Thread Programmierung unter Windows mit Delphi

Michael Puff
mail@michael-puff.de

2010-03-26

Inhaltsverzeichnis

1 Vorwort

Um was soll es in diesem Tutorial gehen? Wie der Titel schon sagt um Threads und ihre Programmierung unter Windows mit Delphi. Dabei werde ich erst mal auf die Theorie eingehen und diese an Hand von Win32API Code veranschaulichen. Dabei werde ich folgende Themen ansprechen:

- Grundlagen (einen Thread erzeugen / abspalten. Einen Thread beenden.)
- Steuerung von Threads (Anhalten / Fortsetzen)
- Thread Prioritäten (Prozess-Prioritäten, Thread-Prioritäten. Programmieren von Prioritäten)
- Synchronisation von Threads (atomarer Variablenzugriff. Kritische Abschnitte)
- Der Stack eines Threads
- Threadpools
- Pseudo-Threads (Fibers)
- Das VCL-Thread-OBjekt

Die Demo-Programme, welche hier angesprochen werden, könne auf meiner Homepage runtergeladen werden: http://delphitutorials.michael-puff.de. Der Code dieser Demo-Anwendungen wird sich in Auszügen auch hier im Text wiederfinden.

2 Einführung

2.1 Geschichtliches

Am Anfang, als die Threads laufen lernten, gab es kein Multitasking. Ja, man konnte es noch nicht mal Singletasking nennen. Man hat sein Programm in Lochkarten gestanzt, hat sie im Rechenzentrum abgegeben, und Tage später bekam man einen Stapel Lochkarten wieder zurück, mit oder auch oft ohne die gewünschten Resultate. War ein Fehler drin, musste die entsprechende Lochkarte ganz neu gestanzt werden.

Aber die Dinge haben sich weiterentwickelt. Das erste Konzept, bei dem mehrere Threads parallel ausgeführt wurden, tauchte bei den so genannten ttime sharing systemsäuf. Es gab einen großen zentralen Computer der mehrere Client Computer mit Rechenleistung versorgte. Man spricht von Mainframes und Workstations. Hier war es wichtig, dass die Rechenleistung bzw. die CPU-Zeit gerecht zwischen den Workstations aufgeteilt wurde. In diesem Zusammenhang erschien auch zum ersten Mal das Konzept von Prozessen und Threads. Desktop-Computer haben eine ähnliche Entwicklung durchgemacht. Frühe DOS- und Windows-Rechner waren Singletasking-Systeme. Das heißt, ein Programm lief exklusiv und allein für sich auf einem Rechner. Selbst das Betriebssystem bekam während der Ausführung eines Programms keine Rechenzeit zugeteilt. Aber die Anforderungen stiegen und die Rechner stellten immer mehr Leistung zur Verfügung. Leistung die auch heute noch die meiste Zeit, die der Rechner in Betrieb ist, brachliegt. Warum sie also nicht so effizient wie möglich nutzen, und den Rechner mehrere Dinge gleichzeitig machen lassen, wenn er kann? Das Multitasking war geboren.

2.2 Begriffsdefinition

2.2.1 Was ist ein Prozess?

Ganz allgemein kann man sagen, dass ein Prozess die laufende Instanz einer Anwendung, oder wenn man so will, eines Programms, ist. Ein Prozess besteht immer aus zwei Teilen:

1. Dem Kernel-Objekt[1], welches dem Betriebssystem zur Verwaltung des Prozesses dient, und dem
2. Adressraum, der den ausführbaren Code, die Daten und die dynamischen Speicherzuweisungen (Stack- und Heap-Zuweisungen) enthält.

[1] Ein Kernel-Objekt ist im Grunde genommen nichts weiter als ein Speicherblock, den der Kernel belegt hat. Dieser Speicherblock stellt eine Datenstruktur da, deren Elemente Informationen über das Objekt verwalten. Hinweis: Kernel-Objekte gehören dem Kernel, nicht dem Prozess.

Jeder Prozess muss mindestens einen Thread besitzen, den primären Thread. Diesen primären Thread braucht man nicht selber zu erzeugen, dies übernimmt der Loader, der den Prozess startet.. Bei Windows Programmen, die ein Fenster besitzen, enthält der primäre Thread die Nachrichtenschleife und die Fenster-Prozedur der Anwendung. Tut er dies nicht, so gäbe es keine Existenzberechtigung für ihn und das Betriebssystem würde ihn mitsamt seines Adressraumes automatisch löschen. Dies wiederum bedeutet, wird der primäre Thread eines Prozesses beendet, wird auch der zugehörige Prozess beendet, das Kernel-Objekt zerstört und der reservierte Adressraum wieder freigegeben. Ein Prozess dient also quasi als Container für den primären Thread und weiteren Threads , die im Laufe der des Programms abgespalten werden können.

Ein Prozess, oder genauer, dessen Threads führen den Code in einer geordneten Folge aus. Dabei operieren sie alle streng getrennt voneinander. Das heißt, ein Prozess kann nicht auf den Adressraum eines anderen Prozesses zugreifen. Ein Prozess sieht also die anderen Threads gar nicht und hat den Eindruck, als ob er alle Systemressourcen (Speicher, Speichermedien, Ein- / Ausgabe, CPU) für sich alleine hätte. In Wirklichkeit ist dies natürlich nicht der Fall, so dass das Betriebssystem die Aufgabe übernehmen muss, dem Prozess, dem seine eigenen «virtuellen» Ressourcen zur Verfügung stehen, reale Ressourcen zuzuordnen. In diesem Zusammenhang spricht man auch beispielsweise von virtuellen Prozessoren im System. Es gibt so viele virtuelle Prozessoren, wie es Prozesse / Threads im System gibt. Auf der anderen Seite gibt es aber auch Ressourcen, die von den Prozessen geteilt werden können. Da wären zum Beispiel dynamische Bibliotheken, die DLL's, zu nennen. Eine DLL kann von mehreren Prozessen gleichzeitig genutzt werden. Dabei wird sie nur einmal geladen, ihr Code aber in den Adressraum des Prozesses eingeblendet, welcher die DLL nutzt. Dies kann man unter anderem zur Inter Process Communication nutzen. Dies hat aber des Weiteren noch den Vorteil, dass gleicher Code nur einmal geschrieben werden muss und gleichzeitig von mehreren Prozessen genutzt werden kann.

Das Besondere ist, dass nur allein der Kernel Zugriff auf diesen Speicherblock hat. Anwendungen können ihn weder lokalisieren, noch auf ihn zugreifen. Der Zugriff kann nur über API-Funktionen erfolgen. Dies gewährleistet die Konsistenz des Kernel-Objektes.

2.2.2 Threads, die arbeitende Schicht

Ein Thread beschreibt nun einen Äusführungspfadïnnerhalb eines Prozesses. Und der Thread ist es, der den eigentlichen Programmcode ausführt. Der Prozess dient, wie schon gesagt, lediglich als Container, der die Umgebung (den Adressraum) des Threads bereitstellt.

Threads werden immer im Kontext des ihnen übergeordneten Prozesses erzeugt. Das heißt, ein Thread führt seinen Code immer im Adressraum seines Prozesses aus. Werden beispielsweise zwei oder mehr Threads im Kontext eines Prozesses ausgeführt, teilen sie sich einen einzigen Adressraum, dem des Prozesses nämlich. Sie können also denselben Code ausführen und dieselben Daten bearbeiten. Den Zugriff auf den gleichen Adressraum kann man einerseits als Segen, andererseits auch als Fluch sehen. Denn es ist zwar einfach Daten zwischen den Threads auszutauschen, nur die Synchronisation des Datenaustausches kann mit unter recht schwierig werden.

Wann immer es möglich, wenn Multithreading gefragt ist, sollte man einen eigenen Thread starten, anstatt eines eigenen Prozesses, wie man es unter 16-Bit Windows noch tun musste. Dies kannte zwar Multitasking, aber kein Multithreading. Denn im Gegensatz zum Anlegen eines Prozesses, erfordert das Starten eines Threads weniger Systemressourcen, da nur das Thread-Kernel-Objekt angelegt und verwaltet werden muss. Dass Anlegen und Verwalten eines Adressraumes fällt ja weg, da der Thread ja den Adressraum des übergeordneten Prozesses mitbenutzt. Einzig und allein ein weiterer Stack muss für den Thread im Adressraum des Prozesses angelegt werden.

2.2.3 Multitasking und Zeitscheiben

Wie setzt das Betriebssystem nun die Technik des Multitasking / Multithreading um? Das Problem ist ja, dass in heutigen herkömmlichen Desktop Computern meist nur ein Prozessor zur Verfügung steht. Diesen Prozessor müssen sich also nun die im System laufenden Prozesse teilen. Dafür gibt es zwei Ansätze unter Windows:

1. Kooperatives Multitasking (16-Bit Windows)
2. Präemptives Multitasking (32-Bit Windows)

Beim kooperativen Multitasking ist jeder Prozess selbst dafür verantwortlich Rechenzeit abzugeben und so anderen Prozessen die Möglichkeit zu geben weiter zu arbeiten. Die Prozesse müssen also «kooperieren». Wer hingegen beim präemptiven Multitasking die Zuteilung von Rechenzeit ganz alleine dem Betriebssystem obliegt. Anders wäre es beispielsweise nicht möglich gewesen den Taskmanager in das System einzufügen, denn wie will man einen anderen Prozess starten, wenn ein anderer den Prozessor in Beschlag nimmt und ihn nicht wieder freigibt? Da zudem der Taskmanager mit einer höheren Priorität ausgeführt wird, als andere Prozesse, kann er immer noch dazu genutzt werden «hängengebliebene» Prozesse zu beenden.

Beim präemptiven Multitasking wird die Rechenzeit, die zur Verfügung steht, in so genannte Zeitscheiben eingeteilt. Das heißt, ein Prozess kann die CPU für eine bestimmte Zeit nutzen, bevor das Betriebssystem die CPU diesem Prozess entzieht und sie einem anderen Prozess zuteilt. Dies gilt genauso für Threads innerhalb eines Prozesses. Denn unter 32-Bit-Windows ist die kleinste ausführbare Einheit ein Thread und nicht wie unter 16-Bit-Windows die Instanz eines Prozesses. Wird nun ein Thread unterbrochen, speichert das Betriebssystem den momentanen Zustand des Threads, sprich die Werte der CPU Register usw. und wenn dem Thread Rechenzeit zugeteilt wird, wird der Zustand der Register vom Betriebssystem wiederhergestellt und der Thread kann an der Stelle, an der er unterbrochen wurde, weiter Code ausführen. Tatsächlich wird also nichts parallel ausgeführt, sondern nacheinander. Nur erscheint die Ausführung für den Benutzer parallel, da die Zeitscheiben so kurz sind, dass es nicht auffällt und der Anschein von Parallelität erzeugt wird.

2.3 Wann sollte man Threads einsetzen und wann nicht?

Um es vorweg zu nehmen, es gibt keine festen Regeln, wann man mehrere Threads zu verwenden hat und wann nicht. Dies hängt ganz davon ab, was das Programm tun soll und wie wir das erreichen wollen. Man kann eigentlich nur vage Empfehlungen aussprechen. Was man allerdings jedem raten kann, der Multithread-Anwendungen entwickelt, ist, dass Threads umsichtig eingesetzt werden sollten. Denn: Die Implementation ist eigentlich trivial. Das eigentlich Schwere an der Sache ist, Threads interagieren zu lassen, ohne dass sie sich ins Gehege kommen.

Der Einsatz von Threads ist dann sinnvoll, wenn Code im Hintergrund ausgeführt werden soll, also Code, der keine Benutzereingaben erfordert und auch keine Ausgaben hat, die den Benutzer zum Zeitpunkt der Ausführung interessieren. Als Beispiele wären da zu nennen: Jegliche langen Berechnungen von irgendetwas, eine Folge von Primzahlen oder ähnlichem. Oder die automatische Rechtschreibkorrektur in Textverarbeitungen. Oder die Übertragung / das Empfangen von Daten über ein Netzwerk oder Port. Auch Code der asynchron ausgeführt wird, wie das Pollen eines Ports, ist besser in einem eigenen Thread aufgehoben, anstatt mit dem Vordergrund-Task zu kämpfen, der im gleichen Thread ausgeführt wird.

Ein Beispiel, wann Threads nicht sinnvoll sind: Nehmen wir an, wir haben eine Textverarbeitung und wir wollen eine Funktion zum Drucken implementieren. Schön wäre es, wenn selbige nicht die ganze Anwendung blockieren würde und man mit ihr weiter arbeiten könnte. Diese Situation scheint doch gerade zu prädestiniert für einen Thread zu sein, der das Dokument ausdruckt. Nur bei etwas genaueren Überlegungen stößt man auf Probleme. Wie druckt man ein Dokument aus, welches sich ständig ändert, weil daran weitergearbeitet wird? Eine Lösung wäre, das Dokument für die Dauer des Drucks zu sperren und dem Benutzer nur an Dokumenten arbeiten zu lassen, die sich nicht im Druck befinden. Unschön. Aber wie sieht es mit der Lösung aus? Man speichert das zu druckende Dokument in einer temporären Datei und druckt diese? Ein Thread wäre dazu nicht mehr nötig.

3 Grundlagen

3.1 Veranschaulichung

Veranschaulichen wir uns das ganze mal an Hand einer Grafik.

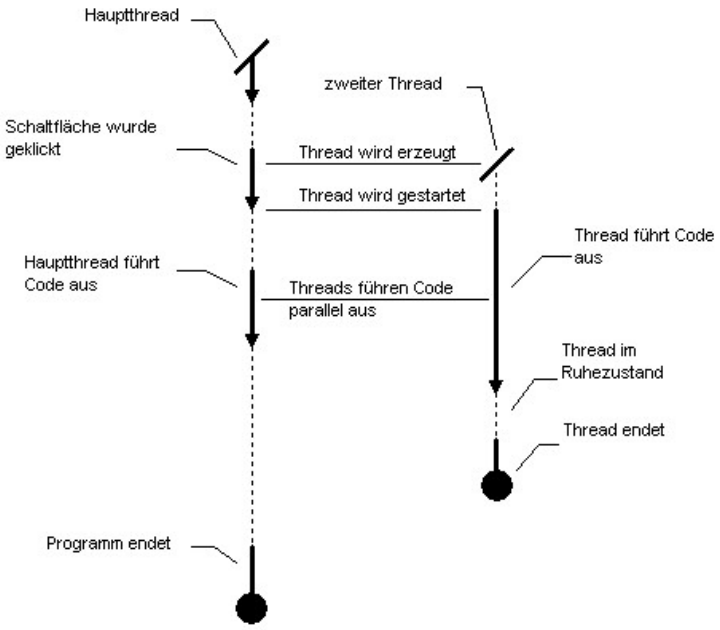

Abb. 3.1: Veranschaulichung von Threads

Legende:

- schräge Striche kennzeichnen den Beginn eines Threads
- durchgezogene Linien, wann der Thread Code ausführt
- gestrichelte Linien, wann sich der Thread im Wartezustand befindet
- Punkte bezeichnen das Ende eines Threads

Zuerst wird der Hauptthread gestartet. Dies übernimmt der Loader, wenn der Prozess erzeugt wird. Als nächstes befindet sich der Hauptthread im Wartezustand, er führt also keinen Code aus. Der Benutzer tätigt keine Eingaben und macht auch sonst nichts mit dem Fenster. Ein Klick auf eine Schaltfläche weckt den Hauptthread und veranlasst ihn Code auszuführen. In diesem Fall wird ein zweiter Thread abgespalten. Dieser wird aber nicht gleich gestartet, sondern verbringt auch erst eine Zeit im Wartezustand bevor er von Hauptthread gestartet wird. Der folgende Abschnitt der Grafik verdeutlicht wie Code von beiden Prozessen parallel

ausgeführt wird. Letztendlich enden beide Threads, wobei der zweite Thread vorher entweder automatisch endet, weil er seinen Code abgearbeitet hat oder sonst wie (dazu später) beendet wurde.

3.2 Die Thread-Funktion

Jeder Thread braucht eine Funktion, die den Code enthält, welchen der Thread ausführen soll. Diese bezeichnet man auch als Startfunktion des Threads. Die Delphi Deklaration sieht wie folgt aus:

```
type TThreadFunc = function(Parameter: Pointer): Integer;
```

Als Parameter kann optional ein Pointer übergeben werden, der auf eine Datenstruktur zeigt mit Werten, die dem Thread zum Arbeiten übergeben werden sollen. Der Rückgabewert der Thread-Funktion ist eine Ganzzahl, welche beim beenden dem Exitcode des Threads entspricht. Im Demo Programm *ThreadTimes* wird demonstriert, wie man das macht.

Zu beachten ist allerdings noch folgendes: Benutzt man *BeginThread* als Wrapper für die API-Funktion *CreateThread*, so darf man nicht die Aufrufkonvention *stdcall* für die Thread-Funktion benutzen. Grund: *BeginThread* benutzt eine TThread-WrapperFunktion anstatt des wirklichen Einsprungspunktes. Diese Funktion ist als *stdcall* definiert, sie verschiebt die Parameter auf dem Stack und ruft dann die eigene Thread-Funktion auf.

Des weiteren sollte man möglichst keine ungeschützten Zugriffe auf globale Variablen durchführen, da ein simultaner Zugriff mehrerer Threads den Variablen-Inhalt „beschädigen“ könnte. Parameter und lokale Variablen hingegen werden auf dem thread-eigenen Stack abgelegt und sind dort weniger anfällig für „Beschädigungen“. Wie man trotzdem einen sichern Zugriff auf globale Variablen realisiert wird im Kapitel sechs besprochen.

3.3 Einen Thread abspalten

Wie schon gesagt, wird der primäre Thread vom Loader automatisch angelegt. Will man nun in seinem Prozess einen weiteren Thread erzeugen, so muss man auf die Funktion CreateThread zurückgreifen:

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

In der folgenden Tabelle (Tabelle 3.1, Seite 11) findet sich eine kurze Erklärung der Parameter.

Was passiert nun nach dem Aufruf von *CreateThread*? Das System legt ein Kernel-Objekt für den Thread an, um ihn verwalten zu können. Des weiteren wird im Adressraum des übergeordneten Prozesses ein separater Stack für den Thread angelegt.

Parameter	Bedeutung
lpThreadAttributes	Zeiger auf eine Struktur, welche festlegt, ob das zurückgegeben Handle vererbt werden kann oder nicht.
dwStackSize	Legt die Größe des Stacks fest. 0 überlässt die Festlegung Windows.
lpStartAddress	Zeiger auf die Thread-Funktion.
lpParameter	Zeiger auf eine Variable, die dem Thread beim Start übergeben werden soll.
dwCreationFlags	Ein Flag, welches festlegt wie der Thread erzeugt werden soll. Dieser Flag kann entweder CREATE_SUSPENDED sein, der Thread wird im angehaltenen Zustand erzeugt oder 0, dann startet der Thread sofort nach seiner Erzeugung.
var lpThreadID	Eindeutiger Bezeichner des Threads.

Tab. 3.1: Parameter CreateThread

Der neue Thread wird im selben Kontext ausgeführt wie der Thread, der ihn erzeugte. Folge: Der neue Thread kann auf alle prozess-eigenen Handles für Kernel-Objekte, den gesamten prozess-eigenen Speicher, sowie auf alle Stacks aller anderen Threads innerhalb des Prozesses zugreifen.

In einem Delphi-Programm sollten Sie nie die Funktion *CreateThread* direkt aufrufen. Benutzen Sie stattdessen die Funktion *BeginThread*. Grund: *BeginThread* kapselt zwar nur die API-Funktion *CreateThread*, setzt aber zusätzlich noch die globale Variable *IsMultiThreaded* und macht somit den Heap thread-sicher.

3.4 Parameter an einen Thread übergeben

Über den Parameter *lpParameter* kann man dem Thread beliebige Daten übergeben. Dazu muss man einen Zeiger auf die Variable mit den Daten übergeben. Einfache Daten (Integers, Strings) kann man ohne weiteres einfach so übergeben. Will man aber mehrere Daten an den Thread übergeben, wird es geringfügig aufwendiger. Dazu erstellt man eine Struktur und übergibt dann einen Zeiger auf diese Struktur der Funktion *BeginThread*.

```
type
  TThreadParams = packed record
    Number: Integer;
    Text: String;
  end;
  PThreadParams = ^TThreadParams;
```

Um einen Zeiger an die Funktion *BeginThread* zu übergeben, hat man jetzt zwei Möglichkeiten. Man kann den Parameter entweder über den Heap übergeben oder über den Stack. Übergibt man den Parameter über den Heap, muss man vor dem Aufruf von *BeginThread* Speicher anfordern und ihn natürlich am Ende vom Thread wieder freigeben:

```
function ThreadFunc(tp: PThreadParams): Integer;
var
  Number            : Integer;
  Text              : string;
  s                 : string;
begin
  // Parameter lokalen Variablen zuweisen.
  Number := PThreadParams(tp)^.Number;
  Text := PThreadParams(tp)^.Text;
  s := 'Zahl: ' + IntToStr(Number) + #13#10 + 'Text: ' + Text;
  // ExitCode enthält Rückgabewert der Messagebox.
  Result := MessageBox(0, PChar(s), 'Thread', MB_YESNO or MB_ICONINFORMATION);
  // Reservierten Speicher für Parameter wieder freigeben.
  Dispose(tp);
end;

procedure RunThread;
var
  tp                : PThreadParams;
  Thread            : THandle;
  ThreadID          : Cardinal;
  ExitCode          : Cardinal;
begin
  // Speicher für Struktur reservieren.
  New(tp);
  // Daten den feldern der Struktur zuweisen.
  tp.Number := 42;
  tp.Text := 'Die Antwort.';
  // Thread erzeugen.
  Thread := BeginThread(nil, 0, @ThreadFunc, tp, 0, ThreadID);
  // Auf Beendigung des Threads warten.
  WaitForSingleObject(Thread, INFINITE);
  // Rückgabewert ermitteln...
  GetExitCodeThread(Thread, ExitCode);
  // ...und auswerten.
  case ExitCode of
    IDYES: Writeln('Benutzer hat "Ja" angeklickt.');
    IDNO: Writeln('Benutzer hat "Nein" angeklickt.');
  end;
  // Thread-Handle schliessen udn somit das Thread-Objekt zerstören.
  CloseHandle(Thread);
end;
```

Alternativ kann man den Parameter auch über den Stack übergeben:

```
function ThreadFunc(tp: PThreadParams): Integer;
var
  Number            : Integer;
  Text              : string;
  s                 : string;
begin
  Number := PThreadParams(tp)^.Number;
  Text := PThreadParams(tp)^.Text;
  s := 'Zahl: ' + IntToStr(Number) + #13#10 + 'Text: ' + Text;
  Result := MessageBox(0, PChar(s), 'Thread', MB_YESNO or MB_ICONINFORMATION);
end;
```

```
procedure RunThread;
var
  tp                : TThreadParams;
  Thread            : THandle;
  ThreadID          : Cardinal;
  ExitCode          : Cardinal;
begin
  tp.Number := 42;
  tp.Text := 'Die Antwort.';
  Thread := BeginThread(nil, 0, @ThreadFunc, @tp, 0, ThreadID);
  WaitForSingleObject(Thread, INFINITE);
  GetExitCodeThread(Thread, ExitCode);
  case ExitCode of
    IDYES: Writeln('Benutzer hat "Ja" angeklickt.');
    IDNO: Writeln('Benutzer hat "Nein" angeklickt.');
  end;
  CloseHandle(Thread);
end;
```

Allerdings ist hier wichtig, dass die Funktion, die den Thread erzeugt, nicht verlassen wird, bevor der Thread beendet ist. Wird die aufrufende Prozedur verlassen, wird der Stack aufgeräumt und die übergeben Daten gehen verloren, weil sie gelöscht werden oder an ihrer Stelle schon andere Daten abgelegt wurden. Schlimmsten Falls kann dies zu einem Programmabsturz führen. Siehe dazu auch die Demo-Programme *ParameterHeap* und *ParameterStack* im Ordner *Parameter*.

3.5 Beenden eines Threads

Ein Thread kann auf vier Arten beendet werden:

1. Die Thread-Funktion endet.
2. Der Thread beendet sich selbst mit der Funktion ExitThread
3. Der Thread wird durch TerminateThread beendet.
4. Der Prozess des Threads endet.

3.5.1 Die Thread-Funktion endet

Threads sollten so entworfen werden, dass sie nach Ausführung ihres Codes selbstständig enden. Grundsätzlich ist es so, dass der Thread automatisch endet, wenn er seinen Code abgearbeitet hat. Handelt es sich um einen Thread, der die ganze Zeit etwas machen soll, also einen Port pollen oder so, dann wird dieser Code normalerweise in eine Schleife gesteckt. Diese Schleife kann man mit einem break oder sonst einer Abruchbedingung verlassen und so den Thread sich selber enden lassen:

```
while bRunning <> 0 do
begin
  ...;
  ...;
end;
```

Nutzt man das Thread-Objekt der VCL kann man alternativ die Eigenschaft Terminated abfragen, welche auch von außerhalb des Threads gesetzt werden kann:

```
for Loop := 0 to ... do
begin
  ...;
  if Terminated then
    breaek;
  ...;
end;
```

Ein Thread sollte sich deshalb selber beenden, weil dann sichergestellt ist, dass

- der thread-eigene Stack wieder freigegeben wird
- der ExitCode den Rückgabewert der Thread-Funktion enthält
- der Zugriffszähler des thread-bezogenen Kernel-Objektes decrementiert wird
- alle Referenzen auf geladene DLL's und ähnlichem decrementiert werden

3.5.2 ExitThread

Eine weitere Möglichkeit einen Thread sich selber beenden zu lassen, ist die Verwendung der API Funktion ExitThread.

```
procedure ExitThread(dwExitCode: DWORD); stdcall;
```

Parameter	Bedeutung
dwExitCode	Exitcode des Threads

Tab. 3.2: Parameter ExitThread

ExitThread hat keinen Rückgabewert, da nach Aufruf, der Thread keinen Code mehr ausführen kann. Auch hier werden alle Ressourcen wieder freigegeben.

Hinweis: In einer Delphi Anwendung sollte statt ExitThread das Äquivalent zu BeginThread EndThread aufgerufen werden.

3.5.3 TerminateThread

Im Gegensatz zu *ExitThread*, welches nur den aufrufenden Thread beenden kann (schon zu erkennen an der Tatsache, dass ExitThread kein Parameter übergeben werden kann, welcher den zu beendenden Thread näher identifiziert.), kann mit TerminateThread jeder Thread beendet werden, dessen Handle man hat.

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

TerminateThread arbeitet asynchron. Das heißt, *TerminateThread* kehrt sofort zurück. Es teilt dem Betriebssystem nur mit, dass der betreffende Thread beendet werden soll. Wann letztendlich das Betriebssystem den Thread beendet, ob überhaupt (fehlende Rechte könnten dies verhindern), steht nicht fest. Kurz: Nach Rückkehr der Funktion ist nicht garantiert, dass der betreffende Thread auch beendet wurde.

Des weiteren ist zu beachten, dass alle vom Thread belegten Ressourcen erst dann wieder freigegeben werden, wenn der übergeordnete Prozess endet. Sprich der thread-eigene Stack wird zum Beispiel nicht abgebaut.

Abschließend kann man sagen, dass man vermeiden sollte einen Thread gewaltsam zu beenden. Und dies aus zwei Gründen:

1. Die belegten Ressourcen werden nicht freigegeben
2. Man weiß nicht, in welchem Zustand sich der Thread gerade befindet. Hat er beispielsweise gerade eine Datei exklusiv geöffnet, können auf diese Datei so lange keine anderen Prozesse zugreifen bis der Prozess des äbgeschossenenTThreads beendet wurde.

3.5.4 Vorgänge beim Beenden eines Threads

Wird ein Thread «sauber» beendet, werden alle zum Thread gehörenden Handles für User-Objekte freigegeben, der Exitcode wechselt von STIIL_ACTIVE auf den Rückgabewert der Thread-Funktion oder dem von *ExitThread* oder *TerminateThread* festgelegten Wert, wird der übergeordnete Prozess beendet, sofern es sich um den letzten Thread des Prozesses handelt und der Zugriffszähler des Thread–Objektes wird um eins verringert.

Mit der Funktion *GetExitCodeThread* lässt sich der Exitcode eines Threads ermitteln.

```
function GetExitCodeThread(hThread: THandle; var lpExitCode: DWORD): BOOL;
  stdcall;
```

So lange der Thread beim Aufruf noch nicht terminiert ist, liefert die Funktion den Wert STILL_ACTIVE ($103) zurück, ansonsten den Exitcode des betreffenden Threads.

Parameter	Bedeutung
hThread	Handle des Threads der beendet werden soll.
dwExitCode	Exitcode des Threads.

Tab. 3.3: Parameter TerminateThread

Parameter	Bedeutung
hThread	Handle des Threads, dessen Exitcode ermittelt werden soll.
var lpExitCode	Out Parameter der den Exitcode nach Aufruf der Funktion enthält.

Tab. 3.4: Parameter GetExitCodeThread

4 Thread Ablaufsteuerung

Beim präemptiven Multitasking – im Gegensatz zum kooperativen Multitasking, wie es bei Windows der Fall ist, wird der Zeitraum in Scheiben definierter Länge unterteilt. Den einzelnen Prozessen werden durch den Scheduler Zeitscheiben zugewiesen. Läuft ihre Zeitscheibe ab, so werden sie unterbrochen und ein anderer Thread erhält eine Zeitscheibe.

Daher ist eine wesentliche Voraussetzung für echtes präemptives Multitasking eine CPU, die verschiedene Berechtigungs-Modi beherrscht und ein Betriebssystemkern, der in der Lage ist, die verschiedenen Modi dahingehend zu nutzen, Anwendungssoftware mit unterschiedlichen Berechtigungen auszuführen. Ein einfaches Beispiel:

- Anwendungssoftware ist „unterprivilegiert“ im Sinne von „sie darf nicht alles“. Sie muss nämlich von einem Prozess mit höher Berechtigungsstufe unterbrechbar sein. Das ist die Grundvoraussetzung dafür, dass der Kernel – wie oben beschrieben – ihr
  1. eine Zeitscheibe zuteilen und den Prozess aufrufen kann,
  2. bei Ablauf der Zeitscheibe den Prozess einfrieren und „schlafen“ legen kann und
  3. bei Zuteilung einer neuen Zeitscheibe den Prozess wieder so aufwecken kann, dass der Prozess nicht merkt, dass er unterbrochen wurde
- Der Systemkern selbst läuft in einem Modus mit maximaler Berechtigungsstufe.
  1. Bei Linux unterscheidet man hier zwischen Kernel-Mode und User–Mode,
  2. bei Windows NT und Nachfolgern spricht man von „Ring 0“ als Kernel-Modus und Ringen höherer Nummerierung für „unterprivilegierte“ Prozesse.

Die Kontextstruktur des Threads enthält den Zustand der Prozessorregister zum Zeitpunkt der letzten Unterbrechung der Thread Ausführung. Das System überprüft nun ca. alle 20 Millisekunden alle vorhandenen Thread-Objekte, dabei werden nur einige Objekte als zuteilungsfähig und damit rechenbereit befunden. Es wird nun ein zuteilungsfähiges Thread-Objekt gewählt und der Inhalt der Prozessorregister aus der Kontextstruktur des Threads in die Register der CPU geladen. Diesen Vorgang bezeichnet man als Kontextwechsel.

Im aktiven Zustand nun, führt ein Thread Code aus und bearbeitet Daten. Nach 20 Millisekunden werden die Registerinhalte wieder gespeichert, der Thread befindet sich nun nicht mehr im aktiven Zustand der Ausführung und ein neues Thread-Objekt wird gewählt. Bei der Rechenzeitzuteilung werden nur zuteilungsfähige Threads berücksichtigt, das heißt Threads mit einem Unterbrechungszähler gleich 0. Besitzt ein Thread einen Unterbrechungszähler größer 0, bedeutet dies, dass er angehalten wurde und das ihm keine Prozessorzeit zugeteilt werden kann. Des weiteren bekommen auch Threads ohne Arbeit keine Rechenzeit, also Threads, die zum Beispiel auf eine Eingabe warten. Startet man zum Beispiel den Editor und gibt nichts ein, hat der Editor-Thread auch nichts zu tun und braucht auch keine Rechenzeit. Sobald aber etwas mit dem Editor geschieht, man nimmt eine Eingabe vor, oder man verschiebt das Fenster oder sollte das Fenster seine Neuzeichnung veranlassen müssen, so wird der Editor-Thread automatisch in den zuteilungsfähigen Zustand überführt.

Das heißt nun aber nicht, dass er sofort Rechenzeit zugeteilt bekommt. Es heißt nur, dass er jetzt arbeitswillig ist und das System ihm bald Rechenzeit gewährt.

4.1 Anhalten und Fortsetzen von Threads

4.1.1 Fortsetzen

Ein Element im thread-bezogenen Kernel-Objekt stellt der Unterbrechungszähler dar. Beim Erzeugen eines Threads wird er mit 1 initialisiert, der Thread ist somit nicht zuteilungsfähig. Dies stellt sicher, dass er erst ausführungsbereit ist, wenn er vollständig initialisiert ist. Nach der Initialisierung wird der Flag CREATE_SUSPENDED (5. Parameter von *CreateThread*) überprüft. Ist er nicht gesetzt, wird der Unterbrechungszähler decrementiert und somit auf 0 gesetzt, der Thread ist zuteilungsfähig. Erzeugt man einen Thread im angehaltenen Zustand, so hat man die Möglichkeit, seine Umgebung, zum Beispiel die Priorität, zu ändern. Er kann dann mit *ResumeThread* in den zuteilungsfähigen Zustand versetzt werden.

```
function ResumeThread(hThread: THandle): DWORD; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads

Tab. 4.1: Parameter ResumeThread

ResumeThread gibt bei erfolgreicher Ausführung den bisherigen Wert des Unterbrechungszählers zurück oder $FFFFFFFF im Fehlerfall.

Das Anhalten eines Threads ist akkumulativ, das heißt, wird ein Thread dreimal angehalten, muss auch dreimal *ResumeThread* aufgerufen werden, um ihn wieder in den zuteilungsfähigen Zustand zu versetzten.

4.1.2 Anhalten

Das Gegenstück zu *ResumeThread* ist *SuspendThread*. Mit dieser Funktion kann ein Thread angehalten werden.

```
function SuspendThread(hThread: THandle): DWORD; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads

Tab. 4.2: Parameter SuspendThread

SuspendThread gibt, wie auch *ResumeThread*, im Erfolgsfall den bisherigen Wert des Unterbrechungsszählers zurück. Beide Funktionen sind von jedem Thread aus aufrufbar, sofern man über das Handle des betreffenden Threads verfügt.

Analog lauten die Methoden des Thread-Objektes *Resume* und *Suspend*.

Hinweis: Ein Thread kann sich mit *SuspendThread* auch selber anhalten. Er ist aber nicht in der Lage seine Fortsetzung zu veranlassen.

Man sollte allerdings vorsichtig mit dem Aufruf von *SuspendThread* sein, da nicht bekannt ist was der Thread gerade tut. Reserviert er zum Beispiel gerade Speicher auf dem Heap und wird dabei unterbrochen, dann würde die Unterbrechung des Threads eine Sperrung des Heapzugriffes bewirken. Würden nun anschließend andere Threads versuchen auf den Heap zuzugreifen, würde ihre Ausführung so lange blockiert bis der andere Thread fortgesetzt wird und seinen Zugriff beenden kann.

4.1.3 Zeitlich begrenztes Unterbrechen

Ein Thread kann auch dem System mitteilen, dass er für eine bestimmte Zeitdauer nicht mehr zuteilungsfähig sein möchte. Dazu dient die Funktion Sleep.

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

Parameter	Bedeutung
dwMilliseconds	Dauer in Millisekunden der Unterbrechung.

Tab. 4.3: Parameter Sleep

Diese Prozedur veranlasst den Thread sich selbst, so lange anzuhalten wie in *dwMilliseconds* angegeben. Dabei gilt zu beachten: Wird *Sleep* mit 0 aufgerufen, verzichtet der Thread freiwillig auf den Rest seiner Zeitscheibe. Die Zuteilungsfähigkeit wird nur um ungefähr die angegebene Dauer in Millisekunden verzögert. Da es immer darauf ankommt, was sonst noch im System los ist. Ruft man *Sleep* mit dem Parameter INFINITE auf, wird Thread überhaupt nicht mehr zuteilungsfähig. Ist aber nicht sehr sinnvoll, da es besser wäre ihn zu beenden und somit die Ressourcen wieder freizugeben.

4.2 Temporärer Wechsel zu einem anderen Thread

Mit der Funktion SwitchToThread stellt das System eine Funktion bereit, um zu einem anderen Thread umzuschalten. Gibt es allerdings keinen zuteilungsfähigen Thread, kehrt SwitchToThread sofort zurück.

```
function SwitchToThread: BOOL; stdcall;
```

SwitchToThread entspricht fast dem Aufruf von *Sleep(0)* mit dem Unterschied, dass *SwitchToThread* die Ausführung von Threads mit niedriger Priorität gestattet, während *Sleep(0)* den aufrufenden Thread sofort neu aktiviert, auch wenn dabei Threads mit niedriger Priorität verdrängt werden.

Hinweis: Windows98 besitzt keine sinnvolle Implementation von *SwitchToThread*.

4.3 Thread Ausführungszeiten, Beispielanwendung *ThreadTimes*

Es kann vorkommen, dass man ermitteln will wie viel Zeit ein Thread für die Durchführung einer bestimmten Aufgabe benötigt. Oft findet man zu diesem Zweck Code wie diesen:

```
var
  StartTime, TimeDiff: DWORD;
begin
  StartTime := GetTickCount();

  // zu untersuchender Code

  TimeDiff := GetTickCount() - Starttime;
```

Mit diesem Code geht man allerdings davon aus, dass der Thread nicht unterbrochen wird, was aber bei einem präemptiven Betriebssystem nicht der Fall sein wird. Wird einem Thread die CPU entzogen, ist es entsprechend schwierig zu ermitteln, wie lange der Thread für die Erledigung verschiedener Aufgaben benötigt. Windows stellt allerdings eine Funktion bereit, die den Gesamtbetrag an Rechenzeit zurück gibt, die dem Thread tatsächlich zugeteilt worden ist. Diese Funktion heißt: *GetThreadTimes*.

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

Die Funktion gibt vier verschiedene Zeitwerte zurück:

Parameter	Bedeutung
CreationTime	Zeitpunkt, an dem der Thread erzeugt wurde.
ExitTime	Zeitpunkt, an dem der Thread beendet wurde. Wird der Thread noch ausgeführt, ist der Wert undefiniert.
KernelTime	... gibt an wie viel Zeit der Thread Code des Betriebssystems ausgeführt hat.
UserTime	... gibt an wie viel Zeit der Thread Anwendungscode ausgeführt hat.

Tab. 4.4: Parameter GetThreadTimes

CreationTime enthält die Zeit seit dem 1. Januar 1601 in Nanosekunden–Intervallen, das gilt auch für *ExitTime*. *KernelTime* enthält den Wert in 100 Nanosekunden Intervallen. Gilt auch für *UserTime*.

Mit Hilfe dieser Funktion lässt sich nun genau ermitteln wie viel Zeit der Thread wirklich mit dem Ausführen von Code verbracht hat. Auszug aus dem Demo *ThreadTimes*:

```
function Thread(p: Pointer): Integer;
var
  lblCounter: TStaticText;
  lblTickCount, lblThreadTimes: TStaticText;
  Counter: Integer;
  TickCountStart, TickCountEnd: Cardinal;
  KernelTimeStart, KernelTimeEnd: FileTime;
  UserTimeStart, UserTimeEnd: FileTime;
  Dummy: TFileTime;
  KernelTimeElapsed, UserTimeElapsed, TotalTime: Int64;
begin
  // Variablen initialisieren
  result := 0;
  Counter := 1;
  lblCounter := PTThreadParameter(p)^.FLabel;
  lblTickCount := PTThreadParameter(p)^.FlblTickCount;
  lblThreadTimes := PTThreadParameter(p)^.FlblThreadTimes;
  // Anfangszeit stoppen
  TickCountStart := GetTickCount();
  GetThreadTimes(GetCurrentThread, Dummy, Dummy, KernelTimeStart, UserTimeStart);
  // Code ausführen
  while (Counter < MAXCOUNT + 1) and (bRunning = 0) do
  begin
    lblCounter.Caption := IntToStr(Counter);
    Inc(Counter);
    sleep(1);
  end;
  // Endzeit stoppen
  GetThreadTimes(GetCurrentThread, Dummy, Dummy, KernelTimeEnd, UserTimeEnd);
  TickCountEnd := GetTickCount();
  // mit FileTime soll man laut PSDK nicht rechnen,
  // Konvertierung nach Int64
  // Berechnung der Differenz
  KernelTimeElapsed := Int64(KernelTimeEnd) - Int64(KernelTimeStart);
  UserTimeElapsed := Int64(UserTimeEnd) - Int64(UserTimeStart);
  TotalTime := (KernelTimeElapsed + UserTimeElapsed) div 10000;
  // Ausgabe der Ergebnisse
  lblTickCount.Caption := IntToStr(TickCountEnd - TickCountStart);
  lblThreadTimes.Caption := IntToStr(TotalTime);
  // freigeben des Speichers für die Struktur auf der der zeiger zeigt
  FreeMem(p);
end;
```

Die Beispielanwendung *ThreadTimes* (siehe Abbildung 4.1, Seite 22) demonstriert dies. In einem Label wird ein Zähler hochgezählt. Betätigt man die Schaltfläche „Stop“, wird die Zählschleife verlassen und die Ausführungszeiten genommen. Ein *Sleep(0)* sorgt dafür, dass man auch bei relativ kurzen Zeitspannen etwas „sieht“.

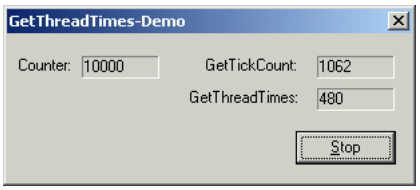

Abb. 4.1: Programmoberfläche *ThreadTimes*

5 Thread-Prioritäten

Wie schon gesagt, bekommt jeder Thread ungefähr 20 Millisekunden Rechenzeit der CPU. Dies gilt allerdings nur, wenn alle Threads die gleiche Priorität hätten, dem ist aber nicht so. Jeder Thread besitzt auch eine Priorität, das heißt „Wichtigkeit“. Diese Priorität bestimmt, welcher Thread als nächstes von der Ablaufsteuerung bei der Zuteilung von Rechenzeit berücksichtigt wird.

Windows kennt Prioritätsstufen zwischen 0 und 31, wobei 0 die niedrigste Priorität ist und 31 die höchste. Die Ablaufsteuerung berücksichtigt nun erst alle Threads mit der Prioritätsstufe 31. Findest es keine zuteilungsfähigen Threads mehr mit der Prioritätsstufe 31, kommen die anderen Threads dran. Jetzt könnte man davon ausgehen, dass Threads mit niedrigerer Priorität gar keine Rechenzeit mehr bekommen. Dies trifft nicht zu, da sich die meisten Threads im System im nichtzuteilungsfähigen Zustand befinden.

Beispiel: So lange in der Nachrichtenschlage keine Nachricht auftaucht, die von GetMessage übermittelt werden könnte, verbleibt der primäre Thread im angehaltenen Zustand. Wird nun eine Nachricht eingereiht, merkt das System dies und versetzt den Thread in den zuteilungsfähigen Zustand. Wird nun kein zuteilungsfähiger Thread mit einer höheren Priorität gefunden, wird dem Thread Rechenzeit zugewiesen.

5.1 Darstellung der Prioritäten

Die Windows-API definiert eine abstrakte Schicht oberhalb der Ablaufsteuerung, das heißt die Anwendung / der Programmierer kommuniziert niemals direkt mit der Ablaufsteuerung. Stattdessen werden Windows-Funktionen aufgerufen, die die übergebenen Parameter je nach verwendeten Betriebssystem interpretieren.

Bei der Wahl der Prioritätsklasse sollte man unter anderem berücksichtigen, welche anderen Anwendungen eventuell noch parallel mit der eigenen ausgeführt werden. Des weiteren hängt die Wahl der Prioritätsklasse auch von der Reaktionsfähigkeit des Threads ab. Diese Angaben sind sehr vage, aber sie hängen eben von der individuellen Aufgabe des Threads ab und müssen dementsprechend gewählt werden.

5.1.1 Prozess-Prioritäten

Windows kennt sechs Prioritätsklassen: „Leerlauf“, „Normal“, „Niedriger als normal“, „Normal“, „Höher als normal“, „Hoch“, „Echtzeit“. Siehe dazu Tabelle 5.1 auf Seite 24.

Prioritätsklasse	Beschreibung
Echtzeit	Die Threads in diesem Prozess müssen unmittelbar auf Ereignisse reagieren können, um zeitkritische Aufgaben durchführen zu können. Threads dieser Klasse, können sogar mit Threads des Betriebssystems konkurrieren. Nur mit äußerster Vorsicht benutzen.
Hoch	Es gilt das gleiche wie für die Klasse „Echtzeit“, nur dass Prozesse mit dieser Priorität unter der Priorität von Ëchtzeitlliegen. Der Taskmanger wird zum Beispiel mit dieser Priorität ausgeführt, um eventuell noch andere Prozesse beenden zu können.
Höher als normal	Die Ausführung erfolgt mit einer Priorität zwischen „Normal“ und „Hoch“. (neu in Windows2000)
Normal	Es werden keine besonderen Anforderungen an die Rechenzeitzuteilung gestellt.
Niedriger als normal	Die Ausführung erfolgt mit einer Priorität zwischen „Normal“ und „Leerlauf“. (neu in Windows2000)
Leerlauf	Threads in diesem Prozess werden nur ausgeführt, wenn es sonst nichts zu tun gibt. Beispielsweise Bildschirmschoner oder Hintergrundprogramme wie der Indexdienst.

Tab. 5.1: Beschreibung der Prozess–Prioritätsklassen

Die Prioritätsklasse „Hoch“ sollte nur dann Verwendung finden, wenn es unbedingt nötig ist, da auch Systemprozesse mit dieser Prioritätsklasse laufen. Die Threads des Explorers werden mit dieser Priorität ausgeführt. Dadurch wird gewährleistet, dass die Shell eine hohe Reaktionsfähigkeit besitzt und noch entsprechend reagiert, wenn andere Prozesse ansonsten das System belasten. Da die Explorer-Threads aber die meiste Zeit im angehaltenen Zustand verweilen belasten sie das System nicht, sind aber trotzdem „zur Stelle“ wenn der Benutzer sie braucht.

Die Prioritätsklasse „Echtzeit“ sollte hingegen sogar vermieden werden, da sie dann eine höhere Priorität als die System-Threads / –Prozesse hätten und sie so das Betriebssystem bei der Durchführung seiner Aufgaben behindern würden. Es könnten zum Beispiel E/A Operationen oder Netzwerkübertragungen ausgebremst werden. Auch Tastatur- und / oder Mauseingaben könnten unter Umständen nicht schnell genug verarbeitet werden, so dass für den Benutzer der Eindruck entstünde, dass das System blockiert sei. Die Prioritätsklasse „Echtzeit“ sollte nur dann Verwendung finden, wenn der Thread / Prozess die meiste Zeit nicht zuteilungsfähig ist und / oder der auszuführende Code schnell abgearbeitet ist.

Hinweis: Ein Prozess kann nur dann die Prioritätsklasse „Echtzeit“ zugewiesen werden, wenn der angemeldete Benutzer über die Berechtigung „Anheben der Zeitplanungspriorität“ verfügt. Standardmäßig haben diese Berechtigung Mitglieder der Gruppe Administratoren und Hauptbenutzer.

Die meisten Threads gehören der Prioritätsklasse „Normal“ an. Die Prioritätsklassen „Höher als normal“ und „Niedriger als normal“ wurden von Microsoft neu bei Windows2000 hinzu-

gefügt, weil sich Firmen beklagt hatten, dass die bisher verfügbaren Prioritäten nicht flexibel genug wären.

Auf Abbildung (Abb. 5.1, Seite 25 sieht man den Taskmanagers von Windows 2000 mit Darstellung der Basisprioritätsklassen der Prozesse.

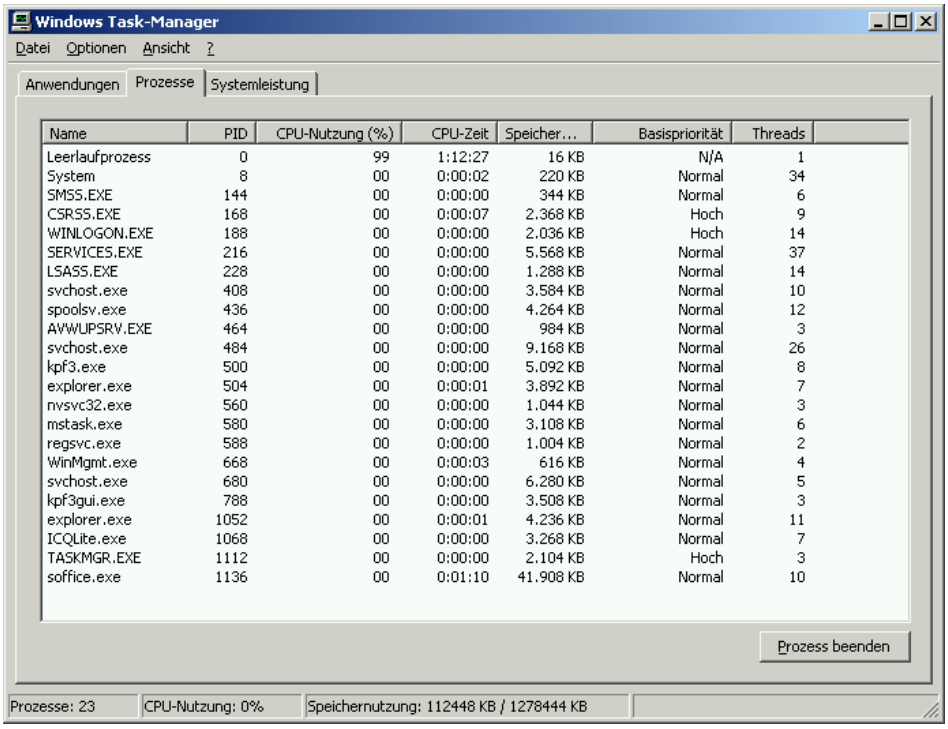

Abb. 5.1: Ansicht des Taskmanagers von Windows 2000 mit Prioritätsklassen

Wie man sieht, läuft der Taskmanager selber mit der Basisprioritätsklasse „Hoch“. Damit hat er die Möglichkeit immer noch zu reagieren, auch wenn andere Prozesse dies nicht mehr tun. Und hier besteht auch die Gefahr, wenn man einem Prozess die Basispriorität „Echtzeit“ zuweist. Dieser Prozess hätte dann eine höhere Priorität als der Taskmanager.

5.1.2 Thread-Prioritätsklassen

Windows unterstützt sieben Thread-Prioritätsklassen (siehe Tabelle 5.2, Seite 26): „Leerlauf“, „Minimum“, „Niedriger als normal“, „Normal“, „Höher als normal“, „Maximum“, „Zeitkritisch“. Diese Prioritäten sind relativ zur Prioritätsklasse des Prozesses definiert.

Die Basispriorität bezeichnet die durch die Prioritätsklasse des Prozesses festgelegte Prioritätsstufe.

Die Zuordnung der prozess-bezogenen Prioritätsklassen und der relativen Thread-Prioritäten zu einer Prioritätsstufe übernimmt das Betriebssystem. Diese Zuordnung wird von Microsoft nicht festgeschrieben, damit Anwendungen unter Umständen auch lauffähig belieben, falls

Relative Thread-Priorität	Beschreibung
Zeitkritisch	Bei geltender Prioritätsklasse Ëchtzeitärbeitet der Thread mit der Priorität 31, bei allen anderen Prioritätsklassen mit Priorität 15.
Maximum	Der Thread arbeitet mit zwei Prioritätsstufen über der Basispriorität1.
Höher als Normal	Der Thread arbeitet mit einer Prioritätsstufe über der Basispriorität.
Normal	Der Thread arbeitet mit der Basispriorität.
Niedriger als normal	Der Thread arbeitet mit einer Prioritätsstufe unterhalb der Basispriorität.
Minimum	Der Thread arbeitet mit zwei Prioritätsstufen unterhalb der Basispriorität.
Leerlauf	Bei geltender Prioritätsklasse Ëchtzeit„' arbeitet der Thread mit Priorität 16, bei allen anderen Prioritätsklassen mit Priorität 1.

Tab. 5.2: Beschreibung der Thread-Prioritätsklassen

Microsoft an dieser Stelle etwas ändert. Tatsächlich hat sich diese Zuordnung beim Wechsel zwischen den verschiedenen Versionen des Betriebssystems geändert.

Relative Thread-Priorität	Prioritätsklasse des Prozesses					
	Leerlauf	Niedriger als normal	Normal	Höher als normal	Hoch	Echtzeit
Zeitkritisch	15	15	15	15	15	31
Maximum	6	8	10	12	15	26
Höher als normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Niedriger als normal	3	5	7	9	12	23
Maximum	2	4	6	8	11	22
Leerlauf	1	1	1	1	1	16

Abb. 5.2: Zuordnung der prozess-bezogenen Prioritätsklassen und den relativen Thread-Prioritäten

Beispiel für die Tabelle (Abb.: 5.2, Seite 27):
Ein normaler Thread in einem Prozess mit hoher Priorität besitzt die Prioritätsstufe 13. Wird nun die Prioritätsklasse des Prozesses von „Hoch“ in „Leerlauf“ geändert, erhält der Thread die Prioritätsstufe 4. Grund: Die Thread-Prioritäten sind relativ zur Prioritätsklasse des Prozesses definiert. Wird die Prioritätsklasse eines Prozesses geändert, hat dies keine Auswirkung auf die relative Thread-Priorität, sondern nur auf die Zuordnung zu einer Prioritätsstufe. Die prozess-bezogene Prioritätsklasse stellt eine Abstraktion von Microsoft dar, um den internen Mechanismus der Ablaufsteuerung von der Anwendungsprogrammierung zu trennen.

5.2 Programmieren von Prioritäten

5.2.1 festlegen der Prioritätsklasse

Beim Aufruf von *CreateProcess* kann die gewünschte Prioritätsklasse über den Parameter *dwCreationFlags* festgelegt werden. Zulässige Werte sind in der folgenden Tabelle (siehe 5.3, Seite 28) aufgelistet.

Prioritätsklasse	Konstante
Echtzeit	REAL_TIME_PRIORITY_CLASS
Hoch	HIGH_PRIORITY_CLASS
Höher als normal	ABOVE_NORMAL_PRIORITY_CLASS
Normal	NORMAL_PRIORITY_CLASS
Niedriger als normal	BELOW_NORMAL_PRIORITY_CLASS
Leerlauf	IDLE_PRIORITY_CLASS

Tab. 5.3: Konstanten der Prozess-Prioritäten

Hinweis: Delphi 6 kennt die Konstanten für die Prioritätsklassen ABOVE_NORMAL_PRIORITY_CLASS und BELOW_NORMAL_PRIORITY_CLASS nicht. Grund ist der, dass diese erst ab Windows2000 verfügbar sind und von Borland in der Unit Windows.pas nicht berücksichtigt wurden. Abhilfe schafft eine eigene Deklaration dieser Konstanten:

```
BELOW_NORMAL_PRIORITY_CLASS = \$4000;
ABOVE_NORMAL_PRIORITY_CLASS= \$8000;
```

Erzeugt ein Prozess einen untergeordneten Prozess, so erbt dieser die Prioritätsklasse des Erzeuger Prozesses. Nachträglich kann die Prioritätsklasse mit der Funktion *SetPriorityClass* geändert werden.

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

Parameter	Bedeutung
hProcess	Handle für den Prozess
dwPriorityClass	Konstante für die Priorität. (Ein Wert aus Tabelle 5.3, Seite 28.)

Tab. 5.4: Parameter SetPriorityClass

Das Gegenstück dazu ist die Funktion *GetPriorityClass*, welchen die Prioritätsklasse von dem mit *hProcess* angegebenen Prozess zurück gibt.

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

Rückgabewert ist die Priorität als eine Konstante aus Tabelle 5.1 (Seite 24).

Parameter	Bedeutung
hProcess	Handle für den Prozess

Tab. 5.5: Parameter GetPriorityClass

5.2.2 Festlegen der Thread-Priorität

Da *CreateThread* keine Möglichkeit bietet die Thread-Priorität festzulegen, wird jeder neu erzeugte Thread mit der relativen Priorität „Normal“ erzeugt. Die relative Thread-Priorität kann aber mit der Funktion *SetThreadPriority* festgelegt werden.

```
function SetThreadPriority(hThread: THandle; nPriority: Integer): BOOL; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Threads
nPriority	Priorität (Ein Wert aus Tabelle 5.7, Seite 29.)

Tab. 5.6: Parameter SetThreadPriority

Parameter	Bedeutung
Zeitkritisch	THREAD_PRIORITY_TIME_CRITICAL
Maximum	THREAD_PRIORITY_HIGHEST
Höher als normal	THREAD_PRIORITY_ABOV_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Niedriger als normal	THREAD_PRIORITY_BELOW_NORMAL
Minimum	THREAD_PRIORITY_LOWEST
Leerlauf	THREAD_PRIORITY_IDLE

Tab. 5.7: Konstanten für die Thread-Prioritär

Will man nun einen Thread mit einer anderen Priorität erzeugen, so erzeugt man ihn mit *CreateThread* (oder dem Wrapper *BeginThread*) im angehaltenen Zustand CREATE_SUSPENDED), setzt mit *SetThreadPriority* die Priorität und ruft dann *ResumeThread* auf, um ihn zuteilungsfähig zu machen.

Mit dem Gegenstück zu *SetThreadPriority*, *GetThreadPriority*, lässt sich die aktuelle Thread-Priorität ermitteln.

```
function GetThreadPriority(hThread: THandle): Integer; stdcall;
```

Parameter	Bedeutung
hProcess	Handle für den Prozess

Tab. 5.8: Parameter GetThreadPriority

Rückgabewert ist die Priorität als eine Konstante aus Tabelle 5.7, Seite 29.

Hinweis: Windows stellt keine Funktion zur Verfügung, mit der sich die Prioritätsstufe eines Threads ermitteln lässt. Microsoft behält es sich vor den Algorithmus zur Ermittlung der Prioritätsstufe zu ändern. Es sollte also keine Anwendung entworfen werden, die eine Kenntnis darüber erfordert. Eine Beschränkung auf die Prioritätsklasse und der relativen Thread-Priorität sollte sicherstellen, dass die Anwendung auch unter zukünftigen Windows-Versionen lauffähig ist.

5.2.3 Prioritätsanhebung durch das System

Wie schon gesagt, ermittelt das System die Prioritätsstufe eines Threads durch die Kombination der Prioritätsklasse des übergeordneten Prozesses mit der relativen Thread-Priorität. Es kann aber die Prioritätsstufe dynamisch anheben in Folge eines E/A Ereignisses, Einreihen einer Fensternachricht in die Warteschlage oder bei einem Zugriff auf die Festplatte.

Beispiel:
Ein Thread mit der Priorität „Normal“, der in einem Prozess mit der Basispriorität „Hoch“ läuft hat die Prioritätsstufe 13. In Folge eines Tastendrucks, welcher dazu führt, dass eine WM_KEYDOWN Nachricht in die Warteschlage eingereiht wird, wird der Thread nun zuteilungsfähig und der Tastaturtreiber kann das System veranlassen die Prioritätsstufe dieses Threads um zwei Stufen zu erhöhen. Für die Dauer der nächsten Zeitscheibe läuft dieser Thread also mit der Prioritätsstufe 15. Über die nächsten Zeitscheiben hinweg erfolgt dann eine stufenweise Absenkung der Prioritätsstufe um jeweils eins, bis der Thread wieder bei seiner Basisprioritätsstufe angekommen ist.

Es ist zu beachten, dass die Prioritätsstufe niemals unter der Basisprioritätsstufe abgesenkt werden kann. Des weiteren behält es sich Microsoft vor, dieses Konzept in Zukunft zu ändern / zu verbessern, um ein optimales Reaktionsverhalten des Gesamtsystems zu erzielen. Aus diesem Grund ist es von Microsoft auch nicht dokumentiert.

Eine dynamische Prioritätsanhebung erfolgt nur bei Threads mit einer Prioritätsstufe zwischen 1 und 15. Deshalb bezeichnet man diese Prioritätsstufen auch als dynamische Prioritäten. Es erfolgt auch nie eine dynamische Anhebung in den Echtzeitbereich, da viele Systemprozesse in diesem Bereich laufen und so verhindert wird, dass diese Threads mit dem Betriebssystem konkurrieren.

Diese dynamische Anhebung kann, laut einiger Entwickler, zu einem nachteiligen Leistungsverhalten der eigenen Anwendung führen. Microsoft hat daraufhin zwei Funktionen eingeführt, um die dynamische Prioritätsstufenanhebung zu deaktivieren.

```
function SetProcessPriorityBoost(hThread: THandle; DisablePriorityBoost: Bool):
    BOOL; stdcall;

function SetThreadPriorityBoost(hThread: THandle; DisablePriorityBoost: Bool):
    BOOL; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Prozesses / Threads
DisablePriorityBoost	Aktivieren oder deaktivieren der dynamischen Prioritätsanhebung

Tab. 5.9: Parameter SetProcessPriorityBoost / SetThreadPriorityBoost

Mit den Gegenstücken

```
function GetProcessPriorityBoost(hThread: THandle; var DisablePriorityBoost: Bool
    ): BOOL; stdcall;

function GetThreadPriorityBoost(hThread: THandle; var DisablePriorityBoost: Bool)
    : BOOL; stdcall;
```

Parameter	Bedeutung
hThread	Handle des Prozesses / Threads
var DisablePriorityBoost	Boolsche Variable die den Status aufnimmt

Tab. 5.10: Parameter GetProcessPriorityBoost / GetThreadPriorityBoost

lässt sich der Status (aktiv / inaktiv) der dynamischen Prioritätsanhebung abfragen.

Es gibt noch eine weitere Situation in der eine dynamische Prioritätsstufenanhebung erfolgen kann. Gegeben sei folgendes Szenario: Thread A läuft mit der Priorität von 4. Im System existiert noch ein Thread B mit der Prioritätsstufe 8, der ständig zuteilungsfähig ist. In dieser Konstellation würde Thread A nie Rechenzeit zugeteilt bekommen. Bemerkt das System, dass ein Thread über drei bis vier Sekunden derart blockiert wird, setzt es die Prioritätsstufe des Threads für zweit aufeinander folgende Zeitscheiben auf die Stufe 15 und senkt sie danach sofort wieder die Basisprioritätsstufe ab.

5.3 Beispielanwendung *Priority-Demo*

Anhand dieser Beispielanwendung kann mit den Prioritäten experimentiert werden. Es empfiehlt sich, zwei Instanzen des Programms zu starten und den Taskmanager nebenbei zu öffnen, um zu beobachten, wie sich die verschiedenen Kombinationen der Prioritäten auf das Verhalten der Anwendungen auswirken.

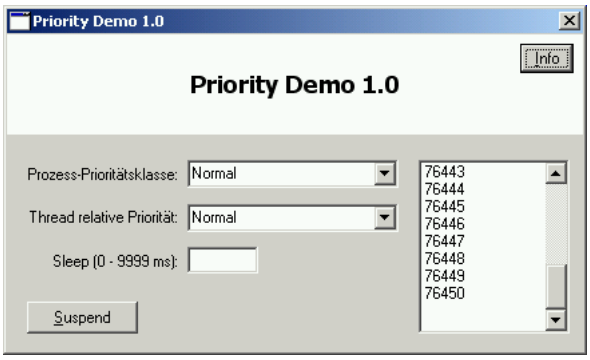

Abb. 5.3: Beispielanwendung “Priority-Demo“

Ein Klick auf den Button „Suspend“ erzeugt einen zweiten Thread, welcher einfach nur eine Messagebox anzeigt und den primären Thread lahmlegt. Dadurch kann das Hauptfenster keine Fensternachrichten mehr verarbeiten, auch kein WM_PAINT, was durch Verschieben der Messagebox deutlich wird. Die macht hinreichend deutlich, dass sich der primäre Thread im angehaltenen Zustand befindet.

Anmerkung:
Seltsamerweise scheint die Messagebox nicht angezeigt zu werden, wenn das Programm aus der IDE gestartet wird. Startet man die Anwendung direkt aus dem Explorer, funktioniert sie wie gewünscht.

6 Thread-Synchronisation

Im System haben alle Threads Zugang zu allen Systemressourcen, ansonsten könnten sie ihre Arbeit nicht verrichten. Da es aber nun nicht sein darf, dass jeder Thread, zu jeder Zeit, jede Ressource manipulieren kann – während ein Thread eine Ressource liest, darf kein anderer Thread im gleichen Moment schreibend auf die Ressource / Speicherblock zugreifen – muss der Zugriff „synchronisiert“ werden. Die Synchronisation im Benutzermodus kann entweder über einen atomaren Variablenzugriff mit den Interlocked–Funktionen erfolgen, wenn es sich um einen 32-Bit-Wert handelt oder über kritische Abschnitte, wenn es komplexe Datenstrukturen sind.

6.1 Atomarer Variablenzugriff

Unter atomaren Zugriffen versteht man, dass ein Thread mit der Gewähr auf eine Ressource zugreifen kann, ohne dass eine anderer Thread gleichzeitig auf sie zugreift. Ein Beispiel:

```
var
  g_Int: Integer = 0;
function ThreadOne(p: Pointer): Integer;
begin
  Inc(g_Int);
  result := 0;
end;

function ThreadTwo(p: Pointer): Integer;
begin
  Inc(g_Int);
  result := 0;
end;
```

Hier inkrementieren zwei Threads eine globale Integer-Variable um 1. Nun, was erwartet man, wenn beide Threads die Variable bearbeitet haben? Dass sie den Wert 2 hat, da sie ja in beiden Threads um 1 erhöht wurde. Aber muss das so sein? Nehmen wir an der Compiler generiert folgenden Code:

```
    mov eax, [g_Int]    // Wert von g_Int in Register kopieren
    inc eax             // Wert im Register inkrementieren
    mov [g_Int], eax    // Neuen Wert wieder in g_Int speichern
```

Führen nun beide Threads ihren Code nacheinander aus, passiert folgendes:

```
    mov eax, [g_Int]    // Thread 1: Kopiert 0 in Register
    inc eax             // Thread 1: Inkrementiert Registerinhalt um 1
    mov [g_Int], eax    // Thread 1: Speichert 1 in g_Int
    mov eax, [g_Int]    // Thread 2: Kopiert 1 in Register
    inc eax             // Thread 2: Inkrementiert Registerinhalt um 1
    mov [g_Int], eax    // Thread 2: Speichert 2 in g_Int
```

Das Ergebnis ist wie erwartet 2. Aber muss das so ablaufen? Windows ist ja ein präemptives Multitasking Betriebssystem, es ist also nicht gewährleistet, dass beide Threads schön nach einander durchlaufen. Wie könnte es aussehen, wenn ein Thread unterbrochen wird?

```
    mov eax, [g_Int]    // Thread 1: Kopiert 0 in Register
    inc eax             // Thread 1: Inkrementiert Registerinhalt um 1

    mov eax, [g_Int]   // Thread 2: Kopiert 0 in Register
    inc eax             // Thread 2: Inkrementiert Registerinhalt um 1
    mov [g_Int], eax   // Thread 2: Speichert 1 in g_Int

    mov [g_Int], eax    // Thread 1: Speichert 1 in g_Int
```

Und schon lautet unser Ergebnis nicht mehr 2 sondern 1! Das Ergebnis hängt von vielerlei Gegebenheiten ab: Wie wird der Code vom Compiler generiert? Welcher Prozessor führt den Code aus? Wie viele Prozessoren gibt es im System? Windows wäre als Multitasking Betriebssystem nicht zu gebrauchen, wenn es für unser Problem keine Lösung bereitstellen würde.

Und zwar stellt Windows die Interlocked-Funktionen bereit. Alle Funktionen dieser Familie haben eins gemeinsam, sie bearbeiten einen Wert auf atomare Weise. Nehmen wir als Beispiel mal die Funktion InterlockedExchangeAdd() (weitere Funktionen der Interlocked-Familie finden sich in den Demos: InterlockedExchangeAdd_Demo und SpinLock):

```
function InterlockedExchangeAdd(Addend: PLongint; Value: Longint): Longint
    stdcall; overload;

function InterlockedExchangeAdd(var Addend: Longint; Value: Longint): Longint
    stdcall; overload;
```

Parameter	Bedeutung
Addend (Plongint)	Zeiger auf ein Integer, der den neuen Wert aufnimmt.
Addend (Longint)	Variable, die den neuen Wert aufnimmt.
Value	Wert, um den inkrementiert werden soll.

Tab. 6.1: Parameter InterlockedExchangeAdd

InterlockedExchangeAdd inkrementiert einfach den ersten Parameter um den in Value angegebenen Wert und stellt sicher, dass der Zugriff atomar erfolgt. Unseren obigen Code können wir also wie folgt umschreiben:

```
function ThreadOne(p: Pointer): Integer;
begin
  InterlockedExchangeAdd(g_Int, 1);
  result := 0;
```

```
end;

function ThreadTwo(p: Pointer): Integer;
begin
  InterlockedExchangeAdd(g_Int, 1);
  result := 0;
end;
```

und so sicherstellen, dass unsere globale Variable *g_Int* korrekt und wie erwartet inkrementiert wird. Wichtig dabei ist, dass alle beteiligten Threads die Änderung an der Variablen nur atomar vornehmen.

Hinweis: Die Interlocked-Funktionen sind zu dem auch noch sehr schnell. Ein Aufruf einer Interlocked-Funktion verbraucht während ihrer Ausführung nur einige CPU-Befehlszyklen (in der Regel weniger als 50) und es findet kein Wechsel vom Benutzer- in den Kernel-Modus statt, was gewöhnlich mehr als 1000 Zyklen benötigt.

Werfen wir noch einen Blick auf die Funktion *InterlockedExchange()*:

```
function InterlockedExchange(var Target: Integer; Value: Integer): Integer;
    stdcall;
```

Parameter	Bedeutung
var Target	Variable, die den Ziel-Wert aufnimmt
Value	Wert für das Ziel

Tab. 6.2: Parameter InterlockedExchange

Diese Funktion ersetzt den Wert, der in *Target* steht, durch den Wert der in *Value* steht. Das besondere ist, dass *InterlockedExchange* den ursprünglichen Wert von *Target* als Funktions-Ergebnis zurückliefert. Damit lässt sich ein so genanntes SpinLock implementieren:

```
var
  g_IsInUse: Integer = 1;  // Status Flag; 1: in Gebrauch, 0: frei

function WatchDogThread(p: Pointer): Integer;
begin
  //  läuft so lange g_IsInUse 1 ist
  while InterlockedExchange(g_IsInUse, 1) <> 0 do
    Sleep(0);
  //  g_IsInUse ist 0 geworden, Schleife wurde verlassen
  writeln('frei.');
  result := 0;
end;
```

Die while-Schleife wird immer wieder durchlaufen, wobei sie den Wert von *g_IsInUse* auf 1 setzt und prüft, ob er vorher ungleich 0 war (1 bedeutet Ressource ist in Gebrauch, 0 auf Ressource kann zugegriffen werden). Ein anderer Thread bearbeitet die zu überwachende Ressource und setzt *g_IsInUse* auf 0, wenn er damit fertig ist. Damit liefert die while-Schleife irgendwann einmal *False*, die Ressource ist frei und wird verlassen. Dieses Vorgehen bezeichnet man als SpinLock.

Diese Technik ist allerdings nicht sehr effizient, da ein SpinLock viel Rechenzeit verbraucht, denn es muss ja ständig ein Wert vergliechen werden, der auf „wundersame“ Weise an anderer Stelle geändert wird. Außerdem setzt dieser Code voraus, dass alle Threads, die mit dem SpinLock arbeiten auf der gleichen Prioritätsstufe laufen.

Hinweis: Es gibt keine Interlocked-Funktion, die nur einen Wert liest, da eine solche Funktion überflüssig ist. Wenn ein Thread einen Wert liest, deren Inhalt mit einer Interlocked-Funktion geändert wurde, handelt es sich immer um einen brauchbaren Wert.

6.2 Kritische Abschnitte

Handelt es sich bei den Ressourcen nicht um reine 32-Bit Werte, dann kommt man nicht mehr mit den Interlocked-Funktionen aus - hier helfen jetzt kritische Abschnitte. Ein kritischer Abschnitt stellt einen Bereich innerhalb des Programmcodes dar, der bei seiner Ausführung exklusiven Zugriff auf eine oder mehrere Ressourcen benötigt. Hier mal ein Beispiel, wie kritische Abschnitte eingesetzt werden:

```
var
  g_Index: Integer = 0;
  g_cs: RTL_CRITICAL_SECTION;
  F: TextFile;
function FirstThread(p: Pointer): Integer;
begin
  result := 0;
  while g_Index < MAX_TIMES do
  begin
    // in CriticalSection eintreten, Ressource sperren
    EnterCriticalSection(g_cs);
    writeln(F, IntToStr(GetTickCount()));
    Inc(g_Index);
    // CriticalSection verlassen
    LeaveCriticalSection(g_cs);
    Sleep(0);
  end;
end;

function SecondThread(p: Pointer): Integer;
begin
  result := 0;
  while g_Index < MAX_TIMES do
  begin
    // in CriticalSection eintreten, Ressource sperren
    EnterCriticalSection(g_cs);
    Inc(g_Index);
    writeln(F, IntToStr(GetTickCount()));
    // CriticalSection verlassen
    LeaveCriticalSection(g_cs);
    Sleep(0);
  end;
end;
```

Wie man sieht greifen beide Threads auf die gleiche Datei zu (F) und schreiben einen Wert (*GetTickCount()*) in sie hinein, gleichzeitig wird eine globale Variable (*g_Index*) erhöht, welche für die Abbruchbedingung benötigt wird. Damit beim Schreiben nichts durcheinander geht, sind in den Threads die Zugriffe auf die Datei und die globale Variable mit kritischen Abschnitten geschützt.

Wie funktionieren kritische Abschnitte nun? Es wurde eine Struktur des Typs CRITICAL_SECTION mit dem Namen *g_cs* angelegt und jede Befehlsfolge, die auf gemeinsame Ressourcen zugreift wird in einen Aufruf von *EnterCriticalSection* und *LeaveCriticalSection* geklammert, denen man einen Zeiger auf die CRITICAL_SECTION-Struktur als Parameter übergibt. Das Prinzip ist nun folgendes: Tritt nun ein Thread in einen kritischen Abschnitt ein oder versucht es, wird mittels der CRITICAL_SECTION-Struktur überprüft, ob nicht schon auf die Ressource / den Code-Abschnitt zugegriffen wird. Ist die Ressource „frei“, betritt es den kritischen Abschnitt und hängt ein „Besetzt“-Schild an die Tür. Hat es die Ressourcen manipuliert und verlässt den kritischen Abschnitt wieder, wird das „Besetzt“-Schild praktisch wieder mit *LeaveCriticalSection* umgedreht, so dass es jetzt wieder „frei“ zeigt.

Wichtig ist, wie auch bei den Interlocked-Funktionen, dass jeder Thread, der auf gemeinsame Ressourcen zugreift, dies nur innerhalb eines kritischen Abschnittes tut. Ein Thread, der dies nicht tut, missachtet quasi das „Besetzt“-Schild – er prüft es gar nicht erst – und könnte so die Ressourcen kompromittieren. Andersherum, vergisst ein Thread nach Manipulation der Ressourcen ein *LeaveCriticalSection* aufzurufen, ähnelt dies dem Verhalten beim Verlassen das „Besetzt“-Schild nicht wieder umzudrehen und so zu signalisieren, dass die Ressourcen wieder «frei» sind.

6.2.1 Wie funktionieren kritische Abschnitte

Wie funktionieren jetzt aber kritische Abschnitte? Grundlage ist die Datenstruktur CRITICAL_SECTIONS. Sie ist nicht dokumentiert, da Microsoft der Meinung ist, dass sie vom Programmierer nicht verstanden werden müsste. Dies ist auch nicht nötig, da man API-Funktionen aufruft, um sie zu manipulieren. Diese Funktionen kennen die Details, die zur Manipulation nötig sind und garantieren so den konsistenten Zustand der Struktur.

In der Regel werden Variablen dieser Struktur global angelegt damit alle Threads auf sie zugreifen können. Des weiteren ist es nötig, dass die Elemente der Struktur initialisiert werden. Dies geschieht mit dem Aufruf von *InitializeCriticalSection()*:

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

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 6.3: Parameter InitializeCriticalSection

Sobald die kritischen Abschnitte nicht mehr gebraucht werden, sollte die CRITICAL_SECTION-Struktur zurückgesetzt werden mit dem Aufruf von *DeleteCriticalSection*:

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

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 6.4: Parameter DeleteCriticalSection

Wie schon im Beispiel zu sehen war, muss jedem Codestück, welches auf eine gemeinsame Ressource zugreift ein Aufruf von *EnterCriticalSection* vorangehen.

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

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 6.5: Parameter EnterCriticalSection

EnterCriticalSection untersucht dabei die Elementvariablen der CRITICAL_SECTION-Struktur und entnimmt ihnen den Zustand der Ressourcen. Dabei geschieht folgendes:

- Greift gerade kein Thread auf die Ressource zu, aktualisiert *EnterCriticalSection* die Elemente entsprechend, um anzuzeigen, dass dem aufrufenden Thread Zugriff gewährt wurde.
- Zeigt das Element an, dass der aufrufende Thread schon Zugang zu den Ressourcen hat, wird der Zähler erhöht, der festhält, wie oft dem aufrufenden Thread schon Zugriff gewährt wurde.
- Stellt die Funktion *EnterCriticalSection* fest, dass die Ressource sich im Zugriff eines anderen Threads befindet, überführt *EnterCriticalSection* den aufrufenden Thread in den Wartezustand. Das System merkt sich das und sorgt dafür dass der aufrufende Thread sofort zuteilungsfähig wird, sobald die Ressourcen wieder freigegeben werden.

Am Ende eines Codestücks, das auf eine gemeinsame Ressource zugreift, muss die Funktion *LeaveCriticalSection* aufgerufen werden.

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

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.

Tab. 6.6: Parameter LeaveCriticalSection

Die Funktion decrementiert einen Zähler, der anzeigt wie oft dem aufrufenden Thread der Zugriff auf die Ressource gewährt wurde. Fällt der Zähler auf 0, prüft die Funktion, ob weitere Threads auf die Freigabe dieses kritischen Abschnittes warten. Ist dies der Fall überführt es einen wartenden Thread in den zuteilungsfähigen Zustand. Gibt es keine wartenden Threads, aktualisiert *LeaveCriticalSection* das Element ebenfalls, um anzuzeigen, dass im Moment kein Thread auf die Ressourcen zugreift.

6.2.2 Kritische Abschnitte und SpinLocks

Wird ein Thread in den Wartezustand überführt, hat dies zur Folge, dass er von den Benutzer-Modus in den Kernel-Modus überwechseln muss. Dies nimmt etwa 1000 CPU-Befehlszyklen in Anspruch. Um nun die Effizienz von kritischen Abschnitten zu verbessern kann man den kritischen Abschnitt mit *InitializeCriticalSectionAndSpinCount* initialisieren:

```
function InitializeCriticalSectionAndSpinCount(var lpCriticalSection:
    TRTLCriticalSection;
  dwSpinCount: DWORD): BOOL; stdcall;
```

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.
dwSpinCount	Anzahl für die Schleifendurchläufe für die Abfrage des SpinLocks.

Tab. 6.7: Parameter InitializeCriticalSectionAndSpinCount

Der Parameter *dwSpinCount* gibt dabei die Anzahl der Schleifendurchläufe der Abfrage des SpinLocks an. Also wie oft der Thread versuchen soll Zugriff auf die Ressourcen zu bekommen bevor er in den Wartezustand geht und so in den Kernel-Modus wechselt. Erfolgt der Aufruf auf einem Einzelprozessorsystem wird dieser Parameter ignoriert und auf 0 gesetzt, da es nutzlos ist in diesem Fall einen SpinLockzähler zusetzen: Der Thread, der die Ressource kontrolliert, kann die Kontrolle nicht abgeben, solange ein anderer Thread aktiv wartet, also das SpinLock testet.

Der SpinCount eines kritischen Abschnittes kann mit *SetCriticalSectionSpinCount* nachträglich geändert werden:

```
function SetCriticalSectionSpinCount(var lpCriticalSection: TRTLCriticalSection;
  dwSpinCount: DWORD): DWORD; stdcall;
```

Parameter	Bedeutung
var lpCriticalSection	Zeiger auf eine RTL_CRITICAL_SECTION Struktur.
dwSpinCount	Anzahl für die Schleifendurchläufe für die Abfrage des SpinLocks.

Tab. 6.8: Parameter SetCriticalSectionSpinCount

Siehe dazu auch die Demos: *InterLockedExchangeAdd*, *CriticalSection*, *SpinLock*.

6.3 Thread-Synchronisation mit Kernel-Objekten

Im vorherigen Kapitel habe ich die Thread-Synchronisation im Benutzermodus besprochen. Ihr großer Vorteil ist, dass sie sich positiv auf das Leistungsverhalten auswirken. Bei der

Synchronisation mit Kernelobjekten ist es nötig, dass der aufrufende Thread vom Benutzermodus in den Kernelmodus wechselt. Dies ist jedoch mit nicht unerheblichen Rechenaufwand verbunden. So braucht auf der x86-Plattform die CPU ca. 1.000 Befehlszyklen für jede Richtung des Moduswechsels. Die Thread Synchronisation im Benutzermodus hat jedoch den Nachteil, dass sie gewissen Einschränkungen unterliegt und einfach eine Funktionalität nicht bietet, die man aber das eine oder andere mal braucht. Beispiele hierzu findet man in den Demos WaitForSingleObject, HandShake, WaitForMultipleObject und in den folgenden Unterkapiteln.

Hier soll es nun um die Thread-Synchronisation mit Kernelobjekten gehen. Was Kernelobjekte sind habe ich weiter oben schon mal erläutert. Im Prinzip geht es in diesem Kapitel darum eben diese Kernelobjekte, in diesem Fall beschränke ich mich des Themas willen hauptsächlich auf Thread- und Prozessobjekte, miteinander zu synchronisieren.

Hinsichtlich der Thread–Synchronisation wird bei jedem Kernelobjekt zwischen dem Zustand signalisiert und nicht–signalisiert unterschieden. Nach welchen Regeln dies geschieht ist in Windows für jeden Objekttypen eigens festgelegt. So werden zum Beispiel Prozessobjekte immer im Zustand nicht-signalisiert erzeugt und gehen in den Zustand signalisiert über, wenn der der Prozess endet. Diesen Zustandswechsel übernimmt das Betriebssystem automatisch. Innerhalb des Prozessobjektes gibt es einen bool'schen Wert, der beim Erzeugen des Objektes mit FALSE (nicht-signalisiert) initialisiert wird. Bei der Beendigung des Prozesses ändert das Betriebssystem automatisch diesen bool'schen Wert und setzt ihn auf TRUE, um den signalisierten Zustand des Objektes anzuzeigen. Soll also Code erstellt werden, der prüft, ob ein bestimmter Prozess noch aktiv ist, braucht nur, mit der dazu zur Verfügung gestellten Funktion, dieser Wert abgefragt zu werden.

Im speziellen werde ich in diesem Kapitel darauf eingehen, wie es einem Thread ermöglicht werden kann, darauf zu warten, dass ein bestimmtes Kernel-Objekt signalisiert wird und wie so etwas zur Thread-Synchronisation genutzt werden kann

6.3.1 WaitForxxx-Funktionen

WaitForSingleObject

Die Wait-Funktionen veranlassen einen Thread, sich freiwillig so lange in den Ruhezustand zu begeben, bis ein bestimmtes Kernel-Objekt signalisiert wird. Während der Thread wartet braucht er keine Rechenzeit, das Warten ist also äußerst ressourcenschonend.

Die wohl am meisten gebrauchte Wait-Funktion ist wohl WaitForSingleObject:

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

Im Demo-Programm *wfso* im Unterordner *WaitForSingleObject* habe ich die Benutzung von *WaitForSingleObject* demonstriert: program wfso;

Parameter	Bedeutung
hHandle	Handle des Kernel-Objektes auf das gewartet werden soll
dwMilliseconds	Zeit in Millisekunden die gewartet werden soll (INFINITE = keine zeitliche Begrenzung)
Rückgabewerte	WAIT_OBJECT_0: Objekt wurde signalisierten, WAIT_TIMEOUT: Zeitüberschreitung, WAIT_FAILED: Fehler beim Aufruf der Funktion (ungültiges Handle oder ähnliches)

Tab. 6.9: Parameter WaitForSingleObject

```
{\$APPTYPE CONSOLE}

uses
  Windows;

function Thread(p: Pointer): Integer;
resourcestring
  rsReturn = 'Druecken Sie die Eingabetaste - oder auch nicht...';
begin
  Writeln(rsReturn);
  Readln;
  result := 0;
end;

var
  hThread: THandle;
  ThreadID: Cardinal;
  wf: DWORD;
resourcestring
  rsThreadStart = 'Thread wird gestartet:';
  rsWO = 'Thread wurde vom Benutzer beendet';
  rsWT = 'Thread wurde nicht innerhalb von 10 Sekunden beendet';
  rsWF = 'Fehler}';
begin
  Writeln(rsThreadStart);
  Writeln('');
  hThread := BeginThread(nil, 0, @Thread, nil, 0, ThreadID);
  if hThread <> INVALID_HANDLE_VALUE then
  begin
    wf := WaitForSingleObject(hThread, 5000);
    case wf of
      WAIT_OBJECT_0: Writeln(rsWO);
      WAIT_TIMEOUT: Writeln(rsWT);
      WAIT_FAILED: Writeln(rsWT);
    end;
    CloseHandle(hThread);
  end;
end.
```

Im Hauptprogramm wird ein Thread abgespalten, den der Benutzer durch Drücken der Eingabetaste beenden kann. Das Hauptprogramm wartet mit WaitForSingleObject auf die Signalisierung des abgespaltenen Threads und zwar genau fünf Sekunden lang. Oder anders ausgedrückt: Der Aufruf von WaitForSingleObject weist das Betriebssystem an, den Thread

so lange nicht als zuteilungsfähig zu behandeln, bis entweder der Thread signalisiert oder die fünf Sekunden überschritten wurden. Dann wertet es den Rückgabewert von WaitForSingleObject aus und gibt eine entsprechende Meldung in der Konsole aus. Betätigt der Benutzer die Eingabetaste wird der Thread normal beendet und das Thread-Objekt wird signalisiert; Rückgabewert ist WAIT_OBJECT_0. Macht der Benutzer gar nichts, bricht nach fünf Sekunden die Funktion *WaitForSingleObject* das Warten ab und kehrt mit WAIT_TIMEOUT zurück. Sollte ein Fehler auftreten, weil zum Beispiel das Handle ungültig war, gibt die Funktion WAIT_FAILED zurück.

Eine praktische Anwendung wäre, wenn zum Beispiel in einem Thread etwas initialisiert wird und der Hauptthread warten muss bis der abgespaltene Thread seine Arbeit getan hat.

WaitForMultipleObjects

Des weiteren stellt Windows noch eine Funktion bereit, um den Signalisierungssatus mehrerer Kernel-Objekte gleichzeitig abzufragen: *WaitForMultipleObjects*. Siehe Demo *WaitForMultipleObjects*.

MsgWaitForMultipleObject

Überlegen wir uns mal folgenden Ablauf: Der Hauptthread startet einen weiteren Thread in dem eine längere Berechnung zum Beispiel durchgeführt wird. Natürlich soll irgendwann mit dem Ergebnis dieser Berechnung weitergearbeitet werden. Die Auslagerung in den Thread erfolgte aus dem Grund, damit das Fenster noch reagiert und der Vorgang eventuell abgebrochen werden kann. Um auf den abgespaltenen Thread zu warten benutzen wir *WaitForSingleObject*. Unser Code sähe dann ungefähr so aus:

```
hThread := BeginThread(...);
wf := WaitForSingleObject(hThread, INFINITE);
    case wf of
      WAIT_OBJECT_0: Writeln(rsWO);
      WAIT_TIMEOUT: Writeln(rsWT);
      WAIT_FAILED: Writeln(rsWT);
    end;
```

So weit so gut. Nur leider haben wir jetzt wieder genau das Problem, was wir versucht haben mit dem Thread zu umgehen: Unser Fenster reagiert nicht mehr, weil es darauf wartet, dass *WaitForSingleObject* zurückkehrt. Eine Lösung wäre jetzt mit zwei Threads zu arbeiten: Vom Hauptthread wird Thread A abgespalten, der Thread B startet und auf dessen Rückkehr wartet. Da das warten auch in einem Thread geschieht, reagiert unser Fenster noch und wir haben unser Problem gelöst. Es geht jedoch auch einfacher und ressourcenschonender mit der API Funktion *MsgWaitForMultipleObjects*. Diese Funktion bewirkt, dass der Thread sich selbst suspendiert, bis bestimmte Ereignisse auftreten oder die TimeOut-Zeit überschritten wird. Diese Ereignisse können Fensternachrichten oder Benutzereingaben sein. Dies demonstriert ganz gut folgendes kleines Programm:

```
function Thread(p: Pointer): Integer;
var
  i                     : Integer;
begin
  for i := 0 to 99 do
  begin
    Form1.ListBox1.Items.Add(IntToStr(i));
    sleep(50);
  end;
  result := 0;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  hThread               : THandle;
  ThreadID              : Cardinal;
  WaitResult            : DWORD;
  Msg                   : TMsg;
begin
  hThread := BeginThread(nil, 0, @Thread, nil, 0, ThreadID);
  repeat
    WaitResult := MsgWaitForMultipleObjects(1, hThread, False, INFINITE,
        QS_MOUSEMOVE);
    if WaitResult = WAIT_OBJECT_0 + 1 then
    begin
      while PeekMessage(Msg, Handle, 0, 0, PM_REMOVE) do
      begin
        TranslateMessage(Msg);
        DispatchMessage(Msg);
      end;
    end;
  until WaitResult = WAIT_OBJECT_0
end;
```

Führt man dieses Programm aus, wird sich die Listbox nur füllen, wenn man die Maus über dem Fenster bewegt. Benutzt man anstatt dem Flag QS_MOUSEMOVE zum Beispiel QS_KEY wird der Thread nur ausgeführt, wenn eine Taste gedrückt wird. Setzen wir nun an dieser Stelle QS_ALLINPUT ein, wird unser Thread gar nicht suspendiert, durch die Schleife und dem Aufruf von *PeekMessage* im Falle das WAIT_OBJECT_0 + 1 zurückgeliefert wird, kann der Hauptthread auf Fensternachrichten abarbeiten und unser Fenster reagiert weiterhin. Welche Flags es gibt und was sie bewirken kann man im MSDN oder PSDK unter dem entsprechenden Stichwort nachlesen.

6.4 Ereignisobjekte

Ereignisse finden meist dann Verwendung, wenn ein Thread signalisieren will, dass es mit etwas fertig ist oder mit etwas begonnen hat und andere Threads über seinen Zustand informieren will.

Mit der Funktion *CreateEvent* erzeugt man ein Ereignisobjekt:

```
function CreateEvent(lpEventAttributes: PSecurityAttributes;
  bManualReset, bInitialState: BOOL; lpName: PChar): THandle; stdcall;
```

Parameter	Bedeutung
lpEventAttributes	Zeiger auf eine Sicherheitsattributstruktur
bManuelReset	Manuel-Reset (True) oder Auto-Reset
bInitialState	Erzeugung des Objektes im signalisierten Zustand oder nicht
lpName	Dient zur Identifizierung des Objektes

Tab. 6.10: Parameter CreateEvent

Als Rückgabewert gibt *CreateEvent* ein prozessbezogenes Handle auf das Ereignisobjekt zurück.

Hat man einen Namen für das Ereignisobjekt vergeben, so kann man wieder Zugriff auf selbiges erlangen mit der Funktion *OpenEvent*:

```
function OpenEvent(dwDesiredAccess: DWORD; bInheritHandle: BOOL; lpName: PChar):
    THandle; stdcall;
```

Parameter	Bedeutung
dwDesiredAccess	Sicherheitsbeschreiber für das Objekt
bInheritHandle	Soll das Handle vererbt werden oder nicht
lpName	Name des Events

Tab. 6.11: Parameter OpenEvent

Wie üblich sollte die Funktion *CloseHandle* aufgerufen werden, wenn das Objekt nicht mehr länger gebraucht wird.

So bald das Ereignisobjekt erzeugt wurde, kann man dessen Zustand mit den Funktionen *SetEvent* und *ResetEvent* kontrollieren:

```
function SetEvent(hEvent: THandle): BOOL; stdcall;

function ResetEvent(hEvent: THandle): BOOL; stdcall;
```

Als Parameter erwarten beide Funktionen jeweils ein gültiges Handle auf ein Ereignisobjekt. *SetEvent* ändert den Zustand des Ereignisobjektes in signalisiert und *ResetEvent* ändert ihn wieder in nicht-signalisiert.

Zusätzlich hat Microsoft noch einen Nebeneffekt für erfolgreiches Warten für Auto-Reset-Ereignisse definiert: Ein Auto-Reset-Ereignisobjekt wird automatisch in den Zustand nicht-signalisiert zurückgesetzt, wenn ein Thread erfolgreich auf das Ereignis wartet. Es ist deshalb nicht nötig, extra noch ResetEvent für ein Auto-Reset-Ereignis aufzurufen. Im Gegensatz dazu hat Microsoft für ein manuelle Ereignisse keinen solchen Nebeneffekt definiert.

7 Der Stack eines Threads

7.1 Was ist der Stack?

Wenn ein neuer Prozess erzeugt wird, wird von Windows der Adressraum des Prozesses eingerichtet. Am einem Ende des Adressbereiches wird der Heap eingerichtet, der so genannte standardmäßige Heap eines Prozesses und am anderen Ende des Adressraumes der Stack des Hauptthreads. Der Heap ist ein dynamischer Speicherbereich, aus dem zur Laufzeit eines Programmes zusammenhängende Speicherabschnitte angefordert und in beliebiger Reihenfolge wieder freigegeben werden können. Die Freigabe kann sowohl manuell als auch mit Hilfe einer automatischen Speicherbereinigung erfolgen. Eine Speicheranforderung vom Heap bzw. Freispeicher wird auch dynamische Speicheranforderung genannt.

Der Unterschied zum Stack (Stapel- oder Kellerspeicher) besteht darin, dass beim Stack angeforderte Speicherabschnitte in der umgekehrten Reihenfolge wieder freigegeben werden müssen, in der sie angefordert wurden. Dies geschieht aber automatisch durch das Betriebssystem.

Globale Variablen und Instanzen von Objekten werden auf dem Heap abegelegt, während hingegen lokale Variablen und Funktionsparameter auf dem Stack abgelegt werden. So werden zum Beispiel bei einem Funktionaufruf die Parameter, je nach Aufrufkonvention auf dem Stack geschoben, um dann von der aufgerufenen Funktion von dort wieder abgeholt zu werden. Deutlich wird dass, wenn man sich den Aufruf einer einfachen MessageBox mal in Assemblercode anguckt. Die MesageBox-Funktion ist wie folgt deklariert:

```
int MessageBox(
  HWND hWnd,
  LPCTSTR lpText,
  LPCTSTR lpCaption,
  UINT uType
);
```

Als ersten Parameter erwartet die Funktion das Handle des aufrufenden Fensters, der zweite und dritte sind der Text und der Fenstertitel und der letzte definiert die Schaltflächen und das Icon, was angezeigt werden soll. Was für Code generiert nun der Compiler? Gucken wir mal:

```
push 0 ; uType
push offset Titel ; lpCaption
push offset Message ; lpText
push 0
call MessageBoxA
```

Mit `push` werden die erforderlichen Aufrufparameter auf den Stack geschoben. Und wie man sieht, geschieht das mit dem letzten zuerst, also wenn man sich die Liste der Parameter nebeneinander vorstellt, von rechts nach links. Dann wird mit `call` die Funktion selber aufgerufen. Sie findet ihre Parameter auf dem Stack, da wir sie genauso abgelegt haben, wie sie es erwartet. Kein Problem, funktioniert, alles prima. Funktion wird aufgerufen, die erwartete Messagebox erscheint, wir klicken sie weg und die Funktion kehrt zurück. So, danach kommt nichts mehr. Und warum kommt nichts mehr? Was ist mit dem Stack? Da liegen doch noch unsere Parameter rum. Nein tun sie nicht, denn die Funktion hat hinter uns hergeräumt und das Aufräumen für uns übernommen und den Stack wieder aufgeräumt. Das Ablegen von rechts nach links und dass die Funktion selber wieder aufräumt, wird mit der Aufrufkonvention stdcall vereinbart.

Des weiteren werden die Rücksprungadressen von Funktionen auf dem Stack abgelegt. Da der Stack wie ein Stapel funktioniert – bevor ich das unterste Elemente wegnehmen kann, muss ich zu vor alle nachträglich auf dem Stack abgelegten Elemente vom Stapel nehmen – lässt sich so die Verschachtelung von Funktionen sehr einfach zurückverfolgen.

7.2 Verwaltung des Stacks

Jedes mal, wenn innerhalb eines Prozesses ein neuer Thread erzeugt wird, reserviert das Betriebssystem den für den privaten Stack des Threads erforderlichen Bereich im Adressraum des Pozesses und stellt ausserdem für diesen Bereich sogleich eine bestimmte Menge physischen Speicher bereit. Standardmäßig werden dabei 1 MByte des Prozessadressraums reserviert und dafür zwei Seiten[1] physischen Speichers bereitgestellt. Diesen Standardwert kann man entweder mit einem Linker-Schalter oder mit einem Parameter der Funktion `CreateThread` beeinflussen. Siehe dazu Tabelle 3.1 auf Seite 11.

Tabelle 7.1 zeigt einen möglichen Aufbau des Stackbereichs auf eiem Rechner mit 4Kbyte Seitengröße, wobei die Reservierung ab der Adresse 0x0800000 erfolgte.

Speicheradresse	**Zustand der Speicherseite**
0x080FF000	Obergrenze des Stacks: Bereit gestellte Seite
0x080FE000	Bereit gestellte Seite mit dem Schutzattribut PAGE_GUARD
0x080FD000	Reservierte Seite
⋮	⋮
0x08003000	Reservierte Seite
0x08002000	Reservierte Seite
0x08001000	Reservierte Seite
0x08000000	Untergrenze des Stacks: Reservierte Seite

Tab. 7.1: Der Stackbereich eines Threads nach der Erstellung

Der Stack eines Threads wächst in Richtung der abnehmenden Adressen. So bald der Thread versucht, auf die mittels des Flag PAGE_GUARD geschützte „Wächterseite“ zu zugreifen, wird automatisch das Betriebssystem benachrichtigt, welches daraufhin, unmittelbar

[1] Eine Seite ist die Grundeinheit, die bei der Speicherverwaltung verwendet wird.

auf die „Wächterseite“ folgende Speicherseite bereitstellt, den Flag PAGE_GUARD von der aktuellen „Wächterseite“ entfernt und ihn stattdessen der neu bereit gestellten Seite zuweist. So wird für einen Thread immer nur so viel Speicher bereit gestellt, wie er aktuelle benötigt.

Nehmen wir an, dass der vom Thread erzeugte Aufrufbaum ziemlich umfangreich geworden ist und das Stackzeiger-Register verweist bereits auf die Adresse 0x08003004. Bei einem weiteren Funktionsaufruf muss das Betriebssystem noch mehr physischen Speicher bereitstellen. Bei der Bereitstellung physischen Speichers für die an der Adresse 0x08001000 beginnenden Seite verfährt das das Betriebssystem jedoch etwas anders. Der Unterschied besteht darin, das die neu bereit gestellte Speicherseite nicht mehr mit dem Flag PAGE_GUARD versehen wird. Siehe Tabelle 7.2.

Speicheradresse	**Zustand der Speicherseite**
0x080FF000	Obergrenze des Stacks: Bereit gestellte Seite
0x080FE000	Bereit gestellte Seite
0x080FD000	Bereit gestellte Seite
⋮	⋮
0x08003000	Bereit gestellte Seite
0x08002000	Bereit gestellte Seite
0x08001000	Bereit gestellte Seite
0x08000000	Untergrenze des Stacks: Reservierte Seite

Tab. 7.2: Vollständig Stackbereich belegter Stackbereich eines Threads

Das bedeutet jedoch, dass damit dem für den Stack reservierten Adressbereich kein zusätzlicher physischer Speicher mehr bereit gestellt werden kann. Die unterste Speicherseite bleibt immer reserviert, und es kann kein physischer Speicher für diese Seite bereit gestellt werden. Bei der Bereitstellung physishcen Speichers an der Adresse 0x0801000 löst das Betriebssystem des weiteren eine Ausnahmebedingung (Exception) des Typs EXCEPTION_STACK_OVERFLOW aus. Diese kann man sich zunutze machen, um auf einen Stacküberlauf zu reagieren, um ein Datenverlust im Programmablauf vorzubeugen. Wie dies konkret aussieht kann man in der Beispielanwendung *StackOverflow* am Ende des Kapitel sehen.

Falls der Thread trotz der den Stacküberlauf ignalisierten Ausnahmebedingung mit der Verwendung des Stacks fortfährt und versucht auf den Speicher, der sich an Adresse 0x08000000 befindet, zu zugreifen, generiert das Betriebssystem eine Speicherschutzverletzung, da dieser Speicherbereich lediglich reserviert, aber nicht bereit gestellt ist. Das Betriebssystem übernimmt dann die Kontrolle und beendet nicht nur den „schuldigen“ Thread, sondern mit diesem auch den gesamten Prozess. Dieser verschwindet dabei kommentarlos ohne dass der Benutzer davon unterrichtet wird.

Aber warum wird die unterste Seite eines jeden Stackbereichs überhaupt stets reserviert? Die Reservierung dieser niemals bereit gestellten Seite verhindert ungewolltes überschreiben anderer vom Prozess verwendeter Daten. Es wäre ja durchaus möglich, dass an der Adresse 0x07FFF000, also eine Seite unter der Adresse 0x08000000, für einen weiteren Adressbereich physischer Speicher bereit gestellt worden ist. Falls darüber hinaus auch für die an 0x08000000 beginnende Seite physischer Speicher bereitstünde, würden Zugriffe auf

diese Seite vom Betriebssystem nicht mehr abgefangen. Ein Überlauf des Stacks bliebe demnach unentdeckt und der Thread könnte ungehindert andere im Adressraum des Prozesses befindliche Daten überschreiben. Derartige Programmfehler sind äußerst schwierig aufzuspüren.

Damit lässt sich auch die Frage beantworten, wie viele Threads man maximal in einem Prozess starten kann. Die Anzahl der Threads ist nur durch den Speicher begrenzt. Geht man davon aus, dass jeder Thread einen Stack der Größe 1 Megabyte zugewiesen bekommt, kann man theoretisch 2028 Threads erzeugen. Reduziert man die Stackgröße pro Thread, sind auch mehr möglich. Ob es sinnvoll ist so viele Threads zu erzeugen sei dahingestellt. Besser wäre es wahrscheinlich so gennante Threadpools zu verwenden, die man entweder selber implementiert oder in dem man die von Windows bereitgestellten API-Funktionen (`QueueUserWorkItem`) nutzt und Windows die Verwaltung überlässt.

7.3 Beispielanwendung *Stackoverflow*

Die Beispielanwendung *StackOverflow* berechnet die Summe aller Zahlen von 0 bis zum angegebenen Endwert. Dabei wird ein rekursiver Algorithmus benutzt, um eine Exception des Typs EXCEPTION_STACK_OVERFLOW (in Delphi EStackOverflow) zu provozieren.

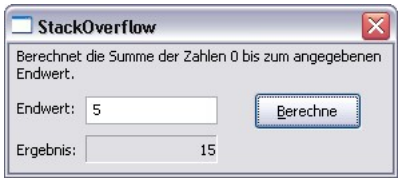

Abb. 7.1: Beispielanwendung *Stackoverflow*

Das Prinzip des Programms ist recht einfach. Nach dem der Benutzer einen Endwert eingegeben hat, wird ein neuer Thread gestartet.

```
MaxValue := GetDlgItemInt(Handle, IDC_EDT_VALUE, Translated, False);
hThread := BeginThread(nil, 0, @SumThread, Pointer(MaxValue), 0, ThreadID);
// Auf Thread warten.
WaitForSingleObject(hThread, INFINITE);
// ExitCode enthält die Summe.
GetExitCodeThread(hThread, Summe);
// Thread-Handle schliessen
CloseHandle(hThread);
// Wenn Rückgabewert = MaxInt dann ist ein Stacküberlauf aufgetreten.
if Summe = MaxInt then
  SetDlgItemText(Handle, IDC_STC_RESULT, PChar('Überlauf'))
else
  SetDlgItemInt(Handle, IDC_STC_RESULT, Summe, False);
```

Dieser Thread wiederum ruft rekursiv eine Funktion zum Berechnen der Summe auf:

```
function Sum(Num: Integer): Integer;
begin
  if Num = 0 then
```

```
    result := 0
  else
    result := Num + Sum(Num - 1);
end;

function SumThread(p: Pointer): Integer;
var
  Max               : Integer;
  Summe             : Integer;
begin
  // Parameter p enthält die Anzahl der zu addierenden Zahlen.
  Max := Integer(p);
  try
    // Um eine durch einen Stackueberlauf ausgelöste Ausnahme abzufangen,
    // muss die Funktion Sum innerhalb eines try-except-Blockes aufgerufen werden
    Summe := Sum(Max);
  except
    on E: EStackOverflow do
      // Bei einem Stackoverflow bekommt Sum den Wert MaxInt.
      Summe := MaxInt;
  end;
  Result := Summe;
end;
```

Der Thread gibt die berechnete Summe in seinem ExitCode zurück oder, wenn ein Stacküberlauf aufgetreten ist, den Wert der Konstante `MaxInt`.

8 Threadpools

In viele Situationen werden Threads erzeugt, die dann einen Großteil ihrer Zeit entweder damit verbringen zu warten und auf ein Ereignis warten. Andere Threads wiederum warten nur, um dann periodisch etwas abzufragen bzw. Statusinformationen zu aktualisieren. Mit Hilfe eines Thread Pools kann man nun mehrere Threads effizient verwalten bzw. vom System selber verwalten lassen. Man spricht dabei von so genannten *Worker Threads*, die von einem übergeordneten Thread verwaltet werden. So bald etwas anliegt, erzeugt der übergeordnete Thread einen Worker Thread bzw. aktiviert einen Worker Thread aus dem Thread Pool, der dann die zugehörige Callback-Funktion ausführt.

Die Implementierung ist relativ einfach, da das System die gesamte Verwaltung übernimmt. Um eine Funktion dem Threadpool hinzuzufügen reicht es die Funktion

```
function QueueUserWorkItem(LPTHREAD_START_ROUTINE: Pointer; Context: Pointer;
    Flags: DWORD): DWORD; stdcall; external
  'kernel32.dll';
```

aufzurufen. Die Methode wird ausgeführt, wenn ein Thread des Threadpools verfügbar wird.

Parameter	Bedeutung
LPTHREAD_START_ROUTINE	Zeiger auf die Thread-Funktion, die den auszuführenden Code enthält.
Context	Parameter, die an den Thread übergeben werden sollen.
Flags	Zusätzliche Flags, die das Verhalten steuern.

Tab. 8.1: Parameter QueueUserWorkItem

Die möglichen Flags können im SDK bzw. MSDN[1] nachgelesen werden. An dieser Stelle nur mal die wichtigsten.

[1] http://msdn2.microsoft.com/en-us/library/ms684957.aspx

Flag	Bedeutung
WT_EXECUTEDEFAULT	Die Callback-Funktion wird an einen Thread übergeben, der I/O completion ports benutzt, was bedeutet, dass er nicht in einen signalisierten Zustand übergehen kann.
WT_EXECUTEINIOTHREAD	Die Callback-Funktion wird an einen I/O Worker Thread übergeben, welcher seinen Zustand signalisieren kann. Dies ist allerdings nicht sehr effizient, deshalb sollte dieser Flag nur benutzt werden, wenn die Callback-Funktion APCs generiert.
WT_EXECUTELONGFUNCTION	Signalisiert dem System, dass die Callback-Funktion länger benötigt, um zurückzukehren. Dies hilft dem System zu entscheiden, ob ein neuer Thread erzeugt werden oder auf bereits laufende Threads gewartet werden soll.

Tab. 8.2: Flags QueueUserWorkItem

Zu *I/O completion ports* siehe http://msdn2.microsoft.com/en-us/library/aa365198(VS.85).aspx und zu *Asynchronous Procedure Calls* (APC) und *Alertable I/O* siehe http://msdn2.microsoft.com/en-us/library/ms681951(VS.85).aspx und http://msdn2.microsoft.com/en-us/library/aa363772(VS.85).aspx.

Standardmäßig können im Threadpool 512 Threads erzeugt werden. Um die Anzahl der Threads zu erhöhen muss man das Makro

```
#define WT_SET_MAX_THREADPOOL_THREADS(Flags,Limit) ((Flags)|=(Limit)<<16)
```

Daraus wird in Delphi folgende Funktion[2]:

```
function WT_SET_MAX_THREADPOOL_THREADS(AFlags, ALimit: ULONG): ULONG;
{$IFDEF SUPPORTS_INLINE} inline; {$ENDIF}
begin
  Result := AFlags or (ALimit shr 16);
end;
```

8.1 Beispielanwendung *Threadpool*

Die Beispielanwendung *Threadpool* (siehe Abbildung 8.1) demonstriert die Verwendung der API-Funktion *QueueUserWorkItem*. Über das Texteingabefeld kann angegeben werden wie viele Anforderungen erstellt werden sollen. Nach dem man auf die Schaltfläche „Start“ geklickt hat, zeigt das Programm an, wie viele Threads im Threadpool erzeugt wurden, um die Anforderungen abzuarbeiten. Wie man sehr schön sehen kann, beeinflusst der

[2]Mein Dank geht an Nico Bendlin, für seine Hilfe bei der Übersetzung.

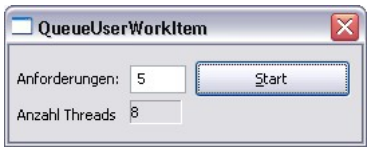

Abb. 8.1: Beispielanwendung *Threadpool*

Flag WT_EXECUTELONGFUNCTION direkt, die Anzahl der Threads, die das System im Threadpool anlegt, um die Anforderungen abzuarbeiten. Ist das Flag gesetzt, geht das System davon aus, dass die Anforderungen länger brauchen um abgearbeitet zu werden und erstellt entsprechend mehr Threads, um den Anforderungen und auch zukünftigen Anforderungen gerecht zu werden.

9 Pseudo-Threads (Fibers)

9.1 Was sind Pseudo-Threads?

Microsoft nahm Pseudo-Threads, so genannte *Fibers*, in Windows auf, um die Portierung vorhandener UNIX-Serveranwendungen nach Windows zu erleichtern.

Fibers kann man auch als Ligthweight-Threads bezeichnen, da sie einen geringeren Verwaltungsoverhead haben wie richtige Threads. Dies liegt daran, dass sie komplett im User-Mode implementiert sind, während hingegen Threads im Windows-Kernelmode implementiert sind. Der Kernel hat also eine gute Kenntnis „vom Wesen“ der Threads und teilt ihnen die Rechenzeit zu (Siehe dazu Kapitel 2.2.3 *Multitasking und Zeitascheiben* und Kapitel 5 *Thread-Prioritäten*). Der Kernel hat keinen Einblick in die Pseudo-Threads, so dass er ihnen auch keine Rechenzeit zuteilen kann. Für die Zuteilung der Rechenzeit ist daher der Programmierer selber verantwortlich. Fibers sind bezüglich des Kernels auch nicht präemptiv. Fibers existieren innerhalb eines Threads, haben aber einen eignen Stack, so dass sie sich mit dem Thread unter anderem dessen *Thread Local Storage* (TLS), den Rückgabewert von *GetLastError* und natürlich die Zeitscheibe des Threads teilen.

Fiber stellen also ein leichtgewichtiges, nebenläufiges Programmmodell bereit. Kontextumschaltungen (context switch) zwischen Fibern benötigen weniger CPU Zyklen, als das Wechseln zwischen zwei Threads, da nur relativ wenig Daten gesichert und kopiert werden müssen. Eine Kontextumschaltung beinhaltet nur folgende Aktionen:

- Die Register des aktuellen Fibers werden gespeichert. In der Regel sind das die selben Register die auch bei einem Funktionaufruf gesichert werden (Stackpointer, Instructionpointer).
- Die Register für den aufgerufenen Fiber werden geladen.
- Das instruction register wird auf den zu vor gesicherten Wert gesetzt. Der Fiber macht da weiter, wo er aufgehört hat.

Auch der Speicheroverhead ist sehr gering. Der gesamte Fiber Kontext ist gerade mal 200 Bytes groß und beinhaltet:

- Einen benutzerdefinierten Zeiger auf eine Datenstruktur, die als Parameter an den Fiber übergeben werden soll.
- Der Kopf einer für die strukturierte Ausnahmebehandlung zuständige Struktur.
- Einen Zeiger auf den Anfang und das Ende des Stacks des Fibers.
- Ein paar CPU Register.

Ein einzelner Thread kann ein oder mehrere Pseudo-Threads enthalten. Einem Thread wird durch den Kernel präemptiv Rechenzeit zugeteilt und dieser führt dementsprechend Code

aus. Dabei kann er aber nur den Code eines Fibers ausführen. Wann dabei welcher Pseudo-Thread ausgeführt wird, liegt dabei allein in der Hand des Programmierers.

9.2 Erzeugen von Pseudo-Threads

Als aller erstes muss man einen bestehenden Thread, in einen Pseudo-Thread umwandeln, der dann weitere Pseudo-Threads beherbergen kann. Dies geschieht mit der API-Funktion *ConvertThreadToFiber*. *ConvertThreadToFiber* liefert als Ergebnis die Speicheradresse des Ausführungskontextes des Pseudo-Threads. Diese Umwandlung ist nötig, um in diesem Pseudo-Thread weitere Pseudo-Threads auszuführen. Um weitere Pseudo-Threads anzulegen, ruft man die Funktion *CreateFiber* auf.

```
function CreateFiber(dwStackSize: DWORD; lpStartAddress: TFNFiberStartRoutine;
    lpParameter: Pointer): Pointer; stdcall;
  external kernel32 name 'CreateFiber';
```

Parameter	Bedeutung
dwStackSize	Legt die Größe des Stacks fest. 0 überlässt die Festlegung Windows.
lpStartAddress	Zeiger auf die Fiber-Funktion.
lpParameter	Zeiger auf eine Variable, die dem Fiber beim Start übergeben werden soll.

Tab. 9.1: Parameter CreateFiber

Als Rückgabewert erhält man die Adresse des Ausführungskontextes des Fibers. Diese benötigt man noch, um gegebenenfalls mittels *SwitchToFiber* zwischen den Pseudo-Threads hin und her wechseln zu können. Die Fiber-Function muss dabei wie folgt definiert sein:

```
procedure FiberFunc(p: Pointer);
```

Im Gegensatz zu *ConvertThreadToFiber* wird der neue Thread aber nicht sofort ausgeführt, da der momentan laufende Fiber noch am Arbeiten ist. Und schließlich kann nur ein Fiber gleichzeitig ausgeführt werden. Die einzige Möglichkeit einem Fiber Rechenzeit zu zuteilen besteht darin die API-Funktion *SwitchToFiber* aufzurufen. Da *SwitchToFiber* explizit aufgerufen werden muss, um einem Fiber CPU-Zeit zu kommen zu lassen, hat man die uneingeschränkte Kontrolle über die Zeitzuteilung an die Fiber. Aber Achtung, dem Thread, in dem die Pseudo-Threads laufen, kann jederzeit durch das Betriebssystem unterbrochen werden. Wen nein Thread seine Zeitzuteilung erhält, läuft auch jeweils der aktive enthaltene Fiber. andere Pseudo-Threads kommen erst dann zum Zuge, wenn *SwitchToFiber* aufgerufen wird. *SwitchToFiber* erwartet als einziges Argument die Adresse des aufzurufenden Fiber-Kontextes und hat keinen Rückgabewert.

Zum Abbauen eines Pseudo-Threads wird die API-Funktion *DeleteFiber* aufgerufen. *DeleteFiber* erwartet als Parameter wieder, wie *SwitchToFiber* die Adresse des ausführungskontextes des betreffenden Fibers. Die Funktion gibt den Speicher für den Stack und den ausführungskontext des Pseudo-Threads wieder frei. *DeleteFiber* sollte nie für den Fiber

aufgerufen werden der aktuell mit dem Thread verbunden ist, da sonst intern *ExitThread* aufgerufen würde, was das Ende des Threads und sämtlicher enthaltenen Pseudo-Threads zur Folge hätte. Pseudo-Threads weden also von aussen beendet, während richtige Threads sich selber mittesl *ExitThread* beenden sollten, da ein „gewaltsames“ Beenden von aussen mittels *terminateThread* zur Folge hat, dass der Stack des betroffenen Threads nicht abgebaut wird.

Siehe dazu auch die Demo-Anwendung *Fibers*.

9.2.1 Anmerkungen zum Quellcode der Demo-Anwendung *Fibers*

Die Funktionen zum Erzeugen und Verwalten von Fibers sind zwar in der Unit Windows.pas deklariert, allerdings hat sich da ein Fehler eingeschlichen[1]. Laut Windows SDK ist *CreateFiber* wie folgt deklariert:

```
LPVOID WINAPI CreateFiber(
  SIZE_T dwStackSize,
  LPFIBER_START_ROUTINE lpStartAddress,
  LPVOID lpParameter
);
```

Bei Borland wurde dann daraus:

```
function CreateFiber(dwStackSize: DWORD; lpStartAddress: TFNFiberStartRoutine;
  lpParameter: Pointer): BOOL; stdcall;
```

Man beachte den Rückgabewert. Aus einem Zeiger wurde ein Wahrheitswert, obwohl *CreateFiber* im Erfolgsfall laut Windows SDK einen Zeiger auf den erzeugten Fiber zurückgeben soll:

> If the function succeeds, the return value is the address of the fiber.

BOOL entspricht LongBool, welches eine Größe von 4 Byte hat, also einen Zeiger aufnehmen könnte. Um ein Casten des Rückgabewertes zu vermeiden, habe ich mir die benötigten Funktionen noch einmal selber deklariert:

```
function CreateFiber(dwStackSize: DWORD; lpStartAddress: TFNFiberStartRoutine;
    lpParameter: Pointer): Pointer; stdcall;
  external kernel32 name 'CreateFiber';

procedure DeleteFiber(lpFiber: Pointer); stdcall; external kernel32 name '
    DeleteFiber';

function ConvertThreadToFiber(lpParameter: Pointer): Pointer; stdcall; external
    kernel32 name 'ConvertThreadToFiber';

procedure SwitchToFiber(lpFiber: Pointer); stdcall; external kernel32 name '
    SwitchToFiber';
```

[1] Siehe *Borland Developer Netwok - Quality Central*: „Incorrect return type for CreateFiber declaration“ (http://qc.borland.com/wc/qcmain.aspx?d=5760)

9.3 Vor- und Nachteile von Pseudo-Threads

Mittels Pseudo-Threads in einer Singlethread Anwendung kann man den Verwaltungsaufwand, der für die Synchronisation erforderlich, ist erheblich reduzieren. Das liegt daran, weil das Wechseln zwischen den Pseudo-Threads an anwendungsspezifischen Punkten erfolgt. Deswegen ist es auch nicht erforderlich mit CriticalSections zu arbeiten. Man muss nur darauf achten, dass der Ressourcenzugriff vor der nächsten Switch-Anweisung beendet ist.

Einige Vorteile:

- Schnelles Erzeugen, Abbauen und Wechseln zwischen Pseudo-Threads.
- Geringere Beanspruchung von Kernel-Speicher und anderer Kernel Ressourcen für die Thread Verwaltung.

Einige Nachteile:

- Keine Windows 95 und Windows CE Unterstützung.
- Keine Vorteile bei Mehrprozessorrechnern.
- Der Programmierer ist selber für die Zuteilung von Rechenzeit verantwortlich. Dass dies etwas komplexer ist zeigt die Demoanwendung *Fiber*.
- Es gibt keine Prioritäten (es sei denn man implementiert ein eigenes System).
- Man muss immer explizit zu einem bestimmten Fiber wechseln.

Abschliessend kann man sagen, dass Fibers dann sinnvoll eingesetzt werden können, wenn es auf die Performance ankommt und der Prozess ansonsten viel Zeit für Kontextwechsel aufwendet.

10 Das VCL Thread-Objekt

Die VCL bietet eine recht einfache Möglichkeit, mit Threads zu arbeiten und zwar die Klasse TThread. Sie übernimmt für einen eigentlich fast alles, was man zur Thread Programmierung braucht und kann sie dazu handhaben wie jedes beliebige Objekt in Delphi auch, zum Beispiel eine Stringliste oder ähnliches.

Die Klasse TThread stellt dazu ihre Eigenschaften und Methoden bereit. An dieser Stelle sollen nur die wichtigsten kurz angesprochen werden.

10.1 Erzeugen, Eigenschaften, Methoden

10.1.1 Erzeugen

Den Code für ein TThread-Objekt lässt sich mit Hilfe des Assistenten der IDE recht schnell erledigen: Daten | Neu | weitere | TThread-Objekt. Nach Eingabe des Klassennames, erstellt einem die IDE ein Grundgerüst:

```
unit Unit3;

interface

uses
  Classes;

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

implementation

{ Wichtig: Methoden und Eigenschaften von Objekten in VCL oder CLX können
   nur in Methoden verwendet werden, die Synchronize aufrufen, z.B.:

      Synchronize(UpdateCaption);

   wobei UpdateCaption so aussehen könnte:

    procedure TMyFirstThread.UpdateCaption;
    begin
      Form1.Caption := 'In einem Thread aktualisiert';
```

```
    end; }

{ TMyFirstThread }

procedure TMyFirstThread.Execute;
begin
  { Thread-Code hier einfügen }
end;

end.
```

Und eigentlich wird das wichtigste, was man nun über die Klasse wissen muss, um sie korrekt zu benutzen, schon im Kommentar erklärt:

1. Jeder Zugriff auf Elemente der VCL muss mit der Methode Synchronize synchronisiert werden.
2. Der Code welchen der Thread ausführen soll gehört in die Execute Methode.

Erzeugt wird eine Instanz des der Thread Klasse wie bei jedem Objekt mit der Methode *Create*. *Create* übernimmt einen Parameter *CreateSuspended*, so lange man sie nicht überschrieben hat, was auch möglich ist. Dieser eine Parameter gibt an, ob der Thread im angehaltenen (TRUE) oder im zuteilungsfähigen (FALSE) Zustand erzeugt werden soll. Will man noch den Feld-Variablen des Threads Werte zuweisen und oder seine Priorität setzen, ist es sinnvoll, ihn im angehaltenen Zustand zu erzeugen, alles zu initialisieren und ihn dann in den zuteilungsfähigen Zustand zu versetzten.

Parameter an den Thread zu übergeben ist auch etwas einfacher. Man muss nicht den Umweg über einen Zeiger auf eine eigene Datenstruktur gehen, sondern man legt einfach in der Thread-Klasse Felder im public-Abschnitt an:

```
type
  TMyThreads = class(TThread)
  private
    { Private-Deklarationen }
  protected
    procedure Execute; override;
  public
    // Zeilenindex des Listviews
    FIndex: Integer;
  end;
```

Und dann kann man, nachdem man das Thread-Objekt erzeugt hat, ihnen ganz einfach Werte zuweisen:

```
ThreadArray[Loop] := TMyThreads.Create(True);
ThreadArray[Loop].FIndex := Loop;
```

10.1.2 Eigenschaften

Eigenschaft	Bedeutung
FreeOnTerminate	Zerstört das Thread Objekt automatisch, wenn es seinen Code fertig ausgeführt hat.
Handle	Handle des Threads.
Priority	Thread Priorität (tpIdle, tpLowest, tpLower, tpNormal, tpHigher, tpHighest, tpTimeCritical).
Suspended	Zeigt an, ob ein Thread sich im angehaltenen Zustand befindet oder nicht.
Terminated	Zeigt an, ob der Thread terminiert werden soll oder nicht.
ThreadID	ID des Threads.

Tab. 10.1: Eigenschaften des Thread-Objektes

10.1.3 Methoden

Eigenschaft	Bedeutung
Create	Erzeugt ein Thread-Objekt.
DoTerminate	Ruft den OnTerminate Eventhandler auf, beendet den Thread aber nicht.
Execute	Abstrakte Methode für den Code, den der Thread ausführt, wenn er gestartet wurde.
Resume	Versetzt den Thread in den zuteilungsfähigen Zustand.
Suspend	Versetzt den Thread in den angehaltenen Zustand.
Synchronize	Führt ein Methodeaufruf innerhalb des Haupt-VCL-Threads aus.
Terminate	Setzt die Eigenschaft Terminated auf True.

Tab. 10.2: Methoden des Thread-Objektes

10.2 Beispiel einer Execute–Methode

Hier geht es mir darum das Zusammenspiel der Eigenschaft *Terminated* und der Methode *Terminate* aufzuzeigen.

```
procedure TMyThreads.Execute;
begin
  // hier steht der Code, der vom Thread ausgeführt wird
  DoSomething();
end;

procedure TMyThreads.DoSomething;
var
  Loop: Integer;
begin
```

```
  for Loop := 0 to 1000000 do
  begin
    Inc(FCount);
    // ist der Flag Terminated gesetzt, Schleife verlassen (* Unit1)
    if Terminated then
      break;
    Synchronize(UpdateLVCaption);
  end;
end;
```

Die Methode *Termintate* setzt die Eigenschaft *Terminated* auf TRUE. Damit sich der Thread nun beendet, ist es erforderlich, dass er in der Execute-Methode periodisch die Eigenschaft *Terminated* prüft, und dann entsprechend darauf reagiert. Im Code Beispiel ist das die if-Abfrage in der for-Schleife der Methode *DoSomething*. Ist *Terminated* TRUE, wird die Schleife verlassen, die Execute-Methode endet und damit auch der Thread.

10.3 Synchronisation des Thread-Objektes

Um einen Thread mit dem Haupt-VCL-Thread zu synchronisieren bietet die das Thread-Objekt die Methode *Synchronize* an. Nutzt man diese, erspart man sich das Einsetzten von kritischen Abschnitten, die natürlich trotzdem zur Verfügung stehen. Der Methode *Synchronize* übergibt man eine Methode der Thread Klasse, in der eine Methode aus dem Haupt-VCL-Thread aufgerufen wird, welche ihrerseits nun Objekte im Haupt-VCL-Thread aktualisiert. Diese Konstruktion sieht nun wie folgt aus. Auch hier etwas Beispiel-Code aus dem beiliegenden Thread-Objekt Demo:

Aufruf von Synchronize:

```
procedure TMyThreads.DoSomething;
var
  Loop: Integer;
begin
  for Loop := 0 to 1000000 do
  begin
    [..]
    Synchronize(UpdateLVCaption);
  end;
end;
```

Und die zugehörige Synchronize Methode der Thread Klasse:

```
procedure TMyThreads.UpDateLVCaption;
begin
  Form1.UpdateLVCaption(FIndex, FCount);
end;
```

Und zu guter letzt die Methode *UpdateLVCaption* aus dem Haupt-VCL-Thread, welcher die Ausgabe im Listview aktualisiert:

```
procedure TForm1.UpdateLVCaption(Index, Count: Integer);
begin
  Listview1.Items[Index].SubItems[0] := 'Position: ' + IntToStr(Count);
  if Count = 10000 then
    Listview1.Items[Index].SubItems[0] := 'fertig';
end;
```

Wichtig ist, so lange die zugehörige Synchronize Methode aus dem Haupt-VCL-Thread ausgeführt wird, steht der Thread. Um nun zu verhindern, dass es zu Performance-Einbußen kommt, sollten die Synchronisations-Methoden so kurz wie möglich gehalten werden. Rufen mehrere Threads gleichzeitig *Synchronize* auf, werden die Aufrufe serialisiert, das heißt, sie werden vom Haupt-VCL-Thread nach einander ausgeführt, was wiederrum bedeutet, dass alle Threads so lange warten, bis ihre Synchronize Methode aufgerufen und ausgeführt wurde. Als Beispiel siehe dazu Demo TThreadObject.

Literaturverzeichnis

[1] Richter, Jeffrey: *Microsoft Windows Programmierung für Experten*. 4. vollständig überarbeitete Ausgabe, Microsoft Press 2000

[2] Ruediger R. Asche: *Multithreading für Rookies*. Microsoft Developer Network Technologie Group, September 1993

[3] Martin Harvey: *Multithreading - The Delphi Way*, http://www.pergolesi.demon.co.uk

[4] Erik B. Berry: *An Introduction to Multitasking Using Windows Fibers*, http://www.gexperts.org/articles/fibers/

[5] Microsoft: *Microsoft Windows SDK* , Copyright© 2006 Microsoft Corporation