

# Синхронизация потоков

Синхронизация необходима для координации выполнения потоков. Такая *координация* необходима для согласования порядка выполнения потоков или для согласования доступа потоков к разделяемому ресурсу.

В основе синхронизации лежит понятие блокировки – один *поток* блокируется в ожидании определенного события от других потоков.

Ожидание может быть:

- активным,
- пассивным.

При *активном* ожидании поток циклически проверяет статус ожидаемого события. Фактически поток не прекращает своей работы и не освобождает процессорное время для других потоков. Активное ожидание эффективно только при незначительном времени ожидания.

*Пассивное* ожидание реализуется с помощью ОС, которая сохраняет контекст потока и выгружает его, предоставляя возможность выполняться другим потокам. При наступлении ожидаемого события ОС "будит" поток – загружает контекст потока и выделяет ему процессорное время. Пассивное ожидание требует дополнительного времени, но позволяет использовать вычислительные ресурсы во время ожидания для выполнения других задач.

*Пример.* Использовать два типа ожидания. В первом случае применить циклическую проверку статуса. Во втором случае использовать метод **Join**.

```
class Program
{
    static bool b;
    static double res;
    static void SomeWork()
    {
        for (int i=0; i<1000; i++)
            for(int j=0; j<1000; j++)
                res += Math.Pow(i, 2);
        b = true;
    }
}
```

```
static void Main()
{
    Thread thr1 = new Thread(SomeWork);
    thr1.Start();
    // Активное ожидание в цикле
    while(!b);
    Console.WriteLine("Result = " + res);
    res = 0;
    Thread thr2 = new Thread(SomeWork);
    thr2.Start();
    // Ожидание с выгрузкой контекста
    thr2.Join();
    Console.WriteLine("Result = " + res);
}
}
```

```
Result = 332833500000
Result = 332833500000
```

# Средства для взаимного исключения

Одно из основных назначений средств синхронизации заключается в организации взаимно исключительного доступа к разделяемому ресурсу. Изменения общих данных одним потоком не должны прерываться другими потоками.

Фрагмент кода, в котором осуществляется работа с разделяемым ресурсом и который должен выполняться только в одном потоке одновременно, называется *критической секцией*.

# Оператор **lock**

Предназначен для того, чтобы одному потоку не дать войти в важный раздел кода в тот момент, когда в нем находится другой поток.

```
Object thisLock = new Object();  
lock (thisLock)  
{ // Критический фрагмент кода }
```

где **thisLock** – ссылка на синхронизируемый объект, который гарантирует, что защищенный фрагмент кода будет использоваться только в потоке, получающем эту блокировку, а все остальные потоки блокируются до тех пор, пока блокировка не будет снята. Блокировка снимается по завершении защищаемого ею фрагмента кода.

*Пример.* Два потока изменяют значение общей переменной data. После внесения изменений поток выводит на экран новое значение data. Без средств синхронизации:

```
using System;
using System.Threading;
class Program
{
    static string data;
    static void DoSomeWork1()
    {
        data = "AAAA";
        Console.WriteLine("Data 1: {0}", data);
    }
    static void DoSomeWork2()
    {
        data = "BBBB";
        Console.WriteLine("Data 2: {0}", data);
    }
}
```



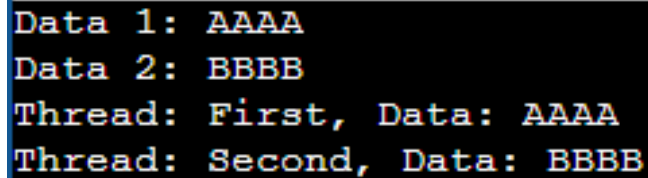
```
static void Main()
{ Thread thr1 = new Thread(DoSomeWork1);
  thr1.Name = "First";
  thr1.Start();
  Thread thr2 = new Thread(DoSomeWork2);
  thr2.Name = "Second";
  thr2.Start();
  Console.WriteLine("Thread: {0}, Data:
{1}",thr1.Name,data);
  Console.WriteLine("Thread: {0}, Data:
{1}",thr2.Name,data);
}
```

Поток *First* внес свои изменения и собирался вывести сообщение, но второй поток успел вклиниться и изменить данные.

```
Data 1: AAAA
Data 2: BBBB
Thread: First, Data: BBBB
Thread: Second, Data: BBBB
```

Выделение критической секции с помощью конструкции **lock** позволяет избежать такой ситуации:

```
static void DoSomeWork1()  
{  
    Object thisLock = new Object();  
    lock (thisLock)  
    {  
        data = "AAAA";  
        Console.WriteLine("Data 1: {0}", data);  
    }  
}
```



```
Data 1: AAAA  
Data 2: BBBB  
Thread: First, Data: AAAA  
Thread: Second, Data: BBBB
```

Когда один поток входит в критическую секцию (захватывает объект синхронизации), другой поток ожидает завершения всего блока (освобождения объекта синхронизации).

# Класс Monitor

Предназначен для того, чтобы контролировать доступ к объектам, предоставляя блокировку объекта одному потоку. В классе **Monitor** определено несколько методов синхронизации:

- **Enter()** – чтобы получить возможность блокировки для некоторого объекта,
- **Exit()** – чтобы снять блокировку.

```
public static void Enter(object syncOb)
```

```
public static void Exit(object syncOb)
```

где **syncOb** – синхронизируемый объект. Если при вызове метода **Enter()** заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока объект не станет доступным.

Использование оператора **lock** эквивалентно вызову метода **Enter()** с последующим вызовом метода **Exit()** класса **Monitor**. Класс **Monitor** позволяет добавлять значение тайм-аута для ожидания получения блокировки. Т.о., вместо того, чтобы ожидать блокировку до бесконечности, можно вызвать метод **TryEnter()** и передать в нем значение тайм-аута, указывающее, сколько максимум времени должно ожидаться получение блокировки. Один из форматов использования метода **TryEnter()**:

```
public static bool TryEnter(object syncOb)
```

Метод возвращает значение **true**, если вызывающий поток получает блокировку для объекта **syncOb**, и значение **false** в противном случае. Если заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока он не станет доступным.

Конструкция

```
lock(sync_obj)
```

```
{ // критическая секция  
}
```

аналогична применению объекта Monitor:

```
try
```

```
{ Monitor.Enter(sync_obj);  
  // критическая секция  
}
```

```
finally
```

```
{ Monitor.Exit(sync_obj);  
}
```

Кроме "обычного" входа в критическую секцию класс Monitor предоставляет "условные" входы:

```
b = Monitor.TryEnter(sync_obj);  
if(!b)  
{ // Выполняем полезную работу  
  DoWork();  
  // Снова пробуем войти в критическую секцию  
  Monitor.Enter(sync_obj);  
}  
// Критическая секция  
ChangeData();  
// Выходим  
Monitor.Exit(sync_obj);
```

Если *критическая секция* уже выполняется кем-то другим, то поток не блокируется, а выполняет полезную работу. После завершения всех полезных работ поток пытается войти в критическую секцию с блокировкой.

Если доступ к критической секции достаточно интенсивен со стороны *множества* потоков, то полезным может быть метод **TryEnter** с указанием интервала в миллисекундах, в течение которого поток пытается захватить блокировку.

```
while(! Monitor.TryEnter(sync_obj, 100))  
{ // Полезная работа  
    DoWork();  
}  
// Критическая секция  
ChangeData();  
// Выходим  
Monitor.Exit(sync_obj);
```

# Методы класса Monitor

- Wait()** – вызывается когда выполнение потока временно блокируется, т.е. он переходит в режим ожидания и снимает блокировку с объекта, позволяя другому потоку использовать этот объект.

- Pulse()** – возобновляет выполнение потока, стоящего первым в очереди потоков, пребывающих в режиме ожидания.

- PulseAll()** – сообщает о снятии блокировки всем ожидающим потокам.

Могут вызываться только из заблокированного фрагмента блока (lock-оператор).

Полезны для предотвращения взаимоблокировки в случае работы с несколькими разделяемыми ресурсами.



Форматы использования метода `Wait()`:

- ожидание до уведомления

```
public static bool Wait(object waitOb)
```

- ожидает до уведомления или до истечения периода времени, заданного в миллисекундах

```
public static bool Wait(object waitOb, int ms)
```

Форматы использования методов `Pulse()` и `PulseAll()`:

```
public static void Pulse(object waitOb)
```

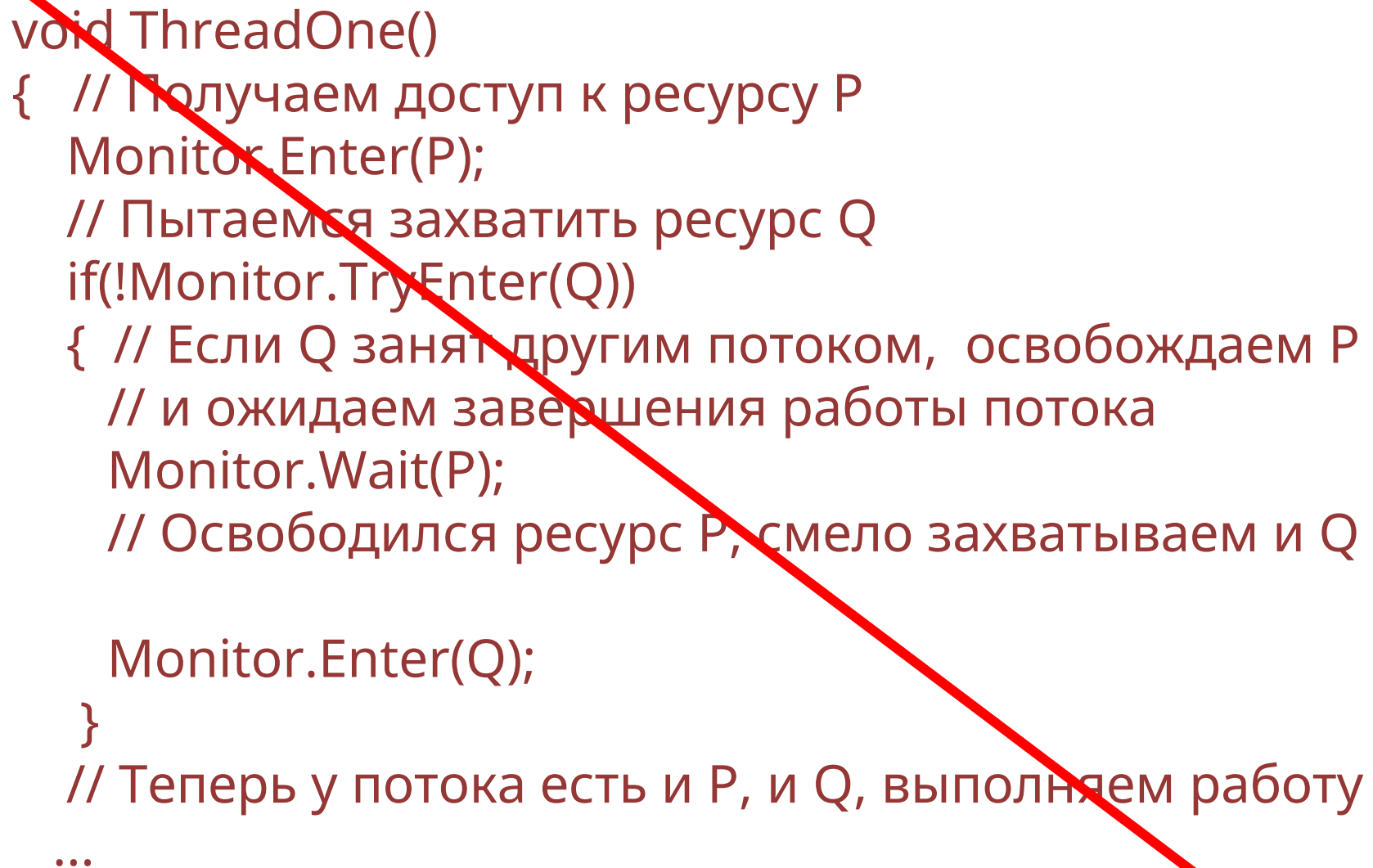
```
public static void PulseAll(object waitOb)
```

где параметр `waitOb` – означает объект, освобождаемый от блокировки.

Если метод `Wait()`, `Pulse()` или `PulseAll()` вызывается из кода, который находится вне lock блока, генерируется исключение типа `SynchronizationLockException`.

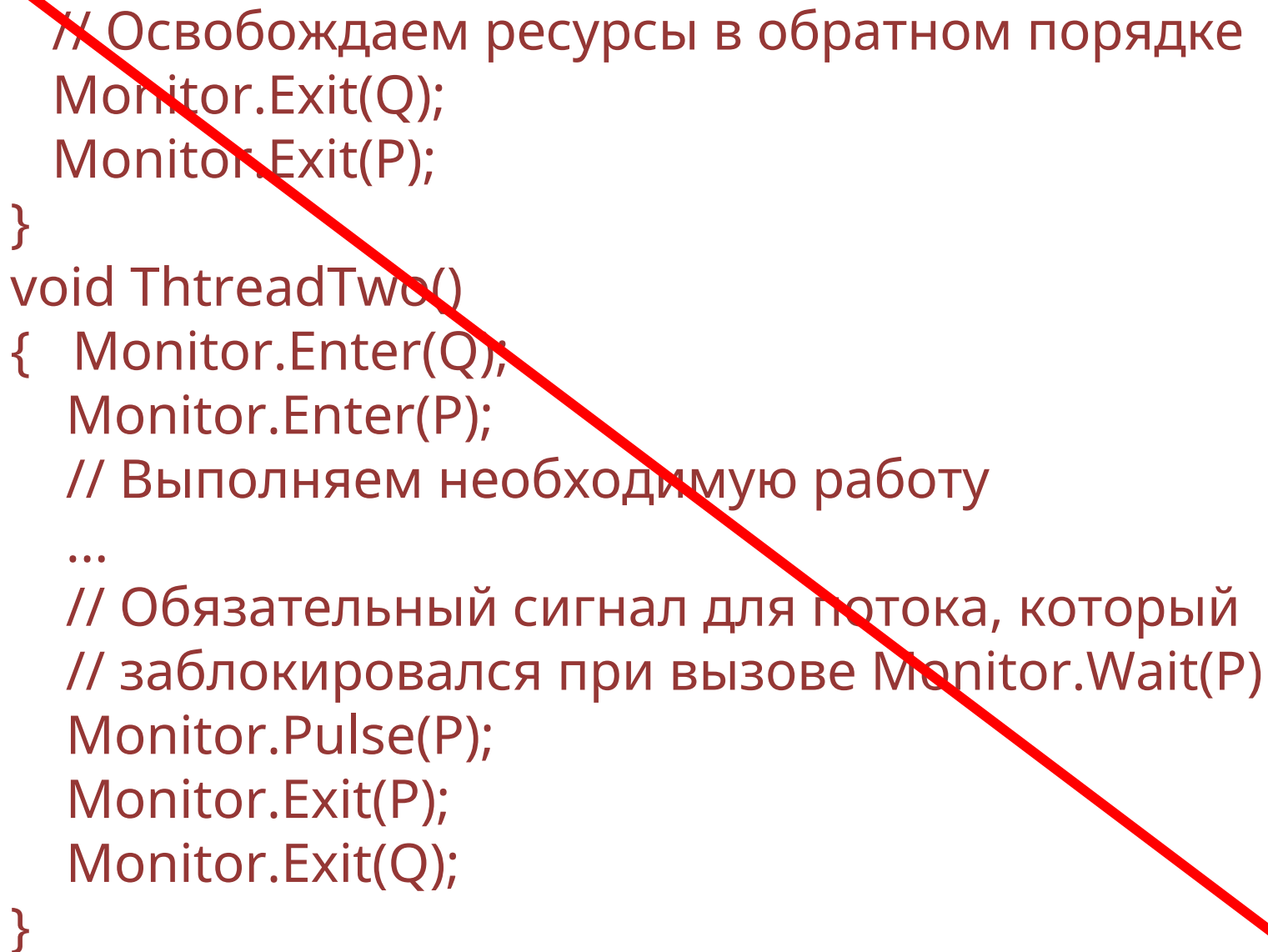
*Пример.* Два потока пытаются захватить ресурсы **P** и **Q**. Первый поток захватывает сначала **P**, затем пытается захватить **Q**. Второй поток сначала захватывает ресурс **Q**, а затем пытается захватить **P**.

Применение обычной конструкции **lock** привело бы в некоторых случаях к взаимоблокировке потоков – потоки успели захватить по одному ресурсу и пытаются получить доступ к недостающему ресурсу. Следующий фрагмент решает проблему с помощью объекта Monitor:



```
void ThreadOne()
{ // Получаем доступ к ресурсу P
  Monitor.Enter(P);
  // Пытаемся захватить ресурс Q
  if(!Monitor.TryEnter(Q))
  { // Если Q занят другим потоком, освобождаем P
    // и ожидаем завершения работы потока
    Monitor.Wait(P);
    // Освободился ресурс P, смело захватываем и Q

    Monitor.Enter(Q);
  }
  // Теперь у потока есть и P, и Q, выполняем работу
  ...
}
```



```
// Освобождаем ресурсы в обратном порядке
Monitor.Exit(Q);
Monitor.Exit(P);
}
void ThreadTwo()
{  Monitor.Enter(Q);
   Monitor.Enter(P);
   // Выполняем необходимую работу
   ...
   // Обязательный сигнал для потока, который
   // заблокировался при вызове Monitor.Wait(P)
   Monitor.Pulse(P);
   Monitor.Exit(P);
   Monitor.Exit(Q);
}
```

Первый поток после захвата ресурса **P** пытается захватить **Q**. Если **Q** уже занят, то первый поток, зная, что второму нужен еще и **P**, освобождает его и ждет завершения работы второго потока с обоими ресурсами. Вызов **Wait** блокирует первый поток и позволяет другому потоку (одному из ожидающих) войти в критическую секцию для работы с ресурсом **P**. Работа заблокированного потока может быть продолжена после того как выполняющийся поток вызовет метод **Pulse** и освободит критическую секцию. Таким образом, первый поток возобновляет работу не после вызова **Pulse**, а после вызова **Exit(P)**.

*Пример.* Создать программу, которая имитирует работу часов посредством отображения на экране слов "тик" и "так". Для этого создать класс TickTock, который содержит два метода: tick() и tock(), отображающих слова "тик" и "так".

```
using System;
using System.Threading;
class TickTock
{ public void tick(bool running)
  { lock (this)
    { if (!running)
      { // Остановка часов
        // разрешает выполнение любого потока, ожидающего
очереди
        Monitor.Pulse(this);
        return;
      }
      Console.Write("тик ");
      Monitor.Pulse(this); // Разрешает выполнение метода
tock()
      Monitor.Wait(this); // Ожидаем завершения метода tock()
    }
  }
}
```

```
public void tock(bool running)
{ lock (this)
  { if (!running)
    { // Остановка часов.
      Monitor.Pulse(this); // Уведомление ожидающих потоков
      return;
    }
    Console.WriteLine("так");
    Monitor.Pulse(this); // Разрешает выполнение метода tick()
    Monitor.Wait(this);  // Ожидаем завершения метода tick()
  }
}
```

/\*Каждый из их методов вызывается пять раз подряд с передачей логического значения true в качестве аргумента. Часы идут до тех пор, пока этим методам передается логическое значение true, и останавливаются, как только передается логическое значение false\*/



```
class MyThread
{   public Thread thrd;
    TickTock ttOb;
    // Создаем НОВЫЙ ПОТОК
    public MyThread(string name, TickTock tt)
    {   thrd = new Thread(new ThreadStart(this.run));
        ttOb = tt;
        thrd.Name = name;
        thrd.Start();
    }
```

// Начинаем выполнение нового потока

void run()

{ // Если имя потока "тик", то вызывается метод tick()

if (thrd.Name == "тик")

{ for (int i = 0; i < 5; i++) ttOb.tick(true);

ttOb.tick(false);

}

else

// иначе вызывается метод tock()

{ for (int i = 0; i < 5; i++) ttOb.tock(true);

ttOb.tock(false);

}

}

}

```
class Program
{ static void Main(string[] args)
{   TickTock tt = new TickTock(); //создается объект tt
    класса TickTock
    //запуск двух потоков на выполнение
    MyThread mt1 = new MyThread("тик", tt);
    MyThread mt2 = new MyThread("так", tt);
    mt1.thrd.Join();
    mt2.thrd.Join();
    Console.WriteLine("Часы остановлены");
}
}
```

Если в методе Run() из класса MyThread обнаруживается имя потока mt1, соответствующее ходу часов "тик", то вызывается метод tick(). А если это имя потока mt2, соответствующее ходу часов "так", то вызывается метод tock().

## Результат работы программы с синхронизацией

```
ТИК  ТАК  
ТИК  ТАК  
ТИК  ТАК  
ТИК  ТАК  
ТИК  ТАК  
Часы остановлены
```

## Без синхронизации (без Pulse, Wait)

```
ТИК  ТАК  
так  
так  
так  
так  
ТИК  ТИК  ТИК  ТИК  Часы остановлены
```

```
ТИК  ТИК  ТИК  ТИК  ТИК  ТАК  
так  
так  
так  
так  
Часы остановлены
```

# Класс Mutex

от *mutually exclusive* (взаимно исключающий) – это примитив, который предоставляет эксклюзивный доступ к общему ресурсу только одному потоку синхронизации.

Класс Mutex очень похож на класс Monitor тем, что тоже допускает наличие только одного владельца. Только один поток может получить блокировку и иметь доступ к защищаемым Mutex синхронизированным областям кода.

Наиболее употребительные конструкторы:

`public Mutex();` – создается Mutex, которым первоначально никто не владеет

`public Mutex(bool initOw);`

`public Mutex(bool initOw, string name_mutex);` –  
исходным состоянием Mutex завладевает  
вызывающий поток, если параметр `initOw`  
имеет логическое значение `true`, если `false`,  
то объектом Mutex никто не владеет

## Метод **WaitOne()**

Ожидает до тех пор, пока не будет получен Mutex, для которого он был вызван, т.о. блокирует выполнение вызывающего потока до тех пор, пока не станет доступным указанный Mutex.

Возвращает логическое значение **true**.

Форма объявления метода **WaitOne()**:

```
Mutex mutex = new Mutex(false);  
mutex.WaitOne();
```

Когда в коде не требуется использовать Mutex, он освобождается с помощью метода **ReleaseMutex()**:

```
mutex.ReleaseMutex();
```

## Пример. Работа Mutex

```
using System;
using System.Threading;
class Program
{ private static Mutex mut = new Mutex();
  //Синхронизируем данный метод
  private static void MyThreadProc()
  { mut.WaitOne();
    Console.WriteLine("{0} зашел в защищенную
зону",Thread.CurrentThread.Name);
    Thread.Sleep(100);// Имитируем работу
    Console.WriteLine("{0} покинул защищенную
зону",Thread.CurrentThread.Name);
    mut.ReleaseMutex();
  }
```



```
private static void Main(string[] args)
{ // Создаем потоки, которые будут использовать
  // защищенный ресурс
  for (int i = 0; i < 3; i++)
  { Thread myThread = new Thread(new
ThreadStart(MyThreadProc));
    myThread.Name = String.Format("Поток {0}", i +
1);
    myThread.Start();
  }
}
}
```

## Результат работы программы с синхронизацией

```
Поток 1 зашел в защищенную зону  
Поток 1 покинул защищенную зону  
Поток 3 зашел в защищенную зону  
Поток 3 покинул защищенную зону  
Поток 2 зашел в защищенную зону  
Поток 2 покинул защищенную зону
```

```
Поток 2 зашел в защищенную зону  
Поток 2 покинул защищенную зону  
Поток 1 зашел в защищенную зону  
Поток 1 покинул защищенную зону  
Поток 3 зашел в защищенную зону  
Поток 3 покинул защищенную зону
```

## Без синхронизации (без WaitOne() и ReleaseMutex() )

```
Поток 1 зашел в защищенную зону  
Поток 2 зашел в защищенную зону  
Поток 3 зашел в защищенную зону  
Поток 1 покинул защищенную зону  
Поток 2 покинул защищенную зону  
Поток 3 покинул защищенную зону
```

# Класс *Semaphore*

Класс **Semaphore** предназначен для управления доступом к пулу ресурсов. Потоки производят вход в семафор, вызывая метод **WaitOne()**, и освобождают семафор при вызове метода **Release()**.

Semaphore похож на Mutex, но он предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам.

Счетчик на семафоре уменьшается на 1 каждый раз, когда в семафор входит поток, и увеличивается на 1, когда поток освобождает семафор.

Когда счетчик равен 0, последующие запросы блокируются, пока другие потоки не освободят семафор. Когда семафор освобожден всеми потоками, счетчик имеет максимальное значение, заданное при создании семафора.

## Конструктор класса Semaphore:

`public Semaphore(int initCount, int maxCount)`

где `initCount` – первоначальное значение для счетчика разрешений семафора (количество первоначально доступных разрешений); `maxCount` – максимальное значение данного счетчика (максимальное количество разрешений, которые может дать семафор).

Семафор применяется таким же образом, как и Mutex. В целях получения доступа к ресурсу в коде программы используется метод `WaitOne()` для семафора. Этот метод ожидает до тех пор, пока не будет получен семафор, для которого он вызывается. Таким образом, он блокирует выполнение вызывающего потока до тех пор, пока указанный семафор не предоставит разрешение на доступ к ресурсу.

Если коду больше не требуется владеть семафором, он освобождает его, вызывая метод **Release()**.

Формы этого метода:

**public int Release();** – высвобождает только одно разрешение

**public int Release(int releaseCount);** – высвобождает количество разрешений, определяемых параметром **releaseCount**.

В обеих формах данный метод возвращает подсчитанное количество разрешений, существовавших до высвобождения.

## Пример. Работа Semaphore

```
using System;
using System.Threading;
class Program
{
    static Semaphore s = new Semaphore(2, 3);
    static void Fun()
    {
        s.WaitOne();
        Console.WriteLine("{0} зашел в защищенную
зону", Thread.CurrentThread.Name);
        Thread.Sleep(100);
        Console.WriteLine("{0} покинул защищенную
зону", Thread.CurrentThread.Name);
        s.Release();
    }
}
```

```

static void Main(string[] args)
{
    for (int i = 1; i < 4; i++)
    {
        Thread myThread = new Thread(new
ThreadStart(Fun));
        myThread.Name = String.Format("Поток {0}", i );
        myThread.Start();
    }
}

```

```

Поток 1 зашел в защищенную зону
Поток 3 зашел в защищенную зону
Поток 1 покинул защищенную зону
Поток 3 покинул защищенную зону
Поток 2 зашел в защищенную зону
Поток 2 покинул защищенную зону

```

Если `static Semaphore s = new Semaphore(3, 3);`

```

Поток 2 зашел в защищенную зону
Поток 1 зашел в защищенную зону
Поток 3 зашел в защищенную зону
Поток 2 покинул защищенную зону
Поток 3 покинул защищенную зону
Поток 1 покинул защищенную зону

```

Если `static Semaphore s = new Semaphore(1, 3);`

```

Поток 1 зашел в защищенную зону
Поток 1 покинул защищенную зону
Поток 3 зашел в защищенную зону
Поток 3 покинул защищенную зону
Поток 2 зашел в защищенную зону
Поток 2 покинул защищенную зону

```

# Класс Barrier

Реализует барьер потока исполнения, который позволяет множеству потоков встречаться в определенном месте во времени. Данный метод применяется для участников, нуждающихся в синхронизации, до тех пор, пока задание остается активным, динамически могут добавляться дополнительные участники, например, дочерние задачи, создаваемые из родительской задачи. Эти участники могут ожидать, пока все остальные участники не выполнят свою работу.



Для использования этого класса необходимо:

- Создать экземпляр, указав количество потоков, которые будут встречаться одновременно;
- Каждый поток, должен вызывать метод `SignalAndWait()`.

Метод `SignalAndWait()` сообщает, что участник достиг барьера (Barrier) и ожидает достижения барьера другими участниками.

Форма метода:

```
public void SignalAndWait()
```

## *Пример.* Работа Barrier

```
using System;
using System.Threading;
namespace BarrierProgram
{ class Program
    { static Barrier barrier = new Barrier(3);
      static void Speak()
      { for (int i = 0; i < 5; i++)
        { Console.Write(i + " ");
          barrier.SignalAndWait();
        }
      }
    }
```

```
static void Main(string[] args)
{
    new Thread(Speak).Start();
    new Thread(Speak).Start();
    new Thread(Speak).Start();
}
}
```

Каждый из трех потоков выводит числа от 0 до 4, одновременно с другими потоками

```
0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

Если не использовать класс **Barrier** программа выведет числа в случайном порядке

```
0 1 2 3 4 0 1 2 3 4 0 1 2 3 4
```