China-pub.com 下载

第11章 编写多线程应用程序

本章内容：

- 对线程的解释
- TThread对象
- 管理多线程
- 一个多线程的示范程序
- 多线程与数据库
- 多线程与图形处理

Win32操作系统提供了在应用程序中执行多线程的能力。从16位的Windows升级到Win32的一个最大受益便是：它允许多线程同时运行。这也是要升级到32位Delphi的一个最主要的原因。本章提供了程序中如何进行多线程编程的所有细节。

11.1 对线程的解释

如同在第3章“Win32 API”中所讨论过的，线程是一种操作系统对象，它表示在进程中代码的一条执行路径。在每一个Win32的应用程序中都至少有一个线程，它通常被称为主线程或默认线程。在应用程序中也可以自由地创建别的线程去执行其他任务。

线程技术使不同的代码可以同时运行。当然，只有在多CPU的计算机上，多个线程才能够真正地同时运行。然而，由于操作系统把CPU的时间分成很短的片段分配给每个线程，这样给人的感觉好像是多个线程真的同时运行。

提示 线程不能也从来没有被16位的Windows支持。这就意味着，任何32位版本的Delphi的多线程程序代码都有无法在Delphi 1环境下编译。如果你在为这两个平台开发程序，请一定记住这一点。

11.1.1 一种新型的多任务

线程的概念与16位环境中的多任务有很大的不同。或许曾听人们这样讲：Win32是一种抢占式操作系统，而Windows 3.1是一种协作式的多任务环境。

其关键区别在于：在抢占式多任务环境中，操作系统负责管理哪个线程在什么时候执行。如果当线程1暂停执行时，线程2才有机会获得CPU时间，我们说线程1是抢占的。如果某个线程的代码陷入死循环，这并不可怕，操作系统仍会安排时间给其他线程。

在Windows 3.1下，程序员必须保证应用程序能够把控制权返还给Windows。如果这一步失败，将导致整个操作环境锁死，或许你已经有过这样的痛苦经历。只要稍微想想便会明白，16位的Windows是如此脆弱，它依赖于应用程序的运行情况，并且不允许程序陷入死循环或无穷递归以及任何封闭状态。这是因为所有的应用程序都必须协助Windows工作，这种工作类型被称为协作式多任务系统。

11.1.2 在Delphi程序中使用多线程

对一个Windows程序员来说，线程提供了非常大的好处。可以在应用程序中的任何地方创建多个

China-pub.com 下载

附属线程，它们在后台进行各种类型的处理。例如：在一个电子表格程序中计算单元格，或是脱机打印Word文档。即使后台正在处理许多工作，也不会影响前台的用户界面。

大多数VCL在被设计时，都只考虑了在任何时刻只有一个线程来访问它。其局限性尤其体现在VCL的用户界面部分。同时，一些非用户界面部分也不是线程安全的。

1. 非用户界面的VCL

实际上VCL只有很少的部分保证是线程安全的。可能在这很少的部分中，最让人注意的是 VCL的属性流机制。VCL的流机制确保了组件流能被多线程安全地读写。请记住即使最基础的 VCL类(诸如TList)，也不是为安全地同时操作多个线程而设计的。对某些情况， VCL提供了一些线程安全的替代，比如，用TThreadList 来替代TList可以解决多个线程操作的问题。

2. 用户界面的VCL

VCL要求所有的用户界面控制要发生在一个应用程序的主线程的环境中 (线程安全的TCanvas类除外，本章后面就此将说明)。当然，利用技术手段是可以有效地利用附属线程更新用户界面的 (后面将会讨论)。本章的例子将介绍一些在Delphi程序中使用多线程的方法。

11.1.3 关于线程的滥用

好事太多也可能会是件坏事，线程正是这样。从程序设计的角度看，尽管线程能帮助我们解决一些问题，但同时它又带来新的问题。例如，在编写一个集成开发环境时，也许想使编译器在一个专门的线程中运行，这样才能使你在编译时照常做其他的工作。可问题是，如果正在编译时修改了一个文件，会出现什么情况呢？为解决这个问题，你可以临时复制那个文件的副本，或者干脆禁止用户在编译期间修改尚未编译的文件。这说明，线程可以解决一些问题，但同时也带来了其他问题。一个更为严重的问题是多线程依赖于时间片，在多线程中的缺陷非常难以调试。同时，编写和实现线程安全的代码也比较困难，程序员需要考虑很多的因素。

11.2 TThread 对象

Delphi把有关线程的API封装在TThread 这个Object Pascal的对象中 。虽然 TThread已经封装了几乎所有与线程有关的API。但在某些情况下，尤其在处理线程同步的问题时，仍需调用一些别的 API函数。本节将介绍TThread 用法。

11.2.1 TThread基础

下面是TThread在Classes单元中的声明：

```
type
  TThread = class
  private
    FHandle: THandle;
    FThreadID: THandle;
    FTerminated: Boolean;
    FSuspended: Boolean;
    FFreeOnTerminate: Boolean;
    FFinished: Boolean;
    FReturnValue: Integer;
    FOnTerminate: TNotifyEvent;
    FMethod: TThreadMethod;
    FSynchronizeException: TObject;
    procedure CallOnTerminate;
    function GetPriority: TThreadPriority;
```

China-pub.com 下载

```
    procedure SetPriority(Value: TThreadPriority);
    procedure SetSuspended(Value: Boolean);
  protected
    procedure DoTerminate; virtual;
    procedure Execute; virtual; abstract;
    procedure Synchronize(Method: TThreadMethod);
    property ReturnValue: Integer read FReturnValue write FReturnValue;
    property Terminated: Boolean read FTerminated;
  public
    constructor Create(CreateSuspended: Boolean);
    destructor Destroy; override;
    procedure Resume;
    procedure Suspend;
    procedure Terminate;
    function WaitFor: Integer;
    property FreeOnTerminate: Boolean read FFreeOnTerminate write FFreeOnTerminate;
    property Handle: THandle read FHandle;
    property Priority: TThreadPriority read GetPriority write SetPriority;
    property Suspended: Boolean read FSuspended write SetSuspended;
    property ThreadID: THandle read FThreadID;
    property OnTerminate: TNotifyEvent read FOnTerminate write FOnTerminate;
  end;
```

从声明中可以看出，TThread是直接从TObject继承下来的，因此，它不是组件。其中，Execute()是抽象的，说明TThread类是抽象的。这意味着，不能创建TThread的实例，而只能创建其派生类的实例。可以选择Delphi的主菜单中的File | New命令，然后在New Items对话框中双击Thread Object。New Items对话框如图11-1所示。

当双击Thread Object后，将出现一个对话框，它会提示输入线程对象的名称。例如，输入TTestThread。Delphi会自动创建一个包括新创建的线程对象的单元，如下所示：

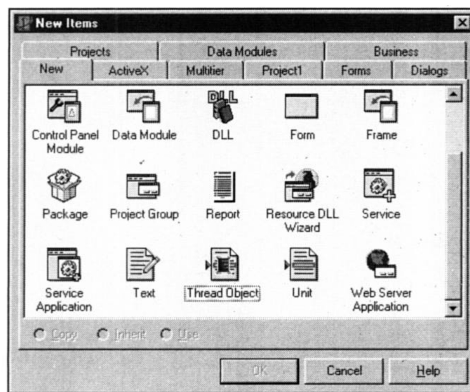

图11-1 在New Items对话框中的Thread Object项

```
type
  TTestThread = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;
```

正如你看到的一样，TThread的派生类中唯一必须覆盖的方法是Execute()。假设你要在TTestThread中进行复杂的计算，可以如下定义Execute()：

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  for i := 1 to 2000000 do
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
end;
```

当然，本段代码只是为了演示一个长循环。本线程的唯一目的是解决长时间的计算工作。

现在，可以通过调用线程对象的Create()使上述代码执行。在下例中，看到在主窗体上有一个按钮，单击此按钮就会调用Create()(请注意，不要忘记在主窗体单元的uses子句中包含TTestThread单元，否则会导致编译错误)。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewThread: TTestThread;
begin
  NewThread := TTestThread.Create(False);
end;
```

如果运行此程序并单击按钮，上述的那个长循环就会执行，但同时，你会注意到仍然可以操纵窗口，做各种其他操作。

注意 当TThread的Create()被调用时，需要传递一个布尔型的参数CreateSuspended。如果把这个参数设成False，那么当调用Create()后，Excute()会被自动地调用，也就是自动地执行线程代码。如果该参数设为True，则需要运行TThread的Resume()来唤醒线程。一般情况下，当你调用Create()后，还会有一些其他的属性要求设置。所以，应当把CreateSuspended参数设为True，因为在TThread已执行的情况下设置TThread的属性可能会引起麻烦。

再深入一点讲，在构造函数Create()中隐含调用了一个RTL例程BeginThread()，而它又调用了一个API函数CreateThread()来创建一个线程对象的实例。CreateSuspended参数表明是否传递CREATE_ SUSPEDED标志给CreateThread()。

11.2.2 TThread实例

回到TTestThread对象的Excute()，我们注意到它声明了一个局部变量 i ，试想一下，当存在两个TTestThread对象的实例时，i的值会怎么样呢？它会覆盖掉另一个实例的值吗？或者第一个实例具有优先级？回答是否定的。因为Win32会为每个线程都分配一个单独的栈。这意味着，当你创建了TTestThread对象的多个实例后，在每个线程实例所在的栈里都有一份 i 的副本。因此，所有的线程都是独立地运行的。

然而，上述内容并不适合于线程中的全局变量。在本章的11.3节中我们将讨论此问题。

11.2.3 线程的终止

当线程对象的Excute()执行完毕，我们就认为此线程终止了。这时，它会调用Delphi的一个标准例程EndThread()，这个例程再调用API函数ExitThread()。由ExitThread()来清除线程所占用的栈。

当结束使用TThread对象时，应该确保已经把这个Object Pascal对象从内存中清除了。这才能确保所有内存占有都释放掉。尽管在进程终止时会自动清除所有的线程对象，但及时清除已不再用的对象，可以使内存的使用效率提高。利用将FreeOnTerminate的属性设为True的方法来及时清除线程对象是最方便的办法，这只需要在Excute()退出前设置就行了。设置方法如下：

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  FreeOnTerminate := True;
  for i := 1 to 2000000 do
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
end;
```

这样，当一个线程终止时，就会触发OnTerminate事件，就有机会在事件处理过程内清除线程对

China-pub.com 下载

象了。

提示 OnTerminate事件是在主线程的环境中发生的。这意味着，在处理这个事件的处理过程中，你可以不需要借助于Synchronize()而自由地访问VCL。

要记住Execute() 需要经常地检查Terminated属性的值，来确认是否要提前退出。尽管这将意味着当使用线程工作时，你必须关心更多的事情，但它能确保在线程结束时，能够完成必要的清除。下面是一段在Execute()增加处理操作的简单代码：

```
procedure TTestThread.Execute;
var
  i: integer;
begin
  FreeOnTerminate := True;
  for i := 1 to 2000000 do begin
    if Terminated then Break;
    inc(Answer, Round(Abs(Sin(Sqrt(i)))));
  end;
end;
```

注意 某些紧急情况下，你可以使用Win32 API函数TerminateThread()来终止一个线程。但是，除非没有别的办法了，否则不要用它。例如，当线程代码陷入死循环时。TerminateThread()的声明如下：

function TerminateThread(hThread:THandle;dwExitCode:DWORD);

TThread的Handle属性可以作为第一个参数，因此，TerminateThread()常这样调用：

TerminateThread(MyHosedThread.Handle,0)

如果选择使用这个函数，应该考虑到它的负面影响。首先，此函数在 Windows NT与在Windows 95/98下并不相同。在Windows 95/98 下，这个函数能够自动清除线程所占用的栈；而在Windows NT下，在进程被终止前栈仍然保留。其次，无论线程代码中是否有 try...finally块，这个函数都会使线程立即停止执行。这意味着，被线程打开的文件没有被关闭、由线程申请的内存没有被释放等情况。而且，这个函数在终止线程的时候也不通知DLL，当DLL关闭时，这也容易出现问题。在第9章“动态链接库”中有关于这方面的更多内容。

11.2.4 与VCL同步

如同在前面多次提到的，对 VCL的访问只能在主线程中。这将意味着：所有需要与用户打交道的代码都只能在主线程的环境中执行。这是其结构上明显的不足，并且这种需求看起来只局限在表面上，但它实际上有一些优点。

1. 单线程用户界面的好处

首先，只有一个线程能够访问用户界面，这减少了编程的复杂性。 Win32要求每个创建窗口的线程都要使用GetMessage()建立自己的消息循环。正如你所想的，这样的程序将会非常难于调试，因为消息的来源实在太多了。

其次，由于VCL只用一个线程来访问它，那些用于把线程同步的代码就可以省略了，从而改善了应用程序的性能。

2. Synchronize()方法

在TThread中有一个方法叫 Synchronize()，通过它可以让线程的一些方法在主线程中执行。Synchronize()的声明如下：

```
procedure Synchronize(Method: TThreadMethod);
```

参数Method的类型是TThreadMethod类型(这是一个无参数的过程)，类型声明如下：

```
type
  TThreadMethod = procedure of object;
```

Method参数用来传递要在主线程中执行的方法。以TTestThread对象为例，如果要在一个编辑框中显示计算的结果。首先要在 TTestThread中增加能对编辑控件的 Text属性进行修改的方法，然后，用Synchronize()来调用此方法。

我们给这个方法取名为GiveAnswer()。在清单11-1中列出了例子的代码，其中包含了更新主窗体的编辑控件的代码。

清单11-1 ThrdU.PAS单元

```
unit ThrdU;

interface

uses
  Classes;

type
  TTestThread = class(TThread)
  private
    Answer: integer;
  protected
    procedure GiveAnswer;
    procedure Execute; override;
  end;

implementation
uses SysUtils, Main;

{ TTestThread }

procedure TTestThread.GiveAnswer;
begin
  MainForm.Edit1.Text := InttoStr(Answer);
end;

procedure TTestThread.Execute;
var
  I: Integer;
begin
  FreeOnTerminate := True;
  for I := 1 to 2000000 do
  begin
    if Terminated then Break;
    Inc(Answer, Round(Abs(Sin(Sqrt(I)))));
    Synchronize(GiveAnswer);
  end;
end;

end.
```

你已经知道Synchronize()的作用是在主线程中执行一个方法。但是，你或许已把 Synchronize()当成一个黑匣子，不清楚它是如何工作的。如果愿意揭开这个谜，请继续向下读。

当你在程序中第一次创建一个附属线程时， VCL将会从主线程环境中创建和维护一个隐含的线程

China-pub.com 下载

窗口。此窗口唯一的目的是把通过Synchronize()调用的方法排队。

Synchronize()把由Method参数传递过来的方法保存在TThread的FMethod字段中，然后，给线程窗口发一个CM_EXECPROC消息，并且把消息的lParam参数设为self(这里指线程对象)。当线程窗口的窗口过程收到这个消息后，它就调用 FMethod字段所指定的方法。由于线程窗口是在主线程内创建的，线程窗口的窗口过程也将被主线程执行。因此，FMethod字段所指定的方法就在主线程内执行。

图11-2形象地说明了Synchronize()的内部机制。

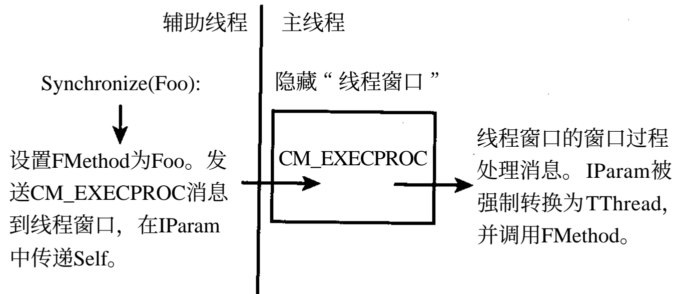

图11-2 Synchronize()的原理

3. 用消息来同步

可以利用在线程之间使用消息同步以替代 TThread.Synchronize()方法。可以使用 API函数SendMessage() 或PostMessage()来发送消息。例如，下面是一段用来在一个线程中设置另一个线程中的编辑框文本的代码：

```
var
  S: string;
begin
  S := 'hello from threadland';
  SendMessage(SomeEdit.Handle, WM_SETTEXT, 0, Integer(PChar(S)));
end;
```

11.2.5 一个演示程序

为了充分地说明在Delphi中的多线程编程，下面介绍一个演示程序。可以把它存为 EZThrd。这个演示程序上有一个文本编辑器、一个按钮、一个编辑框和若干个标签，如图 11-3所示。

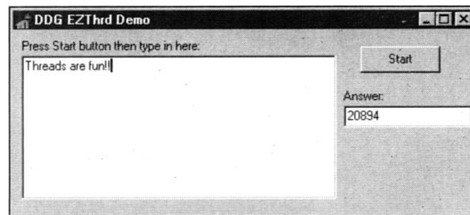

图11-3 EZThrd程序的主窗体

此程序的主窗体单元代码在清单 11-2中。

清单11-2 MAIN.PAS单元

```
unit Main;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
   Dialogs, StdCtrls, ThrdU;

type
  TMainForm = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Memo1: TMemo;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.Button1Click(Sender: TObject);
var
  NewThread: TTestThread;
begin
  NewThread := TTestThread.Create(False);
end;

end.
```

当单击按钮创建附属线程后，仍然可以在多行文本编辑器中输入文本，就好像附属线程不存在一样。当线程代码执行完后，计算结果将会显示在编辑框中。

11.2.6 优先级和时序安排

正如前面提到的，操作系统会负责为每个线程分配CPU时间。一个线程所分配到的CPU时间主要取决于该线程的优先级，而线程的优先级又取决于进程的优先级类和线程本身的相对优先级。

1. 进程的优先级类

进程的优先级类用来描述一个进程的优先程度。Win32支持四种不同的优先级类：Idle、Normal、High 和Realtime。其中，Normal是默认的优先级。在Windows单元中，每一种优先级类都对应着一个标志。当要进行进程的优先级设置时，可以用一种优先级类与CreateProcess()的参数dwCreationFlags进行或操作。另外，还可以动态地为一个已有的进程调整优先级类。每个优先级类也对应一个数字，值在4~ 24之间。

注意 在Windows NT/2000下，要有特殊的权限才能修改进程的优先类。默认的设置允许进程设置它们的优先级类，但是，这些都可以由系统管理员来关闭，尤其是在高负载的 Windows NT/2000服务器上。

表11-1列出了所有的优先级类以及对应的标志和数值。

China-pub.com 下载

表11-1 进程优先级类

类	标 志	值
Idle	IDLE_PRIORITY_CLASS	$40
Below normal[①]	BELOW_NORMAL_PRIORITY_CLASS	$4000
Normal	NORMAL_PRIORIT_CLASS	$20
Above normal[①]	ABOVE_NORMAL_PRIORITY_CLASS	$8000
High	HIGH_PRIORITY_CLASS	$80
Realtime	REALTIME_PRIORITY_CLASS	$100

① 只在Windows 2000上有效，这些标志常量在Delphi 5版本的Windows，pas中没有。

我们可以利用GetPriorityClass() 和SetPriorityClass()这两个函数来动态地获取或设置一个进程的优先级类。这两个函数声明如下：

```
function GetPriorityClass(hProcess: THandle): DWORD; stdcall;

function SetPriorityClass(hProcess: THandle; dwPriorityClass: DWORD): BOOL;
  stdcall;
```

hProcess参数用于指定一个进程的句柄。在大多数情况下，调用这两个函数是为了获取或设置本进程的优先级类。这时，可以使用API函数GetCurrentProcess()。此函数声明如下：

```
function GetCurrentProcess: THandle; stdcall;
```

此函数的返回值是一个当前进程的假句柄。之所以说它假，是因为这个函数并没有创建新的句柄，并且，返回值不必由CloseHandle()来关闭。它仅仅是用来定位一个已存在的句柄。

要设置High优先级类，可参照如下代码：

```
if not SetPriorityClass(GetCurrentProcess, HIGH_PRIORITY_CLASS) then
  ShowMessage('Error setting priority class.');
```

注意 大多数情况下，进程的优先级类不要被设为Realtime。因为，大多数操作系统本身的线程的优先级类比Realtime低。如果一个进程得到的CPU时间比操作系统本身还多，后果是无法想象的。

即使将进程的优先级类设为High,也可能引起问题。因为，当高优先级的线程没有大部分空时间或等待外部事件时，它要从低优先级的线程和进程中抢夺CPU时间，直到它被一事件阻塞或处于空闲状态或处理消息。所以，在抢占式多任务操作系统中如果不能合理地安排优先级，就很容易崩溃。

2. 相对优先级

决定一个线程全面的优先级的另一方面是相对优先级。优先级类是针对进程的，它对进程内部的所有线程都有效。而相对优先级是针对某个线程的。一个线程的相对优先级可设为以下七种： Idle、Lowest、Below Normal、Normal、Above Normal、Highest 和Time Critical。

TThread中声明了一个枚举类型TThreadPriority，它列举了此七种相对优先级：

```
type
  TThreadPriority = (tpIdle, tpLowest, tpLower, tpNormal, tpHigher,
    tpHighest, tpTimeCritical);
```

你可以通过读写 TThread中TThreadPriority属性来获取或设置一个线程的优先级。下面把线程MyThread的优先级设置为Highest:

```
MyThread.Priority := tpHighest.
```

如同优先级类一样，每个相对优先级也对应着一个数字。但是，相对优先级的数字是有符号的，它们分别表示在优先级类的基础上调高或调低优先级。因此，有时候相对优先级又叫做增量优先级。一个线程的全面优先级可以取从 1~31之间的任何值(1是最低的)。在Windows单元中声明了一些变量，

China-pub.com
下载

它们分别对应于一个优先级，见表11-2。

表11-2　线程的优先级

TThreadPriority	常　量	值
tpIdle	THREAD_PRIORITY_IDLE	-15*
tpLowest	THREAD_PRIORITY_LOWEST	-2
tpBelow Normal	THREAD_PRIORITY_BELOW_NORMAL	-1
tpNormal	THREAD_PRIORITY_NORMAL	0
tpAbove Normal	THREAD_PRIORITY_ABOVE_NORMAL	1
tpHighest	THREAD_PRIORITY_HIGHEST	2
tpTimeCritical	THREAD_PRIORITY_TIME_CRITICAL	15*

在表11-2中，在tpIdle和tpTimeCritical的对应数字旁边打上星号，是因为它不像别的常量，它不能真正地把值加入优先级类。任何一个相对优先级被设为tpIdle的线程，无论它的优先级类是什么，其优先级将总是1。只有在进程的优先级类设为Realtime,并且相对优先级为tpIdle时，线程的优先级将为16。同样，对于相对优先级tpTimeCritical，无论它的优先级类是什么，其优先级将总是15。只有在进程的优先级类设为Realtime，并且相对优先级为tpTimeCritical时，线程的优先级将为31。

11.2.7　挂起和唤醒线程

回顾本章在先前学习TThread 的构造器Create()时讲过，当创建一个线程时，可以先使它处于挂起状态，在调用了Resume()唤醒线程后再执行线程代码。你可能已经想到，对线程可以调用 Suspend()和Resume()来动态地挂起或唤醒。

11.2.8　测试线程的时间

在16位时代，当我们在Window 3.x下编程时，经常会用到GetTickCount() 或timeGetTime()来判断一段代码的执行时间。示例如下：

```
var
  StartTime, Total: Longint;
begin
  StartTime := GetTickCount;
  { 进行一些操作 }
  Total := GetTickCount - StartTime;
```

在多线程环境下，这是很困难的，因为在执行程序中间，操作系统可能会把 CPU时间片分给别的进程。所以，用上述方法测出的时间会不真实。

为了解决这个问题， Windows NT提供了一个函数GetThreadTimes(),它可以提供详细的时间信息。其声明如下：

```
function GetThreadTimes(hThread: THandle; var lpCreationTime, lpExitTime,
    lpKernelTime, lpUserTime: TFileTime): BOOL; stdcall;
```

hThread参数是线程的句柄，其他参数都是变参，由GetThreadTimes()函数返回它们的值。其中：

- lpCreationTime　线程创建的时间。
- lpExitTime　线程退出的时间。如果线程还在执行，此值无意义。
- lpKernelTime　执行操作系统代码所用的时间。
- lpUserTime　执行应用程序本身代码所用的时间。

以上四个参数都是TFileTime类型。此类型在Windows单元中声明如下：

```
type
```

China-pub.com 下载

```
TFileTime = record
  dwLowDateTime: DWORD;
  dwHighDateTime: DWORD;
end;
```

此类型的声明有些不寻常。由dwLowDateTime和dwHighDateTime组成一个64位值，代表从1601年1月1日以来的计数(单位：千万分之一秒)。

提示 因为TFileTime的长度是64位，所以，为了进行数学运算你可以把它转换为Int64。这样，我们可以对两个TFileTime的值比较大小，例如：

```
if Int64(UserTime) > Int64(KernelTime) then Beep;
```

为了帮助你学会TFileTime的用法，下面的代码将演示如何把TFileTime 和TDateTime相互转换：

```
function FileTimeToDateTime(FileTime: TFileTime): TDateTime;
var
  SysTime: TSystemTime;
begin
  if not FileTimeToSystemTime(FileTime, SysTime) then
    raise EConvertError.CreateFmt('FileTimeToSystemTime failed. ' +
      'Error code %d', [GetLastError]);
  with SysTime do
    Result := EncodeDate(wYear, wMonth, wDay) +
      EncodeTime(wHour, wMinute, wSecond, wMilliseconds)
end;

function DateTimeToFileTime(DateTime: TDateTime): TFileTime;
var
  SysTime: TSystemTime;
begin
  with SysTime do
  begin
    DecodeDate(DateTime, wYear, wMonth, wDay);
    DecodeTime(DateTime, wHour, wMinute, wSecond, wMilliseconds);
    wDayOfWeek := DayOfWeek(DateTime);
  end;
  if not SystemTimeToFileTime(SysTime, Result) then
    raise EConvertError.CreateFmt('SystemTimeToFileTime failed. ' +
      + 'Error code %d', [GetLastError]);
end;
```

注意 请记住函数GetThreadTimes()只适用于Windows NT/2000。如果你在Windows 95/98下调用，它总是返回False。非常不幸，Windows 95/98没有提供获取线程时间的手段。

11.3 管理多线程

正如前面介绍的，尽管线程能够解决许多的问题，但同时它又给我们带来了许多的问题。其中主要的问题是：对全局变量或句柄这样的全局资源如何访问？另外，当必须确保一个线程中的某些事件要在另一个线程中的其他事件之前(或之后)发生时，该怎么办？在本节里你将学习通过使用由Delphi提供的线程局部存储和API为线程提供同步的方法。

11.3.1 线程局部存储

由于每个线程都代表了一个不同的执行路径，因此，最好有一种只限于一个线程内部使用的数据

China-pub.com
下载

存储方式。要实现上述目的有3种方式：第一种也是最简单的一种方式是局部变量(基于栈)。由于每个线程都在各自的栈中，各个线程将都有一套局部变量的副本，这样，就不会相互影响。第二种方式是把有关信息保存在各自的线程对象中。第三种方式是用Object Pascal的关键字threadvar来声明变量，以利用操作系统级的线程局部存储。

1. 把信息保存在TThread派生对象中

将相关信息保存在TThread派生对象中，这是一种线程局部存储可选用的技术。相对于使用关键字threadvar的技术，这种方式更加简单、更加有效。例如，你可以在一个线程对象中加入下列信息：

```
type
  TMyThread = class(TThread)
  private
    FLocalInt: Integer;
    FLocalStr: String;
    .
    .
    .
  end;
```

提示 由于访问线程对象中的数据比访问线程局部变量要快10倍，因此，你应当尽可能地把线程专用的信息保存在线程对象中。对于那些只在过程或函数的生存期有意义的变量，应当把它们声明为局部变量。

2. threadvar：API线程局部存储

在前面我们了解到：虽然对于局部变量，在每个线程中都一个副本，然而应用程序的全局变量是被所有线程所共享的。例如，假设现在有一个用于设置和显示一个全局变量的值的过程。如果传递一个字符串给它，就设置这个全局变量；如果传递一个空字符串给它，就显示这个全局变量的值。这个过程定义如下：

```
var
  GlobalStr: String;
procedure SetShowStr(const S: String);
begin
  if S = '' then
    MessageBox(0, PChar(GlobalStr), 'The string is...', MB_OK)
  else
    GlobalStr := S;
end;
```

如果在某段代码中调用这个过程，则有可能出现一些问题。因为在调用它时，可以先设置而后显示。可问题是，有可能有两个或更多的线程存在。当一个线程调用此过程来设置字符串时，而另一个线程也可能调用了此过程来设置GlobalStr变量的值。当操作系统把CPU时间又分配给前一个线程时，该线程所设置的值有可能已经令人失望地丢失。

为解决此类问题，Win32提供了一种称为线程局部存储的方式，它能使你在第一个运行的线程中创建一个全局变量的拷贝。Delphi利用关键字threadvar封装此功能。在threadvar关键字下你可以声明任何局部存储的变量。下面是全局变量GlobalStr的声明：

```
threadvar
  GlobalStr: String;
```

在清单11-3中演示了线程局部存储的用法。在程序的主窗体上有一个按钮，单击此按钮，就设置并显示GlobalStr变量的值。然后再创建一个线程，GlobalStr变量的值又被设置并显示。在创建了一个线程后，从主线程再次设置并显示GlobalStr变量的值。

先后用var和threadvar来声明GlobalStr变量并运行此程序。你会在输出中看到不同。

China-pub.com 下载

清单11-3 MAIN.PAS

```
Done. -sunit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;
implementation

{$R *.DFM}

{ NOTE: Change GlobalStr from var to threadvar to see difference }
var
//threadvar
  GlobalStr: string;

type
  TTLSThread = class(TThread)
  private
    FNewStr: String;
  protected
    procedure Execute; override;
  public
    constructor Create(const ANewStr: String);
  end;

procedure SetShowStr(const S: String);
begin
  if S = '' then
    MessageBox(0, PChar(GlobalStr), 'The string is...', MB_OK)
  else
    GlobalStr := S;
end;

constructor TTLSThread.Create(const ANewStr: String);
begin
  FNewStr := ANewStr;
  inherited Create(False);
end;

procedure TTLSThread.Execute;
begin
```

China-pub.com
下载


```
    FreeOnTerminate := True;
    SetShowStr(FNewStr);
    SetShowStr('');
  end;

  procedure TMainForm.Button1Click(Sender: TObject);
  begin
    SetShowStr('Hello world');
    SetShowStr('');
    TTLSThread.Create('Dilbert');
    Sleep(100);
    SetShowStr('');
  end;

  end.
```

注意　演示程序中，在创建了线程之后，调用了一个Win32 API过程Sleep()。此过程声明如下：

procedure Sleep(dwMilliseconds:DWORD); stdcall;

Sleep()过程用来告诉操作系统，当前的线程在参数dwMilliseconds指定的时间内不需要分配任何CPU时间。插入这个调用是使很多的任务在发生时，使执行哪个线程有一些随机性。

通常，可以把参数dwMilliseconds设为0。尽管，这并没有使当前的线程真的“睡眠”，但它使操作系统把CPU时间分给了其他优先级相等或更高的线程。

要小心Sleep()神秘的时间调整问题。Sleep()可能会使你的机器出现特别的问题。这种问题在另一台机器上可能无法再现。

11.3.2　线程同步

当有多个线程的时候，经常需要去同步这些线程以访问同一个数据或资源。例如，假设有一个程序，其中一个线程用于把文件读到内存，而另一个线程用于统计文件中的字符数。当然，在把整个文件调入内存之前，统计它的计数是没有意义的。但是，由于每个操作都有自己的线程，操作系统会把两个线程当作是互不相干的任务分别执行，这样就可能在没有把整个文件装入内存时统计字数。为解决此问题，你必须使两个线程同步工作。

存在一些线程同步地址的问题， Win32提供了许多线程同步的方式。在本节你将看到使用临界区、互斥、信号量和事件来解决线程同步的问题。

为了检验这些技术，首先来看一个需要用线程同步来解决的问题。假设有一个整数数组，需要按升序赋初值。现在要在第一遍把这个数组赋初值为 1至128，第二遍将此数组赋初值为128至255，然后结果显示在列表框中。要用两个线程来分别进行初始化。清单 11-4列出了程序代码。

清单11-4　在两个线程中对数组进行初始化

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
```

China-pub.com 下载

```
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  // return global var
  Inc(NextNumber);       // inc global var
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  OnTerminate := MainForm.ThreadsDone;
  for i := 1 to MaxSize do
  begin
    GlobalArray[i] := GetNextNumber;  // set array element
    Sleep(5);                         // let thread intertwine
  end;
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then        // make sure both threads finished
    for i := 1 to MaxSize do
      { fill listbox with array contents }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
```

China-pub.com 下载


```
  TFooThread.Create(False);  // create threads
  TFooThread.Create(False);
end;

end.
```

因为两个线程同时运行，同一个数组在两个线程中被初始化会出现什么呢？你可以从图11-4中看到结果。

这个问题的解决方案是：当两个线程访问这个全局数组时，为防止它们同时执行，需要使用线程的同步。这样，你就会得到一组合理的数值。

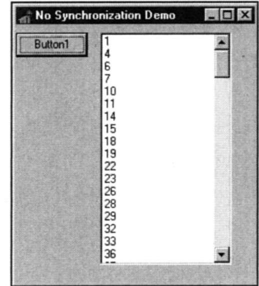

图11-4　线程没有同步的输出

1. 临界区

临界区是一种最直接的线程同步方式。所谓临界区，就是一次只能由一个线程来执行的一段代码。如果把初始化数组的代码放在临界区内，另一个线程在第一个线程处理完之前是不会被执行的。

在使用临界区之前，必须使用InitializeCriticalSection()过程来初始化它。其声明如下：

```
procedure InitializeCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
```

lpCriticalSection参数是一个TRTLCriticalSection类型的记录，并且是变参。至于TRTLCriticalSection是如何定义的，这并不重要，因为很少需要查看这个记录中的具体内容。只需要在lpCriticalSection中传递未初始化的记录，InitializeCriticalSection()过程就会填充这个记录。

注意　Microsoft故意隐瞒了TRTLCriticalSection的细节。因为，其内容在不同的硬件平台上是不同的。在基于Intel的平台上，TRTLCriticalSection包含一个计数器、一个指示当前线程句柄的域和一个系统事件的句柄。在Alpha平台上，计数器被替换为一种Alpha-CPU 数据结构，称为spinlock。

在记录被填充后，我们就可以开始创建临界区了。这时我们需要用 EnterCriticalSection()和LeaveCriticalSection()来封装代码块。这两个过程的声明如下：

```
procedure EnterCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
procedure LeaveCriticalSection(var lpCriticalSection:
  TRTLCriticalSection); stdcall;
```

正如你所想的，参数lpCriticalSection就是由InitializeCriticalSection()填充的记录。

当你不需要TRTLCriticalSection记录时，应当调用DeleteCriticalSection()过程，下面是它的声明：

procedure DeleteCriticalSection(var lpCriticalSection: TRTLCriticalSection);stdcall;

清单11-5演示了利用临界区来同步数组初始化线程的技术。

清单11-5　使用临界区

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
```

China-pub.com 下载

```
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;
implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  CS: TRTLCriticalSection;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  // return global var
  inc(NextNumber);       // inc global var
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  OnTerminate := MainForm.ThreadsDone;
  EnterCriticalSection(CS);             // CS begins here
  for i := 1 to MaxSize do
  begin
    GlobalArray[i] := GetNextNumber;    // set array element
    Sleep(5);                           // let thread intertwine
  end;
  LeaveCriticalSection(CS);             // CS ends here
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  inc(DoneFlags);
  if DoneFlags = 2 then
  begin // make sure both threads finished
    for i := 1 to MaxSize do
      { fill listbox with array contents }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
```

China-pub.com 下载


```
      DeleteCriticalSection(CS);
    end;
  end;

  procedure TMainForm.Button1Click(Sender: TObject);
  begin
    InitializeCriticalSection(CS);
    TFooThread.Create(False);   // create threads
    TFooThread.Create(False);
  end;

  end.
```

在第一个线程调用了EnterCriticalSection()之后，所有别的线程就不能再进入代码块。下一个线程要等第一个线程调用LeaveCriticalSection()后才能被唤醒。图11-5显示了同步后的程序的输出结果。

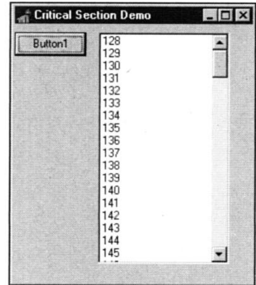

图11-5　同步的数组初始化的输出

2. 互斥

互斥非常类似于临界区，除了两个关键的区别：首先，互斥可用于跨进程的线程同步。其次，互斥能被赋予一个字符串名字，并且通过引用此名字创建现有互斥对象的附加句柄。

提示　临界区与事件对象(比如互斥对象)的最大的区别是在性能上。临界区在没有线程冲突时，要用10~15个时间片，而事件对象由于涉及到系统内核要用400~600个时间片。

可以调用函数CreateMutex()来创建一个互斥量。下面是函数的声明：

```
function CreateMutex(lpMutexAttributes: PSecurityAttributes;
  bInitialOwner: BOOL; lpName: PChar): THandle; stdcall;
```

lpMutexAttributes参数为一个指向TSecurityAttributtes记录的指针。此参数通常设为0，表示默认的安全属性。

bInitalOwner参数表示创建互斥对象的线程是否要成为此互斥对象的拥有者。当此参数为 False时，表示互斥对象没有拥有者。

lpName参数指定互斥对象的名称。设为nil表示无命名，如果参数不是设为nil，函数会搜索是否有同名的互斥对象存在。如果有，函数就会返回同名互斥对象的句柄。否则，就新创建一个互斥对象并返回其句柄。

当使用完互斥对象时，应当调用CloseHandle()来关闭它。

清单11-6演示了利用互斥技术来使两个进程对一个数组的初始化同步。

清单11-6　使用互斥同步

```
Done. -sunit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
```

China-pub.com 下载

```
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  hMutex: THandle = 0;

function GetNextNumber: Integer;
begin
  Result := NextNumber;  // return global var
  Inc(NextNumber);       // inc global var
end;

procedure TFooThread.Execute;
var
  i: Integer;
begin
  FreeOnTerminate := True;
  OnTerminate := MainForm.ThreadsDone;
  if WaitForSingleObject(hMutex, INFINITE) = WAIT_OBJECT_0 then
  begin
    for i := 1 to MaxSize do
    begin
      GlobalArray[i] := GetNextNumber;  // set array element
      Sleep(5);                         // let thread intertwine
    end;
  end;
  ReleaseMutex(hMutex);
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then     // make sure both threads finished
  begin
    for i := 1 to MaxSize do
      { fill listbox with array contents }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    CloseHandle(hMutex);
```

China-pub.com 下载


```
    end;
  end;

  procedure TMainForm.Button1Click(Sender: TObject);
  begin
    hMutex := CreateMutex(nil, False, nil);
    TFooThread.Create(False);  // create threads
    TFooThread.Create(False);
  end;

  end.
```

你将注意到，在程序中使用WaitForSingleObject()来防止其他线程进入同步区域的代码。此函数声明如下：

```
function WaitForSingleObject(hHandle: THandle; dwMilliseconds: DWORD):
  DWORD; stdcall;
```

这个函数可以使当前线程在dwMilliseconds指定的时间内睡眠，直到hHandle参数指定的对象进入发信号状态为止。一个互斥对象不再被线程拥有时，它就进入发信号状态。当一个进程要终止时，它就进入发信号状态。dwMilliseconds参数可以设为0，这意味着只检查hHandle参数指定的对象是否处于发信号状态，而后立即返回。dwMilliseconds参数设为INFINITE，表示如果信号不出现将一直等下去。这个函数的返回值列在表11-3中。

表11-3 WaitFor SingleObject()函数使用的返回值

返 回 值	含 义
WAIT_ABANDONED	指定的对象是互斥对象，并且拥有这个互斥对象的线程在没有释放此对象之前就已终止。此时就称互斥对象被抛弃。这种情况下，这个互斥对象归当前线程所有，并把它设为非发信号状态
WAIT_OBJECT_0	指定的对象处于发信号状态
WAIT_TIMEOUT	等待的时间已过，对象仍然是非发信号状态

再次声明，当一个互斥对象不再被一个线程所拥有，它就处于发信号状态。此时首先调用WaitForSingleObject()函数的线程就成为该互斥对象的拥有者，此互斥对象设为不发信号状态。当线程调用ReleaseMutex()函数并传递一个互斥对象的句柄作为参数时，这种拥有关系就被解除，互斥对象重新进入发信号状态。

注意 除WaitForSingleObject()函数外，你还可以使用WaitForMultipleObject()和MsgWaitForMultipleObject()函数，它们可以等待几个对象变为发信号状态。这两个函数的详细情况请看Win32 API联机文档。

3. 信号量

另一种使线程同步的技术是使用信号量对象。它是在互斥的基础上建立的，但信号量增加了资源计数的功能，预定数目的线程允许同时进入要同步的代码。可以用CreateSemaphore()来创建一个信号量对象，其声明如下：

```
function CreateSemaphore(lpSemaphoreAttributes: PSecurityAttributes;
  lInitialCount, lMaximumCount: Longint; lpName: PChar): THandle;stdcall;
```

和CreateMutex()函数一样，CreateSemaphore()的第一个参数也是一个指向TSecurityAttributes记录的指针，此参数的缺省值可以设为nil。

lInitialCount参数用来指定一个信号量的初始计数值，这个值必须在0和lMaximumCount之间。此参数大于0，就表示信号量处于发信号状态。当调用WaitForSingleObject()函数(或其他函数)时，此计

China-pub.com 下载

数值就减1。当调用ReleaseSemaphore()时，此计数值加1。

参数lMaximumCount指定计数值的最大值。如果这个信号量代表某种资源，那么这个值代表可用资源总数。

参数lpName用于给出信号量对象的名称，它类似于CreateMutex()函数的lpName参数。

清单11-7中是使用信号量技术来同步初始化数组的代码

清单11-7 使用信号量技术同步

```
Done. -sunit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    procedure ThreadsDone(Sender: TObject);
  end;

  TFooThread = class(TThread)
  protected
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
  MaxSize = 128;

var
  NextNumber: Integer = 0;
  DoneFlags: Integer = 0;
  GlobalArray: array[1..MaxSize] of Integer;
  hSem: THandle = 0;

function GetNextNumber: Integer;
begin
  Result := NextNumber;   // return global var
  Inc(NextNumber);        // inc global var
end;

procedure TFooThread.Execute;
var
  i: Integer;
  WaitReturn: DWORD;
```

China-pub.com 下载


```
begin
  OnTerminate := MainForm.ThreadsDone;
  WaitReturn := WaitForSingleObject(hSem, INFINITE);
  if WaitReturn = WAIT_OBJECT_0 then
  begin
    for i := 1 to MaxSize do
    begin
      GlobalArray[i] := GetNextNumber;  // set array element
      Sleep(5);                         // let thread intertwine
    end;
  end;
  ReleaseSemaphore(hSem, 1, nil);
end;

procedure TMainForm.ThreadsDone(Sender: TObject);
var
  i: Integer;
begin
  Inc(DoneFlags);
  if DoneFlags = 2 then      // make sure both threads finished
  begin
    for i := 1 to MaxSize do
      { fill listbox with array contents }
      Listbox1.Items.Add(IntToStr(GlobalArray[i]));
    CloseHandle(hSem);
  end;
end;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  hSem := CreateSemaphore(nil, 1, 1, nil);
  TFooThread.Create(False);  // create threads
  TFooThread.Create(False);
end;

end.
```

因为只允许一个线程进入要同步的代码，所以信号量的最大计数值(lMaximumCount)要设为1。

ReleaseSemaphore()函数将使信号量对象的计数加1。请注意此函数比ReleaseMutex()更复杂。ReleaseSemaphore()函数声明如下：

```
function ReleaseSemaphore(hSemaphore: THandle; lReleaseCount: Longint;
    lpPreviousCount: Pointer): BOOL; stdcall;
```

lReleaseCount参数用于指定每次使计数值加多少。如果参数lpPreviousCount不为nil,原有的计数值将存储在lpPreviousCount里。信号量对象并不属于某个线程。例如，假设一个信号量对象的最大计数值是10，并且有10个线程调用WaitForSingleObject(),计数值将减至0。只要有一个线程调用ReleaseSemaphore()，并且把lReleaseCount参数设为10。这时，计数值又恢复为10，同时10个线程又恢复发信号状态。不过，此函数使调试变得困难，使用时要小心。

记住，最后一定要调用CloseHandle()函数来释放由CreateSemaphore()创建的信号量对象的句柄。

11.4 一个多线程的示范程序

为了演示TThread对象在现实程序设计中的用法，这一节将创建一个文件搜索程序，此程序用一个专门的线程来搜索文件。程序为DelSrch,表示Delphi Search，程序的主窗体如图11-6所示。

China-pub.com 下载

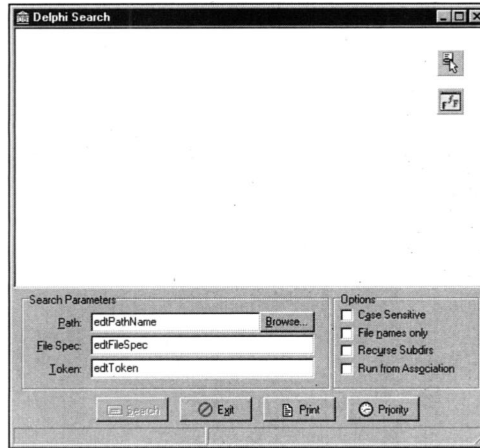

图11-6 DelSrch项目的主窗体

此程序是这样工作的。用户先选择一个路径，然后指定要搜索的文件类型。用户也可以在适当的编辑框内输入要搜索的内容。在窗体的右下部分有一组复选框，用于设置搜索选项。当用户单击Search按钮时，程序将创建一个线程，并把用户输入的信息传递给新创建的线程对象。当线程找到指定的内容时，在列表框中将显示有关信息。如果用户在列表框中双击一个文件，就启动一个文本编辑器或其他程序来打开这个文件。

尽管这个程序的功能很全，但我们主要讲解其与多线程有关的功能。

11.4.1 用户界面

这个程序的主单元文件叫Main.pas，其源代码列在清单11-8中。此单元负责管理用户界面，包含拥有者绘制列表框、打开一个文件观看器、创建一个线程、打印列表框中的信息、读写一个INI文件的逻辑。

清单11-8 DelSrch项目的Main.pas单元

```
unit SrchU;

interface

uses Classes, StdCtrls;

type
  TSearchThread = class(TThread)
  private
    LB: TListbox;
    CaseSens: Boolean;
    FileNames: Boolean;
    Recurse: Boolean;
    SearchStr: string;
    SearchPath: string;
    FileSpec: string;
    AddStr: string;
    FSearchFile: string;
    procedure AddToList;
    procedure DoSearch(const Path: string);
    procedure FindAllFiles(const Path: string);
```

China-pub.com 下载


```
    procedure FixControls;
    procedure ScanForStr(const FName: string; var FileStr: string);
    procedure SearchFile(const FName: string);
    procedure SetSearchFile;
  protected
    procedure Execute; override;
  public
    constructor Create(CaseS, FName, Rec: Boolean; const Str, SPath,
      FSpec: string);
    destructor Destroy; override;
  end;

implementation

uses SysUtils, StrUtils, Windows, Forms, Main;

constructor TSearchThread.Create(CaseS, FName, Rec: Boolean; const Str,
  SPath, FSpec: string);
begin
  CaseSens := CaseS;
  FileNames := FName;
  Recurse := Rec;
  SearchStr := Str;
  SearchPath := AddBackSlash(SPath);
  FileSpec := FSpec;
  inherited Create(False);
end;

destructor TSearchThread.Destroy;
begin
  FSearchFile := '';
  Synchronize(SetSearchFile);
  Synchronize(FixControls);
  inherited Destroy;
end;

procedure TSearchThread.Execute;
begin
  FreeOnTerminate := True;     // set up all the fields
  LB := MainForm.lbFiles;
  Priority := TThreadPriority(MainForm.SearchPri);
  if not CaseSens then SearchStr := UpperCase(SearchStr);
  FindAllFiles(SearchPath);      // process current directory
  if Recurse then                // if subdirs, then...
    DoSearch(SearchPath);        // recurse, otherwise...
end;

procedure TSearchThread.FixControls;
{ Enables controls in main form. Must be called through Synchronize }
begin
  MainForm.EnableSearchControls(True);
end;

procedure TSearchThread.SetSearchFile;
{ Updates status bar with filename. Must be called through Synchronize }
begin
  MainForm.StatusBar.Panels[1].Text := FSearchFile;
```

China-pub.com 下载

```
end;

procedure TSearchThread.AddToList;
{ Adds string to main listbox. Must be called through Synchronize }
begin
  LB.Items.Add(AddStr);
end;

procedure TSearchThread.ScanForStr(const FName: string; var FileStr: string);
{ Scans a FileStr of file FName for SearchStr }
var
  Marker: string[1];
  FoundOnce: Boolean;
  FindPos: integer;
begin
  FindPos := Pos(SearchStr, FileStr);
  FoundOnce := False;
  while (FindPos <> 0) and not Terminated do
  begin
    if not FoundOnce then
    begin
      { use ":" only if user doesn't choose "filename only" }
      if FileNames then
        Marker := ''
      else
        Marker := ':';
      { add file to listbox }
      AddStr := Format('File %s%s', [FName, Marker]);
      Synchronize(AddToList);
      FoundOnce := True;
    end;
    { don't search for same string in same file if filenames only }
    if FileNames then Exit;

    { Add line if not filename only }
    AddStr := GetCurLine(FileStr, FindPos);
    Synchronize(AddToList);
    FileStr := Copy(FileStr, FindPos + Length(SearchStr), Length(FileStr));
    FindPos := Pos(SearchStr, FileStr);
  end;
end;

procedure TSearchThread.SearchFile(const FName: string);
{ Searches file FName for SearchStr }
var
  DataFile: THandle;
  FileSize: Integer;
  SearchString: string;
begin
  FSearchFile := FName;
  Synchronize(SetSearchFile);
  try
    DataFile := FileOpen(FName, fmOpenRead or fmShareDenyWrite);
    if DataFile = 0 then raise Exception.Create('');
    try
      { set length of search string }
      FileSize := GetFileSize(DataFile, nil);
```

China-pub.com
下载


```
        SetLength(SearchString, FileSize);
        { Copy file data to string }
        FileRead(DataFile, Pointer(SearchString)^, FileSize);
      finally
        CloseHandle(DataFile);
      end;
      if not CaseSens then SearchString := UpperCase(SearchString);
      ScanForStr(FName, SearchString);
    except
      on Exception do
      begin
        AddStr := Format('Error reading file: %s', [FName]);
        Synchronize(AddToList);
      end;
    end;
  end;

procedure TSearchThread.FindAllFiles(const Path: string);
{ procedure searches Path subdir for files matching filespec }
var
  SR: TSearchRec;
begin
  { find first file matching spec }
  if FindFirst(Path + FileSpec, faArchive, SR) = 0 then
    try
      repeat
        SearchFile(Path + SR.Name);              // process file
      until (FindNext(SR) <> 0) or Terminated; // find next file
    finally
      SysUtils.FindClose(SR);                  // clean up
    end;
end;

procedure TSearchThread.DoSearch(const Path: string);
{ procedure recurses through a subdirectory tree starting at Path }
var
  SR: TSearchRec;
begin
  { look for directories }
  if FindFirst(Path + '*.*', faDirectory, SR) = 0 then
    try
      repeat
        { if it's a directory and not '.' or '..' then... }
        if ((SR.Attr and faDirectory) <> 0) and (SR.Name[1] <> '.') and
          not Terminated then
        begin
          FindAllFiles(Path + SR.Name + '\');  // process directory
          DoSearch(Path + SR.Name + '\');      // recurse
        end;
      until (FindNext(SR) <> 0) or Terminated;     // find next directory
    finally
      SysUtils.FindClose(SR);                      // clean up
    end;
end;

end.
```

在本单元中有几件事值提一下。首先，你会注意到在程序使用了 PrintStrings()过程来打印一个

China-pub.com 下载

TString对象。为了完成这项工作，此过程利用了 Delphi的AssignPrn()标准过程，它被用来把一个TextFile类型的变量指派给打印机。这样，写到TextFile中的内容会自动输出到打印机。在打印完毕后，你必须调用CloseFile()来关闭到打印机的连接。

对Win32 API ShellExecte()的使用也很有趣，它主要用来执行一个启动一个在列表框中显示的文件的观看器。不过，它不仅可以调用一个可执行程序，而且还可以按扩展名打开一个文件。例如，如果用ShellExecte()来打开一个以.pas为扩展名的文件，它将自动装入Delphi并打开文件。

提示 如果ShellExecte()返回一个值表明有错误，程序将调用RaiseLastWin32Error()。这个过程在SysUtils单元中，它将调用API函数GetLastError()和Delphi的SysErrorMessage()以获得关于错误的更详细的信息。之后，把错误信息格式化为一个字符串。如果想让用户获得详细的 API级错误信息，可以调用RaiseLastWin32Error()。

11.4.2 搜索线程

在SrchU.pas中包含了一个搜索引擎，其源代码列在清单11-9中。这个单元做了一些有意义的事情：把整个文件拷贝到一个字符串中，递归子目录，与主窗体通信等。

清单11-9 SrchU.pas单元

```
unit SrchU;

interface

uses Classes, StdCtrls;

type
  TSearchThread = class(TThread)
  private
    LB: TListbox;
    CaseSens: Boolean;
    FileNames: Boolean;
    Recurse: Boolean;
    SearchStr: string;
    SearchPath: string;
    FileSpec: string;
    AddStr: string;
    FSearchFile: string;
    procedure AddToList;
    procedure DoSearch(const Path: string);
    procedure FindAllFiles(const Path: string);
    procedure FixControls;
    procedure ScanForStr(const FName: string; var FileStr: string);
    procedure SearchFile(const FName: string);
    procedure SetSearchFile;
  protected
    procedure Execute; override;
  public
    constructor Create(CaseS, FName, Rec: Boolean; const Str, SPath,
      FSpec: string);
    destructor Destroy; override;
  end;

implementation
```

China-pub.com
下载


```
uses SysUtils, StrUtils, Windows, Forms, Main;

constructor TSearchThread.Create(CaseS, FName, Rec: Boolean; const Str,
  SPath, FSpec: string);
begin
  CaseSens := CaseS;
  FileNames := FName;
  Recurse := Rec;
  SearchStr := Str;
  SearchPath := AddBackSlash(SPath);
  FileSpec := FSpec;
  inherited Create(False);
end;

destructor TSearchThread.Destroy;
begin
  FSearchFile := '';
  Synchronize(SetSearchFile);
  Synchronize(FixControls);
  inherited Destroy;
end;

procedure TSearchThread.Execute;
begin
  FreeOnTerminate := True;      // set up all the fields
  LB := MainForm.lbFiles;
  Priority := TThreadPriority(MainForm.SearchPri);
  if not CaseSens then SearchStr := UpperCase(SearchStr);
  FindAllFiles(SearchPath);      // process current directory
  if Recurse then                // if subdirs, then...
    DoSearch(SearchPath);        // recurse, otherwise...
end;

procedure TSearchThread.FixControls;
{ Enables controls in main form. Must be called through Synchronize }
begin
  MainForm.EnableSearchControls(True);
end;

procedure TSearchThread.SetSearchFile;
{ Updates status bar with filename. Must be called through Synchronize }
begin
  MainForm.StatusBar.Panels[1].Text := FSearchFile;
end;

procedure TSearchThread.AddToList;
{ Adds string to main listbox. Must be called through Synchronize }
begin
  LB.Items.Add(AddStr);
end;

procedure TSearchThread.ScanForStr(const FName: string;
  var FileStr: string);
{ Scans a FileStr of file FName for SearchStr }
var
  Marker: string[1];
  FoundOnce: Boolean;
```

China-pub.com 下载

```
  FindPos: integer;
begin
  FindPos := Pos(SearchStr, FileStr);
  FoundOnce := False;
  while (FindPos <> 0) and not Terminated do
  begin
    if not FoundOnce then
    begin
      { use ":" only if user doesn't choose "filename only" }
      if FileNames then
        Marker := ''
      else
        Marker := ':';
      { add file to listbox }
      AddStr := Format('File %s%s', [FName, Marker]);
      Synchronize(AddToList);
      FoundOnce := True;
    end;
    { don't search for same string in same file if filenames only }
    if FileNames then Exit;

    { Add line if not filename only }
    AddStr := GetCurLine(FileStr, FindPos);
    Synchronize(AddToList);
    FileStr := Copy(FileStr, FindPos + Length(SearchStr),
      Length(FileStr));
    FindPos := Pos(SearchStr, FileStr);
  end;
end;

procedure TSearchThread.SearchFile(const FName: string);
{ Searches file FName for SearchStr }
var
  DataFile: THandle;
  FileSize: Integer;
  SearchString: string;
begin
  FSearchFile := FName;
  Synchronize(SetSearchFile);
  try
    DataFile := FileOpen(FName, fmOpenRead or fmShareDenyWrite);
    if DataFile = 0 then raise Exception.Create('');
    try
      { set length of search string }
      FileSize := GetFileSize(DataFile, nil);
      SetLength(SearchString, FileSize);
      { Copy file data to string }
      FileRead(DataFile, Pointer(SearchString)^, FileSize);
    finally
      CloseHandle(DataFile);
    end;
    if not CaseSens then SearchString := UpperCase(SearchString);
    ScanForStr(FName, SearchString);
  except
    on Exception do
    begin
      AddStr := Format('Error reading file: %s', [FName]);
      Synchronize(AddToList);
```

China-pub.com
下载


```
    end;
  end;
end;

procedure TSearchThread.FindAllFiles(const Path: string);
{ procedure searches Path subdir for files matching filespec }
var
  SR: TSearchRec;
begin
  { find first file matching spec }
  if FindFirst(Path + FileSpec, faArchive, SR) = 0 then
    try
      repeat
        SearchFile(Path + SR.Name);               // process file
      until (FindNext(SR) <> 0) or Terminated; // find next file
    finally
      SysUtils.FindClose(SR);                    // clean up
    end;
end;

procedure TSearchThread.DoSearch(const Path: string);
{ procedure recurses through a subdirectory tree starting at Path }
var
  SR: TSearchRec;
begin
  { look for directories }
  if FindFirst(Path + '*.*', faDirectory, SR) = 0 then
    try
      repeat
        { if it's a directory and not '.' or '..' then... }
        if ((SR.Attr and faDirectory) <> 0) and (SR.Name[1] <> '.') and
          not Terminated then
        begin
          FindAllFiles(Path + SR.Name + '\');  // process directory
          DoSearch(Path + SR.Name + '\');      // recurse
        end;
      until (FindNext(SR) <> 0) or Terminated;      // find next directory
    finally
      SysUtils.FindClose(SR);                       // clean up
    end;
end;

end.
```

在创建后，这个线程首先调用了FindAllFiles()。而FindAllFiles()又调用FindFirst()和FindNext()在当前目录中搜索指定的文件。如果用户指定要包含子目录，就会调用 DoSearch()。每找到一个子目录，就会调用FindAllFiles()在这个子目录中搜索匹配的文件。

提示 在DoSearch()中使用了递归的算法，这是为了遍历所有的子目录。不过，递归算法非常难于调试，程序员必须确保递归的代码能够正确运行。这个方法值得保存，以便在以后的应用程序中使用。

你可能已注意到，为了处理每个文件， DoSearch()使用了TMemMapFile对象用于在一个文件中搜索指定的内容。TMemMapFile封装了Win32的内存映射文件。在第12章“文件处理”中我们将讨论此对象。在这里你只需要知道， TMemMapFile能够把一个文件的内容映射到内存中。 DoSearch()的算法是这样的：

China-pub.com 下载

1) 当FindAllFiles()找到一个匹配的文件时，就调用SearchFile(),并且把该文件的内容复制到一个字符串中。

2) 调用ScanForStr()在这个字符串中搜索一个特定的文本。

3) 如果找到了特定的文本，文件名以及文本所在的行就加到了列表框中。如果用户选中了 File Name Only 复选框，则只把文件名加到列表框中。

注意，TSearchThread对象的所有方法都要定期检查StopIt标志和Terminated标志的状态。前者用于表明线程是否停止，后者表示线程是否终止。

注意 一定要记住使用任何线程对象的方法来操作用户界面时，一定要通过Synchronize()方法来调用，或者通过消息来控制用户界面。

11.4.3 调整优先级

现在在DelSrch中加入新特点，使它可以允许用户动态地调整线程的优先级。图11-7显示了一个窗体，这个窗体的作用就是调整线程的优先级。清单11-10列出了这个窗体的对应单元文件。

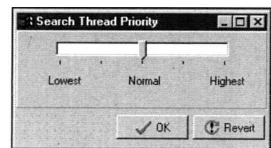

图11-7 从DelSrch调整线程的优先级

本单元的代码很简单。它所要做的只是根据跟踪条的位置来设置SearchPri变量的值。如果线程正在运行，就直接设置该线程的优先级。由于TThreadPriority是一个枚举类型，跟踪条的每个位置都对应着一个从1到5的序号。

清单11-10 PriU.Pas单元

```
unit PriU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, Buttons, ExtCtrls;

type
  TThreadPriWin = class(TForm)
    tbrPriTrackBar: TTrackBar;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    btnOK: TBitBtn;
    btnRevert: TBitBtn;
    Panel1: TPanel;
    procedure tbrPriTrackBarChange(Sender: TObject);
    procedure btnRevertClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure FormShow(Sender: TObject);
    procedure btnOKClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    OldPriVal: Integer;
  public
    { Public declarations }
  end;
```

China-pub.com 下载


```
var
  ThreadPriWin: TThreadPriWin;

implementation

{$R *.DFM}

uses Main, SrchU;

procedure TThreadPriWin.tbrPriTrackBarChange(Sender: TObject);
begin
  with MainForm do
  begin
    SearchPri := tbrPriTrackBar.Position;
    if Running then
      SearchThread.Priority := TThreadPriority(tbrPriTrackBar.Position);
  end;
end;

procedure TThreadPriWin.btnRevertClick(Sender: TObject);
begin
  tbrPriTrackBar.Position := OldPriVal;
end;

procedure TThreadPriWin.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Action := caHide;
end;

procedure TThreadPriWin.FormShow(Sender: TObject);
begin
  OldPriVal := tbrPriTrackBar.Position;
end;

procedure TThreadPriWin.btnOKClick(Sender: TObject);
begin
  Close;
end;

procedure TThreadPriWin.FormCreate(Sender: TObject);
begin
  tbrPriTrackBarChange(Sender);         // initialize thread priority
end;

end.
```

11.5　多线程与数据库

尽管要到第28章“编写桌面数据库应用程序”才开始讨论数据库的编程问题，但在这一节里，我们还是要介绍一些有关多线程在数据库领域的应用。如果你对数据库编程不太熟悉，可以先阅读第28章。

在Win32下从事数据库开发，会经常需要在后台进行复杂的查询或执行存储过程。非常幸运，32位的Borland数据库引擎(BDE)完全支持这个功能。

通过TQuery组件可以在后台进行查询，但要注意下面两点：

China-pub.com 下载

- 每个线程化的查询必须在它自己的会话中。为此，需要把一个TSession组件加到窗体上，然后设置TQuery的SessionName属性为TSession的名称。这样，如果TQuery组件通过TDatabase组件连接数据库的话，必须保证每个会话所使用的TDatabase对象是唯一的。
- 当一个辅助线程打开了查询后，TQuery组件就不能与TDataSource组件有任何联系。否则，查询必须在主线程内进行。这是因为，TDataSource的作用是把查询的结果与用户界面相关联，而涉及到用户界面的操作必须在主线程内进行。

为了说明后台查询的技术，图11-8显示了一个叫BDEThrd的示范程序的主窗体。这个窗体允许你指定一个线程来进行查询,查询的结果显示在一个子窗体中。

此子窗体叫TQueryForm,如图11-9所示。注意，这个子窗体上也有TQuery、TDataSource、TSession、TDatabase和TDBGrid组件，也就是说，每个TQueryForm实例都拥有这些组件的实例。

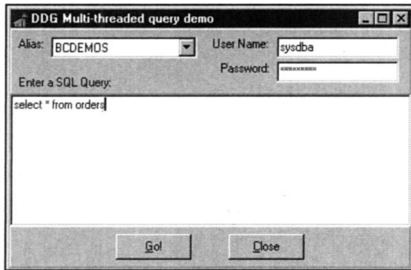

图11-8 BDEThrd的主窗体

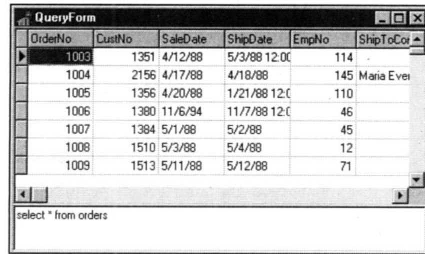

图11-9 BDEThrd的查询子窗体

清单11-11列出了这个程序的主窗体单元的代码。

清单11-11 Main.pas单元

```
Fixed. -sunit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, StdCtrls, ExtCtrls;

type
  TMainForm = class(TForm)
    pnlBottom: TPanel;
    pnlButtons: TPanel;
    GoButton: TButton;
    Button1: TButton;
    memQuery: TMemo;
    pnlTop: TPanel;
    Label1: TLabel;
    AliasCombo: TComboBox;
    Label3: TLabel;
    UserNameEd: TEdit;
    Label4: TLabel;
    PasswordEd: TEdit;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure GoButtonClick(Sender: TObject);
```

China-pub.com
下载


```
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses QryU, DB, DBTables;

var
  FQueryNum: Integer = 0;

procedure TMainForm.Button1Click(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.GoButtonClick(Sender: TObject);
begin
  Inc(FQueryNum);   // keep querynum unique
  { invoke new query }
  NewQuery(FQueryNum, memQuery.Lines, AliasCombo.Text, UserNameEd.Text,
    PasswordEd.Text);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  { fill drop-down list with BDE Aliases }
  Session.GetAliasNames(AliasCombo.Items);
end;

end.
```

从本单元可以看出，代码很简单。在AlisasCombo下列框内将由BDE别名来填充，这一步将OnCreate事件处理过程中调用TSession的GetAliasNames()来完成。当用户单击按钮Go!时，就调用NewQuery()开始一个新的查询。在下面要介绍的 QryU.pas单元里有NewQuery()代码。请注意，在每次按键过程中，NewQuery()将传递不同的FQueryNum参数。这是为了使每个查询线程所使用的会话对象和数据库对象的名称是唯一的。

清单11-12列出QryU.pas单元。

清单11-12 QryU.pas单元

```
unit QryU;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  Grids,
DBGrids, DB, DBTables, StdCtrls;
```

China-pub.com 下载

```
type
  TQueryForm = class(TForm)
    Query: TQuery;
    DataSource: TDataSource;
    Session: TSession;
    Database: TDatabase;
    dbgQueryGrid: TDBGrid;
    memSQL: TMemo;
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

procedure NewQuery(QryNum: integer; Qry: TStrings; const Alias, UserName,
  Password: string);

implementation

{$R *.DFM}
type
  TDBQueryThread = class(TThread)
  private
    FQuery: TQuery;
    FDataSource: TDataSource;
    FQueryException: Exception;
    procedure HookUpUI;
    procedure QueryError;
  protected
    procedure Execute; override;
  public
    constructor Create(Q: TQuery; D: TDataSource); virtual;
  end;

constructor TDBQueryThread.Create(Q: TQuery; D: TDataSource);
begin
  inherited Create(True);         // create suspended thread
  FQuery := Q;                    // set parameters
  FDataSource := D;
  FreeOnTerminate := True;
  Resume;                         // thread that puppy!
end;

procedure TDBQueryThread.Execute;
begin
  try
    FQuery.Open;                  // open the query
    Synchronize(HookUpUI);        // update UI from main thread
  except
    FQueryException := ExceptObject as Exception;
    Synchronize(QueryError);      // show exception from main thread
  end;
end;

procedure TDBQueryThread.HookUpUI;
begin
```

```
  FDataSource.DataSet := FQuery;
end;

procedure TDBQueryThread.QueryError;
begin
  Application.ShowException(FQueryException);
end;

procedure NewQuery(QryNum: integer; Qry: TStrings; const Alias, UserName,
  Password: string);
begin
  { Create a new Query form to show query results }
  with TQueryForm.Create(Application) do
  begin
    { Set a unique session name }
    Session.SessionName := Format('Sess%d', [QryNum]);
    with Database do
    begin
      { set a unique database name }
      DatabaseName := Format('DB%d', [QryNum]);
      { set alias parameter }
      AliasName := Alias;
      { hook database to session }
      SessionName := Session.SessionName;
      { user-defined username and password }
      Params.Values['USER NAME'] := UserName;
      Params.Values['PASSWORD'] := Password;
    end;
    with Query do
    begin
      { hook query to database and session }
      DatabaseName := Database.DatabaseName;
      SessionName := Session.SessionName;
      { set up the query strings }
      SQL.Assign(Qry);
    end;
    { display query strings in SQL Memo }
    memSQL.Lines.Assign(Qry);
    { show query form }
    Show;
    { open query in its own thread }
    TDBQueryThread.Create(Query, DataSource);
  end;
end;

procedure TQueryForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caFree;
end;

end.
```

在程序中， NewQuery()创建了一个新的TQueryForm子窗体实例，并且设置数据访问组件的属性，还给TDatabase组件和TSession组件命名。查询组件的SQL属性是由Qry参数传递的TStrings填充的，随后，查询线程就产生了。

TDBQueryThread对象本身的代码比较简单。它的构造器建立了几个实例变量。在它的 Execute()方

China-pub.com 下载

法中，打开了查询并通过Synchronize()调用了HookupUI()，用于把查询与数据源挂钩。其try...except结构通过Synchronize()从主线程环境中显示异常信息。

11.6 多线程与图形处理

在前面曾提到，VCL不应当被几个线程同时访问，但是，这句话并不完全正确。对于VCL中的图形对象来讲，可以让多个线程同时访问。这需要感谢新的 Lock() 和Unlock方法。有了它们，整个Graphics单元现在是线程安全的，包括 TCanvas、TPen、TBrush、TFont、TBitmap、TMetafile、TPicture 和TIcon类。

Lock()其实类似于前面介绍的临界区的EnterCriticalSection()函数，它的作用就是不让其他线程访问图形对象。当一个线程调用了Lock()后，这个线程就可以独占图形对象。其他想进入这部分代码的线程调用了Lock()后将处于睡眠状态直到这个线程调用了Unlock()，Unlock()调用LeaveCriticalSection以释放临界区并让其他等待着的线程(如果有的话)进入代码的受保护部分。下面一段代码演示了如何使用这些方法来控制访问TCanvas对象:

```
Form.Canvas.Lock;
// 在这里操纵TCanvas对象
Form.Canvas.Unlock;
```

为更进一步说明此功能，清单11-13列出了一个示范程序的代码，这个程序演示了多个线程是如何访问同一个窗体的画布的。

清单11-13 MTGraph程序的Main.pas单元

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,  Menus;

type
  TMainForm = class(TForm)
    MainMenu1: TMainMenu;
    Options1: TMenuItem;
    AddThread: TMenuItem;
    RemoveThread: TMenuItem;
    ColorDialog1: TColorDialog;
    Add10: TMenuItem;
    RemoveAll: TMenuItem;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure AddThreadClick(Sender: TObject);
    procedure RemoveThreadClick(Sender: TObject);
    procedure Add10Click(Sender: TObject);
    procedure RemoveAllClick(Sender: TObject);
  private
    ThreadList: TList;
  public
    { Public declarations }
  end;

  TDrawThread = class(TThread)
  private
    FColor: TColor;
```

China-pub.com
下载


```
    FForm: TForm;
  public
    constructor Create(AForm: TForm; AColor: TColor);
    procedure Execute; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

{ TDrawThread }

constructor TDrawThread.Create(AForm: TForm; AColor: TColor);
begin
  FColor := AColor;
  FForm := AForm;
  inherited Create(False);
end;

procedure TDrawThread.Execute;
var
  P1, P2: TPoint;

  procedure GetRandCoords;
  var
    MaxX, MaxY: Integer;
  begin
    // initialize P1 and P2 to random points within Form bounds
    MaxX := FForm.ClientWidth;
    MaxY := FForm.ClientHeight;
    P1.x := Random(MaxX);
    P2.x := Random(MaxX);
    P1.y := Random(MaxY);
    P2.y := Random(MaxY);
  end;

begin
  FreeOnTerminate := True;
  // thread runs until it or the application is terminated
  while not (Terminated or Application.Terminated) do
  begin
    GetRandCoords;             // initialize P1 and P2
    with FForm.Canvas do
    begin
      Lock;                    // lock canvas
      // only one thread at a time can execute the following code:
      Pen.Color := FColor;     // set pen color
      MoveTo(P1.X, P1.Y);      // move to canvas position P1
      LineTo(P2.X, P2.Y);      // draw a line to position P2
      // after the next line executes, another thread will be allowed
      // to enter the above code block
      Unlock;                  // unlock canvas
    end;
  end;
end;
```

China-pub.com 下载

```
{ TMainForm }

procedure TMainForm.FormCreate(Sender: TObject);
begin
  ThreadList := TList.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  RemoveAllClick(nil);
  ThreadList.Free;
end;

procedure TMainForm.AddThreadClick(Sender: TObject);
begin
  // add a new thread to the list... allow user to choose color
  if ColorDialog1.Execute then
    ThreadList.Add(TDrawThread.Create(Self, ColorDialog1.Color));
end;

procedure TMainForm.RemoveThreadClick(Sender: TObject);
begin
  // terminate the last thread in the list and remove it from list
  TDrawThread(ThreadList[ThreadList.Count - 1]).Terminate;
  ThreadList.Delete(ThreadList.Count - 1);
end;

procedure TMainForm.Add10Click(Sender: TObject);
var
  i: Integer;
begin
  // create 10 threads, each with a random color
  for i := 1 to 10 do
    ThreadList.Add(TDrawThread.Create(Self, Random(MaxInt)));
end;

procedure TMainForm.RemoveAllClick(Sender: TObject);
var
  i: Integer;
begin
  Cursor := crHourGlass;
  try
    for i := ThreadList.Count - 1 downto 0 do
    begin
      TDrawThread(ThreadList[i]).Terminate; // terminate thread
      TDrawThread(ThreadList[i]).WaitFor;   // make sure thread terminates
    end;
    ThreadList.Clear;
  finally
    Cursor:= crDefault;
  end;
end;

initialization
  Randomize;  // seed random number generator
end.
```

China-pub.com
下载

这个程序有一个包含四个菜单项的主菜单，如图11-10所示。第一项Add Thread用来创建一个新的TDrawThread实例，并在主窗体上画一些随机的直线。这一项可以被多次选择以创建更多的线程实例来访问主窗体。下一项是Remove Thread，用于删除被增加的线程中的最后一个。第三项是 Add 10，用于创建10个新的TDrawThread实例。最后，第四项是Remove All，用于终止和删除所有TDrawThread实例。

图11-10显示了10个线程同时在窗体上画线的样子。

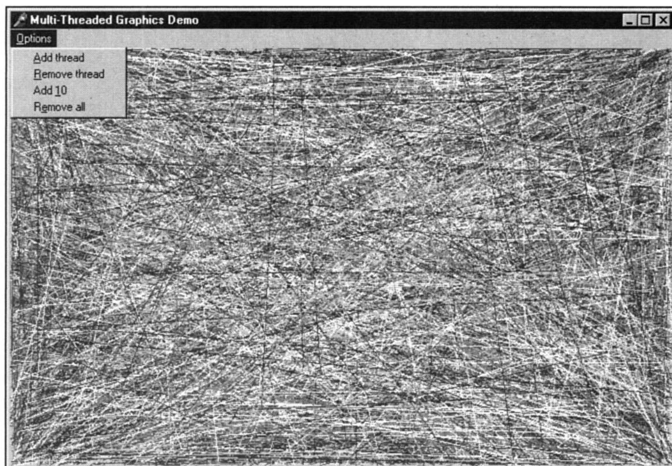

图11-10　MTGraph的主窗体

画布的锁定规则规定，画布的每一个使用者在每次画前要对画布加锁，每次操作完后要对画布解锁，这样多线程就可以不冲突地使用画布。要注意所有的 OnPaint() 和Paint()方法都通过VCL自动锁定和解锁画布来初始化；因此，一般的Delphi代码都能与后台的图形操作线程友好共处。

使用这个程序作为例子，我们来看一下如果执行锁定画布失败会是什么样的后果和情况。如果线程1设置了一个画布的笔的颜色为红色，而后再画一条线。线程 2设置了一个画布的笔的颜色为蓝色，而后再画一个圆。如果这两个线程在操作前都没有锁定画布，可能会出现下列冲突：当线程 1把画笔设为红色后，操作系统有可能切换到线程 2，而在线程2中又把画笔设为了蓝色并且画了一个圆。当重新切换到线程1时，线程1会画出一条线，但是它不是红色而是蓝色的，因为线程 2有机会在线程1的操作中间获得控制。

即使只有一个线程锁定了画布也会出现上面的情况。如果线程 1锁定了画布，而线程2没有，那么上述情况将会发生。两个线程必须都在其操作时锁定画布才可以防止线程冲突。

11.7　总结

现在，对线程的介绍完毕，你应当知道在 Delphi环境中该如何使用它们。在这一章里，讲述了几种线程同步的技术以及一个附属线程与一个程序中的主线程之间通信的技术。另外，本章通过一个文件搜索的例子程序，演示了多线程的使用。最后，介绍了多线程在数据和图形处理方面的应用。在下一章“文件处理”中，你将学习到对Delphi中的各种类型文件的多种处理技术。