Многопоточность

Обзор и ключевые понятия

Поток (thread) является единицей обработки данных. Многозадачность (multithreading) — это одновременное выполнение нескольких потоков. Существует два вида многопоточности — совместная (cooperative) и вытесняющая (preemptive). Ранние версии Microsoft Windows, IBM System/38 и OS/2 поддерживали совместную многопоточность. Это означало, что каждый поток отвечал за возврат управления процессору, чтобы тот смог обработать другие потоки. Безусловно, это является источником проблем, т.к. поток может просто «забыть» вернуть управление процессору, и это приведёт к «зависанию» всей операционной системы.

С течением времени, совместная многопоточность (главным достоинством которой является, по сути, простота реализации) потеряла свою популярность. Практически все широко используемые операционные системы стали поддерживать вытесняющую многопоточность (и вытесняющую многозадачность). При этом операционная система отвечает за выдачу каждому потоку определенного количества времени, в течение которого поток может выполняться, — кванта времени (timeslice). Далее процессор переключается между разными потоками, выдавая каждому потоку его квант времени. При использовании вытесняющей многопоточности программист может не заботится о том, как и когда возвращать управление операционной системе – это происходит автоматически, по истечению отведённого кванта времени. Таким образом, переключение потоков происходит автоматически (независимо от «желания» или «нежелания» потока отдавать управление обратно операционной системе). Как следствие, за счёт быстрого переключения потоков создаётся иллюзия параллельного выполнения многопоточной программы. С другой стороны, при использовании вытесняющей многопоточности, потоки, работающие параллельно могут даже не подозревать о существовании друг друга.

Обратите внимание, даже в случае вытесняющей многопоточности, если вы работаете на однопроцессорной машине, то все равно в любой момент времени реально будет исполняться только один поток. Но поскольку интервалы между переключениями процессора от потока к потоку измеряются миллисекундами, возникает иллюзия многозадачности. Чтобы несколько потоков на самом деле работали одновременно, вам потребуется многопроцессорная машина (или машина с многоядерным процессором).

Далее в этой работе мы будем исследовать вытесняющую многопоточноть, характерную (в частности) для Microsoft Windows. В качестве инструментария для работы с потоками мы будем использовать классы платформы Java 7SE и платформы .NET (пространство имён System.Threading), это позволит работать с потоками операционной системы, не прибегая к использованию низкоуровневых функций WinAPI.

Переключение контекста потока

Неотъемлемый атрибут многопоточности — переключение контекста потоков (context switching). Именно переключение контекста вызывает, как правило, трудности при

изучении многопоточного программирования. Итак, что же представляет собой переключение контекста потока. Процессор с помощью аппаратного таймера определяет момент окончания кванта времени, выделенного для данного потока. Когда аппаратный таймер генерирует прерывание, процессор сохраняет содержимое всех регистров для данного потока. Затем процессор перемещает содержимое этих же регистров в структуру данных CONTEXT. При необходимости переключения обратно на выполнение потока, выполнявшегося прежде, процессор выполняет обратную процедуру и восстанавливает содержимое регистров из структуры CONTEXT, ассоциированной с потоком. Весь этот процесс называется переключением контекста.

Пример многопоточного приложения ()

Прежде чем приступить к детальному изучению классов для работы с потоками, создадим простое многопоточное приложение. Обсудив этот пример, мы перейдём к рассмотрению пространства имен System. Threading и класса Thread. В этом примере создаётся второй поток в методе Main. Затем метод, ассоциированный со вторым потоком, выводит строку, сигнализирующую о вызове этого потока. Это действительно простой пример.

Пример Java

```
public class SimpleThread extends Thread {

        // Код потока, который будет выполняться параллельно главному потоку.
        public void run() {
            System.out.println("Второй поток работает...");
        }

        // Главный поток приложения.
        public static void main( String[] args) {

            System.out.println("Главнй поток - создаём ещё один поток.");

            Thread t = new SimpleThread();
            t.start();

            System.out.println("Главный поток - второй поток запущен.");
        }
}
```

Пример C#

```
using System;
using System.Threading;

namespace multithreading_01
{
    class SimpleThreadApp {

        /// <summary>
        /// Код потока, который будет выполняться параллельно главному потоку.
        /// </summary>
        public static void ThreadProc()
        {
            Console.WriteLine("Второй поток работает...");
```

```
        }

        /// <summary>
        /// Главный поток приложения.
        /// </summary>
        public static void Main() {

            ThreadStart worker = new ThreadStart(ThreadProc);

            Console.WriteLine("Главнй поток - создаём ещё один поток.");

            Thread t = new Thread(worker);
            t.Start();

            Console.WriteLine ("Главный поток - второй поток запущен.");
        }
    }
}
```

Результат выполнения этой, казалось бы простой, программы:

```
Главный поток - создаём ещё один поток.

Главный поток - второй поток запущен.

Второй поток работает...

Press any key to continue . . .
```

А может быть и такой:

```
Главный поток - создаём ещё один поток.

Второй поток работает...

Главный поток - второй поток запущен.

Press any key to continue . . .
```

Это действительно не предсказуемо. Дело в том, что в данном случае мы имеем дело с асинхронным выполнением кода. Именно операционная система (а не программа!) определяет, что произойдёт после запуска второго потока: будет продолжено выполнение главного потока или управление будет передано новому потоку.

При рассмотрении этого примера мы несколько раз упоминали понятие «главный поток приложения». Главным потоком называется поток, который запускается операционной системой при старте приложения. В данном случае при старте приложения запускается метод Main(), таким образом можно сказать, что код метода Main() выполняется в главном потоке приложения. Главный поток может запустить несколько других потоков, которые называются дочерними. Все эти потоки (и главный, и дочерние) будут работать параллельно, разделяя между собой ресурсы, выделенные данному приложения операционной системой.

Работа с потоками в Java 7SE

Класс Thread и интерфейс Runnable

Поточная модель Java реализуется иерархией классов, описывающих потоки. Основу этой иерархии составляют класс Thread и интерфейс Runnable. Для создания потока необходимо либо расширить класс Thread, либо реализовать интерфейс Runnable. При этом класс Thread инкапсулирует поток исполнения. При запуске Java-программы начинает выполняться главный поток. Главным потоком можно управлять с помощью методов класса Thread. Ссылку на поток получают с помощью статического метода Thread.currentThread().

Создание потока

Сначала рассмотрим создание потока путем реализации интерфейса Runnable.

Создать поток можно на базе любого класса, который реализует интерфейс Runnable. При реализации интерфейса Runnable достаточно определить всего один метод run(), который определяет точку входа в поток. Общая последовательность действий при создании нового потока путем реализации интерфейса Runnable следующая.

1. Определяется класс, реализующий интерфейс Runnable. В этом классе определяется метод run().

2. В этом классе создается объект класса Thread. Конструктору класса передается два аргумента: объект класса, реализующего интерфейс Runnable, и текстовая строка — название потока.

3. Для запуска потока из объекта класса Thread вызывается метод start().

Один экземпляр Runnable можно передать нескольким объектам Thread:

```
MyRunnable r = new MyRunnable();
Thread foo = new Thread(r);
Thread bar = new Thread(r);
Thread bat = new Thread(r);
```

Это означает что несколько потоков будут делать одну и ту же работу.

Практически также создаются потоки наследованием класса Thread. Класс Thread сам наследует интерфейс Runnable. Поэтому принцип создания потока остается неизменным, просто вместо непосредственной реализации в создаваемом классе интерфейса Runnable этот интерфейс реализуется путем расширения (наследования) класса Thread. Реализация метода run() в классе Thread не предполагает каких-либо действий. Выход из ситуации -- путем создания подкласса. Как и в случае с интерфейсом Runnable, в подклассе, создаваемом на основе класса Thread, необходимо переопределить метод run() и запустить его унаследованным из Thread методом start().

Управление потоками

Рассмотрим некоторые приёмы работы с классом Thread:

```
public class SimpleThread2 extends Thread {

        // Код потока, который будет выполняться параллельно главному потоку.
        public void run() {
            System.out.println("Второй поток работает...");
            int sleepTime = 5000;
            System.out.printf("Второй поток засыпает на %d секунд.%n",
                  sleepTime/1000);
            try {
                Thread.sleep(sleepTime);
            } catch ( InterruptedException ex ) {}
            System.out.println("Второй поток снова работает.");
        }

        // Главный поток приложения.
        public static void main( String[] args) {

            System.out.println("Главнй поток - создаём ещё один поток.");

            Thread t = new SimpleThread();
            t.start();

            System.out.println("Главный поток - второй поток запущен.");
        }
}
```

Во многом этот пример эквивалентен предыдущему. Что же делает функция run(). Итак, результаты работы это программы:

Главный поток - создаём ещё один поток.

Главный поток - второй поток запущен.

Второй поток работает.

Второй поток засыпает на 5 секунд.

Второй поток снова работает.

Вам уже известно, что порядок строк, выводимых программой, может немного отличаться от приведенного выше. Тем не менее, строка «Второй поток снова работает.» не появится раньше, чем через 5 секунд после старта программы. Дело в том, что второй поток «засыпает» на 5 секунд, т.е. сообщает ОС, что в ближайшие 5000 миллисекунд ему не понадобятся кванты времени для работы. Как мы видим, делает от это сам (т.е. «добровольно», по собственному желанию) вызывая метод sleep().

В принципе, есть два варианта вызова метода Thread.sleep(). Первый — вызов Thread. sleep() со значением 0. При этом вы заставите текущий поток освободить неиспользованный остаток своего кванта. С другой стороны, передача в качестве параметра положительного числа приведёт к остановке выполнения потока на заданное количество миллисекунд.

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления Thread.State:

NEW – поток создан, но еще не запущен;

RUNNABLE – поток выполняется;

BLOCKED – поток блокирован;

WAITING – поток ждет окончания работы другого потока;

TIMED_WAITING – поток некоторое время ждет окончания другого потока;

TERMINATED — поток завершен.

Получить значение состояния потока можно вызовом метода getState().

При создании потока он получает состояние "новый" (NEW) и не выполняется. Для перевода потока из состояния "новый" в состояние "работоспособный" (RUNNABLE) следует выполнить метод start(), который вызывает метод run() – основной метод потока.

Поток переходит в состояние "неработоспособный" (WAITING) вызовом метода wait() или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания (TIMED_WAITING) с помощью методов sleep(long millis) и wait(long timeout), при выполнении которого может генерироваться прерывание InterruptedException. Вернуть потоку работоспособность после вызова метода wait() можно методами notify() или notifyAll().

Поток переходит в "пассивное" состояние (TERMINATED), если вызван метод interrupt() или метод run() завершил выполнение. После этого, чтобы запустить поток еще раз, необходимо создать новый объект потока. Метод interrupt() успешно завершает поток, если он находится в состоянии "работоспособность". Если же поток неработоспособен, то метод генерирует исключительные ситуации разного типа в зависимости от способа остановки потока.

Планирование потоков

Потоку можно назначить приоритет от 1 (константа MIN_PRIORITY) до 10 (MAX_PRIORITY) с помощью метода setPriority(int prior). Получить значение приоритета можно с помощью метода getPriority().

Потоки объединяются в группы потоков. После создания потока нельзя изменить его принадлежность к группе.

```
ThreadGroup tg = new ThreadGroup( "Группа потоков 1");

Thread t0 = new Thread(tg, "поток 0");
```

Все потоки, объединенные группой, имеют одинаковый приоритет. Чтобы определить, к какой группе относится поток, следует вызвать метод getThreadGroup().

Если поток до включения в группу имел приоритет выше приоритета группы потоков, то после включения в группу значение его приоритета станет равным приоритету группы. Поток со значением приоритета более низким, чем приоритет группы после включения в группу, значения своего приоритета не изменит. Поток с более высоким приоритетом, как правило, монополизирует вывод на консоль.

Пример программы, в которой конкурируют потоки с разными приоритетами:

```
class MyThread extends Thread {
    MyThread( String name ) {
        super(name);
    }
    public void run() {
        for ( int i = 0; i < 20; i++) {
            System.out.print( getName() + " " + i + "; ");
            try {
                sleep(1); //test with sleep(0)
            } catch ( InterruptedException e ) {
                System.err.print("Error" + e);
            }
        }
        System.out.println( getThreadGroup() + ":" +
                  getPriority() +":" + getName() +  " finished." );
    }
}

public class PriorThread {
    public static void main( String[] args) {
        Thread min = new MyThread("Min");//1
        Thread max = new MyThread("Max");//10
        Thread norm = new MyThread("Norm");//5
        min.setPriority(Thread.MIN_PRIORITY);
        max.setPriority(Thread.MAX_PRIORITY);
        norm.setPriority(Thread.NORM_PRIORITY);
        max.start();
        norm.start();
        min.start();
   }
}
```

Результаты работы это программы:

Max 0; Min 0; Norm 0; Norm 1; Min 1; Max 1; Max 2; Norm 2; Min 2;

Norm 3; Max 3; Min 3; Norm 4; Max 4; Min 4; Norm 5; Max 5; Min 5;

Max 6; Norm 6; Min 6; Norm 7; Min 7; Max 7; Norm 8; Max 8; Min 8;

Norm 9; Max 9; Min 9; Norm 10; Max 10; Min 10; Norm 11; Min 11; Max 11;

Norm 12; Max 12; Min 12; Norm 13; Max 13; Min 13; Max 14; Min 14; Norm 14;

Norm 15; Max 15; Min 15; Norm 16; Max 16; Min 16; Norm 17; Min 17; Max 17;

Norm 18; Min 18; Max 18; Norm 19; Min 19; Max 19;

java.lang.ThreadGroup[name=main,maxpri=10]:10:Max finished.

java.lang.ThreadGroup[name=main,maxpri=10]:1:Min finished.

java.lang.ThreadGroup[name=main,maxpri=10]:5:Norm finished.

Приостановить (задержать) выполнение потока можно с помощью метода sleep(время задержки) класса Thread. Альтернативный способ состоит в вызове метода yield(), который сообщает планировщику выделить квант времени потоку ожидающему в очереди.

Ожидание завершения выполнения потоков является одним из важнейших механизмов работы с потоками. Метод join() блокирует работу потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока.

Чтобы определить состояние потока используется метод isAlive().

```
Thread ct = Thread.currentThread();

ct.isAlive();
```

Если поток запушен или заблокирован, то возвращается значение true, а если поток является созданным (еще не запущенным) или остановленным, то возвращается значение false.

Работа с потоками в C#

Класс System.Threading.Thread

Операционная система позволяет программисту осуществлять управление потоками (при наличии соответствующих прав у пользователя от лица которого выполняется программа). Как правило, это происходит посредством вывоза соответствующих функций ядра операционной системы, что осуществляется через интерфейс прикладного программирования (Application Programming Interface - API) операционной системы. В случае с Microsoft Windows – WinAPI. Классы платформы .NET скрывают низкоуровневые API ОС, позволяя использовать более простую объектно-ориентированную модель. Работы с потоками в .NET осуществляется при помощи классов пространства имён System.Threading, в основном – при помощи класса System.Threading.Thread.

Создание потоков и объектов Thread

Как отмечалось, работа с потоками происходит посредством класса System.Threading.Thread. Вы можете создавать экземпляры Thread двумя способами. Один вы уже видели: это создание нового потока и получение объекта Thread, позволяющего манипулировать новым потоком. Другой способ получить объект Thread для потока, который выполняется в данный момент, — вызов статического метода Thread.CurrentThread (результат – объект Thread для того потока, который вызвал этот метод).

Управление потоками

Рассмотрим некоторые приёмы работы с классом Thread:

```
using System;
using System.Threading;

namespace multithreading_02
{
    class ThreadSleepApp
    {
        /// <summary>
        /// Код потока, который будет выполняться параллельно главному потоку.
        /// </summary>
        public static void ThreadProc()
        {
            Console.WriteLine("Второй поток работает.");
            int sleepTime = 5000;
            Console.WriteLine("Второй поток засыпает на {0} секунд.",
                  sleepTime/1000);
            Thread.Sleep(sleepTime);
            Console.WriteLine("Второй поток снова работает.");
        }

        /// <summary>
        /// Главный поток приложения.
        /// </summary>
        public static void Main()
        {
            Console.WriteLine("Главный поток - создаём ещё один поток.");
            Thread t = new Thread(ThreadProc);
            t.Start();
            Console.WriteLine("Главный поток - второй поток запущен.");
        }
    }
}
```

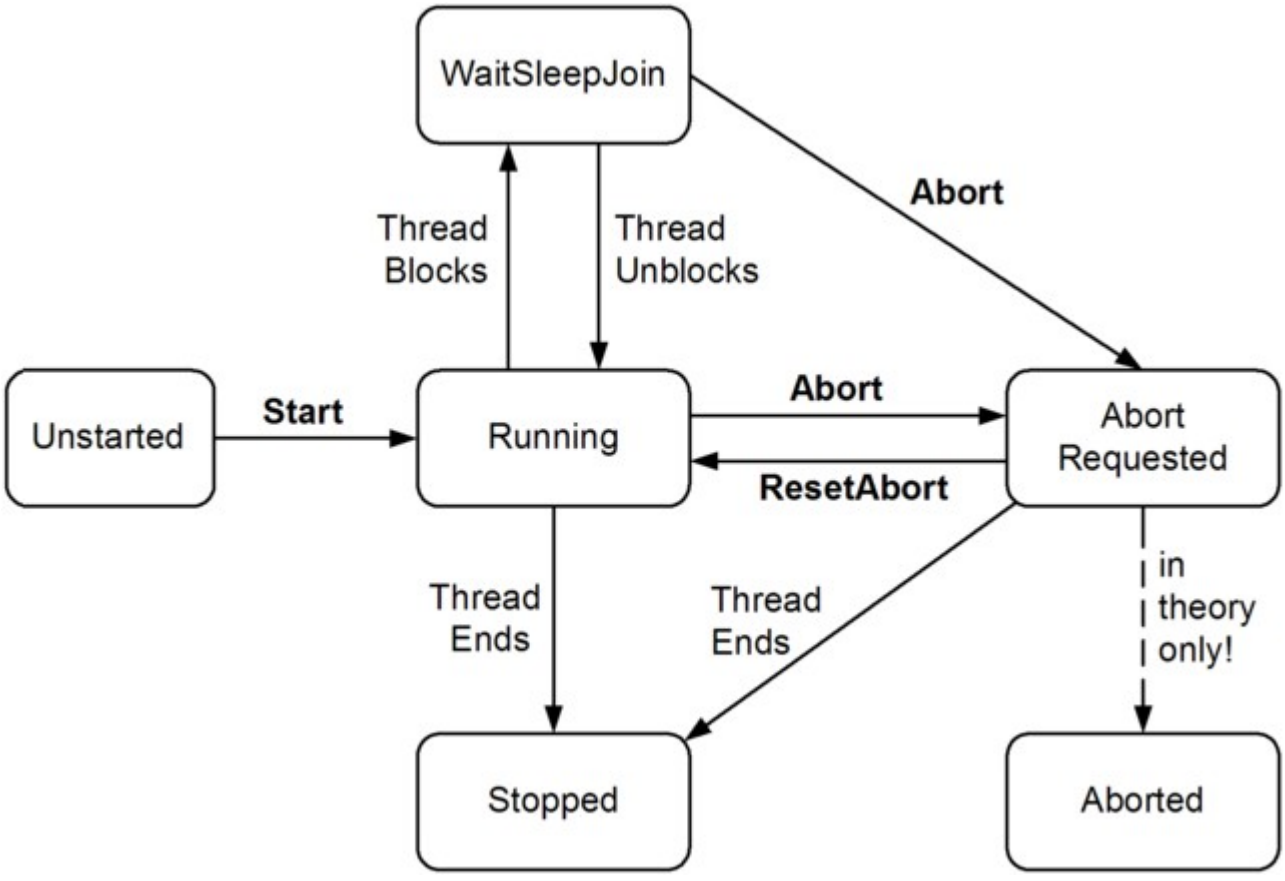
Как видите, мы ещё больше упростили создание объекта Thread в Main(). В остальном функция Main() эквивалента предыдущему примеру. Остаётся выяснить что же делает функция ThreadProc(). Итак, результаты работы это программы:

```
Главный поток - создаём ещё один поток.

Главный поток - второй поток запущен.

Второй поток работает.

Второй поток засыпает на 5 секунд.

Второй поток снова работает.

Press any key to continue . . .
```

Вам уже известно, что порядок строк, выводимых программой, может немного отличаться от приведенного выше. Тем не менее, строка «Второй поток снова работает.» не появится раньше, чем через 5 секунд после старта программы. Дело в том, что второй поток «засыпает» на 5 секунд, т.е. сообщает ОС, что в ближайшие 5000 миллисекунд ему не

понадобятся кванты времени для работы. Как мы видим, делает от это сам (т.е. «добровольно», по собственному желанию) вызывая метод Sleep().

В принципе, есть два варианта вызова метода Thread.Sleep(). Первый — вызов Thread. Sleep() со значением 0. При этом вы заставите текущий поток освободить неиспользованный остаток своего кванта. С другой стороны, передача в качестве параметра положительного числа приведёт к остановке выполнения потока на заданное количество миллисекунд. При передаче же значения Timeout.Infinite поток будет приостановлен на неопределенно долгий срок, пока это состояние потока не будет прервано другим потоком, вызвавшим метод приостановленного потока Thread.Interrupt() – прервать работу потока, находящегося в состоянии WaitSleepJoin. Перевод потока в состояние бесконечного ожидания и вывод его из этого состояния при помощи метода Thread.Interrupt() не является хорошей (и как следствие широко распространённой практикой), но это приводит нас к пониманию того, что поток обладает «состоянием»: он может «выполняться», «быть остановленным», «спать» и так далее. Полный граф состояний потока приведён ниже. Обратите внимание, что переход между состояниями можно инициировать соответствующими методами. Узнать текущее состоянии потока можно с помощью его свойства ThreadState.



Вернёмся к управлению потоками. Второй способ приостановить исполнение потока — вызов метода Thread.Suspend(). Между этими методиками (Interrupt() и Suspend()) есть несколько важных отличий. Во-первых, можно вызвать метод Thread. Suspend() для потока, выполняющегося в текущий момент, или для любого другого потока. Во-вторых, если таким образом приостановить выполнение потока, любой другой поток способен возобновить

его выполнение с помощью метода Thread.Resume(). Обратите внимание, что, когда один поток приостанавливает выполнение другого, первый поток не блокируется. Возврат управления после вызова происходит немедленно. Кроме того, единственный вызов Thread.Resume() возобновит исполнение данного потока независимо от числа вызовов метода Thread.Suspend(), выполненных ранее.

Таким образом, пара методов Suspend()/Resume() является предпочтительной, когда речь идёт о временной приостановке работы потока. Здесь мы имеем в виду, что этот механизм является удобным, когда речь идёт об управлении одним потоком выполнением другого потока. Когда же поток хочет приостановить своё собственное выполнение, то более применим метод Sleep(), как сделано в предыдущем примере.

Уничтожение потоков

Уничтожить поток можно вызовом метода Thread.Abort(). Исполняющая среда насильно завершает выполнение потока, генерируя исключение ThreadAbortException (это исключение возникнет в коде потока, для которого был вызван метод Abort()). Даже если поток попытается обработать ThreadAbortException, исполняющая среда этого не допустит. Однако она исполнит код из блока finally потока, выполнение которого прервано, если этот блок присутствует (что позволяет среагировать на аварийное завершение). Проиллюстрируем это следующим примером. После запуска главный поток создаёт и запускает второй (рабочий) поток. Далее, выполнение метода Main приостанавливается на 5 секунд, чтобы дать исполняющей среде время для запуска рабочего потока. После запуска рабочий поток считает до десяти, останавливаясь после каждого отсчета на секунду. Когда выполнение метода Main возобновляется после пятисекундной паузы, он прерывает выполнение рабочего потока (после этого исполняется блок finally).

```
using System;
using System.Threading;

class ThreadAbortApp {

    public static void ThreadProc()
    {
        try
        {
            Console.WriteLine("Рабочий поток запущен.");
            for (int i = 0; i < 10; i++) {
                Thread. Sleep(1000);
                Console.WriteLine("Рабочий поток -> {0}", i);
            }
            Console.WriteLine("Рабочий поток завершён");
        }
        catch(ThreadAbortException e)
        {
            // ThreadAbortException здесь обработано не будет!
        }
        finally
        {
           Console.WriteLine("В рабочем потоке возникло необработанное исключение!");
        }
    }

    public static void Main()
```

```
    {
        Console.WriteLine("Главный поток - запускаем рабочий поток.");
        Thread t = new Thread(ThreadProc);
        t.Start();

        Console.WriteLine("Главный поток - засыпаем на 5 секунд.");
        Thread.Sleep(5000);

        Console.WriteLine("Главный поток - прерываем рабочий поток.");
        t.Abort();
    }
}
```

Результаты работы программы:

```
Главный поток - запускаем рабочий поток.

Рабочий поток запущен.

Главный поток - засыпаем на 5 секунд.

Рабочий поток -> 0

Рабочий поток -> 1

Рабочий поток -> 2

Рабочий поток -> 3

Рабочий поток -> 4

Главный поток - прерываем рабочий поток.

В рабочем потоке возникло необработанное исключение!

Press any key to continue . . .
```

Планирование потоков

При переключении процессора по окончании выделенного потоку кванта времени, процесс выбора следующего потока, предназначенного для исполнения, далеко не произволен. У каждого потока есть приоритет, указывающий процессору, как должно планироваться выполнение этого потока по отношению к другим потокам системы. Для потоков, создаваемых в период выполнения, уровень приоритета по умолчанию равен Normal. Для просмотра и установки этого значения служит свойство Thread.Priority. Свойству Thread.Priority можно присвоить значение типа Thread.ThreadPriority, которое представляет собой перечисление, определяющее значения Highest, AboveNormal, Normal, BelowNormal и Lowest (наивысший, выше нормального, нормальный, ниже нормального, низший).

Рассмотрим пример:

```
using System;
using System.Threading;

class ThreadAbortApp
{
    /// <summary>
```

```
    /// Код рабочего потока.
    /// </summary>
    public static void ThreadProc(object args)
    {
        string name = args.ToString();
        Console.WriteLine("Рабочий поток запущен ({0}).", name);
        for (int i = 0; i < 10; i++)
        {
            // Цикл с бесполезной работой (для загрузки процессора).
            for (int j = 0; j < 1000000; j++)
                Math.Sin(Math.Cos(j) + Math.Atan(j) * Math.Sin(Math.Cos(j)
                  + Math.Atan(i)));
            Console.WriteLine("{0} -> {1}", name, i);
        }
        Console.WriteLine("Рабочий поток завершён ({0}).", name);
    }

    public static void Main()
    {
        // Разрешить данному процессу использовать только один процессор.
        System.Diagnostics.Process.GetCurrentProcess().ProcessorAffinity
            = new IntPtr(0x0001);

        // Создаём два рабочих потока.
        Console.WriteLine("Главный поток - запускаем рабочие потоки.");
        Thread t1 = new Thread(ThreadProc);
        Thread t2 = new Thread(ThreadProc);

        // Устанавливаем приоритеты потоков.
        t1.Priority = ThreadPriority.BelowNormal;
        t2.Priority = ThreadPriority.AboveNormal;

        // Запускаем потоки.
        t1.Start("первый поток");
        t2.Start("второй поток");

        // Ждём завершения обоих потоков.
        t1.Join();
        t2.Join();
        Console.WriteLine("Главный поток - выполнение обоих потоков завершено.");
    }
}
```

Результаты работы программы (потоки запущены с одинаковым приоритетом):

```
Главный поток - запускаем рабочие потоки.

Рабочий поток запущен (первый поток).

Рабочий поток запущен (второй поток).

первый поток -> 0

второй поток -> 0

первый поток -> 1

второй поток -> 1

первый поток -> 2
```

```
второй поток -> 2
первый поток -> 3
второй поток -> 3
второй поток -> 4
первый поток -> 4
второй поток -> 5
первый поток -> 5
второй поток -> 6
первый поток -> 6
второй поток -> 7
первый поток -> 7
второй поток -> 8
первый поток -> 8
второй поток -> 9
Рабочий поток завершён (второй поток).
первый поток -> 9
Рабочий поток завершён (первый поток).
Главный поток - выполнение обоих потоков завершено.
Press any key to continue . . .
```

Потоки запущены с разными приоритетами:

```
Главный поток - запускаем рабочие потоки.
Рабочий поток запущен (первый поток).
Рабочий поток запущен (второй поток).
второй поток -> 0
второй поток -> 1
второй поток -> 2
второй поток -> 3
второй поток -> 4
второй поток -> 5
```

```
второй поток -> 6

второй поток -> 7

второй поток -> 8

второй поток -> 9

Рабочий поток завершён (второй поток).

первый поток -> 0

первый поток -> 1

первый поток -> 2

первый поток -> 3

первый поток -> 4

первый поток -> 5

первый поток -> 6

первый поток -> 7

первый поток -> 8

первый поток -> 9

Рабочий поток завершён (первый поток).

Главный поток - выполнение обоих потоков завершено.

Press any key to continue . . .
```

Обратите внимание, что второй поток завершился раньше, несмотря на то, что был запушен позже первого.

В приведённом примере также демонстрируются ещё два новых механизма работы с потоками:

- передача параметров потоку;
- ожидание завершения выполнения потоков.

Каждому потоку передаётся его «имя» - это позволяет различать вывод первого и второго потоков на консоли. В реальных программах для этой цели используется свойство Thread.Name, что избавляет от необходимости передавать лишние параметры потоку.

Ожидание завершения выполнения потоков также является одним из важнейших механизмов работы с потоками. В .NET это осуществляется при помощи метода Thread.Join(), который блокирует вызывающий поток до завершения потока, метод Join() которого был вызван. Т.е. в нашем примере:

```
public static void Main()
{
	...
	t1.Join();
	t2.Join();
	...
}
```

выполнение метода Main() будет приостановлено сначала до тех пор, пока первый поток не завершится, а затем – до завершения и второго потока. Таким образом, мы дождёмся завершения обоих потоков.

См. также:

- System.Threading - пространство имен
- Thread - класс
- Создание потоков (Руководство по программированию на C#)
- Использование потоков (Руководство по программированию на C#)
- Синхронизация потоков (Руководство по программированию на C#)

Задание категории А.

Написать приложение, содержащее не менее двух тредов. Каждый из этих тредов должен искать файлы:

- **с определенным заданным шаблоном;**
- **содержащие в своем составе определенную строку;**
- **начиная с определенного директория;**
- **обеспечить возможность поиска в поддиректориях.**

Каждый тред должен помещать результаты своей работы в свой список типа List.

Задание категории Б.

Разработать многопоточное приложение, моделирующее движение бильярдных шаров по игровому столу. Поведение каждого шара (т.е. вычисление новых координат и перерисовка) программируется как отдельный поток. На игровом столе действуют обычные физические законы - шары отскакивают от стенок и углов стола так, что угол падения равен углу отражения, единственным исключением для данной задачи является отсутствие взаимодействий между шарами (т.е. проще говоря, они не сталкиваются).

При запуске процесса моделирования каждый шар получает некоторый (случайный) импульс, под действием которого он движется по инерции, постепенно останавливаясь. Когда шар останавливается, соответствующий поток должен завершиться. Приложение следит за тем, чтобы был хотя бы один поток, который ещё не закончил свою работу. Когда все потоки будут завершены, требуется выдать соответствующее сообщение.

Программа должна предоставлять пользователю возможность приостановить/продолжить или прервать процесс имитации движения.

Пример подобного приложения находится в папке либо архиве MultiThreadBalls.

Недостатки данного приложения:

- **шары движутся бесконечно долго, с постоянной скоростью;**
- **скорость всех шаров одинакова и не от чего не зависит;**
- **в программе используются целочисленные координаты и, следовательно, проиллюстрирована возможность движения только по прямым с углами ±45°.**