Maarten Struys is .NET Compact Framework MVP bij PTS Software bv

.NET Compact Framework

DEEL 2: MULTITHREADED APPLICATIES VOOR WINDOWS CE

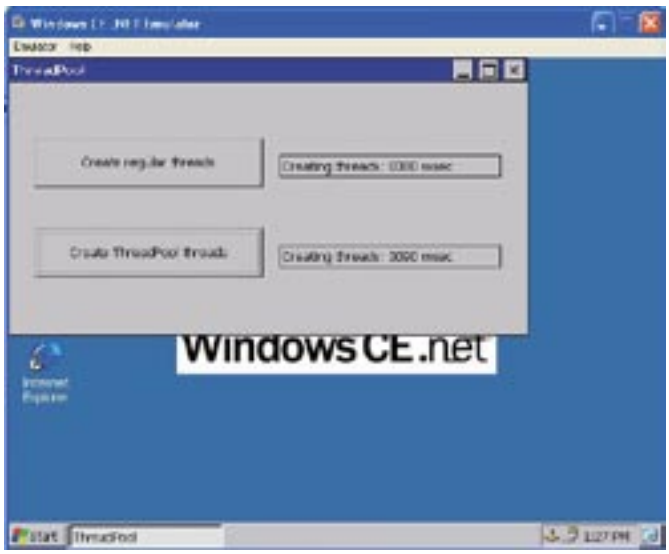
In nummer 8 van het .NET Magazine stonden we stil bij het creëren, starten en stoppen van threads en het gebruik van userinterface-elementen vanuit worker threads. In dit tweede artikel over multithreaded applicaties voor Windows CE kijken we naar synchronisatiemogelijkheden tussen verschillende threads. Daarnaast behandelen we de ThreadPool-class. Vanwege het verschijnen van bèta 2 van Visual Studio 2005 zullen we ook stilstaan bij de nieuwe mogelijkheden die versie 2.0 van het .NET Compact Framework biedt aan multithreaded applicaties.

Ondanks het feit dat we ons specifiek richten op het .NET Compact Framework is de informatie in dit artikel ook relevant voor ontwikkelaars die gebruik maken van het volledige .NET Framework. De omvang van het .NET Compact Framework is ongeveer 10% van de omvang van het .NET Framework. Omdat het .NET Compact Framework slechts een deel van de functionaliteit van het .NET Framework bevat, zullen sommige voorbeeldcodes voor ervaren .NET-ontwikkelaars wellicht inefficiënt lijken.

Thread versus ThreadPool

In het eerste deel van dit artikel hebben we kunnen lezen dat het eenvoudig is om verschillende threads in een applicatie te creëren en te starten met de Thread-class. Elke thread moet een eigen kopie van de processorregisters bewaren en een eigen stack hebben, omdat het Windows CE-besturingssysteem continu schakelt tussen verschillende threads. Tijdens het creëren van een thread wordt daarom een eigen kopie van deze gegevens gemaakt, waardoor het creëren van een thread een tamelijk dure operatie is. Als threads in een applicatie slechts een korte levensduur hebben is dit niet de efficiëntste manier van omgaan met verschillende threads. Een alternatief is het gebruik van de ThreadPool-class. De ThreadPool-class beheert een pool die herbruikbare threads bevat. In versie 1.0 van het .NET Compact Framework kan de ThreadPool-class maximaal 256 threads bevatten. Elke keer als een applicatie gebruik maakt van de ThreadPool zal gekeken worden of een ongebruikte thread beschikbaar is in de pool. Als dat zo is zal die thread worden gebruikt om functionaliteit van de applicatie op deze thread uit te voeren. Als er geen vrije thread beschikbaar is zal een nieuwe thread worden aangemaakt totdat het maximum aantal van 256 threads is bereikt. Een applicatie zal normaal gesproken geen 256 verschillende threads bevatten, waardoor het niet echt belangrijk is ThreadPool-threads alleen te gebruiken voor threads met een relatief korte levensduur. In versie 2.0 van het .NET Compact Framework wordt de grootte van de pool, in overeenstemming met het .NET Framework, teruggebracht tot 25 threads waardoor het wellicht belangrijker wordt ThreadPool-threads niet te gebruiken voor threads die gedurende de gehele levensduur van een applicatie beschikbaar moeten zijn. System.Threading.Timer maakt bijvoorbeeld ook gebruik van de ThreadPool om een timer te maken..Omdat ThreadPool-threads hergebruikt worden, wijkt het gebruik van ThreadPool af van dat van de Thread-class. Met behulp van de methode QueryUserWorkItem op de ThreadPool-class kan een callback meegeven worden die door een ThreadPool-thread uitgevoerd moet worden. Als een thread in de ThreadPool beschikbaar is, zal deze thread direct starten. Indien er geen thread in de ThreadPool beschikbaar is, zal een nieuwe thread in de ThreadPool worden aangemaakt. Als alle threads in de ThreadPool bezet zijn, zal het uitvoeren van de callback-methode worden uitgesteld totdat weer een thread beschikbaar komt in de ThreadPool.

Er zijn een paar dingen waarmee rekening moet worden houden. Het is mogelijk de prioriteit van een ThreadPool-thread te wijzigen in de callback-methode. Zorg er alleen wel voor dat de prioriteit weer teruggezet wordt naar ThreadPriority.Normal voordat de callback-methode wordt beëindigd. In versie 1.0 van het .NET Compact Framework wordt de prioriteit namelijk niet automatisch teruggezet als de callback-methode afloopt. Daardoor zou een andere thread ongewenst op een andere prioriteit kunnen gaan werken. Houd er ook rekening mee dat, net zoals 'gewone' threads, ThreadPool-threads in het .NET Compact Framework 'foreground threads' zijn. Dit impliceert dat een applicatie pas volledig is beëindigd als alle callback-methodes in de ThreadPool zijn gestopt. In tijdkritische situaties is het ook verstandig rekening te houden met het feit dat ongebruikte threads in de ThreadPool na verloop van tijd door de garbage collector zullen worden opgeruimd, waardoor bij opnieuw gebruik maken van een ThreadPool-thread deze wellicht opnieuw door de CLR moet worden aangemaakt.



Afbeelding 1. Gebruik van ThreadPool-threads kan veel snelheidswinst opleveren

```
using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Data;
using System.Threading;

namespace ThreadvsThreadPool
{
  public class Form1 : System.Windows.Forms.Form
  {
    internal System.Windows.Forms.TextBox TextBox2;
    internal System.Windows.Forms.TextBox TextBox1;
    internal System.Windows.Forms.Button Button2;
    internal System.Windows.Forms.Button Button1;

    private System.Threading.AutoResetEvent doneEvent;
    private int threadCounter = 0;
    private int threadPoolCounter = 0;

    public Form1()
    {
      InitializeComponent();
    }

    protected override void Dispose( bool disposing )
    {
      base.Dispose( disposing );
    }
    static void Main()
    {
      Application.Run(new Form1());
    }
    /// <summary>
    /// Create a number of worker threads and wait until all worker
    /// threads have finished executing. The time between starting the
    /// first threadand terminating the last thread is displayed.
    /// </summary>
    private void Button1_Click(object sender, System.EventArgs e)
    {
      threadCounter = 0;
      doneEvent = new AutoResetEvent(false);
      TextBox1.Text = "";
      int elapsedTime = Environment.TickCount;
      for (int i = 0; i < 200; i++)
      {
        Thread workerThread = new Thread(
        new ThreadStart(MyWorkerThread));
        workerThread.Start();
      }
      doneEvent.WaitOne();
      elapsedTime = Environment.TickCount - elapsedTime;
      TextBox1.Text = "Creating threads: " +
        elapsedTime.ToString() + " msec";
    }
    /// <summary>
    /// All worker threads are the same. Once the last worker thread is
    /// executing, it will set an event to trigger the main thread,
    /// allowing it to continue running.
    /// </summary>
    private void MyWorkerThread()
    {
      threadCounter++;
      Thread.Sleep(150);
      if (threadCounter == 200)
      {
        doneEvent.Set();
      }
    }
    /// <summary>
    /// Use a number of threads in the ThreadPool. Since ThreadPool
    /// threads are not created but rather shared, they are much more
    /// performant, especially for short lasting operations.
    /// </summary>
    private void Button2_Click(object sender, System.EventArgs e)
    {
      threadPoolCounter = 0;
      doneEvent = new AutoResetEvent(false);
      TextBox2.Text = "";
      int elapsedTime = Environment.TickCount;
      for (int i = 0; i < 200; i++)
      {
        ThreadPool.QueueUserWorkItem(
          new WaitCallback(MyWaitCallBack));
      }
      doneEvent.WaitOne();
      elapsedTime = Environment.TickCount - elapsedTime;
      TextBox2.Text = "Creating threads: " +
        elapsedTime.ToString() + " msec";
    }
    /// <summary>
    /// This function executes on a ThreadPool thread. Again, all
    /// ThreadPool callbacks execute the same code, identical to the
    /// 'normal' threads.
    /// </summary>
    private void MyWaitCallBack(object stateInfo)
    {
      threadPoolCounter++;
      Thread.Sleep(150);
      if (threadPoolCounter == 200)
        doneEvent.Set();
    }
  }
}
```

Codevoorbeeld 1. Thread versus ThreadPool

```
public void Function1()
{
  for (int i = 0; i < nrLoops; i++)
  {
    counter++;
  }
}

public void Function2()
{
  for (int i = 0; i < nrLoops; i++)
  {
    counter--;
  }
}
```

Codevoorbeeld 2. Wijzigen van een variabele in twee verschillende methoden

Om het verschil in performance te kunnen zien tussen Thread en ThreadPool kunnen we een simpele applicatie gebruiken. Deze applicatie creëert respectievelijk 200 gewone threads en plaatst 200 callbacks in de ThreadPool die door ThreadPool-threads uitgevoerd worden..In beide gevallen wordt de totale tijd gemeten van het aanmaken van de eerste thread tot het afsluiten van de laatste thread. Om activiteit in de thread te simuleren maken we gebruik van de Sleep-methode. Om er zeker van te zijn dat in de ThreadPool meer threads worden aangemaakt, zorgen we er voor dat de wachttijd groter is dan de Windows CE quantumtijd (de tijd die een thread maximaal mag benutten voordat een eventueel andere thread met gelijke prioriteit door het besturingssysteem geactiveerd

Afbeelding 2. Simultaan wijzigen van de instance-variabele kan leiden tot foutieve resultaten

wordt). Daarmee garanderen we dat in ieder geval verscheidene threads in de ThreadPool moeten worden aangemaakt. Telkens als een nieuwe thread in de ThreadPool moet worden aangemaakt, zal dit extra processortijd kosten. Vanwege het hergebruik van deze threads zal de totale benodigde tijd om de voorbeeldapplicatie uit te voeren echter korter zijn als we de ThreadPool gebruiken in vergelijking met het aanmaken van 'gewone' threads. Afbeelding 1 laat het verschil in performance zien tussen Thread en ThreadPool, waarbij de laatste bijna 3 keer sneller is. Nogmaals uitvoeren van dezelfde test zonder de applicatie af te sluiten levert nog meer tijdwinst op, omdat dan voldoende threads in de ThreadPool beschikbaar zijn om alle 200 callbacks uit te voeren zonder de noodzaak nieuwe threads aan te maken. De snelheidswinst kan dan oplopen tot wel 25 keer, tenzij de garbage collector natuurlijk inmiddels de niet in gebruik zijnde threads uit de ThreadPool heeft verwijderd.

Synchronisatie tussen threads

Elke Windows CE-applicatie bestaat uit één of meer threads. Al deze threads kunnen verschillende prioriteiten hebben. Het Windows CE-besturingssysteem is verantwoordelijk voor het verdelen van de totale processortijd over alle verschillende threads die code moeten uitvoeren. Threads met een hogere prioriteit gaan voor threads met een lagere prioriteit. Als meer threads dezelfde prioriteit hebben, krijgen deze om beurten door het besturingssysteem een beetje processortijd toebedeeld. Van tevoren is normaal gesproken niet te voorspellen welke thread wanneer aan de beurt is. In situaties waarin meer threads bewerkingen moeten uitvoeren op gemeenschappelijke data kan dit tot problemen leiden. Laten we bijvoorbeeld eens naar het volgende scenario kijken.

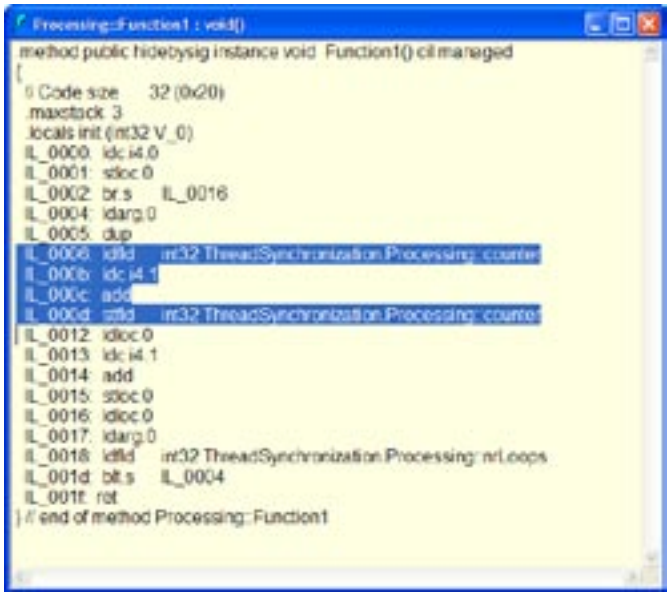
Een applicatie maakt gebruik van twee verschillende threads die beide onafhankelijk van elkaar methoden uit een andere class, genaamd Processing, aanroepen. De Processing-class is heel eenvoudig en bevat twee methoden die elk dezelfde variabele modificeren. Deze variabele met de naam **counter** wordt in beide methoden in een for-loop gewijzigd.
Het aantal keren dat de for-loop wordt doorlopen is in te stellen in de applicatie. De threads in de applicatie roepen beide Processing.Function1 en Processing.Function2 aan. Direct daarna stoppen de threads. Zodra beide threads zijn gestopt, wordt de waarde van counter aan de gebruiker getoond. Omdat beide threads op identieke wijze de methoden uit de Processing-class aanroepen, moet de uiteindelijke waarde van counter altijd 0 zijn. Wanneer de variabele `nrLoops` groot genoeg is blijkt echter dat de uiteindelijke waarde van counter af en toe niet 0 is. Dit gedrag is te verklaren wanneer we begrijpen hoe de Windows CE scheduler werkt en door de gegenereerde *Intermediate Language* van Processing.Function1 en Processing.Function2 te bestuderen.

De Windows CE scheduler werkt volgens het 'round-robin'-principe. Dit betekent dat threads met dezelfde prioriteit om beurten een zelfde hoeveelheid processortijd krijgen, de quantum-tijd. Als een thread niet alle processortijd nodig heeft, zal de scheduler een andere thread actief laten worden. Omdat Windows CE prioriteiten kent, worden threads met hogere prioriteit altijd uitgevoerd vóór threads met lagere prioriteit. In het hier beschreven voorbeeld hebben beide threads dezelfde prioriteit. Als de loop-counter groot genoeg wordt gemaakt, zullen de threads meer quanta nodig hebben om hun werk te beëindigen. Daardoor zullen beide threads beurtelings actief worden; aangenomen dat er geen andere activiteit in het systeem plaatsvindt.
Met deze kennis wordt het tijd om in detail naar de *Intermediate Language* van de Processing-class te kijken. Ondanks het feit dat het verhogen of verlagen van de counter-variabele in C# één enkel statement lijkt, counter++ en counter--, bestaat dit statement uit een aantal *Intermediate Language*-instructies. In afbeelding 3 zijn deze instructies blauw gemarkeerd.

Wat er onder water gebeurt als in C# counter++ wordt uitgevoerd, zijn de volgende instructies:

- De waarde van variabele counter wordt uit het geheugen opgehaald en op de stack geplaatst.
- Het getal 1 wordt op de stack geplaatst.
- De twee bovenste waardes worden van de stack gehaald, bij elkaar opgeteld en het resultaat wordt teruggezet op de stack.
- De bovenste waarde wordt van de stack gehaald en geplaatst in de variabele counter in het geheugen.

Tussen elk van deze instructies kan de scheduler besluiten dat een andere thread aan de beurt is. Stel nu dat één van de worker-threads de waarde van counter op de stack heeft geplaatst en wordt onderbroken door de scheduler. Alle registers en de stack van die thread worden tijdelijk bewaard, waarna de tweede thread actief wordt. Deze thread haalt ook de waarde van counter op, hoogt deze op en plaatst hem terug in het geheugen. Dit zal een aantal keren gebeuren totdat de quantumtijd van de tweede thread is



Afbeelding 3. Intermediate Language voor Processing.Function1

```
public void Function1()
{
   Monitor.Enter(this);

   try
   {
      for (int i = 0; i < nrLoops; i++)
      {
         counter++;
      }
   }
   finally
   {
      Monitor.Exit(this);
   }
}

public void Function2()
{
   Monitor.Enter(this);

   try
   {
      for (int i = 0; i < nrLoops; i++)
      {
         counter--;
      }
   }
   finally
   {
      Monitor.Exit(this);
   }
}
```

Codevoorbeeld 3. Beschermd wijzigen van een variabele in twee verschillende methoden

verstreken, waarna de eerste thread weer actief wordt. Deze zal precies verder gaan op de plaats waar hij eerder door de scheduler werd onderbroken; dus met een eigen locale waarde van counter op de stack. Die waarde zal worden opgehoogd, waarna het resultaat wordt teruggeplaatst in het geheugen. Hiermee wordt dus de waarde overschreven die de tweede thread daar had geplaatst. Dit gedrag zorgt ervoor dat soms foutieve resultaten ontstaan, zoals is te zien in afbeelding 2. Om dit gedrag tegen te gaan moet het modificeren van de instance-variabele worden beschermd door een synchronisatie-object. Hiervoor zijn verscheidene objecten beschikbaar, Monitor, Mutex en in het geval van integer-operaties ook Interlocked. We zullen nu de methodes van de Processing-class gaan beschermen met een Monitor-object. Daarvoor moet de code als volgt worden aangepast.

In codevoorbeeld 3 beschermt het Monitor-object de variabele counter, waardoor deze slechts door één thread kan worden aangepast. Een tweede thread die dezelfde variabele wil aanpassen zal blokkeren totdat het Monitor-object wordt vrijgegeven. Let wel op het gebruik van Monitor. Een aanroep van Monitor.Enter **moet altijd** worden gevolgd door een aanroep van Monitor.Exit, omdat anders andere threads die ook hetzelfde stuk code willen doorlopen eeuwig kunnen blokkeren. Het is daarom goed gebruik om Monitor.Enter en Monitor.Exit als paar binnen één enkele methode uit te voeren. Om de code echt veilig te maken is het ook belangrijk het gebruik van Monitor te beveiligen met een Try – Finally blok, om te garanderen dat het Monitor-object altijd vrijgegeven wordt - zelfs als er excepties in de beschermde code optreden. Achterwege laten van Try – Finally zou in het geval van fouten tot resultaat kunnen hebben dat een Monitor-object nooit meer wordt vrijgegeven, waardoor zelfs deadlock-situaties kunnen ontstaan. In C# is het ook mogelijk het lock-statement te gebruiken. Dit statement combineert een Monitor-object met een Try – Finally blok in één enkel statement, waardoor de leesbaarheid van de code wordt vergroot. Helaas is het lock-statement niet beschikbaar in Visual Basic.NET, zodat in die taal de structuur van codevoorbeeld 3 moet worden gebruikt. De gegenereerde Intermediate Language-code van codevoorbeeld 3 en codevoorbeeld 4 is nagenoeg identiek. Het gebruik van lock heeft als voordeel dat minder code hoeft te worden geschreven en dat niet per ongeluk een Try – Finally blok kan worden vergeten.

Stel nu dat één van de worker-threads de waarde van counter op de stack heeft geplaatst en wordt onderbroken door de scheduler. Alle registers en de stack van die thread worden tijdelijk bewaard, waarna de tweede thread actief wordt. De tweede thread zal nu de waarde van counter willen ophalen, maar blokkeert in Monitor.Enter, omdat de eerste thread op dit moment het Monitor-object in bezit heeft. Doordat de tweede thread direct blokkeert, en onze aanname was dat er geen andere activiteit in het systeem plaatsvindt, zal de scheduler direct de eerste thread weer actief maken. Deze thread kan dus verder gaan met het modificeren van de variabele.

Naast Monitor kan ook gebruik gemaakt worden van Mutex. In het .NET Compact Framework verschillen beide functioneel niet veel van elkaar. Het belangrijkste verschil is dat Monitor een echt managed object is en dat Mutex gebruik maakt van de onderliggende native Win32 mutex. Daardoor zal Monitor iets efficiënter in gebruik zijn. In het .NET Framework is er een ander belangrijk verschil. Daar kan Mutex ook worden gebruikt voor interprocess-synchronisatie, iets dat in het .NET Compact Framework als subset van het .NET Framework niet mogelijk is.

Thread-safe classes

Door het eerder beschreven gedrag is de verleiding nu wellicht groot geworden om elke class-methode te voorzien van een Monitor.Enter–Monitor.Exit-paar. Dit is niet verstandig, omdat dit ten koste gaat van performance. Als in het voorgaande voorbeeld namelijk gegarandeerd maar één enkele thread gelijktijdig de methodes van de Processing-class kan aanroepen is er geen reden de variabelen in Processing te beschermen. Vanwege performance zijn ook lang niet alle classes in het .NET (Compact) Framework thread-safe. Dit betekent dat elke ontwikkelaar zich goed moet realiseren welke stukken code thread-safe moeten zijn. Dit is op zich een juiste keuze, omdat een low level class zelf meestal niet kan bepalen of hij thread-safe moet zijn., Dat hangt namelijk erg af van het gebruik van die class in een bepaalde applicatie. Natuurlijk zijn

```
public void Function1()
{
   lock(this)
   {
      for (int i = 0; i < nrLoops; i++)
      {
         counter++;
      }
   }
}

public void Function2()
{
   lock(this)
   {
      for (int i = 0; i < nrLoops; i++)
      {
         counter--;
      }
   }
}
```

Codevoorbeeld 4. Beschermd wijzigen van een variabele met behulp van het lock-statement

```
private void btnStartThread_Click(object sender, System.EventArgs e)
{
   label1.Text = "Worker Thread started";
   btnStartThread.Enabled = false;
   btnTerminateThread.Enabled = true;
   workerThreadDone = false;
   threadTerminatedEvent = new AutoResetEvent(false);
   Thread myThread = new Thread(new ThreadStart(MyWorkerThread));
   myThread.Start();
}

private void btnTerminateThread_Click(object sender, System.EventArgs e)
{
   workerThreadDone = true;
   threadTerminatedEvent.WaitOne();
   label1.Text = "Worker Thread terminated";
   btnTerminateThread.Enabled = false;
   btnStartThread.Enabled = true;
}

private void MyWorkerThread()
{
   while (! workerThreadDone)
   {
      // simulate some processing
      Thread.Sleep(0);
   }
   threadTerminatedEvent.Set();
}
```

Codevoorbeeld 5. Weten wanneer een thread precies eindigt met behulp van een AutoResetEvent

alle synchronisatie-classes in het .NET (Compact) Framework zelf wel thread-safe, omdat die classes zeker tegelijkertijd door meer threads kunnen worden aangeroepen.

Events, een andere vorm van thread-synchronisatie

In de context van een applicatie die meer threads bevat, is een Event een object dat kan worden gebruikt om een signaal aan een thread te kunnen geven dat 'iets is gebeurd'. In dit geval is een Event een synchronisatieobject dat niet moet worden verward met bijvoorbeeld User Interace Events die zijn gekoppeld aan een event-handler. Voor thread-synchronisatie zijn in het .NET Compact Framework twee verschillende Event-classes beschikbaar, te weten AutoResetEvent en ManualResetEvent. Het verschil tussen beide is dat een AutoResetEvent automatisch wordt gereset, nadat een op het event wachtende thread verder kan gaan, terwijl een ManualResetEvent gezet blijft totdat deze expliciet wordt gereset. Een situatie waarin Events goed van pas komen in het .NET Compact Framework 1.0 is bij het beëindigen van een thread. Stel dat het noodzakelijk is om in een applicatie te weten wanneer een thread exact eindigt. In het .NET Framework en in het .NET Compact Framework 2.0 bestaat daarvoor de Thread.Join-methode, die blokkeert totdat een thread eindigt. In het .NET Compact Framework 1.0 kunnen we deze functionaliteit zelf maken door gebruik te maken van een AutoResetEvent.

In codevoorbeeld 5 zien we dat we in de btnStartThread_Click-methode een AutoResetEvent creëren en een nieuwe thread starten. De thread is erg eenvoudig en simuleert activiteit in een while-loop. De thread eindigt als de boolean variabele workerThreadDone op true wordt gezet. Als laatste wordt in de thread het AutoResetEvent gezet. Stoppen van de thread gebeurt in de btnTerminateThread_Click-methode, waarin workerThreadDone op true wordt gezet. Daarna wordt in deze methode gewacht totdat het Event gezet is, waarna we verder gaan en zeker weten dat de thread is gestopt.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace CS20Sample
{
    public partial class Form1 : Form
    {
        private Thread myThread;
        private delegate void StatusBarUpdater(string newText);

        public Form1()
        {
            InitializeComponent();
        }

        private void btnStart_Click(object sender, EventArgs e)
        {
            btnStart.Enabled = false;

            myThread = new Thread(delegate()
            {
                // The thread itself is defined as an anonymous method
                StatusBarUpdater sbUpd = UpdateStatusBar;

                try
                {
                    this.Invoke(sbUpd, new object[] { "Thread started" });
                    while (true)
                    {
                        Thread.Sleep(0);
                        // simply yield to give up processor time
                    }
                }
                catch (ThreadAbortException exc)
                {
                    MessageBox.Show("Thread aborted (" + exc.Message + ")");
                }
                finally
                {
                  this.BeginInvoke(sbUpd, new object[] { "Thread terminated" });
                }
            });

            myThread.Start();
            btnStop.Enabled = true;
        }

        private void btnStop_Click(object sender, EventArgs e)
        {
            btnStop.Enabled = false;
            myThread.Abort();   // Terminate the thread
            myThread.Join();    // and wait until the thread is really gone.
            btnStart.Enabled = true;
        }

        private void UpdateStatusBar(string newText)
        {
            statusBar1.Text = newText;
        }
    }
}
```

Codevoorbeeld 6. .NET Compact Framework 2.0-functionaliteit en C# 2.0-constructies

.NET Compact Framework 1.0	.NET Compact Framework 2.0
Thread	Thread
CurrentThread	CurrentThread
	IsBackGround
	Name
Priority	Priority
	Abort
	Join
Sleep	Sleep
Start	Start

Tabel 1. Uitbreidingen in de System.Threading-namespace

Events worden typisch gebruikt in situaties waarin een thread een andere thread op de hoogte moet stellen van een bepaalde gebeurtenis, waarna de wachtende thread verder kan gaan. Denk bijvoorbeeld aan het volgende scenario. Een applicatie ontvangt gegevens via een seriële port. In een multithreaded scenario zou er bijvoorbeeld een thread kunnen zijn die ruwe data vanaf de seriële poort leest. Deze data worden lokaal opgeslagen totdat een speciaal karakter binnenkomt; bijvoorbeeld een 'einde bericht'-indicatie). Nadat dit speciale karakter is ontvangen, kunnen de data door de applicatie worden verwerkt. De thread die de ruwe data verzamelde kan nu een AutoResetEvent zetten waar een andere thread op wacht die het complete bericht moet verwerken. Die thread zal nu actief worden. Ook de andere thread blijft actief, omdat er tussentijds misschien nog meer gegevens worden ontvangen. Hierdoor hoeft elke thread slechts beperkte functionaliteit te bevatten. Gegevensverwerking en gegevensontvangst zijn zo op eenvoudige wijze van elkaar gescheiden.

Nieuwe multithreading-mogelijkheden

Tot nu toe hebben we gebruik gemaakt van functionaliteit van het .NET Compact Framework 1.0 voor multithreaded applicaties. Met de komst van Visual Studio 2005 (waarvan een bèta-versie bij het .NET Magazine is ingesloten) verschijnt ook een nieuwe versie van het .NET Compact Framework. Versie 2.0 van het .NET Compact Framework bevat veel meer functionaliteit dan de eerste versie, zo ook voor multithreaded applicaties. In tabel 1 is een overzicht te zien van de belangrijkste properties en methodes die in versie 2.0 in de System.Threading-namespace beschikbaar zijn.

Niet alleen de System.Threading-namespace is behoorlijk uitgebreid. Er is in versie 2.0 van het .NET Compact Framework ook veel betere ondersteuning voor het updaten van userinterface-controls vanuit threads. In versie 1.0 was alleen synchroon updaten mogelijk. Bovendien konden ook geen parameters worden doorgegeven aan een update-delegate. In versie 2.0 kunnen parameters worden meegegeven en is ook asynchroon updaten van userinterface-controls mogelijk. Een belangrijke verbetering is ook dat een *NotSupportedException* wordt gegenereerd als onverhoopt een userinterface-control direct vanuit een thread wordt benaderd die niet de eigenaar is van de betreffende control.
Naast deze nieuwe functionaliteit in het .NET Compact Framework is het natuurlijk ook mogelijk gebruik te maken van nieuwe functies in versie 2.0 van C#. Zo worden *anonymous methodes* ondersteund, waarmee het mogelijk is een methode zonder naam te definiëren die via een delegate kan worden aangeroepen. Dit kan zorgen voor compactere en soms ook beter leesbare code, zoals te zien is in codevoorbeeld 6.

In versie 2.0 van het .NET Compact Framework zijn verschillende manieren beschikbaar om een thread te stoppen. Natuurlijk kan de ontwikkelaar nog steeds gebruik maken van een constructie met een while-loop in combinatie met een boolean variabele, maar daarnaast kan hij de Thread.Abort-methode toepassen. Aanroepen van deze methode zorgt ervoor dat een thread wordt beëindigd en dat een ThreadAbortException wordt gegenereerd. Door in een thread een exception-handler op te nemen met een *finally*-blok, kan zelfs bij een Thread.Abort-aanroep gegarandeerd worden dat de thread een hoeveelheid code nog uitvoert alvorens te stoppen; zie codevoorbeeeld 6. Dit is vooral belangrijk in die situaties waarin een thread resources moet vrijgeven, of connecties moet afsluiten voor beëindiging.

Background-threads zijn ook nieuw in versie 2.0 van het .NET Compact Framework. Een thread wordt een background-thread door simpelweg de property IsBackground op true te zetten. Hiermee wordt het gedrag van een thread vooral anders bij het afsluiten van een applicatie. Een applicatie sluit pas af als alle foreground-threads zijn afgesloten. Met andere woorden, we zijn er zelf verantwoordelijk voor om die threads netjes af te sluiten. Als een applicatie moet worden afgesloten, terwijl die applicatie nog actieve background-threads heeft, worden deze door de Common Language Runtime afgesloten. Onder water gebeurt dit via een Thread.Abort-aanroep. Een aantal uitzonderingen op dit gedrag is specifiek voor het .NET Compact Framework, met name als een background-thread via P/Invoke op een native synchronisatie-object staat te wachten. Het zou echter te ver gaan deze uitzonderingen in dit artikel te behandelen.

Updaten van userinterface-controls

Multithreaded applicaties zijn complexer dan applicaties die geen extra threads creëren. Zoals we in dit artikel hebben gelezen is het vaak nodig threads met elkaar te synchroniseren om geen onverwachte foutieve resultaten te krijgen. In versie 2.0 biedt het .NET Compact Framework behoorlijk veel ondersteuning voor het werken met meer threads. Vooral de nieuwe manieren voor het updaten van userinterface-controls vanuit verschillende threads zijn een aanwinst.

Maarten Struys is werkzaam bij PTS Software bv dat zich specialiseert in technische systeemontwikkeling. Hij heeft in die hoedanigheid veel expertise opgebouwd op het gebied van embedded oplossingen met behulp van Windows CE en Windows XP Embedded. Maarten is .NET Compact Framework MVP en Windows Embedded Evangelist bij PTS Software bv. Hij is regelmatig als spreker te beluisteren tijdens MSDN WebCasts. Maarten schrijft veelvuldig over het .NET Compact Framework op zijn eigen website: www.dotnetfordevices.com.

Nuttige internetadressen

Webcast over multithreading: http://msevents.microsoft.com/cui/WebCastEventDetails.aspx?EventID=1032257390
http://msevents.microsoft.com/CUI/WebCastEventDetails.aspx?EventID=1032265119
MSDN Webcast: Synchronisation possibilities for multithreaded .NET Compact Framework applications (9 februari 2005)
MSDN Webcast: What's new for multithreaded applications in the .NET CF 2.0? (13 april 2005)
http://msdn.microsoft.com/smartclient/understanding/netcf/
http://msdn.microsoft.com/library/en-us/dnanchor/html/NETCompactFrame.asp