Threads von Martin Strohal

Einleitung

Im Gegensatz zu anderen Programmiersprachen ist es in Delphi kein großes Problem, Thread-Objekte zu erstellen. Es gibt sogar einen Assistenten dafür. Dennoch sollte man natürlich wissen, was Threads überhaupt sind, wofür sie eingesetzt werden und worauf man achten muss.

Da es sich bei Threads um ein sehr komplexes Thema handelt, kann das Gebiet hier nur angeschnitten werden. Wer sich darüber hinaus intensiver damit beschäftigen will, dem seien die Beispielprogramme in den Verzeichnissen Demos\IPCDemos und Demos\Threads sowie Fachliteratur (z. B. "Delphi Win32-Lösungen" von Andreas Kosch oder "Delphi in Team" von der Toolbox-Redaktion) empfohlen. Die Online-Hilfe ist bei diesem Thema nicht besonders mitteilungsfreudig.

Begriffe

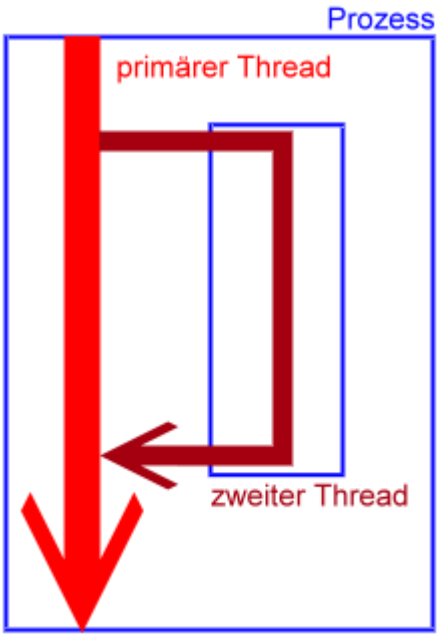
Bevor es an's eigentliche Programmieren geht, ist ein wenig Theorie unerlässlich. Deshalb zunächst ein kurzer Überblick über die wichtigsten Begriffe. Wer sich mit Betriebssystem-Grundlagen auskennt, kann diesen Abschnitt getrost überspringen.

Dass Windows "multitasking-" und "multithreadingfähig" ist, ist fast allgemein bekannt. Häufig geht das Wissen noch so weit, dass damit gemeint ist, dass mehrere Anwendungen gleichzeitig laufen können - was unter DOS nicht möglich war.
Nun müssen wir uns die Aussage "mehrere Anwendungen laufen gleichzeitig" jedoch aus Sicht des Betriebssystems genauer anschauen. Zunächst einmal werden laufende Programme als **Prozess** bezeichnet. Unter Windows 3.x sprach Microsoft noch von **Tasks** (deshalb auch "multitasking" und "Taskleiste"); da besonders auf Unix-Seite jedoch immer von Prozessen die Rede war, schwenkte Microsoft mit Windows 95 auch auf diesen Begriff über.
Ein Prozess ist also eine laufende Anwendung, zu der der geladene Programmcode und Daten gehören.
Jeder Prozess besteht aus mindestens einem **Thread**. Darunter kann man den Pfad verstehen, an dem ein Prozess abläuft. Der "Hauptthread", der beim Starten eines Prozesses erzeugt wird, wird auch als **primärer Thread** bezeichnet. Wird dieser primäre Thread beendet, ist auch der Prozess zu Ende.
Während unter dem 16-Bit-Windows (3.x) jeder Task nur aus einem Thread bestehen konnte, können 32-Bit-Anwendungen mehrere Threads gleichzeitig ausführen, um verschiedene Aktionen parallel auszuführen. Dies wird als **Multithreading** bezeichnet.

Mit dem "gleichzeitig" und "parallel" ist das natürlich so eine Sache. Die meisten Desktop-PCs besitzen lediglich einen Prozessor, der natürlich nur einen Befehl auf einmal ausführen kann. Um trotzdem den Eindruck entstehen zu lassen, mehrere Anwendungen würden gleichzeitig ablaufen, werden jedem Thread sog. **Zeitscheiben** zugeteilt. Dadurch erhält jeder Thread der Reihe nach eine bestimmte Zeit, in der er die CPU nutzen darf. Ist die Zeit um, wird dem Thread die Rechenleistung entzogen. Er wird in den Wartezustand versetzt, und der nächste Thread ist an der Reihe.

Nun haben Threads unterschiedliche **Prioritäten**. Threads mit hoher Priorität werden vom Betriebssystem bevorzugt behandelt. Beispielsweise hat der Taskmanager unter Windows die höchste Priorität, so dass man über ihn in jedem Fall Threads, die sich vermeintlich aufgehängt haben, beenden kann. Denn tritt ein Thread mit hoher Priorität auf den Plan, werden alle Threads mit niedrigerer Priorität sofort in Wartestellung versetzt. Eine normale Anwendung besitzt normale Priorität, woran ohne Grund auch nichts geändert werden sollte.
Unter Windows NT können Rechner mit mehreren Prozessoren betrieben werden. Dies ermöglicht es dann auch, dass mehrere Threads wirklich gleichzeitig abgearbeitet werden können.

Multithreading



Jede Anwendung besteht also aus mindestens einem Thread, dem primären Thread. Dafür muss nichts Besonderes getan werden. Interessant wird es erst, wenn man weitere Threads einsetzen will, um die Rechenzeit des Prozessors besser auszunutzen und um die Anwender nicht mit Sanduhren warten zu lassen.

Nebenstehende Grafik zeigt den Ablauf einer Anwendung (Prozess) mit ihrem primären Thread (roter Pfeil). Zwischendrin wird ein zweiter Thread erzeugt (dunkelroter Pfeil), der irgendwelche Aufgaben im Hintergrund und gleichzeitig zum Ablauf des primären Threads erledigt.
Als Beispiel kann man sich die Rechtschreibprüfung in Word vorstellen, die den Text überprüft, ohne den Anwender derweil mit einer Sanduhr zum Warten zu zwingen.

Allgemein gilt, dass Multithreading (also der Einsatz mehrerer Threads innerhalb eines Prozesses) verwendet werden sollte, wenn sich dadurch die Rechenzeit besser ausnutzen lässt.

Dabei sollten auch die **Nachteile** berücksichtigt werden:

- Jeder Thread muss verwaltet werden, was eine Anwendung langsamer macht und Arbeitsspeicher benötigt. Nicht umsonst weist Borland darauf hin, dass bei einem Ein-Prozessor-System 16 aktive Threads eine praktikable Obergrenze darstellen.
- Da mehrere Threads gleichzeitig Daten ändern können, muss darauf geachtet werden, dass keine Probleme auftreten. Dies kann beispielsweise durch den Einsatz von kritischen Abschnitten (Critical Section) geschehen. Dadurch wird gewährleistet, dass sich immmer nur ein einziger Thread in einem Abschnitt befindet. Alle anderen Thread müssen warten, bis dieser fertig ist, bevor sie diesen Abschnitt ebenfalls durchlaufen können. Besonders bei Mehrprozessorsystemen, auf denen Threads wirklich gleichzeitig ablaufen können, muss man als Programmierer sehr vorsichtig sein.

Letzterer Punkt bezieht sich auf synchronisiertes Multithreading. Natürlich kann es auch **unsynchronisiertes Multithreading** geben. Dabei sind die Threads voneinander völlig unabhängig und dürfen auch nicht auf globale Variablen zugreifen. Dies ist für den Programmierer am einfachsten umzusetzen, jedoch in der Praxis recht selten.
In der Regel wird synchronisiertes Multithreading eingesetzt. Darum geht es auch im nächsten Abschnitt.

Thread-Synchronisierung

Beim synchronisierten Multithreading gelten folgende Regeln:

- Nur ein Thread darf zur gleichen Zeit die Daten verändern.
- Während ein Thread die Daten verändert, darf kein zweiter Thread die Daten lesen.
- Während ein Thread die Daten liest, darf kein zweiter Thread die Daten verändern.
- Während ein Thread die Daten liest, dürfen alle anderen Threads auch die Daten lesen.

Deadlock

Beim Synchronisieren von Threads muss darauf geachtet werden, dass keine sog. Deadlock-Situation entsteht. Dabei handelt es sich um eine Verklemmung, die z.B. auftritt, wenn zwei Threads beide auf ein Signal des anderen warten. Es handelt sich also um ein Vergehen wie eine Endlosschleife, das unbedingt vermieden werden sollte.

Kritischer Abschnitt (Critical Section)

Bei einem kritischen Abschnitt handelt es sich um ein Stück im Programmablauf, das immer nur von einem einzigen Thread gleichzeitig abgearbeitet werden darf. Für alle anderen Threads ist dieser Abschnitt gesperrt. Betritt ein Thread diesen kritischen Abschnitt, aktiviert er die Sperrung, so dass der Wechsel zu einem anderen Thread des gleichen Prozesses verhindert wird. Wenn er den Abschnitt verlassen hat, muss er die Sperrung wieder aufheben, da ansonsten eine Deadlock-Situation entsteht. Es ist klar, dass kritische Abschnitte immer so kurz wie möglich gehalten werden sollten. Zum Einsatz kommt diese Technik an Stellen, an denen z.B. globale Variablen verändert werden. Hierbei muss sichergestellt werden, dass diese Daten nicht gleichzeitig von einem anderen Thread ausgelesen oder verändert werden. Delphi stellt dafür die Klasse TCriticalSection (Unit SyncObjs) bereit, auf die später eingegangen wird.

Event

Mit TEvent (Unit SyncObjs) lässt sich signalisieren, dass ein Ereignis aufgetreten ist oder ein bestimmter Status erreicht wurde. In einer Multithread-Anwendung kann TEvent verwendet werden, damit ein Thread anderen Threads signalisieren kann, dass ein Ereignis ausgelöst wurde.

TMultiReadExclusiveWriteSynchronizer

Während beim kritischen Abschnitt ein Bereich komplett für andere Threads gesperrt ist, arbeitet dieses Objekt (Unit SysUtils) - wie der Name bereits sagt - differenzierter. Nur das Schreiben wird auf einen Thread begrenzt. Das Lesen ist weiterhin allen Threads möglich.

CreateThread (Win32-API)

Die Win32-API stellt die Funktion CreateThread zur Erzeugung eines Threads zur Verfügung.

```
function CreateThread(lpSecurityAttributes: Pointer;
    dwStackSize: DWORD;
    lpStartAddress: TFNThreadStartRoutine;
    lpParameter: Pointer;
    dwCreationFlags: DOWRD;
    var lpThreadId: DWORD): THandle; stdcall;
```

Die Parameter:

lpSecurityAttributes: Zeiger auf eine Struktur mit Sicherheitsattributen (siehe auch Win32-SDK-Hilfe).

dwStackSize: Gibt die Stack-Größe für den neuen Thread in Byte an. Wird der Wert 0 übergeben, verwednet der Thread die gleiche Stack-Größe wie der primäre Thread des Prozesses. Der Speicherbereich wird automatisch im Speicherbereich des Prozesses reserviert und wird freigegeben, wenn der Thread beendet wird. Die Stack-Größe wächst, wenn nötig. Wenn nicht genügend Speicher reserviert werden, schlägt der Aufruf von CreateThread fehl.

lpStartAddress: Die Startadresse des neuen Threads. Normalerweise ist das die 32-Bit-Adresse einer Funktion, die über den neuen Thread ausgeführt werden soll.

lpParameter: Gibt einen 32-Bit-Parameter für die Thread-Funktion (unter lpStartAddress angegeben) an.

dwCreationFlags: Legt zusätzliche Eigenschaften zur Erzeugung des Threads fest. Wenn die CREATE_SUSPEND-Eigenschaft angegeben wird, wird der Thread in suspendiertem Zustand erzeugt und läuft erst ab, wenn er über ResumeThread aufgerufen wird. Wenn dieser Wert 0 ist, wird der Thread sofort nach Erzeugung gestartet.

lpThreadID: Diese Variable enthält nach der Erzeugung des Threads dessen ID, über die während seiner Laufzeit auf ihn zugegriffen werden kann.

Der **Rückgabewert** der Funktion ist ungleich 0, sofern der Aufruf erfolgreich war. Im Fehlerfall wird 0 zurückgegeben. Der genaue Fehler lässt sich dann über die Win32-API-Funktion GetLastError ermitteln.

Beispiel: Unsynchronisierter Thread

Als Beispiel erstellen wir nun eine kleine Konsolenanwendung (Datei/Neu/Konsolenanwendung) mit folgendem Code:

```
program thread;
{$APPTYPE CONSOLE}

uses Windows;

function UnserThread(zahl: Pointer): LongInt; stdcall;
begin
  Sleep(2000);
  WriteLn('UnserThread ist fertig');
  Result:=0;
```

```
end;

var
  ThreadID: DWORD;        //Thread-ID
  ThreadHandle: THandle; //Rückgabewert von CreateThread

begin
  WriteLn;
  WriteLn('Unser Thread-Testprogramm ist gestartet.');
  WriteLn('Nun erzeugen wir den neuen Thread!');

  ThreadHandle:=CreateThread(nil, 0, TFNThreadStartRoutine(@UnserThread),
nil,
                             0, ThreadID);

  //wenn der Thread erfolgreich gestartet wurde (ThreadHandle<>0), können
wir
  //ThreadHandle wieder freigeben:
  if ThreadHandle<>0 then CloseHandle(ThreadHandle);

  WriteLn('Das Hauptprogramm ist nun am Ende angekommen.');

  //Automatisches Schließen der Konsole verhindern:
  ReadLn;
end.
```

Ob überraschend oder nicht - die Ausgabe dieses Programms sieht so aus:

Warum?
Durch das Sleep wird der Thread (bestehend aus der Funktion UnserThread) so lange ausgebremst, bis das Hauptprogramm am Ende angekommen ist und das auch auf dem Bildschirm ausgegeben hat. Hätten wir auf die ReadLn-Bremse am Ende verzichtet, wäre das Konsolenfenster bereits geschlossen, noch bevor der Thread durchgelaufen ist.

Hätten wir den Thread mit dem Parameter **CREATE_SUSPEND** (fünftes Argument von CreateThread) erzeugt, wäre dieser erst losgelaufen, wenn wir ihn im primären Thread (also dem Hauptprogramm) mit

```
ResumeThread(ThreadHandle);
```

aktiviert hätten. Logischerweise muss dieser Aufruf *vor* dem Freigeben des Handles mit CloseHandle erfolgen und sollte von der Bedingung ThreadHandle<>0 abhängig sein. Denn wer einen Thread, der aus irgend einem Grund nicht erzeugt werden konnte, starten will, wird mit einer Fehlermeldung bestraft.

BeginThread

In der Unit SysUtils befindet sich eine Funktion namens BeginThread. Sie kapselt den Aufruf der oben beschriebenen CreateThread-Funktion. Zusätzlich setzt sie die globale Variable IsMultiThread und macht dabei den Heap thread-sicher.

Deshalb sollte auf die Verwendung von CreateThread verzichtet und stattdessen BeginThread verwendet werden.

```
function BeginThread(SecurityAttributes: Pointer; StackSize: LongWord;
  ThreadFunc: TThreadFunc; Parameter: Pointer; CreationFlags: LongWord;
  var ThreadId: LongWord): Integer;
```

Die Parameter entsprechen den bei CreateThread beschriebenen. Außerdem befinden sich Informationen dazu in der VCL-Hilfe.

Steuerung

In vielen Fällen ist es nötig, dass sich ein Thread selbst steuert. Dazu bestehen folgende Möglichkeiten (API-Funktionen):

MsgWaitForMultipleObjects

Diese Funktion bewirkt, dass der Thread sich selbst suspendiert, bis bestimmte Ereignisse auftreten oder die TimeOut-Zeit überschritten wird.

```
function MsgWaitForMultipleObjects(
    nCount: DWORD;
    var pHandles;
    fWaitAll: BOOL;
    dwMilliseconds,
    dwWakeMask: DWORD): DWORD;
    stdcall;
```

nCount: gibt die Anzahl der Objekthandles in dem Array an, auf die pHandles zeigt.

pHandles: Zeigt auf ein Array mit Objekthandles.

fWaitAll: Legt den Wartetyp fest. Bei true kehrt die Funktion zurück, sobald alle Objekte im pHandles-Array, Input-Ereignisse eingeschlossen, auf signalisiert gesetzt worden sind. Bei false kehrt die Funktion zurück, wenn bereits eines der Objekte auf signalisiert gesetzt wurde. Im letzten Fall verweist der Rückgabewert der Funktion auf das Objekt, dessen Zustand die Rückkehr der Funktion ausgelöst hat.

dwMilliseconds: Legt das Time-Out-Intervall in Millisekunden fest. Die Funktion kehrt zurück, sobald die Zeit abgelaufen ist, auch wenn die Kriterien, die durch fWaitAll oder dwWakeMask festgelegt wurden, nicht erfüllt sind. Ist dwMilliseconds 0, prüft die Funktion die Zustände der angegebenen Objekte und kehrt sofort zurück. Wenn dwMilliseconds INFINITE ist, läuft das Time-Out-Intervall niemals ab.

dwWakeMask: Legt Input-Typen fest, die miteinander kombiniert werden können (siehe Win32-SDK-Hilfe).

Sleep

Mit Sleep suspendiert sich ein Thread für die Dauer der angegebenen Millisekunden selbst.

```
function Sleep(cMilliseconds: DWORD); stdcall;
```

WaitForInputIdle

Beim Starten eines neuen Prozesses (Anwendung) vergeht einige Zeit, bis der Prozess vollständig initialisiert ist und auf Botschaften reagieren kann. Die Funktion WaitForInputIdle sorgt dafür, dass ein Thread, der einen neuen Prozess startet, so lange wartet, bis der neue Prozess "empfangsbereit" ist.

```
function WaitForInputIdle(
    hProcess: THandle;
    dwMilliseconds: DWORD): DWORD; stdcall;
```

hProcess: Identifiziert den Prozess, auf dessen Empfangsbereitschaft gewartet werden soll.

dwMilliseconds: Legt einen Time-Out in Millisekunden fest. Wenn dwMilliseconds INFINITE ist, wartet die Funktion so lange, bis der Prozess empfangsbereit ist.

WaitForMultipleObjects

Mit dieser Funktion kann auf das Ende mehrerer beliebiger Objekte gewartet werden.

```
function WaitForMultipleObjects(
    nCount: DWORD;
    var lpHandles;
    bWaitAll: BOOL;
    dwMilliseconds: DWORD): DWORD; stdcall;
```

nCount: Legt die Anzahl der Objekt-Handles im Array lpHandles fest.

lpHandles: Zeigt auf ein Array von Objekt-Handles.

fWaitAll: Legt den Wartetyp fest. Bei true kehrt die Funktion zurück, sobald alle Objekte im pHandles-Array, Input-Ereignisse eingeschlossen, auf signalisiert gesetzt worden sind. Bei false kehrt die Funktion zurück, wenn bereits eines der Objekte auf signalisiert gesetzt wurde. Im letzten Fall verweist der Rückgabewert der Funktion auf das Objekt, dessen Zustand die Rückkehr der Funktion ausgelöst hat.

dwMilliseconds: Legt das Time-Out-Intervall in Millisekunden fest. Die Funktion kehrt zurück, sobald die Zeit abgelaufen ist, auch wenn die Kriterien, die durch fWaitAllfestgelegt wurden, nicht erfüllt sind. Ist dwMilliseconds 0, prüft die Funktion die Zustände der angegebenen Objekte und kehrt sofort zurück. Wenn dwMilliseconds INFINITE ist, läuft das Time-Out-Intervall niemals ab.

WaitForSingleObject

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

lHandle: Identifiziert das Objekt.

dwMilliseconds: Legt das Time-Out-Intervall in Millisekunden fest. Die Funktion kehrt zurück, sobald die Zeit abgelaufen ist, auch wenn die Kriterien, die durch fWaitAllfestgelegt wurden, nicht erfüllt sind. Ist dwMilliseconds 0, prüft die Funktion die Zustände der angegebenen Objekte und kehrt sofort zurück. Wenn dwMilliseconds INFINITE ist, läuft das Time-Out-Intervall niemals ab.

TThread

Bisher haben wir nur über Funktionen aus dem Win32-API gesprochen. Aber Delphi wäre nicht Delphi, wenn es hierfür keine Vereinfachung bieten würde. So enthält die VCL eine Klasse TThread. Wir werden uns nun anschauen, wie man diese nutzen kann. Eine Vereinfachung bringt natürlich auch mit sich, dass man nicht so viele Vorgaben (mittels Parameter) machen kann. In den meisten Fällen dürfte TThread jedoch ausreichend sein.

Wichtig zu wissen ist, dass man nicht einfach eine Instanz von TThread erzeugen und verwenden kann. Man muss zunächst eine neue Klasse von TThread ableiten und an seine Anforderungen anpassen. Wenn Sie über Datei/Neu/Thread-Objekt eine Unterklasse zu TThread erzeugen, besitzt diese folgende Deklaration:

```
MyThread = class(TThread)
private
  { Private-Deklarationen }
protected
  procedure Execute; override;
end;
```

Um der Objektorientierung gerecht zu werden, sollten Sie dieser Klasse im private-Teil alle Objektfelder hinzufügen, auf die der Thread Zugriff haben soll. Desweiteren sollte der Konstruktor überschrieben werden (im zu ergänzenden Abschnitt public) und die Objektfelder initialisieren. Außerdem fehlt die Funktion, die vom Thread durchlaufen werden soll (entsprechend der Funktion UnserThread, wenn Sie sich an die Konsolenanwendung in einem früheren Abschnitt erinnern).
Unbedingt überschrieben werden muss die Methode Execute. Genau genommen steht hierin der Thread-Code. Eine zusätzliche Funktion erhöht jedoch die Übersichtlichkeit. In unserem Fall ruft die Methode Execute also nur die Methode UnserThread auf. Ist Execute abgearbeitet, beendet sich der Thread. Dabei wird das Ereignis OnTerminate ausgelöst, auf das Ihr Programm bei Bedarf reagieren kann.

Auf etwas Wichtiges weist der Kommentar in Delphis Code-Gerüst für eine neue Thread-Klasse hin: Methoden und Eigenschaften von Objekten in der VCL können nur in einer Methode namens Synchronize verwendet werden. Synchronize löst den Aufruf einer bestimmten Methode aus, die vom VCL-Haupt-Thread ausgeführt werden soll. Durch dieses indirekte Verfahren werden Konflikte in Multithread-Anwendungen vermieden.

Beispielprogramm

Um das viele Hintergrundwissen, das für den Einsatz von Threads erforderlich ist, ein wenig zu veranschaulichen, hat Borland ein Beispielprojekt erstellt, das im Unterverzeichnis Demos\Threads zu finden ist.

Öffnen Sie das Projekt thrddemo.dpr, dann können wir kurz den Aufbau betrachten. Als erstes können Sie das Programm jedoch einmal laufen lassen, um zu sehen, was überhaupt passiert. Sie sehen ein Fenster, das in drei Spalten aufgeteilt ist. In jeder befinden sich unterschiedlich lange rote Balken, völlig unsortiert. Wie Sie im Code sehen können, repräsentieren diese Balken 115 Integer-Zahlen zwischen 0 und 169. Beim Klick auf den Button sollen diese Zahlen, die sich jeweils in einem Array befinden, sortiert werden und zwar nach unterschiedlichen Sortierverfahren.
Probieren Sie es aus, welcher Sortieralgorithmus am schnellsten arbeitet! Sie sollten dann in etwa folgendes Ergebnis erhalten, wobei QuickSort am schnellsten fertig ist:

Da die Zahlen zufällig erzeugt werden, werden sich die Grafiken nach jedem Programmstart leicht unterscheiden. Wie beim Programmablauf zu sehen ist, laufen die drei Sortierverfahren gleichzeitig und nicht nacheinander ab. Hier sind also mehrere Threads im Einsatz. Genau gesagt sind es vier: der primäre Thread sowie für jedes der drei Sortierverfahren ein zusätzlicher.

Das Programm besteht aus zwei Units. ThSort enthält den Code zur Erzeugung der Zufallszahlen, zur Anzeige und zur Verwaltung der Threads. In SortThds sind die Thread-Klassen deklariert. Außerdem finden sich hier die drei eingesetzten Sortieralgorithmen.

Die Sortier-Threads

Werfen wir nun einen Blick auf die Threads, also die Unit SortThds. Da sich die drei Threads lediglich durch den verwendeten Sortieralgorithmus unterscheiden und ansonsten gleich sind, ist das ein Fall für Vererbung. Die Klasse **TSortThread** ist zunächst einmal direkter Nachkomme von TThread. Wie im letzten Abschnitt erläutert, kann TThread ja nicht direkt verwendet werden. Stattdessen muss die Klasse konkretisiert werden. In diesem Fall wird der Konstruktor Create überschrieben, um den Threads Zeiger auf das mitzugeben, mit dem sie zu arbeiten haben: eine Paintbox für die Anzeige und das Array, in dem sich die Zahlen befinden. (Objektreferenzen und var-Parameter sind immer Zeiger auf einen Speicherbereich.) Im Implementation-Teil ist zu sehen, dass im Kunstruktor die privaten Felder initialisiert werden.
Im protected-Abschnitt sind drei Methoden deklariert: Execute, die - wie wir wissen - immer überschrieben werden muss und in diesem Fall einfach Sort aufruft; VisualSwap, eine Methode, die für die grafische Ausgabe zuständig ist, und da aus einem Thread heraus bekanntlich nicht direkt auf VCL-Teile (die Paintbox) zugegriffen werden darf, macht sie Gebrauch von Synchronize. Dadurch wird die DoVisualSwap-Methode aufgerufen, die vom primären Thread ausgeführt wird, damit es nicht zu Konflikten kommt. Und schließlich ist da noch die Sortier-Methode, die jedoch abstrakt deklariert wird, da der Code (logischerweise) bei jedem Sortierverfahren unterschiedlich aussieht und deshalb in den Unterklassen implementiert werden kann.
Nun folgen die drei Thread-Klassen, die letztlich direkt zum Einsatz kommen. Sie sind Nachkommen der eben beschriebenen TSortThread-Klasse und erben somit alle Methoden und Felder. Da Sort in der Oberklasse nur virtuell-abstrakt deklariert wurde, muss nun eine Implementierung folgen. Und da jede der drei Klassen für ein anderes Sortierverfahren steht, ist das auch ohne Probleme möglich.

Mehr ist zu dieser Unit eigentlich nicht zu sagen. Der Code ist recht übersichtlich und zusätzlich kommentiert, so dass man sich schnell darin zurecht finden sollte.

Der primäre Thread

Gehen wir den primären Thread so durch, wie er auch abläuft. Es beginnt mit dem OnCreate-Ereignis des Hauptfensters. Hier werden über RandomizeArrays die drei Arrays mit Zufallszahlen gefüllt. Genauer gesagt wird nur ein Array mit Zufallszahlen gefüllt. Die anderen beiden Arrays erhalten anschließend die gleichen Werte zugewiesen.
Über "Repaint" wird ausgelöst, dass sich das Fenster neu zeichnet. Jeder der drei Paintboxen ist eine OnPaint-Ereignisbehandlungsmethode zugeordnet. BubbleSortBoxPaint für die erste, SelectionSortBoxPaint für die zweite und QuickSortBoxPaint für die dritte. Der Code in diesen drei Methoden ähnelt sich; es wird jedes Mal Paint Array aufgerufen. Nur die Parameter unterscheiden sich. Beim ersten Wert handelt es sich um die Paintbox, in der gezeichnet werden soll, und beim zweiten um das Array, das grafisch dargestellt werden soll.
So viel zu den "Vorarbeiten". Nun wird darauf gewartet, dass der Anwender auf den Button "Start Sorting" klickt. Dadurch wird die StartBtnClick-Methode aufgerufen. Und hier wird es nun spannend. Nachdem die Arrays noch einmal neu gefüllt werden, wird ein interner Thread-Zähler auf 3 gesetzt. Dann werden die Threads gestartet:

```
  with TBubbleSort.Create(BubbleSortBox, BubbleSortArray) do
    OnTerminate := ThreadDone;
```

Durch den Aufruf des Konstruktors Create wird auch automatisch die Execute-Methode aufgerufen, so dass der Thread startet. Außerdem wird dem Thread hier noch mitgeteilt, dass er auf das OnTerminate-Ereignis mit dem Ausführen der Methode "ThreadDone" reagieren soll. OnTerminate wird ausgelöst, sobald ein Thread fertig abgelaufen ist.
Nun laufen die Threads also, und in der Variablen ThreadsRunning steht, wie viele es sind. Ist ein Thread fertig, wird der Wert in ThreadDone um 1 reduziert. Sind alle Threads mit dem Sortieren am Ende angekommen, ist die Variable 0, und der Start-Button wird wieder verfügbar gemacht.

TCriticalSection, Abschluss

Auch für kritische Abschnitte gibt es eine Delphi-Vereinfachung: die VCL-Klasse TCriticalSection. Die Anwendung dieser Klasse erfolgt so:

- Einbindung der Unit SyncObjs
- Objektreferenz erstellen (also zunächst eine Variable `kritischerAbschnitt: TCriticalSection;` dann bei Programmstart die Instanz erzeugen `kritischerAbschnitt:=TCriticalSection.Create` und beim Beenden mit `Destroy` wieder freigeben.)
- Nun kann der kritische Abschnitt im Thread-Code festgelegt werden:
- `  kritischerAbschnitt.Enter;`
- `  // hier der geschützte Block`
- `  kritischerAbschnitt.Leave;`

Die Objektreferenz kritischerAbschnitt muss globale Gültigkeit haben, damit alle Threads darauf zugreifen können.
Da alle anderen Threads blockiert werden, während sich ein Thread im kritischen Abschnitt befindet, wird die Anwendung langsamer. Kritische Abschnitte sollten also nur verwendet werden, wenn es erforderlich ist.

Abschluss

Wie bereits in der Einführung erwähnt, ist dies natürlich nur ein grober Überblick, sozusagen eine Einstiegshilfe in die Arbeit mit Threads. Nach der Lektüre dieses Tutorials sind die Texte in der Online-Hilfe zu den Themen TThread, TCriticalSection, TEvent usw. möglicherweise etwas verständlicher. Für weitergehende Informationen soll nochmal auf die eingangs erwähnten Bücher verwiesen werden.
