SINCRONIZAÇÃO DE THREADS COM C#

Como foi dito, threads são partes de um mesmo programa que são executadas separadamente pelo(s) core(s) da CPU. Elas podem ou não interferirem no trabalho de outra. Quando há interferência, por exemplo, devido a variáveis em comum envolvidas, deve haver uma maneira de evitar incoerências que **surgem** caso nenhum método de controle seja utilizado.

Para tanto, o próximo passo é aprender sobre *sincronização*, que é o modo de coordenar as ações das threads para que seus resultados sejam os esperados.

Em C# existem, basicamente, 4 categorias:

Métodos de bloqueio simples

Fazem com que uma thread espere pelo término de outra que estar a executar ou que um tempo limite seja estourado. `Sleep`, `Join` e `Task.Wait` são dessa categoria.

Construtores de locking

Limitam o número de threads que podem realizar alguma atividade ou executar uma parte do código por vez.

Os de tipo exclusivo são mais comuns, eles permitem apenas **uma** thread por vez e que threads concorrentes acessem o mesmo dado sem interferência entre elas. `Mutex`, `lock` e `SpinLock` são locking do tipo exclusivo.

Outro tipo bastante conhecido, os não exclusivos, permitem mais de uma thread por vez. `Samaphore` é o melhor exemplo.

Construtores de sinalização

Eles permitem que uma thread fique pausada até receber uma notificação de outra, fato que evita polling ineficiente. `Wait`/`Pulse` são exemplos.

Apresentadas as categorias, façamos um estudo mais profundo.

LOCKING

Locking exclusivo é utilizado para garantir que apenas uma thread acesse uma parte do código por vez. Os dois principais construtores de locking são `lock` e `Mutex`. O primeiro é mais eficiente, porém este último possui seu nicho de aplicação.

```
class ThreadUnsafe
{
  static int a = 1, b = 1;

  static void Go()
  {
    if(b != 0){
      Console.WriteLine(a/b);
      b = 0;
    }
  }
}
```

```
class ThreadSafe
{
  static int a = 1, b = 1;
  static readonly object locker;
  locker = new object();

  static void Go()
  {
   lock (locker){
     if(b != 0){
        Console.WriteLine(a/b);
        b = 0;
     }
  }
```

Na classe da esquerda, caso o método `Go()` fosse chamado por duas threads, seria possível haver uma divisão por 0, pois uma thread poderia setar b em 0 bem no momento que a outra tenta fazer a divisão.

Na classe da direita, o problema foi resolvido pois apenas uma única thread pode segurar o objeto de sincronização (nesse caso, `locker`) por vez, fato que faz com que qualquer thread concorrente seja bloqueada até que o objeto seja liberado.

Se há disputa pelo objeto de sincronização por duas ou mais threads, elas são enfileiradas numa fila de pronto do tipo primeiro a chegar, primeiro servido.

Então, quando usar locking? Como regra básica, você precisa usá-lo ao redor de qualquer espaço de memória escrevível mesmo nos casos mais simples, como os das duas variáveis acima.

Apesar de muito eficiente, locking pode gerar um dos problemas de resolução mais árdua dentre os problemas de multithreading: deadlocks.

Um deadlock acontece entre duas threads quando ambas esperam por um recurso mantida pela outra, assim nenhuma consegue proceder. É o que acontece no exemplo abaixo:

```
object locker1 = new object();
object locker2 = new object();

new Thread (() => {
                    lock (locker1) {
                         Thread.Sleep (1000);
                         lock (locker2); // Deadlock
                    }
                  }).Start();
lock (locker2)
{
     Thread.Sleep (1000);
     lock (locker1);                    // Deadlock
}
```

SEMAPHORE

A maneira como um semáforo trabalha é análoga a uma boate: ela tem uma certa capacidade e possui seguranças na entrada. Quando ela está cheia, ninguém pode entrar e uma fila se forma na porta. Então, para cada pessoa que sai, uma pessoa do início da fila entra. Assim, para trabalhar com sincronização, os semáforos precisam de dois parâmetros: o número de lugares disponíveis e a capacidade total suportada.

Um semáforo com capacidade igual a 1 é similar a um `lock`, exceto pelo fato de que semáforos não têm donos. Qualquer thread pode chamá-lo, enquanto um `lock` pode ser obtido por apenas uma thread por vez.

Semáforos são úteis quando se deseja limitar uma quantidade de threads concorrentes, evitando que muitas delas executem por vez.

O exemplo a seguir ilustra 5 threads tentando entrar numa boate de capacidade igual a 3.

```
class TheClub
{
     static SemaphoreSlim sem;
     sem = new SemaphoreSlim (3); // Capacity of 3

     static void Main()
     {
          for (int i = 1; i <= 5; i++)
               new Thread (Enter).Start (i);
     }

     static void Enter (object id)
     {
          Console.WriteLine (id + " wants to enter");
          sem.Wait(); // Chamada do semáforo
```

```
                Console.WriteLine (id + " is in!");
                Thread.Sleep (1000 * (int) id); // Tempo "curtindo"

                Console.WriteLine (id + " is leaving");
                sem.Release(); // Libera o semáforo
            }
        }

Saída:
        1 wants to enter
        1 is in!
        2 wants to enter
        2 is in!
        3 wants to enter
        3 is in!
        4 wants to enter
        5 wants to enter
        1 is leaving
        4 is in!
        2 is leaving
        5 is in!
```