UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

SILVIO MANÇANO DE MATTOS JUNIOR

MECANISMOS DE PROGRAMAÇÃO CONCORRENTE EM C#

RIO DE JANEIRO
2021

SILVIO MANÇANO DE MATTOS JUNIOR

MECANISMOS DE PROGRAMAÇÃO CONCORRENTE EM C#

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Profa. Silvana Rossetto

RIO DE JANEIRO
2021

CIP - Catalogação na Publicação

M444m

Mattos Junior, Silvio Mançano de
Mecanismos de programação concorrente em C# / Silvio Mançano de Mattos Junior. -- Rio de Janeiro, 2021.
79 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2021.

1. Concorrência.. 2. Linguagens de programação. 3. C#. 4. NET. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

SILVIO MANÇANO DE MATTOS JUNIOR

MECANISMOS DE PROGRAMAÇÃO CONCORRENTE EM C#

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 11 de Agosto de 2021

BANCA EXAMINADORA:

Silvana Rossetto
D.Sc. (Instituto de Computação - UFRJ)

Presença virtual

Nelson Quilula Vasconcelos
M.Sc. (Instituto de Computação - UFRJ)

Presença virtual

Valeria Menezes Bastos
D.Sc. (Instituto de Computação - UFRJ)

Dedico este trabalho às mais de quinhentos e cinquenta e três mil pessoas vítimas do Corona Vírus *(COVID-19)* no Brasil até o momento. Entre elas, meu grande amigo e ex-aluno de Ciência da Computação na UFRJ, Matheus Pinheiro.

AGRADECIMENTOS

Agradeço primeiramente à minha mãe Márcia Machado, aos meus irmãos Bruno Ignácio e Danielle Ignácio, e ao meu pai Silvio Mattos, por todos os esforços para que este momento fosse possível; à minha orientadora e professora Silvana Rossetto, por toda a paciência e ajuda na construção desse trabalho; agradeço também aos meus amigos que me acompanharam nessa jornada de estudo na UFRJ, Daniel Artine, Leonardo Dagnino, Mateus Villas Boas, Vitor Trentin e William Lacerda; aos meus amigos Igor Fonseca, Kaique Rodrigues e Lucas Carneiro, por todo o apoio na vida pessoal e profissional; e por fim à Camila Simonin, Carlos Ney, e todas as pessoas importantes que contribuíram de alguma forma nessa caminhada.

RESUMO

Este estudo visa explorar os mecanismos de programação concorrente oferecidos pela linguagem C#. São apresentadas soluções de problemas clássicos de concorrência como produtor-consumidor, mostrando o uso dos recursos da linguagem e suas abstrações. Avalia-se o desempenho e a facilidade de uso dos mecanismos de concorrência da linguagem no problema particular de multiplicação de matrizes. Realiza-se um estudo mais aprofundado sobre o funcionamento do async-await, recurso de programação assíncrona oferecido pela linguagem. Por fim, avalia-se a possibilidade e dificuldade de extensão e modificação do escalonador de tarefas padrão da linguagem, base do funcionamento do async-await, para a criação de um escalonador de tarefas com prioridade.

Palavras-chave: Concorrência. Linguagens de Programação. C#. .NET.

ABSTRACT

This study aims to explore the concurrent computing mechanisms offered by the C# language. First it presents concurrent data structures and threading abstractions, then it explains how to use it while solving classical concurrent computing problems like the producer-consumer. The performance of the mechanisms offered by the language are evaluated in the matrix multiplication problem. There is also a deeper study of how async-await works, the asynchronous programming feature offered by the language. Finally, the possibility and difficulty of extending and modifying the standard language task scheduler, which is the basis for the operation of async-await, is evaluated in order to create a priority task scheduler.

Keywords: Concurrency. Programming Language. C#. .NET.

LISTA DE ILUSTRAÇÕES

Figura 1 – Visualização da diferença entre ForEach e ForAll 30
Figura 2 – Gráfico comparação de velocidade de execução da multiplicação de matrizes com diferente quantidade de threads 35

LISTA DE CÓDIGOS

Código 1 Exemplo criação de thread 16
Código 2 Exclusão Mútua - Lock 18
Código 3 Construtores Semaphore 20
Código 4 Exclusão Mútua - Semaphore 21
Código 5 Exclusão Mútua - SemaphoreSlim 23
Código 6 Produtor/Consumidor com BlockingCollection 25
Código 7 Produtor/Consumidor para inserção em massa 27
Código 8 PLINQ - Consulta em lista de forma sequencial e paralela 28
Código 9 PLINQ - Utilização do método ForAll 30
Código 10 Multiplicação de Matrizes versão sequencial 33
Código 11 PLINQ - Multiplicação de Matrizes versão paralela 33
Código 12 Algoritmo Café da Manhã Síncrono 38
Código 13 Algoritmo Café da Manhã Assíncrono 39
Código 14 Realizando extensão da classe TaskScheduler 44
Código 15 Testando escalonador simples 45
Código 16 Método assíncrono para transformação 48
Código 17 Método assíncrono transformado 49
Código 18 Método estático criado na compilação 51
Código 19 Método com duas chamadas assíncronas 51
Código 20 Método para teste de desempenho assíncrono 52
Código 21 Modelo Tarefa com Prioridade 57
Código 22 Gerente do Escalonador de Prioridade - ConsumerThread 58
Código 23 Trabalhador do Escalonador de Prioridade - QueueTask 60
Código 24 Verificação Escalonador com Prioridade 61
Código 25 Método de múltiplas chamadas assíncronas descompilado 68
Código 26 Código para Teste de desempenho assíncrono 71
Código 27 Classe PriorityTaskSchedulerManager 72
Código 28 Classe PriorityTaskSchedulerWorker 74
Código 29 Implementação Fila de Prioridade - Visual Studio Magazine 77

LISTA DE TABELAS

Tabela 1 – Disposição da média, desvio padrão (DP) e percentil 95 (P95), em segundos, dos testes de multiplicação de matrizes para 1 e 2 threads. . 34
Tabela 2 – Disposição da média, desvio padrão (DP) e percentil 95 (P95), em segundos, dos testes de multiplicação de matrizes para 6 e 12 threads. . 34
Tabela 3 – Aceleração do tempo de execução com 2, 6 e 12 threads, em comparação com a versão sequencial 35
Tabela 4 – Disposição das médias de tempo de execução de 6 threads, 12 threads e comparação através da aceleração entre elas 36
Tabela 5 – Disposição da média, erro e razão, em milissegundos, da execução do código 20 para mil, dez mil e cem mil elementos 53

LISTA DE ABREVIATURAS E SIGLAS

CLR	Commom Language Runtime
JIT	Just in Time compilation
POSIX	Portable Operating System Interface
LINQ	Language Integrated Query
PLINQ	Parallel Language Integrated Query
CPU	Central Processing Unit
FIFO	First in First Out
LIFO	Last in First Out
TAP	Task asynchronous programming model
APM	Asynchronous Programming Model

SUMÁRIO

1 INTRODUÇÃO 13

2 COMPUTAÇÃO CONCORRENTE EM C# 15
2.1 ESTRUTURAS BÁSICAS DE CONCORRÊNCIA 15
2.1.1 Threads 15
2.1.2 Locks 17
2.1.3 Semáforos 19
2.1.3.1 Semaphore 20
2.1.3.2 SemaphoreSlim 22
2.1.4 Estruturas de dados concorrentes 23
2.2 PLINQ - BIBLIOTECA DE PROGRAMAÇÃO PARALELA 27
2.2.1 Diferença das Enumerações 29
2.2.2 Análise de Desempenho 32
2.3 PROGRAMAÇÃO ASSÍNCRONA 36

3 DISSECANDO O ASYNC/AWAIT 41
3.1 ESCALONADOR DE TAREFAS 41
3.1.1 Escalonador Padrão 41
3.1.2 Estendendo a classe TaskScheduler 43
3.2 MÁQUINA DE ESTADOS 47
3.3 DESEMPENHO DO ASYNC/AWAIT 52

4 ESCALONADOR 55
4.1 ESTRUTURA DE DADOS PARA FILA DE PRIORIDADE 55
4.2 MODELO DE TAREFA COM PRIORIDADE 56
4.3 LÓGICA INTERNA DO ESCALONADOR 57
4.3.1 Gerente 58
4.3.2 Trabalhador 59
4.4 AVALIAÇÃO 60
4.4.1 Discussão 63

5 CONCLUSÃO 64

REFERÊNCIAS 66

APÊNDICE A – MÉTODO DE MÚLTIPLAS CHAMADAS ASSÍNCRONAS DESCOMPILADO **68**

APÊNDICE B – CÓDIGO PARA TESTE DE DESEMPENHO ASSÍNCRONO. **71**

APÊNDICE C – ESCALONADOR DE PRIORIDADE - GERENTE **72**

APÊNDICE D – ESCALONADOR DE PRIORIDADE - TRABALHADOR **74**

ANEXO A – IMPLEMENTAÇÃO FILA DE PRIORIDADE - VISUAL STUDIO MAGAZINE **77**

1 INTRODUÇÃO

Quando falamos em programação concorrente, estamos falando da implementação de fluxos de execução diferentes dentro de uma aplicação e que podem ser executados de forma simultânea em uma mesma máquina. Com o avanço da tecnologia nos últimos anos, ocorreu uma explosão de dispositivos de múltiplos núcleos no mercado, incluindo *notebooks*, *smartphones*, *tablets*, além de computadores pessoais com processadores *multi-core*. Nesse momento a computação concorrente se torna ainda mais relevante, permitindo que os programas executados possam explorar os recursos das máquinas, além de facilitar a modelagem de problemas que, mesmo sem requisitos de desempenho, se encaixam melhor em uma solução concorrente.

Em conjunto com a evolução do *hardware*, houve também uma maior necessidade das linguagens de programação e ambientes de execução se modernizarem, oferecendo abstrações e mecanismos de programação concorrente de níveis mais altos, tornando mais acessível a utilização dos recursos agora disponíveis, além de contribuir para uma modelagem mais condizente com o problema.

Na programação *multithreading* convencional, o programador precisa criar e disparar de forma explícita os fluxos de execução internos de uma aplicação. Esses fluxos de execução são normalmente escalonados para execução diretamente pelo sistema operacional, seguindo um modelo clássico de execução preemptiva com tempo compartilhado. Além disso, o próprio programador é o responsável por lidar com primitivas de sincronização e problemas de concorrência como *starvations* ou *deadlocks*.

O objetivo deste trabalho é estudar as opções de abstrações e mecanismos de programação concorrente, desde as básicas para a criação de fluxos concorrentes, até as de mais alto nível, oferecidos e gerenciados dentro da linguagem C# e o seu ambiente de execução .NET. São realizados testes de desempenho a fim de avaliar possíveis ganhos ou perdas ao se utilizar desses mecanismos, assim como é avaliada a possibilidade de estender a linguagem, provendo uma implementação própria de um escalonador de tarefas.

Inicialmente é feito o estudo dos mecanismos básicos de concorrência como threads e estruturas de dados concorrentes, assim como primitivas de sincronização como *locks* e semáforos. É feito também um estudo de desempenho da biblioteca *PLINQ*, focada em programação paralela. Posteriormente é realizado um detalhamento mais aprofundado do mecanismo *async/await*, comandos criados para programação assíncrona dentro do C#, e o escalonador de tarefas padrão do .NET. Por último, é feito um estudo da extensibilidade e praticidade na criação de um escalonador de tarefas próprio, baseado em prioridade, para suprir a falta de recursos do escalonador padrão em definir ordenação na execução das tarefas.

O restante deste trabalho está organizado da seguinte forma. Inicialmente, no capítulo

2, são apresentados os mecanismos básicos de programação concorrente da linguagem C#. No capítulo 3 é realizado um detalhamento do mecanismo *async/await*, explicitando as transformações realizadas pelo compilador, assim como o funcionamento do escalonador de tarefas padrão. Continuando, no capítulo 4 é realizada a implementação de um escalonador de tarefas baseado em prioridade, avaliando a extensibilidade da linguagem, assim como trazendo novas funcionalidades que o escalonador padrão não apresenta. Por fim, o capítulo 5 apresenta as conclusões obtidas neste trabalho.

2 COMPUTAÇÃO CONCORRENTE EM C#

C# é uma linguagem de programação moderna, orientada a objetos e fortemente tipada. Programas escritos em C# são compilados para uma linguagem intermediária, e posteriormente são executados pelo CLR *(Common Language Runtime)* do ambiente .NET. O CLR realiza um processo de JIT *(Just in Time compilation)* que converte a linguagem intermediária em instruções de máquina, além de fornecer facilidades como coleta de lixo, gerenciamento de memória e de threads (.NET Team, 2021d). Outros exemplos de linguagens que rodam no CLR são o Visual Basic e F#.

Neste capítulo, são apresentados os mecanismos de concorrência do C#. A seção 2.1 contém as estruturas clássicas de concorrência, com criação de threads, locks, semáforos e coleções concorrentes. Na seção 2.2 é apresentada a biblioteca PLINQ, que tem como objetivo tornar a computação paralela mais acessível e menos suscetível à erros. É apresentado como utilizar os principais métodos da biblioteca e também é feita uma análise de desempenho com um algoritmo de multiplicação de matrizes. Por fim, na seção 2.3 é introduzido o funcionamento de código assíncrono dentro da linguagem e como utilizar esse mecanismo.

2.1 ESTRUTURAS BÁSICAS DE CONCORRÊNCIA

Estruturas básicas de concorrência faz referência às estruturas mais comuns, apresentadas na maioria das linguagens, para a criação e sincronização dos fluxos concorrentes. Nesta seção, são apresentadas algumas dessas estruturas e como elas se comportam e são utilizadas dentro do C#, desde a criação e execução de threads, passando por sincronização dos fluxos com locks e semáforos e terminando com coleções como listas, filas e dicionários *thread-safe*, disponibilizados pela linguagem.

2.1.1 Threads

Em ciência da computação, de acordo com Lamport, uma *thread* é a menor sequência de instruções programadas que podem ser gerenciadas por um escalonador, que normalmente faz parte de um sistema operacional (LAMPORT, 1979).

O C# dá suporte à criação, manipulação e execução de threads através das classes do *namespace System.Threading*. Com elas é possível criar programas *multithreading* de uma maneira mais clássica, onde fica a cargo do programador a criação das *threads* e o uso dos mecanismos de sincronização.

É importante ressaltar que o objeto do tipo *Thread* criado no programa se refere à uma *managed thread*. *Managed thread* é uma thread gerenciada pelo CLR, que é mapeada para uma thread nativa do sistema operacional. Uma *managed thread* só é associada a

uma thread nativa quando ela é iniciada com o método *Start()*. Ou seja, quando o objeto thread é criado mas não iniciado, a *managed thread* é criada dentro do CLR, porém não é associada a nenhuma thread nativa (.NET Team, 2017).

O exemplo 1 abaixo demonstra a utilização básica de threads, que é a criação de fluxos de execução concorrentes. Enquanto a thread principal realiza uma tarefa, a thread filha também está realizando outra tarefa, como visto pela saída do programa:

Código 1 – Exemplo criação de thread

```
public static void MainThreadExamples()
{
    Thread t = new Thread(() => DoNothing("Trabalhadora"));
    Logger.Log($"Thread t \"isAlive\"? {t.IsAlive}");
    t.Start();
    Logger.Log($"Thread t \"isAlive\"? {t.IsAlive}");
    DoNothing("Principal");
    Logger.Log("Thread principal chamando Join()");
    t.Join();
    Logger.Log($"Thread t \"isAlive\"? {t.IsAlive}");
    Logger.Log("Terminando Exemplo");
}

private static void DoNothing(string name, int sleepMili = 100)
{
    for (int i = 0; i < 3; i++)
    {
        Logger.Log($"Thread \"{name}\" fazendo algo...");
        Thread.Sleep(sleepMili);
    }
}
```

Saída do programa 1:

```
Time: 51:54:919 - Thread Id: 1 - Thread t "isAlive"? False
Time: 51:54:933 - Thread Id: 1 - Thread t "isAlive"? True
Time: 51:54:933 - Thread Id: 1 - Thread "Principal" fazendo algo...
Time: 51:54:933 - Thread Id: 4 - Thread "Trabalhadora" fazendo algo...
Time: 51:55:33 - Thread Id: 4 - Thread "Trabalhadora" fazendo algo...
Time: 51:55:33 - Thread Id: 1 - Thread "Principal" fazendo algo...
Time: 51:55:134 - Thread Id: 4 - Thread "Trabalhadora" fazendo algo...
Time: 51:55:134 - Thread Id: 1 - Thread "Principal" fazendo algo...
Time: 51:55:234 - Thread Id: 1 - Thread principal chamando Join()
Time: 51:55:235 - Thread Id: 1 - Thread t "isAlive"? False
Time: 51:55:235 - Thread Id: 1 - Terminando Exemplo
```

É possível observar o identificador da thread *(Thread Id)* na saída do programa. Essa

informação é dada pela propriedade *Thread.CurrentThread.ManagedThreadId* que o CLR dispõe para cada thread no programa.

Para dar início ao trabalho do objeto thread, é necessário chamar o método *Start()*. Este método altera o valor da propriedade *isAlive* para *true*, e faz com que a *managed thread* do CLR seja associada a uma thread do sistema operacional, e seu estado seja atualizado para *running*, indicando que a thread em questão está em execução. Quando o valor desta propriedade é *false*, indica que a thread ou não foi iniciada ou já terminou a sua execução. Um mesmo objeto thread não pode ser reiniciado, lançando um erro caso isso seja tentado. É necessário criar um novo objeto para que seja possível executar novamente aquele pedaço de código.

2.1.2 Locks

Quando dois ou mais fluxos de execução ocorrem ao mesmo tempo numa máquina, é possível haver concorrência por recursos compartilhados, como arquivos ou variáveis. Na maioria dos casos, esse comportamento é prejudicial ao programa, levando a erros como uma escrita sobrescrever a outra, uma leitura ser realizada antes de outro fluxo terminar a escrita, entre outros erros. As primitivas de sincronização foram criadas para lidar com esse tipo de problema, sincronizando o acesso de fluxos concorrentes a recursos compartilhados.

O *lock* é uma primitiva de sincronização de exclusão mútua, delimitando uma seção crítica de maneira que apenas um fluxo de execução acesse essa área por vez. A lógica é que antes de entrar na seção crítica, o código requisite o *lock*, a "chave" para a "fechadura". A partir daí, apenas o fluxo com a chave é executado. Ao final da seção, o fluxo em questão devolve a chave para o sistema, para que assim outros fluxos possam executar a seção crítica.

```
adquireLock()
// código seção crítica, acessando algum recurso compartilhado
liberaLock()
```

No C#, essa primitiva é realizada através da palavra reservada *lock*, que delimita o bloco correspondente a seção crítica. Além disso, é necessário informar ao *lock* qual variável será usada como "chave". No exemplo 2, a variável *sharedCounter* atua como um contador compartilhado entre 2 threads, a "Principal" e a "Trabalhadora". Ambas necessitam acessar e incrementar esse contador 3 vezes, e por isso utilizam de um *lock* para delimitar essa seção crítica.

Código 2 – Exclusão Mútua - Lock

```
public static int sharedCounter = 0;
public static readonly object block = new object();

private static void IncrementCounterLock(string name, int sleepMili =
    100)
{
    for (int i = 0; i < 3; i++)
    {
        lock (block)
        {
            Logger.Log($"Thread \"{name}\" incrementando contador: {++
                sharedCounter}");
        }
        Thread.Sleep(sleepMili);
    }
}
```

Saída do programa 2:

```
Time: 55:16:97 - Thread Id: 1 - Thread "Principal"
incrementando contador: 1
Time: 55:16:112 - Thread Id: 4 - Thread "Trabalhadora"
incrementando contador: 2
Time: 55:16:212 - Thread Id: 1 - Thread "Principal"
incrementando contador: 3
Time: 55:16:212 - Thread Id: 4 - Thread "Trabalhadora"
incrementando contador: 4
Time: 55:16:313 - Thread Id: 1 - Thread "Principal"
incrementando contador: 5
Time: 55:16:313 - Thread Id: 4 - Thread "Trabalhadora"
incrementando contador: 6
Time: 55:16:413 - Thread Id: 1 - Thread principal chamando Join()
Time: 55:16:414 - Thread Id: 1 - Terminando LockExample
```

O bloco delimitado pela instrução *lock* em conjunto com o objeto *block* são criados para garantir essa exclusão mútua no acesso à seção crítica, acessando a variável compartilhada, impedindo que ocorram problemas de concorrência. Caso o mesmo objeto fosse utilizado para garantir a exclusão mútua no acesso a outra variável compartilhada, não faria sentido, e poderia causar lentidão no programa, uma vez que ambos estariam concorrendo pela mesma chave *(block)* sem necessidade, podendo haver duas chaves separadas.

2.1.3 Semáforos

Semáforo é um mecanismo de sincronização que tem sua invenção atribuída ao matemático E.W.Dijkstra, em 1965, e apesar de antigo, continua sendo um dos mecanismos de sincronização mais utilizados na construção de aplicações concorrentes.

Um semáforo pode ser definido como uma variável composta de uma variável contadora e uma fila de fluxos de execução (MAZIERO, 2020). Inicialmente é definido o valor máximo do contador. Após isso, são utilizados dois métodos, um para incrementar e outro para decrementar o valor do semáforo, de forma atômica.

Quando um fluxo chama a operação de decremento, o contador é diminuído em 1, e caso ainda assim seja maior que 0, o fluxo segue sua lógica. Caso o contador esteja zerado, o fluxo fica bloqueado na fila de espera do semáforo. Quando um fluxo chama a operação de incremento, o contador é incrementado de 1, e, caso algum fluxo esteja bloqueado na fila de espera, ele é desbloqueado.

O semáforo pode ser utilizado como um *lock* realizando uma exclusão mútua, quando o valor máximo de seu contador é definido como 1. Porém, seu uso pode ir além, realizando lógicas condicionais baseadas no valor do contador, para atender requisitos específicos do problema. Por exemplo, é possível imaginar um semáforo controlando o número de pessoas numa piscina de um clube. A carrocinha de sorvete do clube não é algo que pode ficar o tempo todo na beira da piscina, para preservar a temperatura dos produtos. É possível então basear o tempo que a carrocinha fica na beira da piscina pelo valor do semáforo. Quando atinge um certo número de pessoas ao mesmo tempo, a carrocinha, em outro ponto do programa, é acionada. Quando esse valor cai conforme as pessoas saem da piscina, a carrocinha é chamada de volta.

Sistemas operacionais que seguem o padrão POSIX, devem implementar dois tipos de semáforos: *named* e *unnamed* (KERRISK, 2006). Semáforos *named* são semáforos nomeados no sistema operacional, e que podem ser compartilhados por diferentes processos. Já semáforos *unnamed* não podem ser utilizados por outros processos, sendo assim, apenas threads do processo que criou o semáforo tem acesso a ele. Essa diferença faz com que semáforos *named* e *unnamed* sejam chamados também de "globais" e "locais", respectivamente.

Em C#, existem duas implementações nativas para semáforos, a *Semaphore* e a *SemaphoreSlim*, ambas do *namespace System.Threading*. A *SemaphoreSlim* é uma implementação mais recente, mais leve, e que apresenta suporte a métodos assíncronos. Porém, através dessa classe não é possível criar semáforos do tipo *named*, ao contrário da *Semaphore*, que têm essa funcionalidade.

2.1.3.1 Semaphore

A classe Semaphore do C# provê três construtores. O primeiro recebe apenas a quantidade inicial e máxima de sinais do semáforo. O segundo recebe o mesmo, porém também é possível passar o nome do semáforo, e portanto criando um semáforo do tipo *named*. E o terceiro contém todos os parâmetros anteriores, e além disso indica em um quarto argumento, booleano, se o semáforo nomeado será criado no sistema ou se já existe. Caso o semáforo nomeado já exista no sistema, o mesmo é retornado pelo método e os parâmetros iniciais serão desconsiderados.

Através dessa classe, é possível criar tanto semáforos locais, através do primeiro construtor citado, como semáforos globais, utilizando os outros dois construtores. Porém, é importante ressaltar que até a data deste trabalho, o CLR não oferece suporte para semáforos nomeados em sistemas Unix, apenas em Win32 (.NET Team, 2016). Quando este código é executado em plataforma Unix, uma exceção do tipo *"PlatformNotSupportedException"* é lançada.

O código 3 demonstra a criação de semáforos pela classe *Semaphore* com todas as três opções de construtores disponíveis. No primeiro é criado um semáforo *unnamed* com valor inicial dois e máximo de dois. No segundo é criado um semáforo *named*, chamado "global1", com valor inicial um e valor máximo três. No terceiro construtor, é criado um semáforo *named*, de valor inicial dois e valor máximo três, chamado "global2". Além disso, no terceiro construtor é passada a variável *created* por referência. Essa variável será preenchida com o valor *true*, caso o semáforo seja criado, ou com o valor *false* caso ele já exista no sistema no momento da chamada do construtor.

Código 3 – Construtores Semaphore

```
// Cria semaforo local, iniciado com valor 2 e maximo de 2
Semaphore local = new Semaphore(2, 2);

// Ambos os semaforos seguintes lancam excecao do tipo System.
   PlatformNotSupportedException ao rodar em Unix

// Cria semaforo global de nome "global1", com valor inicial 1 e maximo
   3
Semaphore global1 = new Semaphore(1, 3, "global1");
// Cria semaforo global de nome "global2", com valor inicial 2 e maximo
   3, e retorna na variavel "created" se o semaforo foi criado no
   sistema
Semaphore global2 = new Semaphore(2, 3, "global2", out var created);
```

O uso de um semáforo para implementar sincronização por exclusão mútua é simples. Após a sua criação, a thread que desejar entrar na seção crítica deve chamar o método *WaitOne()*, e quando essa seção crítica terminar, deve chamar o método *Release()*, para liberar a entrada de outra thread que esteja esperando.

Código 4 – Exclusão Mútua - Semaphore

```
public static void WaitReleaseExample()
{
    Semaphore local = new Semaphore(1, 1);
    for (int i = 0; i < 3; i++)
    {
        new Thread(() =>
        {
            Logger.Log($"esperando...");
            local.WaitOne();
            Logger.Log($"executando!");
            Thread.Sleep(1000);
            Logger.Log($"liberando...");
            local.Release();
        }).Start();
    }
}
```

Saída do programa 4:

```
Time: 32:56:578 - Thread Id: 9 - esperando...
Time: 32:56:578 - Thread Id: 5 - esperando...
Time: 32:56:595 - Thread Id: 9 - executando!
Time: 32:56:578 - Thread Id: 8 - esperando...
Time: 32:57:596 - Thread Id: 9 - liberando...
Time: 32:57:596 - Thread Id: 5 - executando!
Time: 32:58:597 - Thread Id: 5 - liberando...
Time: 32:58:598 - Thread Id: 8 - executando!
Time: 32:59:598 - Thread Id: 8 - liberando...
```

Na função de exemplo *WaitReleaseExample()*, primeiramente um semáforo é criado, com valor inicial um e valor máximo um. Dessa forma, o semáforo é capaz de imitar um *lock*, realizando uma exclusão mútua na seção crítica. É executado um *loop* de tamanho três, que contém a criação e inicialização da thread através do construtor e do método *Start()*, respectivamente. Para a thread, é passado como parâmetro a função a ser executada.

Nessa função é feito um *log* como primeiro comando, e em seguida a thread tenta entrar na seção crítica. Para realizar a sincronia, é utilizado o semáforo criado anteriormente, chamando a função *WaitOne()*, que irá decrementar o contador, caso esteja positivo, e entrar na seção crítica. Caso contrário, a thread é bloqueada nesse comando até que o semáforo a libere. A seção crítica é executada primeiramente pela thread 9, enquanto as threads 5 e 8 ficam bloqueadas. Após a execução, a thread 9 incrementa o semáforo em

um através do método *Release()*, liberando assim outra thread para execução da seção crítica.

É importante ressaltar que ambos os métodos *WaitOne()* e *Release()* apresentam variações. O *WaitOne()* pode receber um valor inteiro, indicando em milissegundos, o tempo máximo de espera pelo semáforo. Enquanto o *Release()* apresenta uma variação onde é possível liberar mais de um sinal com uma única chamada, informando esse valor dentro do método.

2.1.3.2 SemaphoreSlim

A classe *SemaphoreSlim* é uma implementação mais recente de semáforo em C#. Essa classe tem seu comportamento muito parecido com a classe *Semaphore*, com as principais diferenças sendo o não suporte a semáforos nomeados, porém com suporte à espera assíncrona. Além disso, a documentação indica que é uma implementação mais leve, ganhando da implementação da classe *Semaphore* em questões de desempenho.

A *SemaphoreSlim* apresenta os mesmos métodos da classe *Semaphore*, com a adição da versão assíncrona do *WaitOne()*, o *WaitAsync()*. O funcionamento interno e utilização dos métodos assíncronos serão abordados em maior profundidade na seção 2.3. Mas em resumo, o *WaitAsync()* faz com que a espera pelo semáforo possa ser realizada de forma não-bloqueante. Dessa maneira, a *managed thread* que executa esse fluxo é liberada para executar outro fluxo enquanto o semáforo não é liberado. Além disso, por ser assíncrono, o método *WaitAsync()* dispõe de sobrecargas úteis para realização de lógicas extras. É possível passar como parâmetro um *cancellationToken*, que pode cancelar a tarefa que espera pelo semáforo de qualquer outro lugar do código. Além disso, é possível passar um parâmetro também de *timeout*, delimitando um tempo máximo para espera.

O código 5 apresenta uma versão assíncrona do código 4, utilizando a classe *SemaphoreSlim* e o método *WaitAsync()*, que espera de forma assíncrona o semáforo ser liberado para poder executar a seção crítica. Além disso, um *timeout* de no máximo dois segundos é configurado. Caso o semáforo demorasse mais do que isso para ser liberado, o código lançaria uma exceção informando que o tempo máximo para o fluxo ser liberado foi atingido.

Código 5 – Exclusão Mútua - SemaphoreSlim

```
public static void WaitReleaseExample()
{
    SemaphoreSlim local = new SemaphoreSlim(1, 1);
    for (int i = 0; i < 3; i++)
    {
        new Thread(async () =>
        {
            Logger.Log($"esperando...");
            await local.WaitAsync(TimeSpan.FromSeconds(2));
            Logger.Log($"executando!");
            Thread.Sleep(1000);
            Logger.Log($"liberando...");
            local.Release();
        }).Start();
    }
}
```

O código 5 utilizando *SemaphoreSlim* demonstra a diferença entre a classe *Semaphore*, que é a possibilidade de utilizar o *Wait()* de forma assíncrona, com o *WaitAsync()*. Para finalizar, ela também dispõe de uma propriedade *CurrentCount*, que indica o valor atual do semáforo, algo não presente na classe *Semaphore*.

2.1.4 Estruturas de dados concorrentes

Para facilitar o trabalho do programador e incentivar o uso de códigos *multithreading*, o C# disponibiliza estruturas de dados preparadas para concorrência, já sincronizadas e com operações atômicas. As estruturas de dados preparadas para concorrência estão localizadas no *namespace System.Collections.Concurrent* (.NET Team, 2020), e fornecem coleções *thread-safe* para adição e remoção de itens. Isto é, o programador não precisa se preocupar com o uso de mecanismos de sincronização ao utilizar-se das mesmas. Nesse *namespace*, são oferecidas as estruturas *BlockingCollection*, *ConcurrentBag*, *ConcurrentDictionary*, *ConcurrentQueue* e *ConcurrentStack*, todas *thread-safe*. A *ConcurrentBag* é uma estrutura desordenada onde é possível a repetição de itens. O *ConcurrentDictionary* é um dicionário chave-valor, enquanto *ConcurrentQueue* e *ConcurrentStack* são estruturas de fila e pilha, respectivamente.

A *BlockingCollection* (.NET Team, 2021a) é uma estrutura que implementa o paradigma produtor/consumidor, onde existem os métodos *Add()* e *Take()*, para produzir e consumir elementos de maneira *thread-safe*.

Ao criar uma *BlockingCollection*, é possível especificar um limite de dados na coleção. O limite ajuda a controlar o tamanho máximo de dados na memória, e quando ele é

atingido, as chamadas ao método de *Add()* são bloqueadas enquanto pelo menos um dado não for removido da coleção. Além disso, é possível também especificar qual estrutura de dados será utilizada por dentro da coleção. Por padrão, a estrutura de dados utilizada é a *ConcurrentQueue*, garantindo assim um consumo dos dados em ordenação FIFO, porém é possível trocar para a estrutura *ConcurrentStack*, que fará com que os dados sejam consumidos em ordem LIFO.

Na *BlockingCollection*, diversas threads podem, ao mesmo tempo, inserir e consumir da coleção. O código 6 demonstra a utilização da *BlockingCollection*, onde duas tarefas são criadas para inserção na coleção através do método *Add()*, enquanto duas tarefas consomem os valores através do método *Take()*. A coleção é limitada a cinco elementos na sua criação, e cada produtor irá produzir seis elementos. A inserção é atrasada pelo método *Task.Delay()*, para uma melhor visualização da saída do programa.

Código 6 – Produtor/Consumidor com BlockingCollection

```
public static void ProducerConsumerExample()
{
    BlockingCollection<string> bc = new BlockingCollection<string>(5);

    for (int i = 0; i < 2; i++)
    {
        var taskId = i;
        Task.Run(() =>
        {
            int j = 0;
            while (j < 6)
            {
                Logger.Log($"PRODUTOR {taskId}: Inserindo: {taskId}-{j}")
                    ;
                bc.Add($"{taskId}-{j}");
                j++;
                Thread.Sleep(TimeSpan.FromSeconds(1));
            }
        });
    }
    for (int i = 0; i < 2; i++)
    {
        var taskId = i;
        Task.Run(() =>
        {
            while (true)
            {
                var item = bc.Take();
                Logger.Log($"CONSUMIDOR {taskId}: Consumido: {item}");
            }
        });
    }
}
```

Saída do programa 6:

```
Time: 50:42:3 - Thread Id: 7 - PRODUTOR 0: Inserindo: 0-0
Time: 50:42:3 - Thread Id: 5 - PRODUTOR 1: Inserindo: 1-0
Time: 50:42:19 - Thread Id: 8 - CONSUMIDOR 0: Consumido: 1-0
Time: 50:42:19 - Thread Id: 9 - CONSUMIDOR 1: Consumido: 0-0
Time: 50:43:23 - Thread Id: 7 - PRODUTOR 1: Inserindo: 1-1
Time: 50:43:23 - Thread Id: 5 - PRODUTOR 0: Inserindo: 0-1
Time: 50:43:23 - Thread Id: 8 - CONSUMIDOR 0: Consumido: 1-1
Time: 50:43:23 - Thread Id: 9 - CONSUMIDOR 1: Consumido: 0-1
```

```
Time: 50:44:24 - Thread Id: 5 - PRODUTOR 1: Inserindo: 1-2
Time: 50:44:25 - Thread Id: 7 - PRODUTOR 0: Inserindo: 0-2
Time: 50:44:26 - Thread Id: 8 - CONSUMIDOR 0: Consumido: 1-2
Time: 50:44:26 - Thread Id: 9 - CONSUMIDOR 1: Consumido: 0-2
Time: 50:45:23 - Thread Id: 13 - PRODUTOR 1: Inserindo: 1-3
Time: 50:45:23 - Thread Id: 8 - CONSUMIDOR 0: Consumido: 1-3
Time: 50:45:27 - Thread Id: 7 - PRODUTOR 0: Inserindo: 0-3
Time: 50:45:28 - Thread Id: 9 - CONSUMIDOR 1: Consumido: 0-3
Time: 50:46:24 - Thread Id: 7 - PRODUTOR 1: Inserindo: 1-4
Time: 50:46:24 - Thread Id: 8 - CONSUMIDOR 0: Consumido: 1-4
Time: 50:46:28 - Thread Id: 7 - PRODUTOR 0: Inserindo: 0-4
Time: 50:46:28 - Thread Id: 9 - CONSUMIDOR 1: Consumido: 0-4
Time: 50:47:24 - Thread Id: 13 - PRODUTOR 1: Inserindo: 1-5
Time: 50:47:25 - Thread Id: 8 - CONSUMIDOR 0: Consumido: 1-5
Time: 50:47:29 - Thread Id: 13 - PRODUTOR 0: Inserindo: 0-5
Time: 50:47:30 - Thread Id: 9 - CONSUMIDOR 1: Consumido: 0-5
```

Na saída do programa, é possível observar o funcionamento do produtor e consumidor. São inseridos elementos com o formato *"Id do produtor - Id do elemento"*, e esses mesmos elementos são consumidos e escritos no *log* do consumidor. Logo nas primeiras linhas é possível observar a inserção simultânea de ambos os produtores, assim como é possível observar logo na sequência ambos os consumidores imprimirem os elementos consumidos. Apesar da garantia de ordenação FIFO, essa propriedade é difícil de observar na saída do programa, uma vez que após o consumo por cada tarefa, a ordenação do log de saída não é garantida. Porém, é possível observar que nenhum consumidor imprime o mesmo elemento que outro, garantindo o consumo paralelo da estrutura de forma nativa, e também que ao observar os *Ids* dos elementos de apenas um consumidor, a ordem FIFO do mesmo é garantida.

A classe *BlockingCollection* também disponibiliza sobrecargas dos métodos *Add()* e *Take()*. Ambos apresentam versões de tentativa, com nomes *TryAdd()* e *TryTake()*, que ao invés de ficarem bloqueados caso não seja possível inserir ou consumir, retornam um *booleano* indicando se foi possível concluir o método. Além disso, ambos os métodos podem receber como parâmetro um *TimeSpan*, indicando um tempo máximo de espera por um lugar para inserir ou um elemento para consumir.

Uma aplicação comum desse padrão produto/consumidor que pode se utilizar do tempo máximo de consumo, são aplicações de inserção em massa no banco de dados. Diversas threads podem inserir ao mesmo tempo na *BlockingCollection*, enquanto uma thread consumidora fica com a tarefa de agrupar os elementos para inserir todos em apenas uma interação com o banco de dados.

O código 7 imita esse comportamento de inserção em massa utilizando a sobrecarga do método *TryTake()*. O consumidor no caso esperaria no máximo um segundo por um elemento novo na coleção. Caso esse elemento não seja inserido, a inserção em massa é feita com os elementos já consumidos. Além disso, é comum também limitar a quantidade de elementos da inserção. Para isso, os elementos consumidos são inseridos em uma estrutura local, e quando a mesma atinge esse limite (nesse caso dez elementos), o consumo da *BlockingCollection* é interrompido para realizar a inserção em massa, antes de voltar a consumir.

Código 7 – Produtor/Consumidor para inserção em massa

```
Task.Run(() =>
{
    while (true)
    {
        var consumed = new List<string>();
        while (consumed.Count < 10 && bc.TryTake(out var item, TimeSpan.
            FromSeconds(1)))
        {
            consumed.Add(item);
        }

        Console.WriteLine($"CONSUMIDOR: Inserindo em massa: {consumed.
            Count} elementos");
        consumed.Clear();
    }
});
```

2.2 PLINQ - BIBLIOTECA DE PROGRAMAÇÃO PARALELA

A *Parallel LINQ*, ou *PLINQ*, é uma implementação da biblioteca *LINQ*, com foco em programação paralela. A mesma é conhecida por trazer facilidade na manipulação de objetos enumeráveis e por oferecer uma sintaxe mais legível das operações, além da possibilidade de encadeá-las, inspirado na estrutura de linguagens funcionais.

Ambas a LINQ e PLINQ, realizam suas operações usando a estratégia *lazy*, ou seja, só começam a executar após toda a *query* ser montada. A diferença principal é a estratégia da PLINQ de tentar aproveitar ao máximo os recursos do processador, dividindo o conjunto de dados a ser manipulado em partes menores, e executando as manipulações em *worker threads* separadas em diferentes núcleos do processador. Por meio dessa estratégia, a biblioteca pode obter ganhos de desempenho de maneira simples, com poucas modificações de um código já existente.

A PLINQ implementa todos os métodos base da LINQ, e os expõe através da classe *ParallelEnumerable*. Para fazer uso da biblioteca, é necessário informar que determinada

fonte de dados deve ser tratada de forma paralela, e isso pode ser feito através de métodos de extensão, como o *AsParallel()*. Um exemplo simples é o cálculo de quais números são pares dentro de uma lista. A PLINQ ajuda a realizar essa conta de forma paralela, sem esforço adicional, apenas acrescentando o método *AsParallel()* na fonte de dados.

No código 8, é criada a lógica para verificar quais números dentro de uma lista são pares em duas funções diferentes, que serão chamadas em sequência para avaliação. Na primeira função, *GetSequential()*, há a inicialização da lista com números de zero até o valor passado na variável *size*. Após essa criação, é efetuada uma *query LINQ Where*, que recebe como parâmetro a função a ser realizada em cada elemento, no caso, verificando que o número é divisível por dois. E por fim, a *query Select* é efetuada apenas para que o *log* seja efetuado.

Na função *GetParallel()*, é realizado quase inteiramente a mesma lógica, porém com a diferença que no final da criação da lista, é chamada a função *AsParallel()* mencionada anteriormente. Essa função não retorna um *IEnumerable<T>*, mas sim uma *ParallelQuery<T>*, na qual é possível chamar as funções da PLINQ para o processamento dos dados.

Código 8 – PLINQ - Consulta em lista de forma sequencial e paralela

```
public static List<int> GetSequential(int size)
{
    IEnumerable<int> sequentialSource = Enumerable.Range(0, size);
    return sequentialSource.Where(x => x % 2 == 0).Select(e =>
    {
        Logger.Log($"Numero par: {e}");
        return e;
    }).ToList();
}

public static List<int> GetParallel(int size)
{
    ParallelQuery<int> parallel = Enumerable.Range(0, size).AsParallel();
    return parallelSource.Where(x => x % 2 == 0).Select(e =>
    {
        Logger.Log($"Numero par: {e}");
        return e;
    }).ToList();
}
```

Saída do programa 8 com uma lista de zero a dez:

```
Time: 58:2:468 - Thread Id: 1 - (Sequencial) Iniciando processamento
de números pares:
Time: 58:2:493 - Thread Id: 1 - Número par: 0
Time: 58:2:493 - Thread Id: 1 - Número par: 2
```

```
Time: 58:2:493 - Thread Id: 1 - Número par: 4
Time: 58:2:493 - Thread Id: 1 - Número par: 6
Time: 58:2:493 - Thread Id: 1 - Número par: 8

Time: 58:2:493 - Thread Id: 1 - (Paralelo) Iniciando processamento
de números pares:
Time: 58:2:508 - Thread Id: 1 - Número par: 2
Time: 58:2:508 - Thread Id: 8 - Número par: 4
Time: 58:2:508 - Thread Id: 10 - Número par: 0
Time: 58:2:508 - Thread Id: 13 - Número par: 6
Time: 58:2:508 - Thread Id: 5 - Número par: 8
```

É possível observar que a versão sequencial, *GetSequential()*, se utiliza da mesma thread principal *(Id: 1)* para executar toda a sua lógica de verificação, imprimindo os números pares em ordem. Já no método *GetParallel()*, onde é utilizada a extensão *AsParallel()* da biblioteca PLINQ, a lógica acontece em threads diferentes *(Ids: 10, 4, 7, 11 e 5)*, executando a *query* de forma paralela. Com isso, a saída do programa traz os resultados de forma não ordenada.

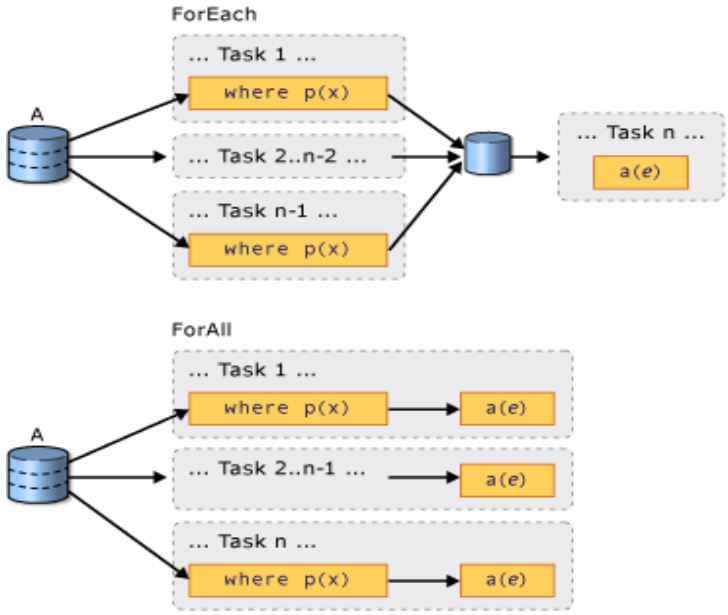
2.2.1 Diferença das Enumerações

Como citado anteriormente, a PLINQ utiliza uma estratégia *lazy*, onde o cálculo só é realizado quando a consulta for requerida. Isso significa que o processamento será feito quando for necessária uma enumeração, através de um *ForEach()* ou métodos como *ToArray()*, *ToDictionary()*, ou, como no exemplo 8, o *ToList()*. Até lá, a consulta permanece com o seu tipo original, *ParallelQuery*$<T>$, onde é possível continuar acumulando outras consultas antes de realizar o processamento.

No caso da PLINQ, quando é realizada a enumeração, é necessário um processo extra em comparação com a LINQ, pois os dados da consulta estão divididos em diferentes threads e portanto devem ser sincronizados. Porém, essa sincronização pode ser atrasada propositalmente pelo programador. É possível que seja de seu desejo manter os dados da consulta separados em cada uma das threads trabalhadoras, e realizar outras operações em cima do resultado anterior. Para que isso seja possível, é necessário utilizar um tipo diferente de enumeração, disponibilizado pela PLINQ através do método *ForAll()*. De forma geral, ao não utilizar o *ForAll()*, o processamento inicial será feito em paralelo mas posteriormente esse resultado será sincronizado na thread principal para realizar os próximos cálculos, enquanto com o *ForAll()* isso será realizado de forma paralela.

Na Figura 1, é possível observar a diferença na utilização entre *ForEach()* e *ForAll()*, onde no *ForEach()*, os dados são sincronizados para a aplicação da próxima transformação, enquanto no *ForAll()* esse processamento continua paralelo. Reutilizando o exemplo 8,

Figura 1 – Visualização da diferença entre ForEach e ForAll



Fonte: docs.microsoft.com, PLINQ (O Operador ForAll)

porém retirando o *ToList()* no final do método, podemos realizar uma operação em cima do resultado da consulta, como somar um aos números pares encontrados, e comparar a saída dos métodos *ForEach()* e *ForAll()*:

Código 9 – PLINQ - Utilização do método ForAll

```
public static void ExemploForAllPLINQ()
{
    Logger.Log("(foreach) Iniciando processamento de numeros pares:");
    ParallelQuery<int> s = EvenNumbers.GetParallel(10);
    foreach (var i in s)
    {
        Logger.Log($"Somando 1 sequencialmente: {i+1}");
    }

    Logger.Log("(ForAll) Iniciando processamento de numeros pares:");
    ParallelQuery<int> p = EvenNumbers.GetParallel(10);
    p.ForAll(e =>
    {
        Logger.Log($"Somando 1 em paralelo: {e+1}");
    });
}
```

Saída do programa 9 com uma lista de zero a dez:

```
Time: 36:59:103 - Thread Id: 1 - (foreach) Iniciando
processamento de números pares:
```

```
Time: 36:59:136 - Thread Id: 12 - Número par: 2
Time: 36:59:136 - Thread Id: 15 - Número par: 4
Time: 36:59:136 - Thread Id: 10 - Número par: 8
Time: 36:59:136 - Thread Id: 14 - Número par: 0
Time: 36:59:136 - Thread Id: 16 - Número par: 6
Time: 36:59:136 - Thread Id: 1 - Somando 1 sequencialmente: 5
Time: 36:59:137 - Thread Id: 1 - Somando 1 sequencialmente: 7
Time: 36:59:137 - Thread Id: 1 - Somando 1 sequencialmente: 9
Time: 36:59:137 - Thread Id: 1 - Somando 1 sequencialmente: 3
Time: 36:59:137 - Thread Id: 1 - Somando 1 sequencialmente: 1

Time: 36:59:137 - Thread Id: 1 - (ForAll) Iniciando
processamento de números pares:
Time: 36:59:139 - Thread Id: 11 - Número par: 0
Time: 36:59:139 - Thread Id: 10 - Número par: 2
Time: 36:59:139 - Thread Id: 12 - Número par: 4
Time: 36:59:139 - Thread Id: 17 - Número par: 6
Time: 36:59:139 - Thread Id: 17 - Somando 1 em paralelo: 7
Time: 36:59:139 - Thread Id: 17 - Número par: 8
Time: 36:59:139 - Thread Id: 17 - Somando 1 em paralelo: 9
Time: 36:59:139 - Thread Id: 11 - Somando 1 em paralelo: 1
Time: 36:59:139 - Thread Id: 10 - Somando 1 em paralelo: 3
Time: 36:59:139 - Thread Id: 12 - Somando 1 em paralelo: 5
```

O método *GetParallel()* é executado duas vezes, para obter objetos de consulta diferentes. Na primeira ocasião, é utilizado o *ForEach()* tradicional. Este não é um método preparado para ser executado em paralelo, e por isso há a necessidade de um sequenciamento dos resultados na thread principal, para após isso realizar as operações informadas dentro do bloco, de forma sequencial. Também é possível observar esse comportamento, onde antes os cálculos haviam sido realizados de forma paralela, posteriormente esses resultados foram sequencializados na thread 1 para execução do resto da lógica.

Já na segunda consulta é utilizado o método *ForAll()*. Através dele, é informado que não há necessidade de sincronizar os resultados das threads da consulta, e que qualquer operação dentro desse bloco pode continuar sendo feita em paralelo. Com isso, a soma dos números pares é feita em quatro threads diferentes *(Ids: 17, 11, 10 e 12)*. Além disso, é possível observar que antes mesmo de todos os números pares serem descobertos, a thread 17 já havia começado a segunda parte da lógica e somado um ao número par seis encontrado por ela, resultando em sete. O número par oito foi descoberto por essa mesma thread logo após isso, uma vez que ela já havia acabado o seu trabalho anterior. Isso demonstra outra vantagem da biblioteca. Uma vez que os cálculos são independentes,

não há necessidade de esperar que todos os números pares sejam encontrados para iniciar o próximo trabalho. Ao contrário de como funciona na lógica do *ForEach()*, que torna necessário esperar o cálculo anterior em todos os elementos para realizar o próximo passo.

2.2.2 Análise de Desempenho

A abstração de alto nível da biblioteca pode esconder a possibilidade de que a execução em paralelo piore de desempenho para qualquer sistema. No caso da PLINQ, *overheads* de divisão de *dataset*, criação de threads, disputa das threads por tempo de CPU, e após isso a junção dos dados, fazem com que em casos menores, seja pior usar a biblioteca para executar o código sequencialmente.

É possível observar esse comportamento ao experimentar com um algoritmo que realiza cálculos mais elaborados, como o de multiplicação de matrizes. Considerando uma matriz qualquer $A(NxM)$ e outra matriz qualquer $B(MxG)$. Sendo a matriz C resultante do cálculo $C = A * B$, esta terá dimensões NxG. Para cada par i,j, onde $1 \leq i \leq N$ e $1 \leq j \leq G$, sendo $C_{i,j}$ cada elemento da matriz final, temos que o cálculo de um elemento é igual a:

$$C_{i,j} = \sum_{k=1}^{M} (A_{i,k} * B_{k,j}) \tag{2.1}$$

Em que $A_{i,k}$ é um elemento da matriz A e $B_{k,j}$ é um elemento da matriz B. Assim, é possível observar que cada elemento da matriz final C depende exclusivamente dos valores lidos de A e de B. Dado que as matrizes A e B não são alteradas a partir do início da multiplicação, o cálculo de cada campo é independente da execução do outro campo.

A versão clássica desse algoritmo consiste em três *loops* aninhados, percorrendo, nessa ordem, as linhas A, as colunas de B e as colunas de A. A complexidade de tempo dessa lógica é de $O(N * G * M)$, ou $O(N^3)$, em casos de matrizes quadradas.

O código 10 implementa esse algoritmo. Primeiramente é feita uma verificação das dimensões das matrizes para saber se as mesmas podem ser multiplicadas. Após isso é feita a inicialização da matriz resultante com as dimensões corretas, tendo o número de linhas da matriz A e o número de colunas da matriz B.

A partir disso, é possível aplicar o algoritmo. Primeiramente é feito um *Select()* utilizando a biblioteca LINQ, que itera por cada linha da matriz A criando um objeto contendo a linha e o índice dessa linha. Após isso, é feito o algoritmo com três loops aninhados, sendo o primeiro o *ForEach()* passando pela lista de linhas de A, o segundo um *loop* for passando pelas colunas de B, e o terceiro passando pelas colunas de A. Com todos os valores em mãos, é possível realizar a multiplicação de cada elemento e acumular esse resultado na posição correta da matriz final.

Código 10 – Multiplicação de Matrizes versão sequencial

```
public static double[][] Sequential(double[][] a, double[][] b)
{
    AbMatrix abMatrix = CheckMultiplicity(a, b);

    var result = Init(abMatrix.ARows, abMatrix.BCols);

    a.Select((row, i) => new {row, i}).ToList().ForEach(objA =>
    {
        for (var j = 0; j < abMatrix.BCols; ++j)
        for (var k = 0; k < abMatrix.ACols; ++k)
            result[objA.i][j] += a[objA.i][k] * b[k][j];
    });
    return result;
}
```

Devido à natureza da complexidade de tempo $O(N^3)$, conforme N aumenta, rapidamente esse algoritmo pode custar bastante tempo para executar. Considerando isso e a independência do cálculo, pode ser que o uso de múltiplas threads seja benéfico. A abordagem utilizada no algoritmo paralelo 11 foi a de criar uma thread para cada linha, se aproveitando da facilidade em paralelizar a consulta com a biblioteca PLINQ, como visto anteriormente. Dessa maneira, cada thread pode calcular a linha da matriz resultante ao mesmo tempo, dividindo o trabalho.

Código 11 – PLINQ - Multiplicação de Matrizes versão paralela

```
public static double[][] Parallel(double[][] a, double[][] b)
{
    AbMatrix abMatrix = CheckMultiplicity(a, b);
    var result = Init(abMatrix.ARows, abMatrix.BCols);

    a.AsParallel().Select((row, i) => new {row, i}).ForAll(objA =>
    {
        for (var j = 0; j < abMatrix.BCols; j++)
        for (var k = 0; k < abMatrix.ACols; k++)
            result[objA.i][j] += a[objA.i][k] * b[k][j];
    });
    return result;
}
```

Novamente pode-se observar a facilidade em transformar um código LINQ em paralelo, com o uso dos métodos *AsParallel()* e *ForAll()*. O restante da lógica do algoritmo permaneceu a mesma.

Para a realização de testes de desempenho, foi utilizado um processador *Intel Core i7-9750H 2.6GHz*, com 6 núcleos físicos e 6 lógicos, juntamente com a *runtime .NET Core 3.1.407*, rodando em um ambiente *Linux Ubuntu 20.04*.

Os testes foram realizados com o algoritmo paralelo, em matrizes quadradas de tamanho 1000, 2000, 3000, utilizando 1, 2, 6 e 12 threads. Através da biblioteca *BenchmarkDotNet*[1] *(versão utilizada v0.12.1)*, foram realizadas dez rodadas para cada matriz e número de threads, coletando a média, desvio padrão e percentil 95 de cada teste. Os resultados obtidos seguem nas tabelas 1 e 2, e no gráfico 2.

Tabela 1 – Disposição da média, desvio padrão (DP) e percentil 95 (P95), em segundos, dos testes de multiplicação de matrizes para 1 e 2 threads.

Tamanho Matriz	Número de Threads					
	1			2		
	Média	DP	P95	Média	DP	P95
1000x1000	5,54	0,06	5,65	3,02	0,17	3,30
2000x2000	77,37	0,44	78,05	38,18	0,28	38,71
3000x3000	289,40	1,51	291,20	135,82	0,14	136,00

Tabela 2 – Disposição da média, desvio padrão (DP) e percentil 95 (P95), em segundos, dos testes de multiplicação de matrizes para 6 e 12 threads.

Tamanho Matriz	Número de Threads					
	6			12		
	Média	DP	P95	Média	DP	P95
1000x1000	1,26	0,12	1,45	1,23	0,06	1,30
2000x2000	14,25	0,25	14,47	11,48	0,35	11,91
3000x3000	49,69	0,25	49,92	41,56	0,62	42,17

Inicialmente é possível observar no gráfico 2, a evolução do tempo médio de execução entre diferentes threads, conforme o tamanho da entrada aumenta. Mesmo em matrizes de tamanho menor, como $1000x1000$ e $2000x2000$, a diferença do tempo médio de execução foi significativa, sendo o maior tempo médio a versão com 1 (uma) thread, levando *5,537s*, em comparação à execução com 12 threads, levando em média *1,227s*. Apesar de absolutamente a diferença de *4,31s* ser pequena, calculando a aceleração (tempo sequencial dividido pelo tempo paralelo), proporcionalmente houve um ganho de *4,51* vezes no desempenho.

Em todos os tamanhos de entrada, a versão com 1 (uma) thread foi, em média, a mais lenta, enquanto a versão com 12 threads a mais rápida. É possível observar a aceleração conforme o aumento do número de threads através da tabela 3, apenas por adicionar dois comandos novos ao código, o *ForAll()* e o *AsParallel()*.

[1] Biblioteca para medição de desempenho: https://benchmarkdotnet.org/articles/overview.html

Figura 2 – Gráfico comparação de velocidade de execução da multiplicação de matrizes com diferente quantidade de threads

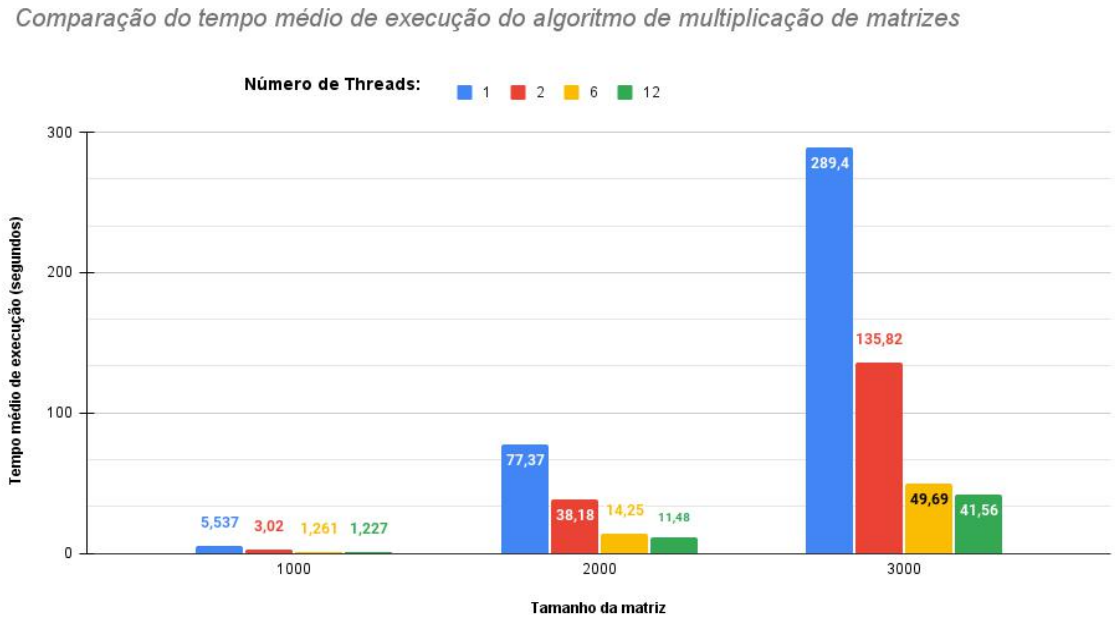


Tabela 3 – Aceleração do tempo de execução com 2, 6 e 12 threads, em comparação com a versão sequencial

Tamanho Matriz	Aceleração		
	2 Threads	6 Threads	12 Threads
1000x1000	1,83	4,39	4,51
2000x2000	2,03	5,43	6,74
3000x3000	2,13	5,82	6,96

Outro resultado interessante retirado desses testes, é a diferença de desempenho entre a versão de 6 e 12 threads, que se mantém baixa ao longo de todos os tamanhos testados. Isso pode ser explicado devido a natureza do processador utilizado, onde apenas 6 dos 12 núcleos são físicos, enquanto os outros 6 são núcleos lógicos, que utilizam a tecnologia *Intel Hyper-Threading* [2].

Apesar dessa tecnologia trazer ganhos de desempenho em contextos com CPU ociosa, não foi o caso do algoritmo de multiplicação de matrizes. Isso pois as threads de execução estariam concorrendo pelos mesmos recursos físicos do núcleo do processador, por realizarem o mesmo tipo de cálculo, e portanto não havendo uma margem grande para ociosidade. o ganho de desempenho foi de no máximo 0,24 vezes na matriz de tamanho dois mil, enquanto na matriz de tamanho mil a aceleração foi de apenas 0,03 vezes, conforme mostra a tabela 4. Porém, o custo de ocupar o dobro do número de núcleos, mesmo que lógicos, para este nível de ganho tão baixo, pode não ser justificável, uma vez que a diferença em tempo absoluto é baixa.

[2] https://www.intel.com.br/content/www/br/pt/gaming/resources/hyper-threading.html

Tabela 4 – Disposição das médias de tempo de execução de 6 threads, 12 threads e comparação através da aceleração entre elas

Tamanho Matriz	6 Threads	12 Threads	Aceleração
1000x1000	1,26	1,23	1,03
2000x2000	14,25	11,48	1,24
3000x3000	49,69	41,56	1,20

Por fim, a PLINQ se mostra uma poderosa aliada do programador C#, ao trazer recursos e extensões simples à recursos já utilizados no dia-a-dia, com a possibilidade de altos ganhos de desempenho. Quanto maior o projeto, mais complicada se torna a manutenção do código e é comum aplicações perderem desempenho ao longo do tempo. Porém parte dessa perda pode ser evitada ao utilizar os múltiplos núcleos do processador, de maneira fácil como a apresentada pela biblioteca.

2.3 PROGRAMAÇÃO ASSÍNCRONA

A programação ser síncrona ou assíncrona, diz respeito ao fluxo de execução de um programa. Quando o código é executado de forma direta e o computador executa cada comando completamente antes de passar para o próximo, esse código é chamado de síncrono. Na programação assíncrona, as operações seguintes a um comando não necessariamente precisam esperar o comando anterior finalizar. Esse controle passa a ficar na mão do programador, que irá decidir quando um código necessita esperar a execução por completo do comando anterior, e quando pode realizar outras ações sem esse resultado.

Essa abordagem pode ser muito vantajosa quando é conhecido que uma determinada função pode demandar um tempo considerável de processamento, e não é necessária para a execução das funções que serão invocadas em seguida. Dessa maneira, o programa não fica bloqueado esperando essa resposta, enquanto pode realizar outras ações. Isso é bastante comum em aplicações *web*, por exemplo, onde o clique em um botão no navegador realiza uma requisição para o servidor, mas o usuário pode continuar realizando outras ações no *site* enquanto essa requisição, que é demorada, não é finalizada. Caso não fosse utilizado programação assíncrona, o site ficaria travado, não deixando o usuário realizar qualquer outra ação enquanto aquela requisição não retornar uma resposta.

É importante notar que esse tipo de comportamento pode ser obtido utilizando-se de múltiplas threads, como visto anteriormente. Mas ao mesmo tempo, coordenar tantas threads em contextos pequenos, ao longo de um extenso programa, pode ser complicado e gerar problemas de concorrência como condições de corrida ou *deadlocks*. Por isso, as linguagens de programação trazem cada vez mais recursos para lidar com a programação assíncrona de maneira mais fácil e eficiente.

Na versão 5 do C#, disponibilizada em 2012, foi introduzido o padrão TAP *(Task asynchronous programming model)*, em português *"modelo de programação assíncrona com*

tarefas", para lidar com código assíncrono; e os comandos *async* e *await* (.NET Team, 2012). Com esses recursos, a linguagem provê uma abstração para lidar com esse tipo de problema, onde o programador escreve o código quase idêntico à sua versão síncrona, porém obtém a vantagem de desempenho e experiência para o usuário que a execução assíncrona possibilita.

O TAP se baseia nas classes do *namespace System.Threading.Tasks*, que são usados para representar operações assíncronas. Esse modelo é hoje o recomendado pelo C# para a criação e manipulação de código assíncrono. Em outra época, antes dos comandos *async* e *await*, o padrão para desenvolvimento de tarefas assíncronas era o APM *(Asynchronous Programming Model)* onde existia a necessidade de definir métodos de *Begin()* e *End()* da operação assíncrona. Com o TAP, isso não é mais necessário, fazendo assim com que a operação assíncrona esteja contida totalmente em um único método.

A programação seguindo o padrão TAP é bastante parecida com como nós, humanos, realizamos tarefas assíncronas no nosso dia-a-dia. Tomando como exemplo um algoritmo para fazer um café da manhã clássico americano:

- 1- Encher uma xícara de café.
- 2- Aquecer uma frigideira e, em seguida, fritar dois ovos.
- 3- Fritar três fatias de bacon.
- 4- Torrar dois pedaços de pão.

Existem diversas formas de fazer o preparo desse café da manhã. A maneira mais usual, para uma pessoa pouco experiente na cozinha, seria executar cada tarefa do início ao fim, de maneira síncrona, para que não houvesse confusão. Um programa em C# que imita esse comportamento ficaria como o programa 12:

Código 12 – Algoritmo Café da Manhã Síncrono

```
private void CafeDaManha() {
        ServirCafe();
        FritarOvo();
        FritarBacon();
        TorrarPao();
}

private Ovo FritarOvo() {
        Frigideira f = GetFrigideira();
        // fritar ovo...
        return ovo;
}

private Bacon FritarBacon() {
        Frigideira f = GetFrigideira();
        // fritar bacon...
        return bacon;
}
```

Dessa maneira, cada etapa da preparação do café da manhã no programa 12, seria realizada sequencialmente. Porém podemos modelar esse problema de forma mais real, e assim, tentar aproveitar melhor os recursos (como os diversos núcleos do processador) e otimizar esse processo de preparação do café da manhã, tornando-o mais rápido.

A maneira como isso pode ser feito no C# gira em torno do padrão TAP, da execução de tarefas assíncronas e dos comandos *async* e *await*. A ideia é que o programador escreva métodos assíncronos quase idênticos aos métodos síncronos, porém com possibilidade de ganho de desempenho na execução concorrente das tarefas. O compilador do C# se encarrega de interpretar o uso dos comandos *async* e *await* e realizar uma série de transformações no código da aplicação para que a tarefa seja executada e o contexto de execução se mantenha, enquanto o *runtime* provê o gerenciamento da execução concorrente. Dessa forma, o programa fica de fácil leitura e compreensão.

A tarefa é um objeto do tipo *Task*, que representa uma operação assíncrona que será realizada no futuro, e é o objeto central do padrão TAP. Por meio desse objeto, é possível informar qual trabalho será feito de maneira assíncrona e acompanhar o andamento da tarefa. Esse trabalho é executado em um *thread pool* (.NET Team, 2021c), e esse tópico será aprofundado no capítulo 3. Porém de forma sucinta, quando uma tarefa é requisitada, ela entra numa fila compartilhada por toda a aplicação, na qual as threads do *thread pool* ficam consumindo e executando.

No código 13, é feita a modelagem do café da manhã apresentado anteriormente, porém seguindo o padrão TAP de desenvolvimento. Os métodos são transformados em métodos

assíncronos através do comando *async* no cabeçalho. Além disso, é uma convenção da comunidade C#, e aconselhado pelo padrão, que os métodos assíncronos tenham o sufixo *Async* em seu nome. Isso é realizado, pois é comum que em uma mesma biblioteca existam as versões síncrona e assíncrona do método, e portanto é necessário uma diferenciação. Além disso, os métodos também tiveram seu retorno alterado para o tipo *Task* ou *Task<T>*, algo obrigatório quando um método se utiliza dos comandos *async* e *await*. Cada tarefa poderia ser esperada separadamente com um *await*, porém, para diminuir o número de linhas de código, é possível ir acumulando essas tarefas em uma lista e posteriormente esperá-las todas de uma vez. Além disso, esperá-las cada uma separadamente tiraria a vantagem de executá-las todas ao mesmo tempo.

Código 13 – Algoritmo Café da Manhã Assíncrono

```
private async Task CafeDaManhaAsync() {
      IList<Task> tasks = new List<Task>();
        tasks.Add(ServirCafeAsync());
        tasks.Add(FritarOvosAsync());
        tasks.Add(FritarBaconAsync());
        tasks.Add(TorrarPaoAsync());
      await Task.WhenAll(tasks);
}

private async Task<Ovo> FritarOvosAsync() {
        Frigideira f = await GetFrigideiraAsync();
        // Frita ovos
      return ovo;
}

private async Task<Bacon> FritarBaconAsync() {
        Frigideira f = await GetFrigideiraAsync();
        // Frita bacon
      return bacon;
}
```

No código 13, é possível observar diversos padrões na utilização de *Tasks*, e acompanhar o funcionamento das mesmas. Primeiro, ao utilizar o comando *await* dentro de um método, o compilador da linguagem obriga o programador a identificar o método com a palavra chave *async* em seu cabeçalho. Além disso, um método *async* também é obrigado a retornar um objeto do tipo *Task<T>*, onde *T* é o valor que a tarefa retorna. No exemplo, podemos verificar esse comportamento nos métodos *FritarBaconAsync()* e *FritarOvosAsync()*. Apesar dos métodos já retornarem o objeto pronto, é necessário o tipo de retorno ser do tipo *Task<T>*, já que por utilizar um método assíncrono dentro de sua implementação, temos apenas uma *promessa* de que um ovo/bacon será devolvido

no final da execução da *Task*.

Quando um método síncrono não apresenta retorno, ele é anotado com o tipo *void* como retorno. Nos métodos assíncronos, o *void* é substituído pelo objeto *Task*. Porém, diferentemente do *void*, essa *Task* retornada é um objeto que carrega informações sobre a tarefa, como por exemplo, se ela foi finalizada com sucesso ou erro. A obrigatoriedade do retorno do tipo *Task* também se torna necessária, pois o comando *await* espera um objeto desse tipo para ser executado.

Conforme os métodos assíncronos são disparados, eles entram numa fila de execução providenciada pelo escalonador de tarefas definido (caso nenhum seja definido, é utilizado o escalonador padrão). Essa fila é consumida, começando a execução do método em uma thread do *thread pool*. Ao chamar um método assíncrono, o fluxo principal de execução não é interrompido, portanto, a thread principal do *CafeDaManha* continua sua execução, mesmo que nenhuma das tarefas tenha sido completada. Apenas quando a palavra chave *await* é utilizada que o fluxo principal de execução é interrompido, e há então a espera da finalização da tarefa. Esse comportamento é visto em todos os três métodos do exemplo.

Nos métodos *FritarBaconAsync()* e *FritarOvosAsync()*, o fluxo é interrompido logo na primeira linha, quando o recurso compartilhado *Frigideira* necessita ser usado através de um outro método assíncrono. O fluxo para fritar a comida e retornar o alimento pronto só é executado quando essa chamada *GetFrigideiraAsync()* for concluída. Já no método principal *CafeDaManha*, o fluxo só é interrompido na última linha, quando é utilizada a palavra chave *await* em conjunto com o método *Task.WhenAll()*. Esse método nada mais faz do que começar uma nova *Task* que é encerrada quando todas as tarefas da lista passada são finalizadas.

É possível observar que o código utilizando TAP é quase idêntico ao código síncrono do primeiro exemplo, porém com toda a vantagem de um código concorrente, onde múltiplas tarefas são executadas ao mesmo tempo, aproveitando ao máximo a capacidade de processamento, e sem precisar se preocupar com threads. Isso demonstra que a utilização de *Tasks* e os comandos *async* e *await* ajudam o programador a criar programas mais eficientes em comparação com sua versão síncrona, mantendo um código legível e de fácil manutenção.

3 DISSECANDO O ASYNC/AWAIT

Os comandos *async* e *await* são uma abstração criada na linguagem C# para lidar com códigos assíncronos e facilitar a vida do programador, como visto anteriormente. Porém, esse mecanismo de tão alto-nível esconde muita complexidade realizada por parte da linguagem e do CLR.

Quando uma tarefa é iniciada dentro do .NET, diversos mecanismos são acionados para que a mesma seja executada, e esse capítulo se propõe a explicar esses mecanismos e transformações internas da plataforma. Inicialmente é aprofundada a descrição sobre o escalonador de tarefas padrão, como as tarefas são escalonadas por ele para execução dentro do *thread pool*, e como é possível realizar uma extensão do mesmo para criar o seu próprio escalonador. Também é feito um estudo sobre as transformações feitas no código pelo compilador, que cria uma máquina de estados internamente para realizar as execuções assíncronas. Ao final, são realizados testes de desempenho a fim de medir o impacto desses mecanismos na velocidade de execução do programa.

3.1 ESCALONADOR DE TAREFAS

A classe *TaskScheduler*[1] é uma classe abstrata do pacote *Threading.Tasks* do .NET que define o comportamento necessário para a organização e execução das tarefas lançadas pelo programa principal. O .NET apresenta uma implementação padrão, a partir da classe *ThreadPoolTaskScheduler*[2], que utiliza o *ThreadPool* do .NET como base para execução das tarefas, aproveitando assim as lógicas de balanceamento de carga e divisão de trabalho que o *ThreadPool* traz. Apesar desse escalonador padrão apresentar bom desempenho e ser a implementação aconselhada pela *Microsoft*, é possível sobrescrever a classe abstrata *TaskScheduler*, e assim, criar escalonadores com características específicas.

3.1.1 Escalonador Padrão

O *TaskScheduler* padrão de C# utiliza como base a implementação de *ThreadPool* do .NET. A classe *ThreadPool* mantém uma fila global FIFO para receber os pedidos de execução de tarefas. Para inserir uma tarefa nessa fila, basta chamar o método *QueueUserWorkItem(Task)*, e assim que disponível, uma thread executará o trabalho.

Dentro do .NET, existem duas classes de tarefas, as de alto e baixo nível. Tarefas de alto nível são tarefas "pai", as primeiras da família, e elas são sempre alocadas nessa fila global FIFO. Já tarefas "filhas", ou seja, tarefas criadas por outras tarefas, são alocadas

[1] https://docs.microsoft.com/pt-br/dotnet/api/system.threading.tasks.taskscheduler?view=netcore-3.1

[2] https://referencesource.microsoft.com/#mscorlib/system/threading/tasks/ThreadPoolTaskScheduler.cs

em uma fila local da thread que está executando a tarefa pai, e essa fila segue a política LIFO.

Quando a thread que executa a tarefa pai está livre para executar mais trabalho, primeiro ela consulta a fila local, e caso exista alguma tarefa filha nessa fila, ela será executada nessa mesma thread. Vale ressaltar que pelo fato da fila local ser do tipo LIFO, a última tarefa filha lançada será a primeira a ser executada. Essa estratégia é utilizada para aproveitar melhor os recursos de memória *cache* e evitar *lock contention* no acesso à uma única fila global.

O uso de filas locais em cada thread alivia a fila principal e traz a vantagem da localidade de dado, uma vez que tarefas filhas tendem a acessar dados referenciados pela tarefa pai, e portanto estão fisicamente mais próximas na memória. Bibliotecas como a PLINQ têm ganho de desempenho ao se aproveitar dessa estratégia (.NET Team, 2021b). A documentação sobre o porquê desse ganho de desempenho é escassa. Porém, é possível inferir que como as operações são focadas em uma fonte de dados, e a divisão dos dados está sob controle da biblioteca, ela se utiliza desse artifício da localidade lançando tarefas filhas, sabendo que serão executadas na mesma thread da tarefa pai e por isso terão um acesso à *cache* de dados quando necessário, ganhando desempenho.

Outra funcionalidade aproveitada com a estrutura da classe *ThreadPool* é a estratégia de roubo de trabalho. Caso o número de tarefas pai seja menor que o número de threads disponíveis no *thread pool*, algumas dessas threads que sobraram ficariam ociosas. Enquanto isso, as tarefas filhas lançadas e que são alocadas nas filas locais ficam esperando que a thread em execução termine e consuma a fila local. É possível observar então um desperdício de recursos, pois enquanto existem tarefas filhas em filas locais a serem executadas, existem threads no *thread pool* ociosas esperando por trabalho.

Nesse momento a estratégia de roubo de trabalho entra em ação. Cada thread, ao terminar o seu trabalho, segue uma linha de procura por outras tarefas.

- 1- Fila local própria, LIFO.
- 2- Fila global, FIFO.
- 3- Fila local de outra thread, FIFO.

Primeiramente, é verificada a fila local de maneira LIFO, e após isso, a thread procura na fila global de maneira FIFO. No caso de não existirem tarefas em ambas as filas, a thread passa a procurar tarefas nas filas locais de outras threads. Ao contrário do que acontece com a própria thread pai, essa procura nas filas locais é feita de maneiro FIFO, ou seja, contrária à ordenação LIFO que a thread pai busca, evitando assim colisões e tentando manter ainda a otimização de *cache*.

Uma maneira de evitar qualquer uma das filas (seja local ou global), é marcar a tarefa com a opção *long-running*. Isso deve ser feito durante a criação da tarefa, e é vantajoso

quando é conhecido previamente que essa tarefa terá uma longa duração. Isso porque uma tarefa de longa duração pode ''alugar" uma thread do *thread pool* por muito tempo, podendo causar *starvation* tanto na fila global quanto local, bloqueando outras tarefas de serem executadas rapidamente. Quando a tarefa é marcada com essa opção, o escalonador padrão cria uma thread totalmente independente do *thread pool* para a sua execução.

3.1.2 Estendendo a classe TaskScheduler

Para a criação de um escalonador personalizado, é necessário criar uma classe que estenda da classe abstrata *TaskScheduler*. Através dela, os métodos *GetScheduledTasks()*, *QueueTask()* e *TryExecuteTaskInline()* ficam disponíveis para a sobrescrita. Esses métodos são utilizados internamente por outras classes do .NET que irão se comunicar com o escalonador.

O método *GetScheduledTasks()* precisa retornar uma lista de tarefas que estão a espera de serem executadas. O método *QueueTask()* é chamado quando uma tarefa é lançada no sistema, e tem como intuito enfileirar essa tarefa para futura execução. E por fim, o método *TryExecuteTaskInline()* é chamado para tentar realizar a execução de uma tarefa na thread corrente que chamou o *await* da tarefa.

Para exemplificar, o código 14 mostra a implementação de um escalonador bem simples, utilizando os conhecimentos já apresentados neste trabalho, onde as tarefas são enfileiradas em uma fila, e consumidas dela de dez em dez segundos por uma thread independente.

Código 14 – Realizando extensão da classe TaskScheduler

```
public class SimpleTaskScheduler : TaskScheduler  {
        private ConcurrentQueue<Task> tasks = new ConcurrentQueue<Task
            >();
        public SimpleTaskScheduler() {
          var T = new Thread(() => {
              Logger.Log("Comecando execucao de tarefas...");
              while (true)
              {
                  Thread.Sleep(10_000);
                  ExecuteTask();
              }});
          T.Start();
        }

        private void ExecuteTask() {
          if (tasks.Count <= 0)
              Logger.Log($"Sem tasks para executar");
          else
          {
              Logger.Log($"Tentando retirar task da fila. Qntd: {tasks.
                  Count}");
              if (tasks.TryDequeue(out Task task))
              {
                  Logger.Log($"Retirada task {task.Id}");
                  TryExecuteTask(task);
              }
              else
              {
                  Logger.Log($"Falha ao retirar da fila");
              }
          }
        }

        protected override void QueueTask(Task task) {
          Logger.Log($"Enfileirando task: {task.Id}");
          tasks.Enqueue(task);
        }
}
```

Nesse escalonador, é criada uma thread no momento em que seu construtor é chamado. Essa thread possui um laço infinito, que dorme por dez segundos e chama a função *ExecuteTask()*, que verifica se existe alguma tarefa a ser realizada na fila, e caso positivo, a executa. O método *TryExecuteTask()* é um método herdado da classe pai *TaskScheduler*,

que executa a *Task* passada como parâmetro.

Para utilizar um escalonador próprio, é necessário informar para a classe *Task* (que é responsável por criar tarefas), que um escalonador diferente do padrão será utilizado. Para isso, na criação da tarefa, é passado como parâmetro uma instância do escalonador a ser utilizado, nesse caso, o *SimpleTaskScheduler*.

O código 15 realiza um teste desse escalonador. Primeiramente uma instância dele é criada, e portanto, a thread com o *loop* infinito já está em execução, procurando por tarefas a serem realizadas. Após isso, é realizado um *loop* for que cria três tarefas utilizando o escalonador criado.

Código 15 – Testando escalonador simples

```
class Program
{
        private const int NumberOfTasks = 3;

        static void Main(string[] args)
        {
            SimpleTaskScheduler simpleTaskScheduler = new
               SimpleTaskScheduler();
            for (int i = 0; i < NumberOfTasks; i++)
            {
                var i1 = i + 1;
                Task.Factory.StartNew(
                  () => {
                     Logger.Log($"eu sou a task {i1}")
                  },
                  CancellationToken.None,
                  TaskCreationOptions.None,
                  simpleTaskScheduler);
            }
        }
}
```

Nesse programa 15, é utilizado o método estático *StartNew()* da classe *Task*. Esse método realiza a criação de uma nova tarefa e também permite passar alguns parâmetros de personalização da mesma, como o escalonador necessário para o teste.

Seu primeiro parâmetro é a função que será executada como tarefa. O segundo parâmetro é um *token* de cancelamento, um mecanismo do C# onde é possível passar um objeto que tem o poder de cancelar a tarefa dado uma fração de tempo. O terceiro parâmetro é onde pode ser especificado se a tarefa é de longa duração, como mencionado na seção anterior. E o último parâmetro é um objeto que estenda a classe *TaskScheduler*. Caso não seja especificado, o sistema utilizará o escalonador padrão. No caso, foi utilizado

um objeto do tipo *SimpleTaskScheduler*, o escalonador personalizado criado. Portanto, essa tarefa será agendada para execução seguindo a programação desse escalonador. A saída do programa 15 fica da seguinte maneira:

Saída do programa 15 com uma lista de zero a dez:

```
1:20:42:423 - Executando main...
1:20:42:450 - Criando timer para execução de tasks
a cada 10s...
1:20:42:451 - Começando execução de tarefas...
1:20:42:451 - Enfileirando task: 1
1:20:42:451 - Numero de tasks: 1
1:20:42:451 - Enfileirando task: 2
1:20:42:451 - Numero de tasks: 2
1:20:42:451 - Enfileirando task: 3
1:20:42:451 - Numero de tasks: 3
1:20:52:453 - Tentando retirar task da fila. Qntd: 3
1:20:52:453 - Retirada task 1
1:20:52:454 - eu sou a task 1
1:21:02:455 - Tentando retirar task da fila. Qntd: 2
1:21:02:455 - Retirada task 2
1:21:02:455 - eu sou a task 2
1:21:12:455 - Tentando retirar task da fila. Qntd: 1
1:21:12:456 - Retirada task 3
1:21:12:456 - eu sou a task 3
1:21:22:456 - Sem tasks para executar
```

O programa começa criando o objeto referente ao escalonador, portanto, o conteúdo em seu construtor será executado. Esse construtor cria e inicia a thread que ficará no *loop* infinito buscando por tarefas. Após isso, o programa volta para o fluxo principal e entra em um *loop* que cria três tarefas e as adiciona ao escalonador personalizado.

No momento *1:20:52:453* é retirada a primeira tarefa para ser executada. Após isso, a cada 10 segundos, tempo configurado dentro da lógica do escalonador, uma tarefa é retirada da fila e executada, conforme as saídas de *"eu sou a task i"*.

É possível observar também através da saída, que as tarefas são executadas em ordem FIFO, pois a ordem de execução segue a ordem de criação, com a tarefa de *Id* 1 executando primeiro, em seguida a de *Id* 2 e por fim a de *Id* 3. Isso ocorre pois a estrutura de dados utilizada dentro do escalonador era a *ConcurrentQueue*, uma estrutura de dados do tipo fila e que portanto segue a ordem FIFO. Essa estrutura de dados foi escolhida para permitir que sejam criadas múltiplas threads que enfileiram tarefas para o escalonador. Com múltiplas threads realizando esse trabalho, tanto a inserção quanto remoção de

tarefas da fila precisa ser sincronizada para que não haja problemas de concorrência como condições de corrida.

3.2 MÁQUINA DE ESTADOS

Até aqui, foi possível analisar como é feita a construção de métodos assíncronos, e como múltiplas tarefas são organizadas e selecionadas para execução por parte do .NET. Uma tarefa é uma unidade de trabalho que promete um resultado no futuro. Esse trabalho pode ser uma requisição *I/O* ou uma tarefa intensiva de CPU. Independente disso, a tarefa é auto-suficiente. Tarefas são valores de primeira classe, sendo possível passá-las como parâmetro, agendar continuações e avaliar os resultados em casos de sucesso ou falha.

Para que isso seja possível, existem diferenças fundamentais em um método marcado como assíncrono (que utiliza a palavra-chave *async* em seu cabeçalho). Um método síncrono, tem apenas um ponto de saída. Porém, um método assíncrono não segue essa mesma lógica. Isso porque o compilador C# executa transformações no código, construindo uma máquina de estados para possibilitar a execução assíncrona. Essa máquina de estados é responsável por executar pedaços do código original, guardar o contexto de execução e em qual estado a máquina se encontra, para posteriormente, quando a tarefa for concluída, poder retomar o resto do caminho a ser executado. A lógica principal para criação da máquina de estados segue os seguintes passos:

- Uma classe que implementa a interface *IAsyncStateMachine* é criada, e nela é criado o método *MoveNext*;
- A lógica do método assíncrono é colocada dentro do método *MoveNext()*, com algumas modificações de verificação de estado;
- O método original é substituído por um código que cria um novo objeto dessa classe da máquina de estados, salva o contexto de execução e chama o método *Start()*, que em seguida chama o método *MoveNext()* da máquina de estados.

Como mencionado, a lógica principal do método assíncrono é alterada para se transformar numa máquina de estados, com o código sendo separado entre os pontos de *await* dentro do método. É possível verificar o código gerado pelo compilador através de ferramentas de descompilação, como o *ILSpy*[3].

O código 16 é o método assíncrono que terá a sua transformação analisada. O método inicialmente cria um cliente para realizar requisições *http*. Em seguida, é feita uma requisição assíncrona para o site *https://twitter.com*, e nessa chamada é realizado um *await*. No final do método é impressa a saída "Done".

[3] https://github.com/icsharpcode/AvaloniaILSpy

Código 16 – Método assíncrono para transformação

```
private static async Task DoSomething()
{
    var client = new HttpClient();
    var task2 = await client.GetStringAsync("https://twitter.com");
    Logger.Log("Done");
}
```

O código 16, após passar pelo compilador, se apresenta com a forma do código 17:

Código 17 – Método assíncrono transformado

```
private sealed class <DoSomething>d__3 : IAsyncStateMachine {
  public int <>1__state;
  public AsyncTaskMethodBuilder <>t__builder;
  private HttpClient <client>5__1;
  private string <task2>5__2;
  private string <>s__3;
  private TaskAwaiter<string> <>u__1;

  private void MoveNext() {
    int num = <>1__state;
    try {
      TaskAwaiter<string> awaiter;
      if (num != 0) {
        <client>5__1 = new HttpClient();
        awaiter = <client>5__1.GetStringAsync("https://twitter.com").
           GetAwaiter();
        if (!awaiter.IsCompleted) {
          num = (<>1__state = 0);
          <>u__1 = awaiter;
          <DoSomething>d__3 stateMachine = this;
          <>t__builder.AwaitUnsafeOnCompleted(ref awaiter, ref
             stateMachine);
          return;
        }
      }
      else {
        awaiter = <>u__1;
        <>u__1 = default(TaskAwaiter<string>);
        num = (<>1__state = -1);
      }
      <>s__3 = awaiter.GetResult();
      <task2>5__2 = <>s__3;
      <>s__3 = null;
      Logger.Log("Done");
    }
    catch (Exception exception) {
      <>1__state = -2;
      <client>5__1 = null;
      <task2>5__2 = null;
      <>t__builder.SetException(exception);
      return;
    }
    <>1__state = -2;
    <client>5__1 = null;
    <task2>5__2 = null;
    <>t__builder.SetResult();
}}
```

É possível verificar a criação da classe *<DoSomething>d__3*, que abriga a lógica original da nossa função. A classe é criada com caracteres especiais "<" ">", para evitar conflito de nome, uma vez que os mesmos não podem ser utilizados normalmente. Essa classe implementa a interface *IAsyncStateMachine*, e portanto é necessário que implemente o método *MoveNext()*.

O primeiro condicional do método *MoveNext()* contém a primeira parte do código original, na qual é instanciado um cliente *http* e é feita a chamada assíncrona. A partir daí, a execução pode seguir dois caminhos.

Caso a tarefa não tenha sido finalizada, o contexto de execução é salvo através da variável *stateMachine*, e o seu estado é atualizado para zero. A própria máquina de estados se cadastra como uma continuação da tarefa, através do método *AwaitUnsafeOnCompleted()*. Dessa maneira, quando a tarefa estiver concluída, o .NET irá chamar o método *MoveNext()* dessa máquina de estado novamente.

Quando a tarefa for concluída e o método for chamado, seu estado estará com valor 0, e portanto irá executar o bloco else do primeiro condicional, onde é recuperado o objeto *awaiter*, que contém o resultado da *Task*, e a máquina tem seu estado atualizado para o valor -1 novamente (valor inicial). O método *awaiter.GetResult()* traz o resultado da tarefa, e a segunda parte do código original é executada (linha de *log Logger.Log("Done")*).

Porém, caso a tarefa já tenha sido finalizada, o restante do código é executado de forma direta, sem precisar salvar o contexto ou passar por um estado intermediário da máquina. O método pula para a parte onde o objeto *awaiter* é recuperado, e termina a sua execução normalmente.

É importante perceber uma otimização nesse código da máquina de estados, chamado de *Hot Path*, quando a tarefa é finalizada rapidamente. A máquina antes de salvar o contexto de execução e se cadastrar como *callback* da tarefa, realiza uma verificação através do código *if(!awaiter.IsCompleted)*, e a máquina então pula diretamente para a execução do próximo passo. Isso traz o ganho de que a execução de mais de um passo da máquina estará executando na mesma thread, sem precisar salvar o contexto de execução nem gerar outras chamadas e inicializações. Um método assíncrono no qual as tarefas já estejam completas no momento dessa verificação, será executado quase como um método síncrono.

Além da criação da máquina de estados, há também o passo de criação do método estático com o mesmo nome do método original, como mostrado no código 18.

Código 18 – Método estático criado na compilação

```
private static Task DoSomething()
{
  <DoSomething>d__3 stateMachine = new <DoSomething>d__3();
  stateMachine.<>t__builder = AsyncTaskMethodBuilder.Create();
  stateMachine.<>1__state = -1;
  stateMachine.<>t__builder.Start(ref stateMachine);
  return stateMachine.<>t__builder.Task;
}
```

Essa função realiza o trabalho de instanciar um objeto do tipo máquina de estado (que executa efetivamente a lógica), o primeiro estado da máquina é configurado como -1, e o método *Start()* é chamado, dando início a execução da máquina.

A lógica da criação da máquina de estados é bastante parecida quando aplicada a códigos com diversos pontos de chamada da função *await*. Porém, conforme esse número de seções aumenta, o compilador opta por se utilizar de estruturas como *switch-case* e *goto*, para facilitar a transição de estados. O seguinte método representado no código 19 conta com três chamadas assíncronas, sendo uma delas uma chamada para o método do primeiro exemplo, o *DoSomething()*. Sabendo a lógica utilizada pelo compilador, é possível prever as seções que serão criadas, e que estão identificadas através dos comentários.

Código 19 – Método com duas chamadas assíncronas

```
private static async Task DoubleAsyncAwait()
{
    // parte 1
    var client = new HttpClient();
    var task = await client.GetStringAsync("https://google.com");
    // parte 2
    await DoSomething();
    var task2 = await client.GetStringAsync("https://twitter.com");
    // parte 3
    Logger.Log("Terminado double async");
}
```

É possível observar o código completo descompilado do método 19 no Apêndice A. A primeira parte da lógica principal fica localizada no case *default*. Em todas as chamadas assíncronas, antes de salvar o contexto e se cadastrar como *callback*, o programa verifica o estado da tarefa. Caso todas as tarefas tenham sido concluídas, o método executa por completo no mesmo contexto, utilizando o *hot path*, evitando assim um *overhead* na execução. Sendo este o caso, os blocos condicionais com essa verificação não serão executados, e os códigos *goto* indicarão qual será o próximo passo a ser executado imediatamente.

Caso contrário, a máquina salva seu estado para que quando a tarefa esteja pronta, saiba voltar para o pedaço correto de código.

3.3 DESEMPENHO DO ASYNC/AWAIT

Sabendo das transformações realizadas pelo compilador para a execução de um método assíncrono, é possível perceber que existem custos associados. Nesta seção, é apresentada uma avaliação de desempenho da execução dos métodos assíncronos a partir de uma aplicação exemplo.

A aplicação representada pelo código 20 consiste de uma lista de preços de ações, e três métodos de busca. Um síncrono, que realiza a busca através de um loop for. Um assíncrono, que lança uma tarefa para executar essa mesma busca. E um síncrono, que chama a busca diretamente, porém anotado com o comando *async* em seu cabeçalho.

O método de busca varre a lista um a um, procurando o valor da ação. A utilização de uma lista em vez de um dicionário é proposital. Para avaliar a sobrecarga de processamento dentro de métodos assíncronos, seria necessário simular algum tipo de trabalho, e ao utilizar a lista, esse trabalho se torna a busca linear $O(n)$ pelo preço da ação.

Código 20 – Método para teste de desempenho assíncrono

```
 // Full Async
public async Task<decimal> FullAsync(string companyId) => await
    GetPriceAsync(companyId);

// Full Sync
public decimal FullSync(string companyId) => GetPrice(companyId);

// Async that is Sync
public async Task<decimal> FakeAsync(string companyId) => GetPrice(
    companyId);

private decimal GetPrice(string name) => Search(name);

private async Task<decimal> GetPriceAsync(string name) => await Task.Run
    (() => Search(name));

private decimal Search(string name)
{
    foreach (var kvp in _arrayStocks)
        if (kvp.name == name)
            return kvp.price;
}
```

Os testes foram realizados utilizando a biblioteca *BenchmarkDotnet* (.NET Founda-

tion, 2013) e o código pode ser encontrado no Apêndice B. A ação buscada foi sempre a última da lista, para forçar que a busca percorresse a lista em sua totalidade, com complexidade $O(n)$. Foram feitos testes para listas de tamanho mil, dez mil e cem mil elementos. O método "FullSync" testa a versão síncrona, o método "FullAsync" testa a versão assíncrona, e o método "FakeAsync" testa a versão síncrona, com o cabeçalho modificado. Nesse último, queremos observar o impacto das transformações no código, como a criação da máquina de estados, mesmo chamando a versão síncrona internamente. Por causa da funcionalidade do *hot path*, esse código será sempre executado síncronamente, porém terá o impacto da criação de objetos e fluxos vistos na transformação pelo compilador.

Os resultados estão dispostos na tabela 5. Foram coletados os dados da média e erro para cada um dos tamanhos, ambos em milissegundos. O erro é calculado pela biblioteca *BenchmarkDotNet*, como sendo metade do intervalo de confiança de 99%. Além disso, na tabela é apresentada a razão do tempo médio entre o método síncrono e os outros dois. Através dessa razão é possível observar o quanto a adição da camada de tarefas impacta no desempenho do programa.

Tabela 5 – Disposição da média, erro e razão, em milissegundos, da execução do código 20 para mil, dez mil e cem mil elementos

Tamanho	1000			10000			100000		
	Média	Erro	Razão	Média	Erro	Razão	Média	Erro	Razão
FullSync	0,057	0,001	1,000	0,706	0,014	1,000	10,520	0,159	1,000
FullAsync	0,074	0,001	1,290	0,816	0,016	1,157	10,460	0,207	0,994
FakeAsync	0,058	0,001	1,010	0,717	0,014	1,016	10,700	0,209	1,017

Comparando o método assíncrono (*FullAsync*) com o método totalmente síncrono (*FullSync*), a diferença chegou em 29% a mais no tempo de execução, com a lista de tamanho mil. Conforme o tamanho da lista aumenta, o tempo de execução do método de busca aumenta junto, e a diferença entre esses métodos cai para 15% na lista de tamanho dez mil. Já na lista de tamanho cem mil, o método assíncrono empata com o método síncrono, quando considerado o erro. Já no método assíncrono *FakeAsync*, a diferença se manteve sempre pequena, tendo menos de 1% de queda de desempenho.

Essa diferença no tempo de execução entre os métodos totalmente síncrono e assíncrono é esperada, devido a sobrecarga que é necessária para executar uma tarefa. Como explicado anteriormente, um método assíncrono passa por transformações do compilador, é preciso que a tarefa passe pelo escalonador padrão, para enfim ser executada em uma thread do *thread pool*.

Apesar disso, é importante lembrar o impacto dessa sobrecarga em diferentes situações e aplicabilidades. O exemplo realiza uma simples busca linear numa lista, algo trivial e bastante rápido. A busca na lista de cem mil elementos demorou em média dez milissegundos, e já nessa média de tempo foi possível observar que o impacto da sobrecarga por utilizar tarefas foi nulo. Quando aplicada essa estratégia em aplicações focadas em *I/O*,

como acesso a banco de dados ou à camada de rede, que costumam ser chamadas mais demoradas, o uso de tarefas e dos comandos *async* e *await* pode ser bem aproveitado, tendo pouco ou nenhum impacto no desempenho, e ganhando na produtividade do código e aproveitamento de recursos, como reaproveitamento de threads com o *thread pool* e a possibilidade de estruturar o código de maneira não bloqueante.

4 ESCALONADOR

Após um maior entendimento do funcionamento do escalonador padrão e também de como implementar e utilizar um escalonador diferente deste, é apresentado neste capítulo uma sugestão de implementação de um escalonador de tarefas por prioridade. Essa sugestão vem do fato que no escalonador padrão não há mecanismos de prioridade para as tarefas. A única maneira de forçar a execução de uma tarefa à frente das outras, seria marcar a tarefa como *long-running*, criando assim uma thread separada apenas para ela. Porém esse mecanismo seria ineficiente para escalonar diversas tarefas pequenas de prioridade alta, além de não poder especificar diversos níveis de prioridade.

Neste capítulo é apresentada uma proposta de escalonador que dispõe da funcionalidade de poder criar diversos níveis de prioridade para as tarefas dentro do programa, e que seja possível garantir que tarefas de prioridade mais alta serão executadas primeiro. O escalonador garante também que tarefas de prioridade igual serão executadas em ordem FIFO, e segue um modelo não-preemptivo, ou seja, não interrompendo uma tarefa corrente para executar outra.

4.1 ESTRUTURA DE DADOS PARA FILA DE PRIORIDADE

Para ter a funcionalidade de priorizar tarefas, foi escolhida a *PriorityQueue* (fila de prioridade), como estrutura de dados. O .NET, até a data desse trabalho, não tem uma implementação nativa dessa estrutura, essa novidade está por vir na próxima versão 6.0[1]. Porém, por ser uma estrutura de dados tão famosa, existem diversas implementações disponíveis, e a da *Visual Studio Magazine*[2] foi escolhida para ser utilizada nesse escalonador. Essa escolha se deu pelos seguintes motivos: código open-source, ou seja, código aberto e livre para uso de terceiros; e implementação utilizando uma heap binária de mínimo. O código da *PriorityQueue* pode ser encontrado no Anexo A.

Uma maneira simples de implementar uma fila de prioridade seria utilizar uma lista encadeada, e após cada inserção, executar um algoritmo de ordenação que tomasse a prioridade como parâmetro. Porém essa abordagem seria ineficiente por necessitar verificar a lista por completo e executar a ordenação a cada inserção de elemento. Uma *heap* binária de mínimo é uma implementação mais comum e eficiente para esse problema (CORMEN et al., 2009).

A *heap* binária é uma árvore binária representada na forma de um vetor, e a palavra "mínimo" significa que o elemento de menor valor será mais prioritário, ou seja, ficará mais perto da raiz da árvore. Cada nó de uma *heap* binária de mínimo é garantidamente menor

[1] https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.priorityqueue-2?view=net-6.0

[2] Revista online focada em assuntos sobre o mundo .NET https://visualstudiomagazine.com/Home.aspx

ou igual em prioridade que os seus nós filhos. Seja n o número de elementos inseridos, a complexidade de tempo de inserção em uma *heap* binária será $O(log(n))$. A remoção se dá sempre pela raiz da árvore por ser o item mais prioritário, tendo assim a complexidade de tempo $O(1)$. Além disso, a complexidade de espaço é $O(n)$, por ser apenas uma lista contendo os elementos inseridos.

A implementação escolhida disponibiliza uma classe *PriorityQueue<T>* que representa a fila de prioridade, e dispõe dos métodos *Queue(T elem)* para inserir um elemento, *Dequeue()* para remover o elemento mais prioritário (a raiz da árvore), além dos métodos *Peek* que retorna uma cópia do objeto mais prioritário e *Count* que retorna quantos elementos existem na fila. Além disso, *T* é obrigatoriamente uma classe que implemente a interface *IComparable<T>*[3].

A interface *IComparable<T>* determina a implementação do método *CompareTo()*, que recebe uma outra instância de *T*, e retorna um *int*. Esse método é chamado internamente por outros métodos como *Sort()* ou *Add()*, quando há necessidade de determinar a ordem entre dois elementos. O método *CompareTo()* compara duas instâncias de *T*, que serão chamadas de fixa e desafiante. Se o valor retornado pelo método *CompareTo()* for menor que zero, isso quer dizer que a fixa deve vir antes que a desafiante em uma ordenação. Caso o valor retornado seja maior que zero, então a instância fixa deve vir depois que a desafiante. E caso retorne zero, quer dizer que em questões de ordenação, ambas as instâncias têm a mesma prioridade. Utilizando como exemplo uma lista de inteiros, ao comparar a instância de valor "5" com a de valor "6", o método deveria retornar "-1", indicando que "5" vem antes do "6" numa ordenação. Caso a comparação fosse entre "15" e "10", o método deve retornar "1", indicando que "15" vem depois de "10". Enquanto uma comparação entre "8" e "8" deve retornar "0", pois os números são iguais e não tem como ordená-los.

4.2 MODELO DE TAREFA COM PRIORIDADE

Para modelar uma tarefa que contenha a propriedade de prioridade, foi criada a classe *PriorityTask*, mostrada no código 21. Além da prioridade, era necessário fazer com que essa classe fosse possível de ser ordenada dentro da fila de prioridade. Para isso, foi implementada a interface *IComparable<T>*, que exige o método *CompareTo()*.

No construtor da classe *PriorityTask* é informado um identificador para a tarefa, a prioridade e a tarefa a ser executada. Além disso, internamente o construtor preenche uma propriedade *CreationTime*, que é inicializada com o horário de criação do objeto. No método *CompareTo()*, a lógica é igual a apresentada anteriormente, onde é comparado o valor da prioridade através da propriedade *Priority* para saber qual virá antes na ordenação. Porém, no caso de empate na prioridade, queremos manter uma ordenação FIFO se

[3] https://docs.microsoft.com/pt-br/dotnet/api/system.icomparable-1?view=netcore-3.1

baseando no horário de criação da tarefa, onde a *PriorityTask* criada primeiro terá maior prioridade. Por isso, é adicionada uma verificação para empate, e quando verdadeira, são comparadas as datas de criação *CreationTime* de cada objeto.

Código 21 – Modelo Tarefa com Prioridade

```
public class PriorityTask : IComparable<PriorityTask>
{
    public int Id { get; set; }
    public Task Task { get; set; }
    public int Priority { get; set; }
    private DateTime CreationTime { get; set; }

    public PriorityTask(int id, int priority, Task task)
    {
        Id = id;
        CreationTime = DateTime.Now;
        Priority = priority;
        Task = task;
    }
    public int CompareTo(PriorityTask other)
    {
        if (Priority == other.Priority)
            return CreationTime.CompareTo(other.CreationTime);
        return Priority - other.Priority;
    }
}
```

4.3 LÓGICA INTERNA DO ESCALONADOR

Como visto no capítulo 3, para criar um escalonador personalizado, é necessário que se estenda da classe *TaskScheduler*. Além disso, para utilizar o escalonador criado, é necessário que no momento de criação da tarefa através do método *Task.Factory.StartNew*, seja repassado um objeto do tipo do escalonador novo.

Para a implementação desse escalonador, foi escolhida uma estratégia de Gerente-Trabalhador. Essa estratégia foi escolhida para manter centralizadas as instâncias da fila de prioridade e as threads consumidoras de tarefas. Ambas são mantidas dentro do gerente, de forma com que as threads não sejam dedicadas à uma prioridade específica, mas sim compartilhadas, podendo executar qualquer nível de tarefa. Para isso, foram criadas duas classes, *PriorityTaskSchedulerWorker* e *PriorityTaskSchedulerManager*, representando o trabalhador e o gerente, respectivamente.

4.3.1 Gerente

O gerente é responsável por manter uma *PriorityQueue<PriorityTask>*, que será compartilhada por todos os trabalhadores. Essa fila será utilizada para armazenar todas as tarefas, em ordem de prioridade, iniciadas pelos trabalhadores cadastrados nesse gerente. Além disso, na criação do gerente, é possível informar o número de threads que irão consumir dessa fila compartilhada, através da propriedade *concurrencyLevel*. Se não for especificado o número de threads consumidoras, o escalonador utiliza o valor retornado pela variável *Environment.ProcessorCount*, que obtém o número de núcleos (físicos e lógicos) do computador.

O código 22 contém o método principal de execução das tarefas do gerente, enquanto no Apêndice C pode ser visto a classe completa. No construtor, a variável *concurrencyLevel* é recebida como parâmetro, e caso um valor não seja informado, o padrão será o número de processadores da máquina. Após isso, as threads consumidoras são criadas, e todas irão executar o método *ConsumerThread*, que consiste de um *loop* infinito, procurando por tarefas na fila de prioridade. Caso tenha alguma tarefa, ela será executada, caso contrário, a thread adormece por um segundo. Além disso, há a definição do método *EnqueuePriority*, que insere na lista de prioridade uma tarefa a ser executada.

Código 22 – Gerente do Escalonador de Prioridade - ConsumerThread

```
private void ConsumerThread()
{
    while (true)
    {
        _semaphore.Wait();
        var success = _taskQueue.TryDequeue(out PriorityTask pt);
        _semaphore.Release();
        if (success)
        {
            if(_schedulers.TryGetValue(pt.Priority, out
               PriorityTaskSchedulerWorker sch))
            {
                Logger.Log($"(Consumidora) Comecando a executar tarefa
                   de Id {pt.Id} e prioridade: {pt.Priority}");
                sch.ExecuteTask(pt.Task);
            }
        }
        else
        {
            Thread.Sleep(TimeSpan.FromSeconds(1));
            Logger.Log("sleeping 1s");
        }
    }
}
```

É importante notar que o acesso à fila de prioridade *_taskQueue*, realizada nos métodos *EnqueuePriority* e *ConsumerThread*, são feitos numa seção crítica, definida pelas chamadas do semáforo *_semaphore.Wait* e *_semaphore.Release*. Isso porque a estrutura de dados *PriorityQueue<T>* não é *thread-safe*. Como diversas threads podem inserir e retirar tarefas da fila ao mesmo tempo, é necessário um mecanismo de sincronização.

O método *AddScheduler* é chamado pelas instâncias trabalhadoras do tipo *PriorityTaskSchedulerWorker*, para serem inseridas no contexto do gerente. A classe *PriorityTaskSchedulerWorker* (definida na próxima seção, 4.3.2) implementa a interface *TaskScheduler* que contém o método *TryExecuteTask* para executar uma tarefa. Quando a instância trabalhadora se "cadastra" no gerente, ela é salva em um dicionário onde a chave é a prioridade da fila, e o valor é a instância de *PriorityTaskSchedulerWorker*. Quando a thread consumidora encontra uma tarefa para realizar, esse dicionário é acessado, e a instância referente àquela prioridade é usada para chamar o método *TryExecuteTask* e executar efetivamente a tarefa.

4.3.2 Trabalhador

O trabalhador foi implementado criando a classe *PriorityTaskSchedulerWorker*, disponível no Apêndice D. Ela é responsável por implementar a classe abstrata *TaskScheduler* e assim tornando suas instâncias aptas a serem passadas como parâmetro para o método que cria as tarefas, o *Task.Factory.StartNew*. Além disso, na criação de um objeto trabalhador, é necessário informar a prioridade, e quem é o seu gerente. Nesse mesmo construtor é chamado o método *AddScheduler* do gerente, descrito anteriormente. Caso já exista um escalonador para aquela prioridade, o construtor lança um erro e o objeto trabalhador não é criado.

O método *QueueTask* representado no código 23, necessário para a classe abstrata *TaskScheduler*, enfileira a tarefa em uma *ConcurrentQueue<Task>* local, e também na fila de prioridades do seu gerente, criando um novo objeto do tipo *PriorityTask*. O tipo da fila local é o *Task*, diferentemente da fila de prioridade onde o tipo é o *PriorityTask*. Isso é necessário pois outro método obrigatório pela classe *TaskScheduler* é o *GetScheduledTasks*, que retorna a lista de tarefas a serem executadas por aquele escalonador. Além disso, essa fila precisa ser concorrente, pois o escalonamento de tarefas pode ser feito em qualquer ponto da aplicação, e por diferentes threads.

Código 23 – Trabalhador do Escalonador de Prioridade - QueueTask

```
protected override void QueueTask(Task task)
{
    _tasks.Enqueue(task);
    _manager.EnqueuePriority(new PriorityTask(Interlocked.Increment(ref
        _taskCounter), Priority, task));
}
```

4.4 AVALIAÇÃO

Para utilizar o escalonador de tarefas por prioridade, é necessário criar um objeto gerente e n objetos trabalhadores, que indicarão cada prioridade. Para averiguar o funcionamento correto do escalonador, foi realizado um teste que verifica as propriedades principais:

- Tarefas devem ser executadas de acordo com a ordem de prioridade.
- Caso haja empate na prioridade, a fila deve seguir uma ordenação FIFO
- A execução das tarefas segue um modelo não-preemptivo

Para esse teste, como mostrado pelo código 24, foram criados um objeto gerente e dois objetos trabalhadores, indicando a prioridade de nível um e nível dois. Lembrando que por ser uma heap de mínimo, o nível um é o mais prioritário. O gerente foi criado com apenas uma thread para execução das tarefas.

Após a inicialização dos objetos, foram criadas três tarefas de prioridade dois, e que demoram cinco segundos para serem executadas. Após isso, a thread principal espera três segundos antes de criar uma nova tarefa, agora de prioridade um, que também demora cinco segundos para executar.

A intenção do teste é verificar que as tarefas de nível dois são enfileiradas antes da de nível um, porém a de nível um passará a frente e será executada antes. Além disso, é possível verificar por meio desse teste que tarefas de mesma prioridade são executadas em ordem FIFO, de acordo com a ordem de inserção na fila de prioridade.

Código 24 – Verificação Escalonador com Prioridade

```
public static async void PrioritySimpleExample()
{
    PriorityTaskSchedulerManager manager = new
        PriorityTaskSchedulerManager(1);
    PriorityTaskSchedulerWorker worker1 = new
        PriorityTaskSchedulerWorker(1, manager);
    PriorityTaskSchedulerWorker worker2 = new
        PriorityTaskSchedulerWorker(2, manager);

    for (int i = 0; i < 3; i++)
    {
        Task.Factory.StartNew(() =>
        {
            Logger.Log($"(Task) Executando tarefa de prioridade: {
                worker2.Priority}");
            Thread.Sleep(TimeSpan.FromSeconds(5));
            Logger.Log($"(Task) Terminada tarefa de prioridade: {worker2
                .Priority}");
        }, CancellationToken.None, TaskCreationOptions.None, worker2);
    }
    Thread.Sleep(TimeSpan.FromSeconds(3));
    Task.Factory.StartNew(() =>
    {
        Logger.Log($"(Task) Executando tarefa de prioridade: {worker1.
            Priority}");
        Thread.Sleep(TimeSpan.FromSeconds(5));
        Logger.Log($"(Task) Terminada tarefa de prioridade: {worker1.
            Priority}");
    }, CancellationToken.None, TaskCreationOptions.None, worker1);
}
```

Saída do programa 24:

```
Time: 20:44:593 - Thread Id: 1 - Criando 1 consumidores
Time: 20:44:614 - Thread Id: 1 - Enfileirando tarefa
de Id: 1 e Prioridade: 2
Time: 20:44:614 - Thread Id: 1 - Enfileirando tarefa
de Id: 2 e Prioridade: 2
Time: 20:44:615 - Thread Id: 1 - Enfileirando tarefa
de Id: 3 e Prioridade: 2
Time: 20:45:619 - Thread Id: 5 - (Consumidora) Começando a
executar tarefa de Id 1 e prioridade: 2
```

```
Time: 20:45:621 - Thread Id: 5 - (Task) Executando tarefa
de prioridade: 2
Time: 20:47:618 - Thread Id: 7 - Enfileirando tarefa
de Id: 4 e Prioridade: 1
Time: 20:50:622 - Thread Id: 5 - (Task) Terminada tarefa
de prioridade: 2
Time: 20:50:622 - Thread Id: 5 - (Consumidora) Começando a
executar tarefa de Id 4 e prioridade: 1
Time: 20:50:623 - Thread Id: 5 - (Task) Executando tarefa
de prioridade: 1
Time: 20:55:624 - Thread Id: 5 - (Task) Terminada tarefa
de prioridade: 1
Time: 20:55:624 - Thread Id: 5 - (Consumidora) Começando a
executar tarefa de Id 2 e prioridade: 2
Time: 20:55:625 - Thread Id: 5 - (Task) Executando tarefa
de prioridade: 2
Time: 21:0:626 - Thread Id: 5 - (Task) Terminada tarefa
de prioridade: 2
Time: 21:0:627 - Thread Id: 5 - (Consumidora) Começando a
executar tarefa de Id 3 e prioridade: 2
Time: 21:0:628 - Thread Id: 5 - (Task) Executando tarefa
de prioridade: 2
Time: 21:5:628 - Thread Id: 5 - (Task) Terminada tarefa
de prioridade: 2
```

A saída do programa demonstra a corretude do escalonador. Primeiramente são criados os objetos do escalonador, o gerente e os dois trabalhadores, de prioridade um e dois. O gerente é criado com apenas uma thread consumidora. As tarefas de Id 1, 2 e 3 são enfileiradas, com prioridade dois. A thread consumidora pega a tarefa de Id 1, e começa a executá-la. Enquanto isso, a tarefa de Id 4 e prioridade um (mais prioritária) é inserida na fila.

Como esperado, a thread consumidora termina a tarefa de Id 1, e parte para a próxima tarefa que é a de Id 4 e prioridade um. Apesar dela ter sido a última a ser inserida, ela é a de maior prioridade, apresentando a corretude de ordem de prioridade. Após isso, as tarefas de Id 2 e 3 são executadas na ordem em que foram inseridas, demonstrando a corretude da ordenação FIFO quando há empate na prioridade.

4.4.1 Discussão

Esse escalonador teve como motivação a falta de recursos do escalonador padrão em poder definir a ordem de execução das tarefas. Dessa maneira, caso haja uma definição de tarefas mais importante que outras, será possível garantir que essas serão executadas assim que possível.

Um exemplo desse tipo de aplicação poderia ser a de execução de procedimentos médicos. Os médicos são alocados em salas, recebendo os pacientes a serem tratados. Uma vez que um paciente começa a ser tratado, não é mais possível parar o tratamento (não-preemptivo). Porém, pacientes com condições mais graves devem passar à frente na fila, havendo diversos níveis de prioridade.

Através do escalonador proposto, um sistema que organizasse esses procedimentos necessitaria apenas da avaliação da prioridade do paciente e após isso, iniciar a tarefa no sistema. Esse tipo de abordagem não seria possível com o escalonador padrão, onde seria necessário por parte do programador se preocupar em criar os mecanismos de ordenação e escalonamento dessas tarefas.

Além disso, o número de médicos atendendo no hospital na vida real é limitado e bem conhecido. No escalonador proposto, esse número poderia ser definido na criação do gerente, através da variável *concurrencyLevel*. Já no escalonador padrão, o número de threads do *thread pool* podem variar de acordo com a quantidade de tarefas na fila, não sendo possível modelar esse tipo de problema diretamente através do escalonador, necessitando a criação de outros mecanismos de gerenciamento.

5 CONCLUSÃO

Cada vez mais há uma necessidade de se aproveitar melhor os recursos computacionais oferecidos pelo avanço da tecnologia. Uma das maneiras de realizar isso é através do uso da computação concorrente, aproveitando processadores *multicore*. Entretanto, a programação concorrente traz desafios para os desenvolvedores com os múltiplos fluxos de execução criados dentro das aplicações. Com isso, as linguagens de programação têm se modernizado, oferecendo mecanismos de concorrência de nível mais alto para facilitar o desenvolvimento de aplicações concorrentes.

A proposta deste trabalho foi de apresentar os diversos recursos de computação concorrente oferecidos pela linguagem C#, desde os mais clássicos como *threads* e primitivas de sincronização, até outros mais complexos como os comandos *async/await* e a biblioteca *PLINQ*.

Primeiramente foram apresentados os mecanismos básicos de computação concorrente, como a criação e uso de *threads* e os mecanismos de sincronização como *locks* e semáforos de diferentes tipos. Em seguida foram apresentadas as estruturas de dados próprias para programação concorrente, e exemplos de utilização da *BlockingCollection* para implementar o padrão produtor/consumidor. Tanto o conhecimento de semáforos como o das estruturas concorrentes como a *ConcurrentQueue* foram utilizados para a criação de um escalonador de tarefas com prioridade.

Após isso foi apresentada a biblioteca *PLINQ*, que tem como objetivo trazer facilidade no uso do processamento paralelo de dados, junto com os ganhos de desempenho que essa abordagem pode trazer. Em todos os testes, a adição de poucos comandos e funções da biblioteca proporcionou ganho de desempenho. No algoritmo de multiplicação de matrizes, a versão utilizando a *PLINQ* com duas e seis threads obteve ganhos de até 2,13 e 5,82 vezes, respectivamente, atendendo as expectativas de paralelização, dobrando a velocidade ao executar com duas *threads*, e chegando bem próximo de seis vezes mais rápido ao executar com seis threads.

Foi apresentado o escalonador de tarefas padrão do .NET, estrutura principal para execução das tarefas lançadas com os comandos *async/await*, e junto com a ferramenta *ILSpy*, foi possível observar as modificações que o compilador realiza em um código assíncrono dentro do C#, como a criação da máquina de estados e mecanismos como o *hot path* para otimização na execução das tarefas. O escalonador padrão foi analisado e junto com ele o funcionamento do *thread pool*, mostrando a utilização da fila global de tarefas e as filas internas de cada *thread* para a otimização do uso de *cache*, além da política de roubo de tarefas. No final, foi realizado um teste de desempenho para avaliar a sobrecarga que esse mecanismo de alto nível traz para a aplicação. Como esperado, o uso de tarefas e todos os seus mecanismos trazem uma perda de desempenho mínima. Considerando

a dificuldade de criação e manutenção de códigos de grandes aplicações, o *async/await* oferece facilidade no uso da computação concorrente sem deixar a desejar no desempenho, principalmente quando utilizado para aplicações com muitas operações de *I/O*, que tendem a ter um processo mais demorado e assim tornando a perda de desempenho menos significativa.

Ao final, foi proposto um escalonador de tarefas com prioridade com intuito de explorar a funcionalidade de extensão e utilização de um escalonador próprio proporcionado pela plataforma .NET e de suprir a falta de recursos do escalonador padrão em definir ordem e prioridade na execução das tarefas. Para a criação do mesmo, foram utilizados os conceitos estudados nos capítulos anteriores, como estruturas de dados concorrentes e mecanismos de sincronização. Além disso, foi necessário um estudo aprofundado sobre como o escalonador padrão funciona e como é possível implementar e utilizar um escalonador próprio. Através desse estudo foi possível observar que a extensibilidade da linguagem torna a criação de políticas de execução diferentes algo mais fácil, provendo ao programador mecanismos de personalização e modelagem de problemas específicos para sua área, como o exemplo demonstrado de uma aplicação para gerenciamento de atendimento de pacientes em um hospital.

O C# e a plataforma .NET se mostraram capazes de oferecer suporte a mecanismos básicos de concorrência até mecanismos de mais alto nível. A principal fonte de informações para este trabalho foi a documentação oficial da linguagem que demonstrou ser, na maioria das vezes, bem completa. Desde 2016[1], a plataforma .NET se tornou *open-source* e também possível de ser executada em plataformas Linux, tornando-se uma opção mais concebível para a criação de sistemas e também para utilização no estudo de mecanismos e ferramentas para computação concorrente.

Como trabalho futuro, um estudo mais aprofundado de outros modelos de escalonamento poderia ser realizado. Isso, em conjunto com a implementação através da extensão da classe *TaskScheduler*, pode trazer uma visão mais ampla na utilização de escalonadores diferentes dentro da plataforma .NET, com relação a desempenho e funcionalidades não atendidas. Além disso, esse trabalho poderia gerar uma biblioteca com foco em escalonadores com políticas diferentes da padrão.

[1] https://dotnet.microsoft.com/platform/support/policy/dotnet-core

REFERÊNCIAS

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844.

KERRISK, M. **sem_overview(7) - Linux manual page**. 2006. Disponível em: https://man7.org/linux/man-pages/man7/sem_overview.7.html. Acesso em: 27 jul.2021.

LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. **IEEE Transactions on Computers C-28**, v. 9, p. 690–691, September 1979. Disponível em: https://www.microsoft.com/en-us/research/publication/make-multiprocessor-computer-correctly-executes-multiprocess-programs/.

MAZIERO, C. **Sistemas Operacionais: Conceitos e Mecanismos**. [S.l.: s.n.], 2020. ISBN 978-85-7335-340-2.

.NET Foundation. **BenchmarkDotNet**. 2013. Disponível em: https://benchmarkdotnet.org/index.html. Acesso em: 28 jul.2021.

.NET Team. **C# Versão 5.0**. 2012. Disponível em: https://docs.microsoft.com/pt-br/dotnet/csharp/whats-new/csharp-version-history#c-version-50. Acesso em: 28 jul.2021.

.NET Team. **Named mutex not supported on Unix 5211**. 2016. Disponível em: https://github.com/dotnet/runtime/issues/5211. Acesso em: 27 jul.2021.

.NET Team. **CLR Threading Overview**. 2017. Disponível em: https://github.com/dotnet/coreclr/blob/master/Documentation/botr/threading.md. Acesso em: 27 jul.2021.

.NET Team. **System Collections Concurrent Namespace**. 2020. Disponível em: https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent?view=netcore-3.1. Acesso em: 27 jul.2021.

.NET Team. **BlockingCollection<T> Class**. 2021. Disponível em: https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1?view=netcore-3.1. Acesso em: 27 jul.2021.

.NET Team. **TaskScheduler Class**. 2021. Disponível em: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.taskscheduler?wt.mc_id=MVP&view=net-5.0#the-global-queue-vs-local-queues. Acesso em: 28 jul.2021.

.NET Team. **ThreadPool Class**. 2021. Disponível em: https://docs.microsoft.com/en-us/dotnet/api/system.threading.threadpool?view=netcore-3.1. Acesso em: 28 jul.2021.

.NET Team. **A Tour of the C# language**. 2021. Disponível em: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. Acesso em: 27 jul.2021.

APÊNDICES

APÊNDICE A – MÉTODO DE MÚLTIPLAS CHAMADAS ASSÍNCRONAS DESCOMPILADO

Código 25 – Método de múltiplas chamadas assíncronas descompilado

```
private void MoveNext()
        {
                int num = <>1__state;
                try
                {
                        TaskAwaiter<string> awaiter3;
                        TaskAwaiter awaiter2;
                        TaskAwaiter<string> awaiter;
                        switch (num)
                        {
                        default:
                                <client>5__1 = new HttpClient();
                                awaiter3 = <client>5__1.GetStringAsync("
                                    https://google.com").GetAwaiter();
                                if (!awaiter3.IsCompleted)
                                {
                                        num = (<>1__state = 0);
                                        <>u__1 = awaiter3;
                                        <DoubleAsyncAwait>d__4
                                            stateMachine = this;
                                        <>t__builder.
                                            AwaitUnsafeOnCompleted(ref
                                            awaiter3, ref stateMachine);
                                        return;
                                }
                                goto IL_0095;
                        case 0:
                                awaiter3 = <>u__1;
                                <>u__1 = default(TaskAwaiter<string>);
                                num = (<>1__state = -1);
                                goto IL_0095;
                        case 1:
                                awaiter2 = <>u__2;
                                <>u__2 = default(TaskAwaiter);
                                num = (<>1__state = -1);
                                goto IL_010c;
                        case 2:
                                {
                                        awaiter = <>u__1;
                                        <>u__1 = default(TaskAwaiter<
```

```
                                        string>);
                                num = (<>1__state = -1);
                                break;
                        }
                        IL_010c:
                        awaiter2.GetResult();
                        awaiter = <client>5__1.GetStringAsync("
                            https://twitter.com").GetAwaiter();
                        if (!awaiter.IsCompleted)
                        {
                                num = (<>1__state = 2);
                                <>u__1 = awaiter;
                                <DoubleAsyncAwait>d__4
                                    stateMachine = this;
                                <>t__builder.
                                    AwaitUnsafeOnCompleted(ref
                                    awaiter, ref stateMachine);
                                return;
                        }
                        break;
                        IL_0095:
                        <>s__4 = awaiter3.GetResult();
                        <task>5__2 = <>s__4;
                        <>s__4 = null;
                        awaiter2 = DoSomething().GetAwaiter();
                        if (!awaiter2.IsCompleted)
                        {
                                num = (<>1__state = 1);
                                <>u__2 = awaiter2;
                                <DoubleAsyncAwait>d__4
                                    stateMachine = this;
                                <>t__builder.
                                    AwaitUnsafeOnCompleted(ref
                                    awaiter2, ref stateMachine);
                                return;
                        }
                        goto IL_010c;
                }
                <>s__5 = awaiter.GetResult();
                <task2>5__3 = <>s__5;
                <>s__5 = null;
                Logger.Log("Terminado double async");
        }
        catch (Exception exception)
        {
                <>1__state = -2;
                <client>5__1 = null;
```

```
                        <task>5__2 = null;
                        <task2>5__3 = null;
                        <>t__builder.SetException(exception);
                        return;
                    }
                    <>1__state = -2;
                    <client>5__1 = null;
                    <task>5__2 = null;
                    <task2>5__3 = null;
                    <>t__builder.SetResult();
                }
```

APÊNDICE B – CÓDIGO PARA TESTE DE DESEMPENHO ASSÍNCRONO.

Código 26 – Código para Teste de desempenho assíncrono

```
public class StockPriceBenchmark
{
    [Params(1_000, 10_000, 100_000)] public int Tam;

    [Benchmark(Baseline = true)]
    public decimal FullSync()
    {
        var _sp = new StockPrices(Tam);
        return _sp.FullSync((Tam-1).ToString());
    }

    [Benchmark]
    public async Task<decimal> FullAsync()
    {
        var _sp = new StockPrices(Tam);
        return await _sp.FullAsync((Tam-1).ToString());
    }

    [Benchmark]
    public async Task<decimal> FakeAsync()
    {
        var _sp = new StockPrices(Tam);
        return await _sp.AsyncThatIsSync((Tam-1).ToString());
    }
}
```

APÊNDICE C – ESCALONADOR DE PRIORIDADE - GERENTE

Código 27 – Classe PriorityTaskSchedulerManager

```
public class PriorityTaskSchedulerManager
{
    private readonly SemaphoreSlim _semaphore = new SemaphoreSlim(1, 1);
    private readonly PriorityQueue<PriorityTask> _taskQueue = new
        PriorityQueue<PriorityTask>();
    private readonly Dictionary<int, PriorityTaskSchedulerWorker>
        _schedulers = new Dictionary<int, PriorityTaskSchedulerWorker>();

    public PriorityTaskSchedulerManager(int concurrencyLevel = -1)
    {
        int c = concurrencyLevel <= 0 ? Environment.ProcessorCount :
            concurrencyLevel;
        Logger.Log($"Criando {c} consumidores");

        // Cria threads consumidoras
        for (int i = 0; i < c; i++)
        {
            new Thread(ConsumerThread).Start();
        }
    }

    public void EnqueuePriority(PriorityTask pt)
    {
        _semaphore.Wait();
        Logger.Log($"Enfileirando tarefa de Id: {pt.Id} e Prioridade: {
            pt.Priority}");
        _taskQueue.Enqueue(pt);
        _semaphore.Release();
    }

    private void ConsumerThread()
    {
        while (true)
        {
            _semaphore.Wait();
            var success = _taskQueue.TryDequeue(out PriorityTask pt);
            _semaphore.Release();
            if (success)
            {
                if(_schedulers.TryGetValue(pt.Priority, out
                    PriorityTaskSchedulerWorker sch))
```

```
                {
                    Logger.Log($"(Consumidora) Comecando a executar
                        tarefa de Id {pt.Id} e prioridade: {pt.Priority}"
                        );
                    sch.ExecuteTask(pt.Task);
                }
            }
            else
            {
                Thread.Sleep(TimeSpan.FromSeconds(1));
                Logger.Log("sleeping 1s");
            }
        }
    }

    public bool AddScheduler(PriorityTaskSchedulerWorker scheduler)
    {
        try
        {
            _schedulers.Add(scheduler.Priority, scheduler);
            return true;
        }
        catch
        {
            Logger.Log($"Escalonador de prioridade {scheduler.Priority}
                ja existe neste gerente");
            return false;
        }
    }
}
```

APÊNDICE D – ESCALONADOR DE PRIORIDADE - TRABALHADOR

Código 28 – Classe PriorityTaskSchedulerWorker

```
public class PriorityTaskSchedulerWorker : TaskScheduler
{
    private readonly PriorityTaskSchedulerManager _manager;
    public int Priority { get; set; }
    private static int _taskCounter = 0;
    private readonly ConcurrentQueue<Task> _tasks = new ConcurrentQueue<
        Task>();

    public PriorityTaskSchedulerWorker(int priority,
        PriorityTaskSchedulerManager manager)
    {
        _manager = manager;

        Priority = priority;
        if (!manager.AddScheduler(this))
        {
            throw new Exception("Could not create priorityTaskScheduler"
                );
        }
    }

    public bool ExecuteTask(Task t)
    {
        _tasks.TryDequeue(out var deq);
        return TryExecuteTask(t);
    }

    protected override IEnumerable<Task>? GetScheduledTasks()
    {
        return _tasks;
    }

    protected override void QueueTask(Task task)
    {
        _tasks.Enqueue(task);
        _manager.EnqueuePriority(new PriorityTask(Interlocked.Increment(
            ref _taskCounter), Priority, task));
    }

    protected override bool TryExecuteTaskInline(Task task, bool
        taskWasPreviouslyQueued)
```

```
        {
            return true;
        }
    }
```

ANEXOS

ANEXO A – IMPLEMENTAÇÃO FILA DE PRIORIDADE - VISUAL STUDIO MAGAZINE

Código 29 – Implementação Fila de Prioridade - Visual Studio Magazine

```
using System;
using System.Collections.Generic;

namespace TCC.Helpers
{
    // This project uses .net core 3.1, but in .NET 6 there is a
       official implementation of a PriorityQueue
    // https://github.com/dotnet/runtime/blob/main/src/libraries/System.
       Collections/src/System/Collections/Generic/PriorityQueue.cs
    // From http://visualstudiomagazine.com/articles/2012/11/01/priority
       -queues-with-c.aspx
    public class PriorityQueue<T> where T : IComparable<T>
    {
        private List<T> data;

        public PriorityQueue()
        {
            this.data = new List<T>();
        }

        public void Enqueue(T item)
        {
            data.Add(item);
            int ci = data.Count - 1; // child index; start at end
            while (ci > 0)
            {
                int pi = (ci - 1) / 2; // parent index
                if (data[ci].CompareTo(data[pi]) >= 0)
                    break; // child item is larger than (or equal)
                       parent so we're done
                T tmp = data[ci];
                data[ci] = data[pi];
                data[pi] = tmp;
                ci = pi;
            }
        }

        public bool TryDequeue(out T obj)
        {
            obj = default(T);
```

```
        if (data.Count == 0) return false;
        obj = Dequeue();
        return true;
    }

    public T Dequeue()
    {
        // assumes pq is not empty; up to calling code
        int li = data.Count - 1; // last index (before removal)
        T frontItem = data[0]; // fetch the front
        data[0] = data[li];
        data.RemoveAt(li);

        --li; // last index (after removal)
        int pi = 0; // parent index. start at front of pq
        while (true)
        {
            int ci = pi * 2 + 1; // left child index of parent
            if (ci > li) break; // no children so done
            int rc = ci + 1; // right child
            if (rc <= li && data[rc].CompareTo(data[ci]) < 0
            ) // if there is a rc (ci + 1), and it is smaller than
                left child, use the rc instead
                ci = rc;
            if (data[pi].CompareTo(data[ci]) <= 0)
                break; // parent is smaller than (or equal to)
                    smallest child so done
            T tmp = data[pi];
            data[pi] = data[ci];
            data[ci] = tmp; // swap parent and child
            pi = ci;
        }

        return frontItem;
    }

    public T Peek()
    {
        T frontItem = data[0];
        return frontItem;
    }

    public int Count()
    {
        return data.Count;
    }
```

```
        public override string ToString()
        {
            string s = "";
            for (int i = 0; i < data.Count; ++i) s += data[i].ToString()
                + " ";
            s += "count = " + data.Count;
            return s;
        }

        public bool IsConsistent()
        {
            // is the heap property true for all data?
            if (data.Count == 0) return true;
            int li = data.Count - 1; // last index
            for (int pi = 0; pi < data.Count; ++pi)
            {
                // each parent index
                int lci = 2 * pi + 1; // left child index
                int rci = 2 * pi + 2; // right child index

                if (lci <= li && data[pi].CompareTo(data[lci]) > 0)
                    return false; // if lc exists and it's greater than
                        parent then bad.
                if (rci <= li && data[pi].CompareTo(data[rci]) > 0)
                    return false; // check the right child too.
            }

            return true; // passed all checks
        }

        // IsConsistent
    }
}
```