Работа с потоками

Выделяют два типа *многозадачности*: основанную на *процессах* и основанную на *потоках*, т.е. в системах с общей памятью параллельные вычисления могут выполняться:

- многопроцессно,
- многопоточно.

Многопроцессное выполнение – оформление каждой подзадачи в виде отдельной программы (процесса).

Процесс – это по сути запущенная *программа*. Следовательно, основанная на процессах многозадачность – средство, позволяющее компьютеру выполнять несколько операций (программ) одновременно.

Пример

Основанная на процессах многозадачность предоставляет одновременно редактировать текст в текстовом редакторе и работать с другой запущенной программой.

Недостаток – 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).

Поток – это управляемая *единица* исполняемого кода. В многозадачной среде, основанной на потоках, у всех работающих процессов обязательно имеется *основной поток*, но их может быть и больше. Это означает, что в одной программе могут выполняться несколько задач асинхронно.

Пример

Редактирование текста в текстовом редакторе во время печати, т.к. эти две задачи выполняются в различных потоках.

Потоки позволяют выделить подзадачи в рамках *одного процесса*.

Все потоки одного приложения работают в рамках одного адресного процесса, средства коммуникации не нужны. Потоки могут непосредственно обращаться к общим переменным, что приводит к необходимости использования *средств синхронизации*, регулирующих порядок работы потоков с данными.

Потоки являются более легковесной структурой по сравнению с процессам. Поэтому параллельная работа *множества потоков*, решающих общую задачу, более эффективна в плане временных затрат, чем параллельная работа *множества процессов*.

Структура потока

- *Ядро потока* – содержит информацию о текущем состоянии потока: *приоритет* потока, к*онтекст* потока (программный и стековый указатели).
- *Блок окружения* потока – содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока, структуры данных, используемые интерфейсом графических устройств.
- *Стек пользовательского режима* – для передаваемых в методы локальных переменных и аргументов.
- *Стек режима ядра* – для передачи аргументов в функцию операционной системы, находящуюся в режиме ядра.
- *Ядро ОС* – вызывает собственные методы и использует *стек* режима ядра для передачи локальных аргументов и сохранения локальных переменных.

Состояния потоков

- "Готовый" – поток, готовый к выполнению и ожидающий предоставления доступа к центральному процессору.
- "Выполняющийся" – поток, который выполняется в текущий момент времени.
- "Ожидает" – при выполнении операций ввода-вывода или обращений к функциям ядра операционной системы.
- *Очередь* готовых потоков – при завершении операций ввода-вывода или возврате из функций ядра или при переключении контекста.

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.

Стратегии разделения работы между потоками

- *Параллелизм данных* (data parallelism) – используется, если необходимо над большим объемом данных выполнить некий набор задач, разбиваем данные между потоками.
- *Параллелизм задач* (task parallelism) – распараллеливание задач, каждый поток выполняет свою задачу.

Примечание. Параллелизм данных проще и лучше масштабируется на высокопроизводительном оборудовании. Данных обычно значительно больше, чем отдельных задач.

Работа с потоками в C#

Классы, поддерживающие многопоточное программирование, определены в пространстве имен System.Threading.

Поэтому любая многопоточная программа включает в себя следующую строку кода:

```
using System.Threading;
```

Одним из основных классов в данном пространстве имен является класс Thread.

Основные свойства и методы класса Thread

1. ExecutionContext – позволяет получить контекст, в котором выполняется поток
2. IsAlive – указывает, работает ли поток в текущий момент
3. IsBackground – указывает, является ли поток фоновым
4. Name – содержит имя потока
5. ManagedThreadId – возвращает числовой идентификатор текущего потока
6. Priority – хранит приоритет потока
7. ThreadState – возвращает состояние потока (одно из значений перечисления ThreadState)

8. CurrentThread – свойство только для чтения, возвращает ссылку на текущий выполняемый поток
9. Sleep() – метод, приостанавливающий текущий поток на заданное время
10. Abort() – прерывает поток как только это возможно
11. Join() – блокирует вызывающий поток до тех пор, пока указанный поток не завершится
12. Resume() – возобновляет ранее приостановленный поток
13. Start() – запускает поток

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

Пример. Вывести информацию о потоке.

по умолчанию свойство Name у объектов Thread не установлено

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

```
using System;
using System.Threading;
class HelloWorld {
  static void Main() {
    // получаем текущий поток
    Thread myThread = Thread.CurrentThread;
    //получаем имя потока
    Console.WriteLine($"Имя потока: {myThread.Name}");
    myThread.Name = "Метод Main";
    Console.WriteLine($"Имя потока: {myThread.Name}");
    Console.WriteLine($"Запущен ли поток:
          {myThread.IsAlive}");
    Console.WriteLine($"Id потока:
          {myThread.ManagedThreadId}");
    Console.WriteLine($"Приоритет потока:
          {myThread.Priority}");
    Console.WriteLine($"Статус потока:
          {myThread.ThreadState}");
  }
}
```

Основные этапы работы с потоком

- Инициализация потока

  Thread *Имя_потока* = new Thread(*рабочий_элемент*);
- Запуск потока

  Имя_потока.Start();
- Ожидание завершения потока

  Имя_потока.Join();

  В качестве *рабочего элемента* используются:
- метод класса,
- делегат метода,
- лямбда-выражение.

Пример 1

Создать три потока:

1-й поток – в качестве рабочего элемента принимает статический метод LocalWorkItem;

2-й поток – инициализируется с помощью лямбда-выражения;

3-й поток – связывается с методом общедоступного класса.

```
using System;
using System.Threading;
namespace Simp {
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 S;
   public ThreadClass(string sS)
   { S= sS;
   }
   public void Run()
   { Console.WriteLine(S);
   }
}
```

```
Hello from lambda-expression
Hello from static method
Hello from thread-class
```

Пример 2

Создать три потока, инициализирующихся с помощью лямбда-выражений, каждый поток выводит на экран 5 раз заданный символ (A, B, C).

```
using System;
using System.Threading;
namespace Simple
{ class Program
  { static void Main()
    { 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();
            thr3.Start();
        }
    }
}
```

AAAAABBBBBCCCCC

AAAAACCCCCBBBBB

BBBBBCCCCCAAAAA

AAABBBBBAACCCCC

CCCCCAAAAABBBBB

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

```
thr1.Start();
thr2.Start();
thr1.Join();
thr2.Join();
thr3.Start();
```

```
AAAAABBBBBCCCCC
```

```
BBBBBAAAAACCCCC
```

```
AAABBBBBAACCCCC
```

```
AABBAABBBACCCCC
```

В общем случае порядок вывода 1-го и 2-го потоков не определен. Вывод 3-го потока осуществляется только после завершения работы потоков thr1 и thr2.

Передача параметров

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

Пример. Создать 2 потока, вызывающих функцию вычисления квадрата числа.

```
using System;
using System.Threading;
class Program  {
 static int Kv(int n)
 { int res = n * n;
   return res;
 }
```

```
 static void Main()
 { int res1 = 0, res2 = 0;
   int n1 = 5, n2 = 7;
   // Описываем потоки
   Thread t1 = new Thread(() => { res1 = Kv(n1); });
   Thread t2 = new Thread(() => { res2 = Kv(n2); });
   // Запускаем потоки
   t1.Start();  t2.Start();
   // Ожидаем завершения потоков
   t1.Join();    t2.Join();
   Console.WriteLine("Kvadrat {0} = {1}", n1, res1);
   Console.WriteLine("Kvadrat {0} = {1}", n2, res2);
 }
}
```

```
Kvadrat 5 = 25
Kvadrat 7 = 49
```

Приостановление потока

Метод Sleep() приостанавливает 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*. Это может быть полезно в отладочных целях для обеспечения параллельности выполнения определенных фрагментов кода.

Пример

Описать статический метод, выводящий 20 раз заданный символ. Создать массив из четырех потоков, вызывающих этот метод. Запустить потоки для вывода разных символов с блокировкой для каждого.

```
using System;
using System.Threading;
class Program  {
  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();
    }
}
```

```
AAAAAAAAAAAAAAAAAAAACCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAAAAADDDDDDDDDDDDDDDDDDDD
```

Параллельность не наблюдается, так как каждый поток за выделенный *квант процессорного времени* успевает обработать все 20 итераций.

Если изменить *тело цикла рабочей функции:*

```
static void ThreadFunc(object o)
{ for (int i=0; i<20; i++)
  {  Console.Write(o);
     Thread.Sleep(0);
  }
}
```

```
ACDCDCDCDCDCDCDCDCDCDCDCDCDCDADCDCDADCDADCACACAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBB
```

```
ABCCCCCCCCABCBCABACCABCABABABABABABABABABABABABABABABABABCCCCCCCCCDDDDDDDDDDDDDDDDDDDDD
```

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 (по умолчанию),
- BelowNormal – ниже среднего,
- Lowest – низший.

Для изменения приоритета потока или чтения текущего используется свойство Priority.

Пример. Описать пять потоков с разными приоритетами. Каждый поток увеличивает свой счетчик.

```
using System;
using System.Threading;
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, 15}, Counts: {1}", (ThreadPriority)i, counts[i]);
    }
}
```

```
Thread with priority       Lowest, Counts: 203782912
Thread with priority    BelowNormal, Counts: 76703341
Thread with priority       Normal, Counts: 177292744
Thread with priority    AboveNormal, Counts: 145372094
Thread with priority       Highest, Counts: 65902341
```

```
Thread with priority       Lowest, Counts: 177004644
Thread with priority    BelowNormal, Counts: 128938932
Thread with priority       Normal, Counts: 113779681
Thread with priority    AboveNormal, Counts: 123883892
Thread with priority       Highest, Counts: 106339481
```

```
Thread with priority       Lowest, Counts: 153347089
Thread with priority    BelowNormal, Counts: 132257779
Thread with priority       Normal, Counts: 123948692
Thread with priority    AboveNormal, Counts: 122822651
Thread with priority       Highest, Counts: 147074334
```

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.

Т.о., вопросы эффективной загрузки оптимального числа потоков решаются не программистом, а исполняющей средой.

Для управления списком потоков предусмотрен *класс* ThreadPool, который по мере необходимости уменьшает и увеличивает количество потоков в пуле до максимально допустимого значения.

Для того чтобы запросить поток из пула для обработки вызова метода, можно использовать метод QueueUserWorkItem().

Этот метод перегружен, чтобы в *дополнение* к экземпляру делегата WaitCallback позволить указывать необязательный *параметр* System.Object для специальных данных состояния.

Добавление метода без параметров

```
ThreadPool.QueueUserWorkItem(SomeWork);
```

Добавление метода с параметром

```
ThreadPool.QueueUserWorkItem(SomeWork, data);
```

Максимальное количество рабочих потоков kP и потоков ввода-вывода kPV

ThreadPool.GetMaxThreads(out kP, out kPV);

Пример

```
int kP;
int kPV;
ThreadPool.GetMaxThreads(out kP, out kPV);
Console.WriteLine("\nMaximum worker threads: \t{0}
\nMaximum completion port threads: {1}", kP, kPV);
```

```
Maximum worker threads:          800
Maximum completion port threads: 200
```

Пример. Добавить в пул потоков 10 экземпляров безымянного делегата, объявленного в виде лямбда-выражения. В рабочем элементе осуществить вывод значения индекса (номера) потока и признака того, что поток принадлежит пулу.

…

```
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);
    });
    Thread.Sleep(100);          }
```

```
i: 0, ThreadId: 4, IsPoolThread: True
i: 1, ThreadId: 4, IsPoolThread: True
i: 2, ThreadId: 4, IsPoolThread: True
i: 3, ThreadId: 4, IsPoolThread: True
i: 4, ThreadId: 4, IsPoolThread: True
i: 5, ThreadId: 4, IsPoolThread: True
i: 6, ThreadId: 4, IsPoolThread: True
i: 7, ThreadId: 4, IsPoolThread: True
i: 8, ThreadId: 5, IsPoolThread: True
i: 9, ThreadId: 5, IsPoolThread: True
```

```
i: 0, ThreadId: 4, IsPoolThread: True
i: 1, ThreadId: 5, IsPoolThread: True
i: 2, ThreadId: 4, IsPoolThread: True
i: 3, ThreadId: 5, IsPoolThread: True
i: 4, ThreadId: 5, IsPoolThread: True
i: 5, ThreadId: 4, IsPoolThread: True
i: 6, ThreadId: 5, IsPoolThread: True
i: 7, ThreadId: 4, IsPoolThread: True
i: 8, ThreadId: 5, IsPoolThread: True
i: 9, ThreadId: 4, IsPoolThread: True
```

Все рабочие элементы выполнялись потоками пула (признак IsPoolThread равен true). Всего в обработке участвовало только два потока.

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();
}
```

Каждый рабочий элемент обрабатывался в своем потоке, не входящем в состав пула. 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.

```
i: 8, ThreadId: 11,IsPoolThread: False
i: 4, ThreadId: 7,IsPoolThread: False
i: 7, ThreadId: 10,IsPoolThread: False
i: 1, ThreadId: 4,IsPoolThread: False
i: 2, ThreadId: 5,IsPoolThread: False
i: 1, ThreadId: 3,IsPoolThread: False
i: 4, ThreadId: 6,IsPoolThread: False
i: 7, ThreadId: 9,IsPoolThread: False
i: 9, ThreadId: 12,IsPoolThread: False
i: 6, ThreadId: 8,IsPoolThread: False
```

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.Write(y));
    Console.WriteLine();
    new Thread(() => Console.Write(y)).Start();
 }
```