MERA

Многопоточность в CLR и C#

Что такое поток?

Поток – это сущность операционной системы, набор инструкций и данных, выполняемый на процессоре.

По сути – это **виртуальный процессор**.

Процесс – набор ресурсов, используемый отдельным экземпляром приложения.

У каждого процесса есть отдельное виртуальное адресное
пространство и как минимум один поток.

Приоритеты потоков

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below Normal	Normal	Above Normal	High	Realtime
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
Above Normal	5	7	9	11	14	25
Normal	4	6	8	10	13	24
Below Normal	3	5	7	9	12	23
Lowest	2	4	6	8	11	22

Когда использовать потоки

- Длительные вычисления в фоновом режиме
- Изоляция одного кода от другого
- Распараллеливание интенсивных вычислений

Недостатки использования потоков

- Значительное увеличение сложности программы
- Повышенный расход ресурсов процессора на создание и переключение между потоками

Потоки в C#

```
namespace System.Threading
{
    // Summary:
    //     Creates and controls a thread, sets its priority, and gets its status.
    [ClassInterface(ClassInterfaceType.None)]
    [ComDefaultInterface(typeof(_Thread))]
    [ComVisible(true)]
    public sealed class Thread : CriticalFinalizerObject, _Thread
    {
        [SecuritySafeCritical]
        public Thread(ParameterizedThreadStart start);

        [SecuritySafeCritical]
        public Thread(ThreadStart start);

     …

    }
}
```

Пример создания потока

```
class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread(WriteY);
        t.Start();            // Выполнить WriteY в новом потоке
        while (true)
            Console.Write("x"); // Все время печатать 'x'
    }

    static void WriteY()
    {
        while (true)
            Console.Write("y"); // Все время печатать 'y'
    }
}
```

Вывод:

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxyyyy
yyyyyyyyyyyyxxxxy...
```

Основные и фоновые потоки

Процесс не будет завёршен, пока хотя бы один его основной поток ещё исполняется. По умолчанию все создаваемые потоки являются основными.

Фоновые потоки не продлевают жизнь процессу. Они завершаются автоматически при завершении всех основных потоков.

Статус потока переключается с основного на фоновый при помощи свойства **IsBackground** класса **Thread**.

Разделение данных между потоками

```
class ThreadTest
{
    bool done;

    static void Main()
    {
        ThreadTest tt = new ThreadTest(); // Создаем общий объект
        new Thread(tt.Go).Start();
        tt.Go();
    }

    void Go()
    {
        if (!done) { done = true; Console.WriteLine("Done"); }
    }
}
```

Так как оба потока вызывают метод **Go()** одного и того же экземпляра **ThreadTest**, они разделяют поле **done.**
Результат – “**Done”**, напечатанное один раз вместо двух.

Завершение работы потока

Поток завершает свою работу при выходе из основного метода. Можно попытаться принудительно завершить поток с помощью метода **Abort()**. Ожидать завершения другого потока можно с помощью метода **Join()**.

```
Thread t = new Thread(Go);
t.Start();
// doing stuff...
t.Abort();
t.Join();       // Ожидаем завершения потока
```

Обработка исключений в потоках

```
try
{
    new Thread(Go).Start();
}
catch (Exception ex)
{
    // Сюда мы никогда не попадем!
    Console.WriteLine("Исключение!");
}
```

Начиная с .NET 2.0 необработанное исключение в любом потоке приводит к закрытию всего приложения.

Мораль – обрабатывать исключения необходимо в самом методе потока.

Thread.Abort и Thread.Interrupt

Thread.Abort генерирует в потоке исключение **ThreadAbortException**. Даже если это исключение будет перехвачено, оно будет сгенерировано снова в конце блока **catch**. Повторную генерацию этого исключения можно предотвратить путём вызова **Thread.ResetAbort**.

Вызов **Interrupt** для блокированного потока принудительно освобождает его с генерацией **ThreadInterruptedException**. Если **Interrupt** вызывается для неблокированного потока, поток продолжает своё исполнение до точки следующей блокировки, где и генерируется исключение.

Потоковая безопасность

Потокобезопасный код – это код, не имеющий никаких неопределенностей при любых сценариях многопоточного исполнения.

Потокобезопасность достигается прежде всего блокировками и сокращением возможностей взаимодействия между потоками.

Метод, который является потокобезопасным при любых сценариях, называется реентерабельным.

Недостатки потокобезопасного кода

MERA

- Трудоёмкость написания
- Производительность (даже если многопоточность реально не используется)
- Потокобезопасный тип не обязательно делает саму программу потокобезопасной

Блокировка потока

Блокированный поток немедленно перестает получать время CPU, устанавливает свойство ThreadState в **WaitSleepJoin** и остается в таком состоянии, пока не разблокируется.

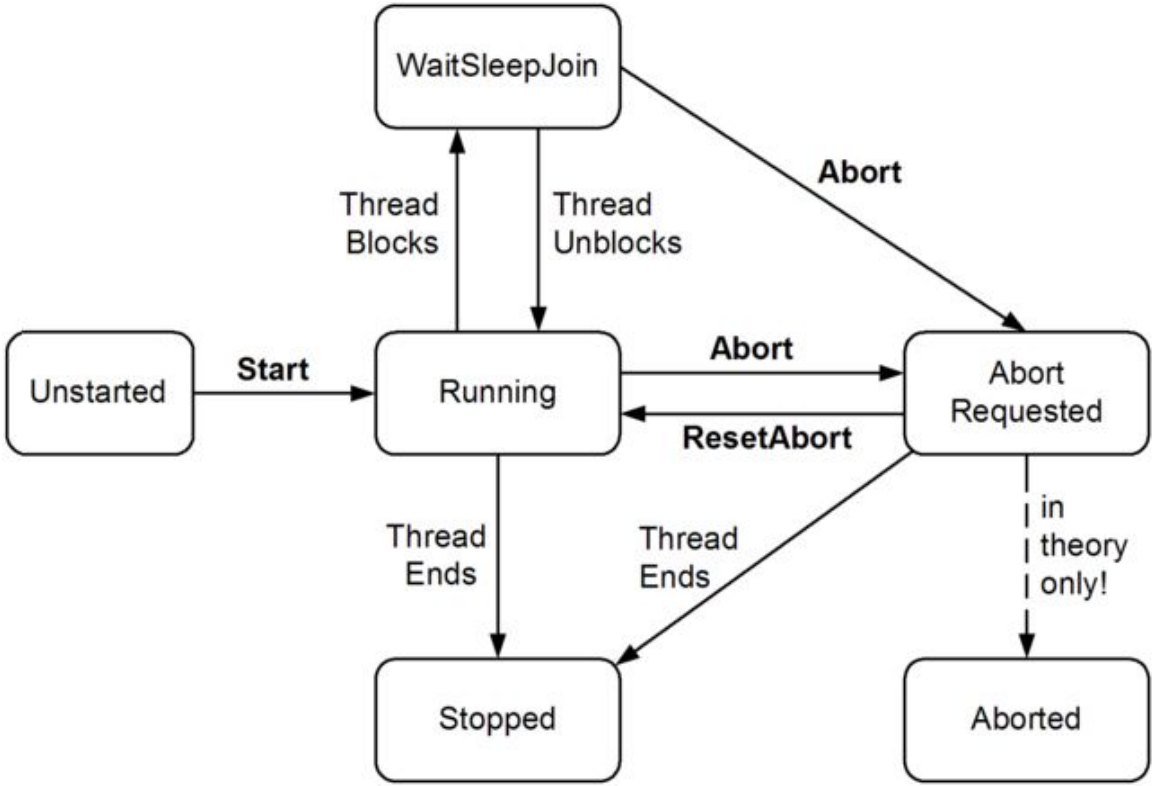
Разблокировка может произойти в следующих случаях:

- выполнится условие разблокировки;
- истечёт таймаут операции (если он был задан);
- по прерыванию через **Thread.Interrupt**;
- по аварийному завершению через **Thread.Abort**.

Основные средства синхронизации потоков

Sleep	Блокировка на указанное время.
Join	Ожидание окончания другого потока.
lock (Monitor)	Гарантирует, что только один поток может получить доступ к ресурсу или секции кода.
Mutex	То же. Может использоваться для предотвращения запуска нескольких экземпляров приложения.
Semaphore	Гарантирует, что не более заданного числа потоков может получить доступ к ресурсу или секции кода.
EventWaitHandle	Позволяет потоку ожидать сигнала от другого потока.
Interlocked	Выполнение простых не блокирующих атомарных операций.

Состояния потока

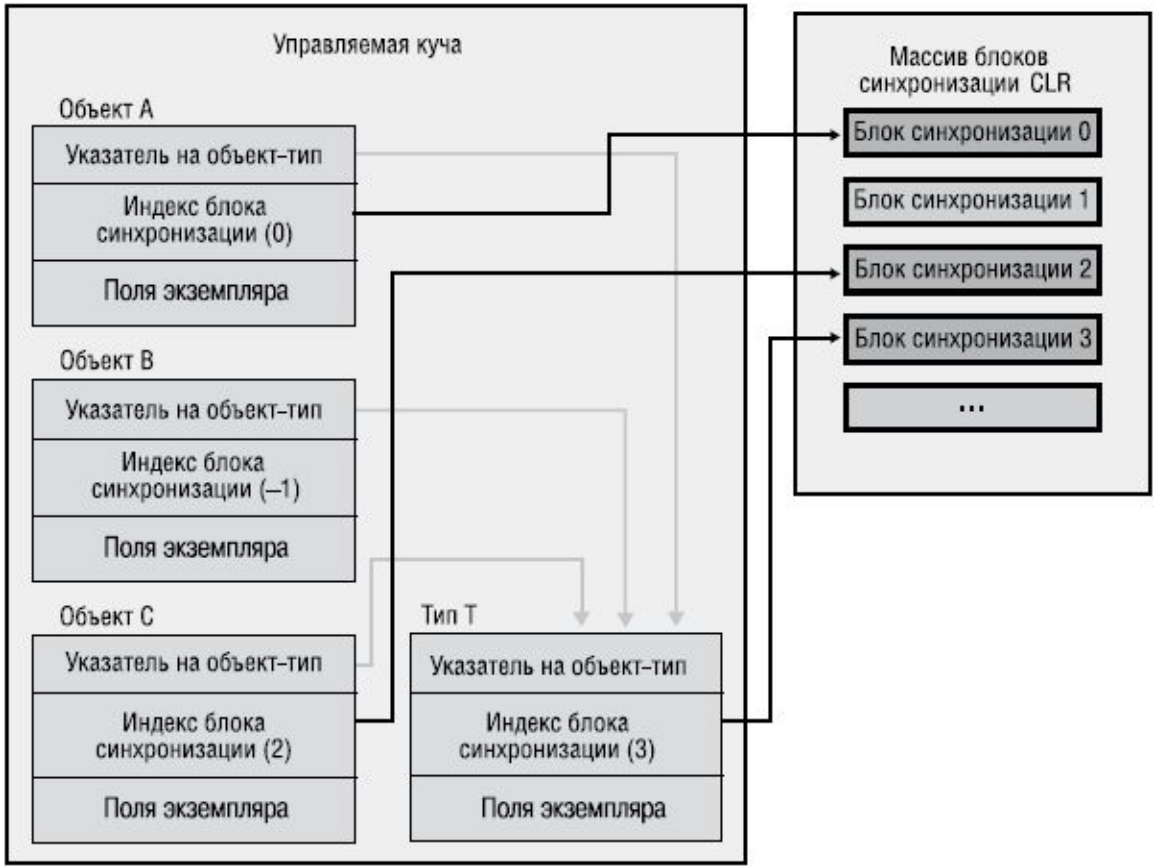


Конструкция lock (Monitor.Enter – Monitor.Exit)

```
class ThreadSafe
{
    static object locker = new object();
    static int val1, val2;

    static void Go()
    {
        lock (locker)
        {
            if (val2 != 0)
                Console.WriteLine(val1 / val2);

            val2 = 0;
        }
    }
}
```

```
class ThreadSafe
{
    static object locker = new object();
    static int val1, val2;

    static void Go()
    {
        Boolean lockTaken = false;
        object obj = (System.Object)locker;
        try
        {
            Monitor.Enter(obj, ref lockTaken);
            if (val2 != 0)
                Console.WriteLine(val1 / val2);

            val2 = 0;
        }
        finally
        {
            if (lockTaken) Monitor.Exit(obj);
        }
    }
}
```

Блоки синхронизации



Использование Mutex

```
// Используем уникальное имя приложения,
// например, с добавлением имени компании
static Mutex mutex = new Mutex(false, "MERA.NN TestConsoleApplication");

static void Main()
{
    // Ожидаем получения мьютекса 5 сек - если уже есть запущенный
    // экземпляр приложения - завершаемся.
    if(!mutex.WaitOne(TimeSpan.FromSeconds(5)))
    {
        Console.WriteLine("В системе запущен другой экземпляр программы!");
        return;
    }

    try
    {
        Console.WriteLine("Работаем - нажмите Enter для выхода...");
        Console.ReadLine();
    }
    finally { mutex.ReleaseMutex(); }
}
```

Иерархия WaitHandle

MERA

System.Threading.WaitHandle
- System.Threading.EventWaitHandle
  - System.Threading.AutoResetEvent
  - System.Threading.ManualResetEvent
- System.Threading.Mutex
- System.Threading.Semaphore

Использование EventWaitHandle

MERA

```
class BasicWaitHandle
{
    static EventWaitHandle wh = new EventWaitHandle(false, EventResetMode.AutoReset);

    static void Main()
    {
        new Thread(Waiter).Start();
        Thread.Sleep(1000);                  // Подождать некоторое время...
        wh.Set();                             // ОК – можно разбудить
    }

    static void Waiter()
    {
        Console.WriteLine("Ожидание...");
        wh.WaitOne();                         // Ожидать сигнала
        Console.WriteLine("Получили сигнал");
    }
}
```

AutoResetEvent и ManualResetEvent

MERA

```
new AutoResetEvent(false) ==
new EventWaitHandle(false, EventResetMode.AutoReset)
```

```
new ManualResetEvent(false) ==
new EventWaitHandle(false, EventResetMode.ManualReset)
```

Создание межпроцессных EventWaitHandle

```
EventWaitHandle wh = new EventWaitHandle(false, EventResetMode.AutoReset,
  "MyCompany.MyApp.SomeName");
```

Конструктор **EventWaitHandle** также позволяет создавать именованные **EventWaitHandle**, способные действовать через границы процессов.

Если задаваемое имя уже используется на компьютере, возвращается ссылка на существующий **EventWaitHandle**, в противном случае операционная система создает новый.

Semaphore

```
class SemaphoreTest
{
    static Semaphore s = new Semaphore(3, 3);  // Available=3; Capacity=3

    static void Main()
    {
        for (int i = 0; i < 10; i++)
            new Thread(Go).Start();
    }

    static void Go()
    {
        while (true)
        {
            s.WaitOne();
            // Только 3 потока могут находиться здесь одновременно
            Thread.Sleep(100);
            s.Release();
        }
    }
}
```


Вопросы?
MERA

Список литературы

- Джеффри Рихтер. CLR via C# (3е издание)
- Эндрю Троелсен. Язык программирования C# и платформа .NET 4.5
- RSDN Magazine. Работа с потоками в C#, часть 1. http://rsdn.ru/article/dotnet/CSThreading1.xml
- Хабрахабр. Процессы и потоки in-depth. Обзор различных потоковых моделей. http://habrahabr.ru/post/40227/

Задание для работы в аудитории

Написать программу, которая в реальном времени будет рисовать на форме узор из разноцветных окружностей.
Для этого создать несколько потоков, которые и будут отвечать за отрисовку, каждый своим цветом.
Позиции окружностей выбирать рандомно.
Сделать возможность ставить отрисовку на паузу, и возможность продолжения рисовки.
По желанию сделать возможность прямо в программе выбирать количество потоков и цвета для каждого из них.

Задание на дом

Написать программу, которая в реальном времени будет обрабатывать очень длинный список чисел, находя числа, удовлетворяющие некоторому условию (например, числа делящиеся на 3, или простые числа, и т.д.). Обработку списка производить в нескольких потоках, каждому потоку выделить свой диапазон. Найденные числа выводить в реальном времени в таблицу на форме. В таблице должно быть число, позиция числа в списке, и номер (или имя) потока, нашедшего число. Сделать возможность в самой программе задавать число потоков и диапазоны.