内存栅栏和volatile关键字

前言

本次主要讲解关于内存栅栏的一点小东西,主要是扫盲,给大家普及普及概念性的东西.以前我们说过在一些简单的案例中,比如一个字段赋值或递增该字段,我们需要对线程进行同步.

虽然lock可以满足我们的需要,但是一个竞争锁一定会导致阻塞,然后忍受线程上下文切换和调度的开销.有些高并发和性能比较关键的地方,这些是不能忍受的.

.net提供了非阻塞同步构造,为一些简单的操作提高了性能,它甚至都没有阻塞,暂停,和等待线程.

引入

Memory Barries and Volatility(内存栅栏和易失字段)

考虑下面的代码

```
1.  int _answer;
2.          bool _complete;
3.          void A()
4.          {
5.              _answer = 123;
6.              _complete = true;
7.          }
8.
9.          void B()
10.         {
11.             if (_complete)
12.             {
13.                 Console.WriteLine(_answer);
14.             }
15.         }
```

如果方法A和B都在不同的线程下并发的执行,方法B可能输出”0”吗?回答是”yes”,原因如下:

1.编译器,CLR或CPU可能会为了性能而重新为程序的指令进行排序,例如可能会将方法A中的两句代码的顺序进行调整.

2.编译器,CLR或CPU可能会为变量 的赋值采用缓存策略,这样这些变量就不会立即对其他变量可见了,例如方法A中的变量赋值,不会立即刷新到内存中,变量B看到的变量并不是最新的值.

C#在运行时非常小心的保证这些优化策略不会影响正常的单线程的代码和多线程环境下加锁的代码.

除此之外,你必须显示的通过创建内存屏障(Memory fences)来限制指令重新排序和读写缓存对程序造成影响.

Full Fences:

最简单的完全栅栏的方法莫过于使用Thread.Memory.Barrier方法了.这个方法就是写完数据之后,调用MemoryBarrier,数据就会立即刷新,另外在读取数据之前调用MemoryBarrier可以确保读取的数据是最新的,并且处理器对Memorybarrier的优化小处理.so,以上代码可以下称如下这样:

```
1.  int _answer;
2.          bool _complete;
3.          void A()
4.          {
5.              _answer = 123;
6.              Thread.MemoryBarrier();
7.              _complete = true;
8.              Thread.MemoryBarrier();
9.          }
10.
11.         void B()
12.         {
```

```
13.
14.          Thread.MemoryBarrier();
15.          if (_complete)
16.          {
17.
18.              Thread.MemoryBarrier();
19.              Console.WriteLine(_answer);
20.          }
21.      }
```

一个完全的栅栏在现在桌面应用程序中,大约需要花费10ns.

下面的一些构造都隐式的生成完全栅栏.

1.C# Lock 语句(Monitor.Enter / Monitor.Exit)

2.在Interlocked类的所有方法。

3.使用线程池的异步回调，包括异步的委托，APM 回调，
和 Task continuations.

4.在一个信号构造中的发送(Settings)和等待(waiting)

你不需要对每一个变量的读写都使用完全栅栏，假设你有三个answer 字段，我们仍然可以使用4个栅栏。例如:

```
1.  int _answer1, _answer2, _answer3;
2.          bool _complete;
3.
4.          void A()
5.          {
6.              _answer1 = 1; _answer2 = 2; _answer3 = 3;
7.              Thread.MemoryBarrier();
8.              _complete = true;
9.              Thread.MemoryBarrier();
```

```
10.         }
11.
12.         void B()
13.         {
14.             Thread.MemoryBarrier();
15.             if (_complete)
16.             {
17.                 Thread.MemoryBarrier();
18.                 Console.WriteLine(_answer1 + _answer2 + _answer3);
19.             }
20.         }
21.
22.
```

我们真的需要lock和内存栅栏吗?

在一个共享可写的字段上不使用lock或者栅栏就是在自找麻烦,看一下面的代码:

```
1.  class Program
2.      {
3.          static void Main(string[] args)
4.          {
5.              bool complete = false;
6.              var t = new Thread(() =>
7.              {
8.                  bool toggle = false;
9.                  while (!complete)
10.                 {
11.                     toggle = !toggle;
12.                 }
13.             });
14.
```

```
15.             t.Start();
16.             Thread.Sleep(1000);
17.             complete = true;
18.             t.Join();
19.         }
20.     }
21.
```

如果你在Visual Studio中选择发布(Release)模式，生成该应用程序，那么如果你直接运行应用程序(不使用VS调试器，直接双击运行exe文件.)，程序都不会中止.

因为CPU寄存器把complete变量的值给缓存了.在寄存器中,complete永远都是false.

通过在while循环中插入Thread.MemoryBarrier,或者是在读取complete的时候加锁都可以解决这个问题.

volatile关键字

为_complete字段加上volatile关键字也可以解决这个问题.

volatile bool _complete;

volatile关键字指导道编译器自动的为读写字段加屏障,以下是MSDN的解析:

volatile关键字指示一个字段可以由多个同时执行的线程修改.声明为volatile的字段不受编译器优化(假定由单个线程访问)的限制.这样可以确保该字段在任何时间呈现的都是最新的值.

使用volatile字段可以被总结成下表:

第一条指令	第二条指令	可以被交换吗?
Read	Read	No
Read	Write	No

Write	Write	No(CLR会确保写和写的操作不被交换,甚至不使用volatile关键字)
Write	**Read**	**Yes!**

注意到应用volatile关键字,并不能保证写后面跟读的操作不被交换.这就可能会造成莫名其妙的问题.例如:

```
1.  volatile int x, y;
2.      void Test1()
3.      {
4.          x = 1;
5.          int a = y;
6.      }
7.      void Test2()
8.      {
9.          y = 1;
10.         int b = x;
11.     }
```

这是Test1和Test2在不同的线程中并发执行,有可能a和b字段的值都是0(尽管在x和y上应用了volatile关键字).这段代码的意思是说,即使使用了volatile,也无法保证操作的顺序不被交换.volatile关键字可以确保线程读取到最新的值,但保证不了操作顺序.

这是一个避免使用volatile关键字的好例子,甚至假设你彻底明白了这段代码,是不是其他在你的代码上工作的人也全部明白呢?

在Test1和Test2方法中使用完全栅栏或者是lock都可以解决这个问题.

还有一个不适用volatile关键字的原因是性能问题,因为每次读写都会创建内存栅栏,例如:

```
1.  volatile m_amount;
2.  m_amount+=m_amount;
```

volatile关键字不支持引用传递的参数,和局部变量.再这样的情况下,你必须使用VolatileRead和VolatileWrite方法.例如:

```
1. volatile int m_amount;
2. Boolean success=Int32.TryParse(“123”,out m_amount);
3.
```

VolatileRead和VolatileWrite

从技术上来说,Thread类的静态方法VolatileRead和VolatileWrite在读取一个变量上和volatile关键字作用一致.

它们的实现是一样低效的,尽管事实上它们都创建了内存栅栏.下面是它们在Integer类型上的实现:

```
1. public static void VolatileWrite(ref int address, int value)
2.         {
3.             Thread.MemoryBarrier(); address = value;
4.         }
5.
6.         public static int VolatileRead(ref int address)
7.         {
8.             int num = address; Thread.MemoryBarrier(); return num;
9.         }
```

你可以看到如果你在调用VolatileWrite之后调用VolatileRead,在中间没有栅栏会被创建,着同样会导致我们上面讲到写之后再读顺序可能变换的问题.

小小的结一下

volatile修饰的变量对于读操作它具有acquire语意,而对于写操作它具有release语意.

```
1.  int _answer;
2.  volatile  bool _complete;
3.
4.   void A()
5.   {
6.       _answer = 123;
7.       _complete = true;
8.   }
9.
10.  void B()
11.  {
12.      if (_complete)
13.          Console.WriteLine(_answer);
14.  }
```

这样可以确保CW方法输出123了.