Writing Multithreaded Applications

CHAPTER

11

IN THIS CHAPTER

- Threads Explained 476
- The TThread Object 478
- Managing Multiple Threads 493
- A Sample Multithreaded Application 510
- Multithreading Database Access 524
- Multithreaded Graphics 530
- Summary 535

The Win32 operating system provides you with the capability to have multiple threads of execution in your applications. Arguably the single most important benefit Win32 has over 16-bit Windows, this feature provides the means for performing different types of processing simultaneously in your application. This is one of the primary reasons for upgrading to a 32-bit version of Delphi, and this chapter gives you all the details on how to get the most out of threads in your applications.

Threads Explained

As discussed in Chapter 3, "The Win32 API," a *thread* is an operating system object that represents a path of code execution within a particular process. Every Win32 application has at least one thread—often called the *primary thread* or *default thread*—but applications are free to create other threads to perform other tasks.

Threads provide a means for running many distinct code routines simultaneously. Of course, unless you have more than one CPU in your computer, two threads can't truly run simultaneously. However, each thread is scheduled in fractions of seconds of time by the operating system in such a way as to give the feeling that many threads are running simultaneously.

TIP

Threads are not and never will be supported under 16-bit Windows. This means that any 32-bit Delphi code you write using threads will never be backward-compatible to Delphi 1.0. Keep this in mind if you develop applications for both platforms.

A New Type of Multitasking

The notion of *threads* is much different from the style of multitasking supported under 16-bit Windows platforms. You might hear people talk about Win32 as a *preemptive multitasking* operating system, whereas Windows 3.1 is a *cooperative multitasking* environment.

The key difference here is that under a preemptive multitasking environment the operating system is responsible for managing which thread executes when. When execution of thread one is stopped in order for thread two to receive some CPU cycles, thread one is said to have been *preempted*. If the code that one thread is executing happens to put itself into an infinite loop, it's usually not a tragic situation because the operating system will continue to schedule time for all the other threads.

Under Windows 3.1, the application developer is responsible for giving control back to Windows at points during the application's execution. Failure of an application to do so causes the operating environment to appear locked up, and we all know what a painful experience that

can be. If you take a moment to think about it, it's slightly amusing that the very foundation of 16-bit Windows depends on all applications behaving themselves and not putting themselves into infinite loops, a recursion, or any other unneighborly situation. It's because all applications must cooperate for Windows to work correctly that this type of multitasking is referred to as *cooperative*.

Using Multiple Threads in Delphi Applications

It's no secret that threads represent a serious boon for Windows programmers. You can create secondary threads in your applications anywhere that it's appropriate to do some sort of background processing. Calculating cells in a spreadsheet or spooling a word processing document to the printer are examples of situations where a thread would commonly be used. The goal of the developer will most often be to perform necessary background processing while still providing the best possible response time for the user interface.

Most of VCL has a built-in assumption that it's being accessed by only one thread at any given time. While this limitation is especially apparent in the user interface portions of VCL, it's important to note that even many non-UI portions of VCL are not thread-safe.

Non-UI VCL

There are actually very few areas of VCL that are guaranteed to be thread-safe. Perhaps the most notable among these thread-safe areas is VCL's property streaming mechanism, which ensures that component streams can be effectively read and written by multiple threads. Remember that even very basic classes in VCL, such as TList, are not designed to be manipulated from multiple simultaneous threads. In some cases, VCL provides thread-safe alternatives that you can use in cases where you need them. For example, use a TThreadList in place of a TList when the list will be subject to manipulation by multiple threads.

UI VCL

VCL requires that all user-interface control happens within the context of an application's primary thread (the exception is the thread-safe `TCanvas`, which is explained later in this chapter). Of course, techniques are available to update the user interface from a secondary thread (which we discuss later), but this limitation essentially forces you to use threads a bit more judiciously than you might do otherwise. The examples given in this chapter show some ideal uses for multiple threads in Delphi applications.

Misuse of Threads

Too much of a good thing can be bad, and that's definitely true in the case of threads. Even though threads can help to solve some of the problems you may have from an application design standpoint, they do introduce a whole new set of problems. For example, suppose you're writing an integrated development environment, and you want the compiler to execute

in its own thread so the programmer will be free to continue work on the application while the program compiles. The problem here is this: What if the programmer changes a file that the compiler is in the middle of compiling? There are a number of solutions to this problem, such as making a temporary copy of the file while the compile continues or preventing the user from editing not-yet-compiled files. The point is simply that threads are not a panacea; although they solve some development problems, they invariably introduce others. What's more, bugs due to threading problems are also much, much harder to debug because threading problems are often time-sensitive. Designing and implementing thread-safe code is also more difficult because you have a lot more factors to consider.

The TThread Object

Delphi encapsulates the API thread object into an Object Pascal object called `TThread`. Although `TThread` encapsulates almost all the commonly used thread API functions into one discrete object, there are some points—particularly those dealing with thread synchronization—where you have to use the API. In this section, you learn how the `TThread` object works and how to use it in your applications.

TThread Basics

The `TThread` object is found in the `Classes` unit and is defined as follows:

```
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;
    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;
```

As you can tell from the declaration, `TThread` is a direct descendant of `TObject` and is therefore not a component. You might also notice that the `TThread.Execute()` method is `abstract`. This means that the `TThread` class itself is abstract, meaning that you will never create an instance of `TThread` itself. You will only create instances of `TThread` descendants. Speaking of which, the most straightforward way to create a `TThread` descendant is to select Thread Object from the New Items dialog box provided by the File, New menu option. The New Items dialog box is shown in Figure 11.1.

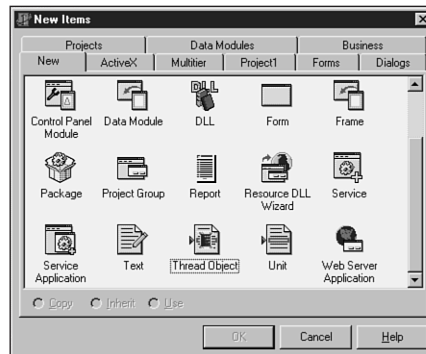

FIGURE 11.1
The Thread Object item in the New Items dialog box.

After choosing Thread Object from the New Items dialog box, you'll be presented with a dialog box that prompts you to enter a name for the new object. You could enter `TTestThread`, for example. Delphi will then create a new unit that contains your object. Your object will initially be defined as follows:

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

As you can see, the only method that you *must* override in order to create a functional descendant of `TThread` is the `Execute()` method. Suppose, for example, that you want to perform a complex calculation within `TTestThread`. In that case, you could define its `Execute()` method as follows:

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

Admittedly, the equation is contrived, but it still illustrates the point in this case because the sole purpose of this equation is to take a relatively long time to execute.

You can now execute this sample thread by calling its `Create()` constructor. For now, you can do this from a button click in the main form, as shown in the following code (remember to include the unit containing `TTestThread` in the `uses` clause of the unit containing `TForm1` to avoid a compiler error):

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

If you run the application and click the button, you'll notice that you can still manipulate the form by moving it or resizing it while the calculation goes on in the background.

NOTE

The single `Boolean` parameter passed to `TThread`'s `Create()` constructor is called `CreateSuspended`, and it indicates whether to start the thread in a suspended state. If this parameter is `False`, the object's `Execute()` method will automatically be called following `Create()`. If this parameter is `True`, you must call `TThread`'s `Resume()`

method at some point to actually start the thread running. This will cause the `Execute()` method to be invoked at that time. You would set `CreateSuspended` to `True` if you need to set additional properties on your thread object before allowing it to run. Setting the properties after the thread is running would be asking for trouble.

To go a little deeper, the `constructor` of `Create()` calls the `BeginThread()` Delphi Runtime Library (RTL) function, which calls the `CreateThread()` API function in order to create the new thread. The value of the `CreateSuspended` parameter indicates whether to pass the `CREATE_SUSPENDED` flag to `CreateThread()`.

Thread Instances

Going back to the `Execute()` method for the `TTestThread` object, notice that it contains a local variable called `i`. Consider what might happen to `i` if you create two instances of `TTestThread`. Does the value for one thread overwrite the value for the other? Does the first thread take precedence? Does it blow up? The answers are no, no, and no. Win32 maintains a separate stack for each thread executing in the system. This means that as you create multiple instances of the `TTestThread` object, each one keeps its own copy of `i` on its own stack. Therefore, all the threads will operate independently of one another in that respect.

An important distinction to make, however, is that this notion of the same variable operating independently in each thread doesn't carry over to global variables. This topic is explored in detail in the "Thread-Local Storage" and "Thread Synchronization" sections, later in this chapter.

Thread Termination

A `TThread` is considered terminated when the `Execute()` method has finished executing. At that point, the `EndThread()` Delphi standard procedure is called, which in turn calls the `ExitThread()` API procedure. `ExitThread()` properly disposes of the thread's stack and deallocates the API thread object. This cleans up the thread as far as the API is concerned.

You also need to ensure that the Object Pascal object is destroyed when you're finished using a `TThread` object. This will ensure that all memory occupied by that object has been properly disposed of. Although this will automatically happen when your process terminates, you might want to dispose of the object earlier so that your application doesn't leak memory as it runs. The easiest way to ensure that the `TThread` object is disposed of is to set its `FreeOnTerminate` property to `True`. This can be done any time before the `Execute()` method finishes executing. For example, you could do this for the `TTestThread` object by setting the property in the `Execute()` method as follows:

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

The `TThread` object also has an `OnTerminate` event that's called when the thread terminates. It's also acceptable to free the `TThread` object from within a handler for this event.

TIP

The `OnTerminate` event of `TThread` is called from the context of your application's main thread. This means that you can feel free to access VCL properties and methods from within a handler for this event without using the `Synchronize()` method, as described in the following section.

It's also important to note that your thread's `Execute()` method is responsible for checking the status of the `Terminated` property to determine the need to make an earlier exit. Although this means one more thing that you must worry about when working with threads, the flip side is that this type of architecture ensures that the rug isn't pulled out from under you, and that you'll be able to perform any necessary cleanup on thread termination. To add this code to the `Execute()` method of `TTestThread` is rather simple, and the addition is shown here:

```
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;
```

CAUTION

In case of emergency, you can also use the Win32 API `TerminateThread()` function to terminate an executing thread. You should do this only when no other options exist, such as when a thread gets caught in an endless loop and stops responding. This function is defined as follows:

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

The `Handle` property of `TThread` provides the API thread handle, so you could call this function with syntax similar to that shown here:

```
TerminateThread(MyHosedThread.Handle, 0);
```

If you choose to use this function, you should be wary of the negative side effects it will cause. First, this function behaves differently under Windows NT/2000 and Windows 95/98. Under Windows 95/98, `TerminateThread()` disposes of the stack associated with the thread; under Windows NT/2000, the stack sticks around until the process is terminated. Second, on all Win32 operating systems, `TerminateThread()` simply halts the execution, wherever it may be, and does not allow `try..finally` blocks to clean up resources. This means that files opened by the thread would not be closed, memory allocated by the thread would not be freed, and so forth. Also, DLLs loaded by your process won't be notified when a thread destroyed with `TerminateThread()` goes away, and this may cause problems when the DLL closes. See Chapter 9, "Dynamic Link Libraries," for more information on thread notifications in DLLs.

Synchronizing with VCL

As mentioned several times earlier in this chapter, you should only access VCL properties or methods from the application's primary thread. This means that any code that accesses or updates your application's user interface should be executed from the context of the primary thread. The disadvantages of this architecture are obvious, and this requirement might seem rather limiting on the surface, but it actually has some redeeming advantages that you should know about.

Advantages of a Single-Threaded User Interface

First, it greatly reduces the complexity of your application to have only one thread accessing the user interface. Win32 requires that each thread that creates a window have its own message loop using the `GetMessage()` function. As you might imagine, having messages coming into your application from a variety of sources can make it extremely difficult to debug. Because an application's message queue provides a means for serializing input—fully processing one condition before moving on to the next—you can depend in most cases on certain messages coming before or after others. Adding another message loop throws this serialization of input out the door, thereby opening you up to potential synchronization problems and possibly introducing a need for complex synchronization code.

Additionally, because VCL can depend on the fact that it will be accessed by only one thread at any given time, the need for code to synchronize multiple threads inside VCL is obviated.

The net result of this is better overall performance of your application due to a more streamlined architecture.

The Synchronize() Method

`TThread` provides a method called `Synchronize()` that allows for some of its own methods to be executed from the application's primary thread. `Synchronize()` is defined as follows:

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

Its `Method` parameter is of type `TThreadMethod` (which means a procedural method that takes no parameter), which is defined as follows:

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

The method you pass as the `Method` parameter is the one that's then executed from the application's primary thread. Going back to the `TTestThread` example, suppose you want to display the result in an edit control on the main form. You could do this by introducing to `TTestThread` a method that makes the necessary change to the edit control's `Text` property and calling that method by using `Synchronize()`.

In this case, suppose this method is called `GiveAnswer()`. Listing 11.1 shows the complete source code for this unit, called `ThrdU`, which includes the code to update the edit control on the main form.

LISTING 11.1 The *ThrdU.PAS* Unit

```
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.
```

You already know that the `Synchronize()` method enables you to execute methods from the context of the primary thread, but up to this point you've treated `Synchronize()` as sort of a mysterious black box. You don't know *how* it works—you only know that it does. If you'd like to take a peek at the man behind the curtain, read on.

The first time you create a secondary thread in your application, VCL creates and maintains a hidden *thread window* from the context of its primary thread. The sole purpose of this window is to serialize procedure calls made through the `Synchronize()` method.

The `Synchronize()` method stores the method specified in its `Method` parameter in a private field called `FMethod` and sends a VCL-defined `CM_EXECPROC` message to the thread window, passing `Self` (`Self` being the `TThread` object in this case) as the `lParam` of the message. When the thread window's window procedure receives this `CM_EXECPROC` message, it calls the method specified in `FMethod` through the `TThread` object instance passed in the `lParam`. Remember, because the thread window was created from the context of the primary thread, the window procedure for the thread window is also executed by the primary thread. Therefore, the method specified in the `FMethod` field is also executed by the primary thread.

To see a more visual illustration of what goes on inside `Synchronize()`, look at Figure 11.2.

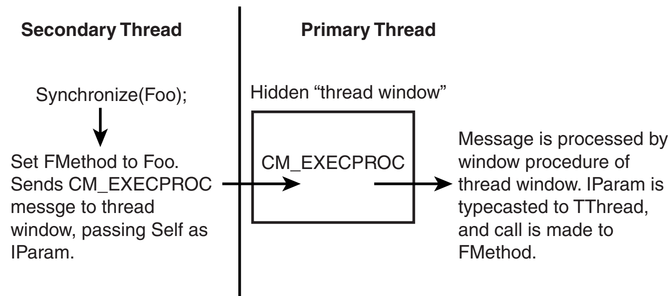

FIGURE 11.2
A road map of the `Synchronize()` method.

Using Messages for Synchronization

As an alternative to the `TThread.Synchronize()` method, another technique for thread synchronization is to use messages to communicate between threads. You can use the `SendMessage()` or `PostMessage()` API function to send or post messages to windows operating in the context of another thread. For example, the following code could be used to set the text in an edit control residing in another thread:

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

A Demo Application

To fully illustrate how multithreading in Delphi works, you can save the current project as `EZThrd`. Then you can also put a memo control on the main form so that it resembles what's shown in Figure 11.3.

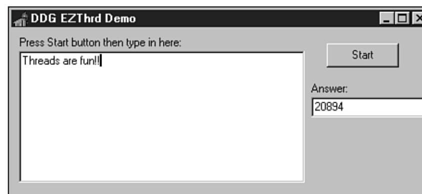

FIGURE 11.3
The main form of the `EZThrd` demo.

The source code for the main unit is shown in Listing 11.2.

LISTING 11.2 The *MAIN.PAS* Unit for the *EZThrd* Demo

```
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.
```

Notice that after you click the button to invoke the secondary thread, you can still type in the memo control as if the secondary thread doesn't exist. When the calculation is completed, the result will be displayed in the edit control.

Priorities and Scheduling

As mentioned earlier, the operating system is in charge of scheduling each thread some CPU cycles in which it may execute. The amount of time scheduled for a particular thread depends on the priority assigned to the thread. An individual thread's overall priority is determined by a combination of the priority of the process that created the thread—called the *priority class*—and the priority of the thread itself—called the *relative priority*.

Process Priority Class

The *process priority class* describes the priority of a particular process running on the system. Win32 supports four distinct priority classes: Idle, Normal, High, and Realtime. The default priority class for any process, of course, is Normal. Each of these priority classes has a corresponding flag defined in the `Windows` unit. You can `or` any of these flags with the `dwCreationFlags` parameter of `CreateProcess()` in order to spawn a process with a specific priority. Additionally, you can use these flags to dynamically adjust the priority class of a given process, as shown in a moment. Furthermore, each priority class can also be represented by a numeric priority level, which is a value between `4` and `24` (inclusive).

NOTE

Modifying a process's priority class requires special process privileges under Windows NT/2000. The default settings allow processes to set their priority classes, but these can be turned off by system administrators, particularly on high-load Windows NT/2000 servers.

Table 11.1 shows each priority class and its corresponding flag and numeric value.

TABLE 11.1 Process Priority Classes

Class	*Flag*	*Value*
Idle	`IDLE_PRIORITY_CLASS`	`$40`
Below normal*	`BELOW_NORMAL_PRIORITY_CLASS`	`$4000`
Normal	`NORMAL_PRIORITY_CLASS`	`$20`Above normal*
	`ABOVE_NORMAL_PRIORITY_CLASS`	`$8000`
High	`HIGH_PRIORITY_CLASS`	`$80`
Realtime	`REALTIME_PRIORITY_CLASS`	`$100`

**Available only on Windows 2000, and flag constant is not present in the Delphi 5 version of Windows.pas.*

To get and set the priority class of a given process dynamically, Win32 provides the `GetPriorityClass()` and `SetPriorityClass()` functions, respectively. These functions are defined as follows:

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

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

The `hProcess` parameter in both cases represents a handle to a process. In most cases, you'll be calling these functions in order to access the priority class of your own process. In that case, you can use the `GetCurrentProcess()` API function. This function is defined as follows:

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

The return value of these functions is a pseudo-handle for the current process. We say *pseudo* because the function doesn't create a new handle, and the return value doesn't have to be closed with `CloseHandle()`. It merely provides a handle that can be used to reference an existing handle.

To set the priority class of your application to High, use code similar to the following:

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

CAUTION

In almost all cases, you should avoid setting the priority class of any process to Realtime. Because most of the operating system threads run in a priority class lower than Realtime, your thread will receive more CPU time than the OS itself, and that could cause some unexpected problems.

Even bumping the priority class of the process to High can cause problems if the threads of the process don't spend most of their time idle or waiting for external events (such as file I/O). One high-priority thread is likely to drain all CPU time away from lower-priority threads and processes until it blocks on an event, goes idle, or processes messages. Preemptive multitasking can easily be defeated by abusing scheduler priorities.

Relative Priority

The other thing that goes into determining the overall priority of a thread is the *relative priority* of a particular thread. The important distinction to make is that the priority class is associated with a process and the relative priority is associated with individual threads within a process. A thread can have any one of seven possible relative priorities: Idle, Lowest, Below Normal, Normal, Above Normal, Highest, or Time Critical.

TThread exposes a Priority property of an enumerated type TThreadPriority. There's an enumeration in this type for each relative priority:

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

You can get and set the priority of any TThread object simply by reading from or writing to its Priority property. The following code sets the priority of a TThread descendant instance called MyThread to Highest:

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

Like priority classes, each relative priority is associated with a numeric value. The difference is that the relative priority is a signed value that, when added to a process's class priority, is used to determine the overall priority of a thread within the system. For this reason, relative priority is sometimes called *delta priority*. The overall priority of a thread can be any value from 1 to 31 (1 being the lowest). Constants are defined in the Windows unit that represents the signed value for each priority. Table 11.2 shows how each enumeration in TThreadPriority maps to an API constant.

TABLE 11.2 Relative Priorities for Threads

TThreadPriority	*Constant*	*Value*
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*

The reason the values for the tpIdle and tpTimeCritical priorities are marked with asterisks is that, unlike the others, these relative priority values are not truly added to the class priority to determine overall thread priority. Any thread that has the tpIdle relative priority, regardless of its priority class, has an overall priority of 1. The exception to this rule is the Realtime priority class, which, when combined with the tpIdle relative priority, has an overall value of 16. Any thread that has a priority of tpTimeCritical, regardless of its priority class, has an overall priority of 15. The exception to this rule is the Realtime priority class, which, when combined with the tpTimeCritical relative priority, has an overall value of 31.

Suspending and Resuming Threads

Recall when you learned about `TThread`'s `Create()` constructor earlier in this chapter. At the time, you discovered that a thread could be created in a suspended state, and that you must call its `Resume()` method in order for the thread to begin execution. As you might guess, a thread can also be suspended and resumed dynamically. You accomplish this using the `Suspend()` method in conjunction with the `Resume()` method.

Timing a Thread

Back in the 16-bit days when we programmed under Windows 3.*x*, it was pretty common to wrap some portion of code with calls to `GetTickCount()` or `timeGetTime()` to determine how much time a particular calculation may take (something like the following, for example):

```
var
  StartTime, Total: Longint;
begin
  StartTime := GetTickCount;
  { Do some calculation here }
  Total := GetTickCount - StartTime;
```

In a multithreaded environment, this is much more difficult to do, because your application may be preempted by the operating system in the middle of the calculation in order to provide CPU cycles to other processes. Therefore, any timing you do that relies on the system time can't provide a true measure of how long it spends crunching the calculation in your thread.

To avoid such problems, Win32 under Windows NT/2000 provides a function called `GetThreadTimes()`, which provides quite detailed information on thread timing. This function is declared as follows:

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

The `hThread` parameter is the handle to the thread for which you want to obtain timing information. The other parameters for this function are passed by reference and are filled in by the function. Here's an explanation of each:

- `lpCreationTime`. The time when the thread was created.
- `lpExitTime`. The time when the thread was exited. If the thread is still running, this value is undefined.
- `lpKernelTime`. The amount of time the thread has spent executing operating system code.
- `lpUserTime`. The amount of time the thread has spent executing application code.

Each of the last four parameters is of type `TFileTime`, which is defined in the `Windows` unit as follows:

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

The definition of this type is a bit unusual, but it's a part of the Win32 API, so here goes: `dwLowDateTime` and `dwHighDateTime` are combined into a quad word (64-bit) value that represents the number of 100-nanosecond intervals that have passed since January 1, 1601. This means, of course, that if you wanted to write a simulation of English fleet movements as they defeated the Spanish Armada in 1588, the `TFileTime` type would be a wholly inappropriate way to keep track of time…but we digress.

TIP

Because the `TFileTime` type is 64 bits in size, you can typecast a `TFileTime` to an `Int64` type in order to perform arithmetic on `TFileTime` values. The following code demonstrates how to quickly tell whether one `TFileTime` is greater than another:

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

In order to help you work with `TFileTime` values in a manner more native to Delphi, the following functions allow you to convert back and forth between `TFileTime` and `TDateTime` types:

```
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;
```

CAUTION

Remember that the `GetThreadTimes()` function is implemented only under Windows NT/2000. The function always returns `False` when called under Windows 95 or 98. Unfortunately, Windows 95/98 doesn't provide any mechanism for retrieving thread-timing information.

Managing Multiple Threads

As indicated earlier, although threads can solve a variety of programming problems, they're also likely to introduce new types of problems that you must deal with in your applications. Most commonly, these problems revolve around multiple threads accessing global resources, such as global variables or handles. Additionally, problems can arise when you need to ensure that some event in one thread always occurs before or after some other event in another thread. In this section, you learn how to tackle these problems by using the facilities provided by Delphi for thread-local storage and those provided by the API for thread synchronization.

Thread-Local Storage

Because each thread represents a separate and distinct path of execution within a process, it logically follows that you will at some point want to have a means for storing data associated with each thread. There are three techniques for storing data unique to each thread: the first and most straightforward involves local (stack-based) variables. Because each thread gets its own stack, each thread executing within a single procedure or function will have its own copy of local variables. The second technique is to store local information in your `TThread` descendant object. Finally, you can also use Object Pascal's `threadvar` reserved word to take advantage of operating system–level thread-local storage.

TThread Storage

Storing pertinent data in the `TThread` descendant object should be your technique of choice for thread-local storage. It's both more straightforward and more efficient than using `threadvar` (described later). To declare thread-local data in this manner, simply add it to the definition of your `TThread` descendant, as shown here:

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

TIP

It's about 10 times faster to access a field of an object than to access a `threadvar` variable, so you should store your thread-specific data in your `TThread` descendant, if possible. Data that doesn't need to exist for more than the lifetime of a particular procedure or function should be stored in local variables, because those are faster still than the fields of a `TThread` object.

threadvar: API Thread-Local Storage

Earlier we mentioned that each thread is provided with its own stack for storing local variables, whereas global data has to be shared by all threads within an application. For example, say you have a procedure that sets or displays the value of a global variable. When you call the procedure passing a text string, the global variable is set, and when you call the procedure passing an empty string, the global variable is displayed. Such a procedure might look like this:

```
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;
```

If this procedure is called from within the context of one thread only, there wouldn't be any problems. You'd call the procedure once to set the value of `GlobalStr` and call it again to display the value. However, consider what can happen if two or more threads call this procedure

at any given time. In such a case, it's possible that one thread could call the procedure to set the string and then get preempted by another thread that might also call the function to set the string. By the time the operating system gives CPU time back to the first thread, the value of `GlobalStr` for that thread will be hopelessly lost.

For situations such as these, Win32 provides a facility known as *thread-local storage* that enables you to create separate copies of global variables for each running thread. Delphi nicely encapsulates this functionality with the `threadvar` clause. Just declare any global variables you want to exist separately for each thread within a `threadvar` (as opposed to `var`) clause, and the work is done. A redeclaration of the `GlobalStr` variable is as simple as this:

```
threadvar
  GlobalStr: String;
```

The unit shown in Listing 11.3 illustrates this very problem. It represents the main unit to a Delphi application that contains only a button on a form. When the button is clicked, the procedure is called to set and then to show `GlobalStr`. Next, another thread is created, and the value internal to the thread is set and shown again. After the thread creation, the primary thread again calls `SetShowStr` to display `GlobalStr`.

Try running this application with `GlobalStr` declared as a `var` and then as a `threadvar`. You'll see a difference in the output.

LISTING 11.3 The *MAIN.PAS* Unit for Thread-Local Storage Demo

```
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;
```

continues

LISTING 11.3 Continued

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

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

```
  Sleep(100);
  SetShowStr('');
end;

end.
```

NOTE

The demo program calls the Win32 API `Sleep()` procedure after creating the thread. `Sleep()` is declared as follows:

```
procedure Sleep(dwMilliseconds: DWORD); stdcall;
```

The `Sleep()` procedure tells the operating system that the current thread doesn't need any more CPU cycles for another `dwMilliseconds` milliseconds. Inserting this call into the code has the effect of simulating system conditions where more multitasking is occurring and introducing a bit more "randomness" into the application as to which threads will be executing when.

It's often acceptable to pass zero in the `dwMilliseconds` parameter. Although that doesn't prevent the current thread from executing for any specific amount of time, it does cause the operating system to give CPU cycles to any waiting threads of equal or greater priority.

Be careful of using `Sleep()` to work around mysterious timing problems. `Sleep()` may work around a particular problem on your machine, but timing problems that are not solved conclusively will pop up again on somebody else's machine, especially when the machine is significantly faster or slower or has a different number of processors than your machine.

Thread Synchronization

When working with multiple threads, you'll often need to synchronize the access of threads to some particular piece of data or resource. For example, suppose you have an application that uses one thread to read a file into memory and another thread to count the number of characters in the file. It goes without saying that you can't count all the characters in the file until the entire file has been loaded into memory. However, because each operation occurs in its own thread, the operating system would like to treat them as two completely unrelated tasks. To fix this problem, you must synchronize the two threads so that the counting thread doesn't execute until the loading thread finishes.

These are the types of problems that thread synchronization addresses, and Win32 provides a variety of ways to synchronize threads. In this section, you'll see examples of thread synchronization techniques using critical sections, mutexes, semaphores, and events.

In order to examine these techniques, first take a look at a problem involving threads that need to be synchronized. For the purpose of illustration, suppose you have an array of integers that needs to be initialized with ascending values. You want to first go through the array and set the values from `1` to `128` and then reinitialize the array with values from `128` to `255`. You'll then display the final thread in a list box. An approach to this might be to perform the initializations in two separate threads. Consider the code in Listing 11.4 for a unit that attempts to perform this task.

LISTING 11.4 A Unit That Attempts to Initialize an Array in Threads

```
unit 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;

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

end.
```

Because both threads will execute simultaneously, what happens is that the contents of the array are corrupted as it's initialized. As proof, take a look at the output of this code, as shown in Figure 11.4.

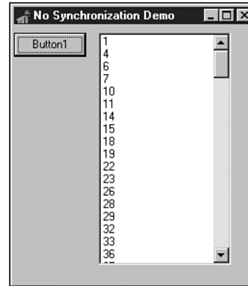

FIGURE 11.4
Output from unsynchronized array initialization.

The solution to this problem is to synchronize the two threads as they access the global array so that they don't both dive in at the same time. You can take any of a number of valid approaches to this problem.

Critical Sections

Critical sections provide one of the most straightforward ways to synchronize threads. A *critical section* is some section of code that allows only one thread to execute through it at a time. If you wrap the code used to initialize the array in a critical section, other threads will be blocked from entering the code section until the first finishes.

Prior to using a critical section, you must initialize it using the `InitializeCriticalSection()` API procedure, which is declared as follows:

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

`lpCriticalSection` is a `TRTLCriticalSection` record that's passed by reference. The exact definition of `TRTLCriticalSection` is unimportant, because you'll rarely (if ever) actually look at the contents of one. You'll pass an uninitialized record in the `lpCriticalSection` parameter, and the record will be filled by the procedure.

NOTE

Microsoft deliberately obscures the structure of the `TRTLCriticalSection` record because the contents vary from one hardware platform to another, and because tinkering with the contents of this structure can potentially wreak havoc on your process. On Intel-based systems, the critical section structure contains a counter, a field containing the current thread handle, and (potentially) a handle of a system event. On Alpha hardware, the counter is replaced with an Alpha-CPU data structure called a *spinlock*, which is more efficient than the Intel solution.

When the record is filled, you can create a critical section in your application by wrapping some block of code with calls to `EnterCriticalSection()` and `LeaveCriticalSection()`. These procedures are declared as follows:

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

As you might guess, the `lpCriticalSection` parameter you pass these guys is the same one that's filled in by the `InitializeCriticalSection()` procedure.

When you're finished with the `TRTLCriticalSection` record, you should clean up by calling the `DeleteCriticalSection()` procedure, which is declared as follows:

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

Listing 11.5 demonstrates the technique for synchronizing the array-initialization threads with critical sections.

LISTING 11.5 Using Critical Sections

```
unit 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;
```

continues

LISTING 11.5 Continued

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

```
end;

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

end.
```

After the first thread passes through the call to `EnterCriticalSection()`, all other threads are prevented from entering that block of code. The next thread that comes along to that line of code is put to sleep until the first thread calls `LeaveCriticalSection()`. At that point, the second thread is awakened and allowed to take control of the critical section. Figure 11.5 shows the output of this application when the threads are synchronized.

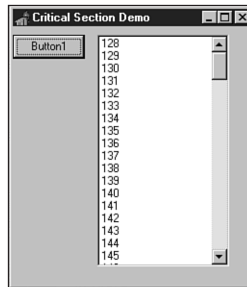

FIGURE 11.5
Output from synchronized array initialization.

Mutexes

Mutexes work very much like critical sections except for two key differences: First, mutexes can be used to synchronize threads across process boundaries. Second, mutexes can be given a string name, and additional handles to existing mutex objects can be created by referencing that name.

TIP

Semantics aside, the biggest difference between critical sections and event objects such as mutexes is performance: Critical sections are very lightweight—as few as

continues

10–15 clock cycles to enter or leave the critical section when there are no thread collisions. As soon as there is a thread collision for that critical section, the system creates an event object (a mutex, probably). The cost of using event objects such as mutexes is that it requires a roundtrip into the kernel, which requires a process context switch and a change of ring levels, which piles up to 400 to 600 clock cycles each way. All this overhead is incurred even if your app doesn't currently have multiple threads, or if no other threads are contending for the resource you're protecting.

The function used to create a mutex is appropriately called `CreateMutex()`. This function is declared as follows:

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

`lpMutexAttributes` is a pointer to a `TSecurityAttributes` record. It's common to pass `nil` in this parameter, in which case the default security attributes will be used.

`bInitialOwner` indicates whether the thread creating the mutex should be considered the owner of the mutex when it's created. If this parameter is `False`, the mutex is unowned.

`lpName` is the name of the mutex. This parameter can be `nil` if you don't want to name the mutex. If this parameter is non-`nil`, the function will search the system for an existing mutex with the same name. If an existing mutex is found, a handle to the existing mutex is returned. Otherwise, a handle to a new mutex is returned.

When you're finished using a mutex, you should close it using the `CloseHandle()` API function.

Listing 11.6 again demonstrates the technique for synchronizing the array-initialization threads, except this time it uses mutexes.

LISTING 11.6 Using Mutexes for Synchronization

```
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;
  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;
```

continues

LISTING 11.6 Continued

```
  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);
  end;
end;

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

end.
```

You'll notice that in this case the `WaitForSingleObject()` function is used to control thread entry into the synchronized block of code. This function is declared as follows:

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

The purpose of this function is to sleep the current thread up to `dwMilliseconds` milliseconds until the API object specified in the `hHandle` parameter becomes signaled. *Signaled* means different things for different objects. A mutex becomes signaled when it's not owned by a thread, whereas a process, for example, becomes signaled when it terminates. Apart from an actual period of time, the `dwMilliseconds` parameter can also have the value `0`, which means to check the status of the object and return immediately, or `INFINITE`, which means to wait forever for the object to become signaled. The return value of this function can be any one of the values shown in Table 11.3.

TABLE 11.3 WAIT constants used by WaitForSingleObject() API function.

Value	*Meaning*
`WAIT_ABANDONED`	The specified object is a mutex object, and the thread owning the mutex was exited before it freed the mutex. This circumstance is referred to as an *abandoned mutex*; in such a case, ownership of the mutex object is granted to the calling thread, and the mutex is set to nonsignaled.
`WAIT_OBJECT_0`	The state of the specified object is signaled.
`WAIT_TIMEOUT`	The timeout interval elapsed, and the object's state is nonsignaled.

Again, when a mutex isn't owned by a thread, it's in the signaled state. The first thread to call `WaitForSingleObject()` on this mutex is given ownership of the mutex, and the state of the mutex object is set to nonsignaled. The thread's ownership of the mutex is severed when the thread calls the `ReleaseMutex()` function, passing the mutex handle as the parameter. At that point, the state of the mutex again becomes signaled.

NOTE

In addition to `WaitForSingleObject()`, the Win32 API also has functions called `WaitForMultipleObjects()` and `MsgWaitForMultipleObjects()`, which enable you to wait for the state of one or more objects to become signaled. These functions are documented in the Win32 API online help.

Semaphores

Another technique for thread synchronization involves using semaphore API objects. *Semaphores* build on the functionality of mutexes while adding one important feature: They offer the capability of resource counting so that a predetermined number of threads can enter synchronized pieces of code at one time. The function used to create a semaphore is `CreateSemaphore()`, and it's declared as follows:

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

Like `CreateMutex()`, the first parameter to `CreateSemaphore()` is a pointer to a `TSecurityAttributes` record to which you can pass `Nil` for the defaults.

`lInitialCount` is the initial count of the semaphore object. This is a number between `0` and `lMaximumCount`. A semaphore is signaled as long as this parameter is greater than zero. The count of a semaphore is decremented whenever `WaitForSingleObject()` (or one of the other wait functions) releases a thread. A semaphore's count is increased by using the `ReleaseSemaphore()` function.

`lMaximumCount` specifies the maximum count value of the semaphore object. If the semaphore is used to count some resources, this number should represent the total number of resources available.

`lpName` is the name of the semaphore. This parameter behaves the same as the parameter of the same name in `CreateMutex()`.

Listing 11.7 demonstrates using semaphores to perform synchronization of the array-initialization problem.

LISTING 11.7 Using Semaphores for Synchronization

```
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;
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
```

continues

LISTING 11.7 Continued

```
  TFooThread.Create(False);
end;

end.
```

Because you allow only one thread to enter the synchronized portion of code, the maximum count for the semaphore is `1` in this case.

The `ReleaseSemaphore()` function is used to increase the count for the semaphore. Notice that this function is a bit more involved than its cousin, `ReleaseMutex()`. The declaration for `ReleaseSemaphore()` is as follows:

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

The `lReleaseCount` parameter enables you to specify the number by which the count of the semaphore will be increased. The old count will be stored in the `longint` pointed to by the `lpPreviousCount` parameter if its value is not `Nil`. A subtle implication of this capability is that a semaphore is never really owned by any thread in particular. For example, suppose the maximum count of a semaphore is `10`, and 10 threads call `WaitForSingleObject()` to set the count of the thread to `0` and put the thread in a nonsignaled state. All it takes is one of those threads to call `ReleaseSemaphore()` with `10` as the `lReleaseCount` parameter in order to not only make the thread signaled again, but to increase the count back to `10`. This powerful capability can introduce some hard-to-track-down bugs into your applications, so you should use it with care.

Be sure to use the `CloseHandle()` function to free the semaphore handle allocated with `CreateSemaphore()`.

A Sample Multithreaded Application

To demonstrate the usage of `TThread` objects within the context of a real-world application, this section focuses on creating a file-search application that performs its searches in a specialized thread. The project is called `DelSrch`, which stands for *Delphi Search*, and the main form for this utility is shown in Figure 11.6.

The application works like this. The user chooses a path through which to search and provides a file specification to indicate the types of files to be searched. The user also enters a token to search for in the appropriate edit control. Some option check boxes on one side of the form enable the user to tailor the application to suit his or her needs for a particular search. When the user clicks the Search button, a search thread is created and the appropriate search information—such as token, path, and file specification—is passed to the `TThread` descendant object.

When the search thread finds the search token in certain files, information is appended to the list box. Finally, if the user double-clicks a file in the list box, the user can browse it with a text editor or view it from its desktop association.

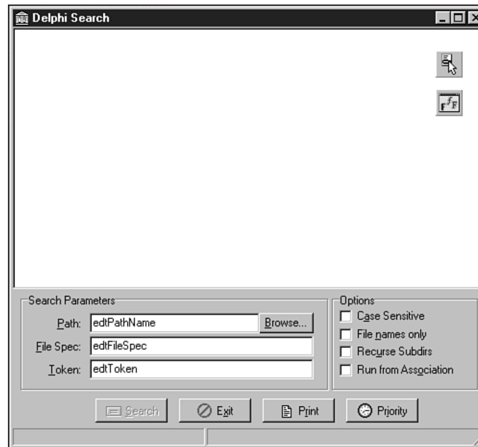

FIGURE 11.6
The Main form for the `DelSrch` project.

Although this is a fairly full-featured application, we'll focus mainly on explaining the application's key search features and how they relate to multithreading.

The User Interface

The main unit for the application is called `Main.pas`. Shown in Listing 11.8, this unit is responsible for managing the main form and the overall user interface. In particular, this unit contains the logic for owner-drawing the list box, invoking a viewer for files in the list box, invoking the search thread, printing the list box contents, and reading and writing UI settings to an INI file.

LISTING 11.8 The *`Main.pas`* Unit for the *`DelSrch`* Project

```
unit SrchU;

interface

uses Classes, StdCtrls;

type
  TSearchThread = class(TThread)
  private
```

continues

LISTING 11.8 Continued

```
    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

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;
  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" }
```

continues

LISTING 11.8 Continued

```
      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);
    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.
```

Several things worth mentioning happen in this unit. First, you'll notice the fairly small `PrintStrings()` procedure that's used to send the contents of `TStrings` to the printer. To accomplish this, the procedure takes advantage of Delphi's `AssignPrn()` standard procedure, which assigns a `TextFile` variable to the printer. That way, any text written to the `TextFile` is automatically written to the printer. When you're finished writing to the printer, be sure to use the `CloseFile()` procedure to close the connection to the printer.

Also of interest is the use of the `ShellExecute()` Win32 API procedure to launch a viewer for a file that will be shown in the list box. `ShellExecute()` not only enables you to invoke executable programs but also to invoke associations for registered file extensions. For example, if you try to invoke a file with a `.pas` extension using `ShellExecute()`, it will automatically load Delphi to view the file.

TIP

If `ShellExecute()` returns a value indicating an error, the application calls `RaiseLastWin32Error()`. This procedure, located in the `SysUtils` unit, calls the `GetLastError()` API function and Delphi's `SysErrorMessage()` in order to obtain more detailed information about the error and to format that information into a string. You can use `RaiseLastWin32Error()` in this manner in your own applications if you want your users to obtain detailed error messages on API failures.

The Search Thread

The searching engine is contained within a unit called `SrchU.pas`, which is shown in Listing 11.9. This unit does a number of interesting things, including copying an entire file into a string, recursing subdirectories, and communicating information back to the main form.

LISTING 11.9 The *SrchU.pas* Unit

```
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

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;
```

continues

LISTING 11.9 Continued

```
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;
  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]);
```

continues

LISTING 11.9 Continued

```
      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.
```

When created, this thread first calls its `FindAllFiles()` method. This method uses `FindFirst()` and `FindNext()` to search for all files in the current directory matching the file specification indicated by the user. If the user has chosen to recurse subdirectories, the `DoSearch()` method is then called in order to traverse down a directory tree. This method again makes use of `FindFirst()` and `FindNext()` to find directories, but the twist is that it calls itself recursively in order to traverse the tree. As each directory is found, `FindAllFiles()` is called to process all matching files in the directory.

TIP

The recursion algorithm used by the `DoSearch()` method is a standard technique for traversing a directory tree. Because recursive algorithms are notoriously difficult to debug, the smart programmer will make use of ones that are already known to work. It's a good idea to save this method so that you can use it with other applications in the future.

To process each file, you'll notice that the algorithm for searching for a token within a file involves using the `TMemMapFile` object, which encapsulates a Win32 memory-mapped file. This object is discussed in detail in Chapter 12, "Working with Files," but for now you can just assume that this provides an easy way to map the contents of a file into memory. The entire algorithm works like this:

1. When a file matching the file spec is found by the `FindAllFiles()` method, the `SearchFile()` method is called and the file contents are copied into a string.
2. The `ScanForStr()` method is called for each file-string. `ScanForStr()` searches for occurrences of the search token within each string.
3. When an occurrence is found, the filename and/or the line of text is added to the list box. The line of text is added only when the user unchecks the File Names Only check box.

Note that all the methods in the `TSearchThread` object periodically check the status of the `StopIt` flag (which is tripped when the thread is told to stop) and the `Terminated` flag (which is tripped when the `TThread` object is to terminate).

CAUTION

Remember that any methods within a `TThread` object that modify the application's user interface in any way must be called through the `Synchronize()` method, or the user interface must be modified by sending messages.

Adjusting the Priority

Just to add yet another feature, `DelSrch` enables the user to adjust the priority of the search thread dynamically. The form used for this purpose is shown in Figure 11.7, and the unit for this form, `PRIU.PAS`, is shown in Listing 11.10.

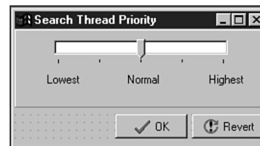

FIGURE 11.7
The thread priority form for the `DelSrch` project.

LISTING 11.10 The *PriU.pas* Unit

```
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;
```

```
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.
```

The code for this unit is fairly straightforward. All it does is set the value of the `SearchPri` variable in the main form to match that of the track bar position. If the thread is running, it also sets the priority of the thread. Because `TThreadPriority` is an enumerated type, a straight typecast maps the values `1` to `5` in the track bar to enumerations in `TThreadPriority`.

Multithreading Database Access

Although database programming isn't really discussed until Chapter 28, "Writing Desktop Database Applications," this section is intended to give you some tips on how to use multiple threads in the context of database development. If you're unfamiliar with database programming under Delphi, you might want to look through Chapter 28 before reading on in this section.

The most common request for database applications developers in Win32 is for the capability to perform complex queries or stored procedures in a background thread. Thankfully, this type of thing is supported by the 32-bit Borland Database Engine (BDE) and is fairly easy to do in Delphi.

There are really only two requirements for running a background query through, for example, a `TQuery` component:

- Each threaded query must reside within its own session. You can provide a `TQuery` with its own session by placing a `TSession` component on your form and assigning its name to the `TQuery`'s `SessionName` property. This also implies that, if your `TQuery` uses a `TDatabase` component, a unique `TDatabase` must also be used for each session.
- The `TQuery` must not be attached to any `TDataSource` components at the time the query is opened from the secondary thread. When the query is attached to a `TDataSource`, it must be done through the context of the primary thread. `TDataSource` is only used to connect datasets to user interface controls, and user interface manipulation must be performed in the main thread.

To illustrate the techniques for background queries, Figure 11.8 shows the main form for a demo project called `BDEThrd`. This form enables you to specify a BDE alias, user name, and password for a particular database and to enter a query against the database. When the Go! button is clicked, a secondary thread is spawned to process the query and the results are displayed in a child form.

The child form, `TQueryForm`, is shown in Figure 11.9. Notice that this form contains a `TQuery`, `TDatabase`, `TSession`, `TDataSource`, and `TDBGrid` component. Therefore, each instance of `TQueryForm` has its own instances of these components.

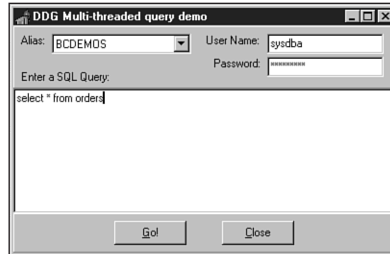

FIGURE 11.8
The main form for the `BDEThrd` *demo.*

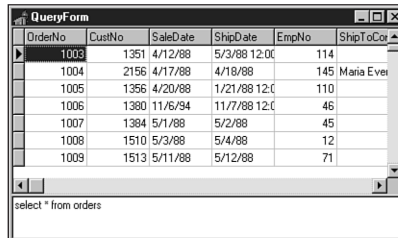

FIGURE 11.9
The child query form for the `BDEThrd` *demo.*

Listing 11.11 shows `Main.pas`, the application's main unit.

LISTING 11.11 The *`Main.pas`* Unit for the *`BDEThrd`* Demo

```
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;
```

continues

LISTING 11.11 Continued

```
    pnlTop: TPanel;
    Label1: TLabel;
    AliasCombo: TComboBox;
    Label3: TLabel;
    UserNameEd: TEdit;
    Label4: TLabel;
    PasswordEd: TEdit;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure GoButtonClick(Sender: TObject);
    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.
```

As you can see, there's not much to this unit. The `AliasCombo` combobox is filled with BDE aliases in the `OnCreate` handler for the main form using `TSession`'s `GetAliasNames()` method. The handler for the Go! button OnClick event is in charge of invoking a new query by calling the `NewQuery()` procedure that lives in a second unit, `QryU.pas`. Notice that it passes a new unique number, `FQueryNum`, to the `NewQuery()` procedure with every button click. This number is used to create a unique session and database name for each query thread.

Listing 11.12 shows the code for the `QryU` unit.

LISTING 11.12 The *QryU.pas* Unit

```
unit QryU;

interface

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

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}
```

continues

LISTING 11.12 Continued

```
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.
```

The `NewQuery()` procedure creates a new instance of the child form `TQueryForm`, sets up the properties for each of its data-access components, and creates unique names for its `TDatabase` and `TSession` components. The query's `SQL` property is filled from the `TStrings` passed in the `Qry` parameter, and the query thread is then spawned.

The code inside the `TDBQueryThread` itself is rather sparse. The constructor merely sets up some instance variables, and the `Execute()` method opens the query and calls the `HookupUI()` method through `Synchronize()` to attach the query to the data source. You should also take note of the `try..except` block inside the `Execute()` procedure, which uses `Synchronize()` to show exception messages from the context of the primary thread.

Multithreaded Graphics

We mentioned earlier that VCL isn't designed to be manipulated simultaneously by multiple threads, but this statement isn't entirely accurate. VCL has the capability to have multiple threads manipulate individual graphics objects. Thanks to new `Lock()` and `Unlock()` methods introduced in `TCanvas`, the entire `Graphics` unit has been made thread-safe. This includes the `TCanvas`, `TPen`, `TBrush`, `TFont`, `TBitmap`, `TMetafile`, `TPicture`, and `TIcon` classes.

The code for these `Lock()` methods is similar in that it uses a critical section and the `EnterCriticalSection()` API function (described earlier in this chapter) to guard access to the canvas or graphics object. After a particular thread calls a `Lock()` method, that thread is free to exclusively manipulate the canvas or graphics object. Other threads waiting to enter the portion of code following the call to `Lock()` will be put to sleep until the thread owning the critical section calls `Unlock()`, which calls `LeaveCriticalSection()` to release the critical section and let the next waiting thread (if any) into the protected portion of code. The following portion of code shows how these methods can be used to control access to a canvas object:

```
Form.Canvas.Lock;
// code which manipulates canvas goes here
Form.Canvas.Unlock;
```

To further illustrate this point, Listing 11.13 shows the unit `Main` of the `MTGraph` project—an application that demonstrates multiple threads accessing a form's canvas.

LISTING 11.13 The *Main.pas* Unit of the *MTGraph* Project

```
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;
    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;
```

continues

LISTING 11.13 Continued

```
  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;

{ 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.
```

This application has a main menu containing four items, as shown in Figure 11.10. The first item, Add thread, creates a new `TDrawThread` instance, which paints random lines on the main form. This option can be selected repeatedly in order to throw more and more threads into the mix of threads accessing the main form. The next item, Remove thread, removes the last thread

added. The third item, Add 10, creates 10 new `TDrawThread` instances. Finally, the fourth item, Remove all, terminates and destroys all `TDrawThread` instances. Figure 11.10 also shows the results of 10 threads simultaneously drawing to the form's canvas.

Canvas-locking rules dictate that as long as every user of a canvas locks it before drawing and unlocks it afterwards, multiple threads using that canvas will not interfere with each other. Note that all `OnPaint` events and `Paint()` method calls initiated by VCL automatically lock and unlock the canvas for you; therefore, existing, normal Delphi code can coexist with new background thread graphics operations.

Using this application as an example, examine the consequences or symptoms of thread collisions if you fail to properly perform canvas locking. If thread one sets a canvas's pen color to red and then draws a line, and thread two sets the pen color to blue and draws a circle, and these threads do not lock the canvas before starting these operations, the following thread collision scenario is possible: Thread one sets the pen color to red. The OS scheduler switches execution to thread two. Thread two sets the pen color to blue and draws a circle. Execution switches to thread one. Thread one draws a line. However, the line is not red, it is blue, because thread two had the opportunity to slip in between the operations of thread one.

Note also that it only takes one errant thread to cause problems. If thread one locks the canvas and thread two does not, the scenario just described is unchanged. Both threads must lock the canvas around their canvas operations to prevent that thread collision scenario.

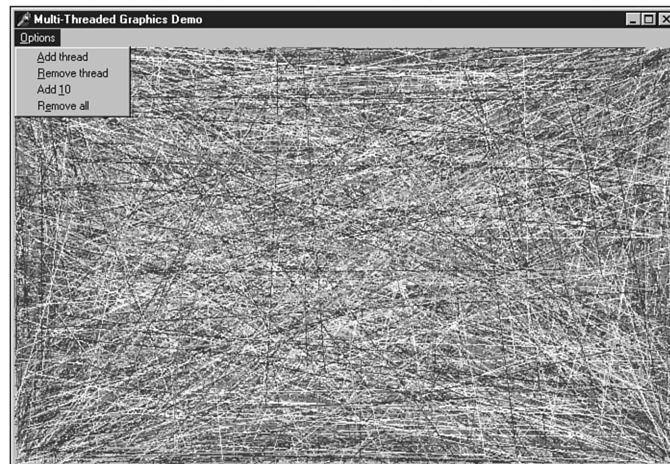

FIGURE 11.10
The `MTGraph` main form.

Summary

By now you've had a thorough introduction to threads and how to use them properly in the Delphi environment. You've learned several techniques for synchronizing multiple threads, and you've learned how to communicate between secondary threads and a Delphi application's primary thread. Additionally, you've seen examples of using threads within the context of a real-world file-search application, you've gotten the lowdown on how to leverage threads in database applications, and you've learned about drawing to a `TCanvas` with multiple threads. In the next chapter, "Working with Files," you'll learn a multitude of techniques for working with different types of files in Delphi.