Лекция 8 Использование данных разными нитями одного процесса

8.1 Использование локальных переменных разными нитями

В предыдущей лекции мы рассмотрели работу двух нитей, которые выводили символы «A» и «B» в консольное окно экрана монитора. Нити работали независимо друг от друга. Это происходит, потому что в языке C# каждому объекту выделяется свой стек данных и локальные переменные хранятся отдельно.

Если создать один объект для некоторого класса, содержащего **метод** вывода чисел в консольное окно **и локальную переменную**, то можно проверить, можно ли изменять локальную переменную объекта этого класса при работе разных нитей. Это **можно организовать, запустив метод** нашего объекта **в главной нити** и **в** одной **дополнительной**.

Исходный код программы:

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
    class Danie_1
    {
          class Pervaj_1
          {
              bool flag;
              public void Niti_1()
              {
                  Console.WriteLine("Работает
нить");
                  if (!flag)
                  {
                      flag = true;
                      for (int i = 0; i < 10; i++)
                          Console.Write(i + " ");
                      Console.WriteLine();
                  }
              }
          }
        static void Main()
        {
            Pervaj_1 tt = new Pervaj_1();
            new Thread(tt.Niti_1).Start();
            tt.Niti_1();
            Console.ReadLine();
        }
```

```
    }
}
```
Результат работы программы:
Работает нить
Работает нить
0 1 2 3 4 5 6 7 8 9
или

Работает нить
0 1 2 3 4 5 6 7 8 9
Работает нить

Как видно из работы нашей программы обе нити работают с локальной переменной **flag**. Выводится только одна последовательность чисел, а другая блокируется – нить «видит» значение переменной **flag**.

2) Если создать два объекта и для каждого запустить **свою** нить, то для каждого объекта (соответственно каждой нити) будет создаваться своя локальная переменная **flag** и взаимное использование локальной переменной разными нитями прекратиться.
Исходный код программы:
```
using System;
using System.Threading;

namespace ConsoleApplication1
{
    class Danie_1
    {
        class Pervaj_1
        {
            bool flag;
            public void Niti_1()
            {
                Console.WriteLine("Работает нить");
                if (!flag)
                {
                    flag = true;
                    for (int i = 0; i < 10; i++)
                        Console.Write(i + " ");
                    Console.WriteLine();
                }
            }
        }
        static void Main()
        {
            Pervaj_1 tt = new Pervaj_1();
            new Thread(tt.Niti_1).Start();
            Pervaj_1 vv = new Pervaj_1();
            new Thread(vv.Niti_1).Start();
            //tt.Niti_1();
```

```
            Console.ReadLine();
        }
    }
}
```

Работает нить
Работает нить
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
или

Работает нить
0 1 2 Работает нить
0 1 2 3 4 5 6 7 8 9
3 4 5 6 7 8 9
и т.д.

Таким образом, о **совместном** использовании данных разными нитями имеет смысл говорить, если нити используют методы или данные, принадлежащие **одному объекту**.

В теории использования разными нитями данных и методов, как правило, рассматриваются и вопросы **работы нитей со «статическими» элементами**, т.е. элементами, имеющими **спецификатор доступа static.** Объявим в нашей программе **одну статическую переменную** и один метод и попробуем их использовать в разных нитях.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
 class Danie_1
 {
  static int k = 0;
  static void cikl()
  {
   k++;
   Console.WriteLine("Работает поток " + k);
   int n1, n2;
   n1 = k * 10; n2 = (k + 1) * 10;
   for (int i = n1; i < n2; i++)
        Console.Write(i + " ");
   Console.WriteLine();
  }

  static void Main()
  {
   new Thread(cikl).Start();
   new Thread(cikl).Start();
   new Thread(cikl).Start();
   new Thread(cikl).Start();
```

```
    cikl();
    Console.ReadLine();
   }
  }
}
```

Работа программы:
Работает поток 1
Работает поток 3
50 Работает поток 2
50 51 52 53 54 55 56 57 58 59
50 51 52 53 54 55 56 57 58 59
51 52 53 54 55 56 57 58 59
Работает поток 4
Работает поток 5
50 51 52 53 54 55 56 57 58 59
50 51 52 53 54 55 56 57 58 59

Все запущенные нити используют одну и ту же статическую переменную и работают с одним методом. И если использовать термин безопасности работы нитей – рентабельность, то все они не рентабельны.

Один из вариантов обеспечения безопасности работы нитей и использования общих данных заключается в использовании в них оператора **lock**, позволяющего блокировать работу всех остальных нитей (фактически блокирование всей программы) до завершения работы текущей нити.

Оператор **lock** относится к специальным блокирующим конструкциям языка C#.

Рассмотрим, как оператор **lock** определяют различные авторы учебников по программированию на языке **C#.**

Оператор **lock** является синтаксическим сокращением для вызова методов **Enter** и **Exit** класса **Monitir** с блоком **try/finally** [Албахари стр.743].

Ключевое слово **lock** позволяет блокировать блок кода таким образом, что в каждый момент времени его может использовать только одна нить [Троелсен стр.312].

Павловская Т.А. определяет **lock** как оператор со следующим форматом записи:

lock (выражение)

{ блок операторов}

Выражение определяет объект, который требуется заблокировать. Для обычных методов в качестве выражения используется ключевое слово **this**, для статических методов – **typeof(**класс). Блок операторов задает критическую секцию кода, которую требуется заблокировать [Павловская стр. 242].

Во всех определениях присутствует единое утверждение, что **lock** включает некоторый блок кода, который блокируется для использования многими нитями. С методической точки зрения лучшим и понятным

определением является определение Павловской Т.А., которого мы будем придерживаться в наших лекциях.

Итак, **lock** это оператор, позволяющий блокировать секцию кода таким образом, что в каждый момент времени его может использовать только одна нить.

Исходный код программы:

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
 class Danie_1
 {
  static int k = 0;
  static object t = typeof(object);
  static void cikl()
  {
   Console.WriteLine("Работает нить ---- " + k);
   lock (t)
   {
    k++;
    Console.WriteLine("Работает нить № " + k);
    int n1, n2;
    n1 = k * 10; n2 = (k + 1) * 10;
    for (int i = n1; i < n2; i++)
         Console.Write(i + " ");
    Console.WriteLine();
   }
  }

  static void Main()
  {
   new Thread(cikl).Start();
   new Thread(cikl).Start();
   new Thread(cikl).Start();
   new Thread(cikl).Start();
   cikl();
   Console.ReadLine();
  }
 }
}
```

Работа программы:

Работает нить ---- 0
Работает нить ---- 0
Работает нить ---- 0
Работает нить ---- 0
Работает нить ---- 0
Работает нить № 1
10 11 12 13 14 15 16 17 18 19
Работает нить № 2

20 21 22 23 24 25 26 27 28 29
Работает нить № 3
30 31 32 33 34 35 36 37 38 39
Работает нить № 4
40 41 42 43 44 45 46 47 48 49
Работает нить № 5
50 51 52 53 54 55 56 57 58 59

В нашей программе одновременно запускаются все нити, но выполнение очередной нити приводит к блокировке выполнения остальных нитей программы с помощью оператора **lock**.

Подобные задачи возникают, когда необходимо, чтобы при работе нити значения некоторых переменных, используемых и другими нитями программы, не изменялись до окончания работы нити, например, при выполнении операций со счетами клиентов банка.

Нить, закончившая работу, не может быть начата заново.

8.2 Передача данных нитям

Рассмотренный нами делегат **ThreadStart**, используемый при создании объектов нити, не имеет формальных параметров для передачи данных нитям – делегат способен только определять имя **запускаемого** метода.

Однако в языке **C#** конструктор нити перегружен и допускает использование и другого делегата платформы **.NET Framework**, способного не только создавать объект, но и передавать некоторый параметр – **ParameterizedThreadStart**. Этот делегат имеет следующий формат записи:

public delegate void ParameterizedThreadStart(object obj);

Метод **ParameterizedThreadStart** способен принимать только один параметр типа **object** – любой объект в языке **C#** является производным от **object**.

Рассмотрим учебный пример, в котором будем явно указывать название используемого делегата (в учебных целях) и передавать некоторый аргумент методу, для которого запускаются нити.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
 class Nit_Par
 {
  static void cikl(object ob)
  {
   int k = (int)ob;
   Console.WriteLine("Работает нить ---- " + k);
   k++;
   Console.WriteLine("Работает нить № " + k);
```

```
    for (int i = 0; i <= k; i++)
         Console.Write(i + " ");
    Console.WriteLine();
   }

   static void Main()
   {
       Thread[] name = new Thread[5];
       for (int j = 0; j < 5; j++)
       {
        name[j] = new Thread(new ParameterizedThreadStart(cikl));
        name[j].Start(j);
       }
      Console.ReadLine();
   }
  }
}
```

Работа программы:
Работает нить ---- 0
Работает нить № 1
0 1
Работает нить ---- 1
Работает нить ---- 2
Работает нить ---- 3
Работает нить № 4
0 1 2 Работает нить № 3
0 1 2 3
3 4
Работает нить ---- 4
Работает нить № 5
0 1 2 3 4 5
Работает нить № 2
0 1 2

В приведенном примере создается массив объектов класса **Thread** с помощью делегата ParameterizedThreadStart(object obj);, который позволяет во время запуска нитевого метода передавать ему один параметр типа **object**.

Запускаемый в нити метод должен выполнить явное преобразование полученного параметра в целое число.

Как мы уже отмечали, в языке **C#** конструктор нити перегружен и допускает использование нескольких делегатов платформы **.NET Framework** по умолчанию. Поэтому обычно в программах не прописывается название используемого делегата, например, в нашей программе метод **Main** обычно записывается следующим образом:

```
static void Main()
  {
```

```
        Thread[] name = new Thread[5];
        for (int j = 0; j < 5; j++)
        {
            name[j] = new Thread(cikl);
            name[j].Start(j);
        }

      Console.ReadLine();
   }
```

Упорядочение работы нитей можно осуществлять с помощью, например, оператора **lock**.

Другой способ передачи данных в нить состоит в запуске в нити метода определенного экземпляра объекта, который позволяет работать с полями объекта, а не статического метода, используемого без объекта и без его полей.

Свойства выбранного экземпляра объекта будут определять поведение нитей.

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
 class Program
 {
  public static int kol=0;
  class T
  {
   private double data;
   private int c, n;
   public Thread thread1;

   public T()
   {
    thread1 = new Thread(run);
    thread1.Start(kol);
   }
   public void run(object ob)
   {
    lock (this)
    {
     kol++; n = kol;
     Console.WriteLine("Нить {0} стартовала ",kol);
     Console.WriteLine("Введите число");
     c = int.Parse(Console.ReadLine());
     if (kol == 4) data = c*40;
     if (kol < 4)
     {
      switch (kol)
      {
       case 1: data = Math.Cos(c); break;
```

```
            case 2: data = Math.Exp(c); break;
            case 3: data = c + 30; break;
           }
           T thr2 = new T();
           thr2.thread1.Join();
          }
          Console.WriteLine("Нить {0} завершена! Значение равно:
                                                    {1}",n,data);
        }
       }
      }
      static void Main()
      {
       T thr1 = new T();
       thr1.thread1.Join();
       Console.ReadKey();
      }
     }
}
```

Работа программы:
Нить 1 стартовала
Введите число
1
Нить 2 стартовала
Введите число
2
Нить 3 стартовала
Введите число
3
Нить 4 стартовала
Введите число
4
Нить 4 завершена! Значение равно: 160
Нить 3 завершена! Значение равно: 33
Нить 2 завершена! Значение равно: 7,38905609893065
Нить 1 завершена! Значение равно: 0,54030230586814

В нашем примере запускаются 4 нити, каждая из которых использует метод **run** класса **T** и, в качестве параметра, получает переменную **kol**, значение которой соответствует номеру запущенной нити. Метод **run** является методом класса **T**, поэтому он может работать с полями объекта этого класса, например, вычислять некоторое выражение, используя значение поля **c**, индивидуально заданное для каждой нити. В нашей программе в каждой нити использовано свое выражение, результат которого присваивается вещественной переменной **data**.

В программе использован рекурсивный способ запуска очередной нити – ограничение числа запускаемых нитей осуществляется с помощью условия

if (kol < 4). Переменная **kol** изменяется от 0. В каждой запущенной нити создается свой объект T thr2 = new T().

Внутри созданного объекта запускается метод Join() (thr2.thread1.Join();), который болкирует вызывающий объект до его завершения.

О целесообразности использования многонитевых приложений – обзор из Интернета (нити иногда называются потоками).

« Когда использовать потоки

Многопоточность имеет смысл, если выполняемая задача может занять много времени, так как требуется ожидание ответа от другого компьютера (сервера приложений, сервера баз данных или клиента).

Типовое приложение с многопоточностью выполняет длительные вычисления в фоновом режиме. Главный поток продолжает выполнение, в то время как рабочий поток выполняет фоновую задачу.

В приложениях Windows Forms, когда главный поток занят длительными вычислениями, он не может обрабатывать сообщения клавиатуры и мыши, и приложение перестает откликаться. По этой причине следует запускать отнимающие много времени задачи в рабочем потоке, даже если главный поток в это время демонстрирует пользователю модальный диалог с надписью *“Работаю... Пожалуйста, ждите”*, так как программа не может перейти к следующей операции, пока не закончена текущая. Такое решение гарантирует, что приложение не будет помечено операционной системой как “Не отвечающее”, соблазняя пользователя с горя прикончить процесс. Опять же, в этом случае модальный диалог может предоставить кнопку “Отмена”, так как форма продолжает получать сообщения, пока задача выполняется в фоновом потоке.

Однопоточный web-сервер не просто плох, он попросту невозможен! К счастью, в случае серверов приложений, не хранящих состояние (stateless), многопоточность реализуется обычно довольно просто, сложности возможны разве что в синхронизации доступа к данным в статических переменных.

Когда потоки не нужны

Многопоточность наряду с достоинствами имеет и свои недостатки. Самое главный из них – значительное увеличение сложности программ. Сложность

увеличивают не дополнительные потоки сами по себе, а необходимость организации их взаимодействия. От того, насколько это взаимодействие является преднамеренным, зависит продолжительность цикла разработки, а также количество спорадически проявляющихся и трудноуловимых ошибок в программе. Таким образом, нужно либо поддерживать дизайн взаимодействия потоков простым, либо не использовать многопоточность вообще, если только вы не имеете противоестественной склонности к переписыванию и отладке кода. »