05

Usando a propriedade IsAlive e a Thread

Transcrição

No último vídeo, conhecemos a propriedade da classe `Thread`, `IsAlive`. Vamos usá-la em nosso código para pausar a linha de execução principal que criou as duas *threads*, para esperar que o trabalho delas sejam finalizadas e possamos prosseguir. Com esta propriedade, pode-se ficar dentro do laço de repetição, de que sairemos apenas quando a execução das *threads* terminar:

```
thread_parte1.Start();
thread_parte2.Start();

while (thread_parte1.IsAlive || thread_parte2.IsAlive)
{
        //Não vou fazer nada
}
```

Se temos a `thread_parte1` ou a `thread_parte2` trabalhando, não vamos fazer nada, permanecendo neste laço de repetição até que elas sejam finalizadas. Verificaremos como a aplicação é rodada agora, apertando "*Start*". Abriremos também o Gerenciador de Tarefas para acompanharmos o uso da CPU, indo à aba de "Desempenho", que nos mostra os *cores*.

Clicaremos em "Fazer Processamento" no ByteBank. Lembrem-se que no vídeo anterior só havia um núcleo em funcionamento (`CPU 0`), e agora temos mais um. Ou seja, provavelmente o primeiro que estava sendo utilizado está executando uma *thread* da aplicação, e o outro (`CPU 1`), a outra *thread*. Desta forma, teoricamente, a aplicação rodará mais rapidamente - mas isto não é uma regra. Muitos fatores influenciam a velocidade de execução de uma aplicação: o sistema operacional pode estar aguardando respostas da rede, tentando ler o disco rígido, entre outros.

A execução foi finalizada, vemos os outros núcleos da máquina sem processamento algum, já que criamos apenas duas *threads*. A aplicação realmente rodou mais rápido; antes, o processamento levava quase `1min`, e este número desceu para `48s`.

Não parece um ganho tão significativo, mas se pararmos para pensar que estamos com várias aplicações abertas no computador, e que nosso código está sempre em um laço de repetição, verificando as duas propriedades `IsAlive`, talvez faça mais sentido. Por que digo isto? Porque o código ainda está sendo executado na *thread* principal.

Aquele comentário que colocamos, `//Não vou fazer nada`, não é totalmente verdadeiro. Momento após momento, a app verifica o `IsAlive` da `thread_parte1` e, depois, da `thread_parte2`. É um trabalho, esta execução incessante do laço de repetição, causando uso de CPU.

É como quando viajamos com um irmão mais novo, ou quando somos pais, e as crianças ficam nos perguntando "pai, já chegou?", "e agora? Falta muito?". No nosso código, não precisamos fazer isto. Entre uma pergunta e outra, podemos, de fato, não fazer nada. Para isto, ou seja, para não usarmos a CPU, pode-se usar um método estático da classe `Thread` chamado `Sleep`:

```
thread_parte1.Start();
thread_parte2.Start();

while (thread_parte1.IsAlive || thread_parte2.IsAlive)
```

```
{
        Thread.Sleep(250);
        //Não vou fazer nada
}
```

O `Sleep` é um método que, literalmente, coloca a `Thread` para dormir. Ela recebe por parâmetro um número inteiro que representa o número de milisegundos durante os quais a `Thread` ficará sem fazer nada. Vamos checar como a aplicação será rodada. O processamento foi finalizado... Se antes demorava-se mais de `40s`, a linha de código que acabamos de acrescentar para fazer a `Thread` principal parar de ficar perguntando incessantemente se as *threads* estão trabalhando, nos fez diminuir este tempo consideravelmente, passando a levar `30s` para consolidar a informação de todos os clientes de desenvolvimento.

Note que isto é importante: permanecer no laço de repetição consome bastante a CPU. Nossa aplicação está bem mais rápida, criamos duas *threads*, utilizamos o `Sleep`. No entanto, se abrirmos o Gerenciador de Tarefas, vimos que apenas dois dos seis núcleos estão sendo utilizados. Podemos usar todos os oito núcleos, deixando a aplicação ainda mais rápida.

Começaremos a alterar o código a partir da criação de quatro *threads*. Também precisaremos criar uma nova variável para guardar a terceira parte, e mais uma para a quarta parte da nossa lista de contas. Para deixar o código mais limpo e legível, criaremos uma variável para armazenar a quantidade de contas por *thread*, totalizando-se quatro *threads* para esta lista inteira.

```
var contas = r_Repositorio.GetContaClientes();

var contasQuantidadePorThread = contas.Count() / 4;

var contas_parte1 = contas.Take(contasQuantidadePorThread);
var contas_parte2 = contas.Skip(contasQuantidadePorThread).Take(contasQuantidadePorThread);
var contas_parte3 = contas.Skip(contasQuantidadePorThread*2).Take(contasQuantidadePorThread);
var contas_parte4 = contas.Skip(contasQuantidadePorThread*3);
```

Se antes usávamos a metade como parâmetro deste `Take` de `contas_parte1`, agora isto será substituído pela variável `contasQuantidadePorThread`, ou a porção para cada *thread*. A mesma variável é utilizada para o `Skip` na primeira porção, sendo que a segunda usará esta mesma `contasQuantidadePorThread`.

Criadas as variáveis, precisaremos também das *threads* responsáveis por cada uma delas. Usaremos a classe `Thread` do .NET chamada de `thread_parte3` e um construtor *default* que recebe por parâmetro um *delegate* que estamos representando por uma expressão lambda, que por sua vez criará um laço de repetição de cada conta, com todas da parte `3`, cujo resultado será armazenado na variável `r_Servico.ConsolidarMovimentacao`, enviando-o para a conta e adicionando-o em nossa lista de resultados. Faremos o mesmo para a `thread_parte4`, iniciando-se o trabalho destas *threads* em seguida:

```
Thread thread_parte3 = new Thread(() =>
{
  foreach (var conta in contas_parte3)
  {
    var resultadoProcessamento = r_Servico.ConsolidarMovimentacao(conta);
    resultado.Add(resultadoProcessamento);
  }
});

Thread thread_parte4 = new Thread(() =>
{
  foreach (var conta in contas_parte4)
```

```
    {
      var resultadoProcessamento = r_Servico.ConsolidarMovimentacao(conta);
      resultado.Add(resultadoProcessamento);
    }
  });

  thread_parte1.Start();
  thread_parte2.Start();
  thread_parte3.Start();
  thread_parte4.Start();

  while (thread_parte1.IsAlive || thread_parte2.IsAlive || thread_parte3.IsAlive || thread_parte4.IsAl
```

Vamos executar a aplicação e verificar o ganho de performance dela. A execução se acabou em `23s`, em comparação com os `30s` de antes. Verificaremos como isto é demonstrado no processador por meio do Gerenciador de Tarefas, acessando-se a aba "Desempenho". Clicaremos novamente em "Fazer Processamento" no ByteBank, e veremos que as CPUs `0`, `1`, `2` e `3` estão com processamento em `100%`. As outras continuam entre `20%` e `30%`, pois estão executando outras aplicações.

Podemos criar novas *threads* para que estas outras CPUs (`4`, `5`, `6` e `7`) sejam utilizadas. No Visual Studio, vimos que repetimos o construtor `foreach` para todas as *threads*. Para criarmos uma quinta *thread*, teríamos que aumentar o código da mesma forma que fizemos até então. Será que não existe jeito melhor de fazer isto?