第 3 章　C++中的多线程

在现代程序设计中，多线程变得越来越重要。原因之一是多线程可以使得程序更加充分地利用 CPU，从而使得编写的程序更加高效。另一个原因是对于处理事件驱动的代码，多线程是一种很自然的选择，在现在高度分布式的、网络化的、基于 GUI 的环境中，事件驱动代码非常普遍。当然，使用地最广泛的操作系统 Windows 支持多线程也是一个因素。无论因为什么原因，不断增加的多线程的使用改变了程序员对程序基本结构的认识。尽管 C++没有内建的对多线程程序的支持，但是它却非常适合这一领域。

由于多线程变得越来越重要，因此在本章将尝试使用 C++创建多线程程序。本章开发了两个多线程应用程序。第一个是线程控制面板，可以使用它来控制程序中线程的执行。这不仅是一个有趣的多线程示例，而且是一个在开发多线程应用程序时可以使用的实用工具。第二个程序创建了第 2 章垃圾回收器的修订版本，使得它在后台线程运行，从而说明了如何将多线程应用到实际的示例中。

本章还有另外一个目的：显示 C++是如何熟练地与操作系统直接对接。一些其他语言，如 Java，在您的程序和 OS 之间有一个处理层。对于某些类型的程序(如实时环境中使用的那些程序)而言，这个层会严重影响系统性能。与之形成鲜明对比的是，C++可以直接访问操作系统提供的底层功能。这也是 C++能够创造高性能代码的原因之一。

3.1　什么是多线程

在开始之前，有必要准确地定义术语多线程的含义。多线程是多任务的特殊形式。通常，有两种类型的多任务：基于进程和基于线程的多任务。进程本质上是正在执行的程序。因此，基于进程的多任务就是允许您的计算机同时运行两个或者更多程序的特性。例如，基于进程的多任务允许您在使用电子制表软件或者浏览 Internet 的同时运行文字处理程序。在基于进程的多任务中，程序是调度程序可以分派的最小代码单元。

线程是可执行代码的可分派单元。这个名称来源于“执行的线索”的概念。在基于线程的多任务的环境中，所有进程有至少一个线程，但是它们可以具有多个任务。这意味着单个程序可以并发执行两个或者多个任务。例如，文本编辑器可以在打印文本的同时格式化文本，只要这两个动作是被两个独立的线程执行。基于进程的多任务与基于线程的多任务之间的区别可以归纳如下：基于进程的多任务处理程序的并发执行，基于线程的多任务处理相同程序的不同片断的并发执行。

在前面的讨论中，需要明确：只有在多 CPU 的系统中，才可能有真正的并发执行，在那里每个进程或者线程可以不受限制地访问 CPU。对于单个 CPU 的系统(在当前使用的系统中，这是主流)，仅能够在表面上做到并发执行。在单个 CPU 的系统中，每个进程或者线程都接收一部分 CPU 时间，时间的数量由几个因素来确定，包括进程或线程的优先级。尽管大多数计

算机并没有真正意义上的并发执行，但是在编写多线程应用程序的时候，您应该假定它确实有并发能力。这是因为您不能够知道单个线程执行的确切顺序，或者它们是否能够按照相同的顺序执行两次。因此，最好假定程序确实是在并发执行。

多线程对程序结构的改变

多线程改变了程序的基本结构。不同于按照严格的线性方式执行的单线程程序，多线程程序并发地执行它自身的各个部分。这样，所有的多线程程序都包含了相似的元素。因此，多线程程序的主要问题是管理线程之间的交互。

如前所述，所有的进程都至少包含一个执行线程，称之为主线程。主线程在程序开始时创建。在多线程程序中，主线程创建一个或者多个子线程。因此，每个多线程的进程都以一个执行线程开始，然后创建一个或者多个附加的线程。在设计合理的程序中，每个线程都代表一个逻辑上独立的活动单元。

多线程的主要优点是可以让您编写非常高效的程序，因为它使得您可以利用大多数程序都具有的空闲时间。大多数的 I/O 设备，无论是网络端口、磁盘驱动器还是键盘，速度都比 CPU 慢很多。通常，程序将主要的执行时间都花费在等待接收或者发送数据上。通过谨慎地使用多线程，您的程序可以在空闲的时候执行另一个任务。例如，当程序的一部分通过 Internet 发送文件时，另一个部分可以读取键盘的输入，还有一个部分可以将下一步要发送的数据块缓存。

3.2 为什么 C++没有内建支持多线程

C++没有包含任何对多线程应用程序的内建的支持。相反，它依赖于操作系统提供这个特性。考虑到 Java 和 C# 都提供了内建的多线程支持，您会很自然地问，为什么 C++没有提供，答案是因为效率、控制以及 C++适用的应用程序的范围。让我们逐一分析。

由于没有内建对多线程的支持，因此 C++没有尝试定义一种“万能的”解决方案。相反，C++允许您直接使用操作系统提供的多线程特性。这种方法意味着您的程序可以使用执行环境支持的、最高效的方法来实现多线程。由于许多的多任务环境提供了对多线程丰富的支持，因此能够访问这些支持对于创建高性能的多线程程序至关重要。

使用操作系统的函数来支持多线程使得可以全面地使用执行环境提供的控制。考虑 Windows 环境。它提供了多组线程相关函数来有条理地控制线程的创建和管理。例如，Windows 有多种方法来控制对共享资源的访问，共享资源包括信号、互斥体、事件对象、可等待定时器以及临界区。由于操作系统能力的不同，很难将这种灵活性设计到一门语言中。因此，对于多线程语言层次的支持通常意味着仅提供特性的“最小公倍数”。通过 C++，您可以访问操作系统提供的所有特性。当编写高性能的代码时，这是非常重要的优点。

C++是为所有类型的程序设计类型设计的，从嵌入式系统(在执行环境中没有操作系统)到高度分布的、基于 GUI 的终端用户应用程序以及介于二者之间的一切程序。因此，C++不能够对它的执行环境加入明显的限制。内建的对多线程的支持将会从根本上将 C++限制在那些支持多线程的环境中，从而阻止了在不使用线程的环境中开发软件时使用 C++。

在最后的分析中，没有内建的对多线程的支持是 C++的一个主要优点，因为这样可以使用对目标执行环境最高效的方式编写程序。记住，C++的功能无处不在。多线程的情况很明显是

一种“简单就好”的情况。

3.3 选用什么样的操作系统和编译器

由于 C++依赖于操作系统提供对多线程程序设计的支持，因此在本章有必要选择一种操作系统作为多线程应用程序的平台。由于 Windows 是世界上使用最广泛的操作系统，因此本章使用了这个操作系统。然而，许多信息都可以适用于任何支持多线程的 OS。

由于在设计 Windows 程序时，Visual C++是应用最广泛的编译器，它也就是本章示例使用的编译器。在下面的部分中这个重要性很明显。然而，如果您使用了其他编译器，就很容易修改这些代码来适应它。

提示：

本章的示例假定读者具有 Windows 程序设计的基本知识。

3.4 Windows 线程函数概述

Windows 提供了多组支持多线程的应用程序接口(API)函数。许多读者已经对 Windows 提供的多线程函数有一定程度的了解，但是对于那些不熟悉这些的读者，本章提供了这些函数的概述。记住，Windows 提供了许多其他的基于多线程的函数，这些函数需要您自己去探索。

为了使用 Windows 的多线程函数，必须在程序中包含<Windows.h>。

3.4.1 线程的创建和终止

Windows API 提供了 CreateThread()函数来创建一个线程。其原型如下所示：

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES secAttr,
                    SIZE_T stackSize,
                    LPTHREAD_START_ROUTINE threadFunc,
                    LPVOID param,
                    DWORD flags,
                    LPDWORD threadID);
```

在此，secAttr 是一个用来描述线程的安全属性的指针。如果 secAttr 是 NULL，就会使用默认的安全描述符。

每个线程都具有自己的堆栈。可以使用 stackSize 参数来按字节指定新线程堆栈的大小。如果这个整数值为 0，那么这个线程堆栈的大小与创建它的线程相同。如果需要的话，这个堆栈可以扩展。(通常使用 0 来指定线程堆栈的大小)。

每个线程都在创建它的进程中通过调用线程函数来开始执行。线程的执行一直持续到线程函数返回。这个函数的地址(也就是线程的入口点)在 threadFunc 中指定。每个线程函数都必须具有这样的原型：

```
DWORD WINAPI threadfunc(LPVOID param);
```

需要传递给新线程的任何参数都在 CreateThread()的 param 中指定。线程函数在它的参数中

接收这个 32 位的值。这个参数可以用作任何目的。函数返回它的退出状态。

参数 flags 确定了线程的执行状态。如果它是 0，线程会立即执行。如果是 CREATE_SUSPEND，线程则以挂起状态创建并等待执行。(可以通过调用 ResumeThread()来开始执行，稍后讨论)。

与线程相关的标识符以 threadID 所指向的长整型返回。

如果成功，函数则向线程返回一个句柄。如果失败，则返回 NULL。可以通过调用 CloseHandle()来显式销毁这个线程。否则，会在父进程结束时自动销毁它。

如前所述，当线程的入口函数返回时终止执行线程。进程也可以使用 TerminateThread()或者 ExitThread()来手动终止线程，这两个函数的原型如下：

```
BOOL TerminateThread(HANDLE thread, DWORD status);

VOID ExitThread(DWORD status);
```

对于 TerminateThread()，thread 是将要终止的线程的句柄。ExitThread()只能用来终止调用了 ExitThread()的线程。对于两个函数而言，status 是终止状态。TerminateThread()如果成功，则会返回非 0 值，否则返回 0。

调用 ExitThread()在功能上等价于允许线程函数正常返回。这意味着堆栈会正确地重新设置。当使用 TerminateThread()结束线程时，线程会立刻终止，而会执行任何特定的清理任务。另外，TerminateThread()可能会停止正在执行重要操作的线程。为此，当入口函数返回时，通常最好(也是最容易的)让线程正常终止。

3.4.2 Visual C++对 CreateThread()和 ExitThread()的替换

尽管 CreateThread()和 ExitThread()是用来创建并终止线程的 Windows API 函数，我们在本章并不会使用它们。原因是在 Visual C++中(其他的 Windows 兼容的编译器也可能有这个问题)使用这两个函数时，会导致内存泄漏，丢失少量的内存。对于 Visual C++，如果多线程程序利用了 C/C++标准库函数并使用了 CreateThread()和 ExitThread()，就会丢失少量的内存。(如果您的程序没有使用 C/C++的标准库，就不会发生这样的内存丢失)。为了避免这种情况，必须使用 Visual C++运行库中定义的函数来开始和终止线程，而不是使用由 Win32 API 指定的函数。这些函数类似于 CreateThread()和 ExitThread()，但是不会产生内存泄漏。

提示：

如果使用非 Visual C++的编译器，如果需要的话，检查它的文档来确定是否需要忽略 CreateThread()和 ExitThread()，以及如何做到这一点。

Visual C++用_beginthreadex()和_endthreadex()来取代 CreateThread()和 ExitThread()。这两个函数都需要头文件<process.h>。下面是_beginthreadex()函数的原型：

```
uintptr_t _beginthreadex(void *secAttr, unsigned stackSize,
                         unsigned (__stdcall *threadFunc)(void *),
                         void *param, unsigned flags,
                         unsigned *threadID);
```

正如您看到的那样，_beginthreadex()的参数类似于 CreateThread()的参数。另外，这些参数与 CreateThread()指定的参数有相同的含义。secAttr 是一个用来描述线程安全性属性的指针。

然而，如果 secAttr 为 NULL，则会使用默认的安全描述符。新线程堆栈的大小由 stackSize 参数按字节数传递。如果这个值为 0，那么这个线程堆栈的大小与进程中创建它的主线程的大小相同。

线程函数的地址(也就是线程的入口点)在 threadFunc 中指定。对于_beginthreadex()，线程函数的原型如下：

```
unsigned_stdcall threadfunc(void *param)
```

这个原型在功能上等价于 CreateThread()的线程函数的原型，但是它使用了不同的类型名称。想要传递给新线程的任何参数都在 param 参数中指定。

flags 参数确定线程的执行状态。如果 flags 参数为 0，线程会立即开始执行。如果 flags 参数为 CREATE_SUSPEND，则以挂起状态创建线程。(可以调用 ResumeThread()来开始它)。与线程相关的标识符以 threadID 指向的 double word 返回。

如果成功，则这个函数返回一个线程的句柄；如果失败，则返回 0。类型 uintptr_t 指定了可以拥有指针或者句柄的 Visual C++类型。

_endthreadex()的原型如下：

```
void _endthreadex(unsigned status);
```

它的功能就像 ExitThread()那样，停止线程并返回 *status* 中指定的退出代码(exit code)。

由于 Windows 下使用最广泛的编译器是 Visual C++，因此本章示例将使用_beginthreadex()和_endthreadex()而不是使用它们的等价的 API 函数。如果您使用了非 Visual C++的编译器，那么只需要用 CreateThread()和 EndThread()来替代这两个函数。

当使用_beginthreadex()和_endthreadex()时，必须记住链接多线程库。这随编译器的不同而不同。在此有一些示例。当使用 Visual C++的命令行编译器时，包括-MT 选项。为了在 Visual C++ 6 IDE 中使用多线程库，首先要激活“Project | Settings”属性页。然后选择“C/C++”选项卡。接着在“Category”下拉列表框中选择“Code Generation”，然后在“Use Runtime Library ”下拉列表框中选择“Multithreaded”。对于 Visual C++ 7 .NET IDE，选择“Project |Properties”。然后选择“C/C++”条目，并加亮显示“Code Generation”。最后，将“Multithreaded”选择为运行库。

3.4.3 线程的挂起和恢复

线程的执行可以通过调用 SuspendThread()来挂起。可以通过调用 ResumeThread()来恢复它。这两个函数的原型如下：

```
DWORD SuspendThread(HANDLE hThread);

DWORD ResumeThread(HANDLE hThread);
```

两个函数都通过 hThread 来传递线程的句柄。

每个执行的线程都有与其相关的挂起计数。如果这个计数为 0，那么不会挂起线程。如果为非 0 值，则线程就会处于挂起状态。每次调用 SuspendThread()都会增加这个计数。每次调用 ResumeThread()都会减小这个挂起计数。挂起的线程只有在它的挂起计数达到 0 时才会恢复。因此，为了恢复一个挂起的线程，对 ResumeThread()的调用次数必须与对 SuspendThread()的调

用次数相等。

这两个函数都返回线程先前的挂起计数，如果发生错误，返回值为 - 1。

3.4.4 改变线程的优先级

在 Windows 中，每个线程都与一个优先级设置相关。线程的优先级决定了线程接收的 CPU 时间的多少。低优先级的线程接收比较少的时间，高优先级的线程接收比较多的时间。当然，线程接收的 CPU 时间的多少对于它的执行性能以及它与系统中当前执行的其他线程之间的交互有着深远的影响。

在 Windows 中，线程优先级的设置是两个值的组合：进程总体的优先级类别以及相对于这个优先级类别的各个线程的优先级设置。也就是说，线程实际的优先级由进程的优先级类别和各个线程的优先级的组合来确定。后面会逐一讲述。

1. 优先级类别

在默认情况下，进程具有普通的优先级类别，大多数程序在其执行的声明周期内保持这个普通的优先级类别。尽管在本章的示例中没有改变优先级类别，但是为了完整起见，在此给出了线程优先级类别的简单概况。

Windows 定义了 6 个优先级类别，相应的值以从高到低的顺序显示如下：

REALTIME_PRIORITY_CLASS

HIGH_PRIORITY_CLASS

ABOVE_NORMAL_PRIORITY_CLASS

NORMAL_PRIORITY_CLASS

BELOW_NORMAL_PRIORITY_CLASS

IDLE_PRIORITY_CLASS

在默认情况下，程序的优先级类别为 NORMAL_PRIORITY_CLASS。通常，您不需要改变程序的优先级类别。事实上，改变进程的优先级类别对于整个计算机系统的性能会有负面的影响。例如，如果您将一个程序的优先级类别增加到 REALTIME_PRIORITY_CLASS，它就会支配 CPU。对于某些特殊的应用程序，可能需要增加应用程序的优先级类别，但通常并不需要。如前所述，本章的应用程序没有改变优先级类别。

当确实需要改变程序的优先级类别时，可以调用 SetPriorityClass()。可以调用 GetPriorityClass()来获取当前的优先级类别。这两个函数的原型如下：

```
DWORD GetPriorityClass(HANDLE hApp);

BOOL SetPriorityClass(HANDLE hApp, DWORD priority);
```

在此，*hApp* 是进程的句柄。GetPriorityClass()返回应用程序的优先级类别，如果失败的话，返回 0。对于 SetPriorityClass()，*priority* 指定了进程的新优先级类别。

2. 线程优先级

对于给定的优先级类别，各个线程的优先级确定了它在进程内接收的 CPU 时间的多少。当线程第一次创建时，它具有普通的优先级，但是您可以改变线程的优先级——即使在它执行时。

可以通过调用 GetThreadPriority()来获取线程的优先级设置。可以使用 SetThreadPriority()来增加或者减小线程的优先级。这两个函数的原型如下：

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

int GetThreadPriority(HANDLE hThread);
```

对于这两个函数而言，*hThread* 是线程的句柄。对于 SetThreadPriority()，*priority* 是新的优先级设置。如果发生错误，则返回值为 0；否则，返回非 0 值。GetThreadPriority()会返回当前的优先级设置。优先级设置按照从高到低的顺序如表 3-1 所示。

表 3-1 优先级设置

线程优先级	值
THREAD_PRIORITY_TIME_CRITICAL	15
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_IDLE	-15

这些值相对于进程的优先级类别或增或减。通过组合进程的优先级类别和线程的优先级，Windows 向应用程序提供了 31 个不同的优先级设置的支持。

如果有错误发生，则 GetThreadPriority()返回 THREAD_PRIORITY_ERROR_RETURN。

在大多数情况下，如果线程具有普通的优先级类别，那么可以随意地改变它的优先级设置，而不必担心会给整个系统的性能带来灾难性的影响。您将会看到，在下面部分开发的线程控制面板中，可以改变进程内线程的优先级设置(但是不能改变优先级类别)。

3.4.5 获取主线程的句柄

主线程的执行是可以控制的。为此，需要获取它的句柄。做到这一点最简单的方法是调用 GetCurrentThread()，其原型如下：

```
HANDLE GetCurrentThread(void);
```

这个函数返回当前线程的伪句柄(pseudohandle)。之所以称之为伪句柄，是因为它是一个预定义的值，总是引用当前的线程，而不是引用指定的调用线程。然而，它能够用在任何可以使用普通线程处理的地方。

3.4.6 同步

在使用多线程或多进程时，有时需要调整两个或者多个线程(或者进程)之间的活动。这个过程称为同步。当两个或者多个线程需要访问共享资源，而这个共享资源在同一时刻只能由一个线程使用时，就需要使用同步。例如，当一个线程在写文件时，在此时必须阻止另一个线程

也这么做。同步的另一个原因是有时线程需要等待由另一个线程引发的事件。在此情况下，必须采取某种措施将第一个线程保持挂起状态，直到这个事件发生。随后等待的线程必须恢复执行。

通常某个任务会处于两种状态。首先，它可能正在执行(或者在获得它的时间段时就开始执行)。另外，任务可能被阻塞，等待某个资源或者事件。在此情况下其执行被挂起，直到所需的资源可以使用或者所等待的事件发生。

如果您对于同步问题或者它的常用解决方案(信号量)不熟悉，下面的部分将对此进行讨论。

1. 理解同步问题

Windows 必须提供某种特殊的服务来允许对共享资源的访问同步，因为如果没有操作系统的协助，进程或者线程就没有办法得知它是否在单独访问某个资源。为了理解这个问题，假定您在为一个没有提供任何同步支持的多任务操作系统编写程序，并且假定您具有两个并发执行的线程 A 和 B，它们都不时地访问某个资源 R(如磁盘文件)，这个资源在某个时刻只能被一个线程访问。为了在一个线程使用这个资源时阻止另一个线程访问它，您尝试了下面的解决方案。首先，创建了一个初始化值为 0 并且两个线程都可以访问的变量，名为 flag。然后，在使用访问 R 的每段代码之前，等待 flag 被清 0，然后设置 flag，访问 R，最后将 flag 清 0。也就是说，在每个线程访问 R 之前，执行如下的代码：

```
while(flag) ; // wait for flag to be cleared
flag = 1; // set flag

// ... access resource R ...

flag = 0; // clear the flag
```

这段代码隐含的概念是，如果设置了 flag，则两个线程都不能够访问 R。从概念上讲，这种方法是正确的解决方案。然而，实际上它远远没有达到要求，原因很简单：它并非总是有效！让我们看一下原因。

使用刚才给定的代码，有可能两个进程同时访问 R。while 循环在本质上执行重复的加载和比较 flag 上的指令。换句话说，它一直在测试 flag 的值。当 flag 被清 0 的时候，代码的下一行将设置 flag 的值。问题在于，这两个操作有可能在两个不同的时间段执行。在两个时间段之间，flag 的值有可能被另一个线程访问，从而 R 被两个线程同时访问。为了理解这一点，假定线程 A 进入 while 循环，发现 flag 为 0，这是访问 R 的绿灯。然而，在将 flag 设置为 1 之前，其时间段用尽，线程 B 恢复执行。如果 B 执行了它的 while，它也发现 flag 没有被设置，并且认为它可以安全地访问 R。然而，当 A 重新开始时，它也会访问 R。问题的关键在于对 flag 的测试和设置没有包含在一个连续的操作中，而是可以被分为两个部分，正如刚才说明的那样。无论您如何努力，都没有办法只使用应用层的代码以绝对保证在同一时刻只有一个线程访问 R。

对同步问题的解决方案简单而优雅。操作系统(在 Windows 中)提供了一个例程，在一个连续的操作中完成对 flag 的测试和设置(如果可能的话)。用操作系统工程师的话来说，这就是所谓的测试和置位(test and set)操作。由于历史的原因，用来控制对共享资源的访问并提供线程(以及进程)间同步的标记被称为信号量。信号量是 Windows 同步系统的核心。

2. Windows 的同步对象

Windows 支持几种类型的同步对象。第一种类型是经典的信号量。当使用信号量时，可以完全同步资源，在此情况下只有一个进程或者线程在同一时刻可以访问这个资源，或者信号量允许不超过一定数量的进程或者线程在同一时刻访问资源。信号量使用计数器来实现，当某个任务使用信号量时，计数器减小；当这个任务释放信号量时，计数器增加。

第二个同步对象是互斥体信号量，或者简称为互斥体。互斥体将一个资源同步，保证在任何时候都只有一个线程或者进程来访问它。在本质上，互斥体是标准信号量的一个特殊版本。

第三个同步对象是事件对象。它可以用来阻塞对某个资源的访问，直到某个其他的进程或者线程发送信号，通知可以使用资源(也就是一个事件对象发送某个指定的事件发生的信号)。

第四个同步对象是可等待计时器。可等待计时器阻塞线程的执行，直到指定的时间。也可以创建计时器序列，这是一个计时器的列表。

可以使用临界区对象将一个代码段放入临界区，从而阻止在同一时刻一个以上的线程使用这段代码。当一个线程进入临界区时，其他线程只有在第一个线程离开整个临界区时才可以使用它。

本章使用的惟一的同步对象是互斥体，下面的部分将对其进行描述。然而，C++程序员可以使用所有的 Windows 定义的同步对象。如前所述，这是使得 C++依赖于操作系统处理多线程的主要优点之一：所有的多线程特性都在您的控制之中。

3. 使用互斥体同步线程

如前所述，互斥体是一种特殊的信号量，在给定的时间内，只允许一个线程访问某个资源。在使用互斥体之前，必须使用 CreatMutex()创建一个互斥体，函数原型如下：

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES secAttr,
                   BOOL acquire,
                   LPCSTR name);
```

在此，*secAttr* 是用来描述安全属性的指针。如果 *secAttr* 为 NULL，则使用默认的安全描述符。

如果创建的线程需要互斥体的控制，则 *acquire* 为 true，否则为 false。

name 参数指向一个字符串，这个字符串是互斥体对象的名称。互斥体是一个全局对象，它可能被其他进程使用。为此，当两个进程都打开了使用相同名称的互斥体时，二者引用了相同的互斥体。使用这种方法可以将两个进程同步。这个名称也可以为 NULL，在此情况下这个信号量被限制在一个进程之内。

如果成功，则 CreatMutex()函数返回信号量的句柄，否则，返回 NULL。当主进程结束时，互斥体的句柄则自动关闭。当不再需要时，可以调用 CloseHandle()来显式地关闭互斥体的句柄。

当创建信号量时，可以调用两个相关的函数来使用它：WaitForSingleObject()和 ReleaseMutex()。这两个函数的原型如下：

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

BOOL ReleaseMutex(HANDLE hMutex);
```

WaitForSingleObject()等待一个同步对象，直到这个对象可以使用或者超时之后才会返回。

在使用互斥体时，*hObject* 是互斥体的句柄。*howLong* 参数以毫秒为单位指定调用例程的等待时间。当这个时间用尽时，会返回超时错误。为了无限期地等待，可以使用值 INFINITE。当成功时(也就是访问被准许)，这个函数返回 WAIT_OBJECT_0。当发生超时时，返回 WAIT_TIMEOUT。

ReleaseMutex()释放互斥体，并允许其他线程获取它。在此，*hMutex* 是互斥体的句柄。如果成功，则函数返回非 0 值；如果失败，则返回 0。

为了使用互斥体控制对共享资源的访问，封装了访问在调用 WaitForSingleObject()和 ReleaseMutex()之间的资源的代码，如下面的代码所示(当然，超时期限随应用程序的不同而不同)。

```
if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT) {
    // handle time-out error
}

  // access the resource

ReleaseMutex(hMutex);
```

通常，您会选择足够长的超时期限来适应程序的操作。如果在开发多线程应用程序时重复出现超时错误，那么通常意味着您创建了死锁条件。当一个线程等待另一个线程永远都不会释放的互斥体时，就会发生死锁。

3.5 创建线程控制面板

当开发多线程程序时，体验不同的优先级设置通常是有用的。能够动态地挂起或者重新启动线程，甚至终止线程也是有用的。您将会看到，使用刚才描述的线程函数来创建允许完成这些任务的线程控制面板相当容易。另外，可以在您的多线程程序运行时使用控制面板。线程控制面板的动态特性使得您可以轻易地改变线程的执行配置文件并观察结果。

这个部分开发的线程控制面板能够控制一个线程。然而，可以创建任意多个面板，每个面板都可以控制不同的线程。为了简单起见，线程控制面板用非模式对话框实现，这个面板属于桌面而不是属于应用程序，这个面板控制此应用程序的线程。

线程控制面板可以用来执行下面的操作：

- 设置线程的优先级
- 挂起线程
- 重新执行线程
- 终止线程

另外它还显示了线程的当前优先级设置。线程控制对话框如图 3-1 所示。

如前所述，控制面板是非模式对话框。当非模式对话框激活时，应用程序的其余部分仍然可以运行。因此，控制面板的运行不依赖于它所使用的应用程序。

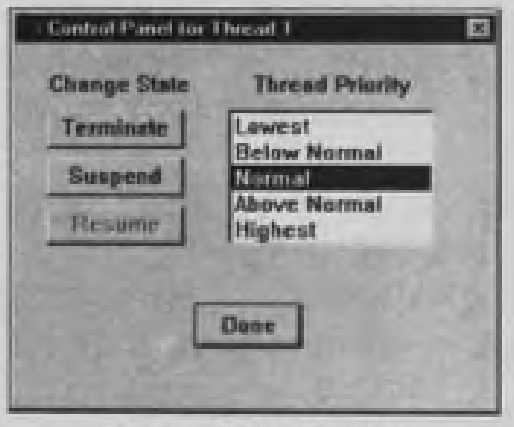

图 3-1 线程控制对话框

3.5.1 线程控制面板

线程控制面板的代码如下所示。文件名为 tcp.cpp。

```
// A thread control panel.

#include <map>
#include <windows.h>
#include "panel.h"
using namespace std;

const int NUMPRIORITIES = 5;
const int OFFSET = 2;

// Array of strings for priority list box.
char priorities[NUMPRIORITIES][80] = {
  "Lowest",
  "Below Normal",
  "Normal",
  "Above Normal",
  "Highest"
};

// A Thread Control Panel Class.
class ThrdCtrlPanel {
  // Information about the thread under control.
  struct ThreadInfo {
    HANDLE hThread;  // handle of thread
    int priority;    // current priority
    bool suspended;  // true if suspended
    ThreadInfo(HANDLE ht, int p, bool s) {
      hThread = ht;
      priority = p;
      suspended = s;
    }
  };

  // This map holds a ThreadInfo for each
```

```
  // active thread control panel.
  static map<HWND, ThreadInfo> dialogmap;

public:

  // Construct a control panel.
  ThrdCtrlPanel(HINSTANCE hInst, HANDLE hThrd);
  // The control panel's callback function.
  static LRESULT CALLBACK ThreadPanel(HWND hwnd, UINT message,
                          WPARAM wParam, LPARAM lParam);
};

// Define static member dialogmap.
map<HWND, ThrdCtrlPanel::ThreadInfo>
  ThrdCtrlPanel::dialogmap;

// Create a thread control panel.
ThrdCtrlPanel::ThrdCtrlPanel(HINSTANCE hInst,
                             HANDLE hThrd)
{
  ThreadInfo ti(hThrd,
                GetThreadPriority(hThrd)+OFFSET,
                false);

  // Owner window is desktop.
  HWND hDialog = CreateDialog(hInst, "ThreadPanelDB",
                        NULL,
                        (DLGPROC) ThreadPanel);

  // Put info about this dialog box in the map.
  dialogmap.insert(pair<HWND, ThreadInfo>(hDialog, ti));

  // Set the control panel's title.
  char str[80] = "Control Panel for Thread ";
  char str2[4];
  _itoa(dialogmap.size(), str2, 10);
  strcat(str, str2);
  SetWindowText(hDialog, str);

  // Offset each dialog box instance.
  MoveWindow(hDialog, 30*dialogmap.size(),
          30*dialogmap.size(),
          300, 250, 1);

  // Update priority setting in the list box.
  SendDlgItemMessage(hDialog, IDD_LB, LB_SETCURSEL,
                        (WPARAM) ti.priority, 0);

// Increase priority to ensure control. You can
// change or remove this statement based on your
// execution environment.
SetThreadPriority(GetCurrentThread(),
```

```
                    THREAD_PRIORITY_ABOVE_NORMAL);
}


// Thread control panel dialog box callback function.
LRESULT CALLBACK ThrdCtrlPanel::ThreadPanel(HWND hwnd,
                                            UINT message,
                                            WPARAM wParam,
                                            LPARAM lParam)
{
  int i;
  HWND hpbRes, hpbSus, hpbTerm;

  switch(message) {
    case WM_INITDIALOG:
      // Initialize priority list box.
      for(i=0; i<NUMPRIORITIES; i++) {
        SendDlgItemMessage(hwnd, IDD_LB,
            LB_ADDSTRING, 0, (LPARAM) priorities[i]);
        }

      // Set suspend and resume buttons for thread.
      hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
      hpbRes = GetDlgItem(hwnd, IDD_RESUME);
      EnableWindow(hpbSus, true); // enable Suspend
      EnableWindow(hpbRes, false); // disable Resume
      return 1;
  case WM_COMMAND:
    map<HWND, ThreadInfo>::iterator p = dialogmap.find(hwnd);

    switch(LOWORD(wParam)) {
      case IDD_TERMINATE:
        TerminateThread(p->second.hThread, 0);

        // Disable Terminate button.
        hpbTerm = GetDlgItem(hwnd, IDD_TERMINATE); }
        EnableWindow(hpbTerm, false); // disable

        // Disable Suspend and Resume buttons.
        hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
        hpbRes = GetDlgItem(hwnd, IDD_RESUME);
        EnableWindow(hpbSus, false); // disable Suspend
        EnableWindow(hpbRes, false); // disable Resume

        return 1;
      case IDD_SUSPEND:
        SuspendThread(p->second.hThread);

        // Set state of the Suspend and Resume buttons.
        hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
        hpbRes = GetDlgItem(hwnd, IDD_RESUME);
        EnableWindow(hpbSus, false); // disable Suspend
```

```
        EnableWindow(hpbRes, true); // enable Resume

        p->second.suspended = true;
        return 1;
      case IDD_RESUME:
        ResumeThread(p->second.hThread);

        // Set state of the Suspend and Resume buttons.
        hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
        hpbRes = GetDlgItem(hwnd, IDD_RESUME);
        EnableWindow(hpbSus, true); // enable Suspend
        EnableWindow(hpbRes, false); // disable Resume

        p->second.suspended = false;
        return 1;
      case IDD_LB:
        // If a list box entry was clicked,
        // then change the priority.
        if(HIWORD(wParam)==LBN_DBLCLK) {
          p->second.priority = SendDlgItemMessage(hwnd,
                               IDD_LB, LB_GETCURSEL,
                               0, 0);
        SetThreadPriority(p->second.hThread,
                                  p->second.priority-OFFSET);
        }
        return 1;
      case IDCANCEL:
        // If thread is suspended when panel is closed,
        // then resume thread to prevent deadlock.
        if(p->second.suspended) {
           ResumeThread(p->second.hThread);
          p->second.suspended = false;
        }

        // Remove this thread from the list.
        dialogmap.erase(hwnd);

        // Close the panel.
        DestroyWindow(hwnd);
        return 1;
      }
    }
    return 0;
}
```

控制面板需要如下的资源文件，称为 tcp.rc。

```
#include <windows.h>
#include "panel.h"

ThreadPanelDB DIALOGEX 20, 20, 140, 110
CAPTION "Thread Control Panel"
```

```
STYLE WS_BORDER | WS_VISIBLE | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
  DEFPUSHBUTTON "Done", IDCANCEL, 55, 80, 33, 14
  PUSHBUTTON "Terminate", IDD_TERMINATE, 10, 20, 42, 12
  PUSHBUTTON "Suspend", IDD_SUSPEND, 10, 35, 42, 12
  PUSHBUTTON "Resume", IDD_RESUME, 10, 50, 42, 12
  LISTBOX IDD_LB, 65, 20, 63, 42, LBS_NOTIFY | WS_VISIBLE |
          WS_BORDER | WS_VSCROLL | WS_TABSTOP
  CTEXT "Thread Priority", IDD_TEXT1, 65, 8, 64, 10
  CTEXT "Change State", IDD_TEXT2, 0, 8, 64, 10
}
```

控制面板使用了如下的头文件 panel.h

```
#define IDD_LB         200
#define IDD_TERMINATE  202
#define IDD_SUSPEND    204
#define IDD_RESUME     206
#define IDD_TEXT1      208
#define IDD_TEXT2      209
```

为了使用线程控制面板，需要遵循如下步骤：

(1) 在程序中包含 tcp.cpp

(2) 在程序的资源文件中包含 tcp.rc

(3) 创建想要控制的一个线程或多个线程

(4) 为每个线程实例化 ThrdCtrlPanel 对象

每个 ThrdCtrlPanel 对象链接线程和控制这个线程的对话框。

对于多个文件需要访问 ThrdCtrlPanel 的大项目，可以使用头文件 tcp.h 来包含 ThrdCtrlPanel 的声明。tcp.h 的内容如下：

```
// A header file for the ThrdCtrlPanel class.

class ThrdCtrlPanel {
public:

  // Construct a control panel.
  ThrdCtrlPanel(HINSTANCE hInst, HANDLE hThrd);

  // The control panel's callback function.
  static LRESULT CALLBACK ThreadPanel(HWND hwnd, UINT message,
                              WPARAM wParam, LPARAM lParam);
};
```

3.5.2 线程控制面板的详细分析

让我们仔细观察这个线程控制面板。开始它进行了如下的全局定义：

```
const int NUMPRIORITIES = 5;
const int OFFSET = 2;

// Array of strings for priority list box.
```

```
char priorities[NUMPRIORITIES][80] = {
  "Lowest",
  "Below Normal",
  "Normal",
  "Above Normal",
  "Highest"
};
```

priorities 数组包含了对应于线程的优先级设置的字符串。它初始化控制面板内的列表框，这个列表框显示了当前线程优先级。优先级的数量由 NUMPRIORITIES 指定，对于 Windows 这个值为 5。因此，NUMPRIORITIES 定义了线程可能具有的不同优先级的数量。(如果您在另一个操作系统中使用了这些代码，则可能需要其他的值)。通过使用控制面板，可以为线程设置如下的优先级：

THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_LOWEST

另外两个线程优先级的设置：

THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_IDLE

没有被支持，是因为对于这个控制面板，它们没有实际的意义。例如，如果想要创建一个 time-critical 应用程序，最好将它的优先级类别设置为 time-critical。

OFFSET 定义了在列表框索引和线程优先级之间转换的偏移量。应该记得普通优先级的值为 0。在此示例中，最高的优先级为 THREAD_PRIORITY_HIGHEST，其值为 2。最低的优先级为 THREAD_PRIORITY_LOWEST，其值为 -2。由于列表框索引以 0 开始，因此使用了这个偏移量来在索引和优先级设置之间转换。

随后声明了 ThrdCtrlPanel 类。下面给出类的声明代码：

```
// A Thread Control Panel Class.
class ThrdCtrlPanel {
  // Information about the thread under control.
  struct ThreadInfo {
    HANDLE hThread;  // handle of thread
    int priority;    // current priority
    bool suspended; // true if suspended
    ThreadInfo(HANDLE ht, int p, bool s) {
      hThread = ht;
      priority = p;
      suspended = s;
    }
  };

  // This map holds a ThreadInfo for each
  // active thread control panel.
  static map<HWND, ThreadInfo> dialogmap;
```

关于被控制线程的信息保存在 ThreadInfo 类型的结构中。这个线程的句柄保存在 hThread 中。它的优先级保存在 priority 中。如果这个线程被挂起，suspended 则为 true，否则为 false。

静态成员 dialogmap 是一个 STL map 对象，链接线程信息与用来控制线程的对话框的句柄。由于在给定的时间可以具有多个活动的线程控制面板，因此必须有某种方法来判断某个线程与哪个面板相关。dialogmap 提供了这个链接。

1. ThreadCtrlPanel 的构造函数

ThreadCtrlPanel 的构造函数如下所示。应用程序的实例句柄和被控制的线程的句柄传递给这个构造函数。实例句柄用来创建控制面板对话框。

```
// Create a thread control panel.
ThrdCtrlPanel::ThrdCtrlPanel(HINSTANCE hInst,
                             HANDLE hThrd)
{
  ThreadInfo ti(hThrd,
                GetThreadPriority(hThrd)+OFFSET,
                false);

  // Owner window is desktop.
  HWND hDialog = CreateDialog(hInst, "ThreadPanelDB",
                       NULL,
                       (DLGPROC) ThreadPanel);

  // Put info about this dialog box in the map.
  dialogmap.insert(pair<HWND, ThreadInfo>(hDialog, ti));

  // Set the control panel's title.
  char str[80] = "Control Panel for Thread ";
  char str2[4];
  _itoa(dialogmap.size(), str2, 10);
  strcat(str, str2);
  SetWindowText(hDialog, str);

  // Offset each dialog box instance.
  MoveWindow(hDialog, 30*dialogmap.size(),
             30*dialogmap.size(),
             300, 250, 1);

  // Update priority setting in the list box.
  SendDlgItemMessage(hDialog, IDD_LB, LB_SETCURSEL,
                           (WPARAM) ti.priority, 0);

  // Increase priority to ensure control. You can
  // change or remove this statement based on your
  // execution environment.
  SetThreadPriority(GetCurrentThread(),
                    THREAD_PRIORITY_ABOVE_NORMAL);
}
```

在这个构造函数的开始，创建了一个 ThreadInfo 实例 ti，ti 包含了线程的初始设置。注意，这个优先级是通过为被控制的线程调用 GetThreadPriority()获得的。随后，通过调用 CreateDialog()创建控制面板对话框。CreateDialog()是用来创建非模式对话框的 Windows API 函数，使得对话框与创建它的应用程序无关。这个对话框的句柄被返回并存储在 hDialog 中。然后，hDialog 和包含在 ti 中的线程信息保存在 dialogmap 中。从而线程与控制它的对话框链接起来。

随后设置对话框的标题以表示线程的数量。线程数量的获取基于 dialogmap 中条目的数量。您可能想要实现另一种方法，显式地将每个线程的名称传递给 ThrdCtrlPanel 的构造函数。然而对于本章而言，只要有线程的数量就足够了。

随后，通过调用另一个 Windows API 函数 MoveWindow()，控制面板在屏幕上的位置就可以移动。这使得被显示的多个面板之间不会出现遮挡的情况。然后通过调用 Windows API 函数 SendDlgItemMessage()，线程的优先级设置就显示在优先级列表框中。

最后，当前线程的优先级增加到高于普通值。从而确保了应用程序接收足够的 CPU 时间来响应用户的输入，无论被控制线程的优先级如何。并不是所有的情况都需要这个步骤，您可以尝试它。

2. ThreadPanel()函数

ThreadPanel()是 Windows 的回调函数，用来响应用户与线程控制面板的交互。像所有的对话框回调函数一样，每当用户改变了控制状态时，它就会收到一个消息。它传递发生操作的那个对话框的句柄、消息以及这条消息所需的任何附加信息。它的常用操作模式与对话框使用的其他回调函数相同。下面的讨论描述了每条消息发生的事件。

在线程控制面板对话框首次创建时，它接收到 WM_INITDIALOG 消息，由下面的 case 序列处理：

```
case WM_INITDIALOG:
  // Initialize priority list box.
  for(i=0; i<NUMPRIORITIES; i++) {
    SendDlgItemMessage(hwnd, IDD_LB,
        LB_ADDSTRING, 0, (LPARAM) priorities[i]);
    }

  // Set Suspend and Resume buttons for thread.
  hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
  hpbRes = GetDlgItem(hwnd, IDD_RESUME);
  EnableWindow(hpbSus, true); // enable Suspend
  EnableWindow(hpbRes, false); // disable Resume
  return 1;
```

这段代码初始化了优先级列表框，并为初始状态设置了 Suspend 和 Resume 按钮，Suspend 按钮可用，Resume 按钮不可用。

用户的每一个交互都会产生 WM_COMMAND 消息。每次接收到这个消息时，获取指向 dialogmap 中这个对话框条目的迭代器，如下所示：

```
case WM_COMMAND:
  map<HWND, ThreadInfo>::iterator p = dialogmap.find(hwnd);
```

使用 p 指向的信息恰当地处理每个操作。由于 p 是一个 map 对象的迭代器，它指向一个 pair 类型的对象，pair 是 STL 定义的一个结构。这个结构包含了两个字段：first 和 second。这两个字段分别对应于由键和值组成的信息。在此情况下，句柄为键，线程信息为值。

精确指示所发生的操作的代码包含在 wParam 的低位字中，用它来控制处理剩余消息的 switch 语句。下面将逐一讲述。

当用户按下 Terminate 按钮时，终止受控的线程。这由下面的 case 序列来处理：

```
case IDD_TERMINATE:
  TerminateThread(p->second.hThread, 0);

  // Disable Terminate button.
  hpbTerm = GetDlgItem(hwnd, IDD_TERMINATE);
  EnableWindow(hpbTerm, false); // disable

  // Disable Suspend and Resume buttons.
  hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
  hpbRes = GetDlgItem(hwnd, IDD_RESUME);
  EnableWindow(hpbSus, false); // disable Suspend
  EnableWindow(hpbRes, false); // disable Resume

  return 1;
```

通过调用 TerminateThread()来终止线程。注意线程句柄的获取方式。如前所述，由于 p 是 map 对象的一个迭代器，因此它指向 pair 类型的一个对象，pair 包含了在 first 字段中的键以及 second 字段中的值。这就是可以使用表达式 p->second.hThread 获取线程句柄的原因。该线程停止后，Terminate 按钮将失效。

一旦线程终止，将不可恢复。注意控制面板使用 TerminateThread()来终止线程的执行。如前所述，必须小心使用这个函数。如果您使用控制面板来试验您自己的线程，就要确保不会出现有害的影响。

当用户按下 Suspend 按钮时，挂起线程。下面的序列完成了这个操作：

```
case IDD_SUSPEND:
  SuspendThread(p->second.hThread);

  // Set state of the Suspend and Resume buttons.
  hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
  hpbRes = GetDlgItem(hwnd, IDD_RESUME);
  EnableWindow(hpbSus, false);  // disable Suspend
  EnableWindow(hpbRes, true);   // enable Resume

  p->second.suspended = true;
  return 1;
```

通过调用 SuspendThread()来挂起线程。随后，Suspend 和 Resume 按钮的状态被更新，Resume 按钮可以使用，Suspend 按钮不可使用。从而避免了用户试图将一个线程挂起两次。

当按下 Resume 按钮时，恢复挂起的线程。由下面的代码来处理：

```
case IDD_RESUME:
```

```
  ResumeThread(p->second.hThread);

  // Set state of the Suspend and Resume buttons.
  hpbSus = GetDlgItem(hwnd, IDD_SUSPEND);
  hpbRes = GetDlgItem(hwnd, IDD_RESUME);
  EnableWindow(hpbSus, true);   // enable Suspend
  EnableWindow(hpbRes, false); // disable Resume

  p->second.suspended = false;
  return 1;
```

通过调用 ResumeThread()来恢复线程，恰当地设置 Suspend 和 Resume 按钮。

为了改变线程的优先级，用户可以双击优先级列表框中的条目。对这个事件的处理如下所示：

```
case IDD_LB:
  // If a list box entry was double-clicked,
  // then change the priority.
  if(HIWORD(wParam)==LBN_DBLCLK) {
    p->second.priority = SendDlgItemMessage(hwnd,
                         IDD_LB, LB_GETCURSEL,
                         0, 0);

    SetThreadPriority(p->second.hThread,
    p->second.priority-OFFSET);
  }
  return 1;
```

列表框生成了不同类型的提示消息来描述所发生事件的精确类型。提示消息包含在 wParam 的高位字中。这些消息的其中之一是 LBN_DBLCLK，这意味着用户双击了列表框中的一个条目。当接收到这个提示消息时，通过调用 Windows API 函数 SendDlgItemMessage()来获取这个条目的索引，来请求当前的选择。然后使用这个值来设置线程的优先级。注意减去 OFFSET 来规范索引的值。

最后，当用户关闭线程控制面板对话框时，会发送 IDCANCEL 消息。它由如下序列处理：

```
case IDCANCEL:
  // If thread is suspended when panel is closed,
  // then resume thread to prevent deadlock.
  if(p->second.suspended) {
    ResumeThread(p->second.hThread);
    p->second.suspended = false;
  }

  // Remove this thread from the list.
  dialogmap.erase(hwnd);

  // Close the panel.
  DestroyWindow(hwnd);
  return 1;
```

如果挂起线程，则会重新开始线程。为了防止不小心死锁线程，这样做是有必要的。随后，

删除 dialogmap 中这个对话框的条目。最后，调用 Windows API 函数 DestoryWindow()来删除此对话框。

3.5.3 控制面板的演示

在此有一个包含了线程控制面板的程序来说明它的使用。示例输出在图 3-2 中显示。程序创建了一个主窗口并定义两个子线程。当开始的时候，这些线程简单地从 0 到 50000 计数，并在主窗口中显示这个计数。这些线程可以通过激活一个线程控制面板来控制。

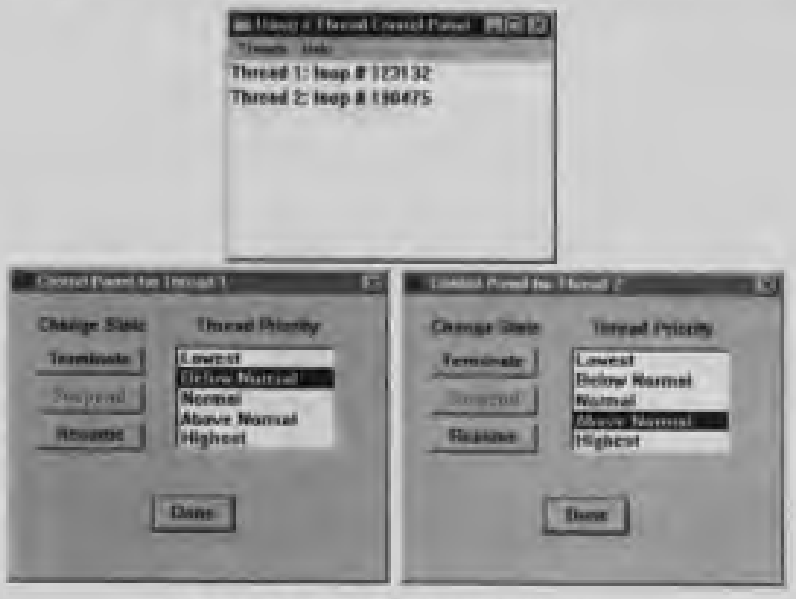

图 3-2 线程控制面板示例程序的输出

为了使用这个程序，首先在 Threads 菜单中选择 Start Threads(或者按 F2)来开始执行线程，然后选择 Threads 菜单中的 Control Panels(或者按 F3)来激活线程控制面板。当控制面板激活时，您可以设置不同的优先级。

提示：

对 Windows 程序设计的讲述超出了本书的范围。然而，示例程序的操作相当的直接，所有的 Windows 程序员应该都可以很容易地理解它们。

```
// Demonstrate the thread control panel.
#include <windows.h>
#include <process.h>
#include "thrdapp.h"
#include "tcp.cpp"

const int MAX = 50000;

LRESULT CALLBACK WindowFunc(HWND, UINT, WPARAM, LPARAM);

unsigned __stdcall MyThread1(void * param);
unsigned __stdcall MyThread2(void * param);

char str[255]; // holds output strings
```

```
unsigned tid1, tid2; // thread IDs
HANDLE hThread1, hThread2; // thread handles

HINSTANCE hInst; // instance handle

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst,
                   LPSTR args, int winMode)
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcl;
  HACCEL hAccel;

  // Define a window class.
  wcl.cbSize = sizeof(WNDCLASSEX);

  wcl.hInstance = hThisInst; // handle to this instance
  wcl.lpszClassName = "MyWin"; // window class name
  wcl.lpfnWndProc = WindowFunc; // window function
  wcl.style = 0; // default style

  wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // large icon
  wcl.hIconSm = NULL; // use small version of large icon
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW); // cursor style
  wcl.lpszMenuName = "ThreadAppMenu"; // main menu

  wcl.cbClsExtra = 0; // no extra memory needed
  wcl.cbWndExtra = 0;

  // Make the window background white.
  wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);

  // Register the window class.
  if(!RegisterClassEx(&wcl)) return 0;

  /* Now that a window class has been registered, a window
     can be created. */
  hwnd = CreateWindow(
    wcl.lpszClassName, // name of window class
    "Using a Thread Control Panel", // title
    WS_OVERLAPPEDWINDOW, // window style - normal
    CW_USEDEFAULT, // X coordinate - let Windows decide
    CW_USEDEFAULT, // Y coordinate - let Windows decide
    260,           // width
    200,           // height
    NULL,          // no parent window
    NULL,          // no override of class menu
    hThisInst,     // instance handle
    NULL           // no additional arguments
  );
```

```
hInst = hThisInst; // save instance handle

// Load the keyboard accelerators.
hAccel = LoadAccelerators(hThisInst, "ThreadAppMenu");

// Display the window.
ShowWindow(hwnd, winMode);
UpdateWindow(hwnd);

// Create the message loop.
while(GetMessage(&msg, NULL, 0, 0))
{
  if(!TranslateAccelerator(hwnd, hAccel, &msg)) {
    TranslateMessage(&msg); // translate keyboard messages
    DispatchMessage(&msg); // return control to Windows
  }
}
  return msg.wParam;
}

/* This function is called by Windows and is passed
   messages from the message queue.
*/
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
                            WPARAM wParam, LPARAM lParam)
{
  int response;

  switch(message) {
    case WM_COMMAND:
      switch(LOWORD(wParam)) {
        case IDM_THREAD: // create the threads
          hThread1 = (HANDLE) _beginthreadex(NULL, 0,
                              MyThread1, (void *) hwnd,
                              0, &tid1);
          hThread2 = (HANDLE) _beginthreadex(NULL, 0,
                              MyThread2, (void *) hwnd,
                              0, &tid2);
          break;
        case IDM_PANEL: // activate control panel
          ThrdCtrlPanel(hInst, hThread1);
          ThrdCtrlPanel(hInst, hThread2);
          break;
        case IDM_EXIT:
          response = MessageBox(hwnd, "Quit the Program?",
                              "Exit", MB_YESNO);
          if(response == IDYES) PostQuitMessage(0);
          break;
        case IDM_HELP:
          MessageBox(hwnd,
                     "F1: Help\nF2: Start Threads\nF3: Panel",
                     "Help", MB_OK);
```

```
        break;
      }
      break;
    case WM_DESTROY: // terminate the program
      PostQuitMessage(0);
      break;
    default:
      return DefWindowProc(hwnd, message, wParam, lParam);
  }
  return 0;
}

// First thread.
unsigned __stdcall MyThread1(void * param)
{
  int i;
  HDC hdc;

  for(i=0; i<MAX; i++) {
    wsprintf(str, "Thread 1: loop # %5d ", i);
    hdc = GetDC((HWND) param);
    TextOut(hdc, 1, 1, str, lstrlen(str));
    ReleaseDC((HWND) param, hdc);
  }

  return 0;
}

// Second thread.
unsigned __stdcall MyThread2(void * param)
{
  int i;
  HDC hdc;

  for(i=0; i<MAX; i++) {
    wsprintf(str, "Thread 2: loop # %5d ", i);
    hdc = GetDC((HWND) param);
    TextOut(hdc, 1, 20, str, lstrlen(str));
    ReleaseDC((HWND) param, hdc);
  }

  return 0;
}
```

此程序需要的头文件 thrdapp.h 如下所示：

```
#define IDM_THREAD 100
#define IDM_HELP 101
#define IDM_PANEL 102
#define IDM_EXIT 103
```

程序需要的资源文件如下所示：

```
#include <windows.h>
#include "thrdapp.h"
#include "tcp.rc"

ThreadAppMenu MENU
{
  POPUP "&Threads" {
    MENUITEM "&Start Threads\tF2", IDM_THREAD
    MENUITEM "&Control Panels\tF3", IDM_PANEL
    MENUITEM "E&xit\tCtrl+X", IDM_EXIT
  }
  MENUITEM "&Help", IDM_HELP
}

ThreadAppMenu ACCELERATORS
{
  VK_F1, IDM_HELP, VIRTKEY
  VK_F2, IDM_THREAD, VIRTKEY
  VK_F3, IDM_PANEL, VIRTKEY
  "^X", IDM_EXIT
}
```

3.6 一个多线程的垃圾回收器

在开发多线程程序时，尽管通过线程控制面板来控制线程是有用的，然而最终是使用线程以使得它们重要。为此，本章给出了第 2 章开发的原始的 GCPtr 垃圾回收器类的多线程版本。第 2 章显示的 GCPtr 的版本在 GCPtr 对象超出作用域时回收不再使用的内存。尽管这种方法适合于某些应用程序，然而通常更好的选择是使得垃圾回收器作为后台任务运行，在 CPU 循环空闲的时候回收内存。这里开发的产品是为 Windows 设计的，但是相似的基本技术可以应用到其他的多线程环境。

将 GCPtr 转换为后台任务相当容易，但是会涉及到一些改变。主要有以下几点：

(1) 支持线程的成员变量必须加入到 GCPtr 中。这些变量包括线程句柄、互斥体句柄以及跟踪现有的 GCPtr 对象数量的实例计数器。

(2) GCPtr 的构造函数必须开始垃圾回收线程。另外构造函数还必须创建控制同步的互斥体。这只能在创建第一个 GCPtr 对象时发生一次。

(3) 必须定义另外的异常来指示超时情况。

(4) GCPtr 析构函数不应该再调用 collect()。垃圾回收由垃圾回收线程来处理。

(5) 必须为垃圾回收器定义一个名为 gc()的函数作为线程的入口点。

(6) 必须定义一个函数 isRunning()，如果正在使用垃圾回收，它就返回 true。

(7) 访问包含在 gclist 中的垃圾回收链表的 GCPtr 的成员函数必须被同步，从而在某个时刻只有一个线程可以访问这个链表。

下面的部分给出了这些改变。

3.6.1 附加的成员变量

GCPtr 的多线程版本要求加入如下的成员变量：

```
// These support multithreading.
unsigned tid; // thread id
static HANDLE hThrd;    // thread handle
static HANDLE hMutex;   // handle of mutex

static int instCount;   // counter of GCPtr objects
```

垃圾回收器使用的线程的 ID 存储在 tid 中。只有在调用_beginthreadex()时才会使用这个成员。这个线程的句柄存储在 hThrd 中。互斥体用来同步访问 GCPtr 的句柄存储在 hMutex 中。当前 GCPtr 对象的数量保存在 instCount 中。最后 3 个变量为静态变量，它们被 GCPtr 的所有实例共享。它们在 GCPtr 的外部定义如下：

```
template <class T, int size>
  int GCPtr<T, size>::instCount = 0;

template <class T, int size>
  HANDLE GCPtr<T, size>::hMutex = 0;

template <class T, int size>
  HANDLE GCPtr<T, size>::hThrd = 0;
```

3.6.2 多线程的 GCPtr 构造函数

除了原先的任务之外，多线程的 GCPtr()必须创建互斥体，开始垃圾回收器线程，并更新实例计数器。在此是更新了的版本：

```
// Construct both initialized and uninitialized objects.
GCPtr(T *t=NULL) {

  // When first object is created, create the mutex
  // and register shutdown().
  if(hMutex==0) {
    hMutex = CreateMutex(NULL, 0, NULL);
    atexit(shutdown);
  }

  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  list<GCInfo<T> >::iterator p;

  p = findPtrInfo(t);

  // If t is already in gclist, then
  // increment its reference count.
  // Otherwise, add it to the list.
  if(p != gclist.end())
```

```
    p->refcount++; // increment ref count
  else {
    // Create and store this entry.
    GCInfo<T> gcObj(t, size);
    gclist.push_front(gcObj);
  }

  addr = t;
  arraySize = size;
  if(size > 0) isArray = true;
  else isArray = false;

  // Increment instance counter for each new object.
  instCount++;

  // If the garbage collection thread is not
  // currently running, start it running.
  if(hThrd==0) {
    hThrd = (HANDLE) _beginthreadex(NULL, 0, gc,
            (void *) 0, 0, (unsigned *) &tid);

    // For some applications, it will be better
    // to lower the priority of the garbage collector
    // as shown here:
    //
    // SetThreadPriority(hThrd,
    //                   THREAD_PRIORITY_BELOW_NORMAL);
  }

  ReleaseMutex(hMutex);
}
```

让我们仔细分析这段代码。首先，如果 hMutex 为 0，则意味着将要创建第一个 GCPtr，并且还没有为垃圾回收器创建互斥体。如果是这种情况，则创建互斥体，将其句柄赋给 hMutex。同时，通过调用 atexit()，将函数 shutdown()注册为终止函数。

注意，在多线程的垃圾回收器中，shutdown()有两个作用。首先，如同原先版本的 GCPtr，shutdown()释放任何不再使用的但是由于循环引用而没有被释放的内存。其次，当使用了多线程的垃圾回收器结束的时候，它终止了垃圾回收线程。这意味着仍然可能存在没有被释放的动态分配的内存。这很重要，因为这些对象可能具有需要被调用的析构函数。由于 shutdown()释放所有仍然存在的对象，因此它也会释放这些对象。

随后，通过调用 WaitForSingleObject()来获取互斥体。为了阻止两个线程在同一时刻访问 gclist，这是必须的。一旦获取了互斥体，就开始搜索 gclist，查找是否存在与 t 中的地址匹配的条目。如果能够找到，其引用计数增加。如果没有先前存在的条目与 t 匹配，就创建新的 GCInfo 对象来包含这个地址，并且将这个对象加入到 gclist 中。然后设置 addr、arraySize 和 isArray。这些操作与先前的 GCPtr 的版本相同。

然后 instCount 递增。记得 instCount 是初始化为 0 的。每当有对象创建的时候，都会递增以跟踪现有的 GCPtr 对象的数目。只要这个计数大于 0，垃圾回收器就会持续运行。

随后，如果 hThrd 为 0(如同它的初始化状态)，则说明还没有为垃圾回收器创建线程。在此情况下，调用_beginthreadex()来开始这个线程。这个线程的句柄被赋给 hThrd。这个线程的入口函数称为 gc()，对它会进行简短的检查。

最后，释放互斥体，返回构造函数。有必要指出，每一个对 WaitForSingleObject()的调用都必须通过调用 ReleaseMutex()来平衡，在 GCPtr 的构造函数中显示了这一点。释放互斥体失败会引发死锁。

3.6.3 TimeOutExc 异常

在前面部分对 GCPtr()代码的描述中，您可能已经发现，当 10 秒钟之后仍然不能获取互斥体时，就会抛出 TimeOutExc 异常。坦白的说，10 秒种是一个非常长的时间，因此除非操作系统的任务调度程序被破坏，否则不会发生超时。然而，在确实发生超时的情况下，您的应用程序代码可能想要捕捉这个异常。TimeOutExc 类如下所示：

```
// Exception thrown when a time-out occurs
// when waiting for access to hMutex.
//
class TimeOutExc {
  // Add functionality if needed by your application.
};
```

注意，它没有包含数字。在本章，它作为一个独特的类型存在就足够了。当然，如果需要，可以添加功能。

3.6.4 多线程的 GCPtr 析构函数

不同于单线程版本的 GCPtr 的析构函数，~GCPtr()的多线程版本没有调用 collect()。取而代之的是，它只是简单地递减超出作用域的 GCPtr 所指向内存的引用计数。实际的垃圾回收(如果存在的话)由垃圾回收线程处理。析构函数同时还递减实例计数器 instCount 的值。

多线程版本的~GCPtr()如下所示：

```
// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~GCPtr() {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  list<GCInfo<T> >::iterator p;

  p = findPtrInfo(addr);
  if(p->refcount) p->refcount--; // decrement ref count

  // Decrement instance counter for each object
  // that is destroyed.
  instCount--;

  ReleaseMutex(hMutex);
}
```

3.6.5 gc()函数

垃圾回收器的入口函数称为gc()，如下所示：

```
// Entry point for garbage collector thread.
template <class T, int size>
unsigned __stdcall GCPtr<T, size>::gc(void * param) {
  #ifdef DISPLAY
    cout << "Garbage collection started.\n";
  #endif

  while(isRunning()) {
     collect();
  }

  collect(); // collect garbage on way out

  // Release and reset the thread handle so
  // that the garbage collection thread can
  // be restarted if necessary.
  CloseHandle(hThrd);
  hThrd = 0;

  #ifdef DISPLAY
    cout << "Garbage collection terminated for "
         << typeid(T).name() << "\n";
  #endif

  return 0;
}
```

gc()函数非常简单：只要使用垃圾回收器它就会运行。如果 instCount 大于 0，isRunning()函数就返回true(这意味着仍然需要垃圾回收器)，否则，就返回false。在循环内连续调用collect()。为了说明多线程垃圾回收器的运行，这种方法是恰当的，但是对于实际的应用，它可能效率太低了。您可能想对 collect()的调用不那么频繁，例如只有在内存变少时才调用它。您还可以在每次调用 collect()之后调用 Windows API 函数 Sleep()。Sleep()将调用的线程暂停执行指定的毫秒数。当睡眠时，线程不会消耗 CPU 时间。

当isRunning()返回值为 false 时，循环结束，导致 gc()最终结束，从而终止了垃圾回收线程。由于多线程，虽然 isRunning()返回了 false，但是在 gclist 中可能仍然存在没有被释放的条目。为了处理这种情况，在 gc()结束之前，最后一次调用 collect()。

最后，通过调用 Windows API 函数CloseHandle()释放了线程句柄，其值被设置为0。将 hThrd 设置为 0 可以使得后来在程序中创建新的 GCPtr 对象时，GCPtr 构造函数重新启动这个线程。

3.6.6 isRunning()函数

isRunning()函数如下所示：

```
// Returns true if the collector is still in use.
static bool isRunning() { return instCount > 0; }
```

它只是简单地将 instCount 与 0 进行比较。只要 instCount 大于 0，则当前至少存在一个 GCPtr 指针，仍然需要垃圾回收器。

3.6.7 gclist 的同步访问

GCPtr 中的许多函数访问了 gclist，gclist 拥有垃圾回收链表。对 gclist 的访问必须同步，以阻止在同一时刻两个或者多个线程试图使用它。原因很容易理解。如果访问没有同步，那么某个线程可能从这个链表的结尾获取了一个迭代器，同时另一个线程增加或者删除了这个链表的元素。在此情况下，这个迭代器变得无效。为了阻止这个问题，互斥体必须准许每一个访问 gclist 的代码序列。在此给出了 GCPtr 的复制构造函数的一个示例：

```
// Copy constructor.
GCPtr(const GCPtr &ob) {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  list<GCInfo<T> >::iterator p;

  p = findPtrInfo(ob.addr);
  p->refcount++; // increment ref count

  addr = ob.addr;
  arraySize = ob.arraySize;
  if(arraySize > 0) isArray = true;
  else isArray = false;

  instCount++; // increase instance count for copy

  ReleaseMutex(hMutex);
}
```

注意复制构造函数做的第一件事是获取互斥体。然后创建这个对象的副本，并调整指向内存的引用计数。当它结束时，复制构造函数释放互斥体。相同的基本方法适用于所有访问 gclist 的函数。

3.6.8 其他两个改变

必须对原先的垃圾回收器做另外两个修改。首先，原始版本的 GCPtr 定义了静态变量 first，来指示创建第一个 GCPtr 的时刻。由于 hMutex 执行了这个功能，因此这个变量不再需要，从而从 GCPtr 中删除 first。因为它是一个静态变量，所以还需要在 GCPtr 外部删除它的定义。

在原先单线程版本的垃圾回收器中，如果您定义了 DISPLAY 宏，就可以观察垃圾回收器的运转状态。在多线程版本中，这些代码的大部分都被删除了，因为在大多数情况下，多线程使得输出混乱而难以理解。对于多线程的版本，定义 DISPLAY 宏只能让您知道垃圾回收器的开始和结束时间。

3.6.9 完整的多线程垃圾回收器

多线程垃圾回收器的完整版本如下所示。这个文件称为 gcthrd.h。

```
// A garbage collector that runs as a background task.

#include <iostream>
```

```
#include <list>
#include <typeinfo>
#include <cstdlib>
#include <windows.h>
#include <process.h>

using namespace std;

// To watch the action of the garbage collector, define DISPLAY.
// #define DISPLAY

// Exception thrown when an attempt is made to
// use an Iter that exceeds the range of the
// underlying object.
//
class OutOfRangeExc {
  // Add functionality if needed by your application.
};

// Exception thrown when a time-out occurs
// when waiting for access to hMutex.
//
class TimeOutExc {
  // Add functionality if needed by your application.
};

// An iterator-like class for cycling through arrays
// that are pointed to by GCPtrs. Iter pointers
// ** do not ** participate in or affect garbage
// collection. Thus, an Iter pointing to
// some object does not prevent that object
// from being recycled.
//
template <class T> class Iter {
  T *ptr; // current pointer value
  T *end; // points to element one past end
  T *begin; // points to start of allocated array
  unsigned length; // length of sequence
public:

  Iter() {
    ptr = end = begin = NULL;
    length = 0;
  }

  Iter(T *p, T *first, T *last) {
    ptr = p;
    end = last;
    begin = first;
    length = last - first;
  }
```

```
// Return length of sequence to which this
// Iter points.
unsigned size() { return length; }

// Return value pointed to by ptr.
// Do not allow out-of-bounds access.
T &operator*() {
  if( (ptr >= end) || (ptr < begin) )
    throw OutOfRangeExc();
  return *ptr;
}

// Return address contained in ptr.
// Do not allow out-of-bounds access.
T *operator->() {
  if( (ptr >= end) || (ptr < begin) )
    throw OutOfRangeExc();
  return ptr;
}

// Prefix ++.
Iter operator++() {
ptr++;
return *this;
}

// Prefix --.
Iter operator--() {
  ptr--;
  return *this;
}

// Postfix ++.
Iter operator++(int notused) {
T *tmp = ptr;
  ptr++;
  return Iter<T>(tmp, begin, end);
}

// Postfix --.
Iter operator--(int notused) {
  T *tmp = ptr;

  ptr--;
  return Iter<T>(tmp, begin, end);
}

// Return a reference to the object at the
// specified index. Do not allow out-of-bounds
// access.
T &operator[](int i) {
  if( (i < 0) || (i >= (end-begin)) )
```

```
    throw OutOfRangeExc();
  return ptr[i];
}

// Define the relational operators.
bool operator==(Iter op2) {
  return ptr == op2.ptr;
}

bool operator!=(Iter op2) {
return ptr != op2.ptr;
}

bool operator<(Iter op2) {
  return ptr < op2.ptr;
}

bool operator<=(Iter op2) {
  return ptr <= op2.ptr;
}

bool operator>(Iter op2) {
  return ptr > op2.ptr;
}

bool operator>=(Iter op2) {
  return ptr >= op2.ptr;
}

// Subtract an integer from an Iter.
Iter operator-(int n) {
  ptr -= n;
  return *this;
}

// Add an integer to an Iter.
Iter operator+(int n) {
  ptr += n;
  return *this;
}

  // Return number of elements between two Iters.
  int operator-(Iter<T> &itr2) {
    return ptr - itr2.ptr;
  }
};

// This class defines an element that is stored
// in the garbage collection information list.
//
template <class T> class GCInfo {
public:
```

```
  unsigned refcount; // current reference count

  T *memPtr; // pointer to allocated memory

  /* isArray is true if memPtr points
     to an allocated array. It is false
     otherwise. */
  bool isArray; // true if pointing to array

  /* If memPtr is pointing to an allocated
     array, then arraySize contains its size */
  unsigned arraySize; // size of array

  // Here, mPtr points to the allocated memory.
  // If this is an array, then size specifies
  // the size of the array.
  GCInfo(T *mPtr, unsigned size=0) {
    refcount = 1;
    memPtr = mPtr;
    if(size != 0)
      isArray = true;
    else
      isArray = false;

    arraySize = size;
  }
};

// Overloading operator== allows GCInfos to be compared.
// This is needed by the STL list class.
template <class T> bool operator==(const GCInfo<T> &ob1,
                  const GCInfo<T> &ob2) {
  return (ob1.memPtr == ob2.memPtr);
}


// GCPtr implements a pointer type that uses
// garbage collection to release unused memory.
// A GCPtr must only be used to point to memory
// that was dynamically allocated using new.
// When used to refer to an allocated array,
// specify the array size.
//
template <class T, int size=0> class GCPtr {

  // gclist maintains the garbage collection list.
  static list<GCInfo<T> > gclist;

  // addr points to the allocated memory to which
  // this GCPtr pointer currently points.
  T *addr;
```

```
  /* isArray is true if this GCPtr points
     to an allocated array. It is false
     otherwise. */
  bool isArray; // true if pointing to array

  // If this GCPtr is pointing to an allocated
  // array, then arraySize contains its size.
  unsigned arraySize; // size of the array

  // These support multithreading.
  unsigned tid; // thread id
  static HANDLE hThrd; // thread handle
  static HANDLE hMutex; // handle of mutex
  static int instCount; // counter of GCPtr objects

  // Return an iterator to pointer info in gclist.
  typename list<GCInfo<T> >::iterator findPtrInfo(T *ptr);

public:

  // Define an iterator type for GCPtr<T>.
  typedef Iter<T> GCiterator;

  // Construct both initialized and uninitialized objects.
  GCPtr(T *t=NULL) {

    // When first object is created, create the mutex
    // and register shutdown().
    if(hMutex==0) {
      hMutex = CreateMutex(NULL, 0, NULL);
      atexit(shutdown);
    }

    if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
      throw TimeOutExc();

    list<GCInfo<T> >::iterator p;

    p = findPtrInfo(t);

    // If t is already in gclist, then
    // increment its reference count.
    // Otherwise, add it to the list.
    if(p != gclist.end())
      p->refcount++; // increment ref count
    else {
      // Create and store this entry.
      GCInfo<T> gcObj(t, size);
      gclist.push_front(gcObj);
    }

    addr = t;
```

```
  arraySize = size;
  if(size > 0) isArray = true;
  else isArray = false;

  // Increment instance counter for each new object.
  instCount++;
  // If the garbage collection thread is not
  // currently running, start it running.
  if(hThrd==0) {
    hThrd = (HANDLE) _beginthreadex(NULL, 0, gc,
            (void *) 0, 0, (unsigned *) &tid);

    // For some applications, it will be better
    // to lower the priority of the garbage collector
    // as shown here:
    //
    // SetThreadPriority(hThrd,
    //                   THREAD_PRIORITY_BELOW_NORMAL);
  }

  ReleaseMutex(hMutex);
}

// Copy constructor.
GCPtr(const GCPtr &ob) {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  list<GCInfo<T> >::iterator p;

  p = findPtrInfo(ob.addr);
  p->refcount++; // increment ref count

  addr = ob.addr;
  arraySize = ob.arraySize;
  if(arraySize > 0) isArray = true;
  else isArray = false;

  instCount++; // increase instance count for copy

  ReleaseMutex(hMutex);
}

// Destructor for GCPTr.
~GCPtr();

// Collect garbage. Returns true if at least
// one object was freed.
static bool collect();

// Overload assignment of pointer to GCPtr.
T *operator=(T *t);
```

```
// Overload assignment of GCPtr to GCPtr.
GCPtr &operator=(GCPtr &rv);

// Return a reference to the object pointed
// to by this GCPtr.
T &operator*() {
  return *addr;
}

// Return the address being pointed to.
T *operator->() { return addr; }

// Return a reference to the object at the
// index specified by i.
T &operator[](int i) {
  return addr[i];
}

// Conversion function to T *.
operator T *() { return addr; }

// Return an Iter to the start of the allocated memory.
Iter<T> begin() {
  int size;

  if(isArray) size = arraySize;
  else size = 1;

  return Iter<T>(addr, addr, addr + size);
}

// Return an Iter to one past the end of an allocated array.
Iter<T> end() {
  int size;

  if(isArray) size = arraySize;
  else size = 1;

  return Iter<T>(addr + size, addr, addr + size);
}

// Return the size of gclist for this type
// of GCPtr.
static int gclistSize() {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  unsigned sz = gclist.size();

  ReleaseMutex(hMutex);
  return sz;
```

```
  }

  // A utility function that displays gclist.
  static void showlist();

  // The following functions support multithreading.
  //
  // Returns true if the collector is still in use.
  static bool isRunning() { return instCount > 0; }

  // Clear gclist when program exits.
  static void shutdown();

  // Entry point for garbage collector thread.
  static unsigned __stdcall gc(void * param);
};

// Create storage for the static variables.
template <class T, int size>
  list<GCInfo<T> > GCPtr<T, size>::gclist;

template <class T, int size>
  int GCPtr<T, size>::instCount = 0;

template <class T, int size>
  HANDLE GCPtr<T, size>::hMutex = 0;

template <class T, int size>
  HANDLE GCPtr<T, size>::hThrd = 0;

// Destructor for GCPtr.
template <class T, int size>
GCPtr<T, size>::~GCPtr() {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  list<GCInfo<T> >::iterator p;
  p = findPtrInfo(addr);
  if(p->refcount) p->refcount--; // decrement ref count

  // Decrement instance counter for each object
  // that is destroyed.
  instCount--;

  ReleaseMutex(hMutex);
}

// Collect garbage. Returns true if at least
// one object was freed.
template <class T, int size>
bool GCPtr<T, size>::collect() {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
```

```
    throw TimeOutExc();

  bool memfreed = false;

  list<GCInfo<T> >::iterator p;
  do {

    // Scan gclist looking for unreferenced pointers.
    for(p = gclist.begin(); p != gclist.end(); p++) {
      // If in-use, skip.
      if(p->refcount > 0) continue;

      memfreed = true;

      // Remove unused entry from gclist.
      gclist.remove(*p);

      // Free memory unless the GCPtr is null.
      if(p->memPtr) {
        if(p->isArray) {
          delete[] p->memPtr; // delete array
        }
        else {
          delete p->memPtr; // delete single element
        }
      }

      // Restart the search.
      break;
    }
  } while(p != gclist.end());

  ReleaseMutex(hMutex);

  return memfreed;
}

// Overload assignment of pointer to GCPtr.
template <class T, int size>
T * GCPtr<T, size>::operator=(T *t) {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  list<GCInfo<T> >::iterator p;

  // First, decrement the reference count
  // for the memory currently being pointed to.
  p = findPtrInfo(addr);
  p->refcount--;

  // Next, if the new address is already
  // existent in the system, increment its
```

```
  // count. Otherwise, create a new entry
  // for gclist.
  p = findPtrInfo(t);
  if(p != gclist.end())
    p->refcount++;
  else {
    // Create and store this entry.
    GCInfo<T> gcObj(t, size);
    gclist.push_front(gcObj);
  }

  addr = t; // store the address.

  ReleaseMutex(hMutex);

  return t;
}

// Overload assignment of GCPtr to GCPtr.
template <class T, int size>
GCPtr<T, size> & GCPtr<T, size>::operator=(GCPtr &rv) {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();
  list<GCInfo<T> >::iterator p;

  // First, decrement the reference count
  // for the memory currently being pointed to.
  p = findPtrInfo(addr);
  p->refcount--;

  // Next, increment the reference count
  // of the new object.
  p = findPtrInfo(rv.addr);
  p->refcount++; // increment ref count

  addr = rv.addr;// store the address.

  ReleaseMutex(hMutex);

  return rv;
}

// A utility function that displays gclist.
template <class T, int size>
void GCPtr<T, size>::showlist() {
  if(WaitForSingleObject(hMutex, 10000)==WAIT_TIMEOUT)
    throw TimeOutExc();

  list<GCInfo<T> >::iterator p;

  cout << "gclist<" << typeid(T).name() << ", "
       << size << ">:\n";
```

```
  cout << "memPtr refcount value\n";

  if(gclist.begin() == gclist.end()) {
    cout << " -- Empty --\n\n";
    return;
  }

  for(p = gclist.begin(); p != gclist.end(); p++) {
    cout << "[" << (void *)p->memPtr << "]"
         << " " << p->refcount << " ";
    if(p->memPtr) cout << " " << *p->memPtr;
    else cout << " ---";
    cout << endl;
  }
  cout << endl;

  ReleaseMutex(hMutex);
}

// Find a pointer in gclist.
template <class T, int size>
typename list<GCInfo<T> >::iterator
  GCPtr<T, size>::findPtrInfo(T *ptr) {

  list<GCInfo<T> >::iterator p;

  // Find ptr in gclist.
  for(p = gclist.begin(); p != gclist.end(); p++)
    if(p->memPtr == ptr)
      return p;

  return p;
}

// Entry point for garbage collector thread.
template <class T, int size>
unsigned __stdcall GCPtr<T, size>::gc(void * param) {
  #ifdef DISPLAY
    cout << "Garbage collection started.\n";
  #endif

  while(isRunning()) {
     collect();
  }

  collect(); // collect garbage on way out

  // Release and reset the thread handle so
  // that the garbage collection thread can
  // be restarted if necessary.
  CloseHandle(hThrd);
  hThrd = 0;
```

```
    #ifdef DISPLAY
      cout << "Garbage collection terminated for "
           << typeid(T).name() << "\n";
    #endif

    return 0;
  }

// Clear gclist when program exits.
template <class T, int size>
void GCPtr<T, size>::shutdown() {

  if(gclistSize() == 0) return; // list is empty

  list<GCInfo<T> >::iterator p;

  #ifdef DISPLAY
    cout << "Before collecting for shutdown() for "
         << typeid(T).name() << "\n";
  #endif

  for(p = gclist.begin(); p != gclist.end(); p++) {
    // Set all remaining reference counts to zero.
    p->refcount = 0;
  }

  collect();

  #ifdef DISPLAY
  cout << "After collecting for shutdown() for "
       << typeid(T).name() << "\n";
  #endif
}
```

3.6.10 多线程垃圾回收器的使用

为了使用多线程的垃圾回收器，需要在您的程序中包含 gcthrd.h。然后，以第 2 章描述的方式使用 GCPtr。当编译这个程序时，必须要记住链接多线程库，如同本章前面对_beginthreadex()和_endthreadex()的描述中所讲述的那样。

为了观察多线程垃圾回收器的效果，尝试原先在第 2 章中给出的这个版本的加载测试程序：

```
// Demonstrate the multithreaded garbage collector.
#include <iostream>
#include <new>
#include "gcthrd.h"

using namespace std;

// A simple class for load testing GCPtr.
class LoadTest {
```

```
  int a, b;
public:
  double n[100000]; // just to take-up memory
  double val;

  LoadTest() { a = b = 0; }

  LoadTest(int x, int y) {
    a = x;
    b = y;
    val = 0.0;
  }

  friend ostream &operator<<(ostream &strm, LoadTest &obj);
};

// Create an inserter for LoadTest.
ostream &operator<<(ostream &strm, LoadTest &obj) {
  strm << "(" << obj.a << " " << obj.b << ")";
  return strm;
}

int main() {
  GCPtr<LoadTest> mp;
  int i;

  for(i = 1; i < 2000; i++) {
    try {
      mp = new LoadTest(i, i);
      if(!(i%100))
        cout << "gclist contains " << mp.gclistSize()
             << " entries.\n";
    } catch(bad_alloc xa) {
      // For most users, this exception won't
      // ever occur.
      cout << "Last object: " << *mp << endl;
      cout << "Length of gclist: "
           << mp.gclistSize() << endl;
    }
  }

  return 0;
}
```

在此有一个运行的样本。(当然，您的输出可能会不同)。在 gcthrd.h 中定义了 DISPLAY 宏并开启显示选项时，输出如下：

```
Garbage collection started.
gclist contains 42 entries.
gclist contains 35 entries.
gclist contains 29 entries.
gclist contains 22 entries.
```

```
gclist contains 18 entries.
gclist contains 11 entries.
gclist contains 4 entries.
gclist contains 51 entries.
gclist contains 47 entries.
gclist contains 40 entries.
gclist contains 33 entries.
gclist contains 26 entries.
gclist contains 19 entries.
gclist contains 15 entries.
gclist contains 10 entries.
gclist contains 3 entries.
gclist contains 53 entries.
gclist contains 46 entries.
gclist contains 42 entries.
Before collecting for shutdown() for class LoadTest
After collecting for shutdown() for class LoadTest
```

正如您所看到的那样，由于 collect()在后台运行，虽然有成千个对象被分配并丢弃，但是 gclist 也不会非常大。

3.7 试着完成下面的任务

创建一个成功的多线程程序具有相当的挑战性。原因之一是多线程要求您用并行的而不是线性的观点来考虑程序。另外，在运行时，线程交互的方式通常难以预测。因此，您可能会对多线程程序的操作感到惊奇(甚至迷惑)。掌握多线程的最好方法是使用它。为此，您或许会尝试这些想法。

试着在线程控制面板内添加另外一个列表框，以使得用户除了调整线程的优先级值之外，还可以调整线程的优先级类别。试着在控制面板内加入不同的同步对象，这些对象可以在用户的控制下开启或者关闭。这将会让您体验不同的同步选择。

对于多线程的垃圾回收器，试着减少垃圾回收的次数，如 gclist 达到某个大小时或者在自由内存低于某个预定值时才进行回收。作为另一种选择，可以使用可等待计时器有规律地激活垃圾回收。最后，您可以试验不同的垃圾回收器的优先级类别和设置来找到最适合您使用的优先级。