《Win32多线程程序设计》

图书基本信息

书名：《Win32多线程程序设计》

13位ISBN编号：9787560926384

10位ISBN编号：756092638X

出版时间：2002-1

出版社：华中科技大学出版社

作者：[美] Jim Beveridge,Robert Wiener

页数：453

译者：侯捷

《Win32多线程程序设计》

内容概要

《Win32多线程程序设计》全书共分三篇。第一篇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；第二篇介绍C runtime函数库和MFC对线程的支持、如何在USER和GDI的限制之下施行对线程等内容；第三篇谈论如何组织一个程序，使它有效支持多线程。

《Win32多线程程序设计》

作者简介

Jim Beveridge进入操作系统的研究领域已有15年，从多处理器数据库到microkernel操作系统的开发，他都有经验。他于Rochoester Institure of Technology获得计算机科学学士学位。他目前受聘为Turning Point Software公司顾问。

《Win32多线程程序设计》

书籍目录

函数索引（Function Index）常见问答集（Frequently Asked Questions）第一篇 上路吧，线程第1章 为什么要“千头万绪”一条曲折的路与线程茶枕为什么最终用户也需要多线程多任务Win32基础Context SwitchingRace Conditions（竞争条件）Atomic Operations（原子操作）线程之间如何通讯好消息与坏消息第2章 线程的第一次接触产生一个线程使用多个线程的结果核心对象（Kernel Objects）线程结束代码（Exit Code）结束一个线程错误处理后台打印（Background Printing）成功的秘诀第3章 快跑与等待看似闲暇却忙碌（Busy Waiting）性能监视器（Performance Monitor）等待一个线程的结束在一个GUI程序中等待提要第4章 同步控制(Synchronization)Critical Sections（关键区域、临界区域）死锁（Deadlock）哲学家进餐问题（The Dining Philosophers）互斥器（Mutexes）信号量（Semaphores）事件（Event Objects）从Worker线程中显示输出Interlocked Variables同步机制摘要第5章 不要让线程成为脱疆野马干净地终止一个线程线程优先权（Thread Priority）初始化一个线程提要第6章 Overlapped I/O在你身后变戏法Win32文件操作函数被激发的File Handles被激发的Event对象异步过程调用（Asynchronous Procedure Calls,APCs）对文件进行Overlapped I/O的缺点I/O Completion Ports……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)第8章 使用C Run-time library第9章 使用C++第10章 MFC中的线程第11章 GDI与窗口管理第12章 调试第13章 进程之间的通讯(Interprocess Communication)第三篇 真实世界中的多线程应用程序第14章 建造DLLs第15章 规划一个应用程序第16章 ISAPI第17章 OLE ActiveX COM附录A MTVERIFY宏附录B 更多的信息

《Win32多线程程序设计》

编辑推荐

使用线程，你可以产生高效率的服务器。建立Internet服务器扩充软件，获得多CPU系统的好处，建立精巧的COM/OLE对象，并改善程序的反应度。写出这样的软件，需要更多的理论基础，以及一本参考手册；需要广泛了解每一件事情如何彼此相称；需要一份指南，告诉你什么可以正常动作，什么不能！通过Win32 API，Windows NT以及Windows 95都可以支持多线程程序设计，但是这个重要主题的信息却极稀有而不够详尽。在《Win32多线程程序设计》书中，Jim Beveridge和Robert Wiener告诉你什么时机、什么地点、什么方法可以使用多线程。

《Win32多线程程序设计》

精彩短评

1、系统的介绍了Win32下多线程编程，值得购买。 介绍了Win32下创建关闭多线程的几个函数，并详细的进行了比较。还有，讲述了多线程通信的几种方式。
2、光盘内容太少，也不知道多加点示例！书的内容不错！
3、实用的工具手册
4、CPP系Q1
5、1、电子书：图版（可标注）
2、已买书
6、存在一些错误
7、内容比较浅显
8、比较详细，深入浅出地讲解windows下多线程的书籍，值得一读，。。。
9、win32多线程入门书籍
10、windows下多线程编程的经典书，很久以前就看过的。
11、线程入门 一地API
12、写的不错，但是外国人写的东西总有些不好消化
13、不错的入门书，但是多线程要写好，经验很重要。
14、可能是当时唯一的win多线程专著，奇怪的是有些列子极有水准，有些则显得无厘头
15、当年的经典，适合多线程编程入门
16、工作之余的读书时间极为有限，因而“ 读书只读经典 ”成为我的读书准则--不是说非经典书籍不好，只是与我而言，少了经典的畅快。本书是侯捷先生的经典，声名早已远扬，更喜欢侯捷老师严谨的治学作风，故而得到此书时心情极为兴奋，读起来更是若品美酒，沉醉其中
17、多线程入门
18、书是好书，可惜比较老了
19、多线程实践启蒙
20、不错，依然不过时.
21、候捷译得不赖
22、通俗易懂的语言，简洁的描述！让人轻轻松松的学下去
23、这本书太旧了，Vista有新的api，且CPU的性能已很大提升，原书是九十年代了，过于老旧，C++标准委员会在今年将推出针对多线程和多核处理器的标准库，这本书的可读性就大大降低。我仔细读了，翻译得很适合中国人的语言口味，但过于肤浅。不知大家有没读过windows核心编程（五版），比较一下，你就知道这本书很多东西过时了，且没深度。多线程编程是对程序员的巨大挑战，现在多线程编程还只是初步。这书还是可以作为入门级，只需把最重要的部分看看就行，像里面的DLLS部分就应该省略，没深度，有很多Vista和WINDOWS XP的DLL细节没有，且不符合这两个系统的实际情况，因为操作系统已经不是那个时代的操作系统。
24、书讲得很细致，不知道侯捷大人写作的时候是否运用了在我们这儿讲课时提倡的深夜泡咖啡精神。
25、比起《Windows核心编程》这本非常简单易懂，侯捷的翻译太赞了
26、图书借还处(老馆二楼流通书库） TP316.7 160 在架上
27、基本啃完了
28、送货太慢，用了20天
29、Windows平台多线程编程入门. 这本书涉及内容非常少, 后半部内容过时, 只能用于削减对多线程编程的畏惧心理, 如果已经入门, 建议找本大部头正经的相关部分读一读.
30、多线程的问题的很多，还是需要系统的看下。
31、经典！多线程必备！
32、快速翻完了，到后面就迷迷糊糊了，感觉还是要实践才能真正弄懂线程
33、通俗易懂
34、不错不错
35、绝对过时的一本书，建议直接看《windows 核心编程》。如果看过《windows 核心编程》，再看这

本书你会觉得它是鸡肋,这本书实在贵了点又没深度。
36、经典跟过时不过时没一点关系
37、展现windows下多线程概貌
38、印象中多线程都是很复杂,但这书讲的还是很不算,例子都详细
39、系统地介绍了WIndows平台下多线程开发的体系和方法，对Windows平台诸多底层机制亦有一定介绍，侯捷的翻译浅显易懂，还加入了不少独到的理解作为注释，实在是不可多得的佳作。
40、貌似讲了很多东西, 对于具体的API解释不少, 但是实际对于多线程貌似只需要理解每个线程执行的顺序是不可预期的就行了, 具体的技术也就是怎么同步和传递信息.
41、内容不错， 不过不知道是我RP 问题， 还是此书在印刷方面在某一批中有问题， 有几页是空白的， 并没有被印刷上。 还好空白的几页都是无管紧要的东西。 书的内容还是不错。
42、老书了，启发不够震撼性
43、编写Win32多线程程序的最好的入门之作，没有之一，另外，译者的翻译很棒，赞！
44、比较基础， 这是我接触多线程的第一步书，很不错
45、参考下windows下的多线程与Posix的区别
46、没看完，感觉有点罗嗦，也许讲的太详细，叫“ 线程完全手册 ”，感觉还不够完全。
47、读过最好的多线程书
48、学习多线程编程不错的一本书。我读过全部内容。
49、读得时候我总在思考，他讲的的有些内容现在还适用吗？
50、描述windows线程以及线程开发很全面的一本书
51、有点老。。
52、侯捷老师的译作，一如既往的好，我的多线程启蒙书籍。
53、看过核心编程之后再看这本书，觉得讲的很基础。
54、这本书能帮助写出干净合理的线程封装。
55、排版不错，源代码量适中，2011-04-18 在读。
侯老湿翻译的又一本好书，对线程进行了深度的思索。此书似乎已经被我送人了... 貌似没有通览，拿来查字典式的反复看过些章节。
56、虽然在编程时早就用到了多线程，也有一些多线程的知识，但是总感觉比较零碎，不够系统化。这本书恰好符合我的需要，它系统地讲述了多线程的方方面面，由主及次，由浅入深。虽然是将近十年前的一本书，现在读起来丝毫没有过时的感觉。
必须要提一句的是：侯杰老师的文笔真好，行云流水，翻译的没有任何生涩感。而且侯杰好师极其认真，对原版中出现的错误或者容易产误解的地方都做了注释。感谢作者和侯杰。

《Win32多线程程序设计》

精彩书评

1、虽然在编程时早就用到了多线程，也有一些多线程的知识，但是总感觉比较零碎，不够系统化。这本书恰好符合我的需要，它系统地讲述了多线程的方方面面，由主及次，由浅入深。虽然是将近十年前的一本书，现在读起来丝毫没有过时的感觉。必须要提一句的是：侯杰老师的文笔真好，行云流水，翻译的没有任何生涩感。而且侯杰好师极其认真，对原版中出现的错误或者容易产误解的地方都做了注释。感谢作者和侯杰。
2、已经读完了第一篇，觉得讲得很细致，也很易懂，但有些哆嗦，很多内容不用看例子也知道是怎么回事，就直接跳过了。做过多线程的人，可以当作复习教材，系统地回顾一下。适合入门。为什么说我的评论太短了？？
3、原著应该很强~就是有点老，很多例子都是那时候的事了。这本台湾人翻译的算是相对来讲很用心的了，甚至还有对原著的更傻瓜版解释，国内翻译有几人能做到？不过还是有翻译上的错误，迅速啃完之后还是应该去看原著。
4、使用 C RUNTIME LIBRARY 这一章节不少内容都可以说是过时的（或者说是错误的），主要是关于 CRT 的线程安全的问题的描述在现在的一些 CRT 实现上，完全不是那么回事，CreateThread 也没有描述中的那么糟糕。总之对于此问题，读者需要自己细心品味，另外本书的“过时”的问题，应该也在其他章节有所表现，务必注意：）
5、陆陆续续终于读完了，花了大概3个礼拜吧。一直觉得这是一本相当有深度的书（为什么会有这个感觉？估计是当年大学时看到一个牛人书架上有这本书，牛人嘛，都读牛书的~~~）。但前段时间做了点多线程的程序，于是想到了读一读这本书，结果发现，正如本书第一篇的名字一样，这本书其实只是让你”上路“而已。第一篇无论是多线程里如context switch，race condition，synchronization等等的概念，还是每个线程相关的Windows API使用及原理的讲解，以及最后Overlapped I/O的介绍, 作者都极其小心，详细的解释着。第二篇主要是讲了C, C++, MFC中线程的用法的，其实也就是：CreateThread,_beginthread, _beginthreadex, CWinThread,AfxBeginTrhread这些API之间的区别。调试讲的很简单，IPC除了SendMessage和共享内存，其他都只是一带而过。不过像MTVerify那样的宏在那个时候可能是比较NB的，只不过现在大家都见多了。第三篇第一章讲了些如何规划一个多线程程序的“道理”，但相信对于一个有一定经验的人，这些应该都已经懂了的。（或者看了本书前面的内容后到这里就懂了）COM的多线程提到了，赞一个的，但对于刚接触的人，那些篇幅的解释还是不够的，更多可以参考《COM技术内幕》第十二章。本书本来已经很罗嗦了，想到不到侯先生比作者更罗嗦 - 不过这样也好，理解起来就相当容易了，整篇读下来还是比较顺畅的，应该说明翻译的也比较好吧。英文版是有的，但不打算读了 - 有必要吗？
6、很久没看中文的技术书了，不知道是中文本身就表达能力不行，还是因为台湾人讲话习惯不一样，感觉很多地方都表述地不清楚，主谓不清楚，不知所云。还有就是太罗嗦了，不像AUPE这类书写得很“专业”（个人更想他从多扯扯信号量机制、算法、数据结构、系统设计方面讨论问题），我觉得不适合科班出生的读者。linux给人的感觉是思想简单，直达本质，自由，windows则是封装、隐藏细节、规范、快速批量生产。———一个刚从linux到windows的说
7、到今天为止大致过了一遍 因时间关系和应用需要，这几章跳过没看： 6.Oeverlapped I/O 8.使用C Run-time Library 13.进程之间的通讯 14.建造DLLs 16.ISAPI 17.OLE,ActiveX,COM 总的来说非常好，一边看一边在本子上记笔记，收获很大 :)多线程编程是win32程序员的基本技能，get used to it！
8、评4分觉得太高，3分又太低，还是3分吧。是本不错的多线程入门数据，方便没有太多多线程编程经验的人建立关于多线程编程的whole picture，有一点“深入浅出”的味道。副标题称之为“线程完全手册”，有些名不符实。
9、初级读物吧，很多东西也不是很透彻，但是在win上的多线程底层模型与原理还是可以了解到的，一些深入的东西好像讲得不怎么好，像异步IO和IOCP这些觉得讲得太肤浅了，没看懂，还是其他资料找到的比较详细。除了一些基本的东西外，其他的东西就当随便读读吧，了解下就差不多了，读这本书还是达到了我预期的目的：了解基本的东西。推荐菜鸟多线程从这里开始……
10、有一些地方存在一些笔误，看了英文版才知道：读写锁部分，谈到读取锁定和解锁时，ReaderCount 错误写为 ReadCount，不过这个无关要紧，严重的是为 Reader 锁定的相关代码：Lock(ReaderMutex)ReadCount = ReaderCount + 1if (ReaderCount ==

0)Unlock(DataSemaphore)Unlock(ReaderMutex)这段代码基本没有正确的地方，正确的（原文中）应该为：Lock(ReaderMutex)ReaderCount = ReaderCount + 1 if (ReaderCount == 1) Lock(DataSemaphore)Unlock(ReaderMutex)请大家注意：）
11、原书90，91页在解释MsgWaitForMultipleObjects时，有两处不明，恳请高手指教：1、收到WM_QUIT之后，书中旨在等待后台线程结束时仍然能处理消息，若此时用于又在新建后台打印线程，不是程序就一直结束不了了吗？先前发的退出命令是否就不合逻辑了呢？2、WM_THREADCOUNT是如何迫使MsgWaitForMultipleObjects返回的？谢谢！
12、提供的源代码如何编译呢
13、基本上是在地铁上腾出时间阅读的。这本书是我在大学时候读过，那个时候才刚刚读过核心编程，对系统本身的理解尚且不够，所以读下来似懂非懂。后来毕业后，又看了几遍核心编程，可是由于主要开发的环境是linux，所以很少再拿起这本有些reference性质的小书了。最近由于阅读一个跨平台的网络库，需要了解win32下面的设计。我再次利用每天地铁的时间，再看了一遍这本书。不得不说，书是好书，可是内容我觉得尚且不够，我基本上是这本书加上核心编程加上windows网络编程三本书互相印证，才有更深刻的理解。此外这本书有一些篇幅在com上面，一些在mfc上面，这些篇幅我都跳过了
14、本书对win32线程同步api的讲解没有《windws核心编程》详细对线程同步算法又没有os书籍如《os concept》来得全面本书内容没有什么出彩的地方 个人认为这本书比较鸡肋

《Win32多线程程序设计》

章节试读

1、《Win32多线程程序设计》的笔记-第45页

BOOL GetExitCodeThread(HANDLE hThread,LPWORD lpExitCode)

TRUE代表有成功获得线程的状态
FALSE代表没有成功
这个返回值与线程的状态无关
书中对此描述的并不准确

BOOL ExitThread(DWORD dwExitCode)
书中线程的描述中，在主线程退出后，会强迫其他线程结束
事实并非如此，对于上面这句话，可能是翻译问题
当主线程正常退出时，其他线程的确被强制结束了
但是使用ExitThread退出时，其他线程并没有被强制结束

2、《Win32多线程程序设计》的笔记-第190页

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。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。

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。

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。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。CreateFile() 可以用来打开各式各样的资源，包括（但不限制于）：
1.文件（硬盘、软盘、光盘或其他）
2. 串行口和并行口（serial and parallel ports）
3.Named pipes
4.console
以下为原型：
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。

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。
以下为函数原型：

```
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
);
```

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
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。
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。
以下时例子：

```
int ReadSomething()
 {
 BOOL rc;
 HANDLE hFile;
 DWORD numread;
 OVERLAPPED overlap;
 char buf[512];

// open the file for overlapped reads
 hFile = CreateFile( "C:\\WINDOWS\\WINFILE.EXE",
GENERIC_READ,
 FILE_SHARE_READ|FILE_SHARE_WRITE,
NULL,
 OPEN_EXISTING,
 FILE_FLAG_OVERLAPPED,
 NULL
 );
 if (hFile == INVALID_HANDLE_VALUE)
 return -1;


// Initialize the OVERLAPPED structure
memset(&amp;overlap, 0, sizeof(overlap));
 overlap.Offset = 1500;
```

```
// Request the data
rc = ReadFile(
hFile,
buf,
300,
&amp;numread,
&amp;overlap
);

if (rc)
{
// The data was read successfully
}
else
{
// Was the operation queued ?
if (GetLastError() == ERROR_IO_PENDING)
{
// We could do something else for awhile here...

WaitForSingleObject(hFile, INFINITE);
rc = GetOverlappedResult(
hFile,
&amp;overlap,
&amp;numread,
FALSE
);
}
else
{
// Something went wrong
}
}

CloseHandle(hFile);
```

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。

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。

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

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()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。

3、《Win32多线程程序设计》的笔记-第365页

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。
卧槽，Linux设计哲学上就说是要尽量避免用多线程，提倡用多进程，以免引起多线程引起的复杂性问题。 宁愿代价开销更大。 话说，Unix编程艺术那本书说得好：编写软件就是尽量降低软件复杂度。--！话说，我在GitHub上看过很多别人自己的项目，都很少用到多线程。能尽量避免多线程就避免多线程。
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。
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。

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;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。

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。
使用共享内存（Shared Memory）
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）。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。

为了从共享内存中获得一个指针，我们必须使用 MapViewOfFile()。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。
较高层次的进程通讯（IPC）
匿名管道，命名管道等

4、《Win32多线程程序设计》的笔记-第257页

如果你使用 MFC 来开发程序，注意，不要在一个MFC 程序中使用_beginthreadex() 或
CreateThread()。

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()。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。
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。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。

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。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。每个版本对应一个调试版本针
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（Release）使
用。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
以 C Runtime Library 启动线程
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。

_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。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() ？
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()
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()：
1.在 C 程序中使用 malloc() 和 free()，或是在 C++ 程序中使用 new 和 delete。

2.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。

3. 使用浮点变量或浮点运算函数。

4.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。
避免stdio.h
尽量避免使用sprintf，printf，用console API来实现，文件，屏幕的重定向。GetStdHandle
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。
为什么你应该避免_beginthread()
_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。

另有一个函数对应于 ExitThread()，名为 _endthread();

_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。

5、《Win32多线程程序设计》的笔记-第305页

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。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。
MFC框架及其丑陋，然后被引诱进MFC坑的人，估计是不好出来了，呵呵。写windows程序我及其讨
厌写UI，特别麻烦。

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）。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
。CWinThread 被极其小心地设计，解决了许多我们在 CreateThread() _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。

6、《Win32多线程程序设计》的笔记-第180页

同步和异步行为
一个函数是异步还是同步的分别：

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。
临界区
千万不要在一个 critical section 之中调用 Sleep() 或任何 Wait...() API 函数。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。
临界区与互斥体区别：
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：

1. 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。（ring3到ring0的切换代价昂贵）

2.Mutexes 可以跨进程使用。Critical section 则只能够在同一个进程中使用。（MUtex是内核对象，所以可以跨进程）

3.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。

使用一个 completion port
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。

7、《Win32多线程程序设计》的笔记-第11页

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：

i Code

i Data

i 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。

“ 如果不需要，操作系统会将它动态扩充 ”？多了个“ 不 ”吧

8、《Win32多线程程序设计》的笔记-第400页

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）。

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，或者说是导入库）。
DLL 的通告消息（Notifications）
如果你使用 C runtime library（在DLL的源码中），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。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。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）
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。
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。
i 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）
i DllMain() 不接受第一个线程的 DLL_THREAD_ATTACH， 而以 DLL_PROCESS_ATTACH 取代之。
i 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。

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。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：
i 不要使用全局变量。用来储存 TLS 槽（slot）者例外。
i 不要使用静态变量。
i 如有必要，尽量使用 TLS（Thread Local Storage）。

i 如有必要，尽量使用你的堆栈。

9、《Win32多线程程序设计》的笔记-第322页

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。

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（那个CALLBACK函数，窗口回调）处理，并因此被产生该窗口的线程处理。
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。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。
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。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()。
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。
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。

线程之间的通讯
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。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。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。

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。

10、《Win32多线程程序设计》的笔记-第8页

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。

11、《Win32多线程程序设计》的笔记-第10页

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。
先记一下疑问：file handles是文件句柄吗，hInstance那个吗？

12、《Win32多线程程序设计》的笔记-第6页

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。
这段话不理解啊，Windows 3.x 不是可以运行多个Windows程序的吗，只不过相对于DOS，它只能单独运行，和这里说得不一样啊。
翻译很像台湾普通话呢。

13、《Win32多线程程序设计》的笔记-第106页

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”（就是无界面显示任务的线程，只做后台运算）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。
线程对象与线程的不同
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。可以使用 GetExitCodeThread( )等待一个线程的结束，然而这并不是好方法。有隐患：

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”
结束主线程
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。
微软的多线程模型
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。
多线程成功设计的关键
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）。
等待线程结束的糟糕方法：
1.调用Sleep()------。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。

2.是使用所谓的 busy loop，循环调用 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。

warning：绝对不要在 Win32 中使用 busy loop
等待一个线程结束的方法：
使用同步对象----调用WaitSingleObject()；

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。

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）。
等待多个对象：
调用WaitForMultipleObjects()

主线程消息循环的等待：
MsgWaitForMultipleObjects()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 函数。

《Win32多线程程序设计》

版权说明