

Многопоточное программирование

Тема 18

Многопоточная программа

- Многопоточная программа состоит из двух или больше частей, которые могут выполняться одновременно.
- Каждая часть такой программы называется **потоком** (thread), и каждый поток определяет собственный путь выполнения инструкций. При выполнении программы каждому потоку процессор выделяет определенный квант времени.
- Различают два вида многозадачности:
 - с ориентацией на процессы,
 - с ориентацией на потоки.

Процесс

- **Процесс** — это программа, т.е. многозадачность, ориентированная на процессы, — это средство, позволяющее компьютеру выполнять две или больше программ одновременно.
- **Поток** — это управляемая единица выполняемого кода (функция).
- Все процессы имеют по крайней мере один поток (м.б. больше).
- Т.е. одна программа может выполнять сразу две и более задач.

Многозадачность

- **Процессно-**ориентированная многозадачность обеспечивает одновременное выполнение программ, а **поточно-**ориентированная —одновременное выполнение частей одной и той же программы.
- Управление многопоточностью осуществляет планировщик потоков (ОС).
- На однопроцессорных компьютерах планировщик потоков использует квантование времени – быстрое переключение между выполнением каждого из активных потоков.
- На многопроцессорных компьютерах многопоточность реализована как смесь квантования времени и подлинного параллелизма, когда разные потоки выполняют код на разных процессорах.
- Говорят, что поток **вытесняется**, когда его выполнение приостанавливается из-за внешних факторов. В большинстве случаев поток не может контролировать, когда и где он будет вытеснен.

Класс Thread

- Все процессы имеют, по крайней мере, один поток управления, который обычно называется основным (main thread), поскольку именно с этого потока начинается выполнение программы.
- Многопоточная система C# встроена в класс **Thread**, который инкапсулирует поток управления.

Класс Thread

- Классы, которые поддерживают многопоточное программирование, определены в пространстве имен `System.Threading`.
- Класс `Thread` является **sealed**-классом, т.е. он не может иметь наследников.
- В классе `Thread` определен ряд методов и свойств для управления потоками.

Методы и свойства класса Thread

- **IsAlive** - указывает, работает ли поток в текущий момент
- **IsBackground** - указывает, является ли поток фоновым
- **Name** - содержит имя потока
- **ManagedThreadId** - возвращает числовой идентификатор текущего потока

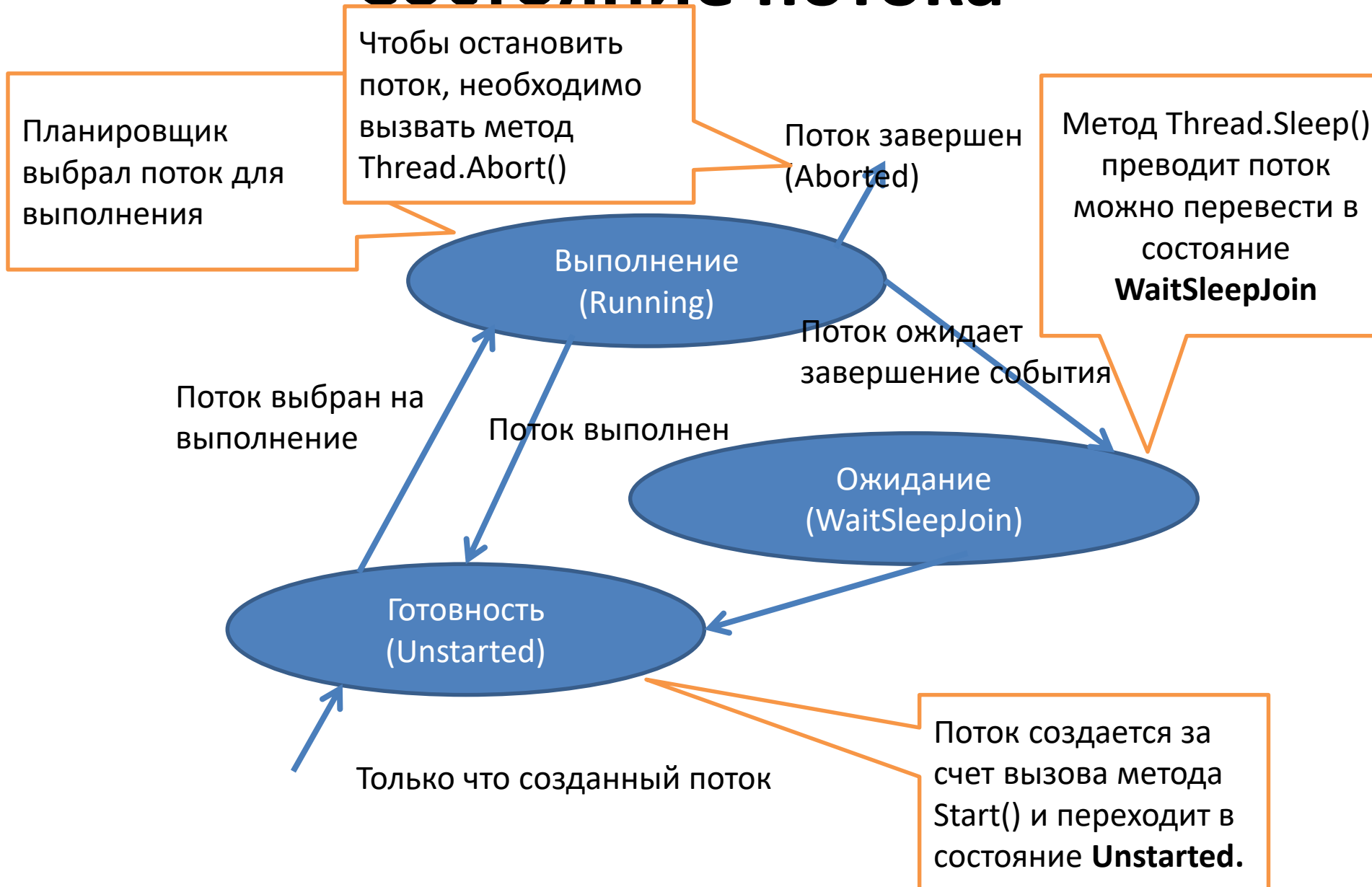
Методы и свойства класса Thread

- **Priority**: хранит приоритет потока - значение перечисления **ThreadPriority**:
 - Lowest
 - BelowNormal
 - Normal
 - AboveNormal
 - Highest

Методы и свойства класса Thread

- **ThreadState** возвращает состояние потока - одно из значений перечисления **ThreadState**:
 - **Aborted**: поток остановлен, но пока еще окончательно не завершен
 - **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла
 - **Background**: поток выполняется в фоновом режиме
 - **Running**: поток запущен и работает (не приостановлен)
 - **Stopped**: поток завершен
 - **StopRequested**: поток получил запрос на остановку
 - **Suspended**: поток приостановлен
 - **SuspendRequested**: поток получил запрос на приостановку
 - **Unstarted**: поток еще не был запущен
 - **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join

Состояние потока



Создание главного потока и получение информации

```
// получаем текущий поток
```

```
Thread t = Thread.CurrentThread;
```

```
//получаем имя потока
```

```
Console.WriteLine($"Имя потока: {t.Name}") ; //пустая строка
```

```
t.Name = "Метод Main";
```

```
Console.WriteLine($"Имя потока: {t.Name}");
```

```
Console.WriteLine($"Запущен ли поток: {t.IsAlive}");
```

```
Console.WriteLine($"Приоритет потока: {t.Priority}" );
```

```
Console.WriteLine($"Статус потока: {t.ThreadState} ");
```

```
Имя потока:
```

```
Имя потока: Метод Main
```

```
Запущен ли поток: True
```

```
Приоритет потока: Normal
```

```
Статус потока: Running
```

Создание второго потока в функции `main()`

- `Thread (ThreadStart)` – параметром является объект делегата `ThreadStart`, с помощью объекта делегата передается метод, который должен быть запущен в отдельном потоке.
- `Thread (ParametrizedThreadStart)` – параметром является объект делегата `ParametrizedThreadStart`, с помощью объекта делегата передается метод, который должен быть запущен в отдельном потоке, метод имеет параметр.

Создание второго потока в функции `main()`

- Пример 2: поток на базе метода без параметров
- Пример 3: поток на базе метода с параметром `object`
- Пример 4: поток на базе метода с параметром типа `class...`
- Пример 5: поток на базе делегата

Создание второго потока в функции main()

```
class Program
{
    public static void Count()
    {
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Второй поток:");
            Console.WriteLine(i);
            Thread.Sleep(400);
        }
    }

    static void Main(string[] args)
    {
        //создаем объект потока с помощью делегата ThreadStart
        Thread myThread = new Thread(new ThreadStart(Count));
        myThread.Start(); // запускаем поток

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

Целевой метод
делегата без
параметров

C:\WINDOWS\system32\cmd.exe

```
Главный поток:
1
Второй поток:
1
Главный поток:
4
Второй поток:
2
Главный поток:
9
Второй поток:
3
Главный поток:
16
Второй поток:
4
Главный поток:
25
Главный поток:
36
Второй поток:
5
Главный поток:
49
Второй поток:
```

Создание второго потока в функции main()

```
class Program
{
    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);
        }
    }
    static void Main(string[] args)
    {
        int number = 4; // параметр
        // создаем новый поток
        Thread myThread1 = new Thread(new ParameterizedThreadStart(Count));
        myThread1.Start(number);

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

Целевой метод
делегата с
параметрами

Параметр метода

Создание второго потока в функции main()

```
public class Counter
{
    private int x;
    private int y;
    public Counter(int _x, int _y)
    {
        this.x = _x;
        this.y = _y;
    }

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

```
class Program
{
    static void Main(string[] args)
    {
        Counter counter = new Counter(5, 4);
        // создаем новый поток
        Thread myThread = new Thread(new ThreadStart(counter.Count));
        myThread.Start();

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

Метод класса

Создание потока с помощью делегата

```
internal class Program
{
    public delegate int DisplayHandler();

    static int Display()
    {
        Console.WriteLine("Method Display is working...");
        int res = 0;
        for (int i = 0; i < 10; i++)
        {
            res += i * i;
        }
        Thread.Sleep(1000);
        Console.WriteLine("The end of method Display");
        return res;
    }

    static void Main(string[] args)
    {
        DisplayHandler dh = new DisplayHandler(Display);
        int result=dh.Invoke();//вызов метода Display
        Console.WriteLine("Method Main is working...");
        Console.WriteLine($"res={result}");
    }
}
```

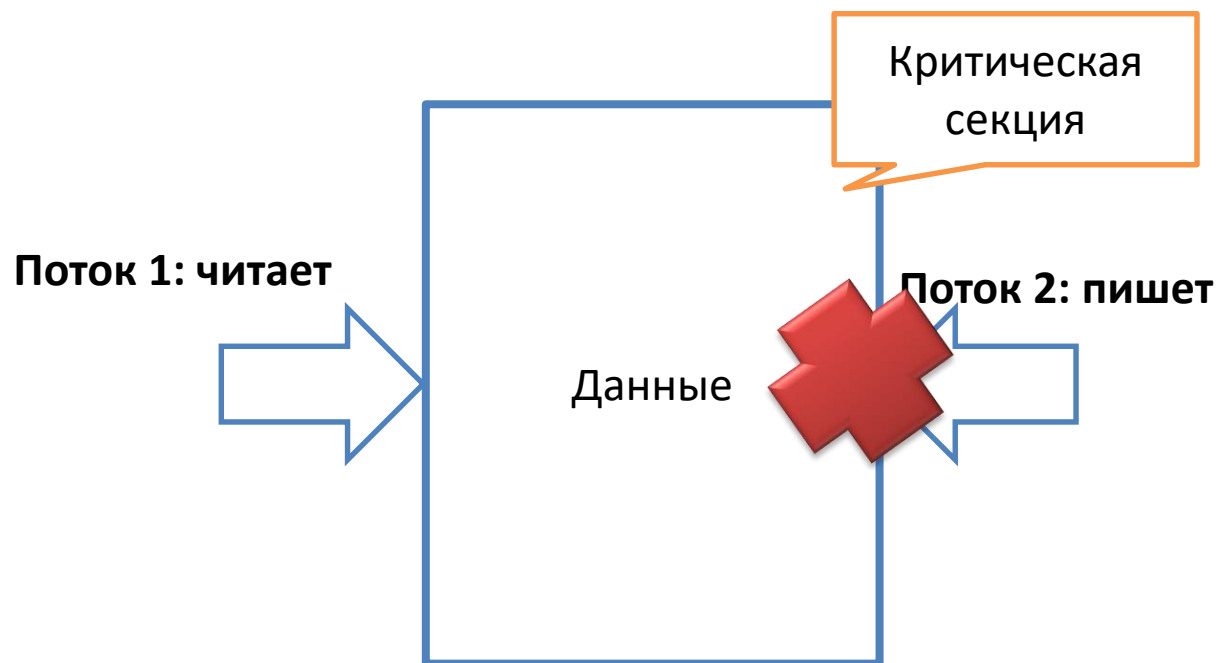
Алгоритм создания потоков

1. Создать метод, который будет точкой входа для нового потока.
2. Создать новый делегат **ParametrizedThreadStart/ThreadStart**, передав конструктору адрес метода, определенного на предыдущем шаге.
3. Создать объект **Thread**, передав в качестве аргумента конструктора **ParametrizedThreadStart/ThreadStart**.
4. Вызвать метод **Thread.Start()**. Это запустит поток на методе, который указан делегатом, созданным на втором шаге, как только это будет возможно.

Синхронизация

- **Синхронизация** – это процесс координации потоков, если:
 - двум потокам требуется один ресурс;
 - выполнение второго потока зависит от результатов первого потока.
- В этом случае необходимо выполнять синхронизацию потоков, которая заключается в согласовании их скоростей путем приостановки потока до наступления какого-то события, а потом его активизации при наступлении этого события.
- **Критическая секция** – это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные этой части программы изменяются другими потоками, в то время, когда выполнение этой части еще не завершено. Критическая секция определяется по отношению к определенным **критическим данным**, при несогласованном изменении которых и может возникнуть нежелательный эффект.
- **Блокировка** - управление доступом к некоторому блоку кода в объекте.
- На то время, когда объект заблокирован одним потоком, никакой другой поток не может получить доступ к заблокированному блоку кода. Когда поток снимет блокировку, объект станет доступным для использования другим потоком.

Синхронизация



Синхронизация

- Пример 6: Использование общей переменной
- Запускаются пять потоков, которые работают с общей переменной x .
- Предполагаем, что метод выведет все значения x от 1 до 8 для каждого потока.
- В процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Синхронизация

```
class Program
{
    static int x = 0; //общая переменная
    public static void Count()
    {
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}") ;
            x++;
            Thread.Sleep(100);
        }
    }

    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
}
```

4 потока

Thread3
Thread4
Thread0
Thread1
Thread2
Thread0
Thread4
Thread2
Thread3
Thread1
Thread3
Thread4
Thread1
Thread2
Thread0
Thread3
Thread2
Thread1
Thread0
Thread4
Thread2
Thread4

Синхронизация

- Ситуация, когда два или более потоков обрабатывают разделяемые данные и конечный результат зависит от соотношения скоростей потоков называется **гонками**.
- Чтобы исключить эффект гонок по отношению к критическим данным, необходимо, чтобы в каждый момент времени с ними работал только **один поток**.
- Т.е. необходимо синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком

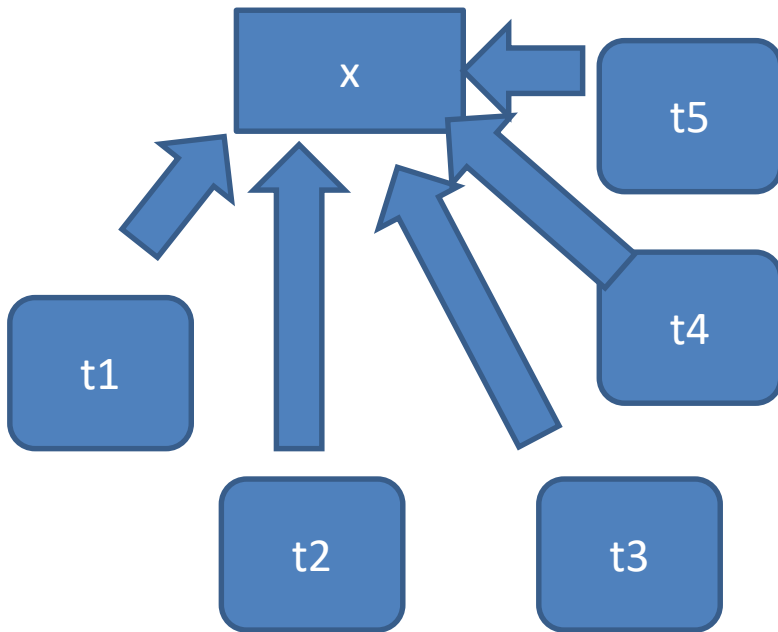
Синхронизация

```
lock(object)
{
    // Инструкции, подлежащие
    // синхронизации.
}
```

где object - ссылка на
синхронизируемый объект .

- **Инструкция lock**
гарантирует, что указанный блок кода, защищенный блокировкой для данного объекта, может быть использован только потоком, который получает эту блокировку.
- Все другие потоки остаются заблокированными до тех пор, пока блокировка не будет снята.
- А снята она будет лишь при выходе из этого блока.

Синхронизация



- Когда выполнение доходит до оператора `lock`, объект-заглушка блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток.
- **Пример 7:**
Использование блокировки с помощью оператора `lock`

Блокировка с помощью lock

```
class Program
```

```
{
```

```
    static int x=0; //общая переменная
```

```
    static object locker = new object(); //объект-заглушка
```

```
    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);
```

```
            }
```

```
        }
```

```
    }
```

```
    static void Main(string[] args)
```

```
    {
```

```
        for (int i = 0; i < 5; i++)
```

```
        {
```

```
            Thread myThread = new Thread(Count);
```

```
            myThread.Name = "Поток " + i.ToString();
```

```
            myThread.Start();
```

```
        }
```

```
    }
```

```
}
```

```
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
```

Класс Monitor

- Класс **Monitor** предоставляет средства для синхронизации потоков.
- В классе Monitor определено несколько методов синхронизации:
 - `public static void Enter(object syncOb);` - предоставляет возможность блокировки для объекта, syncOb.
 - `public static void Exit(object syncOb);` – снимает блокировку для объекта, syncOb.
 - `public static bool TryEnter(object syncOb);`— 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.
- Если при вызове метода **Enter()** заданный объект недоступен, вызывающий поток будет ожидать до тех пор, пока объект не станет доступным.

Класс Monitor

- В классе Monitor также определены методы Wait(), Pulse() и PulseAll().
- **Форматы использования методов:**
 - `public static bool Wait(object waitOb)` - ожидание до уведомления;
 - `public static bool Wait(object waitOb, int milliseconds)` - ожидание до уведомления или до истечения периода времени, заданного в миллисекундах;
 - `public static void Pulse(object waitOb)` возобновляет выполнение потока;
- 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 получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки.
- Если метод Monitor.Pulse отправлен, поток, находящийся во главе очереди ожидания, получает сигнал и блокирует освободившийся объект.

Класс Monitor

- Пример 8: Использование блокировки с помощью монитора

Блокировка с помощью класса Monitor

```
class Program
{
    static int x=0; //общая переменная
    static object locker = new object(); //объект-заглушка
    public static void Count()
    {
        Monitor.Enter(locker); //монитор блокирует заглушку

        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
        Monitor.Exit(locker); //заглушка становится доступной для других потоков
    }

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

Класс Mutex

- Класс Mutex (mutual exclusion — взаимное исключение или мьютекс) позволяет обеспечить синхронизацию среди множества процессов.
 - Класс Mutex является классом-оболочкой над соответствующим объектом ОС Windows.
1. Создаем объект мьютекса: `Mutex mutexObj = new Mutex();`
 2. Метод `mutexObj.WaitOne()` приостанавливает выполнение потока до тех пор, пока не будет получен мьютекс `mutexObj`.
 3. После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода `mutexObj.ReleaseMutex()`.
 4. Таким образом, когда выполнение дойдет до вызова `mutexObj.WaitOne()`, поток будет ожидать, пока не освободится мьютекс. И после его получения продолжит выполнять свою работу.
- **Мьютексы могут также применяться не только внутри одного процесса, но и между процессами.**

Класс Mutex

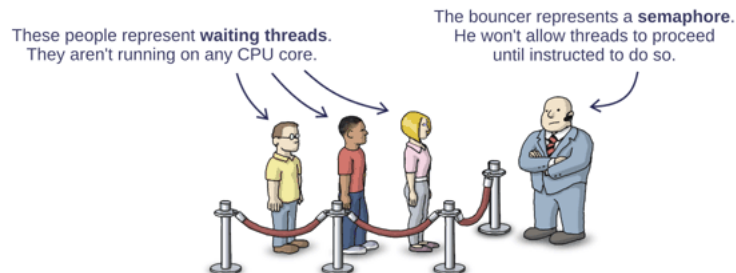
- **Пример 9:** Использование блокировки с помощью мьютекса
- Для создания мьютекса мы используем другую перегрузку конструктора: `Mutex mutexObj = new Mutex(true, guid, out existed);`
- Первый параметр: `true` - указывает, что приложение будет запрашивать владение мьютексом.
- Второй параметр: `guid` - указывает на уникальное имя мьютекса. В данном случае в качестве имени выбран `guid` приложения, то есть глобальный универсальный идентификатор.
- Третий параметр: `out existed` - возвращает значение из конструктора. Если он равен `true`, то это означает, что мьютекс запрошен и получен. А если `false` - то запрос на владение мьютексом отклонен.

Блокировка с помощью класса Mutex

```
class Program
{
    static int x = 0; //общая переменная
    static Mutex mutexObj = new Mutex(); //мьютекс
    public static void Count()
    {
        mutexObj.WaitOne(); //текущий поток переводится в состояние ожидания,
        x = 1;
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine($"{ Thread.CurrentThread.Name}: { x}");
            x++;
            Thread.Sleep(100);
        }
        mutexObj.ReleaseMutex(); //поток переводится в сигнальное состояние
    }
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }
    }
}
```

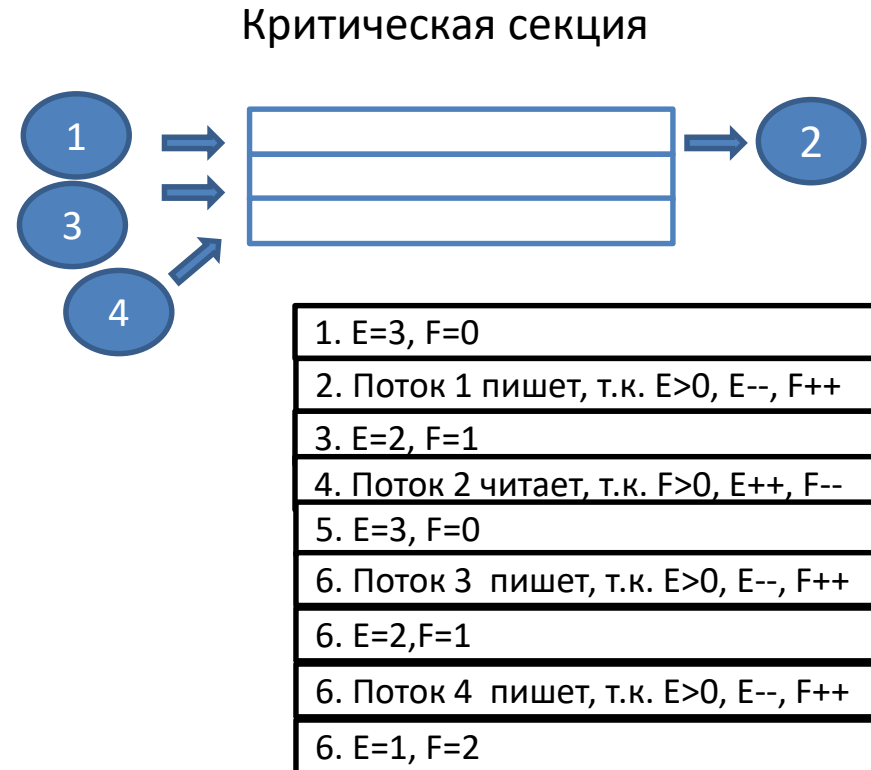
Класс Semaphore

- **Семафор** предоставляет одновременный доступ к общему ресурсу не одному, а нескольким потокам, т.е. семафор пригоден для синхронизации целого ряда ресурсов. Для доступа к ресурсу поток должен получить разрешение от семафора
- Семафор управляет доступом к общему ресурсу, используя для этой цели **счетчик**. Если значение счетчика больше нуля, то доступ к ресурсу разрешен. А если это значение равно нулю, то доступ к ресурсу запрещен.
- С помощью счетчика ведется подсчет количества разрешений.
- Если создать семафор, одновременно разрешающий только один доступ, то такой семафор будет действовать как мьютекс.



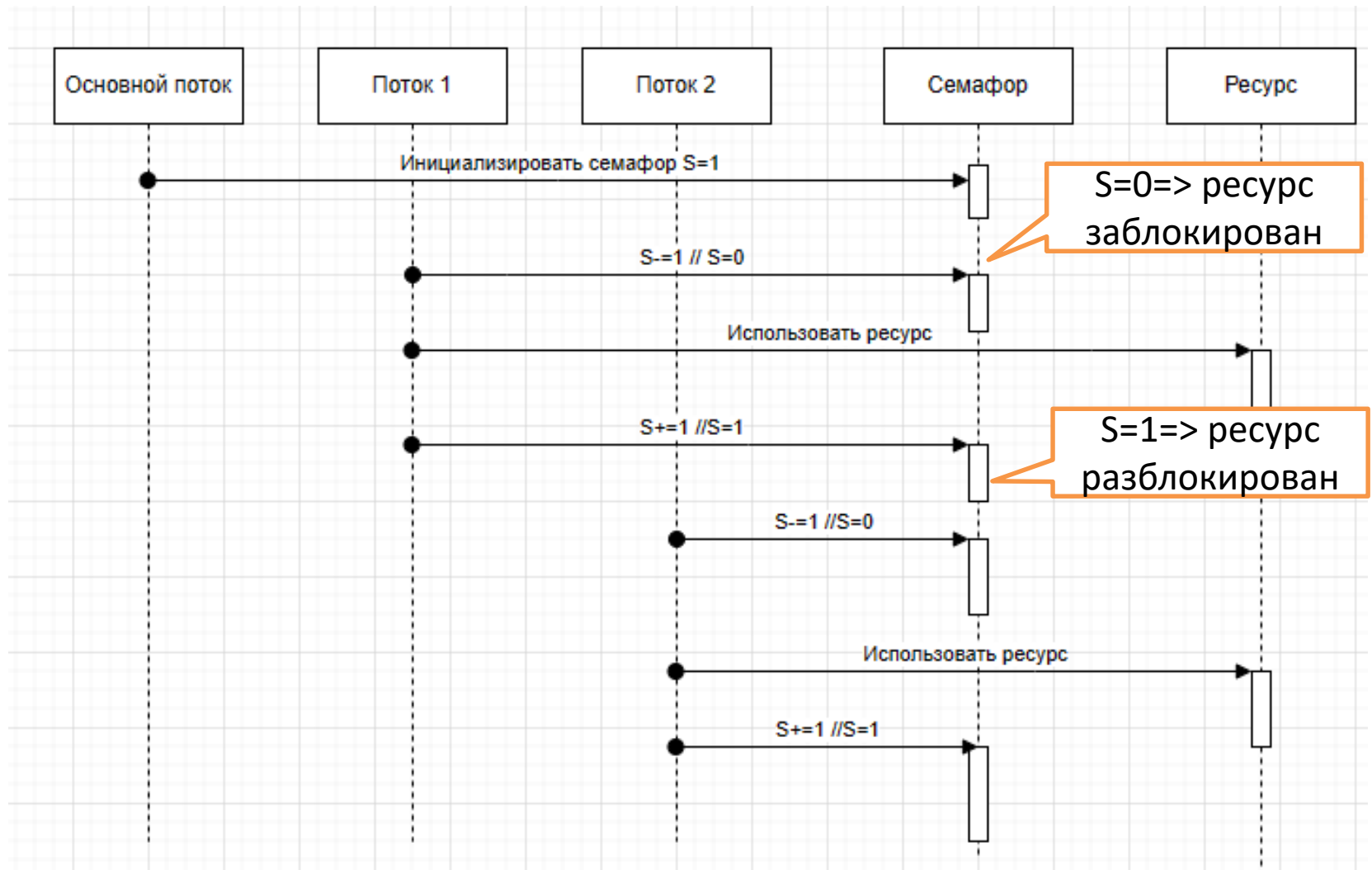
Класс Semaphore

- Для работы с семафорами используются 2 примитива:
- $V(S)$: $S++$ – переменная S (семафор) увеличивается на 1, во время выполнения этой операции другим потокам доступа к переменной S нет.
- $P(S)$: $S--$ – переменная S (семафор) уменьшается на 1, если это возможно. Если $S=0$, то поток, вызывающий операцию P ждет, когда это уменьшения станет возможным.
- Поток-писатель прежде всего выполняет операцию $P(e)$ и проверяет, есть ли пустые буферы. Если $e==0$, то он переходит в состояние ожидания. Если $e>0$, то он уменьшает число свободных буферов и записывает данные в очередной свободный буфер. После чего увеличивает значение f операцией $V(f)$.
- Поток-читатель действует аналогично, но начинает работу с проверки занятых буферов.
- Критическим ресурсом здесь является буферный пул, с которым могут работать столько потоков, сколько буферов в нем содержится.



Если $E=0$, то писать нельзя, Если $F=0$, то читать нельзя

Семафор



Класс Semaphore

- **Пример 10:** Есть некоторое число читателей, которые приходят в библиотеку три раза в день. Ограничение: одновременно в библиотеке не может находиться больше трех читателей.

Синхронизация с помощью семафора

```
class Reader
{
    static Semaphore sem = 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)
        {
            sem.WaitOne(); // ожидания получения семафора
            Console.WriteLine($"{Thread.CurrentThread.Name} входит в библиотеку");
            Console.WriteLine($"{Thread.CurrentThread.Name} читает");
            Thread.Sleep(1000);
            Console.WriteLine($"{Thread.CurrentThread.Name} покидает библиотеку");
            sem.Release(); // высвобождаем семафор
            count--;
            Thread.Sleep(1000);
        }
    }
}
```

какому числу потоков изначально
будет доступен семафор;
какое максимальное число
потоков будет использовать
данный семафор

Синхронизация с помощью семафора

5 читателей

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

        Console.ReadLine();
    }
}
```

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

Простейшие методы блокировки

Конструкция	Назначение	Доступность из других процессов	Скорость
lock/Monitor	Гарантирует, что только один поток может получить доступ к ресурсу или секции кода.	нет	быстро
Mutex	Гарантирует, что только один поток может получить доступ к ресурсу или секции кода.	да	средне
Semaphor	Гарантирует, что не более заданного числа потоков может получить доступ к ресурсу или секции кода.	да	средне

Класс Task

- Задача – это отдельная продолжительная операция (абстракция асинхронной операции).
- Библиотека параллельных задач TPL (Task Parallel Library) располагается в пространстве имен `System.Threading.Tasks`.
- Класс `Task` описывает отдельную задачу, которая запускается асинхронно в одном из потоков (обычно в `Main`).

Способы создания объектов Task

1.

```
Console.WriteLine("Hello World!")  
Task task1 = new Task(() =>
```

Параметр –
делегат Action

2.

```
    Console.WriteLine("first task")  
task1.Start(); // запуск задачи
```

Используется
статический
метод StartNew()

3.

```
Task task2 = Task.Factory.StartNew(() =>  
    Console.WriteLine("second task"));  
Task task3 = Task.Run(() =>  
    Console.WriteLine("third task"));
```

Используется
статический
метод Run()

По умолчанию задачи запускаются асинхронно

Основные свойства класса Task

- AsyncState: возвращает объект состояния задачи
- CurrentId: возвращает идентификатор текущей задачи (статическое свойство)
- Id: возвращает идентификатор текущей задачи
- Exception: возвращает объект исключения, возникшего при выполнении задачи
- Status: возвращает статус задачи.
- IsCompleted: возвращает true, если задача завершена
- IsCanceled: возвращает true, если задача была отменена
- IsFaulted: возвращает true, если задача завершилась при возникновении исключения
- IsCompletedSuccessfully: возвращает true, если задача завершилась успешно

Пример

```
static void MyTask()
{
    Console.WriteLine($"MyTask {Task.CurrentId} запущен");
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(500);
        Console.WriteLine($"В методе MyTask {Task.CurrentId} счетчик = {i}");
    }
    Console.WriteLine($"MyTask {Task.CurrentId} завершен");
}

static void Main(string[] args)
{
    Console.WriteLine("Hello, world!");
    Task t1=new Task(MyTask);
    Task t2=new Task(MyTask);
    t1.Start();
    t2.Start();
    Console.WriteLine("The end of Main()");
}
```

Чтобы приложение ожидало завершения задачи, можно использовать метод **Wait()** объекта Task

```
Hello, world!
The end of Main()
MyTask 1 запущен
MyTask 2 запущен
Для продолжения нажмите любую клавишу . . . |
```

Пример

```
Task task1 = new Task(MyTask);
Task task2 = new Task(MyTask);
task1.Start();
task2.Start();
//приостановка Main до завершения об
task1.Wait();
task2.Wait();
Console.WriteLine("The end of Main")
```

Или так

```
task1.Start();
//приостановка Main до завершения обеих за
Task.WaitAll(task1, task2);
```

Или так

```
Task.WaitAny(task1, task2);
```

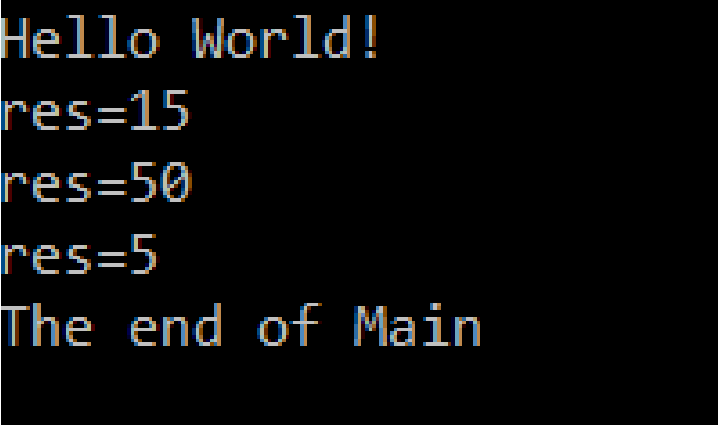
```
Hello World!
MyTask 1 запущен
MyTask 2 запущен
В методе MyTask 2 счетчик равен 0
В методе MyTask 1 счетчик равен 0
В методе MyTask 2 счетчик равен 1
В методе MyTask 1 счетчик равен 1
В методе MyTask 2 счетчик равен 2
В методе MyTask 1 счетчик равен 2
В методе MyTask 2 счетчик равен 3
В методе MyTask 1 счетчик равен 3
В методе MyTask 2 счетчик равен 4
В методе MyTask 1 счетчик равен 4
В методе MyTask 2 счетчик равен 5
В методе MyTask 1 счетчик равен 5
В методе MyTask 2 счетчик равен 6
В методе MyTask 1 счетчик равен 6
В методе MyTask 2 счетчик равен 7
В методе MyTask 1 счетчик равен 7
В методе MyTask 2 счетчик равен 8
В методе MyTask 1 счетчик равен 8
В методе MyTask 2 счетчик равен 9
MyTask 2 завершен
В методе MyTask 1 счетчик равен 9
MyTask 1 завершен
The end of Main
```

Возвращение результатов из задач

- Используется обобщенный класс Task<T>

```
static int Sum(int a, int b) => a + b;
static int Mult(int a, int b) => a * b;
static int Sub(int a, int b) => a - b;

static void Main(string[] args)
{
    int x1 = 10, x2 = 5;
    Console.WriteLine("Hello World!");
    Task<int>[] tasks = new Task<int>[3];
    tasks[0] = new Task<int>(() => Sum(x1, x2));
    tasks[1] = new Task<int>(() => Mult(x1, x2));
    tasks[2] = new Task<int>(() => Sub(x1, x2));
    foreach (var item in tasks)
    {
        item.Start();
        Console.WriteLine($"res={ item.Result}");
    }
    Console.WriteLine("The end of Main");
}
```



```
Hello World!
res=15
res=50
res=5
The end of Main
```

Параллельное выполнение задач

```
static void Sum(int a, int b)
{
    Console.WriteLine($"выполняется задача {Task.CurrentId}");
    Thread.Sleep(1000);
    Console.WriteLine($"Результат={ a + b}");
}

static void Print()
{
    Console.WriteLine($"выполняется задача {Task.CurrentId}");
    Thread.Sleep(1000);
}

static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    Stopwatch sw = new Stopwatch();
    sw.Start();
    Parallel.Invoke(Print, Print, ()=>Sum(2,5));
    sw.Stop();
    Console.WriteLine($"Время выполнения {sw.ElapsedMilliseconds}");
    Console.WriteLine("The end of Main");
}
```

```
Hello World!
выполняется задача 1
выполняется задача 3
выполняется задача 2
Результат=7
Время выполнения 1025
The end of Main
```

Параллельное выполнение задач

```
static void Pow(int n)
{
    Console.WriteLine($"выполняется задача {Task.CurrentId}");
    Console.WriteLine($"2^{n}={Math.Pow(2,n)}");
    Thread.Sleep(1000);
}

static void Main(string[] args)
{
    invoke

    #region For
    sw.Restart();
    Parallel.For(1, 5, Pow);
    sw.Stop();
    Console.WriteLine($"Время выполнения {sw.ElapsedMilliseconds}");
    #endregion

    Console.WriteLine("The end of Main");
}
```

```
выполняется задача 6
выполняется задача 5
выполняется задача 7
выполняется задача 4
2^4=16
2^3=8
2^1=2
2^2=4
Время выполнения 1012
The end of Main
```


Параллельное выполнение задач

```
List<int> power = new List<int> { 1, 2, 4, 6, 8, 10 };  
sw.Restart();  
ParallelLoopResult res = Parallel.ForEach<int>(power, Pow);  
sw.Stop();  
Console.WriteLine($"Время выполнения {sw.ElapsedMilliseconds}");  
#endregion  
Console.WriteLine("The end of Main");
```

```
выполняется задача 8  
выполняется задача 9  
2^1=2  
выполняется задача 12  
2^10=1024  
выполняется задача 10  
2^8=256  
выполняется задача 13  
2^6=64  
выполняется задача 11  
2^2=4  
2^4=16  
Время выполнения 1019  
The end of Main
```

Асинхронное программирование

- Программа может выполнять такие операции, которые могут занять продолжительное время, например, обращение к сетевым ресурсам, чтение-запись файлов, обращение к базе данных и т.д.
- **Асинхронность** позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы и при этом более экономно использовать потоки.
- Асинхронные методы выполняются в отдельных потоках. При выполнении продолжительной операции поток асинхронного метода возвратится в пул потоков и будет использоваться для других задач.
- Когда продолжительная операция завершит свое выполнение, для асинхронного метода опять выделяется поток из пула потоков, и асинхронный метод продолжает свою работу.

Асинхронное программирование

- В заголовке метода используется модификатор `async`
- Метод содержит одно или несколько выражений `await`
- Метод возвращает:
 - `void`
 - `Task`
 - `Task<T>`
 - `ValueTask<T>`
- Асинхронный метод не может определять параметры с модификаторами `out`, `ref` и `in`.

Асинхронное программирование

Суффикс Async

```
static async Task PrintNameAsync(string name)
{
    await Task.Delay(3000);
    Console.WriteLine(name);
}
static async Task Main(string[] args)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    await PrintNameAsync("Ann");
    await PrintNameAsync("Jane");
    await PrintNameAsync("Kate");
    sw.Stop();
    Console.WriteLine($"Время выполнения {sw.ElapsedMilliseconds}");
}
```

Ann

Jane

Kate

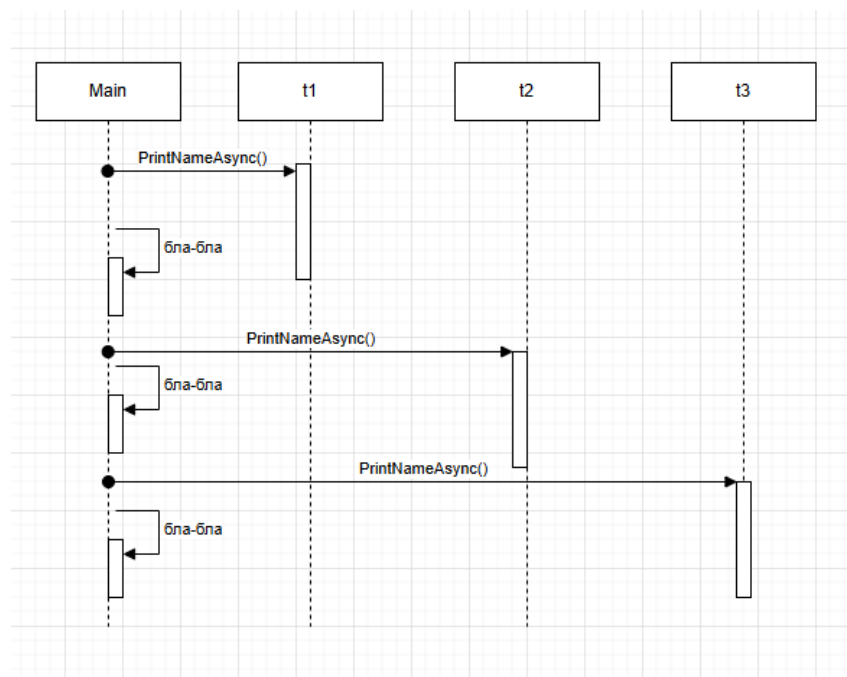
Время выполнения 9051
The end of Main

Задержка 3с при
выводе каждого
имени

Асинхронное программирование

```
static async Task PrintNameAsync(string name)
{
    await Task.Delay(1000);
    Console.WriteLine(name);
}

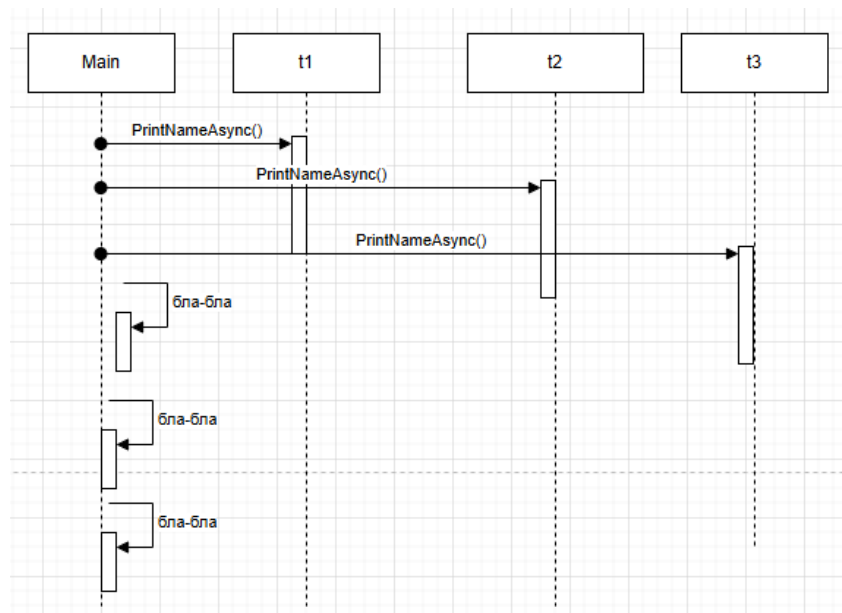
static async Task Main(string[] args)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    await PrintNameAsync("Viktor");
    Console.WriteLine("bla-bla");
    await PrintNameAsync("Mickle");
    Console.WriteLine("bla-bla");
    await PrintNameAsync("Robert");
    sw.Stop();
    Console.WriteLine($"{sw.ElapsedMilliseconds}"); //3004
}
```



Асинхронное программирование

```
static async Task Main(string[] args)
{
    Stopwatch sw = new Stopwatch();
    var viktor = PrintNameAsync("Viktor");
    var mickle = PrintNameAsync("Mickle");
    var robert = PrintNameAsync("Robert");
    sw.Start();
    await viktor;
    Console.WriteLine("bla-bla");
    await mickle;
    Console.WriteLine("bla-bla");
    await robert;
    Console.WriteLine("bla-bla");
    sw.Stop();
    Console.WriteLine($"{sw.ElapsedMilliseconds}"); //1015
}
```

Задачи запускаются
при определении



Возвращение результата

Нет return, т.к.
возвращаем Task

```
static async Task PrintNameAsync(string name)
{
    await Task.Delay(3000);
    Console.WriteLine(name);
}
static async Task Main(string[] args)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    await PrintNameAsync("Ann");
    await PrintNameAsync("Jane");
    await PrintNameAsync("Kate");
    sw.Stop();
    Console.WriteLine($"Время выполнения {sw.ElapsedMilliseconds}");
}
```

Возвращение результата

```
static async Task<int> SumAsync(int a, int b)|...|
static async Task<int> MultAsync(int a, int b)|...|
static async Task<int> SubAsync(int a, int b)
{
    await Task.Delay(0);
    var result = a - b;
    Console.WriteLine($"a={a}, b={b}, a-b={result}");
    return result;
}
```

```
static async Task Main(string[] args)
{
    Console.WriteLine("Hello World!");
    var sum = SumAsync(10, 5);
    var mult = MultAsync(10, 5);
    var sub = SubAsync(10, 5);..
    Console.WriteLine("Остальные действия в Main");
    Thread.Sleep(1000);
    int r1 = await sum;
    int r2 = await mult;
    int r3 = await sub;
    Console.WriteLine($"r1={r1}, r2={r2}, r3={r3}");
}
```

Hello World!

a=10, b=5, a+b=15

a=10, b=5, a*b=50

a=10, b=5, a-b=5

Остальные действия в Main

r1=15, r2=50, r3=5

Возвращение результата

```
static async ValueTask PrintEmployeeAsync(Employee e)
{
    await Task.Delay(0);
    Console.WriteLine(e);
}
static async ValueTask<Employee>MakeEmployeeAsync(string name)
{
    await Task.Delay(0);
    Console.WriteLine("Make employee");
    return new Employee(name);
}
static async Task Main(string[] args)
{
    Console.WriteLine("Hello World!");
    Employee e = await MakeEmployeeAsync("Иванов");
    await PrintEmployeeAsync(e);
    Console.WriteLine("The end of Main");
}
```

Пример

- Найти сумму элементов строк двумерного массива. Каждая строка массива обрабатывается в отдельном потоке.
- 1) использовать Thread
- 2) использовать Task
- 3) использовать параллельные процессы
- Матрицу представим в виде
- `List<List<int>> collection`

Использовать Thread

```
static int CalcSum(List<int>row, int number)
{
    Console.WriteLine($"запуск вычислений в строки {number}");
    int sum = 0;
    foreach (var item in row)
    {
        sum += item;
    }
    return sum;
}
```

Вычисление суммы строки с номером number

```

#region Thread
List<(Thread, Action<int, int>)> threadActions =
    new List<(Thread, Action<int, int>)>(); // список потоков и результатов
for (int i = 0; i < list.Count; i++)
{
    int index = i;
    // Делегат для сохранения результата
    Action<int, int> action = (index, result) =>
    { Console.WriteLine($"сумма {index} равна {result} "); };
    // Создаем поток
    Thread thread = new Thread(() =>
    {
        int result = CalcSum(list[index], index);
        action(index, result);
    });
    threadActions.Add((thread, action));
    thread.Start();
}
// Ждем завершения всех созданных потоков
foreach ((Thread thread, _) in threadActions)
{
    if (thread.IsAlive)
    {
        thread.Join(); // Ожидаем завершения потока
    }
}

#endregion

```

Запомнить `i` в отдельной переменной, т.к. `i` может измениться после запуска функции, но до ее фактического выполнения

Дискард-символ, вместо результата вычислений

Использовать Task

```
static int CalcSum(List<int>row, int number)
{
    Console.WriteLine($"запуск вычислений в строки {number}");
    int sum = 0;
    foreach (var item in row)
    {
        sum += item;
    }
    return sum;
}
```

Вычисление суммы строки с номером number

Использовать Task

```
#region Task
Task<int>[] tasks = new Task<int>[list.Count]; // массив для результатов задач
for (int i = 0; i < list.Count; i++)
{
    int index = i;
    tasks[i] = Task.Run(() => CalcSum(list[index], index));
}
Task.WaitAll(tasks);
foreach (var item in tasks)
{
    Console.WriteLine(item.Result);
}
```

```
60 80 22 53 96 50 49 78 49 33 54
38
14
запуск вычислений для строки 1
запуск вычислений для строки 0
запуск вычислений для строки 2
950
38
14
```

Запомнить *i* в отдельной переменной, т.к. *i* может измениться после запуска функции, но до ее фактического выполнения

Использовать параллельные процессы

```
static int CalcSum(List<int> row)
{
    int sum = 0;
    foreach (var item in row)
    {
        sum += item;
    }
    return sum;
}
```



Вычисление суммы строки

```

#region Parallel
int totalSum = 0;
// блокировка для безопасной работы с общей переменной
object lockObject = new object();
Parallel.ForEach(list, (row, parallelLoopState, index) =>
{
    int sum = CalcSum(row);
    // Синхронизированный доступ к общей сумме
    lock (lockObject)
    {
        totalSum += sum;
    }
    Console.WriteLine($"Сумма строки {index} = {sum}");
});
Console.WriteLine(totalSum);
#endregion

```

```

20  63  86  65   3   0  86  16  10   6  92   9  37   0
14  21  91  79  41  50  81  46  21  59   5
45  95   2  31  49  45  72  76  65  97  76  94  17  25  67  62  88  32  93
Сумма строки 2 = 1131
Сумма строки 1 = 508
Сумма строки 0 = 493
2132

```