3. Работа с потоками

В системах с общей памятью, включая многоядерные архитектуры, параллельные вычисления могут выполняться как при многопроцессном выполнении, так и при многопоточном выполнении. Многопроцессное выполнение подразумевает оформление каждой подзадачи в виде отдельной программы (процесса). Недостатком такого подхода является сложность взаимодействия подзадач. Каждый процесс функционирует в своем виртуальном адресном пространстве, не пересекающемся с адресным пространством другого процесса. Для взаимодействия подзадач необходимо использовать специальные средства межпроцессной коммуникации (интерфейсы передачи сообщений, общие файлы, объекты ядра операционной системы).

Потоки позволяют выделить подзадачи в рамках одного процесса. Все потоки одного приложения работают в рамках одного адресного процесса. Для взаимодействия потоков не нужно применять какие-либо средства коммуникации. Потоки могут непосредственно обращаться к общим переменным, которые изменяют другие потоки. Работа с общими переменными приводит к необходимости использования средств синхронизации, регулирующими порядок работы потоков с данными.

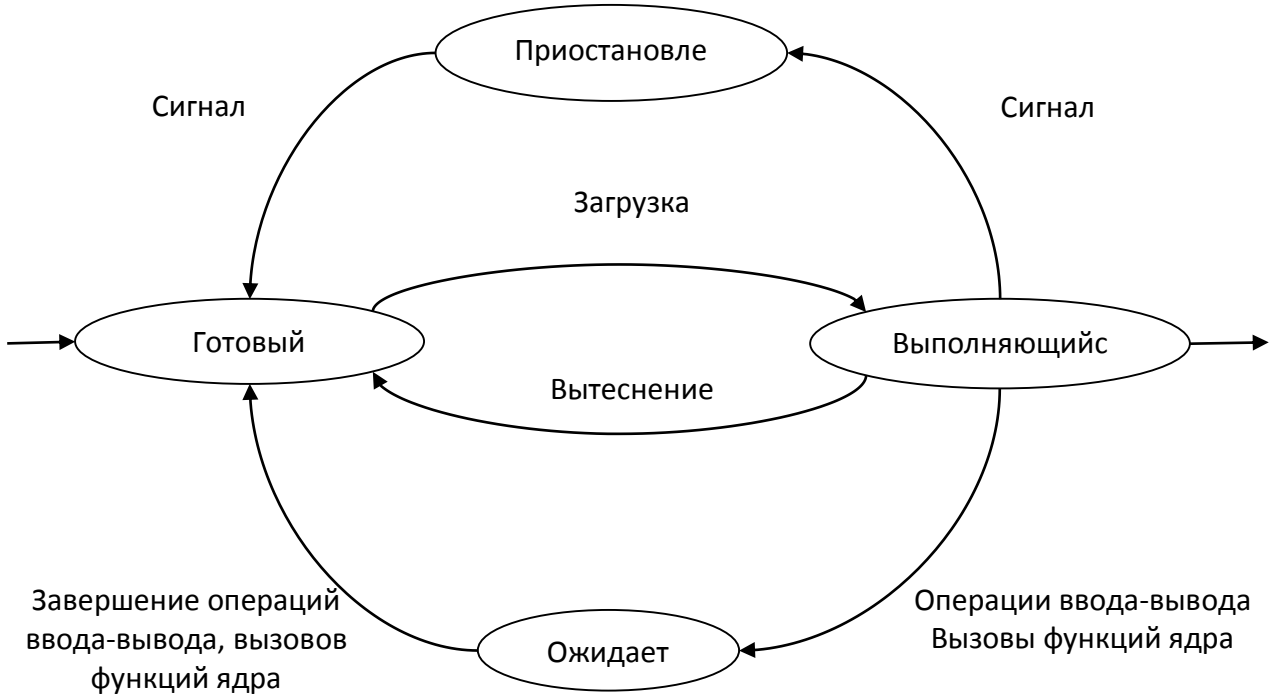
Потоки являются более легковесной структурой по сравнению с процессами («облегченные» процессы). Поэтому параллельная работа множества потоков, решающих общую задачу, более эффективна в плане временных затрат, чем параллельная работа множества процессов.

Структура потока

Поток состоит из нескольких структур. ***Ядро потока*** – содержит информацию о текущем состоянии потока: приоритет потока, программный и стековый указатели. Программный и стековые указатели образуют контекст потока и позволяют восстановить выполнение потока на процессоре. **Блок окружения потока** - содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL. **Стек пользовательского режима** – используется для передаваемых в методы локальных переменных и аргументов. **Стек режима ядра** - используется, когда код приложения передает аргументы в функцию операционной системы, находящуюся в режиме ядра. Ядро ОС вызывает собственные методы и использует стек режима ядра для передачи локальных аргументов, а также для сохранения локальных переменных**.**

Состояния потоков

Каждый поток может находиться в одном из нескольких состояний. Поток, готовый к выполнению и ожидающий предоставления доступа к центральному процессору, находится в состоянии «Готовый». Поток, который выполняется в текущий момент времени, имеет статус «Выполняющийся». При выполнении операций ввода-вывода или обращений к функциям ядра операционной системы, поток снимается с процессора и находится в состоянии «Ожидает». При завершении операций ввода-вывода или возврате из функций ядра поток помещается в очередь готовых потоков. При переключении контекста поток выгружается и помещается в очередь готовых потоков.



Переключение контекста

1. Значения регистров процессора для исполняющегося в данный момент потока сохраняются в структуре контекста, которая располагается в ядре потока.
2. Из набора имеющихся потоков выделяется тот, которому будет передано управление. Если выбранный поток принадлежит другому процессу, Windows переключает для процессора виртуальное адресное пространство.
3. Значения из выбранной структуры контекста потока загружаются в регистры процессора.

Работа с потоками в C#

Среда исполнения .NET CLR предоставляет возможность работы с управляемыми потоками через объект Thread пространства имен `System.Threading`. Среда исполнения стремится оптимизировать работу управляемых потоков и использовать для их выполнения потоки процесса, существующие на уровне операционной системы. Поэтому создание потоков типа Thread не всегда сопряжено с созданием потоков процесса.

Основные этапы работы

```
// Инициализация потока
Thread oneThread = new Thread(Run);
// Запуск потока
oneThread.Start();
// Ожидание завершения потока
oneThread.Join();
```

В качестве рабочего элемента можно использовать метод класса, делегат метода или лямбда-выражение. В следующем фрагменте создаются три потока. Первый поток в качестве рабочего элемента принимает статический метод `LocalWorkItem`. Второй поток инициализируется с помощью лямбда-выражения, третий поток связывается с методом общедоступного класса.

```
class Program
{
     static void LocalWorkItem()
     {
          Console.WriteLine("Hello from static method");
     }
     static void Main()
     {
          Thread thr1 = new Thread(LocalWorkItem);
          thr1.Start();
          Thread thr2 = new Thread(() =>
               {
                    Console.WriteLine("Hello from
                                        lambda-expression");
               });
          thr2.Start();
          ThreadClass thrClass = new  ThreadClass("Hello from
                                              thread-class");
          Thread thr3 = new Thread(thrClass.Run);
          thr3.Start();
     }
}
class ThreadClass
{
     private string greeting;
     public ThreadClass(string sGreeting)
     {
```

```
            greeting = sGreeting;
      }
      public void Run()
      {
            Console.WriteLine(greeting);
      }
}
```

Вызов метода `thr1.Join()` блокирует основной поток до завершения работы потока `thr1`.

```
      Thread thr1 = new Thread(() =>
            {
                  for(int i=0; i<5; i++)
                        Console.Write("A");
            });
      Thread thr2 = new Thread(() =>
            {
                  for(int i=0; i<5; i++)
                        Console.Write("B");
            });
      Thread thr3 = new Thread(() =>
            {
                  for(int i=0; i<5; i++)
                        Console.Write("C");
            });
      thr1.Start();
      thr2.Start();
      thr1.Join();
      thr2.Join();
      thr3.Start();
```

В общем случае порядок вывода первого и второго потоков не определен. Вывод третьего потока осуществляется только после завершения работы потоков `thr1` и `thr2`.

Можно получить такие результаты:

```
AAAAABBBBBCCCCC
```

или

```
BBBBBAAAAACCCCC
```

Маловероятны, но возможны варианты

```
AABBAAABCCCCC
```

или

```
AAAABBBBBACCCCC
```

Передача параметров

Общение с потоком (передача параметров, возвращение результатов) можно реализовать с помощью глобальных переменных.

```
class Program
{
     static long Factorial(long n)
     {
          long res = 1.0;
          do
          {
               res = res * n;
          } while(--n > 0);
          return res;
     }
     static void Main()
     {
          long res1, res2;
          long n1 = 5000, n2 = 10000;
          Thread t1 = new Thread(() =>
               {
                    res1 = Factorial(n1))
               });
          Thread t2 = new Thread(() => { res2=Factorial(n2); });
          // Запускаем потоки
          t1.Start();    t2.Start();
          // Ожидаем завершения потоков
          t1.Join();     t2.Join();
          Console.WriteLine("Factorial of {0} equals {1}",
                                        n1, res1);
          Console.WriteLine("Factorial of {0} equals {1}",
                                        n2, res2);
     }
}
```

Существует возможность передать параметры в рабочий метод потока с помощью перегрузки метода `Start`. Сигнатура рабочего метода строго фиксирована – либо без аргументов, либо только один аргумент типа `object`. Поэтому при необходимости передачи нескольких параметров в рабочем методе необходимо выполнить правильные преобразования.

```
     class Program
     {
          static double res;
          static void ThreadWork(object state)
          {
               string sTitle = ((object[])state)[0] as string;
               double d = (double)(((object[])state)[1]);
               Console.WriteLine(sTitle);
               res = SomeMathOperation(d);
```

```
        }
        static void Main()
        {
            Thread thr1 = new Thread(ThreadWork);
            thr1.Start(new object[] {"Thread #1", 3.14});
            thr1.Join();
            Console.WriteLine("Result: {0}", res);

        }
    }
```

Работа в лямбда-выражениях и анонимных делегатах с общими переменными может приводить к непредсказуемым результатам.

```
    for(int i=0;i<=10; i++)
    {
        Thread t = new Thread(() =>
                    Console.Write("ABCDEFGHIJK"[i]));
        t.Start();
    }
```

Ожидаем получить все буквы в случайном порядке, а получаем

```
    BDDDEEJJKK
```

Если в строковой константе оставить только 10 букв, полагая, что индекс i может быть от 0 до 9, получаем ошибку «Индекс вышел за границы массива».

Проблема связана с тем, что при объявлении потока делегат метода или лямбда-выражение содержит только ссылку на индекс i. Когда созданный поток начинает свою работу фактическое значение индекса уже давно убежало вперед. Последнее значение индекса равно 10, что и приводит к возникновению исключения. Исправить данный фрагмент можно с помощью дополнительной переменной, которая на каждой итерации сохраняет текущее значение индекса.

```
    for(int i=0; i<=10; i+
    +) {
        int i_copy = i;
        Thread t = new Thread(new delegate(
                    Console.Write("ABCDEFGHIJK"[i_copy]))
        t.Start();
    }
```

Теперь у каждого потока своя независимая переменная `i_copy` с уникальным значением. В результате получаем:

```
    ABCDEFGHIJK
```

или в произвольном порядке, но все буквы по одному разу

```
BDACFGHEIJK
```

Приостановление потока

Метод `Sleep()` позволяет приостановить выполнение текущего потока на заданное число миллисекунд:

```
// Приостанавливаем поток на 100 мс
Thread.Sleep(100);
// Приостанавливаем поток на 5 мин
Thread.Sleep(TimeSpan.FromMinute(5));
```

Если в качестве аргумента указывается ноль `Thread.Sleep(0)`, то выполняющийся поток отдает выделенный квант времени и без ожидания включается в конкуренцию за процессорное время. Такой прием может быть полезен в отладочных целях для обеспечения параллельности выполнения определенных фрагментов кода.

Например, следующий фрагмент

```
static void ThreadFunc(object o)
{
     for(int i=0; i<20; i++)
          Console.Write(o);
}
static void Main()
{
     Thread[] t = new Thread[4];
     for(int i=0; i<4; i++)
          t[i] = new Thread(ThreadFunc);

     t[0].Start("A"); t[1].Start("B");
     t[2].Start("C"); t[3].Start("D");

     for(int i=0; i<4; i++)
          t[i].Join();
}
```

Выводит на консоль:

```
BBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCDDDDD
DDDDDDDDDDDDDD
```

Параллельность не наблюдается, так как каждый поток за выделенный квант процессорного времени успевает обработать все 20 итераций. Изменим тело цикла рабочей функции:

```
     static void ThreadFunc(object o)
     {
          for(int i=0; i<20; i++)
          {
```

```
                Console.Write(o);
                Thread.Sleep(0);
            }
        }
```

Вывод стал более разнообразный:

```
AAAACACACACACACACACACACACACACACACACAACCBBDDDBDBDBDBDBDBDBDBDBDBDBDBDBC
CCDBBBDDBBDD
```

Существует аналог метода `Thread.Sleep(0)`, который позволяет вернуть выделенный квант – `Thread.Yield()`. При этом возврат осуществляется только в том случае, если для ядра, на котором выполняется данный поток, есть другой готовый к выполнению поток. Неосторожное применение методов `Thread.Sleep(0)` и `Thread.Yield()` может привести к ухудшению быстродействия из-за не оптимального использования кэш-памяти.

Свойства потока

Каждый поток имеет ряд свойств: `Name` - имя потока, `ManagedThreadId` – номер потока, `IsAlive` - признак существования потока, `IsBackground` – признак фонового потока, `ThreadState` – состояние потока. Эти свойства доступны и для внешнего вызова.

```
        // Объявляем массив потоков
        Thread[] arThr = new Thread[N];
        for(int i=0; i<arThr.Length; i++)
        {
            arThr[i] = new Thread(SomeFunc);
            arThr[i].Start();
        }
        for(int i=0; i<arThr.Length; i++)
        {
            // Выводим информацию о потоках
            Console.WriteLine("Thread Id: {0},
                        name: {1}, IsAlive: {2}",
                            arThr[i].ManagedThreadId,
                            arThr[i].Name,
                            arThr[i].IsAlive);
        }
```

Свойства текущего потока можно получить с помощью объекта `Thread.CurrentThread`. Следующий фрагмент с помощью механизма рефлексии выводит все свойства текущего потока.

```
using System;
using System.Reflection;
using System.Threading;
class ThreadInfo
{
```

```
     static void Main()
     {
          Thread t = Thread.CurrentThread;
          t.Name = "MAIN THREAD";
          foreach(PropertyInfo p in t.GetType().GetProperties())
          {
               Console.WriteLine("{0}:{1}",
                         p.Name,p.GetValue(t, null));
          }

     }
```

}

Вывод всех свойств текущего потока:

```
ManagedThreadId: 10
ExecutionContext: System.Threading.ExecutionContext
Priority: Normal
IsAlive: True
IsThreadPoolThread: False
CurrentThread: System.Threading.Thread
IsBackground: False
ThreadState: Running
ApartmentState: MTA
CurrentUICulture: ru-RU
CurrentCulture: ru-RU
CurrentContext: ContextID: 0
CurrentPrincipal: System.Security.Principal.GenericPrincipal
Name: MAIN THREAD
```

Приоритеты потоков

Приоритеты потоков определяют очередность выделения доступа к ЦП. Высокоприоритетные потоки имеют преимущество и чаще получают доступ к ЦП, чем низкоприоритетные. Приоритеты потоков задаются перечислением `ThreadPriority`, которое имеет пять значений: `Highest` - наивысший, `AboveNormal` – выше среднего, `Normal` - средний, `BelowNormal` – ниже среднего, `Lowest` - низший. По умолчанию поток имеет средний приоритет. Для изменения приоритета потока или чтения текущего используется свойство `Priority`. Влияние приоритетов сказывается только в случае конкуренции множества потоков за мощности ЦП.

В следующем фрагменте пять потоков с разными приоритетами конкурируют за доступ к ЦП с двумя ядрами. Каждый поток увеличивает свой счетчик.

```
 class PriorityTesting
{
     static long[] counts;
     static bool finish;
     static void ThreadFunc(object iThread)
```

```
    {
        while(true)
        {
            if(finish)
                break;
            counts[(int)iThread]++;
        }
    }
    static void Main()
    {
        counts = new long[5];
        Thread[] t = new Thread[5];
        for(int i=0; i<t.Length; i++)
        {
            t[i] = new Thread(ThreadFunc);
            t[i].Priority = (ThreadPriority)i;
        }
        // Запускаем потоки
        for(int i=0; i<t.Length; i++)
            t[i].Start(i);

        // Даём потокам возможность поработать 10 с
        Thread.Sleep(10000);

        // Сигнал о завершении
        finish = true;

        // Ожидаем завершения всех потоков
        for(int i=0; i<t.Length; i++)
            t[i].Join();
        // Вывод результатов
        for(int i=0; i<t.Length; i++)
        Console.WriteLine("Thread with priority {0},
                Counts: {1}", (ThreadPriority)i, counts[i]);
    }
}
```

Вывод программы

```
Thread with priority         Lowest, Counts:    7608195
Thread with priority    BelowNormal, Counts:   10457706
Thread with priority         Normal, Counts:   17852629
Thread with priority    AboveNormal, Counts:  297729812
Thread with priority        Highest, Counts:  302506232
```

Локальное хранилище потока

Типы, объявленные внутри рабочей функции потока, являются локальными – у каждого потока, выполняющего функцию, свои копии (приватные данные). Типы,

объявленные вне рабочей функции потока, являются общими для всех потоков переменными. Изменения общих данных в одном потоке отражаются в другом потоке.

Существует возможность оперировать с локальными данными потока, объявленными вне рабочей функции, например, в общедоступном классе. Первый способ заключается в объявлении статического поля, локального для каждого потока.

В следующем фрагменте рабочая функция потока использует объект пользовательского типа Data. В этом классе

```
public class Data
{
     public static int sharedVar;
     [ThreadStatic] public static int localVar;
}
static void threadFunc(object i)
{
     Console.WriteLine("Thread {0}: Before changing.. Shared:
{1}, local: {2}",
       i, Data.sharedVar, Data.localVar);
     Data.sharedVar = (int)i;
     Data.localVar = (int)i;
     Console.WriteLine("Thread {0}: After changing.. Shared:
{1}, local: {2}",
       i, Data.sharedVar, Data.localVar);
}
static void Main()
{
     Thread t1 = new Thread(threadFunc);
     Thread t2 = new Thread(threadFunc);
     Data.sharedVar = 3; Data.localVar = 3;
     t1.Start(1); t2.Start(2);
     t1.Join(); t2.Join();
}
```

Вывод:

```
Thread 1: Before changing.. Shared: 3, local: 0
Thread 1: After  changing.. Shared: 1, local: 1
Thread 2: Before changing.. Shared: 1, local: 0
Thread 2: After  changing.. Shared: 2, local: 2
```

Ограничения этого способа связаны с тем, что атрибут используется только со статическими полями и инициализация поля осуществляется только одним потоком.

Второй способ объявления локальных данных заключается в использовании объекта `ThreadLocal<T>`:

```
static void Main()
{
     ThreadLocal<int> localSum = new ThreadLocal<int>(() => 0);
     Thread t1 = new Thread(() => {
               for(int i=0; i<10; i++)
                    localSum.Value++;
               Console.WriteLine(localSum.Value);
          });
     Thread t2 = new Thread(() => {
               for(int i=0; i<10; i++)
                    localSum.Value--;
               Console.WriteLine(localSum.Value);
          });

     t1.Start(); t2.Start();
     t1.Join(); t2.Join();

     Console.WriteLine(localSum.Value);
}
```

Получаем:

```
10
-10
0
```

Первый поток увеличивал свой счетчик, второй уменьшал, а третий (главный поток) ничего не делал со своим локальным счетчиком, поэтому получаем 0.

Третий способ заключается в использовании локальных слотов потока. Доступ к слотам обеспечивается с помощью методов `GetData`, `SetData`. Объект, идентифицирующий слот, можно получить с помощью строковой константы. Применение слотов является менее производительным по сравнению с локальными статическими полями. Но может быть полезным, если работа потока структурирована в нескольких методах. В каждом методе можно получить доступ к слоту потока по его имени.

```
public class ThreadWork
{
     private string sharedWord;
     public void Run(string secretWord)
     {
          sharedWord = secretWord;
          Save(secretWord);
          Thread.Sleep(500);
          Show();
     }
```

```
     private void Save(string s)
     {
          // Получаем идентификатор слота по имени
          LocalDataStoreSlot slot =
Thread.GetNamedDataSlot("Secret");
          // Сохраняем данные
          Thread.SetData(slot, s);
     }
     private void Show()
     {
          LocalDataStoreSlot slot =
Thread.GetNamedDataSlot("Secret");
          string secretWord = (string)Thread.GetData(slot);
          Console.WriteLine("Thread {0}, secret word: {1},
shared word: {2}",
          Thread.CurrentThread.ManagedThreadId, secretWord,
sharedWord);
     }
}
static void Main()
{
     ThreadWork thr = new ThreadWork();
     new Thread(() => thr.Run("one")).Start();
     new Thread(() => thr.Run("two")).Start();
     new Thread(() => thr.Run("three")).Start();
     Thread.Sleep(1000);
}
```

Вывод программы

```
Thread: 15, secret word: one, shared word: three
Thread: 16, secret word: two, shared word: three
Thread: 17, secret word: three, shared word: three
```

Переменная `sharedWord` является разделяемой, поэтому выводится последнее изменение, выполненное третьим потоком.

Пул потоков

Пул потоков предназначен для упрощения многопоточной обработки. Программист выделяет фрагменты кода (рабочие элементы), которые можно выполнять параллельно. Планировщик (среда выполнения) оптимальным образом распределяет рабочие элементы по рабочим потокам пула. Таким образом, вопросы эффективной загрузки оптимального числа потоков решаются не программистом, а планировщиком (исполняющей средой). Еще одним достоинством применения пула является уменьшение накладных расходов, связанных с ручным созданием и завершением потоков для каждого фрагмента кода,

допускающего распараллеливание. Пул потоков используется для обработки задач типа Task. Задачи обладают рядом полезных встроенных механизмов (ожидания, отмены, продолжения и т.д.). Поэтому для распараллеливания рабочих элементов рекомендуется использовать именно задачи или шаблоны класса Parallel. Непосредственная работа с пулом без явного определения задач может быть полезна, когда нет необходимости в дополнительных возможностях объекта Task.

Для добавления рабочего элемента используется метод

```
      // Добавление метода без параметров
      ThreadPool.QuequeUserWorkItem(SomeWork);
      // Добавление метода с параметром
      ThreadPool.QuequeUserWorkItem(SomeWork, data);
```

В следующем фрагменте проиллюстрируем основные особенности пула потоков.

```
for (int i = 0; i < 10; i++)
{
  ThreadPool.QueueUserWorkItem((object o) =>
  {
    Console.WriteLine("i: {0}, ThreadId: {1}, IsPoolThread: {2}",
      i, Thread.CurrentThread.ManagedThreadId,
      Thread.CurrentThread.IsThreadPoolThread);
  });
}
```

Добавляем в пул потоков 10 экземпляров безымянного делегата, объявленного в виде лямбда-выражения. В рабочем элементе осуществляется вывод значения индекса i, номер потока и признак того, что поток принадлежит пулу.

```
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 3, IsPoolThread: True
i: 10, ThreadId: 4, IsPoolThread: True
```

Убеждаем, что действительно все рабочие элементы выполнялись потоками пула (признак `IsPoolThread` равен true). Всего в обработке участвовало только два потока. Для приведенного кода возможна ситуация, когда все рабочие элементы будут обрабатываться в одном потоке пула. Во всех рабочих элементах осуществляется вывод одного и того же значения индекса i, равного 10. Это связанно с асинхронностью запуска рабочих элементов – основной поток продолжает обрабатывать цикл и увеличивать значение индекса, а рабочие элементы фактически еще не выполняются. Для

предотвращения такой ситуации необходимо использовать индивидуальные копии для каждого элемента.

Заменим работу с пулом на ручную работу с потоками:

```
for (int i = 0; i < 10; i++)
{
  new Thread((object o) =>
  {
    Console.WriteLine("i: {0}, ThreadId: {1}, IsPoolThread: {2}",
      i, Thread.CurrentThread.ManagedThreadId,
      Thread.CurrentThread.IsThreadPoolThread);
  }).Start();
}
```

Вывод кода будет следующим:

```
i: 1, ThreadId: 5, IsPoolThread: False
i: 1, ThreadId: 6, IsPoolThread: False
i: 3, ThreadId: 7, IsPoolThread: False
i: 4, ThreadId: 8, IsPoolThread: False
i: 5, ThreadId: 9, IsPoolThread: False
i: 6, ThreadId: 10, IsPoolThread: False
i: 7, ThreadId: 11, IsPoolThread: False
i: 8, ThreadId: 12, IsPoolThread: False
i: 9, ThreadId: 13, IsPoolThread: False
i: 10, ThreadId: 14, IsPoolThread: False
```

Каждый рабочий элемент обрабатывался в своем потоке, не входящем в состав пула. С индексом i опять есть проблемы, но возникают они гораздо реже и не так явно проявляются – какое-то разнообразие в выводе все-таки есть. Это объясняется тем, что операция добавления делегата в очередь выполняется гораздо быстрее, чем инициализация запуска нового потока. Пока на второй итерации осуществляется запуск потока, первый поток уже приступил к работе и прочитал текущее значение индекса. Для гарантированной работы каждого потока с уникальным индексом необходимо использовать копии индексов, создаваемые на каждой итерации:

```
for(int i=0; i<10; i++)
{
     int y = i;
     ThreadPool.QueueUserWorkItem((object o) =>
                                  Console.WriteLine(y));
     new Thread(() => Console.WriteLine(y)).Start();
}
```

Основным неудобством работы с пулом, является отсутствие механизма ожидания завершения рабочих элементов. Необходимо использовать либо какие-то средства синхронизации (например, сигнальные сообщения `ManualResetEvent`, шаблон синхронизации `CountdownEvent`), либо общие переменные.

```
static void Func1(object o)
{
     var ev = (ManualResetEventSlim)o;
     // Работа
     Console.WriteLine("Func1: Working..");
     ev.Set();
}
static void Main()
{
     ManualResetEvent ev = new ManualResetEvent();
     ThreadPool.QuequeWorkItem(Fun1, ev);
     ev.Wait()
}
```

В рабочую функцию передаем сигнальное сообщение. Основной поток блокируется в ожидании сигнала. Рабочий поток после завершения работы генерирует сигнал. В случае работы нескольких потоков можно использовать массив сообщений и метод `WaitHandle.WaitAll` для ожидания сигналов от всех потоков.

```
class Program
{
     static void Main()
     {
          var events = new ManualResetEvent[10];
          for(int i=0; i<10; i++)
          {
               int y = i;
               events[i] = new ManualResetEvent(false);
               ThreadPool.QuequeWorkItem(() => {
                    SomeWork();
                    events[y].Set();
               });
          }
          WaitHandle.WaitAll(events);
     }
}
```

Каждый рабочий элемент оперирует со своим уникальным сигнальным объектом. После завершения работы, в делегате устанавливается сигнальный объект. Основной поток дожидается завершения всех рабочих элементов.