04

Mantendo a interface gráfica ativa

Transcrição

A aplicação dá ao usuário a impressão de estar travada pois a tela deixa de responder ou mover. Quando passamos o mouse, nada acontece, e o mesmo ocorre ao clicarmos em qualquer parte da tela. Isto ocorre porque a app (a interface gráfica) roda em uma *thread* separada, ou "principal" em uma WPF, ou aplicação Windows Forms. No momento em que se clica nela, para que a ação seja executada de um botão, o processamento é pausado.

Todas as animações e movimentos do mouse e da própria janela são pausadas, começando-se a executar o código referente ao botão. Se ali é solicitado que se aguarde o processamento de várias outras tarefas, a interface gráfica trava, dando uma impressão negativa ao usuário.

Esta é uma reclamação do ByteBank. Como podemos melhorar isto? Em vez de pararmos a aplicação e esperarmos a finalização de tudo para atualizar a *view*, podemos solicitar ao .NET para que ele execute outro código ao fim de todas as tarefas, liberando a *thread* de execução do código principal.

A partir do atalho `F12`, verificaremos melhor a classe `Task`. Estamos utilizando `WaitAll`, método que não possui retorno, pausando a execução da *thread* principal, ou daquela que estiver rodando-a, até que seu parâmetro termine, no caso a lista de *tasks*.

```
public static void WaitAll(params Task[] tasks);
```

Porém, em vez de `WaitAll`, podemos optar pelo método `WhenAll`, cuja diferença consiste em retornar uma tarefa que espera o término de todas as demais recebidas por parâmetro. Voltando ao código, trocaremos `WaitAll` por `WhenAll`:

```
Task.WhenAll(contasTarefas);
```

Parece que não mudou muito, mas em uma tarefa, podemos encadear outras. É o que faremos com o método `ContinueWith`, que recebe outro *delegate* como parâmetro. Este por sua vez, só será executado quando a tarefa anterior for finalizada:

```
Task.WhenAll(contasTarefas)
  .ContinueWith(task) => {

    });
```

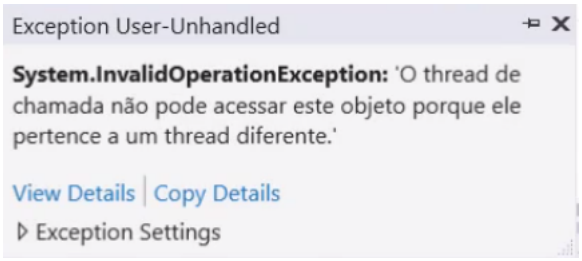
Observe que a expressão lambda tem um parâmetro, falamos do `task`, e dentro do corpo da função temos disponível a task que originou a execução. Ou seja, a `task` será a tarefa que espera todas as outras (`Task.WhenAll()`). Isso é bom pois teremos informações, por exemplo, saberemos se a *task* anterior lançou uma exceção, entre outras. Mas não nos preocuparemos com isso agora.

Em seguida, atualizaremos a *view* e vamos obter o tempo final de execução. Este código, portanto, não será executado na *thread* principal, por isso, vamos movê-lo para dentro da *task* recém criada:

```
Task.WhenAll(contasTarefas)
  .ContinueWith(task) => {
```

```
        var fim = DateTime.Now;
        AtualizarView(resultado, fim - inicio);
      });
```

Vamos verificar como a aplicação é executada após esta mudança. A CPU está sendo utilizada em `100%`, conseguimos mover a janela da app, as animações continuam acontecendo. Porém, temos uma mensagem de erro:



O erro ocorreu porque estávamos na *thread* principal, atualizando a lista de resultados (`LstResultados`) e o texto de resumo (`TxtTempo.Text = mensagem`) na mesma *thread*. Colocamos os códigos criados agora em uma diferente, sem saber qual, cuja responsabilidade não é nossa. Uma *thread* diferente não pode acessar o controle da interface gráfica. O .NET protege isto, e quando uma tentativa de acesso a um objeto da interface gráfica ocorre por uma *thread* diferente, a aplicação é travada, lançando-se a exceção.

É necessário informar ao .NET que o código recém criado será executado na interface gráfica, sem ser necessário esperar todas as outras, mantendo-se a interface gráfica travada até que a execução do código. Lembra que tínhamos um intermediário responsável por delegar quem executaria qual tarefa, delegando assegurando que todas as *threads* fossem executadas da forma mais otimizada possível?

O `TaskScheduler` também existe na *thread* principal, sendo obtido a partir de uma que estiver em execução, com o método estático `FromCurrentSynchronizationContext()`, que retorna o `TaskScheduler` atuante no momento.

```
TaskScheduler.FromCurrentSynchronizationContext();
Task.WhenAll(contasTarefas)
  .ContinueWith(task) => {
      var fim = DateTime.Now;
      AtualizarView(resultado, fim - inicio);
    });
```

Caso executemos o `TaskScheduler` dentro do método `WhenAll()`, ou seja, dentro de outra tarefa, obteremos um resultado diferente. Para deixar isto ainda mais claro, colocaremos o método que obtém isto no início do código, na parte de clique do botão, guardando na variável `TaskSchedulerUI` - lembrando que UI é a sigla para "interface gráfica" em inglês, *"user interface"*).

```
private void BtnProcessar_Click(object sender, RoutedEventArgs e)
{
  var taskSchedulerUI = TaskScheduler.FromCurrentSynchronizationContext();

  var contas = r_Repositorio.GetContaClientes();

  var resultado = new List<string>();
//...
}
```

Assim, informamos ao .NET nosso desejo de que a tarefa referente à finalização e atualização de visualização seja feita de acordo com a demanda do `TaskScheduler`, podendo ser enviado no método `ContinueWith`, como um parâmetro.

```
Task.WhenAll(contasTarefas)
  .ContinueWith(task => {
    var fim = DateTime.Now;
    AtualizarView(resultado, fim - inicio);
  }, taskSchedulerUI);
```

Feito isso, a tarefa só será executada após todas as outras, como determinamos, porém, de acordo com o `TaskScheduler` da interface gráfica. Vamos ver se aquele erro continua acontecendo? Pressionaremos "*Start*", clicaremos em "Fazer Processamento", mexendo no cursor, movimentando a tela e abrindo o "Gerenciador de Tarefas". Verificaremos que a CPU está sendo utilizada e o processamento foi realizado. Tudo foi feito paralelamente, e a tela não travou.

Repetiremos todo este procedimento. Desta vez, a aplicação irá demorar um pouco mais, pois ela executa várias tarefas ao mesmo tempo, usando a CPU depois de clicarmos diversas vezes em "Fazer Processamento". Vamos melhorar isto? Deixaremos o botão desabilitado no começo da execução, voltando ao funcionamento normal ao fim da mesma.

```
private void BtnProcessar_Click(object sender, RoutedEventArgs e)
{
  var taskSchedulerUI = TaskScheduler.FromCurrentSynchronizationContext();
  BtnProcessar.IsEnabled = false;

  var contas = r_Repositorio.GetContaClientes();

  var resultado = new List<string>();
//...
}
```

No começo da execução, alteraremos o botão de processamento, depois, definiremos a propriedade `IsEnabled` como `false`. Porém, no fim, precisaremos que ele fique habilitado novamente caso o usuário queira fazer outro processamento. Temos uma nova tarefa a ser executada depois da atualização de *view*, capaz de tornar o botão disponível novamente.

```
Task.WhenAll(contasTarefas)
  .ContinueWith(task => {
    var fim = DateTime.Now;
    AtualizarView(resultado, fim - inicio);
  }, taskSchedulerUI);
  .ContinueWith(task => {
    BtnProcessar.IsEnabled = true;
    });
```

Vamos executar a aplicação novamente. O botão fica desabilitado após o primeiro clique, isto impossibilitará que ele fique rodando desnecessariamente, enquanto se realiza um processamento. No entanto, veremos o mesmo erro por descuido, pois esquecemos de colocar que é preciso executar no `taskSchedulerUI`. Corrigiremos isso:

```
Task.WhenAll(contasTarefas)
  .ContinueWith(task => {
    var fim = DateTime.Now;
    AtualizarView(resultado, fim - inicio);
```

```
    }, taskSchedulerUI);
    .ContinueWith(task => {
      BtnProcessar.IsEnabled = true;
      }, taskSchedulerUI);
```

Após apertaremos "*Start*" de novo, rodaremos a aplicação, depois, clicaremos em "Fazer Processamento". O botão será travado e conseguiremos mover a janela sem problemas. Ao término do processamento da aplicação, o botão voltará a ficar disponível para nós. Mas o código está ficando complicado... Veja quantas funções encadeadas temos, com várias tarefas sendo criadas também. Estamos voltando àquele estado anterior de código confuso e prolixo. A seguir, vamos melhorar isso.