А.В. Макаров, С.Ю. Скоробогатов, А.М. Чеповский

Учебный курс “CIL и системное программирование в Microsoft .NET”

Лекция 23.

Параллельные операции в .NET

Введение

Реализация параллельного выполнения кода в .NET основана на базовых механизмах, предоставляемых ядром операционной системы Windows:

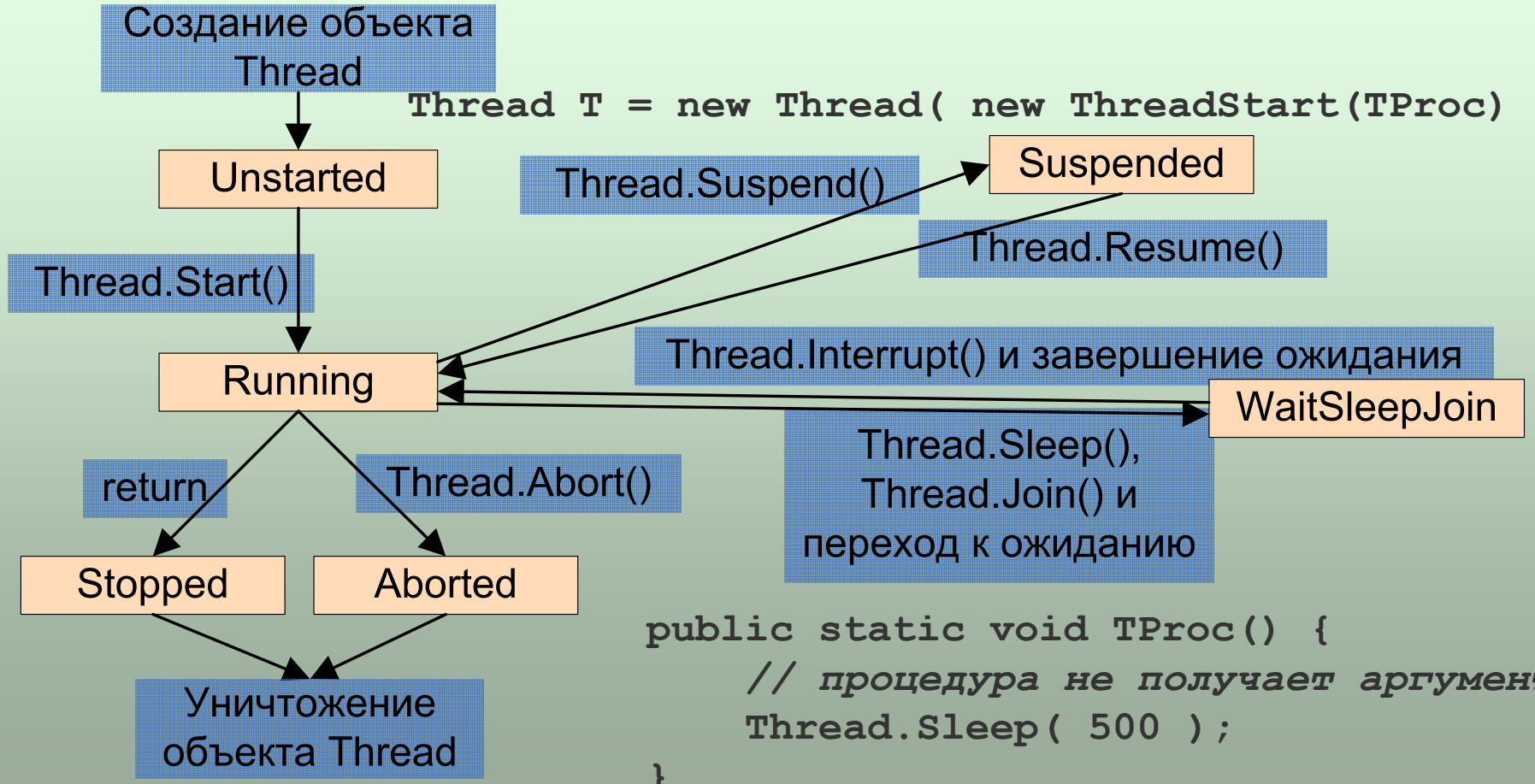
- **Обработка асинхронных запросов**. Сюда относятся средства для выполнения асинхронных операций ввода-вывода, средства для работы с очередями сообщений, некоторые надстройки для работы в сетевой среде (ASP, XML и др.) и средства поддержания инфраструктуры COM объектов.
- **Организация многопоточных приложений**. Сюда относятся средства создания потоков, управления ими, включая пулы потоков, средства взаимной синхронизации, а также организация локальной для потоков памяти. По большей части .NET Framework предоставляет надстройку над средствами операционной системы.
- **Работа с различными процессами и доменами приложений**. В основном являются надстройками над абстракциями высокого уровня, такими как RPC, COM объекты и пр.

Введение


CLI
Расширенные библиотеки
Базовые библиотеки
Механизмы исполнения
Platform Adaptation Layer (PAL)
ОС Windows

23.1. Потоки и пул потоков

Пространство имен: **System.Threading**
Класс: sealed **Thread**




```
public static void TProc() {
    // процедура не получает аргументов
    Thread.Sleep( 500 );
}
```

23.1. Потоки и пул потоков
Пример реализации (начало)

```
using System;
using System.Threading;
namespace TestNamespace {
   class TestApp {
      const int m_size = 600, m_stripsize = 50, m_stipmax = 12;
      private static int m_stripused = 0;
      private static double[,] m_A = new double[m_size,m_size],
                               m_B = new double[m_size,m_size],
                               m_C = new double[m_size,m_size];
      public static void ThreadProc()
      {
         int   i,j,k, from, to;
         from = ( m_stripused++ ) * m_stripsize;
         to = from + m_stripsize;
         if ( to > m_size ) to = m_size;
         for ( i = 0; i < m_size; i++ ) {
            for ( j = 0; j < m_size; j++ ) {
               for ( k = from; k < to; k++ )
                  m_C[i,j] += m_A[i,k] * m_B[k,j];
            }
         }
      }
```

Определяем квадратные матрицы m_A, m_B и m_C

Процедура потока, умножает вертикальную полосу из m_A на горизонтальную из m_B и получает частичный результат для m_C

23.1. Потоки и пул потоков
Пример реализации (продолжение)

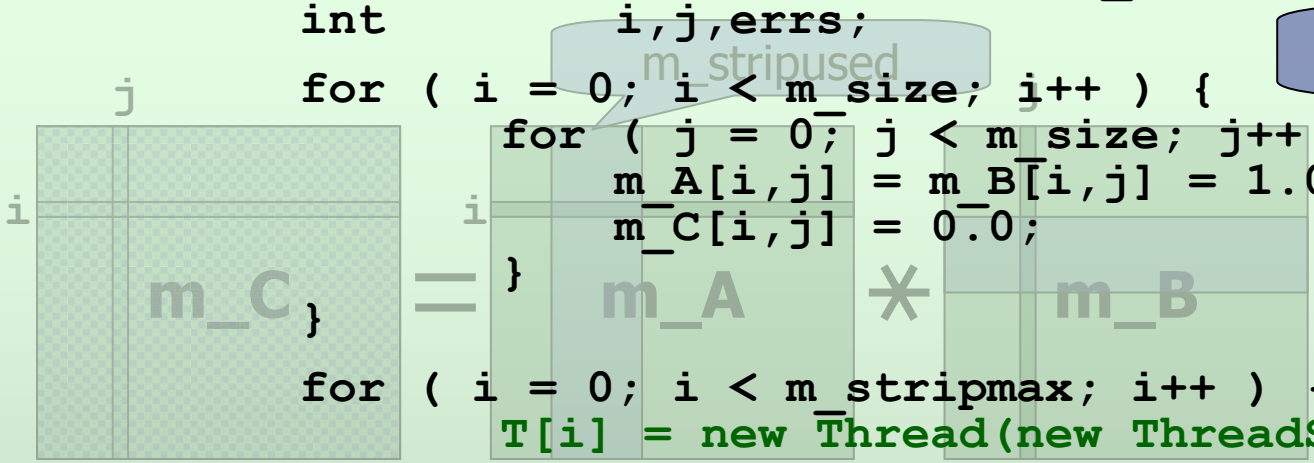
```
public static void Main() {
   Thread[]   T = new Thread[ m_stripmax ];
   int        i,j,errs;

   for ( i = 0; i < m_size; i++ ) {
      for ( j = 0; j < m_size; j++ ) {
         m_A[i,j] = m_B[i,j] = 1.0;
         m_C[i,j] = 0.0;
      }
   }

   for ( i = 0; i < m_stripmax; i++ ) {
      T[i] = new Thread(new ThreadStart(ThreadProc));
      T[i].Start();
   }

   for ( i = 0; i < m_stripmax; i++ ) T[i].Join();

   errs = 0;
   for ( i = 0; i < m_size; i++ )
      for ( j = 0; j < m_size; j++ )
         if ( m_C[i,j] != m_size ) errs++;

   Console.WriteLine("Error count = {0}", errs );
  }
 }
}
```

Начальные значения

Запуск потоков

Ожидание

Проверка результата



23.2. Асинхронный ввод-вывод

Пространство имен: **System.IO**

Класс: **Stream, FileStream, MemoryStream, BufferedStream, …**

Асинхронный ввод вывод в .NET Framework основан на выполнении синхронных операций ввода-вывода отдельными фоновыми потоками.

Пул потоков, предназначенный для обработки асинхронных операций создается CLR при запуске приложения.

```
public abstract class Stream : MarshalByRefObject, IDisposable {
    public abstract int  Read( byte[] buf, int offs, int cnt );
    public abstract void Write( byte[] buf, int offs, int cnt );
    public abstract long Seek( long offset, SeekOrigin origin );
    public abstract void Flush();

    public virtual int   ReadByte();
    public virtual void  WriteByte( byte value );

    public virtual IAsyncResult BeginRead(
      byte[] buf, int offs, int cnt, AsyncCallback clbck, object state
    );
    public virtual int EndRead( IAsyncResult asyncResult );

    public virtual IAsyncResult BeginWrite(
      byte[] buf, int offs, int cnt, AsyncCallback clbck, object state
    );
    public virtual void EndWrite( IAsyncResult asyncResult );
    ...
```

23.2. Асинхронный ввод-вывод

```
using System;   using System.IO;
namespace TestNamespace {
   class TestApp {
      private const int     m_size = 100000000;
      private static byte[] m_data = new byte [m_size];
      public static void Done( IAsyncResult state ) {
          Console.WriteLine( "Async Write ended" );
      }
      static void Main(string[] args) {
         IAsyncResult  state;
         Stream  file = new FileStream(
            "test.dat", FileMode.OpenOrCreate,
            FileAccess.ReadWrite, FileShare.Read, 1, true
         );
         state = file.BeginWrite(
            m_data, 0, m_size, new AsyncCallback(Done), null
         );
         Console.WriteLine( "Async Write started" );
         file.EndWrite( state );
         file.Close();
      }
   }
}
```

23.3. Асинхронные процедуры

Пространство имен: **System.Threading**
Класс: **ThreadPool, WaitHandle**

Пространство имен: **System**
Класс: **Delegate**

.NET предоставляет три основных механизма асинхронного вызова процедур:

- Явное размещение вызовов в очереди
  `ThreadPool.QueueUserWorkItem`
- Размещение вызова в очереди, когда объект переходит в свободное состояние
  `ThreadPool.RegisterWaitForSingleObject`
- С помощью поддерживаемых компилятором методов **`BeginInvoke`** и **`EndInvoke`** класса **`Delegate`**.

Асинхронный вызов выполняется потоком пула, а не потоком, поставившим вызов в очередь.

23.3. Асинхронные процедуры
Явное размещение вызова в очереди

```
using System;  using System.Threading;

namespace TestNamespace {
   class GreetingData {
      private string     m_greeting;

      public GreetingData( string text ) {m_greeting = text;}
      public void Invoke() { Console.WriteLine(m_greeting); }
   }

   class TestApp {

      static void AsyncProc( Object arg ) {
         GreetingData    gd = (GreetingData)arg;
         gd.Invoke();
      }

      public static void Main() {
         GreetingData gd = new GreetingData( "Hello, world!" );

         ThreadPool.QueueUserWorkItem(
            new WaitCallback(AsyncProc), gd
         );

         Thread.Sleep( 1000 );
      }
   }
}
```

23.3. Асинхронные процедуры
Асинхронный вызов при освобождении объекта

```
using System;       using System.Threading;

namespace TestNamespace {
   class TestApp {

      private RegisteredWaitHandle      m_whandle;

      static void AsyncProc( Object arg, bool isTimeout ) {
         TestApp    ta = (TestApp)arg;

         if ( !isTimeout ) ta.m_whandle.Unregister( null );
      }

      public static void Main() {
         TestApp              ta = new TestApp();
         AutoResetEvent       ev = new AutoResetEvent( false );

         ta.m_whandle = ThreadPool.RegisterWaitForSingleObject(
            ev,    new WaitOrTimerCallback(AsyncProc),
            ta,    1000,    false
         );

         Thread.Sleep( 2500 );
         ev.Set();

         Console.ReadLine();
      }
   }
}
```

23.3. Асинхронные процедуры
С использованием делегатов

```
using System;   using System.Threading;
namespace TestNamespace {
   public class GreetingData {
      private string      m_greeting;
      public GreetingData( string text ) { m_greeting = text; }
      public static void Invoke( GreetingData arg ) {
         Console.WriteLine( arg.m_greeting );
      }
   }
   public delegate void AsyncProcCallback( GreetingData gd );
   class TestApp {
      public static void Main() {
         GreetingData  gd = new GreetingData( "Hello!!!" );
         AsyncProcCallback  apd = new AsyncProcCallback(
            GreetingData.Invoke
         );
         IAsyncResult  ar = apd.BeginInvoke( gd, null, null );
         ar.AsyncWaitHandle.WaitOne(); // apd.EndInvoke();
      }
   }
}
```

Можно указать объект, который будет освобожден при завершении асинхронной операции

23.4. Синхронизация и изоляция потоков

Потоки и основные средства взаимодействия потоков .NET основаны на базовых механизмах операционной системы. Многие из них нашли свое отражение в аналогичных средствах .NET Framework.

К этим средствам относятся:

Атомарные операции.

Синхронизирующие примитивы.

Локальная для потока память.

Таймеры.

Применение этих средств в основе своей похоже на применение аналогичных средств Win32 API.

23.4.1. Атомарные операции

Пространство имен: **System.Threading**

Класс: **Interlocked**

```
public static void ThreadProc()
{
    int         i,j,k, from, to;
    from = ( Interlocked.Increment(ref m_stripused)-1 )
           * m_stripsize;
    to = from + m_stripsize;
    …
```

```
public sealed class Interlocked {
   public static int Increment(ref int);
   public static long Increment(ref long);
   public static int Decrement(ref int);
   public static long Decrement(ref long);
   public static int Exchange(ref int, int);
   public static object Exchange(ref object, object);
   public static float Exchange(ref float, float);
   public static int CompareExchange(ref int, int, int);
   public static object CompareExchange(ref obj, obj, obj);
   public static float CompareExchange(ref float, float, float);
}
```

23.4.2. Синхронизация потоков

Основные средства взаимной синхронизации потоков в .NET обладают заметным сходством со средствами операционной системы. Среди них можно выделить:

Мониторы,
близкие аналоги критических секций Win32 API, используются в качестве базового синхронизирующего объекта в .NET;

События и мьютексы,
основанные на соответствующих аналогичных объектах ядра; .NET предоставляет базовый класс **WaitHandle**, на основе которого строятся остальные классы, использующие для синхронизации объекты ядра операционной системы;

Объект **ReaderWriterLock** закрывает очень типичный класс задач синхронизации – для многих объектов является типичным конкурентный доступ для чтения и исключительный для изменения данных. Аналога в Win32 API этому объекту нет.

23.4.2.1. Синхронизация потоков
Мониторы

Пространство имен: **System.Threading**
Класс: **Monitor**

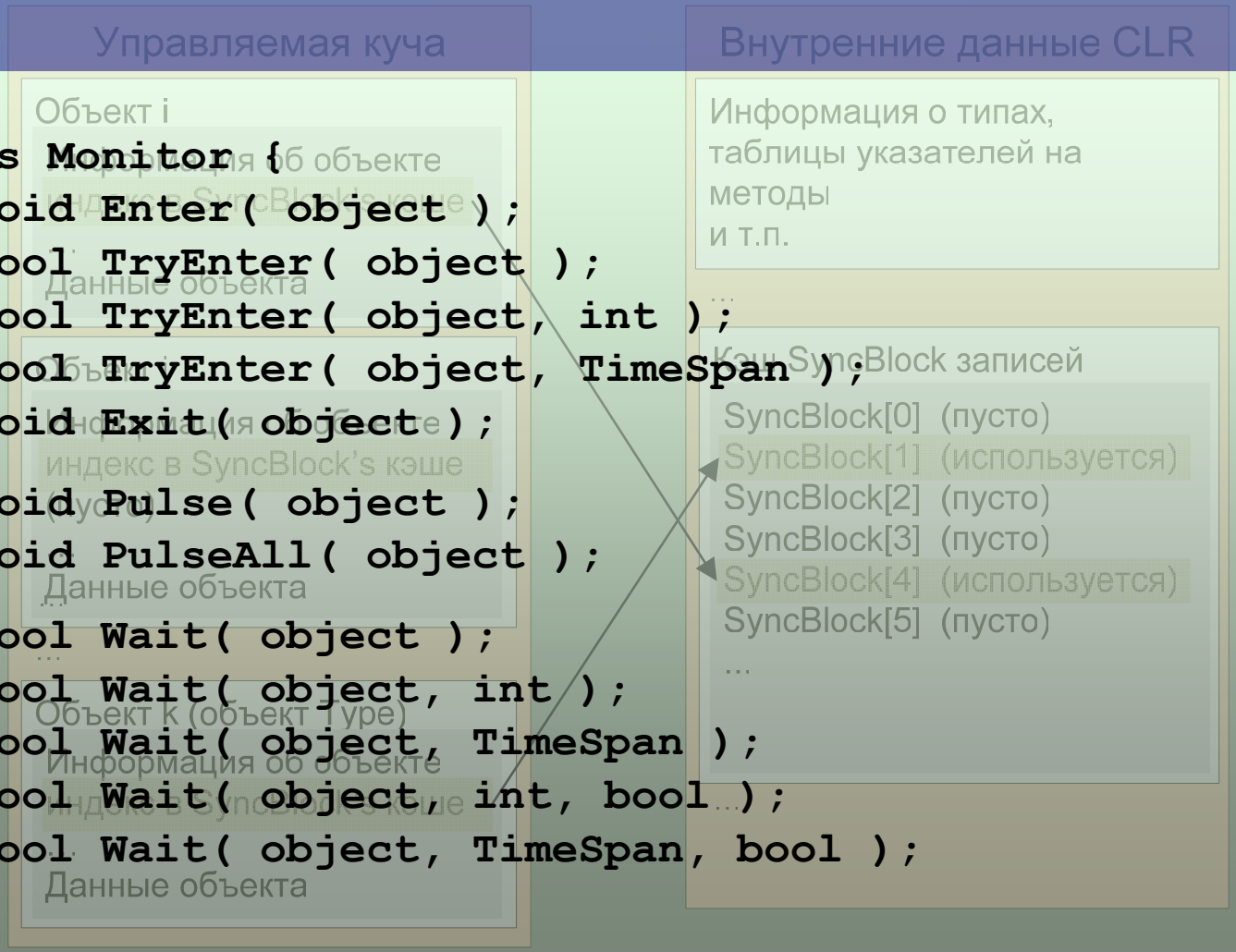
Использование мониторов настолько эффективно, что в .NET был предусмотрен механизм, позволяющий использовать практически любой объект, хранящийся в управляемой куче, для синхронизации доступа. С каждым объектом, производным от **`Object`**, сопоставляется запись **`SyncBlock`**.

23.4.2.1. Синхронизация потоков Мониторы

Пространство имен: **System.Threading**

Класс: **Monitor**

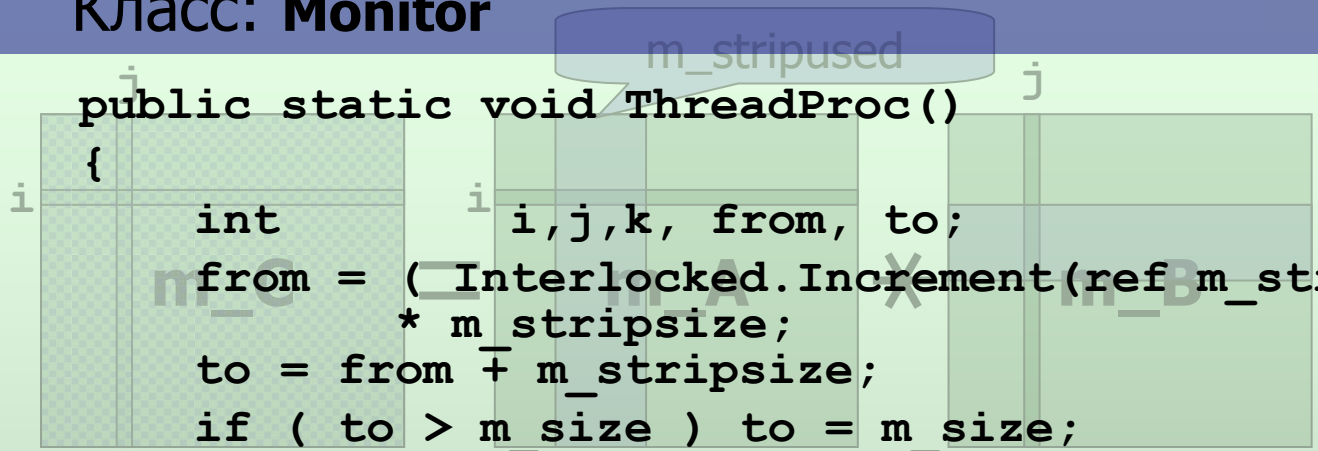
```
public sealed class Monitor {
   public static void Enter( object );
   public static bool TryEnter( object );
   public static bool TryEnter( object, int );
   public static bool TryEnter( object, TimeSpan );
   public static void Exit( object );

   public static void Pulse( object );
   public static void PulseAll( object );

   public static bool Wait( object );
   public static bool Wait( object, int );
   public static bool Wait( object, TimeSpan );
   public static bool Wait( object, int, bool );
   public static bool Wait( object, TimeSpan, bool );
}
```



23.4.2.1. Синхронизация потоков Мониторы

Пространство имен: **System.Threading**

Класс: **Monitor**




```
public static void ThreadProc()
{
    int         i,j,k, from, to;
    from = ( Interlocked.Increment(ref m_stripused)-1 )
           * m_stripsize;
    to = from + m_stripsize;
    if ( to > m_size ) to = m_size;
    for ( j = 0; j < m_size; j++ ) {
        for ( k = from; k < to; k++ ) {
            Monitor.Enter( m_C );
            try {
                m_C[i,j] += m_A[i,k] * m_B[k,j];
            } finally {
                Monitor.Exit( m_C );
            }
        }
    }
}
```

23.4.2.1. Синхронизация потоков Мониторы

Пространство имен: **System.Threading**
Класс: **Monitor**

```
Monitor.Enter(m_C); try { ... } finally { Monitor.Exit(m_C); }
lock ( m_C ) { m_C[i,j] += m_A[i,k] * m_B[k,j]; }
```

C# предоставляет синтаксическую поддержку мониторов

```
Monitor.Enter( m_C[i,j] );
try {
    m_C[i,j] += m_A[i,k] * m_B[k,j];
} finally {
    Monitor.Exit( m_C[i,j] );
}

Monitor.Enter( typeof(double) );
try {
    m_C[i,j] += m_A[i,k] * m_B[k,j];
} finally {
    Monitor.Exit( typeof(double) );
}
```

Использование типов-значений ошибочно; Метод Enter требует ссылочный тип, так что для него будет создано упакованное **временное** представление, а для метода Exit – еще одно представление, на которое блокировки не накладывалось.

23.4.2.2. Синхронизация потоков Ожидающие объекты

Пространство имен: **System.Threading**

Класс: **WaitHandle**

Класс **`WaitHandle`** является базовым для описания объектов, находящихся в одном из двух состояния – занятом или свободном. **`Wait`**… методы этого класса являются оберткой функций **`WaitFor...`** операционной системы.

```
public abstract class WaitHandle:MarshalByRefObject,IDisposable {
    public virtual IntPtr Handle {get; set;}
    public virtual void Close();
    public virtual bool WaitOne();
    public virtual bool WaitOne(int, bool);
    public virtual bool WaitOne(TimeSpan, bool);
    public static int   WaitAny(WaitHandle[]);
    public static int   WaitAny(WaitHandle[], int, bool);
    public static int   WaitAny(WaitHandle[], TimeSpan, bool);
    public static bool  WaitAll(WaitHandle[]);
    public static bool  WaitAll(WaitHandle[], int, bool);
    public static bool  WaitAll(WaitHandle[], TimeSpan, bool);
    ...
```

23.4.2.2. Синхронизация потоков Ожидающие объекты (пример, начало)

Пространство имен: **System.Threading**

Класс: **WaitHandle, ManualResetEvent, AutoResetEvent, Mutex**

Основные типы синхронизирующих объектов .NET и их применение соответствуют стандартным объектам ядра ОС Windows «событие» и «мьютекс».

```
using System;   using System.Threading;
namespace TestNamespace {
   public class SomeData {
      public const int                  m_queries = 10;
      private static int                m_counter = 0;
      private static Mutex              m_mutex = new Mutex();
      private static ManualResetEvent m_event =
                                    new ManualResetEvent( false );
      public static void Invoke( int no ) {
         m_mutex.WaitOne();
         m_counter++;
         if ( m_counter >= m_queries ) m_event.Set();
         m_mutex.ReleaseMutex();
         m_event.WaitOne();
      }
   }
```

23.4.2.2. Синхронизация потоков Ожидающие объекты (пример, окончание)

Пространство имен: **System.Threading**

Класс: **WaitHandle, ManualResetEvent, AutoResetEvent, Mutex**

В примере используются объекты разных типов (события с ручным сбросом, мьютекс и WaitHandle), методы синхронизации с одиночными объектами и с группой объектов.

```
public delegate void AsyncProcCallback( int no );
class TestApp {
   public static void Main() {
      int                 i;
      WaitHandle[]        wh;
      AsyncProcCallback   apd;
      wh = new WaitHandle[ SomeData.m_queries ];
      apd = new AsyncProcCallback( SomeData.Invoke );
      for ( i = 0; i < SomeData.m_queries; i++ ) {
         wh[i]=apd.BeginInvoke(i,null,null).AsyncWaitHandle;
      }
      WaitHandle.WaitAll( wh );
   }
}
}
```

23.4.2.3. Синхронизация потоков
Один «писатель», много «читателей»

Пространство имен: **System.Threading**
Класс: **ReaderWriterLock**

Класс **`ReaderWriterLock`** позволяет организовать доступ к объекту, для которого разрешается множественный конкурентный доступ для чтения и требуется исключительный доступ для записи. Прямого аналога среди объектов ядра для этого типа нет, он является сложным объектом.

```
public sealed class ReaderWriterLock {
   public void AcquireReaderLock(int);
   public void AcquireReaderLock(TimeSpan);
   public void ReleaseReaderLock();

   public void AcquireWriterLock(int);
   public void AcquireWriterLock(TimeSpan);
   public void ReleaseWriterLock();

   public LockCookie UpgradeToWriterLock(int);
   public LockCookie UpgradeToWriterLock(TimeSpan);
   public void DowngradeFromWriterLock(ref LockCookie lockCookie);

   public LockCookie ReleaseLock();
   public void RestoreLock( ref LockCookie lockCookie );

   ...
```

23.4.2.3. Синхронизация потоков
Один «писатель», много «читателей»

```
using System;  using System.Threading;
namespace TestNamespace {
   public class SomeData {
      public const int          m_queries = 10;
      private ReaderWriterLock m_rwlock = new ReaderWriterLock();
      private int               m_a = 0, m_b = 0;
      public int summ() {
         int    r;
         m_rwlock.AcquireReaderLock( -1 ); // infinite
         try {
            r = m_a; Thread.Sleep( 1000 ); return r + m_b;
         } finally {
            m_rwlock.ReleaseReaderLock();
         }
      }
      public int inc() {
         m_rwlock.AcquireWriterLock( -1 );
         try {
            m_a++;  Thread.Sleep( 500 );  m_b++;
            return m_a + m_b;
         } finally {
            m_rwlock.ReleaseWriterLock();
         }
      }
```

23.4.2.3. Синхронизация потоков
Один «писатель», много «читателей»

```
public static void Invoke( SomeData sd, int no ) {
      if ( no % 2 == 0 ) {
         Console.WriteLine( sd.inc() );
      } else {
         Console.WriteLine( sd.summ() );
      }
   }
}
public delegate void AsyncProcCallback( SomeData sd, int no );
class TestApp {
   public static void Main() {
     int                  i;
     SomeData              sd = new SomeData();
     WaitHandle[]          wh;
     AsyncProcCallback     apd;
     wh = new WaitHandle[ SomeData.m_queries ];
     apd = new AsyncProcCallback( SomeData.Invoke );
     for ( i = 0; i < SomeData.m_queries; i++ ) {
        wh[i]=apd.BeginInvoke(sd,i,null,null).AsyncWaitHandle;
     }
     WaitHandle.WaitAll( wh );
   }
}
}
```

23.4.3. Локальная для потока память Декларативный подход

.NET предоставляет два способа работы с локальной для потока памятью – декларативный (с использованием **`ThreadStaticAttribute`**) и императивный (с использованием методов класса **`Thread`**).

Пространство имен: **System**
Класс: **ThreadStaticAttribute**

```
class SomeData {
    [ThreadStatic]
    public static double xxx;
    …
```

Пространство имен: **System.Threading**
Класс: **Thread**

```
class SomeData {
   private static LocalDataStoreSlot m_tls=Thread.AllocateDataSlot();
    public static void ThreadProc() {
        Thread.SetData( m_tls, … );
        …
    }
    public void Main() {
        SomeData    sd = new SomeData();
        …
        // создание и запуск потоков
    }
}
```

```
AllocateDataSlot()
AllocateNamedDataSlot(string)
GetNamedDataSlot(string)
FreeNamedDataSlot(string)
GetData(slot)
SetData(slot,object)
```

23.5. Таймеры

Пространство имен: **System.Threading**
Класс: **Timer**

Таймер пространства имен **`System.Threading`** является опечатанным и предназначен для вызова указанной асинхронной процедуры с заданным интервалом времени. Обработка процедуры выполняется потоком пула.

Пространство имен: **System.Timers**
Класс: **Timer**

Таймер пространства имен **`System.Timers`** может быть использован для создания собственных классов-потомков, вместо процедуры асинхронного вызова используется обработка события, с которым может быть сопоставлено несколько обработчиков. Кроме того, этот таймер может вызывать обработку события конкретным потоком, а не произвольным потоком пула.

23.5. Таймеры
Пример использования System.Timers.Timer

```
using System;    using System.Timers;
namespace TestNamespace {
   class TestTimer : Timer {
      private int  m_minimal, m_maximal, m_counter;
      public int   count { get { return m_counter - m_minimal; } }
      public TestTimer( int mn, int mx ) {
         Elapsed += new ElapsedEventHandler( OnElapsed );
         m_minimal = m_counter = mn;
         m_maximal = mx;
         AutoReset = true;
         Interval = 400;
      }
      static void OnElapsed( object src, ElapsedEventArgs e ) {
         TestTimer    tt = (TestTimer)src;
         if ( tt.m_counter < tt.m_maximal ) tt.m_counter++;
         if ( tt.m_counter >= tt.m_maximal ) tt.Stop();
      }
      static void Main(string[] args) {
         TestTimer    tm = new TestTimer( 0, 10 );
         tm.Start();
         Thread.Sleep( 5000 );
         tm.Stop();
      }
   }
}
```

Назначение обработчика таймера

Таймер может быть периодическим или однократным

Аналогично: `tm.Enabled = true;`

`tm.Enabled = false;`