第二十章　多工和多执行绪

多工是一个作业系统可以同时执行多个程式的能力。基本上，作业系统使用一个硬体时钟为同时执行的每个程序配置「时间片段」。如果时间片段够小，并且机器也没有由於太多的程式而超出负荷时，那么在使用者看来，所有的这些程式似乎在同时执行著。

多工并不是什么新的东西。在大型电脑上，多工是必然的。这些大型主机通常有几十甚至几百个终端机和它连结，而每个终端机使用者都应该感觉到他或者她独占了整个电脑。另外，大型主机的作业系统通常允许使用者「提交工作到背景」，这些背景作业可以在使用者进行其他工作时，由机器执行完成。

个人电脑上的多工花了更长的时间才普及化。但是现在 PC 多工也被认为是很正常的了。我马上就会讨论到，Microsoft Windows 的 16 位元版本支援有限度的多工，Windows 的 32 位元版本支援真正的多工，而且，还多了一种额外的优点，多执行绪。

多执行绪是在一个程式内部实作多工的能力。程式可以把它自己分隔为各自独立的「执行绪」，这些执行绪似乎也同时在执行著。这一概念初看起来似乎没有什么用处，但是它可以让程式使用多执行绪在背景执行冗长作业，从而让使用者不必长时间地无法使用其电脑进行其他工作（有时这也许不是人们所希望的，不过这种时候去冲冲凉或者到冰箱去看看总是很不错的）！但是，即使在电脑繁忙的时候，使用者也应该能够使用它。

多工的各种模式

在 PC 的早期，有人曾经提倡未来应该朝多工的方向前进，但是大多数的人还是很迷惑：在一个单使用者的个人电脑上，多工有什么用呢？好了，最後事实表示即使是不知道这一概念的使用者也都需要多工的。

DOS 下的多工

在最初 PC 上的 Intel 8088 微处理器并不是为多工而设计的。部分原因（我在上一章中讨论过）是记忆体管理不够强。当启动和结束多个程式时，多工的作业系统通常需要移动记忆体块以收集空闲记忆体。在 8088 上是不可能透明於应用系统来做到这一点的。

DOS 本身对多工没有太大的帮助，它的设计目的是尽可能小巧，并且与独立於应用程式之外，因此，除了载入程式以及对程式提供档案系统的存取功能，

它几乎没有提供任何支援。

不过，有创意的程式写作者仍然在 DOS 的早期就找到了一种克服这些缺陷的方法，大多数是使用常驻（TSR：terminate-and-stay-resident）程式。有些 TSR，比如背景列印伫列程式等，透过拦截硬体时钟中断来执行真正的背景处理。其他的 TSR，诸如 SideKick 等突现式工具，可以执行某种型态的工作切换——暂停目前的应用程式，执行突现式工具。DOS 也逐渐有所增强以便提供对 TSR 的支援。

一些软体厂商试图在 DOS 之上架构出工作切换或者多工的外壳程式(shell)（诸如 Quarterdeck 的 DesqView），但是在这些环境中，仅有其中一个占据了大部分市场，当然，这就是 Windows。

非优先权式的多工

当 Microsoft 在 1985 年发表 Windows 1.0 时，它是最成熟的解决方案，目的是突破 DOS 的局限。Windows 在实际模式下执行。但是即使这样，它已可以在实体记忆体中移动记忆体块。这是多工的前提，虽然移动的方法尚未完全透明於应用程式，但是几乎可以忍受了。

在图形视窗环境中，多工比在一种命令列单使用者作业系统中显得更有意义。例如，在传统的命令列 UNIX 中，可以在命令列之外执行程式，让它们在背景执行。然而，程式的所有显示输出必须被重新转向到一个档案中，否则输出将和使用者正在做的事情混在一起。

视窗环境允许多个程式在相同萤幕上一起执行，前後切换非常容易，并且还可以快速地将资料从一个程式移动到另一个程式中。例如，将绘图程式中建立的图片嵌入由文书处理程式编辑的文字档案中。在 Windows 中，以多种方式支援资料转移，首先是使用剪贴簿，後来又使用动态资料交换（DDE），而现在则是透过物件连结和嵌入（OLE）。

不过，早期 Windows 的多工实作还不是多使用者作业系统中传统的优先权式的分时多工。这些作业系统使用系统时钟周期性地中断一个工作并开始另一个工作。Windows 的这些 16 位元版本支援一种被称为「非优先权式的多工」，由於 Windows 讯息驱动的架构而使这种型态的多工成为可能。通常情况下，一个 Windows 程式将在记忆体中睡眠，直到它收到一个讯息为止。这些讯息通常是使用者的键盘或滑鼠输入的直接或间接结果。当处理完讯息之後，程式将控制权返回给 Windows。

Windows 的 16 位元版本不会绝对地依据一个 timer tick 将控制权从一个 Windows 程式切换到另一个，任何的工作切换都发生在当程式完成对讯息的处理

後将控制权返回给 Windows 时。这种非优先权式的多工也被称为「合作式的多工」，因为它要求来自应用程式方面的一些合作。一个 Windows 程式可以占用整个系统，如果它要花很长一段时间来处理讯息的话。

虽然非优先权式的多工是 16 位元 Windows 的一般规则，但仍然出现了某些形式的优先权式多工。Windows 使用优先权式多工来执行 DOS 程式，而且，为了实作多媒体，还允许动态连结程式库接收硬体时钟中断。

16 位元 Windows 包括几个功能特性来帮助程式写作者解决（或者，至少可以说是对付）非优先权式多工中的局限，最显著的当然是时钟式滑鼠游标。当然，这并非一种解决方案，而仅仅是让使用者知道一个程式正在忙於处理一件冗长作业，因而让使用者在一段时间内无法使用系统。另一种解决方案是 Windows 计时器，它允许程式周期性地接收讯息并完成一些工作。计时器通常用於时钟应用和动画。

针对非优先权式多工的另一种解决方案是 PeekMessage 函式呼叫，我们曾在第五章中的 RANDRECT 程式里看到过。一个程式通常使用 GetMessage 呼叫从它的讯息伫列中找寻下一个讯息，不过，如果在讯息伫列中没有讯息，那么 GetMessage 不会传回，一直到出现一个讯息为止。而另一方面，PeekMessage 将控制权传回程式，即使没有等待的讯息。这样，一个程式可以执行一个冗长作业，并在程式码中混入 PeekMessage 呼叫。只要没有这个程式或其他任何程式的讯息要处理，那么这个冗长作业将继续执行。

Presentation Manager 和序列化的讯息伫列

Microsoft 在一种半 DOS/半 Windows 的环境下实作多工的第一个尝试（和 IBM 合作）是 OS/2 和 Presentation Manager（缩写成 PM ）。虽然 OS/2 明确地支援优先权式多工，但是这种多工方式似乎并未在 Presentation Manager 中得以落实。问题在於 PM 序列化来自键盘和滑鼠的使用者输入讯息。这意味著，在前一个使用者输入讯息被完全处理以前，PM 不会将一个键盘或者滑鼠讯息传送给程式。

尽管键盘和滑鼠讯息只是一个 PM（或者 Windows）程式可以接收的许多讯息中的几个，大多数的其他讯息都是键盘或者滑鼠事件的结果。例如，功能表命令讯息是使用者使用键盘或者滑鼠进行功能表选择的结果。在处理功能表命令讯息时，键盘或者滑鼠讯息并未完全被处理。

序列化讯息伫列的主要原因是允许使用者的预先「键入」键盘按键和预先「按入」滑鼠按钮。如果一个键盘或者滑鼠讯息导致输入焦点从一个视窗切换到另一个视窗，那么接下来的键盘讯息应该进入拥有新的输入焦点的视窗中去。

因此，系统不知道将下一个使用者输入讯息发送到何处，直到前一个讯息被处理完为止。

目前的共识是不应该让一个应用系统有可能占用整个系统，而这需要非序列化的讯息伫列，32 位元版本的 Windows 支援这种讯息伫列。如果一个程式正在忙著处理一项冗长作业，那么您可以将输入焦点切换到另一个程式中。

多执行绪解决方案

我讨论 OS/2 的 Presentation Manager，只是因为它是第一个为早期的 Windows 程式写作者（比如我自己）介绍多执行绪的环境。有趣的是，PM 实作多执行绪的局限为程式写作者提供了应该如何架构多执行绪程式的必要线索。即使这些限制在 32 位元的 Windows 中已经大幅减少，但是从更有限的环境中学到的经验仍然是非常有效的。因此，让我们继续讨论下去。

在一个多执行绪环境中，程式可以将它们自己分隔为同时执行的片段（叫做执行绪）。对执行绪的支援是解决 PM 中存在的序列化讯息伫列的最好方法，并且在 Windows 中执行绪有更实际的意义。

就程式码来说，一个执行绪简单地被表示为可能呼叫程式中其他函式的函式。程式从其主执行绪开始执行，这个主执行绪是在传统的 C 程式中叫做 main 的函式，而在 Windows 中是 WinMain。一旦执行起来，程式可以通过在系统呼叫 CreateThread 中指定初始执行绪函式的名称来建立新的执行绪的执行。作业系统在执行绪之间优先权式地切换控制项，和它在程序之间切换控制权的方法非常类似。

在 OS/2 的 Presentation Manager 中，每个执行绪可以建立一个讯息伫列，也可以不建立。如果希望从执行绪建立视窗，那么一个 PM 执行绪必须建立讯息伫列。否则，如果只是进行许多的资料处理或者图形输出，那么执行绪不需要建立讯息伫列。因为无讯息伫列的程序不处理讯息，所以它们将不会当住系统。唯一的限制是一个无讯息伫列执行绪无法向一个讯息伫列执行绪中的视窗发送讯息，或者呼叫任何发送讯息的函式（不过，它们可以将讯息递送给讯息伫列执行绪）。

这样，PM 程式写作者学会了如何将它们的程式分隔为一个讯息伫列执行绪（在其中建立所有的视窗并处理传送给视窗的讯息）和一个或者多个无讯息伫列执行绪，在其中执行冗长的背景工作。PM 程式写作者还了解到「1/10 秒规则」，大体上，程式写作者被告知，一个讯息伫列执行绪处理任何讯息都不应该超过 1/10 秒，任何花费更长时间的事情都应该在另一个执行绪中完成。如果所有的程式写作者都遵循这一规则，那么将没有 PM 程式会将系统当住超过 1/10 秒。

多执行绪架构

我已经说过 PM 的限制让程式写作者理解如何在图形环境中执行的程式里头使用多个执行绪提供了必要的线索。因此在这里我将为您的程式建议一种架构：您的主执行绪建立您程式所需要的所有视窗，并在其中包含所有的视窗讯息处理程式，以便处理这些视窗的所有讯息；所有其他执行绪只进行一些背景处理，除了和主执行绪通讯，它们不和使用者进行交流。

可以把这种架构想像成：主执行绪处理使用者输入（和其他讯息），并建立程序中的其他执行绪，这些附加的执行绪完成与使用者无关的工作。

换句话说，您程式的主执行绪是一个老板，而您的其他执行绪是老板的职员。老板将大的工作丢给职员处理，而他自己保持和外界的联系。因为那些执行绪仅仅是职员，所以其他执行绪不会举行它们自己的记者招待会。它们会认真地完成自己的工作，将结果报告给老板，并等待他们的下一个任务。

一个程式中的执行绪是同一程序的不同部分，因此他们共用程序的资源，如记忆体和打开的档案。因为执行绪共用程式的记忆体，所以他们还共用静态变数。然而，每个执行绪都有他们自己的堆叠，因此动态变数对每个执行绪是唯一的。每个执行绪还有各自的处理器状态（和数学辅助运算器状态），这个状态在进行执行绪切换期间被储存和恢复。

执行绪间的「争吵」

正确地设计、写作和测试一个复杂的多执行绪应用程式显然是 Windows 程式写作者可能遇到的最困难的工作之一。因为优先权式多工系统可以在任何时刻中断一个执行绪，并将控制权切换到另一个执行绪中，在两个执行绪之间可能有无法预料的随机交互作用的情况。

多执行绪程式中的一个常见的错误被称为「竞争状态（race condition）」，这发生在程式写作者假设一个执行绪在另一个执行绪需要某资料之前已经完成了某些处理（如准备资料）的时候。为了帮助协调执行绪的活动，作业系统要求各种形式的同步。一种是同步信号（semaphore），它允许程式写作者在程式码中的某一点阻止一个执行绪的执行，直到另一个执行绪发信号让它继续为止。类似於同步信号的是「临界区域（critical section）」，它是程式码中不可中断的部分。

但是同步信号还可能产生称为「锁死（deadlock）」的常见执行绪错误，这发生在两个执行绪互相阻止了另一个的执行，而继续执行的唯一办法又是它们继续向前执行。

幸运的是，32 位元程式比 16 位元程式更能抵抗执行绪所涉及的某些问题。例如，假定一个执行绪执行下面的简单叙述：

```
lCount++ ;
```

其中 lCount 是由其他执行绪使用的一个 32 位元的 long 型态变数，C 中的这个叙述被编译为两条机械码指令，第一条将变数的低 16 位元加 1，而第二条指令将任何可能的进位加到高 16 位上。假定作业系统在这两个机械码指令之间中断了执行绪。如果 lCount 在第一条机械码指令之前是 0x0000FFFF，那么 lCount 在执行绪被中断时为 0，而这正是另一个执行绪将看到的值。只有当执行绪继续执行时，lCount 才会增加到正确的值 0x00010000。

这是那些偶尔会导致操作问题的错误之一。在 16 位元程式中，解决此问题正确的方法是将叙述包含在一个临界区域中，在这期间执行绪不会被中断。然而，在一个 32 位元程式中，该叙述是正确的，因为它被编译为一条机械码指令。

Windows 的好处

32 位元 Windows 版本（包括 Windows NT 和 Windows 98）有一个非序列化的讯息伫列。这种实作似乎非常好：如果一个程式正在花费一段长时间处理一个讯息，那么滑鼠位於该程式的视窗上时，滑鼠游标将呈现为一个时钟，但是当将滑鼠移到另一个程式的视窗上时，滑鼠游标将变为正常的箭头形状。只需按一下就可以将另一个视窗提到前面来。

然而，使用者仍然不能使用正在处理大量工作的那个程式，因为那些工作会阻止程式接收其他讯息，这不是我们所希望的。一个程式应该总是能随时处理讯息的，所以这时就需要使用从属执行绪了。

在 Windows NT 和 Windows 98 中，没有讯息伫列执行绪和无讯息伫列执行绪的区别，每个执行绪在建立时都会有它自己的讯息伫列，从而减少了 PM 程式中关於执行绪的一些不便规定（然而，在大多数情况下，您仍然想通过一条专门处理讯息的执行绪中的讯息程序处理输入，而将冗长作业交给那些不包含视窗的执行绪处理，这种结构几乎总是最容易理解的，我们将看到这一点）。

还有更好的事情：Windows NT 和 Windows 98 中有个函式允许执行绪杀死同一程序中的另一个执行绪。当您开始编写多执行绪程式码时，您将会发现这种功能在有时是很方便的。OS/2 的早期版本没有「杀死执行绪」的函式。

最後的好讯息（至少对这里的话题是好讯息）是 Windows NT 和 Windows 98 实作了一些被称为「执行绪区域储存空间（TLS：thread local storage）」的功能。为了了解这一点，回顾一下我在前面提到过的，静态变数（对一个函式来说，既是整体又是区域变数）在执行绪之间是被共用的，因为它们位於程序

的资料储存空间中。动态变数（对一个函式来说总是区域变数）对每一个执行绪则是唯一的，因为它们占据堆叠上的空间，而每个执行绪都有它自己的堆叠。

有时让两个或多个执行绪使用相同的函式，而让这些执行绪使用唯一於执行绪的静态变数，那会带来很大便利。这就是执行绪区域储存空间，其中涉及一些 Windows 函式呼叫，但是 Microsoft 还为 C 编译器进行扩展，使执行绪区域储存空间的使用更透明于程式写作者。

新改良过的！支援多执行绪了！

既然已经介绍了执行绪的现状，让我们来展望一下执行绪的未来。有时，有人会出现一种使用作业系统所提供的每一种功能特性的冲动。最坏的情况是，当您的老板走到您的桌前并说：「我听说这种新功能非常炫，让我们在自己的程式中用一些这种新功能吧。」然後您将花费一个星期的时间，试图去了解您的应用程式如何从这种新功能获益。

应该注意的是，在并不需要多执行绪的应用系统中加入多执行绪是没有任何意义的。如果您的程式显示沙漏游标的时间太长，或者如果它使用 PeekMessage 呼叫来避免沙漏游标的出现，那么请重新规划您的程式架构，使用多执行绪可能会是一个好主意。其他情形，您是在为难您自己，并可能会在程式码中产生新的错误。

在某些情况下，沙漏游标的出现可能是完全适当的。我在前面提到过「1/10 秒规则」，而将一个大档案载入记忆体可能会花费多於 1/10 秒的时间，这是否意味著档案载入常式应该在分离的执行绪中实作呢？没有必要。当使用者命令一个程式打开档案时，他或者她通常想立即完成该操作。将档案载入常式放在分离的执行绪中只会增加额外的负担。即使您想向您的朋友夸耀您在编写多执行绪程式，也完全不值得这样做！

WINDOWS 的多执行绪处理

建立新的执行绪的 API 函式是 CreateThread，它的语法如下：

```
hThread = CreateThread (&security_attributes, dwStackSize, ThreadProc,
                        pParam, dwFlags, &idThread) ;
```

第一个参数是指向 SECURITY_ATTRIBUTES 型态的结构的指标。在 Windows 98 中忽略该参数。在 Windows NT 中，它被设为 NULL。第二个参数是用於新执行绪的初始堆叠大小，预设值为 0。在任何情况下，Windows 根据需要动态延长堆叠的大小。

CreateThread 的第三个参数是指向执行绪函式的指标。函式名称没有限制，

但是必须以下列形式宣告：

```
DWORD WINAPI ThreadProc (PVOID pParam) ;
```

CreateThread 的第四个参数为传递给 ThreadProc 的参数。这样主执行绪和从属执行绪就可以共用资料。

CreateThread 的第五个参数通常为 0，但当建立的执行绪不马上执行时为旗标 CREATE_SUSPENDED。执行绪将暂停直到呼叫 ResumeThread 来恢复执行绪的执行为止。第六个参数是一个指标，指向接受执行绪 ID 值的变数。

大多数 Windows 程式写作者喜欢用在 PROCESS.H 表头档案中宣告的 C 执行时期程式库函式_beginthread。它的语法如下：

```
hThread = _beginthread (ThreadProc, uiStackSize, pParam) ;
```

它更简单，对於大多数应用程式很完美，这个执行绪函式的语法为：

```
void __cdecl ThreadProc (void * pParam) ;
```

再论随机矩形

程式 20-1 RNDRCTMT 是第五章里的 RANDRECT 程式的多执行绪版本，您将回忆起 RANDRECT 使用的是 PeekMessage 回圈来显示一系列的随机矩形。

程式 20-1　RNDRCTMT

```
RNDRCTMT.C
/*------------------------------------------------------------------------
-
      RNDRCTMT.C --     Displays Random Rectangles
                                                (c) Charles Petzold, 1998

------------------------------------------------------------------------*/

#include <windows.h>
#include <process.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
HWND  hwnd ;
int   cxClient, cyClient ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
      static TCHAR      szAppName[] = TEXT ("RndRctMT") ;
      MSG         msg ;
      WNDCLASS          wndclass ;

      wndclass.style                    = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc              = WndProc ;
      wndclass.cbClsExtra               = 0 ;
```

```
     wndclass.cbWndExtra              = 0 ;
     wndclass.hInstance               = hInstance ;
     wndclass.hIcon                   = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor                 = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground      = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName       = NULL ;
     wndclass.lpszClassName      = szAppName ;

     if (!RegisterClass (&wndclass))
     {
      MessageBox (    NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
               return 0 ;
    }
     hwnd = CreateWindow (  szAppName, TEXT ("Random Rectangles"),
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
               TranslateMessage (&msg) ;
               DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

VOID Thread (PVOID pvoid)
{
     HBRUSH     hBrush ;
     HDC             hdc ;
     int             xLeft, xRight, yTop, yBottom, iRed, iGreen, iBlue ;

     while (TRUE)
    {
               if (cxClient != 0 || cyClient != 0)
               {
                         xLeft           = rand () % cxClient ;
                         xRight          = rand () % cxClient ;
                         yTop       = rand () % cyClient ;
                         yBottom    = rand () % cyClient ;
                         iRed            = rand () & 255 ;
                         iGreen          = rand () & 255 ;
                         iBlue           = rand () & 255 ;
```

```
                        hdc = GetDC (hwnd) ;
                        hBrush = CreateSolidBrush (RGB (iRed, iGreen, iBlue)) ;
                        SelectObject (hdc, hBrush) ;

                    Rectangle (hdc,  min (xLeft, xRight), min (yTop, yBottom),
              max (xLeft, xRight), max (yTop, yBottom)) ;

                                    ReleaseDC (hwnd, hdc) ;
                                    DeleteObject (hBrush) ;
                        }
      }
}
LRESULT CALLBACK WndProc (   HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      switch (message)
      {
      case WM_CREATE:
                        _beginthread (Thread, 0, NULL) ;
                        return 0 ;

      case  WM_SIZE:
                        cxClient = LOWORD (lParam) ;
                        cyClient = HIWORD (lParam) ;
                        return 0 ;

      case  WM_DESTROY:
                        PostQuitMessage (0) ;
                        return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

在建立多执行绪的 Windows 程式时，需要在「Project Settings」对话方块中做一些修改。选择「C/C++」页面标签，然後在「Category」下拉式清单方块中选择「Code Generation」。在「Use Run-Time Library」下拉式清单方块中，可以看到用於「Release」设定的「Single-Threaded」和用於 Debug 设定的「Debug Single-Threaded」。将这些分别改为「Multithreaded」和「Debug Multithreaded」。这将把编译器旗标改为/MT，它是编译器在编译多执行绪的应用程式所需要的。具体地说，编译器将在.OBJ 档案中插入 LIBCMT.LIB 档案名，而不是 LIBC.LIB。连结程式使用这个名称与执行期程式库函式连结。

LIBC.LIB 和 LIBCMT.LIB 档案包含 C 语言程式库函式，有些 C 语言程式库函式包含静态资料。例如，由於 strtok 函式可能被连续地多次呼叫，所以它在静态记忆体中储存了一个指标。在多执行绪程式中，每个执行绪必须在 strtok 函

式中有它自己的静态指标。因此，这个函式的多执行绪版本稍微不同於单执行绪的 strtok 函式。

同时请注意，我在 RNDRCTMT.C 中包含了表头档案 PROCESS.H，这个档案定义一个名为_beginthread 的函式，它启动一个新的执行绪。只有定义了_MT 识别字，才会宣告这个函式，这是/MT 旗标的另一个结果。

在 RNDRCTMT.C 的 WinMain 函式中，由 CreateWindow 传回的 hwnd 值被储存在一个整体变数中，因此 cxClient 和 cyClient 值也可以由视窗讯息处理程式的 WM_SIZE 讯息获得。

视窗讯息处理程式以最容易的方法呼叫_beginthread——简单地以执行绪函式的位址（称为 Thread）作为第一个参数，其他参数使用 0，执行绪函式传回 VOID 并有一个参数,该参数是一个指向 VOID 的指标。在 RNDRCTMT 中的 Thread 函式不使用这个参数。

在呼叫了_beginthread 函式之後，执行绪函式（以及该执行绪函式可能呼叫的其他任何函式）中的程式码和程式中的其他程式码同时执行。两个或者多个执行绪使用一个程序中的同一函式，在这种情况下，动态区域变数（储存在堆叠上）对每个执行绪是唯一的。对程序中的所有执行绪来说，所有的静态变数都是一样的。这就是视窗讯息处理程式设定整体的 cxClient 和 cyClient 变数并由 Thread 函式使用的方式。

有时您需要唯一於各个执行绪的持续储存性资料。通常，这种资料是静态变数，但在 Windows 98 中，您可以使用「执行绪区域储存空间」，我将在本章後面进行讨论。

程式设计竞赛的问题

1986 年 10 月 3 日，Microsoft 举行了为期一天，针对电脑杂志出版社的技术编辑和作者的简短的记者招待会，来讨论他们当时的一组语言产品，包括他们的第一个交谈式开发环境，QuickBASIC 2.0。当时，Windows 1.0 出现还不到一年，但是没有人知道我们什么时候能得到与该环境类似的东西（这花了好几年）。这一事件与众不同的部分原因是由於 Microsoft 的公关人员所举办的「Storm the Gates」程式设计竞赛。Bill Gates 使用 QuickBASIC 2.0，而电脑出版社的人员可以使用他们选择的任何语言产品。

竞赛的问题是从公众提出的题目中挑选出来的（挑选那些需要写大约半小时程式来解决的问题），问题如下：

建立一个包含四个视窗的多工模拟程式。第一个视窗必须显示一系列的递增数，第二个必须显示一系列的递增质数，而第三个必须显示 Fibonacci 数列

（Fibonacci 数列以数字 0 和 1 开始，後头每一个数都是其前两个数的和——即 0、1、1、2、3、5、8 等等）。这三个视窗应该在数字达到视窗底部时或者进行滚动，或者自行清除视窗内容。第四个视窗必须显示任意半径的圆，而程式必须在按下一个 Escape 键时终止。

当然，在 1986 年 10 月，在 DOS 下执行的这样一个程式最多只能是模拟多工而已，而且没有一个竞赛者具有足够的勇气——并且其中大多数也没有足够的知识——来为 Windows 编写这个程式。再者，如果真要这么做，当然不会只花半小时了！

参加这次竞赛的大多数人编写了一个程式来将萤幕分为四个区域，程式中包含一个回圈，依次更新每个视窗，然後检查是否按下了 Escape 键。如同 DOS 环境下的传统习惯，程式占用了百分之百的 CPU 处理时间。

如果在 Windows 1.0 中写程式，那么结果将是类似程式 20-2 MULTI1 的结果。我说「类似」，是因为我编写的程式是 32 位元的，但程式结构和相当多的程式码——除了变数和函式参数定义以及 Unicode 支援——都是相同的。

程式 20-2 MULTI1

```
MULTI1.C
/*------------------------------------------------------------------------
      MULTI1.C --      Multitasking Demo
                                                (c) Charles Petzold, 1998
------------------------------------------------------------------------
*/

#include <windows.h>
#include <math.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int cyChar ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Multi1") ;
      HWND                              hwnd ;
      MSG                         msg ;
      WNDCLASS                    wndclass ;

      wndclass.style              = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc        = WndProc ;
      wndclass.cbClsExtra         = 0 ;
      wndclass.cbWndExtra         = 0 ;
      wndclass.hInstance          = hInstance ;
      wndclass.hIcon              = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor            = LoadCursor (NULL, IDC_ARROW) ;
```

```
      wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName  = NULL ;
      wndclass.lpszClassName = szAppName ;

      if (!RegisterClass (&wndclass))
      {
       MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                  szAppName, MB_ICONERROR) ;
                  return 0 ;
    }
    hwnd = CreateWindow (    szAppName, TEXT ("Multitasking Demo"),
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL) ;

      ShowWindow (hwnd, iCmdShow) ;
      UpdateWindow (hwnd) ;

      while (GetMessage (&msg, NULL, 0, 0))
      {
                  TranslateMessage (&msg) ;
                  DispatchMessage (&msg) ;
      }
      return msg.wParam ;
}

int CheckBottom (HWND hwnd, int cyClient, int iLine)
{
      if (iLine * cyChar + cyChar > cyClient)
      {
                  InvalidateRect (hwnd, NULL, TRUE) ;
                  UpdateWindow (hwnd) ;
                  iLine = 0 ;
    }
      return iLine ;
}

// ------------------------------------------------------------------------
// Window 1: Display increasing sequence of numbers
// ------------------------------------------------------------------------

LRESULT APIENTRY WndProc1 ( HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      static int       iNum, iLine, cyClient ;
      HDC                           hdc ;
      TCHAR                         szBuffer[16] ;
```

```
     switch (message)
     {
     case  WM_SIZE:
                       cyClient = HIWORD (lParam) ;
                       return 0 ;

     case  WM_TIMER:
                       if (iNum < 0)
                                        iNum = 0 ;

                       iLine = CheckBottom (hwnd, cyClient, iLine) ;
                       hdc = GetDC (hwnd) ;

                       TextOut (hdc, 0, iLine * cyChar, szBuffer,
                                  wsprintf (szBuffer, TEXT ("%d"), iNum++)) ;

                       ReleaseDC (hwnd, hdc) ;
                       iLine++ ;
                       return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

// ------------------------------------------------------------------------
// Window 2: Display increasing sequence of prime numbers
// ------------------------------------------------------------------------

LRESULT APIENTRY WndProc2 (  HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static int       iNum = 1, iLine, cyClient ;
     HDC                         hdc ;
     int                         i, iSqrt ;
     TCHAR                       szBuffer[16] ;

     switch (message)
     {
     case  WM_SIZE:
                       cyClient = HIWORD (lParam) ;
                       return 0 ;

     case  WM_TIMER:
                       do          {
                                        if (++iNum < 0)
                                                  iNum = 0 ;

                                        iSqrt = (int) sqrt (iNum) ;
```

```
                                        for (i = 2 ; i <= iSqrt ; i++)
                                                if (iNum % i == 0)
                                                        break ;
                        }
                        while (i <= iSqrt) ;

                        iLine = CheckBottom (hwnd, cyClient, iLine) ;
                        hdc = GetDC (hwnd) ;

                        TextOut (  hdc, 0, iLine * cyChar, szBuffer,
                                   wsprintf (szBuffer, TEXT ("%d"), iNum)) ;
                        ReleaseDC (hwnd, hdc) ;
                        iLine++ ;
                        return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}

// ------------------------------------------------------------------------
// Window 3: Display increasing sequence of Fibonacci numbers
// ------------------------------------------------------------------------

LRESULT APIENTRY WndProc3 ( HWND hwnd, UINT message, WPARAM wParam,LPARAM
lParam)
{
      static int       iNum = 0, iNext = 1, iLine, cyClient ;
      HDC                        hdc ;
      int                        iTemp ;
      TCHAR                      szBuffer[16] ;

      switch (message)
      {
      case WM_SIZE:
                        cyClient = HIWORD (lParam) ;
                        return 0 ;

      case  WM_TIMER:
                        if (iNum < 0)
                        {
                                   iNum  = 0 ;
                                   iNext = 1 ;
                        }

                        iLine = CheckBottom (hwnd, cyClient, iLine) ;
                        hdc = GetDC (hwnd) ;

                        TextOut (  hdc, 0, iLine * cyChar, szBuffer,
```

```
                                                    wsprintf (szBuffer, "%d",
iNum)) ;
                        ReleaseDC (hwnd, hdc) ;
                        iTemp       =     iNum ;
                        iNum        =     iNext ;
                        iNex        +=    iTemp ;
                        iLine++ ;
                        return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}

// ---------------------------------------------------------------------------
// Window 4: Display circles of random radii
//
---------------------------------------------------------------------------

LRESULT APIENTRY WndProc4 ( HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      static int       cxClient, cyClient ;
      HDC                        hdc ;
      int                        iDiameter ;

      switch (message)
      {
      case  WM_SIZE:
                        cxClient = LOWORD (lParam) ;
                        cyClient = HIWORD (lParam) ;
                        return 0 ;

      case  WM_TIMER:
                        InvalidateRect (hwnd, NULL, TRUE) ;
                        UpdateWindow (hwnd) ;

                        iDiameter = rand() % (max (1, min (cxClient, cyClient))) ;
                        hdc = GetDC (hwnd) ;

                        Ellipse (hdc,          (cxClient - iDiameter) / 2,
                                   (cyClient - iDiameter) / 2,
                                   (cxClient + iDiameter) / 2,
                                   (cyClient + iDiameter) / 2) ;

                        ReleaseDC (hwnd, hdc) ;
                        return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

```
// ------------------------------------------------------------------------
// Main window to create child windows
// ------------------------------------------------------------------------

LRESULT APIENTRY WndProc (   HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      static HWND            hwndChild[4] ;
      static TCHAR *   szChildClass[] = { TEXT ("Child1"), TEXT ("Child2"),
       TEXT ("Child3"), TEXT ("Child4") } ;
      static WNDPROC         ChildProc[] =    { WndProc1, WndProc2, WndProc3,
WndProc4 } ;
      HINSTANCE                    hInstance ;
      int                                i, cxClient, cyClient ;
      WNDCLASS                        wndclass ;

      switch (message)
      {
      case  WM_CREATE:
                  hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

                        wndclass.style              = CS_HREDRAW | CS_VREDRAW ;
                        wndclass.cbClsExtra         = 0 ;
                        wndclass.cbWndExtra         = 0 ;
                        wndclass.hInstance          = hInstance ;
                        wndclass.hIcon              = NULL ;
                        wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
                        wndclass.hbrBackground =    (HBRUSH)    GetStockObject
(WHITE_BRUSH) ;
                        wndclass.lpszMenuName  = NULL ;

                        for (i = 0 ; i < 4 ; i++)
                        {
                                    wndclass.lpfnWndProc   = ChildProc[i] ;
                                    wndclass.lpszClassName = szChildClass[i] ;

                                    RegisterClass (&wndclass) ;

                        hwndChild[i] = CreateWindow (szChildClass[i], NULL,
                            WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
                            0, 0, 0, 0,
                            hwnd, (HMENU) i, hInstance, NULL) ;
                        }

                        cyChar = HIWORD (GetDialogBaseUnits ()) ;
                        SetTimer (hwnd, 1, 10, NULL) ;
                        return 0 ;
```

```
     case  WM_SIZE:
                    cxClient = LOWORD (lParam) ;
                    cyClient = HIWORD (lParam) ;

                    for (i = 0 ; i < 4 ; i++)
               MoveWindow (hwndChild[i],    (i % 2) * cxClient / 2,
                                  (i > 1) * cyClient / 2,
                         cxClient / 2, cyClient / 2, TRUE) ;
                               return 0 ;

     case  WM_TIMER:
                for (i = 0 ; i < 4 ; i++)
                      SendMessage (hwndChild[i], WM_TIMER, wParam, lParam) ;

                      return 0 ;

     case  WM_CHAR:
                      if (wParam == '\x1B')
                                        DestroyWindow (hwnd) ;

                      return 0 ;

     case  WM_DESTROY:
                      KillTimer (hwnd, 1) ;
                      PostQuitMessage (0) ;
                      return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

在这个程式里实际上没有什么我们没见过的东西。主视窗建立四个子视窗，每个子视窗占据显示区域的一个象限。主视窗还设定一个 Windows 计时器并发送 WM_TIMER 讯息给四个子视窗中的每一个。

通常一个 Windows 程式应该保留足够的资讯以便在 WM_PAINT 讯息处理期间重建其视窗中的内容。MULTI1 没有这么做，既然它绘制和清除视窗的速度如此之快，所以我认为那是不必要的。

WndProc2 中的质数产生器的效率并不很高，但是有效。如果一个数除了 1 和它自身以外没有别的因数，那么这个数就是质数。当然，要检查一个数是否是质数并不要求使用小於被检查数的所有数来除这个数并检查余数，而只需使用所有小於被检查数的平方根的数。平方根计算是发表浮点数的原因，否则，该程式将是完全依据整数的程式。

MULTI1 程式没有什么不好的地方。使用 Windows 计时器是在 Windows 的早期（和目前）版本中模拟多工的一种好方法，然而，计时器的使用有时限制了

程式的速度。如果程式可以在 WM_TIMER 讯息处理中更新它的所有视窗而还有时间剩余下来的话，那就意味著它并没有充分利用我们的机器资源。

一种可能的解决方案是在单个 WM_TIMER 讯息处理期间进行两次或者更多次的更新，但是到底多少次呢？这不得不依赖於机器的速度，而有很大的变动性。您当然不会想编写一个只能适用於 25MHz 的 386 或 50MHz 的 486 或 100-GHz 的 Pentium VII 上的程式吧。

多执行绪解决方案

让我们来看一看关於这个程式设计问题的一种多执行绪解决方案。如程式 20-3 MULTI2 所示。

程式 20-3 MULTI2

```
MULTI2.C
/*-------------------------------------------------------------------------
-
      MULTI2.C --      Multitasking Demo
                                                (c) Charles Petzold, 1998
-------------------------------------------------------------------------
*/

#include <windows.h>
#include <math.h>
#include <process.h>

typedef struct
{
      HWND        hwnd ;
      int         cxClient ;
      int         cyClient ;
      int         cyChar ;
      BOOL bKill ;
}
PARAMS, *PPARAMS ;
LRESULT APIENTRY WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("Multi2") ;
      HWND                              hwnd ;
      MSG                         msg ;
      WNDCLASS                    wndclass ;

      wndclass.style                    = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc              = WndProc ;
```

```
     wndclass.cbClsExtra                = 0 ;
     wndclass.cbWndExtra                = 0 ;
     wndclass.hInstance                 = hInstance ;
     wndclass.hIcon                     = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor                   = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground       = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName        = NULL ;
     wndclass.lpszClassName       = szAppName ;

     if (!RegisterClass (&wndclass))
     {
      MessageBox (NULL, TEXT ("This program requires Windows NT!"),
                szAppName, MB_ICONERROR) ;
                return 0 ;
     }

     hwnd = CreateWindow (  szAppName, TEXT ("Multitasking Demo"),
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      CW_USEDEFAULT, CW_USEDEFAULT,
                      NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

int CheckBottom (HWND hwnd, int cyClient, int cyChar, int iLine)
{
     if (iLine * cyChar + cyChar > cyClient)
     {
                InvalidateRect (hwnd, NULL, TRUE) ;
                UpdateWindow (hwnd) ;
                iLine = 0 ;
     }
     return iLine ;
}

//---------------------------------------------------------------------------
// Window 1: Display increasing sequence of numbers
//---------------------------------------------------------------------------
```

```
void Thread1 (PVOID pvoid)
{
      HDC                   hdc ;
      int                   iNum = 0, iLine = 0 ;
      PPARAMS          pparams ;
      TCHAR                 szBuffer[16] ;

      pparams     = (PPARAMS) pvoid ;

      while (!pparams->bKill)
      {
                  if (iNum < 0)
                                    iNum = 0 ;
                  iLine = CheckBottom (  pparams->hwnd,
      pparams->cyClient,
   pparams->cyChar,    iLine) ;

                  hdc = GetDC (pparams->hwnd) ;

                  TextOut (   hdc, 0, iLine * pparams->cyChar, szBuffer,
                                    wsprintf (szBuffer, TEXT ("%d"), iNum++)) ;

                  ReleaseDC (pparams->hwnd, hdc) ;
                  iLine++ ;
      }
      _endthread () ;
}

LRESULT APIENTRY WndProc1 (  HWND  hwnd,  UINT  message,  WPARAM  wParam,  LPARAM
lParam)
{
      static PARAMS params ;
      switch (message)
      {
      case  WM_CREATE:
                        params.hwnd = hwnd ;
                        params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
                        _beginthread (Thread1, 0, 秭 s) ;
                        return 0 ;

      case  WM_SIZE:
                        params.cyClient = HIWORD (lParam) ;
                        return 0 ;

      case  WM_DESTROY:
                        params.bKill = TRUE ;
                        return 0 ;
      }
```

```
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

// ---------------------------------------------------------------------------
// Window 2: Display increasing sequence of prime numbers
// ---------------------------------------------------------------------------

void Thread2 (PVOID pvoid)
{
     HDC                    hdc ;
     int                    iNum = 1, iLine = 0, i, iSqrt ;
     PPARAMS          pparams ;
     TCHAR                  szBuffer[16] ;

     pparams = (PPARAMS) pvoid ;
     while (!pparams->bKill)
     {
               do
               {
                         if (++iNum < 0)
                                   iNum = 0 ;
                         iSqrt = (int) sqrt (iNum) ;
                         for (i = 2 ; i <= iSqrt ; i++)
                                   if (iNum % i == 0)
                                             break ;
               }
               while (i <= iSqrt) ;
               iLine = CheckBottom    (     pparams->hwnd,
     pparams->cyClient,
   pparams->cyChar,    iLine) ;

               hdc = GetDC (pparams->hwnd) ;

               TextOut (  hdc, 0, iLine * pparams->cyChar, szBuffer,
                              wsprintf (szBuffer, TEXT ("%d"), iNum)) ;

               ReleaseDC (pparams->hwnd, hdc) ;
               iLine++ ;
     }
     _endthread () ;
}

LRESULT APIENTRY WndProc2 ( HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static PARAMS params ;
     switch (message)
     {
```

```
     case  WM_CREATE:
                        params.hwnd = hwnd ;
                        params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
                        _beginthread (Thread2, 0, 秏 s) ;
                        return 0 ;

     case  WM_SIZE:
                        params.cyClient = HIWORD (lParam) ;
                        return 0 ;

     case  WM_DESTROY:
                        params.bKill = TRUE ;
                        return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

// Window 3: Display increasing sequence of Fibonacci numbers
// ----------------------------------------------------------

void Thread3 (PVOID pvoid)
{
     HDC                    hdc ;
     int                    iNum = 0, iNext = 1, iLine = 0, iTemp ;
     PPARAMS          pparams ;
     TCHAR                  szBuffer[16] ;

     pparams = (PPARAMS) pvoid ;
     while (!pparams->bKill)
    {
                        if (iNum < 0)
                        {
                                   iNum  = 0 ;
                                   iNext = 1 ;
                }
                iLine = CheckBottom (  pparams->hwnd,   pparams->cyClient,
   pparams->cyChar, iLine) ;

                hdc = GetDC (pparams->hwnd) ;

                TextOut (hdc, 0, iLine * pparams->cyChar, szBuffer,
                                   wsprintf (szBuffer, TEXT ("%d"), iNum)) ;

                ReleaseDC (pparams->hwnd, hdc) ;
                iTemp  = iNum ;
                iNum   = iNext ;
                iNext += iTemp ;
                iLine++ ;
```

```
      }
      _endthread () ;
}

LRESULT APIENTRY WndProc3 ( HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      static PARAMS params ;
      switch (message)
      {
      case  WM_CREATE:
                        params.hwnd = hwnd ;
                        params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
                        _beginthread (Thread3, 0, 秏s) ;
                        return 0 ;

      case  WM_SIZE:
                        params.cyClient = HIWORD (lParam) ;
                        return 0 ;

      case  WM_DESTROY:
                        params.bKill = TRUE ;
                        return 0 ;
      }
      return DefWindowProc (hwnd, message, wParam, lParam) ;
}

// ------------------------------------------------------------------------
// Window 4: Display circles of random radii
// ------------------------------------------------------------------------

void Thread4 (PVOID pvoid)
{
      HDC                     hdc ;
      int                     iDiameter ;
      PPARAMS           pparams ;

      pparams = (PPARAMS) pvoid ;
      while (!pparams->bKill)
      {
                  InvalidateRect (pparams->hwnd, NULL, TRUE) ;
                  UpdateWindow (pparams->hwnd) ;

                  iDiameter =       rand() % (max (1,
               min (pparams->cxClient, pparams->cyClient))) ;

                  hdc = GetDC (pparams->hwnd) ;
```

```
                Ellipse (hdc,    (pparams->cxClient - iDiameter) / 2,
                                 (pparams->cyClient - iDiameter) / 2,
                                 (pparams->cxClient + iDiameter) / 2,
                                (pparams->cyClient + iDiameter) / 2) ;

                ReleaseDC (pparams->hwnd, hdc) ;
     }
     _endthread () ;
}

LRESULT APIENTRY WndProc4   (HWND  hwnd,  UINT  message,WPARAM  wParam,LPARAM
lParam)
{
     static PARAMS params ;
     switch (message)
     {
     case  WM_CREATE:
                     params.hwnd = hwnd ;
                     params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
                     _beginthread (Thread4, 0, 秏s) ;
                     return 0 ;

     case  WM_SIZE:
                     params.cxClient = LOWORD (lParam) ;
                     params.cyClient = HIWORD (lParam) ;
                     return 0 ;

     case  WM_DESTROY:
                     params.bKill = TRUE ;
                     return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}

//---------------------------------------------------------------------------
// Main window to create child windows
//---------------------------------------------------------------------------

LRESULT APIENTRY WndProc (   HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static HWND            hwndChild[4] ;
     static TCHAR * szChildClass[] = { TEXT ("Child1"), TEXT ("Child2"),
      TEXT ("Child3"), TEXT ("Child4") } ;
     static WNDPROC   ChildProc[] = { WndProc1, WndProc2, WndProc3, WndProc4 } ;
     HINSTANCE              hInstance ;
     int                              i, cxClient, cyClient ;
     WNDCLASS                   wndclass ;
```

```
switch (message)
{
case  WM_CREATE:
           hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;
                 wndclass.style               = CS_HREDRAW | CS_VREDRAW ;
                 wndclass.cbClsExtra                = 0 ;
                 wndclass.cbWndExtra                = 0 ;
                 wndclass.hInstance                 = hInstance ;
                 wndclass.hIcon                     = NULL ;
                 wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
                 wndclass.lpszMenuName  = NULL ;

                 for (i = 0 ; i < 4 ; i++)
                 {
                           wndclass.lpfnWndProc         = ChildProc[i]
                           wndclass.lpszClassName = szChildClass[i] ;

                           RegisterClass (&wndclass) ;

                 hwndChild[i] =   CreateWindow (szChildClass[i], NULL,
                  WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
                  0, 0, 0, 0,
                  hwnd, (HMENU) i, hInstance, NULL) ;
                 }

                 return 0 ;

case  WM_SIZE:
                 cxClient = LOWORD (lParam) ;
                 cyClient = HIWORD (lParam) ;

                 for (i = 0 ; i < 4 ; i++)
                       MoveWindow (hwndChild[i], (i % 2) * cxClient / 2,
                                                  (i > 1) * cyClient / 2,
                                   cxClient / 2, cyClient / 2, TRUE) ;
                 return 0 ;

case  WM_CHAR:
                 if (wParam == '\x1B')
                                  DestroyWindow (hwnd) ;

                 return 0 ;

case  WM_DESTROY:
                 PostQuitMessage (0) ;
                 return 0 ;
```

```
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

MULTI2.C 的 WinMain 和 WndProc 函式非常类似於 MULTI1.C 中的同名函式。WndProc 为四个视窗注册了四种视窗类别，建立了这些视窗，并在 WM_SIZE 讯息处理期间缩放这些视窗。WndProc 的唯一不同是它不再设定 Windows 计时器，也不再处理 WM_TIMER 讯息。

MULTI2 中较大的改变是每个子视窗讯息处理程式透过在 WM_CREATE 讯息处理期间呼叫_beginthread 函式来建立另一个执行绪。总括来说，MULTI2 程式有五个同时执行的执行绪，主执行绪包含主视窗讯息处理程式和四个子视窗讯息处理程式，其余的四个执行绪使用名为 Thread1、Thread2 等的函式，这四个执行绪负责绘制四个视窗。

我在RNDRCTMT程式中给出的多执行绪程式码没有使用_beginthread的第三个参数，这个参数允许一个建立另一个执行绪的执行绪在 32 位元变数中将资讯传递给其他执行绪。通常，这个变数是一个指标，而且是指向一个结构的指标，这允许原来的执行绪和新执行绪共用资讯，而不必借助於整体变数。您可以看到，在 MULTI2 中没有整体变数。

对 MULTI2 程式，我在程式开头定义了一个名为 PARAMS 的结构和一个名为 PPARAMS 的指向结构的指标，这个结构有五个栏位——视窗代号、视窗的宽度和高度、字元的高度和名为 bKill 的布林变数。最後的结构栏位允许建立执行绪告知被建立执行绪何时终止。

让我们来看一看 WndProc1，这是显示增加数序列的子视窗讯息处理程式。视窗讯息处理程式变得非常简单，唯一的区域变数是一个 PARAMS 结构。在 WM_CREATE 讯息处理期间，它设定这个结构的 hwnd 和 cyChar 栏位，呼叫 _beginthread 来建立一个使用 Thread1 函式的新执行绪，并传递给新执行绪一个指向该结构的指标。在 WM_SIZE 讯息处理期间，WndProc1 设定结构的 cyClient 栏位，而在 WM_DESTROY 讯息处理期间，它将 bKill 栏位设定为 TRUE。Thread1 函式通过对_endthread 的呼叫而告结束。这并不是绝对必要的，因为执行绪将在退出执行绪函式之後被清除。不过，要退出一个深陷入复杂的处理程序的执行绪时，_endthread 是很有用的。

Thread1 函式完成在视窗上的实际绘图，并且和程式的其他四个执行绪同时执行。函式接收指向 PARAMS 结构的一个指标，并进入一个 while 回圈，不断检查 bKill 是 TRUE 还是 FALSE。如果是 FALSE，那么函式必须进行 MULTI1.C 中的 WM_TIMER 讯息处理期间所作的同样处理——格式化数字、取得装置内容代号并使用 TextOut 显示数字。

当您在 Windows 98 中执行 MULTI2 时，将会看到，视窗更新要比在 MULTI1 中快得多，这表示程式在更加有效地利用处理器的资源。在 MULTI1 和 MULTI2 之间还有另一种区别：通常，当您移动或者缩放一个视窗时，内定视窗讯息处理程式进入一种模态回圈，而视窗的所有输出都将停止。在 MULTI2 中，输出将继续。

有问题吗?

似乎 MULTI2 程式并没有达到它应该有的稳固性。我为什么会这样认为呢？让我们来看一看 MULTI2.C 中的一些多执行绪「缺陷」，以 WndProc1 和 Thread1 为例。

WndProc1 在 MULTI2 的主执行绪中执行，而 Thread1 与它同时执行，Windows 98 在这两个执行绪之间进行切换是不可预测的。假定 Thread1 正在执行，并且刚好执行了检查 PARAMS 结构的 bKill 栏位是否为 TRUE 的程式码。发现不为 TRUE，但是这之後 Windows 98 将控制权切换到主执行绪，这时使用者终止了程式，WndProc1 收到一个 WM_DESTROY 讯息并将 bKill 参数设为 TRUE。哦，这参数设定得太晚了！作业系统突然切换到 Thread1 中，而该函式会试图取得一个不存在的视窗的装置内容代号。

事实证明，这不是一个问题。Windows 98 够稳固，以致另一条执行绪呼叫的图形处理函式只是失败而已，而不会引起任何问题。

正确的多执行绪程式写作技术涉及执行绪同步的使用（尤其是临界区域的使用），我将马上加以详细地讨论。大体上，临界区域通过对 EnterCriticalSection 和 LeaveCriticalSection 的呼叫而加以界定。如果一个执行绪进入一个临界区域，那么另一个执行绪将无法再进入这个临界区域。後一个执行绪被阻档在对 EnterCriticalSection 的呼叫上，直到第一个执行绪呼叫 LeaveCriticalSection 时为止。

在 MULTI2 中的另一个可能存在的问题是，当另外一个执行绪显示其输出时，主执行绪可能会收到一个 WM_ERASEBKGND 或 WM_PAINT 讯息。这里，使用临界区域有助於避免当两个程序试图在同一个视窗上绘图时可能导致的任何问题。但是，经验显示，Windows 98 很恰当地序列化了对图形绘制函式的存取。亦即，当另一个执行绪正在绘图的时候，一个执行绪不能在同一个视窗上绘图。

Windows 98 文件提醒说，有一种未进行图形函式序列化的情形，这就是 GDI 物件（如画笔、画刷、字体、点阵图、区域和调色盘等）的使用。有可能发生一个执行绪清除了一个物件，而另一个执行绪仍然在使用它的情况。解决这个问题的方法要求使用临界区域，或者最好不要在执行绪之间共用 GDI 物件。

Sleep 的好处

我曾经提到，我认为对一个多执行绪程式来说，最好的架构是主执行绪建立程式中的所有视窗，以及所有的视窗讯息处理程式，并处理所有的视窗讯息。其他执行绪完成背景工作或者冗长作业。

不过，假设您想在另一个执行绪中做动画。通常，Windows 中的动画是使用 WM_TIMER 讯息来实作的。如果这个执行绪没有建立视窗，那么它也不会收到这些讯息。如果没有计时器，动画又可能会执行得太快。

解决方案是 Sleep 函式。实际上，执行绪呼叫 Sleep 函式来自动暂停执行，该函式唯一一个参数是以毫秒计的时间。Sleep 函式呼叫在指定的时间过去以前不会传回控制权。在这段时间内，执行绪被暂停，并且不会被配置给时间片段（尽管该执行绪显然仍然要求在 tick 时给予一小段的处理时间，因为系统必须确定执行绪是否应该重新开始执行）。给 Sleep 一个值为 0 的参数将导致执行绪交回它尚未使用完的时间片段。

当一个执行绪呼叫 Sleep 时，只是该执行绪被暂停指定的时间。系统仍然执行其他的执行绪，这些执行绪和暂停的执行绪可以是在同一个程序中，也可以是在另一个程序中。我在第十四章中的 SCRAMBLE 程式中使用了 Sleep 函式，以放慢画面清除的操作。

通常，您不应该在您的主执行绪中使用 Sleep 函式，因为这会减慢对讯息的处理速度，但是因为 SCRAMBLE 没有建立任何视窗，因此在那里使用 Sleep 应该没有问题。

执行绪同步

大约每年一次，在我公寓窗外的交通繁忙地段的红绿灯会停止工作。结果是造成交通的混乱，虽然轿车一般能避免撞上别的轿车，但是这些车经常挤在一起。

我用术语称两条路相交的十字路口为「临界区域」。一辆向南的车和一辆向西的车不可能同时通过一个十字路口而不撞著对方。依赖於交通流量，可以采用不同的方法来解决这个问题。对於视野清楚车辆稀少的路口，可以相信司机有处理的能力。车辆增多可能会要求一个停车号志，而更加繁忙的交通则将要求有红绿灯，红绿灯有助於协调路口的交通（当然，这些灯号必须正常工作）。

临界区域

在单工作业系统中，传统的电脑程式不需要红绿灯来帮助协调它们之间的

行为。它们在执行时似乎独占了整条路，而且也确实是这样，没有什么会干扰它们的工作。

即使在多工作业系统中，大多数的程式也似乎各自独立地在执行，但是可能会发生一些问题。例如，两个程式可能会需要同时从同一个档案中读或者对同一档案进行写。在这种情况下，作业系统提供了一种共用档案和记录上锁的技术来帮助解决这个问题。

然而，在支援多执行绪的作业系统中，情况会变得混乱而且存在潜在的危险。两个或多个执行绪共用某些资料的情况并不罕见。例如，一个执行绪可以更新一个或者多个变数，而另一个执行绪可以使用这些变数。有时这会引发一个问题，有时又不会（记住作业系统将控制权从一个执行绪切换到另一个执行绪的操作，只能在机器码指令之间发生。如果只是一个整数被执行绪共用，那么对这个变数的改变通常发生在单个指令中，因此潜在的问题被最小化了）。

然而，假设执行绪共用几个变数或者资料结构。通常，这么多个变数或者结构的栏位在它们之间必须是一致的。作业系统可以在更新这些变数的程序中间中断一个执行绪，那么使用这些变数的执行绪得到的将是不一致的资料。

结果是冲突发生了，并且通常不难想像这样的错误将对程式造成怎样的破坏。我们所需要的是类似於红绿灯的程式写作技术，以帮助我们对执行绪交通进行协调和同步，这就是临界区域。大体上，一个临界区域就是一块不可中断的程式码。

有四个函式用於临界区域。要使用这些函式，您必须定义一个临界区域物件，这是一个型态为 CRITICAL_SECTION 的整体变数。例如：

```
CRITICAL_SECTION cs ;
```

这个 CRITICAL_SECTION 资料型态是一个结构，但是其中的栏位只能由 Windows 内部使用。这个临界区域物件必须先被程式中的某个执行绪初始化，通过呼叫：

```
InitializeCriticalSection (&cs) ;
```

这样就建立了一个名为 cs 的临界区域物件。该函式的线上辅助说明包含下面的警告：「临界区域物件不能被移动或者复制，程序也不能修改该物件，但必须在逻辑上把它视为不透明的。」这句话，可以被解释为：「不要干扰它，甚至不要看它。」

当临界区域物件被初始化之後，执行绪可以通过下面的呼叫进入临界区域：

```
EnterCriticalSection (&cs) ;
```

在这时，执行绪被认为「拥有」临界区域物件。两个执行绪不可以同时拥有同一个临界区域物件，因此，如果一个执行绪进入了临界区域，那么下一个使用同一临界区域物件呼叫 EnterCriticalSection 的执行绪将在函式呼叫中被

暂停。只有当第一个执行绪通过下面的呼叫离开临界区域时，函式才会传回控制权：

```
LeaveCriticalSection (&cs) ;
```

这时，在 EnterCriticalSection 呼叫中被停住的那个执行绪将拥有临界区域，其函式呼叫也将传回，允许执行绪继续执行。

当临界区域不再被程式所需要时，可以通过呼叫

```
DeleteCriticalSection (&cs) ;
```

将其删除，该函式释放所有被配置来维护此临界区域物件的系统资源。

这种临界区域技术涉及「互斥」（此术语在我们继续讨论执行绪同步时将再次出现）。在任何时刻，只有一个执行绪能拥有一个临界区域。因此，一个执行绪可以进入一个临界区域，设定一个结构的栏位，然後退出临界区域。另一个使用该结构的执行绪在存取结构中的栏位之前也要先进入该临界区域，然後再退出临界区域。

注意，您可以定义多个临界区域物件，比如 cs1 和 cs2。例如，如果一个程式有四个执行绪，而前两个执行绪共用一些资料，那么它们可以使用一个临界区域物件，而另外两个执行绪共用一些其他的资料，那么它们可以使用另一个临界区域物件。

您在主执行绪中使用临界区域时应该小心。如果从属执行绪在它自己的临界区域中花费了一段很长的时间，那么它可能会将主执行绪的执行阻碍很长一段时间。从属执行绪可能只是使用临界区域复制该结构的栏位到自己的区域变数中。

临界区域的一个限制是它们只能用於在同一程序内的执行绪之间的协调。但是在某些情况下，您需要协调两个不同程序对同一资源的共用（如共用记忆体等）。在此其况下不能使用临界区域，但是可以使用一种被称为「互斥物件（mutex object）」的技术。「mutex」是个合成字，代表「mutual exclusion（互斥）」，它在这里精确地表达了我们的目的。我们想防止一个程式的执行绪在更新资料或者使用共用记忆体与其他资源时被中断。

事件信号

多执行绪通常是用於那些必须执行长时间处理的程式。我们可以将一个「大作业」定义为一个可能会违反 1/10 秒规则的程式。显然大作业包括文书处理程式中的拼写检查、资料库程式中的档案排序或者索引、试算表的重新计算、列印，甚至包括复杂的绘图。当然，迄今为止我们知道，遵循 1/10 秒规则的最好方法是将大作业放到另一个执行绪去执行。这些额外的执行绪不会建立视窗，因此它们不受 1/10 秒规则的限制。

通常希望这些额外的执行绪在完成其任务时能够通知主执行绪，或者主执行绪能够停止其他执行绪正在进行的作业。这就是我们下面将要讨论的。

BIGJOB1 程式

作为一个想像的大作业，我将使用一系列浮点运算，有时这种运算被称为「暴力的」性能测试指标。这种计算以一种间接的方式递增一个整数的值：它求一个数的平方，再对结果取平方根（得到原来的整数），然後使用 log 和 exp 函式（同样得到原来的整数），接著使用 atan 和 tan 函式（还是得到原来的整数），最後对结果加 1。

BIGJOB1 程式如程式 20-4 所示。

程式 20-4　BIGJOB1

```
BIGJOB1.C
/*-------------------------------------------------------------------------
-
      BIGJOB1.C -- Multithreading Demo
                                                (c) Charles Petzold, 1998
-------------------------------------------------------------------------
*/

#include <windows.h>
#include <math.h>
#include <process.h>

#define REP                                          1000000

#define STATUS_READY                    0
#define STATUS_WORKING                1
#define STATUS_DONE                     2

#define WM_CALC_DONE               (WM_USER + 0)
#define WM_CALC_ABORTED            (WM_USER + 1)

typedef struct
{
      HWND hwnd ;
      BOOL bContinue ;
}
PARAMS, *PPARAMS ;
LRESULT APIENTRY WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                szCmdLine, int iCmdShow)
{
      static TCHAR           szAppName[] = TEXT ("BigJob1") ;
```

```
     HWND                                    hwnd ;
     MSG                                msg ;
     WNDCLASS                           wndclass ;
     wndclass.style               = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc         = WndProc ;
     wndclass.cbClsExtra          = 0 ;
     wndclass.cbWndExtra          = 0 ;
     wndclass.hInstance           = hInstance ;
     wndclass.hIcon               = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;

     if (!RegisterClass (&wndclass))
     {
      MessageBox (    NULL, TEXT ("This program requires Windows NT!"),
                      szAppName, MB_ICONERROR) ;
                return 0 ;
    }

     hwnd = CreateWindow (  szAppName, TEXT ("Multithreading Demo"),
                            WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void Thread (PVOID pvoid)
{
     double           A = 1.0 ;
     INT                    i ;
     LONG                   lTime ;
     volatile         PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;
     lTime = GetCurrentTime () ;
     for (i = 0 ; i < REP && pparams->bContinue ; i++)
```

```
                              A = tan (atan (exp (log (sqrt (A * A))))) + 1.0 ;
     if (i == REP)
     {
                lTime = GetCurrentTime () - lTime ;
                SendMessage (pparams->hwnd, WM_CALC_DONE, 0, lTime) ;
     }
     else
                SendMessage (pparams->hwnd, WM_CALC_ABORTED, 0, 0) ;
     _endthread () ;
}

LRESULT CALLBACK WndProc (   HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static     INT         iStatus ;
     static     LONG        lTime ;
     static     PARAMS      params ;
     static          TCHAR *     szMessage[] = { TEXT ("Ready (left mouse button
begins)"),
                TEXT ("Working (right mouse button ends)"),
                TEXT ("%d repetitions in %ld msec") } ;
     HDC                                   hdc ;
     PAINTSTRUCT                      ps ;
     RECT                                  rect ;
     TCHAR                                 szBuffer[64] ;

     switch (message)
     {
     case  WM_LBUTTONDOWN:
                    if (iStatus == STATUS_WORKING)
                    {
                               MessageBeep (0) ;
                               return 0 ;
                    }

                    iStatus = STATUS_WORKING ;

                    params.hwnd = hwnd ;
                    params.bContinue = TRUE ;

                    _beginthread (Thread, 0, 秏s) ;

                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;

     case  WM_RBUTTONDOWN:
                    params.bContinue = FALSE ;
                    return 0 ;
```

```
     case  WM_CALC_DONE:
                         lTime = lParam ;
                         iStatus = STATUS_DONE ;
                         InvalidateRect (hwnd, NULL, TRUE) ;
                         return 0 ;

     case  WM_CALC_ABORTED:
                         iStatus = STATUS_READY ;
                         InvalidateRect (hwnd, NULL, TRUE) ;
                         return 0 ;

     case  WM_PAINT:
                         hdc = BeginPaint (hwnd, &ps) ;

                         GetClientRect (hwnd, &rect) ;

                         wsprintf (szBuffer, szMessage[iStatus], REP, lTime) ;
                         DrawText (hdc, szBuffer, -1, &rect,
                           DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

                         EndPaint (hwnd, &ps) ;
                         return 0 ;

     case  WM_DESTROY:
                         PostQuitMessage (0) ;
                         return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

这是一个相当简单的程式，但是我认为您将看到它如何展示在多执行绪程式中完成大作业的通用方法。为了使用 BIGJOB1 程式，在视窗的显示区域中按下滑鼠左键，从而开始暴力的性能测试计算的 1,000,000 次重复，这在一台 300MHz 的 Pentium II 机器上将花费 2 秒。当完成计算时，花费的时间将显示在视窗上。当正在进行计算时，您可以通过在显示区域中按下滑鼠右键来终止它。

让我们来看一看这是如何实作的：

视窗讯息处理程式拥有了一个被叫做 iStatus 的静态变数（该变数可以被设定为在程式开始处定义的三个常数之一，常数以 STATUS 为字首），该变数表示程式是否准备好进行一次计算，是否正在进行一次计算，或者是否完成了计算。程式在 WM_PAINT 讯息处理期间使用 iStatus 变数在显示区域的中央显示一个适当的字串。

视窗讯息处理程式还拥有一个静态结构（型态为 PARAMS，也定义在程式的顶部），该结构是在视窗讯息处理程式和其他执行绪之间的共用资料。结构只

有两个栏位——hwnd（程式视窗的代号）和 bContinue，这是一个布林变数，用於指示执行绪是否继续计算或者停止。

当您在显示区域中按下滑鼠左键时，视窗讯息处理程式将 iStatus 变数设为 STATUS_WORKING，并设定 PARAMS 结构中的两个栏位。结构的 hwnd 栏位被设定为视窗代号，当然，bContinue 被设定为 TRUE。

然後视窗程序呼叫_beginthread 函式。执行绪函式 Thread 以呼叫 GetCurrentTime 开始,GetCurrentTime 取得以毫秒计的 Windows 启动以来已经-执行了的时间。然後它进入一个 for 回圈，重复 1,000,000 次的暴力测试计算。还要注意，如果 bContinue 被设为了 FALSE，那么执行绪将退出回圈。

在 for 回圈之後，执行绪函式检查它是否确实完成了 1,000,000 次计算。如果是，那么它再次呼叫 GetCurrentTime 获得所经过的时间，然後使用 SendMessage 向视窗讯息处理程式发送一个由程式定义的 WM_USER_DONE 讯息，并以经过的时间作为 lParam 参数。如果计算是在未完成之前被终止的（即，如果在回圈期间 PARAMS 结构的 bContinue 栏位变为 FALSE），那么执行绪将发送给视窗讯息处理程式一个 WM_USER_ABORTED 讯息。然後，执行绪通过呼叫_endthread 正常地结束。

在视窗讯息处理程式中，当您在显示区域中按下滑鼠右键时，PARAMS 结构的 bContinue 栏位被设为 FALSE。这是如何在完成计算之前结束计算的方法。

注意 Thread 中的 pparams 变数定义为 volatile，这种型态限定字向编译器指出变数可能会在实际的程式叙述外被修改（例如被另一个执行绪）。否则，最佳化的编译器会假设 pparams->bContinue 不能被 for 回圈内的程式码修改，没有必要在每层回圈中检查变数。volatile 关键字防止这样的最佳化进行。

视窗讯息处理程式处理 WM_USER_DONE 讯息时，首先储存经过的时间。对 WM_USER_DONE 和 WM_USER_ABORTED 讯息的处理都是透过对 InvalidateRect 的呼叫产生 WM_PAINT 讯息并在显示区域显示一个新的字串。

提供一个方法（如结构中的 bContinue 栏位）允许执行绪正常终止，通常是一个好主意。KillThread 函式只有在正常终止执行绪比较困难时才应该使用，原因是执行绪可以配置资源，如记忆体等。如果当执行绪终止时没有释放所配置的记忆体，那么记忆体将仍然是被配置了的。执行绪不是程序：所配置的资源在一个程序的所有执行绪之间是共用的，因此当执行绪终止时，资源不会被自动释放。好的程式结构要求一个执行绪释放由它配置的所有资源。

您还应该知道当第二个执行绪仍在执行时，可以建立第三个执行绪。如果 Windows 在 SendMessage 呼叫和_endthread 呼叫之间，将控制权从第二个执行绪切换到第一个执行绪，那么视窗讯息处理程式就可能回应滑鼠按键而建立一

个新的执行绪，从而出现了上述的情况。这不是什么问题，但是如果这对您自己的应用来说是一个问题的话，那么您可能会考虑使用临界区域来避免执行绪之间的冲突。

事件物件

BIGJOB1 在每次需要执行暴力测试计算时，就建立一个执行绪。执行绪在完成计算之後自动终止。

另一种可用的方法是在程式的整个生命周期内保持执行绪的执行，但是只在必要时才启动它。这是一个应用事件物件的理想情况。

事件物件可以是「有信号的」（也称为「被设立的」）或「没信号的」（也称为「被重置的」）。您可以通过下面呼叫来建立事件物件：

```
hEvent = CreateEvent (&sa, fManual, fInitial, pszName) ;
```

第一个参数（指向一个 SECURITY_ATTRIBUTES 结构的指标）和最後一个参数（一个事件物件的名字）只有在事件物件被多个程序共用时才有意义。在同一程序中，这些参数通常被设定为 NULL。如果您希望事件物件被初始化为有信号的，那么将 fInitial 参数设定为 TRUE。而如果希望事件物件被初始化为无信号的，则将 fInitial 参数设定为 FALSE。稍後，我将简短地描述 fManual 参数。

要设立一个现存的事件物件，呼叫

```
SetEvent (hEvent) ;
```

要重置一个事件物件，呼叫

```
ResetEvent (hEvent) ;
```

一个程式通常呼叫：

```
WaitForSingleObject (hEvent, dwTimeOut) ;
```

并且将第二个参数设定为 INFINITE。如果事件物件目前是被设立的，那么函式将立即传回，否则，函式将暂停执行绪直到事件物件被设立。如果您将第二个参数设定为一个以毫秒计的超时时间值，这样函式也可能在事件物件被设立之前传回。

如果最初的 CreateEvent 呼叫的 fManual 参数被设定为 FALSE，那么事件物件将在 WaitForSingleObject 函式传回时自动重置。这种功能特性通常使得事件物件没有必要使用 ResetEvent 函式。

现在，我们可以来看一看程式 20-5 所示的 BIGJOB2.C 程式。

程式 20-5　BIGJOB2

```
BIGJOB2.C
/*-------------------------------------------------------------------------
--
      BIGJOB2.C -- Multithreading Demo
                                                (c) Charles Petzold, 1998
```

```
------------------------------------------------------------------------
-*/

#include <windows.h>
#include <math.h>
#include <process.h>

#define REP                                          1000000

#define STATUS_READY                    0
#define STATUS_WORKING                 1
#define STATUS_DONE                     2

#define WM_CALC_DONE                    (WM_USER + 0)
#define WM_CALC_ABORTED                (WM_USER + 1)

typedef struct
{
      HWND                    hwnd ;
      HANDLE                hEvent ;
      BOOL                    bContinue ;
}
PARAMS, *PPARAMS ;
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  PSTR szCmdLine, int iCmdShow)
{
      static TCHAR     szAppName[] = TEXT ("BigJob2") ;
      HWND                              hwnd ;
      MSG                         msg ;
      WNDCLASS                        wndclass ;

      wndclass.style               = CS_HREDRAW | CS_VREDRAW ;
      wndclass.lpfnWndProc         = WndProc ;
      wndclass.cbClsExtra          = 0 ;
      wndclass.cbWndExtra          = 0 ;
      wndclass.hInstance           = hInstance ;
      wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName  = NULL ;
      wndclass.lpszClassName = szAppName ;

      if (!RegisterClass (&wndclass))
      {
      MessageBox (     NULL, TEXT ("This program requires Windows NT!"),
                  szAppName, MB_ICONERROR) ;
                   return 0 ;
```

```
    }

     hwnd = CreateWindow (  szAppName, TEXT ("Multithreading Demo"),
                            WS_OVERLAPPEDWINDOW,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, CW_USEDEFAULT,
                            NULL, NULL, hInstance, NULL) ;

     ShowWindow (hwnd, iCmdShow) ;
     UpdateWindow (hwnd) ;

     while (GetMessage (&msg, NULL, 0, 0))
     {
                TranslateMessage (&msg) ;
                DispatchMessage (&msg) ;
     }
     return msg.wParam ;
}

void Thread (PVOID pvoid)
{
     double           A = 1.0 ;
     INT                   i ;
     LONG                  lTime ;
     volatile         PPARAMS pparams ;

     pparams = (PPARAMS) pvoid ;
     while (TRUE)
     {
                WaitForSingleObject (pparams->hEvent, INFINITE) ;
                lTime = GetCurrentTime () ;
                for (i = 0 ; i < REP && pparams->bContinue ; i++)
                           A = tan (atan (exp (log (sqrt (A * A))))) + 1.0 ;
                if (i == REP)
                {
                      lTime = GetCurrentTime () - lTime ;
                      PostMessage (pparams->hwnd, WM_CALC_DONE, 0, lTime) ;
                }
                else
                      PostMessage (pparams->hwnd, WM_CALC_ABORTED, 0, 0) ;
     }
}

LRESULT CALLBACK WndProc (   HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
     static     HANDLE     hEvent ;
     static     INT        iStatus ;
```

```
     static     LONG       lTime ;
     static     PARAMS     params ;
     static          TCHAR *     szMessage[] = { TEXT ("Ready (left mouse button
begins)"),
                TEXT ("Working (right mouse button ends)"),
                TEXT ("%d repetitions in %ld msec") } ;
     HDC                                   hdc ;
     PAINTSTRUCT                      ps ;
     RECT                                  rect ;
     TCHAR                                 szBuffer[64] ;

     switch (message)
     {
     case  WM_CREATE:
                    hEvent = CreateEvent (NULL, FALSE, FALSE, NULL) ;

                    params.hwnd = hwnd ;
                    params.hEvent = hEvent ;
                    params.bContinue = FALSE ;

                    _beginthread (Thread, 0, 秏s) ;

                    return 0 ;

     case  WM_LBUTTONDOWN:
                    if (iStatus == STATUS_WORKING)
                    {
                              MessageBeep (0) ;
                              return 0 ;
                    }
                    iStatus = STATUS_WORKING ;
                    params.bContinue = TRUE ;

                    SetEvent (hEvent) ;

                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;

     case  WM_RBUTTONDOWN:
                    params.bContinue = FALSE ;
                    return 0 ;

     case  WM_CALC_DONE:
                    lTime = lParam ;
                    iStatus = STATUS_DONE ;
                    InvalidateRect (hwnd, NULL, TRUE) ;
                    return 0 ;
```

```
     case  WM_CALC_ABORTED:
                        iStatus = STATUS_READY ;
                        InvalidateRect (hwnd, NULL, TRUE) ;
                        return 0 ;

     case  WM_PAINT:
                        hdc = BeginPaint (hwnd, &ps) ;

                        GetClientRect (hwnd, &rect) ;

                        wsprintf ( szBuffer, szMessage[iStatus], REP, lTime) ;
                        DrawText ( hdc, szBuffer, -1, &rect,
                                   DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;

                        EndPaint (hwnd, &ps) ;
                        return 0 ;

     case  WM_DESTROY:
                        PostQuitMessage (0) ;
                        return 0 ;
     }
     return DefWindowProc (hwnd, message, wParam, lParam) ;
}
```

处理 WM_CREATE 讯息时，视窗讯息处理程式首先建立一个初始化为没信号的自动重置事件物件，然後建立执行绪。

Thread 函式进入一个无限的 while 回圈，在回圈开始时首先呼叫 WaitForSingleObject（注意 PARAMS 结构包括一个包含事件物件代号的栏位）。因为事件被初始化为重置的，所以执行绪的执行被阻挡在函式呼叫中。按下滑鼠左键将导致视窗程序呼叫 SetEvent，这将释放由 WaitForSingleObject 呼叫产生的第二个执行绪，并开始暴力测试计算。当计算完之後，执行绪再次呼叫 WaitForSingleObject，但是由於第一次呼叫已经使事件物件重置，因此，执行绪将被暂停，直到再次按下滑鼠。

在其他方面，程式几乎和 BIGJOB1 完全一样。

执行绪区域储存空间（TLS）

多执行绪程式中的整体变数（以及任何被配置的记忆体）被程式中的所有执行绪共用。在一个函式中的局部静态变数也被使用函式的所有执行绪共用。一个函式中的局部动态变数是唯一於各个执行绪的，因为它们被储存在堆叠上，而每个执行绪有它自己的堆叠。

对各个执行绪唯一的持续性储存空间有存在的必要。例如，我在本章前面提到过的 C 中的 strtok 函式要求这种型态的储存空间。不幸的是，C 语言不支

援这类储存空间。但是 Windows 中提供了四个函式，它们实作了一种技术来做到这一点，并且 Microsoft 对 C 的扩充语法也支援它，这就叫做执行绪区域储存空间。

下面是 API 工作的方法：

首先，定义一个包含需要唯一於执行绪的所有资料的结构，例如：

```
typedef struct
{
      int a ;
      int b ;
}
DATA, * PDATA ;
```

主执行绪呼叫 TlsAlloc 获得一个索引值：

```
dwTlsIndex = TlsAlloc () ;
```

这个值可以储存在一个整体变数中或者通过参数结构传递给执行绪函式。

执行绪函式首先为该资料结构配置记忆体，并使用上面所获得的索引值呼叫 TlsSetValue：

```
TlsSetValue (dwTlsIndex, GlobalAlloc (GPTR, sizeof (DATA)) ;
```

该函式将一个指标和某个执行绪及某个执行绪索引相关联。现在，任何需要使用这个指标的函式（包括最初的执行绪函式本身）都可以包含如下所示的程式码：

```
PDATA pdata ;
...
pdata = (PDATA) TlsGetValue (dwTlsIndex) ;
```

现在函式可以设定或者使用 pdata->a 和 pdata->b 了。在执行绪函式终止以前，它释放配置的记忆体：

```
GlobalFree (TlsGetValue (dwTlsIndex)) ;
```

当使用该资料的所有执行绪都终止之时，主执行绪将释放索引：

```
TlsFree (dwTlsIndex) ;
```

这个程序刚开始可能令人有些迷惑，因此如果能看一看如何实作执行绪区域储存空间可能会有帮助（我不知道 Windows 实际上是如何实作的，但下面的方案是可能的）。首先，TlsAlloc 可能只是配置一块记忆体（长度为 0）并传回一个索引值，即指向这块记忆体的一个指标。每次使用该索引呼叫 TlsSetValue 时，通过重新配置将记忆体块增大 8 个位元组。在这 8 个位元组中储存的是呼叫函式的执行绪 ID（通过 GetCurrentThreadId 来获得）以及传递给 TlsSetValue 函式的指标。TlsSetValue 简单地使用执行绪 ID 来搜寻作业系统管理的执行绪区域储存空间位址表，然後传回指标。TlsFree 将释放记忆体块。所以您看，这可能是一件容易得可以由您自己来实作的事情。不过，既然已经有工具为您做好了这些工作，那也不错。

Microsoft 对 C 的扩充功能使这件工作更加容易。只要在要对每个执行绪都保留不同内容的变数前加上__declspec (thread)就好了。对於任何函式的外部静态变数，则为：

```
__declspec (thread) int iGlobal = 1 ;
```

对於函式内部的静态变数，则为：

```
__declspec (thread) static int iLocal = 2 ;
```