02

Criando e executando Tasks

Transcrição

Pessoal, se vamos abstrair nosso problema em tarefas, em vez de simplesmente nos preocuparmos com o trabalho que as *threads* fazem, como dividir isso entre elas, vamos apagar o código antigo e confuso que acabamos criando para gerenciarmos aquelas já existentes.

Qual é a nossa lista de tarefas básica? A menor unidade delas seria consolidar a conta de um cliente, e a tarefa completa seria a consolidação de todos eles. Temos então pequenas tarefas até o término de todo o trabalho. Após deletarmos os códigos referentes às *threads*, criaremos uma variável `contasTarefas`, utilizando o método de extensão chamado `Select`, que facilitará o mapeamento da tarefa para a conta, recebendo uma expressão lambda como parâmetro, que por sua vez recebe por parâmetro uma conta, elemento de nossa lista de contas.

```
var contasTarefas = contas.Select(conta =>
  {
  return Task.Factory.StartNew(() =>
    {
    var resultadoConta = r_Servico.ConsolidarMovimentacao(conta);
    resultado.Add(resultadoConta);
    });
  });
```

Em seu corpo, faremos todo o processamento de criação de uma tarefa. Estamos enfatizando o termo "tarefa" pois, com .NET `4`, surgiu um novo objeto, desenvolvido pela Microsoft e denominado ***Task*** ("tarefa" em inglês). Ele possui uma propriedade estática, `Factory`, que constrói tarefas. Nossa nova tarefa é constituída por outra expressão lambda, a qual não recebe nenhum parâmetro, em que faremos a consolidação desta conta.

A variável `resultadoConta` chama o `Servico`, que consolida a `conta`. Depois disto, precisamos adicioná-la a nossa lista de resultado. Este é o trabalho da nossa tarefa. Definimos assim a tarefa de cada conta, para o processamento de cada uma delas.

Conversamos sobre o gestor de tarefas, sobre qual *thread* resolverá que tarefa, tudo feito internamente por meio da `Factory`. Já construída no .NET, ela é utilizada na criação de uma tarefa, tomando a responsabilidade de delegar funções ao gestor, outro objeto nascido do .NET `4`, chamando-se `TaskScheduler`. Ele define quando uma tarefa será executada ou não, em que lugar isto ocorrerá, tentando sempre otimizar ao máximo a execução da aplicação.

Se há `10` contas sendo executadas simultaneamente, e `8` núcleos, sendo que um deles está parado, o `TaskScheduler` coloca-o para trabalhar. Não precisamos mais nos preocupar em dividir tarefas para cada *thread*, pois ele faz isto. Quando usamos a `Factory`, propriedade estática da classe `Task`, utilizamos o `TaskScheduler` *default* que o .NET nos fornece.

Relembremos que `Select` é uma função do LINQ, que age de forma mais "preguiçosa" possível. Neste momento, na execução da próxima linha: `var fim = DateTime.Now;`. Nenhuma tarefa foi criada, pois o LINQ executa *queries* somente quando necessário. Se não usamos a variável `contasTarefas`, o código que vem a seguir não é executado. Vamos forçar todas as tarefas a serem criadas, gerando um *array* (`.ToArray();`) e obrigando o LINQ a executá-las:

```
var contasTarefas = contas.Select(conta =>
  {
  return Task.Factory.StartNew(() =>
```

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

var fim = DateTime.Now;
```

Vamos rodar a aplicação para ver como ela se comporta agora, pressionando o botão "*Start*" e depois, abrindo o "Gerenciador de Tarefas" para acompanharmos o uso da CPU. Ao iniciarmos o processamento (por meio de "Fazer Processamento"), já notamos que a CPU se encontra em `100%`, com todas as *threads* no topo do gráfico, o que significa que todos núcleos estão trabalhando para resolver estas tarefas.

Então, estamos usando o `TaskScheduler`, que já foi construído no .NET, assim como o objeto `Task`, deixando o código mais limpo e rápido. A aplicação em si não nos mostra nenhum resultado, sendo o mesmo problema enfrentado anteriormente: terminamos a execução da *thread* principal, ou seja, da interface gráfica, antes de obtermos o resultado das contas.

Por isso, criaremos um laço de repetição `while` para verificar se a *thread* `IsAlive` ... Não nos preocupamos mais com *threads*, tampouco com esta propriedade. Faremos isto de um jeito diferente: o `Task` possui um método estático chamado `WaitAll` que não faz nada até que todas as outras tarefas sejam finalizadas. Este método recebe uma lista de tarefas como parâmetro, travando a execução deste até que todas as tarefas terminem.

```
var contasTarefas = contas.Select(conta =>
  {
  return Task.Factory.StartNew(() =>
    {
    var resultadoConta = r_Servico.ConsolidarMovimentacao(conta);
    resultado.Add(resultadoConta);
    });
  }).ToArray();

Task.WaitAll(contasTarefas);

var fim = DateTime.Now;

AtualizarView(resultado, fim - inicio);
```

No momento em que se alcança o código `Task.WaitAll(contasTarefas)`, a aplicação é pausada, e as linhas seguintes - de finalização e atualização da tela - só serão executadas quando terminarmos todas as tarefas existentes. Verificaremos seu funcionamento apertando "*Start*" mais uma vez e acompanhando o uso das CPUs pelo Gerenciador de Tarefas.

A tela da app continua com aspecto de estar travada, porém o uso da CPU está em `100%` e, usando os gráficos, iremos ver que todos os *cores* estão trabalhando. Desta vez, temos um resultado na tela. No começo do curso a aplicação levava cerca de `58s` para realizar o processamento. Este número caiu para `12s`, é um ganho e tanto.

No entanto, continuamos enfrentando o problema de que ao clicarmos em "Fazer Processamento", a tela trava. Vamos melhorar isso?