BOSCH
Invented for life

Docupedia Export

Author: Goncalves Donathan (CtP/ETS)
Date: 11-May-2023 13:32

Table of Contents

- Sistemas Operacionais, Escalonamento de Tarefas e Preempção
- Deadlocks, Starvation, Priority Inversion e Algoritmo do Avestruz
- Desenvolvimento Paralelo, Threads e Lei de Amdahl
- Thread-Safe, SpinLocks, Mutex, Semaphore Monitor e Lock
- Exemplo: Computando uma gaussiana e avaliando sua qualidade
- Exemplo: Soma de 100 milhões de números
- Async e Await

1 Sistemas Operacionais, Escalonamento de Tarefas e Preempção

Ao compreender o funcionamento de um processador contemplamos que é impossível um único processador executar duas tarefas ao mesmo tempo. Dito isso, é impossível que em um computador controlemos a interação do cursor e a execução de uma aplicativo qualquer. O que acontece na verdade é que existem sistemas que gerenciam a execução de multiplas aplicações alternando entre elas, esses sistemas são os Sistemas Operacionais. Neles, cada aplicação torna-se uma tarefa e eles trocados rapidamente no processador de forma que se torna impossível que o usuário perceba que na verdade uma tarefa está executando por pouquissímo tempo e então ficando parado no resto do tempo. É imperceptível que uma aplicação fica congelada por algum tempo já que o mesmo trata-se de poucos microssegundos. Lembrando: Em 1 microssegundo, um computador é capaz de executar cerca de poucas milhares de instruções e fazer com que uma tarefa realmente avance em seu processamento. Ao mesmo tempo, 50 microssegundos é tão pouco tempo que em um segundo teríamos 20 mil tarefas colocadas ao processador. Isso significa que mesmo que você possua 1000 tarefas no seu computador, uma aplicação que você está usando entra no processador e executa poucas millhões de instruções.

Para trocar as terefas do processador por outro realizamos uma troca de contexto: Salvamos todas os registradores na memória e compiamos os registradores que estavam salvos da tarefa que irá entrar no processador. Em geral trabalhamos com interrupções: Recursos do processador que "pausam" o processador obrigando a execução de uma determinada secção de código. Note que não tem como o Sistema Operacional agir enquanto outra aplicação está rodando. Por isso, é necessário que o hardware pare uma tarefa e guarde qual é a linha onde o código está para futuro retorno. O nome dessa operação é chamada de Preempção.

Após a preempção o processador é entregue ao Sistema Operacional (que assim não é nada mais nada menos que uma tarefa) que irá realizar o Escalonamento de Tarefas: Escolher uma terefa para entrar no processador. O escalonamento pode ser de várias formas: Escolher qualquer uma; Considerar quanto tempo faz que uma tarefa não é executada; Considerar um nível de prioridade para que uma tarefa ganhe o processador; E assim por diante.

Outra operação que resulta em uma Preempção é o acesso a recursos. Se duas tarefas estão trabalhando com o mesmo recurso na memória, existem alguns cuidados que se necessita ter com a aplicação para que ela não perca dados e tenha comportamente inesperado. Assim quando uma tarefa acessa um recuros já usado ou que demora para ser acessado a tarefa é posta para dormir e aguardar, deixando assim, o processador.

2 Deadlocks, Starvation, Priority Inversion e Algoritmo do Avestruz

Alguns problemas acontecem quando estamos em sistemas multi-tarefas:

- Starvation: A Inanição acontece quando o Sistema Operacional não percebe que uma tarefa deve executar por causa do algoritmo e de como o algoritmo funciona. Por exemplo existem tarefas com prioridades muito altas que nunca deixam uma tarefa executar e ela acaba nunca executando.
- Priority Inversion: A inversão de prioridade acontece quando uma tarefa de menor prioridade toma o processador e não deixa uma tarefa de maior prioridade executar. Isso acontece pois uma tarefa mais importante pode estar esperando um recurso que a tarefa de baixa prioridade está usando.
- Deadlocks: Deadlocks talvez sejam os problemas mais danosos dessa lista. Um impasse acontece de várias formas, mas a mais básica é quando uma tarefa A e outra B precisam ambas usar recursos R e S. Logo após a tarefa A tomar posse do recurso R ela sofre preempção pelo sistema operacional que agora da o controle para a tarefa B. A tarefa B toma posse do recurso S e então tenta tomar posse do recurso R, mas percebe que já está em uso por outra tarefa e então é preemptada pelo SO que agora devolve o processador para a tarefa A. A tarefa A ao retornar percorre o código e agora chega o momento de torma posse do recurso S, que já está em uso, logo A é preemptada também. Assim A e B estão como inativas: A espera o recurso S que B está usando e B espera o recurso R que A está usando. Logo elas nunca serão executadas e pior: Os recursos R e S ficarão retidos para sempre e qualquer outra tarefa que precise deles também cairá num impasse que não será capaz de executar.

Muitos desses problemas podem ser resolvidos com o Algoritmo do Avestruz mostrado abaixo:

```
1  public void OstrichAlgorithm()
2  {
3
4  }
```

Comicamente, o Algoritmo do Avestruz é basicamente ignorar o problema. Deadlocks são raros e alguns outros ocorrências dos problemas acimas podem ser raros ou até não impactarem o sistema severamente. Então não fazer nada, simplismente pode ser uma boa opção. Outras soluções podem gastar processamento desnecessário em 99.9% dos cenários para resolver um problema que ocorre em 0.1% destes. Mesmo assim são problemas que devemos conhecer porquê nem sempre o algoritmo do avestruz é uma opção. Ao desenvolver aplicações que trabalham com muitas tarefas é importante conhecer os possíveis problemas que a mesma resultará.

3 Desenvolvimento Paralelo, Threads e Lei de Amdahl

Podemos aumentar o desempenho de um uma aplicação com desenvolvimento paralelo, ou seja, a criação de várias tarefas para a mesma aplicação o que chamamos de Threads. Uma Thread é um código executado em paralelo a aplicação inicial. Note que em um sistema mono-core isso apenas divide o trabalho em vários componentes de código. Porém, em sistemas muilt-core, cada core pode executar uma parte do trabalho em um processador diferente. Deadlocks pode ocorrer aqui e é necessário saber trabalhar com eles, contudo, ainda sim podemos conseguir um grande speedup (ganho de velocidade) separando o código em várias Threads.

É importante perceber que existem processos não paralelizáveis que precisam ser executados sequêncialmente, caso contrário, não seria possível obter o resultado correto. Por exemplo a sequência de Fibonacci, que precisa dos elementos anteriores para que o mesmo funcione. Assim separar o processo em 2 seria complicado para não dizer impossível.

A Lei de Amdahl diz que o aumento do desempenho de um sistema dado a melhora e uma parte do sistema é proporcional ao impacto da parte do sistema no todo. Na programação paralela ela serve para indicar o aumento máximo teórico de uma aplicação cuja parte paralelizável equivale a uma proporção **p** e a quantidade cores é **k** onde o trabalho pode ser dividido.

O ganho de velocidade **S** é escrito como o trabalho inicial **T** sobre o trabalho com as melhorias aplicadas:

$$S = \frac{T}{T'}$$

Com o paralelismo a parte do código paralelisada é dividida entre os processadores:

$$T' = T(\frac{p}{k} + 1 - p)$$

Assim o ganho teórico de velocidade é dado por:

$$S = \frac{T}{T'} = \frac{T}{T(\frac{p}{k}+1-p)} = \frac{1}{\frac{p}{k}+1-p} = \frac{k}{p+k-pk} = \frac{k}{p+k(1-p)}$$

Ou seja, caso uma aplicação possua 50% de código (com 50% de código queremos dizer 50% de trabalho não quantidade de linhas) paralelizável em um sistema com 4 cores teremos o seguinte ganho teórico:

$$S = \frac{4}{0.5+4*0.5} = \frac{4}{2.5} = 1.6$$

Ou seja, o código deve se tornar 60% mais rápido. Se antes ele demorava 16 segundos agora ele deve demorar apenas 10.

4 Thread-Safe, SpinLocks, Mutex, Semaphore Monitor e Lock

Podemos ganhar muito com Threads, mas precisamos aprender a fazer com que tudo funcione, e para isso temos que garantir que o processo seja Thread-Safe. Isso significa que nenhum dado é perdido e nenhuma condição de corrida (onde você não sabe qual código irá acessar um recurso primeiro) irá ocorrer. Para isso, no exemplo a seguir usaremos muitos recursos de controle de dados e criação de Threads em C# para optimizar a computação de um trabalho.

5 Exemplo: Computando uma gaussiana e avaliando sua qualidade

Realizaremos o seguinte experimento para compreender a disputa e resolução dessa disputa por recursos. Para isso realizaremos a seguinte computação: Iremos computar uma distribuição Gaussiana que pode ser calculada com a soma de muitos valores aleatórios entre 0 e 1. Depois usaremos conceitos de estatística para apontar se a distribuição tem as qualidades de uma distrubuição gaussiana. Repetiremos o processo várias vezes para tirar uma média do resultado:

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4  using System.Collections.Concurrent;
5
6  int N =
   1000;                                    // Tamanho da amostras
7  int M =
   5000;                                    // Amostras a serem utilizadas
8  int K =
   50;                                      // Vezes que o Monte Carlo será
   executado
9
10 var seed = DateTime.Now.Millisecond;
11 var rand = new Random(seed);                    // Objeto para gerar
   números aleatórios
12 float sum = 0f;
13 float sqSum = 0f;
14 float[] dist = new float[M];
15
16 float aval = 0;
17 double time = 0;
18 for (int i = 0; i < K; i++)
19 {
20     for (int j = 0; j < dist.Length; j++)
21         dist[j] = 0;
22     sum = 0f;
23     sqSum = 0f;
24
25     var dt = DateTime.Now;
26
   run();                                       // Código entra aqui
27     var span = DateTime.Now - dt;
28     time += span.TotalMilliseconds;
29     aval += avaliate();
30 }
31
32 Console.WriteLine($"{100 * aval / K}% of precision in {time / K} ms per
   computation.");
33
34 void compute(int index)
35 {
36     var value = 0f;
37     for (int i = 0; i < N; i++)
38         value += rand.NextSingle();              // Geramos muitos valores
   aleatórios e somamos para tirar a média
39     value /= N;
40     dist[index] = value;
```

```
41      sum += value;
42      sqSum += value * value;
43  }
44
45  float avaliate()
46  {
47      float avg = sum / M;
48
49      float std = sqSum - sum * sum / M;
50      std /= M;
51      std = MathF.Sqrt(std);
52
53      int count1 = 0;
54      int count2 = 0;
55      int count3 = 0;
56      foreach (var y in dist)
57      {
58          if (y > avg + 2 * std)
59          {
60              count1++;
61              count2++;
62              count3++;
63          }
64          else if (y > avg + std)
65          {
66              count1++;
67              count2++;
68          }
69          else if (y > avg)
70          {
71              count1++;
72          }
73      }
74
75      var exp1 = M * 0.5f;
76      var exp2 = M * 0.159f;
77      var exp3 = M * 0.022f;
78
79      var err1 = MathF.Abs(count1 - exp1) / exp1;
80      var err2 =  MathF.Abs(count2 - exp2) / exp2;
81      var err3 = MathF.Abs(count3 - exp3) / exp3;
82
83      var prec = 1f - (err1 + err2 + err3) / 3;
84      return prec < 0 ? 0 : prec;
85  }
86
87  void run()
88  {
89      for (int i = 0; i < M; i++)
90          compute(i);
91  }
```

A função *run* executa *compute* M vezes até que a computação esteja concluida. O resultado é >95% de precisão e ~80ms por processamento. Note que estamos em um quad-core onde, teoricamente, cada soma pode ser realizada em paralelo, logo p=1. Assim esperamos aumentar a velocidade da aplicação em 4 vezes. Agora vamos tentar melhorar a execução aprendendo a utilizar Threads:

```
1   void threadWrongRun()
2   {
```

```
3       for (int i = 0; i < M; i++)
4       {
5           Thread thread = new Thread(() =>
6           {
7               compute(i);
8           });
9           thread.Start();
10      }
11  }
```

Pelo nome da função você sabe que algo está errado, mas o quê? Ao executar temos o seguinte erro: '*Index was outside the bounds of the array.*' Porquê disso acontecer é simples: Ao criarmos a variável *i* e passarmos ela dentro da função anonima na Thread estamos fazendo com que essa função capture a variável. Isso significa que quando a thread rodar ela usará o valor atual de *i* no momento da execução. O problema é que algumas threads só executarão de fato após o for ser completo. Quando isso acontece *i* já vale *N* (o que ocasionou na quebra do for) e então todas as threads usam o valor atual de *i* que é *N* resultando num estouro de vetor dentro da função *compute*.

Para resolver isso precisamos fazer uma cópia imutável de *i*:

```
1   void threadRun()
2   {
3       for (int i = 0; i < M; i++)
4       {
5           int j = i;
6           Thread thread = new Thread(() =>
7           {
8               compute(j);
9           });
10          thread.Start();
11      }
12  }
```

Agora o código funciona pois cada Thread possui uma cópia da variável *i*. Note que *j* é definido a cada loop, assim existem vários *j's* na memória e não só 1 como poderia se imaginar. Cada um com um valor fixo.

Mas de repente surge um novo problema: Obtivémos <70% e ~680 ms para a execução desta solução. Perdemos tanto precisão quanto desempenho. O motivo é porque a criação e gerenciamento de threads é pesada, criamos *M* threads que no caso são 5000 threads sendo que só temos 4 núcleos. A queda na precisão é interessante de se avaliar também. A classe Random não é Thread-Safe, isso significa que podemos literalmente quebrar a classe se usarmos ela várias vezes ao mesmo tempo. Vamos resolver um problema de cada vez. Primeiramente o desempenho. Para isso vamos utilizar a classe Parallel que usa um *ThreadPool*, ou seja, reuitliza as threads e considera a quantidade de cores na hora de executá-las:

```
1   void wrongParallelForRun()
2   {
3       Parallel.For(0, M, i =>
4       {
5           compute(i);
6       });
7   }
```

Um dado importante: Ao executar a função *start* do objeto thread nós iniciamos a thread em segundo plano e continuamos a executar o programa. O Parallel para o programa na linha da chamada de qualquer função e só continua a executar quando o trabalho em paralelo for completamente executado.

Bom, resolvemos noss problema de desemepnho, embora ainda não esteja bom, alcançando ~180 ms de execução. Mas continua errado, visoto que nossa precisão é <1%. Agora entra o trabalho de tentar tornar nossa aplicação Thread-Safe:

```
1  void spinLockWrongParallelForRun()
2  {
3      SpinLock sl = new SpinLock();
4      bool blocked = false;
5
6      Parallel.For(0, M, i =>
7      {
8          try
9          {
10             sl.Enter(ref blocked);
11             compute(i);
12         }
13         finally
14         {
15             if (blocked)
16                 sl.Exit();
17         }
18     });
19 }
```

Novamente, o '*Wrong'* no nome já mostra que algo não está certo. O mesmo erro da primeira tentativa: '*The tookLock argument must be set to false before calling this method.*'. A mensagem é diferente, o erro é o mesmo. Ao deixar a variável *blocked* para fora do Parallel For estamos comprometendo o funcionamento da struct SpinkLock. Vamos corrigir isso antes de falarmos da técnica empregada:

```
1  void spinLockParallelForRun()
2  {
3      SpinLock sl = new SpinLock();
4
5      Parallel.For(0, M, i =>
6      {
7          bool blocked = false;
8          try
9          {
10             sl.Enter(ref blocked);
11             compute(i);
12         }
13         finally
14         {
15             if (blocked)
16                 sl.Exit();
17         }
18     });
19 }
```

O SpinLock é uma implementação simples que bloqueia um Thread para permitir que apenas uma thread entre dentro da zona crítica onde usamos o *compute*. Como todas as threads enxergam o mesmo objeto SpinLock, ao chamar a função *Enter*, se alguma thread já chamou essa função e ainda não chamou a função Exit, então isso significa que os recursos estão sendo utilizados e a thread entra em um loop até que outra thread passe na função Exit. Com elas melhoramos em todas as perspectivas tendo >95% de precisão e ~120 ms de execução. Ainda não estamos próximos da precisão do código sequêncial, mas corrigimos a questão da precisão. Vejamos outras opções ainda mais leves que o SpinLock:

```
1   void mutexParallelForRun()
2   {
3       using Mutex mt = new Mutex();
4
5       Parallel.For(0, M, i =>
6       {
7           mt.WaitOne();
8           compute(i);
9           mt.ReleaseMutex();
10      });
11  }
```

O Mutex é como o SpinLock. O que muda é o que acontece com a thread bloqueada, Equanto o SpinLock coloca a thread em um loop a Mutex coloca a tarefa para domir no Sistema Operacional. Isso sigfinica que ela não executará nem entrerá no processador até que seja acordada pela Mutex quando a secção crítica for liberada. Isso evita que ela tome o processador para ficar presa em um loop, dando o processador para threads que realmente precisião dele. Note que colocar uma thread para dormir custa bastante, se a operação dentro da secção crítica for pequena, vale mais a pena usar outra o SpinLock. Com a mutex mantivemos a precisão >95%, mas economizamos um pouco e agora usamos apenas ~110 ms por execução.

```
1   void semaphoreParallelForRun()
2   {
3       int count = 4;
4       using Semaphore sm = new Semaphore(count, count);
5
6       Parallel.For(0, M, i =>
7       {
8           sm.WaitOne();
9           compute(i);
10          sm.Release();
11      });
12  }
```

O semáforo também vale a pena ser comentado. Ele é como um Mutex, porém ele permite a passagem de uma quantidade de *N* threads na secção crítica. Note que um semáforo unitário é o mesmo que um Mutex. O código acima, por permitir que 4 threads adentrem a secção crítica teve o desempemho destruído. Mesmo assim é útil em algumas situações.

```
1   void monitorParallelForRun()
2   {
3       Parallel.For(0, M, i =>
4       {
5           Monitor.Enter(rand);
6           compute(i);
7           Monitor.Exit(rand);
8       });
9   }
```

O monitor é um Mutex que não pode se compartilhado enter várias aplicações não estando no nível do sistema operacional. Mas ele é bem mais leve que um Mutex ou SpinLock. Além disso, ele permite que você associe um objeto que deve ser protegido na secção crítica. No caso, todas as threads que estejam com a mesma referência de *rand*, estarão sujeitas a exclusão mútua, ou seja, serão barradas e colocadas fora de execução até que a área seja liberada. O código acima mantém a precisão e roda em ~90 ms.

```
1  void monitorWithRefParallelForRun()
2  {
3      Parallel.For(0, M, i =>
4      {
5          bool lockTaken = false;
6          try
7          {
8              Monitor.Enter(rand, ref lockTaken);
9              compute(i);
10         }
11         finally
12         {
13             if (lockTaken)
14             {
15                 Monitor.Exit(rand);
16             }
17         }
18     });
19 }
```

Acima você pode observar um Monitor usado de uma forma um pouco mais cuidadosa/profissional evitando alguns errors com um try/finally e usando um booleano para compreender se realmente a computação entrou na secção crítica ou obteve algum erro antes (e assim não deve chamar Exit). O mesmo código acima pode ser reescrito abaixo de forma simples usando um recurso do C# chamado lock:

```
1  void lockParallelForRun()
2  {
3      Parallel.For(0, M, i =>
4      {
5          lock(rand)
6          {
7              compute(i);
8          }
9      });
10 }
```

Nosso último exemplo mostra como pode ser simples escrever códigos Thread-Safe em C#. Note que o código é basicamente sequêncial e nunca vai ter desempenho melhor que o tradicional pois todo trabalho está dentor de um lock. Além disso, é bom apontar que se deve evitar lock dentro de outro lock para evitar DeadLocks.

6 Exemplo: Soma de 100 milhões de números

Não iremos sair sem um exemplo real de ganho de performance daqui:

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks;
4
5  var seed = DateTime.Now.Millisecond;
6  var rand = new Random(seed);
7
8  DateTime dt = DateTime.Now;
9  int sum = 0;
10 byte[] arr = new byte[100_000_000];
11 rand.NextBytes(arr);
12 for (int j = 0; j < arr.Length; j++)
13     sum += arr[j];
14 var span = DateTime.Now - dt;
15 Console.WriteLine($"{sum} - {span.TotalMilliseconds}");
16
17 dt = DateTime.Now;
18 sum = 0;
19 int threadCount = 250 * Environment.ProcessorCount; // Obtém o número de
   Processadores Lógicos (no caso 250 threads por processador)
20 Parallel.For(0, threadCount, i =>
21 {
22     int _sum = 0;
23     byte[] arr = new byte[100_000_000 / threadCount];
24     lock (rand)
25     {
26         rand.NextBytes(arr);
27     }
28
29     for (int j = 0; j < arr.Length; j++)
30         _sum += arr[j];
31
32     Interlocked.Add(ref sum, _sum); // Realiza uma Soma Thread-Safe
33 });
34 span = DateTime.Now - dt;
35 Console.WriteLine($"{sum} - {span.TotalMilliseconds}");
```

Enquanto sequêncialmente levamos ~1200 ms, paralelamente levamos ~900ms. Um ganho pequeno mas real.

Ainda existem outros recursos não mostrados, como o ConcurrentQueue e outras coleções ThreadSafe no *namespace System.Collections.Concurrent*.

7 Async e Await

Outro uso para threads é a programação assíncrona, ou seja, rodar algo em segundo plano para não comprometer o funcionamento da aplicação principal. Observe o seguinte exemplo:

```
1  using System;
2  using System.Threading;
3
4  while (true)
5  {
6      var key = Console.ReadKey(true);
7      if (key.Key == ConsoleKey.Escape)
8          return;
9
10     MostrarNumeroComplicado();
11 }
12
13 long calcularNumeroComplicado()
14 {
15     Thread.Sleep(2000); // Fazendo calculo
16     return 3141592653589793238;
17 }
18
19 void MostrarNumeroComplicado()
20 {
21     var resultado = calcularNumeroComplicado();
22     Console.WriteLine(resultado);
23 }
```

Ao clicar em qualquer botão vários caractéres entram no nosso buffer e ao entrar no *calcularNumeroComplicado* a aplicação fica congelada enquanto espera o cálculo (aqui representado pela função Sleep que faz a função dormir por 2000 ms, ou 2 segundos). Ao segurar qualquer tecla exceto o Esc por alguns segundos a aplicação fica trava mostrando o número a cada 2 segundos e não responde mais ao Esc do usuário. Em uma tela normal isso faria com que a aplicação travasse enquanto espera calcular algo, ou conectar ao banco de dados, por exemplo.

```
1  void MostrarNumeroComplicadoAsync()
2  {
3      var estado = calcularNumeroComplicadoAsync();
4      Thread thread = new Thread(() =>
5      {
6          while (!estado.EstaCompleto)
7              Thread.Yield();
8
9          Console.WriteLine(estado.resultado);
10     });
11     thread.Start();
12 }
13
14 EstadoAssincrono<long> calcularNumeroComplicadoAsync()
15 {
16     EstadoAssincrono<long> estado = new EstadoAssincrono<long>();
17     var thread = new Thread(() =>
18     {
19         var result = calcularNumeroComplicado();
20         estado.resultado = result;
21         estado.EstaCompleto = true;
```

```
22      });
23      thread.Start();
24      return estado;
25  }
26
27  public class EstadoAssincrono<T>
28  {
29      public T resultado { get; set; }
30      public bool EstaCompleto { get; set; }
31  }
```

Neta nova solução ciramos um estádo assincrono que controla se uma dada operação já terminou e seu resultado. Quando *MostrarNumeroComplicadoAsync* é chamada ela chama *calcularNumeroComplicadoAsync* que por sua vez criará uma Thread que interage com um estado assíncrono, Quando o número for calculado, a thread irá modificar o estado. Esse estádo é imediatamente devolvido e *MostrarNumeroComplicadoAsync* verifica continuament em uma segunda thread para ver se ela já terminou. Caso não tenha terminado ela chama a função Thread.Yield que faz a preempção da Thread atual dando espaço no processador para outra thread que tenha algo a fazer. Quando o calculo termina *EstaCompleto* é definida como verdadeiro, interrompendo o Loop na *MostrarNumeroComplicadoAsync* e fazendo com que a função void seja executada. Ao usar o programa com essa função, todo clique cria uma Thread sendo executada em paralelo mas não deixa de executar o Loop principal, assim, a qualquer momento o usuário pode usar a tecla Esc e fechar a aplicação sem que ela congele.

Felizmente, o C# já tem uma estrutura (que no fundo é bem mais complexa) para trabalhar com esses tipos de sistemas, trata-se do Async e Await. Observe:

```
1   using System;
2   using System.Threading;
3   using System.Threading.Tasks;
4
5   while (true)
6   {
7       var key = Console.ReadKey(true);
8       if (key.Key == ConsoleKey.Escape)
9           return;
10
11      MostrarNumeroComplicadoAsync();
12  }
13
14  long calcularNumeroComplicado()
15  {
16      Thread.Sleep(2000);
17      return 3141592653589793238;
18  }
19
20  Task<long> calcularNumeroComplicadoAsync()
21      => Task<long>.Factory.StartNew(() => calcularNumeroComplicado());
22
23  async void MostrarNumeroComplicadoAsync()
24  {
25      var estado = await calcularNumeroComplicadoAsync();
26      Console.WriteLine(estado);
27  }
```

Task é a classe que representa uma tarefa, ou seja, uma thread que retorna resultado. Usando o Task.Factory.StartNew podemos criar novas tarefas e aguardá-las em funções assíncronas, isto é, com a palavra reservada async antes do retorno, com a palavra await. Aguardá-las faz com que o código pause naquela linha e retorne ao primeiro chamador não assíncrono. Ou seja, ao usar o await no final do código, voltamos a chamada de método de *MostrarNumeroComplicadoAsync*, e continuamos a executar de lá.

Quando o calculo for completo, a execução pausa e volta para linha do await para completar o que estava sendo feito.

```
            using System;
            using System.Threading;
            using System.Threading.Tasks;

1 13        while (true)
2 14        {
3 15            var key = Console.ReadKey(true);
4 18            if (key.Key == ConsoleKey.Escape)
5                   return;
6
7               MostrarNumeroComplicadoAsync();
12          }

            long calcularNumeroComplicado()
            {
12 13 14        Thread.Sleep(2000);
15              return 3141592653589793238;
            }

            Task<long> calcularNumeroComplicadoAsync()
10              => Task<long>.Factory.StartNew(() =>
calcularNumeroComplicado());

            async void MostrarNumeroComplicadoAsync()
8           {
9 11 16         var estado = await calcularNumeroComplicadoAsync();
17              Console.WriteLine(estado);
            }

1-6         Código executando normalmente
7           Chamada de Função
8           Código executando normalmente
9           Chamada de Função
10          Tarefa criada nesta linha
11          await chamado para Tarefa
12-14       Código retorna ao chamado enquanto thread executada
paralelamente
15          Thread terminou
16          Resultado retornado
17          Resultado apresentado na tela
18          Código executando normalmente
```

Você ainda pode fazer uma sequência de funções assincronas. Quando uma função retorna Task<T> e é assícnrona, retorne T ao invés de uma Task:

```
using System;
using System.Threading.Tasks;

var result = await calcularSomaComplicadaAsync();
Console.WriteLine(result);

Task<long> calcularNumeroComplicado()
    => Task<long>.Factory.StartNew(() => 910);

```

```
10
11  Task<long> calcularOutroNumeroComplicado()
12      => Task<long>.Factory.StartNew(() => 90);
13
14  // Criando uma Task como chamadas assíncronas de outras Tasks
15  async Task<long> calcularSomaComplicadaAsync()
16  {
17      var x = await calcularNumeroComplicado();
18      var y = await calcularOutroNumeroComplicado();
19      return x + y; // Não retorna Task<long>, mas sim long, pois é
    assíncrona e já é tratada como uma tarefa
20  }
```

Inclusive, na maioria das vezes, criamos funções assíncronas sobre outras funções assíncronas que já existem.

Outra informação importante é que caso você chame uma função assíncrona sem o await você chamará ela sincronamente. Ela retornará o controle para você assim que criada a Thread paralela. Note que se o restorno for uma Task você estará retornando uma tarefa e executando-a em paralelo.

```
1   using System;
2   using System.Threading;
3   using System.Threading.Tasks;
4
5   var result = await PegarSoma();     // Pegando o número...
6   Console.WriteLine(result);          // 1500
7
8                                       // Retornamos a tarefa e não a
    aguardamos, assim temos um objeto Tarefa e não um número, assim o retorno
    vem antes da tarefa executar
9   var result2 = PegarSoma();          //
    System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1+AsyncStateMachine
    Box`1[System.Int64,Program+<<<Main>$>g__PegarSoma|0_1>d]
10  Console.WriteLine(result2);         // Pegando o número...
11
12  await MostrarSoma();                // Pegando o número...
13                                      // 1500
14
15  // await MostrarSomaVoid();         //não é possível aguardar void
16
17  MostrarSoma();                      // Essa função foi executada
    sincronamente, não aguardando neste ponto e mostrando o Pegando número da
    linha abaixo imediatamente
18  MostrarSomaVoid();                  // Pegando o número...
19                                      // Pegando o número...
20                                      // 1500
21                                      // 1500
22
23  Console.ReadKey(true);
24
25  Task<long> PegarNumero()
26      => Task<long>.Factory.StartNew(() =>
27      {
28          Thread.Sleep(500);
29          Console.WriteLine("Pegando o número...");
30          return 1000;
31      });
32
33  async Task<long> PegarSoma()
```

```
34  {
35      var x = await PegarNumero();
36      var y = 500;
37      return x + y;
38  }
39
40  async Task MostrarSoma()
41  {
42      var x = await PegarNumero();
43      var y = 500;
44      Console.WriteLine(x + y);
45  }
46
47  async void MostrarSomaVoid()
48  {
49      var x = await PegarNumero();
50      var y = 500;
51      Console.WriteLine(x + y);
52  }
```

Felizmente, você verá que Tasks é muito mais fácil de usar do dia-a-dia do que entender.