Aula 4

Programação I

Prof. Alan Matheus Pinheiro Araya

1

Conversa Inicial

2

Threads e Tasks

- **Threads**
- **Tasks**
- **Async/Await**

3

Introdução a Threads

4

Funcionamento das Threads

- **Até o início dos anos 2000, era comum computadores conterem apenas um "core" (núcleo)**
- **Seu computador realiza um processo chamado Preempção para poder executar várias tarefas simultaneamente, mesmo possuindo um "core"**
- **Round Robin – um algoritmo para definir o tempo de preempção dos processos**

5

Processos e Threads

- **Processos são como "containers" que executam um programa**
  - **Main Thread**
- **O SO realiza preempção entre "espaços de espera" das Threads**
  - **Multitarefas**
- **Cada processo possui seu próprio contexto de execução**

6

Threads no C#

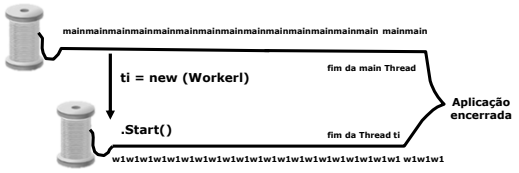
- No C# as Threads são gerenciadas pela CLR
  - Depois pelo Sistema Operacional
- Para manipular Threads vamos usar o namespace:
  - System.Threading

7

- Vamos ver uma Thread funcionando?

8

O trecho de código que vimos terá o seguinte comportamento:



Tempo decorrido >

Fonte: Araya, 2021

9

- Para "esperarmos" uma Thread terminar sua execução antes de continuar determinada tarefa, podemos utilizar o conceito do "Join". Com ele a Thread atual (em que o "Join()" é executado), ficará suspensa até que a Thread que estamos esperando termine

10

```
public ExemploThread2()
{
    Thread t2 = new Thread(Execute);
    t2.Start();
    t2.Join();

    Console.WriteLine("fim da execução");
}

public void Execute()
{
    for (int i = 0; i < 50; i++)
    {
        Console.Write("x");
    }
}
//output no console:
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
fim da execuçao
```

11

- Dizemos que o "Join()" cria uma flag de "sincronização" para as Threads. Colocando a Thread atual em "suspensão" até que a outra termine
- A mensagem "fim de execução" somente foi escrita após o bloco de 50 vezes "x" ser escrito

12

- Para "esperarmos" uma Thread podemos usar o "Join"
- Dizemos que o "Join()" cria uma flag de "sincronização" para as Threads
  - Colocando a Thread atual em "suspensão" até que a outra termine

13

- O Thread.Sleep() é outra forma de esperarmos uma Thread
  - O Sleep coloca a Thread ATUAL em estado de suspenção por um período de tempo
- Executa um "block" na Thread atual
- "Pode" ser mais performático para frações de tempo muito curtas
  - 10-60 milissegundos
  - Gera uma suspenção agendada no OS

14

- Vamos ver um exemplo prático do Join e "Thread.Sleep"?

15

Blocking Threads

- Uma Thread é considerada como "bloqueada" sempre que sua execução é pausada por algum motivo
  - Sleep
  - Join

16

- Toda Thread possui uma propriedade (Enum) ThreadState que representa o seu "status"
- Os principais status de um Thread são:
  - Unstarted
  - Running
  - WaitSleepJoin
  - Stopped

17

Exceções em Threads

- Qualquer bloco try/catch que englobe a declaração da Thread não terá efeito sobre exceções lançadas dentro da Thread

18

Veja o exemplo abaixo, ele não produz efeito algum para tratar o erro, causando uma exceção da Thread principal:

```
public void Exemplo5Exception()
{
    try
    {
        var t5 = new Thread(() =>
        {
            DividePorZero(100);
        });
        t5.Start();
    }
    catch (Exception ex)
    {
        Console.WriteLine("execption tratada!");
    }
}
public int DividePorZero(int valor)
{
    return valor / 0;
}
```

19

Tasks no C#

20

As dificuldades de lidar com Threads

- **Threads são ferramentas de "baixo-nível" para lidar com concorrência no C#. E por conta disso, possuem algumas limitações, em particular (Albahari, 2017, p. 595):**

21

- **Obter valores de retorno**
  - **Não podemos "facilmente" obter valores de retorno de Threads**
- **Agendar novas tarefas encadeadas**

22

As Tasks

- **Comparada a uma Thread, uma Task é uma abstração de nível superior**
- **Representando uma operação simultânea que pode ou não ser apoiada por uma nova Thread**

23

- **Tasks são "composicionais"**
  - **Podemos encadeá-las**
  - **Podem retornar valores**
    - **Mesmo valores "tipados" (<T>)**

24

- Tasks usam Threads do "ThreadPool"
- 1 Task = Thread?
  - Não exatamente...
- O ThreadPool atua como um gestor das Threads em execução pelo Processo
  - É o Framework quem gerencia

25

- Vamos ver alguns exemplos de inicialização de Tasks e compará-las com as Threads

26

```
//Os dois trechos de código abaixo são equivalentes:
Task.Run(() => { Console.WriteLine("Hello World!"); });
new Thread(() => Console.WriteLine("Hello World!")).Start();
```

Observe as diferenças entre as inicializações de Tasks e Threads

```
Task task1 = Task.Run(() =>
{
    Thread.Sleep(2000);
    Console.WriteLine("Task 1 terminando...");
});

Console.WriteLine(task1.IsCompleted); // False
task1.Wait(); // Espera até que a Task finalize
```

Podemos (mas não devemos) fazer Thread.Sleep em Tasks.

Para esperar uma Task finalizar podemos utilizar o .Wait()

```
Task<int> taskNumerosPrimos = Task.Run(() =>
{
    var rangeValores = Enumerable.Range(2, 3000000);
    return rangeValores.Count(v => Enumerable.Range(2,(int)Math.Sqrt(v)-1)
                                   .All(i => v % i > 0));
});

Console.WriteLine("Task em execução...");
Console.WriteLine("Quantidade números primos: " + taskNumerosPrimos.Result);
```

Tasks podem retornar valores.

Ao acessar o ".Result", caso Task esteja em execução forçamos um "Wait()" implícito.

27

Valores de Retorno

- Tasks suportam funções com e sem retorno
  - Também suportam Generics
- Tasks com Type de retorno definido, recebem como parâmetro uma Lambda Function do tipo Func<T>, onde T é o Type esperado no retorno

28

Exceptions em Tasks

- Ao contrário das Threads, as Tasks propagam exceções de maneira conveniente
- Quando uma exceção é lançada
  - A exceção é automaticamente relançada para quem chamar o Wait()
  - Também pode ocorrer quando propriedade Result de uma Task<TResult> for acessada

29

- Pode retornar uma exceção do tipo AggregateException
- Existe para agregar exceções de várias Tasks (encadeadas, por exemplo)
  - IsFaulted
  - IsCanceled

30

Continuação de Tasks

- Uma das maiores vantagens das Tasks é a capacidade que temos de encadeá-las formando uma "sequência de execução"

31

- Vamos ver um exemplo de como encadear algumas Tasks?

32

- A classe Task também nos provê com alguns métodos estáticos que gerenciam um conjunto de Tasks:
- WhenAll
  - Recebe como parâmetro uma ou mais Tasks
  - Retorna uma nova Task que somente estará completa quando todas as Tasks de input estiverem concluídas
    - Mesmo com falha

33

- WhenAny
- Recebe como parâmetro uma ou mais Tasks
- Retorna uma nova Task que estará concluída quando qualquer uma das Tasks de input finalizar
  - Pode ser útil para cenários onde você precisa "testar" condições de forma simultânea e qualquer uma que lhe retornar o valor primeiro ganha

34

Processos Assíncronos e o Await

35

Introdução ao Async/Await

- Todo processo de I/O é uma "longa" espera para o processador
  - Eles "predem" a Main Thread
- Usar Tasks para operações simples é "moroso"
  - Gera muito uso de "Waits"
    - WhenAll, WhenAny, WaitAll etc.

36

- Até agora usamos as Tasks esperando seu término de forma que bloqueamos a Main Thread
- Mas é possível fazer isso e "liberar" a Main Thread para outras atividades enquanto esperamos:
  - Processamento pesado
  - I/O

37

- O Async/Await foi introduzido no C# para facilitar a manipulação de Tasks
- Além disso ele libera nossa Thread para outras operações enquanto ela estiver "esperando" o resultado

38

- Vamos montar um cenário para utilizarmos o Async/Await no C# e compará-lo com "Wait"?

39

Async/Await e o ThreadPool

- O principal benefício que o uso do Async/Await lhe entrega é: "não bloquear a Thread em andamento e permitir que a mesma continue a ser utilizada para outras tarefas da aplicação"
- Para compreender isso, precisamos primeiramente comentar sobre a existência e funcionamento básico do Thread Pool

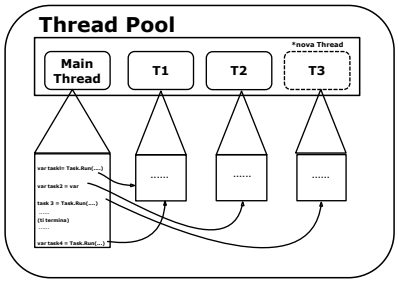
40

- O Thread Pool é uma estrutura gerenciada pela CLR e que toda aplicação .NET tem
- Ela controla:
  - Quantidade de Threads disponíveis
  - Threads em execução
- É o Thread Pool quem aloca uma nova Thread junto ao SO

41

- Alocar uma nova Thread custa alguns microssegundos de contexto e inicialização, além de memória
- O Thread Pool mantém uma lista (Pool) reserva de Threads prontas para a aplicação consumir
  - As Tasks podem iniciar em uma Thread e terminar em outra!
  - A Task termina, mas a Thread continua

42



43

Async/Await e "Bounded Conext"

44

O que é "bounded context"?

- **Faz diferença utilizar "Task.Wait()" ou "await taskX"?**
- **Para entender o impacto de cada um vamos precisar entender os conceitos de:**
  - **I/O Bounded**
  - **CPU Bounded**

45

- **O termo "bound" vem de "vinculado" ou "anexado"**
- **Quando o utilizamos no contexto de Tasks/multitarefa, estamos falando de "blocking"**

46

- **CPU Bounded:**
  - **Uma Thread (pode estar rodando uma task) com alto processamento de CPU (ex: um processamento intensivo matemático).**
- **I/O Bounded:**
  - **Uma Thread esperando um processo de I/O terminar**

47

- **Vamos montar um cenário para podermos visualizar o I/O Bounded?**

48

Explorando Processos de I/O

49

Processos de I/O

- Como vimos, processos de I/O (input e output) geram "block" em Threads
- Processos que envolvem I/O fazem interface direta com o SO e para o sistema operacional, TODOS os processos de I/O são assíncronos por definição

50

- Exemplos de processos assíncronos que envolvem I/O em nosso dia a dia:
  - Leitura e escrita de arquivos em disco
  - Envio e recebimento de dados pela rede
  - Operações com banco de dados
  - Stream de dados, mesmo em memória, envolvendo grandes volumes
  - Transferências de dados por bluethooh

51

- O que a CLR faz quando, em uma operação síncrona, ela é bloqueada por uma operação de I/O, já que o sistema operacional implementa chamadas assíncronas?
- ELA ESPERA
- Logo seu código ficará "bloqueado" de executar outras coisas, esperando também

52

Streams e o System.IO

- No C# várias das classes que lidam com IO, em especial, fluxo de dados e arquivos, estão no namespace:
  - System.IO

53

- Streams são sequências de dados ordenados (bytes), que representam um fluxo de dados contíguo entre uma origem e destino
- Você pode ler ou gravar dados em uma Stream.
  - Sendo também possível gravar um fluxo (Stream) em um arquivo, por exemplo

54

- Vejamos um exemplo prático de uso de Streams

55

Async "all the way"!?

- Durante o processo de leitura ou escrita de um arquivo, sempre ocorre o "blocking"
- O uso do Async em I/O é preferencial, mas NEM SEMPRE desejável
  - Pois é ligeiramente mais lento

56

- Quando usar o Async/Await?
  - Aplicações Web / Mobile
  - Contexto geral
- Quando não usar?
  - Operações de alta performance e curto espaço de "espera"
    - ✓ Tempo de espera conhecido e curto
  - Programação de baixo nível ou em hardwares limitados

57