Mark Blomsma is medeoprichter van en softwarearchitect bij Omnext.NET bv.

Threading: synchroon versus asynchroon

HET INZETTEN VAN MULTI-THREADING

Multi-threading kan je gebruiken als je wachttijden in je applicatie hebt die je zou kunnen invullen met andere werkzaamheden om de totale doorlooptijd van je applicatie te verlagen.

Even 20 jaar terug in de tijd. Ik zit op m'n Commodore Amiga een beetje 68000 assembler te programmeren. Op Intel gebaseerde machines en de pc zoals we hem vandaag de dag kennen staan in hun kinderschoenen. Inmiddels zijn we 20 jaar verder. Ik ontwikkel nu software voor bedrijven. Veelal administratieve toepassingen of webapplicaties voor het Internet. Multi-tasking is het laatste waar ik aan denk. Data ophalen, tonen, wijzigen, CRUD-functionaliteit, dat is aan de orde van de dag. Een gebruiker doet toch maar een ding tegelijk? Het .NET Framework biedt een aantal prachtige base-classes om eenvoudig multi-tasking toe te voegen aan een eigen applicatie. Wanneer zou je dan gebruik moeten maken van de classes die in de system.threading namespace zitten? Laten we eerst eens kijken naar het thread-model van de Common Language Runtime (CLR) om vervolgens te gaan kijken naar hoe en waar we multi-threading kunnen inzetten.

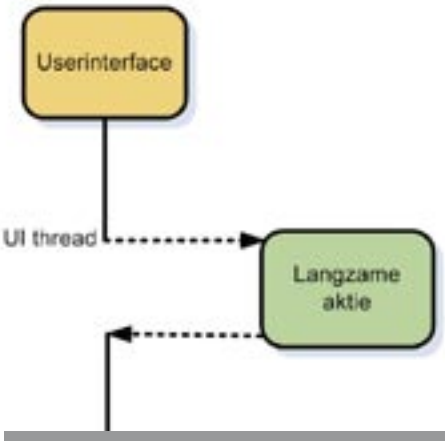
Een belangrijke opmerking over het gebruik van threading vooraf. Threading kan serieuze problemen en zeer complexe bugs veroorzaken in je code. Het advies is om zorgvuldig met threading om te gaan. Neem de tijd om de 'Threading Design Guidelines' op de MSDN-site door te nemen voordat je overgaat tot het implementeren van multi-threading.

CLR thread-model

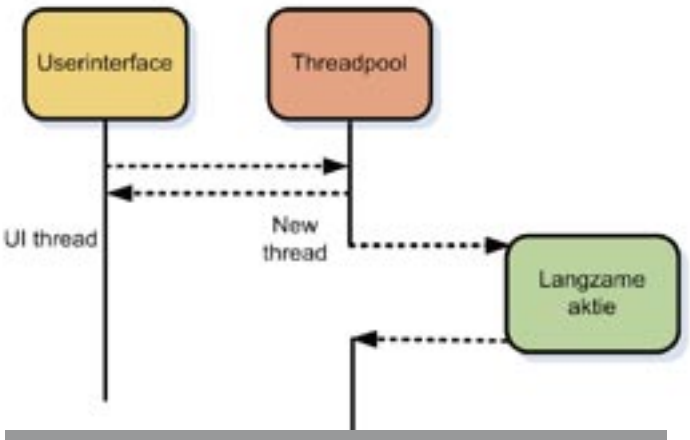
Een veel voorkomend scenario is dat gebruikersinteractie via een applicatie resulteert in een verzoek om de bijbehorende taak door de server af te laten handelen. De server maakt een *thread* om de taak af te handelen, het verzoek wordt afgehandeld, het resultaat teruggegeven en de thread wordt beëindigd. Gedurende de levensloop van de thread kunnen andere threads gestart worden. Vergeleken met het starten en beëindigen van processen (om een taak in af te handelen) is dit een goedkope actie, maar in termen van resources en CPU zeker niet gratis. In het pre-Windows 2000-tijdperk werd, om een thread te aan te maken, voor elke thread een kernel-object geïnstantieerd en bij het beëindigen werd het kernel-object definitief opgeruimd. Voor de liefhebbers: hiervoor werden de DLL_THREAD_ATTACH en DLL_THREAD_DETACH notifications gebruikt. In Windows 2000 werd voor het eerst een *threadpool* geïmplementeerd zodat threads via een poolmanager konden worden hergebruikt.

Het CLR-ontwikkelteam heeft hiervan geleerd en bij het ontwikkelen van de CLR heeft het team direct een managed threadpool ontwikkeld. Je krijgt toegang tot de managed threadpool via de system.Threading namespace. Hoe werkt de managed threadpool? Op het moment dat de CLR wordt geïnitialiseerd, is er een lege threadpool. Op het moment dat een applicatie om de eerste thread vraagt wordt deze geïnitialiseerd en aan de applicatietaak beschikbaar gesteld. De applicatietaak doet zijn werk en geeft de thread vrij. De threadpool schoont de thread nu niet op, maar bewaart de thread om deze te hergebruiken op het moment dat weer om een nieuwe thread wordt gevraagd. Precies wat je verwacht van een pooled resource. De administratieve 'overhead' wordt zo beperkt.

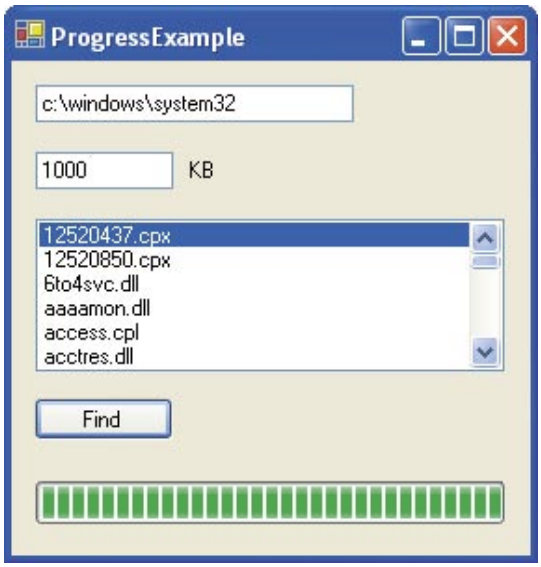
Het zojuist beschreven model is van toepassing voor .NET Framework 1.x en is met .NET Framework 2.0 niet aangepast.



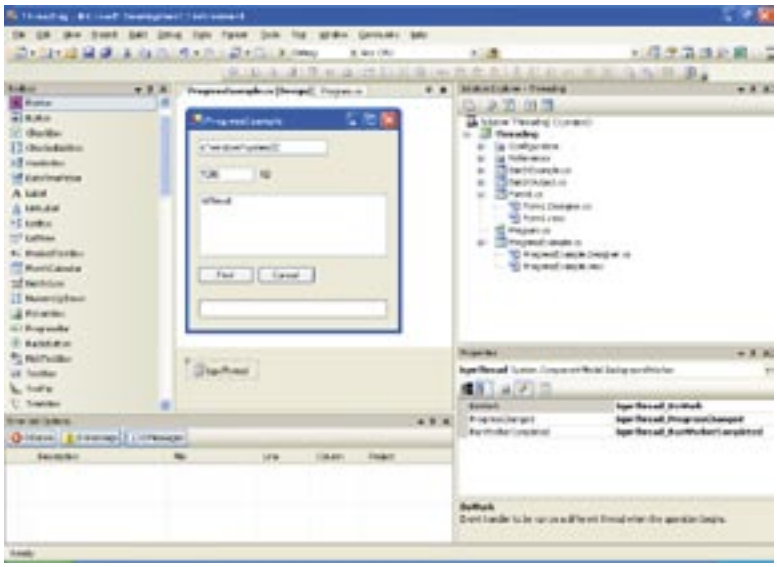
Afbeelding 1. Synchrone executie



Afbeelding 2. Asynchrone executie



Afbeelding 3. Screenshot van onze zoekapplicatie



Afbeelding 4. Gebruik de properties-toolbox om het DoWork-event af te handelen

Wel is de API in system.threading uitgebreid met extra mogelijkheden. Tijd om even een bijzonderheid aan te kaarten. De Windows Forms-architectuur heeft strikte regels voor het gebruik van threads. Regel nummer één is dat interactie via GUI-controls alleen kan plaatsvinden vanuit de thread die de GUI-control heeft gecreëerd. Uit deze regel vloeit voort dat controls die onderdeel uitmaken van andere controls, zoals een button op een form, onderdeel moeten uitmaken van dezelfde thread.

Twee scenario's

De CLR biedt dus multi-threading-mogelijkheden. Gebruiken dus! Abstract gesteld is er maar één reden waarom een applicatie zou moeten 'multi-threaden': je wilt zo min mogelijk wachten op het systeem om zelf een volgende taak te kunnen starten. Wanneer is dat wenselijk? Er zijn twee scenario's waarbij dit relevant is:

- responsetijden verkorten in interactieve applicaties door middel van asynchrone taakafhandeling in de gebruikersinterface.
- doorlooptijden verkorten van niet-interactieve applicaties zoals transactiegeoriënteerde applicaties; bijvoorbeeld een batchjob.

Asynchrone taakafhandeling in de gebruikersinterface

Multi-threading is nuttig op het moment dat de gebruiker een afhandeling van een taak activeert in de applicatie, maar zonder te wachten door wil werken. Hoe lang die afhandeling duurt is afhankelijk van de toegangstijden tot de veelal langzame, externe resources. Het laatste wat de gebruiker wil is op een zandloper wachten. Afbeelding 1 laat zien hoe de UI-thread op de afloop van de langzame actie moet wachten om verder te gaan.
Veel mooier is het, zoals staat in afbeelding 2, wanneer in een fractie van een seconde een nieuwe thread wordt gestart zodra de gebruiker de taak start. De gebruiker kan vervolgens op de UI-thread gewoon doorwerken.

Een voorbeeld: de gebruiker voert een postcode en huisnummer in, verlaat het veld met tab en gaat het telefoonnummer invoeren terwijl het systeem op de achtergrond in de postcodetabel de straat en plaats opzoekt. Een ander voorbeeld: de gebruiker start een taak die lang duurt. Je wilt netjes de voortgang laten zien in een progressbar, maar de gebruiker moet ook de taak kunnen beëindigen. Met Visual Studio.NET 2003 is het programmeren van dergelijke functionaliteit nog een redelijke hersenkraker (een goed artikel hierover vindt u overigens op www.devtips.net). Met Visual Studio 2005 wordt dit scenario echter een stuk eenvoudiger door de BackgroundWorker-control. Aan de hand van een voorbeeld kijken we naar de werking van de BackgroundWorker. Stel, ik wil een applicatie maken die zoekt naar bestanden die groter zijn dan een bepaalde omvang. Afbeelding 3 laat zien hoe de GUI van een dergelijke applicatie eruit komt te zien. We hebben een scherm, een textbox voor het pad waar we willen zoeken, een textbox voor de minimale omvang van het bestand, een listbox voor de gevonden bestanden en een progressbar om te laten zien hoever we zijn.

Als we de gewenste functionaliteit ontwikkelen zonder gebruik te maken van threading zou de code eruit zien zoals die staat in codevoorbeeld 1. We zien dat we de cursor in wachtstand zetten. Alle code wordt uitgevoerd, de progressbar wordt bijgewerkt, het resultaat in de listbox wordt getoond en vervolgens wordt de cursor weer teruggezet. In dit voorbeeld duurt het wachten niet al te lang, maar als je dit recursief voor de subdirectories zou doen, dan duurt een complete harddiskscan toch al snel langer. Verder is het onmogelijk om de taak af te breken. De UI-thread wordt volledig in beslag genomen door het zoeken naar de bestanden.
We gaan het mooier maken. In Visual Studio 2005 sleep ik een BackgroundWorker-control op het formulier en hernoem dit naar

```
private void btnFind_Click(object sender, EventArgs e)
{
  this.Cursor = System.Windows.Forms.Cursors.WaitCursor;
  DirectoryInfo di = new DirectoryInfo(this.txtPath.Text);
  FileInfo[] files = di.GetFiles();
  List<FileInfo> result = new List<FileInfo>();

  int count = files.Length;
  long minSize = long.Parse(this.txtSize.Text);
  double progress = 0;

  for (int i = 0; i < files.Length; i++)
  {
    FileInfo file = files[i];
    if (file.Length > minSize)
    {
      result.Add(file);
    }
    progress = ((i+1) / count) * 100;
    this.pbProgress.Value =(int)progress;
  }
  this.lstResult.DataSource = result;
  this.Cursor = System.Windows.Forms.Cursors.Default ;
}
```

Codevoorbeeld 1.

```
private void bgwThread_DoWork(object sender, DoWorkEventArgs e)
{
   // This method will run on a thread other than the UI-thread.
   // Be sure not to manipulate any Windows Forms controls created
   // on the UI-thread from this method.
}
```

Codevoorbeeld 2.

```
private void btnFind_Click(object sender, EventArgs e)
{
   this.Cursor = System.Windows.Forms.Cursors.WaitCursor;
   bgwThread.RunWorkerAsync();
   this.Cursor = System.Windows.Forms.Cursors.Default;
}

private void bgwThread_DoWork(object sender, DoWorkEventArgs e)
{
   DirectoryInfo di = new DirectoryInfo(this.txtPath.Text);
   FileInfo[] files = di.GetFiles();
   List<FileInfo> result = new List<FileInfo>();

   int count = files.Length;
   long minSize = long.Parse(this.txtSize.Text);
   double progress = 0;

   for (int i = 0; i < files.Length; i++)
   {
      FileInfo file = files[i];
      if (file.Length > minSize)
      {
         result.Add(file);
      }
      progress = ((i + 1) / count) * 100;
      this.pbProgress.Value = (int)progress;
   }
   this.lstResult.DataSource = result;
}
```

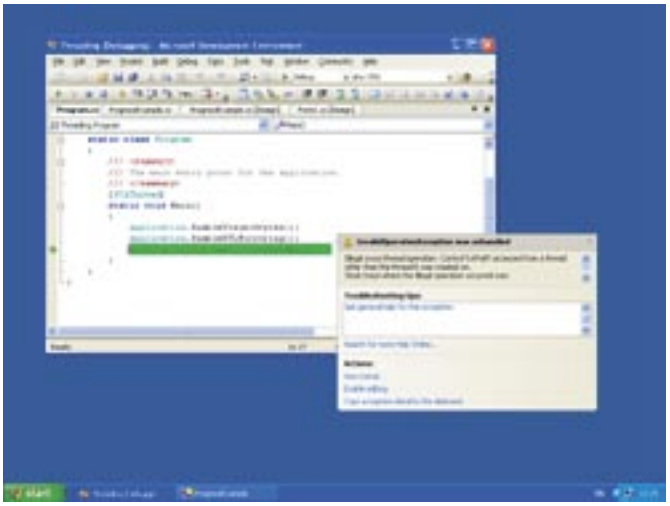
Codevoorbeeld 3.

'bgwThread'. Afbeelding 4 laat zien hoe je na het selecteren van bgwThread op het bliksempictogram kunt klikken in de properties-toolbox om vervolgens op het event DoWork te klikken. Er wordt in de code nu een methode gegenereerd die het DoWork-event afhandelt; zie codevoorbeeld 2. Dit event wordt aangeroepen op het moment dat we op ons 'bgwThread'-object de 'RunWorkerAsync'-methode aanroepen. Codevoorbeeld 3 laat zien dat we de code verplaatsen.
De code in codevoorbeeld 3 lijkt het te gaan doen, maar dit is een illusie. Op het moment dat de code van DoWork wordt uitgevoerd, zitten we in een andere thread. Zoals we al eerder hadden gezien staat de Windows Forms-architectuur niet toe dat we vanuit een andere thread de controls in de UI-thread benaderen. De code in codevoorbeeld 3 kan gecompileerd worden, maar zal runtime leiden tot een 'Illegal cross-thread operation' - zie afbeelding 5.

Om te beginnen moeten we de data die we uit de velden van het formulier willen lezen als parameter meegeven aan de DoWork-methode. In codevoorbeeld 4 is te zien dat we hiervoor even een 'HashTable'-object gebruiken, maar je kunt hiervoor elk willekeurig objecttype inzetten. Binnen de 'DoWork'-methode kunnen we deze data benaderen door de waarde van 'Argument'-property van parameter 'e' uit te lezen.
Vervolgens moeten we informatie teruggeven vanuit onze BackgroundWorker-thread naar de UI-thread. Voor het melden van de progress gebruiken we de 'ReportProgress'-methode op ons 'BackgroundWorker'-object ('bgwThread'). We kunnen aan deze methode een integer meegeven om de voortgang aan te geven. De ReportProgress-methode zorgt ervoor dat een eventhandler aangeroepen wordt. Deze genereer ik in de Designer door op het event te dubbelklikken waarna de 'bgwThread_ProgressChanged'-eventhandler voor me wordt aangemaakt. De code binnen deze eventhandler wordt uitgevoerd in de UI-thread. Dit is handig, want nu is het weer mogelijk om controls te wijzigen. In dit geval werk ik de progressbar bij. Merk op: er kunnen ook extra parameters meegegeven worden aan ReportProgress. Zo kun je bijvoorbeeld de listbox stapje voor stapje vullen. Dit doen we in dit voorbeeld niet. We willen de listbox graag 'binden' aan onze lijst van gevonden bestanden. Een generic list is hierbij een handige toepassing. Op het moment dat we klaar zijn met zoeken, wordt het resultaat in de 'Result'-property van 'e' opgeslagen. Om nu het resultaat te 'binden' aan de listbox moeten we de 'bgwThread_ProgressChanged'-eventhandler implementeren. Wederom is het klikken op het event in de Visual Designer erg handig om de methode te genereren. In deze methode wordt het resultaat weer uit 'e' gehaald en gebonden aan de control. Ook de WorkerCompleted-eventhandler draait in de UI-thread waardoor dit geen runtime-problemen oplevert.
Als laatste voegen we nog een cancel-knop toe aan ons formulier en roepen we de 'CancelAsync'-methode aan op de background-worker. We moeten nu wel wat code toevoegen om binnen de thread regelmatig te controleren of er een afbreekverzoek is geweest. Zo ja, dan kun je correctieve actie ondernemen, of zoals in dit voorbeeld, stoppen met zoeken.

Doorlooptijden verkorten van niet-interactieve applicaties

De andere reden om multi-threading in te bouwen in applicaties is om de gemiddelde doorlooptijd van een batchprogramma te verkleinen. Een batch is een programma waarin bulkverwerking van gegevens plaatsvindt zonder gebruikersinteractie. Hierin wordt in hoog tempo data gelezen, weggeschreven en/of andere vormen van output gegenereerd. Een typisch voorbeeld is de 'facturatie-batch'. Eens per maand start een programma dat alle gegevens van de verwerkte orders uit het systeem leest, facturen aanmaakt, facturen in de database zet en vervolgens de factuur afdrukt. Voor een groot aantal facturen is het belangrijk dat het batchproces optimaal ingericht wordt. Bij een scenario waarin een programma opstart dat één voor één de facturen gaat aanmaken, is de kans groot dat de hardware niet optimaal benut wordt. Dit is natuurlijk zonde, zeker als dit betekent dat de facturatie-batch twee dagen nodig heeft in plaats van één nacht. Elke batchopdracht is uiteindelijk uniek, dus is het moeilijk om hier 'de gouden oplossing' voor alle batchopdrachten te geven. Iedere batchopdracht gebruikt tenslotte systeembronnen. Een van de bronnen zal uiteindelijk de bottleneck vormen. De voornaamste bronnen zijn geheugen, CPU, IO en gekoppeld aan IO



Afbeelding 5. Vanuit de verkeerde thread UI-controls benaderen leidt tot een 'Illegal cross-thread operation'-exception

het transaction-management. IO moet vanuit design geminimaliseerd worden. CPU- en transactiemanagement kunnen beide beïnvloed worden door meer of minder threads en meer of minder parallelle processen. Geheugengebrek kan leiden tot administratieve overhead op het IO-kanaal en de CPU. Als je voor transactiemanagement gebruik maakt van Enterprise Services en gedistribueerde transacties dan zal Microsoft Transaction Server de bronnen (lees: connections naar de database) delen per proces ('Resource Pooling'). Het gebruik van een of meer threads binnen een proces zorgt ervoor dat bronnen gedeeld worden. Hierdoor wordt zuiniger omgesprongen met de beschikbare bronnen. Maar pas op, aan het delen van bronnen zit een maximum vast.

```
private void btnFind_Click(object sender, EventArgs e)
{
   this.Cursor = System.Windows.Forms.Cursors.WaitCursor;
   bgwThread.WorkerReportsProgress  = true;
   bgwThread.WorkerSupportsCancellation = true;
   Hashtable parameters = new Hashtable();
   parameters.Add("dir", this.txtPath.Text);
   parameters.Add("size", long.Parse(this.txtSize.Text));
   bgwThread.RunWorkerAsync(parameters);
   this.Cursor = System.Windows.Forms.Cursors.Default;
}

private void bgwThread_DoWork(object sender, DoWorkEventArgs e)
{
   Hashtable parameters = (Hashtable) e.Argument;
   DirectoryInfo di = new DirectoryInfo((string) parameters["dir"]);
   FileInfo[] files = di.GetFiles();
   List<FileInfo> result = new List<FileInfo>();

   double count = files.Length;
   long minSize = (long) parameters["size"];
   double progress = 0;

   for (int i = 0; i < files.Length; i++)
   {
      FileInfo file = files[i];
      if (file.Length > minSize)
      {
         result.Add(file);
      }
      progress = (((i + 1) / count) * 100);
      bgwThread.ReportProgress((int)progress);
      if (bgwThread.CancellationPending)
      {
         return;
      }
   }
   e.Result = result;
}

private void bgwThread_ProgressChanged(object sender,
   ProgressChangedEventArgs e)
{
   int p = e.ProgressPercentage;
   this.pbProgress.Value = p;
}

private void bgwThread_RunWorkerCompleted(object sender,
   RunWorkerCompletedEventArgs e)
{
   List<FileInfo> result = (List<FileInfo>)e.Result;
   this.lstResult.DataSource = result;
}

private void btnCancel_Click(object sender, EventArgs e)
{
   bgwThread.CancelAsync();
}
```

Codevoorbeeld 4.

```
private void btnStart_Click(object sender, EventArgs e)
{
   ThreadStart ts = new ThreadStart(BatchExample.DoBatch);
   Thread t = new Thread(ts);
   t.Start();
}
```

Codevoorbeeld 5.

Een onderzoek waar ik bij betrokken was heeft aangetoond dat in een specifiek project bij meer dan acht threads per proces de DTC een bottleneck vormt. We hebben toen gekeken naar verscheidene processen. Het blijkt dat wanneer de DTC de flessenhals vormt, de prestatie van acht separate processen hoger is dan de prestatie van één proces met acht threads. Het onderzoek werd uitgevoerd op een multi-processor server. Het resultaat is redelijk te verklaren. Een .NET-proces zal door de CLR namelijk nooit over meer processoren verdeeld worden. Dit houdt in dat een proces wordt toegewezen aan één processor. De andere processor heeft op dat moment dus niets te doen. Echter, door het aantal threads per proces te verhogen, neemt het aantal transacties per seconde nog verder toe. In ons geval blijkt, dat vier processen en vier threads per proces de optimale, gemiddelde doorstroom per machine oplevert. Let op: dit is geen magisch getal dat voor elke batchopdracht zo zal liggen. Het leerpunt is dat bij belangrijke batchopdrachten gezocht moet worden naar het optimum van het aantal processen en het aantal threads.

Om in dit artikel nu even een heel batch-framework uit de doeken te doen gaat te ver. De complexiteit zit veelal toch op een iets functioneler vlak. Hoe verdeel ik het werk (taken) over meer threads? Dit moet op een dusdanige manier gebeuren dat er geen dubbele resultaten ontstaan. In het voorbeeld van facturatie moet per klant één factuur gemaakt worden, geen twee facturen. Een handige verdeelsleutel is dus om per thread een reeks van klantnummers toe te kennen. Maar niet elke klant bestelt elke maand iets, maar we willen wel dat elke thread een vergelijkbare hoeveelheid werk heeft. Om dit te bewerkstelligen moet voorbereidend werk gedaan worden. Dit voorbereidende werk bestaat uit het vinden van alle klanten die in een maand orders hebben geplaatst, deze gelijkmatig verdelen over het aantal threads en vervolgens per thread de lijst met klanten meegeven. Dit is nog een eenvoudig voorbeeld. Het bepalen van klanten die orders hebben geplaatst zal geen zwaar proces zijn, maar als het bepalen van de verdeling over de threads zelf een zwaar proces wordt, dan wordt het allemaal een stuk lastiger.

Tijd voor code. Het aanmaken van threads is eigenlijk heel eenvoudig. De complexiteit van de threadpool wordt afgeschermd door de Thread-class. Door een nieuw object te maken wordt automatisch gekeken of er een thread uit de pool gehaald moet worden.
In codevoorbeeld 5 is te zien dat we eerst een ThreadStart-object aanmaken. Dit is om aan te geven welke methode aangeroepen wordt bij het starten van de nieuwe thread. Deze methode is onderdeel van de constructor van de thread. Met de methode 'Start' wordt de thread vervolgens daadwerkelijk gestart. Merk op dat dit wel een static methode moet zijn. In dit voorbeeld geven we geen parameters mee aan onze methode 'DoBatch'. Met Visual Studio 2005 wordt dit op eenvoudige wijze mogelijk gemaakt door aan 'Start' parameters toe te kennen. Op het moment van schrijven werkte dit echter nog niet in de bètaversie.
Via de threadpool kunnen we nog wel het een en ander optimali-

```
private void btnStart_Click(object sender, EventArgs e)
{
   ThreadPool.SetMaxThreads(5, 5);
   WaitCallback w = new WaitCallback(BatchExample.DoBatch);
   ThreadPool.QueueUserWorkItem(w);
}
```
Codevoorbeeld 6.

```
Thread thread = new Thread(new ThreadStart(Work));
thread.CurrentCulture = Thread.CurrentThread.CurrentCulture;
thread.CurrentUICulture = Thread.CurrentThread.CurrentUICulture;
thread.Start();
```
Codevoorbeeld 7.

```
public static Thread CreateThread(ThreadStart start)
{
   Thread t = new Thread(start);
   t.CurrentCulture = Thread.CurrentThread.CurrentCulture;
   t.CurrentUICulture = Thread.CurrentThread.CurrentUICulture;
   return t;
}
```
Codevoorbeeld 8.

seren met betrekking tot het maximum aantal toegestane threads. Codevoorbeeld 6 laat zien dat we het maximum aantal threads instellen op vijf. Stel dat ik nu acht keer op de button klik, dan zullen er drie items in de wachtrij worden gezet. Er wordt automatisch een item uit de wachtrij gehaald op het moment dat de resources van de thread vrijkomen. In de broncode die te downloaden is (zie nuttige internetadressen) is een aardig voorbeeld te vinden dat dit gedrag laat zien.

Threads en globalization

Een aandachtspunt tijdens het werken met threads en globalization is dat een nieuwe thread standaard de Culture-instellingen van de gebruiker aanneemt. Mocht je dus op basis van de gebruikersinstellingen met de hand de Culture van de applicatie hebben aangepast, dan wordt deze niet automatisch doorgezet naar de nieuwe thread. Dit kan opgelost worden door de code uit codevoorbeeld 7 te implementeren.

Als je op meer plekken in de code een nieuwe thread start is het handiger om hiervoor een class te maken die het Factory-designpattern implementeert. Zie codevoorbeeld 8.

Je zou vervolgens een FxCop-regel kunnen maken om te controleren of deze factory overal in de code wordt gebruikt om nieuwe threads te starten.

Merk op dat de codevoorbeelden in dit artikel gebaseerd zijn op de bètaversies van Visual Studio 2005 en dus mogelijk wijzigen in aanloop naar de definitieve versie. Succes met het toepassen van deze mogelijkheden in jouw eigen applicatie!

Mark Blomsma is medeoprichter van en softwarearchitect bij Omnext.NET bv (www.omnext.net). Hij is lead architect van het Nuclis.NET Framework en houdt zich onder andere bezig met het software renovatie en het migreren van oude systemen naar .NET en .NET 2.0. Naast zijn werkzaamheden bij Omnext is hij voorzitter van het Software Developer Network (www.sdn.nl). Een vereniging van onder andere C# en Visual Basic .NET ontwikkelaars.

Nuttige internetadressen

http://www.microsoft.nl/netmagazine9 (voor de broncode bij dit artikel)
http://msdn.microsoft.com/msdnmag/issues/05/03/AdvancedBasics/default.aspx
http://mtaulty.com/blog/archive/2004/05/24/430.aspx
http://msdn.microsoft.com/msdnmag/issues/03/06/NET/
http://www.codeproject.com/csharp/ManagedThreadCS-Projects.asp
http://msdn.microsoft.com/msdnmag/issues/03/02/Multithreading/default.aspx
http://www.sdn.nl
http://www.devtips.net/article.aspx?id=109