# Macoratti.net C# - Programação Assíncrona com async e await (revisitado) - II

Neste artigo vou revisar os conceitos da programação assíncrona com **async e await** da linguagem C# mostrando como evitar deadlocks.



Continuando o **artigo anterior** agora veremos como evitar o temido **deadlock**.

Se você ainda não sabe um deadlock segue a definição Wikipédia do termo:

*"Um deadlock (interbloqueio, blocagem, impasse) refere-se a uma situação em que ocorre um impasse, e dois ou mais processos ficam impedidos de continuar suas execuções - ou seja, ficam bloqueados, esperando uns pelos outros."*

De uma forma bem simples podemos dizer que em um deadlock um processo A fica bloqueado esperando por dados do processo B, que por sua vez esta bloqueado esperando dados do processo A.

**Recursos Usados neste artigo :**

- VS 2017 Community (update 15.5)

## Evitando DeadLocks quando usar async/await

Vamos usar o projeto **WF_ProgAssinc** criado no artigo anterior.

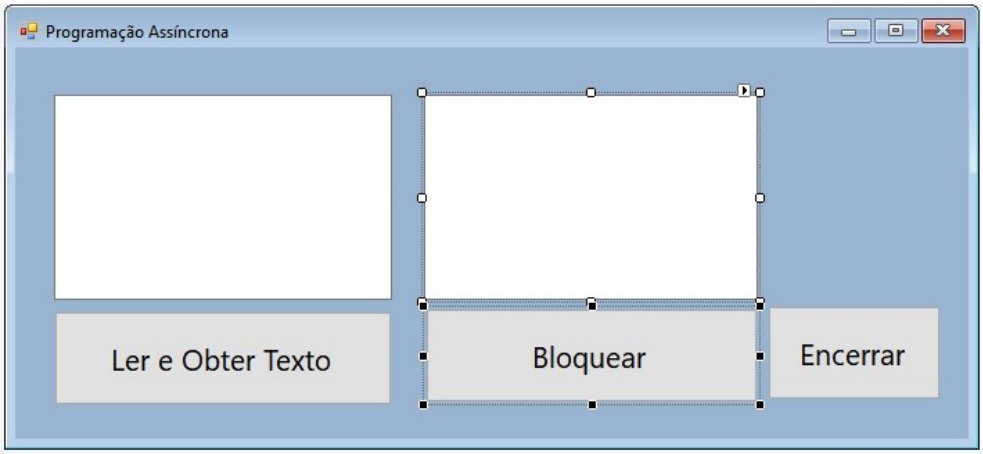
Se você não tomar cuidado pode ser levado a causar um **deadlock** na sua aplicação fazendo que ela trave.

Você pode fácilmente causar um deadlock na sua aplicação misturando código assíncrono com código síncrono.

Vou mostrar isso neste exemplo. Inclua um novo botão de comando e um novo **TextBox** no formulário :

- **1 Button - btnDeadLock**
- **1 TextBox - txtTexto2**

Abaixo vemos o novo leiaute do formulário:




A seguir defina um novo método chamado **LerTexto2Async()** no formulário conforme o código abaixo:

```
private static async Task<string> LerTexto2Async(string valor)
{
    await Task.Delay(TimeSpan.FromMilliseconds(8000));
    return $"Olá, {valor}";
```

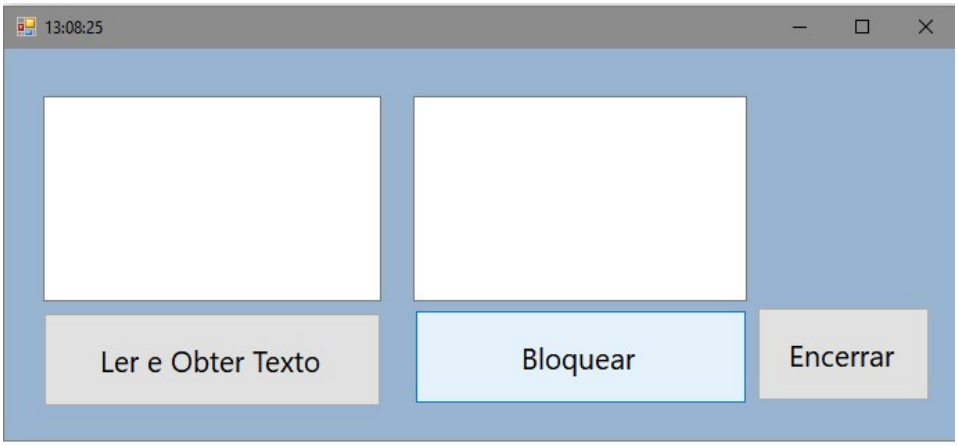
```
}
```

Este método é um método assíncrono que espera por 8 segundos para retornar uma valor string concatenado, estamos usando a interpolação de strings, com o parâmetro valor que é uma string.

Para chamar este método inclua o código abaixo no evento **Click** do botão - **Bloquear** :

```
private void btnDeadLock_Click(object sender, EventArgs e)
{
        var valorLido = LerTexto2Async("Macoratti");
        txtTexto2.Text = valorLido.Result;
}
```

Agora execute a aplicação e observe o resultado:




A aplicação vai travar pois entrou em deadlock e assim você não pode realizar nenhuma operação e a hora também deixou de ser atualizada na barra de títulos do formulário.

Para encerrar a aplicação teremos que recorrer a força bruta.

## Então, o que aconteceu aqui ?

Bem, a resposta tem a ver com contextos.

- A aplicação Windows Forms usa um segmento UI e, portanto, o contexto é um contexto de interface do usuário;
- Ao responder a requisições ASP.NET, o contexto é um contexto de requisição ASP.NET.
- Se não for o caso, o contexto do pool de threads será usado.

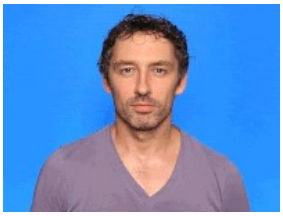
Existe uma thread que é responsável pela interface do usuário: a UI Thread. A UI só pode ser atualizada se for chamada a partir desta thread, portanto, se essa thread estiver bloqueada, o aplicativo não responderá.

Ao clicar no botão **Bloquear** o código bloqueia a thread de contexto porque está aguardando o método **assíncrono** terminar.

O método assíncrono, por outro lado, aguarda pacientemente o contexto para ser liberado e assim concluir. Isso acontece porque ele continua no mesmo contexto que o iniciou.

Assim o método síncrono fica esperando pelo método assíncrono e vice-versa. E temos a aplicação bloqueada.

## Como resolver este problema ?



Existem uma forma de evitar esse problema.

1- A técnica é usar **async/await** para evitar o bloqueio da tarefa no código síncrono.

Vamos aplicar a primeira técnica modificando o código do evento **Click** do botão **Bloquear** incluindo um **async** e um **await** conforme abaixo:

```
private async void btnDeadLock_Click(object sender, EventArgs e)
{
    var valorLido = await LerTexto2Async("Macoratti");
    txtTexto2.Text = valorLido;
}
```

Execute a aplicação e a seguir clique no botão **'Bloquear'** você verá que a aplicação agora se mantém responsiva e não ocorre o deadlock.

Outra forma encontrada para evitar deadlocks é usar **ConfigureAwait(false).**

No nosso exemplo usaríamos o **ConfigureAwait(false)** no método **LerTexto2Async()** que ficaria assim:

```
private static async Task<string> LerTexto2Async(string valor)
{
    await Task.Delay(TimeSpan.FromMilliseconds(8000)).ConfigureAwait(false);
    return $"Olá, {valor}";
}
```

Mas essa não é uma boa prática.

Usar **ConfigureAwait(false)** para evitar deadlocks é uma prática perigosa.

Você precisaria usar **ConfigureAwait (false)** para cada **await** no fechamento transitivo de todos os métodos chamados pelo código de bloqueio, incluindo todos os códigos de terceiros.

E estamos conversados. Até o próximo artigo.

**'E o testemunho é este: que Deus nos deu a vida eterna; e esta vida está em seu Filho.(Jesus Cristo)'**
**1 João 5:11**

Referências:


- Seção VB .NET do Site Macoratti.net
- Super DVD .NET - A sua porta de entrada na plataforma .NET
- Super DVD Vídeo Aulas - Vídeo Aula sobre VB .NET, ASP .NET e C#
- Super DVD C# - Recursos de aprendizagens e vídeo aulas para C#
- Seção C# do site Macoratti.net
- Seção ASP .NET do site Macoratti .net
- Curso Básico VB .NET - Vídeo Aulas
- Curso C# Básico - Vídeo Aulas
- Curso Fundamentos da Programação Orientada a Objetos com VB .NET NEW
- **Macoratti .net | Facebook**
- **macoratti - YouTube**
- **Jose C Macoratti (@macorati) | Twitter**
- C# - Usando Reflection na prática - I - Macoratti.net
- C# - Programação Assincrona - Macoratti.net
- VB .NET - Programação assíncrona com Async e Wait - Macoratti.net
- WPF - Usando async/await para melhorar o desempenho - Macoratti.net
- VB .NET - Usando Tasks - Macoratti.net
- C# - Executando Tasks - Macoratti.net
- C# - Tasks x Threads. Qual a diferença - Macoratti.net


---

José Carlos Macoratti