Capítulo 14
Threading

Introdução

A criação de *threads* permitem aumentar consideravelmente a performance das aplicações. Elas fornecem a habilidade de conseguirmos delegar processamentos em diversas unidades de execução, aumentando a capacidade de processamento de uma aplicação. Mas utilizando isso de forma errada, poderá piorar ao invés de melhorar, consumindo mais recursos do que o necessário, tendo um comportamento inesperado e retornando valores diferentes do esperado.

O .NET Framework fornece várias classes que podemos utilizar para criação e gerenciamento de *threads*, bloqueio de recursos em um ambiente *multi-threading* e sincronização. Este capítulo irá ajudá-lo a conhecer alguns problemas existentes em aplicações que fazem o uso de *threads* e como contorná-los, utilizando as classes fornecidas pelo .NET Framework.

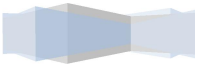
É importante dizer que cada tecnologia (Windows Forms, ASP.NET, WCF, etc.) temos suas peculiaridades em relação a execução dos mesmos. Essas peculiaridades definem a característica das aplicações desenvolvidas sob alguma dessas tecnologias, adotando uma melhor estratégia para a execução das mesmas, tirando o mesmo proveito possível em um ambiente *multi-threading*. O que é importante dizer é que não será abordado neste capítulo essas diferenças mas que, com certeza, fazem o uso dos recursos que serão apresentados aqui.

CPU vs. I/O Bound

Toda operação pode ser considerada CPU *bound* ou I/O *bound*. Operações que são determinadas como CPU *bound*, indicam que dependem da velocidade do processador e quantidade de memória disponível para ser executada. Já as operações de I/O *bound* são o oposto, ou seja, a CPU muitas vezes deverá aguardar o processamento de outros dispositivos (serviços, bancos de dados, acesso à sistema de arquivos, etc.) para dar execução à aplicação e, sendo assim, não importa o quanto o processador é rápido ou o quanto de memória temos disponível, já que essa operação não faz uso destes recursos.

A classe Thread

O .NET Framework possui uma classe chamada *Thread*. Esta classe refere-se à uma unidade lógica de execução. Essa classe fornece diversas propriedades para que o desenvolvedor possa interagir, fornecendo métodos para iniciar, suspender, pausar e propriedades para controlar o status, prioridade, etc. Grande parte das classes que veremos no decorrer deste capítulo, estão dentro do *namespace System.Threading*.



Ao criar uma instância da classe *Thread*, você obrigatoriamente precisa informar qual o método que ela deverá processar, ou melhor, qual tarefa ele deverá executar. Esse vínculo se faz através de um dos seguintes *delegates*: *ThreadStart* ou o *ParameterizedThreadStart*. A escolha entre eles consiste em saber se o método a ser disparado nesta *Thread* irá ou não aceitar um parâmetro. O primeiro *delegate* não define parâmetros; já o segundo, o método a ser vinculado à *Thread* obrigatoriamente precisará ter receber uma instância da classe *Object* como parâmetro. Com isso, dependendo do *delegate* selecionado, você utilizará uma versão específica do método *Start*, ou seja, se está utilizando o *delegate ParameterizedThreadStart*, provavelmente irá utilizar o método *Start* que aceita uma instância de *Object* como parâmetro. Ao invocar este método, a execução será agendada para um posterior processamento. O trecho de código abaixo ilustra o uso destes dois tipos de *delegates*:

VB.NET

```
Imports System.Threading

Sub Main()
    Dim semParametro As New _
        Thread(New ThreadStart(AddressOf ExecutarSemParametro))
    Dim comParametro As New _
        Thread(New             ParameterizedThreadStart(AddressOf
ExecutarComParametro))

    semParametro.Start()
    comParametro.Start("Testando")
End Sub

Sub ExecutarSemParametro()
    Console.WriteLine("ExecutarSemParametro")
End Sub

Sub ExecutarComParametro(ByVal value As Object)
    Console.WriteLine("ExecutarComParametro. Valor: " & value)
End Sub
```

C#

```
using System.Threading;

static void Main(string[] args)
{
    Thread semParametro =
        new Thread(new ThreadStart(ExecutarSemParametro));
    Thread comParametro =
        new                                              Thread(new
ParameterizedThreadStart(ExecutarComParametro));

    semParametro.Start();
    comParametro.Start("Testando");
```

```
}

static void ExecutarSemParametro()
{
    Console.WriteLine("ExecutarSemParametro");
}

static void ExecutarComParametro(object value)
{
    Console.WriteLine("ExecutarSemParametro. Valor: " + value);
}
```

A classe *Thread* fornece uma propriedade chamada *Name* que, como o próprio nome diz, podemos especificar uma *string* contendo o nome da *Thread*. Isso poderá facilitar muito durante a depuração do código. Além disso, ainda temos uma outra propriedade chamada *Priority* que recebe uma das opções fornecidas pelo enumerador *ThreadPriority* (*Highest*, *AboveNormal*, *Normal*, *BelowNormal* e *Lowest*). Os itens fornecidos por esse enumerador definem a prioridade que a *Thread* terá durante a execução, permitindo que uma determinada *Thread* tenha maior prioridade em relação à outra.

As *threads* podem ser de dois tipos: *foreground* ou *background*. Os dois tipos são idênticos em todos os aspectos, com exceção de um: um processo não pode ser encerrado até que *threads* definidos como *foreground* sejam finalizadas. Ao criar uma *thread* a partir da classe *Thread*, por padrão, ela será do tipo *foreground*. Você poderá controlar o tipo da *thread* a partir da propriedade *IsBackground* que recebe um valor booleano indicando se ela é ou não do tipo *background*.

Como vimos no exemplo de código acima, o método *Start* agenda o processamento do método indicado no construtor para ser executado. Uma vez que o processo inicia você pode abortá-lo através do método *Abort*, fornecido também pela instância da classe *Thread*. Há também uma método estático chamado *Sleep* que suspende a execução da *thread* corrente por N milissegundos. Durante a vida/execução de uma *thread* ela pode passar por diversos estados (rodando, suspensa, parada, etc.) e, podemos recorrer a uma propriedade de somente leitura chamada *ThreadState* que, por sua vez, retorna uma das opções do enumerador *ThreadState* contendo a estado atual em que a *thread* se encontra.

Ainda a partir da instância da classe *Thread* podemos utilizar um método chamado *Join*. Quando invocamos este método, a *thread* corrente aguardará até que esta segunda *thread* finalize. Alternativamente podemos informar um *timeout*, especificando o tempo máximo que devemos aguardar até que a mesma retorne; caso isso não aconteça, então a execução do sistema continuará normalmente, enquanto essa segunda *thread* ocorrerá em paralelo.

ThreadPool

Como vimos anteriormente, a criação de *threads* permitem uma melhoria em termos de performance para nossas aplicações já que podemos executar um processo em paralelo enquanto fazemos outros, sem a necessidade de executar tais tarefas de forma seqüencial.

Mas o grande problema com a técnica que vimos acima é que a criação de *threads* é uma tarefa extremamente custosa e, além disso, é muito difícil gerenciá-las depois de criadas. Para resolver este problema, a Microsoft disponibilizou um *pool* de *threads*. Basicamente este é um repositório de *threads* que permite ao desenvolvedor ao invés de criar as *threads* manualmente, delegar ao *pool* a tarefa a ser executada que ele, por sua vez, se encarrega de criar ou reutilizar alguma *thread* para executar a tarefa desejada. O grande benefício de utilizar o *pool* é que as *threads* não criadas a todo momento, para todas as tarefas. O *runtime* do .NET dimensiona a quantidade de *threads* no *pool* de acordo com o *workload* que é realizado. Isso garantirá que ao delegar uma tarefa para o *pool*, muito provavelmente ele utilizará uma *thread* que ele já criou e, ao finalizar a tarefa, a *thread* não será descartada, apenas reciclada e devolvida ao *pool* para uso posterior.

Há dentro do .NET Framework uma classe estática chamada *ThreadPool*, que tem a responsabilidade de gerenciar o *pool* de *threads*. Todas as threads que pool armazena internamente são do tipo *background* (a propriedade *IsBackground* está definida como *True*), o que significa que se a aplicação (processo) finalizar, não aguardará que as tarefas que estão sendo executadas finalizem.

Para cada processo há um *ThreadPool* correspondente. O *ThreadPool* tem inicialmente 25 *worker threads* e 1000 I/O *threads* por processador, podendo estes valores serem alterados através do método *SetMaxThreads*. Além deste método, temos também o *SetMinThreads*, onde podemos especificar a quantidade mínima de *threads*. Isso ajuda a diminuir o custo que existe na criação das primeiras *threads*, evitando que se gaste tempo para criá-las no momento em que se enfileira um pedido. Quando há mais *threads* ociosas do que o estipulado pela quantidade mínima, o *runtime* se encarrega de finalizá-las para liberar os possíveis recursos que elas prendem. Assim como os métodos *SetXXX*, existem os métodos que recuperar o valor mínimo e máxima do *ThreadPool*: *GetMaxThreads* e *GetMinThreads*.

Para colocar uma tarefa (método) para ser executado, podemos recorrer ao método *QueueUserWorkItem*. Esse método recebe uma instância do *delegate WaitCallback* que determina que o método a ser executado, obrigatoriamente, deverá ter a mesma assinatura deste *delegate*, ou seja, não retornar nada e receber uma instância da classe *Object* como parâmetro (podendo ser nulo). Utilizando o mesmo exemplo que vimos acima para exemplificar a classe *Thread*, podemos delegar o processamento do método *"ExecutarComParametro"* para o *ThreadPool*. O trecho de código abaixo ilustra isso:

```
VB.NET
Imports System.Threading

Sub Main()
```

```
    ThreadPool.QueueUserWorkItem( _
        New     WaitCallback(AddressOf     ExecutarComParametro),
"Testando")
End Sub

Sub ExecutarComParametro(ByVal value As Object)
    Console.WriteLine("ExecutarComParametro. Valor: " & value)
End Sub
```

C#

```
using System.Threading;

static void Main(string[] args)
{
    ThreadPool.QueueUserWorkItem(
        new WaitCallback(ExecutarComParametro), "Testando");
}

static void ExecutarComParametro(object value)
{
    Console.WriteLine("ExecutarSemParametro. Valor: " + value);
}
```

Ainda há alguns outros métodos que podemos utilizar para registrar uma tarefa que o *ThreadPool* a execute, mas veremos mais tarde, quando falarmos de sincronização.

Observação: As *threads* criadas manualmente bem como as *threads* que são geridas pela classe *ThreadPool*, são executadas em paralelo à aplicação. Isso quer dizer que ao chamar o método *Start* de uma *Thread* ou o método *QueueUserWorkItem* da classe *ThreadPool*, o controle da execução voltará para a aplicação, não aguardando o processamento desta *thread*.

APM – Asynchronous Programming Model

O processamento assíncrono permite executarmos uma determinada tarefa em uma outra *thread*, conhecida também como *worker thread*. Recorremos a este tipo de técnica quando queremos ter um boa performance, permitindo executarmos uma determinada tarefa em uma *thread* secundária, fora da *thread* que é utilizada pela aplicação.

Felizmente o .NET Framework já traz suporte nativo a esta técnica. Para operações que exigem um processamento mais custoso, como é o caso acesso ao sistema de arquivos, banco de dados, acesso a network, etc., há versão síncrona e assíncrona para cada uma delas. Uma vez que entendemos esta técnica, ela poderá ser aplicada para qualquer um destes recursos, já que a Microsoft manteve o mesmo padrão.

Como exemplo vamos adotar a classe *FileStream* do namespace *System.IO*. Essa classe permite ler ou escrever um conteúdo em um determinada arquivo. Para ler o conteúdo

deste arquivo, poderíamos utilizar o método *Read* que esta classe fornece, que é a versão síncrona do método. Para a versão assíncrona, segundo o padrão da Microsoft, há dois métodos que irão compor esta tarefa: o método que inicia (*BeginRead*) e o que finaliza (*EndRead*). Como podemos notar, a versão assíncrona do método *Read* prefixa os métodos da versão assíncrona em *BeginXXX* e *EndXXX*.

A versão 2.0 do ADO.NET também já fornece métodos para processamento assíncrono, como por exemplo: o método *ExecuteReader* retorna um *DataReader* que representa o conjunto de dados que a consulta retornou. Se quisermos efetuar o processamos assíncrono deste método, temos as opções: *BeginExecuteReader* e *EndExecuteReader*.

Segundo o padrão definido para o processamento assíncrono, o método que inicia o processo assíncrono (BeginXXX) recebe os parâmetros necessários para a execução da tarefa, uma instância de um *delegate* do tipo *AsyncCallback* e um *object* (informação que você poderá passar para o processamento assíncrono). A instância do *delegate* do tipo *AsyncCallback* irá conter o nome do método a ser chamado quando o processado for finalizado. Todos os métodos que inicializam um processo assíncrono, devem retornar uma instância de uma objeto que implementa a *interface IAsyncResult*. Essa *interface* traz informações a respeito do método que está sendo executado assincronamente e também traz membros que podemos utilizar para interagir com este processamento (falaremos mais sobre ele logo abaixo). O método que finaliza o processo assíncrono (*EndXXX*) é responsável por retornar o resultado do processamento assíncrono. O método que finaliza deve receber como parâmetro um objeto que implemente a *interface IAsyncResult*. Isso é necessário para que ele consiga saber onde está o processamento assíncrono de onde ele precisa recuperar a informação.

Observação: Chame o método *EndXXX* somente quando você tiver certeza de que o processo realmente finalizou, caso contrário, a *thread* corrente será bloqueada até que o resultado do processamento assíncrono finalize.

Abaixo veremos um comparativo entre o método *Read* da classe *FileStream* e os métodos *BeginRead* e *EndRead*. Notem que a forma de programar muda bastante, mas temos um ganho de performance, já que a *thread* corrente não ficará bloqueada enquanto o conteúdo do arquivo está sendo lido.

VB.NET

```
Imports System.IO

Sub Main()
    Using fs As New FileStream("Conteudo.txt", FileMode.Open)
        Dim buffer(fs.Length) As Byte
        fs.Read(buffer, 0, fs.Length)
        Console.WriteLine("Processo concluído.")
    End Using
End Sub
```

C#
```
using System.IO;

static void Main(string[] args)
{
    using (FileStream fs =
        new FileStream("Conteudo.txt", FileMode.Open))
    {
        byte[] buffer = new byte[fs.Length];
        fs.Read(buffer, 0, (int)fs.Length);
        Console.WriteLine("Processo concluído.");
    }
}
```

VB.NET
```
Imports System.IO

Private _fs As FileStream
Private _buffer() As Byte

Sub Main()
    _fs = New FileStream("Conteudo.txt", FileMode.Open)
    Array.Resize(_buffer, _fs.Length)
    _fs.BeginRead(_buffer, 0, _fs.Length, _
                New AsyncCallback(AddressOf Resultado), Nothing)

    //continua o trabalho
    Console.ReadLine()
End Sub

Sub Resultado(ByVal result As IAsyncResult)
    Dim total As Integer = _fs.EndRead(result)
    Console.WriteLine("Processo concluído.")
End Sub
```

C#
```
using System.IO;

private static FileStream _fs;
private static byte[] _buffer;

static void Main(string[] args)
{
    _fs = new FileStream("Conteudo.txt", FileMode.Open);
    _buffer = new byte[_fs.Length];
    _fs.BeginRead(_buffer,      0,      (int)_fs.Length,      new
AsyncCallback(Resultado), null);
```

```
    'continua o trabalho
    Console.ReadLine();
}

private static void Resultado(IAsyncResult result)
{
    int total = _fs.EndRead(result);
    Console.WriteLine("Processo concluído.");
}
```

Apesar do método *BeginXXX* retornar uma instância de um objeto que implemente a *interface IAsyncResult*, não estamos neste momento utilizando o mesmo. Isso se deve ao fato de que estamos optando por ser notificado via *callback* quando o processo finalizar, ou seja, quando a leitura do arquivo for concluída, o runtime irá disparar o método *"Resultado"* e, via método *EndXXX* poderemos recuperar o resultado.

Além da técnica de callback, temos a *polling*. Esta técnica consiste em testar de tempos em tempos se o processo assíncrono finalizou ou não. Para verificarmos se o processo finalizou, podemos utilizar a instância do objeto que implementa a *interface IAsyncResult* que é retornado pelo método *BeginXXX*. Essa *interface* fornece uma propriedade chamada *IsCompleted* que retorna uma valor booleano indicando se o processo assíncrono foi ou não finalizado. O código abaixo ilustra esta técnica:

VB.NET

```
Imports System.IO

Sub Main()
    Dim fs As New FileStream("Conteudo.txt", FileMode.Open)
    Dim buffer(fs.Length) As Byte
    Dim result As IAsyncResult = _
        fs.BeginRead(buffer, 0, fs.Length, Nothing, Nothing)

    'continua o trabalho

    If result.IsCompleted Then
        Dim total As Integer = fs.EndRead(result)
        Console.WriteLine("Processo concluído.")
    End If
End Sub
```

C#

```
using System.IO;

static void Main(string[] args)
{
    FileStream fs =
        new FileStream("Conteudo.txt", FileMode.Open);
```

```
    byte[] buffer = new byte[fs.Length];
    IAsyncResult result =
        fs.BeginRead(buffer, 0, (int)fs.Length, null, null);

    //continua o trabalho

    if (result.IsCompleted)
    {
        int total = fs.EndRead(result);
        Console.WriteLine("Processo concluído.");
    }
}
```

Note que antes de chamar o método *EndRead* foi verificado se o processo foi finalizado. Como dito anteriormente, caso você não teste isso e chame o método *EndXXX*, a *thread* corrente irá bloquear, aguardando que o processo de leitura do arquivo seja finalizado.

O processamento assíncrono não é uma característica apenas das classes de acesso à recursos externos, como banco de dados, sistema de arquivos, etc. O .NET também traz esse suporte nos *delegates*. Como já sabemos, *delegates* nada mais são do que ponteiros para métodos. Ao construir um *delegate*, devemos especificar a assinatura (tipo de retorno, quantidade e tipo de parâmetros) que o mesmo irá ter. Ao instanciar um objeto do tipo *delegate*, obrigatoriamente eu deve apontar para um método que tenha a mesma assinatura. O exemplo abaixo ilustra a criação do *delegate* e como utilizá-lo:

VB.NET

```
Delegate Function Operacao(ByVal v1 As Decimal, ByVal v2 As
Decimal)

Sub Main()
    Dim op As New Operacao(AddressOf Somar)
    Console.WriteLine(op(2, 3))
    Console.WriteLine(op.Invoke(2, 3))
End Sub

Function Somar(ByVal v1 As Decimal, ByVal v2 As Decimal) As
Decimal
    Return v1 + v2
End Function
```

C#

```
delegate decimal Operacao(decimal v1, decimal v2);

static void Main(string[] args)
{
    Operacao op = new Operacao(Somar);
    Console.WriteLine(op(2, 3));
    Console.WriteLine(op.Invoke(2, 3));
```

```
}

static decimal Somar(decimal v1, decimal v2)
{
    return v1 + v2;
}
```

Chamar o método *Invoke* é opcional, pois se omitirmos, ele também chamará o método *Invoke*. O método *Invoke* é a versão síncrona, ou seja, ao invocá-lo, ele bloqueará a *thread* corrente até que o método ao qual ele está vincula seja finalizado. Ao instanciar o *delegate* temos a possibilidade de invocar o método qual ele está referenciando de forma assíncrona. Como já era de se esperar, temos os métodos *BeginInvoke* e *EndInvoke*.

Assim como os exemplos anteriores, o método *BeginInvoke* irá retornar uma instância de um objeto que implementa a *interface IAsyncResult* e, além do tradicional *AsyncCallback* e *Object* que, por padrão, ele já tem, temos os parâmetros que são definidos na construção do *delegate* que, no nosso caso, são dois valores do tipo *decimal*. Já o método *EndInvoke* retornará um valor *decimal* pois, como falamos acima, o método *EndXXX* é responsável por retornar o resultado da operação e, como o nosso *delegate* retorna um decimal, ele também retornará. Abaixo temos o mesmo código do exemplo dos *delegates*, mas agora, utilizando a versão assíncrona:

VB.NET

```
Delegate Function Operacao(ByVal v1 As Decimal, ByVal v2 As
Decimal)
Private _op As Operacao

Sub Main()
    _op = New Operacao(AddressOf Somar)
    _op.BeginInvoke(2, 3, New AsyncCallback(AddressOf Resultado),
Nothing)

    Console.ReadLine()
End Sub

Function Somar(ByVal v1 As Decimal, ByVal v2 As Decimal) As
Decimal
    Return v1 + v2
End Function

Sub Resultado(ByVal result As IAsyncResult)
    Console.WriteLine(_op.EndInvoke(result))
End Sub
```

C#

```
delegate decimal Operacao(decimal v1, decimal v2);
private static Operacao _op;
```

```
static void Main(string[] args)
{
    _op = new Operacao(Somar);
    _op.BeginInvoke(2, 3, new AsyncCallback(Resultado), null);

    Console.ReadLine();
}

static decimal Somar(decimal v1, decimal v2)
{
    return v1 + v2;
}

static void Resultado(IAsyncResult result)
{
    Console.WriteLine(_op.EndInvoke(result));
}
```

O exemplo acima está descartando o uso da *interface IAsyncResult* que é retornado pelo método *BeginInvoke*. Assim como nos outros casos, você poderia armazená-lo e utilizar para testar se o processo finalizou ou não ao invés de recorrer a técnica de *callback* que utilizamos aqui.

Sincronização

Quando trabalhamos com uma aplicação *multi-threading* precisamos saber como lidar com possíveis conflitos (entre eles os *deadlocks*) que podem acontecer quando estamos acessando um determinado recurso por múltiplas *threads* ao mesmo tempo.

Esses recursos são geralmente um conteúdo estático ou até mesmo uma instância *singleton* de uma classe e que, possivelmente, múltiplas *threads* podem chegar até eles. Além disso, muitas vezes precisamos ter uma comunicação entre *threads*, notificando uma ou muitas *threads* que um processo finalizou, etc..

Apesar do .NET Framework fornecer diversos recursos para possibilitar a comunicação e sincronização entre múltiplas *threads*, precisamos nos atentar a forma que utilizamos esses recursos, para que possamos tirar o melhor proveito deles, pois quando não são corretamente utilizados, podem prejudicar a aplicação/tarefa. Essa parte do capítulo consiste em apresentar cada um destes recursos, como funcionam e exemplos de implementação que ajudarão a entender melhor o seu funcionamento.

WaitHandle

Esta classe é utilizada como classe base para grande parte dos objetos de sincronização fornecidos pelo .NET Framework. Classes que derivam de *WaitHandle* definem um mecanismo de sinalização que permite indicar o bloqueio ou a liberação de algum recurso

compartilhado. A sinalização consiste em enviar um sinal entre *threads*, notificando a outra *thread* que algo aconteceu e, permitindo assim, que você prossiga ou aguarde pelo processamento de uma ou mais *threads* que estão executando. A imagem abaixo ilustra a hierarquia das classes que herdam direta ou indiretamente da classe *WaitHandle*:

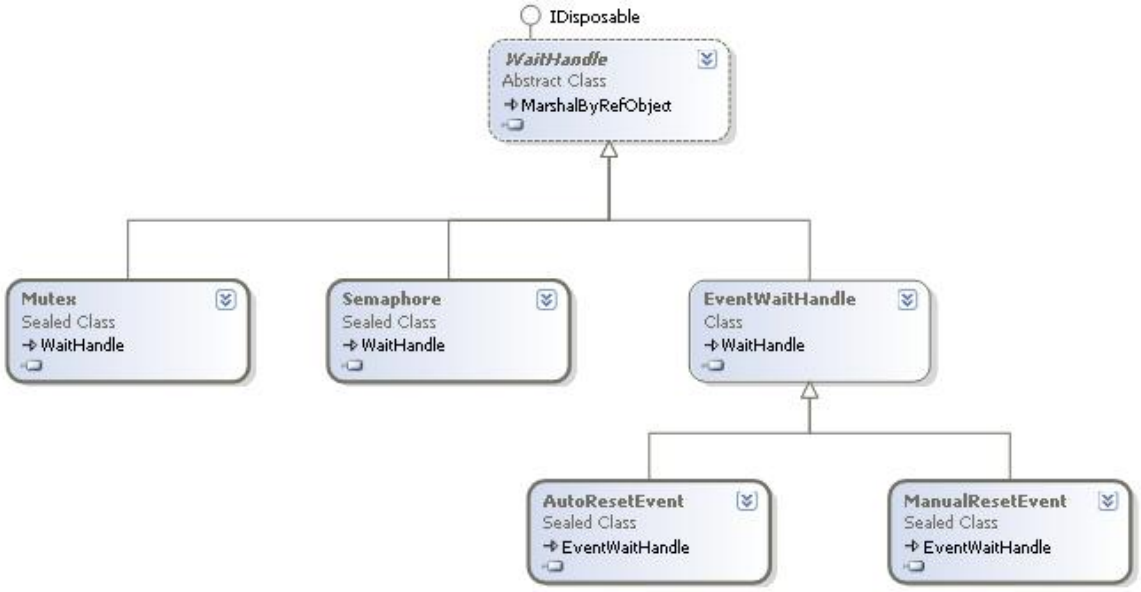


Imagem 14.1 – Hierarquia das classes de sincronização

Veremos detalhadamente cada uma das classes derivadas mais abaixo, ainda neste capítulo.

É importante dizer que esta classe ainda fornece alguns métodos estáticos que bloqueiam uma determinada *thread* até que um ou mais objetos de sincronização recebem um sinal. Os métodos estáticos são:

- **SignalAndWait:** Dados dois *WaitHandles*, esse método chama o método *WaitOne* de um deles e invoca o método *Set* de outro *WaitHandle*, em uma operação atômica.
- **WaitAll:** Recebe um *array* contendo instâncias da classe do tipo *WaitHandle*. Quando você invocar este método, ele aguardará o sinal de todos os elementos contidos no *array*, não prosseguindo até que todos eles notifiquem a *thread* corrente.
- **WaitAny:** Também recebe um *array* contendo instâncias da classe do tipo *WaitHandle*. Só que ao invés de esperar todos os elementos, qualquer um deles que sinalizar, a execução irá continuar.

Além dos métodos estáticos, temos um método de instância chamado *WaitOne*. Este método, uma vez chamado, aguardará que o respectivo *WaitHandle* receba o sinal para que ele consiga prosseguir. O sinal é dado por um outro método, chamado de *Set*.

Como podemos notar na imagem acima, a classe *WaitHandle* implementa a *interface IDisposable*. A finalidade do método *Dispose* aqui implementado é liberar recursos que são gerenciados e utilizados pela propriedade *SafeWaitHandle*.

EventWaitHandle

Esta fornece um funcionamento para que duas *threads* se comuniquem através de sinais. Tipicamente uma ou mais *threads* bloqueiam até que uma delas chame o método *Set*, liberando uma ou mais *threads* para prosseguir a execução. O comportamento desta classe depende do modo de *reset* que você especifica para ela. Este modo é definido através do construtor da mesma, utilizando uma das duas opções fornecidas pelo enumerador *EventResetMode*: *AutoReset* e *ManualReset*. Veremos a diferença entre essas duas opções a seguir.

Essa classe é base para as duas classes que veremos a seguir: *AutoResetEvent* e *ManualResetEvent*.

AutoResetEvent

A classe *AutoResetEvent* herda diretamente da classe *EventWaitHandle*, informando em seu construtor a opção *AutoReset*. Quando estamos aguardando por múltiplas *threads* serem finalizadas, ao ser sinalizado, ele será automaticamente reinicializado e, sendo assim, a execução continuará mas, ele garantirá que apenas uma das *threads* que estão aguardando seja executada. O código abaixo ilustra como podemos proceder para utilizar este objeto:

```
VB.NET
Imports System.Threading

Private _are As AutoResetEvent

Sub Main()
    _are = New AutoResetEvent(False)

    Dim t1 As New Thread(New ThreadStart(AddressOf Tarefa1))
    Dim t2 As New Thread(New ThreadStart(AddressOf Tarefa2))
    Dim t3 As New Thread(New ThreadStart(AddressOf Tarefa3))

    t1.Start()
    t2.Start()
    t3.Start()

    Console.ReadLine()
End Sub

Sub Tarefa1()
```

```
    Thread.Sleep(3000)
    _are.Set()
End Sub

Sub Tarefa2()
    _are.WaitOne()
    Console.WriteLine("Tarefa2")
End Sub

Sub Tarefa3()
    _are.WaitOne()
    Console.WriteLine("Tarefa3")
End Sub
```

C#

```
using System.Threading;

private static AutoResetEvent _are;

static void Main(string[] args)
{
    _are = new AutoResetEvent(false);

    new Thread(new ThreadStart(Tarefa1)).Start();
    new Thread(new ThreadStart(Tarefa2)).Start();
    new Thread(new ThreadStart(Tarefa3)).Start();

    Console.ReadLine();
}

static void Tarefa1()
{
    Thread.Sleep(3000);
    _are.Set();
}

static void Tarefa2()
{
    _are.WaitOne();
    Console.WriteLine("Tarefa2");
}

static void Tarefa3()
{
    _are.WaitOne();
    Console.WriteLine("Tarefa3");
}
```

No exemplo acima, o método *WaitOne* aguardará até que o sinal seja recebido. Quando o método *Set* for disparado, apenas um dos dois métodos (*"Tarefa2"* ou *"Tarefa3"*) que estão aguardando, será disparado.

ManualResetEvent

A classe *ManualResetEvent* herda diretamente da classe *EventWaitHandle*, informando em seu construtor a opção *ManualReset*. Quando estamos aguardando por múltiplas *threads* serem finalizadas, ao ser sinalizado, todas as *threads* que estão aguardando (*WaitOne*) serão disparadas. O código abaixo ilustra como podemos proceder para utilizar este objeto:

VB.NET
```
Imports System.Threading

Private _mre As ManualResetEvent

Sub Main()
    _mre = New ManualResetEvent(False)

    Dim t1 As New Thread(New ThreadStart(AddressOf Tarefa1))
    Dim t2 As New Thread(New ThreadStart(AddressOf Tarefa2))
    Dim t3 As New Thread(New ThreadStart(AddressOf Tarefa3))

    t1.Start()
    t2.Start()
    t3.Start()

    Console.ReadLine()
End Sub

Sub Tarefa1()
    Thread.Sleep(3000)
    _mre.Set()
End Sub

Sub Tarefa2()
    _mre.WaitOne()
    Console.WriteLine("Tarefa2")
End Sub

Sub Tarefa3()
    _mre.WaitOne()
    Console.WriteLine("Tarefa3")
End Sub
```

C#
```
using System.Threading;
```

```
private static ManualResetEvent _mre;

static void Main(string[] args)
{
    _mre = new ManualResetEvent(false);

    new Thread(new ThreadStart(Tarefa1)).Start();
    new Thread(new ThreadStart(Tarefa2)).Start();
    new Thread(new ThreadStart(Tarefa3)).Start();

    Console.ReadLine();
}

static void Tarefa1()
{
    Thread.Sleep(3000);
    _mre.Set();
}

static void Tarefa2()
{
    _mre.WaitOne();
    Console.WriteLine("Tarefa2");
}

static void Tarefa3()
{
    _mre.WaitOne();
    Console.WriteLine("Tarefa3");
}
```

Ao contrário do *AutoResetEvent*, ao ser sinalizado, o objeto *ManualResetEvent* irá permitir que todas as *threads* que estão aguardando a sinalização, sejam disparadas e, no caso acima, os dois métodos (*"Tarefa2"* e *"Tarefa3"*) irão continuar a sua execução.

A classe *ThreadPool* fornece um método chamado *RegisterWaitForSingleObject* que nos permite agendar uma tarefa (método) para processamento mas, além disso, é possível passar um *handle* (*Auto* ou *Manual*) para que ela aguarde até que o sinal seja dado por ele ou um possível *timeout* ocorra. Este método retorna um objeto do tipo *RegisteredWaitHandle* que representa a tarefa que foi agendada e, a qualquer momento, podemos chamar o método *Unregister* do mesmo para cancelar a operação em questão.

O código abaixo mostrará como devemos fazer o uso deste método, especificando um *timeout* de dois segundos (lembrando que o valor deve ser passado em milissegundos). Caso o timeout ocorra antes do método receber a sinalização, ele disparará o mesmo método informado, mas internamente você conseguirá tratar se ocorreu ou não o *timeout* de acordo com o valor do parâmetro *"ocorreuTimeout"*.

VB.NET

```
Imports System.Threading

Sub Main()
    Dim are As New AutoResetEvent(False)

    Dim reg As RegisteredWaitHandle = _
        ThreadPool.RegisterWaitForSingleObject( _
            are, _
            New WaitOrTimerCallback(AddressOf Tarefa), _
            Nothing, _
            2000, _
            True)

    are.Set()
    Console.ReadLine()
End Sub

Sub Tarefa(ByVal value As Object, ByVal ocorreuTimeout As
Boolean)
    If ocorreuTimeout Then
        Console.WriteLine("Ocorreu timeout")
    Else
        Console.WriteLine("Ok. Executado")
    End If
End Sub
```

C#

```
using System.Threading;

static void Main(string[] args)
{
    AutoResetEvent are = new AutoResetEvent(false);

    RegisteredWaitHandle reg =
        ThreadPool.RegisterWaitForSingleObject(
            are,
            new WaitOrTimerCallback(Tarefa),
            null,
            2000,
            true);

    are.Set();
    Console.ReadLine();
}

static void Tarefa(object value, bool ocorreuTimeout)
{
```

```
    if(ocorreuTimeout)
        Console.WriteLine("Ocorreu timeout");
    else
        Console.WriteLine("Ok. Executado");
}
```

Caso queira esperar por um tempo infinito, você pode especificar -1 ao invés do tempo em milissegundos.

Semaphore

Esta classe permite controlar o número de *threads* concorrentes em um determinada recurso. Em seu construtor especificamos um valor inteiro que representa o número de requisições que esta classe pode executar concorrentemente. Além disso, o construtor também permite que você informe o nome que o semáforo irá ter. Fazendo isso, você pode ter um *Semaphore* a nível de sistema, ou seja, poderá sincronizar os recursos entre processos. Quando o nome é omitido, o *Semaphore* apenas servirá para sincronizar os recursos localmente.

Esta classe fornece o método *WaitOne* (que foi herdado da classe *WaitHandle*). Ao chamar este método, ele decrementa o número de *threads* que estão habilitadas (via construtor) a entrar/acessar o recurso que está sendo compartilhado. O método *Release* tem a habilidade de incrementar o número de *threads* disponíveis. Para cada método *WaitOne* chamado, sempre haverá uma chamada para o método *Release*. Caso você chame o método *Release* mais do que o necessário, uma exceção do tipo *SemaphoreFullException* será disparada. O código abaixo ilustra como devemos proceder para utilizar a classe *Semaphore*:

```
VB.NET
Imports System.Threading

Private _sem As Semaphore

Sub Main()
    _sem = New Semaphore(0, 3)

    'Liberamos as 3 threads para execução
    _sem.Release(3)

    For i As Integer = 0 To 10
        Dim t As New Thread(New
ParameterizedThreadStart(AddressOf Tarefa))
        t.Start(i)
    Next

    Console.ReadLine()
End Sub
```

```
Sub Tarefa(ByVal value As Object)
    _sem.WaitOne()

    Thread.Sleep(2000)
    Console.WriteLine(value)

    _sem.Release()
End Sub
```

C#

```
using System.Threading;

private static Semaphore _sem;

static void Main(string[] args)
{
    _sem = new Semaphore(0, 3);

    //Liberamos as 3 threads para execução
    _sem.Release(3);

    for (int i = 0; i < 10; i++)
        new                                                    Thread(new
ParameterizedThreadStart(Tarefa)).Start(i);

    Console.ReadLine();
}

static void Tarefa(object value)
{
    _sem.WaitOne();

    Thread.Sleep(2000);
    Console.WriteLine(value);

    _sem.Release();
}
```

Mutex

Quando duas ou mais *threads* precisamos acessar um recurso compartilhado, precisamos de alguma forma, assegurar que este recurso seja somente acessado por uma única *thread*. *Mutex* é uma das alternativas que temos para conceder acesso exclusivo para um recurso compartilhado, e para uma única *thread*. Se uma *thread* utiliza o *Mutex* para acessar o recurso, a segunda *thread* que quer utilizar o mesmo recurso será bloqueada, aguardando a liberação da primeira *thread* para prosseguir.

Assim como o *Semaphore*, se nomearmos o *Mutex*, ele também poderá ser sincronizado entre processos. Além disso, temos o método *WaitOne* (que foi herdado da classe *WaitHandle*) que tem a finalidade de bloquear o acesso à um determinado recurso, liberando o acesso através do método *ReleaseMutex*. O código abaixo ilustra como podemos utilizar a classe *Mutex*:

VB.NET

```
Imports System.Threading

Private _mutex As Mutex

Sub Main()
    _mutex = New Mutex()

    For i As Integer = 0 To 10
        Dim t As New Thread(New
ParameterizedThreadStart(AddressOf Tarefa))
        t.Start(i)
    Next

    Console.ReadLine()
End Sub

Sub Tarefa(ByVal value As Object)
    _mutex.WaitOne()

    'acesso ao recurso compartilhado

    'simula processo custoso
    Thread.Sleep(2000)
    Console.WriteLine(value)

    _mutex.ReleaseMutex()
End Sub
```

C#

```
using System.Threading;

private static Mutex _mutex;

static void Main(string[] args)
{
    _mutex = new Mutex();

    for (int i = 0; i < 10; i++)
        new Thread(new
ParameterizedThreadStart(Tarefa)).Start(i);
```

```
    Console.ReadLine();
}

static void Tarefa(object value)
{
    _mutex.WaitOne();

    //acesso ao recurso compartilhado

    //simula processo custoso
    Thread.Sleep(2000);
    Console.WriteLine(value);

    _mutex.ReleaseMutex();
}
```

Monitor (lock)

Esta classe também tem a habilidade de controlar o acesso à um determinado código, permitindo que apenas uma única *thread* por vez acesse o recurso. A idéia desta classe é proteger um trecho do código, muitos vezes chamado de seção crítica. Quando uma *thread* adquire o bloqueio, nenhuma outra *thread* será capaz de acessar, tendo que aguardar a liberação para que assim consiga acessar o referido recurso. Além disso é importante dizer que a classe *Monitor* apenas trabalha com tipos por referência.

Esta classe possui um conjunto de métodos estáticos que podemos utilizar:

- **Enter, TryEnter:** Adquire um bloqueio para um determinado objeto. Este método determina o início da seção crítica.
- **Wait:** Bloqueia a execução corrente, aguardando até que um determinado recurso seja liberado ou até que um *timeout* aconteça.
- **Pulse, PulseAll:** Notifica uma (ou todas) as *threads* que estão aguardando o recurso bloqueado.
- **Exit:** Libera o objeto. Este método determina o final da seção crítica.

Utilizar os métodos *Enter* e *Exit* tem funcionalidade semelhante ao utilizar a *keyword lock* do C# (*SyncLock* no Visual Basic), exceto que, quando compilado, as *keywords* de *lock* são colocadas dentro de um bloco *Try/Finally*, garantindo que o método *Exit* seja chamando mesmo que algum problema ocorra. Independente se utilizar as *keywords* de *lock* ou os métodos *Enter/Exit*, isso te dará um maior controle sobre o código a ser protegido. Caso você tenha um método todo a ser protegido, você poderá tornar o código mais simples, aplicando o atributo *MethodImplAttribute* que está contido no *namespace System.Runtime.CompilerServices*, definindo a propriedade *Value* como *Synchronized*. Neste caso, os métodos *Enter/Exit* não são necessário, pois o *runtime* bloqueará até que o método retorne. Caso você pode liberar o recurso um pouco antes do retorno do método, então o mais viável é utilizar as *keywords* de *lock*.

O código abaixo faz o uso da *keyword lock* (*SyncLock* em Visual Basic), mas lembrando que isso apenas é um forma diferente de expressar os método *Enter/Exit*.

VB.NET

```
Imports System.Threading
Imports System.Collections.Generic

Private _items As List(Of String)

Sub Main()
    _items = New List(Of String)

    For i As Integer = 10 To 10
        Dim t As New Thread(New
ParameterizedThreadStart(AddressOf Tarefa))
        t.Start(i)
    Next

    Console.ReadLine()
    Console.WriteLine(_items.Count)
End Sub

Sub Tarefa(ByVal value As Object)
    SyncLock _items
        _items.Add(value.ToString())
    End SyncLock
End Sub
```

C#

```
using System.Threading;
using System.Collections.Generic;

private static List<string> _items;

static void Main(string[] args)
{
    _items = new List<string>();

    for (int i = 0; i < 10; i++)
        new Thread(new
ParameterizedThreadStart(Tarefa)).Start(i);

    Console.ReadLine();
    Console.WriteLine(_items.Count);
}

static void Tarefa(object value)
{
```

```
    lock (_items)
    {
        _items.Add(value.ToString());
    }
}
```

Interlock

Utilizar os mecanismos de bloqueio que vimos até o momento são extremamente úteis quando desejamos sincronizar o acesso à um determinado recurso. Como vimos anteriormente, ao utilizar a classe *Monitor*, somente podemos bloquear o acesso à tipos por referência, e não por valor. Neste caso, como poderíamos efetuar a sincronização de um número inteiro, que efetua a contagem de alguma coisa ou qualquer outra tarefa? Aplicar o bloqueio na instância como um todo (utilizando as *keywords this*/*Me*) apenas para incrementar ou decrementar esse conteúdo diminuirá o *throughput* do objeto.

Felizmente, a Microsoft criou a classe *Interlocked* que já fornece métodos para incrementar (*Increment*) e decrementar (*Decrement*) uma variável inteira (atentando-se que ela deva ser passada por referência). Ela realiza essa operação de forma atômica, garantindo que as operações ocorram de forma única, sem que uma *thread* passe na frente de outra (independentemente se for ler ou escrever um conteúdo), comprometendo o resultado final da operação. Além disso, essa classe ainda fornece um método chamado *Exchange* que, por sua vez, permite alterar o valor de uma variável também de forma atômica.

ReaderWriterLock

Esta classe também tem o papel de efetuar sincronização entre *threads* mas esta classe define um tipo de bloqueio específico, onde ela suporta múltiplas *threads* para ler um determinado recurso, mas permite apenas que uma única *thread* altere este mesmo recurso. Enquanto existir uma thread alterando o conteúdo, as *threads* de leitura serão colocadas em uma fila, aguardando até que a *thread* de escrita finalize ou um possível *timeout* aconteça.

Quando você precisa de um melhor *throughput*, a utilização desta classe tem uma melhor performance em relação a classe *Monitor*, pois ela permite o acesso concorrente para leitura, enquanto a classe *Monitor* não.

Basicamente esta classe fornece dois pares de métodos que são utilizados para você adquirir o bloqueio para leitura ou escrita. Os métodos são: *AcquireReaderLock*, *ReleaseReaderLock*, *AcquireWriterLock* e *ReleaseWriterLock*. Quando invocamos alguns dos métodos utilizados para aquisição do bloqueio (para escrita ou leitura), devemos informar um *timeout*, que é o tempo de espera até que o direito ao acesso para o recurso seja concedido. Caso o *timeout* ocorra, uma exceção do tipo *ApplicationException* será disparada.

O exemplo abaixo ilustra a utilização desta classe. Note que há dois métodos, um que adquire o acesso para escrita e outro que adquire o acesso para leitura. O método que adquire o acesso para escrita, espera por apenas dois segundos; já o método que é responsável pela leitura aguardará, por no máximo, oito segundos.

VB.NET

```
Imports System.Threading

Private _lock As New ReaderWriterLock

Sub Main()
    _lock = New ReaderWriterLock()

    Dim t1 As New Thread(New ThreadStart(AddressOf
TarefaParaEscrita))
    t1.Start()

    Dim t2 As New Thread(New ThreadStart(AddressOf
TarefaParaLeitura))
    t2.Start()

    Console.ReadLine()
End Sub

Sub TarefaParaEscrita()
    _lock.AcquireWriterLock(2000)

    'Simula processo custoso
    Thread.Sleep(5000)

    _lock.ReleaseWriterLock()
End Sub

Sub TarefaParaLeitura()
    _lock.AcquireReaderLock(8000)

    Console.WriteLine("Conseguiu Ler")

    _lock.ReleaseReaderLock()
End Sub
```

C#

```
using System.Threading;

private static ReaderWriterLock _lock;

static void Main(string[] args)
{
    _lock = new ReaderWriterLock();
```

```
    new Thread(new ThreadStart(TarefaParaEscrita)).Start();
    new Thread(new ThreadStart(TarefaParaLeitura)).Start();

    Console.ReadLine();
}

static void TarefaParaEscrita()
{
    _lock.AcquireWriterLock(2000);

    //Simula processo custoso
    Thread.Sleep(5000);

    _lock.ReleaseWriterLock();
}

static void TarefaParaLeitura()
{
    _lock.AcquireReaderLock(8000);

    Console.WriteLine("Conseguiu Ler.");

    _lock.ReleaseReaderLock();
}
```

Por questões de espaço, o código acima não está sendo envolvido em um bloco *Try/Finally* mas, como já vimos acima, esses métodos podem disparar uma exceção do tipo *ApplicationException*.

Além dos métodos tradicionais que vimos para aquisição e liberação do recurso, temos ainda dois outros métodos (*UpgradeToWriterLock* e *DowngradeFromWriterLock*) que nos permite fazer um *downgrade* ou *upgrade* de *writer* para *reader* ou de *reader* para *writer*.

SynchronizationContext

Esta classe permite executarmos uma determinada tarefa em uma outra *thread*, diferente da qual estamos atualmente, representando uma espécie de canal entre as duas *threads* envolvidas. Uma grande utilidade desta classe é quando estamos utilizando uma aplicação Windows Forms. Os controles criados em Windows Forms possuem uma afinidade com a *thread* que iniciou a aplicação e, sendo assim, você somente poderá alterar a propriedade de algum controle a partir desta mesma *thread*. Quando estamos em uma *worker thread*, fazendo alguma tarefa custosa, precisamos notificar a aplicação de que o processo foi finalizado ou até mesmo notificações que falam a respeito do andamento do processamento.

Ela possui uma propriedade estática chamada *Current*, que retorna uma instância da mesma classe, representando o contexto da *thread* atual. A partir daí, podemos utilizar um dos métodos *Send* ou *Post*. A única diferença entre eles é que o método *Send* executa a tarefa de forma síncrona, ao contrário do *Post*, que permite que a tarefa seja executada de forma assíncrona, não havendo necessidade de esperar pela execução desta tarefa.

Segurança

A classe *Thread* possui uma propriedade de escrita/leitura chamada *CurrentPrincipal* que, recebe uma instância de uma classe que implemente a *interface IPrincipal*. As classes do tipo *Principal* definem o contexto de segurança em que uma *thread* está executando, ou seja, define a identidade de um usuário (autenticação) e a possibilidade de saber em quais papéis ele se encontra (autorização).

É baseado nesta propriedade que podemos concedemos ou negamos acesso à um determinado recurso dentro da aplicação. Claro que o tipo de classe a ser definido nesta propriedade muitas vezes acontece automaticamente, e dependendo do tipo de tecnologia utilizada.

Esse contexto refere-se à uma unidade de execução, ou seja, à uma *thread*. Mas podemos utilizar uma técnica para que isso seja propagado entre as outras *threads* que são criadas por uma aplicação. Para isso, poderemos utilizar a classe *ExecutionContext*, abordada abaixo:

ExecutionContext

Quando invocamos um método assíncrono, via *delegate* ou via *ThreadPool*, o contexto é sempre propagado entre as várias *threads* que a aplicação está utilizando e, conseqüentemente, você terá acesso as mesmas informações com relação a execução da *thread* principal e, é neste cenário, que a classe em questão, *ExecutionContext*, entra em ação. Ela fornece quatro métodos importantes, a saber:

- **Capture:** Método estático que captura o contexto de execução atual.
- **SupressFlow:** Suprime a propagação do contexto entre as *threads* assíncronas.
- **Run:** Executa um método (através de um *delegate ContextCallback*) em um contexto específico.
- **RestoreFlow:** Restaura a propagação do contexto entre as *threads* assíncronas.

Para exemplificar esses quatro métodos, o código abaixo utiliza uma função chamada *"LogonUser"* que permite efetuar o *logon* de um determinado usuário. Se a autenticação for realizada com sucesso, personificamos a *thread* atual para este usuário.

VB.NET

```
Imports System
```

```
Imports System.Threading
Imports System.Globalization
Imports System.Security.Principal
Imports System.Runtime.InteropServices

Declare Auto Function LogonUser Lib "advapi32.dll" (ByVal
lpszUsername As String, _
    ByVal lpszDomain As String, ByVal lpszPassword As String,
ByVal dwLogonType As Integer, _
    ByVal dwLogonProvider As Integer, ByRef phToken As IntPtr) As
Boolean

Sub Main()
    Dim token As IntPtr = Nothing
    If LogonUser("Teste", String.Empty, "123456", 2, 0, token)
Then
        WindowsIdentity.Impersonate(token)
        ThreadPool.QueueUserWorkItem(New    WaitCallback(AddressOf
Callback), 1)
        Dim ec As ExecutionContext = ExecutionContext.Capture()
        ExecutionContext.SuppressFlow()
        ThreadPool.QueueUserWorkItem(New    WaitCallback(AddressOf
Callback), 2)
        ExecutionContext.Run(ec,   New   ContextCallback(AddressOf
Callback), 3)
        ThreadPool.QueueUserWorkItem(New    WaitCallback(AddressOf
Callback), 4)
        ExecutionContext.RestoreFlow()
        ThreadPool.QueueUserWorkItem(New    WaitCallback(AddressOf
Callback), 5)
        token = IntPtr.Zero
    End If

    Console.ReadLine()
End Sub

Sub Callback(ByVal state As Object)
    Console.WriteLine("Id: {0}\tThread: {1}\tUser: {2}", _
        state, _
        Thread.CurrentThread.ManagedThreadId, _
            WindowsIdentity.GetCurrent().Name)
End Sub
```

C#

```
using System;
using System.Threading;
using System.Globalization;
using System.Security.Principal;
using System.Runtime.InteropServices;
```

```
[DllImport("advapi32.dll")]
private static extern bool LogonUser(
    String lpszUsername,
    String lpszDomain,
    String lpszPassword,
    int dwLogonType,
    int dwLogonProvider,
    out IntPtr phToken);

static void Main(string[] args)
{
    IntPtr token;
    if (LogonUser("Teste", string.Empty, "123456", 2, 0, out
token))
    {
        WindowsIdentity.Impersonate(token);
        ThreadPool.QueueUserWorkItem(Callback, 1);
        ExecutionContext ec = ExecutionContext.Capture();
        ExecutionContext.SuppressFlow();
        ThreadPool.QueueUserWorkItem(Callback, 2);
        ExecutionContext.Run(ec, new ContextCallback(Callback),
3);
        ThreadPool.QueueUserWorkItem(Callback, 4);
        ExecutionContext.RestoreFlow();
        ThreadPool.QueueUserWorkItem(Callback, 5);
        token = IntPtr.Zero;
    }
    Console.ReadLine();
}

static void Callback(object state)
{
    Console.WriteLine("Id: {0}\tThread: {1}\tUser: {2}",
        state,
        Thread.CurrentThread.ManagedThreadId,
        WindowsIdentity.GetCurrent().Name);
}
```

O método chamado "*Callback*" é utilizado pelos *delegates* para a execução do trabalho assíncrono. Ele exibe em seu interior informações como o estado (*object*) que é passado como parâmetro para o método, *id* da *thread* corrente, o nome do usuário corrente.

O resultado da execução deste código é:

Id: 1 Thread: 3 User: ISRAELAECE\Teste
Id: 2 Thread: 3 User: ISRAELAECE\Israel Aece
Id: 3 Thread: 1 User: ISRAELAECE\Teste
Id: 4 Thread: 3 User: ISRAELAECE\Israel Aece
Id: 5 Thread: 3 User: ISRAELAECE\Teste

Analisando o código do método *Main*, se o *logon* for realizado com sucesso, personificamos a *thread* atual para o usuário chamado "*Teste"*. Em seguida, delegamos para a classe *ThreadPool* a execução do método "*Callback"* de forma assíncrona e resultará na exibição do usuário *"Teste (Id: 1)"*. A partir daqui entra em cena a classe que é tema desta seção.

Através do método estático *Capture* da classe *ExecutionContext*, ela retorna um objeto do mesmo tipo, representando o contexto atual e armazena em um objeto chamado *"ec"*. Agora, através do método *SupressFlow*, ele evita de propagar o contexto para os métodos assíncronos e, conseqüentemente, as requisições realizadas através do *ThreadPool*, resultarão em um outro nome de usuário *(Id: 2)*. Como vimos acima, o método *Run*, dado um contexto e um *delegate*, é responsável por executar o método no contexto informado e, como podemos notar no código, passamos para o método a instância do contexto que capturamos mais acima e que está armazenado no objeto *"ec"* e, além dele, passamos também uma instância do *delegate ContextCallback*, apontando qual será o método que deve ser executado dentro daquele contexto específico. O resultado, como poderíamos esperar, é o usuário *"Teste" (Id: 3)*. Finalmente, restauramos a propagação do contexto quando invocamos o método *RestoreFlow* e, se analisarmos a última execução a partir da *ThreadPool*, temos como resultado o usuário *"Teste" (Id: 5)* novamente.

E, como a documentação diz, enquanto o contexto está suprimido, se invocar o método *Capture*, ele retornará nulo. Se ainda desejar verificar se o contexto está ou não suprimido, pode analisar isso através de um método chamado *IsFlowSuppressed* que retorna um valor booleano indicando essa informação.