

31 Mehreres zur gleichen Zeit erledigen – Threading in .NET

-
- 936 Threads durch ein Thread-Objekt initiieren
 - 939 Synchronisieren von Threads
 - 953 Verwenden von Steuerelementen in Threads
 - 954 Managen von Threads
 - 959 Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen
 - 970 Verwenden des Thread-Pools
 - 975 Thread-sichere Formulare in Klassen kapseln
 - 978 Threads durch den Background-Worker initiieren
 - 981 Threads durch asynchrone Aufrufe von Delegationen initiieren
-

Dass Gleichzeitigkeit eigentlich eine Illusion ist, hat Einstein mit seiner Relativitätstheorie bewiesen.¹ Wenn auch aus anderen Gründen, besteht dennoch der Eindruck, dass ein normaler Computer Dinge wirklich gleichzeitig erledigen könnte. Auch wenn er im Hintergrund Robbie Williams spielt, eine seiner Festplatten defragmentiert und mich gleichzeitig diese Zeilen mit Word schreiben lässt, so zerlegt er diese drei Sachen in klitzekleine Aufgaben und arbeitet sie im Grunde genommen nacheinander ab. Aus der Geschwindigkeit, mit der er diese kleinen Dinge hintereinander macht, entsteht dann der Eindruck, er mache sie wirklich gleichzeitig.

Ausnahmen davon bilden Multiprozessor- oder Multicore-Systeme, die bestimmte Aufgaben tatsächlich gleichzeitig erledigen können. Anmerkung am Rande: Die so genannten *Hyperthreading*-Prozessoren von Intel nehmen dabei eine Art Zwitterstellung ein – sie nutzen Pausen, die der Prozessor von Zeit zur Zeit einlegen muss, wenn er beispielsweise auf Daten aus dem Hauptspeicher wartet, um schon mal Teile anderer Threads zu verarbeiten, sodass dieser Prozessor auf unterster Ebene betrachtet Gleichzeitigkeit ziemlich perfekt vortäuscht. In der Tat führt das zu einer Leistungssteigerung von durchschnittlich 10 % bis zu ca. maximal und unter günstigsten Umständen 20 %, doch im Grunde genommen arbeiten auch Hyperthreading-Prozessoren die zu erledigenden Threads auch nur nacheinander ab. Echte Gleichzeitigkeit ist allein Multiprozessor- bzw. Multicore-Systemen vorbehalten.

¹ Siehe auch die Folge von Alpha Centauri (vom 10.6.2001), zu erreichen über den IntelliLink G3101.

Doch gerade die letzten Systemtypen machen genau in diesem Moment, in dem die Zeilen entstehen, Schluss mit dem Taktfrequenzrennen der verschiedenen Prozessorhersteller. Da die Grenze des physikalisch Möglichen quasi erreicht ist, können Prozessoren nicht mehr wesentlich schneller werden: man kann Ihnen nur beibringen, mehrere Dinge wirklich gleichzeitig zu machen, und genau dahin geht der Trend. Dummerweise bedeutet es nicht automatisch, dass ein Prozessor, der im Grunde genommen zwei oder vier Prozessoren in sich vereint, auch automatisch das zwei- bzw. vierfache an Leistung bringt. Nur, wenn eine Software auch in der Lage ist, die beiden Prozessorkerne auch wirklich zu nutzen, indem sie eine oder mehrere Aufgaben (so genannte *Tasks*) in verschiedene Threads verlagert, die dann wiederum unabhängig auf mehreren Prozessorkernen laufen können, können Sie als Anwender (und als Entwickler zeitkritischer Systeme) auch wirklich von dieser neuen Technologie profitieren.

Sie sehen also, dass dieses Kapitel eines der wirklich wichtigen Themen der kommenden Zeit behandelt.

Unter dem Namen »Multitasking« hat wohl jeder, der sich nur ein wenig mit Computern beschäftigt, diese Fähigkeit schon einmal kennen gelernt. Multitasking ist die Kombination der englischen Wörter »multi« – für »viel« – und »task« – für »Aufgabe« – und bedeutet im Deutschen das, was Frauen beneidenswerter- und normalerweise eher können als Männer: nämlich mehrere Dinge zur gleichen Zeit erledigen.

Ein weiterer, ähnlicher Begriff, stammt ebenfalls aus dem Englischen, aber er ist nicht ganz so bekannt wie »Multitasking«. Gemeint ist »Multithreading«², wieder abgeleitet von »multi« – für »viel« – nur im zweiten Teil des Wortes diesmal von »thread« – für »Faden«. Man könnte eine Multithreading-fähige Anwendung also als »mehrfädiges« Programm bezeichnen, wollte man den Ausdruck unbedingt übersetzen.

Und was bedeutet dieser Ausdruck jetzt genau? Dazu folgender Hintergrund: wenn Sie eine Windows-Applikation starten, dann besteht sie aus mindestens einem so genannten Thread. In diesem Zusammenhang ist »Thread« eine abstrakte Bezeichnung für einen bestimmten Programmverlauf. Ein Thread startet, löst eine bestimmte Aufgabe und wird beendet, wenn diese Aufgabe erledigt ist. Wenn diese Aufgabe sehr rechenintensiv ist, dann bedeutet das in der Regel für das Programm, dass es nicht weiter bedienbar ist. Der aktuelle Thread beansprucht die gesamte Rechenleistung eines Prozessorkerns (und wenn der Computer nur über einen Prozessorkern verfügt, seine komplette Rechenleistung), sodass für die Behandlung der Bedienungselemente nichts mehr übrig bleibt, wie das folgende kleine Beispielprogramm eindrucksvoll zeigt:

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap31\Single-Threading\`.

Dieses Programm macht nichts anderes, als eine Zählvariable bis auf einen bestimmten Wert hinaufzuzählen und den aktuellen Wert im Label-Steuer-element anzuzeigen.

² Ausgesprochen etwa »Maltiþrædding«, und wenn Sie es ganz perfekt machen wollen, lispeln Sie das scharfe S.

```
Public Class Form1
```

```
    Private Sub btnZählenStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
        Handles btnZählenStarten.Click  
        For z As Integer = 0 To 100000  
            lblAusgabe.Text = z.ToString  
            lblAusgabe.Refresh()  
        Next  
    End Sub  
End Class
```

Wenn Sie dieses Programm gestartet haben, klicken Sie auf die Schaltfläche *Zählen starten*.

Ein Blick in den Task-Manager offenbart, dass es fast alles an Prozessorleistung verschlingt. Da es darüber hinaus keine Anstalten macht, das Formular die Warteschleife verarbeiten zu lassen, lässt es sich, während es läuft, auch nicht bedienen – Sie können das Programmfenster nicht verschieben oder schließen; obwohl die Zählung läuft, scheint es zu hängen.

Sie sehen: In dem Moment, in dem Sie Ihr Programm eine umfangreiche Verarbeitung von Daten durchführen lassen müssen, sollten Sie auf eine andere Technik ausweichen, damit andere Funktionen des Programms nach wie vor zur Verfügung stehen können. Und hier kommt die Thread-Programmierung ins Spiel. Threads sind Programmteile, die unabhängig voneinander und quasi gleichzeitig operieren können. So könnte beispielsweise die eigentliche Zählschleife des Programms in einem eigenen Thread laufen. Sie würde in diesem Fall parallel zum eigentlichen Programm ausgeführt werden. Das Programm könnte sich dann nicht nur um seine Nachrichtenwarteschlange kümmern und das Programm so bedienbar halten, sondern sich zusätzlich um die Ausführung weiterer Aufgaben kümmern.

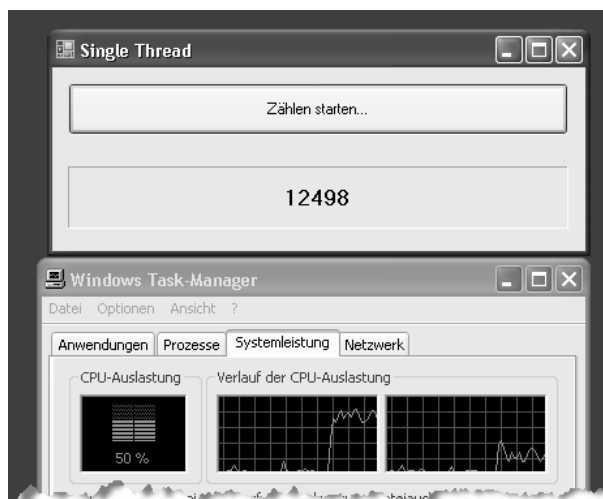


Abbildung 31.1: Sobald Sie das Programm starten, verschlingt es die komplette Prozessorleistung eines Prozessorkerns. Darüber hinaus lässt die Schleife keine Nachrichtenauswertungen zu, sodass sich das Programm während des Zählens nicht anderweitig bedienen lässt.

Beispiele, wann Threads »angesagt« sind, gibt es viele:

- Ihr Programm muss eine umfangreiche Auswertung von Daten zusammenstellen und drucken. Diese Aufgabe könnte in einem eigenen Thread im Hintergrund passieren, ohne dass es den weiteren Arbeitsablauf des Programms stören würde.

- Ihr Programm muss umfangreiche Datensicherungen durchführen. Ein Thread könnte Momentaufnahmen dieser Dateien erstellen und sie anschließend sozusagen im Hintergrund auf eine andere Ressource kopieren.
- Ihr Programm muss Zustände bestimmter Hardwarekomponenten aufzeichnen – beispielsweise Produktionsdaten von Maschinen abrufen. Auch diese Funktion könnte im Hintergrund ablaufen; Auswertungsfunktionen könnten dennoch zu jeder Zeit parallel laufen und vom Anwender verwendet werden, wenn die Produktionsdatenerfassung als Thread im Hintergrund läuft.
- Bei komplizierten Berechnungen oder Auswertungen könnten zwei oder mehrere Threads diese Aufgabe übernehmen. In diesem Fall würde auf Hyperthreading- oder Multiprozessorsystemen eine Auslastung mehrerer Prozessoren durch *jeweils* einen eigenen Thread die Verarbeitungsgeschwindigkeit der gesamten Aufgabe deutlich erhöhen.

Anwendungen gibt es also viele, um Threads einzusetzen. Allerdings gibt es bei Threads auch ein paar Dinge, die beachtet werden müssen, denn: Sie dürfen sich nicht gegenseitig ins Gehege kommen. Doch dazu später mehr.

Betrachten wir zunächst die Grundlagen, also wie wir das Framework überhaupt dazu bewegen können, dass bestimmte Teile einer Anwendung als eigener Thread ausgeführt werden können.

HINWEIS: Sollten Sie einfach nur die Anforderung haben, eine bestimmte Aufgabe in einem anderen Thread erledigen zu lassen, und möchten Sie dazu nicht tiefer in die Materie einsteigen, empfehle ich Ihnen den Einsatz der `BackgroundWorker`-Komponente, die Sie im ► Abschnitt »Threads durch den Background-Worker initiieren« ab Seite 978 beschrieben finden.

Threads durch ein Thread-Objekt initiieren

Das folgende Beispiel zeigt am einfachsten Beispiel, wie Sie eine Prozedur Ihrer Klasse mithilfe der `Thread`-Klasse als Thread parallel zum aktuellen Programm ablaufen lassen können. Alle weiteren Techniken des Threading wird dieses Kapitel übrigens anhand dieser Klasse besprechen.

BEGLEITDATEIEN: Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap31\SimpleThread01\`.

WICHTIG: In den Beispielen dieses Kapitels finden Sie des Öfteren eine Klasse, die die Ausgabe von Zeilen in einem speziellen Dialog ermöglicht. Diese Klasse nennt sich `ADThreadSafeInfoBox`, und sie stellt die Methoden `TSWrite` und `TSWriteLine` zur Verfügung. Sie ist für die Verwendung in Windows Forms-Anwendungen gedacht, und um sie zu verwenden, müssen Sie sie nicht instanzieren, sondern können ihre Funktionen wie die der `Console`-Klasse direkt verwenden, da sie statisch sind. Ihre Verwendung ist notwendig, da die Aktualisierung von Steuerelementen in Formularen aus Thread-Prozeduren eine besondere Vorgehensweise erforderlich macht. Auf dieses Thema werde ich jedoch am Ende dieses Kapitels genauer eingehen. Für den Moment arbeiten Sie mit der Klasse einfach so, als wäre sie fest im Framework vorhanden.

Wenn Sie dieses Programm starten, sehen Sie einen simplen Dialog, der lediglich aus zwei Schaltflächen besteht. Sobald Sie die Schaltfläche *Thread starten* anklicken, öffnet sich ein weiteres Fenster, in dem ein Wert von 0 bis 50 hoch gezählt wird. Soweit ist das noch nichts Besonderes. Allerdings können Sie die Schaltfläche ein weiteres Mal anklicken, um einen weiteren Thread zu starten. Auch der zweite Thread führt die Zählung durch, und das Ausgabefenster zeigt dabei die Ergebnisse beider Zahlenfolgen an – etwa wie in Abbildung 31.2 zu sehen:

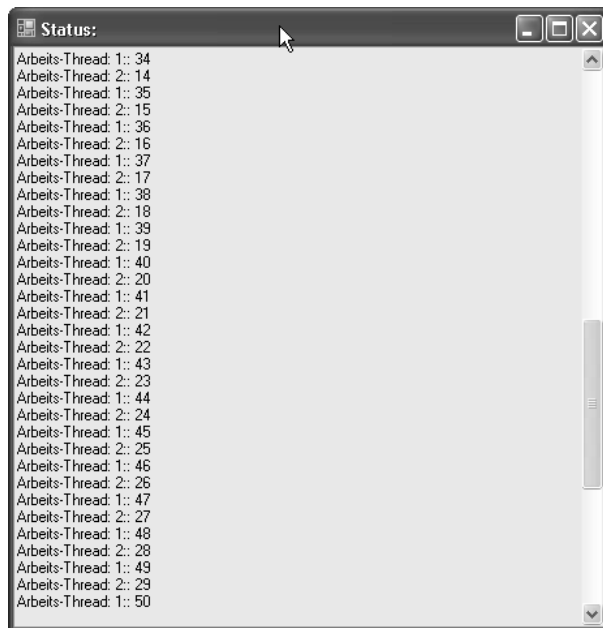


Abbildung 31.2: Zwei Threads laufen parallel und teilen sich das Ausgabefenster, um Ergebnisse ihres Schaffens anzuzeigen

Soweit, so gut. Nun lassen Sie uns als nächstes den Code betrachten, der dieses Ergebnis im Ausgabefenster zustande bringt:

```
Imports System.Threading
```

```
Public Class frmMain
```

```
    'Member-Variable, damit die Threads durchnummeriert werden können.
```

```
    'Dient nur zur späteren Unterscheidung des laufenden Threads, wenn
```

```
    'er Ergebnisse im Ausgabefenster darstellt.
```

```
    Private myArbeitsThreadNr As Integer = 1
```

```
    'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
```

```
    'Hochzählen und die Werteausgabe übernimmt
```

```
    Private Sub UmfangreicheBerechnung()
```

```
        For c As Integer = 0 To 50
```

```
            'Dient zur Ausgabe des Wertes. TSWriteLine ist eine statische
```

```
            'Prozedur, die für die Darstellung des Fensters selbst sorgt,
```

```
            'sobald sie das erste Mal verwendet wird.
```

```
            ADThreadSafeInfoBox.TSWriteLine(Thread.CurrentThread.Name + ":: " + c.ToString)
```

```

        'Aktuellen Thread um 100ms verzögern, damit die ganze
        'Geschichte nicht zu schnell vorbei ist.
        Thread.Sleep(100)
    Next
End Sub

Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnThreadStarten.Click
    'Dieses Objekt kapselt den eigentlichen Thread
    Dim locThread As Thread
    'Dieses Objekt benötigen Sie, um die Prozedur zu bestimmen,
    'die den Thread ausführt.
    Dim locThreadStart As ThreadStart

    'Threadausführende Prozedur bestimmen
    locThreadStart = New ThreadStart(AddressOf UmfangreicheBerechnung)
    'ThreadStart-Objekt dem Thread-Objekt übergeben
    locThread = New Thread(locThreadStart)
    'Thread-Namen bestimmen
    locThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString
    'Thread starten
    locThread.Start()
    'Zähler, damit die Threads durch ihre Namen unterschieden werden können
    myArbeitsThreadNr += 1

End Sub

Private Sub btnBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
    btnBeenden.Click
    'Einfach so geht's normalerweise nicht
    Me.Close()
End Sub

End Class

```

Starten von Threads

Wie aus dem Listing des vorherigen Beispielprogramms ersichtlich, benötigen Sie zwei Objekte, um einen Thread tatsächlich zum Laufen zu bringen: das Thread- und das ThreadStart-Objekt. Das ThreadStart-Objekt dient lediglich dazu, die Adresse der Prozedur aufzunehmen, die als Thread ausgeführt werden soll. Sie können es als Delegaten mit besonderen Eigenschaften betrachten, der darauf ausgerichtet ist, mit dem eigentlichen Thread-Objekt zusammenzuarbeiten. Nachdem Sie das ThreadStart-Objekt instanziiert und ihm dabei die Thread-Prozedur mit AddressOf zugespült haben, übergeben Sie die generierte Instanz dem Thread-Konstruktor. Haben Sie auch dieses Objekt erzeugt, starten Sie den Thread mit der Start-Methode.

Grundsätzliches über Threads

Jedes Thread-Objekt, das Sie auf die beschriebene Weise erzeugt haben, speichert spezifische Informationen über den eigentlichen Thread. Das ist notwendig, da Windows auf Basis des so genannten *Preemptive Multitasking*³ arbeitet, bei dem Prozessorzeit den verschiedenen Threads durch das Betriebssystem zugewiesen wird.⁴ Wenn das Betriebssystem bestimmt, dass es Zeit für die Ausführung eines bestimmten Threads wird, müssen beispielsweise der komplette Zustand der CPU-Register für den laufenden Thread gesichert und der ursprüngliche Zustand der Register für den als nächstes auszuführenden Thread wiederhergestellt werden. Diese Daten werden unter anderem in einem bestimmten Speicherbereich gesichert, den das Thread-Objekt kapselt. Sie werden für gewöhnlich auch als *Thread Context* bezeichnet.

Thread-Prozeduren sind im Grunde genommen nichts Besonderes. Eine Thread-Prozedur ist grundsätzlich – wie jede andere Prozedur auch – ein Teil einer Klasse oder eines Moduls (das ja im Grunde genommen auch nicht weiter als eine Klasse mit nur statischen Funktionen ist). Die lokalen Variablen, die die Thread-Prozedur verwendet, sind für den jeweils ausgeführten Thread unterschiedlich. Anders ausgedrückt, können lokale Variablen eines Threads also nicht denen eines anderen ins Gehege kommen. Anders ist das beim Zugriff auf Klassen-Member: Für jeden Thread gilt dieselbe Instanz einer Member-Variablen, und dabei können sich besondere Probleme ergeben:

Stellen Sie sich vor, ein Thread greift auf eine Member-Variable der Klasse zu, um sie beispielsweise neu zu berechnen und anschließend auszugeben. Genau in dem Moment, in dem der erste Thread sie berechnet hat, aber noch bevor er dazu gekommen ist, das berechnete Ergebnis tatsächlich auf dem Bildschirm auszugeben, hat ein zweiter Thread die Berechnung mit dem Member abgeschlossen. In der Member-Variablen steht nun ein aus Sicht des ersten Threads völlig falsches Ergebnis, und das ausgegebene Ergebnis des ersten Threads ist ebenso falsch.

Synchronisieren von Threads

Damit Zugriffs- bzw. Synchronisationsprobleme verhindert werden können, gibt es eine ganze Reihe von Techniken, die in diesem Abschnitt besprochen werden sollen.

Die einfachste Methode erlaubt das automatische Synchronisieren eines Codeblocks auf Grund einer verwendeten Objektvariablen. Der folgende Abschnitt erläutert das Problem und dessen Lösung anhand eines konkreten Beispiels.

Synchronisieren der Codeausführung mit SyncLock

Das nächste Beispielprogramm lehnt sich an das des vorherigen Beispiels an – es ist nur ein wenig »eloquenter«. Lediglich um zu zeigen, inwieweit nicht synchronisierte Vorgänge beim Threading richtig daneben gehen können, modifiziert es die Ausgaberoutine der Thread-Routine auf folgende Weise:

³ Etwa »bevorrechtigt«, »präventiv«.

⁴ Im Gegensatz dazu gibt es das so genannte *Cooperative Multitasking*, bei dem ein Thread selber bestimmt, wie viel Prozessorzeit er benötigt. Dadurch kann ein Thread, der in einer nicht enden wollenden Operation fest hängt, die Stabilität des gesamten Systems gefährden.

BEGLEITDATEIEN: Sie finden dieses Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap31\SimpleThread02 (SyncLock)`.

```
'Member-Variablen, mit der demonstrativ Mist gebaut wird...
Private myThreadString As String

'Dies ist der eigentliche Arbeits-Thread (auch "Worker Thread" genannt),
'der das Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim strTemp As String

    For c As Integer = 0 To 50
        strTemp = Thread.CurrentThread.Name + ":: " + c.ToString

        'Nehmen Sie die Auskommentierung von SyncLock zurück,
        'um den "Fehler" des Programms zu beheben.
        'SyncLock Me
        myThreadString = ""

        For z As Integer = 0 To strTemp.Length - 1
            myThreadString += strTemp.Substring(z, 1)
            Thread.Sleep(5)
        Next

        ADThreadSafeInfoBox.TSWriteLine(myThreadString)
        'End SyncLock

    Next
End Sub
```

Hier passiert folgendes: `myThreadString` ist eine Member-Variable der Klasse, und sie wird für jedes Zeichen, das im Ausgabefenster für einen Zählungseintrag erscheinen soll, Zeichen für Zeichen zusammengesetzt. Die `TSWriteLine`-Methode gibt diesen String anschließend aus, wenn das Zusammenbasteln des Strings für einen Eintrag erledigt ist.

Wenn Sie das Programm starten, und die Schaltfläche nur ein einziges Mal anklicken, sodass auch nur ein einziger Thread läuft, bleibt alles beim Alten. Doch wehe, sie starten, schon während der erste Threads läuft, auch nur einen weiteren, dann nämlich ist Chaos angesagt, wie auch in Abbildung 31.3 zu sehen.

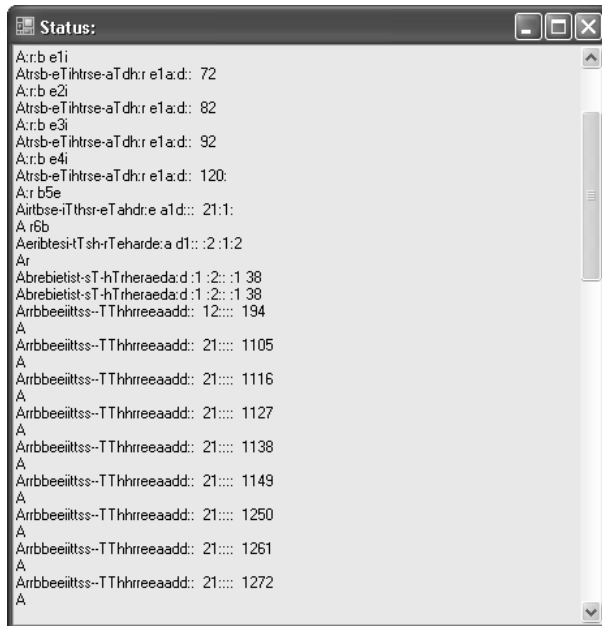


Abbildung 31.3: Zwei Threads im Kampf um eine Member-Variable – und das daraus resultierende und nicht wirklich zufrieden stellende Ergebnis

Das Problem ist, nicht genau voraussehen zu können, wann eine Thread-Prozedur zum Zug kommt – denn beim *Preemptive Multitasking* bestimmt diesen Zeitpunkt das Betriebssystem. Im hier vorliegenden Fall »meint« das Betriebssystem, dass ein anderer Thread am besten zum Zuge kommen kann, wenn der eine Thread gerade zu warten beginnt (was beide durch die *Sleep*-Methode in regelmäßigen Abständen machen).

Dieses Problem können Sie durch das Blocken von Codeabschnitten in Abhängigkeit von verwendeten Objekten aus der Welt schaffen. Die *SyncLock*-Anweisung ist hier der Schlüssel zur Lösung. Wenn Sie den Code folgendermaßen umbauen, kann sich das Ergebnis wieder sehen lassen:

```
'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Wertausgabe übernimmt.
Private Sub UmfangreicheBerechnung()
    Dim strTemp As String

    For c As Integer = 0 To 50
        strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
        SyncLock Me
            myThreadString = ""
            For z As Integer = 0 To strTemp.Length - 1
                myThreadString += strTemp.Substring(z, 1)
                Thread.Sleep(5)
            Next
            'ADThreadSafeInfoBox.TSWriteLine(myThreadString)
            Console.WriteLine(myThreadString)
        End SyncLock

    Next
End Sub
```

SyncLock verwendet ein Objekt, um den nachfolgenden Code für alle anderen Threads zu schützen, die einen Verweis auf dasselbe Objekt halten. Dieses Objekt sollte daher für die Dauer des Zugriffs nicht verändert werden, da der Schutz sonst nicht oder nicht mehr zuverlässig funktioniert.

WICHTIG: Im Gegensatz zu vielen vorherrschenden Meinungen schützt SyncLock nicht das Objekt selbst vor Veränderungen, sondern den Code gegen den gleichzeitigen Zugriff von einem anderen Thread. Natürlich kann jeder andere Thread das Objekt verändern und so die ganze Bemühung zur Synchronisation des Programms an dieser Stelle zunichte machen. Außerdem sollte am Rande erwähnt werden: Da SyncLock intern einen Try/Catch-Block verwendet und damit auch im Falle einer Ausnahme der Schutz des Blocks wieder aufgehoben werden kann, dürfen Sie nicht mit der Goto-Anweisung in einen SyncLock-Block springen.

Was passiert also im Detail bei der Ausführung dieser Routine? Nun, sobald der erste Thread den Block erreicht, sperrt er den Zugriff auf den Code für jeden weiteren Thread durch Zuhilfenahme des Objektes. Für dieses Objekt müssen folgende Bedingungen gelten:

- Es muss sich um ein Objekt handeln, das auf eine gültige Adresse im Managed Heap verweist – darf also nicht Nothing sein.
- Es muss eine Member-Variable sein, also jedem Thread den Zugriff darauf ermöglichen.
- Es muss zwingend ein Referenztyp sein (logisch, denn Wertetypen liegen nicht auf dem Managed Heap).
- Es darf durch die geschützte Routine nicht verändert werden.

Im Beispiel wird das einfach durch die Verwendung der eigenen Instanz Me erreicht. Me erfüllt diese Bedingungen stets. Bei statischen Routinen, bei denen Me natürlich nicht anwendbar ist, können Sie das Typ-Objekt der verwendeten Klasse verwenden, um es als Hilfe zum Schutz einer Routine zu verwenden, etwa:

```
.
.  
SyncLock (GetType(Klassenname))  
  
    'Hier steht der Code, der zu schützenden Routine  
  
End SyncLock  
.
```

Wenn nun ein zweiter Thread den geschützten Programmteil erreicht, dann wird er so lange in die Knie gezwungen, bis der Thread, der den Programmteil bereits durchläuft, End SyncLock erreicht hat.

In unserem Beispiel kann also der Member-String ohne Sorge zu Ende aufgebaut und anschließend verarbeitet werden. Dass ein anderer Thread den Member-String an dieser Stelle in irgendeiner Form verändert, ist ausgeschlossen, weil er auf alle Fälle zu warten hat.

HINWEIS: Einen Programmteil, der besonderen Synchronisationsschutz benötigt bzw. erfährt, nennt man *kritischer Abschnitt* oder neudeutsch: *Critical Section*.

Mehr Flexibilität in kritischen Abschnitten mit der Monitor-Klasse

Die SyncLock-Funktion hat einen großen Nachteil: Sie ist unerbittlich. Wenn ein anderer Thread bereits einen kritischen Abschnitt durchläuft, dann warten alle anderen Threads am Anfang des kritischen Abschnitts – ob sie wollen oder nicht. Sie können dagegen nichts tun. Die Monitor-Klasse bietet an dieser Stelle die größere Flexibilität, da sie ein paar zusätzliche Methoden im Angebot hat, mit der ein Thread auch nachschauen kann, ob er warten müsste, wenn er einen kritischen Bereich beträte. Ein weiteres Beispiel soll das verdeutlichen:

BEGLEITDATEIEN: Betrachten Sie den Code des folgenden Beispielprogramms, das Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap31\SimpleThread03 (Monitor)\`.

```
'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim strTemp As String

    For c As Integer = 0 To 50
        strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
        'Nehmen Sie die Auskommentierung von SyncLock zurück,
        'um den "Fehler" des Programms zu beheben.
        If Not Monitor.TryEnter(myLock, 1) Then
            ADThreadSafeInfoBox.TSWriteLine(Thread.CurrentThread.Name + " meldet: Gerade besetzt!")
            Thread.Sleep(50)
        Else
            myThreadString = ""
            For z As Integer = 0 To strTemp.Length - 1
                myThreadString += strTemp.Substring(z, 1)
                Thread.Sleep(5)
            Next
            ADThreadSafeInfoBox.TSWriteLine(myThreadString)
            Monitor.Exit(myLock)
        End If
    Next
End Sub
```

In dieser Version des Arbeits-Thread erfolgt die Synchronisation durch die Monitor-Klasse. Allerdings ist der Thread, der auf seinen Vorgänger warten muss, ein wenig ungeduldig. Bekommt er nicht innerhalb von einer Millisekunde Zugriff auf den kritischen Abschnitt, verarbeitet er den Code für die entsprechenden Werterhöhung nicht, sondern gibt nur lapidar die Meldung *Threadname meldet: Gerade besetzt!* aus.

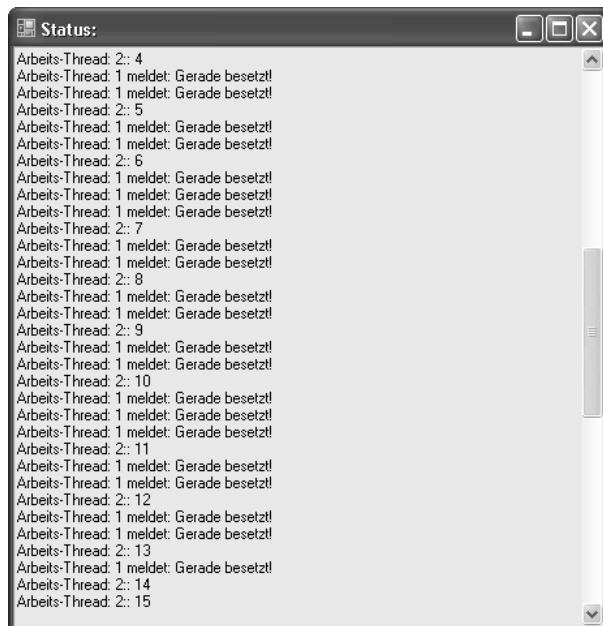


Abbildung 31.4: Ein Thread, der auf den kritischen Abschnitt nicht zugreifen kann, wartet maximal eine Millisekunde, bevor er mit einer lapidaren Meldung den Wartevorgang abbricht

Möglich wird das durch die Verwendung der TryEnter-Methode der Monitor-Klasse – die im Übrigen statisch ist; Sie brauchen die Monitor-Klasse also nicht zu instanzieren (Sie könnten es auch gar nicht). Diese Methode überprüft, ob der Zugriff auf einen kritischen Abschnitt, der durch ein angebbares Objekt genau wie bei SyncLock gesteuert wird, freigegeben ist. Ist er freigegeben, liefert sie True als Funktionsergebnis zurück. Ist er nicht freigegeben, ergibt sie False.

Auf diese Weise gibt es beim Ablauf des Programms Effekte, wie Sie sie auch in Abbildung 31.4 beobachten können.

Doch die Monitor-Klasse kann noch mehr, wie das folgende Beispiel zeigt.

Angenommen Sie möchten, dass im bislang gezeigten Beispielprogramm die durchgeführten Operationen in Fünferportionen durchgeführt werden. In diesem Fall benötigen Sie drei weitere Funktionen oder besser: Funktionalitäten:

- Sie benötigen eine Methode, die einen Thread in einen Wartezustand versetzt, wenn er sich innerhalb eines kritischen Bereichs befindet.
- Sie benötigen eine Methode, die einen Thread, der sich in einen Wartezustand versetzt hat, wieder aufwecken kann. Gibt es mehrere wartende Threads, sollte die Funktion in der Lage sein, nicht nur alle, sondern auch nur einen (beliebigen) Thread wieder zum Leben zu erwecken.

Diese Möglichkeiten bzw. Methoden gibt es. Sie heißen Monitor.Wait, Monitor.PulseAll und Monitor.Pulse. Für unser Vorhaben brauchen wir darüber hinaus eine Technik, die es uns ermöglicht, zwischen einem und mehreren laufenden Threads zu unterscheiden. Denn solange nur ein einzelner Thread läuft, darf der sich natürlich nicht in den Schlafmodus versetzen, da es keinen anderen gibt, der ihn wieder wach rütteln könnte. Doch diese Technik zu implementieren ist simpel: Eine einfache, statische Thread-Zählvariable vollführt diesen Trick. Wird ein neuer Thread gestartet, wird der Zähler zu Beginn der Thread-Prozedur hoch gezählt; beendet er seine Arbeitsroutine, zählt er ihn wieder

runter. Er versetzt sich nur dann in Schlaf, wenn noch mindestens zwei Threads laufen. Getreu dem Motto »Der Letzte macht das Licht aus« muss jeder Thread vor dem Verlassen seiner Arbeitsprozedur noch überprüfen, ob es weitere Threads gibt und den jeweils letzten prophylaktisch aus seinem Dornröschenschlaf erwecken.

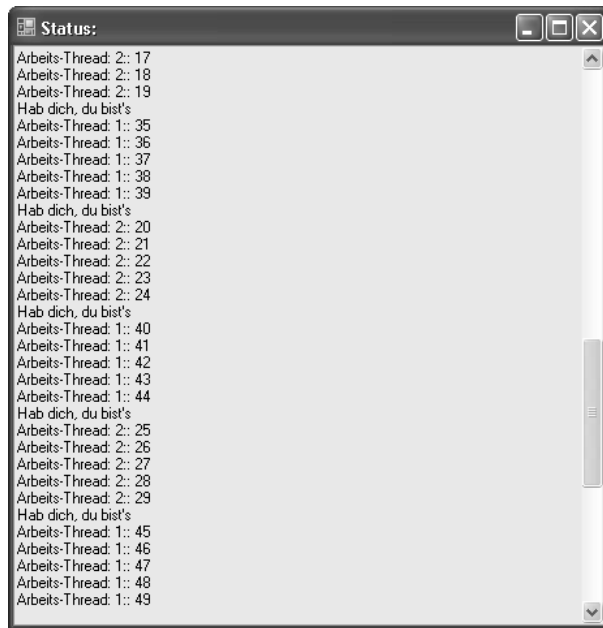


Abbildung 31.5: Ein Thread, der auf den kritischen Abschnitt nicht zugreifen kann, wartet maximal eine Millisekunde, bevor er mit einer lapidaren Meldung den Wartevorgang abbricht

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap31\SimpleThread04 (Monitor Wait Pulse)`

Wenn Sie dieses Programm starten und mindestens zwei Mal auf die Schaltfläche zum Starten des Threads klicken, sehen Sie nach einer Weile ein Ergebnis, wie es in etwa dem in Abbildung 31.5 gezeigten entspricht.

Das entsprechende Listing des Arbeits-Threads sieht folgendermaßen aus:

```
'Die brauchen wir, um festzustellen, wie viele Threads unterwegs sind.
Private myThreadCount As Integer

'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim strTemp As String
    Dim stepCount As Integer

    'Neuer Thread: Zählen!
    myThreadCount += 1
```

```

'Hier beginnt der kritische Abschnitt.
Monitor.Enter(Me)
For c As Integer = 0 To 50
    strTemp = Thread.CurrentThread.Name + ":: " + c.ToString
    'Der Thread wartet maximal eine Sekunde; bekommt er in dieser
    'Zeit keinen Zugriff auf den Code, steigt er aus.
    'Zugriff wurde gewährt - jetzt nimmt der Arbeits-Thread
    'erst seine eigentliche Arbeit auf.
    myThreadString = ""
    For z As Integer = 0 To strTemp.Length - 1
        myThreadString += strTemp.Substring(z, 1)
        Thread.Sleep(1)
    Next
    ADThreadSafeInfoBox.TSWriteLine(myThreadString)
    'Der Thread darf sich nur dann schlafen legen, wenn mindestens
    'ein weiterer Thread unterwegs ist, der ihn wieder wecken kann.
    If myThreadCount > 1 Then
        stepCount += 1
        If stepCount = 5 Then
            stepCount = 0
            'Ablösung naht!
            Monitor.Pulse(Me)
            ADThreadSafeInfoBox.TSWriteLine("Hab dich, du bist's")
            'Abgelöster geht schlafen.
            Monitor.Wait(Me)
        End If
    End If
Next
If myThreadCount > 1 Then
    'Alle anderen schlafen zu dieser Zeit.
    'Also mindestens einen wecken, bevor dieser Thread geht.
    'Passiert das nicht, schlafen die anderen bis zum nächsten Stromausfall...
    Monitor.Pulse(Me)
End If
'Hier ist der kritische Abschnitt wieder vorbei.
Monitor.Exit(Me)
'Thread-Zähler vermindern.
myThreadCount -= 1
End Sub

```

Synchronisieren von beschränkten Ressourcen mit Mutex

Wenn Threads lediglich theoretische Aufgaben lösen müssen, ist es fast egal, wie viele Threads gleichzeitig laufen (natürlich sollten dabei Sie berücksichtigen, dass Sie grundsätzlich nur so viele Threads wie gerade nötig verwenden sollten, da das Umschalten zwischen den Threads selbst natürlich auch nicht wenig an Rechenleistung verschlingt). Doch wenn bestimmte Anwendungen Komponenten der nur begrenzt vorhandenen Hardware benötigen, dann müssen diese Komponenten zwischen den verschiedenen Threads ideal aufgeteilt werden.

An dieser Stelle kommt die `Mutex`-Klasse ins Spiel. Ihr Vorteil: Sie funktioniert zwar prinzipiell wie die `Monitor`-Klasse, doch sie ist instanzierbar. Das bedeutet: Sie können durch `Mutex`-Instanzen, deren Anzahl Sie von der Verfügbarkeit benötigter Hardwarekomponenten abhängig machen, einen Verteiler organisieren, der sich um die Zuteilung der jeweils nächsten freien Hardware-Komponente kümmert.

Das folgende Beispiel demonstriert den Umgang mit `Mutex`-Klassen. Damit dieses Beispiel die `Mutex`-Klasse auch hardwareunabhängig auf jedem Rechner demonstrieren kann, ist es ein wenig anders aufgebaut als die Beispiele, die Sie bisher kennen gelernt haben: Drei Ausgabefelder innerhalb des Formulars dienen zur Emulation von drei Hardwarekomponenten; Ziel des Programms ist es, die laufenden Threads, die inhaltlich nichts anderes machen als das vorherige Beispielprogramm, so auf die drei Komponenten aufzuteilen, dass sie optimal genutzt werden.

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap31\SimpleThread05 (MutexDemo)\`.

Wenn Sie das Programm starten, sucht es sich nach dem Klick auf die Schaltfläche `Thread starten` die erste freie Hardware-Ressource (also ein Ausgabefeld). Ein weiterer Klick auf diese Schaltfläche startet einen weiteren Thread, der das nächste freie Ausgabefenster verwendet. Wenn Sie den vierten Thread starten, während alle vorherigen Threads noch laufen, wartet dieser auf das nächste freie Fenster. Sobald einer der Threads beendet ist, übernimmt er dessen Ausgabefenster. Die Arbeits-Thread-Routine ist so ausgelegt, dass eine einzelne Operation innerhalb des Threads unterschiedlich lange dauert. Nur so kann eine realistische Emulation einer simulierten Hardwarekomponente realisiert werden. Zu diesem Zweck gibt es ein im Konstruktor des Programms initialisiertes `Random`-Objekt, das den jeweils nächsten zufälligen Wert für die `Sleep`-Methode eines Arbeits-Threads generiert.

HINWEIS: Das Programm gibt die Bildschirmmeldungen nicht direkt mit einer bestimmten Anweisung aus – so wie Sie es von den bisherigen Beispielen gewohnt waren. Stattdessen verwendet es drei `TextBox`-Steuerelemente für die Textausgabe.

Bitte beachten Sie, dass die Veränderung der Eigenschaften eines Steuerelements ausschließlich aus dem Thread erfolgen darf, in dem das Steuerelement erstellt wurde. Aus diesem Grund werden Sie die Vorgehensweise zur Aktualisierung von Eigenschaften in diesem Beispiel ein wenig befremdlich finden. Ich werde im ► Abschnitt »Verwenden von Steuerelementen in Threads« ab Seite 953 auf dieses Problem genauer eingehen.

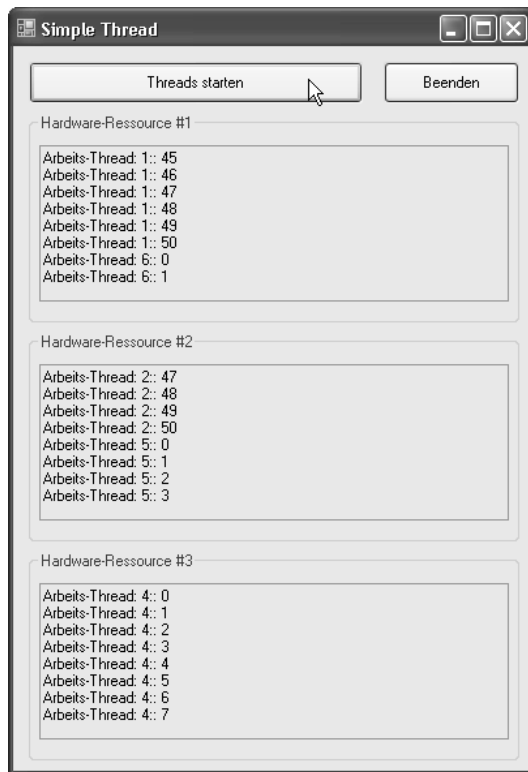


Abbildung 31.6: Drei Ausgabebereiche simulieren drei Hardware-Ressourcen. Über ein Array von *Mutex*-Objekten wird erkannt, welche als Nächstes zur Verfügung steht; der nächste anstehende Thread läuft dann im nächsten freien Fenster.

Das Codelisting:

```
'Speicher für Mutex-Objekte
Private myMutexes() As Mutex

'Speicher für TextBox-Controls
Private myTextBoxes() As TextBox

'Zufallsgenerator für die künstlichen Wartezeiten im
'Arbeitsthread. Damit kann ein Thread unterschiedlich lange dauern.
Private myRandom As Random

'Delegat für das Aufrufen von Invoke, da Controls nicht
'thread-sicher sind.
Delegate Sub AddTBTextTSActuallyDelegate(ByVal tb As TextBox, ByVal txt As String)

Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()
    'Mutexes definieren.
    myMutexes = New Mutex() {New Mutex, New Mutex, New Mutex}
    'Textbox-Array zuweisen.
```

```

myTxtBoxes = New TextBox() {txtHardware1, txtHardware2, txtHardware3}
'Zufallsgenerator initialisieren.
myRandom = New Random(DateTime.Now.Millisecond)

End Sub

'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Werteausgabe übernimmt.
Private Sub UmfangreicheBerechnung()

    Dim locMutexIndex As Integer
    Dim locTextBox As TextBox

    'Hier beginnt der kritische Abschnitt.
    'Warten, bis eine TextBox "frei" wird.
    locMutexIndex = Mutex.WaitAny(myMutexes)
    'Textbox, die dem freien Mutex entspricht, finden.
    locTextBox = myTxtBoxes(locMutexIndex)
    For c As Integer = 0 To 50
        SyncLock Me
            'Text in die TextBox des Threads ausgeben.
            AddTBTextTS(locTextBox, Thread.CurrentThread.Name + ":: " + c.ToString + vbNewLine)
        End SyncLock
        'Eine zufällige Weile lang warten.
        Thread.Sleep(myRandom.Next(50, 400))
    Next
    'Hier ist der kritische Abschnitt wieder vorbei.
    'Verwendete TextBox (Mutex) wieder freigeben.
    myMutexes(locMutexIndex).ReleaseMutex()
End Sub

```

Ein paar zusätzliche Erklärungen zu diesem Programm: Ein Mutex-Array wird als Klassen-Member zur Verwaltung der zur Verfügung stehenden Ressourcen im Konstruktor des Programms erstellt. Mit der statischen Funktion `WaitAny`, der ein Mutex-Array übergeben wird, wartet ein Thread solange, bis eines der Mutex-Objekte im Array frei wird. Wird es frei, liefert `waitAny` den Index auf das jetzt freie Mutex-Objekt zurück, ändert dessen Zustand aber gleichzeitig in »blockiert«. Der zurück gelieferte Index wird anschließend verwendet, um die korrelierende TextBox zu finden, die diesem Mutex (nur über den Indexwert) quasi zugeordnet ist. Der Thread verwendet anschließend diese TextBox (wird über `locTextBox` referenziert), um die Ausgabe durchzuführen. `waitAny` wartet übrigens nicht nur auf einen freien Mutex, sondern blockiert ihn auch wieder für den Thread, der ihn angefordert hat. Der Mutex bleibt solange im »Blockiert-Zustand«, bis der Thread den Mutex mit der `ReleaseMutex`-Methode wieder freigibt.

Weitere Synchronisierungsmechanismen

Neben den bereits beschriebenen Synchronisierungsmechanismen kennt das .NET-Framework weitere Techniken, die in den folgenden Abschnitten kurz angerissen werden sollen:

Synchronization- und MethodImpl-Attribut

Das Synchronization-Attribut erlaubt die Erklärung einer ganzen Klasse zum kritischen Abschnitt. Wenn Sie eine Klasse mit diesem Attribut versehen, kann nur ein einziger Thread zur gleichen Zeit auf die Klasseninstanz zugreifen. Die Anwendung dieses Attributes funktioniert prinzipiell folgendermaßen:

```
<System.Runtime.Remoting.Contexts.Synchronization(> _  
Public Class Komplettsynchronisiert  
    Sub MacheIrgendwas()  
        'Hier die Anweisungen  
        'des Arbeitstreads.  
    End Sub  
    Sub MacheIrgendwasAnderes()  
        'Hier die Anweisungen  
        'des Arbeitstreads.  
    End Sub  
End Class
```

Wenn die Synchronisation einer ganzen Klasse zu viel des Guten wäre, steht Ihnen mit MethodImpl und dessen Parameter MethodImplOptions.Synchronized ein weiteres Attribut zur Verfügung, das nur eine einzelne Prozedur einer Klasse zum kritischen Abschnitt erklärt. Seine Anwendung funktioniert wie folgt:

```
Public Class TeilweiseSynchronisiert  
    <System.Runtime.CompilerServices.MethodImpl(Runtime.CompilerServices.MethodImplOptions.Synchronized)> _  
    Sub MacheIrgendwasSynchronisiertes()  
        'Hier die Anweisungen  
        'des Arbeitstreads.  
    End Sub  
  
    Sub MacheIrgendwasAnderes()  
        'Hier die Anweisungen  
        'des Arbeitstreads.  
    End Sub  
End Class
```

Die Interlocked-Klasse

Mit Hilfe der Interlocked-Klasse können Sie steuern, wie viele Threads einen kritischen Abschnitt betreten dürfen, bevor weitere Threads ausgesperrt werden. Prinzipiell funktioniert sie also wie die Monitor-Klasse und dessen Methode Enter, nur dass sie einen Parameter (eine Member-Variable der Klasse) angeben können, über die gesteuert wird, wie viele Threads den kritischen Abschnitt betreten dürfen.

```
Public Class InterlockedTest  
  
    Dim myThreadZählerFürInterlocked As Integer  
  
    Sub Arbeitstread()  

```

```

        'Maximal 3 Threads kommen hier gleichzeitig hinein,
        'dann ist Schluss!
    If Interlocked.Increment(myThreadZählerFürInterlocked) <= 3 Then
        'Hier die Anweisungen
        'den Arbeitsthreads.
        Interlocked.Decrement(myThreadZählerFürInterlocked)
    End If

    End Sub
End Class

```

Die Klasse selbst bietet übrigens ausschließlich statische Funktionen an; sie muss also nicht instanziiert werden (tatsächlich ist sie eine abstrakte Klasse (`Noninheritable`), und kann auch gar nicht instanziiert werden).

Die ReaderWriterLock-Klasse

Wenn mehrere Threads auf eine Datei zugreifen müssen, dann ist das so lange unkritisch, wie diese Threads aus der Datei lediglich lesen. Beim Schreiben sieht das anders aus: Sobald eine Datei geschrieben wird, darf kein anderer Thread zusätzlich in die Datei schreiben. Auch das Lesen aus einer Datei, in die gerade von einem anderen Thread geschrieben wird, könnte für die Ergebnisse einer Prozedur fatale Folgen haben.

Um dieses Problem zu lösen, gibt es die `ReaderWriteLock`-Klasse. Ihre wichtigsten Funktionen sind `AcquireReaderLock`, `ReleaseReaderLock`, `AcquireWriterLock` und `ReleaseWriterLock`.

Mehrere Threads, die sich über diese Klasse und deren Methoden synchronisieren wollen, müssen Zugriff auf dasselbe `ReaderWriterLock`-Objekt haben – eine Instanz dieser Klasse sollte also mindestens ein Klassen-Member sein oder als statische, öffentliche Eigenschaft programmweit verfügbar sein.⁵

Sowohl der Lesebereich als auch der Schreibbereich eines Threads werden nun als kritische Abschnitte definiert – etwa wie folgt:

```

Public Class ReaderWriteLockTest

    Dim myReaderWriterLock As New ReaderWriterLock

    Sub DateiLeseThread()
        'Hier kommt das Programm nur rein,
        'wenn nicht geschrieben wird, ein Schreibvorgang also
        'nicht zuvor durch ein myReaderWriterLock.AcquireWriterLock
        'eingeleitet wurde!
        '1000 Millisekunden wird auf den Lock gewartet.
        myReaderWriterLock.AcquireReaderLock(1000)
        'Lese-Thread nur ausführen, wenn Lock erteilt wurde.
        If myReaderWriterLock.IsReaderLockHeld Then

```

⁵ Es könnte ja durchaus vorkommen, dass verschiedene Klassen und deren gleichzeitig laufende Arbeits-Threads einer Anwendung auf die gleiche Datei zugreifen müssen. In diesem Fall definieren Sie eine Klasse, die im statischen Konstruktor (`Shared New`) einen statische Member vom Typ `ReaderWriterLock` instanziiert und diesen über eine öffentliche statische Eigenschaft (`Public Shared Property ...`) allen anderen Klassen zur Verfügung stellt.

```

        'Hier die Anweisungen
        'des Arbeitstreads
        'der aus einer Datei liest.
        myReaderWriterLock.ReleaseReaderLock()
    Else
        'Timeout, in die Datei konnte nicht rechtzeitig geschrieben werden.
    End If
End Sub

Sub DateiSchreibThread()
    'Hier kommt ein neuer Thread nicht eher rein,
    'als bis ein anderer Lese-Thread zu Ende gelesen wurde
    'oder ein anderer Schreib-Thread den Schreibvorgang
    'komplettiert hat.
    '1000 Millisekunden wird auf den Lock gewartet.
    myReaderWriterLock.AcquireWriterLock(1000)
    'Schreib-Thread nur ausführen, wenn Lock erteilt wurde.
    If myReaderWriterLock.IsWriterLockHeld Then
        'Hier die Anweisungen
        'des Arbeitstreads
        'der in die Datei schreibt.
        myReaderWriterLock.ReleaseWriterLock()
    Else
        'Timeout, in die Datei konnte nicht rechtzeitig geschrieben werden.
    End If

End Sub
End Class

```

Schon die Kommentare im Beispielprogramm machen die Zusammenhänge der Operationen deutlich:

- Wenn ein Thread die Leseroutine betritt, erhält er einen Leseschutz. Dieser Leseschutz schützt diesen Thread davor, dass ein neuer Schreibvorgang beginnt, *bevor* das Lesen abgeschlossen ist (und der Thread das durch `ReleaseReaderLock` angezeigt hat).
- Betritt ein Thread die Schreibroutine, wartet sein `AcquireWriterLock` solange, bis alle ausstehenden Lesevorgänge abgearbeitet sind. Neue Lese-Anforderungen werden nun solange zurückgestellt, wie `AcquireWriterLock` auf seine Schutzzuteilung wartet und diesen Schutz nach Beenden des Schreibens wieder freigegeben hat.

Synchronisieren von voneinander abhängigen Threads mit `ManualResetEvent` und `AutoResetEvent`

Eines vorweg: Lassen Sie sich von den Begriffen `Event` in den Namen dieser beiden Klassen nicht verwirren. Beide Begriffe bezeichnen instanzierbare Klassen und haben deshalb mit Ereignissen (*Events*), wie Sie sie bisher kennen gelernt haben, nicht das Geringste zu tun.

Sie verwenden beide Objekte, wenn bestimmte Threads voneinander abhängig sind. Sie haben beispielsweise einen Thread, der bestimmte Ergebnisse produziert, und weitere, die diese Ergebnisse anschließend weiterverarbeiten sollen. In diesem Fall verwenden Sie diese Objekte zur Synchronisation untereinander.

Beide Klassen unterscheiden sich lediglich in einem Punkt: Die `AutoResetEvent`-Klasse setzt ihren Zustand automatisch sofort zurück, sobald ein durch `WaitOne` geblockter Thread wieder gestartet wurde. `ManualResetEvent` macht eine Signaländerungsanzeige durch die `Set`-Eigenschaft erforderlich. Ein Beispiel für die `AutoResetEvent`-Klasse finden Sie übrigens im ► Abschnitt »Abbrechen und Beenden eines Threads« auf Seite 955.

Verwenden von Steuerelementen in Threads

Wenn Sie Steuerelemente der Hauptanwendung von einem Thread aus verwenden wollen, müssen Sie folgendes beachten: Steuerelemente sind nicht thread-sicher, das heißt, sie dürfen nur aus dem Thread heraus verwendet werden, der sie erstellt hat. In der Regel ist das der Haupt-Thread der Anwendung, also der Thread, der die Benutzeroberfläche Ihrer Anwendung regelt. Aus diesem Grund nennt man diesen Thread auch den *UI-Thread* (UI als Abkürzung für *User Interface*, oder zu Deutsch: *Benutzeroberfläche*).

Damit ein anderer Thread dennoch das Steuerelement eines Formulars verwenden kann, muss er sich eines Tricks bedienen: Er muss dem Steuerelement mitteilen, dass es eine Prozedur aus seiner Klasse aufrufen soll. Diese Prozedur verändert dann die Eigenschaften des Steuerelements. Und da das Steuerelement diese Prozedur – wenn auch indirekt – selbst aufgerufen hat, wurde sie auch vom richtigen Thread (nämlich dem UI-Thread) aus aufgerufen.

Das Steuerelement selbst stellt dafür eine der wenigen thread-sicheren Prozeduren zur Verfügung, die es besitzt. Ihr Name: `Invoke`. Ihr wird als Parameter ein Delegat übergeben, der als Zeiger auf die eigentliche Prozedur dient, die aus dem Thread aufgerufen werden soll. Im `Mutex`-Beispiel ist diese Vorgehensweise schon zur Anwendung gekommen, und aus diesem Grund werden wir dieses Programm noch einmal unter diesem Aspekt unter die Lupe nehmen.

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap31\SimpleThread05 (MutexDemo)\`.

Dieses Programm besteht aus drei `TextBox`-Steuerelementen, deren `Text`-Eigenschaft verwendet wird, um die Ausgaben des Programms anzuzeigen. Da die `TextBox`-Steuerelemente ausschließlich von der Thread-Prozedur verwendet werden (also in jedem Fall nicht vom UI-Thread), muss die Prozedur, die die Steuerelement-Eigenschaft setzt, zwingend über `Invoke` erfolgen.

```
.
.
    'Delegat für das Aufrufen von Invoke, da Controls nicht thread-sicher sind.
    Delegate Sub AddTBTextTSActuallyDelegate(ByVal tb As TextBox, ByVal txt As String)
.
.
    'Dient zum Setzen einer Eigenschaft auf einer TextBox indirekt über Invoke.
    Private Sub AddTBTextTS(ByVal tb As TextBox, ByVal txt As String)
        Dim locDel As New AddTBTextTSActuallyDelegate(AddressOf AddTBTextTSActually)

        Me.Invoke(locDel, New Object() {tb, txt})

    End Sub
```

```

Private Sub AddTBTextTSActually(ByVal tb As TextBox, ByVal txt As String)
    tb.Text += txt
    tb.SelectionStart = tb.Text.Length - 1
    tb.ScrollToCaret()
End Sub

```

Bitte lassen Sie sich in diesem Beispiel nicht von der Tatsache irritieren, dass hier ein `TextBox`-Steuerelement selbst als Parameter mit übergeben wird. Diese Vorgehensweise ist für das Beispiel deshalb notwendig, da die zu verwendende `TextBox` vom nächsten freien `Mutex`-Objekt im `Thread` abhängig ist. Da das Steuerelement hier als Parameter selbst mit übergeben wird, ist es eigentlich egal, über welches Steuerelement im Formular die `Invoke`-Methode aufgerufen wird. In diesem Beispiel ist es das Formular selbst, dessen `Invoke` verwendet wird. Genauso gut könnte die entsprechende Zeile, die `Invoke` ausführt, auch folgendermaßen ausschauen:

```
tb.Invoke(1000, New Object() {tb, txt})
```

Voraussetzung dafür ist allerdings, dass `tb` aus dem `UI-Thread` stammt.

Letzten Endes kommt es also nur darauf an, dass ein Steuerelement des `UI-Threads` den Aufruf des Delegates durchführt. Um das sicherzustellen, sollten Sie, von Ausnahmen wie in diesem Fall einmal abgesehen, die `Invoke`-Methode des Steuerelements, auf das sich die Änderungen auch beziehen, für einen indirekten Delegationaufruf verwenden.

Managen von Threads

`Thread`-Objekte verfügen über einige Methoden, mit denen sie sich sowohl selbst steuern als auch von außen steuern lassen können. Zwei dieser Methoden haben Sie bereits kennen gelernt: die `Start`- und die `Sleep`-Methode. Die nächsten Abschnitte nehmen diese und weitere Methoden der `Thread`-Klasse ein wenig genauer unter die Lupe.

Starten eines Threads mit Start

Sie starten einen `Thread` mit seiner `Start`-Methode. Voraussetzung dafür ist, dass Sie den `Thread` zuvor mit einem `ThreadStart`-Objekt instanziiert haben, das bei seiner Instanzierung wiederum die Adresse der Prozedur erhalten hat, die den eigentlichen `Thread` darstellt.

Vorübergehendes Aussetzen eines Threads mit Sleep – Statusänderungen im Framework bei Suspend und Resume

Wenn ein `Thread` für eine gewisse Zeit unterbrochen werden soll, verwenden Sie die `Sleep`-Methode des `Thread`-Objektes. `Sleep` übernimmt als Parameter entweder einen *Integer*-Wert, der die abzuwartende Zeitspanne in Millisekunden bestimmt oder einen `TimeSpan`-Wert, der ebenfalls die abzuwartende Dauer bestimmt, nach deren Ablauf der `Thread` wieder aktiv wird. Mit der `Sleep`-Methode kann sich ein `Thread` dadurch selbst »aussetzen« und wieder »aufwecken«.

Die Methoden `Suspend` und `Resume` sind im `.NET 2.0`-Framework übrigens als veraltet markiert, und sollten laut Microsoft nicht mehr verwendet werden. Stattdessen sollten Sie andere Synchronisierungstechniken verwenden.

Wie Sie einen Update Ihres Source-Codes vornehmen, zeigt der Vergleich der beiden Beispiele in den Verzeichnissen *ObsoleteThreadMethodsDemo* und *ThreadPoolThreads* im Verzeichnis der Beispielprojekte dieses Kapitels. Sie entsprechen beide dem Prinzip des Beispiels, das in ► Abschnitt »Verwenden des Thread-Pools« ab Seite 970 besprochen wird. Im ersten Verzeichnis finden Sie jedoch die veraltete Version, im zweiten die entsprechend angepasste.

Abbrechen und Beenden eines Threads

Abort nennt sich die Methode, mit der Sie die Möglichkeit haben, einen Thread abzubrechen. Allerdings sollten Sie von dieser Funktion nur in Ausnahmefällen Gebrauch machen, denn: Abort nicht unmittelbar ausgeführt. Framework-intern wird eine Ausnahme vom Typ *ThreadAbortException* ausgelöst. Diese Ausnahme ist allerdings etwas Besonderes, da Sie sie nicht abfangen können. Falls der Thread jedoch in einem *Try/Catch/Finally*-Codeblock ausgeführt wird, garantiert das Framework, dass der *Finally*-Block bei Abort auf jeden Fall abgearbeitet wird. Ein Thread kann dabei mithilfe seiner *ThreadState*-Eigenschaft feststellen, dass er gerade abgebrochen wird, und das Abbrechen sogar mit *ResetAbort* verhindern.

Allerdings ist das Abbrechen eines Threads keine elegante Vorgehensweise – eher die Brechstangenmethode. Ein Thread sollte ausschließlich beendet werden, indem er seine Arbeitsprozedur verlässt. Auf der anderen Seite muss ein Thread auf jeden Fall dafür sorgen, dass er beendet wird, wenn das Hauptprogramm beendet wird, das ihn beherbergt.

In den vorangegangenen Beispielen ist das bislang nicht der Fall gewesen. Wenn Sie das Hauptfenster schließen, wird zwar das Hauptprogramm beendet – Threads, die zu dieser Zeit noch aktiv sind, laufen aber unbehelligt weiter. Das heißt, für das letzte Beispiel gilt »unbehelligt« nicht wirklich. Tatsächlich steigt das Programm mit einer Fehlermeldung aus, wenn Sie es schließen, während andere Threads noch munter am Werkeln sind. Das liegt daran, dass der Thread seine Ausgabe in eine *TextBox* vornimmt, die es zu diesem Zeitpunkt schon gar nicht mehr gibt.

Die erste Idee, die einem in den Sinn kommen könnte, um dieses Problem zu lösen, wäre ein klassenweites Flag, das dann gesetzt wird, wenn das Formular geschlossen wird. Dazu müsste man lediglich die *OnClosing*-Routine des Formulars überschreiben. Bevor ein Thread eine Ausgabe im Formular vornimmt, könnte er dieses Flag überprüfen. Wäre es gesetzt, beendete der Thread sich mit einem simplen *Exit* Sub auf unproblematische Art und Weise.

Allerdings ist diese Problemlösung – jedenfalls was Windows Forms-Anwendungen anbelangt – nicht konsequent zu Ende gedacht, denn: Wird dieses »Thread-Abbrechen-Flag« in dem Moment auf *True* gesetzt, in dem sich der Thread gerade zwischen der Flag-Abfrage und der Ausgabe des Textes ins Steuerelement befindet, hat die ganze Aktion überhaupt nichts gebracht. Auch in diesem Fall rauscht der Thread ungebremst in die Ausgaberroutine für die *TextBox*, die es auch in diesem Fall nicht mehr gibt. Abermals ist eine Ausnahme die Folge. Was also tun?

Eine Synchronisierungstechnik ist an dieser Stelle wieder indiziert. *OnClosing* muss so lange warten, bis ein Thread den kritischen Bereich des Schreibens in die *TextBox* beendet hat. Dazu muss der Thread diesen kritischen Bereich aber erst einmal definieren und dafür bietet sich in diesem Fall das *AutoRaiseEvent*-Objekt an (das *ManualRaiseEvent*-Objekt täte es übrigens genau so gut für unser Vorhaben).

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap31\SimpleThread06 (MutexDemo proper)`.

Wenn Sie dieses Beispiel starten, können Sie das Formular schließen, egal wie viele Threads noch unterwegs sind. Der veränderte Code für das Beispielprogramm sieht folgendermaßen aus:

```
'Member-Variable, damit die Threads durchnummeriert werden können.
'Dient nur zur späteren Unterscheidung des laufenden Threads, wenn
'er Ergebnisse im Ausgabefenster darstellt.
Private myArbeitsThreadNr As Integer = 1

'Speicher für Mutex-Objekte
Private myMutexes() As Mutex

'Speicher für TextBox-Controls
Private myTextBoxes() As TextBox

'Zufallsgenerator für die künstlichen Wartezeiten im
'Arbeitsthread. Damit kann ein Thread unterschiedlich lange dauern.
Private myRandom As Random

'Delegat für das Aufrufen von Invoke, da Controls nicht
'Thread-Sicher sind.
Delegate Sub AddTBTextTSActuallyDelegate(ByVal tb As TextBox, ByVal txt As String)

'Flag, dass bestimmt, wann alle Threads zu gehen haben
Private myAbortAllThreads As Boolean

'Zusätzliches Synchronisierungsobjekt für "Alle Threads beenden!"
Private myAutoResetEvent As AutoResetEvent

Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()
    'Mutexes definieren
    myMutexes = New Mutex() {New Mutex, New Mutex, New Mutex}
    'Textbox-Array zuweisen
    myTextBoxes = New TextBox() {txtHardware1, txtHardware2, txtHardware3}
    'Zufallsgenerator initialisieren
    myRandom = New Random(DateTime.Now.Millisecond)
    'Zusätzliches Synchronisierungsobjekt für "Alle Threads beenden!" initialisieren
    myAutoResetEvent = New AutoResetEvent(True)

End Sub

'Dies ist der eigentliche Arbeits-Thread (Worker Thread), der das
'Hochzählen und die Wertausgabe übernimmt
Private Sub UmfangreicheBerechnung()
    Dim locMutexIndex As Integer
    Dim locTextBox As TextBox
```

```

'Hier beginnt der 1. kritische Abschnitt der Mutexes
'Warten, bis eine TextBox "frei" wird
locMutexIndex = Mutex.WaitAny(myMutexes)
'Thread beenden, wenn das "Alles-Threads-Beenden-Flag" während
'des Wartens signalisiert wurde
If myAbortAllThreads Then
    'Verwendeten TextBox-Mutex wieder freigeben,
    'sonst knallt es, wenn die anderen Threads in diesem
    'Abschnitt beendet werden sollen!
    myMutexes(locMutexIndex).ReleaseMutex()
    Exit Sub
End If
'Textbox, die dem freien Mutex entspricht finden
locTextBox = myTxtBoxes(locMutexIndex)
'Hier beginnt der 2. kritische Abschnitt für OnClosing
myAutoResetEvent.Reset()
For c As Integer = 0 To 20
    SyncLock Me
        'Falls abgebrochen werden soll,
        If myAbortAllThreads Then
            'OnClosing benachrichtigen
            myAutoResetEvent.Set()
            'Verwendeten TextBox-Mutex wieder freigeben.
            'Grund: Siehe oben.
            myMutexes(locMutexIndex).ReleaseMutex()
            Exit Sub
        End If
        'Text in die TextBox des Threads ausgeben
        AddTBTextTS(locTextBox, Thread.CurrentThread.Name + ":: " + c.ToString + vbNewLine)
        Console.WriteLine(Thread.CurrentThread.Name + ":: " + c.ToString + vbNewLine)
    End SyncLock
    'Eine zufällige Weile lang warten
    Thread.Sleep(myRandom.Next(50, 400))
Next
'2. kritische Abschnitt beendet (OnClosing)
myAutoResetEvent.Set()
'Hier ist der 1. kritische Abschnitt wieder vorbei.
'Verwendete TextBox (Mutex) wieder freigeben
myMutexes(locMutexIndex).ReleaseMutex()
End Sub

Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    'WICHTIG: Ein einfaches Warten mit WaitOne reicht in diesem
    'Fall nicht aus, da es sich um den Hauptthread handelt.
    'Dann aber würde WaitOne die Nachrichtenwarteschlange blockieren,
    'und ohne die funktioniert Invoke nicht. Deswegen wird hier ein Timeout
    'angegeben, der Nachrichtenwarteschlange die Möglichkeit gegeben, einmal
    '"Luft zu holen", und dann weiter gewartet.
    myAbortAllThreads = True
    'DoEvents beim Warten in ein-Millisekunden-Abständen auslösen

```

```

        Do While Not myAutoResetEvent.WaitOne(1, True)
            Application.DoEvents()
        Loop
    End Sub

    'Dient zum Setzen einer Eigenschaft auf einer TextBox indirekt über Invoke
    Private Sub AddTBTextTS(ByVal tb As TextBox, ByVal txt As String)
        Dim locDel As New AddTBTextTSActuallyDelegate(AddressOf AddTBTextTSActually)

        Me.Invoke(locDel, New Object() {tb, txt})
    End Sub

    Private Sub AddTBTextTSActually(ByVal tb As TextBox, ByVal txt As String)
        tb.Text += txt
        tb.SelectionStart = tb.Text.Length - 1
        tb.ScrollToCaret()
    End Sub

    Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnThreadStarten.Click
        'Dieses Objekt kapselt den eigentlichen Thread
        Dim locThread As Thread
        'Dieses Objekt benötigen Sie, um die Prozedur zu bestimmen,
        'die den Thread ausführt.
        Dim locThreadStart As ThreadStart

        'Threadausführende Prozedur bestimmen
        locThreadStart = New ThreadStart(AddressOf UmfangreicheBerechnung)
        'ThreadStart-Objekt dem Thread-Objekt übergeben
        locThread = New Thread(locThreadStart)
        'Thread-Namen bestimmen
        locThread.Name = "Arbeits-Thread: " + myArbeitsThreadNr.ToString
        'Thread starten
        locThread.Start()
        'Zähler, damit die Threads durch ihren Namen unterschieden werden können
        myArbeitsThreadNr += 1
    End Sub

    Private Sub btnBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnBeenden.Click
        'Einfach so geht's normalerweise nicht
        Me.Close()
    End Sub

```

Eine kleine Anmerkung vielleicht noch an dieser Stelle zur überschriebenen OnClosing-Prozedur: Wie aus den Kommentaren im Codelistung schon ersichtlich, gilt es noch ein letztes Problem aus der Welt zu schaffen. Ein einfaches WaitOne würde den UI-Thread einfrieren. Dummerweise benötigt Invoke eines Steuerelements eine aktive Nachrichtenwarteschlange, um zu funktionieren. Aus diesem Grund müssen wir an dieser Stelle wieder einen Trick anwenden, damit sich das Programm beim Warten auf die letzte Textausgabe nicht aufhängt. Als Argument übergeben wir einen Timeout-Wert von einer

Millisekunde, nachdem `WaitOne` zunächst den Wartevorgang auf das Abschließen des kritischen Abschnittes eines Threads unterbricht. Ein anschließendes `DoEvents` erlaubt das Ausführen eines möglicherweise noch ausstehenden `Invoke` im Arbeits-Thread. `DoEvents` wird in einer Schleife abgehandelt, die so lange ausgeführt wird, bis `WaitOne` zurückmeldet, dass es nicht durch `Timeout`, sondern tatsächlich durch die Signalisierung des Arbeits-Threads beendet wurde. In diesem Fall »weiß« `OnClosing`, dass der ausstehende Thread beendet wurde, die `TextBox` zur Ausgabe ergo nicht mehr gebraucht wird und das Formular zu diesem Zeitpunkt sicher geschlossen werden kann.

HINWEIS: Da die Verarbeitung dieser Warteschleife natürlich auch Zeit kostet, sollten Sie bei sehr zeitkritischen Operationen von dieser Methode absehen. Die Prozessorleistung, die für die Warteschleife benötigt wird, fehlt anderen Threads natürlich. Eine Zwischenlösung: Sie können auf Kosten der Reaktionszeit natürlich den `Timeout`-Parameter für `WaitOne` erhöhen, um Leistung zu sparen. Denn während `WaitOne` »ausgeführt« wird, können andere Threads bedient werden.

Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen

Sie haben im Laufe der letzten Kapitel bereits feststellen können, dass Sie beim Einrichten eines Threads keine Parameter an die eigentliche Thread-Prozedur übergeben können. In der Praxis sind Threads, denen keine Daten zum Verarbeiten übergeben werden können, aber eher nutzlos. Doch so groß ist das Problem nicht, denn es gibt einen ganz einfachen Trick, mit dem Sie es meistern können: Sie kapseln die eigentliche Thread-Routine einfach in einer Klasse. Die notwendigen Parameter, die die Thread-Routine benötigt, können Sie dann einfach im Konstruktor der Klasse übergeben.

Da der Thread asynchron läuft, als nur sozusagen »angeschmissen« wird und dann alleine weiterarbeitet, muss die Thread-Klasse natürlich auch in der Lage sein, ein »ich habe fertig« an die ihn einbindende Instanz mitteilen zu können. Doch auch dieses Problem ist vergleichsweise einfach in den Griff zu bekommen: Wenn der Arbeits-Thread innerhalb der Klasse seine Aufgabe abgeschlossen hat, löst er einfach ein Ereignis aus. Die Ergebnisse, die durch ihn zustande gekommen sind, können anschließend durch Eigenschaften der Öffentlichkeit zugänglich gemacht werden.

Das folgende Codelisting zeigt, wie beispielsweise eine Klasse aussehen kann, die ein beliebiges Array in einem Thread sortieren kann.

```
Public Class SortArrayThreaded
    Implements IDisposable

    Public Event SortCompleted(ByVal sender As Object, ByVal e As SortCompletedEventArgs)
    Private myArrayToSort As ArrayList
    Private myAutoResetEvent As AutoResetEvent
    Private myTerminateThread As Boolean
    Private myThreadStart As ThreadStart
    Private myThread As Thread
    Private mySortingComplete As Boolean

    'Konstruktor übernimmt die zu sortierende Liste. Der ThreadName hat nur
    'dokumentarischen Charakter.
    Sub New(ByVal ArrayToSort As ArrayList, ByVal ThreadName As String)
```

```

myArrayToSort = ArrayToSort
myThread = New Thread(New ThreadStart(AddressOf SortArray))
myThread.Name = ThreadName
myAutoResetEvent = New AutoResetEvent(True)
End Sub

```

Der eigentliche Thread kann beginnen, wenn die die Klasse einbindende Instanz nach der Instanzierung die folgende Methode aufruft. Damit der Thread mit anderen Aufgaben synchronisiert werden kann, stellt er ein `AutoResetEvent`-Objekt zur Verfügung, dessen Signalisierung an dieser Stelle zurückgesetzt wird. Ein anderer Thread, der darauf wartet, dass die Sortierung beendet wird, kann nun nicht nur durch das Ereignis, sondern auch über die `WaitOne`-Methode (oder, wenn mehrere Threads zu synchronisieren sind, auch über `WaitAny` oder `WaitAll`) über das Ende der Sortierung informiert werden.

```

'Thread starten und signalisieren, dass blockiert.
Public Sub StartSortingAsynchron()
    myAutoResetEvent.Reset()
    myThread.Start()
End Sub

Private Sub SortArray()

    Dim locSortCompletionStatus As SortCompletionStatus

    If Not myArrayToSort Is Nothing Then
        locSortCompletionStatus = Me.ShellSort()
    End If
    myAutoResetEvent.Set()
    'Dieses Ereignis könnte man einbinden, falls der Sort-Thread über das
    'Sortierende benachrichtigen soll.
    RaiseEvent SortCompleted(Me, New SortCompletedEventArgs(locSortCompletionStatus))
End Sub

```

Kleine Anmerkung am Rande: Die folgende Prozedur stellt den eigentlichen Arbeits-Thread der Klasse dar. Natürlich gibt es eine Reihe von Sortiermöglichkeiten von Elementen, die im Framework eingebaut sind. Aber behalten Sie im Hinterkopf, dass es hier in erster Linie um die Demonstration der grundsätzlichen Threading-Möglichkeiten geht.

```

'Sortiert eine ArrayList, die IComparable-Member enthält.
'Null-Werte oder inkompatible Typen werden nicht geprüft!
Private Function ShellSort() As SortCompletionStatus

    Dim locOutCount, locInCount As Integer
    Dim locDelta As Integer
    Dim locElement As IComparable

    locDelta = 1

    'Größten Wert der Distanzfolge ermitteln.
    Do
        locDelta = 3 * locDelta + 1
    Loop Until locDelta > myArrayToSort.Count

```

```

Do
    'War eins zu groß, also wieder teilen.
    locDelta \= 3

    'Shellsorts Kernalgorithmus
    For locOutCount = locDelta To myArrayToSort.Count - 1
        locElement = CType(myArrayToSort(locOutCount), IComparable)
        locInCount = locOutCount
        Do While CType(myArrayToSort(locInCount - locDelta), IComparable).CompareTo(locElement) = 1
            If myTerminateThread Then
                Return SortCompletionStatus.Aborted
            End If
            myArrayToSort(locInCount) = myArrayToSort(locInCount - locDelta)
            locInCount = locInCount - locDelta
            If (locInCount <= locDelta) Then Exit Do
        Loop
        myArrayToSort(locInCount) = locElement
    Next
    Loop Until locDelta = 0
    Return SortCompletionStatus.Completed

End Function

```

WICHTIG: Damit ein laufender Thread auf elegante Art und Weise beendet werden kann (und nicht mit brachialer Gewalt durch Abort), gibt es eine Member-Variable namens `myTerminateThread`, deren Setzen auf `True` bewirkt, dass der Sortalgorithmus seine Arbeit abbricht, die Prozedur verlässt und damit den Arbeits-Thread sauber zu Ende führt. Um ein Programm nicht durch laufende Sortier-Threads zu blocken, wenn es für die Klasseninstanz Zeit wird, vom Garabage Collector entsorgt zu werden, implementiert die Klasse `IDisposable` und damit `Dispose`, das diesen Member setzt. Der Garbage Collector ist damit spätestens derjenige, der den Thread beendet. Eigentlich ist das aber nur das Netz unter dem Trapez. Sie sollten in jedem Fall selbst durch geschickte Programmierung darauf achten, dass kein noch laufender Thread existiert, wenn Sie das umgebende Programm beenden. Rufen Sie im Zweifelsfall lieber selbst die `Dispose`-Methode Ihrer Thread-Klasse auf, damit der Thread in jedem Fall ordnungsgemäß beendet wird.

```

'Dispose beendet einen vielleicht noch laufenden Sortier-Thread.
Public Sub Dispose() Implements System.IDisposable.Dispose
    myTerminateThread = True
End Sub

'Stellt die benötigten Eigenschaften zur Verfügung.
Public ReadOnly Property AutoResetEvent() As AutoResetEvent
    Get
        Return myAutoResetEvent
    End Get
End Property

Public ReadOnly Property ArrayList() As ArrayList
    Get
        Return myArrayToSort
    End Get
End Property
End Class

```

```
'Die beiden möglichen Ergebnisse, wenn der Sortier-Thread beendet wurde.
'Falls er durch Dispose abgebrochen wurde, liefert er Aborted zurück.
Public Enum SortCompletionStatus
    Completed
    Aborted
End Enum
```

Damit das Ereignis, das ausgelöst wird, wenn der Thread die Sortierung beendet hat, auch ordnungsgemäß über die durchgeführten Vorgänge informieren kann, gibt es zusätzlich zur eigentlichen Klasse noch ein paar Hilfselemente für diesem Zweck:

```
'Dient nur der Ereignisübergabe, wenn Sie das SortCompleted-Ereignis
'nach Abbruch oder Abschluss des Sortierens auslösen wollen.
Public Class SortCompletedEventArgs
    Inherits EventArgs

    Private mySortCompletionStatus As SortCompletionStatus
    Sub New(ByVal scs As SortCompletionStatus)
        mySortCompletionStatus = scs
    End Sub

    Public Property SortCompletionStatus() As SortCompletionStatus
        Get
            Return mySortCompletionStatus
        End Get
        Set(ByVal Value As SortCompletionStatus)
            mySortCompletionStatus = Value
        End Set
    End Property
End Class
```

Soviel zum eigentlichen Arbeitstier dieses Beispiels. Nun ist es an der Zeit herauszufinden, ob es auch wirklich das leistet, was wir von ihm erwarten.

Der Einsatz von Thread-Klassen in der Praxis

Sie finden in den Begleitdateien ein Beispiel, dass die im vorherigen Abschnitt vorgestellte Klasse demonstriert. Und es macht nicht nur das. Es zeigt auch die Geschwindigkeitsverhältnisse von Programmen, wenn Sie mit einem oder mehreren Threads bestimmte Probleme lösen.

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap31\ThreadsInPraxis (MultiThreadingSorting)\`.

Wenn Sie das Programm starten, zeigt es Ihnen einen Dialog, wie Sie ihn in etwa auch in Abbildung 31.7 sehen können. Bevor Sie den Test starten, entscheiden Sie sich durch Anklicken des entsprechenden Feldes, ob Sie für die Sortierung der Daten Single- oder Multithreading verwenden wollen. Mit einem weiteren Klick auf *Benchmark* starten Sie die Threading-Operationen.

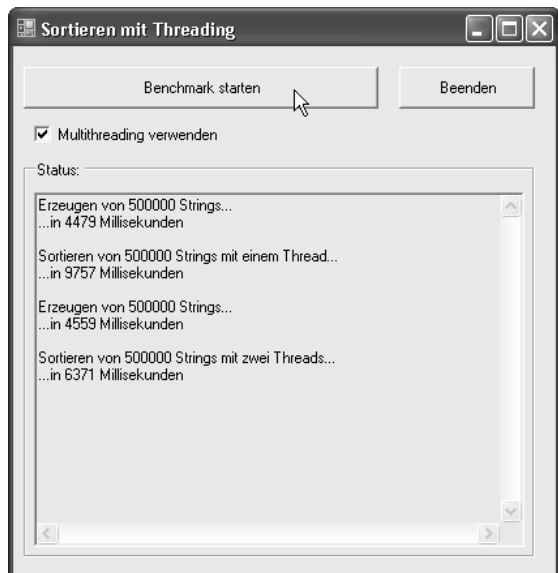


Abbildung 31.7: Solche Unterschiede erreichen Sie nur auf einem Multiprozessor- oder MultiCore-System. Wichtig: Beim Einsatz von Hyperthreading auf einem Prozessor kann das Sortieren unter Umständen sogar länger dauern, als beim ausgeschalteten Hyperthreading.

Das folgende Codelisting zeigt, wie das Programm funktioniert. Längere Kommentare im Listing finden Sie im Folgenden der leichteren Lesbarkeit wegen in Normalschrift gesetzt und dort, wo es sinnvoll erscheint, etwas ausführlicher beschrieben. Sie sind im Listing aber ebenfalls vorhanden.

Einige Worte zur generellen Funktionsweise vorweg: Das Programm arbeitet mit maximal vier Threads, aber mindestens drei Threads, wenn die Sortierung vorgenommen wird. Der UI-Thread – also der Thread, der das Formular verwaltet – startet automatisch mit dem Programm. Wenn der Anwender den Benchmark startet, beginnt ein zweiter Thread, der die zu sortierenden Elemente erzeugt und dann entweder einen weiteren Thread oder zwei weitere Threads startet, um diese Elemente zu sortieren. Mit dieser Vorgehensweise wird erreicht, dass der UI-Thread ungestört weiter operieren kann, ohne durch den eigentlichen Arbeits-Thread angehalten zu werden.

'Dieser Namensbereich enthält die benötigten Threading-Objekte.

```
Imports System.Threading
Imports System.IO
```

```
Public Class frmMain
    Inherits System.Windows.Forms.Form
```

```
#Region " Vom Windows Form Designer generierter Code "
    'Aus Platzgründen ausgelassen
#End Region
```

```
    Private Const cAmountOfElements As Integer = 75000
    Private Const cCharsPerString As Integer = 100
    Private Const cShowResults As Boolean = False
```

Die folgenden Delegationen werden benötigt, um bestimmte Eigenschaften der Steuerelemente des Formulars thread-sicher verändern zu können. Das gilt für das Setzen der Text-Eigenschaft des TextBox-Steuerelements, das für die Darstellung der Kommentare zuständig ist, auf der einen und für das Wiedereinschalten der Schaltflächen auf der anderen Seite.

```

'Delegaten für das Aufrufen von Invoke, da Controls nicht
'thread-sicher sind.
'Für die TextBox zur Ausgabe
Private Delegate Sub AddTBTextTSActuallyDelegate(ByVal txt As String)
'Für das Einschalten der Buttons, wenn der Vorgang beendet ist.
Private Delegate Sub TSEnableControlsDelegate()

'Flag, das bestimmt, wann alle Threads zu gehen haben.
Private myAbortAllThreads As Boolean

'Ausgabe-Textbox. Statisch deswegen, damit jeder Thread
'zu jeder Zeit darauf zugreifen kann.
Private Shared myAusgabeTextBox As TextBox

'Synchronisation für die Ausgabe-Textbox
Private Shared myAutoResetEvent As AutoResetEvent

'Merkt sich thread-safe, ob die Sortierung mit einem oder
'zwei Threads durchgeführt werden soll.
Private myUseTwoThread As Boolean

'Ereignis, um dem UI-Thread mitteilen zu können,
'dass der Hauptarbeits-Thread abgeschlossen ist.
Private Event MainThreadFinished()

Public Sub New()
    MyBase.New()

    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
    InitializeComponent()

    'Synchronisation für die Ausgabe-Textbox
    myAutoResetEvent = New AutoResetEvent(True)

End Sub

```

Wichtig bei der Verwendung der folgenden Funktion ist die Synchronisation mit allen Programmprozeduren, die sie verwenden, denn: Sollte das Formular genau in dem Moment geschlossen werden, in dem ein Thread genau diese Routine erreicht hat, dann würde Invoke auf ein Steuerelement zugreifen, das zu dieser Zeit nicht mehr existierte. Eine Ausnahme wäre die Folge. Aus diesem Grund wird das Schließen des Formulars mit der Ausgabe in der TextBox über ein AutoResetEvent-Objekt synchronisiert. OnClosing fängt das Schließen-Ereignis ab und wartet, bis eine mögliche, gerade laufende Ausgabe in der TextBox abgeschlossen ist.

```

'Dient zum Setzen einer Eigenschaft auf der TextBox indirekt über Invoke.
Public Shared Sub AddTBText(ByVal txt As String)
    'Formular nicht mehr oder noch nicht da...
    If AusgabeTextBox Is Nothing Then
        '...und tschö!
        Exit Sub
    End If
    'Threads synchronisieren: Hier beginnt kritischer Abschnitt.
    myAutoResetEvent.Reset()

```

```

Dim locDel As New AddTBTextTSActuallyDelegate(AddressOf AddTBTextTSActually)
AusgabeTextBox.Invoke(locDel, New Object() {txt})
myAutoResetEvent.Set()
End Sub

```

```

'Diese Routine wird indirekt über Invoke aufgerufen, und ist damit UI-Thread.
Private Shared Sub AddTBTextTSActually(ByVal txt As String)
    AusgabeTextBox.Text += txt
    AusgabeTextBox.SelectionStart = AusgabeTextBox.Text.Length - 1
    AusgabeTextBox.ScrollToCaret()
End Sub

```

```

'Liefert die Textbox als Eigenschaft.
Private Shared ReadOnly Property AusgabeTextBox() As TextBox
    Get
        Return myAusgabeTextBox
    End Get
End Property

```

Die folgende Routine kümmert sich um die notwendigen Schritte, wenn der Anwender die Schaltfläche *Benchmark starten* angeklickt hat. Sie startet dann den Arbeits-Thread (den Haupt-Thread), der zunächst alle zu sortierenden Elemente erzeugt und sie dann anschließend durch die Sortier-Threads ordnen lässt.

```

Private Sub btnThreadStarten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnBenchmarkStarten.Click
    'Dieses Objekt kapselt den eigentlichen Thread.
    Dim locThread As Thread
    'Dieses Objekt benötigen Sie, um die Prozedur zu bestimmen,
    'die den Hauptthread ausführt.
    Dim locThreadStart As ThreadStart

    'Ausgabetextbox bestimmen.
    myAusgabeTextBox = Me.txtAusgabe

```

Damit der Haupt-Thread kein zweites Mal gestartet werden kann, schaltet die Routine die Schaltflächen an dieser Stelle aus. Wenn der komplette Testparcours abgeschlossen ist, löst der Haupt-Thread ein Ereignis aus, das vom Formular eingebunden wird. Die Behandlungsprozedur dieses Ereignisses schaltet die Schaltflächen dann indirekt über Invoke wieder ein. Diese Vorgehensweise dient nur der Demonstration: Genauso gut könnte der Haupt-Thread selbst die Schaltflächen wieder einschalten, wenn alle Operationen abgeschlossen sind.

```

    btnBenchmarkStarten.Enabled = False
    btnBeenden.Enabled = False

    'Festhalten, ob die Sortierung in einem oder zwei Threads erfolgen soll.
    myUseTwoThread = Me.chkMultithreading.Checked

    'Thread ausführende Prozedur bestimmen.
    locThreadStart = New ThreadStart(AddressOf StartMainThread)
    'ThreadStart-Objekt dem Thread-Objekt übergeben
    locThread = New Thread(locThreadStart)
    'Thread-Namen bestimmen.
    locThread.Name = "MainThread"

```

```

'Haupt-Thread starten.
locThread.Start()

'Der UI-Thread (die Nachrichtenwarteschlange) läuft jetzt leer nebenbei.
'Auf diese Weise kann der komplette Testparcours ruhig 100% Prozessorleistung
'verschlingen; da er in einem Extra-Thread läuft, bleibt das Programm
'dennoch bedienbar.
End Sub

```

Dies ist die eigentliche Haupt-Thread-Routine (aber nicht der UI-Thread!), die das Array mit den zu sortierenden Elementen generiert und dann selbst wiederum entweder ein oder zwei Arbeits-Threads aufruft, die die eigentliche Sortierung durchführen.

```

Private Sub StartMainThread()

    Dim locStrings(cAmountOfElements - 1) As String
    Dim locGauge As New HighSpeedTimeGauge
    Dim locAutoResetEvents(1) As AutoResetEvent

    Dim locRandom As New Random(DateTime.Now.Millisecond)
    Dim locChars(cCharsPerString) As Char

    'Damit die Controls nach Abschluss des Sortierens auf dem Formular
    'wieder eingeschaltet werden können, muss der Haupt-Thread das Ende
    'der Sortierung mitbekommen. Deswegen: Ereignis
    AddHandler MainThreadFinished, AddressOf MainThreadFinishedHandler

    'String-Array erzeugen; hatten wir schon zig Mal...
    AddTBText("Erzeugen von " + cAmountOfElements.ToString + " Strings..." + vbNewLine)
    locGauge.Start()
    For locOutCount As Integer = 0 To cAmountOfElements - 1
        For locInCount As Integer = 0 To cCharsPerString - 1
            If myAbortAllThreads Then
                RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
                Exit Sub
            End If
            Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
            If locIntTemp > 26 Then
                locIntTemp += 97 - 26
            Else
                locIntTemp += 65
            End If
            locChars(locInCount) = Convert.ToChar(locIntTemp)
        Next
        locStrings(locOutCount) = New String(locChars)
    Next
    locGauge.Stop()
    AddTBText("...in " + locGauge.DurationInMilliseconds.ToString + " Millisekunden" + vbNewLine)
    AddTBText(vbNewLine)
    locGauge.Reset()

```

Die Member-Variable `myUseTwoThread` wird durch den Ereignis-Handler des CheckBox-Steuerelements gesetzt. Sie bestimmt, ob der Haupt-Thread das Sortieren mit zwei oder nur einem Thread starten soll.

```

If Not myUseTwoThread Then
    'Messung starten - hier wird mit nur einem Thread sortiert.
    locGauge.Start()
    Dim locFirstSortThread As New SortArrayThreaded(New ArrayList(locStrings), "1. SortThread")

    AddTBText("Sortieren von " + cAmountOfElements.ToString + " Strings mit einem Thread..." + _
        vbNewLine)
    locAutoResetEvents(0) = locFirstSortThread.AutoResetEvent
    locFirstSortThread.StartSortingAsynchron()

```

Die folgende Do-While-Schleife wartet, bis der Sortier-Thread abgeschlossen ist. Der Sortier-Thread, der durch die SortArrayThreaded-Klasse gekapselt wird, stellt eine AutoResetEvents-Eigenschaft bereit, die signalisiert wird, wenn der Sortierungsvorgang abgeschlossen ist.

TIPP: Wenn Sie eigene Klassen implementieren, die Aufgaben kapseln, welche in Threads ausgeführt werden, sollten Sie ebenfalls dafür sorgen, dass der Abschluss einer Aufgabe nicht nur durch ein Ereignis, sondern auch mit einem Synchronisierungs-Objekt wie beispielsweise der Mutex- oder der AutoResetEvent-Klasse signalisiert wird.

```

Do While Not locAutoResetEvents(0).WaitOne(1, True)
    'Mit Timeout-Wert warten, damit ein Eingreifen (Abbrechen) im Sortier-Thread möglich
    'bleibt, wenn das komplette Programm beendet werden soll.
    If myAbortAllThreads = True Then
        'Die Dispose-Methode der SortArrayThread-Klasse bricht einen vielleicht laufenden
        'Sortiert-Thread ab.
        locFirstSortThread.Dispose()
        RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
        Exit Sub
    End If
Loop
'Messung beenden.
locGauge.Stop()
AddTBText("...in " + locGauge.DurationInMilliseconds.ToString + " Millisekunden" + vbNewLine)
AddTBText(vbNewLine)
'Array-List zurück-casten.
locStrings = CType(locFirstSortThread.ArrayList.ToArray(GetType(String)), String())
Else
    'Es soll mit zwei Threads sortiert werden.
    locGauge.Start()
    'Zwei ArrayLists erstellen, die jeweils den oberen und den unteren Teil der
    'zu sortierenden Gesamtliste beinhalten.
    Dim locFirstAL As New ArrayList(locStrings.Length \ 2 + 1)
    Dim locSecondAL As New ArrayList(locStrings.Length \ 2 + 1)
    Dim locMitte As Integer = locStrings.Length \ 2

    'Elemente über beide ArrayLists verteilen
    For c As Integer = 0 To locMitte - 1
        locFirstAL.Add(locStrings(c))
        locSecondAL.Add(locStrings(c + locMitte))
    Next

```

```

'Zwei SortArrayThreaded-Instanzen erzeugen und ihnen die Teillisten übergeben.
Dim locFirstSortThread As New SortArrayThreaded(locFirstAL, "1. SortThread")
Dim locSecondSortThread As New SortArrayThreaded(locSecondAL, "2. SortThread")

AddTBText("Sortieren von " + cAmountOfElements.ToString + " Strings mit zwei Threads..." + _
vbNewLine)
'Beide AutoResetEvent-Objekte der beiden Sortier-Threads in ein Array packen,
'damit auf den Abschluss beider Threads gewartet werden kann.
locAutoResetEvents(0) = locFirstSortThread.AutoResetEvent
locAutoResetEvents(1) = locSecondSortThread.AutoResetEvent
'Los geht's!
locFirstSortThread.StartSortingAsynchron()
locSecondSortThread.StartSortingAsynchron()
'Warten, bis beide Sortier-Threads beendet sind; dabei die Erkennung eines
'möglichen Programmabbruchs offen halten.
Do While Not AutoResetEvent.WaitAll(locAutoResetEvents, 1, True)
    If myAbortAllThreads = True Then
        locFirstSortThread.Dispose()
        locSecondSortThread.Dispose()
        RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
        Exit Sub
    End If
Loop

'Beide sortierten String-Arrays wieder zusammenführen.
Dim locIndexOnSecond As Integer
Dim locTempArray As New ArrayList(cAmountOfElements)
For locIndexOnFirst As Integer = 0 To locFirstAL.Count - 1
    Do
        If locIndexOnSecond < locSecondAL.Count Then
            If CType(locFirstAL(locIndexOnFirst), IComparable).CompareTo( _
                CType(locSecondAL(locIndexOnSecond), IComparable)) < 0 Then
                locTempArray.Add(locFirstAL(locIndexOnFirst))
                Exit Do
            Else
                locTempArray.Add(locSecondAL(locIndexOnSecond))
                locIndexOnSecond += 1
            End If
        Else
            locTempArray.Add(locFirstAL(locIndexOnFirst))
            Exit Do
        End If
    Loop
    'Möglichen Abbruch auch an dieser Stelle berücksichtigen.
    If myAbortAllThreads = True Then
        locFirstSortThread.Dispose()
        locSecondSortThread.Dispose()
        RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
        Exit Sub
    End If
Next

```

```

'Falls es Reste aus dem zweiten Array gibt, diese auch noch kopieren.
If locIndexOnSecond < (locSecondAL.Count - 1) Then
    For c As Integer = locIndexOnSecond To locSecondAL.Count - 1
        locTempArray.Add(locSecondAL(c))
    Next
End If
locStrings = CType(locTempArray.ToArray(GetType(String)), String())

locGauge.Stop()
AddTBText("...in " + locGauge.DurationInMilliseconds.ToString + " Millisekunden" + vbNewLine)
AddTBText(vbNewLine)
End If

'Falls das entsprechende Flag gesetzt ist, die Ergebnisse in die Textbox schreiben.
If cShowResults Then
    For Each s As String In locStrings
        If myAbortAllThreads Then
            RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler
            Exit Sub
        End If
        AddTBText(s + vbNewLine)
        AddTBText(vbNewLine)
    Next
End If

'Ereignis auslösen, das die Schaltflächen wieder einschaltet.
RaiseEvent MainThreadFinished()
'Ereignishandler wieder entfernen.
RemoveHandler MainThreadFinished, AddressOf MainThreadFinishedHandler

End Sub

```

WICHTIG: Wenn ein Thread einen Ereignishandler über RaiseEvent aufruft, gehört dieser Ereignishandler immer zum Ereignis auslösenden Thread, ganz egal, welcher Klasse der Ereignis-Handler zugeordnet ist. Deshalb gilt auch für eine Routine, die durch ein Thread-Ereignis aufgerufen wurde: Steuerelemente, die hier manipuliert werden, können nur über Invoke manipuliert werden, damit sie innerhalb des UI-Threads verändert werden.

Diese Tatsache macht die Thread-Programmierung von Windows-Benutzeroberflächen vergleichsweise aufwändig. Abhilfe schafft hier die BackgroundWorker-Komponente, die im ► Abschnitt »Threads durch den Background-Worker initiieren« ab Seite 978 besprochen wird.

```

Private Sub MainThreadFinishedHandler()
    Console.WriteLine("MainThread beendet: " + Thread.CurrentThread.Name)
    Me.Invoke(New TSEnableControlsDelegate(AddressOf TSEnableControlsActually))
End Sub

Private Sub TSEnableControlsActually()
    btnBeenden.Enabled = True
    btnBenchmarkStarten.Enabled = True
End Sub

```

```

Private Sub btnBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnBeenden.Click
    'Nur schließen. OnClosing synchronisiert.
    Me.Close()
End Sub
Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    'Falls eine TextBox-Ausgabe läuft, warten, bis diese abgeschlossen ist.
    'Sonst erfolgt die Ausgabe womöglich gerade zu der Zeit, wenn der Anwender
    'das Formular geschlossen hat. Dann wird die TextBox allerdings entladen,
    'ihr Handle zerstört, und die Ausgabe löst eine Ausnahme aus, weil sie das
    'zerstörte Handle der TextBox verwenden will.
    Do While Not myAutoResetEvent.WaitOne(1, True)
        Application.DoEvents()
    Loop
    'Ausgabe sicher --> jetzt erst abbrechen.
    myAbortAllThreads = True
End Sub

```

End Class

TIPP: Denkbar für die Lösung dieses Beispiels ist auch noch eine andere Vorgehensweise, die obendrein noch effektiver ist: Der Haupt-Thread könnte auf die `AutoResetEvent`-Objekte zum Warten auf das Beenden des Sortiervorgangs komplett verzichten und sich stattdessen selbst mit `Suspend` aussetzen, nachdem er das Sortieren angeworfen hat. Die Sortierrountinen könnten ihrerseits durch das Auslösen des »Fertig!«-Ereignisses einen Ereignis-Handler aufrufen, der den Haupt-Thread wieder in Gang und schließlich zum Abschluss bringt. Das Beispiel des folgenden Abschnittes, dessen Hauptaufgabe eigentlich die Demonstration des Thread-Pools ist, verdeutlicht diese Vorgehensweise nebenbei.

Verwenden des Thread-Pools

In den bisherigen Beispielen haben Sie Threads als Instrument kennen gelernt, die nur dann einmalig hervorgeholt werden, wenn man sie braucht. In Anwendungen, die unter Praxisbedingungen arbeiten, sieht das oft anders aus: Sicherlich lässt sich voraussagen, welche Threads in Anwendungen häufiger und welche weniger häufig benötigt werden; doch in jedem Fall macht es die Praxis notwendig – im Gegensatz zu den bislang gezeigten Beispielen – dass Thread-Objekte ständig erzeugt, verwendet, ausgesetzt und wieder zerstört werden; unter Umständen passiert das Hunderte Male in einer komplexen Applikation.

Das Erstellen eines Threads gehört für das Betriebssystem mit zu den aufwändigeren Dingen, die es zu erfüllen hat. Schon das Erstellen eines Threads kostet, gerade wenn es für ständig sich wiederholende, kleinere Aufgaben zigfach wiederholt werden muss, wertvolle Ressourcen. Aus diesem Grund bedient man sich eines Tricks, wenn innerhalb einer Anwendung viele kleine Threads eingesetzt werden sollen. Threads werden nicht ständig neu, sondern innerhalb eines so genannten Thread-Pools definiert. Wenn eine Anwendung einen Thread aus dem Thread-Pool anfordert, dann wird dieser beim ersten Mal zwar auch neu erstellt, doch anschließend nicht sofort wieder gelöscht, wenn die Anwendung ihn nicht mehr benötigt.

Stattdessen bleibt seine Grunddefinition im Pool erhalten, und das Thread-Objekt muss nicht komplett neu erstellt werden, wenn die Anwendung es für eine andere Aufgabe benötigt. Sie muss lediglich die Adresse der neuen Arbeits-Thread-Prozedur angeben; ansonsten kann das Thread-Objekt des Thread-Pools weiterverwendet werden.

Das Anlegen eines Threads aus dem Thread-Pool gestaltet sich eigentlich noch simpler als das Anlegen eines Threads mit der herkömmlichen Thread-Klasse. Sie übergeben der statischen Methode `QueueUserWorkItem` lediglich die Adresse der Prozedur, die als Thread ausgeführt werden soll. Im Gegensatz zum normalen Thread können Sie der Thread-Routine sogar eine Variable vom Typ `Object` als Parameter mitgeben (optional, Sie müssen also nicht).

HINWEIS: Threads, die über den Thread-Pool erstellt werden, laufen als so genannte Hintergrund-Threads. Diese Threads haben eine besondere Eigenschaft: Wenn der Haupt- bzw. UI-Thread der Applikation endet, werden auch die Hintergrund-Threads automatisch beendet.

Zusammenfassend gesagt, haben also Thread-Pool-Threads folgende Vorteile:

- Sie benötigen keinen zusätzlichen Programmieraufwand, um sie vorzeitig beenden zu können, da sie als Hintergrund-Thread eingerichtet werden.
- Sie benötigen kein zusätzliches `Delegate`-Objekt (Startobjekt), um gestartet zu werden.
- Sie werden in einem Rutsch definiert und gestartet.
- Ihre benötigten Ressourcen lassen sich schneller reservieren, da intern die Thread-Instanz nicht gelöscht sondern nur »ruhiggestellt« wird und damit ohne Aufwand für andere Zwecke wiederverwendet werden kann.
- Der Arbeits-Thread-Routine kann einen Parameter vom Typ `Object` empfangen.

Allerdings gibt es auch Nachteile: Maximal stehen pro verfügbaren Prozessor 25 Threads im Thread-Pool zur Verfügung. Wenn Sie einen neuen Thread darüber hinaus anfordern, wartet die dafür zuständige Methode `QueueUserWorkItem` solange, bis ein zurzeit noch reserviertes Thread-Objekt des Pools wieder verfügbar ist.

BEGLEITDATEIEN: Das folgende Beispiel demonstriert das Sortieren mit Threads aus dem Thread-Pool. Sie finden das Projekt dieses Beispiels im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap31\ThreadPoolThreads\`. Da es weitestgehend dem vorherigen Beispiel entspricht, zeigt das folgende Codelisting nur die geänderten Stellen.

Betrachten wir zunächst das Codelisting des Sortier-Threads, der nunmehr auf einem Threadpool-Thread beruht (die eigentliche Sortieroutine ist in dieser Version aus Platzgründen ausgelassen).

```
Public Class SortArrayThreaded
```

```
    Public Event SortCompleted(ByVal sender As Object, ByVal e As SortCompletedEventArgs)
    Private myArrayToSort As ArrayList
    Private myTerminateThread As Boolean
    Private mySortingComplete As Boolean
    Private myThreadName As String
```

```

'Konstruktor übernimmt die zu sortierende Liste. Der ThreadName hat nur
'dokumentarischen Charakter.
Sub New(ByVal ArrayToSort As ArrayList, ByVal ThreadName As String)
    myArrayToSort = ArrayToSort
    myThreadName = ThreadName
End Sub

'Thread starten
Public Sub StartSortingAsynchron()
    ThreadPool.QueueUserWorkItem(AddressOf SortArray)
End Sub

Private Sub SortArray(ByVal state As Object)
.
. 'Wie gehbat.
.
End Function

Public ReadOnly Property ArrayList() As ArrayList
    Get
        Return myArrayToSort
    End Get
End Property
End Class

```

Damit eine den Thread in Gang setzende Instanz weiß, welcher Thread beendet wurde, wenn mehrere Sortierungen gestartet werden, sind die Ereignisparameter in dieser Version leicht erweitert worden und geben nunmehr auch den Thread-Namen zurück:

```

'Dient nur der Ereignisübergabe, wenn Sie das SortCompleted-Ereignis
'nach Abbruch oder Abschluss des Sortierens auslösen wollen.
Public Class SortCompletedEventArgs
    Inherits EventArgs

    Private mySortCompletionStatus As SortCompletionStatus
    Private myThreadName As String

    Sub New(ByVal scs As SortCompletionStatus, ByVal ThreadName As String)
        mySortCompletionStatus = scs
    End Sub

    Public Property ThreadName() As String
        Get
            Return myThreadName
        End Get
        Set(ByVal Value As String)
            myThreadName = Value
        End Set
    End Property

```

```

Public Property SortCompletionStatus() As SortCompletionStatus
    Get
        Return mySortCompletionStatus
    End Get
    Set(ByVal Value As SortCompletionStatus)
        mySortCompletionStatus = Value
    End Set
End Property

```

End Class

Die wesentlichen Unterschiede im eigentlichen Arbeits-Thread beschränken sich auf das Warten des Sortierungsabschlusses. Während im vorherigen Beispiel eine Warteschleife einiges an Rechenzeit gekostet hat, wird der Arbeits-Thread hier komplett ausgesetzt. Erst die Ereignisbehandlungsroutine des SortCompleted-Ereignisses setzt den Arbeits-Thread wieder in Gang, der seine Aufgabe dann zu Ende führen kann:

```

'Dies ist die eigentliche Haupt-Thread-Routine (aber nicht der UI-Thread!), die das Testarray
'generiert und dann selbst wiederum entweder ein oder zwei Arbeits-Threads aufruft,
'die die eigentliche Sortierung durchführen.
Private Sub StartMainThread()

    Dim locStrings(cAmountOfElements - 1) As String
    Dim locGauge As New HighSpeedTimeGauge

    Dim locRandom As New Random(DateTime.Now.Millisecond)
    Dim locChars(cCharsPerString) As Char

    'Damit die Controls nach Abschluss des Sortierens auf dem Formular
    'wieder eingeschaltet werden können, muss der Haupt-Thread das Ende
    'der Sortierung mitbekommen. Deswegen: Ereignis
    AddHandler MainThreadFinished, AddressOf MainThreadFinishedHandler

    'String-Array erzeugen; hatten wir schon zig Mal...
    AddTBText("Erzeugen von " + cAmountOfElements.ToString + " Strings..." + vbNewLine)
    locGauge.Start()
    For locOutCount As Integer = 0 To cAmountOfElements - 1
        For locInCount As Integer = 0 To cCharsPerString - 1
            Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
            If locIntTemp > 26 Then
                locIntTemp += 97 - 26
            Else
                locIntTemp += 65
            End If
            locChars(locInCount) = Convert.ToChar(locIntTemp)
        Next
        locStrings(locOutCount) = New String(locChars)
    Next
    locGauge.Stop()
    AddTBText("...in " + locGauge.DurationInMilliseconds.ToString + " Millisekunden" + vbNewLine)
    AddTBText(vbNewLine)
    locGauge.Reset()

```

```

If Not myUseTwoThread Then
    'Messung starten - hier wird mit nur einem Thread sortiert.
    locGauge.Start()
    Dim locFirstSortThread As New SortArrayThreaded(New ArrayList(locStrings), "1. SortThread")

    AddTBText("Sortieren von " + cAmountOfElements.ToString + " Strings mit einem Thread..." _
        + vbNewLine)

    'Handler hinzufügen, der das Ende des Sortierens mitbekommt und dann wiederum
'mySortCompletion hochzählt, damit die Warteschleife des Hauptthreads beendet werden kann
AddHandler locFirstSortThread.SortCompleted, AddressOf SortCompletionHandler

    'los geht's
    locFirstSortThread.StartSortingAsynchron()

    'Warten, bis der Sortier-Thread abgeschlossen ist.
'mySortCompletion wird durch das SortCompletion-Ereignis hochgezählt
'wenn es 1 erreicht hat, schmeißt der Ereignishandler diesen Thread wieder an.
mySyncUIResetEvent.WaitOne()

    'Handler wieder entfernen
    RemoveHandler locFirstSortThread.SortCompleted, AddressOf SortCompletionHandler

    'Messung beenden
    locGauge.Stop()
    AddTBText("...in " + locGauge.DurationInMilliseconds.ToString + " Millisekunden" + vbNewLine)
    AddTBText(vbNewLine)
    'Array-List zurück-casten.
    locStrings = CType(locFirstSortThread.ArrayList.ToArray(GetType(String)), String())
Else
    'Es soll mit zwei Threads sortiert werden
    .
    .
    .
End Sub

```

Der Thread hält hier, nachdem er die Ereignisbehandlungsroutine eingerichtet und den Sortiervorgang gestartet hat, komplett an (siehe fett markierte Zeilen im Listing). Durch die folgende Ereignisroutine wird er wieder zum Leben erweckt, wenn der Sortiervorgang abgeschlossen wurde:

```

Private Sub SortCompletionHandler(ByVal sender As Object, ByVal e As SortCompletedEventArgs)
    mySortCompletion += 1
    If myUseTwoThread Then
        If mySortCompletion = 2 Then
            mySyncUIResetEvent.Set()
        End If
    Else
        If mySortCompletion = 1 Then
            mySyncUIResetEvent.Set()
        End If
    End If
End Sub

```

Thread-sichere Formulare in Klassen kapseln

Steuerelemente können von Threads aus nur durch deren Invoke thread-sicher bearbeitet werden – die vergangenen Abschnitte haben das in aller Ausführlichkeit gezeigt. Wenn Sie Ihre Formulare thread-sicher gestalten wollen, dann ist es das Beste, wenn Sie nach Möglichkeit die dazugehörigen Threads in den Formular-Klassen kapseln – etwa, wie es der ► Abschnitt »Datenaustausch zwischen Threads durch Kapseln von Threads in Klassen« ab Seite 959 gezeigt hat. Auch dabei gilt natürlich, dass Sie die Steuerelemente eines solchen Formulars aus seinen Arbeits-Threads heraus nur über Invoke verarbeiten dürfen.

Für einige Anwendungen kann es jedoch sinnvoll sein, dass ein Thread selbst ein Formular erstellt und es unter seine Verwaltung stellt. Denken Sie an die Beispiele, die Sie schon kennen gelernt haben: Einige der ersten Testprogramme haben auf eine `ADThreadSafeInfoBox`-Klasse zurückgegriffen, die die thread-sichere Ausgabe von Text in einem Formular erlaubte, obwohl an keiner Stelle die Instanzierung einer auf *Form* basierten Klasse erfolgte.

Wenn ein Thread völlig unabhängig vom dem ihn einbindenden Programm ein Formular erstellen will, hat er nur eine Chance: Er muss für das Formular (und alle weiteren, die dieses Formular aufruft), eine eigene Warteschlange implementieren.

Wenn Sie also eine komplette Klasse schaffen wollen, die einen Arbeits-Thread beinhaltet, der selbst auf ein Formular zugreifen muss, dann sind folgende Schritte nötig:

- Die den Thread startende Prozedur muss zunächst einen neuen Thread erstellen, der zum zusätzlichen UI-Thread wird.
- Dieser neue Thread instanziiert dann das Hauptformular des neuen UI-Threads und bindet das Ereignis `Application.ThreadExit` ein.
- Mit `Application.Run` kann er nun im neu geschaffenen Thread eine Nachrichtenwarteschlange erstellen und die Instanz des Formulars an diese binden. Der Thread ist dann automatisch beendet, wenn der Anwender (oder eine andere Instanz) das gebundene Formular auf irgendeine Weise schließt.
- Für den Fall, dass die umgebende Applikation zuvor geschlossen wurde, löst das `Application`-Objekt das `ThreadExit`-Ereignis aus. Die Klasse muss in der Ereignisbehandlungsroutine dieses Ereignisses ein paar Aufräumarbeiten durchführen: Durch das eigentliche Programmende ist der zweite UI-Thread nämlich noch nicht ebenfalls automatisch abgeschlossen. Ein explizites `Application.ExitThread` wird an dieser Stelle noch notwendig, um den zweiten UI-Thread spätestens jetzt zu beenden.

Das folgende Beispiel geht sogar noch einen Schritt weiter, um absolute Unabhängigkeit zu erlangen: Es erstellt den zweiten UI-Thread im statischen Konstruktor der Klasse. Wenn eine andere Instanz die statische Funktion `TSWrite` (oder `TSWriteLine`) das erste Mal aufruft, um eine Ausgabe in das Statusfenster durchzuführen, sorgt der statische Konstruktor dafür, dass der neue UI-Thread erstellt wird. Die Ausgabe erfolgt dann anschließend in das nun vorhandene Fenster (durch ein einfaches `TextBox`-Steuerelement simuliert). Damit der Konstruktor dem neuen UI-Thread ein wenig Zeit gibt,

in Gang zu kommen und das Formular zu instanzieren, und damit verhindert wird, dass TSWrite[Line] eine Ausgabe in ein Steuerelement durchführt, das noch nicht existiert, erfolgt die notwendige Synchronisierung mit einem ManualResetEvent-Objekt.

```
Imports System.Threading
```

```
Public Class ADThreadSafeInfoBox
    Private Shared myText As String
    Private Shared myInstance As ADThreadSafeInfoBox
    Private Shared myThread As Thread
    Private Shared myManualResetEvent As ManualResetEvent

    Private Delegate Sub TSWriteActallyDelegate()
    Private Delegate Sub ThisInstanceCloseDelegate()

    Private Shared myThisInstanceCloseDelegate As ThisInstanceCloseDelegate

    Private Shared Sub ThisInstanceCloseActually()
        myInstance.Dispose()
    End Sub

    'Statischer Konstruktor erstellt neuen UI-Thread
    Shared Sub New()
        myText = ""
        myManualResetEvent = New ManualResetEvent(False)
        CreateInstanceThroughThread()
    End Sub

    'Teil des Konstruktors; könnte theoretisch auch von
    'anderswo in Gang gesetzt werden. Diese Routine erstellt
    'den neuen UI-Thread und startet ihn...
    Private Shared Sub CreateInstanceThroughThread()
        myThread = New Thread(New ThreadStart(AddressOf CreateInstanceThroughThreadActually))
        myThread.Name = "2. UI-Thread"
        myThread.Start()
    End Sub

    '...und der neue UI-Thread erstellt jetzt das Formular
    'und bindet es an eine neue Nachrichtenwarteschlange.
    Private Shared Sub CreateInstanceThroughThreadActually()
        'Instanz des Formulars erstellen
        myInstance = New ADThreadSafeInfoBox
        'Wir müssen auf jeden Fall wissen, wann die Hauptapplikation (der Haupt-UI-Thread) geht.
        myThisInstanceCloseDelegate = New ThisInstanceCloseDelegate(AddressOf ThisInstanceCloseActually)

        AddHandler Application.ThreadExit, AddressOf ThreadExitHandler
        'Und wir müssen wissen, ab wann das Formular wirklich existiert;
        'vorher dürfen keine Ausgaben ins Formular erfolgen
        AddHandler myInstance.HandleCreated, AddressOf HandleCreatedHandler
        'und ab wann nicht mehr. Für dieses Beispiel ist dieses Ereignis nicht soooo wichtig.
        AddHandler myInstance.HandleDestroyed, AddressOf HandleDestroyedHandler
    End Sub
End Class
```

```

        'Hier wird die Warteschlange gestartet
        Application.Run(myInstance)
    End Sub

    'Dieser Ereignis-Handler wird aufgerufen, wenn das Hauptprogramm beendet wird.
    'Der zweite UI-Thread wird damit beendet.
    Private Shared Sub ThreadExitHandler(ByVal sender As Object, ByVal e As EventArgs)
        Console.WriteLine("ThreadExit")
        'Keine TextBox mehr vorhanden:
        ' Synchronisationsvoraussetzung für TSWrite schaffen
        myManualResetEvent.Reset()
        Try
            myInstance.Invoke(myThisInstanceCloseDelegate)
        Catch ex As Exception
        End Try
    End Sub

    'TSWrite signalisieren, dass die Ausgabe in die TextBox jetzt sicher ist,
    'da das Formular-Handle erstellt wurde.
    Private Shared Sub HandleCreatedHandler(ByVal sender As Object, ByVal e As EventArgs)
        Console.WriteLine("HandleCreated")
        myManualResetEvent.Set()
    End Sub

    'Vielleicht später mal wichtig; hier nur zur Demo.
    Private Shared Sub HandleDestroyedHandler(ByVal sender As Object, ByVal e As EventArgs)
        'Nur für Testzwecke
        Console.WriteLine("HandleDestroyed")
    End Sub

    'Nutzt TSWrite; siehe dort.
    Public Shared Sub TSWriteLine(ByVal Message As String)
        SyncLock (GetType(ADThreadSafeInfoBox))
            Message += vbNewLine
            TSWrite(Message)
        End SyncLock
    End Sub

    'Ausgabe ohne neue Zeile
    Public Shared Sub TSWrite(ByVal Message As String)
        SyncLock (GetType(ADThreadSafeInfoBox))
            'Synchronisierung: Wenn nach 50 Millisekunden
            'keine TextBox vorhanden ist --> Befehl ignorieren
            If Not myManualResetEvent.WaitOne(50, True) Then
                Exit Sub
            End If
            myText += Message
        Try
            myInstance.Invoke(New TSWriteActuallyDelegate(AddressOf TSWriteActually))
        Catch ex As Exception
        End Try
        End SyncLock
    End Sub

```

```

'Thread-sichere Ausgabe in die TextBox mit Invoke
Private Shared Sub TWriteActually()
    myInstance.txtOutput.Text = myText
    myInstance.txtOutput.SelectionStart = myText.Length - 1
    myInstance.txtOutput.ScrollToCaret()
End Sub
End Class

```

Threads durch den Background-Worker initiieren

Neu im .NET-Framework 2.0 ist die so genannte `BackgroundWorker`-Komponente, die die meiner Meinung nach einfachste Methode darstellt, eine rechenintensive Methode in einem anderen Thread laufen zu lassen.

Falls Sie dieses Kapitel aufmerksam studiert haben, werden Sie feststellen, dass der Einsatz dieser Komponente zwar nicht so viel Flexibilität bietet, wie die Programmierung von Threads über das `Thread`-Objekt; aber durch eine besondere intern angewandte Technik hat sie einen unschlagbaren Vorteil: Sie kann Statusmeldungen über Ereignisse zur Verfügung stellen, die im Ursprungs-Thread (in der Regel also dem UI-Thread) laufen; der Aufwand, sich `Invoke` bedienen zu müssen, um beispielsweise ein `Label`-Steuerelement zu aktualisieren, fällt also weg.

Die Vorgehensweise, um mithilfe der `BackgroundWorker`-Komponente eine Prozedur als neuen Thread laufen zu lassen, ist außergewöhnlich aber simple: Ihr Thread läuft als Ereignisbehandlungsprozedur.

- Sie triggern die `BackgroundWorker`-Komponente mit der Methode `RunWorkerAsync`. Die wiederum sorgt dafür, dass das `DoWork`-Ereignis ausgelöst wird, und ihre implementierte Behandlungsroutine, die dieses Ereignis behandelt, *ist* die Prozedur, die auf einem anderen Thread läuft.
- Möchten Sie, dass Fortschrittsinformationen aus ihrer Thread-Routine heraus gesendet werden, sorgen Sie vor Beginn des Threads, dass die `WorkerReportsProgress`-Eigenschaft auf `True` gesetzt ist, denn nur dann unterstützt die `BackgroundWorker`-Komponente das Berichten von Fortschrittsstatus. Auch die Kommunikation des Fortschritts funktioniert über Ereignisse. Wenn die entsprechenden Vorbereitungen getroffen wurden, lösen Sie zu gegebener Zeit innerhalb Ihres Threads (also dem `DoWork`-Ereignisbehandler) das `ProgressChanged`-Ereignis aus, das Sie mit der `ReportProgress`-Methode initiieren.

WICHTIG: Dieses Ereignis läuft dann, anders als zu erwarten wäre, auf dem Thread, der die `BackgroundWorker`-Komponente kapselt (in der Regel also dem UI-Thread), und nicht auf dem Thread, der das `ProgressChanged`-Ereignis ausgelöst hat. UI-Komponenten können deswegen direkt in diesem Ereignis angesprochen werden!

- Wenn der `DoWork`-Thread beendet wurde, löst die `BackgroundWorker`-Komponente automatisch das `RunWorkerCompleted`-Ereignis aus. Eine Prozedur, die dieses Ereignis behandelt, läuft auch wieder auf dem ursprünglichen Thread (also in der Regel dem UI-Thread) und nicht auf dem Thread, der durch `DoWork` initiiert wurde.

BEGLEITDATEIEN: Das Projekt für das folgende Beispiel finden Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap31\BackgroundWorkerDemo\`.

Das Beispiel dient zur Berechnung von Primzahlen in einem eigenen Thread. Wenn Sie es starten, sehen Sie einen Dialog, wie Sie ihn auch in Abbildung 31.8 sehen können.

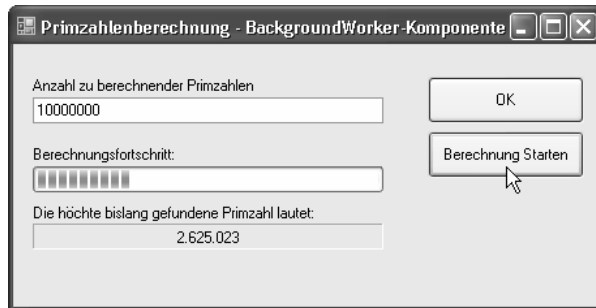


Abbildung 31.8: Mit dem *BackgroundWorker* können Sie auf einfachste Weise neue Threads starten, und bekommen Fortschritts-Informationen auf dem ursprünglichen UI-Thread!

Der entsprechende Code des Programms sieht folgendermaßen aus:

```
Public Class frmMain

    Private myPrimzahlen As List(Of Long)

    Private Sub btnBerechnungStarten_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles btnBerechnungStarten.Click

        'Festlegen, dass der BackgroundWorker über den Fortschritt berichten können soll.
        PrimzahlenBackgroundWorker.WorkerReportsProgress = True

        'Fortschrittsanzeige konfigurieren.
        pbBerechnungsfortschritt.Minimum = 0
        pbBerechnungsfortschritt.Maximum = 1000
        pbBerechnungsfortschritt.Value = 0
        lblGefundenText.Text = "Die höchste bislang gefundene Primzahl lautet:"

        'Ereignis auslösen lassen, damit der Thread
        'in der DoWork-Ereignisbehandlungsroutine gestartet werden kann.
        'Die Obergrenze wird aus dem Textfeld als Argument übergeben.
        Me.PrimzahlenBackgroundWorker.RunWorkerAsync(txtAnzahlPrimzahlen.Text)
    End Sub

    Private Sub PrimzahlenBackgroundWorker_DoWork(ByVal sender As Object, _
        ByVal e As System.ComponentModel.DoWorkEventArgs) _
        Handles PrimzahlenBackgroundWorker.DoWork

        'Das Argument (die Obergrenze) aus den Ereignisparametern wieder herauslesen.
        Dim locMax As Integer = CInt(e.Argument)

        'Ein paar Variablen für die Fortschrittsprozentanzeige einrichten.
        '(OK: Es sind Promille).
        Dim locProgressFaktor As Double = 1000 / locMax
        Dim locProgressAlt As Integer = 0
        Dim locProgressAktuell As Integer
```

```

'Fertig, wenn Obergrenze kleiner 3 ist.
If (locMax < 3) Then Return

'Neues Array definieren, dass die Primzahlen enthält.
'(Wir verwenden als Algorithmus das Sieb des Eratosthenes.)
myPrimzahlen = New List(Of Long)

For z As Integer = 2 To locMax
    If IstPrimzahl(myPrimzahlen, z) Then
        myPrimzahlen.Add(z)

        'Progressinformation senden. Das darf, da der Thread durch SendMessage
        '"gewechselt" wird, nicht zu häufig passieren, sonst hängt der
        'UI-Thread, der die BackgroundWorker-Komponente anfangs initiiert hat.
        locProgressAktuell = Cint(z * locProgressFaktor)
        If locProgressAktuell > locProgressAlt Then
            locProgressAlt = locProgressAktuell
            'Das ProgressChange-Ereignis auslösen, um über den
            'Fortschritt zu informieren.
            PrimzahlenBackgroundWorker.ReportProgress(locProgressAktuell)
        End If
    End If
Next
End Sub

Private Function IstPrimzahl(ByVal Primzahlen As List(Of Long), ByVal Number As Long) As Boolean
    For Each locTestZahl As Long In Primzahlen
        If (Number Mod locTestZahl = 0) Then Return False
        If (locTestZahl >= Math.Sqrt(Number)) Then Exit For
    Next
    Return True
End Function

'Wird aufgerufen, wenn eine BackgroundWorker-Komponente mit
'ReportProgress über den Fortschritt informiert. Dieser Ereignisbehandler
'läuft dennoch auf dem UI-Thread und nicht dem DoWork-Thread. Er kann daher
'ohne Probleme Steuerelemente direkt manipulieren.
Private Sub PrimzahlenBackgroundWorker_ProgressChanged(ByVal sender As Object, _
    ByVal e As System.ComponentModel.ProgressChangedEventArgs) _
    Handles PrimzahlenBackgroundWorker.ProgressChanged
    pbBerechnungsfortschritt.Value = e.ProgressPercentage
    lblHöchstePrimzahl.Text = myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,##0")
End Sub

'Wird aufgerufen, wenn der DoWork-Thread beendet wurde.
'Auch dieser Ereignisbehandler läuft auf dem UI-Thread.
Private Sub PrimzahlenBackgroundWorker_RunWorkerCompleted(ByVal sender As Object, _
    ByVal e As System.ComponentModel.RunWorkerCompletedEventArgs) _
    Handles PrimzahlenBackgroundWorker.RunWorkerCompleted
    lblGefundenText.Text = "Die höchste gefundene Primzahl im Bereich lautet:"
    lblHöchstePrimzahl.Text = myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,##0")
End Sub
End Class

```

HINWEIS: Achten Sie darauf, das `ProgressChanged`-Ereignis nicht zu häufig auszulösen, da Sie den UI-Thread unter Umständen durch das Aktualisieren von Steuerelementen überlasten könnten. Der `DoWork`-Thread läuft dabei zwar regelmäßig weiter, der UI-Thread reagiert dann aber quasi nicht mehr.

Threads durch asynchrone Aufrufe von Delegationen initiieren

Delegationen haben Sie in ► Kapitel 15 kennen gelernt, und für genauere Informationen zu diesem Thema lesen Sie bitte dort nach. Kurz zusammengefasst: Sie dienen zum indirekten Aufruf von Methoden über in bestimmten Objektvariablen abgelegten Adressen dieser Methoden.

Mit der Methode `BeginInvoke` haben Sie die Möglichkeit, eine Prozedur, die durch einen Delegaten dargestellt wird, asynchron, also auf einem eigenen Thread laufen zu lassen. Sie sollten `EndInvoke` aufrufen, wenn die Arbeit erledigt ist. Mit `BeginInvoke` haben Sie die Möglichkeit, einen zweiten Delegaten anzugeben, der aufgerufen wird, wenn die Arbeit abgeschlossen wurde.

Das folgende Beispiel, das ebenfalls Primzahlen berechnet, demonstriert den Einsatz des asynchronen Ausführens von Prozeduren über Delegationen.

BEGLEITDATEIEN: Das Projekt für das folgende Beispiel finden Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap31\AsyncDelegates\AsyncDelegates\`.

```
Imports System.Collections.Generic
Imports System.ComponentModel

Public Class Form1

    Private Delegate Sub PrimzahlenErstellenDelegate(ByVal Max As Integer)
    Private Delegate Sub ErgebnisSetzenDelegate(ByVal ErgebnisText As String)

    Private myPrimzahlen As List(Of Long)
    Private myPZDelegate As PrimzahlenErstellenDelegate
    Private myErgebnisSetzenDelegate As ErgebnisSetzenDelegate
    Private myAsyncOperation As AsyncOperation

    Sub New()

        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()

        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.
        myErgebnisSetzenDelegate = New ErgebnisSetzenDelegate(AddressOf ErgebnisSetzen)
    End Sub

    Private Sub btnBerechnungStarten_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles btnBerechnungStarten.Click
        'Delegate definieren.
        myPZDelegate = New PrimzahlenErstellenDelegate(AddressOf PrimzahlenErstellen)
    End Sub
End Class
```

```

    'Delegate asynchron aufrufen.
    myPZDelegate.BeginInvoke(Integer.Parse(txtAnzahlPrimzahlen.Text), _
        AddressOf BerechnungAbgeschlossenCallback, Nothing)
End Sub

Private Sub PrimzahlenErstellen(ByVal Max As Integer)

    If (Max < 3) Then Return
    myPrimzahlen = New List(Of Long)

    For z As Integer = 2 To Max
        If IstPrimzahl(myPrimzahlen, z) Then
            myPrimzahlen.Add(z)
            'Direkte Manipulation des Steuerelements ist nicht möglich,
            'da dieser Thread nicht dem UI-Thread entspricht.
            lblHöchstePrimzahl.Invoke(myErgebnisSetzenDelegate, _
                myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,##0"))
        End If
    Next
End Sub

Private Function IstPrimzahl(ByVal Primzahlen As List(Of Long), ByVal Number As Long) As Boolean

    For Each locTestZahl As Long In Primzahlen
        If (Number Mod locTestZahl = 0) Then Return False
        If (locTestZahl >= Math.Sqrt(Number)) Then Exit For
    Next
    Return True
End Function

'Delegatroutine für das Aktualisieren der Benutzeroberfläche.
'Notwendig, weil die Aktualisierung auf dem Thread des Delegates ausgeführt wird.
Private Sub ErgebnisSetzen(ByVal ErgebnisText As String)
    lblHöchstePrimzahl.Text = ErgebnisText
End Sub

'Der Rückrufoutine des Delegates, der aufgerufen wird, wenn seine Aufgabe beendet ist.
Private Sub BerechnungAbgeschlossenCallback(ByVal ar As System.IAsyncResult)
    lblHöchstePrimzahl.Invoke(myErgebnisSetzenDelegate, _
        myPrimzahlen(myPrimzahlen.Count - 1).ToString("#,##0"))
    myPZDelegate.EndInvoke(ar)
End Sub

Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
    Me.Close()
End Sub
End Class

```