PROGRAMAÇÃO I

AULA 4

Prof. Alan Matheus Pinheiro Araya

CONVERSA INICIAL

O objetivo desta aula é explicar e detalhar conceitos sobre threads e tasks no .NET de forma geral. Vamos estudar o que é uma thread para o sistema operacional, como o sistema lida com elas, como o computador lida com o paralelismo, e como o C# pode ajudar a resolver problemas relacionados a multitarefas.

Vamos abordar também o funcionamento e o correto uso do padrão Async Await no C#. Trata-se de um padrão muito útil e também comum em aplicações .NET, em especial no mundo web e mobile.

TEMA 1 – INTRODUÇÃO A THREADS

Até o início dos anos 2000, era comum que os computadores tivessem apenas um “core” (núcleo). Isso não significa, no entanto, que o computador somente pudesse executar uma única tarefa por vez. Mesmo computadores mais antigos e seus respectivos sistemas operacionais já eram capazes de manter multitarefas sem grandes problemas. O conceito de *multitarefa* é diferente do conceito de *multicores,* que por sua vez difere de *paralelismo*. Apesar de todos apresentarem conceitos básicos em comum, vamos entender primeiro como o computador lida com a multitarefa.

O sistema operacional (Windows, MAC ou Linux) realiza um processo chamado *preempção*, para poder executar várias tarefas simultaneamente, mesmo que o computador tenha apenas um “core”. Isso é feito executando os processos (programas e rotinas) de forma rápida, trocando entre eles a cada fração de tempo (20 milissegundos, por exemplo). Como nossa percepção de resposta é muito mais lenta (na ordem de 400/900 milissegundos), nosso computador pode executar muitas preempções antes que pareça que “algo está travado ou lento”. Isso é realizado utilizando uma série de algoritmos, como o famoso “Round Robin” (Johann, 2010a).

Com o advento das máquinas multicores/multiprocessadas, vivenciamos grandes avanços na velocidade com que o sistema operacional é capaz lidar com tarefas simultâneas, isso por que ele pode estar lendo um arquivo grande no disco, executando uma música, e ao mesmo tempo processando uma página da web, cada tarefa em diferentes núcleos, sem precisar realizar “preempção” (pausa) nos processos. É claro que existem muitas outras tarefas rodando em

seu computador, em especial nos dias de hoje. Nossos processadores apresentam às vezes 8 núcleos, e muitos suportam “hyperthreading”, um conceito que pode dividir o núcleo em outra unidade, dobando facilmente a capacidade de multiprocessos da máquina. Se você estiver usando um sistema operacional Windows, pode abrir o gerenciador de tarefas e observar quantos processos e threads seu computador está executando no momento.

Os conceitos de processos e threads são um assunto da matéria de sistemas operacionais. Vamos apenas resumir o conceito, de forma generalista, para que possamos avançar com seu uso no C#.

Processos são como containers que executam um programa. Por definição, um processo sempre terá pelo menos uma thread. Uma Thread por sua vez representa uma unidade de trabalho que o sistema operacional utiliza para realizar o processo de preempção e alocação de memória exclusiva, permitindo a execução multitarefas em processos. Cada processo tem um contexto de execução, que não interfere em outros processos. Por exemplo, caso um processo falhe, não afeta outros (Griffiths, 2019, p. 698; Johann, 2010b).

Uma thread é um contexto de execução que pode ser progredido/executado, independentemente dos outros. Cada thread é executada dentro de um processo do sistema operacional, que fornece um ambiente de contexto em que um programa é executado. Com um programa de uma única thread (single-threaded), apenas uma thread é executada no processo. (Albahari; Albahari, 2017, p. 578).

Quando um processo precisa executar mais de uma atividade simultânea, como tocar música e baixar um arquivo pesado da rede, e ainda assim manter a interface responsiva, certamente ele fará uso de threads. Do contrário, alguma dessas tarefas terá que esperar pela outra, causando uma sensação de “travamento” aos olhos de qualquer usuário.

1.1 Threads no C#

No C#, uma thread é gerenciada pela CLR, que por sua vez é gerenciada pelo sistema operacional. Afinal, como definimos, é o sistema operacional que escolhe quando executar cada uma das milhares de threads, das centenas de processos que estão em execução no computador.

Para manipular threads, é necessário referenciar o *namespace*: "***System.Threading***".

Vamos analisar um exemplo de uma thread com seu resultado de execução no C#, de modo a compreender os vários detalhes de sua execução. Observe o trecho de código a seguir, que cria uma nova thread e a executa com um loop em sequência. Escrevemos "*main*" na Thread principal (*main* thread); ao mesmo tempo, em outra Thread, escrevemos "w1" (acrônimo de Worker1):

```
public class ExemploThread1
{
    public ExemploThread1()
    {
        //Inicializa uma Thread que irá executar o método Worker1
        Thread t1 = new Thread(Worker1);
        //A Thread somente começa a executar a partir do "start"
        t1.Start();

        //escreve em loop na main Thread a palavra "main"
        for (int i = 0; i < 1000; i++)
        {
            Console.Write("main");
        }
    }

    public void Worker1()
    {
        for (int i = 0; i < 1000; i++)
        {
            Console.Write("w1");
        }
    }
}
```

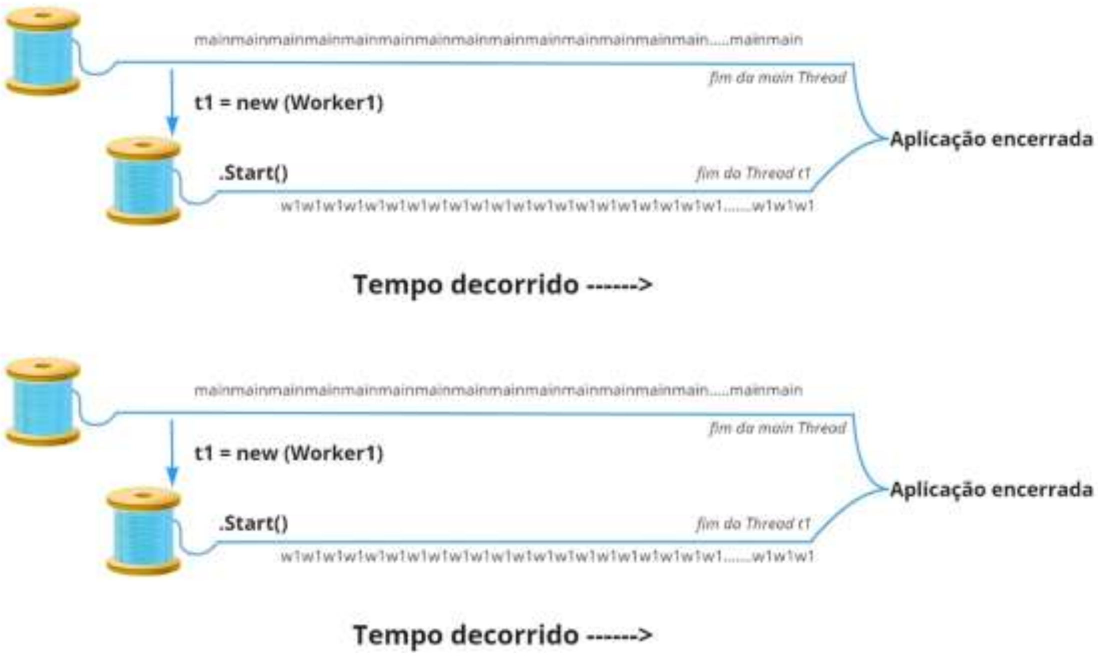
O resultado no console será algo similar a isso:

> *mainmainmainmaainmainmainmainmainmainmainmainmainmain*
> *mainmainmaw1w1w1w1w1w1w1w1w1w1w1w11w1w1w1w1w1w1*
> *w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1ww1w1w1w1w1w*
> *1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w11w1w1w1w1w1w1w1*
> *w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1w1mainmain*
> *mainmainmainmainmainmainmainmainmainmainmainmainm*
> *ainmaiinmainmainw1w1w1w1w1w1w1w1w1w1...*

O que podemos perceber, com este output, além do fato de que os processos realmente ocorreram de forma simultânea, é que houve uma "concorrência" por um recurso especial: o *Console.Write()*. Falaremos sobre esse detalhe mais adiante.

A imagem a seguir ilustra essa execução.

Figura 1 – Execução



Podemos constatar os seguintes fatos e eventos desse exemplo:

- A escrita no console certamente começará pela “main Thread”, pois há um “delay” entre criar uma Thread, pedir para que ela seja executada, e o momento em que de fato começa a execução.
- A thread “t1” não necessariamente completará o seu loop primeiro, mas fará com que a main Thread espere até a conclusão para poder encerrar o processo.
- Para inicializar a thread, nós passamos a referência de um método, utilizando os delegates do C#. Veremos que é possível inicializá-la também com uma Action através de uma Lambda Function.
- A escrita no console ficou dividida em blocos, pois o Console precisa executar uma técnica de “block” de recursos para que gerencie uma concorrência de várias threads que “escrevem” ao mesmo tempo. Veremos esse tema em detalhes nos próximos temas.

1.2 Join e sleep

Existem formas de esperar uma thread executar para que possamos continuar o nosso código. Quando é necessário esperar por completo a execução de uma thread, podemos utilizar o método “Join”:

```
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:
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxfim da execuçao
```

No exemplo, vemos que a mensagem “fim de execução” foi escrita somente após a escrita do bloco de 50 vezes “x”. Isso ocorre porque utilizamos o “Join” para sinalizar para a thread corrente que desejamos “sincronizar” com essa thread em específico (no caso, a main com a “t2”).

Quando utilizamos o “Join”, estamos automaticamente criando uma flag para que o sistema operacional entenda que não precisa executar a nossa thread até que a outra termine. Isso é muito bom, pois poupa recursos de CPU e de troca de contexto da CLR. O Join também pode receber como parâmetro uma quantidade máxima de tempo que ele deve esperar.

Outro modo de “esperar” uma thread é realizando uma “pausa” (Sleep) na thread corrente. Podemos fazer isso em qualquer thread. O mecanismo de Sleep é mais rápido que o Join para a CLR, pois evita trocas de contexto e ao mesmo tempo libera recursos da thread, mas deve ser usado apenas em casos específicos, em que a espera para que uma outra tarefa ocorra seja uma fração de tempo muito curta (alguns milissegundos, por exemplo).

```
public ExemploThread2()
{
    var t3 = new Thread(ExecuteSleep);
    t3.Start();
    Thread.Sleep(100);

    Console.WriteLine("continuando a main Thread...");
    t3.Join();

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

public void ExecuteSleep()
{
    for (int i = 0; i < 50; i++)
    {
        Console.Write("x");
    }
    Thread.Sleep(500);
    Console.WriteLine("fim Thread t3");
}
//Output no console:
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxcontinuando a main Thr
ead...
//fim Thread t3
//fim da execuçao...
```

No exemplo, podemos observar que o ***Sleep*** é um **método estático da classe thread**. Ele executa um "**block**" ou "pausa" na thread corrente em que o trecho de código está inserido. No caso, o método "ExecuteSleep" está executando dentro da thread t3 enquanto os demais trechos estão na "*main Thread*". O método *Sleep* recebe como parâmetro a **quantidade de tempo em milissegundos** durante a qual a thread deve ser suspensa (pode também receber um *TimeSpan*).

1.3 Blocking

Uma thread é considerada como "bloqueada" **sempre que a sua execução é pausada por algum motivo**, seja quando está suspensa, no caso do *Sleep,* ou quando está suspensa esperando uma outra thread através do "Join" (Albahari; Albahari, 2017, p. 580).

Toda thread tem uma propriedade com seu Nome (Name) e status atual (**ThreadState**). O nome de uma thread somente pode ser definido quando ela é inicializada, e não é possível modificá-lo depois. O status atual da thread pode ser consultado através da **propriedade ThreadState**. Essa propriedade retorna um **enumerador** com vários valores de estados possíveis de uma thread. Os

estados mais significativos e úteis são: ***Unstarted, Running, WaitSleepJoin*** e ***Stopped*** (Albahari; Albahari, 2017, p. 581).

Quando uma thread é bloqueada ou desbloqueada, o sistema operacional realiza uma troca de contexto. Isso acarreta uma pequena sobrecarga, normalmente 1 ou 2 microssegundos (Albahari; Albahari, 2017, p. 581).

Uma operação (thread) que passa a maior parte do tempo esperando que algo aconteça é chamada "I**/O-bound**". Por exemplo, fazer o download de uma página da web ou chamar "*Console.ReadLine"*. Operações vinculadas a I/O normalmente envolvem entrada ou saída, mas isso não é obrigatório: a função "*Thread.Sleep"* também é considerada vinculada a I/O. Por outro lado, uma operação que passa a maior parte do tempo realizando o trabalho intensivo da CPU é chamado de "**compute-bounded**" (Albahari; Albahari, 2017, p. 581).

Normalmente, o tipo de operação que mais demanda multitarefas no dia a dia do desenvolvimento é o "I/O-bound" (estudo de um tema mais adiante).

1.4 Variáveis compartilhadas

Observe o exemplo a seguir, em que temos um objeto de "pilha" (*Stack*) que está sendo compartilhado entre a *main Thread* e a *Thread t4* do exemplo. Nesse trecho de código, estamos usando elementos diferentes: **compartilhando variáveis comuns** à instância da classe "ExemploThread3" entre a *main Thread* e a *Thread t4*; e instanciando uma Thread com uma Lambda Function. Perceba que, no corpo da lambda, fazemos uma chamada para o método "ConsomePilha", e depois escrevemos uma mensagem no console.

```
public class ExemploThread3
{
    public bool done = false;
    Stack<int> pilhaCompatilhada = new Stack<int>(new int[] { 0, 1, 2, 3,
4, 5, 6, 7, 8, 9, 10 }); //inicializa a partir de um array

    public ExemploThread3()
    {
        Console.WriteLine("Inicio exemplo 3 - t4");
        Console.WriteLine("ID da main Thread: " + Thread.CurrentThread.Man
agedThreadId.ToString());

        //declara t4 com uma Lambda Function
        var t4 = new Thread(() =>
        {
            ConsomePilha();
            Console.WriteLine("t4 terminou!");
        });
        t4.Name = "Thread t4";
        t4.Start();

        //Sleep na main Thread
        Thread.Sleep(100);

        //Executa o consumo da pilha na main Thread
        ConsomePilha();

        //aguarda t4 finalizar
        t4.Join();
    }

    public void ConsomePilha()
    {
        if (!done)
        {
            while(pilhaCompatilhada.Count > 0)
            {
                Console.Write(pilhaCompatilhada.Pop());
            }
            done = true;
            Console.WriteLine("Acabei! Na Thread: " + Thread.CurrentThread
.Name ?? "main Thread");
        }
        else
        {
            Console.WriteLine("pilha vazia....");
        }
    }
}

//Output no console:
//Inicio exemplo 3 - t4
//ID da main Thread: 1
//109876543210Acabei! Na Thread: Thread t4
//t4 terminou!
//pilha vaiza....
```

No exemplo, você pode notar que criamos uma flag para sinalizar quando a pilha estivesse vazia. Caso alguém tente consumir a pilha já vazia, nosso método escreve uma mensagem avisando que a pilha já se encontra vazia. O que notamos no output do console é que a Thread t4 consegue acessar as variáveis “pilhaCompartilhada” e “done” e manipulá-las. Isso pode ser muito útil e ao mesmo tempo um grande problema se feito de forma incorreta!

1.5 Tratamento de exceções

Qualquer bloco *try/catch* que englobe a declaração da thread não terá efeito sobre exceções lançadas dentro da thread. A seguir, temos um exemplo de código em que o bloco try/catch não tem efeito algum, gerando uma exceção em tempo de execução (Albahari; Albahari, 2017, p. 587):

```
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;
}
```

A mensagem “exception tratada” acima **nunca será exibida**. A Thread t5 irá ocasionar uma **falha no processo e seu programa terá de ser fechado**, pois o bloco *try/catch* não pode ser compartilhado no método que instancia a Thread. Você sempre deve considerar ter um bloco *try/catch* dentro dos seus “métodos principais” executados por threads, a fim de garantir que exceções geradas por elas não afetem o processo. A solução para o exemplo seria mover o bloco *try/catch* para **dentro do método “DividePorZero”**.

TEMA 2 – TASKS NO C#

Threads são ferramentas de “baixo-nível” para lidar com concorrência no C#. Por conta disso, apresentam algumas limitações, em particular (Albahari; Albahari, 2017, p. 595):

- Embora seja fácil passar dados para um thread que você inicia, não existe uma maneira fácil de obter um “valor de retorno” de volta de thread do qual você espera com “Join”. Você terá que configurar algum tipo de campo compartilhado para obter esse retorno. Pode ser bem complexo se a thread lançar uma exceção, capturar e propagar a exceção.
- Você não pode dizer a uma thread para iniciar uma outra coisa quando ela terminar; ao invés disso, você terá que esperar usando o “Join” (bloqueando a sua própria thread no processo). Como exemplo, você não consegue facilmente encadear uma lista de tarefas, que devem ser executadas cada uma em sua thread, mas somente quando a anterior terminar.

Essas limitações desencorajam o uso de threads para opções mais complexas de multitarefas. Em outras palavras, é difícil de compor operações concorrentes maiores combinando umas às outras (algo essencial para a programação assíncrona, como veremos nos próximos temas). Essas limitações, por sua vez, levam a uma maior complexidade no código para a sincronização manual (bloqueio, sinalização com flags etc.), trazendo consigo uma série de outros desafios. O uso direto de threads também tem implicações de desempenho, uma vez que alocar e deslocar threads consome memória e processamento. Se você precisar executar centenas ou milhares de operações de I/O simultâneas, uma abordagem baseada em thread consome centenas ou milhares de MB de memória apenas com o gerenciamento de threads (Albahari; Albahari, 2017, p. 595).

A classe **Task** ajuda com todos esses problemas. Comparada a uma thread, uma *Task* é uma **abstração de nível superior**, representando uma operação simultânea que pode ou não ser apoiada por uma thread. As tarefas são “composicionais” (você pode encadeá-las através do uso de mecanismos de continuação ou de espera em massa). Elas podem usar o pool de threads para diminuir a latência de inicialização. **Contam ainda com um mecanismo para retornar valores computados em suas rotinas**, evitando assim trocas de

contexto e variáveis compartilhadas entre threads (Albahari; Albahari, 2017, p. 595).

Tasks usam threads do “pool” de threads do framework para inicializar tarefas simultâneas. Quem gerencia se uma tarefa (task) será executada dentro de uma nova thread, ou se será utilizada a main Thread, é o próprio framework. Você não precisa se preocupar se uma task está ou não sendo executada por uma nova thread. Você apenas delega uma tarefa simultânea para o framework, e ele escolhe a melhor forma de atendê-lo.

2.1 Principais métodos das tasks

Observe o exemplo de inicialização a seguir, comparando uma inicialização de um task com uma thread:

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

Inicializar um task usando uma Lambda Function é sem dúvida uma das formas mais simples e úteis de se trabalhar com tasks. O método estático Run da classe task automaticamente cria e inicia uma task, como vimos no exemplo.

Outro método muito útil para tratar as tasks é o “Wait”. Como o próprio nome diz, aguarda a task finalizar, de forma similar ao “Join” das 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 finalizePodemo
```

Observe que utilizamos o *Thread.Sleep* **dentro** da Action (método lambda), que passamos para execução na task. Isso pode ser feito do ponto de vista do método lambda (ou qualquer outro que estiver sendo executado por uma task), uma vez que invariavelmente ele estará dentro de uma thread. De antemão, não podemos prever qual thread (uma nova, uma já existente etc.).

A task também contém algumas **propriedades úteis** que podem indicar o seu estado atual, como a *IsCompleted, IsCancelled* ou *IsFaulted*. A task apresenta uma propriedade Status que retorna uma Enum com a sua condição atual dentre uma faixa conhecida de condições. Porém, as propriedades e status **mais comuns já estão bem representadas pelas propriedades citadas**.

2.2 Valores de retorno

As tasks apresentam uma grande vantagem sobre o uso direto de threads para tarefas específicas, como recuperar um dado de rede, fazer um processo de leitura/gravação de arquivos no disco etc. Uma dessas vantagens técnicas é a capacidade de retornar um valor de operação.

Tasks suportam Generics. Isso é muito útil, pois podemos criar tarefas e executá-las apenas no momento correto, sabendo de antemão o tipo de retorno esperado. Tasks com Type de retorno definido recebem como parâmetro uma Lambda Funcation do tipo **Func<T>**, **em que T é o Type esperado no retorno**.

No exemplo a seguir, vamos criar uma task que irá contar a quantidade de números primos entre 0 e 3 milhões. Repare que estamos esperando a task terminar para obter o retorno. A task tem uma propriedade **Result**; se acessada enquanto ela está em execução, **causará um "block" na Thread atual até que a Task termine**, similar ao uso do "Join" em threads.

```
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);
```

2.3 Exceptions em tasks

Ao contrário das threads, as tasks propagam exceções de maneira conveniente. Então, se o código em sua tarefa lançar uma exceção não tratada (ou seja, se a task falhar), a exceção é automaticamente relançada para quem chamar o Wait() ou para quem acessar a propriedade Result de uma Task<TResult>:

```
//Inicia uma nova task que lança uma NullReferenceException:
var task = Task.Run(() => { throw null; });
try
{
    task.Wait();
}
catch (AggregateException aggregateEx)
{
    foreach (var ex in aggregateEx.InnerExceptions)
    {
        Console.WriteLine($"Exception do tipo {ex.GetType()}. Message: {ex.Message
}");
    }
}
```

Repare que uma task pode retornar uma exceção do tipo AggregateException. Esse Type foi criado para que a CLR possa agrupar problemas de várias tasks, em especial em um contexto de paralelismo ou multitarefa.

Você pode verificar se uma task falhou sem relançar a exceção por meio das propriedades IsFaulted e IsCanceled. Se ambas as propriedades retornarem falso, então não ocorreu nenhuma falha; se IsCanceled for verdadeiro, uma OperationCanceledException foi lançada para essa tarefa (pode ocorrer quando a CLR aborta a task por conta de um evento inesperado, como falta de recursos, ou quando a thread foi interrompida de forma abrupta); se IsFaulted for verdadeiro, outro tipo de exceção foi lançada e a propriedade Exception presente na task indicará o erro.

2.4 Continuação/conjunto de tasks

Uma das principais vantagens das tasks é a capacidade de **encadeá-las**, formando uma “sequência de execução”. No exemplo a seguir, podemos ver uma sequência de três tasks encadeadas, utilizando o resultado uma da outra para produzir um novo conjunto de resultados:

```
var task4 = Task.Run(() =>
{
    Thread.Sleep(500);
    return 100;

}).ContinueWith((taskAnterior) =>
{
    Thread.Sleep(500);
    return taskAnterior.Result * 5;
}).ContinueWith((taskAnterior)=>
{
    Thread.Sleep(500);
    return taskAnterior.Result * 5;
});

Console.WriteLine("Resultado encadeado:" + task4.Result);
```

A classe task também nos provê de métodos que gerenciam um **conjunto de tasks**, podendo esperar ou realizar ações sobre um conjunto de Tasks:

- **WhenAll**: recebe como parâmetro uma ou mais tasks, e retorna uma nova task, que estará completa apenas quando todas as tasks de input estiverem concluídas (mesmo com falha);
- **WhenAny**: recebe como parâmetro uma ou mais tasks, e retorna uma nova task, que estará concluída quando qualquer uma das tasks de input finalizar. Pode ser útil para cenários nos quais você precisa "testar" condições de forma simultânea – qualquer uma que lhe retornar o valor primeiro estará "ok".

Repare que os métodos WhenAll e WhenAny retornam uma nova task, por si. Teremos que esperar essa task.

No próximo exemplo, vamos usar o nosso método de cálculo de números primos, levemente modificado, para receber como parâmetro o intervalo de cálculo. Vamos criar três tasks que calculam intervalos diferentes de números primos. Além disso, vamos mensurar o tempo que levaram (ao todo e de forma individual).

Vamos usar a seguinte função para calcular os números primos em um intervalo:

```
public int CalculaPrimos(int intervalo)
{
    var rangeValores = Enumerable.Range(2, intervalo);
    var qtd = rangeValores.Count(v => Enumerable.Range(2,(int)Math.Sqrt(v)-
1).All(i => v % i > 0));

    return qtd;
}
```

No trecho a seguir, observe o uso do "Wait()" após o WhenAll. Também observe o resultado do console, como as tasks calculam de forma individual, com seu próprio "tempo de vida", mesmo dentro do WhenAll. O tempo total decorrido será igual ao tempo gasto pela task mais demorada:

```
//cria um array de cronometros para mensurarmos os tempos decorridos
//no processamento das tasks
var cronometros = new Stopwatch[4];
cronometros[0] = new Stopwatch();
cronometros[1] = new Stopwatch();
cronometros[2] = new Stopwatch();
cronometros[3] = new Stopwatch();

var task1 = new Task<int>(() =>
{
    var qtd = CalculaPrimos(500000);
    cronometros[0].Stop();
    return qtd;
});

var task2 = new Task<int>(() =>
{
    var qtd = CalculaPrimos(2000000);
    cronometros[1].Stop();
    return qtd;
});

var task3 = new Task<int>(() =>
{
    var qtd = CalculaPrimos(4000000);
    cronometros[2].Stop();
    return qtd;
});

//cronometros geral
cronometros[3].Start();

//Vamos iniciar as tasks na ordem inversa
cronometros[2].Start();
task3.Start();
cronometros[1].Start();
task2.Start();
cronometros[0].Start();
task1.Start();

//TASK WHEN ALL -> Wait() - espera todas as Tasks finalizarem
Task.WhenAll(task1, task2, task3).Wait();

//cronometros geral
cronometros[3].Stop();

Console.WriteLine("Conjunto finalizado. Tempo total:" + cronometros[3].Ela
psed.TotalSeconds);
Console.WriteLine($"Tempo Task1: {cronometros[0].Elapsed.TotalSeconds} Res
ultado: {task1.Result}");
Console.WriteLine($"Tempo Task2: {cronometros[1].Elapsed.TotalSeconds} Res
ultado: {task2.Result}");
Console.WriteLine($"Tempo Task3: {cronometros[2].Elapsed.TotalSeconds} Res
ultado: {task3.Result}");

//Output no console:
//Conjunto finalizado. Tempo total:6,5563376
//Tempo Task1: 0,5660922 Resultado: 41538
//Tempo Task2: 2,9045921 Resultado: 148933
```

TEMA 3 – PROCESSOS ASSÍNCRONOS E AWAIT

Como já vimos, o uso de tasks no C# é muito poderoso, pois abstrai do desenvolvedor a necessidade de trabalhar com threads de forma direta, deixando de se preocupar com vários problemas que a manipulação de threads podem trazer diretamente para o contexto do desenvolvimento.

Podem existir cenários em que necessitamos de computação intensiva, ou mesmo de operações de I/O que "demoram" o que chamamos de "**I/O-bound operations**" (pois "prendem" a main Thread, esperando uma operação). Podemos usar tasks para "encapsular" essas operações. Porém, a sintaxe e uso aqui são "morosos" no código, pois requerem o uso massivo de "Task Waits" (WhenAll ,WhenAny, WaitAll e etc.) (Griffiths, 2019, p. 758).

Na versão 5.0 do C#, foi introduzido um novo conceito, chamado de *Async/Await*. Esse conceito não é exclusivo da linguagem C#, sendo muito utilizado em Javascript, Phyton e Kotlin (também apresenta equivalências no Java).

O conceito de Async/Await simplifica o uso de tasks, de forma quase natural, para qualquer operação, em especial para operações nativas do .NET, que suportam rotinas assíncronas. Desde a versão 5 da linguagem, mais e mais funções tem ganhado versões assíncronas, em especial aquelas que lidam com fluxo de dados (*streams*), arquivos, rede (*network*), operações com bancos de dados etc.

Com o avanço do desenvolvimento para web nos últimos anos, e o uso de webservices como modelos de pequenas funções para aplicações, as famosas "Web APIs", muitas linguagens precisaram adaptar as suas características para suportar rotinas assíncronas e liberar recursos de máquina enquanto determinadas operações simplesmente "aguardam" uma resposta – por exemplo, um download de arquivo da nuvem. Ficar esperando, usando por exemplo Thread.Sleep é um grande desperdício de recursos computacionais.

3.1 Async e Await

O grande "problema" do uso das tasks nas abordagens que fizemos durante o Tema 2 é que **temos que esperar pelo fim da task** para **continuar uma rotina síncrona**. Vamos montar um exemplo em que fique claro como isso pode ser trabalho e verboso, usando o *Task.WhenAll*. Para realizar o download

das páginas da web, vamos utilizar o seguinte método simples, que aparece dentro da nossa classe de exemplo, executando de forma síncrona o download do conteúdo de uma determinada url passada como parâmetro:

```
public string ObtemConteudoUrl(string url)
{
    WebClient webClient = new WebClient();
    return webClient.DownloadString(url);
}
```

Observe o exemplo a seguir, em que precisamos fazer o download do conteúdo de páginas da Web. Nesse exemplo, vamos fazer o download dessas páginas de forma assíncrona, utilizando primeiramente tasks. Vamos disparar três tasks simultaneamente e esperar pelos resultados. Além disso, vamos mensurar o tempo decorrido.

```
public void ExecutaTasks()
{
    var task1 = new Task<string>(() =>
    {
        return ObtemConteudoUrl("https://www.bcb.gov.br");
    });
    var task2 = new Task<string>(() =>
    {
        return ObtemConteudoUrl("https://www.gov.br/pt-br");
    });
    var task3 = new Task<string>(() =>
    {
        return ObtemConteudoUrl("https://www.gov.br/saude/pt-br");
    });

    Console.WriteLine("Iniciando tasks para download dos sites....");
    var cronometro = new Stopwatch();
    cronometro.Start();

    task1.Start();
    task2.Start();
    task3.Start();

    //Espera todas as tasks
    Task.WhenAll(task1, task2, task3).Wait();

    //pausa o cronometro
    cronometro.Stop();

    Console.WriteLine($"O total do download das páginas levou: {cronometro.Elapsed
.TotalSeconds} segundos");
    //O total do download das páginas levou: 2,8051061 segundos
}
```

No exemplo, note a necessidade de realizar um Task.WhenAll, independentemente da necessidade de um ou mais tasks. Observe também que escolhemos um método síncrono, que faz o download do conteúdo dentro de uma task, para que ela rode de forma assíncrona e nos retorne o conteúdo.

Agora, vamos usar uma abordagem assíncrona da plataforma .NET. Considere a seguinte modificação do método de download:

```
public Task<string> ObtemConteudoUrlAsync(string url)
{
    HttpClient httpClient = new HttpClient();
    return httpClient.GetStringAsync(url);
}
```

Observe que mudamos, essencialmente, o retorno do método de “string” para “**Task<string>**”. Além disso, o *HttpClient* tem um chamado “*GetString**Async**(string uri)*”, que retorna o conteúdo de uma URL como uma ***Task<string>***. Note o uso da palavra “async” e o retorno de uma *Task<T>*. Isso é tipicamente uma operação assíncrona no C#.

Uma operação assíncrona deve retornar uma *Task / Task<T>,* pois ela é essencialmente uma “promessa”. Ela ainda não aconteceu. Mas você já sabe antecipadamente que, quando ela terminar, terá em mãos um retorno do *Type* T.

Observe como podemos trabalhar com esse retorno, com pequenas alterações em nosso método original:

```
public void ExecutaTasks1()
{
    var task1 = ObtemConteudoUrlAsync("https://www.bcb.gov.br");
    var task2 = ObtemConteudoUrlAsync("https://www.gov.br/pt-br");
    var task3 = ObtemConteudoUrlAsync("https://www.gov.br/saude/pt-br");

    Console.WriteLine("Iniciando tasks para download dos sites, usando GetStringAsyn
c....");
    var cronometro = new Stopwatch();
    cronometro.Start();

    //Espera todas as tasks
    Task.WhenAll(task1, task2, task3).Wait();

    //pausa o cronometro
    cronometro.Stop();

    Console.WriteLine($"O total do download das páginas levou: {cronometro.Elapsed.T
otalSeconds} segundos");
    //O total do download das páginas levou: 2,6077242 segundos
}
```

Destacamos que não foi necessário iniciar as tasks usando o método “Start”.

Agora, vamos supor que você deseje obter o resultado da primeira URL e, apenas depois que ela finalizar com sucesso, obter o resultado da segunda, comparando seus tamanhos de retorno. Veja a seguir um exemplo utilizando as palavras reservadas “**async**” e “**await**”:

```
public async Task ExecutaTasks2Async()
{
    var task1 = ObtemConteudoUrlAsync("https://www.bcb.gov.br");
    var task2 = ObtemConteudoUrlAsync("https://www.gov.br/saude/pt-br");

    Console.WriteLine("Iniciando tasks para download dos sites, usando Async
/Await");
    var cronometro = new Stopwatch();
    cronometro.Start();

    //Espera todas as tasks usando a palavra reservada: await
    //o resultado é uma string retornada pela Task<string>:
    string resultadoTask1 = await task1;
    string resultadoTask2 = await task2;

    //pausa o cronometro
    cronometro.Stop();

    Console.WriteLine($"O total do download das páginas levou: {cronometro.E
lapsed.TotalSeconds} segundos");
    Console.WriteLine($"A página 1 retornou: {resultadoTask1.Length} caracte
res");
    Console.WriteLine($"A página 2 retornou: {resultadoTask2.Length} caracte
res");

    //Output no console:
    //O total do download das páginas levou: 2,7308623 segundos
    //A página 1 retornou: 4756 caracteres
    //A página 2 retornou: 224537 caracteres
}
```

Observe que destacamos de forma proposital as instruções “**async**” e “**await**”. A instrução ***await*** significa basicamente: “**espere a task terminar e me retorne o seu valor**”. A instrução ***async*** significa: “**este método faz uso de funções assíncronas e em algum momento fará uma await de uma task**”.

Métodos que usam o modificador assíncrono são chamados de *funções assíncronas*, pois eles próprios são tipicamente assíncronos. Para entender a razão, vamos analisar como a execução prossegue por meio de uma função assíncrona. Ao encontrar uma expressão “**await**”, **a execução será “suspensa”** e irá **esperar** a task finalizar. A CLR anexa uma **“continuação à tarefa que está em espera”**, como se fosse um “ContinueWith” de uma task, garantindo que, quando a tarefa for concluída, a execução volte ao método/ponto original de onde parou. Se a task gerar uma exceção, ela é relançada; caso contrário, seu valor de retorno é atribuído a quem está esperando a expressão (Albahari; Albahari, 2017, p. 610).

3.2 Entendendo o Async/Await - Threadpool

O uso do padrão *Async/Await* na linguagem C# não traz apenas o benefício de "facilidade" na hora de trabalhar com rotinas assíncronas. Esse padrão faz o uso de uma alocação de contexto e threads de forma inteligente, sendo muito útil em vários cenários do dia a dia.

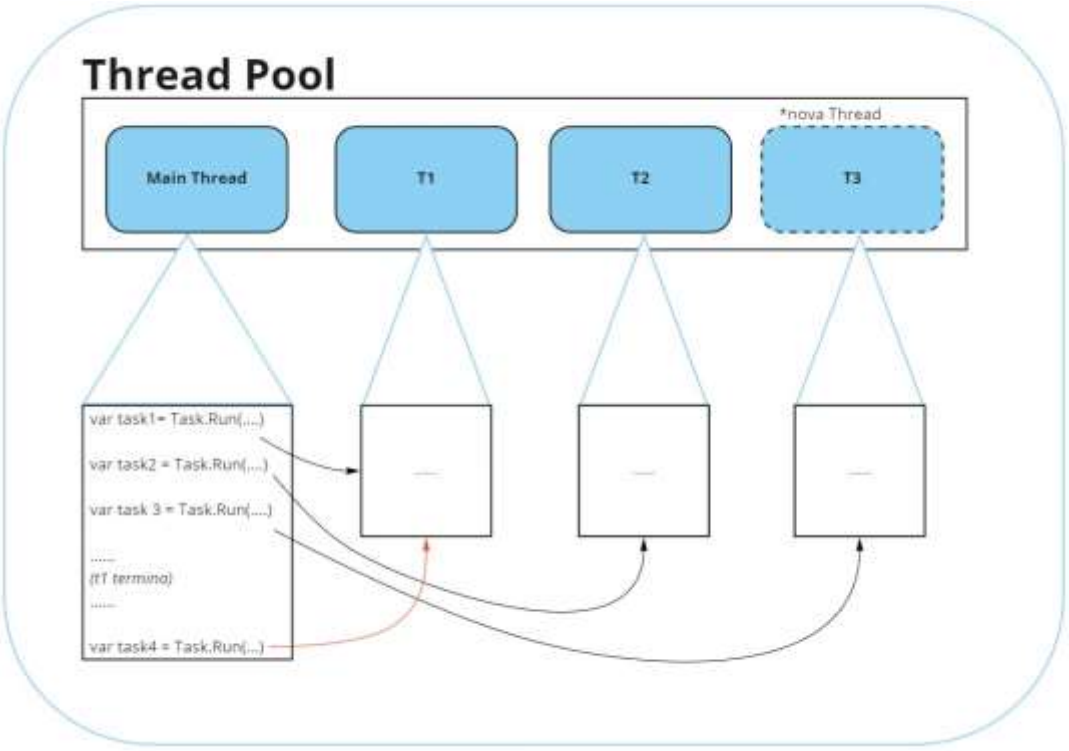
O principal benefício que o uso do *Async/Await* entrega é: não bloquear a thread em andamento e permitir que ela continue a ser utilizada para outras tarefas da aplicação. Para compreender esse ponto, precisamos primeiramente comentar a existência e o funcionamento básico do *Thread Pool*.

O *Thread Pool* é uma estrutura que toda aplicação .NET terá. Ela é gerenciada pela CLR, sendo utilizada para controlar a quantidade de threads que uma aplicação tem à disposição e que está executando. Essa estrutura aloca novas threads junto ao sistema operacional, disponibilizando-as em um "array" de possíveis threads para que a aplicação "consuma".

Alocar uma nova thread para aplicação custa alguns microssegundos de contexto e inicialização, fora a alocação de memória. Utilizar uma "pool" (reserva) de threads pode fazer com que a aplicação apresente desempenho muito melhor, ao consumir threads já criadas, que estão apenas "trocando" de tarefas.

Observe, na imagem a seguir, um exemplo em que nossa aplicação dispara três tasks consecutivas. Vamos supor que a nossa *Thread Pool* contenha apenas duas threads fora a main Thread no pool (aguardando utilização). Nesse caso, uma terceira thread será criada para atender a "task3". Vamos supor que, após a execução de alguns trechos de código da Main Thread, a "task1" termine, liberando a thread. Nesse caso, a "task4" poderá utilizar a mesma thread "t1" que estava atendendo a "task1", passando a executar o código da "task4" na thread "t1".

Figura 2 – Thread pool



O comportamento de compartilhar threads pode ajudar aplicações que demandam execuções rápidas e curtas em um cenário assíncrono, como é o caso de aplicações Web (veremos este tema mais a fundo em aulas futuras). Nesse tipo de aplicação, cada requisição de um cliente (um browser, por exemplo) irá gerar uma nova thread. Com um pool de 5/8 threads, uma aplicação .NET consegue atender facilmente alguns milhares de requisições por segundo, contanto que faça uso de *async/await* e rode em poucos milissegundos (como é comum nesse tipo de aplicação) (Albahari; Albahari, 2017, p. 593).

TEMA 4 – ASYNC/AWAIT E "BOUNDED CONTEXT"

Nos temas anteriores, ressaltamos a importância do uso de threads e utilizamos os termos "CPU Bound" e "I/O Bound". Tais termos são utilizados em referência a códigos que "param ou bloqueiam" a thread, esperando por processamento ou devido à espera de uma operação de I/O.

Vimos isso na prática, em nosso exemplo de download de conteúdo de páginas web utilizando tasks, no Tema 2. Nesse exemplo, nossas tasks ficaram "travadas", esperando que o download das páginas finalizasse, para que então fossem concluídas. Nesse caso, nossa Main Thread também ficou "travada"

esperando a execução do código das tasks. E se fosse necessário manter a Main Thread funcionando?

Nesse caso, o uso do async/await faz muito mais do que facilitar a espera e a criação de tasks. Ele também proporciona a troca de contextos de forma simples e dinâmica. Ao fazer um “await”, a CLR irá colocar a “thread” em um modo de “espera”, liberando-a para atender outros “comandos”/“solicitações”. A verdade é que, quando utilizamos um método com “async”, estamos dizendo para a CLR que aquele método poderá, mas não obrigatoriamente irá, executar uma operação assíncrona. Vamos detalhar esse ponto mais adiante. Primeiramente, vamos elucidar os dois tipos de trabalhos executados por uma task (Pontes, 2021):

- **CPU-Bounded**: nesse caso, o código que está executando é síncrono. Obviamente, do ponto de vista da primeira thread, a que disparou o Task.Run, é como se o código fosse assíncrono, mas apenas porque ela disparou a ação para ser realizada por outra thread, enquanto ela própria pode seguir fazendo outra coisa. **Porém, na prática, alguma thread será bloqueada para o processamento**.
- **I/O-Bounded**: nesse caso, a thread que for elencada para executar aquela tarefa será bloqueada durante a parte síncrona do código, mas quando o código chegar na parte assíncrona (no caso do .net, significa: quando encontrar um await), a execução do código será suspensa, e um callback será chamado quando aquela operação de IO for concluída, para que então o código seja retomado. No caso do .net, isso significa que, quando a operação de IO estiver concluída, uma nova ação será registrada em uma fila das threads, sendo possível que outra thread continue a execução do código. Enquanto a operação de IO está em andamento, a thread que estava executando aquele código fica livre para fazer outras coisas. **Percebam que, nesse caso, enquanto a operação estava em andamento, nenhuma thread está sendo bloqueada**.

Esses cenários fazem muito mais sentido quando lidamos com aplicações web, nas quais o volume de threads será no mínimo igual ao volume de requisições que estão sendo atendidas pela aplicação/site simultaneamente. Nesse caso, enquanto uma thread processa uma requisição de download de um arquivo (é preciso esperar que o arquivo seja transmitido de outro servidor para

o servidor web em questão, para então ser enviado ao cliente), esse código pode ser “congelado” e aguardar a operação, liberando a thread para atender outra requisição de outro cliente, por exemplo. Quando o código retornar, ele **pode não** retornar para a Thread original que o iniciou.

Para exemplificar esse ponto, vamos montar um exemplo bem completo. Vamos criar um “mini servidor” de requisições web, simulando algumas centenas de chamadas Http para uma página. Cada chamada é executada por uma task. Vamos observar como o computador aloca os recursos, em especial o número de threads, e como ocorre o processo de “block” de um thread por ter de esperar um recurso de I/O.

Vamos dividir o exemplo em trechos de blocos de código e explicar cada um isoladamente, para depois demonstrar o resultado final.

```
public class Downloader
{
    public enum Status
    {
        EXECUTANDO,
        FINALIZADO
    }
    public Status StatusRequisicao { get; set; }
    public string ClientId { get; set; }
    public string Resultado { get; set; }
    public Downloader(string clientId)
    {
        ClientId = clientId;
    }
    public Task<string> ObtemConteudoUrlAsync(string url)
    {
       StatusRequisicao = Status.EXECUTANDO;
       HttpClient httpClient = new HttpClient();
       var resposta = httpClient.GetStringAsync(url);
       return resposta;
    }
    public string ObtemConteudoUrl(string url)
    {
       StatusRequisicao = Status.EXECUTANDO;
       WebClient webClient = new WebClient();
       return webClient.DownloadString(url);
    }
}
```

```
public class MiniServer
{
    public List<Downloader> downloaders = new List<Downloader>(1000);

    public void DownloadUrlAsync(string url, string clientId)
    {
        var downloader = new Downloader(clientId);
        downloaders.Add(downloader);

        Task.Run(async () =>
        {
            var resultado = await downloader.ObtemConteudoUrlAsync(url);
            downloader.Resultado = resultado;
            downloader.StatusRequisicao = Downloader.Status.FINALIZADO;
        });
    }

    public void DownloadUrlSync(string url, string clientId)
    {
        var downloader = new Downloader(clientId);
        downloaders.Add(downloader);

        Task.Run(() =>
        {
            var resultado = downloader.ObtemConteudoUrl(url);
            downloader.Resultado = resultado;
            downloader.StatusRequisicao = Downloader.Status.FINALIZADO;
        });
    }
}
```

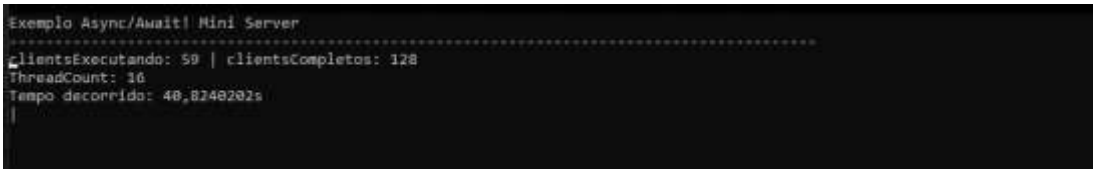
Considere agora a classe (MiniServer) do exemplo como nossa classe "operária". Ela irá executar o download da Url. Ela apresenta dois métodos, um Async e outro Sync, síncrono). Também tem um status, que pode ser: Executando ou Finalizado. Nos indicará ainda se ela terminou ou não de baixar o conteúdo da url.

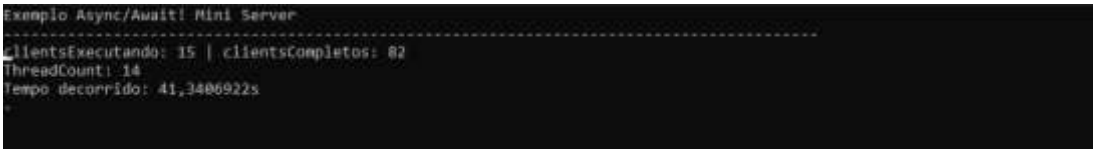
Para executá-la, vamos instanciar uma "MiniServer", com uma lista de "downloaders" e dois métodos, um para download **assíncrono** e outro **síncrono**. Basicamente, os dois fazem "a mesma coisa": inicializam um novo downloader, adicionam ele na fila de "downloaders" e inicializam uma task, através da diretiva **Task.Run**(). Dentro da Lambda Function da task, eles executam o método "ObtemConteudoUrl**Async**" ou "ObemConteudoUrl**Sync**", e modificam o status do downloader para "**Finalizado**". Uma fato **muito** importante: a Lambda Function do método assíncrono tem um "**async** () => { **await** downloader.....}". Isso significa que Lambdas **devem possuir *async*** em sua declaração, sempre que usarem um *await*, tal como qualquer outro método. Isso indica para o .NET que esse método **poderá** fazer uso de uma troca de contextos; ou seja, enquanto o código estiver no "*await*", ele deverá deixar a thread atual "à disposição" para outras operações, aguardando uma operação de CPU ou I/O bound.

```
static async Task Main(string[] args)
{
    Console.WriteLine("Exemplo Async/Await! Mini Server");
    Console.WriteLine("-----------------------------------------------");

    MiniServer miniServer = new MiniServer();
    string urlDownload = "https://www.gov.br/pt-br";
    Stopwatch cronometro = new Stopwatch();
    cronometro.Start();

    for (int i = 0; i < 500; i++)
    {
        //miniServer.DownloadUrlAsync(urlDownload, $"client: {i}");
        miniServer.DownloadUrlSync(urlDownload, $"client: {i}");

         await Task.Delay(100);

        var clientsCompletos = miniServer.downloaders.Count(d => d.StatusRequisicao
== Downloader.Status.FINALIZADO);
        var clientsExecutando = miniServer.downloaders.Count(d => d.StatusRequisicao
 == Downloader.Status.EXECUTANDO);

        Console.WriteLine($"clientsExecutando: {clientsExecutando} | clientsCompleto
s: {clientsCompletos}");
        Console.WriteLine($"ThreadCount: {ThreadPool.ThreadCount}");
        Console.WriteLine($"Tempo decorrido: {cronometro.Elapsed.TotalSeconds}s");
        Console.SetCursorPosition(0, 2);
    }
}
```

No trecho do exemplo, nosso MiniServer faz um loop de 500 iterações. Para cada uma, dispara uma nova tarefa (task) para baixar a url do site do governo federal (“gov.br”). Após cada requisição disparada, basicamente contabilizamos a **quantidade** de downloaders em **execução e finalizados**, além da **quantidade de threads em execução** no processo, e também a quantidade de segundos decorridos no processamento.

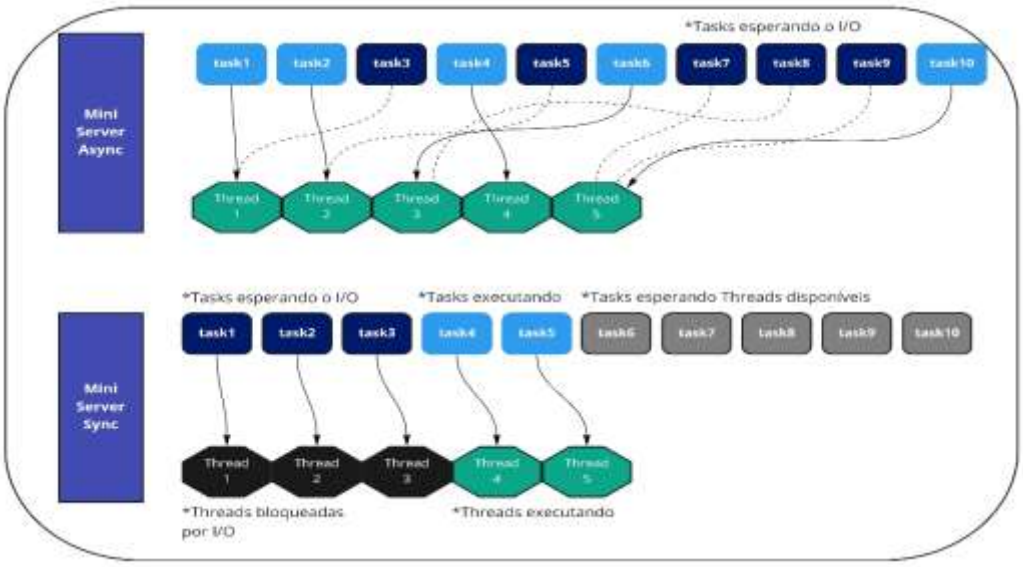
Agora, vamos analisar e comparar os resultados do download usando os métodos assíncronos do MiniServer e do Downloader, além dos métodos síncronos. A primeira imagem a seguir mostra o resultado do processo rodando a função assíncrona; a segunda imagem roda a função síncrona:





Nos dois exemplos, executamos o processo **por 40 segundos** (aproximadamente). Observe a quantidade de **processos completados**, a **quantidade de threads** e a **quantidade de processos em execução** simultaneamente. **Por que temos uma diferença tão grande de processos simultâneos entre os dois modelos**? Como vimos nos temas anteriores, os processos síncronos, quando envolvem threads e tarefas "I/O bounded", fazem um "block" da thread (basicamente um "*Thread.Join()*"), e isso **não libera a thread** para atender **outras tasks**, como ocorre quando fazemos uso do "**await**". A imagem a seguir nos ajudar a entender o cenário proposto de uma forma mais visual.

Figura 3 – Mini server: async e sync



Como podemos ver na imagem, no caso do MiniServer Sync, as tasks são alocadas **uma por thread disponível**. Quando as threads disponíveis acabam, as tasks **ficam esperando** que mais threads sejam **criadas**, ou que sejam serem **liberadas**. Quando uma task desse modelo fica em "waiting" por um processo I/O Bounded, ela bloquea essa thread de atender uma outra task que esteja esperando.

Já no caso do MiniServer Async, as threads estão "**sempre ocupadas**" com tasks que estão sendo executadas. Assim que uma **task executa o "await"**, a CLR **coloca essa Task em "*hold*" (espera)** até que seu I/O finalize. Ainda usa essa mesma thread para **atender outra task**. Eventualmente, uma task

poderá, ao voltar do processo de “await”, continuar a sua execução em outra thread. Essa estratégia nos permitiu, em nosso exemplo anterior, atender **59 processos simultâneos com 16 Threads**, enquanto a estratégia Sync estava atendendo **15 processos simultâneos com 14 threads**.

TEMA 5 – EXPLORANDO PROCESSOS DE I/O

Como vimos nos temas anteriores, processos de I/O (input e output) geram “block” nas threads que estão executando o código que dependem deles. Isso ocorre porque os processos que envolvem I/O fazem interface direta com o sistema operacional e para o sistema operacional. **Todos** os processos de I/O são assíncronos por definição. Isso significa que, para o sistema operacional, quando você salva um arquivo no disco, ele na verdade executa isso de forma assíncrona e “apenas notifica” o seu código de que finalizou.

Para entender melhor esse procedimento, vamos primeiro categorizar alguns tipos de processos de I/O para facilitar o entendimento:

- leitura e escrita de arquivos em disco;
- envio e recebimento de dados pela rede (isso envolve acesos de páginas web e transferências de arquivos/dados);
- operações com banco de dados;
- stream de dados, mesmo em memória, envolvendo grandes volumes;
- transferências de dados por bluethooh;

Entre outras, essas são operações comuns do dia a dia em que estamos lidando com procedimentos de I/O. É importante conceituar que o .NET não faz I/O. Nem o sistema operacional. Quem o realiza é o hardware envolvido na operação. Quando esse mesmo hardware e seu firmware estão transmitindo ou recebendo dados de um meio físico, ocorre o “block” da thread. Isso pode levar alguns microssegundos ou alguns segundos, a depender da operação. Levando em conta que o seu computador realiza algumas bilhões de operações por segundo, mesmo alguns milésimos de segundo podem ser preciosos para os recursos disponíveis.

No C#, possuímos APIs (funções da linguagem) que suportam operações assíncronas e também síncronas de I/O. Como vimos em exemplos anteriores, usamos algumas classes do C# para acesso a rede, de forma síncrona e de forma assíncrona.

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**. De duas formas: simplesmente "ficando em loop" até o processo responder (para operações muito rápidas de I/O), ou colocando a **thread em "sleep" até que o processo termine**. Vamos explorar, com uma série de exemplos, operações síncronas e assíncronas de I/O, em especial de disco.

5.1 Streams e System.io

No C#, temos um conjunto grande de classes que realizam operações de I/O. Muitas estão sob o *namespace* **System.IO**, mas outras estão distribuídas em outros módulos, como System.Net (operações de acesso a rede). Geralmente, ficam na System.IO operações envolvendo acesso a disco (ver mais em: <https://docs.microsoft.com/en-us/dotnet/standard/io/>).

No C#, temos um conceito chamado de *Stream*, que vamos abordar de forma simplificada nesta aula, para nos aprofundarmos mais adiante. *Streams* são sequências de dados ordenados (bytes), que representam um fluxo de dados entre uma origem e um destino. Você pode ler ou gravar dados em uma *Stream*. Também é possível gravar um fluxo (*Stream*) em um arquivo, por exemplo (Albahari; Albahari, 2017, p. 638).

Arquivos são justamente blocos contíguos de dados gravados no disco. A leitura entre o hardware e o sistema operacional ocorre de forma assíncrona. Mesmo linguagens como o C# oferecem bibliotecas e métodos de acesso e gravação síncronos.

Vamos exemplificar como uma *Stream* pode funcionar utilizando-a para gravar um arquivo de teste no disco. Observe, no exemplo de código a seguir, que primeiramente precisamos serializar (converter) nosso texto de string para um array de bytes (byte[]). Depois, criamos uma *FileStream*. Esse é um tipo de *Stream*, quando escreve diretamente em uma estrutura de arquivos (no HD, por exemplo).

No exemplo a seguir, vemos como o comando "*Flush*" é empregado para enviar tudo que está em memória na *Stream* para o HD. É no "*Flush*()", e também no "*Close()"*, que tudo aquilo que escrevemos na *Stream* é enviado para o disco rígido e salvo em arquivo. Durante o processo de escrita dos dados, a nossa

thread fica bloqueada, pois ela precisa esperar o processo de comunicação entre o CLR/sistema operacional/*firmware*/*hardware*.

```
string texto = "um texto para teste das streams!";
//converte o texto em um array de bytes (byte[])
byte[] textoBytes = Encoding.UTF8.GetBytes(texto);

//cria uma Stream para Read/Write
var fileStream = new FileStream("arquivoTeste.txt", FileMode.Create);

//escreve todo o conteúdo do texto (em bye[]) na stream
fileStream.Write(textoBytes, 0, textoBytes.Length);

//cria um novo array com metade do tamanho do anterior
byte[] textoMetade = new byte[textoBytes.Length / 2];

fileStream.Flush(); //escreve o conteúdo atual da stream em disco
fileStream.Position = 0;

//lê parte do texto escrito
fileStream.Read(textoMetade, 0, textoMetade.Length);

fileStream.Close();
fileStream.Dispose();

//escreve o texto pela metade
Console.WriteLine(Encoding.UTF8.GetString(textoMetade));

//Output no console:
//um texto para te
```

Observe como também chamamos o método "*Dispose()"*. Esse método é **muito** importante quando tratamos de *Streams*, pois a interface *IDisposable,* que toda classe de *Stream* implementa, garante que, ao chamar esse método nas classes de *Stream*, a CLR irá efetivamente "limpar" os dados da *Stream* da memória, limpando também suas referências com o sistema operacional. Esse processo garante que o código não deixe "sujeira" na memória. Lembre-se de que *Streams* podem conter blocos muito grandes de dados!

Vamos utilizar, nos próximos exemplos, o padrão "using" do C#. Colocamos dentro dele o nosso bloco de código. Ao sair desse bloco, o C# cuida de chamar para você o método "Dispose" da classe. Você pode encontrar uma boa referência para lidar com interfaces IDisposable no link: <https://docs.microsoft.com/pt-br/dotnet/api/system.idisposable>.

5.2 Blocking IO

Durante o processo de leitura ou escrita de um arquivo, **sempre** ocorre o "**blocking**", pelos motivos que já citamos – **o código precisa esperar o processo físico**. Mas o C#, em suas versões mais recentes, vem trabalhando para que todos os métodos que lidam com I/O apresentem implementações "async". É o que chamamos de "**async all the way**" (assíncrono em todas as partes).

Implementações *async* melhoram a disponibilidade de recursos de um sistema, em especial no mundo mobile e web. No universo mobile, de forma geral, os recursos são muito mais limitados do que em um computador de mesa (desktop). Liberar threads e não gerar *block* em tempo de CPU é fundamental para garantir desempenho, em especial com interações na interface gráfica do sistema/aplicativo.

Deve-se dar preferência às implementações async em **quase** todos os cenários (deve-se levar em conta a plataforma em que o seu código está rodando, assim como outros requisitos). A seguir, deixamos alguns métodos de exemplo para a leitura e a escrita em arquivos, usando as APIs do C# síncronas e assíncronas:

```
var texto = "uma string com um texto de teste simples!";

//Grava um byt[] em um arquivo de modo simples
//Sobre escreve o arquivo se já existir
File.WriteAllBytes("teste.txt", Encoding.UTF8.GetBytes(texto));

//Lê os dados do arquivo gravado
var textolido = Encoding.UTF8.GetString(File.ReadAllBytes("teste.txt"));

Console.WriteLine($"Os texto gravado é igual ao lido = {texto.Length == text
olido.Length}");

//Escreve uma string em um arquivo de forma assíncrona
//utilizando por dentro FileStream com WriteAsync()
await File.WriteAllTextAsync("teste.txt", texto);

//Le todo texto de um arquivo de forma assíncrona
//e converte em string. Utilizando FileStream com ReadAsync()
textolido = await File.ReadAllTextAsync("teste.txt");
```

FINALIZANDO

Esta foi com certeza uma das aulas mais densas e complexas de todo o curso. É recomendável que você faça as aulas práticas e considere os projetos com os códigos de exemplo que desenvolvemos na aula. Eles aparecem como materiais auxiliares da aula.

Nesta aula, falamos sobre conceitos e funcionamento de threads, que são a base para que o sistema não seja “mono-tarefa”. Também abordamos vários aspectos que envolvem a gestão das threads, e como isso pode ser complexo. Por isso, recomendamos o uso de sua “prima mais nova”: as tasks. Tasks abstraem uma série de complexidades ao manipular diretamente uma thread. Tasks ainda proporcionam facilitam o manejo com processos concorrentes e assíncronos. Apresenta uma cadeia de exceção simples e fácil de compreender, e podem ainda ser “esperadas” (await) em métodos “async”, permitindo assim a liberação do recurso enquanto um processo de IO ou CPU intenso está em execução.

Um dos principais objetivos desta aula foi ensinar como é fácil lidar com tasks no C#, e como esse assunto não precisa ser um “bicho de sete-cabeças”. Devemos encarar as tasks como uma “proxy para uma função”, ou seja, como uma “promessa” de que você receberá o retorno <T> quando ela executar. E ela pode executar disparando através do Task.Run() ou simplesmente fazendo uma “await”.

Os princípios aqui demonstrados, se bem praticados e aprendidos, servem de base para outras linguagens, como Kotlin ou Flutter (Dart). Isso porque são princípios “core” para as linguagens modernas.

Não deixe de acessar as referências desta aula e explorar, executar e “debuggar” (depurar) todos os códigos de exemplo do material auxiliar. Em especial, o código de exemplo do nosso MiniServer, feito no Tema 4. Isso vai lhe ajudar a subir de nível no conhecimento da linguagem e do funcionamento do Framework.

Saiba mais

MICROSOFT. **A classe Task C#.** Disponível em: <https://docs.microsoft.com/pt-br/dotnet/api/system.threading.tasks.task>. Acesso em: 10 set. 2021.

MICROSOFT. **Programação Assíncrona com async e await.** Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/async>. Acesso em: 10 set. 2021.

MICROSOFT. **Programação Assíncrona** Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/async>. Acesso em: 10 set. 2021.

MICROSOFT. **Streams.** Disponível em: <https://docs.microsoft.com/pt-br/dotnet/api/system.io.stream>. Acesso em: 10 set. 2021.

MICROSOFT. **Usando threads e threading.** Disponível em: <https://docs.microsoft.com/pt-br/dotnet/standard/threading/using-threads-and-threading>. Acesso em: 10 set. 2021.

REFERÊNCIAS

ALBAHARI, J.; ALBAHARI, B. **C# 7.0 in a Nutshell.** 7. ed. United States of America: O'Reilly Media Inc, 2017.

GRIFFITHS, I. **Programming C# 8.0.** 2.ed. United States of America: O'Reilly Media Inc, 2019.

JOHANN, M. **Algoritmos de Escalonamento.** 2010a. Disponível em: <http://www.inf.ufrgs.br/~johann/sisop1/aula10.scheduling.pdf>. Acesso em: 10 set. 2021.

JOHANN, M. **Threads POSIX**. 2010b. Disponível em: <http://www.inf.ufrgs.br/~johann/sisop1/aula07.threads.pdf>. Acesso em: 10 set. 2021.

PONTES, A. Sincronicidade e Paralelismo. **Linkedin**, 6 abr. 2021. Disponível em: <https://www.linkedin.com/pulse/sincronicidade-e-paralelismo-andr%25C3%25A9-pontes>. Acesso em: 13 de junho de 2021.