() 的消息，但它不处理其他种类（像是位于消息队列中）的消息。如果 destination thread 开启一个对话框（译注：会将其父窗口除能的那种，也就是“modal dialog”），或是进入一个消息循环中，以至于没有返回至 waiting thread 的 SendMessage() 调用处，那么就可能引起一些问题。为了解决这问题，当 destination thread 调用以下任何函数，waiting thread 必须自动醒来：

```
DialogBox()
DialogBoxIndirect()
DialogBoxIndirectParam()
DialogBoxParam()
GetMessage()
MessageBox()
PeekMessage()
```

我们可以测试一个线程（译注：也就是前述的 waiting thread）是否正陷于“SendMessage() 未返回”的进退维谷情况中，而且可以明白地让调用端（译注：也就是前述的 waiting thread）醒来。这可能是有用的——如果你需要开始一个长时间的计算工作，而发动计算的那个人（线程）不需要知道其结果的话。

如果某个线程（译注：也就是前述的 destination thread）正在处理由其他线程“sent”过来的消息，所以你在该线程中调用 IsSendMessage()，会获得 TRUE。IsSendMessage() 不需要指定参数。

为了让调用端线程（译注：也就是前述的 waiting thread）能够继续工作，我们可以调用 ReplyMessage()。

```
BOOL ReplyMessage(
    LRESULT lResult
);
```

参数

lResult	这个值将会被传回给 SendMessage() 的调用端（译注：也就是前述的 waiting thread）。其实际意义因消息而异。

返回值

如果这个线程（译注：也就是前述的 destination thread）原本正在处理一个来自其他线程的消息，那么就传回 TRUE。否则传回 FALSE。

消息在线程间流动的陷阱

当 Windows 必须在线程之间“send”消息时，不论是否这些线程位于相同进程之中，总是有这种可能：destination thread 被锁死，以至于 waiting thread 永远醒不过来。

这个问题在你跨越进程“send”消息时特别突出，因为你没有办法保证传送对象（目标线程）的行为。Win32 提供了两个函数协助解决这个问题。

第一个函数是 SendMessageTimeout()。它允许你指定一个时间，时间终了后不管对方怎么样，SendMessageTimeout() 一定会返回。如果对方“挂”了，它也会自动返回。

第二个函数是 SendMessageCallback()。这个函数会立刻返回，但其参数之一，一个函数地址，应该被以 SendMessage() 的方式调用起来。

你可以在 Win32 Programmer's Reference 中找到这些函数的细节资料。

GUI 效率问题

当你分析为什么要为每一个 MDI 窗口产生一个线程时，最需要回答的一个问题就是：我将获得什么好处？这个问题的答案通常是：我可以让一个窗口（也就是一个线程）做计算而另一个窗口（另一个线程）仍然有活动力。

如果你试着执行一个这样的程序，你会发现那个负责计算工作的窗口看起来好像“死”了一样，它不再回应任何重绘消息，鼠标移到窗口领空也不再能够改变光标形状。菜单和工具栏也都不再能够更新状态。这是因为，线程正忙于计算工作，分身乏术，没有办法去执行窗口的消息循环并处理传进来的消息。整个程序还是有反应，但特定窗口却像“死”了一样。

其他的副作用还包括重绘效率的降低。甚至即使在一部多 CPU 的机器中，通常也只有一张显示卡，那会成为所有屏幕更新操作的瓶颈。不管你执行多少个线程，你就是没有办法将显示卡加速。因此，因为处理重绘操作而发生的所有 context switch，都只是导致重绘效率的降低而已。

我想我听到了你说：但是……但是……但是……。请忍耐一下，我将展示另一种架构，可以适当解决许多问题，这些问题是原本你希望以“每个 MDI 窗口配备一个线程”来解决的。

以 Worker 线程完成多线程版 MDI 程序

欲在一个 MDI 程序中有效率地使用多个线程，我的建议是，以一个线程处理所有的用户输入，以及所有用户界面的管理，然后使用一个以上的线程来负责诸如重绘、打印等工作。或许你还需要一些线程，用来处理你的磁盘 I/O 或网络 I/O。

不管你如何切割你的工作，很重要的一点是，你的主线程（负责主框架窗口）总是应该能够有所回应，不会陷入长时间计算的泥淖中。稍后你会看到一个 CANCEL 范例程序，展示“在一个线程中以消息循环搭配计算工作”所引发的问题。

涂色（Rendering）

理想中你的主线程应该做掉所有绘图工作，或是实际去搬动屏幕上的数据。你还是可以在这个进程中使用多线程：使用 worker 线程来计算什么应该显示，然后再通知主线程去画。做法之一是，让每一个 MDI 窗口拥有一个后台线程，负责涂色。每个线程应该在它自己的“memory device context”（译注）中涂色，这么一来每个线程就非常独立。由于这些涂色线程是在一个非真实屏幕的区域中涂色，所以它们不会和主线程冲突。一旦后台线程完成其工作，就可以通知主线程，把 memory device context 中的内容搬到真正的屏幕上。

译注 “device context”（DC）是一大块数据结构，用来表示一个输出设备。它可以代表打印机（printer DC），可以代表屏幕（screen DC），可以代表 metafile（metafile DC），可以代表窗口（display DC），也可以代表一块模拟屏幕的内存（memory DC）。

微软的某些范例程序建议另外使用一个分离的线程来绘图。除了某些例外情况，我并不同意这个建议。有两个理由：第一，如我刚刚所说，只有一张绘图卡，而它可能就是瓶颈所在。第二，使用一个分离的线程来负责窗口绘图工作，可能会引发一些痛苦而棘手的问题。例如，你必须清楚知道什么是（以及什么不是）被显示于屏幕上，如此一来，用户才能够在绘图时点选屏幕上的东西。

某些情况下，“搬移数据到屏幕上”所能动用的时间，将因为涂色的困难度（大量的计算）而导致萎缩。包括光迹追踪（ray tracing）、电脑辅助设计（CAD）、分形几何的产生（fractal generation）等等，都是这样。这种情况下，如果能找出什么方法可以让多个线程共同分担计算工作，你就是个大赢家。涂色工作需要很小心地切割，才有办法让多个线程分担计算。

让所有的线程在同一张 DC 上作画，是个坏主意。因为每一个线程无可避免需要自己的画笔、画刷或坐标原点。以一个同步机制来保护 DC 是可能的，但是这会降低因多线程而带来的效率利益。

一个可能的解决方法就是把涂色区域切割为许多块小四方形，每一个线程负责一块小四方形。光迹追踪和分形几何都可以这么做，因为每一个像素（pixel）都是独立的。

对于一个 CAD 软件就行不通了。通常 CAD 软件必须把一整个对象完整画出来。比较好的做法是让一个线程产生一个中间过程的数据（intermediate data stream），再让另一个线程解析它并画到 DC 上。运用这个技术，线程就可以把它们的结果排队等待“绘图线程”去取，不会彼此妨碍。

打印（Printing）

虽然绘图和打印都是使用 GDI，但同时做这两件事情是很安全的，因为它们使用完全不同的 device context（DC）。它们可能会共享程序的一些数据结构。如果是这样，就有必要使用一个同步控制机制来保护那些数据。不过，如果在同一时间，不同的线程使用不同的 GDI 对象，是绝对没有问题的。

多个上层窗口（Top Level Windows）如何是好？

“使用不同的线程以控制不同的窗口”所派生的限制问题，适用于 MDI、对话框，以及其他父子窗口关系上面。但相反地，有时候面对多个 top-level 窗口，你会使用另一种程序模型。这也是每次你在桌面上开启一个新的文件夹时，Explorer（资源管理器）的行为。甚至即使有单一程序负责控制所有窗口，在用户的心目中，它们是完全分离的。一个线程负责一个 top-level 窗口很是合理，因为它为每一个窗口提供有自己的消息循环，以及对键盘输入焦点（keyboard focus）的处理。

这样的模式也提供了其他利益，特别是在网络环境中。大部分程序在【File Open】对话框进行过程中是没有其他反应的，因为可能会有许多数据需要载入，而找出适当的程序地点安插一个 PeekMessage() 循环，恐怕十分困难。

然而，如果在多个 top-level 窗口的情况下，当一个窗口正在开启一个文件时，就可以不必担心是否其他窗口会不动如山。以 Explorer 为例，它可能花一分钟或两分钟，以 Remote Access Services 来连接一个网络驱动器。当这件事还在进行之中时，其他的 Explorer 窗口还是有反应。不过，如果你进行某个操作牵扯到桌面窗口（desktop），桌面窗口会冻住，直到该操作完成。

线程之间的通讯

花了本书第一部分的那么多篇幅介绍所谓同步机制和 race condition，现在让我们看看线程间的通讯技术，并试着解决（或绕过）一些问题。

线程常常需要将数据传递给另一个线程。Worker 线程可能需要告诉别人说它的工作完成了，GUI 线程则可能需要交给 worker 线程一件新工作。

PostThreadMessage() 的操作就像 PostMessage() 一样，但是它的参数之一不是窗口 handle，而是线程 ID。当然，收受端线程必须有消息队列，不过至少这个队列是由操作系统管理，你不必操心。

```
BOOL PostThreadMessage(
        DWORD idThread,
        UINT Msg,
        WPARAM wParam,
        LPARAM lParam
    );
```

参数

idThread	线程 ID（不是 handle）。这个 ID 可由 GetCurrentThreadId() 或 CreateThread() 获得。
Msg	消息识别代码
wParam	消息的 wParam
lParam	消息的 lParam

返回值

如果消息被成功地“post”出去，函数传回 TRUE。如果传回的是 FALSE，可利用 GetLastError() 获得失败原因。

PostThreadMessage() 把消息“post”给一个线程，而非一个窗口。如果收受端线程尝试获取目标窗口的 handle，它会得到 NULL。所以，收受端线程的消息循环应该有特殊的处理方式，以处理窗口函数之外的消息。

以消息当做通讯方式，比起标准技术如使用全局变量等，有很大的好处。如果目标线程很忙碌，则负责“post”的那个线程不会因此停滞下来。但如果你设立一个标记，告诉大家数据已经准备好了，欢迎大家来拿，你可能就得等待，直到目标线程有时间收下这张数据收据。此外，我们还能够“post”好多个消息，不需等待目标线程的回应。

如果对象是同一进程中的线程，你可以自定义消息，如 WM_APP + 0x100 等等，并配置一块结构，放置你想传递的数据，然后把结构指针当做 lParam。收受端应该在处理完消息后负责释放内存。

重要！ 如果你使用 PostThreadMessage() 在不同进程的线程之间传递消息，你必须使用 WM_COPYDATA 消息，这样一来数据才能够从一个地址空间中被映射到另一个地址空间。这个技术在第 13 章讨论。

NT 的影子线程（shadow thread）

在 Windows NT 中，另有一个原因使我们不要为每一个 MDI 窗口产生一个线程。在 Windows NT 中，操作系统会为每一个拥有消息队列的应用程序线程，产生另一个操作系统线程。这个所谓的影子线程用来服务你对 GDI 的调用。因此，每当你产生一个 UI 线程，其实是产生两个线程。

这么做的原因是为了使其他程序没有可能破坏你的程序（不管是有意或无心）。如果你交出不合理的值，或是绘图时发生什么错误，系统线程可能会崩溃掉，但其他影子线程不受影响，还是安全的。

关于“Cancel”对话框

有时候我们不十分清楚什么时候为一个用户界面使用多线程才是适当的。我曾经用过一个程序，它有一个 data import filter，需要好几分钟才能运行完。理所当然，用户希望看到所谓的进度对话框，其中有一个【Cancel】钮，以备在他们打算放弃等待时按下去。用户也期望，如果切换到其他程序然后又回来时，进度对话框仍然能够适时地重画。

我的第一版作品是以偶尔调用 PeekMessage() 来实现【Cancel】机能。然后我发现 import filter 比以前慢三倍。我想现在是使用第二个线程的理想情况。让对话框在一个线程中运行，主线程则负责 import filter。如

果用户按下【Cancel】，我就在主线程中设立一个标记，这使我能够常常检查其值。我不需要调用 PeekMessage()，执行对话框的那个线程将可以很有效率地等待。

我的错误是多么大呀！

书附盘片中的 CANCEL 程序示范了我所犯的各种错误。CANCEL 是一个 MFC 程序，丢出一个 modeless 对话框，希望当主线程忙碌时，它还是有作用。

在 CANCEL 程序中我启动一个新线程，然后让新线程产生那个对话框。在启动该线程之后，主线程进入一个紧绷的循环中，不让它一下子回到主消息循环。用以模拟 import filter。

CANCEL 尝试以三种做法解决问题，条列于下。你可以选择【View】选单中的各个项目，试试每一种做法。

1. 使用一个 MFC UI 线程，带出一个 CDialog。
2. 使用 CreateThread() 和 CreateDialog()，让主窗口成为对话框的父窗口。
3. 使用 CreateThread() 和 CreateDialog()，让对话框无父窗口。

第一种方法，对话框将不会出现，直到主线程中的循环结束为止。你可以在 CANCEL 程序选择【View】菜单中的【Launch MFC Dialog】，亲自体会一下。在看过 SPY 的结果之后，我明白有一些不为人知的 MFC 内部消息，所以我决定使用单纯的 Win32 API 函数再试一遍。

你可以在 CANCEL 程序选择【View】菜单中的【Launch Win32 Dialog】，尝试一下第二种方法。这个技术还是没有办法有效运作，非得等主线程开始回到主消息循环之后，对话框才会出现。

再次借助于 SPY，我看到在对话框和其父窗口之间，有许多消息。为了产生一个新窗口，系统送给父窗口一大堆消息，其中包括“deactivate”消息。如果父窗口忙碌于其他事情，没有理会这些消息，那么子窗口（本例为对话框）就会被阻塞住而停止执行。一旦父窗口开始处理消息，子窗口才活了起来——尽管它其

实是在一个完全独立的线程中运行。

我尝试对对话框的启动给予同步控制，使主线程能够在新线程产生之后的一小段时间内继续处理消息。然而问题只不过是延缓发生而已：在用户企图移动对话框或操作某个控件时发生。

以这些经验为基础，于是我尝试让对话框的父窗口为 NULL。你可以在 CANCEL 程序选择【View】菜单中的【Launch Win32 (no parent)】。于是对话框会立刻出现并且运作良好。但是却有另外的问题出现。

没有了父窗口，于是对话框在 taskbar 上就有了自己的图标（icon），而且它不会随着主窗口的最小化而最小化。这是标准行为，与多线程无关。虽然我们可以想办法绕道解决这些问题，但那又是另一个大题目了。由于主线程十分忙碌，没有机会处理重绘的要求，所以窗口的显像会有问题。

裁决

所有这些问题或许都可以绕道解决，但是很明显我正在强迫系统做一些它并没有被设计那么做的事情。我愈是想办法挖东墙补西墙，整个机制就愈是变得脆而虚弱。最后我终于回到原路，把 PeekMessage() 循环最优化，使我能够分头控制主线程和 modeless 对话框。

这个例子提醒我不适当使用线程所带来的危险。不幸的是，在“适当使用”和“不适当使用”之间，界线模糊。

锁住 GDI 对象

使用 GDI 对象绘制窗口，几乎是任何程序中的关键部分。这意味着使用 GDI 对象如 device contexts（DCs）、笔、刷等等。

GDI 对象被每一个进程拥有，而被每一个线程锁定。如果一个线程正在使用一个 GDI 对象，它将会被锁住，其他线程没有办法再使用它。与大部分其他的 Windows 资源不同，GDI 对象的取用并不是连续性的：线程不会排队等待一个忙碌的 GDI 对象，函数调用直接就传回失败。更糟的是，如果有一个线程删除了一个 DC，它就那么走了，即使有另一个线程正在使用它。如果你选择一支新笔放进某一线程正在使用的 DC 之中，那么第二个线程也会获得这支新笔。这样的结果并不是我们所期望的。

如果你使用自己的同步机制来保护它们，GDI 对象就有可能被共享。然而，这样的机能并没有在系统核心中实现，因为它太没有效率了。你应该谨慎地遵循核心设计者的领导，并找出其他方法来解决你的应用程序的问题。

我强烈反对在不同线程之间共享 GDI 对象。微软的技术文件对此的讨论非常少。有些范例程序一涉及这个主题，只会说“别做它”！

提要

这一章告诉你，一个窗口总是被一个特定的线程拥有，该线程将负责处理所有流至该窗口的消息。你也看到了，主线程应该是唯一和用户界面有所交谈的线程，所有耗时的工作都应该移交给 worker 线程来做。最后，你还看到了，当一个线程正使用 GDI 对象时，它们将被锁住。在线程之间共享 GDI 对象是相当困难的，额外的负担也过重。

调　试

Debugging

这一章描述多线程程序的各种撰写、调试、测试技术。

对一个多线程程序调试，如果没有经过小心的策划，很可能就像抓蝴蝶一样。你可以看到问题，接近它，但是当你几乎已经就要用手抓着它时，它飞走了，然后在另一个地方出现。我们曾经在第 2 章的 NUMBERS 范例程序中看到类似的行为：一个“输出重定向（至文件）”的细微改变，竟然带来戏剧性的变化。

如果你曾经尝试在 Windows 中对鼠标消息和键盘消息的处理函数调试，你所遭遇的问题将非常类似在多线程程序中所看到的种种问题。例如，在绘图软件中一个常见的功能是允许用户拉一个橡皮圈四方形，以便能够将一组形状对象包围起来。这通常是藉由处理鼠标左键的按下、放开、移动而完成。如果你尝试对“左键按下”或“鼠标移动”设立中断点，那么当你在中断点之后继续执行程序，调试器和系统会被混淆，因为鼠标按键处在错误的状态下。

这一章将讨论的调试动作，包括改变你的程序代码，以及使用调试工具两种。我们将以 Visual C++ 4.x 集成环境中的调试器为例，举例说明这些工具的应用。

使用 Windows NT

第一点并且也是很重要的一点是：如果你要开发多线程程序，请你使用 Windows NT。在开发本书所需的范例程序过程中，我花 95% 的时间在 Windows NT，5% 的时间在 Windows 95。尽管只有这么少的时间放在 Windows 95 身上，它却当机数次。Visual C++ 调试器在 Windows 95 之中似乎特别脆弱——当数个线程同时执行而调试器又企图中断（breakpoint）它们时。至于 Windows NT 则像岩石一样坚硬，不管我丢什么东西过去！

当你在 Windows NT 上开发程序而市场目标摆在 Windows 95 时，你必须注意哪一个 API 函数在 Windows 95 中有支持，哪一个没有。Windows 95 中有一些明显的 Win32 遗漏，像是 I/O completion ports。另有一些 API 函数在 Windows 95 中只是部分被支持。

有计划地对付错误

对应用程序展开适当的调试，应该是远在你使用调试器之前就开始了。程序总难免有错，尤其对大部分无心的程序员而言。因此，如果你知道难免会有“臭虫”，何不做点计划以确保将来一定可以抓到它们呢？

如果说 Win32 有一件好事情，我想“Win32 能够产生错误代码”应该可以上榜。大部分 Win32 函数如果发生错误，都会传回一个错误代码，调用 GetLastError() 可获得之。因此，你所做的每一件事都有必要做合理性检验。甚至即使“绝对安全”的动作如 GDI 的 SelectObject()，都有可能在多线程程序中失败。所以我的结论是：检查每一件事。

从第 2 章开始我便使用 MTVERIFY 宏，它会检查是否 Win32 函数传回 FALSE，并且在那种情况发生时调用 GetLastError()，然后显示一个字符串，代表错误信息。这个宏为我节省许多小时的调试时间。当我把错误的参数交给

CreateThread()时，它会捕捉到“Invalid Parameter”错误。当我使用错误的变量时，它会捕捉到“Invalid Handle”错误。它甚至可以捕捉到更复杂的错误，如“在一个 event 对象已经被摧毁之后才调用 WaitForSingleObject()”之类。

MFC 和 C runtime library 都有 ASSERT 调试措施。ASSERT 背后的观念就是安全和速度之间的一个交换。这很类似 Windows 3.1 和 Windows 95 中的“Debug Build”，以及 Windows NT 中的“Checked Build”。如果你正在执行一个“Debug Build”程序（Visual C++ 所定义的），“assertion checking”就会全部打开。当然程序也就因此运行得比较慢一些。但是之所以要制造“Debug Build”为的就是调试，所以一切都在理解范围内。一旦你比较有信心了，就改用“Release Build”，所有的“assertion checking”就会全部关闭，当然程序也就因此运行得比较快一些。

在每一个你的假设之处做检验工作。进入一个函数时，确认所有状态。不要只是检查指针是否合法；如果可能，检查一下指针所指的结构中的数据是否一致。

Bench Testing

如果你需要设计一个新的算法、一个新的 C++类、一个新的模块，或用以实现出一组新机能的任何东西，请建立之并以它自己测试它自己。不要把它放到已经稳定的程序代码中，直到你能够让新代码顺利运作为止。我们的目标是，首先，能够在不考虑多线程环境的情况下测试这些机能。如果你有逻辑问题，也有 race condition，那么在一个多线程环境下很难想象什么会真正出错。如果你把机能隔离出来，确定逻辑是正确的，剩下的问题就很可能是与线程有关了。

这样的开发型态称为“Bench Testing”。如果汽车工人修理一辆汽车上的交流发电机，他会先在工作台上以测试工具测试之，一切正常后才置回车内。如果汽车还是没有办法正常地跑，至少他排除了一个可能，可以把注意力放到其他地方。

如果可能，我甚至建议对所有核心的多线程算法都进行“Bench Testing”。例如，你在第 7 章看到了 Readers/Writers 算法，我开发那个程序，并使用一个小小的测试程序。这个测试程序可以很容易修改以改变测试环境。在 Readers/Writers 的实现过程中，有太多太多微妙复杂的问题，如果我企图直接在一个大程序中开发它，可能我到现在还在调试。

线程对话框

? FAQ 48：我如何对一个特定的线程调试？

Visual C++ 调试器可以支持“多线”情况。例如，当一个线程遭遇一个断点时，调试器会自动切换到该线程的 context 去。这所产生的一些行为，可能会令第一次见到的你感到惊讶。如果你在一个“许多线程都会调用的函数”内设立断点，每当一个线程遭遇此断点，调试器便会在线程之间循环切换。

我遇过这种情况数次。我设立一个断点，并且尝试执行下一步，结果却再次回到断点。我想调试器大概正在切换线程。这让我十分困惑。

有一些方法可以避开这些问题。方法之一是挂起所有的线程——你感兴趣的那个除外。在调试器中，如果你打开【Debug】菜单并选择【Threads】，你就会获得图 12-1 所示的画面。

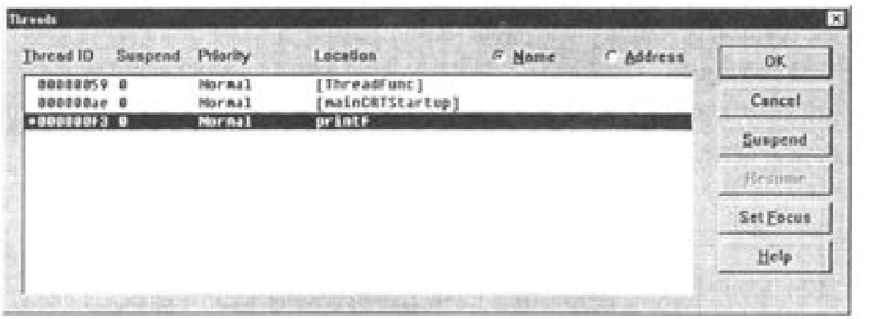


图 12-1 Visual C++ 4.0 的线程对话框

在此对话框中你可以挂起（suspend）或重新执行（resume）任何线程，并且将调试器的焦点设定在某个线程身上。为解决前述问题，我应该选择图 12-1 中的两个线程，并按下【Suspend】按钮，然后再选择第三个线程，按下【Set Focus】按钮。

运转记录（Logging）

所谓运转记录（Logging），是指让程序关键部分显示其活动的一种方法。运转记录是有用的，因为它允许你把焦点放在程序某些特定部位的运作顺序，甚至即使它们其实分隔得远远地。我将在第 14 章使用这项技术，在那里我将让 printf() 插遍整份程序代码，于是我就可以看到 DLL 的初始化过程。下面就是其输出模样：

程序输出：

```
DLL:   Process attach (tid = 180)
       Thread launched
DLL:   Thread attach (tid = 54)
       Thread running
DLL:   TheFunction() called
DLL:   Thread detach (tid = 54)
DLL:   Process detach (tid = 180)
```

这结果并不太令人感动。但是考虑到“在所有那些地方设立断点，并动手抄下所发生的结果”的困难度，你会发现，如果要了解某一段代码，运转记录可以带来很大的便利。

记住，即使在一个 GUI 程序中你也可以拥有一个 console 窗口。这意味你可以在 GUI 程序中使用 printf() 和 puts()。不要尝试使用列表框（listbox）或多行文字编辑框（multiline edit）等控件，因为它们需要仰赖消息的传递，以及适当的重绘。而这两种行为在程序已经出错的情况下是很难保

证的。Console 窗口由系统的设备驱动程序负责，即使你的程序当掉或在调试器中停止，console 窗口仍然有反应。

把运转记录输往 stdout 还有另一个利益：很容易导向到文件。把运转记录导到文件中，可以降低运转记录所花费的时间，同时也降低运转记录带给应用程序的冲击。你可以在命令行中使用 I/O 重定向，决定把输出导到何处。一旦你手上有了一些运转记录文件，你就可以利用 WINDIFF.EXE 或其他类似工具程序分析其间的差异。

C runtime library 的多线程版本使用 mutexes 以确保一次只有一个线程写东西到 stdout 中。从调试角度来看，这有好有坏。好处是，这增加了“信息以其发生之真正次序，被打印出来”的可能性。坏处是，线程必须等待，因此，时间因素可能相当程度地有所变化。请注意，在 stdout 中记录运转过程，并不能够保证你所看到的输出次序都真正是执行时的次序：在“程序的某个动作发生”和“运转记录被打印到文件”之间，有许多机会可能发生 context switch。

请记住，printf() 有一个不算轻的额外负担（overhead），因为它必须解析其“格式字符串”。为了降低这个负担，输出单纯的文字时，不妨以 puts() 取而代之。

MFC 还有 TRACE 宏，可以把文字输出到一个调试窗口中——通常是 Visual C++ 的“Output”窗口中的“Debug”附页。这个机制比较慢，而且“Output”窗口常常远远落在程序的执行之后。我避免使用这种方法来对多线程程序调试，因为它所带来的额外负担以及效率冲击不是十分清楚。

内存记号（Memory Trails）

甚至即使你把输出导向到文件中，运转记录花费的时间所带来的冲击，仍然足够改变程序的执行结果．如果要改善这种情况，我必须回到一个我所谓的“Memory Trails”（内存记号）的低阶技术中．

为了使用 memory trail，你必须产生一个全局缓冲区，以及一个指向该缓冲区的全局指针．例如：

```
char gMemTrail[16384];
char *pMemTrail = gMemTrail;
```

每当想印出某些东西到屏幕上或文件中，你就写一个记号到 memory trail 中．例如：

```
*pMemTrail++ = 'D';
```

你的程序中的每一个追踪点都应该写出一个不同的记号．不论什么时候你想要，或是在程序当掉之后，你可以利用调试器看看 memory trail 的内容，分析其间到底发生了什么事．它当然不像文字那么容易阅读，但总比乱猜的好．

有一个鲜为人知的调试器特性，可以帮助你观看这个缓冲区内容．当程序停在断点上时，选择 Visual C++ 的【View/Memory】，打开“内存窗口”．

一旦窗口开启，双击程序中的全局变量，使它成为高亮度，然后把它拖拉到“内存窗口”中，于是你就会看到数据以字节形式表现出来．如果要切换为文字形式，请选按【Tools/Options】菜单项，并选择其中的【Debug】附页，然后在“内存窗口”中把【Format】设定为 ASCII．现在每当你的程序在调试器中停下来，“内存窗口”就会把有变化的数据高亮度起来．这样就可以很方便地观察 memory trail 中哪些数据被加了进来．

如果你需要储存更多信息，你可以使用一个 DWORDs 数组，放置整个观察集合．例如：

```
*pBuf++ = (5 << 16 | some_useful_value);
```

这可以把 5 存放到较高字，把另一个数值存放在较低字。“内存窗口”可以被设定为 Long Hex 模式，于是数据可以比较容易被解读。

Memory trails 可以大量降低彼此干扰的可能性，因为它既没有用到系统函数，也没有用到同步机制。然而也由于它不是同步操作，当两个线程同时写入一笔数据，memory trails 还是有可能遗失数据。如果你有许多线程，而其中有许多断点，这可能会造成严重的问题。

硬件调试寄存器（Hardware Debug Registers）

重要！ 这一节只适用于 Intel 机器。PowerPC 以及大部分能够执行 Windows NT 的其他 CPUs 并没有所谓的硬件调试寄存器。

你曾经身处这种情况之下吗？内存中的某个位置已经发生错误，但是你无法知道它是从哪里发生的。这种问题最常因为指针的迷失而引起，但很难抓出元凶。Intel CPU 自从 386 之后便拥有一个十分奇妙的性质，称为硬件调试寄存器，可以帮助你抓出像这样的问题。

调试器（软件）有能力操控调试寄存器，让它观察最多四个 location。调试寄存器监视 CPU 的操作，如果 CPU 改变了任何一个 location，程序就会立刻在调试器中停下来，这时候你就可以看看是什么指令引起了改变。最棒的是，调试寄存器的使用不会影响程序的速度。

重要！ 硬件调试寄存器并不能够在 Windows 95 中确实而可靠地运作，甚至虽然 Visual C++ 尝试使用它们。在 Windows 95 中设定“数据断点”（Data breakpoint），可能会引起某个程序（不一定是你的调试对象）突然当掉，请看 Microsoft Developer Network（MSDN）光盘中的 Microsoft Knowledge Base，第 Q137199 号文章。

调试寄存器的使用有一点点晦暗不明。如果你需要监视一个全局变量或是一个全局数组中的某个元素，请选择【Edit/Breakpoints】菜单项，然后选择【Data】附页。在【Expression】编辑栏位中输入变量名称，例如：

```
ghThreadHandle
gDataArray[12]
```

完成后的对话框画面显示于图 12-2。

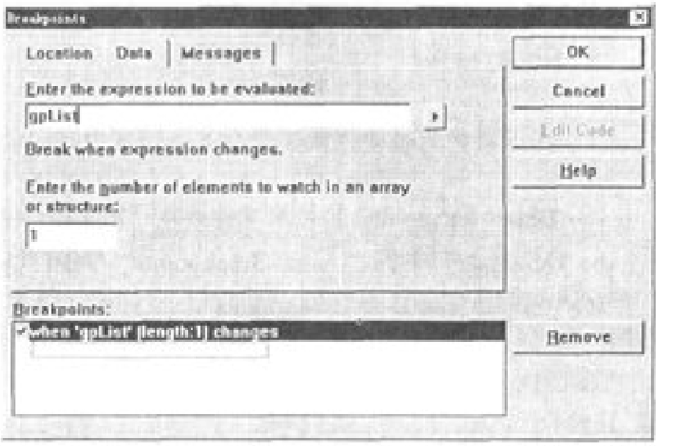


图 12-2 Breakpoints 对话框中的【Data】附页

你可以利用【Breakpoints】对话框中的【Data】附页监视许多其他种类的语句（expression），大部分的它们会引起调试器使用单步（single step），并且在软件中评估其结果。这么做大约会使你的程序慢 100 倍。然而如果你需要监视一个放在堆栈中的变量，这就是办法。

你必须找出变量的地址。在调试器中，选按变量，然后选按【Debug/Quick Watch】，会出现一个对话框，告诉你所选择的变量的现值。

在【QuickWatch】对话框的【Expression】编辑栏位中，在变量名称之前放一个“&”符号，调试器就会取该变量的地址。例如：

```
&slot
```

这会令调试器告诉你变量地址，甚至即使该变量在堆栈之中。我的测试结果的值为 0x0063fdd4。最前面的 0x 至为重要，它告诉调试器说这是 16 进制值。

现在回到 Breakpoints 对话框中的【Data】附页。输入你所记下来的地址（包括最前面的“0x”）。调试器会把这解释为一个命令，表示要观看已知地址上的字节内容。如果要看四个字节，请使用强制转换类型，像这样：

```
DW(0x0063fdd4)
```

调试器将总是使用调试寄存器来观看内存中的内容。

Data breakpoints 的技术文件放在 "Visual C++ User' Guide" 中的 "Using the Debugger" 中的 "Using Breakpoints" 中的 "Using the Breakpoints Dialog Box" 中的 "The Data Breakpoints"。

科学方法

当预防手段全部尝试过了后，程序还是当掉，而且没有什么明显理由，怎么办？为了成功地将一个多线程程序的“臭虫”除尽，你必须具备三种素养：

- 决心
- 耐心
- 创造力

我们在本书一开始就看到了．多线程程序并不总是可以预期的．理想上，当一个多线程程序、对象、或函数被完全调试之后，它应该变得可预期．但是由于你还在读这一章，我猜想你尚未到那种境界。

我在上面所说的那些素养或许也可以用来描述科学家．科学家研究这个世界，设法了解他们所看到的，并且预测是什么因素造成他们之所见．对一个多线程程序调试，情况十分类似．

在多线程程序中，“时间”是唯一考量．一般的调试技术，如设立断点、加上 *printf()*，或放置一个对话框等等，都可能会给你的调试对象带来巨大的冲击，使得你所希望找出来的时间问题被模糊了．时间问题也可能因为系统的活动（其他程序执行得如何？CPU 忙不忙？）、网络的活动（Ethernet 的饱和度如何？）而被影响．你的目标是尽可能稳定环境，使结果得以预期．

你或许常常面对很有次序的、很有逻辑的程序进行调试．但多线程程序既无次序亦无章法，所以需要不同的方法．最容易的方法就是像你管理一个科学实验那样．科学方法摘要如下：

1. 观察
2. 预测
3. 测试

通常调试器只能给你大约印象，告诉你发生了什么事情．剩下的东西都是需要用猜的．如果你看到什么发生了，但不知道它如何发生，你必须开始预测（或说猜想）其原因．尝试你的预测，一次一个就好．改变程序中的一个行为，看看会发生什么事．行为还是一样吗？有什么变化吗？“臭虫”拿掉了吗？

如果行为没有改变，那么就回复原样，再尝试另一个改变．你必须把你的改变尽可能区隔出来，使得程序行为一旦有所变化，你能够确实知道是什么因素造成的．

如果行为改变了，或许你向着问题之所在跨近了一步．不要忽略你所看到的所有“臭虫”．或许还有一些处于潜伏期呢．一个问题之产生可能是因为数个“臭虫”

凑起来爆发的。随时注意你的改变，以便必要时可以回复原状。再也没有什么比“10 分钟前做了一件重要的事情，但我现在忘了”更令人扼腕、挫败。

如果“臭虫”移除了，你必须确定你完全了解问题之所在，并且看看相关的代码，确定问题不会再发生。多线程程序的“臭虫”是不可能自己离开的，它们可能会潜伏起来，半夜 11 点又虫迹乍现。

我要说的是，对多线程程序调试，必须在一种很有控制、很有条理的环境下。没有什么比你的直觉和脑力更有可能解决这些问题。一点点想法可能就要花掉数小时的调试时间。

测试

由于多线程程序对于时间是如此地敏感，因而把它们放在各种环境中测试就成了一件非常重要的事情。较快的 CPU 和较慢的 CPU 都会导致新的 race condition 的可能。把你的程序放在你所能获得的最快电脑上执行，再把它放到像 486/25 这样的慢速机器上执行。如果你的程序有许多 I/O，那么请在活动很少的磁盘或 Ethernet 上测试一遍，再在活动很多的磁盘或 Ethernet 上测试一遍。

你的程序可能遭遇各式各样的效率变化，最佳例子就是当使用 Remote Access Services（RAS）的时候。如果你正通过 Ethernet 和一个服务器交谈，则大部分的请求会在数百个毫秒（ms）内被服务完毕。但是当你通过一个调制解调器来连接时，服务可能需要数秒之久。程序中的时间因素有巨大的变化。

另一个重要的注意事项是：测试你的 Release build 和你的 Debug build。一如我们在第 7 章所见，最佳化可能会对一个多线程程序的逻辑带来破坏。

理想情况下你的 Release build 应该有两个版本：Asserted 版和 Unasserted 版。在 Visual C++ 的预设情况下，Release build 程序将不会引起任何 ASSERT 检验。但由于最优化所引起的一些问题，我们值得在一个最优化版本中保留 assertion 检验。记住，ASSERTs 会改变程序的时间因素，所以 Release build 的两个版本也都需要测试。

提要

我希望这一章不至于让你决定不再使用多线程，但是一个多线程程序的调试困难度的确不应该被过分低估。这些困难度使得在使用多线程之前先加强其成本效益比的分析，变得更为重要。对程序员做适当的训练也非常地重要。

在这一章中你看到了如何使用运转记录（logging）以及内存记号（memory trails）来监视你的程序的所作所为，并且带来最少的冲击。你也看到了调试器中的某些工具可以帮助你追踪问题，并且看到了测试多线程程序所需的一些基本信息。

进程之间的通讯

Interprocess Communication

本章介绍可用于分属不同进程的各个线程之间的通讯方法，以及当你使用这些方法时会遭遇的一些问题。

到目前为止，我们所谈论的都是仅限于在同一个进程之内的线程。当线程分属于不同进程，也就是分驻在不同的地址空间时，有些特别的主题就必须拿出来探讨一下。

虽然我们已经讨论过，使用多重线程比使用多重进程的好处，但还是可能因为某些理由使你决定使用多重进程。我们之所以认为线程有很多优点，主要因为它“重量很轻”，对于系统资源的冲击最小，能够快速启动和结束。多重线程可以共享同一个地址空间和核心对象，所以很容易在它们之间搬移数据。

相反地，进程的设计，在彼此防护上就严谨多了。如果一个进程死亡，系统中的其他进程还是可以继续执行。对于某些应用而言，这样的健壮性（robustness）或许值得花费比较多的额外负担（overhead）。因为如果多个线程在同一个进程中运行，那么一个误入歧途的线程就可能把整个进程给毁了。

在 4.0 版之前，Windows NT 的图形子系统（graphics subsystem）是一个分离的进程。所以即使图形子系统当掉了，系统照样运行。这是个好例子，说明进程如何用来作为“隔板”机制。这有点像船舱被分隔为许多防水壁和许多舱壳一样。利用不同的进程所产生出来的边界，可以阻止错误到处蔓延。

另一个使用多重进程的理由是，当一个程序从一个作业平台被移植到另一个作业平台上。以 Unix 来说吧，Unix 不支持线程，但其进程的产生与结束的代价并不昂贵。因此，Unix 应用程序往往使用多个进程。如果要把它们移植为多线程模式，可能需要大改。在这种情况下，将程序移植到 Win32，可能需要成本效益上的妥协。

这一章所讨论的一些低阶解决方案，可能是你考虑在 Win32 环境中让进程彼此沟通时所需要的。我们将从“如何以消息队列权充数据转运中心”开始，看看如何产生并使用共享内存（shared memory）。本章最后我会谈到一些高阶解决方案，包括它们适用的场合，以及为什么你需要使用它们。

以消息队列权充数据转运中心

在同一个进程的不同线程之间搬移数据，最简单的一种做法就是利用消息队列。我们在第 4 章的 EVENTTST 中就已经这么做了，当时是让 worker 线程告诉主线程去更新列表框（listbox）的内容。定义一个自己的消息，并把一个指向某一额外数据的指针放到消息的 LPARAM 去，是很容易的一件事。

如果你尝试在进程 A 的线程中把一个 LPARAM（内含一个指针）交给进程 B，进程 B 有可能在使用这一指针时当掉。问题出在这个指针所指的数据乃位于进程 A 的地址空间中，进程 B 不可能看到这个地址空间，所以这个指针会被以进程 B 的地址空间解释之。那当然是牛头不对马嘴了。

为解决这个问题，Windows 定义了一个消息，名为 WM_COPYDATA，专门用来在线程之间搬移数据——不管两个线程是否同属一个进程。和其他所有的消息

一样，你必须指定一个窗口，也就是一个 HWND，当做消息的目的地。所以欲接收此消息的线程必须有一个窗口（译注：也就是它必须是个 UI 线程）。

WM_COPYDATA 消息的使用方式如下：

```
SendMessage(hwndReceiver,
          WM_COPYDATA,
          (WPARAM)hwndSender,
          (LPARAM)&cds);
```

LPARAM 参数 cds 必须指向一个特定的 Windows 数据结构：

```
typedef struct tagCOPYDATASTRUCT {  // cds
   DWORD  dwData;
   DWORD  cbData;
   PVOID  lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

结构中的各栏位的意义如下：

dwData	这是一个用户自定义值，它通常被用做一个“行动代码”，指示 lpData 中的内容的用途。
cbData	lpData 所指之数据大小（字节，bytes）。
lpData	一块数据，可以被传送到接收端（目标窗口所属之线程）。

你必须使用 SendMessage() 传送 WM_COPYDATA，不能够使用 PostMessage() 或任何其他变种函数如 PostThreadMessage() 之流。这是因为系统必须管理用以传递数据的缓冲区的生命期。如果你使用 PostMessage()，数据缓冲区会在接收端（线程）有机会处理该数据之前，被系统清除并摧毁。

缓冲区的生命期是一个重点。如果要把数据传送给一个以 CreateThread() 产生出来的线程，那块数据空间必须从 heap 中获得，因为线程将在 CreateThread() 返回之后开始执行（译注）。lpData 所指向的这块数据并不受限于这种方式。因为你用的是 SendMessage()，你可以保证接收端在

SendMessage() 返回之前一定已经完成其对数据的操作（译注：这是因为 SendMessage 有同步特性），所以 lpData 所指的空间可以在 heap 之中，也可以在 stack 之中。

译注 作者这几句话实在太精简了些。“数据从 heap 中获得”和“线程将在 CreateThread() 返回后执行”有什么关系呢？喔，我想他的意思是，数据如果通过 CreateThread() 的第四个参数传给线程，而线程在 CreateThread() 返回后才开始执行，那么传递过去的数据应该在 CreateThread() 返回之后还存在。而我想，这应该是 CreateThread() 内部自己要负责的部分，与我们在其第四个参数所指定的数据究竟在 stack 或 heap 中无关。你可以回头看看第 2 章的列表 2-2，指定给 CreateThread() 的第四个参数 i，是 main() 的局部变量，存在于 main() 的 stack 之中。

然而，接收端（线程）所获得的数据系属于系统。数据区块只是临时存在，一旦消息被处理之后，立刻就会被清除。并且因为是系统拥有这块空间，它被认为应该是只读（read only）属性。如果接收端（线程）需要改变数据内容，或是需要储存一份比较久远的数据副本，接收端应该自行拷贝一份。

产生 COPYDATASTRUCT 缓冲区

产生一个由 lpData 所指的 COPYDATASTRUCT 缓冲区时，你得很小心。系统不知道缓冲区内有什么数据，它只是把那些数据视为一块内存。因此，你可以在 lpData 中放置一个指针，指向下面这样的结构：

```
struct GoodDataBlock
{
    DWORD  dwNumber;
    char   szBuffer[80];
};
```

如果是下面这样的结构就不可以，因为 szBuffer 只是个指针，没有真正的空间：

```
struct BadDataBlock
{
    DWORD  dwNumber;
    char  *szBuffer;
};
```

进入 C++ 世界，情况又更复杂些了。绝对不可以把“指向某一拥有虚函数的对象”的指针当做 lpData 来传递。因为 vtbl（译注：virtual table，因虚函数而形成）指针将因此错误地指向别的进程中的函数。这样的限制也就排除了对“运行时类型检验（runtime type checking）”的使用，因为运行时的类型检验需仰赖虚函数。下面这个例子中，析构函数被声明为虚函数，所以将会拥有一个 vtbl 指针。指向此 BadDataClass 对象之指针若被用做 lpData，vtbl 指针将不正确地指向另一个进程。

```
class BadDataClass
{
public:
    BadDataClass();
    virtual  ~BadDataClass();
};
```

使用拥有虚函数的内嵌类（embedded classes），或是那些“拥有指针，指向它们自己”的内嵌类，也都必须小心。MFC 中有一个容易被人遗忘的数据类型：CString，它内含一个指针指向字符数据，所以 WM_COPYDATA 不能够为它弄出一份正确的拷贝。下面这个例子就有那样的问题，还有因为它派生自 CObject，而 CObject 拥有虚函数。

```
class BadMfcDataClass : public CObject
{
public:
    CString strDescription;
};
```

如果你使用一个指针指向 this，作为 COPYDATASTRUCT 中的 lpData 内容，那绝对是不安全的。那和“拥有虚函数的类”或是“内嵌类”一样地不安全。

COPYDATA 范例程序

为了示范如何使用 WM_COPYDATA，我产生两个 MFC 程序。第一个是 COPYRECV，产生一个窗口，内含多行编辑控件（multiline edit control），并等待一个消息以便接收数据。第二个程序，COPYSEND，允许你输入文字，然后传送到 COPYRECV 窗口中。COPYRECV 会将接收到的文字显示在窗口中，然后你可以在 COPYRECV 中选择、复制、储存这些文字。

COPYRECV 程序利用 MFC 的 CEditView 产生一个 SDI 窗口（译注：SDI，Single Document Interface）。此类提供了一个很方便的方法来产生一个多行编辑窗口。WM_COPYDATA 消息处理函数位于 CMainFrame 之中，因为 WM_COPYDATA 会被送往最上层窗口。列表 13-1 显示 WM_COPYDATA 消息处理函数的内容。

这个处理函数靠着 dwData 决定对结构中的数据做什么样的处理。本例定义了 ACTION_DISPLAY_TEXT 和 ACTION_CLEAR_WINDOW 两个常量。它们并不是 Windows 或 MFC 定义的常量，而是你自定义的。

如果你只是对 MFC 有一点点认识，请你注意 MFC 使字符串的处理变得多么容易。MFC 允许我们使用基本类型的连接操作，也允许 Trim() 函数清除空白字符。

列表 13-1 节录自 COPYRECV——WM_COPYDATA 的处理函数

```
#0001  LONG CMainFrame::OnCopyData( UINT wParam, LONG lParam)
#0002  {
#0003  //  HWND hwnd = (HWND)wParam;       // handle of sending window
#0004      PCOPYDATASTRUCT pcds = (PCOPYDATASTRUCT) lParam;
#0005
#0006      // The view is inside the mainframe,
#0007      // and the edit control is in the view
#0008      CEditView* pView = (CEditView*)GetActiveView();
#0009      ASSERT_VALID(pView);
#0010
#0011      // Find the edit control
```

```
#0012     CEdit& ctlEdit = pView->GetEditCtrl();
#0013
#0014     switch (pcds->dwData)
#0015     {
#0016     case ACTION_DISPLAY_TEXT:
#0017        {
#0018        LPCSTR szNewString = (LPCSTR)(pcds->lpData);
#0019    // Setting a CString equal to a LPCSTR makes a copy of the string.
#0020        CString strTextToDisplay = szNewString;
#0021        // Throw away any \r\n that may already be there
#0022        strTextToDisplay.TrimRight();
#0023        // Now add our own
#0024        strTextToDisplay += "\r\n";
#0025
#0026        // Set the cursor back at the end of the text
#0027        int nEditLen = ctlEdit.GetWindowTextLength();
#0028        ctlEdit.SetSel(nEditLen, nEditLen);
#0029        // Append the text
#0030        ctlEdit.ReplaceSel(strTextToDisplay);
#0031        ctlEdit.ShowCaret();
#0032        break;
#0033        }
#0034
#0035     case ACTION_CLEAR_WINDOW:
#0036        ctlEdit.SetWindowText("");
#0037        break;
#0038
#0039     default:
#0040        break;
#0041     }
#0042
#0043     return 1;
#0044  }
```

COPYSEND 程序是一个对话框程序，那也是 MFC 程序的一种标准形式。COPYSEND 先使用 FindWindow() 找出 COPYRECV 窗口，也就是标题栏为“CopyRecv Display Window”者（译注：这个窗口标题与原书所记载的不同，我是根据书附盘片中的源代码而写）。视你要清除窗口或是要传输文字，

程序会以不同的方式来建造 COPYDATASTRUCT。这块数据将在 SendToServer() 函数中被送往 COPYRECV。列表 13-2 显示此函数之内容。

译注 在作者轻描淡写的几句话中，其实隐藏着一个鲜为一般 MFC 程序员注意的难题。作者说使用 FindWindow() 找出 COPYRECV 窗口，但事实上 MFC 程序的窗口标题往往会被系统自动补加上 document 名称。为了消除额外的窗口标题文字，作者做了这样的手脚：

```
#0001  BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
#0002  {
#0003      // TODO: Modify the Window class or styles here by modifying
#0004      //  the CREATESTRUCT cs
#0005
#0006      cs.style &= ~(FWS_ADDTOTITLE);  // Do not put in a document
name
#0007
#0008      return CFrameWnd::PreCreateWindow(cs);
#0009  }
```

Paul Dilascia 在 Microsoft Systems Journal（MSJ）1995.10 的 C++ Q/A 专栏中，介绍过如何使用 MFC 撰写一个 Debug Window（DBWIN），他用了不同于本书的另一个技巧，以便藉着 FindWindow 找出目标窗口。MSJ 1995.12 的 C++ Q/A 专栏中，他又改用 WM_COPYDATA 消息来传递数据。这两篇文章的主旨也曾在《深入浅出 MFC》第 2 版的附录 D 中被讨论过。

OnOk() 是【Send】按钮的处理函数。把【OK】按钮改装为【Send】按钮，使我能够快速处理用户按下的【Enter】键，而不需要额外的程序代码。

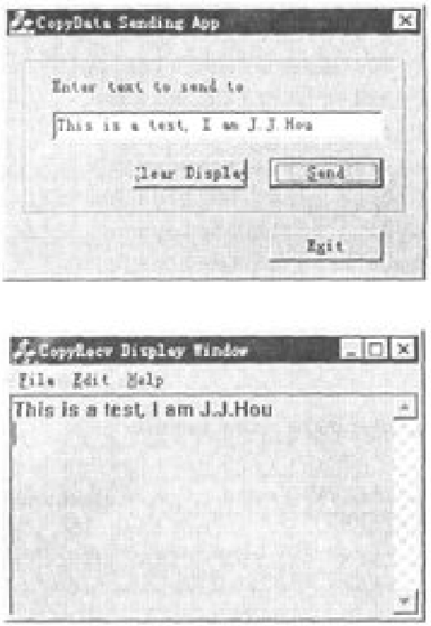
就像你所使用的任何其他 Windows 结构一样，请记住，一定要在你使用它之前，先把结构内容清为 0。即使你的如意算盘是把所有栏位填满，最好事前还是先把所有栏位清为 0，以避免哪一天有人改变了你的如意算盘。

列表 13-2 节录自 COPYSEND——建造并送出 WM_COPYDATA 消息

```
#0001 void CCopySendDlg::OnOk()
#0002 {
#0003     CEdit *pEdit = (CEdit*)GetDlgItem(IDC_EDIT_SENDTEXT);
#0004     ASSERT_VALID(pEdit);
#0005
#0006     // Get the text from the edit control
#0007     CString strDisplayText;
#0008     pEdit->GetWindowText(strDisplayText);
#0009
#0010     COPYDATASTRUCT cds;
#0011     memset(&cds, 0, sizeof(cds));
#0012     cds.dwData = ACTION_DISPLAY_TEXT;
#0013     cds.cbData = strDisplayText.GetLength() + 1; // +1 for the NULL
#0014     cds.lpData = (LPVOID)(LPCTSTR) strDisplayText;
#0015
#0016     SendToServer(cds);
#0017 }
#0018
#0019 void CCopySendDlg::OnClear()
#0020 {
#0021     COPYDATASTRUCT cds;
#0022     memset(&cds, 0, sizeof(cds));
#0023     cds.dwData = ACTION_CLEAR_WINDOW;
#0024
#0025     SendToServer(cds);
#0026 }
#0027
#0028 void CCopySendDlg::SendToServer(const COPYDATASTRUCT& cds)
#0029 {
#0030     CWnd *pDisplayWnd = CWnd::FindWindow(NULL, szDisplayAppName);
#0031     if (pDisplayWnd)
#0032     {
#0033         pDisplayWnd->SendMessage(WM_COPYDATA,
#0034             (WPARAM)GetSafeHwnd(), (LPARAM)&cds);
#0035     }
#0036     else
#0037         AfxMessageBox(IDS_ERR_NOSERVER);
#0038 }
#0039
```

```
#0040  void CCopySendDlg::OnExit()
#0041  {
#0042      EndDialog(IDOK);
#0043  }
```

译注 下图是 COPYSEND 和 COPYRECV 的执行画面：



WM_COPYDATA 的优缺点

WM_COPYDATA 有一个特殊的性质，使它极不寻常。这个消息可以在 16 位程序和 32 位程序中使用。Win16 程序没有支持大部分的进程通讯机制，WM_COPYDATA 是唯一一个可以在 16 位程序和 32 位程序之间搬移数据的方法。

但是 WM_COPYDATA 也有一些缺点。第一，内部机制的爆发力不够，如果你需要高效率，这个解决方案恐怕不行。任何你所传递的数据都需要先被

拷贝到另一个进程之中。有一个比较快速的机制：共享内存（shared memory），将在下一节讨论。Windows 就是以共享内存来实现 WM_COPYDATA。

第二个缺点是，你没办法使用 PostThreadMessage()，因为 WM_COPYDATA 只能用于 SendMessage()。因此，接收端必须有一个消息队列，以及一个相关的窗口——即使不显示出来也行。如果接收端线程没有窗口，你就没有办法使用 WM_COPYDATA。

最后一点，SendMessage() 是一个同步（synchronous）函数调用。送出消息的一方在收受方尚未处理完毕之前，不能够继续下去。这强迫传送端和接受端有同步效果。如果接受端忙碌，则在 WM_COPYDATA 被处理之前以及传送端继续下去之前，会有时间上的延误。

使用共享内存（Shared Memory）

Win32 的一个基础观念就是，进程之间要有严密的保护。每一个进程认为它拥有整部机器。从一个进程中要看到另一个进程的地址空间的任何一部分，都是不可能的。这样的分离策略是如此完整，以至于每一个进程似乎生活在完全相同的地址范围内。一个程序所存在的实际内存的地址，对另一个程序而言，是朦胧而不可见的。程序能够看到的只是所谓的逻辑地址。事实上，程序所存在的内存的实际地址可能会不断地改变，因为虚拟内存管理器会自动搬移程序的一部分，进出虚拟储存空间（译注：硬盘）中。

对于在进程之间搬移数据，WM_COPYDATA 技术非常简单，但有时候你需要更高效率的技术。你需要让进程真正地共享数据，就像同一进程中的线程一样，于是某个进程对该数据的改变，立刻就能够反应在另一个进程中。要达到这一点，你必须使用 Win32 进程通讯技术中的最低阶一层：共享内存（shared memory）。

所谓共享内存，是一块在设计时即打算给一个以上的进程在同一时间都看得到的内存区域。虽然，从这个区域中获得一个指针，就好像你以 GlobalAlloc() 获得一块空间没有什么两样，但是当使用这块空间时，就有一些重大差异了，稍后我将特别为此辟一小节。

我要带着你以下列次序来学习如何使用共享内存，然后再看看可有什么更好的做法。

1. 设定一块共享内存区域。
2. 使用共享内存。
3. 同步处理共享内存。

你可以在第 16 章看到一个更精巧的共享内存使用实例，那是一个可以安插于 Internet Information Server（IIS）的 ISAPI plug-in 软件。那个 plug-in 软件使用第二个进程来避开一个问题："使用数据库，本身却是单线程程序"。

设定一块共享内存区域（Shared Memory Area）

设定一块共享内存区域，需要两个步骤：

1. 产生一个所谓的 file-mapping 核心对象，并指定共享区域的大小。
2. 将共享区域映射到你的进程的地址空间中。

第一个步骤使用 CreateFileMapping() 函数。一般而言，这个函数允许你存取一个文件，就好像它是内存中的数据一样，但是我们要使用的是另一个特殊的模式，它会在页面文件（paging file）中产生一块空间，使任何一个进程都可以根据其名称而存取到它。CreateFileMapping() 规格如下：

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
```

```
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    LPCTSTR lpName
);
```

参数：

hFile	这个参数正常而言应该是 CreateFile() 传回来的一个有关于文件的 handle，用以告诉系统将它映射到内存中。然而如果指定此参数为 (HANDLE)0xFFFFFFFF，我们就可以使用页面文件（paging file）中的一块空间，取代一般的文件。
lpFileMappingAttributes	安全属性。Windows 95 将忽略此参数。
flProtect	文件的保护属性。可以是 PAGE_READONLY 或 PAGE_READWRITE 或 PAGE_WRITECOPY。针对跨进程的共享内存，你应该指定此参数为 PAGE_READWRITE。
dwMaximumSizeHigh	映射之文件大小的高 32 位。如果使用页面文件（paging file），此参数将总是为 0，因为页面文件没有大到足够容纳 4GB 的共享内存空间。
dwMaximumSizeLow	映射区域的低 32 位。对于共享内存而言，此值应该就是你要共享的内存大小。
lpName	共享内存区域的名称。任何进程或线程都可以根据这个名称，引用到这个 file-mapping 对象。如果你要产生共享内存，此参数不应该是一般情况下所使用的 NULL。

返回值

如果成功，则 CreateFileMapping() 传回一个 handle，否则传回 NULL。GetLastError() 可以获得失败的合理解释。如果参数中所指定的文件已经存在，CreateFileMapping() 便会失败，这时候 GetLastError() 会传回 ERROR_ALREADY_EXISTS。

CreateFileMapping() 产生出一个 file-mapping 核心对象。虽然此核心对象的必要性在本讨论之中并不是很明显，但是当一个进程必须从其他进程继承此一 handle（在启动时刻），或是必须处理超过 2GB 的大文件时，此 handle 的用途就显现出来了。对我们的目标而言，file mapping 核心对象是一个调解用的 handle，其他进程可以用它来存取共享内存。

你们之中有些已经熟悉 CreateFileMapping() 的人可能会惊讶，为什么我使用页面文件（paging file）而不使用文件系统中的某个文件。两个理由，第一，如果使用文件系统中的文件，每个人都得同意其文件名称及其位置。这是多么没有必要的琐屑事情。第二，如果机器当掉了的话，一个文件可能变得陈旧而无用，徒占磁盘空间。如果你以页面文件提供共享内存，上述问题都不会发生。

除了产生共享内存，CreateFileMapping() 还有许多其他用途。请参考 Visual C++ 的联机帮助文件。

现在我们有了一个核心对象，但是我们还没有获得一个指针指向可用的内存。为了从共享内存中获得一个指针，我们必须使用 MapViewOfFile()。

```
LPVOID MapViewOfFile(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    DWORD dwNumberOfBytesToMap
);
```

参数

hFileMappingObject	file-mapping 核心对象的 handle，这是 CreateFileMapping() 或 OpenFileMapping() 的传回物。
dwDesiredAccess	对共享内存而言，此值应该设为 FILE_MAP_ALL_ACCESS。其他目的则使用其他设定。
dwFileOffsetHigh	映射文件的高 32 位偏移值。如果使用页面文件（paging file），该参数应该总是为 0，因为页面文件不可能大到足够容纳 4GB 共享内存区域。
dwFileOffsetLow	映射文件的低 32 位偏移值。对于共享内存而言，该参数应该总是 0 以便能够映射整个共享区域。
dwNumberOfBytesToMap	真正要被映射的字节数量。如果指定为 0，表示要映射整个空间。所以，对共享内存而言，最简单的做法就是将此参数指定为 0。

返回值

如果成功，则传回一个指针，指向被映射出来的“视图”（mapped view）的起头。如果失败，传回 NULL，你可以利用 GetLastError() 找出原因。

下面是一小段程序代码，用以示范如何产生一个共享内存，大得足够持有一个 DWORD。一个 DWORD 之为用大矣，可以存放服务器线程（专门用来处理某些需求）的 ID。我们必须储存线程的 ID 而不是其 handle，因为 handle 只在其自己的地址空间中有意义。为了让程序代码简洁，我没有做任何错误检验。

```
#0001  HANDLE hFileMapping;
#0002  LPDWORD pCounter;
#0003
#0004  hFileMapping = CreateFileMapping(
#0005                   (HANDLE)0xFFFFFFFF,    // File handle
#0006                   NULL,                 // Security attributes
#0007                   PAGE_READWRITE,        // Protection
#0008                   0,                    // Size - high 32 bits
```

```
#0009                   sizeof(DWORD),        // Size - low 32 bits
#0010                   "Server Thread ID");   // Name
#0011
#0012  pCounter = (LPDWORD) MapViewOfFile(
#0013                   hFileMapping,         // File mapping object
#0014                   FILE_MAP_ALL_ACCESS,  // Read/Write
#0015                   0,                   // Offset - high 32 bits
#0016                   0,                   // Offset - low 32 bits
#0017                   0);                  // Map the whole thing
#0018
#0019  *pCounter = GetCurrentThreadId();
```

注意，很妙而且很重要的一点：内存映射文件在内存中产生出一块新的区域，数据可以被放在上面。这有点像 GlobalAlloc()。它并不会要求系统将一个现有的内存区域变成“可共享”。

找出共享内存

前一节中我为你展示了如何产生一个共享内存核心对象，让其他进程能够依据名称找到它，然后将它映射到自己的地址空间中，形成一个“视图”（view）。当你决定要如何使用这块共享内存时，你必须决定这块内存是要以点对点（peer to peer）的形式呈现，还是希望被一个 server 进程产生，然后被数个 client 进程打开取用。

如果共享内存被用于点对点的形式，则每一个进程都必须有相同的能力，产生共享内存并将它初始化。每一个进程都应该调用 CreateFileMapping()，然后调用 GetLastError()。如果传回的错误代码是 ERROR_ALREADY_EXISTS，那么进程就可以假设这一共享内存区域已经被其他进程打开并初始化过了。否则进程就可以合理地认为自己排第一位，并接下来将共享内存初始化。

如果共享内存要被使用于 client/server 架构中，那么就只有 server 进程才应该产生并初始化共享内存。所有的 client 进程都应该使用 OpenFileMapping()。该函数会传回一个 handle，代表一个 file-mapping 核心对象，那是稍早被 server 进程以 CreateFileMapping() 产生出来的。OpenFileMapping() 的规格如下：

```
HANDLE OpenFileMapping(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR lpName
);
```

参数

dwDesiredAccess	对于共享内存，此值正常应该是 FILE_MAP_ALL_ACCESS。其他值适用于其他目的。
bInheritHandle	如果是 TRUE，表示这个 handle 可以被子进程继承。Handle 的继承并未涵盖于本书范围之内。
lpName	共享内存的名称。应该有另一个进程以相同的名称调用 CreateFileMapping()。

返回值

如果成功，则 OpenFileMapping() 传回一个 handle。如果失败，则传回 NULL，这时你可以利用 GetLastError() 获得更多的细节信息。

在调用 OpenFileMapping() 之后，进程应该调用 MapViewOfFile() 以获得一个指针，指向共享内存。这似乎很简单，其背后却有一个很大的意义需要了解。第二个进程以及后续各进程调用 OpenFileMapping() 之后所获得的地址，并不保证和第一个进程所获得的地址相同。事实上，对于不同的进程，共享内存可以被映射到不同的地址。图 13-1 显示出这种情况。本章稍后我还会讨论这一现象。

实际上，Windows 95 的 OpenFileMapping() 的确总是把共享内存映射到同一地址上。这是因为共享内存总是被放在一个可以被大家（每一个进程）都看得到的位置。所以任何一个进程都有可能在你的共享内存上放一些垃圾——甚至即使它们并没有开启你的共享内存。这是 Windows 95 的一个不好的地方。Windows NT 的内存安全防护就强固得多，不会再允许这样的行为。

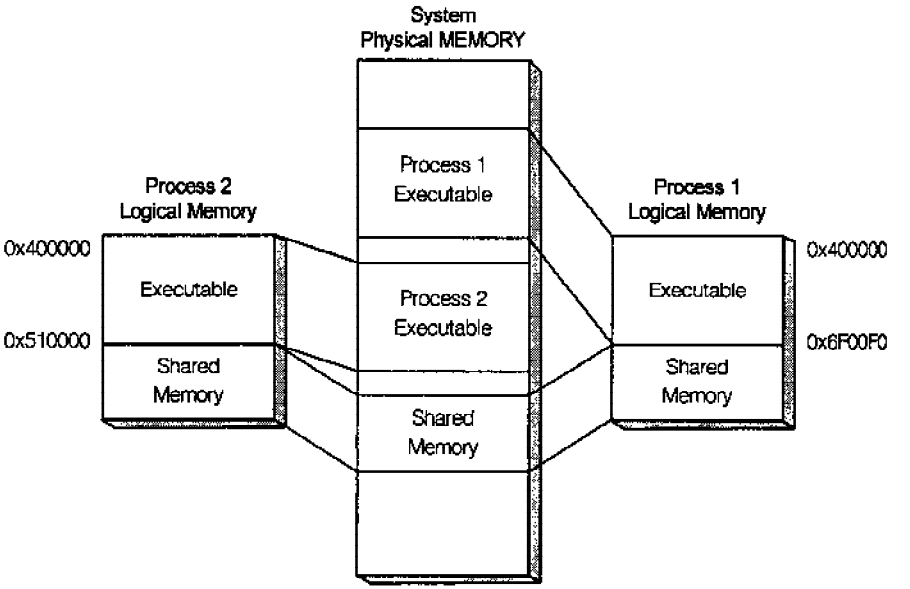


图 13-1 共享内存被映射到许多个进程之中

Windows 95 的这种行为可以视之为一种意外。如果你的程序在不同的进程中使用共享内存，你应该在 Windows NT 环境下测试你的程序，以确保你并不依赖“同一地址”的共享内存。

译注 如果你希望对 Win32 的分离地址空间（separate address spaces）以及共享内存的所在地址有更深刻更具体的了解，请参考 Windows 95 System Programming SECRETS（Matt Pietrek/IDG Books）第 5 章：Memory Management。

清理（Cleaning up）

一旦你完成了对共享内存的操作，你应该调用 UnmapViewOfFile()，交出原本由 MapViewOfFile() 所获得的指针，然后再调用 CloseHandle()，交出 file-mapping 核心对象的 handle。UnmapViewOfFile() 规格如下：

```
BOOL UnmapViewOfFile(
    LPCVOID lpBaseAddress
);
```

参数

lpBaseAddress 指针，指向共享内存．这个值必须符合 MapViewOfFile() 的传回值．

返回值

如果成功，则 UnmapViewOfFile() 传回 TRUE．如果失败，则传回 FALSE，这时你可以利用 GetLastError() 获得更多的细节信息。

同步处理

有可能你需要产生一个全局性标记，用以表示共享内存是否已经被适当地初始化。甚至即使如稍早的例子中，共享内存不过只是个 DWORD，也还是有可能发生 context switch，致使其他进程打开共享内存并抓取其值——在共享内存尚未被正确设定内容之前。

最明显的解决方式就是在共享内存中设定某些魔术数字．但是，如果一个进程寻到那个位置并发现共享内存还未设定，它又该如何等待呢？这个进程势必借助一个 busy loop，而那是我们认为极不妥的操作。

最好的做法就是使用一个 mutex，或是一个 Readers/Writers lock。注意，第 7 章的 Readers/Writers lock 是设计用于单进程之中，必须再加修改才能适用于多重进程。这样的一个 lock 可以在启动时提供保护，并提供共享内存读写时的同步控制（synchronization）．其他进程应该打开那个 mutex 并且等它变成激发状态，才继续进行下去。当共享内存被完全初始化之后，mutex 可以被释放，而处于等待状态中的进程也终于得以继续工作。

使用指针指向共享内存（Shared Memory）

共享内存指针的使用，是非常微妙而不好处理的一件事。稍早我在“产生COPYDATASTRUCT 缓冲区”一节中所讨论过的一些限制，依然适用于共享内存。而由于共享内存通常是一个大区块，问题甚至于会更多。

例如在我所检阅过的一个使用共享内存的 MFC 程序中，我看到这样一小段代码：

```
HANDLE hFileMapping = CreateFileMapping( ... );
LPVOID pSharedView = (LPDWORD) MapViewOfFile( ... );
CStringArray* pStringArray = new(pSharedView)CStringArray;
```

注意：上面最后一行代码使用 C++ 语法中的“placement new”。那是 new 运算符的一个特别版本，用来初始化一个既存的内存区块，而不是配置一个新区块。CStringArray 派生自 CObject，而“placement new”运算符是为所有的 MFC CObject 派生类而定义。之所以命名为 "Placement new" 是因为，它在一个原有的空间中产生了一个新对象。

上述程序片段所造成的情况是，指向共享内存的指针，现在被强制转换类型为一个 CString 数组，然后数组被构造函数（constructor）初始化，然后程序开始填充 CStringArray 的内容。问题是，一个 CStringArray 对象实体只有 16 个字节长，它只不过是内含一个指针，指向数组数据。数组数据本身被配置于 heap，所以这会使得数组数据被置于一般的内存中，而非共享内存中。

甚至即使数组被适当地完成了，单个的 CString 还是会引起错误，因为每一个配置空间（为了储存字符串）都来自 heap，它没有 local 缓冲区。最后，甚至即使数组数据都被置于共享内存中，指向它的那个指针也是错误的，因为它无法在另一个进程中被使用。

我们获得的教训是：任何 collection class（译注：用于 array、list、map 之 C++ 类），不论来自 MFC 或其他地方，都不能够安全地使用于共享内存中，

__based 属性

编译器支持一个鲜为人知的旁门左道，用以解决一些与共享内存有关的问题。那是一个指针修饰词，称为 __based，允许指针被定义为从某一点开始起算的 32 位偏移值，而不是内存中的绝对位置。

下面例子声明一个指针 lpHead，内部储存的是从 pSharedView 开始的偏移值。换句话说，lpHead 是“以 pSharedView 为基准”。ListNode 只不过是个方便的名称，用以代表一个任意结构：

```
HANDLE hFileMapping = CreateFileMapping( ... );
LPVOID pSharedView = (LPDWORD) MapViewOfFile( ... );

ListNode __based( pSharedView ) *lpHead;
```

虽然“based”指针很好用，你可能得在效率上付出一些代价。每次以“based”指针处理数据，CPU 都必须为它加上基地址，才能指向真正的位置。

__based 属性将在即将来到的范例程序中一览无遗。

SHAREMEM 范例程序

SHAREMEM 是一个 MFC 程序，示范如何使用共享内存，并传递一个数组的字符串。程序执行画面显示于图 13-2。你可以执行好几份 SHAREMEM，每一个副本都共享同一块内存。只要有一个人写入，其他人都可以读出。

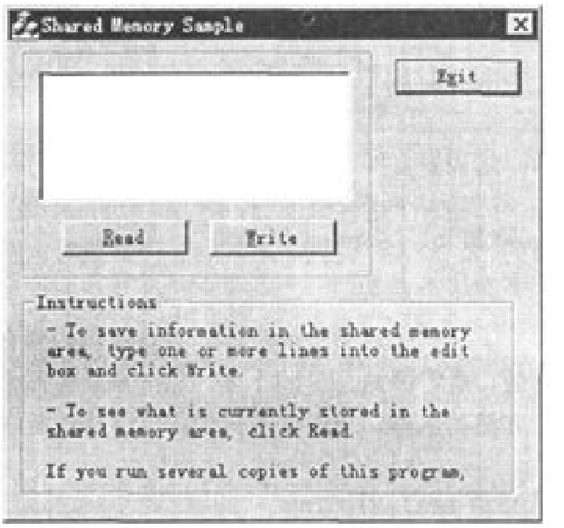


图 13-2 SHAREMEM 范例程序

只有当 SHAREMEM 的【Read】按钮被按下时，它才会读入共享内存的内容。没有任何通告机制（notification mechanism）可以告诉 SHAREMEM 说共享内存的内容有变。

这个范例程序展示了数个观念。它使用来自 file-mapping 核心对象的一个映射后的视图（mapped view），产生出共享内存。它使用一个 mutex 来避免对共享内存的同时读写操作。它也示范了“based”指针的用法。

共享内存在 InitInstance() 中被初始化。这一段代码显示于列表 13-3 中，其操作就像我在前数节所显示的一样。CreateFileMapping() 和 MapViewOfFile() 分别被调用，而当共享内存被创造出来，有一个 mutex 用来保护它。CreateFileMapping() 被调用之后，程序亦调用了 GetLastError() 以检验这是不是第一个产生出共享内存的进程，如果是则进行初始化工作。

当我们离开时，我们不会锁住 mutex，因为当它被解除映射（unmapping）时，其他进程就可以安全地处理共享内存了。由于 file-mapping 核心对象是以引用计数（reference count）来控制其生存与否，所以即使产生它的那个进程已经结束了，它还是可以继续存在。

这一段代码毫不吝啬地到处散布 MTVERIFY 宏，以确保每一件事情都正确地运作。在正式程序中，你应该以更强固的错误处理函数取代 MTVERIFY 宏。

列表 13-3 SHAREMEM 程序片段——初始化操作

```
#0001 // Create a mutex with initial ownership. If it already
#0002 // exists, we will block until it is available.
#0003 ghDataLock = ::CreateMutex(NULL, TRUE, "ShareMem Data Mutex");
#0004 MTVERIFY( ghDataLock != NULL );
#0005
#0006 HANDLE hFileMapping;
#0007
#0008 hFileMapping = ::CreateFileMapping(
#0009                  (HANDLE)0xFFFFFFFF,  // File handle
#0010                  NULL,                // Security attributes
#0011                  PAGE_READWRITE,      // Protection
#0012                  0,                   // Size - high 32 bits
#0013                  1<<16,               // Size - low 32 bits
#0014                  "ShareMem Sample App Data Block"); // Name
#0015 MTVERIFY( hFileMapping != NULL );
#0016 DWORD dwMapErr = GetLastError();
#0017
#0018 gpSharedBlock = (SharedBlock*) ::MapViewOfFile(
#0019                  hFileMapping,        // File mapping object
#0020                  FILE_MAP_ALL_ACCESS, // Read/Write
#0021                  0,                   // Offset - high 32 bits
#0022                  0,                   // Offset - low 32 bits
#0023                  0);                  // Map the whole thing
#0024 MTVERIFY( gpSharedBlock != NULL );
#0025
#0026 // Only initialize shared memory if we actually created.
#0027 if (dwMapErr != ERROR_ALREADY_EXISTS)
#0028    gpSharedBlock->m_nStringCount = 0;
#0029
```

```
#0030  ::ReleaseMutex(ghDataLock);
#0031
#0032  CShareMemDlg dlg;
#0033  m_pMainWnd = &dlg;
#0034  int nResponse = dlg.DoModal();
#0035
#0036  MTVERIFY( ::UnmapViewOfFile(gpSharedBlock) );
#0037  MTVERIFY( ::CloseHandle(hFileMapping) );
#0038  MTVERIFY( ::CloseHandle(ghDataLock) );
```

共享内存的起始处，内含 SharedBlock 结构。此结构定义如下：

```
// Declare a forward reference to cure a circular dependency.
struct SharedBlock;

extern SharedBlock* gpSharedBlock;

struct SharedBlock
{
   short  m_nStringCount;
   char __based( gpSharedBlock ) *m_pStrings[1];
};
```

成员变量 m_pStrings 是一个由“based”指针所组成的数组，每一个指针指向 char。虽然它被定义为 1 个字节大，这只不过是一种欺骗编译器的手法，让它允许我们在这里先开辟出一个数组。这个技术和 DIB（Device Independent Bitmap）管理调色盘的技术如出一辙。

如果有一个以上的字符串，它们的指针就会被紧接着写入 SharedBlock 之后。那些指针之后则是字符串本身，也就是被指针所指的部分。所有指针都是“based”指针，以本质而论，它们其实都是相对于共享内存起头处的索引值。

假设有三个字符串被载入到共享内存中，它们看起来应该是这样：

共享内存起始偏移值	内　容
0	3 (m_nStringCount)
4	16 (m_pStrings[0] - based ptr to string 0)
8	25 (m_pStrings[1] - based ptr to string 1)
12	31 (m_pStrings[2] - based ptr to string 2)
16	String 0\0
25	Str 1\0
31	String number 2\0

用来读写共享内存的程序代码，显示于列表 13-4 中。OnWrite() 中的指针“戏法”用来计算每一样东西应该放置在共享内存中的何处。计算方式相当复杂。是的，如果没有一个类似 heap 管理器之类的东西为我们服务，就只有依赖像这样未经开化的管理了。

OnWrite() 首先计算出 edit 控件之中有多少行字符串，然后把行数写入共享内存中，再计算出需要保留多少空间给指针数组。然后从 edit 控件中一次读取一行，把字符串储存到共享内存中，再储存一个“based”指针，指向数组中的字符串。整个操作被一个 mutex 保护住，所以其他进程不可能拦腰中断之。

列表 13-4 SHAREMEM 程序片段——OnWrite() 和 OnRead()

```
#0001  void CShareMemDlg::OnRead()
#0002  {
#0003      // Make sure the shared memory is available
#0004      ::WaitForSingleObject(ghDataLock, INFINITE);
#0005
#0006      CEdit* pEdit = (CEdit*)GetDlgItem(IDC_EDIT);
#0007      ASSERT_VALID(pEdit);
#0008      pEdit->SetWindowText("");
#0009      pEdit->SetSel(-1, -1);
#0010      pEdit->ShowCaret();
#0011
#0012      for (int i=0; i<gpSharedBlock->m_nStringCount; i++)
#0013      {
```

Win32多线程程序设计

（双击鼠标可读取彩色封面）

```
#0055
#0056          // Skip to the next open space in the buffer
#0057          pTextBuffer += strlen(szLineBuffer) + 1;
#0058      }
#0059
#0060      ::ReleaseMutex(ghDataLock);
#0061  }
```

共享内存的使用摘要

如果你开始有“使用共享内存不是件容易的事”这样的念头，你是对的。为了在共享内存中做出一个链表（linked list），你必须建立一个私人的内存配置系统，它必须有能力像对待一个 heap 那样地对待共享内存，并传回“based”指针。这可不是件小事情。这个困难性彰显了一个重点：尽量把要在线程之间共享的数据保持在最小量，并且严密地定义存取界面（数据靠着这一界面得以被共享）。

下面是一些使用共享内存的指南：

- 不要把 C++ collection classes 放到共享内存中。
- 不要把拥有虚函数之 C++ 类放到共享内存中。
- 不要把 CObject 派生类之 MFC 对象放到共享内存中。
- 不要使用“point within the shared memory”的指针。
- 不要使用“point outside of the shared memory”的指针。
- 使用“based”指针是安全的，但是要小心使用。

较高层次的进程通讯（IPC）

本章一开始我就说过，我们要讨论的是适用于 Win32 环境中的一些有关于进程通讯的低阶机制。还有许多其他方法可以在进程之间通讯，我将在这里大概地对它们做一个简单描述，但是不涉入太多细节。

和其他的平台（诸如 Unix）比较，你在 Win32 中其实不太需要用到低阶的进程通讯（IPC）机制，因为有一些标准的 API 调用已经能够处理大部分事态。例如，系统提供的剪贴板（clipboard）可以被任何程序在任何时间使用，它也允许任意数据在程序之间被传递来传递去，不要求每个程序都得理解数据如何被移动。

Anonymous Pipes

所谓 pipe 有点像是你家里头的管道。你在一端放入一些东西，另一端就会流出一些东西。一个“anonymous pipe”只做单向流动，并且只能够在同一部电脑的各进程之间流动。就是靠“anonymous pipes”，你的程序的 stdout 输出才能够被重定向，成为另一个程序的 stdin 输入。Anonymous pipes 只被使用于点对点通讯。当一个进程产生另一个进程时，这是最有用的一种通讯方法。

Pipes 之所以有用，另一个因素是，它们是操作系统提供的一个环状缓冲区。如果你有一个线程，需要喂数据给其他线程，pipe 是一种非常简单的做法。本书的许多例子中，我使用消息来传递数据给线程。使用消息，就必须要求接收端有一个窗口和一个消息回路。如果使用 pipe，就不必有那种基本条件。

有关 anonymous pipes 的更多信息，请参考 Visual C++ 联机帮助文件中的 Programmer's Reference 之中的 CreatePipe() 相关说明。

Named Pipes

Named pipes 有点像用途较广的 anonymous pipes。一个 named pipe 可以是单向，也可以是双向，并且可以跨越网路，不局限于单机。由于它拥有名称，所以任何进程都可以轻易抓住它。一个 named pipe 也可以轻易被框进所谓的“消息模式”（message mode）中，也就是说，你可以指定每一块被放进 pipe 的数据有着固定大小，而 ReadFile() 只能够读取该大小的倍数。这样的模式对于跨越网络是有帮助的，因为如果 pipes 是在“字节模式”（bytes mode）下执行，系统可能会传回消息不完整的一部分。

Name pipes 可以被设定为“overlapped operations”，并因此能够被使用于 I/O completion ports。Named pipes 是点对点通讯，类似 anonymous pipes。当连接关系产生，server 端可以根据相同的名称产生一个新的 named pipes。所以单独一个 server 便可以为许多个 clients 服务。

由于 named pipes 可以是双向的，所以它们对于在进程或线程之间建立一个双向对话很有帮助。Named pipe 的 handle 对进程而言应该是全局性的，所以数个不同的线程都从 pipe 中读取数据是可能的（在消息模式之下）。这可被使用于“以一个 server 线程将请求喂给同一进程的数个 clients 线程”的情况。

有关 named pipes 的更多信息，请参考 Programmer's Reference 之中有关于 CreateNamedPipe() 的说明。

Mailslots

Pipes 被设计为点对点通讯，mailslots 则被设计为广播式通讯。Mailslots 有点像你家里头的信箱。任何人都可以寄信给你，但只有你才能够从信箱中取出信件。Server 进程可以产生 mailslot，任何 client 进程都可以写数据进去，但只有 server 进程可以读出其中的数据。Mailslots 是具名的，并且可以跨网络。

OLE Automation

OLE Automation 和 Uniform Data Transfer 都是更高阶的机制，允许通讯发生于进程边界，或甚至于机器边界。OLE 允许对象以一种标准方式和其他对象相互沟通。所谓其他对象，可能存在于 DLL 之中或其他进程之中。OLE 可以潜在性地和 non-Win32 平台交谈。OLE 可以使用本章介绍的某些机制，这些机制只不过是 OLE 所支持的传输机制的一部分而已。第 17 章有一个 OLE 实例。

DDE

DDE 的全名是 Dynamic Data Exchange，使用于 16 位 Windows。我之所以提到它纯粹是为了完整性的缘故。OLE 1.0 便是架构在 DDE 之上。DDE 是一个以消息传递为基础的系统，允许程序与程序之间交换字符串或数据。它也被用来告诉一个执行中的程序将文件打开。然而因为 DDE 有一些先天上的限制，包括时间终了（time-out）的处理，以及 DDE client 和 DDE server 之间的连接，所以让我再一次警告，在其他机制可用的情况下，尽量不要使用 DDE。

提要

这一章中你看到了如何在两个进程之间搬移数据，用的是 WM_COPYDATA 消息以及一块内存。你也看到了如何产生并使用共享内存，那是最低阶的程序通讯做法。最后，你看到了因共享内存而起的一些问题，以及“based”指针如何帮助解决这些问题。

OLE Automation

OLE Automation 和 Uniform Data Transfer 都是更高阶的机制，允许通讯发生于进程边界，或甚至于机器边界。OLE 允许对象以一种标准方式和其他对象相互沟通。所谓其他对象，可能存在于 DLL 之中或其他进程之中。OLE 可以潜在性地和 non-Win32 平台交谈。OLE 可以使用本章介绍的某些机制，这些机制只不过是 OLE 所支持的传输机制的一部分而已。第 17 章有一个 OLE 实例。

DDE

DDE 的全名是 Dynamic Data Exchange，使用于 16 位 Windows。我之所以提到它纯粹是为了完整性的缘故。OLE 1.0 便是架构在 DDE 之上。DDE 是一个以消息传递为基础的系统，允许程序与程序之间交换字符串或数据。它也被用来告诉一个执行中的程序将文件打开。然而因为 DDE 有一些先天上的限制，包括时间终了（time-out）的处理，以及 DDE client 和 DDE server 之间的连接，所以让我再一次警告，在其他机制可用的情况下，尽量不要使用 DDE。

提要

这一章中你看到了如何在两个进程之间搬移数据，用的是 WM_COPYDATA 消息以及一块内存。你也看到了如何产生并使用共享内存，那是最低阶的程序通讯做法。最后，你看到了因共享内存而起的一些问题，以及“based”指针如何帮助解决这些问题。

建造 DLLs

这一章叙述如何处理 DLL 之中有关于进程和线程的附着（attach）和解除附着（detach）通告消息，以及你可能面对的其他问题。我同时也描述线程局部存储（thread local storage，TLS）的意义与使用。

所谓 DLL，动态链接库，是 Windows 的一个标准机制，用来完成以下三件事情：

- 提供给每一个程序一组一致的 APIs，让它们去调用。以此角度出发，DLL 是可共享的模块。
- 提供一个函数库，可以在运行时被载入。
- 降低可共享程序代码的内存使用量。

这一章中我们将讨论上述观念之中与多线程有关的部分。如果你需要关于如何产生、编译、使用 DLLs 更多信息，我建议你从 Visual C++ 联机帮助文件中的这一篇文章开始：Overview: Dynamic-Link Libraries（DLLs）。

UNIX DLL 的运作方式和 Unix 中的 shared library 有很大的不同，虽然最后的结果类似，但其机制并不相似。DLL 是一个链接实体，有它自己的实际生存事实，DLL 有自己的数据，独立于进程之外；有它自己的模块识别代码（ID），可以被静态链接，也可以被动态链接（译注）。请看 Win32 说明文件以得到更多相关信息。

译注 我想作者在这里说“DLL 可以被静态链接也可以被动态链接”，意思是指 DLL 的 explicitly linking 和 implicitly linking 两种方式。前者是在程序中很明显地以 LoadLibrary() 将 DLL 载入，后者是在链接时期静态链接 DLL 的一个“替身”，也就是其 import library（输入函数库）。

如果你曾经在 Win16 环境下写过 DLL，特别是在早期的 Windows 3.0 时代，可能你还记得一个神秘的过程，包括以汇编语言（assembly）完成的启动代码、以数值识别的函数、以及一个 WEP（Windows Exit Procedures），还有一大堆的限制。今天，在 Win32 和 Visual C++ 之下，程序员肩膀上的负担已经小多了。DLL 的设计比以前更简单更容易得多。

与 Win16 比较，Win32 带来了许多对 DLLs 的加强，使它们更适合于多任务环境。最重要的一点是，面对每一个 DLL 的用户，DLL 自动为其全局变量提供一份拷贝。在 Win16 之中你必须很辛苦地设计才能够确保“DLL 的每一个用户”彼此隔离。在 Win32 中这一隔离关系是自动而透明的。如果 DLL 之中有一个数组用来记录被打开的文件 handles，那么每当一个程序载入此一 DLL，就会有一个数组空间被产生出来。

由于 DLLs 被设计用来当做可共享的模块或元件，当一个进程或线程使用它，DLLs 需要获得详细的信息。Win32 提供这份信息的做法是，调用 DLL 中的一个函数，并给予当时的状态。

DLL 的通告消息（Notifications）

DLL 是一个奇怪的产品，因为它只存在于某个程序之内（以行家的话来说则是存在于某个程序的“context”之中），但是当它被载入后，它有自己的“人格”。如果有许多个程序都使用同一个 DLL，春去秋来花开花落，DLL 并不依赖哪一个特定的程序而存在。只要还有人需要 DLL，DLL 就会存在。从许多角度来看，DLL 都像是一栋商业大楼的接待员。商务来来往往，人们进进出出，接待员总是在那里。但是如果所有的商业活动都结束了，当然，接待员也就不需要了。

与接待员一样，DLL 必须保持人员进出的记录。实际上我指的是进程或线程使用或放弃 DLL 的次数。为了接收这样的消息，你必须设计一个名为 DllMain()的函数，它的规格如下。

注：如果你使用 C runtime library，那么你应该使用 DllMain() 名称。但如果你没有使用 C runtime library（这是很希罕的情况），你可以利用链接器选项 /ENTRY 定义你自己喜欢的函数名称。

注意：如果你想在联机帮助文件中搜寻这份规格，你应该看 DllEntryPoint() 函数。这是一个你不会真正去用它的抽象名称。

```
BOOL WINAPI DllMain(
  HANDLE hinstDll,
  DWORD  fdwReason,
  LPVOID lpReserved
);
```

参数

hinstDll 这个 DLL 的 module handle。

fdwReason	DllMain() 被调用的原因。可能是以下之一： DLL_PROCESS_ATTACH DLL_THREAD_ATTACH DLL_THREAD_DETACH DLL_PROCESS_DETACH
lpReserved	提供更多信息以补充 fdwReason。如果 fdwReason 是 DLL_PROCESS_ATTACH，那么 lpReserved 为 NULL 表示 DLL 是被 LoadLibrary 载入，non-NULL 则表示 DLL 是被隐式载入（implicitly loaded，也就是在链接时期以 import library 和程序链接在一起）。

返回值

如果 fdwReason 是 DLL_PROCESS_ATTACH，那么 DllMain() 应该在成功情况下传回 TRUE，并在失败情况下传回 FALSE。如果 DLL 是被隐式链接而 DllMain()传回的是 FALSE，程序将没有办法执行下去。如果 DLL 是被显式链接而 DllMain()传回 FALSE，那么 LoadLibrary() 会传回 FALSE。

如果 fdwReason 不是 DLL_PROCESS_ATTACH，那么返回值会被忽略。

? FAQ 49：如果一个新的线程使用了我的 DLL，我如何被告知？

任何时候，当一个进程载入或卸载一个 DLL 时，DllMain() 函数会被调用。线程也是一样。当一个进程开始执行时，它所用到的每一个 DLL 的 DllMain() 都会被系统调用之，并获得 DLL_PROCESS_ATTACH 消息。如果是线程开始执行，进程所用到的每一个 DLL 的 DllMain() 也都会被系统调用之，并获得 DLL_THREAD_ATTACH 消息。

列表 14-1 是一个非常非常小的 DLL 的源代码。只要 DllMain() 被调用，它便印出状态消息，它也提供一个输出函数（exported function）：TheFunction()。程序可以动态链接之并调用之。这一版的 DllMain() 并没有做任何实际工作，只是打印状态消息而已。

列表 14-1 ENTRY.CPP，来自 SMALLDLL 范例程序

```
#0001  /*
#0002   * Entry.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 14, Listing 14-1
#0006   *
#0007   * Demonstrate a very simple DLL that prints
#0008   * status messages when its functions are called
#0009   * a provides a single entry point called
#0010   * TheFunction() for test purposes.
#0011   */
#0012
#0013  #define WIN32_LEAN_AND_MEAN
#0014  #include <stdio.h>
#0015  #include <stdlib.h>
#0016  #include <windows.h>
#0017
#0018
#0019  BOOL WINAPI DllMain(
#0020      HINSTANCE hinstDLL,  // handle to DLL module
#0021      DWORD fdwReason,     // reason for calling function
#0022      LPVOID lpReserved )  // reserved
#0023  {
#0024      DWORD tid = GetCurrentThreadId();
#0025
#0026      // Why are we being called?
#0027      switch( fdwReason )
#0028      {
#0029          case DLL_PROCESS_ATTACH:
#0030              printf("DLL:\tProcess attach (tid = %d)\n", tid);
#0031              break;
#0032
#0033          case DLL_THREAD_ATTACH:
#0034              printf("DLL:\tThread attach (tid = %d)\n", tid);
#0035              break;
#0036
#0037          case DLL_THREAD_DETACH:
#0038              printf("DLL:\tThread detach (tid = %d)\n", tid);
#0039              break;
```

```
#0040
#0041          case DLL_PROCESS_DETACH:
#0042             printf("DLL:\tProcess detach (tid = %d)\n", tid);
#0043             break;
#0044      }
#0045      return TRUE;
#0046  }
#0047
#0048  _declspec ( dllexport ) BOOL TheFunction()
#0049  {
#0050      printf("DLL:\tTheFunction() called\n");
#0051      return TRUE;
#0052  }
```

为了确实掌握这些程序代码的操作，我写了一个小驱动程序 Main1，显示于列表 14-2 中。主线程会自动附着（attaches）DLL，因为程序和这个 DLL 链接在一起。当程序开始执行，另一个线程被产生，调用 DLL 中的 TheFunction()，然后程序结束。

列表 14-2 MAIN1.CPP，来自 SMALLDLL 范例程序

```
#0001  /*
#0002   * Main1.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 14, Listing 14-2
#0006   *
#0007   * Driver to load the simple DLL, create a
#0008   * thread, call a function in the DLL, and exit.
#0009   */
#0010
#0011  #define WIN32_LEAN_AND_MEAN
#0012  #include <stdio.h>
#0013  #include <stdlib.h>
#0014  #include <windows.h>
#0015
#0016
#0017  _declspec(dllimport) BOOL TheFunction();
#0018  DWORD WINAPI ThreadFunc(LPVOID);
```

```
#0040
#0041          case DLL_PROCESS_DETACH:
#0042             printf("DLL:\tProcess detach (tid = %d)\n", tid);
#0043             break;
#0044      }
#0045      return TRUE;
#0046  }
#0047
#0048  _declspec ( dllexport ) BOOL TheFunction()
#0049  {
#0050      printf("DLL:\tTheFunction() called\n");
#0051      return TRUE;
#0052  }
```

为了确实掌握这些程序代码的操作，我写了一个小驱动程序 Main1，显示于列表 14-2 中。主线程会自动附着（attaches）DLL，因为程序和这个 DLL 链接在一起。当程序开始执行，另一个线程被产生，调用 DLL 中的 TheFunction()，然后程序结束。

列表 14-2 MAIN1.CPP，来自 SMALLDLL 范例程序

```
#0001  /*
#0002   * Main1.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 14, Listing 14-2
#0006   *
#0007   * Driver to load the simple DLL, create a
#0008   * thread, call a function in the DLL, and exit.
#0009   */
#0010
#0011  #define WIN32_LEAN_AND_MEAN
#0012  #include <stdio.h>
#0013  #include <stdlib.h>
#0014  #include <windows.h>
#0015
#0016
#0017  _declspec(dllimport) BOOL TheFunction();
#0018  DWORD WINAPI ThreadFunc(LPVOID);
```

```
DLL:    TheFunction() called
DLL:    Thread detach (tid = 54)
DLL:    Process detach (tid = 180)
```

我们从上面的输出中获得的第一个重点是，DllMain() 分别被以 DLL_PROCESS_ATTACH 和 DLL_THREAD_ATTACH 各调用一次。这令我惊讶，因为我明明有两个线程。噢，原来 Win32 定义的这个机制，使得每一个程序的第一个线程调用 DllMain() 时，是以 DLL_PROCESS_ATTACH 调用之。所有后续的线程才是以 DLL_THREAD_ATTACH 调用之。

第二个重点是，DllMain() 系在新线程的 context 中被调用。这一点非常重要，因为你需要一个 context，才能够使用线程局部存储（TLS）。关于 TLS，稍后我再来讨论。

抑制通告消息（Disabling Notifications）

关于“DllMain() 在线程的 context 中被调用”这件事，有一些分歧。很明显地，程序 MAIN1 从未直接调用 DllMain()。DllMain() 是自动被调用，你可以视之为 CreateProcess()（那是 Windows 调用的）或 CreateThread()（那是 MAIN1 调用的）的副作用。现在我们看看，如果有 5 个、10 个、甚至 20 个 DLLs 被附着到进程之中的情况。每当你启动一个新线程，每一个 DLLs 的 DllMain() 都会被调用。突然之间你发现多了好多预期之外的负担。

为了避免这个问题，Win32 提供了一个 DisableThreadLibraryCalls() 函数。该函数只在 Windows NT 中才有，在 Windows 95 中没有效用。你可以一一指定不需要通告消息的 DLLs。如果一个程序常常产生新的线程，这么一点小小的最优化操作会节省不少的负担唷。

```
BOOL DisableThreadLibraryCalls(
    HMODULE hLibModule
);
```

参数

hLibModule DLL 的 module handle.

返回值

如果成功，则 DisableThreadLibraryCalls 传回 TRUE。否则传回 FALSE。如果失败，可调用 GetLastError() 获得详细信息。如果你所指定的 DLL 使用了线程局部存储（TLS），这个函数调用一定会失败。

如果我们修改 SMALLDLL 中的 ENTRY.CPP，将通告消息抑制掉，程序代码像这样：

```
case DLL_PROCESS_ATTACH:
   DisableThreadLibraryCalls(hinstDLL);
   printf("DLL:\tProcess attach (tid = %d)\n", tid);
   break;
```

当我完成这样的修改并重新执行 MAIN1 时，我得到下面的输出（这份修改代码并没有放在书附盘片上）。从中我想你看到了，程序继续接收 DLL_PROCESS_ATTACH 和 DLL_PROCESS_DETACH，但是不再接收线程的相关通告消息。

程序输出（在加入 DisableThreadLibraryCalls() 之后）：

```
Dll:    Process attach (tid = 190)
       Thread launched
       Thread running
Dll:    TheFunction() called
Dll:    Process detach (tid = 190)
```

通告消息（Notifications）的问题

如果你以为每一件关于 DLL 通告消息的东西都井然有序，那么让我把你拉回现实。我第一次写 MAIN1 时忘了加上 WaitForSingleObject()，于是 main() 在 CreateThread() 之后立刻结束。于是我获得这样的输出：

程序输出（WaitForSingleObject() 不存在的情况）：

```
DLL:    Process attach (tid = 185)
        Thread launched
DLL:    Process detach (tid = 185)
```

这很有趣。我没有收到“线程附着”的通告消息，而 TheFunction() 也没有印出该有的消息。似乎，Win32 在 ExitProcess() 函数中对每一个正在运行的线程分别调用了 TerminateThread()。怎么会扯到 ExitProcess()？那是你的 main() 结束时或你调用 exit() 时，由 C runtime library 调用的一个函数。

如果你还记得，第 2 章我曾经说过 TerminateThread() 的危险性。其中之一就是它阻止了 DLL 通告消息，于是 DLL 就没有办法知道一个线程或是一个进程被“解除附着”的时机。由于 TerminateThread() 是在一个进程结束之前被使用，程序没有安全关闭所有正在运行的线程，可能会无意识地留下任何已附着的 DLLs，并使它们处于不稳定状态。

动态载入的效果

? FAQ 50: 为什么我在写 DLL 时需要小心所谓的动态链接？

书附盘片中有第二个驱动程序，名为 MAIN2，以动态链接（而非静态链接）方式载入 DLL。除了 LoadLibrary() 和 FreeLibrary() 的输出信息之外，MAIN2 的输出和 MAIN1 的输出完全相同：

MAIN2 程序输出：

```
        Calling LoadLibrary()
DLL:    Process attach (tid = 169)
        Thread launched
DLL:    Thread attach (tid = 166)
        Thread running
DLL:    TheFunction() called
DLL:    Thread detach (tid = 166)   <-- 译注：原书少了这一行
        Calling FreeLibrary()
DLL:    Process detach (tid = 169)
```

如果程序先启动线程，然后才调用 LoadLibrary()，我们看看会发生什么状况。MAIN3 程序就是这么做的，而我们会收到一个“线程解除附着”消息，但没有收到“线程附着”消息。记住，所有驱动程序（从 MAIN1 到 MAIN3）都使用相同一个 DLL。

MAIN3 程序输出：

```
      Thread launched
      Calling LoadLibrary()
      Thread running
DLL:   Process attach (tid = 178)
DLL:   TheFunction() called
DLL:   Thread detach (tid = 154)
      Calling FreeLibrary()
DLL:   Process detach (tid = 178)
```

当一个 DLL 被 LoadLibrary() 或 LoadLibraryEx() 动态载入时，DllMain() 不会收到任何正在执行的线程的 DLL_THREAD_ATTACH 通告消息，但有一个线程除外，那就是调用 LoadLibrary() 的那一个（译注）。不过，DllMain() 倒是会收到所有那些线程的 DLL_THREAD_DETACH 通告消息。不论你是否认为这样有用，反正技术文件上公开这么说。

译注 如果是这样，上述 MAIN3 程序的输出不就应该有 DLL_THREAD_ATTACH 通告消息吗？的确是，以下是我的实际执行结果（看来是作者的疏忽了）：

```
      Thread launched
      Calling LoadLibrary()
      Thread running
DLL:   Process attach (tid = 178)
DLL:   Thread attach (tid = 154)     <-- 这是作者疏忽的一行
DLL:   TheFunction() called
DLL:   Thread detach (tid = 154)
      Calling FreeLibrary()
DLL:   Process detach (tid = 178)
```

初始化失败

最后还有一个关于通告消息的特性是你必须知道的．如果 DllMain() 收到 DLL_PROCESS_ATTACH 时因不能够正确初始化而传回 FALSE，DllMain() 还是会收到 DLL_PROCESS_DETACH．因此，你必须非常小心地在 FALSE 被传回之前，将每一个有价值的变量初始化为一个已知状态，如此一来，DLL_PROCESS_DETACH 的处理函数才不会因为乱指的指针或是不合理的数组索引而当掉．

通告消息（Notification）摘要

由于上述讨论的那些题目，你必须因此非常小心地安排你的 DllMain() 函数．理想中你最好是像我那样，写一些简单的驱动程序，确保 ATTACH 和 DETACH 通告消息都能够被正确地处理——甚至即使它们遗漏掉了或是乱了次序．

- 如果进程调用 LoadLibrary 时，有一个以上的线程正在运行，那么 DLL_THREAD_ATTACH 不会针对每一个线程送出．（译注：只有调用 LoadLibrary 的那个线程才会发出）
- DllMain() 不接受第一个线程的 DLL_THREAD_ATTACH，而以 DLL_PROCESS_ATTACH 取代之．
- DllMain() 不接收任何因 TerminateThread() 而结束之线程的 DLL_THREAD_DETACH 通告消息．如果程序调用 exit(1) 或 ExitProcess() 结束自己，这种情况就会发生．

DLL 进入点的依序执行（Serialization）特性

我们还没有完成对 DLL 进入点的讨论．Win32 设计之初，下面的问题必须解决：

1. 线程 1 调用 LoadLibrary()，会送出 DLL_PROCESS_ATTACH．
2. 当 DLL 正在处理 DLL_PROCESS_ATTACH，线程 2 调用 LoadLibrary()，于是送出 DLL_THREAD_ATTACH．

3. DllMain()开始处理 DLL_THREAD_ATTACH——在 DLL_PROCESS_ATTACH 未完成之前.

? FAQ 51：为什么我在 DllMain 中启动一个线程时必须特别小心？

一个 race condition 于焉诞生，操作系统不能给你任何帮助．为了解决这个问题，Win32 必须依序调用所有 DLLs 的 DllMain() 函数．在一个进程之中，一次只有一个线程能够执行一个 DLL 的 DllMain()．事实上，每一个线程依次调用每一个附着的 DLLs 的 DllMain() 函数．

偶尔你会进入一种副作用之中．如果你在 DllMain() 内产生一个线程，该线程可能没有办法在其他数个 DllMain() 完成之前顺利初始化它自己．因此，你不可能在 DllMain() 中启动一个线程，等待它初始化，然后才继续执行下去．

译注 关于 DllMain()的 Serialization 性质，请参考 "Advanced Windows 3rd edition" 的第 12 章：Dynamic Link Libraries。

MFC 中的 DLL 通告消息（Notifications）

在 MFC 4.2 中，在一个使用 MFC 的一般性 DLL（而非用来扩充 MFC 的所谓 extension DLLs）之中，DLL 通告消息有一些巧妙．MFC 提供它自己的 DllMain()，用来适当初始化 MFC 本身．除非使用未公开函数，否则你没有办法以自己的函数取代之。

一个使用 MFC 的 DLL，拥有它自己的 CWinThread 对象，那可视为 CWinApp 对象的一部分．当 DLL 接收到 DLL_PROCESS_ATTACH 时，MFC 就会调用 InitInstance()．当 DLL 接收到 DLL_PROCESS_DETACH 时，MFC 就会调用 CWinThread::ExitInstance()．你可以提供自己的两个函数，因为它们都是虚函数．然而，没有任何虚函数在 DLL_THREAD_ATTACH 和 DLL_THREAD_DETACH 发生时被调用．如果你需要在一个 MFC 程序中接收这两个消息，请参阅 Microsoft Knowledge Base Q148791 号文章．

当那些以 CWinThread 产生出来的线程启动时，MFC 4.x 也执行了一些特殊处理。由于前述的 DllMain() 的 Serialization 性质，所以你绝不可能在一个 DLL 的 CWinThread::InitInstance() 中启动一个线程。如果你尝试这么做，DLL 一定会在初始化过程中冻住不动！

喂食给 Worker 线程

让我们分一些时间来讨论如何在 DLL 中产生自己的 worker 线程。一个必须解决的关键问题就是，如何告诉线程它要做些什么事。消息队列是一个很方便的机制，你可以针对 worker 线程定义一些消息，并利用 WPARAM 和 LPARAM 指向一个已配置妥当的数据结构。

其间的障碍是，worker 线程并不拥有任何窗口，也因此其他线程没有办法靠窗口 handle 把消息传递（“send”或“post”）过去。这对 GUI 线程并不构成问题，因为 GUI 线程有消息队列，也有窗口。

同样的问题也发生在一个 DLL 身上，因为不管是多线程情况或不是，DLL 都不会拥有窗口。然而，毕竟 worker 线程和 GUI 线程之间的差异是非常小的（译注：在操作系统的眼中没有区别，是 MFC 把它们这么分）。如果你在一个 worker 线程中调用 GetMessage()，那么消息队列便会产生（译注），纵使你并未拥有窗口。第 11 章也曾经展示过 PostThreadMessage()。

译注 我想你一定会觉得奇怪吧，消息队列有就有，没有就没有，怎会动态产生？如果看过 Windows 95 System Programming SECRETS（Matt Pietrek/IDG Books）第 3 章对于 Thread Database（TDB）的挖掘，你就会知道，消息队列在 Win32 环境中是一个链表（listed list），而不是像 Windows 3.x 的一个数组。所以，动态产生是可能的。

列表 14-3 是一个拥有自己主消息循环的 worker 线程。尽管是放在标题为 DLL 的本章，这一段程序代码和 DLL 并没有关系。不过相同的观念可以被放到 DLL 中，完全一样。

当主线程启动时，它产生一个 worker 线程，然后等待 worker 线程将一个 event 对象激发，代表线程的初始化操作完毕。worker 线程启动后，调用一个 USER 函数，强迫产生一个消息队列，然后激发 event 对象，通知主线程继续执行下去，然后要求系统每两秒送出一个 WM_TIMER 到 worker 线程的消息队列来。

主线程也立刻以 PostThreadMessage() 放置三个消息到 worker 线程的消息队列中，然后等待 worker 线程的结束。PostThreadMessage() 要求第一个参数为线程 ID（而不是窗口 handle），用以识别哪一个消息队列。Worker 线程会处理三个发自主线程的消息，以及发自系统的一些 WM_TIME 消息，然后结束。以 PostThreadMessage() 传送过来的消息，其窗口 handle 将是 NULL。

译注 以下是 WORKER 的执行结果：

```
Thank you for buying this book.

.drawrof daer ot reisae si texT
lower case is for whispering.
```

列表 14-3 WORKER.CPP，一个拥有消息循环的 worker 线程

```
#0001  /*
#0002   * Worker.cpp
#0003   *
#0004   * Sample code for "Multithreading Applications in Win32"
#0005   * This is from Chapter 14, Listing 14-3
#0006   *
#0007   * Demonstrate using worker threads that have
#0008   * their own message queue but no window.
#0009   */
#0010
```

```
#0011  #define WIN32_LEAN_AND_MEAN
#0012  #include <stdio.h>
#0013  #include <stdlib.h>
#0014  #include <windows.h>
#0015  #include <process.h>
#0016  #include <string.h>
#0017  #include "MtVerify.h"
#0018
#0019  unsigned WINAPI ThreadFunc(void* p);
#0020
#0021  HANDLE ghEvent;
#0022
#0023  #define WM_JOB_PRINT_AS_IS          WM_APP + 0x0001
#0024  #define WM_JOB_PRINT_REVERSE        WM_APP + 0x0002
#0025  #define WM_JOB_PRINT_LOWER          WM_APP + 0x0003
#0026
#0027  int main(VOID)
#0028  {
#0029     HANDLE hThread;
#0030     unsigned tid;
#0031
#0032     // Give the new thread something to talk
#0033     // to us with.
#0034     ghEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
#0035
#0036     hThread = (HANDLE)_beginthreadex(NULL,
#0037                    0,
#0038                    ThreadFunc,
#0039                    0,
#0040                    0,
#0041                    &tid );
#0042     MTVERIFY(hThread);
#0043
#0044     // This thread has to wait for the new thread
#0045     // to init its globals and msg queue.
#0046     WaitForSingleObject(ghEvent, INFINITE);
#0047
#0048     // The only place in the book we get to use
#0049     // the thread ID!
#0050     char *szText = strdup("Thank you for buying this book.\n")
```

```
#0051     PostThreadMessage(tid, WM_JOB_PRINT_AS_IS, NULL, (LPARAM)szText);
#0052
#0053     szText = strdup("Text is easier to read forward.\n");
#0054     PostThreadMessage(tid, WM_JOB_PRINT_REVERSE, NULL, (LPARAM)szText);
#0055
#0056     szText = strdup("\nLOWER CASE IS FOR WHISPERING.\n");
#0057     PostThreadMessage(tid, WM_JOB_PRINT_LOWER, NULL, (LPARAM)szText);
#0058
#0059     WaitForSingleObject(hThread, INFINITE);
#0060
#0061     CloseHandle(hThread);
#0062
#0063     return 0;
#0064  }
#0065
#0066  VOID CALLBACK TimerFunc(
#0067      HWND hwnd,  // handle of window for timer messages
#0068      UINT uMsg,  // WM_TIMER message
#0069      UINT idEvent,   // timer identifier
#0070      DWORD dwTime )  // current system time
#0071  {
#0072      UNREFERENCED_PARAMETER(hwnd);
#0073      UNREFERENCED_PARAMETER(uMsg);
#0074
#0075      PostThreadMessage(GetCurrentThreadId(), WM_QUIT,0,0);
#0076  }
#0077
#0078  /*
#0079   * Call a function to do something that terminates
#0080   * the thread with ExitThread instead of returning.
#0081   */
#0082  unsigned WINAPI ThreadFunc(LPVOID n)
#0083  {
#0084      UNREFERENCED_PARAMETER(n);
#0085
#0086      MSG msg;
#0087
#0088      // This creates the message queue.
#0089      PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE);
#0090
#0091      SetEvent(ghEvent);
```

```
#0092
#0093     // We'll run for two seconds
#0094     SetTimer(NULL, NULL, 2000, (TIMERPROC)TimerFunc);
#0095
#0096     while (GetMessage(&msg, NULL, 0, 0))
#0097     {
#0098         char *psz = (char *)msg.lParam;
#0099         switch(msg.message)
#0100         {
#0101         case WM_JOB_PRINT_AS_IS:
#0102             printf("%s", psz);
#0103             free(psz);
#0104             break;
#0105         case WM_JOB_PRINT_REVERSE:
#0106             printf("%s", strrev(psz));
#0107             free(psz);
#0108             break;
#0109         case WM_JOB_PRINT_LOWER:
#0110             printf("%s", _strlwr(psz));
#0111             free(psz);
#0112             break;
#0113         default:
#0114             DispatchMessage(&msg);
#0115         }
#0116     }
#0117
#0118     return 0;
#0119 }
```

线程局部存储（Thread Local Storage，TLS）

我要以“TLS 不是什么”作为这一节的开始。当我第一次听到所谓的TLS，我想那是一种内存保护组织，从中配置得到的内存只在特定的线程中才能够被使用，其他线程对它则是“没办法”。这样的观念其实错误，完全错误！

TLS 是一种机制，藉由它，线程可以持有一个指针，指向它自己的一份数据结构拷贝。C runtime library 和 MFC 都使用 TLS。C runtime library 把 errno 和 strtok() 指针放在 TLS 之中，因为它们的状态必须保留，不能够被其他线程干扰。如果每一个线程自己有一份数据副本，就不会有问题。

MFC 使用 TLS 来追踪每一个线程所使用的 GDI 对象和 USER 对象。CWnds、CPens 以及其他结构只能够使用于产生它们的那些线程之中，关于这一点，MFC 是非常斤斤计较的。如果使用 TLS，MFC 就可以验证对象是不是在线程之间传递。

线程局部存储(TLS)的运作方式是，每一个线程有一个由 4 字节槽(slots)所组成的数组。这个数组保证至少有 TLS_MINIMUM_AVAILABLE 个槽在其中。目前的操作系统保证至少有 64 个槽。每一个槽可被指定放置任何特殊结构。如果结构 WordCount 放在槽 4，那么每一个线程就可以配置一个 WordCount 结构空间，并设定让 TLS 槽 4 指向它。

TLS 对于 DLL 特别有价值，因为它的存在，DLL 不需要不断地要求调用端函数传送一个指向“thread context”的指针过来。如果一个函数库被数百甚至数千个地方调用，TLS 的价值就完全彰显了，因为在所有那些调用处加上一个 context 指针，几乎是不可能办到的。

在你询问有关于“针对每一进程而准备的储存空间”之前，请记住，所有被配置的内存，甚至是 DLL 所配置的内存，都是在调用端进程的 context 中配置的。也请记住，你的 DLL 会获得每一个调用端进程的所有全局变量的一份拷贝。啊，感谢操作系统和虚拟内存的神奇魔法！除非你在“阻遏那些保护措施”的环境下工作，否则你的 DLL 可以很舒服地忽略“它在同一时间被许多程序使用”的这个事实。

设定线程局部存储（TLS）

TLS 的使用起始于 TlsAlloc()。TlsAlloc() 在 TLS 数组中配置一个槽，并传回其数组索引。每一个线程可以自己拥有一份槽内容的拷贝，但是所有拷贝都必须使用同一个槽索引。

```
DWORD TlsAlloc(
    VOID
);
```

返回值

如果成功，传回 TLS 数组中的一个槽。如果失败，传回 0xFFFFFFFF。调用 GetLastError() 可获得失败的详细信息。

在 DLL 之中，典型的用法就是在收到 DLL_PROCESS_ATTACH 时调用此函数。调用一次只能获得一个槽。你可以把所有线程都可以使用的那些全局变量指定到配置而来的槽之中。

有一点很重要，你只配置一个槽就好，而它可以被你的所有数据结构使用。槽的个数是有限的，如果你有一个大程序，需要链接许多个 DLLs，那么 TLS 槽用光就不是没有可能了。如果每个 DLL 为自己配置 4~5 个槽，情况会更糟。

一旦你拥有一个槽，每一个新线程（包括第一个）必须配置一块内存，用来储存线程的局部数据。如果你有数个结构，请把它们封装在一个大结构中，然后你必须为 TLS 槽设定一个值，给目前的线程使用。我们使用的函数是 TlsSetValue()，规格如下：

```
BOOL TlsSetValue(
    DWORD dwTlsIndex,
    LPVOID lpTlsValue
);
```

参数

dwTlsIndex	由 TlsAlloc() 传回的 TLS 槽索引.
lpTlsValue	要储存到上述槽中的数据值.

返回值

如果成功，传回 TRUE. 如果失败，传回 FALSE. 调用 GetLastError() 可获得失败的详细信息。

让我们看看如何进行. 当 DLL 获得 DLL_PROCESS_ATTACH 时，调用 TlsAlloc()，假设传回 4. DLL 把此值储存于一个全局变量之中，然后配置一块内存，假设在 0x1F0000 处. 最后，DLL 调用 TlsSetValue，让槽 4 能够和地址 0x1F0000 产生关系（针对目前的线程而言）.

当下一个线程附着到同一个 DLL 时，DLL 知道已经有一个槽被配置过了，因为全局变量之中已经有值（槽编号），所以它再配置一块内存，假设在 0x203000 处. 为了完成对此线程的初始化操作，DLL 调用 TlsSetValue()，使得槽 4 能够和地址 0x203000 产生关联——只要这个线程执行起来.

现在每一个线程都有它自己的一块空间了，它可以调用 TlsGetValue() 取得其中内容：

```
LPVOID TlsGetValue(
    DWORD dwTlsIndex
);
```

参数

dwTlsIndex	由 TlsAlloc() 传回的 TLS 槽索引.

返回值

如果成功，则传回以 TlsSetValue() 设定在槽中的值. 如果失败，传回 0. 调用 GetLastError() 可获得失败的详细信息. 如果槽中储存的值真的是 0，那么 GetLastError() 会传回 NO_ERROR.

当第一个线程调用 TlsGetValue(4)时，它会获得 0x1F0000。当第二个线程调用 TlsGetValue(4)时，它会获得 0x203000。

一旦 DLL 最终获得了 DLL_PROCESS_DETACH 通告消息，它应该调用 TlsFree() 释放槽：

```
BOOL TlsFree(
    DWORD dwTlsIndex
);
```

参数

dwTlsIndex 由 TlsAlloc() 传回的 TLS 槽索引。

返回值

如果成功，则传回 TRUE。如果失败，传回 FALSE。调用 GetLastError() 可获得失败的详细信息。

隐私权是绝对必要的

一个线程不能够处理另一个线程的 TLS 槽。其实如果能够在程序结束前允许我们这么做，毋宁是比较好的，你可以因此走访所有的 TLS 数组并确定它们都被适当地清理了。

解决之道是维护你自己的表格，内放每一个配置而得的内存区块。虽然系统无论如何会为你清除实际内存内容，但你可以因为关闭文件或其他 handles 而避免数据遗失或过期。

一个 TLS 索引只在同一个进程中才有意义。如果你在不同的进程中以同样的逻辑使用同一个 TLS 索引，没有人能够保证它的效果。

在 DLL 启动时使用 TLS

? FAQ 52:

我如何在 DLL 中设定一个 thread local storage(TLS)?

列表 14-4 展示一个例子，示范如何设定 DLL，使之支持 TLS。下面的程序代码完全遵循以上所说的逻辑大纲。

列表 14-4 DLL 之中标准的 TLS 起始代码

```
#0001  // This is the shared slot
#0002  static DWORD gdwTlsSlot;
#0003
#0004  BOOL DllMain(
#0005           HINSTANCE hinst,
#0006           DWORD fdwReason,
#0007           LPVOID lpReserved)
#0008  {
#0009     LPVOID lpData;
#0010     UNREFERENCED_PARAMETER(hInst);
#0011     UNREFERENCED_PARAMETER(lpReserved);
#0012
#0013     switch( fdwReason )
#0014     {
#0015        case DLL_PROCESS_ATTACH:
#0016           // Find the index that will be global for all threads
#0017           gdwTlsSlot = TlsAlloc();
#0018           if (gdwTlsSlot == 0xFFFFFFFF)
#0019              return FALSE;
#0020
#0021           // Fall through to handle thread attach too
#0022
#0023        case DLL_THREAD_ATTACH:
#0024           // Initialize the TLS index for this thread.
```

```
#0025            lpData = (LPVOID)LocalAlloc(LPTR, sizeof(struct OurData));
#0026            if (lpData != NULL)
#0027               if (TlsSetValue(gdwTlsSlot, lpData) == FALSE)
#0028                  ; // This should be handled
#0029            break;
#0030
#0031         case DLL_THREAD_DETACH:
#0032            // Release the allocated memory for this thread.
#0033            lpData = TlsGetValue(gdwTlsSlot);
#0034            if (lpData != NULL)
#0035               LocalFree((HLOCAL) lpData);
#0036            break;
#0037
#0038         case DLL_PROCESS_DETACH:
#0039            // Release the allocated memory for this thread.
#0040            lpData = TlsGetValue(gdwTlsSlot);
#0041            if (lpData != NULL)
#0042               LocalFree((HLOCAL) lpData);
#0043
#0044            // Give back the TLS slot
#0045            TlsFree(gdwTlsSlot);
#0046            break;
#0047
#0048         default:
#0049            break;
#0050      }
#0051
#0052      return TRUE;
#0053   }
```

_declspec(thread)

另有一个简单的方法，可以降低程序员的负担，又做到线程局部存储的要求。微软的 Visual C++ 允许一个变量或结构被声明为“具有线程局部性”.

例如，下面的声明，如果放在一个 DLL 之中，将产生出一个全局变量，对每一个进程而言独一无二：

```
DWORD gProgressCounter;
```

但是，如果这样声明，它就是对每一个线程独一无二喽：

```
_declspec(thread) DWORD gProgressCounter;
```

这样的机制也可以应用在结构体上．例如

```
struct _ScreenObject
{
   RECT     rectBoundingBox;
   COLORREF crColor;
   POINT   *ptVertices;
}

_declspec(thread) struct _ScreenObject BouncingPolyhedron;
```

把数值或结构体放置于 TLS 上，是编译器的责任。许多情况下，这么做的确是比 TLS 函数好用得多。DllMain() 并不需要知道 TLS 中所有数据的结构长什么样子，也不需要产生一个超级结构来含括所有其他结构。每一个结构体可以声明说它将在哪里被需要；编译器会负责把它们放在一起，放进 TLS 中。

每一个以这种方式声明对象的 EXE 和 DLL，将在可执行文件中有一个特殊的节区（section），内含所有的线程局部变量．当 EXE 或 DLL 被载入时，操作系统会认识这个节区并适当地处理之．这个节区会被操作系统自动设定为“对每一个线程具有局部性”．

当一个 EXE 被载入时，操作系统扫描可执行文件以及所有静态链接（译注，作者指的是 implicitly linked）的 DLLs，以便找出所有的线程局部节区。所有节区的大小被加总在一起以求出每个线程启动时应该配置的内存数量。

Tls...() 函数以及 _declspec(thread) 并不冲突，可以安全地混用．你并没有被限制说只能使用哪一种技术。

限制

? FAQ 53：_declspec (thread) 的限制是什么？

如果你使用 C++，你所产生的类需要一些限制。如果一个对象拥有构造函数或析构函数，它就不能够被声明为 _declspec(thread)。因此，你必须在线程启动时自己动手将对象初始化。

注：Visual C++ 编译器从 1.0 到 4.2 一直存在一个“臭虫”，使得你无法在对象拥有任何 private 或 protected 成员时，声明该对象为 _declspec(thread)。请参阅 Microsoft Knowledge Base 的 C2483 号文章。

另一个限制比较严苛些。一个 DLL 如果使用了 _declspec(thread)，就没有办法被 LoadLibrary() 载入。因为线程局部节区的大小计算是在程序启动时完成的，没有办法在一个新的 DLL 载入时重新计算。虽然 LoadLibrary() 在 Windows NT 中通常会成功，但是当 DLL 之中的函数被调用而开始执行，就会产生一个“protection fault”。请参阅 Microsoft Knowledge Base 的 Q118816 号文章。

数据的一致性

我们已经在本书的其他地方谈过，如何以同步机制保护你的变量。由于 DLL 被设计为可共享，有时候甚至可被重复使用，或被视为元件，所以你绝不可能知道一个程序开发人员打算在同一时刻以 4 个或 5 个或更多线程来调用你的 DLL。

这个问题对于 OLE 和 ActiveX 特别重要，在其中，元件的重复使用是一个更容易被清楚认知的目标。使用控件（controls）或 automation 对象是如此地简单，意味着开发人员可能会在一个古怪的环境中尝试去使用它们。

若想要生产 DLLs 以便安全地在多线程环境中使用，下面是几个大方针：

- 不要使用全局变量。用来储存 TLS 槽（slot）者例外。
- 不要使用静态变量。
- 如有必要，尽量使用 TLS（Thread Local Storage）。
- 如有必要，尽量使用你的堆栈。

如果你来自 Win16 的世界，你可能会做出所有可能的妥协，让 DLL 能够有效运作。特别是当你面临数据节区和堆栈节区之间的差异时。一个典型的 Win32 程序至少拥有 1MB 堆栈，而且所有指针都是 32 位，所以要放一个 16K 对象到堆栈之中，毫无问题。

提要

你在这一章中看到了如何在 DLL 中处理通告消息，那是被用来指示线程或进程的附着（attach）或解除附着（detach）的行为。你看到了那些通告消息带来的一些重大限制，你必须小心地处理它们。你也看到了如何使用线程局部存储（TLS）以及为什么要使用它。最后，你看到了 _declspec(thread)，那是由编译器支持的一种制造“线程局部存储（TLS）”的方法。

第三篇

真实世界中的多线程应用程序

Multithreading in Real-World Applications

第 15 章　规划一个应用程序 397
第 16 章　ISAPI 415
第 17 章　OLE，ActiveX，COM 429

规划一个应用程序

Planning an Application

本章提供一些指南，告诉你哪一种程序最适合使用多线程，以及如何才能够让它们有最佳结构。这一章有数个迷你讨论，主题包括对多线程程序设计的影响、第三方（third party）的函数库产品，以及应用程序的架构。

在软件世界中，有些新技术会突然发烧，然后又快速灰飞烟灭。多线程程序设计有点这种味道.线程早在 Unix 及其他操作系统中就已经以某种形式出现过了，在实验室中的生命更久远。然而，如果你研究它，你会发现线程几乎都只是在理论或操作系统的领域中被讨论，很少有人提到如何把线程应用到实际程序中。

多线程的理由

? FAQ 54: 我应该在什么时候使用多线程?

我们已经看到了一些不使用多线程的理由。例如，Windows 消息架构已经做了很好的努力，将 MDI 的管理打破为更小块的工作。尝试为每一个 MDI 子窗口使用一个线程，是灾难的开始。如果只因为很方便，我们就任意产生线程，那是个坏点子。

那么，线程好在哪里?

线程实际应用于四个主要领域。任何应用程序都可以被归类为其中某些领域。在每个领域之中存在着或可争辩的重叠关系。

1. offloading time-consuming task。由主线程（GUI 线程）负责，让用户界面有较好的反应。
2. Scalability。
3. Fair-share resource allocation。
4. Simulations。

对专家而言

> 为了此处所设定的讨论目的，我把沉重的并行处理（parallel processing）问题视为和 Windows 多线程完全不同的问题。

如果你来自 Unix 世界，我一定得指出，Win32 的线程远比 Unix 中的进程价廉许多。在 Unix 中你原本以多进程和 IPCs 技术解决的问题，现在在 Win32 中可以靠多线程解决。稍后我再来谈这些。

Offloading Time-Consuming Tasks

我们早在第 11 章就花了一些时间在这个题目上。主消息循环的目标是快速反应大部分的 Windows 消息，不因为消息队列的阻塞而导致程序停滞。如果程

序要花 15 分钟才能计算出窗口重绘内容，用户就没有办法选择【File/Exit】立即退出程序，除非有什么办法可以让消息队列仍可操作。

在 Win16 中最普遍的做法就是调用 PeekMessage() 检查队列中是否有消息，以适当地反应它们。这个技术很难有效率地实现出来，因为 PeekMessage() 必须以一定的时间间隔被调用。如果太常调用，程序会遭受效率上的报应。如果调用频率不够，程序反应不足，又显得有点迟疑不决。

在 Win32 中，如果一个 worker 线程做掉那些苦工，主线程和消息循环就可以有理想的表现。然而，最简单的脚本也需要小心地规划。你得让用户觉得你（程序）的确正在进行“正确”的事情，你得让用户安心。用过那种“从来不出现等待光标”的试用版软件吗？你坐在那里东按按西按按，希望程序给你一些反应，而程序却没有任何回应。你于是开始嘀咕：“它当掉了吗？我要不要重新开机？”

多线程程序在幕后执行，应该给用户一些更新消息。“等待光标”并不是个合理的解决之道，因为真正的工作在后台执行，而光标应该只能够显示前台所发生的事情。

想想微软的 Word。当它在后台打印时，你可以在状态栏中看到一些动画：打印机一页一页地吃进纸张。为了证明打印动作的确在进行，而不只是卡通画面，Word 也显示了打印的页数。甚至即使打印了半小时，用户还是很安心，因为他知道一切都正常运作。而且他可以获得其他信息，如花掉了多少时间等等。

让我们看另一个例子。一个像片修饰软件常常需要处理很大的 bitmap 图，2048×2048 并带真实色彩的图片（一张可能需要 12MB）是很稀松平常的。程序没有办法在两秒钟或三秒钟之内就把它完全载入进来。事实上如果机器比较慢或是网络塞车的话，开启这样一张图片文件可能需要好几分钟。因此，“打开文件”操作似乎宜由另一个线程进行。

有数种方法可以表示操作持续在进行，它们统统需要更新屏幕。由于我们希望“打开文件”线程是一个 worker 线程而不是 GUI 线程，所以需要在 worker 线程和主 GUI 线程之间有一些通讯管道。或许我们可以用一个 event

对象对某些数据做记号，然而，从 worker 线程中传送一个消息到主线程的主窗口，大概是最容易的了。

工作状态可以藉由一个计量器或一个百分比进度条，或其他方法来表现。要记住的是，用户可能同时开启数个文件，因此，在状态栏中以一个指示器来表示，并不是最理想的方式。

在 client/server 应用程序中，产生报表也是 worker 线程的一份理想工作。关键的考虑在于这份报表是在本地（local）的 server 产生，还是在远端（remote）的 server 产生。

如果报表是在本地（local）产生，那么这的确是一件适合移往另一个线程的工作。制作报表往往需要数小时，让一个程序“打结”数小时可不是件妙事。

如果报表是在远端（remote）产生，那么或许可以设定一个通告机制，让主消息循环在报表完成时获得通知。使用“pooling”方式来检查对方是否完成工作，是最糟糕的行为。

Scalability

Scalability 意味着程序的效能可以因为增加 RAM、增加磁盘空间、增加 CPU 个数而改善。对于此处的讨论而言，最重要的就是 CPU 个数的增加。我在第 1 章曾经提过所谓的 SMP，也就是“Symmetric Multi-Processor”。单 CPU 系统是以快速切换线程来完成多任务的假象，而 SMP 系统是真正在同一时间执行一个以上的线程，每个 CPU 负责一个线程。

Windows NT 就是 SMP 系统。如果你的程序非常依赖计算（也就是 CPU-bound），那么增加 CPU 个数将是提升效能的一个好方法。

要让 SMP 系统发挥效能，你的应用程序必须支持多线程。如果程序只有一个线程，它还是只能使用一个 CPU。

一个程序如果能够处理数件毫不相干的工作，它就是 scalable 系统的最佳受益人。服务器常常就是这种情况。一个数据库服务器，一个 Web 服务器，或是一个

制造报表的所谓 report 服务器，都是在处理一大堆不相干的请求（requests）。每一份工作应该都能够适当地由一个线程来完成。

另一种应用程序也很容易被“scalable”，就是那种只有一份工作，但很容易被细切的程序。最好的例子就是光迹追踪法（ray tracing）。这种程序必须执行大量的计算，以获得目标图形的每一个像素（pixel）内容。每一个像素可以独立计算，不需依赖其他像素。因此，如果你把 CPU 个数从 1 增加到 2，并且让每一个 CPU 负责一个线程，光迹追踪法的效率真的会提高两倍。

我要强调，线程只不过是一种机制，用来切割工作，使每一小份工作可以被一个 CPU 处理。如果在单 CPU 机器中，在光迹追踪法应用程序里产生多个线程，效率反而会变差，这是因为系统需要额外的负担以管理线程。

大部分的 GUI 程序都不是 scalable，因为程序用户一次只能够做一件事情。电子表格（spread sheet）可以在同一时间被重算、重画、打印，但是这些工作彼此有密切的关联，因为它们统统都是在同一份文件上工作。一个厉害的用户或许可能在同一时间里打开三到四份文件，但是这和服务器每分钟处理数百个请求是不一样的。除了一些例外，如 CAD 软件（其计算有高度并行性），一般 GUI 程序并不适用于此一类型。

在 scalable 环境上使用线程，很容易就可以让程序以最优化的方式充分使用硬件。

Fair-share Resource Allocation

甚至即使在 SMP 系统，一个沉重负荷的（或说饱和的，saturated）服务器仍将使用每一点每一滴的 CPU 时间来服务每一项请求。如果你的服务器有“scalability”能力，你可以花钱解决这事：买更多硬件就是了。然而，如何让这些负荷沉重的服务器公平对待每一项请求，则又是另一个问题。问题也就变成了：一个客户端发出请求之后，多少时间才能获得服务。

如果你限制你所服务之请求的最大并发（concurrent）个数，那么你可以为每个请求指定一个线程，操作系统的调度算法可以帮助你让每一个请求以相同的

优先权被处理；没有任何特殊请求可以把服务器推入动弹不得的泥淖之中。限制“请求的最大个数”是很重要的，因为太多线程会使系统产生极端的“thrash”现象（译注）。如果有 500 个线程，每秒发生 500 次 context switches，那么每个线程每次只能获得比 2ms 更少的 CPU 时间。（译注：很多时间都浪费在 context switch 切换操作上了）。

译注 下面是 Dictionary of Computing 4th edition（OXFORD 出版）对于“thrashing”的解释：

Thrashing: A phenomenon that may arise in paging or other forms of virtual-memory system. If the page-turning rate for a paging system becomes high, usually because the amount of real memory available for holding pages is small compared with the total working set of all the processes currently active, then each process will find itself in a situation in which, on attempting to reference a page, the appropriate page is not in memory. In trying to find space to hold the required page, the system is likely to move out onto backing store a page that will very shortly be required by some other process. As a consequence the paging rate rises to very high levels, the fraction of CPU cycles absorbed in managing page-turning overheads becomes very high, processes become blocked as they wait for page transfers to complete, and system throughput falls sharply.

如果适当实施线程优先权策略，你可以确保“应获得更多重视的线程”的确得到较多的 CPU 时间，而不是被一视同仁地对待。例如，我们可以将系统设定为“只要是来自特定某一群工作站的请求，一定被优先处理”。使用线程优先权，你可以让 Win32 带来你所想要的结果。

Simulations

使用多线程的第四个理由，同时也是最后一个理由，便是仿真推导。这时候 scalability 和 fairness 都不再是兴趣焦点，真正重要的是提供独立的演员，可以和这场仿真游戏中的其他演员互动。

线程在仿真方面的价值，主要是供给开发者一个方便。其实没有什么是线程做得到而一般循序程序做不到的事。

仿真一条高速公路，跑 10000 辆汽车，每一辆车以一个线程表示，这种仿真几乎注定要失败。如果仿真一个停车场，上有 100 辆车，每一辆车以一个线程表示，就比较可能成功。

要线程还是要进程？

下一个问题是，使用多线程好还是使用进程好？尽管本书的重点在线程，我还是要说，有时候使用进程比使用线程适当。

整个讨论之中你必须牢记在心的是，进程是一种代价昂贵的东西。它们在启动、清除、被操作系统追踪记录等方面，都很昂贵。进程和线程之间的代价差异，几乎相差 2~3 order。因此，系统同时存在 500 个线程犹有可能，但同时存在 500 个进程则绝无可能。每秒钟产生和摧毁许多个线程是可能的，每秒钟产生和摧毁许多个进程则是不可能的。

使用多进程的一个考虑因素是应用程序的完整性。在一个多线程程序中，必须有某种层次的信赖感存在，信赖一个线程不会当掉而将其他 50 个线程拉下海。如果整个程序是由一组人马完成的，当然我们有这种信心。但如果你允许加入外部 DLLs，或是有内嵌的程序语言的话，风险就会增加许多。

考虑一下这种情况。你必须使用一段前人留下来的程序代码，它不是十分稳定，但此刻没有时间改写它。如果把这段代码执行于另一个进程之中，你就对伤害有所控制；当这个进程毁了时，操作系统会清除所有尚未关闭的文件和尚未释放的对象。

虽然本书对于安全（security）谈得很少，但我要说，安全性在进程之间是固有的，是天生的，比起线程之间好得多。在一个多线程程序中，所有全局数据对所有线程都是开放的。虽然这在大多数时候是个优点，但对于一个必须很坚固的系统而言，这可能潜在性地招来很大的风险。如果以进程来区隔那些数据，你就产生了一道坚固的藩篱。

多线程程序的架构

设计多线程应用程序的架构时，有三个观念必须常驻于心。我们早已多次在整本书中看过了这些观念，所以我将选择一些范例来示范这些题目。

1. 在线程之间尽量保持最小接触面积。
2. 将 race conditions 的可能性降至最低。
3. 程序代码要有自我防卫性。

第 2 章中我们启动一个后台打印工作，做法是收集一个新线程需要的所有信息，然后把这些信息统统传送到一块区块中。这种做法并不十分有效率，但却十分容易了解，并且也很安全。这个程序是上述第 1 点和第 2 点的极端例子，其主（GUI）线程和 worker 线程之间的接触面积是 0。Worker 线程有它所需要的每一项信息，不需和主线程分享任何东西。主线程则负责处理所有和顺序特别有关的动作。当 worker 线程启动，没有什么和顺序有关的事情会留给它处理。

上述第 3 点可使用像 MTVERIFY 这样的代码而获得，就像本书的其他许多范例程序一样。最不可能出错的东西也有可能在多线程环境下出错，所以务必测试你的每一项假设。

我曾经在第 7 章中谈论如何有效地保护你的数据，避免 race condition 的侵扰。如果操作系统有什么机制能够做你所需要的事情，使用它，不要犹豫。例如，你可以使用“anonymous pipes”，不必再自己写个多线程的环状缓冲区。Pipe 或许不是最快、最精致、最有能力的解决方法，但是它们已经被完成、被测试、被确定在多线程环境中的正确性了。如果效率是你最关心的问题，你可以重写这段代码，否则你最好省点力气。

切割模块

在操作系统的设计理念中，有两个常用的方法用来设计核心。由于多线程是操作系统的天性，于是这也就提供了一个活生生的例子，告诉我们如何组成一个多线程系统。

第一种设计称为 monolithic。在一个 monolithic 系统之中，有一块巨大的代码，用来执行操作系统的所有功能。每样东西都牵扯到每样东西！往往没有合适的界面定义出 monolithic 核心与子系统间的关系。monolithic 核心是最容易实作的一种，因为架构的问题很容易获得解决。

Monolithic 核心终于因为它自己的重量而倒塌。所有对局部瑕疵的修正都导致对其他部分的干扰。问题愈来愈难追踪，因为机能持续扩展并且相互依赖。

Monolithic 核心与众多元件之间有很大的接触面积。几乎不可能对它们完全调试并保证其可依赖性。

Microkernel 架构则是完全相反的极端。这种操作系统的核心倾向于极小化，并且只负责核心机能，像是内存管理啦、进程/线程的调度啦等等。其他的操作系统机能如文件系统、磁盘驱动程序、通讯驱动程序都是在它们自己的进程之中运行，只能够以严谨定义的界面来存取之。Microkernel 架构比较强固（robust），比较可依赖，但是如果没有小心调整，可能在效率方面有比较不佳的表现。

Microkernel 架构的关键就在于分离的元件。这很类似于面向对象设计中的“对象”观念：改变的范围都是局部性的，改变所引起的涟漪有最小化的倾

向。

设计你自己的多线程程序时，请努力遵循 microkernel 架构。虽然它需要更多的前置设计，但是会在维护性以及可信赖度上面回报你。

评估既有程序代码的适用性

如果一个程序从头开始就能够以多线程的方式来设计，很好。但实际情况却是，我们常常需要将一个原有的程序翻新为多线程程序。我将尝试在这一节中提供一些指南，看看如何评估原有的代码适不适合做转换。

? FAQ 55: 我能够对既有程序代码进行多线程操作吗?

第一件要评估的便是全局变量和静态变量的使用度高不高。把状态信息放在全局变量或静态变量中，是 C 程序一种广被使用的手法。然而这种做法会毁掉一个多线程程序。比较常见的就是全局性文件指针或全局性 handles。C++ 程序中比较少这种问题，那是因为大多使用成员变量的缘故。

一如我们在第 8 章所见，C runtime 函数库在许多 runtime 函数中使用静态缓冲区，例如 asctime() 便是。除非你使用多线程版的 C runtime 函数库，否则不同的线程在同一时间调用 asctime() 将会引起奇怪的结果。

在 runtime 函数库中此一问题的解决方法是在 thread local storage（TLS）中配置一个结构空间，取代函数中所使用的静态缓冲区。每一个线程于是有了这个结构空间，并可以在其中放置那些“视各线程之不同而不同”的静态变量和全局变量。例如，当你调用多线程版的 asctime()，你会获得一个指针指向 TLS，而此结果将被放在成员变量 _asctimebuf，而不是函数内的一个静态局部变量。在 Visual C++ 4.2，此结构被设计于 MTDLL.H 文件，显示于列表 15-1。请注意，和 CPU 相关的栏位已经被我略去。

从这份列表，你可以遍览结构内容，并从中获得一些灵感，知道 runtime 函数库必须为每个线程各自留下哪些数据。

列表 15-1 C runtime library 所使用的 thread local structure

```
/* Structure for each thread's data */

struct _tiddata {
    unsigned long   _tid;        /* thread ID */
    unsigned long   _thandle;    /* thread handle */
    int             _terrno;     /* errno value */
    unsigned long   _tdoserrno;  /* _doserrno value */
    unsigned int    _fpds;       /* Floating Point data segment */
    unsigned long   _holdrand;   /* rand() send value */
    char *          _token;      /* ptr to strtok() token */
#ifdef _WIN32
    wchar_t *       _wtoken;     /* ptr to wcstok() token */
#endif /* _WIN32 */
    unsigned char *  _mtoken;    /* ptr to _mbstok() token */

    /* following pointers get malloc'd at runtime */
    char *          _errmsg;     /* ptr to strerror()/_strerror() buff */
    char *          _namebuf0;   /* ptr to tmpnam() buffer */
#ifdef _WIN32
    wchar_t *       _wnamebuf0;  /* ptr to _wtmpnam() buffer */
#endif /* _WIN32 */
    char *          _namebuf1;   /* ptr to tmpfile() buffer */
#ifdef _WIN32
    wchar_t *       _wnamebuf0;  /* ptr to wtmpfile() buffer */
#endif /* _WIN32 */
    char *          _asctimebuf; /* ptr to asctime() buffer */
#ifdef _WIN32
    wchar_t *       _wasctimebuf; /* ptr to _wasctime() buffer */
#endif /* _WIN32 */
    void *          _gmtimebuf;  /* ptr to gmtime() structure */
    char *          _cvtbuf;     /* ptr to ecvt()/fcvt buffer */

    /* following fields are needed by _beginthread code */
    void *          _initaddr;   /* initial user thread address */
    void *          _initarg;    /* initial user thread argument */

    /* following three fields are needed to support signal handling */
    void *          _pxcptacttab;    /* ptr to exception-action table */
    void *          _tpxcptinfoptrs; /* ptr to exception info ptrs */
    int             _tfpecode;       /* float point exception code */
```

```
    /* following field is needed by NLG routines */
    unsigned long    _NLG_dwCodee;

    /*
     * Per-Thread data needed by C++ Exception Handling
    */
    void *          _terminate;     /* terminate() routine */
    void *          _unexpected;    /* unexcepted() routine */
    void *          _translator;    /* S.E. translator */
    void *          _curexception;  /* current exception */
    void *          _curcontext;    /* current exception context */
};
```

另一个常常被放在全局变量中的数据是错误标记，例如 runtime 函数库中的 errno。你可以看到它被包含在列表 15-1 结构中的 _terrno 栏位。这样的解决方案对大部分程序可行，但是 errno 应该以某个函数取得，而不是放在一个大家可以自由拿到的变量中。既有程序代码可以利用 #define 技巧，使 errno 变量自动转换为一个函数调用。

另一件重要的评估标准是：使用了哪些无源代码的函数库（.LIB）或目标文件（.OBJ）。这可能不太容易发现，因为它们都被记录在 makefile 中。如果你找到这样的模块而的确没办法取得其源代码，或许你得在调用它们之前先包装一个 mutex。毕竟你没办法改写它们，不是吗？

使用既有程序代码，最令人感到挫败的，大概就是如何保护出现在各处的数据了。每一个被共享的 array、linked list、hash table，都必须以一个同步机制保护起来。最麻烦的是如何实施所谓的“logically atomic”（逻辑上不可分割）的动作，类似 SQL transaction。后者需要许多改变数据的操作，才能使数据具有一致性。

Windows 95 设计小组尝试让 16 位程序和 32 位程序共存时，也面临相同的问题，而他们的解决办法就是把所有东西都拿到 16 位代码上执行。Windows 95 使用一个 global mutex 来解决问题。只要有一个 16 位程序正在执行一个 API，它就会获得这个 mutex。当这个 global mutex 被设立，没有任何其他程序（包括 32 位程序）能够调用任何 API。

上述的 global mutex 就是所谓的 Win16Mutex。作者说“当这个 global mutex 被设定，没有任何其他程序（包括 Win32 程序）能够调用任何 API”，基本上这样的描述不够精准。

译注 让我从头讲起。

Windows 95 的 USER32.DLL 和 GDI32.DLL 中绝大部分 Win32 API 函数都调用 USER.EXE 和 GDI.EXE 中对应的 Win16 API 函数。在这个范围内，上述作者所言是正确的。但是 KERNEL32.DLL 并不调用 16 位的 KRNL386.EXE（除了极少数例外）。

如果有一个 Win32 线程不牵扯 Win16 API，就不会受到 Win16Mutex 的影响。Worker 线程就是这种情况。

我写了几个小程序来测试上述论点的正确性。操作系统是 Windows 95。首先我写两个 Win32 console 程序，内容都一样：

```
#0001  #include <stdio.h>
#0002
#0003  main()
#0004  {
#0005      int i;
#0006
#0007      for (i=0; i<0xFFFF; i++)
#0008      {
#0009          printf("i= %d \n", i);
#0010      }
#0011  }
```

此二程序不会彼此影响，我们的确看到了抢先式多任务的现象。

接下来我写两个 Win32 程序，内容都一样，当菜单中的 “long chunk” 项目被按下，就进入一个循环，并把循环计数器输出到窗口左上角：

```
#0001  case IDM_LONGCHUNK:  // long chunk 运算
#0002      {
#0003          int i;
```

```
#0004          char str[20];
#0005          HDC hdc = GetDC(hWnd);
#0006
#0007          for (i=0; i< 0xFFFF; i++)
#0008          {
#0009             wsprintf(str, "%010d", i);
#0010             TextOut(hdc, 10, 10, str, 10);
#0011          }
#0012
#0013          ReleaseDC(hWnd, hdc);
#0014          MessageBeep(0);
#0015       }
#0016       break;
```

此二程序亦不会彼此影响，我们可看到抢先式多任务的现象。

如果我把上述 Win32 程序改以 16 位工具编译链接，得到 Win16 程序，结果当我选按菜单中的“long chunk”项目，系统进入循环后，就不再允许任何其他操作（包括切换其他程序，或是缩放窗口大小……等等，唯鼠标仍可自由移动）。

如果我先执行前述两个 Win32 程序，它们会同时计数。在它们尚未结束前，我再执行上述的 Win16 程序，结果该 Win16 程序立刻抓住所有 CPU 时间，两个 Win32 程序立刻停下来。直到该 Win16 程序完毕，那两个 Win32 程序才继续其未完之工作。

接下来我再做一个实验。设计一个多线程程序（注意，要使用编译器选项 /MD 和多线程版的 C runtime 函数库 MSVCRT.LIB），使得一旦用户按下菜单中的“long chunk”项目，程序就产生一个 worker 线程，线程函数如下：

```
#0001  VOID ThreadFunc(VOID)
#0002  {
#0003     unsigned long i;
#0004
#0005     for (i=0; i< 0xFFFFFF; i++)
#0006     {
#0007
#0008     }
#0009
```

```
#0010      MessageBeep(0);
#0011   }
#0012   //注意：不要使用编译器最优化，否则上面那个空的循环可能会被编译器干掉.
```

此函数之执行，在我的机器上大约耗时 1 秒。如果选按“long chunk”之后立刻执行前述的 Win16 程序，则会受到该 Win16 程序的影响。必须等到该 Win16 程序结束，才会听到 MessageBeep() 发出声音。

如果该线程真的是 worker 线程，就不应该受到“素行不良”（如上述例子）之 Win16 程序的影响。但我们看到的结果却是被影响了。所以，是否 MessageBeep() 与 Win16Mutex 有关？我想是。

如何验证一个 worker 线程真的不受“素行不良”之 Win16 程序影响？我的实验方法是，把上述 worker 线程中的循环次数改为 0x7FFFFFF，在我的电脑上大约需时 7 秒。前述的 Win16 程序大约需要 14 秒时间才能运行完其循环。现在，我把有 worker 线程的 Win32 程序执行起来，然后立刻执行前述的 Win16 程序。后者一运行完立刻（而不是又过 7 秒之后）响起 worker 线程中的 MessageBeep() 声音。可见在过去的 14 秒之内，worker 线程的确有效运作，不受 Win16Mutex 被抓住的影响。

对 ODBC 做规划

? FAQ 56：我可以在我的数据库应用程序中使用多线程吗？

撰写多线程版本的数据库 client 程序将非常棘手。问题出在来自不同厂商的许多元件必须一起运作，而每一个都必须具备“多线程安全性”。下面是你决定是否可以对一个数据库 client 软件进行多线程设计时的一些影响因素：

1. 有一个以上的线程和数据库交谈吗？

如果你只有一个线程会和数据库交谈，那么可以不必担心。换句话说，即使你的 client 程序支持多线程，如果你只在单独一个线程中与 ODBC

驱动程序打交道，那么 ODBC 驱动程序绝对不会有问题。唯一例外是 DAO（Data Access Object），我将在第 4 点讨论它。如果你的 client 程序支持多线程能力，而且有数个线程同时在和数据库驱动程序交谈，那么事实上它们共享了数据库的 handles。

2. 你使用 JET 数据库引擎吗？

所谓 JET 数据库是 ODBC 和 DAO 用来与 Access 数据库、xBase 数据库、text 数据库、Excel 数据库交谈的一个数据库引擎。直到 3.00 版，JET 数据库仍然还不是多线程版本，不能够被多线程 client 程序使用。

3. ODBC 驱动程序有多线程安全性吗？

除非 ODBC 驱动程序特别声明它有多线程安全性，否则你应该假设它没有。微软的 SQL Server ODBC 驱动程序声称它有多线程安全性。

4. 你使用 DAO 吗？

DAO 是一个 OLE COM 对象，用来和数据库沟通。DAO 由 MFC v4.0 支持之，建立在 OLE 基础之上。在我下笔时刻，不仅 DAO 不具备多线程安全性，它甚且只能够在程序的主线程中被使用。

5. 你使用 MFC 吗？

虽然 MFC 已经支持 ODBC 好一阵子了，但是直到 1996/07 问世的 MFC v4.2，才把 ODBC 相关类如 CDatabase 和 CRecordSet 改为多线程版本。不过它们的所谓多线程安全性（thread-safe）也只适用于 MFC 自己的 C++ 类。MFC 使用标准的 ODBC 驱动程序，那些驱动程序必须对于 MFC 也具备“多线程安全性”才行。

MFC 4.2 的 DAO 类并未具备“多线程安全性”。第 16 章示范一个例子，告诉你在 IIS（Internet Information Server）上使用 ISAPI 时，如何避开这个问题。

第三方的函数库（Third-Party Libraries）

以今天对元件软件的重视度，以及软件元件的泛滥度而言，写一个应用程序而不至少使用一个第三方的元件，似乎是愈来愈困难了。那些元件可能是以下形式：

- 内存管理器（像是 MicroQuill 的 SmartHeap）
- OLE 控件（像是 Visual C++ 中的“calender”控件）
- 数据库驱动程序（ODBC、dbTools++ 等等）
- 报表产生器（像是 Crystal Reports）
- 影像处理函数库（像是 LeadTools）
- 通讯函数库
- Frameworks（例如 MFC）
- 更多更多其他的东西……

每一样东西都必须在多线程环境下分别验证其安全性以及能力。如果有怀疑，就应该联络产品公司，询问他们是否在多线程环境下做过测试。千万不要认为“我只负责用就是了”。

提要

这一章中你看到了一些主题，围绕在“写出新的多线程程序，或是修改原有程序成为多线程版本”上打转。你也看到了有关于“何时使用多个进程，何时使用多个线程”的讨论。你还看到了如何评估数据库存取代码，以及第三方的函数库。

ISAPI

这一章要为你展示一个 ISAPI extension DLL，适用于一个在 ODBC 规格下使用 JET 引擎的 web 服务器。

我想你一定已经听说过 Internet——一世纪末最后一件大事。气象频道有自己的 web 站点，波士顿有自己的 web 站点，甚至白宫也拿得出一个。当 Web 愈来愈普及时，web 服务器的效率也就愈来愈重要。微软设计 Internet Information Server（IIS）时，他们设计了一种方法，可以附加“服务器扩充软件（server extension）”，使其服务器效能比其竞争对手有戏剧性的进步。这项技术要求那些扩充软件支持多线程，并且，当然，在多线程环境下必须安全无虞。本章将带你看看如何建立一个服务器扩充软件，以 ODBC 规格来和 JET 数据库引擎沟通。

Web 服务器及其工作原理

对于那些从没有剖析过 web 服务器的人而言，这是一个简短的课程。服务器用来接收来自客户端的一个请求（request）。所谓客户端通常是一个 web 浏览器，像是 Microsoft 的 Internet Explorer 或是 Netscape 的 Navigator。所谓的请求（request）是文本模式，并且以一个 socket connection 传达。就像我在第 6 章所示范的 I/O completion ports 一样。

用户通常会指定一个 URL（Universal Resource Locator），像是 http://www.microsoft.com/visualc/vc42，意思是使用 http 通讯协议，站点名称为 www.microsoft.com，资源路径则位于站点下的 /visualc/vc42。

有了一个像这样的简单需求，服务器就会传回被索求的文件。这个文件可能还会指向其他文件，例如 bitmap 图片文件等等，浏览器会自动再为你索求那些关系文件。

有时候静态文字和影像不够，所以必须再做某些运算，以便再产生网页。这些运算可以靠 CGI（Common Gateway Interface）完成。要使用 CGI，客户端必须对一个程序（而不是一个 HTML 网页）发出一个请求。服务器看到 URL 指向程序，于是把 CGI 程序载入进来并执行之。例如，以下这个请求将执行 cgi-bin 子目录中的 search 程序：

```
http://www.bigcompany.com/cgi-bin/search?product=toaster
```

注意：Unix 并不使用 EXE 文件扩展名来表示程序是个可执行文件。文件之所以可执行，靠的是文件系统的许可。

当服务器收到一个请求，它把 CGI 程序执行起来，视为一个新进程。此新进程的环境变量（字符串）被设定为 URL 中所指定的任何参数。上面所举的 URL 例子会使得服务器执行一个名为 search 的程序，并传递“product=toaster”字符串当做一个环境变量。这个程序然后会产生回应，显示于 stdout，被“piped”回到服务器，然后再将信息送回给客户端。

这样的管理运作得十分良好，但是面对每一个请求，都必须有一个进程被产生然后被毁灭。CGI 界面系设计于 Unix 环境下，在那里，进程相当廉价，但也不是免费的。许多 CGI 程序以 Perl 语言撰写，那是一种变种语言。Perl 程序是解释（而非编译）形式，有较高的额外负担（overhead）。

ISAPI

微软在设计 IIS 时就已经很清楚效率上的问题了。微软的解决方案称为 ISAPI，也就是 Internet Server API。ISAPI 的实现品由 Windows NT Advanced Server 4.0 上的 IIS、Windows NT 4.0 Workstation 上的 Peer Web Services（PWS）、以及 Windows 95B 上的 Personal Web Server 提供。（注：Windows 95B 的“B”是一个 OEM 版本，并未对大众公开）

ISAPI 服务器扩充软件可以是一个 filter，也可以是一个 application。对一个 web 浏览器而言，filter 是不可见的。filter 被安装到 web 服务器中，处理（过滤）服务器所送出的每一页。相反地，一个 ISAPI application 必须明白地被一个 URL 提出请求；这个 URL 必须包含 ISAPI application 的名称和路径，有点类似 CGI scripts。

ISAPI 扩充软件是以 DLLs 而非 EXEs 的形式呈现。为了处理一个请求，服务器对着 DLL 发出一个调用，而不再需要产生一个新的进程。DLL 在第一次被使用时被系统载入，并且通常继续留在内存中。处理一个请求（request）所需要的启动工作和清理工作所导致的额外负担（overhead）几乎降至 0，内存需求量也巨幅下降。想象一下为每一个请求配置、初始化、载入 256K（或更多）数据空间与堆栈的惨状吧！我想你终于可以了解为什么许多 Unix web 服务器竟然需要 128MB 内存。

ISAPI DLL 和其服务器处于同一个地址空间中，所以在 ISAPI 扩充软件和服务器之间切换时并没有 context switch 发生。相反地，如果你使用 CGI script

将所有数据利用 stdout 写入服务器中，底层会发生好多好多次的 context switch。所以，在一个忙碌的服务器中，ISAPI DLL 对于效率的提升是十分明显的。

但是 ISAPI 架构也有一些缺点。由于没有进程边界的保护，一个 ISAPI DLL 可以遍访其服务器的整个数据空间。因此，DLL 如果发出不适当的存取操作，可能会把整个服务器搞当掉。但如果使用 CGI，由于有进程边界的保护，即使外界的请求失败了，服务器还是安全的。所以，ISAPI DLLs 一定得经过彻底的测试。

ISAPI 的另一个缺点是，它们的服务器扩充软件只能以 C 或 C++ 写成。这项限制使得 ISAPI DLLs 的写作成了系统程序员的专利。只熟悉 Perl 语言的人就没有办法写一个 ISAPI DLL。然而，IIS 和 PWS 还是容纳 CGI 的存在，所以 Perl 程序员还是可以写 Perl scripts，只是他们无法享受到 ISAPI 带来的效率优势（注）。

注：这项束缚可能会有所改变。当我下笔的同时，我已经看到“以 Perl 解释器为基础的 ISAPI”的一些研讨会，以及 Visual Basic 中的 ISAPI extension 的控件。

撰写 ISAPI DLLs

写一个 ISAPI DLLs 时，你绝对得特别注意资源的运用。一般程序中，如果你忘记调用 CloseHandle()，系统会在用户退出之后帮忙清理。然而，一个 Internet 服务器可能运行好几个月都不会停下来。如果一个 ISAPI 函数对于一个请求忘记释放一项资源——也许它是忘记调用 CloseHandle()，或是忘记调用 DeleteCriticalSection()，在忙碌的服务器中很可能一个小时内就遗失掉数百个甚至数千个 handles。终于，系统耗尽所有空间，动弹不得。

一个 ISAPI extension 所处理的请求数量很难事先掌握。写一个任务非常重要（例如证券交易所所使用）的软件，可能数个星期甚至数个月也没有办法停下来好好检修一下细部。再一次，我要强调，彻底的测试绝对是最重要的原则。

ISAPI extension 是一个 DLL 而不是一个进程，这也带来了一些问题。你绝对不会知道什么线程将被用来调用这个 DLL，也不知道是否有个主线程用来保持反应，或提供其他特殊功能。因此，像线程局部存储（Thread Local Storage，TLS）这样的机制就不显得特别有用，因为没有线程有任何明确的责任。甚至即使你尝试指定责任，你也没有办法控制其他个别线程的行为。

对专家而言

一个 ISAPI DLL 有可能产生它自己的线程，但是这么做时必须注意几点。启动线程的适当地点应该是在 DllMain()，在 DLL 被载入的时候。然而，你会遭遇一些问题，因为 DLL 的启动程序（startup）是异步的（请看第 14 章）。你也应该小心而优雅地把线程结束掉——如果它所维护的任何信息必须被保存下来的话。

IIS 之中的线程

现在我提出一个假设，假设 IIS 是以 I/O completion ports 完成的。这个立场有点危险。一如你在第 6 章所见，completion ports 意味着使用多线程。IIS 是全然的多线程环境，有能力同时处理许多个请求（requests）。因为这样，所以 ISAPI DLLs 必须面对多线程而保证安全（thread-safe），并且必须支持任意量的并行线程（concurrent threads）。

视 DLL 设计方式的不同，让它成为 thread-safe 可能十分简单，也可能十分困难。一个简单的 ISAPI 服务器扩充软件可能只不过是产生一张计数器图片。这种情况就不需要做任何同步控制，因为每一线程可以完全独立计算。

比较复杂的例子是库存管理系统，它必须存取数据库。这可能会非常复杂。不同的线程之间必须分享数据，以便真实世界中所看到的东西呈现一致性。我想你一定不希望看到不同的线程公布三样货物，而其实仓库中只有两样。

通常的解决方法就是挑选一个 thread-safe 数据库，使它在多线程的 IIS 环境中绝对安全，能够有效运作。我们曾经在第 15 章看过一些有关于使用数据库的题目。从微软的眼光来看，只有 SQL Server 的驱动程序才说得上是

thread-safe.

IS2ODBC 范例程序

本章程序示范以一个 non-thread-safe 数据库驱动程序搭配一个 ISAPI DLL 的可能方案. 程序 IS2ODBC (念做 IS-to-ODBC) 使用 IIS 之外的一个进程来执行数据库引擎. 对数据库服务器进程所发出的各项请求, 将被依序处理, 一次一个. 很明显这并不是一种高效率的解决方案. 但是它充填了在"完全没有数据库"和"采购、安装、维护 SQL 服务器"之间的一个大鸿沟.

要让整件事情能够有效运作, 一共有四大块, 分别是:

- Web 服务器
- "IS2ODBC" ISAPI DLL
- 数据库进程
- Web 浏览器

我将讨论每一块, 看看它们是如何工作, 如何和其他项目沟通的. 首先, 你或许对于这个扩充软件的执行感兴趣.

执行范例程序

恐怕 IS2ODBC 不可能赢得"安装容易"这样的赞美. 安装程序得视你使用哪一个服务器而定, 同时 ODBC 也必须正确地安装给 Microsoft Access 数据库使用. 这里有一个基本摘要适用于大部分的微软服务器. 视你的系统配置的不同, 可能会有一些关于使用权限和安全的额外议题需要注意.

在 Windows NT 4.0 Workstation 和 Windows 95B 中, Web 服务器并不是缺省安装项目. 你可以进入【控制面板】中的【网络】去安装它.

下面这些操作假设你在同一部电脑中执行服务器和浏览器。更详细的资料请参考书附盘片之 IS2ODBC 子目录中的 README.TXT 文件。

1. 像 READ ME 文件所说的那样，将数据库配置（configuration）调整好。
2. 把 IS2ODBC.DLL 拷贝到你的 scripts 子目录中。
3. 把 DBSERVER.EXE 拷贝到你的 scripts 子目录中。
4. 把 QUERY.HTM 拷贝到你的 wwwroot 子目录中。
5. 确定你的 Web 服务器正在执行。
6. 使用 http://localhost/Query.htm 命令，将 QUERY.HTM 载入浏览器。
7. 选择某个查询。

运作概观

下面是当 Web 服务器收到一个 URL（其中使用了 IS2ODBC）时的事件序列。为了方便讨论，我们假设这个 URL 是：

```
http://localhost/scripts/Is2Odbc.dll?Sort=Title
```

Host 的名称是“localhost”，那是你正在执行的机器的一个正式别名（alias）。ISAPI extension 的路径是“scripts/Is2Odbc.dll”，服务器发现那是一个 DLL 并假设它是一个 ISAPI DLL。最后，URL 指定参数为“Sort=Title”。稍后我会对这个参数有更多的描述。

Web 服务器首先确定 IS2ODBC.DLL 已经被载入，然后把查询交给那个 DLL。IS2ODBC 需要数据库进程 DBSERVER 以服务这个请求。如果该进程尚未启动，就把它执行起来。

IS2ODBC 然后开始等待，直到数据库进程不再忙碌，就把 URL 的请求交到 DBSERVER 手上，并等待回应。DBSERVER 处理这个请求，产生一份文本结果，并发出信号，要求 IS2ODBC 继续下去。IS2ODBC 取得这份文本结果，把它送回给 web 服务器。

IS2ODBC 和 DBSEVER 都是以 MFC 完成的。MFC 提供了一些辅助类，可以简化 ISAPI extensions 和数据库应用程序的撰写工作。

浏览器网页

每一件事情都从 Web 浏览器开始。图 16-1 显示此例的网页画面之一。这一页主要用来描述范例程序的大纲，然后允许用户选择三个 URL 之一，发出一个请求（request）。

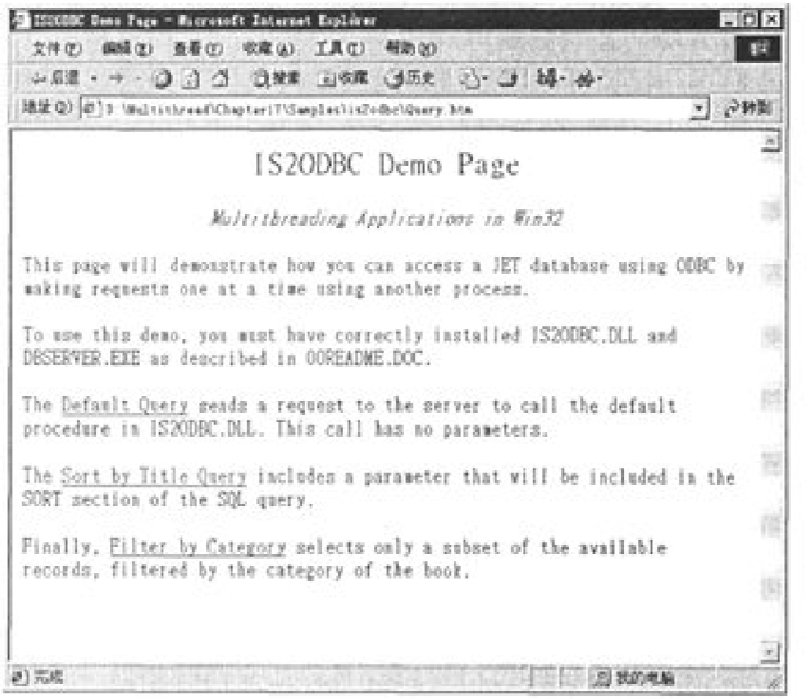


图 16-1 IS2ODBC 的网页画面

这个网页允许三个不同的请求：一个是默认查询（没有参数），一个是排序查询，另一个是过滤查询。这个网页可以利用 forms 加以扩充，如此一来可以对排序查询或过滤查询有更好的控制。

ISAPI 应用程序

范例程序 IS2ODBC 是以 Visual C++ 4.2 中的 ISAPI Extension Wizard 所提供的 CHttpServer 类来完成 ISAPI 扩充软件的功能。这个类处理许多杂务，确保每一件事情都能顺利运作。

当 Web 服务器从 Web 浏览器收到 URL 时，它看到 URL 引用到一个 DLL，于是把请求（request）交给该 DLL，这个时候 MFC 可以帮助你解析 URL 的内容，并把文本字符串转换为成员函数可以处理的参数。

你可以在 IS2ODBC.CPP 的起始处发现用来控制这个转换过程的所谓“parse map”。它显示于列表 16-1。ON_PARSE_COMMAND() 宏声明缺省函数有两个参数，都是 ITS_PSTR 字符串。ON_PARSE_COMMAND_PARAMS() 宏声明参数出现在 URL 上的名称。DEFAULT_PARSE_COMMAND() 声明哪一个函数可以在 URL 未指定函数名称时被调用之。我们并未在此例中使用函数名称，所以 Default() 总是会被调用。

这个 parse map 允许 MFC 调用 Default() 成员函数，那是用来处理外界请求的一个函数。

列表 16-1 IS2ODBC 中的 MFC parse map

```
#0001  BEGIN_PARSE_MAP(CIs2OdbcExtension, CHttpServer)
#0002      // Added Optional Filter and Sort arguments
#0003      //
#0004      ON_PARSE_COMMAND(Default, CIs2OdbcExtension, ITS_PSTR ITS_PSTR)
#0005      ON_PARSE_COMMAND_PARAMS("Filter=~ Sort=~")
#0006      DEFAULT_PARSE_COMMAND(Default, CIs2OdbcExtension)
#0007  END_PARSE_MAP(CIs2OdbcExtension)
```

数据交换

成员函数 Default() 的源代码显示于列表 16-2 中．你可以看到两个参数：pszFilter 和 pszSort，分别对应 URL 中的参数．你需要注意的第一件事情就是，此函数传回的是 void．这似乎颇为奇怪，因为这个函数应该是蛮容易失败的．然而，要知道，此函数的执行结果是产生一个网页．如果函数失败，它会产生一个网页用来描述错误．由于 Web 服务器完全不知道这个扩充软件做些什么事，所以服务器不可能产生一个有意义的错误信息出来．

函数 StartContent() 和 WriteTitle() 都是 MFC 提供的，用以在一个文本为主的网页中产生适当的 HTML 标记（HTML tags）．

列表 16-2 IS2ODBC 中的 URL 处理函数

```
#0001  void CIs2OdbcExtension::Default(
#0002        CHttpServerContext* pCtxt, LPTSTR pszFilter, LPTSTR pszSort)
#0003  {
#0004     StartContent(pCtxt);
#0005     WriteTitle(pCtxt);
#0006
#0007     CMutex myMutex(FALSE, MUTEX_DB_REQUEST);
#0008     CSingleLock myLock(&myMutex, FALSE);
#0009
#0010     // Wait to get access to DbServer
#0011     // If another request is outstanding then we will not get in.
#0012     if (myLock.Lock(TIMEOUT_MUTEX))
#0013     {
#0014        // Create events for signaling DbServer
#0015        CEvent startEvent(FALSE, TRUE, EVENT_START_PROCESSING, &sa);
#0016        CEvent doneEvent(FALSE, TRUE, EVENT_DONE_PROCESSING, &sa);
#0017
#0018        //
#0019        // Allocate memory for our data structure in shared memory.
#0020        //
#0021        HANDLE hFileMapping = NULL;
#0022        DbRequest* pDbRequest = NULL;
#0023
```

```
#0024       MTVERIFY( hFileMapping = ::CreateFileMapping((LPVOID) -1, &sa,
#0025            PAGE_READWRITE, 0, sizeof(DbRequest),
#0026            FILE_DB_REQUEST));
#0027
#0028       MTVERIFY( pDbRequest = (DbRequest*) ::MapViewOfFile(hFileMapping,
#0029            FILE_MAP_ALL_ACCESS, 0, 0, 0));
#0030
#0031         //
#0032         // Fill in our DB request.
#0033         // Make sure we detect the default argument "~" and replace it.
#0034         //
#0035         strcpy(pDbRequest->sqlFilter, "");
#0036         if (strcmp(pszFilter, "~") != 0)
#0037            strcpy(pDbRequest->sqlFilter, pszFilter);
#0038
#0039         strcpy(pDbRequest->sqlSort, "");
#0040         if (strcmp(pszSort, "~") != 0)
#0041            strcpy(pDbRequest->sqlSort, pszSort);
#0042
#0043         // Tell DbServer to do the DB Query
#0044         startEvent.SetEvent();
#0045
#0046         // Wait for result from DbServer
#0047         // If Server has died, we can tell user
#0048         CSingleLock waitForDoneEvent(&doneEvent, FALSE);
#0049         if (waitForDoneEvent.Lock(TIMEOUT_EVENT))
#0050         {   // got result from server
#0051           *pCtxt << _T("The following books match your search
                   request:<p>");
#0052           *pCtxt << pDbRequest->sqlResult << "<p>";
#0053         }   // end if
#0054         else
#0055         {   // Got no response from server
#0056           *pCtxt << _T("Sorry, but DbServer never responded with the
                   results.<p>");
#0057         }   // end else
#0058         doneEvent.ResetEvent();
#0059
#0060         // Close request data structure
#0061         UnmapViewOfFile(pDbRequest);
#0062         CloseHandle(hFileMapping);
#0063
#0064      } // end if
```

```
#0065     else
#0066     {   // Another Thread is busy accessing the server
#0067         *pCtxt << _T("Sorry, another Web Page is busy making a database
                request.<p>");
#0068     }   // end else
#0069
#0070     EndContent(pCtxt);
#0071  }
```

为了与 DBSERVER.EXE 交换数据，Default() 首先锁住一个 mutex，表示服务器正在忙．一次只能有一个 ISAPI 线程可以锁住这个 mutex，如此一来才能确保数据库的存取排他性．

接下来，Default() 产生一个具名的共享内存区域（named shared memory area），就像我们在第 13 章所讨论的那样．这个共享内存产生于页面文件（paging file）之中，所以 IS2ODBC 和 DBSERVER 不需要经过文件系统的同意配置空间．来自 URL 的参数被拷贝到共享内存中，于是 DBSERVER 可以经由它知道是什么样的查询．

Default() 然后又设定一个“startEvent”，告诉 DBSERVER 执行这个查询动作．DBSERVER 将执行结果拷贝至共享内存中，然后设立“doneEvent”．如果 DBSERVER 没有在适当的时间内作出反应，Default() 便假设某些事情出了问题，于是印出一个错误信息．

你也应该注意到，Default() 在等待 mutex 时指定了一个等待时间．如果它没办法在指定的时间 TIMEOUT_MUTEX（本例为 10 秒）内获得 mutex，用户就会被告知说服务器目前太忙碌，没有办法提供服务．如果服务器非常忙碌，许多线程都在排队等待 mutex 以准备使用 DBSERVER，这种情况就会发生．

提要

这一章给你一个 ISAPI extension 范例程序：IS2ODBC。它使用一个独立于 Web 服务器之外的进程，以使对数据库的请求（database request）可以安全送往一个并非 thread-safe 的数据库中。这个程序使用本书的许多观念，包括线程同步化、进程通讯，以及 MFC 同步控制类。

OLE，ActiveX，COM

这一章描述 COM 的多线程模型，并展示一个程序，示范所谓的 free-threading。也讨论了 apartment model 和 free-threading 之间的差异，以及它们在一个 DLL 和一个 EXE 之中的运作情况。

译注　本章中一再出现“interface”这个名词。由于它在 OLE/ActiveX 中有特殊的意义，所以我保留原文，不翻译为“界面”。Interface 其实就是 C++ 类中的虚函数表（vtbl）。

虽然这本书讲的是多线程，而不是 OLE 或 ActiveX，但我想如果这一章从“什么是 OLE、ActiveX 和 COM”开始讲起，应该会比较合适些。如果你对 OLE 已经十分熟悉了，请直接跳过此节，进入“COM 线程模型”那一节。

OLE 是 Object Linking and Embedding 的缩写。对于 20 世纪 90 年代的 1.0 版，OLE 这个字眼可以理想地描绘出技术与目标。例如，它允许你（程序开发者）将 Microsoft Excel 的一份电子表格，嵌入或链接到你所设计的一个新程序里头。

今天，OLE 还是保有那个功能，但是做的事情更多了。总称为 OLE 的这一整组技术包括 drag and drop（对象拖放）、structured storage（结构化存储）、in-place activation（实地激活）、uniform data transfer（统一数据传输），以及许多其他的 interfaces。OLE 其实就是一堆标准的 interfaces，用来完成各种工作，它们统统建立在 COM（Component Object Model）的基础之上。稍后我会对 COM 谈更多一些。这些 interfaces 允许 Word 和 Excel 彼此交谈，允许 Visual Basic 和绘图控件以及行事历控件交谈，允许在 WordPad 和桌面之间拖放对象。由于这些新功能和嵌入、链接几乎没有什么关系，所以微软发展出一个新名词：ActiveX，用来描述这些技术。

大部分对 ActiveX 的讨论都围绕在 Internet 打转。ActiveX 事实上是建立在 COM 基础上的所有技术的总称。换句话说，微软希望 OLE 真正只用来代表其字面意义的“链接与嵌入”技术。但因为 ActiveX 崛起比较晚，许多技术文件都还是以 OLE 涵括所有那些技术。因此你在文件中读到的术语可能不会一致。如果你还是感到迷惑，只要想 OLE 和 ActiveX 是一组建立在 COM 基础之上的技术即可。

COM 的全名是 Component Object Model。描述 COM 的最简单说法就是，它以 C++ 类的形式，提供对象的操作界面，允许你调用 C++ 类的成员函数——跨越应用程序边界、跨越 16/32 位边界，甚至跨越网络和 CPU 类型的边界。你可以询问一个 COM 对象是否支持某一特定的 interface，如果它支持，就会传回一个指向该 interface 的指针给你。另外有一些函数可以让 COM 对象描述其 interfaces，使它们能够被 Visual Basic 程序或 Java 程序调用之。

COM 既是一个规格，也是一组据此规格实现出来的 Windows services。COM 被实作为数个 DLLs，就像 GDI 或 USER 的角色一样。一个 COM 对象就是一个根据 COM 规格撰写出来的对象。COM 提供一些函数，可以找出 COM 对象并与之联络。

举个例子吧。你可以对 COM 说“为我产生一个 Excel document 对象”，于是 COM 会启动 Excel，产生一个 document，然后把一个指向 C++ 类的指针传回来。这个 C++ 类实现出 IUnknown，那是 COM interface 中最基本的一

个．以 C++ 的术语来说，IUnknown 是基类，所有其他 interfaces 都植基于其上．有了 IUnknown，你就可以要求此一对象所支持的任何其他 interfaces．

COM 的观念可能一时之间难以领悟，原因是 C++ 不能够直接解释 COM 背后的基础观念．在 COM 之中你只能够处理 interfaces，不能够处理对象本身．真实世界中的例子大概是，如果你问航空站在哪里，可能会有人告诉你到哪里找到入境厅．你还是不知道航空站在哪里，但是有什么关系呢？你已经获得了所有必要的信息．入境厅当然通往航空站．

在 COM 之中，对象本身有点像航空站，interfaces 就像是入境厅．当你向 COM 要求一个对象，它只给你 interface；它绝不会给你对象指针．在 C++ 中，这种情况可以用“一个拥有 public 成员函数，但未拥有 public 成员变量的 C++ 类”表示之．指向 interface 的指针并不指向对象本身，interface 的观念并不存在于任何语言之中．最值得注意的是 Java，COM 和 Java 可以良好地共存合作．

当你拥有一个对象的 interface，你就能够调用该 interface 拥有的任何函数．一个 interface 可以是 OLE 技术的一部分，或者也可以是你所产生的某些东西．COM 要解决的一个关键问题便是，如何跨越进程边界，使得某个进程中的对象可以使用生存于 DLL，或另一个进程，或甚至另一台机器（也许使用不同种类的 CPU）上的对象的 interface．如有必要，COM 可以透明地在各地址空间之间移动形参（parameters）和实参（arguments）．

COM 的线程模型（COM Threading Models）

OLE 1.0 版早在 Win16 就开始了，那是早在大部分人听过线程和 Win32 之前．当 COM 在 OLE 2.0 引进时，有些人试着在一个新操作系统 Windows NT 3.1 上作业，但大多数人还是固守在 16 位 Windows．因此，根本没有所谓的多线程模型（threading model）．

但是今天却有三个不同的多线程模型。我将对每一个模型做个快速简介，然后再对最后两个模型做详细说明。第一个模型是第二个模型的退化。

第一个模型来自 Win16，并且现今仍是缺省状态下所使用的模型。这是“单线程模型（single-threaded model）”。一个 OLE server 只在单线程下执行，不需要任何其他的东西。很明显地，它不支持多线程。

第二个模型适用于 Windows 95 和 Windows NT 3.51，称为“apartment model”（套间模型）。在此模型中每一个线程都是一个套间，COM 对象可以驻留其中。一个 COM 对象必须在这套间中完成其所有工作。因此，任何特定的 COM 对象都可以在单独一个线程中进行。在 Windows NT 和 Windows 95 中，一个进程如果使用“单线程模型”，我们可以说那是一个拥有“单一套间”的进程。

Apartment model 可以支持多线程，但是因为一个特别的对象必须在一个特别的线程中执行，所以它并不能够因为多个 CPU 而受惠（所谓“scalable”）。对于一个像 OLE 数据库这样的技术而言，如果数据库被强迫在单一线程中运作，它将必须循序处理每一个请求。

最后一种线程模型，第一次出现于 Windows NT 4.0，并不适用于 Windows 95。这个最新的模型称为 "free-threaded model"（自由模型）。在此模型之中，任何对象可以在任何线程中执行。

在我开始详细探讨那些模型之前，我应该先从 OLE 的角度来描述所谓的 client/server 观念。任何程序如果能够实作出一个 OLE 对象，它就是 OLE server。任何程序如果调用 OLE 对象，它就是 OLE client。在一个像实地激活（in-place activation）这样的环境中，一个程序启动于另一个程序的窗口之中，两个程序所做出来的 OLE 对象混在一起运作。这种情况下你不能够说哪个程序是 server 而另一个程序是 client，其所扮演的角色会因为谁调用谁而有所改变。client 或 server 的设计是以每个对象为基础，所以一个程序可以同时既是 client 又是 server。

Apartment Model

在 apartment model 中，一个对象和一个特定的线程有关联，只有在该线程的 context 之中才能够调用该对象。你可以在 Windows NT 3.51 和 Windows 95 和 Windows NT 4.0 中发现这个模型。至于 Windows NT 3.5，只支持单线程模型。

COM 的“apartment”是一个对象产生、运作、毁灭的地方。在一个 server 之中，COM 将循序调用一个 apartment-model 对象，使得一次只有一个调用会发生。这个模型有点像窗口的消息队列：一个特定线程拥有一个窗口，该线程为该窗口的所有消息服务。

这个线程模型在 server 之中的意义非常清楚，因为 COM 必须知道一个对象能够在哪一个线程中执行。但是 client 也必须有这么个线程模型，其意义就比较不那么清楚了。当 client 要求一个指向 interface 的指针，它是从一个特定的线程（也就是“apartment”）中发出请求。这个 client 可能只在该 apartment 中使用该 interface 指针。

当然也有可能产生一个 interface 指针给其他的 apartment 使用啦。若要这么做，你必须使用 COM 函数 CoMarshalInterThreadInterfaceInStream() 和 CoGetInterfaceAndReleaseStream()。但这些函数并不涵盖在本书范围内。

Apartment model 的内部运作系使用消息队列。COM 为每一个在 apartment model 中登记有案的线程产生一个不可见窗口。而我们知道，一个窗口的消息只能够被产生该窗口的线程处理。所以，使用消息队列可以自动强迫适当的线程处理对 COM 对象的调用。这项技术的副作用是，使用 apartment model 的线程必须有一个消息循环，否则调用 COM 对象时消息将不会被派送（dispatched）。

如果有数个调用同时对同一个 COM 对象发出，那么每一个调用都会被放进消息队列之中。每次 server 进入其消息循环之中，就会有一个调用被派送出去。这项技术强迫同步化和排他性，因为一次只有一个调用能够被处理。

当 client 调用一个 COM 对象，COM 会进入自己的消息循环中，因此，允许窗口消息被处理，并允许回到 apartment 内的其他 COM 对象。也因此，一个 apartment 可以在同一时间内既是 client 又是 server。

Free-threaded Model

Free-threaded model 是自从 Windows NT 4.0 之后才引进的模型。在这个模型中，对于 out-of-proc COM server（是个 EXE 而不是 DLL）中的对象的调用动作，是由属于 COM 的线程发出。换句话说，即使你的程序只有一个线程，COM 自己也有一缸子线程，用来调用你的 COM 对象。

如果 server 是 in-proc（也就是 DLL 形式），那么调用通常是直接来自 client，送往 server，其间没有 COM 的介入——虽然其中有一个除外（稍后我会讨论之）。“直接调用”正是 apartment model clients 没办法在线程之间传递指针的原因之一。由于 client 有直接的函数指针，根本不经过 COM，所以 COM 没有机会在多个线程尝试于相同时间发出调用时加以疏导。

在 in-proc server（译注：DLL 形式）和 out-of-proc server（译注：EXE 形式）之间，存在一个基本差异，那就是由谁产生线程。在 in-proc server 中，线程的产生与否完全被 client 程序中的线程个数所控制。一旦 in-proc servers 开始执行，其典型行为类似于一个正常的多线程 DLL 与应用程序之间的互动。但在一个 out-of-proc server 中，线程是由 COM 产生的。

Free threaded model 并不依赖使用 Windows 消息，所以没有必要拥有一个消息循环。避过消息循环，可以使得对 free-threaded 对象的调用能够比对 apartment model 对象的调用被更显著地快速处理。付出的代价则是应用程序必须负起保护对象的完全责任。同步化控制或许只是像“放一个 mutex lock 在整个对象上”那么简单，但是确定对象受到保护是非常重要的。相反地，在 apartment model 中，由于外界请求总是一次一个地被处理，所以同步化控制根本不需要。

Client 程序也可以被声明为 free-threaded．一个 free-threaded client 可以安全地把一个 interface 指针传递给另一个线程，并且可以在任何时间对着任何线程中的 interface 发出调用．这个意思是 client 可以对着同一个对象同时发出数个请求，也因此一个 free-threaded client 可能在同一时间使用五个不同线程中的同一个 interface 指针．

声明一个线程模型

一个 COM client 或 server 必须声明其所支持的线程模型．对于 EXE 文件，每一个即将使用 COM 的线程都必须在调用任何 COM 函数之前先调用 CoInitialize() 或 CoInitializeEx()．线程模型即在那个时候选定．CoInitializeEx() 规格如下：

```
HRESULT CoInitializeEx(
        void* pvReserved,
        DWORD dwCoInit
    );
```

参数

pvReserved	保留给未来使用．目前必须指定为 NULL。
dwCoInit	COINIT enum 中的一个栏位．Windows NT 4.0 允许以下两种数值： COINIT_MULTITHREADED COINIT_APARTMENTTHREADED

返回值

如果 CoInitializeEx() 成功，则传回 S_OK；如果参数不合法，则传回 E_INVALIDARG．如果线程先前已经调用过 CoInitializeEx()，并且给予不同的 dwCoInit 值，则传回 RPC_E_CHANGED_MODE．

CoInitialize() 内部调用 CoInitializeEx()，并指定线程模型为 COINIT_APARTMENTTHREADED。如果你使用复合文档（compound documents），你必须以 OleInitialize() 将 OLE 初始化，而该函数内部会调用 CoInitialize()。如果你使用复合文档，你就不能够在任何已经调用过 OleInitialize() 的线程中使用 free-threaded 对象。

DLL server 的运作又有些不同。它不再是在启动时注册其线程模型，而是把模型种类储存在注册表（registry）之中。在 CLSID/InprocServer32 项目下，DLL 放置一个键（key），名为“ThreadingModel”，其值可能是以下之一：

内　容	意　义
没有出现	单线程模型
Apartment	支持 apartment model
Free	支持 free-threading
Both	支持 apartment model 和 free-threading model

举个例子。作为一个 OLE DLLs，OLEAUT32.DLL 的注册表相关项目将是：

```
HKEY_CLASSES_ROOT
   CLSID
      {00020420-0000-0000-C000-000000000046}
         InprocServer32
            ThreadingModel="Both"
```

混合模型

一个应用程序在某些线程中使用 apartment model 而在另一些线程中使用 free-threaded model 是可能的。为了避免引起术语上的混淆，所有被注册为 free-threaded 的线程被说是生活在它们自己的 apartment 之中。因此，将会有一个 free-threaded apartment 和多个线程，以及一个或多个标准的 apartment，每一个拥有单一线程。

每一个 apartment 的行为如你所预期。标准的 apartments 将利用消息队列来传达数据，并且也因此是同步化的。free-threaded apartment 没有消息队列，同时也异步化。

Interoperability

一个 client 和一个 server 声明为不同的线程模型，也是可能的。如果 client 和 server 都是 out-of-proc，那么 COM 就可以负责让两个人安全地沟通。如果 free-threaded client 调用一个 apartment model server，那么 COM 就会将那些请求一个个排好，使 server 一次处理一个。如果一个 apartment model client 调用一个 free-threaded server，那么 server 被允许正规地处理那些请求。

如果 server 是 in-proc，那么 COM 还是可以负责让通讯的两边“interoperate”，但可能需要额外的一些代价。Servers 通常被设计为 in-proc，为的是让 client 发出的调用能直接到达 server 端，不必透过 COM 的协助。这样的安排相当程度地增加了速度。如果一个 free-threaded client 尝试使用一个 apartment model 的 in-proc server，那么 COM 必须介入，作法是把请求（request）调往 main apartment（也就是第一个调用 CoInitializeEx() 的线程）去，并将每一个请求依序送到 in-proc server 手上。

AUTOINCR 范例程序

在书附盘片之中你会发现一个 COMDEMO 程序。这个范例内含 Visual C++ 的 project 文件，可用以建造一个名为 AUTOINCR 的 COM 对象。盘片中有两个子目录的范例程序用到这个对象，其中之一是 MFCCLI，以 MFC 写成，是个多线程程序。另一个是 VBCLI，以 Visual Basic 写成。CLI 是 Client 的意思。

COM 对象实现出两个会自动累加的变量，每一个变量在它被读取时就自动加 1．一个累加比较慢，大约每 15 秒一次，另一个累加比较快，大约每 2 秒一次．服务器是以 DLL 的形式呈现，所以是个 in-proc server．

虽然这个对象的工作模式十分简单直接，但还是有一些专家等级的性质，可以允许和 Visual Basic 之间有“interoperability”．

这个对象是以 ActiveX Template Library（ATL）1.1（注）版建立起来的，你必须安装有 ATL，才能编译这个程序。

注：在 1996/10，ActiveX Template Library 可以从 http://www.microsoft.com/visualc/vc42/atl 中下载而得．ATL 1.1 版需要搭配 Visual C++ 4.2．

对于产生一个轻量级的 COM 对象，ATL 提供了很大的助力．为了降低其大小，特别是因为要在 Internet 上传送，ATL 可以不靠 MFC 或 C runtime library 而产生一个 COM 对象．使用 Visual C++ 中的一个 wizard，ATL 可以帮助你产生 in-proc server 或 out-of-proc server，实现出一个或一个以上的 COM 对象．ATL 的另一个巧妙处是，它可以产生拥有“dual interfaces”的对象，这样的对象可以被 IDispatch（或说是 OLE Automation）调用，也可以被 custom C++ interface 调用．

ATL 负责产生注册和注销所需的程序代码．在 AUTOINCR.H 文件中，有一个声明看起来像这样（格式可能略有不同）：

```
DECLARE_REGISTRY(CAutoIncr,
        _T("ComDemo.AutoIncr.1"),
        _T("ComDemo.AutoIncr"),
        IDS_AUTOINCR_DESC,
        THREADFLAGS_BOTH)
```

这个声明的意思是，服务器名为“ComDemo”，对象名为“AutoIncr”，版本号码为 1．这个对象的声明可以从 IDS_AUTOINCR_DESC 资源中获得．最后一个参数 THREADFLAGS_BOTH 指示 ATL 服务器注册函数要把“Both”字符串放

到注册表（registry）的“ThreadingModel”键（key）中（译注：代表同时支持 apartment model 和 free-threaded model）。

AUTOINCR.CPP 文件是对象的实现部分。对象名为 CAutoIncr，内含两个计数器变量 m_iValue 和 m_iValue2，它们在构造函数中被初始化为 0。成员函数 get_SlowCounter() 和 get_FasterCounter() 模拟运作方式：它先睡一会儿，然后把计数器现值传回去，然后把计数器加 1。程序代码十分直截了当，显示于列表 17-1。

列表 17-1 CAutoIncr 的实现内容

```
#0001  CAutoIncr::CAutoIncr()
#0002  {
#0003      // Initialize Counters
#0004      m_iValue1 = 0;
#0005      m_iValue2 = 0;
#0006  }
#0007
#0008  STDMETHODIMP CAutoIncr::get_SlowCounter(WORD* pValue)
#0009  {
#0010      Sleep(15000); // simulate work being done.
#0011      if (pValue)
#0012      {   // Return value and increase for next time
#0013          *pValue = m_iValue1++;
#0014          return S_OK;
#0015      }
#0016      return E_POINTER;
#0017  }
#0018
#0019  STDMETHODIMP CAutoIncr::get_FastCounter(WORD* pValue)
#0020  {
#0021      Sleep(2000); // simulate work being done.
#0022      if (pValue)
#0023      {   // Return value and increase for next time
#0024          *pValue = m_iValue2++;
#0025          return S_OK;
#0026      }
#0027      return E_POINTER;
#0028  }
```

AutoIncr 对象并未使用任何同步机制来调整对对象的存取操作，因为即使 context switch 发生在一个坏时机，也不会有什么东西会出错。然而，万一对象的行为比较复杂，它将需要把函数放在一个 critical section 之中。

将对象注册

在你能够执行任何一个 client 程序之前，你必须先将 server 注册到注册表（registry）中。你可以建造 AUTOINCR，注册 DLL 将是建造程序的一部分。或者你也可以执行 REGSVR32.EXE，那是 Visual C++ 的一个工具。例如：

```
regsvr32 comdemo.dll
```

在安装好对象之后，它将显示 Visual C++ 或 Visual Basic 中的可用对象于【Object Browser】窗口之中。Visual Basic 的【Object Browser】窗口展示于图 17-1。显示于其中的信息告诉我们，COMDEMO 内含单一对象，名为 CAutoIncr；这个对象有单独一个 interface，名为 IAutoIncr。

一旦 server 被注册，你就可以执行两个范例程序中的任何一个。其中一个程序以 MFC 完成，另一个程序以 Visual Basic 完成。MFC 程序是 free-threaded（在 Windows NT 4.0 环境中），Visual Basic 程序则使用 apartment model。

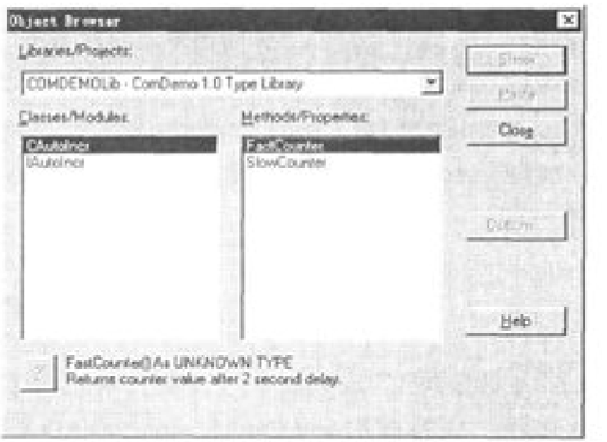


图 17-1 Visual Basic 的【Object Browser】窗口显示 COMDEMO

执行 MFCCLI

第一个范例程序称为 MFCCLI。这是一个以 MFC 完成的 client 程序，使用 COMDEMO server。图 17-2 显示这个程序的执行结果。这个 client 程序把它自己注册为一个 free-threaded client，它可以使用主线程或后台线程来调用 AutoIncr 对象。

虽然整个示范非常简单，还是有一些观念值得说一说。任何时候如果你以主线程取回一个数值，整个程序会停下来，直到数值被取回来。调用 COM 对象是一种同步动作，完成之前不会返回。由于对象的 methods 需要数秒钟来执行，所以整个程序会锁住，直到调用完成，因为消息并未被处理。

如果你选按任何一个【Background】钮，会有一个线程被产生出来，并索取一个对象的数值。如果 server 使用 apartment model，那么所有的请求都将被循序处理；选按【Show Counter】中的【Get (Background Thread)】钮三次，需要 45 秒才能够完成。如果 server 使用 free-threading，所有的请求都可以同时被处理。

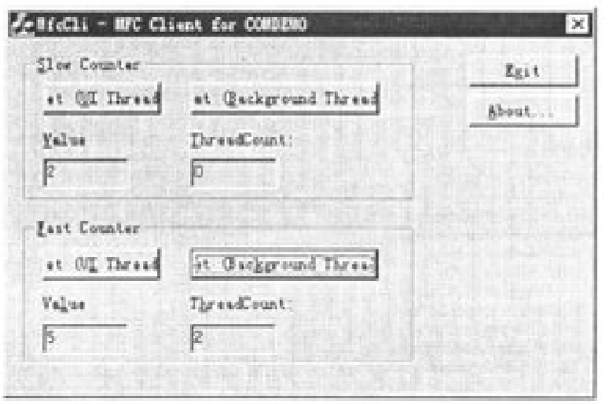


图 17-2 MFCCLI 的主窗口

由于 client 是 free-threaded，interface 指针可以在线程之间被处理，没有问题．如果 client 是 apartment model 而不是 free threaded，在线程之间传递指针将被禁止，必须使用先前描述过的那些函数才有办法做到．

执行 VBCLI

第二个 client 程序称为 VBCLI．其主窗口显示于图 17-3．这个程序以 Visual Basic 4.0 完成．和 MFCCLI 不同，Visual Basic 是一个 apartment model client．COMDEMO 将它自己注册为既支持 apartment model 又支持 free-threading model，所以 VBCLI 可以成功地载入 COMDEMO server．如果 COMDEMO 只将自己注册为 free-threaded，那么 COM 就必须介入 server 和 client 之间，使 free-threaded server 能够正确无误地和 apartment model client 交谈．

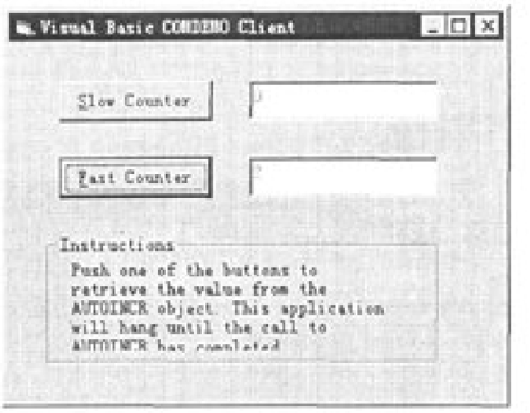


图 17-3 VBCLI 的主窗口

VBCLI 中的每一件事情都必须循序进行，因为 Visual Basic 不能够启动多个线程以解决多个悬而未决的请求．当你在【Slow Counter】或【Fast Counter】上按钮，AutoIncr 中适当的变量将被读出来．慢计数器需要 15 秒而快计数器需要 2 秒．就像在 MFCCLI 中使用主线程一样，应用程序在这些请求被处理期间，将会停滞下来．

如果你同时执行 MFCCLI 和 VBCLI，你会看到一个 server 处于混合模型的情况。COMDEMO server 使用 apartment model 和 Visual Basic 程序沟通，使用 free threading model 和 MFC 程序沟通。每一个程序将产生属于它自己的一个 AutoIncr 对象实体(instance)，所以 VBCLI 的计数器和 MFCCLI 的计数器是不相同的。

Automation 对象通常很容易和 Visual Basic 合用。如果你对实现细节有兴趣，以下的代码可以在应用程序启动时，让 VBCLI 载入 AUTOINCR：

```
Set AutoIncr = CreateObject("ComDemo.AutoIncr.1")
```

对象名称必须和 COMDEMO 中所声明并被放到注册表（registry）中的一样。变量 AutoIncr 被声明为一个全局值，像这样：

```
Dim AutoIncr As Object
```

FastCounter 和 SlowCounter 都是 AutoIncr COM 对象的“properties”。【Fast Counter】按钮以下面的代码读进 FastCounter 的值：

```
FastCounterValue.Text = AutoIncr.FastCounter
```

提要

在这一章中你学到了 COM 的线程模型。Apartment model 是以 Windows 的消息队列为基础，并保证同步。Free threading model 不使用 Windows 消息队列，并因而招至“你得自己完成同步控制”的难题。你也看到了如何在一个 in-proc server 和一个 out-of-proc server 中使用它们。最后你还看到了一个 COMDEMO 示范对象，它解释了本章的许多观念。

MTVERIFY 宏

MTVERIFY 宏使用于本书许多范例程序中，用以捕捉错误并协助找出其原因．下面是其使用实例：

```
MTVERIFY( CloseHandle(hThread) );
```

CloseHandle()与其他许多 Win32 API 函数一样，传回 TRUE 表示成功，传回 FALSE 表示失败．如果失败，你可以调用 GetLastError() 找出其原因．MTVERIFY() 就是使用这样的机能来产生像下面这样的错误信息（如果 CloseHandle() 失败的话）：

```
The following call failed at line 50 in Demo.c:
   CloseHandle(hThread)

Reason: The handle is invalid
```

我将把这个宏的源代码分为一小段一小段并详细解释它．MTVERIFY() 使用数个少见的 Win32 函数和 C runtime library 函数，我将在必要时一并解释它们．以下是文件的头部．由于其间没有什么可执行代码，所以也没有什么好解释的．

```
#0001  /*
#0002   * MtVerify.h
#0003   *
#0004   * Error handling for applications in
#0005   * "Multithreading Applications in Win32"
#0006   *
#0007   * For simplicity, this code includes the complete function PrintError
#0008   * as a static function. For the examples in this book, this works fine.
#0009   * To use the PrintError() in an application, it should be taken out,
#0010   * placed in its own source file, and the "static" declaration removed
#0011   * so the function will be globally available.
#0012   */
#0013
```

下面这个 pragma 将内嵌一个命令到链接器中，使它链接 USER32 函数库——甚至即使这个函数库没有出现在链接器参数中：

```
#0014  #pragma comment( lib, "USER32" )
#0015
```

头文件 crtdbg.h 定义了一组由 Visual C++ 4.0 引进的调试函数。虽然有时候它似乎有点晦暗难明，但是它提供了“切入调试器并放置 Abort, Retry, Ignore 对话框”这样的机能。为了 _ASSERTE() 宏，我们需要它。

```
#0016  #include <crtdbg.h>
```

头文件 MtVerify.h 定义了两个宏。MTVERIFY() 用来检验函数是否成功，并且在函数失败时印出一段根据 GetLastError() 而得的错误信息。MTASSERT() 功能比较少一些，它因为不调用 GetLastError()，所以得不到错误原因。

MTVERIFY() 使用预处理符号 __FILE__ 和 __LINE__ 提供错误描述时所需的细部信息。它也使用“stringizing”运算符，也就是以下宏中的 #a。“stringizing”运算符能够把宏参数视为一个引号字符串。

```
#0017  #define MTASSERT(a) _ASSERTE(a)
```

```
#0018
#0019
#0020  #define MTVERIFY(a) if (!(a))
#0021          PrintError(#a,__FILE__,__LINE__,GetLastError())
```

再来是 PrintError() 的函数原型声明．这个函数在 MTVERIFY() 失败时被调用之。

```
#00xx  // Function prototype, in case the routine is moved out of this
file
#00xx  void PrintError(
#00xx      LPSTR linedesc, LPSTR filename, int lineno, DWORD errnum);
```

译注：光盘中的文件并无此段函数声明。事实上也的确不需要。

接下来是 PrintError() 的实现．它获得经由预处理器所收集的所有关于 MTVERIFY() 宏的信息．如果你的程序执行于一个窗口之中，此函数会放出一个消息框，类似 MFC 用于 asserts 的那一种．如果你的程序是 console 程序，PrintError() 会把错误信息送到 stderr 去．

```
#0022  static void PrintError(LPSTR linedesc, LPSTR filename, int
lineno, DWORD errnum)
#0023  {
#0024      LPSTR lpBuffer;
#0025      char errbuf[256];
#0026  #ifdef _WINDOWS
#0027      char modulename[MAX_PATH];
#0028  #else // _WINDOWS
#0029      DWORD numread;
#0030  #endif // _WINDOWS
#0031
```

译注：#0022 与光盘中的代码有点不同．后者是：

```
#0022  __inline void PrintError(
          LPSTR linedesc, LPSTR filename, int lineno, DWORD errnum)
```

接下来的 FormatMessage()，是最神秘的一个 Win32 函数．这个函数可以根据 GetLastError() 的结果产生出具有阅读价值的错误信息．这些错误信息是本

土文字，所以节省了不少国际化所需的时间。FormatMessage() 会自动配置所需的内存，我们不需要担心缓冲区饱和的情况。

```
#0032   FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER
#0033                | FORMAT_MESSAGE_FROM_SYSTEM,
#0034          NULL,
#0035          errnum,
#0036          LANG_NEUTRAL,
#0037          (LPTSTR)&lpBuffer,
#0038          0,
#0039          NULL );
#0040
```

现在我们要产生一个内含错误信息的字符串。这个函数使用 Windows 的 wsprintf() 函数，取代 C runtime library 的 sprintf() 函数。

```
#0041   wsprintf(errbuf, "\nThe following call failed at line %d in %s:\n\n"
#0042             "    %s\n\nReason: %s\n", lineno, filename, linedesc, lpBuffer);
```

如果这是一个 console 程序，就取得 stderr handle，并使用 WriteFile() 显示错误信息。等候三秒钟以确定使用者看到了它。如果这是一个 GUI 程式，则以消息框显示错误信息，并强迫窗口放在最上层以免被忽略。

```
#0043   #ifndef _WINDOWS
#0044     WriteFile(GetStdHandle(STD_ERROR_HANDLE),
                  errbuf, strlen(errbuf), &numread, FALSE );
#0045      Sleep(3000);
#0046   #else
#0047      GetModuleFileName(NULL, modulename, MAX_PATH);
#0048     MessageBox(NULL, errbuf, modulename,
                   MB_ICONWARNING|MB_OK|MB_TASKMODAL|MB_SETFOREGROUND);
#0049   #endif
```

最后，结束前把一个适当的结束代码 EXIT_FAILURE 交给操作系统。此常量定义于 stdlib.h 中。

```
#0050      exit(EXIT_FAILURE);
#0051  }
```

我发现 MTVERIFY 非常好用。它节省了我许多调试时间，并且使我免于常常需要猜测错误原因。由于使用了预处理的技巧以及少见的 FormatMessage() 函数，MTVERIFY 提供的错误信息可以说是十分完整的，即使你不靠调试器的帮助，依然可以快速标出问题之所在。

更多的信息

译注　这个附录B是我（译者）加上去的，为读者提供更多的相关信息来源。

- Operating System Concept 4th edition
  （A. Silberschatz　J. Peterson　P.Galvin / Addison Wesley）
  Chap4: Processes
  Chap5: CPU Scheduling
  Chap5: Processes Synchronization
  Chap6: DeadLocks

- Win32 System Services - The Heart of Windows NT
  （Marshall Brain / Prentice Hall）
  Chap5: Processes and Threads
  Chap6: Synchronization

- Windows 95 System Programming SECRETS
  （Matt Pietrek / IDG Books）
  Chap3: "Modules, Processes, and Threads"

- Advanced Windows 3rd edition
  （Jeffrey Richter / Microsoft Press）
  Chap2: "Kernel Objects"
  Chap3: "Processes"
  Chap4: "Threads"
  Chap10: "Thread Synchronization"
  Chap14: "File Systems"
  Chap15: "Device I/O"

- Programming Windows 95
  （Charles Petzold / Microsoft Press）
  Chap14: "Multitasking and multithreading"

- Programming Windows 95 with MFC
  （Jeff Prosise / Microsoft Press）
  Chap14: "Threads and Thread Synchronization"

- Inside Visual C++ 4.0
  （David Kraglinski / Microsoft Press）
  Chap11: "Windows Message Processing and Multithreaded Programming"

- The Revolutionary Guide to MFC 4 programming with Visual C++
  （Mike Blaszczak / WROX Press）
  Chap11: "Writing Multithreaded Applications with MFC"

- Programming Windows 95 Unleashed
  （SAMS Publishing）
  Chap3: Multitasking, Processes, and Threads

- Programming Windows NT 4 Unleashed
  （SAMS Publishing）
  Chap22: Threads

- “深入浅出 MFC”第 2 版
  （侯俊杰 / 松岗）
  Chap14：“MFC 多绪程式设计”

Win32 多线程程序设计

Multithreading Applications in Win32

Jim Beveridge & Robert Wiener

Copyright © 1997 by Addison Wesley Longman, Inc.

Simplified Chinese Copyright 2002 by Huazhong Science and Technology University Press and Pearson Education North Asia Limited.

All rights Reserved.

Published by arrangement with Pearson Education North Asia Limited, a Pearson Education Company.

版权所有，翻印必究。

本书封面贴有华中科技大学出版社（原华中理工大学出版社）激光防伪标签，封底贴有"Pearson Education"激光防伪标签，无标签者不得销售。

图书在版编目(CIP)数据

Win32 多线程程序设计/(美)Jim Beveridge & Robert Wiener 著；侯 捷 译

武汉：华中科技大学出版社，2002 年 1 月

ISBN 7-5609-2638-X

Ⅰ. W…

Ⅱ. ①J… ②R… ③侯…

Ⅲ. 程序设计，Win32

Ⅳ. TP311.1

责任编辑：周 筠(http://www.yeka.com.cn)

技术编辑：王伟军 王飞

出版发行：华中科技大学出版社 （武昌喻家山 邮编：430074）

录 排：华中科技大学惠友科技文印中心

印 刷：湖北新华印务有限公司

开 本：787×1092 1/16 印 张：30 字 数：450 000

版 次：2002 年 1 月第 1 版 印 次：2002 年 4 月第 2 次印刷

印 数：6 001—12 000 定 价：59.80 元

ISBN 7-5609-2638-X/TP・456

Win32 多线程程序设计 Multithreading Applications in Win32

Jim Beveridge & Robert Wiener 著 [illegible] 译