Многопоточность

Введение в многопоточность. Класс Thread

Одним из ключевых аспектов в современном программировании является многопоточность. Ключевым понятием при работе с многоопоточностью является поток. Поток предствляет некоторую часть кода программы. При выполнении программы каждому потоку выделяется определенный квант времени. И при помощи многопоточности мы можем выделить в приложении несколько потоков, которые будут выполнять различные задачи одновременно. Если у нас, допустим, графическое приложение, которое посылает запрос к какому-нибудь серверу или считывает и обрабатывает огромный файл, то без многопоточности у нас бы блокировался графический интерфейс на время выполнения задачи. А благодаря потокам мы можем выделить отправку запроса или любую другую задачу, которая может долго обрабатываться, в отдельный поток. Поэтому, к примеру, клиент-серверные приложения (и не только они) практически не мыслимы без многопоточности.

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

Класс Thread определяет ряд методов и свойств, которые позволяют управлять потоком и получать информацию о нем. Основные свойства класса:

- ExecutionContext: позволяет получить контекст, в котором выполняется поток
- IsAlive: указывает, работает ли поток в текущий момент
- IsBackground: указывает, является ли поток фоновым
- Name: содержит имя потока
- ManagedThreadId: возвращает числовой идентификатор текущего потока
- Priority: хранит приоритет потока - значение перечисления ThreadPriority:
    - Lowest
    - BelowNormal
    - Normal
    - AboveNormal
    - Highest

По умолчанию потоку задается значение Normal. Однако мы можем изменить приоритет в процессе работы программы. Например, повысить важность потока, установив приоритет Highest. Среда CLR будет считывать и анализировать значения приоритета и на их основании выделять данному потоку то или иное количество времени.

- 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

  В процессе работы потока его статус многократно может измениться под действием методов. Так, в самом начале еще до применения метода Start его статус имеет значение Unstarted. Запустив поток, мы изменим его статус на Running. Вызвав метод Sleep, статус изменится на WaitSleepJoin.

Кроме того статическое свойство CurrentThread класса Thread позволяет получить текущий поток

В программе на C# есть как минимум один поток - главный поток, в котором выполняется метод Main.

Например, используем вышеописанные свойства для получения информации о потоке:

```
using System.Threading;

// получаем текущий поток
Thread currentThread = Thread.CurrentThread;

//получаем имя потока
Console.WriteLine($"Имя потока: {currentThread.Name}");
currentThread.Name = "Метод Main";
Console.WriteLine($"Имя потока: {currentThread.Name}");

Console.WriteLine($"Запущен ли поток: {currentThread.IsAlive}");
Console.WriteLine($"Id потока: {currentThread.ManagedThreadId}");
Console.WriteLine($"Приоритет потока: {currentThread.Priority}");
Console.WriteLine($"Статус потока: {currentThread.ThreadState}");
```

В этом случае мы получим примерно следующий вывод:

```
Имя потока:
Имя потока: Метод Main
Запущен ли поток: True
Id потока: 1
Приоритет потока: Normal
Статус потока: Running
```

Так как по умолчанию свойство Name у объектов Thread не установлено, то в первом случае мы получаем в качестве значения этого свойства пустую строку.

Также класс Thread определяет ряд методов для управления потоком. Основные из них:

- Статический метод GetDomain возвращает ссылку на домен приложения
- Статический метод GetDomainID возвращает id домена приложения, в котором выполняется текущий поток
- Статический метод Sleep останавливает поток на определенное количество миллисекунд
- Метод Interrupt прерывает поток, который находится в состоянии WaitSleepJoin
- Метод Join блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- Метод Start запускает поток

Например, применим метод Sleep для задания задержки выполнения приложения:

```
using System.Threading;

for(int i = 0; i < 10; i++)
{
    Thread.Sleep(500);      // задержка выполнения на 500 миллисекунд
    Console.WriteLine(i);
}
```

Создание потоков. Делегат ThreadStart

Язык C# позволяет запускать и выполнять в рамках приложения несколько потоков, которые будут выполняться одновременно.

Для создания потока применяется один из конструкторов класса Thread:

- Thread(ThreadStart): в качестве параметра принимает объект делегата ThreadStart, который представляет выполняемое в потоке действие
- Thread(ThreadStart, Int32): в дополнение к делегату ThreadStart принимает числовое значение, которое устанавливает размер стека, выделяемого под данный поток
- Thread(ParameterizedThreadStart): в качестве параметра принимает объект делегата ParameterizedThreadStart, который представляет выполняемое в потоке действие
- Thread(ParameterizedThreadStart, Int32): вместе с делегатом ParameterizedThreadStart принимает числовое значение, которое устанавливает размер стека для данного потока

Вне зависимости от того, какой конструктор будет применяться для создания, нам надо определить выполняемое в потоке действие. В этой статье рассмотрим использование делегата ThreadStart. Этот делегат представляет действие, которое не принимает никаких параметров и не возвращает никакого значения:

```
public delegate void ThreadStart();
```

То есть под этот делегат нам надо определить метод, который имеет тип void и не принимает никаких параметров. Примеры определения потоков:

```
Thread myThread1 = new Thread(Print);
Thread myThread2 = new Thread(new ThreadStart(Print));
Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));

void Print() => Console.WriteLine("Hello Threads");
```

Для запуска нового потока применяется метод Start класса Thread:

```
using System.Threading;

// создаем новый поток
Thread myThread1 = new Thread(Print);
Thread myThread2 = new Thread(new ThreadStart(Print));
Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));

myThread1.Start();  // запускаем поток myThread1
myThread2.Start();  // запускаем поток myThread2
myThread3.Start();  // запускаем поток myThread3

void Print() => Console.WriteLine("Hello Threads");
```

Преимуществом потоком является то, что они могут выполняться одновременно. Например:

```
using System.Threading;

// создаем новый поток
Thread myThread = new Thread(Print);
// запускаем поток myThread
myThread.Start();

// действия, выполняемые в главном потоке
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Главный поток: {i}");
    Thread.Sleep(300);
}

// действия, выполняемые во втором потокке
void Print()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine($"Второй поток: {i}");
        Thread.Sleep(400);
```

```
    }
}
```

Здесь новый поток будет производить действия, определенные в методе Print, то есть выводить числа от 0 до 4 на консоль. Причем после каждого вывода производится задержка на 400 миллисекунд.

В главном потоке - в методе Main создаем и запускаем новый поток, в котором выполняется метод Print:

```
Thread myThread = new Thread(Print);
myThread.Start();
```

Кроме того, в главном потоке производим аналогичные действия - выводим на консоль числа от 0 до 4 с задержкой в 300 миллисекунд.

Таким образом, в нашей программе будут работать одновременно главный поток, представленный методом Main, и второй поток, в котором выполняется метод Print. Как только все потоки отработают, программа завершит свое выполнение. В итоге мы получим следующий консольный вывод:

```
Главный поток: 0
Второй поток: 0
Главный поток: 1
Второй поток: 1
Главный поток: 2
Второй поток: 2
Главный поток: 3
Второй поток: 3
Главный поток: 4
Второй поток: 4
```

Подобным образом мы можем создать и запускать и три, и четыре, и целый набор новых потоков, которые смогут решать те или иные задачи.

Потоки с параметрами и ParameterizedThreadStart

В предыдущей статье было рассмотрено, как запускать в отдельных потоках методы без параметров. А что, если нам надо передать какие-нибудь параметры в поток?

Для этой цели используется делегат ParameterizedThreadStart, который передается в конструктор класса Thread:

```
public delegate void ParameterizedThreadStart(object? obj);
```

Применение делегата ParameterizedThreadStart во многом похоже на работу с ThreadStart. Рассмотрим на примере:

```
using System.Threading;

// создаем новые потоки
Thread myThread1 = new Thread(new ParameterizedThreadStart(Print));
Thread myThread2 = new Thread(Print);
Thread myThread3 = new Thread(message => Console.WriteLine(message));

// запускаем потоки
myThread1.Start("Hello");
myThread2.Start("Привет");
myThread3.Start("Salut");


void Print(object? message) => Console.WriteLine(message);
```

При создании потока в конструктор класса Thread передается объект делегата ParameterizedThreadStart new Thread(new ParameterizedThreadStart(Print)), либо непосредственно метод, который соответствует этому делегату (new Thread(Print)), в том числе в виде лямбда-выражения (new Thread(message => Console.WriteLine(message)))

Затем при запуске потока в метод Start() передается значение, которое передается параметру метода Print. И в данном случае мы получим следующий консольный вывод:

```
Salut
Hello
Привет
```

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 если мы хотим использовать данные других типов, в самом методе необходимо выполнить приведение типов. Например:

```
using System.Threading;

int number = 4;
// создаем новый поток
Thread myThread = new Thread(Print);
myThread.Start(number);     // n * n = 16


// действия, выполняемые во втором потокке
void Print(object? obj)
{
    // здесь мы ожидаем получить число
```

```
    if (obj is int n)
    {
        Console.WriteLine($"n * n = {n * n}");
    }
}
```

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 можно определить свои типы:

```
using System.Threading;

Person tom = new Person("Tom", 37);
// создаем новый поток
Thread myThread = new Thread(Print);
myThread.Start(tom);

void Print(object? obj)
{
    // здесь мы ожидаем получить объект Person
    if (obj is Person person)
    {
        Console.WriteLine($"Name = {person.Name}");
        Console.WriteLine($"Age = {person.Age}");
    }
}

record class Person(string Name, int Age);
```

Сначала определяем специальный класс Person, объект которого будет передаваться во второй поток, а в методе Main передаем его во второй поток.

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. Например:

```
using System.Threading;

Person tom = new Person("Tom", 37);
// создаем новый поток
Thread myThread = new Thread(tom.Print);
myThread.Start();

record class Person(string Name, int Age)
```

```
{
    public void Print()
    {
        Console.WriteLine($"Name = {Name}");
        Console.WriteLine($"Age = {Age}");
    }
}
```

Синхронизация потоков

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

```
int x = 0;

// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";   // устанавливаем имя для каждого потока
    myThread.Start();
}

void Print()
{
    x = 1;
    for (int i = 1; i < 6; i++)
    {
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
        x++;
        Thread.Sleep(100);
    }
}
```

Здесь у нас запускаются пять потоков, которые вызывают метод Print и которые работают с общей переменной x. И мы предполагаем, что метод выведет все значения x от 1 до 5. И так для каждого потока. Однако в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым. Например, в моем случае я получил следующий консольный вывод (он может в каждом конкретном случае различаться):

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

```
Поток 3: 7
Поток 2: 7
Поток 4: 9
Поток 1: 11
Поток 4: 11
Поток 2: 11
Поток 3: 14
Поток 5: 11
Поток 1: 16
Поток 2: 16
Поток 3: 16
Поток 5: 18
Поток 4: 16
Поток 1: 21
Поток 5: 21
Поток 3: 21
Поток 2: 21
Поток 4: 21
```

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. Остальный потоки помещаются в очередь ожидания и ждут, пока текущий поток не освободит данный блок кода. В итоге с помощью lock мы можем переделать предыдущий пример следующим образом:

```
int x = 0;
object locker = new();  // объект-заглушка
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}


void Print()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 6; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
}
```

Для блокировки с ключевым словом lock используется объект-заглушка, в данном случае это переменная locker. Обычно это переменная типа object. И 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.

В этом случае консольный вывод будет более упорядоченным:

```
Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 5: 1
Поток 5: 2
Поток 5: 3
Поток 5: 4
Поток 5: 5
Поток 3: 1
Поток 3: 2
Поток 3: 3
Поток 3: 4
Поток 3: 5
Поток 2: 1
Поток 2: 2
Поток 2: 3
Поток 2: 4
Поток 2: 5
Поток 4: 1
Поток 4: 2
Поток 4: 3
Поток 4: 4
Поток 4: 5
```

Мониторы

Наряду с оператором lock для синхронизации потоков мы можем использовать мониторы, представленные классом System.Threading.Monitor. Для управления синхронизацией этот класс предоставляет следующие методы:

- void Enter(object obj): получает в эксклюзивное владение объект, передаваемый в качестве параметра.
- void Enter(object obj, bool acquiredLock): дополнительно принимает второй параметра - логическое значение, которое указывает, получено ли владение над объектом из первого параметра
- void Exit(object obj): освобождает ранее захваченный объект
- bool IsEntered(object obj): возвращает true, если монитор захватил объект obj

- void Pulse (object obj): уведомляет поток из очереди ожидания, что текущий поток освободил объект obj
- void PulseAll(object obj): уведомляет все потоки из очереди ожидания, что текущий поток освободил объект obj. После чего один из потоков из очереди ожидания захватывает объект obj.
- bool TryEnter (object obj): пытается захватить объект obj. Если владение над объектом успешно получено, то возвращается значение true
- bool Wait (object obj): 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.

Стоит отметить, что фактически конструкция оператора lock инкапсулирует в себе синтаксис использования мониторов. Например, в прошлой теме для синхронизации потоков применялся оператор lock:

```
int x = 0;
object locker = new();  // объект-заглушка
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}


void Print()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 6; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
}
```

Фактически данный пример будет эквивалентен следующему коду:

```
int x = 0;
object locker = new();  // объект-заглушка
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
```

```
    myThread.Start();
}

void Print()
{
    bool acquiredLock = false;
    try
    {
        Monitor.Enter(locker, ref acquiredLock);
        x = 1;
        for (int i = 1; i < 6; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
    finally
    {
        if (acquiredLock) Monitor.Exit(locker);
    }
}
```

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. А в блоке try...finally с помощью метода Monitor.Exit происходит освобождение объекта locker, если блокировка осуществлена успешно, и он становится доступным для других потоков.

Класс AutoResetEvent

Класс AutoResetEvent также служит целям синхронизации потоков. Этот класс представляет событие синхронизации потоков, который позволяет при получении сигнала переключить данный объект-событие из сигнального в несигнальное состояние.

Для управления синхронизацией класс AutoResetEvent предоставляет ряд методов:

- Reset(): задает несигнальное состояние объекта, блокируя потоки.
- Set();: задает сигнальное состояние объекта, позволяя одному или нескольким ожидающим потокам продолжить работу.
- WaitOne(): задает несигнальное состояние и блокирует текущий поток, пока текущий объект AutoResetEvent не получит сигнал.

Событие синхронизации может находиться в сигнальном и несигнальном состоянии. Если состояние события несигнальное, поток, который вызывает метод WaitOne, будет заблокирован, пока состояние события не станет сигнальным. Метод Set, наоборот, задает сигнальное состояние события.

Так, в одной из предыдущих тем для синхронизации потоков применялся оператор lock:

```
int x = 0;
object locker = new();  // объект-заглушка
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}


void Print()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 6; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
}
```

Перепишем этот пример с использованием AutoResetEvent:

```
int x = 0;  // общий ресурс

AutoResetEvent waitHandler = new AutoResetEvent(true);  // объект-событие

// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}


void Print()
{
    waitHandler.WaitOne();  // ожидаем сигнала
    x = 1;
    for (int i = 1; i < 6; i++)
    {
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
        x++;
        Thread.Sleep(100);
    }
```

```
    waitHandler.Set();  //  сигнализируем, что waitHandler в сигнальном состоянии
}
```

Во-первых, создаем переменную типа AutoResetEvent. Передавая в конструктор значение true, мы тем самым указываем, что создаваемый объект изначально будет в сигнальном состоянии.

Когда начинает работать поток, то первым делом срабатывает определенный в методе Print вызов waitHandler.WaitOne(). Метод WaitOne указывает, что текущий поток переводится в состояние ожидания, пока объект waitHandler не будет переведен в сигнальное состояние. И так все потоки у нас переводятся в состояние ожидания.

После завершения работы вызывается метод waitHandler.Set, который уведомляет все ожидающие потоки, что объект waitHandler снова находится в сигнальном состоянии, и один из потоков "захватывает" данный объект, переводит в несигнальное состояние и выполняет свой код. А остальные потоки снова ожидают.

Так как в конструкторе AutoResetEvent мы указываем, что объект изначально находится в сигнальном состоянии, то первый из очереди потоков захватывает данный объект и начинает выполнять свой код.

Но если бы мы написали AutoResetEvent waitHandler = new AutoResetEvent(false), тогда объект изначально был бы в несигнальном состоянии, а поскольку все потоки блокируются методом waitHandler.WaitOne() до ожидания сигнала, то у нас попросту случилась бы блокировка программы, и программа не выполняла бы никаких действий.

Если у нас в программе используются несколько объектов AutoResetEvent, то мы можем использовать для отслеживания состояния этих объектов статические методы WaitAll и WaitAny, которые в качестве параметра принимают массив объектов класса WaitHandle - базового класса для AutoResetEvent.

Так, мы тоже можем использовать WaitAll в вышеприведенном примере. Для этого надо строку

```
waitHandler.WaitOne();
```

заменить на следующую:

```
AutoResetEvent.WaitAll(new WaitHandle[] {waitHandler});
```

Мьютексы

Еще один инструмент управления синхронизацией потоков представляет класс Mutex или мьютекс, который также располагается в пространстве имен System.Threading.

Так, возьмем пример с оператором lock из одной из предыдущих тем, в котором применялась синхронизация потоков:

```
int x = 0;
object locker = new();  // объект-заглушка
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}


void Print()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 6; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
        }
    }
}
```

И перепишем данный пример, используя мьютексы:

```
int x = 0;
Mutex mutexObj = new();

// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}

void Print()
{
    mutexObj.WaitOne();     // приостанавливаем поток до получения мьютекса
    x = 1;
    for (int i = 1; i < 6; i++)
    {
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
        x++;
        Thread.Sleep(100);
    }
    mutexObj.ReleaseMutex();    // освобождаем мьютекс
}
```

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.

Изначально мьютекс свободен, поэтому его получает один из потоков.

После выполнения всех действий, когда мьютекс больше не нужен, поток освобождает его с помощью метода mutexObj.ReleaseMutex(). А мьютекс получает один из ожидающих потоков.

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

Семафоры

Семафоры являются еще одним инструментом, который предлагает нам платформа .NET для управления синхронизацией. Семафоры позволяют ограничить количество потоков, которые имеют доступ к определенным ресурсам. В .NET семафоры представлены классом Semaphore.

Для создания семафора применяется один из конструкторов класса Semaphore:

- Semaphore (int initialCount, int maximumCount): параметр initialCount задает начальное количество потоков, а maximumCount - максимальное количество потоков, которые имеют доступ к общим ресурсам
- Semaphore (int initialCount, int maximumCount, string? name): в дополнение задает имя семафора
- Semaphore (int initialCount, int maximumCount, string? name, out bool createdNew): последний параметр - createdNew при значении true указывает, что новый семафор был успешно создан. Если этот параметр равен false, то семафор с указанным именем уже существует

Для работы с потоками класс Semaphore имеет два основных метода:

- WaitOne(): ожидает получения свободного места в семафоре
- Release(): освобождает место в семафоре

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 единовременно в библиотеке не может находиться больше трех читателей. Данную задачу очень легко решить с помощью семафоров:

```
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
```

```
        Reader reader = new Reader(i);
    }
    class Reader
    {
        // создаем семафор
        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}";
            myThread.Start();
        }

        public void Read()
        {
            while (count > 0)
            {
                sem.WaitOne();  // ожидаем, когда освободиться место

                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();  // освобождаем место

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

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

Сам семафор определяется в виде статической переменной sem:

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 sem.WaitOne():

```
sem.WaitOne();  // ожидаем, когда освободиться место
```

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 sem.Release():

```
sem.Release();  // освобождаем место
```

После этого в семафоре освобождается одно место, которое заполняет другой поток.

Пример консольного вывода:

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

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