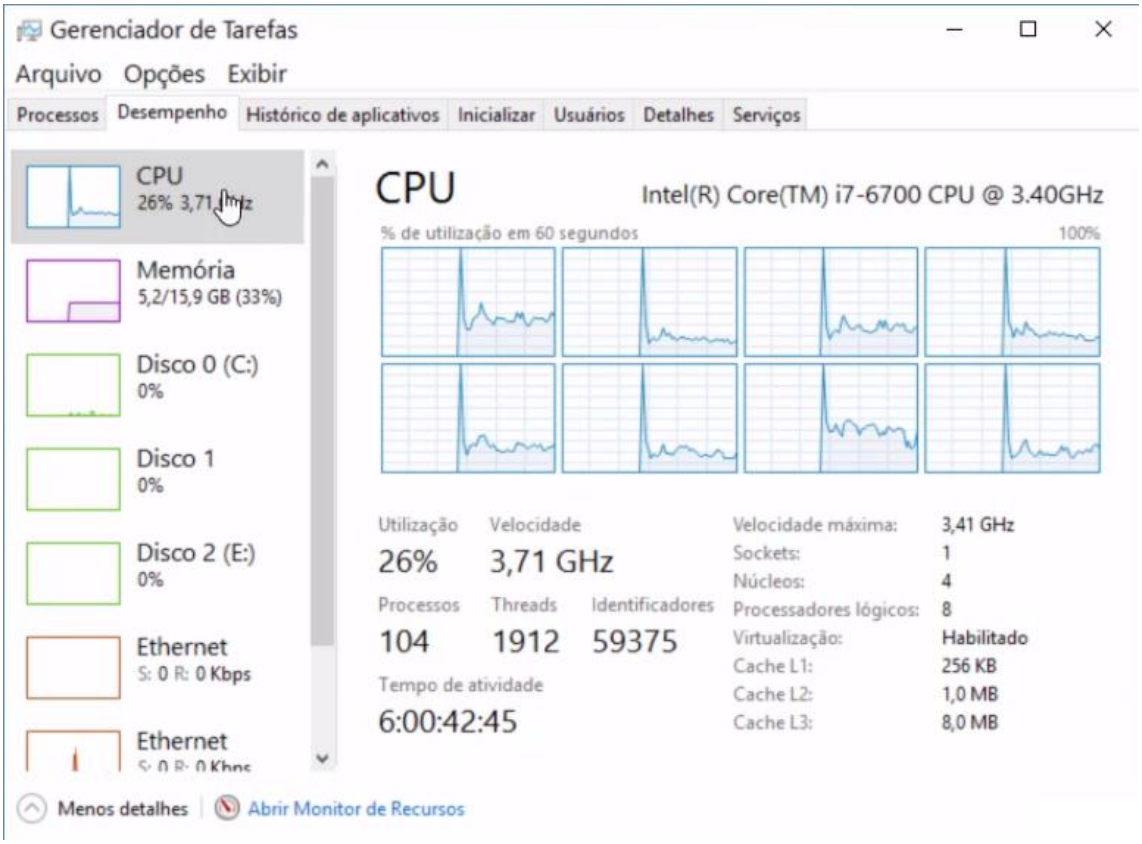
Antes de alterarmos o código para "quebrar" a aplicação de forma a dividi-la em duas partes, voltaremos ao Visual Studio para verificar o que acontece quando a executamos em nossa máquina (a partir da tecla "*Start*"). Abriremos o Gerenciador de Tarefas selecionando a aba "Desempenho", que nos mostra todos os núcleos da máquina e como cada um deles está trabalhando simultaneamente. Neste exemplo, temos uma máquina com oito *cores* trabalhando em mais ou menos `20%`, com vários programas sendo rodados ao mesmo tempo.



Quando executamos o ByteBank e clicamos em "Fazer Processamento", vemos que do *Core* `2` até o último, o funcionamento não se alterou, com `20%`, `30%` de utilização. O único que está realmente fazendo o trabalho é o `CPU 0`. Se pensarmos que a máquina do cliente possui vários *cores*, queremos justamente utilizá-los, dividindo o trabalho entre todos eles.
No código, para cada conta, sempre executaremos uma linha, uma consolidação de cada vez. Se fossemos "quebrar" este código em algum lugar, onde seria? Inicialmente, vamos dividir `contas` em dois. Pensando em executar metade de um processamento destas contas em um *core* e a outra em outro. Mexeremos no código já definindo onde ficará cada metade. Então, vamos criar uma variável denominada `contas_parte1`, a qual armazenará a primeira metade da lista de contas, por meio de um método de extensão de links chamado `Take()`, que recebe, por parâmetro, um número inteiro que representa os `n` primeiros elementos a serem armazenados:

```
var contas_parte1 = contas.Take(contas.Count() / 2);
var contas_parte2 = contas.Skip(contas.Count() / 2);
```

Se vamos usar metade, precisaremos da contagem total da lista, que dividiremos por `2`. Guardaremos na variável `contas_parte2` a segunda metade da lista de contas, utilizando outro método de extensão, chamado `Skip()`, que recebe por parâmetro um número inteiro que representa os `n` primeiros elementos a serem pulados da lista. Neste caso, a primeira parte da lista será pulada e o restante será armazenado.

Ainda não colocamos "a mão na massa". Nota-se que usamos muito o termo "linha de execução", que na realidade é a tradução de "***thread***", termo técnico bastante comum na computação quando falamos sobre Paralelismo. Para criarmos uma nova linha de execução em uma aplicação, é possível fazê-lo com o objeto *thread*, pois é assim que ele é representado no .NET:

```
Thread thread_parte1 = new Thread(() =>
{
  foreach (var conta in contas_parte1)
  {
    var resultadoProcessamento =
r_Servico.ConsolidarMovimentacao(conta);
    resultado.Add(resultadoProcessamento);
  }
});
```

O construtor *default* dele recebe um *delegate* como parâmetro, função representada pela **expressão lambda**, a ser compilada como função, dentro da qual haverá o código a ser executado na nossa *thread*. O que faremos na `thread_parte1` é um processamento de todas as contas da primeira parte da nossa lista. Chamaremos a lista de `contas_parte1` para cada conta na lista de contas, cujo resultado do processamento (`resultadoProcessamento`) será igual à chamada do serviço (`r_Servico`), a ser armazenando na lista de resultados (`resultadoProcessamento`).

O código acima será executado em uma linha de execução diferente desta, que criou a *Thread*:

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

AtualizarView(lew List<string>, TimeSpan.Zero);

var inicio = DateTime.Now;
```

Para continuarmos, vamos criar a *thread* responsável pelo processamento da parte `2`, de forma totalmente independente da `thread_parte1`:

```
Thread thread_parte2 = new Thread(() =>
{
  foreach (var conta in contas_parte2)
  {
    var resultadoProcessamento =
r_Servico.ConsolidarMovimentacao(conta);
    resultado.Add(resultadoProcessamento);
  }
});
```

O construtor `foreach` que existia antes destas duas *threads* que acabamos de criar pode ser deletado:

```
foreach (var conta in contas)
{
```

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

Executaremos a aplicação para ver se vai funcionar. Apertaremos "*Start*", o ByteBank é aberto, clicaremos em "Fazer Processamento". Nos aparece a mensagem "`Processamento de 0 clientes em 0.0 segundos!`", ou seja, não houve processamento de nenhuma conta. No Visual Studio, repassaremos o que fizemos até então: dividimos cada porção... O código parece correto.
O erro é que quando criamos a *Thread*, não o fazemos no momento em que começamos o processamento. Antes de terminar a função de execução do clique no botão, precisamos pedir ao sistema operacional para que ele comece a executar o código contido em `thread_parte1` e `thread_parte2`. Faltou colocarmos isso! Só criamos estas classes, mostrando qual o código e *delegate* a ser executado.
Para que o trabalho delas seja iniciado, existe um método chamado `Start`, implementado na classe `Thread`, que faz exatamente isso. Portanto, acrescentaremos mais estas linhas:

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

Vamos verificar como a aplicação responde a estas alterações, repetindo aquele procedimento pelo "*Start*". A mensagem mostrada agora é "`Processamento de 0 clientes em 0.1 segundos!`". A aplicação demorou um pouco mais, porém ainda não tem nenhum processamento.
Precisamos perceber que estamos lidando com várias linhas de execução diferentes, partindo para o fim logo em seguida, sem esperar que a `thread_parte1` e a `thread_parte2` terminassem seus trabalhos para usarmos os resultados, mostrando-os aos usuários. A classe `Thread` possui uma propriedade denominada `IsAlive`, que retorna "verdadeiro" quando ela está em execução, e "falso" ao fim de seu processamento. Vamos utilizá-la para ficarmos presos a este método até que as *threads* terminem.