

第11章 线程池的使用

第8章讲述了如何使用让线程保持用户方式的机制来实现线程同步的方法。用户方式的同步机制的出色之处在于它的同步速度很快。如果关心线程的运行速度，那么应该了解一下用户方式的同步机制是否适用。

到目前为止，已经知道创建多线程应用程序是非常困难的。需要会面临两个大问题。一个是要对线程的创建和撤消进行管理，另一个是要对线程对资源的访问实施同步。为了对资源访问实施同步，Windows提供了许多基本要素来帮助进行操作，如事件、信标、互斥对象和关键代码段等。这些基本要素的使用都非常方便。为了使操作变得更加方便，唯一的方法是让系统能够自动保护共享资源。不幸的是，在 Windows提供一种让人满意的保护方法之前，我们已经有一种这样的方法。

在如何对线程的创建和撤消进行管理的问题上，人人都有自己的好主意。近年来，我自己创建了若干不同的线程池实现代码，每个实现代码都进行了很好的调整，以便适应特定环境的需要。Microsoft公司的Windows 2000提供了一些新的线程池函数，使得线程的创建、撤消和基本管理变得更加容易。这个新的通用线程池并不完全适合每一种环境，但是它常常可以适合你的需要，并且能够节省大量的程序开发时间。

新的线程池函数使你能够执行下列操作：

- 异步调用函数。
- 按照规定的时间间隔调用函数。
- 当单个内核对象变为已通知状态时调用函数。
- 当异步I/O请求完成时调用函数。

为了完成这些操作，线程池由 4个独立的部分组成。表 11-1显示了这些组件并描述了控制其行为特性的规则。

表11-1 线程池的组件及其行为特性

组 件				
	定 时 器	等 待	I/O	非I/O
线程的初始数值	总是1	0	0	0
当创建一个线程时	当调用第一个线程池函数时	每63个注册对象有一个线程	系统使用试探法，但是这里有一些因素会影响线程的创建： <ul style="list-style-type: none"> • 自从添加线程后已经过去一定的时间（以秒计算） • 使用WT_EXECUTE LONGFUNCTION 标志 • 已经排队的工作项目的数量超过了某个阈值 	
当线程被撤消时	当进程终止运行时	当已经注册的等待对象数量是0时	当线程没有未处理的I/O请求并且已经空闲了一个阈值周期(约1ms)时	当线程空闲了一个阈值周期（约1ms)时
线程如何等待	待命状态	WaitForMultipleOb-jects	待命状态	GetQueued-Comple-tion-Status

(续)

组 件				
定 时 器	等 待	I/O	非I/O	
是什么唤醒了线程	等待定时器通知排队的用户APC	内核对象变为已通知状态	排队的用户APC和已完成的I/O请求	展示已完成的 状态和I/O请示（完成端口最多允许数量为2*的CPU线程同时运行的数量）

当进程初始化时，它并不产生与这些组件相关联的任何开销。但是，一旦新线程池函数之一被调用时，就为进程创建某些组件，并且其中有些组件将被保留，直到进程终止运行为止。如你所见，使用线程池所产生的开销并不小。相当多的线程和内部数据结构变成了你的进程的一个组成部分。因此必须认真考虑线程池能够为你做什么和不能做什么，不要盲目地使用这些函数。

好了，上述说明已经足够了。下面让我们来看一看这些函数能够做些什么。

11.1 方案1：异步调用函数

假设有一个服务器进程，该进程有一个主线程，正在等待客户机的请求。当主线程收到该请求时，它就产生一个专门的线程，以便处理该请求。这使得应用程序的主线程循环运行，并等待另一个客户机的请求。这个方案是客户机 / 服务器应用程序的典型实现方法。虽然它的实现方法非常明确，但是也可以使用新线程池函数来实现它。

当服务器进程的主线程收到客户机的请求时，它可以调用下面这个函数：

```
BOOL QueueUserWorkItem(
    PTHREAD_START_ROUTINE pfnCallback,
    PVOID pvContext,
    ULONG dwFlags);
```

该函数将一个“工作项目”排队放入线程池中的一个线程中并且立即返回。所谓工作项目是指一个（用 pfnCallback 参数标识的）函数，它被调用并传递单个参数 pvContext。最后，线程池中的某个线程将处理该工作项目，导致函数被调用。所编的回调函数必须采用下面的原型：

```
DWORD WINAPI WorkItemFunc(PVOID pvContext);
```

尽管必须使这个函数的原型返回 DWORD，但是它的返回值实际上被忽略了。

注意，你自己从来不调用 CreateThread。系统会自动为你的进程创建一个线程池，线程池中的一个线程将调用你的函数。另外，当该线程处理完客户机的请求之后，该线程并不立即被撤消。它要返回线程池，这样它就可以准备处理已经排队的任何其他工作项目。你的应用程序的运行效率可能会变得更高，因为不必为每个客户机请求创建和撤消线程。另外，由于线程与完成端口相关联，因此可以同时运行的线程数量限制为 CPU 数量的两倍。这就减少了线程的上下文转移的开销。

该函数的内部运行情况是，QueueUserWorkItem 检查非 I/O 组件中的线程数量，然后根据负荷量（已排队的工作项目的数量）将另一个线程添加给该组件。接着 QueueUserWorkItem 执行对 PostQueuedCompletionStatus 的等价调用，将工作项目的信息传递给 I/O 完成端口。最后，在完成端口上等待的线程取出信息（通过调用 GetQueuedCompletionStatus），并调用函数。当函

数返回时，该线程再次调用 `GetQueuedCompletionStatus`，以便等待另一个工作项目。

线程池希望经常处理异步 I/O 请求，即每当线程将一个 I/O 请求排队放入设备驱动程序时，便要处理异步 I/O 请求。当设备驱动程序执行该 I/O 时，请求排队的线程并没有中断运行，而是继续执行其他指令。异步 I/O 是创建高性能可伸缩的应用程序的秘诀，因为它允许单个线程处理来自不同客户机的请求。该线程不必顺序处理这些请求，也不必在等待 I/O 请求运行结束时中断运行。

但是，Windows 对异步 I/O 请求规定了一个限制，即如果线程将一个异步 I/O 请求发送给设备驱动程序，然后终止运行，那么该 I/O 请求就会丢失，并且在 I/O 请求运行结束时，没有线程得到这个通知。在设计良好的线程池中，线程的数量可以根据客户机的需要而增减。因此，如果线程发出一个异步 I/O 请求，然后因为线程池缩小而终止运行，那么该 I/O 请求也会被撤消。因为这种情况实际上并不是你想要的，所以你需要一个解决方案。

如果你想要给发出异步 I/O 请求的工作项目排队，不能将该工作项目插入线程池的非 I/O 组件中。必须将该工作项目放入线程池的 I/O 组件中进行排队。该 I/O 组件由一组线程组成，如果这组线程还有尚未处理的 I/O 请求，那么它们决不能终止运行。因此你只能将它们用来运行发出异步 I/O 请求的代码。

若要为 I/O 组件的工作项目进行排队，仍然必须调用 `QueueUserWorkItem` 函数，但是可以为 `dwFlags` 参数传递 `WT_EXECUTEINIOTHREAD`。通常只需传递 `WT_EXECUTEDEFAULT`（定义为 0），这使得工作项目可以放入非 I/O 组件的线程中。

Windows 提供的函数（如 `RegNotifyChangeKeyValue`）能够异步执行与非 I/O 相关的任务。这些函数也要求调用线程不能终止运行。如果想使用永久线程池的线程来调用这些函数中的一个，可以使用 `WT_EXECUTEINPERSISTENTTHREAD` 标志，它使定时器组件的线程能够执行已排队的工作项目回调函数。由于定时器组件的线程决不会终止运行，因此可以确保最终发生异步操作。应该保证回调函数不会中断，并且保证它能迅速执行，这样，定时器组件的线程就不会受到不利的影响。

设计良好的线程池也必须设法保证线程始终都能处理各个请求。如果线程池包含 4 个线程，并且有 100 个工作项目已经排队，每次只能处理 4 个工作项目。如果个工作项目只需要几个毫秒来运行，那么这是不成问题的。但是，如果工作项目需要运行长得多的时间，那么将无法及时处理这些请求。

当然，系统无法很好地预料工作项目函数将要进行什么操作，但是，如果知道工作项目需要花费很长的时间来运行，那么可以调用 `QueueUserWorkItem` 函数，为它传递 `WT_EXECUTEINPERSISTENTTHREAD` 标志。该标志能够帮助线程池决定是否要将新线程添加给线程池。如果线程池中的所有线程都处于繁忙状态，它就会强制线程池创建一个新线程。因此，如果同时对 10 000 个工作项目进行了排队（使用 `WT_EXECUTEINPERSISTENTTHREAD` 标志），那么这 10 000 个线程就被添加给该线程池。如果不想创建 10 000 个线程，必须分开调用 `QueueUserWorkItem` 函数，这样某些工作项目就有机会完成运行。

线程池不能对线程池中的线程数量规定一个上限，否则就会发生渴求或死锁现象。假如有 10 000 个排队的工作项目，当第 10 001 个项目通知一个事件时，这些工作项目将全部中断运行。如果你已经设置的最大数量为 10 000 个线程，第 10 001 个工作项目没有被执行，那么所有的 10 000 个线程将永远被中断运行。

当使用线程池函数时，应该查找潜在的死锁条件。当然，如果工作项目函数在关键代码段、信标和互斥对象上中断运行，那么必须十分小心，因为这更有可能产生死锁现象。始终都应该

了解哪个组件（I/O、非I/O、等待或定时器等）的线程正在运行你的代码。另外，如果工作项目函数位于可能被动态卸载的DLL中，也要小心。调用已卸载的DLL中的函数的线程将会产生违规访问。若要确保不卸载带有已经排队的工作项目的DLL，必须对已排队工作进行引用计数，在调用QueueUserWorkItem函数之前递增计数器的值，当工作项目函数完成运行时则递减该计数器的值。只有当引用计数降为0时，才能安全地卸载DLL。

11.2 方案2：按规定的時間间隔调用函数

有时应用程序需要在某些时间执行操作任务。Windows提供了一个等待定时器内核对象，因此可以方便地获得基于时间的通知。许多程序员为应用程序执行的每个基于时间的操作任务创建了一个等待定时器对象，但是这是不必要的，会浪费系统资源。相反，可以创建一个等待定时器，将它设置为下一个预定运行的时间，然后为下一个时间重置定时器，如此类推。然而，要编写这样的代码非常困难，不过可以让新线程池函数对此进行管理。

若要调度在某个时间运行的工作项目，首先要调用下面的函数，创建一个定时器队列：

```
HANDLE CreateTimerQueue();
```

定时器队列对一组定时器进行组织安排。例如，有一个可执行文件控制着若干个服务程序。每个服务程序需要触发定时器，以帮助保持它的状态，比如客户机何时不再作出响应，何时收集和更新某些统计信息等。让每个服务程序占用一个等待定时器和专用线程，这是不经济的。相反，每个服务程序可以拥有它自己的定时器队列（这是个轻便的资源），并且共享定时器组件的线程和等待定时器对象。当一个服务程序终止运行时，它只需要删除它的定时器队列即可，因为这会删除该队列创建的所有定时器。

一旦拥有一个定时器队列，就可以在该队列中创建下面的定时器：

```
BOOL CreateTimerQueueTimer(  
    PHANDLE phNewTimer,  
    HANDLE hTimerQueue,  
    WAITORTIMERCALLBACK pfnCallback,  
    PVOID pvContext,  
    DWORD dwDueTime,  
    DWORD dwPeriod,  
    ULONG dwFlags);
```

对于第二个参数，可以传递想要在其中创建定时器的定时器队列的句柄。如果只是创建少数几个定时器，只需要为hTimerQueue参数传递NULL，并且完全避免调用CreateTimerQueue函数。传递NULL，会告诉该函数使用默认的定时器队列，并且简化了你的代码。pfnCallback和pvContext参数用于指明应该调用什么函数以及到了规定的时间应该将什么传递给该函数。dwDueTime参数用于指明应该经过多少毫秒才能第一次调用该函数（如果这个值是0，那么只要可能，就调用该函数，使得CreateTimerQueueTimer函数类似QueueUserWorkItem）。dwPeriod参数用于指明应该经过多少毫秒才能在将来调用该函数。如果为dwPeriod传递0，那么就使它成为一个单步定时器，使工作项目只能进行一次排队。新定时器的句柄通过函数的phNewTimer参数返回。

工作回调函数必须采用下面的原型：

```
VOID WINAPI WaitOrTimerCallback(  
    PVOID pvContext,  
    BOOL fTimerOrWaitFired);
```

当该函数被调用时，fTimerOrWaitFired参数总是TRUE，表示该定时器已经触发。

下面介绍CreateTimerQueueTimer的dwFlags参数。该参数负责告诉函数，当到了规定的时间时，如何给工作项目进行排队。如果想要让非 I/O组件的线程来处理工作项目，可以使用WT_EXECUTEDefault。如果想要在某个时间发出一个异步 I/O请求，可以使用WT_EXECUTEINIOThread。如果想要让一个决不会终止运行的线程来处理该工作项目，可以使用WT_EXECUTEINPERSISTENTThread。如果认为工作项目需要很长的时间来运行，可以使用WT_EXECUTEINLONGFunction。

也可以使用另一个标志，即WT_EXECUTEINTIMERThread，下面将介绍它。在表 11-1 中，能够看到线程池有一个定时器组件。该组件能够创建单个定时器内核对象，并且能够管理它的到期时间。该组件总是由单个线程组成。当调用 CreateTimerQueueTimer函数时，可以使定时器组件的线程醒来，将你的定时器添加给一个定时器队列，并重置等待定时器内核对象。然后该定时器组件的线程便进入待命睡眠状态，等待该等待定时器将一个 APC放入它的队列。当等待定时器将该APC放入队列后，线程就醒来，更新定时器队列，重置等待定时器，然后决定对现在应该运行的工作项目执行什么操作。

接着，该线程要检查下面这些标志：WT_EXECUTEDefault、WT_EXECUTEINIOThread、WT_EXECUTEINPERSISTENTThread、WT_EXECUTEINLONGFunction和WT_EXECUTEINTIMERThread。不过现在可以清楚地看到WT_EXECUTEINTIMERThread标志执行的是什么操作：它使定时器组件的线程能够执行该工作项目。虽然这使工作项目的运行效率更高，但是这非常危险。如果工作项目函数长时间中断运行，那么等待定时器的线程就无法执行任何其他操作。虽然等待定时器可能仍然将APC项目排队放入该线程，但是在当前运行的函数返回之前，这些工作项目不会得到处理。如果打算使用定时器线程来执行代码，那么该代码应该迅速执行，不应该中断。

WT_EXECUTEINIOThread、WT_EXECUTEINPERSISTENTThread和WT_EXECUTEINTIMERThread等标志是互斥的。如果不传递这些标志中的任何一个（或者使用WT_EXECUTEDefault标志），那么工作项目就排队放入I/O组件的线程中。另外，如果设定了WT_EXECUTEINTIMERThread标志，那么WT_EXECUTEINLONGFunction将被忽略。

当不再想要触发定时器时，必须通过调用下面的函数将它删除：

```
BOOL DeleteTimerQueueTimer(  
    HANDLE hTimerQueue,  
    HANDLE hTimer,  
    HANDLE hCompletionEvent);
```

即使对于已经触发的单步定时器，也必须调用该函数。hTimerQueue参数指明定时器位于哪个队列中。hTimer参数指明要删除的定时器，句柄通过较早时调用CreateTimerQueueTimer来返回。

最后一个参数hCompletionEvent告诉你，由于该定时器，什么时候将不再存在没有处理的已排队的工作项目。如果为该参数传递INVALID_HANDLE_VALUE，那么在该定时器的所有已排队工作项目完成运行之前，DeleteTimerQueueTimer函数不会返回。请想一想这将意味着什么。如果在定时器处理自己的工作项目期间对定时器进行一次中断删除，就会造成一个死锁条件。虽然你正在等待工作项目完成处理操作，但是你在等待它完成操作时却中断了它的处理。只有当线程不是处理定时器的的工作项目的线程时，该线程才能进行对定时器的中断删除。

另外，如果你正在使用定时器组件的线程，不应该试图对任何定时器进行中断删除，否则

就会产生死锁。如果试图删除一个定时器，就会将一个 APC 通知放入该定时器组件的线程队列中。如果该线程正在等待一个定时器被删除，而它不能删除该定时器，那么就会发生死锁。

如果不为 `hCompletionEvent` 参数传递 `INVALID_HANDLE_VALUE`，可以传递 `NULL`。这将告诉该函数，你想尽快删除定时器。在这种情况下，`DeleteTimerQueueTimer` 将立即返回，但是你不知道该定时器的所有工作项目何时完成处理。最后，你可以传递一个事件内核对象的句柄作为 `hCompletionEvent` 的参数。当这样操作时，`DeleteTimerQueueTimer` 将立即返回，同时，当定时器的所有已经排队的工作项目完成运行之后，定时器组件的线程将设置该事件。在调用 `DeleteTimerQueueTimer` 之前，千万不要给该事件发送通知，否则你的代码将认为排队的工作项目已经完成运行，但是实际上它们并没有完成。

一旦创建了一个定时器，可以调用下面这个函数来改变它的到期时间和到期周期：

```
BOOL ChangeTimerQueueTimer(  
    HANDLE hTimerQueue,  
    HANDLE hTimer,  
    ULONG dwDueTime,  
    ULONG dwPeriod);
```

这里传递了定时器队列的句柄和想要修改的现有定时器的句柄。可以修改定时器的 `dwDueTime` 和 `dwPeriod`。注意，试图修改已经触发的单步定时器是不起作用的。另外，你可以随意调用该函数，而不必担心死锁。

当不再需要一组定时器时，可以调用下面这个函数，删除定时器队列：

```
BOOL DeleteTimerQueueEx(  
    HANDLE hTimerQueue,  
    HANDLE hCompletionEvent);
```

该函数取出一个现有的定时器队列的句柄，并删除它里面的所有定时器，这样就不必为删除每个定时器而显式调用 `DeleteTimerQueueTimer`。`hCompletionEvent` 参数在这里的语义与它在 `DeleteTimerQueueTimer` 函数中的语义是相同的。这意味着它存在同样的死锁可能性，因此必须小心。

在开始介绍另一个方案之前，让我们说明两个其他的项目。首先，线程池的定时器组件创建等待定时器，这样，它就可以给 APC 项目排队，而不是给对象发送通知。这意味着操作系统能够连续给 APC 项目排队，并且定时器事件从来不会丢失。因此，设置一个定期定时器能够保证每个间隔时间都能为你的工作项目排队。如果创建一个定期定时器，每隔 10s 触发一次，那么每隔 10s 就调用你的回调函数。必须注意这在使用多线程时也会发生必须对工作项目函数的各个部分实施同步。

如果不喜欢这种行为特性，而希望你的工作项目在每个项目执行之后的 10s 进行排队，那么应该在工作项目函数的结尾处创建单步定时器。或者可以创建一个带有高超时值的单个定时器，并在工作项目函数的结尾处调用 `ChangeTimerQueueTimer`。

TimedMsgBox 示例应用程序

清单 11-1 列出的 `TimedMsgBox` 应用程序 (“11 TimedMsgBox.exe”) 显示了如何使用线程池的定时器函数来实现一个用户在规定时间内不作出响应时能自动关闭的消息框。该应用程序的源代码和资源文件位于本书所附光盘上的 11-TimedMsgBox 目录下。

当启动该程序时，它将全局变量 `g_nSecLeft` 设置为 10。这表示用户必须在规定时间内对消息框作出响应的秒数。然后调用 `CreateTimerQueueTimer` 函数，指令线程池每秒钟调用一次

MsgBoxTimeout函数。一旦一切都已初始化，便调用 MessageBox，并向用户显示图 11-1 所示的消息框。

在等待用户作出响应的时候，线程池中的一个线程便调用 MsgBoxTimeout函数。该函数寻找消息框的窗口句柄，对全局变量 g_nSecLeft进行递减，并更新消息框中的字符串。当 MsgBoxTimeout第一次被调用后，消息框就类似下面的样子(见图 11-2)。



图11-1 调用 Message Box 时出现的消息框



图11-2 调用MsgBox Timeout 后出现的消息框

当MsgBoxTimeout第10次被调用时，g_nSecLeft变量变为0，同时MsgBoxTimeout调用 EndDialog函数来撤消该消息框。主线程调用的 MessageBox返回，DeleteTimerQueueTimer被调用，以告诉线程池停止调用 MsgBoxTimeout函数，这时出现图 11-3 所示的消息框，告诉用户他没有在分配给他的时间内对图 11-1 所示的消息框作出响应。

如果用户没有在时间到期之前作出响应，便出现图 11-4 所示的消息框。



图11-3 对图11-1所示的消息框不作出响应时出现的消息框



图11-4 在超时前不作出响应时出现的消息框

清单11-1 TimedMsgBox示例应用程序



TimedMsgBox.cpp

```

/*****
Module: TimedMsgBox.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h"    /* See Appendix A. */
#include <tchar.h>

////////////////////////////////////

// The caption of our message box
TCHAR g_szCaption[] = TEXT("Timed Message Box");

// How many seconds we'll display the message box

```

```

int g_nSecLeft = 0;

// This is STATIC window control ID for a message box
#define ID_MSGBOX_STATIC_TEXT 0x0000ffff

////////////////////////////////////

VOID WINAPI MsgBoxTimeout(PVOID pvContext, BOOLEAN fTimeout) {

    // NOTE: Due to a thread race condition, it is possible (but very unlikely)
    // that the message box will not be created when we get here.
    HWND hwnd = FindWindow(NULL, g_szCaption);

    if (hwnd != NULL) {
        // The window does exist; update the time remaining.
        TCHAR sz[100];
        wsprintf(sz, TEXT("You have %d seconds to respond"), g_nSecLeft--);
        SetDlgItemText(hwnd, ID_MSGBOX_STATIC_TEXT, sz);

        if (g_nSecLeft == 0) {
            // The time is up; force the message box to exit.
            EndDialog(hwnd, IDOK);
        }
    } else {

        // The window does not exist yet; do nothing this time.
        // We'll try again in another second.
    }
}

////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    chWindows9xNotAllowed();

    // How many seconds we'll give the user to respond
    g_nSecLeft = 10;

    // Create a multishot 1-second timer that begins firing after 1 second.
    HANDLE hTimerQTimer;
    CreateTimerQueueTimer(&hTimerQTimer, NULL, MsgBoxTimeout, NULL,
        1000, 1000, 0);

    // Display the message box.
    MessageBox(NULL, TEXT("You have 10 seconds to respond"),
        g_szCaption, MB_OK);

    // Cancel the timer & delete the timer queue
    DeleteTimerQueueTimer(NULL, hTimerQTimer, NULL);
}

```



```
// Let us know if the user responded or if we timed out.
MessageBox(NULL,
    (g_nSecLeft == 0) ? TEXT("Timeout") : TEXT("User responded"),
    TEXT("Result"), MB_OK);

return(0);
}

//////////////////////////////////// End of File //////////////////////////////////////
```

TimedMsgBox.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

////////////////////////////////////
//
// Icon

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_TIMEDMSGBOX ICON DISCARDABLE "TimedMsgBox.ico"

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
```

```

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif // APSTUDIO_INVOKED

#endif // English (U.S.) resources
////////////////////////////////////

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
////////////////////////////////////
#endif // not APSTUDIO_INVOKED

```

11.3 方案3：当单个内核对象变为已通知状态时调用函数

Microsoft发现，许多应用程序产生的线程只是为了等待内核对象变为已通知状态。一旦对象得到通知，该线程就将某种通知移植到另一个线程，然后返回，等待该对象再次被通知。有些编程人员甚至编写了代码，在这种代码中，若干个线程各自等待一个对象。这对系统资源是个很大的浪费。当然，与创建进程相比，创建线程需要的的开销要小得多，但是线程是需要资源的。每个线程有一个堆栈，并且需要大量的CPU指令来创建和撤消线程。始终都应该尽量减少它使用的资源。

如果想在内核对象得到通知时注册一个要执行的工作项目，可以使用另一个新的线程池函数：

```

BOOL RegisterWaitForSingleObject(
    PHANDLE phNewWaitObject,
    HANDLE hObject,
    WAITORTIMERCALLBACK pfnCallback,
    PVOID pvContext,
    ULONG dwMilliseconds,
    ULONG dwFlags);

```

该函数负责将参数传送给线程池的等待组件。你告诉该组件，当内核对象（用 hObject进行标识）得到通知时，你想要对工作项目进行排队。也可以传递一个超时值，这样，即使内核对象没有变为已通知状态，也可以在规定的某个时间内对工作项目进行排队。超时值 0和

INFINITE是合法的。一般来说,该函数的运行情况与 WaitForSingleObject函数(第9章已经介绍)相似。当注册了一个等待组件后,该函数返回一个句柄(通过 phNewWaitObject参数)以标识该等待组件。

在内部,等待组件使用 WaitForMultipleObjects函数来等待已经注册的对象,并且要受到该函数已经存在的任何限制的约束。限制之一是它不能多次等待单个句柄。因此,如果想要多次注册单个对象,必须调用 DuplicateHandle函数,并对原始句柄和复制的句柄分开进行注册。当然,WaitForMultipleObjects能够等待已通知的对象中的任何一个,而不是所有的对象。如果熟悉 WaitForMultipleObjects函数,那么一定知道它一次最多能够等待 64 (MAXIMUM_WAIT_OBJECTS) 个对象。如果用 RegisterWaitForSingleObject函数注册的对象超过 64 个,那么将会出现什么情况呢?这时等待组件就会添加另一个也调用 WaitForMultipleObjects函数的线程。实际上,每隔 63 个对象后,就要将另一个线程添加给该组件,因为这些线程也必须等待负责控制超时的等待定时器对象。

当工作项目准备执行时,它被默认排队放入非 I/O 组件的线程中。这些线程之一最终将会醒来,并且调用你的函数,该函数的原型必须是下面的形式:

```
VOID WINAPI WaitOrTimerCallbackFunc(  
    PVOID pvContext,  
    BOOLEAN fTimerOrWaitFired);
```

如果等待超时了, fTimerOrWaitFired 参数的值是 TRUE。如果等待时对象变为已通知状态,则该参数是 FALSE。

对于 RegisterWaitForSingleObject 函数的 dwFlags 参数,可以传递 WT_EXECUTEINWAITTHREAD,它使等待组件的线程之一运行工作项目函数本身。它的运行速率更高,因为工作项目不必排队放入 I/O 组件中。但是这样做有一定的危险性,因为正在执行工作项目的等待组件函数的线程无法等待其他对象得到通知。只有当工作项目函数运行得很快时,才应该使用该标志。

如果工作项目将要发出异步 I/O 请求,或者使用从不终止运行的线程来执行某些操作,那么也可以传递 WT_EXECUTEINIOTHREAD 或者 WT_EXECUTEINPERSISTENTTHREAD。也可以使用 WT_EXECUTELONGFUNCTION 标志来告诉线程池,你的函数可能要花费较长的时间来运行,而且它应该考虑将一个新线程添加给线程池。只有当工作项目正在被移植到非 I/O 组件或 I/O 组件中时,才能使用该标志,如果使用等待组件的线程,不应该运行长函数。

应该了解的最后一个标志是 WT_EXECUTEONLYONCE。假如你注册一个等待进程内核对象的组件,一旦该进程对象变为已通知状态,它就停留在这个状态中。这会导致等待组件连续地给工作项目排队。对于进程对象来说,可能不需要这个行为特性。如果使用 WT_EXECUTEONLYONCE 标志,就可以防止出现这种情况,该标志将告诉等待组件在工作项目执行了一次后就停止等待该对象。

现在,如果正在等待一个自动重置的事件内核对象。一旦该对象变为已通知状态,该对象就重置为它的未通知状态,并且它的工作项目将被放入队列。这时,该对象仍然处于注册状态,同时,等待组件再次等待该对象被通知,或者等待超时(它已经重置)结束。当不再想让该等待组件等待你的注册对象时,必须取消它的注册状态。即使是使用 WT_EXECUTEONLYONCE 标志注册的并且已经拥有队列的工作项目的等待组件,情况也是如此。调用下面这个函数,可以取消等待组件的注册状态:

```
BOOL UnregisterWaitEx(  

```

```
HANDLE hWaitHandle,  
HANDLE hCompletionEvent);
```

第一个参数指明一个注册的等待（由 RegisterWaitForSingleObject 返回），第二个参数指明当已注册的、正在等待的所有已排队的工作项目已经执行时，你希望如何通知你。与 DeleteTimerQueueTimer 函数一样，可以传递 NULL（如果不要通知的话），或者传递 INVALID_HANDLE_VALUE（中断对函数的调用，直到所有排队的工作项目都已执行），也可以传递一个事件对象的句柄（当排队的工作项目已经执行时，它就会得到通知）。对于无中断的函数调用，如果没有排队的工作项目，那么 UnregisterWaitEx 返回 TRUE，否则它返回 FALSE，而 GetLastError 返回 STATUS_PENDING。

同样，当你将 INVALID_HANDLE_VALUE 传递给 UnregisterWaitEx 时，必须小心避免死锁状态。在试图取消等待组件的注册状态，从而导致工作项目运行时，该工作项目函数不应该中断自己的运行。这好像是说：暂停我的运行，直到我完成运行为止一样——这会导致死锁。然而，如果等待组件的线程运行一个工作项目，而该工作项目取消了导致工作项目运行的等待组件的注册状态，UnregisterWaitEx 是可以用来避免死锁的。还有一点需要说明，在取消等待组件的注册状态之前，不要关闭内核对象的句柄。这会使句柄无效，同时，等待组件的线程会在内部调用 WaitForMultipleObjects 函数，传递一个无效句柄。WaitForMultipleObjects 的运行总是会立即失败，整个等待组件将无法正常工作。

最后，不应该调用 PulseEvent 函数来通知注册的事件对象。如果这样做了，等待组件的线程就可能忙于执行某些别的操作，从而错过了事件的触发。这不应该是个新问题了。PulseEvent 几乎能够避免所有的线程结构产生这个问题。

11.4 方案4：当异步 I/O 请求完成运行时调用函数

最后一个方案是个常用的方案，即服务器应用程序发出某些异步 I/O 请求，当这些请求完成时，需要让一个线程池准备好来处理已完成的 I/O 请求。这个结构是 I/O 完成端口原先设计时所针对的一种结构。如果要管理自己的线程池，就要创建一个 I/O 完成端口，并创建一个等待该端口的线程池。还需要打开多个 I/O 设备，将它们的句柄与完成端口关联起来。当异步 I/O 请求完成时，设备驱动程序就将“工作项目”排队列入该完成端口。

这是一种非常出色的结构，它使少数线程能够有效地处理若干个工作项目，同时它又是一种很特殊的结构，因为线程池函数内置了这个结构，使你可以节省大量的设计和精力。若要利用这个结构，只需要打开设备，将它与线程池的非 I/O 组件关联起来。记住，I/O 组件的线程全部在一个 I/O 组件端口上等待。若要将一个设备与该组件关联起来，可以调用下面的函数：

```
BOOL BindIoCompletionCallback(  
    HANDLE hDevice,  
    POVERLAPPED_COMPLETION_ROUTINE pfnCallback,  
    ULONG dwFlags);
```

该函数在内部调用 CreateIoCompletionPort，传递 hDevice 和内部完成端口的句柄。调用 BindIoCompletionCallback 也可以保证至少有一个线程始终在非 I/O 组件中。与该设备相关联的完成关键字是重叠完成例程的地址。这样，当该设备的 I/O 运行完成时，非 I/O 组件就知道要调用哪个函数，以便它能够处理已完成的 I/O 请求。该完成例程必须采用下面的原型：

```
VOID WINAPI OverlappedCompletionRoutine(  
    DWORD dwErrorCode,  
    DWORD dwNumberOfBytesTransferred,
```

```
POVERLAPPED pOverlapped);
```

你将会注意到没有将一个 OVERLAPPED 结构传递给 BindIoCompletionCallback。OVERLAPPED 结构被传递给 ReadFile 和 WriteFile 之类的函数。系统在内部始终保持对这个带有待处理 I/O 请求的重叠结构进行跟踪。当该请求完成时，系统将该结构的地址放入完成端口，从而使它能够被传递给你的 OverlappedCompletionRoutine 函数。另外，由于该完成例程的地址是完成的关键，因此，如果要将更多的上下文信息放入 OverlappedCompletionRoutine 函数，应使用将上下文信息放入 OVERLAPPED 结构的结尾处的传统方法。

还应该知道，关闭设备会导致它的所有待处理的 I/O 请求立即完成，并产生一个错误代码。要作好准备，在你的回调函数中处理这种情况。如果关闭设备后你想确保没有运行任何回调函数，那么必须引用应用程序中的计数特性。换句话说，每次发出一个 I/O 请求时，必须使计数器的计数递增，每次完成一个 I/O 请求，则递减计数器的计数。

目前没有特殊的标志可以传递给 BindIoCompletionCallback 函数的 dwFlags 参数，因此必须传递 0。相信你能够传递的标志是 WT_EXECUTEINIOTHREAD。如果一个 I/O 请求已经完成，它将被排队放入一个非 I/O 组件线程。在 OverlappedCompletionRoutine 函数中，可以发出另一个异步 I/O 请求。但是记住，如果发出 I/O 请求的线程终止运行，该 I/O 请求也会被撤消。另外，非 I/O 组件中的线程是根据工作量来创建或撤消的。如果工作量很小，该组件中的线程就会终止运行，其 I/O 请求仍然处于未处理状态。如果 BindIoCompletionCallback 函数支持 WT_EXECUTEINIOTHREAD 标志，那么在完成端口上等待的线程就会醒来，并将结果移植到一个 I/O 组件的线程中。由于在 I/O 请求处于未处理状态下时这些线程决不会终止运行，因此可以发出 I/O 请求而不必担心它们被撤消。

虽然 WT_EXECUTEINIOTHREAD 标志的作用不错，但是可以很容易模仿刚才介绍的行为特性。在 OverlappedCompletionRoutine 函数中，只需要调用 QueueUserWorkItem，传递 WT_EXECUTEINIOTHREAD 标志和想要的任何数据（至少是重叠结构）。这就是线程池函数能够为你执行的全部功能。