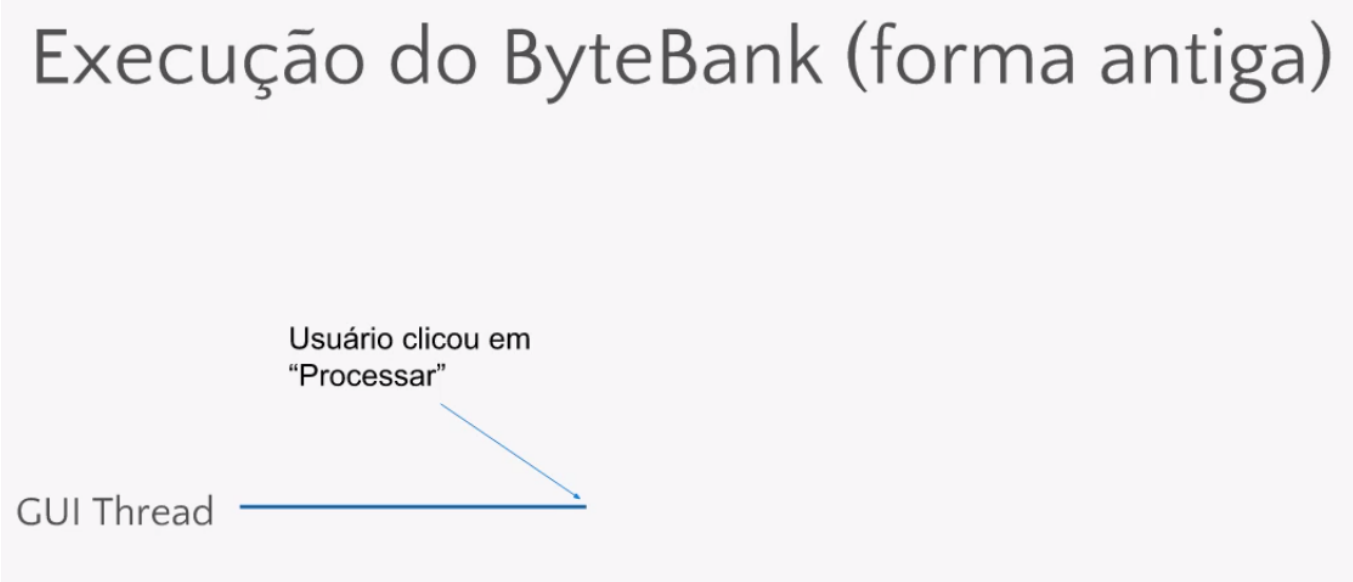
01

Tasks com retorno

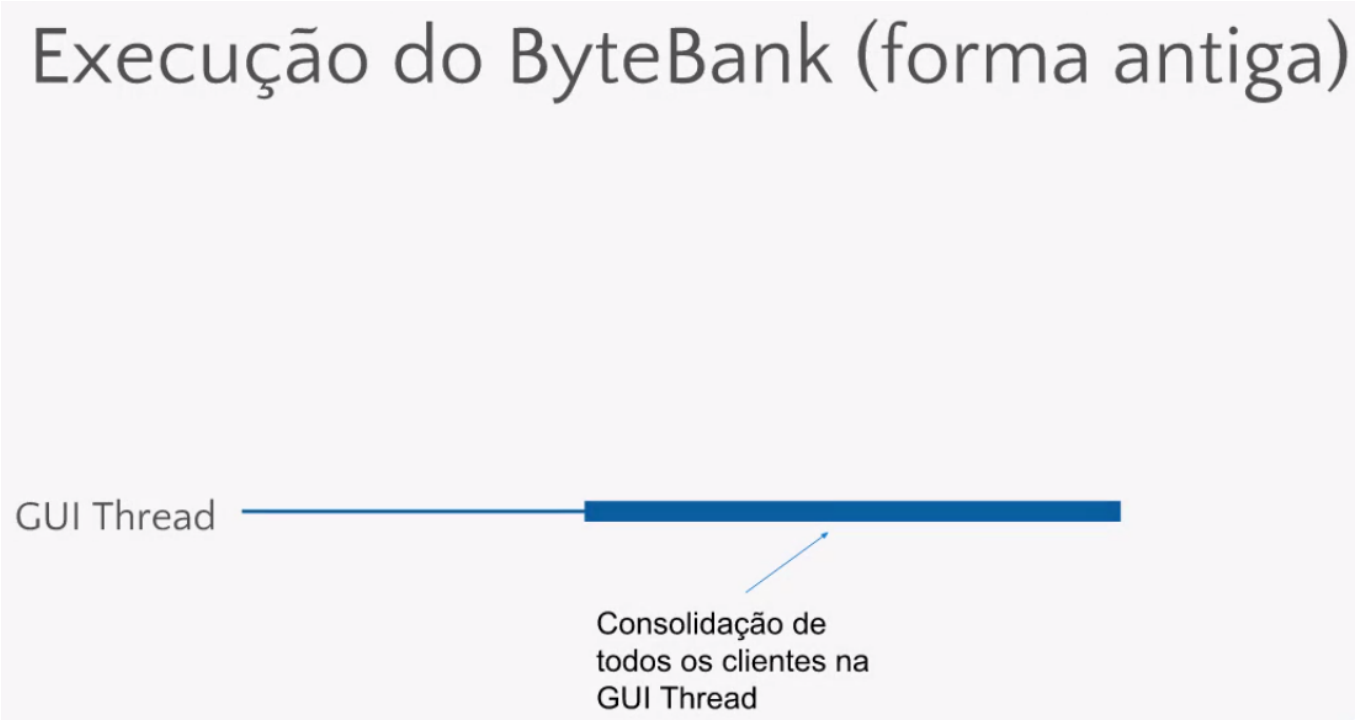
Transcrição

Por hora avançamos muito em nosso curso: alteramos a visão antiga de como codificávamos, de forma procedural, aprendemos sobre *threads* e *tasks*. Porém, percebemos que o código está um pouco complicado, feio, como já ocorreu antes. Vamos dar uma revisada no que vimos até aqui, continuar e deixá-lo cada vez melhor.

No início do curso, tínhamos o programa construído de forma antiga, com que todos estão acostumados: uma tarefa após a outra, nada em paralelo, sem pensar em performance ou processador. Ilustrei esta forma antiga como construímos o programa, sendo que a linha representa a linha de execução da aplicação, uma *thread* chamada GUI. Não é Gui porque meu nome é Guilherme, é porque na literatura técnica este termo é bastante recorrente. Trata-se de um acrônimo que significa *Graphical User Interface*, sempre associado à *thread* principal, que cuida da interface gráfica, a primeira que a aplicação possui quando começa a ser executada.

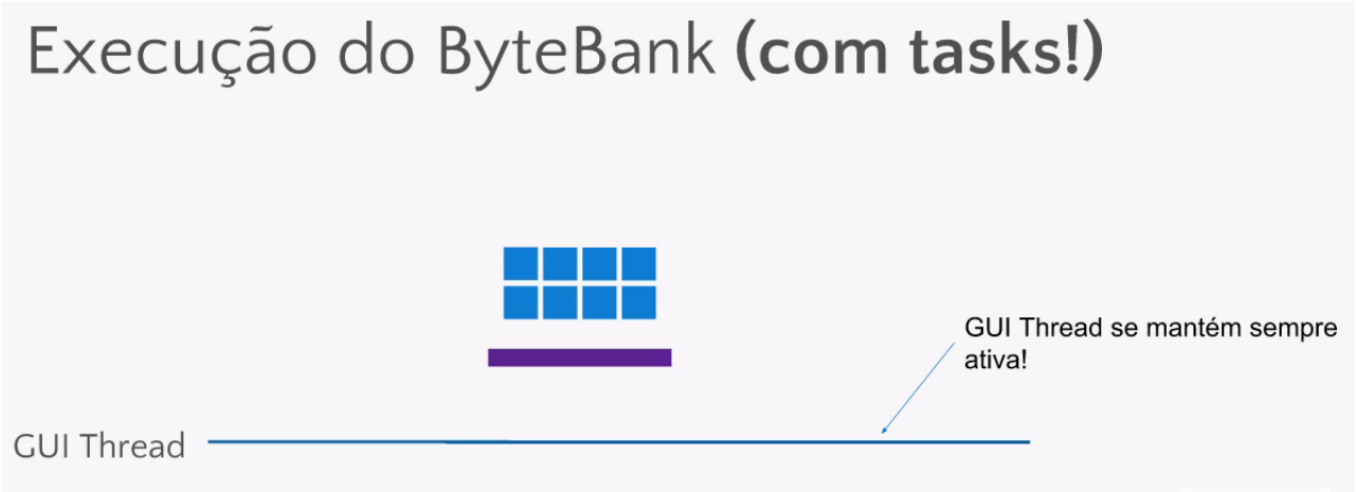


Nesta forma antiga, portanto, o usuário clica em "Fazer Processamento", e o que ocorre é que aquele trabalho pesado do processador não envolve simplesmente clicar em um botão, ou tem a ver apenas com uma animação que sobe ou desce. Deste modo, a consolidação foi representada por uma linha mais grossa, travando qualquer outro evento da *thread* principal, a GUI *Thread*. Assim, a aplicação fica com aparência de estar travada, deixando de responder a cliques ou eventos do mouse, e o usuário não fica feliz com isto.



Diante disto, fizemos melhorias na aplicação utilizando-se *threads*, através das quais determinamos qual fará que trabalho, construindo-as manualmente. Isto nos causou, porém, mais este trabalho de decisão de qual *thread* fará o quê, de pensar em quantos núcleos a máquina possui no momento de execução. Mesmo assim, com estas *threads*, a aplicação continua travada, e o usuário continua insatisfeito.

No entanto, ganhamos performance e a aplicação é executada de forma mais rápida. Melhoramos isto com *tasks*, jogando a responsabilidade de definição dos trabalhos e da quantidade de *threads* a serem trabalhadas para o `TaskScheduler` do .NET, construído para tal. Isso facilitou a execução da aplicação, mantendo-se a interface gráfica responsiva. Ou seja, a tela não trava mais enquanto o usuário estiver fazendo um processamento de carga pesada.



Qual é a nossa preocupação atual? Precisamos nos atentar à obtenção do contexto de sincronização da *thread* principal (GUI), encadeando uma nova tarefa com contexto de execução dela ao término de todas as tarefas de consolidação, informando-se ao .NET que a *task* a ser executada no final precisa ser feita no contexto de execução da *thread* principal. Precisamos criar uma nova tarefa, definindo-se que ela precisa ocorrer naquele `TaskScheduler` em todas as tarefas que precisam ser executadas na *thread* principal.

O código voltou a ficar um pouco bagunçado, com várias chaves e encadeamentos, com `taskSchedulerUI` por todo lado, causando preocupações acerca de sua localização exata. Deixaremos o código mais legível criando-se um método que retornará uma lista com tudo aquilo que for processado e consolidado, e poderemos assim deletar algumas variáveis, bem como o `WhenAll` criado anteriormente.

Começaremos construindo um método privado que retornará uma lista de `string` , afinal, é este que utilizamos na lista da interface gráfica. Chamaremos-na de `ConsolidarContas` , com parâmetro `IEnumerable` de `ContaCliente` , nosso modelo de negócio. O processamento será feito, teremos um `var resultado` , uma nova lista de `string` . Usaremos um método `Select` do LINQ que irá mapear cada conta da lista de contas, para uma nova tarefa, utilizando-se também o método `Factory` , o qual fará o trabalho pesado de consolidar a conta.

```
private List<string> ConsolidarContas(IEnumerable<ContaCliente> contas)
{
  var resultado = new List<string>();

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

      Task.WhenAll(tasks);

      return resultado;
}
```

Então, o que precisamos fazer é esperar todas as *tasks*. Utilizaremos um método estático da classe `Task` chamado `WhenAll` . Quando as tarefas forem finalizadas, será retornado o resultado. Podemos deletar a variável `contasTarefas` e a lista de resultado antes da atualização do *view*, acrescentando-se em outro lugar:

```
var inicio = DateTime.Now;

var resultado = ConsolidarContas(contas);
```

Parece que algo está estranho, pois estamos retornando uma lista vazia. Até o método `ConsolidarContas` retornar, nenhuma tarefa terá o processamento terminado. Na realidade, o que vamos retornar neste método não será uma lista de `string`, e sim uma tarefa que retorna uma lista de `string`. Ainda não vimos uma tarefa com retorno, mas isto é possível, sendo do mesmo tipo, porém genérico. Vamos utilizá-la como uma lista. Teremos uma *task* e, entre os sinais de maior e menor, indicamos o retorno da tarefa correspondente, no caso, uma lista de `string`.

```
private Task<List<string>> ConsolidarContas(IEnumerable<ContaCliente> contas)
```

Como retornaremos uma tarefa que por sua vez retorna uma lista e que, na verdade, precisa esperar todas as outras tarefas? O `WhenAll` já vai esperar a execução de todas as tarefas. E ainda temos o método `ContinueWith`, o qual possibilita duas sobrecargas: uma normal, que retorna uma *task*, e outra genérica, que retorna uma *task* com retorno de tipo genérico. Vamos usar esta construção, e ela receberá um *delegate*, uma ação, que irá retornar o resultado. Como estamos executando esta tarefa de forma encadeada em outra (aquela que espera todas as demais serem executadas), neste momento sabemos que o resultado está populado, bastando retorná-lo.

```
return Task.WhenAll(tasks).ContinueWith(t =>
{
  return resultado;
  });
```

Podemos deletar `var resultado = ConsolidarContas(contas);` e substituir o `Task.WhenAll(contasTarefas)` por `ConsolidarContas(contas)`. Sempre que temos uma tarefa encadeando outra, por parâmetro, recebemos a tarefa anterior, finalizada. Agora veremos por que isto é importante. Não se trata de uma tarefa qualquer, e sim com retorno de lista de `string`. Teremos portanto uma propriedade disponível chamada `task.Result`, pois as tarefas que possuem retorno são de classe genérica.

```
var inicio = DateTime.Now;

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

O código ficou melhor, conhecemos um novo tipo de *task*, agora vamos verificá-lo em execução. Apertaremos em "*Start*", "Fazer Processamento" quando a app abrir, e vemos que a tela continua respondendo. Conferindo o Gerenciador de Tarefas, vemos que a CPU está sendo bastante utilizada, e a aplicação continua fazendo o processamento de forma rápida sem o uso de *tasks*, *threads*, e aquela forma procedural de antes. Mesmo assim, ainda nos preocupamos em guardar o contexto de execução da *thread* principal, em informar o .NET para que ele seja usado... Ainda dá para melhorar!