Синхронизация

1. Проблемы синхронизации .
2. Объекты синхронизации:
   - мьютексы;
   - семафоры;
   - события;
   - критические секции;
   - взаимоисключающий доступ.
3. Критическая секция в .NET Framework:
   - класс Monitor;
   - ключевое слово lock .
4. Мьютексы в .NET Framework. Класс Mutex .
5. Семафоры в .NET Framework. Класс Semaphore .
6. События в .NET Framework .
   - класс ManualResetEvent;
   - класс AutoResetEvent .
7. Взаимоисключающий доступ в .NET Framework . Класс Interlocked.


Theory
Actual

Основным преимуществом использования нескольких потоков в приложении является **”параллельное” выполнение каждого потока**.

- В приложениях Windows это позволяет выполнять длительные задачи в фоновом режиме, при этом окно приложения и элементы управления остаются активными.
- Для серверных приложений многопоточность обеспечивает возможность обработки каждого входящего запроса в отдельном потоке. В противном случае ни один новый запрос не будет обработан, пока не завершена обработка предыдущего запроса.

Проблема параллельности выполнения потоков

потоки работают “параллельно” следовательно доступ к ресурсам, таким как файлы, сетевые подключения и память, должен быть скоординирован. Иначе два или более потоков **могут получить доступ к одному и тому же ресурсу одновременно, причем один поток не будет учитывать действия другого**. В результате данные могут быть повреждены непредсказуемым образом.

Синхронизация – это средство, которое позволяет координировать выполнение нескольких параллельно выполняемых потоков с целью получения предсказуемых результатов.

Виды синхронизации

Монопольное блокирование

Конструкции монопольного блокирования позволяют выполнять некоторое действие только одному потоку одновременно. Их основное назначение заключается в том, чтобы предоставить потокам возможность доступа к записываемому разделяемому состоянию, не влияя друг на друга (*lock, Mutex и SpinLock*).

Немонопольное блокирование

Немонопольное блокирование позволяет ограничивать параллелизм. Конструкциями немонопольного блокирования являются *Semaphore (SemaphoreSlim) и ReaderWriterLock (ReaderWriterLockSlim)*.

Сигнализация

Сигнализация позволяет потоку блокироваться вплоть до получения одного или большего числа уведомлений от другого потока (потоков). Сигнализирующие конструкции включают *ManualResetEvent (ManualResetEventSlim), AutoResetEvent, CountdownEvent и Barrier*.

Объекты синхронизации

Мью́текс (англ. mutex, от mutual exclusion - «взаимное исключение») - элемент, служащий в программировании для синхронизации одновременно выполняющихся потоков.

Основное назначение - организация взаимного исключения для потоков из одного и того же или из разных процессов

Особенности:
Мьютекс может перевести в отмеченное состояние только владеющий им поток.

Принципы работы:
Мьютекс - защищает объект от доступа к нему других потоков, отличных от того, который завладел мьютексом. В каждый конкретный момент только один поток может владеть объектом, защищённым мьютексом. Если другому потоку будет нужен доступ к переменной, защищённой мьютексом, то этот поток блокируется до тех пор, пока мьютекс не будет освобождён.

Объекты синхронизации

Семафо́р (англ. semaphore) - объект, ограничивающий количество потоков, которые могут войти в заданный участок кода.

Основное назначение - синхронизация и защита передачи данных через разделяемую память, а также для синхронизации работы процессов и потоков.
Объект ядра ОС.

Принцип работы:
Семафор используются для ограничения одновременного доступа к ресурсу нескольких потоков. Используя семафор, можно организовать работу программы таким образом, что к ресурсу одновременно смогут получить доступ несколько потоков, однако количество этих потоков будет ограничено.
Создавая семафор, указывается максимальное количество потоков, которые одновременно смогут работать с ресурсом. Каждый раз, когда программа обращается к семафору, значение счетчика ресурсов семафора уменьшается на единицу. Когда значение счетчика ресурсов становится равным нулю, семафор недоступен.

Объекты синхронизации

Критическая секция (англ. critical section) - объект синхронизации потоков, позволяющий предотвратить одновременное выполнение некоторого набора операций (обычно связанных с доступом к данным) несколькими потоками. Критическая секция выполняет те же задачи, что и ***мьютекс***.

Особенности:

В операционных системах семейства Microsoft Windows разница между мьютексом и критической секцией в том, что мьютекс является объектом ядра и может быть использован несколькими процессами одновременно, критическая секция же принадлежит процессу и служит для синхронизации только его потоков.

Принцип работы:

Критические секции Windows имеют оптимизацию, заключающуюся в использовании атомарно изменяемой переменной. Захват критической секции означает атомарное увеличение переменной на 1. Переход к ожиданию осуществляется только в случае, если значение переменной до захвата было уже больше 0, то есть происходит реальное «соревнование» двух или более потоков за ресурс.

Объекты синхронизации

События представляют ресурс для обеспечения синхронизации в масштабе операционной системы.

Особенности: события оповещают потоки об окончании какой-либо операции. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта события, другая – его состояние (свободен или занят).
События являются объектами ядра операционной системы.

Принцип работы:
Событие находится в сигнальном состоянии, если его установил какой-нибудь поток. Если для работы программы требуется, чтобы в случае возникновения события на него реагировал только один из потоков, в то время как все остальные потоки продолжали ждать, то используют ***single event***.
Manual event - предоставляет доступ при сигнальном состоянии множеству потоков.

Объекты синхронизации

Взаимоисключающий доступ представляют собой еще один ресурс для обеспечения синхронизации в масштабе операционной системы.

Функции атомарного доступа к переменным:

- Присваивание целого числа
- Присваивание указателя
- Условное присваивание целого числа
- Условное присваивание указателя
- Прибавление целого числа
- Инкремент целого числа
- Декремент целого числа

Монопольное блокирование

В основу синхронизации положено понятие ***блокировки***, посредством которой организуется управление доступом к кодовому блоку в объекте. Когда объект заблокирован одним потоком, остальные потоки не могут получить доступ к заблокированному кодовому блоку.

Особенности использования блокировки.

- Если блокировка любого заданного объекта получена в одном потоке, то после блокировки объекта она не может быть получена в другом потоке.
- Остальным потокам, пытающимся получить блокировку того же самого объекта, придется ждать до тех пор, пока объект не окажется в разблокированном состоянии.
- Когда поток выходит из заблокированного фрагмента кода, соответствующий объект разблокируется.

Блокировка и безопасность потоков

Оператор **lock** определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока.

Оператор lock используется для создания критических секций.

```
lock (lockObj)
{
    // синхронизируемые операторы
}
```

где **lockObj** - **обозначает ссылку на синхронизируемый объект**.

Требование:

- Использовать в качестве объекта синхронизации можно любой объект, видимый каждому участвующему потоку;
- **Объект синхронизации должен быть ссылочного типа**;
- Объект синхронизации обычно является закрытым (потому что это помогает инкапсулировать логику блокирования) и, как правило, представляет собой поле экземпляра или статическое поле.

Блокировка и безопасность потоков

Класс Monitor предоставляет механизм для синхронизации доступа к объектам.

Пространство имен: **System.Threading**

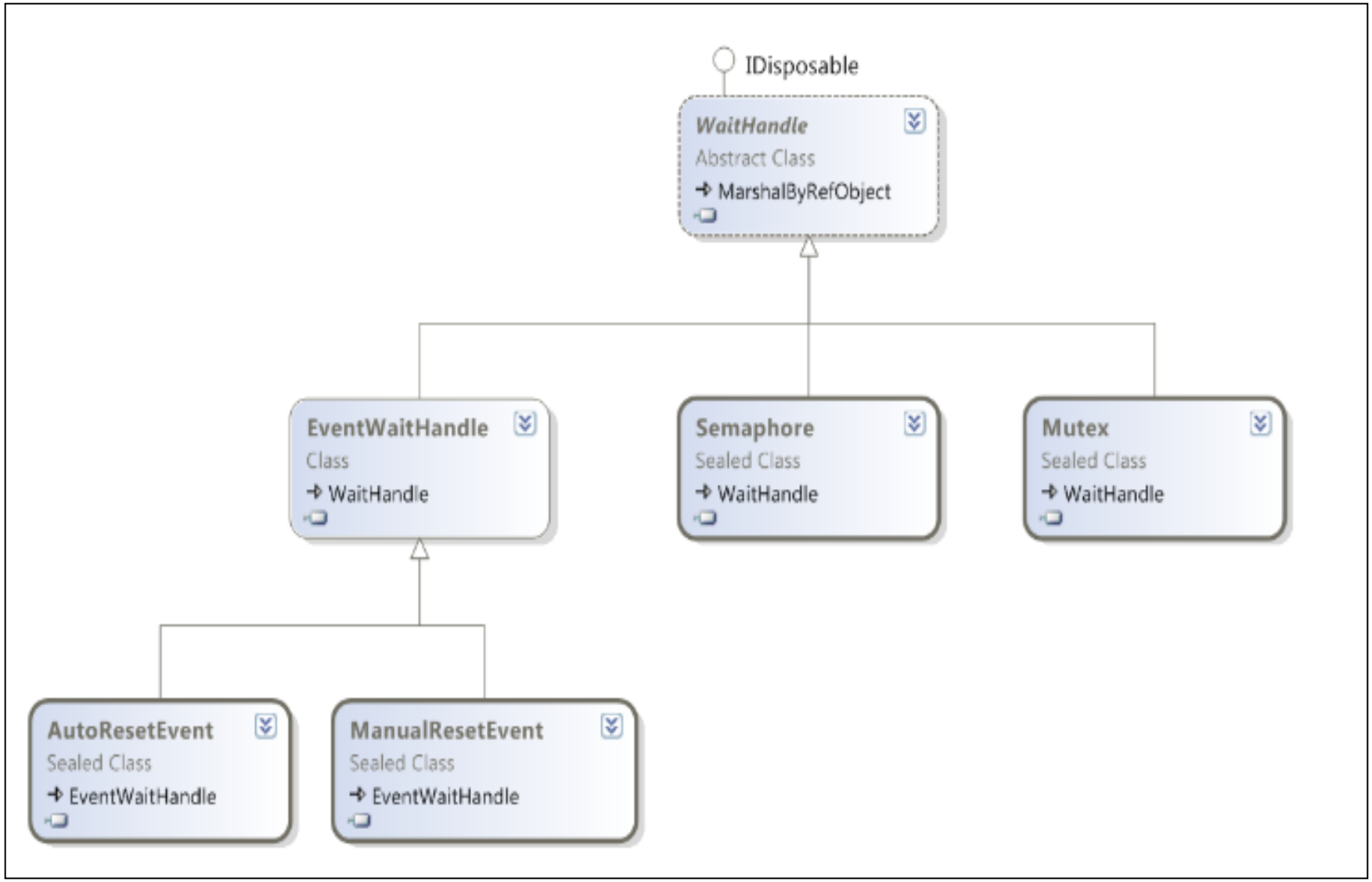
Класс Monitor используется для создания критических секций.

```
lock (obj)
{
    DoSomething();
}
```

```
object tmp = obj;
try
{
    Monitor.Enter(tmp);
    DoSomething();
}
finally
{
    Monitor.Exit(tmp);
}
```

- Метод **Monitor.Enter()** блокирует объект tmp для доступа из других потоков
- Метод **Monitor.Exit()** осуществляет освобождение объекта tmp, который становится доступным для других потоков.
- Метод **Monitor.Wait()** освобождает блокировку объекта и переводит поток в очередь ожидания объекта.

Объекты ядра Windows



Классы WaitHandle и EventWaitHandle

- **Класс `WaitHandle` обычно используется в качестве базового для объектов синхронизации.**

Классы, производные от WaitHandle, определяют механизм сигнализации о предоставлении или освобождении монопольного доступа к общему ресурсу, но используют унаследованные методы WaitHandle для блокирования во время ожидания доступа к общим ресурсам.

- **Класс `EventWaitHandle` позволяет потокам взаимодействовать друг с другом путем передачи сигналов.**

Обычно один или несколько потоков блокируются на EventWaitHandle до тех пор пока не заблокированные потоки не вызывают метод Set() для освобождения одного или нескольких заблокированных потоков.

Mutex

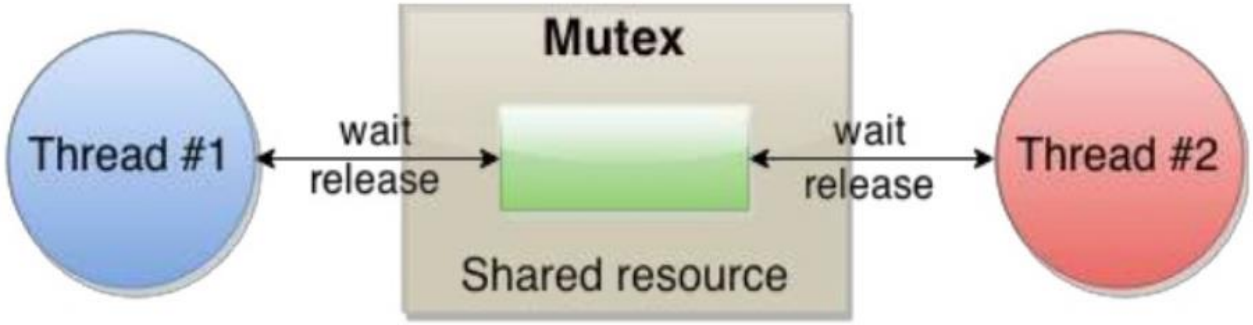
Класс **Mutex** является классом-оболочкой .NET над соответствующим объектом ОС Windows "мьютекс« и позволяет обеспечить синхронизацию потоков среди множества процессов.

Мьютексы бывают двух типов:

- локальные мьютексы (без имени)
- именованные системные мьютексы.

Локальный мьютекс существует только внутри одного процесса. Он может использоваться любым потоком в процессе, который содержит ссылку на объект Mutex, представляющий мьютекс.

Именованный объект Mutex представляет отдельный локальный мьютекс. Именованные системные мьютексы доступны в пределах всей операционной системы и могут быть использованы для синхронизации действий процессов.



Mutex

Основную работу по синхронизации выполняют методы:

- WaitOne()
- ReleaseMutex()

Метод **WaitOne**() (определен в классе WaitHandle) приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс. Мьютекс захватывает первый пришедший поток. После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода **ReleaseMutex**().Следовательно, когда выполнение дойдет до вызова **WaitOne**(), следующий поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

```
public sealed class Mutex : WaitHandle
{
    public Mutex();
    public Mutex(bool initiallyOwned);
    public Mutex(bool initiallyOwned, string name);
    public Mutex(bool initiallyOwned, string name, out bool createdNew);
    public Mutex(bool initiallyOwned, string name, out bool createdNew,
                                                    MutexSecurity mutexSecurity);
    public MutexSecurity GetAccessControl();
    public static Mutex OpenExisting(string name);
    public static Mutex OpenExisting(string name, MutexRights rights);
    public void ReleaseMutex();
    public void SetAccessControl(MutexSecurity mutexSecurity);
    public static bool TryOpenExisting(string name, out Mutex result);
    public static bool TryOpenExisting(string name, MutexRights rights,
                                                        out Mutex result);
}
```

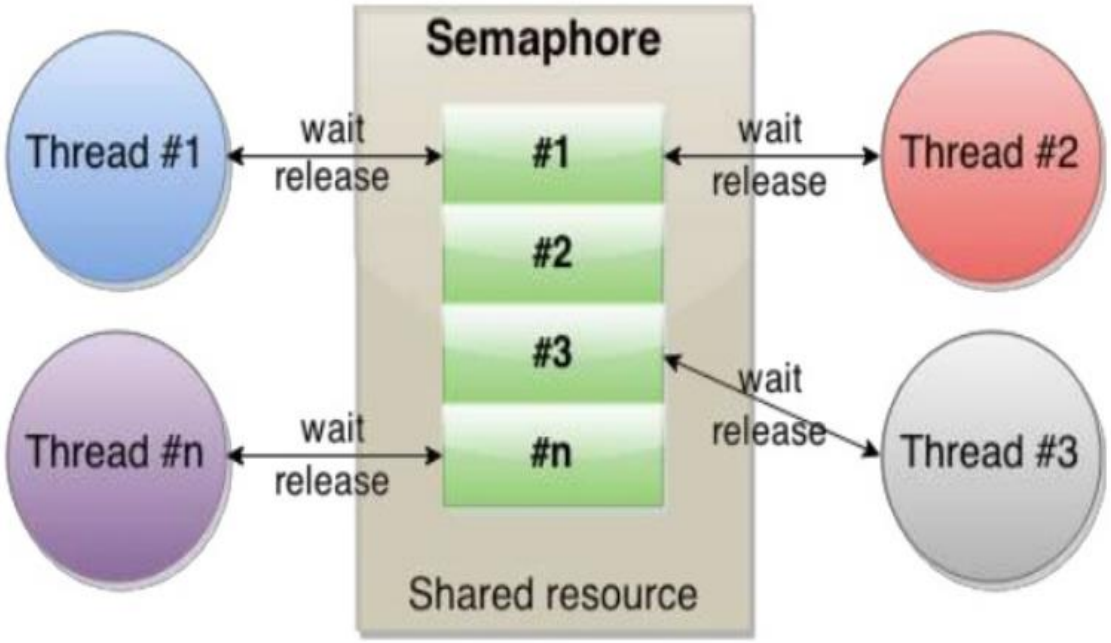
Semaphore

Класс Semaphore *обеспечивает синхронизацию потоков* подобно мьютексу, но позволяет предоставить одновременный доступ к общему ресурсу не одному, а нескольким потокам.

```
public Semaphore(int initialCount, int maximumCount)
```

`initialCount` - количество первоначально доступных разрешений;
`maximumCount` - максимальное количество разрешений, которые может дать семафор.

Семафор управляет **доступом к общему ресурсу**, используя для этой цели **счетчик**.



Semaphore

- Метод **WaitOne**() используется для ожидания получения семафора После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия.
- Метод **Release**() используется для высвобождения семафора. После этого в семафоре освобождается одно место, которое заполняет другой поток.

```
public sealed class Semaphore : WaitHandle
{
    public Semaphore(int initialCount, int maximumCount);
    public Semaphore(int initialCount, int maximumCount, string name);
    public Semaphore(int initialCount, int maximumCount,
                                                string name, out bool createdNew);
    public Semaphore(int initialCount, int maximumCount, string name,
                     out bool createdNew, SemaphoreSecurity semaphoreSecurity);
    public SemaphoreSecurity GetAccessControl();
    public static Semaphore OpenExisting(string name);
    public static Semaphore OpenExisting(string name, SemaphoreRights rights);
    public int Release();
    public int Release(int releaseCount);
    public void SetAccessControl(SemaphoreSecurity semaphoreSecurity);
    public static bool TryOpenExisting(string name, out Semaphore result);
}
```

AutoResetEvent

Класс **AutoResetEvent** позволяет потокам взаимодействовать друг с другом путем передачи сигналов. **Особенность**: как правило, этот класс используется, когда потокам требуется исключительный доступ к ресурсу.

```
// initialState:
// Значение true для задания начального состояния сигнальным;
// false для задания несигнального начального состояния.
public AutoResetEvent(bool initialState);
```

- Метод **WaitOne**() приостанавливает выполнение вызывающего потока до тех пор, пока не будет получено уведомление о событии.
- Метод **Reset**() возвращает событийный объект в несигнальное состояние.
- Метод **Set**() устанавливает событийный объект в сигнальное состояние

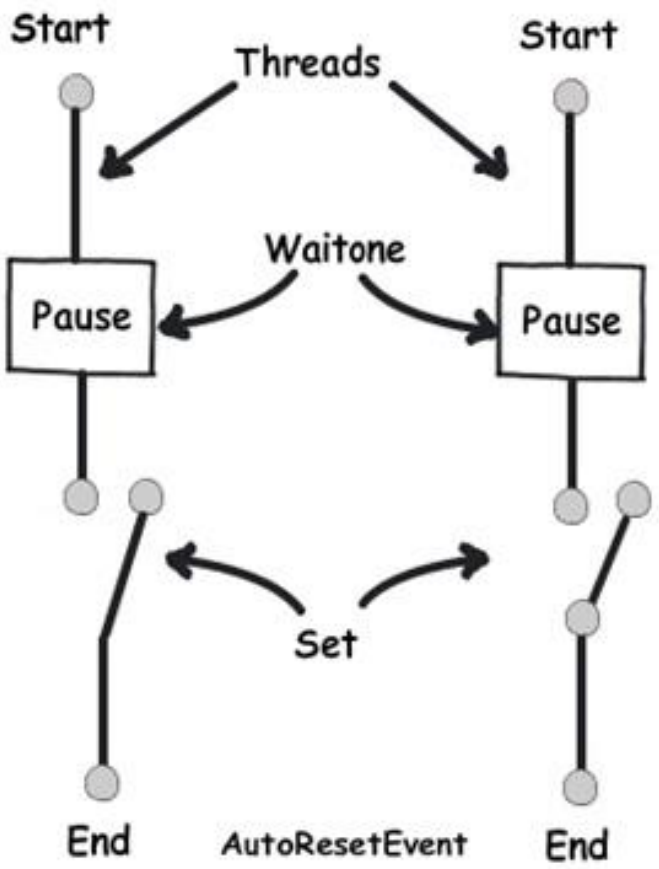
ManualResetEvent

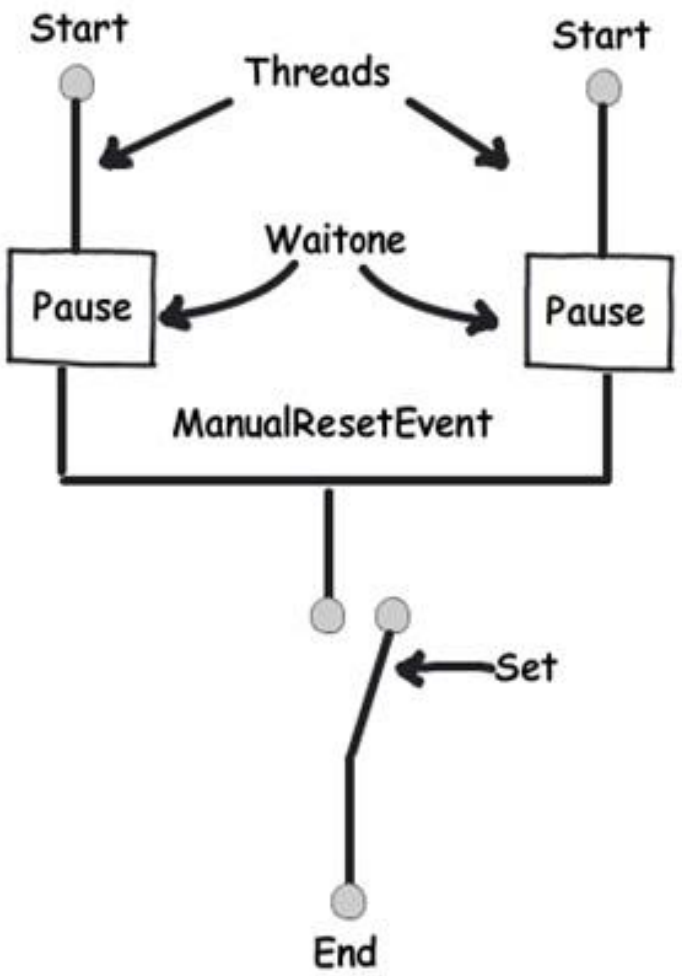
Класс **ManualResetEvent** позволяет потокам взаимодействовать друг с другом путем передачи сигналов.
Особенность: как правило, этот класс используется, когда потокам требуется исключительный доступ к ресурсу.

```
// initialState:
// Значение true для задания начального состояния сигнальным;
// false для задания несигнального начального состояния.
public ManualResetEvent (bool initialState);
```

- Метод **WaitOne**() приостанавливает выполнение вызывающего потока до тех пор, пока не будет получено уведомление о событии.
- Метод **Reset**() возвращает событийный объект в несигнальное состояние.
- Метод **Set**() устанавливает событийный объект в сигнальное состояние

ManualResetEvent vs AutoResetEvent





Принцип работы – «турникет» (приложил карту, дверь открылась, прошел, двери закрылись)

Принцип работы – «ворота» (ворота открылись, кто хотел, те прошли)

AutoResetEvent

```
public class Server
{
    private AutoResetEvent autoEventReset;
    public Server()
    {  autoEventReset = new AutoResetEvent(false); // нач. состояние - несигнальное
    }

    public void Ping()
    {   // Пропускает поток (если доступ не заблокирован)
        // затем блокирует текущий поток до получения сигнала объектом
        autoEventReset.WaitOne();
        Console.WriteLine("Подключился поток: {0}", Thread.CurrentThread.Name);
        Console.Write("Запросы: ");
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(1000);  Console.Write(" {0}", i);
        }
        Console.WriteLine();
    }

    public void Stop()
    {   // Устанавливает несигнальное состояние события, что блокирует доступ всех потоков
        autoEventReset.Reset();
    }

    public void Start()
    {   // Устанавливает сигнальное состояние события,
        // что позволяет продолжить выполнение одному или нескольким ожидающим потокам
        autoEventReset.Set();
    }
}
```

ManualResetEvent

```
public class Server
{
    private ManualResetEvent manualEventReset;
    public Server()
    {manualEventReset = new ManualResetEvent (false); // нач. состояние - несигнальное
    }

    public void Ping()
    {   // Пропускает поток (если доступ не заблокирован)
        // затем блокирует текущий поток до получения сигнала объектом
        manualEventReset.WaitOne();
        manualEventReset.Reset();
        Console.WriteLine("Подключился поток: {0}", Thread.CurrentThread.Name);
        Console.Write("Запросы: ");
        for (int i = 0; i < 5; i++)
        {
            Thread.Sleep(1000);  Console.Write(" {0}", i);
        }
        Console.WriteLine();
    }

    public void Stop()
    {   // Устанавливает несигнальное состояние события, что блокирует доступ всех потоков
        manualEventReset.Reset();
    }

    public void Start()
    {   // Устанавливает сигнальное состояние события,
        // что позволяет продолжить выполнение одному или нескольким ожидающим потокам
        manualEventReset.Set();
    }
}
```

Взаимоисключающий доступ в .NET Framework . Класс Interlocked

Класс **Interlocked** предоставляет доступ к атомарным операциям, доступным в нескольких потоках.

Особенности: операции, выполняемые при помощи методов класса Interlocked гарантировано блокируются для остальных потоков, что позволяет избежать некоторых проблем синхронизации.

Атомарные операции - операции, выполняющиеся как единое целое либо не выполняющиеся вовсе. Атомарность операций имеет особое значение в многопроцессорных компьютерах (и многозадачных операционных системах), так как доступ к неразделяемым ресурсам должен быть обязательно **атомарным**.

Взаимоисключающий доступ в .NET Framework . Класс Interlocked

```
public static class Interlocked
{
    public static int Add(ref int location1, int value);
    public static long Add(ref long location1, long value);
    public static double CompareExchange(ref double location1, double value, double comparand);
    public static float CompareExchange(ref float location1, float value, float comparand);
    public static int CompareExchange(ref int location1, int value, int comparand);
    public static IntPtr CompareExchange(ref IntPtr location1, IntPtr value, IntPtr comparand);
    public static long CompareExchange(ref long location1, long value, long comparand);
    public static object CompareExchange(ref object location1, object value, object comparand);
    public static T CompareExchange<T>(ref T location1, T value, T comparand) where T : class;
    public static int Decrement(ref int location);
    public static long Decrement(ref long location);
    public static double Exchange(ref double location1, double value);
    public static float Exchange(ref float location1, float value);
    public static int Exchange(ref int location1, int value);
    public static IntPtr Exchange(ref IntPtr location1, IntPtr value);
    public static long Exchange(ref long location1, long value);
    public static object Exchange(ref object location1, object value);
    public static T Exchange<T>(ref T location1, T value) where T : class;
    public static int Increment(ref int location);
    public static long Increment(ref long location);
    public static void MemoryBarrier();
    public static long Read(ref long location);
}
```

Подводим итоги

Достоинства:

1. Синхронизация позволяет предотвратить повреждение общих данных при одновременном доступе к этим данным разных потоков.

Недостатки:

1. **Использование блокирования приводит к снижению производительности.** (Установление и снятие блокировки требуют времени, так как для этого вызываются
дополнительные методы, причем необходимо координировать совместную работу, определяя, который из потоков нужно блокировать первым.
2. **Писать код синхронизации крайне утомительно и при этом легко допустить ошибку.** В коде следует выделить все данные, которые потенциально могут обрабатываться различными потоками в одно и то же время. Затем все эти данные заключаются в другой код, обеспечивающий их блокировку и разблокирование. Блокирование гарантирует, что доступ к ресурсу в каждый момент времени сможет получить только один поток. Однако достаточно при программировании забыть заблокировать хотя бы один фрагмент кода, и ваши данные будут повреждены.