1. 为什么要“千头万绪”

线程被定义在进程的某块内存中，CPU的寄存器中，线程产生时配有一个堆栈存储函数的局部变量，CPU的寄存器存储线程的工作状态。进程拥有其线程公有的全局变量和静态变量。同一进程的线程间共享内存，正常情况下进程间没有共享内存。

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。handle 只在其诞生地（进程中）才有意义，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。

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，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。

1. 线程的第一次接触

图形用户界面, 文本

描述已自动生成

多线程程序是无法预测其行为的，线程的执行顺序无法保证，context switch可能在任何时刻任何地点发生

1.多线程程序无法预期。2.执行次序随机，Task Switches任何时刻任何时间发生，编译时使用/MT或/MD,使用多线程版C runtime library。3.线程并不总是立刻启动。

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。

核心对象：

1进程（processes） 2 线程（threads） 3 文件（files） 4 事件（events） 5 信号量（semaphores） 6 互斥器（mutexes） 7 管道（Pipes。分为 named 和 anonymous 两种）

文本

描述已自动生成

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。

线程的 handle 是指向“线程核心对象”，而不是指向线程本身。所以可以在不结束线程的情况下关闭其handle。

线程对象的默认引用计数是 2

**图形用户界面, 文本, 应用程序

描述已自动生成**

 GetExitCodeThread() 的返回值只表示该函数本身执行是否成功并不是说对应线程如果结束了GetExitCodeThread() 函数就必须返回其结束代码,线程是否结束与该函数的返回值根本就是两码事,GetExitCodeThread() 函数返回 FALSE 的唯一情况应该是它无法确定其检测线程的状态.（[(61条消息) 线程函数GetExitCodeThread和ExitThread\_灰灰君呀的博客-CSDN博客\_exitthread](https://blog.csdn.net/onlyfunboy/article/details/24371221)）

文本

描述已自动生成

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）。

问题：图形用户界面, 文本, 应用程序

描述已自动生成

1. 快跑与等待

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。

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(线程#2为正在执行的线程)。

关于 tim e-out，有一个特别重要的用途，但很少被人注意。设定 tim e-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 。

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。

图形用户界面, 文本

描述已自动生成

图形用户界面, 文本

描述已自动生成

MAXIMUM\_WAIT\_OBJECTS为64

WAIT\_ABANDONED 0x00000080：当hHandle为mutex时，如果拥有mutex的线程在结束时没有释放核心对象会引发此返回值。

WAIT\_OBJECT\_0 0x00000000 ：指定的对象处于有信号状态

WAIT\_TIMEOUT 0x00000102：等待超时

WAIT\_FAILED 0xFFFFFFFF ：出现错误，可通过GetLastError得到错误代码

图形用户界面

中度可信度描述已自动生成

1. 同步控制

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

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。

千万不要在一个 critical section 之中调用 Sleep() 或任何 Wait...() API 函数

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 mu tex……等等），会怎样？答案是：不会怎样！

Win32 的 Mutex 用途和 critical section 非常类似，但是它牺牲速度以增 加弹性

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 m ode”就可以进行操作。（译注：作者这里所谓的“user mode”，是相对于 Window s NT 的“kernel mode”而言。至于 Window s 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（mutex名称对整个系统而言是全局性的）。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。

图形用户界面, 文本, 应用程序

描述已自动生成

一旦没有任何线程拥有 mutex，这个 mutex 便处于激发状态。（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。）

文本, 信件

描述已自动生成

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。

图形用户界面, 文本

描述已自动生成

文本

描述已自动生成

semaphore 可以被任意线程锁定，被任意线程释放

文本

描述已自动生成

图形用户界面, 文本

描述已自动生成

文本

描述已自动生成

使用CloseHandle函数关闭句柄。当进程终止时，系统将自动关闭句柄。事件对象会被销毁，当最后一个句柄被关闭。

文本

描述已自动生成

文本, 信件

描述已自动生成

、

Event 对象的激发状态是不能够保留的。

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

1. 不要让线程成为脱缰野马

文本, 信件

描述已自动生成

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。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

文本, 信件

描述已自动生成

文本, 信件

描述已自动生成

文本, 信件

描述已自动生成

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。

6. Overlapped I/O，在你身后变戏法

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。

文本

描述已自动生成

如果你发出许多个 overlapped 请求，那么执行次序无法保证。

文本

描述已自动生成

如果函数成功，则返回值为非零 (TRUE) 。如果函数失败或异步完成，则返回值为零 (FALSE) 。 若要获取扩展的错误信息，请调用 GetLastError 函数。

文本

描述已自动生成

图形用户界面, 文本

描述已自动生成

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 。

图形用户界面, 文本, 应用程序

描述已自动生成

如果你要求一个文件操作为overlapped操作，而操作系统把这个“操作请求”放到队列中等待执行，那么ReadFile()或WriteFile()都会传回FALSE以示失败。但函数返回FALSE有多种原因，可调用GetLastError()调查原因，如果是ERROR\_IO\_PENDING，那说明“操作请求”被放入队列等待执行；其它值，那可能真正代表一个错误了。

图形用户界面, 文本

描述已自动生成

异步I/O调用时，我们会用到OVERLAPPED结构和函数GetOverlappedResult，这个函数的主要作用是将Overlapped返回的结果进行一次简单的分析。[(61条消息) OVERLAPPED结构与GetOverlappedResult函数\_brk1985的博客-CSDN博客](https://blog.csdn.net/brk1985/article/details/28436003?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-28436003-blog-105388181.pc_relevant_3mothn_strategy_and_data_recovery&spm=1001.2101.3001.4242.1&utm_relevant_index=3)

文本

描述已自动生成

文本, 信件

描述已自动生成

是 FileIOCompletionRoutine()。这是一个 callback 函数， 当 overlapped I/O 完成的时候，由系统调用之。

我的测试显示，对于一系列少于 32K B 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。

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。

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。

文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

合理的线程个数应该是 CPU 个数的两倍再加２。

图形用户界面, 文本

描述已自动生成

7. 数据一致性

程序经过编译器的编译之后， 编译器可能会将某些数据放到寄存器中，如果有一个线程改变了变量在内存中的值，其寄存器中的拷贝没有同步的被改，造成了数据的不一致。volatile关键字告诉编译器不要持有变量的临时性拷贝。它可以适用于基本类型，C结构和C++类，如果是后者，那么类的所有成员都会被视为volatile。使用这个关键字，并不会否定 线程同步的需要。

图形用户界面, 文本

描述已自动生成

8. 使用 C Run-time Library

什么是C runtim e library[(62条消息) CRT (C run-time library)简介\_lhw---9999的博客-CSDN博客](https://blog.csdn.net/lhwhit/article/details/109005895)

\_beginthreadex()和CreateThread()的异同 ：\_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 Window s 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。

图形用户界面, 文本, 应用程序

描述已自动生成

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。

文本

描述已自动生成

图形用户界面, 文本, 电子邮件

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

被 \_beginthread() 产生出来的线程所做的第 一件事就是关闭自己的 handle 。

15. 规划一个应用程序

使用多线程的四个原因：

1、  offloading time-consuming task。由辅助线程来执行耗时计算，而使GUI有更好的反应。我想这应该是我们考虑使用线程最多的一种情况吧。  
  
2、  Scalability。服务器软件最常考虑的问题，在程序中产生多个线程，每个线程做一份小的工作，使每个CPU都忙碌，使CPU（一般是多个）有最佳的使用率，达到负载的均衡。  
  
3、  Fair-share resource allocation。当你向一个负荷沉重的服务器发出请求，多少时间才能获得服务。一个服务器不能同时为太多的请求服务，必须有一个请求的最大个数，而且有时候对某些请求要优先处理，这是线程优先级干的活了。  
  
4、  Simulations。线程用于仿真测试。

多进程和多线程

文本

描述已自动生成