第9章　线程与内核对象的同步

上一章介绍了如何使用允许线程保留在用户方式中的机制来实现线程同步的方法。用户方式同步的优点是它的同步速度非常快。如果强调线程的运行速度，那么首先应该确定用户方式的线程同步机制是否适合需要。

虽然用户方式的线程同步机制具有速度快的优点，但是它也有其局限性。对于许多应用程序来说，这种机制是不适用的。例如，互锁函数家族只能在单值上运行，根本无法使线程进入等待状态。可以使用关键代码段使线程进入等待状态，但是只能用这些代码段对单个进程中的线程实施同步。还有，使用关键代码段时，很容易陷入死锁状态，因为在等待进入关键代码段时无法设定超时值。

本章将要介绍如何使用内核对象来实现线程的同步。你将会看到，内核对象机制的适应性远远优于用户方式机制。实际上，内核对象机制的唯一不足之处是它的速度比较慢。当调用本章中提到的任何新函数时，调用线程必须从用户方式转为内核方式。这个转换需要很大的代价：往返一次需要占用x86平台上的大约1000个CPU周期，当然，这还不包括执行内核方式代码，即实现线程调用的函数的代码所需的时间。

本书介绍了若干种内核对象，包括进程，线程和作业。可以将所有这些内核对象用于同步目的。对于线程同步来说，这些内核对象中的每种对象都可以说是处于已通知或未通知的状态之中。这种状态的切换是由Microsoft为每个对象建立的一套规则来决定的。例如，进程内核对象总是在未通知状态中创建的。当进程终止运行时，操作系统自动使该进程的内核对象处于已通知状态。一旦进程内核对象得到通知，它将永远保持这种状态，它的状态永远不会改为未通知状态。

当进程正在运行的时候，进程内核对象处于未通知状态，当进程终止运行的时候，它就变为已通知状态。进程内核对象中是个布尔值，当对象创建时，该值被初始化为FALSE（未通知状态）。当进程终止运行时，操作系统自动将对应的对象布尔值改为TRUE，表示该对象已经得到通知。

如果编写的代码是用于检查进程是否仍在运行，那么只需要调用一个函数，让操作系统去检查进程对象的布尔值，这非常简单。你也可能想要告诉系统使线程进入等待状态，然后当布尔值从FALSE改为TRUE时自动唤醒该线程。这样，你可以编写一个代码，在这个代码中，需要等待子进程终止运行的父进程中的线程只需要使自己进入睡眠状态，直到标识子进程的内核对象变为已通知状态即可。你将会看到，Microsoft的Windows提供了一些能够非常容易地完成这些操作的函数。

刚才讲了Microsoft为进程内核对象定义了一些规则。实际上，线程内核对象也遵循同样的规则。即线程内核对象总是在未通知状态中创建。当线程终止运行时，操作系统会自动将线程对象的状态改为已通知状态。因此，可以将相同的方法用于应用程序，以确定线程是否不再运行。与进程内核对象一样，线程内核对象也可以处于已通知状态或未通知状态。

下面的内核对象可以处于已通知状态或未通知状态：

- 进程
- 线程
- 作业
- 文件修改通知
- 事件
- 可等待定时器

- 文件
- 信标
- 控制台输入
- 互斥对象

线程可以使自己进入等待状态，直到一个对象变为已通知状态。注意，用于控制每个对象的已通知/未通知状态的规则要根据对象的类型而定。前面已经提到进程和线程对象的规则及作业的规则。

本章将要介绍允许线程等待某个内核对象变为已通知状态所用的函数。然后我们将要讲述Windows提供的专门用来帮助实现线程同步的各种内核对象、如事件、等待计数器，信标和互斥对象。

当我最初开始学习这项内容时，我设想内核对象包含了一面旗帜（在空中飘扬的旗帜，不是耷拉下来的旗帜），这对我很有帮助。当内核对象得到通知时，旗帜升起来；当对象未得到通知时，旗帜就降下来（见图9-1）。

当线程等待的对象处于未通知状态（旗帜降下）中时，这些线程不可调度。但是一旦对象变为已通知状态（旗帜升起），线程看到该标志变为可调度状态，并且很快恢复运行（见图9-2）。

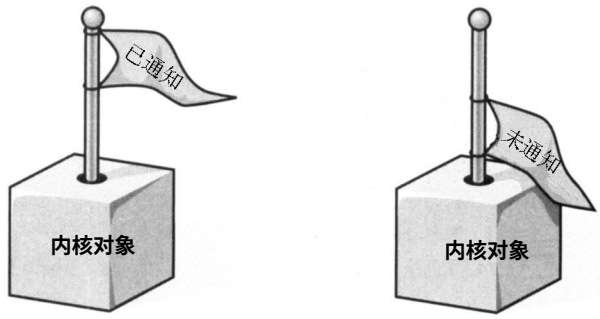

图9-1 内核对象中的旗帜状态

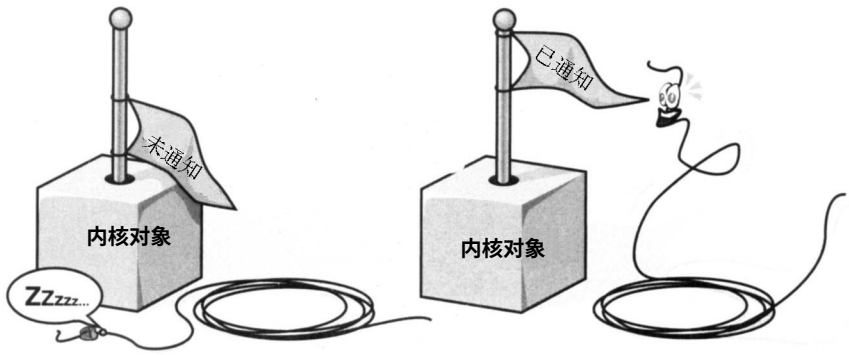

图9-2 内核对象中旗帜状态与线程可调度性示意图

9.1 等待函数

等待函数可使线程自愿进入等待状态，直到一个特定的内核对象变为已通知状态为止。这些等待函数中最常用的是WaitForSingleObject:

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

当线程调用该函数时，第一个参数 hObject标识一个能够支持被通知/未通知的内核对象（前面列出的任何一种对象都适用）。第二个参数dwMilliseconds允许该线程指明，为了等待该对象变为已通知状态，它将等待多长时间。

调用下面这个函数将告诉系统，调用函数准备等待到hProcess句柄标识的进程终止运行为止：

```
WaitForSingleObject(hProcess, INFINITE);
```

第二个参数告诉系统，调用线程愿意永远等待下去（无限时间量），直到该进程终止运行。

通常情况下，INFINITE是作为第二个参数传递给WaitForSingleObject的，不过也可以传递任何一个值（以毫秒计算）。顺便说一下，INFINITE已经定义为0xFFFFFFFF（或-1）。当然，传递INFINITE有些危险。如果对象永远不变为已通知状态，那么调用线程永远不会被唤醒，它将永远处于死锁状态，不过，它不会浪费宝贵的CPU时间。

下面是如何用一个超时值而不是INFINITE来调用WaitForSingleObject的例子：

```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {

   case WAIT_OBJECT_0:
      // The process terminated.
      break;

   case WAIT_TIMEOUT:
      // The process did not terminate within 5000 milliseconds.
      break;

   case WAIT_FAILED:
      // Bad call to function (invalid handle?)
      break;
}
```

上面这个代码告诉系统，在特定的进程终止运行之前，或者在 5000ms时间结束之前，调用线程不应该变为可调度状态。因此，如果进程终止运行，那么这个函数调用将在不到5000ms的时间内返回，如果进程尚未终止运行，那么它在大约 5000ms时间内返回。注意，不能为dwMillisecond传递0。如果传递了0，WaitForSingleObject函数将总是立即返回。

WaitForSingleObject的返回值能够指明调用线程为什么再次变为可调度状态。如果线程等待的对象变为已通知状态，那么返回值是 WAIT_OBJECT_0。如果设置的超时已经到期，则返回值是WAIT_TIMEOUT。如果将一个错误的值（如一个无效句柄）传递给 WaitForSingleObject，那么返回值将是WAIT_FAILED（若要了解详细信息，可调用GetLastError）。

下面这个函数WaitForMultipleObjects与WaitForSingleObject函数很相似，区别在于它允许调用线程同时查看若干个内核对象的已通知状态：

```
DWORD WaitForMultipleObjects(
   DWORD dwCount,
   CONST HANDLE* phObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds);
```

dwCount参数用于指明想要让函数查看的内核对象的数量。这个值必须在 1与MAXIMUM_WAIT_OBJECTS（在Windows头文件中定义为64）之间。phObjects参数是指向内核对象句柄的数组的指针。

可以以两种不同的方式来使用WaitForMultipleObjects函数。一种方式是让线程进入等待状

态，直到指定内核对象中的任何一个变为已通知状态。另一种方式是让线程进入等待状态，直到所有指定的内核对象都变为已通知状态。fWaitAll参数告诉该函数，你想要让它使用何种方式。如果为该参数传递TRUE，那么在所有对象变为已通知状态之前，该函数将不允许调用线程运行。

dwMilliseconds参数的作用与它在WaitForSingleObject中的作用完全相同。如果在等待的时候规定的时间到了，那么该函数无论如何都会返回。同样，通常为该参数传递INFINITE，但是在编写代码时应该小心，以避免出现死锁情况。

WaitForMultipleObjects函数的返回值告诉调用线程，为什么它会被重新调度。可能的返回值是WAIT_FAILED和WAIT_TIMEOUT，这两个值的作用是很清楚的。如果为fWaitAll参数传递TRUE，同时所有对象均变为已通知状态，那么返回值是WAIT_OBJECT_0。如果为fWaitAll传递FALSE，那么一旦任何一个对象变为已通知状态，该函数便返回。在这种情况下，你可能想要知道哪个对象变为已通知状态。返回值是WAIT_OBJECT_0与（WAIT_OBJECT_0+dwCount-1）之间的一个值。换句话说，如果返回值不是WAIT_TIMEOUT，也不是WAIT_FAILED，那么应该从返回值中减去WAIT_OBJECT_0。产生的数字是作为第二个参数传递给WaitForMultipleObjects的句柄数组中的索引。该索引说明哪个对象变为已通知状态。下面是说明这一情况的一些示例代码：

```
HANDLE h[3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dw) {
   case WAIT_FAILED:
      // Bad call to function (invalid handle?)
      break;

   case WAIT_TIMEOUT:
      // None of the objects became signaled within 5000 milliseconds.
      break;

   case WAIT_OBJECT_0 + 0:
      // The process identified by h[0] (hProcess1) terminated.
      break;

   case WAIT_OBJECT_0 + 1:
      // The process identified by h[1] (hProcess2) terminated.
      break;

   case WAIT_OBJECT_0 + 2:
      // The process identified by h[2] (hProcess3) terminated.
      break;
}
```

如果为fWaitAll参数传递FALSE，WaitForMultipleObjects就从索引0开始向上对句柄数组进行扫描，同时已通知的第一个对象终止等待状态。这可能产生一些你不希望有的结果。例如，通过将3个进程句柄传递给该函数，你的线程就会等待3个子进程终止运行。如果数组中索引为0的进程终止运行，WaitForMultipleObjects就会返回。这时该线程就可以做它需要的任何事情，然后循环反复，等待另一个进程终止运行。如果该线程传递相同的3个句柄，该函数立即再次

返回WAIT_OBJECT_0。除非删除已经收到通知的句柄，否则代码就无法正确地运行。

9.2 成功等待的副作用

对于有些内核对象来说，成功地调用WaitForSingleObject和WaitForMultipleObjects，实际上会改变对象的状态。成功地调用是指函数发现对象已经得到通知并且返回一个相对于WAIT_OBJECT_0的值。如果函数返回WAIT_TIMEOUT或WAIT_FAILED，那么调用就没有成功。如果函数调用没有成功，对象的状态就不可能改变。

当一个对象的状态改变时，我称之为成功等待的副作用。例如，有一个线程正在等待自动清除事件对象（本章后面将要介绍）。当事件对象变为已通知状态时，函数就会发现这个情况，并将WAIT_OBJECT_0返回给调用线程。但是就在函数返回之前，该事件将被置为未通知状态，这就是成功等待的副作用。

这个副作用将用于自动清除内核对象，因为它是Microsoft为这种类型的对象定义的规则之一。其他对象拥有不同的副作用，而有些对象则根本没有任何副作用。进程和线程内核对象就根本没有任何副作用，也就是说，在这些对象之一上进行等待决不会改变对象的状态。由于本章要介绍各种不同的内核对象，因此我们将要详细说明它们的成功等待的副作用。

究竟是什么原因使得WaitForMultipleObjects函数如此有用呢，因为它能够以原子操作方式来执行它的所有操作。当一个线程调用WaitForMultipleObjects函数时，该函数能够测试所有对象的通知状态，并且能够将所有必要的副作用作为一项操作来执行。

让我们观察一个例子。两个线程以完全相同的方式来调用WaitForMultipleObjects：

```
HANDLE h[2];
h[0] = hAutoResetEvent1;   // Initially nonsignaled
h[1] = hAutoResetEvent2;   // Initially nonsignaled
WaitForMultipleObjects(2, h, TRUE, INFINITE);
```

当WaitForMultipleObjects函数被调用时，两个事件都处于未通知状态，这就迫使两个线程都进入等待状态。然后hAutoResetEvent1对象变为已通知状态。两个线程都发现，该事件已经变为已通知状态，但是它们都无法被唤醒，因为hAutoResetEvent2仍然处于未通知状态。由于两个线程都没有等待成功，因此没有对hAutoResetEvent1对象产生任何副作用。

接着，hAutoResetEvent2变为已通知状态。这时，两个线程中的一个发现，两个对象都变为已通知状态。等待取得了成功，两个事件对象均被置为未通知状态，该线程变为可调度的线程。但是另一个线程的情况如何呢？它将继续等待，直到它发现两个事件对象都处于已通知状态。尽管它原先发现hAutoResetEvent1处于已通知状态，但是现在它将该对象视为未通知状态。

前面讲过，有一个重要问题必须注意，即WaitForMultipleObjects是以原子操作方式运行的。当它检查内核对象的状态时，其他任何线程都无法背着对象改变它的状态。这可以防止出现死锁情况。试想，如果一个线程看到hAutoResetEvent1已经得到通知并将事件重置为未通知状态，然后，另一个线程发现hAutoResetEvent2已经得到通知并将该事件重置为未通知状态，那么这两个线程均将被冻结：一个线程将等待另一个线程已经得到的对象，另一个线程将等待该线程已经得到的对象。WaitForMultipleObjects能够确保这种情况永远不会发生。

这会产生一个非常有趣的问题，即如果多个线程等待单个内核对象，那么当该对象变成已通知状态时，系统究竟决定唤醒哪个线程呢？Microsoft对这个问题的正式回答是：“算法是公平的。”Microsoft不想使用系统使用的内部算法。它只是说该算法是公平的，这意味着如果多

个线程正在等待，那么每当对象变为已通知状态时，每个线程都应该得到它自己的被唤醒的机会。

这意味着线程的优先级不起任何作用，即高优先级线程不一定得到该对象。这还意味着等待时间最长的线程不一定得到该对象。同时得到对象的线程有可能反复循环，并且再次得到该对象。但是，这对于其他线程来说是不公平的，因此该算法将设法防止这种情况的出现。但是这不一定做得到。

在实际操作中，Microsoft使用的算法是常用的“先进先出”的方案。等待了最长时间的线程将得到该对象。但是系统中将会执行一些操作，以便改变这个行为特性，使它不太容易预测。这就是为什么Microsoft没有明确说明该算法如何起作用的原因。操作之一是让线程暂停运行。如果一个线程等待一个对象，然后该线程暂停运行，那么系统就会忘记该线程正在等待该对象。这是一个特性，因为没有理由为一个暂停运行的线程进行调度。当后来该线程恢复运行时，系统将认为该线程刚刚开始等待该对象。

当调试一个进程时，只要到达一个断点，该进程中的所有线程均暂停运行。因此，调试一个进程会使“先进先出”的算法很难预测其结果，因为线程常常暂停运行，然后再恢复运行。

9.3 事件内核对象

在所有的内核对象中，事件内核对象是个最基本的对象。它们包含一个使用计数（与所有内核对象一样），一个用于指明该事件是个自动重置的事件还是一个人工重置的事件的布尔值，另一个用于指明该事件处于已通知状态还是未通知状态的布尔值。

事件能够通知一个操作已经完成。有两种不同类型的事件对象。一种是人工重置的事件，另一种是自动重置的事件。当人工重置的事件得到通知时，等待该事件的所有线程均变为可调度线程。当一个自动重置的事件得到通知时，等待该事件的线程中只有一个线程变为可调度线程。

当一个线程执行初始化操作，然后通知另一个线程执行剩余的操作时，事件使用得最多。事件初始化为未通知状态，然后，当该线程完成它的初始化操作后，它就将事件设置为已通知状态。这时，一直在等待该事件的另一个线程发现该事件已经得到通知，因此它就变成可调度线程。这第二个线程知道第一个线程已经完成了它的操作。

下面是CreateEvent函数，用于创建事件内核对象：

```
HANDLE CreateEvent(
   PSECURITY_ATTRIBUTES psa,
   BOOL fManualReset,
   BOOL fInitialState,
   PCTSTR pszName);
```

第3章已经介绍了内核对象的操作技巧，比如，如何设置它们的安全性，如何进行使用计数，如何继承它们的句柄，如何按名字共享对象等。由于现在你对所有这些对象都已经熟悉了，所以不再介绍该函数的第一个和最后一个参数。

FMannualReset参数是个布尔值，它能够告诉系统是创建一个人工重置的事件（TRUE）还是创建一个自动重置的事件（FALSE）。fInitialState参数用于指明该事件是要初始化为已通知状态（TRUE）还是未通知状态（FALSE）。当系统创建事件对象后，createEvent就将与进程相关的句柄返回给事件对象。其他进程中的线程可以获得对该对象的访问权，方法是使用在pszName参数中传递的相同值，使用继承性，使用 DuplicateHandle函数等来调用CreateEvent，或者调用OpenEvent,在pszName参数中设定一个与调用CreateEvent时设定的名字相匹配的名字：

```
HANDLE OpenEvent(
   DWORD fdwAccess,
   BOOL fInherit,
   PCTSTR pszName);
```

与所有情况中一样，当不再需要事件内核对象时，应该调用CloseHandle函数。

一旦事件已经创建，就可以直接控制它的状态。当调用SetEvent时，可以将事件改为已通知状态：

```
BOOL SetEvent(HANDLE hEvent);
```

当调用ResetEvent函数时，可以将该事件改为未通知状态：

```
BOOL ResetEvent(HANDLE hEvent);
```

就是这么容易。

Microsoft为自动重置的事件定义了应该成功等待的副作用规则，即当线程成功地等待到该对象时，自动重置的事件就会自动重置到未通知状态。这就是自动重置的事件如何获得它们的名字的方法。通常没有必要为自动重置的事件调用ResetEvent函数，因为系统会自动对事件进行重置。但是，Microsoft没有为人工重置的事件定义成功等待的副作用。

让我们观察一个简单的例子，以便说明如何使用事件内核对象对线程进行同步。下面就是这个代码：

```
// Create a global handle to a manual-reset, nonsignaled event.
HANDLE g_hEvent;

int WINAPI WinMain(...) {

   // Create the manual-reset, nonsignaled event.
   g_hEvent = CreateEve\nt(NULL, TRUE, FALSE, NULL);

   // Spawn 3 new threads.
   HANDLE hThread[3];
   DWORD dwThreadID;
   hThread[0] = _beginthreadex(NULL, 0, WordCount, NULL, 0, &dwThreadID);
   hThread[1] = _beginthreadex(NULL, 0, SpellCheck, NULL, 0, &dwThreadID);
   hThread[2] = _beginthreadex(NULL, 0, GrammarCheck, NULL, 0, &dwThreadID);


   OpenFileAndReadContentsIntoMemory(...);

   // Allow all 3 threads to access the memory.
   SetEvent(g_hEvent);
   ...
}

DWORD WINAPI WordCount(PVOID pvParam) {
   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   return(0);
}

DWORD WINAPI SpellCheck (PVOID pvParam) {
```

```
   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   return(0);
}

DWORD WINAPI GrammarCheck (PVOID pvParam) {

   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   return(0);
}
```

当这个进程启动时，它创建一个人工重置的未通知状态的事件，并且将句柄保存在一个全局变量中。这使得该进程中的其他线程能够非常容易地访问同一个事件对象。现在3个线程已经产生。这些线程要等待文件的内容读入内存，然后每个线程都要访问它的数据。一个线程进行单词计数，另一个线程运行拼写检查器，第三个线程运行语法检查器。这3个线程函数的代码的开始部分都相同，每个函数都调用WaitForSingleObject，这将使线程暂停运行，直到文件的内容由主线程读入内存为止。

一旦主线程将数据准备好，它就调用SetEvent，给事件发出通知信号。这时，系统就使所有这3个辅助线程进入可调度状态，它们都获得了CPU时间，并且可以访问内存块。注意，这3个线程都以只读方式访问内存。这就是所有3个线程能够同时运行的唯一原因。还要注意，如何计算机上配有多个CPU，那么所有3个线程都能够真正地同时运行，从而可以在很短的时间内完成大量的操作。

如果你使用自动重置的事件而不是人工重置的事件，那么应用程序的行为特性就有很大的差别。当主线程调用SetEvent之后，系统只允许一个辅助线程变成可调度状态。同样，也无法保证系统将使哪个线程变为可调度状态。其余两个辅助线程将继续等待。

已经变为可调度状态的线程拥有对内存块的独占访问权。让我们重新编写线程的函数，使得每个函数在返回前调用SetEvent函数（就像WinMain函数所做的那样）。这些线程函数现在变成下面的形式：

```
DWORD WINAPI WordCount(PVOID pvParam) {

   // Wait until the file's data is in memory.
   WaitForSingleObject(g_hEvent, INFINITE);

   // Access the memory block.
   ...
   SetEvent(g_hEvent);
   return(0);
}

DWORD WINAPI SpellCheck (PVOID pvParam) {

   // Wait until the file's data is in memory.
```

```
    WaitForSingleObject(g_hEvent, INFINITE);

    // Access the memory block.
    ...
    SetEvent(g_hEvent);
    return(0);
}

DWORD WINAPI GrammarCheck (PVOID pvParam) {

    // Wait until the file's data is in memory.
    WaitForSingleObject(g_hEvent, INFINITE);

    // Access the memory block.
    ...
    SetEvent(g_hEvent);
    return(0);
}
```

当线程完成它对数据的专门传递时，它就调用 SetEvent函数，该函数允许系统使得两个正在等待的线程中的一个成为可调度线程。同样，我们不知道系统将选择哪个线程作为可调度线程，但是该线程将进行它自己的对内存块的专门传递。当该线程完成操作时，它也将调用SetEvent函数，使第三个即最后一个线程进行它自己的对内存块的传递。注意，当使用自动重置事件时，如果每个辅助线程均以读 /写方式访问内存块，那么就不会产生任何问题，这些线程将不再被要求将数据视为只读数据。这个例子清楚地展示出使用人工重置事件与自动重置事件之间的差别。

为了完整起见，下面再介绍一个可以用于事件的函数：

```
BOOL PulseEvent(HANDLE hEvent);
```

PulseEvent函数使得事件变为已通知状态，然后立即又变为未通知状态，这就像在调用SetEvent后又立即调用ResetEvent函数一样。如果在人工重置的事件上调用 PulseEvent函数，那么在发出该事件时，等待该事件的任何一个线程或所有线程将变为可调度线程。如果在自动重置事件上调用PulseEvent函数，那么只有一个等待该事件的线程变为可调度线程。如果在发出事件时没有任何线程在等待该事件，那么将不起任何作用。

PulseEvent函数并不非常有用。实际上我在自己的应用程序中从未使用它，因为根本不知道什么线程将会看到事件的发出并变成可调度线程。由于在调用 PulseEvent时无法知道任何线程的状态，因此该函数并不那么有用。我相信在有些情况下，虽然 PulseEvent函数可以方便地供你使用，但是你根本想不起要去使用它。关于 PulseEvent函数的比较详细的说明，请参见本章后面对SingleObjectAndWait函数的介绍。

Handshake示例应用程序

清单9-1中列出了Handshake（“09 Handshake.exe ”）应用程序，它展示了自动重置事件的使用情况。该应用程序的源代码文件和资源文件均在本书所附光盘上的 09-Handshake目录下。当运行Handshake应用程序时，就会出现图9-3所示的对话框。

Handshake应用程序接受一个请求字符串，再将该字符串中的所有字符反转，然后将结果放入Result域。Handshake应用程序的出色之处在于它完成这个重要任务时所用的方法不同一般。

Handshake能够解决常见的编程问题。现在有一个客户机和一个服务器，它们之间需要互

相进行通信。开始时，服务器无事可做，因此它进入等待状态。当客户机准备将一个请求提交给服务器时，它将该请求放入一个共享内存缓冲区中，然后发出一个事件通知，这样，服务器线程就会知道查看数据缓冲区并处理客户机的请求。当服务器线程忙于处理该请求的时候，客户机的线程必须进入等待状态，直到服务器准备好请求的结果为止。因此客户机进入等待状态，直到服务器发出另一个事件通知，指明结果已经准备好，可供客户机进行处理。当客户机再次被唤醒的时候，它就知道结果已经放入共享数据缓冲区中，并且可以将结果显示给用户。

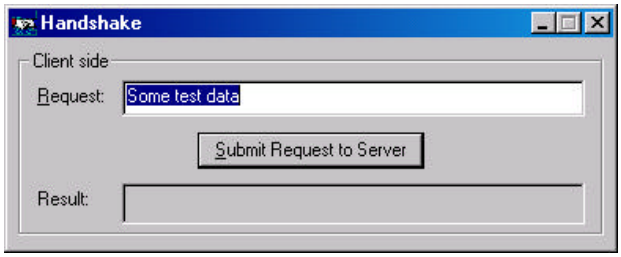

图9-3 Handshake 对话框

当该应用程序启动运行时，它立即创建两个未通知的自动重置的事件对象。一个事件是 g_hevtRequestSubmitted，用于指明何时为服务器准备一个请求。该事件由服务器线程等待，并由客户机线程发出通知。第二个事件是g_hevtResultReturned，用来指明何时为客户机准备好结果。客户机线程等待该事件，而服务器线程则负责发出该事件的通知。

当各个事件创建后，服务器线程就产生并且执行 ServerThread函数。该函数立即让服务器等待客户机的请求。与此同时，主线程（它也是客户机线程）调用 DialogBox函数，该函数负责显示应用程序的用户界面。可以将一些文字输入Request域，然后，当点击Submit Request To Server(将请求提交给服务器)时，请求字符串将被放入由客户机和服务器线程共享的一个缓冲区，并发出g_hevtRequestSubmitted事件的通知。然后客户机线程通过等待g_hevtResultReturened事件来等待服务器的结果。

服务器醒来，将共享内存缓冲区中的字符串反转，然后发出 g_hevtResultReturned事件的通知。服务器的线程循环运行，以便等待客户机的另一个请求。注意，该应用程序决不会调用ResetEvent函数，因为没有必要。自动重置的事件在等待成功后会自动恢复未通知状态。与此同时，客户机线程发现g_hevtResultReturned事件已经变为已通知状态。它醒来，并将字符串从共享内存缓冲区拷贝到用户界面的Result域。

也许这个应用程序剩下的唯一的一个值得注意的特性是它如何关闭。若要关闭该应用程序，只需要关闭它的对话框。这会导致调用的 _tWinMain中的DialogBox函数返回。这时，主线程将一个特殊字符串拷贝到共享缓冲区，并唤醒服务器的线程，以便处理该特殊请求。主线程等待服务器线程确认请求已经收到，并等待服务器线程终止运行。当服务器线程发现该特殊的客户机请求字符串时，它就退出循环，而该线程则终止运行。

我选择让主线程等待服务器线程终止运行，方法是调用WaitForMultipleObjects函数，这样，就可以看到该函数是如何使用的。实际上，也可以调用 WaitForSingleObject函数，传递服务器线程的句柄，一切将以完全相同的方式来运行。

一旦主线程知道服务器线程已经停止运行后，我将 3次调用CloseHandle函数，以便正确地撤消应用程序正在使用的所有内核对象。当然，系统能够自动执行这项操作，但是如果我自己进行操作，我的感觉会更好些。我喜欢能够随时控制我的代码。

清单9-1 Handshake示例应用程序

Handshake.cpp

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


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>       // For beginthreadex
#include "Resource.h"


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


// This event is signaled when the client has a request for the server
HANDLE g_hevtRequestSubmitted;

// This event is signaled when the server has a result for the client
HANDLE g_hevtResultReturned;

// The buffer shared between the client and server threads
TCHAR  g_szSharedRequestAndResultBuffer[1024];
// The special value sent from the client that causes the
// server thread to terminate cleanly.
TCHAR  g_szServerShutdown[] = TEXT("Server Shutdown");


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


// This is the code executed by the server thread
DWORD WINAPI ServerThread(PVOID pvParam) {

   // Assume that the server thread is to run forever
   BOOL fShutdown = FALSE;

   while (!fShutdown) {

      // Wait for the client to submit a request
      WaitForSingleObject(g_hevtRequestSubmitted, INFINITE);

      // Check to see if the client wants the server to terminate
      fShutdown =
         (lstrcmpi(g_szSharedRequestAndResultBuffer, g_szServerShutdown) == 0);

      if (!fShutdown) {
         // Process the client's request (reverse the string)
         _tcsrev(g_szSharedRequestAndResultBuffer);
      }
```

```
      // Let the client process the request's result
      SetEvent(g_hevtResultReturned);
   }

   // The client wants us to shutdown, exit
   return(0);
}

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

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_HANDSHAKE);

   // Initialize the edit control with some test data request
   Edit_SetText(GetDlgItem(hwnd, IDC_REQUEST), TEXT("Some test data"));

   return(TRUE);
}

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

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {

      case IDCANCEL:
         EndDialog(hwnd, id);
         break;

      case IDC_SUBMIT:  // Submit a request to the server thread

         // Copy the request string into the shared data buffer
         Edit_GetText(GetDlgItem(hwnd, IDC_REQUEST),
            g_szSharedRequestAndResultBuffer,
            chDIMOF(g_szSharedRequestAndResultBuffer));

         // Let the server thread know that a request is ready in the buffer
         SetEvent(g_hevtRequestSubmitted);

         // Wait for the server to process the request and give us the result
         WaitForSingleObject(g_hevtResultReturned, INFINITE);

         // Let the user know the result
         Edit_SetText(GetDlgItem(hwnd, IDC_RESULT),
            g_szSharedRequestAndResultBuffer);

         break;
   }
}

//////////////////////////////////////////////////////////////////////////////
```

```
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
    return(FALSE);
}


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


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

   // Create & initialize the 2 nonsignaled, auto-reset events
   g_hevtRequestSubmitted = CreateEvent(NULL, FALSE, FALSE, NULL);
   g_hevtResultReturned   = CreateEvent(NULL, FALSE, FALSE, NULL);

   // Spawn the server thread
   DWORD dwThreadID;
   HANDLE hThreadServer = chBEGINTHREADEX(NULL, 0, ServerThread, NULL,
      0, &dwThreadID);

   // Execute the client thread's user-interface
   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_HANDSHAKE), NULL, Dlg_Proc);

   // The client's UI is closing, have the server thread shutdown
   lstrcpy(g_szSharedRequestAndResultBuffer, g_szServerShutdown);
   SetEvent(g_hevtRequestSubmitted);

   // Wait for the server thread to acknowledge the shutdown AND
   // wait for the server thread to fully terminate
   HANDLE h[2];
   h[0] = g_hevtResultReturned;
   h[1] = hThreadServer;
   WaitForMultipleObjects(2, h, TRUE, INFINITE);

   // Properly clean up everything
   CloseHandle(hThreadServer);
   CloseHandle(g_hevtRequestSubmitted);
   CloseHandle(g_hevtResultReturned);

   // The client thread terminates with the whole process
   return(0);
}
//////////////////////////////// End of File //////////////////////////////////
```

Handshake.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

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_HANDSHAKE DIALOG DISCARDABLE  0, 0, 256, 81
STYLE DS_CENTER | WS_MINIMIZEBOX | WS_CAPTION | WS_SYSMENU
CAPTION "Handshake"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX        "Client side",IDC_STATIC,4,4,248,72
    LTEXT           "&Request:",IDC_STATIC,12,18,30,8
    EDITTEXT        IDC_REQUEST,48,16,196,14,ES_AUTOHSCROLL
    DEFPUSHBUTTON   "&Submit Request to Server",IDC_SUBMIT,80,36,96,14
    LTEXT           "Result:",IDC_STATIC,12,58,23,8
    EDITTEXT        IDC_RESULT,48,56,196,16,ES_AUTOHSCROLL | ES_READONLY
END
/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_HANDSHAKE, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 249
        TOPMARGIN, 7
        BOTTOMMARGIN, 74
    END
END
#endif    // APSTUDIO_INVOKED


#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
////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_HANDSHAKE           ICON    DISCARDABLE     "Handshake.ico"
#endif    // English (U.S.) resources
////////////////////////////////////////////////////////////////////////////

#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

9.4 等待定时器内核对象

等待定时器是在某个时间或按规定的间隔时间发出自己的信号通知的内核对象。它们通常用来在某个时间执行某个操作。

若要创建等待定时器，只需要调用CreateWaitableTimer函数：

```
HANDLE CreateWaitableTimer(
   PSECURITY_ATTRIBUTES psa,
   BOOL fManualReset,
   PCTSTR pszName);
```

psa和pszName这两个参数在第3章中做过介绍。当然，进程可以获得它自己的与进程相关的现有等待定时器的句柄，方法是调用OpenWaitableTimer函数：

```
HANDLE OpenWaitableTimer(
   DWORD dwDesiredAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

与事件的情况一样，fManualReset参数用于指明人工重置的定时器或自动重置的定时器。当发出人工重置的定时器信号通知时，等待该定时器的所有线程均变为可调度线程。当发出自动重置的定时器信号通知时，只有一个等待的线程变为可调度线程。

等待定时器对象总是在未通知状态中创建。必须调用 SetWaitableTimer函数来告诉定时器你想在何时让它成为已通知状态：

```
BOOL SetWaitableTimer(
   HANDLE hTimer,
   const LARGE_INTEGER *pDueTime,
   LONG lPeriod,
   PTIMERAPCROUTINE pfnCompletionRoutine,
   PVOID pvArgToCompletionRoutine,
   BOOL fResume);
```

这个函数带有若干个参数，使用时很容易搞混。显然， hTimer参数用于指明你要设置的定时器。pDueTime和lPeriod两个参数是一道使用的。PDueTimer参数用于指明定时器何时应该第一次报时，而lPeriod参数则用于指明此后定时器应该间隔多长时间报时一次。下面的代码用于将定时器的第一次报时的时间设置在2002年1月1日的下午1点钟，然后每隔6小时报时一次：

```
// Declare our local variables.
HANDLE hTimer;
SYSTEMTIME st;
FILETIME ftLocal, ftUTC;
LARGE_INTEGER liUTC;

// Create an auto-reset timer.
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);

// First signaling is at January 1, 2002, at 1:00 P.M. (local time).
st.wYear         = 2002; // Year
st.wMonth        = 1;    // January
st.wDayOfWeek    = 0;    // Ignored
st.wDay          = 1;    // The first of the month
st.wHour         = 13;   // 1PM
st.wMinute       = 0;    // 0 minutes into the hour
st.wSecond       = 0;    // 0 seconds into the minute
st.wMilliseconds = 0;    // 0 milliseconds into the second

SystemTimeToFileTime(&st, &ftLocal);
// Convert local time to UTC time.
LocalFileTimeToFileTime(&ftLocal, &ftUTC);
// Convert FILETIME to LARGE_INTEGER because of different alignment.
liUTC.LowPart  = ftUTC.dwLowDateTime;
liUTC.HighPart = ftUTC.dwHighDateTime;

// Set the timer.
SetWaitableTimer(hTimer, &liUTC, 6 * 60 * 60 * 1000,
   NULL, NULL, FALSE);
.
.
.
```

代码首先对SYSTEMTIME结构进行初始化，该结构用于指明定时器何时第一次报时（发出信号通知）。我将该时间设置为本地时间，即计算机所在时区的正确时间。 SetWaitableTimer的

第二个参数的原型是个常量LARGE_INTEGER *，因此它不能直接接受SYSTEMTIME结构。但是，FILETIME结构和LARGE_INTEGER结构拥有相同的二进制格式，都包含两个 32位的值。因此，我们可以将SYSTEMTIME结构转换成FILETIME结构。再一个问题是，SetWaitableTimer希望传递给它的时间始终都采用世界协调时（UTC）的时间。调用LocalFileTimeToFileTime函数，就可以很容易地进行时间的转换。

由于FILETIME和LARGE_INTEGER结构具有相同的二进制格式，因此可以像下面这样将FILETIME结构的地址直接传递给SetWaitableTimer：

```
// Set the timer.
SetWaitableTimer(hTimer, (PLARGE_INTEGER) &ftUTC,
   6 * 60 * 60 * 1000, NULL, NULL, FALSE);
```

实际上，这是我最初的做法。但是这是个大错误。虽然 FILETIME和LARGE_INTEGER结构采用相同的二进制格式，但是这两个结构的调整要求不同。所有 FILETIME结构的地址必须从一个32位的边界开始，而所有 LARGE_INTEGER结构的地址则必须从 64位的边界开始。调用SetWaitableTimer函数和给它传递一个 FILETIME结构时是否能够正确地运行，取决于FILETIME结构是否恰好位于64位的边界上。但是，编译器能够确保LARGE_INTEGER结构总是从64位的边界开始，因此要进行的正确操作（也就是所有时间都能保证起作用的操作）是将FILETIME的成员拷贝到LARGE_INTEGER的成员中，然后将LARGE_INTEGER的地址传递给SetWaitableTimer。

注意 x86处理器能够悄悄地处理未对齐的数据引用。因此当应用程序在 x86 CPU上运行时，将FILETIME的地址传递给SetWaitablrTimer总是可行的。但是，其他处理器，如Alpha处理器，则无法像x86处理器那样悄悄地处理未对齐的数据引用。实际上，大多数其他处理器都会产生一个 EXCEPTION_DATATYPE_MISALIGNMENT异常，它会导致进程终止运行。当你将 x86计算机上运行的代码移植到其他处理器时，产生问题的最大原因是出现了对齐错误。如果现在注意对齐方面的问题，就能够在以后省去几个月的代码移植工作。关于对齐问题的详细说明，参见第 13章。

现在，若要使定时器在 2002年1月1日下午1点之后每隔6h进行一次报时，我们应该将注意力转向lPeriod参数。该参数用于指明定时器在初次报时后每隔多长时间（以毫秒为单位）进行一次报时。如果是每隔 6h进行一次报时，那么我传递 21 600 000（6h×每小时60min×每分钟60s×每秒1000ms）。另外，如果给它传递了以前的一个绝对时间，比如 1975年1月1日下午1点，那么SetWaitableTimer的运行就不会失败。

如果不设置定时器应该第一次报时的绝对时间，也可以让定时器在一个相对于调用SetWaitableTimer的时间进行报时。只需要在 pDueTime参数中传递一个负值。传递的值必须是以100ns为间隔。由于我们通常并不以 100ns的间隔来思考问题，因此我们要说明一下 100ns的具体概念：1s=1000ms=1000000μs=1000000000ns 。

下面的代码用于将定时器设置为在调用 SetWaitableTimer函数后5s第一次报时：

```
// Declare our local variables.
HANDLE hTimer;
LARGE_INTEGER li;

// Create an auto-reset timer.
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);

// Set the timer to go off 5 seconds after calling SetWaitableTimer.
```

```
// Timer unit is 100-nanoseconds.
const int nTimerUnitsPerSecond = 10000000;

// Negate the time so that SetWaitableTimer knows we
// want relative time instead of absolute time.
li.QuadPart = -(5 * nTimerUnitsPerSecond);
// Set the timer.
SetWaitableTimer(hTimer, &li, 6 * 60 * 60 * 1000,
   NULL, NULL, FALSE);

:
```

通常情况下，你可能想要一个一次报时的定时器，它只是发出一次报时信号，此后再也不发出报时信号。若要做到这一点，只需要为lPeriod参数传递0即可。然后可以调用CloseHandle函数，关闭定时器，或者再次调用SetWaitableTimer函数，重新设置时间，为它规定一个需要遵循的新条件。

SetWaitableTimer的最后一个参数是fResume，它可以用于支持暂停和恢复的计算机。通常可以为该参数传递FALSE，就像我在上面这个代码段中设置的那样。但是，如果你编写了一个会议安排类型的应用程序，在这个应用程序在中，你想设置一个为用户提醒会议时间安排的定时器，那么应该传递TRUE。当定时器报时的时候，它将使计算机摆脱暂停方式（如果它处于暂停状态的话），并唤醒等待定时器报时的线程。然后该应用程序运行一个波形文件，并显示一个消息框，告诉用户即将举行的会议。如果为fResume参数传递FALSE，定时器对象就变为已通知状态，但是它唤醒的线程必须等到计算机恢复运行（通常由用户将它唤醒）之后才能得到CPU时间。

除了上面介绍的定时器函数外，最后还有一个CancelWaitableTimer函数：

```
BOOL CancelWaitableTimer(HANDLE hTimer);
```

这个简单的函数用于取出定时器的句柄并将它撤消，这样，除非接着调用SetWaitableTimer函数以便重新设置定时器，否则定时器决不会进行报时。如果想要改变定时器的报时条件，不必在调用SetWaitableTimer函数之前调用CancelWaitableTimer函数。每次调用SetWaitableTimer函数，都会在设置新的报时条件之前撤消定时器原来的报时条件。

9.4.1 让等待定时器给APC项排队

到现在为止，你已经学会了如何创建定时器和如何设置定时器。你还知道如何通过将定时器的句柄传递给WaitForSingleObject或WaitForMultipleObjects函数，以便等待定时器报时。Microsoft还允许定时器给在定时器得到通知信号时调用SetWaitableTimer函数的线程的异步过程调用（APC）进行排队。

一般来说，当调用SetWaitableTimer函数时，你将同时为pfnCompletionRoutine和pvArgCompletionRoutine参数传递NULL。当SetWaitableTime函数看到这些参数的NULL时，它就知道，当规定的时间到来时，就向定时器发出通知信号。但是，如果到了规定的时间，你愿意让定时器给一个APC排队，那么你必须传递定时器APC例程的地址，而这个例程是你必须实现的。该函数应该类似下面的形式：

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
   DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

   // Do whatever you want here.
}
```

我已经将该函数命名为TimerAPCRoutine，不过可以根据需要给它赋予任何一个名字。该函数可以在定时器报时的时候由调用SetWaitableTimer函数的同一个线程来调用，但是只有在调用线程处于待命状态下才能调用。换句话说，该线程必须正在 SleepEx,WaitForSngleObjectEx，WaitForMultipleObjectsEx，MsgWaitForMultipleObjectsEx或SingleObject-AndWait等函数的调用中等待。如果该线程不在这些函数中的某个函数中等待，系统将不给定时器 APC例程排队。这可以防止线程的APC队列中塞满定时器APC通知，这会浪费系统中的大量内存。

当定时器报时的时候，如果你的线程处于待命的等待状态中，系统就使你的线程调用回调例程。回调例程的第一个参数的值与你传递给SetWaitableTimer函数的pvArgToCompletionRoutine参数的值是相同的。你可以将某些上下文信息（通常是你定义的某个结构的指针）传递给TimerAPCRoutine。剩余的两个参数dwTimerLowValue和dwTimerHighValue用于指明定时器何时报时。下面的代码使用了该信息，并将它显示给用户：

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
   DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

   FILETIME ftUTC, ftLocal;
   SYSTEMTIME st;
   TCHAR szBuf[256];

   // Put the time in a FILETIME structure.
   ftUTC.dwLowDateTime = dwTimerLowValue;
   ftUTC.dwHighDateTime = dwTimerHighValue;

   // Convert the UTC time to the user's local time.
   FileTimeToLocalFileTime(&ftUTC, &ftLocal);

   // Convert the FILETIME to the SYSTEMTIME structure
   // required by GetDateFormat and GetTimeFormat.
   FileTimeToSystemTime(&ftLocal, &st);

   // Construct a string with the
   // date/time that the timer went off.
   GetDateFormat(LOCALE_USER_DEFAULT, DATE_LONGDATE,
      &st, NULL, szBuf, sizeof(szBuf) / sizeof(TCHAR));
   _tcscat(szBuf, __TEXT(" "));
   GetTimeFormat(LOCALE_USER_DEFAULT, 0,
      &st, NULL, _tcschr(szBuf, 0),
      sizeof(szBuf) / sizeof(TCHAR) - _tcslen(szBuf));

   // Show the time to the user.
   MessageBox(NULL, szBuf, "Timer went off at...", MB_OK);
}
```

只有当所有的APC项都已经处理之后，待命的函数才会返回。因此，必须确保定时器再次变为已通知状态之前，TimerAPCRoutine函数完成它的运行，这样，APC项的排队速度就不会比它被处理的速度快。

下面的代码显示了使用定时器和APC项的正确方法：

```
void SomeFunc() {
   // Create a timer. (It doesn't matter whether it's manual-reset
   // or auto-reset.)
   HANDLE hTimer = CreateWaitableTimer(NULL, TRUE, NULL);

   // Set timer to go off in 5 seconds.
```

```
    LARGE_INTEGER li = { 0 };
    SetWaitableTimer(hTimer, &li, 5000, TimerAPCRoutine, NULL, FALSE

    // Wait in an alertable state for the timer to go off.
    SleepEx(INFINITE, TRUE);

    CloseHandle(hTimer);
}
```

最后要说明的是，线程不应该等待定时器的句柄，也不应该以待命的方式等待定时器。请看下面的代码：

```
HANDLE hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
SetWaitableTimer(hTimer, ..., TimerAPCRoutine,...);
WaitForSingleObjectEx(hTimer, INFINITE, TRUE);
```

不应该编写上面的代码，因为调用WaitForSingleObjectEx函数实际上是两次等待该定时器，一次是以待命方式等待，一次是等待内核对象句柄。当定时器变为已通知状态时，等待就成功了，线程被唤醒，这将使该线程摆脱待命状态，而APC例程则没有被调用。前面讲过，通常没有理由使用带有等待定时器的APC例程，因为你始终都可以等待定时器变为已通知状态，然后做你想要做的事情。

9.4.2 定时器的松散特性

定时器常常用于通信协议中。例如，如果客户机向服务器发出一个请求，而服务器没有在规定的时间内作出响应，那么客户机就会认为无法使用服务器。目前，客户机通常要同时与许多服务器进行通信。如果你为每个请求创建一个定时器内核对象，那么系统的运行性能就会受到影响。可以设想，对于大多数应用程序来说，可以创建单个定时器对象，并根据需要修改定时器报时的时间。

定时器报时时间的管理方法和定时器时间的重新设定是非常麻烦的，只有很少的应用程序采用这种方法。但是在新的线程共享函数（第 11 章中介绍）中有一个新函数，称为CreateTimerQueueTimer，它能够为你处理所有的操作。如果你发现自己创建和管理了若干个定时器对象，那么应该观察一下这个函数，以减少应用程序的开销。

虽然定时器能够给APC项进行排队是很好的，但是目前编写的大多数应用程序并不使用APC，它们使用I/O完成端口机制。过去，我自己的线程池中（由一个 I/O完成端口负责管理）有一个线程，它按照特定的定时器间隔醒来。但是，等待定时器没有提供这个方法。为了做到这一点，我创建了一个线程，它的唯一工作是设置而后等待一个等待定时器。当定时器变为已通知状态时，线程就调用 PostQueuedCompletionStatus函数，将一个事件强加给线程池中的一个线程。

最后要说明的是，凡是称职的 Windows编程员都会立即将等待定时器与用户定时器（用SetTimer函数进行设置）进行比较。它们之间的最大差别是，用户定时器需要在应用程序中设置许多附加的用户界面结构，这使定时器变得资源更加密集。另外，等待定时器属于内核对象，这意味着它们可以供多个线程共享，并且是安全的。

用户定时器能够生成WM_TIMER消息，这些消息将返回给调用 SetTimer（用于回调定时器）的线程和创建窗口（用于基于窗口的定时器）的线程。因此，当用户定时器报时的时候，只有一个线程得到通知。另一方面，多个线程可以在等待定时器上进行等待，如果定时器是个人工重置的定时器，则可以调度若干个线程。

如果要执行与用户界面相关的事件，以便对定时器作出响应，那么使用用户定时器来组织代码结构可能更加容易些，因为使用等待定时器时，线程必须既要等待各种消息，又要等待内核对象（如果要改变代码的结构，可以使用 MsgWaitForMultipleObjects函数）。最后，运用等待定时器，当到了规定时间的时候，更有可能得到通知。正如第 27章介绍的那样，WM_TIMER消息始终属于最低优先级的消息，当线程的队列中没有其他消息时，才检索该消息。等待定时器的处理方法与其他内核对象没有什么差别，如果定时器发出报时信息，而你的线程正在等待之中，那么你的线程就会醒来。

9.5 信标内核对象

信标内核对象用于对资源进行计数。它们与所有内核对象一样，包含一个使用数量，但是它们也包含另外两个带符号的32位值，一个是最大资源数量，一个是当前资源数量。最大资源数量用于标识信标能够控制的资源的最大数量，而当前资源数量则用于标识当前可以使用的资源的数量。

为了正确地说明这个问题，让我们来看一看应用程序是如何使用信标的。比如说，我正在开发一个服务器进程，在这个进程中，我已经分配了一个能够用来存放客户机请求的缓冲区。我对缓冲区的大小进行了硬编码，这样它每次最多能够存放 5个客户机请求。如果5个请求尚未处理完毕时，一个新客户机试图与服务器进行联系，那么这个新客户机的请求就会被拒绝，并出现一个错误，指明服务器现在很忙，客户机应该过些时候重新进行联系。当我的服务器进程初始化时，它创建一个线程池，里面包含 5个线程，每个线程都准备在客户机请求到来时对它进行处理。

开始时，没有客户机提出任何请求，因此我的服务器不允许线程池中的任何线程成为可调度线程。但是，如果3个客户机请求同时到来，那么线程池中应该有 3个线程处于可调度状态。使用信标，就能够很好地处理对资源的监控和对线程的调度，最大资源数量设置为 5，因为这是我进行硬编码的缓冲区的大小。当前资源数量最初设置为 0，因为没有客户机提出任何请求。当客户机的请求被接受时，当前资源数量就递增，当客户机的请求被提交给服务器的线程池时，当前资源数量就递减。

信标的使用规则如下：

- 如果当前资源的数量大于0，则发出信标信号。
- 如果当前资源数量是0，则不发出信标信号。
- 系统决不允许当前资源的数量为负值。
- 当前资源数量决不能大于最大资源数量。

当使用信标时，不要将信标对象的使用数量与它的当前资源数量混为一谈。

下面的函数用于创建信标内核对象：

```
HANDLE CreateSemaphore(
   PSECURITY_ATTRIBUTE psa,
   LONG lInitialCount,
   LONG lMaximumCount,
   PCTSTR pszName);
```

psa和pszName两个参数在第3章中作过介绍。当然，通过调用 OpenSemaphore函数，另一个进程可以获得它自己的进程与现有信标相关的句柄：

```
HANDLE OpenSemaphore(
   DWORD fdwAccess,
```

```
BOOL bInheritHandle,
PCTSTR pszName);
```

lMaximumCount参数用于告诉系统，应用程序处理的最大资源数量是多少。由于这是个带符号的32位值，因此最多可以拥有2 147 483 647个资源。lInitialCount参数用于指明开始时（当前）这些资源中有多少可供使用。当我的服务器进程初始化时，没有任何客户机请求，因此我调用下面这个CreateSemaphore函数：

```
HANDLE hsem = CreateSemaphore(NULL, 0, 5, NULL);
```

该函数创建了一个信标，其最大资源数量为5，但是开始时可以使用的资源为0（由于偶然的原因，该内核对象的使用数量是1，因为我刚刚创建了这个内核对象，请不要与计数器搞混）。由于当前资源数量被初始化为0，因此不发出信标信号。等待信标的所有线程均进入等待状态。

通过调用等待函数，传递负责保护资源的信标的句柄，线程就能够获得对该资源的访问权。从内部来说，该等待函数要检查信标的当前资源数量，如果它的值大于0（信标已经发出信号），那么计数器递减1，调用线程保持可调度状态。信标的出色之处在于它们能够以原子操作方式来执行测试和设置操作，这就是说，当向信标申请一个资源时，操作系统就要检查是否有这个资源可供使用，同时将可用资源的数量递减，而不让另一个线程加以干扰。只有当资源数量递减后，系统才允许另一个线程申请对资源的访问权。

如果该等待函数确定信标的当前资源数量是0（信标没有发出通知信号），那么系统就调用函数进入等待状态。当另一个线程将对信标的当前资源数量进行递增时，系统会记住该等待线程（或多个线程），并允许它变为可调度状态（相应地递减它的当前资源数量）。

通过调用ReleaseSemaphore函数，线程就能够对信标的当前资源数量进行递增：

```
BOOL ReleaseSemaphore(
   HANDLE hsem,
   LONG lReleaseCount,
   PLONG plPreviousCount);
```

该函数只是将lReleaseCount中的值添加给信标的当前资源数量。通常情况下，为lReleaseCount参数传递1，但是，不一定非要传递这个值。我常常传递2或更大的值。该函数也能够在它的*plPreviousCount中返回当前资源数量的原始值。实际上几乎没有应用程序关心这个值，因此可以传递NULL，将它忽略。

有时，有必要知道信标的当前资源数量而不修改这个数量，但是没有一个函数可以用来查询信标的当前资源数量的值。起先我认为调用ReleaseSemaphore并为lReleaseCount参数传递0，也许会在*plPreviousCount中返回资源的实际数量。但是这样做是不行的，ReleaseSemaphore用0填入这个长变量。接着，我试图传递一个非常大的数字，作为第二个参数，希望它不会影响当前资源数量，因为它将取代最大值。同样，ReleaseSemaphore用0填入*plPrevious。可惜，如果不对它进行修改，就没有办法得到信标的当前资源数量。

9.6 互斥对象内核对象

互斥对象（mutex）内核对象能够确保线程拥有对单个资源的互斥访问权。实际上互斥对象是因此而得名的。互斥对象包含一个使用数量，一个线程ID和一个递归计数器。互斥对象的行为特性与关键代码段相同，但是互斥对象属于内核对象，而关键代码段则属于用户方式对象。这意味着互斥对象的运行速度比关键代码段要慢。但是这也意味着不同进程中的多个线程能够访问单个互斥对象，并且这意味着线程在等待访问资源时可以设定一个超时值。

ID用于标识系统中的哪个线程当前拥有互斥对象，递归计数器用于指明该线程拥有互斥对

象的次数。互斥对象有许多用途，属于最常用的内核对象之一。通常来说，它们用于保护由多个线程访问的内存块。如果多个线程要同时访问内存块，内存块中的数据就可能遭到破坏。互斥对象能够保证访问内存块的任何线程拥有对该内存块的独占访问权，这样就能够保证数据的完整性。

互斥对象的使用规则如下：

- 如果线程ID是0（这是个无效ID），互斥对象不被任何线程所拥有，并且发出该互斥对象的通知信号。
- 如果ID是个非0数字，那么一个线程就拥有互斥对象，并且不发出该互斥对象的通知信号。
- 与所有其他内核对象不同， 互斥对象在操作系统中拥有特殊的代码，允许它们违反正常的规则（后面将要介绍这个异常情况）。

若要使用互斥对象，必须有一个进程首先调用CreateMutex，以便创建互斥对象：

```
HANDLE CreateMutex(
   PSECURITY_ATTRIBUTES psa,
   BOOL fInitialOwner,
   PCTSTR pszName);
```

psa和pszName参数在第3章中做过介绍。当然，通过调用 OpenMutex，另一个进程可以获得它自己进程与现有互斥对象相关的句柄：

```
HANDLE OpenMutex(
   DWORD fdwAccess,
   BOOL bInheritHandle,
   PCTSTR pszName);
```

fInitialOwner参数用于控制互斥对象的初始状态。如果传递 FALSE（这是通常情况下传递的值），那么互斥对象的ID和递归计数器均被设置为0。这意味着该互斥对象没有被任何线程所拥有，因此要发出它的通知信号。

如果为fInitialOwner参数传递TRUE，那么该对象的线程ID被设置为调用线程的ID，递归计数器被设置为1。由于ID是个非0数字，因此该互斥对象开始时不发出通知信号。

通过调用一个等待函数，并传递负责保护资源的互斥对象的句柄，线程就能够获得对共享资源的访问权。在内部，等待函数要检查线程的 ID，以了解它是否是0（互斥对象发出通知信号）。如果线程ID是0，那么该线程ID被设置为调用线程的ID，递归计数器被设置为 1，同时，调用线程保持可调度状态。

如果等待函数发现ID不是0（不发出互斥对象的通知信号），那么调用线程便进入等待状态。系统将记住这个情况，并且在互斥对象的 ID重新设置为0时，将线程ID设置为等待线程的ID，将递归计数器设置为 1，并且允许等待线程再次成为可调度线程。与所有情况一样，对互斥内核对象进行的检查和修改都是以原子操作方式进行的。

对于互斥对象来说，正常的内核对象的已通知和未通知规则存在一个特殊的异常情况。比如说，一个线程试图等待一个未通知的互斥对象。在这种情况下，该线程通常被置于等待状态。然而，系统要查看试图获取互斥对象的线程的ID是否与互斥对象中记录的线程ID相同。如果两个线程ID相同，即使互斥对象处于未通知状态，系统也允许该线程保持可调度状态。我们不认为该“异常”行为特性适用于系统中的任何地方的其他内核对象。每当线程成功地等待互斥对象时，该对象的递归计数器就递增。若要使递归计数器的值大于 1，唯一的方法是线程多次等待相同的互斥对象，以便利用这个异常规则。

一旦线程成功地等待到一个互斥对象，该线程就知道它已经拥有对受保护资源的独占访问权。试图访问该资源的任何其他线程（通过等待相同的互斥对象）均被置于等待状态中。当目前拥有对资源的访问权的线程不再需要它的访问权时，它必须调用 ReleaseMutex函数来释放该互斥对象：

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

该函数将对象的递归计数器递减 1。如果线程多次成功地等待一个互斥对象，在互斥对象的递归计数器变成 0之前，该线程必须以同样的次数调用 ReleaseMutex函数。当递归计数器到达0时，该线程ID也被置为0，同时该对象变为已通知状态。

当该对象变为已通知状态时，系统要查看是否有任何线程正在等待互斥对象。如果有，系统将“按公平原则”选定等待线程中的一个，为它赋予互斥对象的所有权。当然，这意味着线程ID被设置为选定的线程的ID，并且递归计数器被置为 1。如果没有其他线程正在等待互斥对象，那么该互斥对象将保持已通知状态，这样，等待互斥对象的下一个线程就立即可以得到互斥对象。

9.6.1 释放问题

互斥对象不同于所有其他内核对象，因为互斥对象有一个“线程所有权”的概念。本章介绍的其他内核对象中，没有一种对象能够记住哪个线程成功地等待到该对象，只有互斥对象能够对此保持跟踪。互斥对象的线程所有权概念是互斥对象为什么会拥有特殊异常规则的原因，这个异常规则使得线程能够获取该互斥对象，尽管它没有发出通知。

这个异常规则不仅适用于试图获取互斥对象的线程，而且适用于试图释放互斥对象的线程。当一个线程调用ReleaseMutex函数时，该函数要查看调用线程的ID是否与互斥对象中的线程ID相匹配。如果两个ID相匹配，递归计数器就会像前面介绍的那样递减。如果两个线程的 ID不匹配，那么ReleaseMutex函数将不进行任何操作，而是将 FALSE（表示失败）返回给调用者。此时调用GetLastError，将返回ERROR_NOT_OWNER（试图释放不是调用者拥有的互斥对象）。

因此，如果在释放互斥对象之前，拥有互斥对象的线程终止运行（使用 ExitThread、TerminateThread、ExitProcess或TerminateProcess函数），那么互斥对象和正在等待互斥对象的其他线程将会发生什么情况呢？答案是，系统将把该互斥对象视为已经被放弃——拥有互斥对象的线程决不会释放它，因为该线程已经终止运行。

由于系统保持对所有互斥对象和线程内核对象的跟踪，因此它能准确的知道互斥对象何时被放弃。当一个互斥对象被放弃时，系统将自动把互斥对象的 ID复置为0，并将它的递归计数器复置为 0。然后，系统要查看目前是否有任何线程正在等待该互斥对象。如果有，系统将“公平地”选定一个等待线程，将 ID设置为选定的线程的ID，并将递归计数器设置为 1，同时，选定的线程变为可调度线程。

这与前面的情况相同，差别在于等待函数并不将通常的 WAIT_OBJECT_0值返回给线程。相反，等待函数返回的是特殊的 WAIT_ABANDONED值。这个特殊的返回值（它只适用于互斥对象）用于指明线程正在等待的互斥对象是由另一个线程拥有的，而这另一个线程已经在它完成对共享资源的使用前终止运行。这不是可以进入的最佳情况。新调度的线程不知道目前资源处于何种状态，也许该资源已经完全被破坏了。在这种情况下必须自己决定应用程序应该怎么办。

在实际运行环境中，大多数应用程序从不明确检查 WAIT_ABANDONED返回值，因为线程很少是刚刚终止运行（上面介绍的情况提供了另一个例子，说明为什么决不应该调用

TerminateThread函数）。

9.6.2 互斥对象与关键代码段的比较

就等待线程的调度而言，互斥对象与关键代码段之间有着相同的特性。但是它们在其他属性方面却各不相同。表9-1对它们进行了各方面的比较。

表9-1 互斥对象与关键代码段的比较

特 性	互斥对象	关键代码段
运行速度	慢	快
是否能够跨进程边界来使用	是	否
声明	HANDLE hmtx；	CRITICAL_SECTION cs；
初始化	hmtx=CreateMutex（NULL，FALSE，NULL）；	InitializeCriticalSection(&es)；
清除	CloseHandle（hmtx）；	DeleteCriticalSection（&cs）；
无限等待	WaitForSingleObject（hmtx,INFINITE）；	EnterCriticalSection（&cs）；
0等待	WaitForSingleObject（hmtx,0）；	TryEnterCriticalSection（&cs）；
任意等待	WaitForSingleObject（hmtx,dwMilliseconds）；	不能
释放	ReleaseMutex（hmtx）；	LeaveCriticalSection（&cs）；
是否能够等待其他内核对象	是（使用WaitForMultipleObjects或类似的函数）	否

9.6.3 Queue示例应用程序

后面的清单9-2列出的Queue（“09 Queue.exe）应用程序使用一个互斥对象和一个信标来控制数据元素的队列。该应用程序的源代码和资源文件位于本书所附光盘中的 09-Queue目录下。当运行Queue应用程序时，就会出现图9-4对话框。

当Queue初始化时，它创建4个客户机线程和两个服务器线程。每个客户机线程均会睡眠一定的时间周期，然后将一个请求元素附加给队列。当每个元素排队时， Client Threads（客户机线程）列表框就被更新。列表框的每一项表示哪个客户机线程附加了这个项，以及它是个什么项。例如，列表框中的第一项表示客户机线程 0附加了它的第一个请求。然后客户机线程 1至3附加它们的第一个请求，接着客户机线程0又附加它的第二个请求，如此类推。

服务器线程在队列中出现第一个元素之前一直无事可做。当第一个元素出现时，一个服务器线程醒来，对该请求进程处理。 Server Threads（服务器线程）列表框显示了服务器线程的状态。它的第一项显示服务器线程 0正在处理来自客户机线程 0的一个请求。正在处理的请求是客户机线程的第一个请求。第二项显示服务器线程 1正在处理客户机线程 1的第一个请求。

在这个例子中，服务器线程无法以足够快的速度来处理客户程序的请求，队列中的请求达到了最大的容量。我对队列数据结构进行了初始化，使它一次能够存放的元素不能超过 10个，这将导致队列很快被填满。另外，客户机线程有 4个，而服务器线程只有两个。我们看到，当客户机线程3试图将它的第5个请求附加给队列时，队列已经填满了。

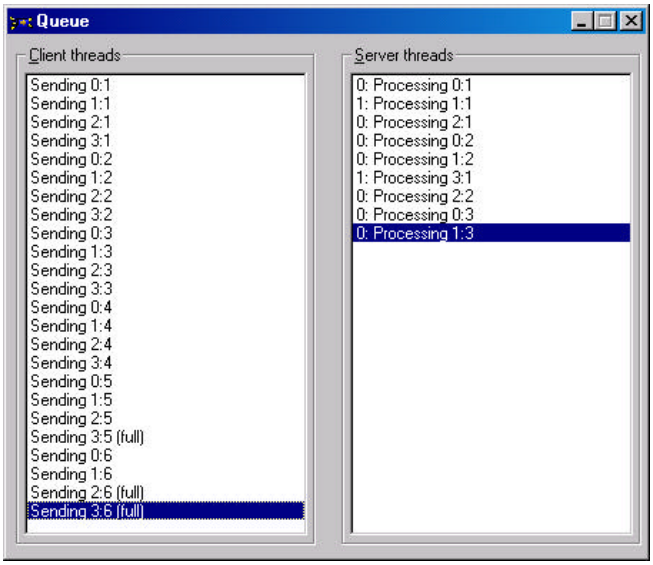

图9-4 Queue 对话框

好了，这就是你看到的情况，更有意思的是它的运行情况。这个队列是由一个 C++类CQueue进行管理和控制的：

```
class CQueue {
public:
   struct ELEMENT {
      int m_nThreadNum, m_nRequestNum;
      // Other element data should go here.
   };
   typedef ELEMENT* PELEMENT;

private:
   PELEMENT m_pElements;        // Array of elements to be processed
   int      m_nMaxElements;     // # of elements in the array
   HANDLE   m_h[2];             // Mutex & semaphore handles
   HANDLE   &m_hmtxQ;           // Reference to m_h[0]
   HANDLE   &m_hsemNumElements; // Reference to m_h[1]

public:
   CQueue(int nMaxElements);
   ~CQueue();

   BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
   BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};
```

这个类中的公有ELEMENT结构用于定义队列数据元素是个什么样子。它的实际内容并不重要。对于这个示例应用程序，客户程序将它们的线程号和它们的请求号放入这个元素，这样，当服务器处理检索出来的元素时，就能在服务器的列表框中显示这些信息。实际运行的应用程序通常不需要这些信息。

至于私有成员，我们有m_pElements，它指向一个固定大小的ELEMENT结构的数组。这是需要保护使之不受多个客户机/服务器线程影响的数据。M_nMaxElements成员用于表示创建

Cqueue对象时该数组被初始化为多大的规模。下一个成员 m_h是由两个内核对象句柄组成的数组。为了正确地保护队列的数据元素，需要两个内核对象，一个是互斥对象，另一个是信标对象。在CQueue构造函数中，这两个对象被创建，它们的句柄放入该数组。

下面很快就会看到，该代码有时调用 WaitForMultipleObjects函数，传递句柄数组的地址。也会看到，有时该代码只需要引用这些内核对象句柄中的一个句柄。为了使代码更容易阅读和维护，我还声明了两个句柄参考成员，即 m_hmtxQ和m_hsemNumElements。当CQueue构造函数运行时，它就将这些句柄参考成员分别初始化为 m_h[0]和m_h[1]。

现在应该能够毫无困难地理解 CQueue的构造函数与析构函数的方法了，因此让我们将注意力转向Append方法。这个方法试图将一个 ELEMENT附加给队列。但是，线程首先必须确保它拥有对该队列的独占访问权。 Append方法通过调用WaitForSingleObject函数，传递m_hmtxQ互斥对象的句柄，实现其独占访问权。如果返回 WAIT_OBJECT_0,那么该线程就拥有对该队列的独占访问权。

接着，Append方法必须调用ReleaseSemaphore函数，传递释放数量 1，以便使队列中的元素数量递增。如果ReleaseSemaphore函数调用成功，队列中的元素没有放满，那么新元素就可以附加给队列。幸好ReleaseSemaphore函数返回了lPreviousCount变量中的队列元素的前一个数量。这确切地告诉你新元素应该放入哪个数组索引中。当该元素被拷贝到队列的数组中后，该函数就返回。一旦该元素完全附加给队列， Append便调用Release互斥对象，这样其他线程就能够访问该队列。Append函数的剩余部分与故障情况和错误处理有关。

现在让我们来看一看服务器线程如何调用 Remove方法，以便从队列中取出元素的。首先，线程必须确保它拥有对队列的独占访问权，同时队列中至少必须拥有一个元素。当然，如果队列中没有任何元素，那么服务器线程就没有理由被唤醒。因此， Remove方法首先要调用WairForMultipleObjects，并且同时传递互斥对象和信标的句柄。只有当这两个对象都得到通知时，服务器线程才被唤醒。

如果返回WAIT_OBJECT_0，该线程将拥有对队列的独占访问权，并且队列中至少必须有一个元素。这时，代码就取出数组中索引号为 0的元素，然后将数组中的剩余元素下移一位。这不是使用队列的最有效的方法，因为用这种方法进行内存的拷贝要占用大量的资源，但是我们在这里是想展示线程同步的情况，所以就使用了这种方法。最后，要调用 ReleaseMutex函数，这样其他线程就能安全地访问该队列。

注意，信标对象能够随时跟踪某个时间队列中存在多少个元素。你能够看到这个数字在递增。当一个新元素被附加给队列时， Append方法就调用ReleaseSemaphore。但是，当一个元素从队列中删除时，你无法立即看到这个数字递减。递减是由Remove方法调用WaitForMultipleObjects函数来进行的。记住，成功地等待信标的副作用是它的数量递减 1。这对我们来说是很方便的。

现在已经懂得CQueue类是如何运行的，源代码的其余部分很容易理解。

清单9-2 CQueue示例应用程序

Queue.cpp

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

```
#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>       // For _beginthreadex
#include "Resource.h"

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

class CQueue {
public:
   struct ELEMENT {
      int m_nThreadNum, m_nRequestNum;
      // Other element data should go here
   };
   typedef ELEMENT* PELEMENT;

private:
   PELEMENT m_pElements;         // Array of elements to be processed
   int      m_nMaxElements;      // Maximum # of elements in the array
   HANDLE   m_h[2];              // Mutex & semaphore handles
   HANDLE   &m_hmtxQ;            // Reference to m_h[0]
   HANDLE   &m_hsemNumElements; // Reference to m_h[1]

public:
   CQueue(int nMaxElements);
   ~CQueue();

   BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
   BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};


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


CQueue::CQueue(int nMaxElements)
   : m_hmtxQ(m_h[0]), m_hsemNumElements(m_h[1]) {

   m_pElements = (PELEMENT)
      HeapAlloc(GetProcessHeap(), 0, sizeof(ELEMENT) * nMaxElements);
   m_nMaxElements = nMaxElements;
   m_hmtxQ = CreateMutex(NULL, FALSE, NULL);
   m_hsemNumElements = CreateSemaphore(NULL, 0, nMaxElements, NULL);
}


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


CQueue::~CQueue() {

   CloseHandle(m_hsemNumElements);
   CloseHandle(m_hmtxQ);
   HeapFree(GetProcessHeap(), 0, m_pElements);
}


//////////////////////////////////////////////////////////////////////////////
```

```
BOOL CQueue::Append(PELEMENT pElement, DWORD dwTimeout) {

   BOOL fOk = FALSE;
   DWORD dw = WaitForSingleObject(m_hmtxQ, dwTimeout);

   if (dw == WAIT_OBJECT_0) {
      // This thread has exclusive access to the queue

      // Increment the number of elements in the queue
      LONG lPrevCount;
      fOk = ReleaseSemaphore(m_hsemNumElements, 1, &lPrevCount);
      if (fOk) {
         // The queue is not full; append the new element
         m_pElements[lPrevCount] = *pElement;
      } else {

         // The queue is full; set the error code and return failure
         SetLastError(ERROR_DATABASE_FULL);
      }

      // Allow other threads to access the queue
      ReleaseMutex(m_hmtxQ);

   } else {
      // Timeout, set error code and return failure
      SetLastError(ERROR_TIMEOUT);
   }

   return(fOk);   // Call GetLastError for more info
}

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

BOOL CQueue::Remove(PELEMENT pElement, DWORD dwTimeout) {

   // Wait for exclusive access to queue and for queue to have element.
   BOOL fOk = (WaitForMultipleObjects(chDIMOF(m_h), m_h, TRUE, dwTimeout)
      == WAIT_OBJECT_0);

   if (fOk) {
      // The queue has an element; pull it from the queue
      *pElement = m_pElements[0];

      // Shift the remaining elements down
      MoveMemory(&m_pElements[0], &m_pElements[1],
      sizeof(ELEMENT) * (m_nMaxElements - 1));

   // Allow other threads to access the queue
   ReleaseMutex(m_hmtxQ);

} else {
   // Timeout, set error code and return failure
   SetLastError(ERROR_TIMEOUT);
}

return(fOk);   // Call GetLastError for more info
```

```
   }

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


   CQueue g_q(10);                        // The shared queue
   volatile BOOL g_fShutdown = FALSE;     // Signals client/server threads to die
   HWND g_hwnd;                           // How client/server threads give status


   // Handles to all client/server threads & number of client/server threads
   HANDLE g_hThreads[MAXIMUM_WAIT_OBJECTS];
   int    g_nNumThreads = 0;


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


   DWORD WINAPI ClientThread(PVOID pvParam) {

      int nThreadNum = PtrToUlong(pvParam);
      HWND hwndLB = GetDlgItem(g_hwnd, IDC_CLIENTS);

      for (int nRequestNum = 1; !g_fShutdown; nRequestNum++) {

         TCHAR sz[1024];
         CQueue::ELEMENT e = { nThreadNum, nRequestNum };

         // Try to put an element on the queue
         if (g_q.Append(&e, 200)) {

            // Indicate which thread sent it and which request
            wsprintf(sz, TEXT("Sending %d:%d"), nThreadNum, nRequestNum);
         } else {
            // Couldn't put an element on the queue
            wsprintf(sz, TEXT("Sending %d:%d (%s)"), nThreadNum, nRequestNum,
               (GetLastError() == ERROR_TIMEOUT)
                  ? TEXT("timeout") : TEXT("full"));
         }

         // Show result of appending element
         ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
         Sleep(2500);   // Wait before appending another element
      }

      return(0);
   }


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


   DWORD WINAPI ServerThread(PVOID pvParam) {

      int nThreadNum = PtrToUlong(pvParam);
      HWND hwndLB = GetDlgItem(g_hwnd, IDC_SERVERS);
```

```
while (!g_fShutdown) {

   TCHAR sz[1024];
   CQueue::ELEMENT e;

   // Try to get an element from the queue
   if (g_q.Remove(&e, 5000)) {

      // Indicate which thread is processing it, which thread
      // sent it, and which request we're processing
      wsprintf(sz, TEXT("%d: Processing %d:%d"),
         nThreadNum, e.m_nThreadNum, e.m_nRequestNum);

      // The server takes some time to process the request
      Sleep(2000 * e.m_nThreadNum);

   } else {
      // Couldn't get an element from the queue
      wsprintf(sz, TEXT("%d: (timeout)"), nThreadNum);
   }

   // Show result of processing element
   ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

   }

   return(0);
}

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

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_QUEUE);

   g_hwnd = hwnd; // Used by client/server threads to show status

   DWORD dwThreadID;

   // Create the client threads
   for (int x = 0; x < 4; x++)
      g_hThreads[g_nNumThreads++] =
         chBEGINTHREADEX(NULL, 0, ClientThread, (PVOID) (INT_PTR) x,
            0, &dwThreadID);

   // Create the server threads
   for (x = 0; x < 2; x++)
      g_hThreads[g_nNumThreads++] =
         chBEGINTHREADEX(NULL, 0, ServerThread, (PVOID) (INT_PTR) x,
            0, &dwThreadID);

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////
```

```
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hwnd, id);
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


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


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

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_QUEUE), NULL, Dlg_Proc);
   InterlockedExchangePointer((PVOID*) &g_fShutdown, (PVOID) TRUE);

   // Wait for all the threads to terminate & then cleanup
   WaitForMultipleObjects(g_nNumThreads, g_hThreads, TRUE, INFINITE);
   while (g_nNumThreads--)
      CloseHandle(g_hThreads[g_nNumThreads]);

   return(0);
}


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

Queue.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

////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_QUEUE DIALOG DISCARDABLE  38, 36, 298, 225
STYLE WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Queue"
FONT 8, "MS Sans Serif"
BEGIN
    GROUPBOX        "&Client threads",IDC_STATIC,4,4,140,216
    LISTBOX         IDC_CLIENTS,8,16,132,200,NOT LBS_NOTIFY |
                    LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_TABSTOP
    GROUPBOX        "&Server threads",IDC_STATIC,156,4,140,216
    LISTBOX         IDC_SERVERS,160,16,132,200,NOT LBS_NOTIFY |
                    LBS_NOINTEGRALHEIGHT | WS_VSCROLL | WS_TABSTOP
END

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

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


//////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_QUEUE, DIALOG
    BEGIN
        RIGHTMARGIN, 244
        BOTTOMMARGIN, 130
    END
END
#endif    // APSTUDIO_INVOKED

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



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

9.7 线程同步对象速查表

表9-2所示的速查表综合列出了各种内核对象与线程同步之间的相互关系。

表9-2 内核对象与线程同步之间的相互关系

对 象	何时处于未通知状态	何时处于已通知状态	成功等待的副作用
进程	当进程仍然活动时	当进程终止运行时（ExitProcess，TerminateProcess）	无
线程	当线程仍然活动时	当线程终止运行时（ExitThread，TerminateThread）	无
作业	当作业的时间尚未结束时	当作业的时间已经结束时	无
文件	当I/O请求正在处理时	当I/O请求处理完毕时	无
控制台输入	不存在任何输入	当存在输入时	无
文件修改通知	没有任何文件被修改	当文件系统发现修改时	重置通知
自动重置事件	ResetEvent,Pulse-Event或等待成功	当调用SetEvent/PulseEvent时	重置事件
人工重置事件	ResetEvent或PulseEvent	当调用SetEvent/PulseEvent时	无

(续)

对 象	何时处于未通知状态	何时处于已通知状态	成功等待的副作用
自动重置等待定时器	CancelWaitableTimer或等待成功	当时间到时(SetWaitableTimer)	重置定时器
人工重置等待定时器	CancelWaitableTimer	当时间到时(SetWaitableTimer)	无
信标	等待成功	当数量>0时(ReleaseSemaphore)	数量递减1
互斥对象	等待成功	当未被线程拥有时(Release互斥对象)	将所有权赋予线程
关键代码段(用户方式)	等待成功((Try) EnterCriticalSection)	当未被线程拥有时(LeaveCriticalSection)	将所有权赋予线程

互锁(用户方式)函数决不会导致线程变为非调度状态,它们会改变一个值并立即返回。

9.8 其他的线程同步函数

WaitForSingleObject和WaitForMultipleObjects是进行线程同步时使用得最多的函数。但是,Windows还提供了另外几个稍有不同的函数。如果理解了WaitForSingleObject和WaitForMultipleObjects函数,那么要理解其他函数如何运行,就不会遇到什么困难。本节简单地介绍一些这样的函数。

9.8.1 异步设备I/O

异步设备I/O使得线程能够启动一个读操作或写操作,但是不必等待读操作或写操作完成。例如,如果线程需要将一个大文件装入内存,那么该线程可以告诉系统将文件装入内存。然后,当系统加载该文件时,该线程可以忙于执行其他任务,如创建窗口、对内部数据结构进行初始化等等。当初始化操作完成时,该线程可以终止自己的运行,等待系统通知它文件已经读取。

设备对象是可以同步的内核对象,这意味着可以调用WaitForSingleObject函数,传递文件、套接字和通信端口的句柄。当系统执行异步I/O时,设备对象处于未通知状态。一旦操作完成,系统就将对象的状态改为已通知状态,这样,该线程就知道操作已经完成。此时,该线程就可以继续运行。

9.8.2 WaitForInputIdle

线程也可以调用WaitForInputIdle来终止自己的运行:

```
DWORD WaitForInputIdle(
  HANDLE hProcess,
  DWORD dwMilliseconds);
```

该函数将一直处于等待状态,直到hProcess标识的进程在创建应用程序的第一个窗口的线程中已经没有尚未处理的输入为止。这个函数可以用于父进程。父进程产生子进程,以便执行某些操作。当父进程的线程调用CreateProcess时,该父进程的线程将在子进程初始化时继续运行。父进程的线程可能需要获得子进程创建的窗口的句柄。如果父进程的线程想要知道子进程何时完成初始化,唯一的办法是等待,直到子进程不再处理任何输入为止。因此,当调用CreateProcess后,父进程的线程就调用WaitForInputIdle。

当需要将击键输入纳入应用程序时,也可以调用WaitForInputIdle。比如说,可以将下面的

消息显示在应用程序的主窗口：

WM_KEYDOWN	使用虚拟键VK_MENU
WM_KEYDOWN	使用虚拟键VK_F
WM_KEYUP	使用虚拟键VK_E
WM_KEYUP	使用虚拟键VK_MENU
WM_KEYDOWN	使用虚拟键VK_O
WM_KEYUP	使用虚拟键VK_O

这个序列将Alt+F,O发送给应用程序，对于大多数使用英语的应用程序来说，它从应用程序的文件菜单中选择 Open命令。该命令打开一个对话框，但是，在对话框出现以前，Windows必须加载来自文件的对话框摸板，遍历摸板中的所有控件，并为每个摸板调用 CreateWindow。这可能需要花费一定的时间。因此，显示 WM_KEY*消息的应用程序可以调用WaitForInputIdle，WaitForlnputIdle将导致应用程序处于等待状态，直到对话框创建完成并准备接受用户的输入。这时，该应用程序可以将其他的击键输入纳入对话框及其控件，使它能够继续执行它需要的操作。

编写16位Windows应用程序的编程人员常常要面对这个问题。应用程序想要将消息显示在窗口中，但是它并不确切知道窗口何时创建完成、作好接受消息的准备。 WaitForInputIdle函数解决了这个问题。

9.8.3 MsgWaitForMultipleObjects(Ex)

线程可以调用MsgWaitForMultipleObjects或MsgWaitForMultipleObjectsEx函数，让线程等待它自己的消息：

```
DWORD MsgWaitForMultipleObjects(
   DWORD dwCount,
   PHANDLE phObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds,
   DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
   DWORD dwCount,
   PHANDLE phObjects,
   DWORD dwMilliseconds,
   DWORD dwWakeMask,
   DWORD dwFlags);
```

这些函数与WaitForMultipleObjects函数十分相似。差别在于它们允许线程在内核对象变成已通知状态或窗口消息需要调度到调用线程创建的窗口中时被调度。

创建窗口和执行与用户界面相关的任务的线程，应该调用 MsgWaitForMultipleObjectsEx函数，而不应该调用WaitForMultipleObjects函数，因为后面这个函数将使线程的用户界面无法对用户作出响应。该函数将在第27章中详细介绍。

9.8.4 WaitForDebugEvent

Windows将非常出色的调试支持特性内置于操作系统之中。当调试程序启动运行时，它将自己附加给一个被调试程序。该调试程序只需闲置着，等待操作系统将与被调试程序相关的调试事件通知它。调试程序通过调用WaitForDebugEvent函数来等待这些事件的发生：

```
BOOL WaitForDebugEvent(
   PDEBUG_EVENT pde,
   DWORD dwMilliseconds);
```

当调试程序调用该函数时，调试程序的线程终止运行，系统将调试事件已经发生的情况通知调试程序，方法是允许调用的WaitForDebugEvent函数返回。pde参数指向的结构在唤醒调试程序的线程之前由系统填入信息。该结构包含了关于刚刚发生的调试事件的信息。

9.8.5 SingleObjectAndWait

SingleObjectAndWait函数用于在单个原子方式的操作中发出关于内核对象的通知并等待另一个内核对象：

```
DWORD SignalObjectAndWait(
   HANDLE hObjectToSignal,
   HANDLE hObjectToWaitOn,
   DWORD dwMilliseconds,
   BOOL fAlertable);
```

当调用该函数时，hObjectToSignal参数必须标识一个互斥对象、信标对象或事件。任何其他类型的对象将导致该函数返回WAIT_FAILED，并使GetLastError函数返回ERROR_INVALID_HANDLE。在内部，该函数将观察对象的类型，并分别运行ReleaseMutex、ReleaseSemaphore(其数量为1)或ResetEvent中的相应参数。

hObjectToWaitOn参数用于标识下列任何一个内核对象：互斥对象、信标、事件、定时器、进程、线程、作业、控制台输入和修改通知。与平常一样， dwMilliseconds参数指明该函数为了等待该对象变为已通知状态，应该等待多长时间，而fAlertable标志则指明线程等待时该线程是否应该能够处理任何已经排队的异步过程调用。

该函数返回下列几个值中的一个：WAIT_OBJECT_0、WAIT_TIMEOUT、WAIT_FAILED、WAIT_ABANDONED（本章前面已经介绍）或WAIT_IO_COMPLETION。

该函数是对Windows的令人欢迎的一个补充，原因有二。首先，因为常常需要通知一个对象，等待另一个对象，用一个函数来执行两个操作可以节省许多处理时间。每次调用一个函数，使线程从用户方式代码变成内核方式代码，大约需要运行1000个CPU周期。例如，运行下面的代码至少需要2000个CPU周期：

```
ReleaseMutex(hMutex);
WaitForSingleObject(hEvent, INFINITE);
```

在高性能服务器应用程序中，SignalObjectAndWait函数能够节省大量的处理时间。

第二，如果没有SignalObjectAndWait函数，一个线程将无法知道另一个线程何时处于等待状态。对于PluseEvent之类的函数来说，知道这个情况是很有用的。本章前面讲过，PulseEvent函数能够通知一个事件，并且立即对它进行重置。如果目前没有任何线程等待该事件，那么就没有事件会抓住这个情况。曾经有人编写过类似下面的代码：

```
// Perform some work.
.
.
.
SetEvent(hEventWorkerThreadDone);
WaitForSingleObject(hEventMoreWorkToBeDone, INFINITE);
// Do more work.
.
.
.
```

一个工作线程负责运行一些代码，然后调用 SetEvent，以指明这项工作已经完成。另一个线程负责执行下面的代码：

```
WaitForSingleObject(hEventWorkerThreadDone);
PulseEvent(hEventMoreWorkToBeDone);
```

这个工作线程的代码段设计得很差，因为它无法可靠地运行。当工作线程调用 SetEvent之后，另一个线程可能立即醒来，并调用 PulseEvent。该工作线程不得不停止运行，没有机会从它对 SetEvent的调用中返回，更不要说调用 WaitForSingleObject函数了。结果， hEvent MoreWorkToBeDone事件的通知就完全被工作线程错过了。

如果像下面所示的那样重新编写工作线程的代码，以便调用 SingleObjectAndWait函数，那么该代码就能够可靠地运行，因为通知和等待都能够以原子操作方式来进行：

```
// Perform some work.
.
.
.
SignalObjectAndWait(hEventWorkerThreadDone,
   hEventMoreWorkToBeDone, INFINITE, FALSE);
// Do more work.
.
.
.
```

当非工作线程醒来时，它能够百分之百地确定工作线程正在等待 hEventMoreWorkToBe Done事件，因此能够确保看到产生该事件。

Windows 98 Windows 98 没有这个函数的可以使用的实现代码。