

## Лабораторная работа 19

### Организация многопоточной обработки на основе класса Thread

**Многопоточность** (англ. Multithreading) — свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени. При выполнении некоторых задач такое разделение может достичь более эффективного использования ресурсов вычислительной машины.

Такие потоки называют также потоками выполнения (от англ. thread of execution); иногда называют «нитеями» (буквальный перевод англ. thread) или неформально «тредами».

Основной функционал для использования потоков в приложении сосредоточен в пространстве имен **System.Threading**. В нем определен класс, представляющий отдельный поток - класс **Thread**.

Для того чтобы создать новую нить (поток) в программе, нужно использовать делегат **ThreadStart**, который получает в качестве параметра метод, который нужно запускать параллельно и передать его в качестве параметра в конструктор класса **Thread**. Для того чтобы запустить, вызываем метод **Start()**

```
class Program
{
    ссылка: 1
    static void SecondThread()
    {
        for (int i = 0; i < 100; i++)
        {
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine($"{new string(' ', 10)} нить SecondThread");
        }
    }
    Ссылка: 0
    static void Main(string[] args)
    {
        ThreadStart secondThread = new ThreadStart(SecondThread);
        Thread threadSecond = new Thread(secondThread);
        threadSecond.Start();
        for (int i = 0; i < 100; i++)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine($"нить main");
        }

        Console.ReadKey();
    }
}
```

В примере выше показано, как запускать в отдельных потоках методы без параметров. А что, если надо передать какие-нибудь параметры в поток?

Для этой цели используется делегат **ParameterizedThreadStart**. Его действие похоже на функциональность делегата **ThreadStart**. Рассмотрим на примере:

```

class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        int number = 4;
        // создаем новый поток
        Thread myThread = new Thread(new ParameterizedThreadStart(Count));
        myThread.Start(number);

        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Главный поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(300);
        }

        Console.ReadLine();
    }

    Ссылка: 1
    public static void Count(object x)
    {
        for (int i = 1; i < 9; i++)
        {
            int n = (int)x;

            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * n);
            Thread.Sleep(400);
        }
    }
}

```

При использовании **ParameterizedThreadStart** мы сталкиваемся с ограничением: мы можем запускать во втором потоке только такой метод, который в качестве единственного параметра принимает объект типа **object**. Поэтому в данном случае нам надо дополнительно привести переданное значение к типу **int**, чтобы его использовать в вычислениях.

Но что делать, если нам надо передать не один, а несколько параметров различного типа? В этом случае на помощь приходит классовый подход:

```

class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        Counter counter = new Counter();
        counter.x = 4;
        counter.y = 5;

        Thread myThread = new Thread(new ParameterizedThreadStart(Count));
        myThread.Start(counter);

        //.....
    }

    Ссылка: 1
    public static void Count(object obj)
    {
        for (int i = 1; i < 9; i++)
        {
            Counter c = (Counter)obj;

            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * c.x * c.y);
        }
    }

    Ссылка: 4
    public class Counter
    {
        public int x;
        public int y;
    }
}

```

Но тут опять же есть одно ограничение: метод **Thread.Start** не является типобезопасным, то есть мы можем передать в него любой тип, и потом нам придется приводить переданный объект к нужному нам типу. Для решения данной проблемы рекомендуется объявлять все используемые методы и переменные в специальном классе, а в основной программе запускать поток через **ThreadStart**. Вот так следует реализовывать многопоточность через класс.

```
class Program
{
    Ссылка: 0
    static void Main(string[] args)
    {
        Counter counter = new Counter(5, 4);

        Thread myThread = new Thread(new ThreadStart(counter.Count));
        myThread.Start();
        //.....
    }
}

Ссылка: 3
public class Counter
{
    private int x;
    private int y;

    ссылка: 1
    public Counter(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    ссылка: 1
    public void Count()
    {
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i * x * y);
            Thread.Sleep(400);
        }
    }
}
```

Нередко в потоках (нитех) используются некоторые разделяемые ресурсы, общие для всей программы. Это могут быть общие переменные, файлы, другие ресурсы. **Например:**

```
Ссылка: 0
class Program
{
    static int x = 0;
    Ссылка: 0
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = $"Поток {i}";
            myThread.Start();
        }

        Console.ReadLine();
    }
    Ссылка: 1
    public static void Count()
    {
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
}
```

```
C:\WINDOWS\system32
Поток 0: 1
Поток 1: 1
Поток 2: 1
Поток 3: 1
Поток 4: 1
Поток 3: 2
Поток 2: 2
Поток 0: 2
Поток 4: 2
Поток 1: 2
Поток 4: 7
Поток 0: 7
Поток 1: 8
Поток 2: 8
Поток 3: 8
Поток 3: 12
Поток 4: 12
Поток 0: 12
Поток 1: 12
Поток 2: 12
Поток 0: 17
Поток 3: 17
Поток 4: 17
Поток 1: 17
Поток 2: 17
Поток 2: 22
Поток 1: 22
Поток 3: 22
Поток 4: 22
Поток 0: 22
Поток 0: 27
Поток 3: 27
Поток 4: 27
Поток 2: 27
Поток 1: 27
Поток 4: 32
Поток 1: 32
Поток 0: 32
Поток 2: 32
Поток 3: 32

C:\WINDOWS\system32
Поток 0: 1
Поток 1: 1
Поток 3: 1
Поток 2: 1
Поток 4: 1
Поток 1: 3
Поток 4: 3
Поток 3: 3
Поток 0: 3
Поток 2: 8
Поток 1: 8
Поток 3: 8
Поток 4: 8
Поток 3: 13
Поток 1: 13
Поток 0: 13
Поток 2: 13
Поток 2: 13
Поток 2: 18
Поток 4: 18
Поток 0: 18
Поток 1: 18
Поток 3: 18
Поток 0: 23
Поток 1: 23
Поток 3: 23
Поток 2: 23
Поток 4: 23
Поток 2: 28
Поток 1: 28
Поток 3: 28
Поток 0: 28
Поток 4: 33
Поток 2: 33
Поток 3: 33
Поток 1: 33

C:\WINDOWS\system32
Поток 0: 1
Поток 1: 1
Поток 2: 1
Поток 3: 1
Поток 4: 1
Поток 4: 6
Поток 0: 6
Поток 2: 6
Поток 3: 6
Поток 1: 6
Поток 3: 11
Поток 2: 11
Поток 0: 11
Поток 4: 11
Поток 1: 11
Поток 0: 16
Поток 4: 16
Поток 1: 16
Поток 2: 16
Поток 3: 16
Поток 2: 21
Поток 3: 21
Поток 4: 21
Поток 1: 21
Поток 0: 21
Поток 0: 26
Поток 1: 26
Поток 3: 26
Поток 2: 26
Поток 4: 26
Поток 4: 31
Поток 1: 31
Поток 0: 31
Поток 2: 31
Поток 3: 31
Поток 3: 36
Поток 0: 36
Поток 2: 36
Поток 1: 36
Поток 4: 36
```

В данном пример запускается пять потоков, которые работают с общей переменной `x`. И мы предполагаем, что метод выведет все значения `x` от 1 до 8. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной `x` становится непредсказуемым. Ниже показан результат, программа запускалась три раза, и все три раза были получены разные значения.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком. Для этого используется ключевое слово **lock**. Оператор **lock** определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока. И мы можем переделать предыдущий пример следующим образом:

```
class Program
{
    static int x = 0;
    static object locker = new object();
    Ссылка: 0
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = $"Поток {i}";
            myThread.Start();
        }

        Console.ReadLine();
    }
    ссылка: 1
    public static void Count()
    {
        lock (locker)
        {
            x = 1;
            for (int i = 1; i < 9; i++)
            {
                Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
                x++;
                Thread.Sleep(100);
            }
        }
    }
}
```

C:\WINDOWS\system32\cmd.exe

Поток 0: 8  
Поток 1: 1  
Поток 1: 2  
Поток 1: 3  
Поток 1: 4  
Поток 1: 5  
Поток 1: 6  
Поток 1: 7  
Поток 1: 8  
Поток 2: 1  
Поток 2: 2  
Поток 2: 3  
Поток 2: 4  
Поток 2: 5  
Поток 2: 6  
Поток 2: 7  
Поток 2: 8  
Поток 3: 1  
Поток 3: 2  
Поток 3: 3  
Поток 3: 4  
Поток 3: 5  
Поток 3: 6  
Поток 3: 7  
Поток 3: 8  
Поток 4: 1  
Поток 4: 2  
Поток 4: 3  
Поток 4: 4  
Поток 4: 5  
Поток 4: 6  
Поток 4: 7  
Поток 4: 8

Для блокировки с ключевым словом `lock` используется объект-заглушка, в данном случае это переменная `locker`. Когда выполнение доходит до оператора `lock`, объект `locker` блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток. После окончания работы блока кода, объект `locker` освобождается и становится доступным для других потоков.

## Синхронизация Monitor

Наряду с оператором `lock` для синхронизации потоков мы можем использовать мониторы, представленные классом **System.Threading.Monitor**. Фактически конструкция оператора `lock` инкапсулирует в себе синтаксис использования мониторов. А рассмотренный в выше пример будет эквивалентен следующему коду:

```
class Program
{
    static int x = 0;
    static object locker = new object();
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = $"Поток {i}";
            myThread.Start();
        }

        Console.ReadLine();
    }
}

public static void Count()
{
    bool acquiredLock = false;
    try
    {
        Monitor.Enter(locker, ref acquiredLock);
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
    finally
    {
        if (acquiredLock) Monitor.Exit(locker);
    }
}
```

```
Поток 0: 1
Поток 0: 2
Поток 0: 3
Поток 0: 4
Поток 0: 5
Поток 0: 6
Поток 0: 7
Поток 0: 8
Поток 2: 1
Поток 2: 2
Поток 2: 3
Поток 2: 4
Поток 2: 5
Поток 2: 6
Поток 2: 7
Поток 2: 8
Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 1: 6
Поток 1: 7
Поток 1: 8
Поток 3: 1
Поток 3: 2
Поток 3: 3
Поток 3: 4
Поток 3: 5
Поток 3: 6
Поток 3: 7
Поток 3: 8
Поток 4: 1
Поток 4: 2
Поток 4: 3
Поток 4: 4
Поток 4: 5
Поток 4: 6
Поток 4: 7
Поток 4: 8
```

Метод **Monitor.Enter** принимает два параметра - объект блокировки и значение типа `bool`, которое указывает на результат блокировки (если он равен `true`, то блокировка успешно выполнена). Фактически этот метод блокирует объект **locker** так же, как это делает оператор **lock**. С помощью `try...finally` с помощью метода **Monitor.Exit** происходит освобождение объекта **locker**, если блокировка осуществлена успешно, и он становится доступным для других потоков.

Кроме блокировки и разблокировки объекта класс **Monitor** имеет еще ряд методов, которые позволяют управлять синхронизацией потоков. Так, метод **Monitor.Wait** освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод **Wait**, остаются в очереди ожидания, пока не получают сигнала от метода **Monitor.Pulse** или **Monitor.PulseAll**, посланного владельцем блокировки. Если метод **Monitor.Pulse** отправил сигнал, то поток, находящийся во главе

очереди ожидания, получает сигнал и блокирует освободившийся объект. Если же метод **Monitor.PulseAll** отправлен, то все потоки, находящиеся в очереди ожидания, получают сигнал и переходят в очередь готовности, где им снова разрешается получать блокировку объекта.

### Синхронизация Mutex.

Еще один инструмент управления синхронизацией потоков представляет класс **Mutex**, также находящийся в пространстве имен **System.Threading**. Данный класс является классом-оболочкой над соответствующим объектом ОС Windows "мьютекс". Перепишем прошлый пример, используя мьютексы:

```
class Program
{
    static int x = 0;
    static Mutex mutex = new Mutex();
    Ссылка: 0
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = $"Поток {i}";
            myThread.Start();
        }

        Console.ReadLine();
    }
    Ссылка: 1
    public static void Count()
    {
        mutex.WaitOne();
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
        mutex.ReleaseMutex();
    }
}
```

Выбрать C:\WINDOWS

Поток 0: 1  
Поток 0: 2  
Поток 0: 3  
Поток 0: 4  
Поток 0: 5  
Поток 0: 6  
Поток 0: 7  
Поток 0: 8  
Поток 1: 1  
Поток 1: 2  
Поток 1: 3  
Поток 1: 4  
Поток 1: 5  
Поток 1: 6  
Поток 1: 7  
Поток 1: 8  
Поток 2: 1  
Поток 2: 2  
Поток 2: 3  
Поток 2: 4  
Поток 2: 5  
Поток 2: 6  
Поток 2: 7  
Поток 2: 8  
Поток 3: 1  
Поток 3: 2  
Поток 3: 3  
Поток 3: 4  
Поток 3: 5  
Поток 3: 6  
Поток 3: 7  
Поток 3: 8  
Поток 4: 1  
Поток 4: 2  
Поток 4: 3  
Поток 4: 4  
Поток 4: 5  
Поток 4: 6  
Поток 4: 7  
Поток 4: 8

Сначала создаем объект мьютекса: **Mutex mutexObj = new Mutex()**.

Основную работу по синхронизации выполняют методы **WaitOne()** и **ReleaseMutex()**. Метод **mutexObj.WaitOne()** приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс **mutexObj**.

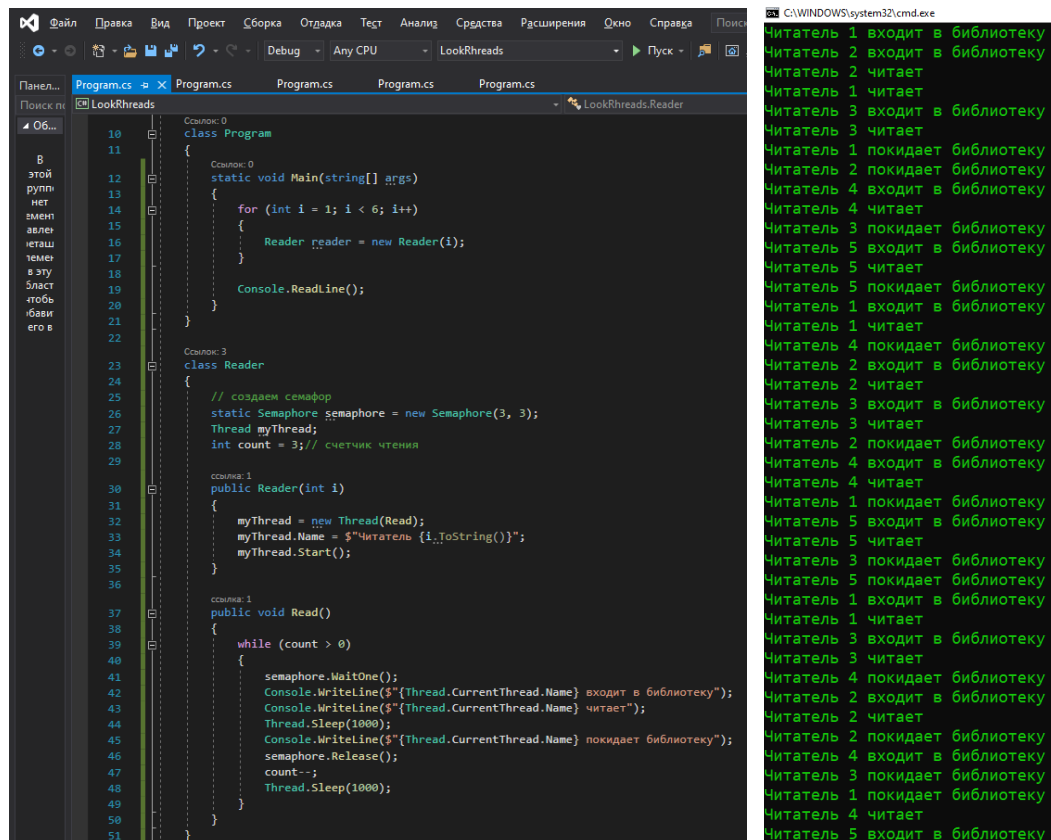
После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода **mutexObj.ReleaseMutex()**

Таким образом, когда выполнение дойдет до вызова **mutexObj.WaitOne()**, поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.

### Синхронизация Semaphore.

Еще один инструмент, который предлагает нам платформа .NET для управления синхронизацией, представляют **семафоры**. Семафоры позволяют ограничить доступ определенным количеством объектов.

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что одновременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:



```
class Program
{
    static void Main(string[] args)
    {
        for (int i = 1; i < 6; i++)
        {
            Reader reader = new Reader(i);
            Console.ReadLine();
        }
    }
}

class Reader
{
    // создаем семафор
    static Semaphore semaphore = new Semaphore(3, 3);
    Thread myThread;
    int count = 3; // СЧЕТЧИК ЧТЕНИЯ

    public Reader(int i)
    {
        myThread = new Thread(Read);
        myThread.Name = $"Читатель {i.ToString()}";
        myThread.Start();
    }

    public void Read()
    {
        while (count > 0)
        {
            semaphore.WaitOne();
            Console.WriteLine($"{Thread.CurrentThread.Name} входит в библиотеку");
            Console.WriteLine($"{Thread.CurrentThread.Name} читает");
            Thread.Sleep(1000);
            Console.WriteLine($"{Thread.CurrentThread.Name} покидает библиотеку");
            semaphore.Release();
            count--;
            Thread.Sleep(1000);
        }
    }
}
```

```
Читатель 1 входит в библиотеку
Читатель 2 входит в библиотеку
Читатель 2 читает
Читатель 1 читает
Читатель 3 входит в библиотеку
Читатель 3 читает
Читатель 1 покидает библиотеку
Читатель 2 покидает библиотеку
Читатель 4 входит в библиотеку
Читатель 4 читает
Читатель 3 покидает библиотеку
Читатель 5 входит в библиотеку
Читатель 5 читает
Читатель 5 покидает библиотеку
Читатель 1 входит в библиотеку
Читатель 1 читает
Читатель 4 покидает библиотеку
Читатель 2 входит в библиотеку
Читатель 2 читает
Читатель 3 входит в библиотеку
Читатель 3 читает
Читатель 2 покидает библиотеку
Читатель 4 входит в библиотеку
Читатель 4 читает
Читатель 1 покидает библиотеку
Читатель 5 входит в библиотеку
Читатель 5 читает
Читатель 5 покидает библиотеку
```

В данной программе читатель представлен классом **Reader**. Он инкапсулирует всю функциональность, связанную с потоками, через переменную **Thread myThread**.

Для создания семафора используется класс **Semaphore**: **static Semaphore sem = new Semaphore(3, 3);**. Его конструктор принимает два параметра: первый указывает, какому числу объектов изначально будет доступен семафор, а второй параметр указывает, какой максимальное число объектов будет использовать данный семафор. В данном случае у нас только три читателя могут одновременно находиться в библиотеке, поэтому максимальное число равно 3.

Основной функционал сосредоточен в методе **Read**, который и выполняется в потоке. В начале для ожидания получения семафора используется метод **semaphore.WaitOne()**. После того, как в семафоре освободится место, данный поток заполняет свободное место и начинает выполнять все дальнейшие действия. После окончания чтения мы



высвобождаем семафор с помощью метода **semaphore.Release()**. После этого в семафоре освобождается одно место, которое заполняет другой поток.

А в методе Main нам остается только создать читателей, которые запускают соответствующие потоки.

### Практическая часть.

Общее:

Каждые t1 секунды вам необходимо из отдельного потока (нити не Main) сохранять процент выполнения делегата, текущий результат, а так же время до миллисекунд в формате \$"Время:{time:hh:mm:ss:ffff} Процент: {perhent,4:00.00} Решение {solution,6}" в файл state.out.

Каждые t2 секунды вам необходимо из отдельного потока (нити не Main) считать итерацию задачи и сохранять в формате \$"Делегат Время:{time:hh:mm:ss:ffff} Процент: {perhent,4:00.00} Решение {solution,6}" те же данные в файл state.out.

Для синхронизированного доступа к файлу использовать указанный ниже объект.

№	Задание
1	Посчитать сумму нечетных цифр числа из файла number.in. T1 – 0.5 t2 – 0.6 <b>Объект синхронизации — monitor</b>
2	Посчитать количество цифр числа, кратных трем из файла number.in. T1 – 0.4, t2 – 0.55 <b>Объект синхронизации — monitor</b>
3	Посчитать сумму цифр числа, кратных двум из файла number.in T1 – 0.4, t2 – 0.55 <b>Объект синхронизации — monitor</b>
4	Посчитать сумму нечетных цифр числа из файле number.in. T1 – 0.4 t2 – 0.5 <b>Объект синхронизации — semaphore</b>
5	Посчитать количество цифр числа, кратных трем из файла number.in. T1 – 0.3, t2 – 0.5 <b>Объект синхронизации — semaphore</b>
6	Посчитать сумму цифр числа, кратных двум из файла number.in T1 – 0.2, t2 – 0.4 <b>Объект синхронизации — semaphore</b>
7	Посчитать сумму нечетных цифр числа из файла number.in. T1 – 0.6 t2 – 0.7 <b>Объект синхронизации — mutex</b>
8	Посчитать количество цифр числа, кратных трем из файла number.in. T1 – 0.6, t2 – 0.8 <b>Объект синхронизации — mutex</b>
9	Посчитать сумму цифр числа, кратных двум из файла number.in. T1 – 0.9, t2 – 1 <b>Объект синхронизации — mutex</b>
10	Посчитать сумму нечетных цифр числа из файла number.in. T1 – 0.6 t2 – 0.7 <b>Объект синхронизации — конструкция lock</b>
11	Посчитать количество цифр числа, кратных трем из файла number.in. T1 – 0.6, t2 – 0.8 <b>Объект синхронизации — конструкция lock</b>
12	Посчитать сумму цифр числа, кратных двум из файла number.in. T1 – 0.9, t2 – 1 <b>Объект синхронизации — конструкция lock</b>

### Контрольные вопросы:

1. Формы параллельных вычислений?
2. Класс Thread? Для чего нужен, где находится?
3. Состояние класса thread?
4. Классы и способы синхронизации потоков?