Лекция 7 Понятие нитей в системном программировании.

7.1 Понятие нити.

Работа процессора компьютера заключается в выполнении команд (инструкций), которые формируются после компиляции программы. Эта последовательность инструкций программы называется потоком команд управления программы. Термин «поток управления» применяется в языке C++. В языке C# принято называть поток команд управления программы нитью – сравнивают с нитью (thread), на которую «нанизаны» инструкции выполняемые процессором компьютера.

Операционная система называется однопрограммной, если в ней может существовать только одна нить.

Операционная система называется мультипрограммной, если в ней может одновременно существовать несколько нитей разных программ.

Эффективность работы компьютера во многом определяется загрузкой процессора компьютера. Поэтому важнейшей задачей мультипрограммной ОС является диспетчеризация нитей программ таким образом, чтобы обеспечить эффективную загрузку процессора.

Нитью программы может являться любой независимо выполняемый фрагмент программы.

Рассмотрим пример учебной программы, в которой вычисляется сумма двух чисел a и b.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
  static void Main(string[] args)
  {
   int a,b,c;
   string buf;
   Console.Write("Введите целое значение a ");
   buf = Console.ReadLine();
   a = Convert.ToInt32(buf);
   Console.Write("Введите целое значение b ");
   buf = Console.ReadLine();
   b = Convert.ToInt32(buf);
   c = a + b;
   Console.WriteLine("a+b={0}", c);
   Console.WriteLine("Для продолжения нажмите клавишу Enter");
   Console.ReadLine();
  }
 }
}
```

Каждое действие этой программы всегда однозначно определено и выполняется одно за другим – программа содержит только одну нить.

Изменим программу, включив в нее функцию вычисления суммы a и b.

```
using System;
namespace ConsoleApplication1
{
 class Program
 {
  public static int sum(int a, int b)
  { return a + b; }
  static void Main(string[] args)
  {
   int a, b, c;
   string buf;
   Console.Write("Введите целое значение a ");
   buf = Console.ReadLine();
   a = Convert.ToInt32(buf);
   Console.Write("Введите целое значение b ");
   buf = Console.ReadLine();
   b = Convert.ToInt32(buf);
   c = sum(a,b);
   Console.WriteLine("a+b={0}", c);
   Console.WriteLine("Для продолжения нажмите клавишу Enter");
   Console.ReadLine();
  }
 }
}
```

Если при выполнении программы функция **Main** ждет выполнение функции **sum**, то программа остается с одной нитью.

Если при выполнении программы функция **Main** не ждет выполнение функции **sum**, а продолжает выполняться, то в программе образуются две нити – главная, нить функции **Main,** и нить функции **sum**. В этом случае значение переменной **c**, выведенное на экран монитора функцией **Main**, может не соответствовать сумме переменных **a и b**.

Еще больше нитей может существовать в программах, использующих операторы условных переходов.

Для организации правильной работы программы необходимо научится передавать данные между нитями, совместно использовать данные и синхронизировать (упорядочить) работу различных нитей или нитевых функций.

7.2 Требования к методам, используемым параллельными нитями

Рассмотрим, каким требованиям должны удовлетворять методы, чтобы их можно было «безопасно» вызывать параллельными нитями.

Для этого определим понятие контекста нити – как область памяти компьютера, к которой может обращаться нить во время своего исполнения.

Рассмотрим работу следующей функции:

```
public static int funk1(int n)
{
```

```
  if (n >= 0) n --;
  if (n  < 0) n ++;
  return n;
}
```

При вызове этой функции создается копия переменной **n,** которая изменяется в соответствии с условием и результат возвращается нити. Далее копия переменной удаляется из памяти. Такая функция называется безопасной для нити.

Изменим функцию **funk1** так, чтобы она работала с глобальной переменной **n**.

```
int n;
. . .
public static void funk2()
{
  if (n >= 0) n --;
  if (n  < 0) n ++;
}
```

При параллельном вызове функции **funk2** несколькими нитями может произойти некорректное изменение значения переменной **n** (может возникнуть изменение переменной **n** больше чем на единицу). Такие функции называют не безопасными для нити.

В некоторых языках программирования, например, в C++, существует понятие статических переменных функций. Статические переменные объявляются внутри функции (локальные переменные), но значение этих переменных сохраняется в памяти по завершении работы функции и может быть использовано при повторном вызове функции. То есть такая статическая переменная ведет себя внутри функции как глобальная переменная программы. При параллельном обращении к такой функции нескольких нитей значение статической переменной может не соответствовать количеству вызовов функции той или иной нити.

В общем случае функция называется безопасной или реентерабельной, если она удовлетворяет следующим требованиям:

- не использует глобальные переменные, значения которых изменяются параллельно работающими нитями;
- не использует статические переменные, определенные внутри функции и изменяемые параллельно работающими нитями;
- не возвращает указатели на статические данные, определенные внутри функции.

Перечисленные требования к безопасным функциям можно обеспечить, если использовать различные приемы блокировки доступа к ресурсам функций различных нитей, которые они совместно используют.

7.3 Состояния нити

Состояние нити зависит от готовности процессора компьютера работать с программой, включающей нить и от готовности самой программы.

Г5отовность процессора определяется его «выделением» на исполнение данной программы – т.е. процессор получает квант времени для работы с данной программой.

Готовность программы зависит от получения ей всех ресурсов необходимых для выполнения программы. В зависимости от процессора и программы возможны следующие состояния потока:

- нить готова к выполнению («не выделен», «готова»);
- нить блокирована («не выделен», «не готова»);
- нить выполняется («выделен», «готова»).

Обычно к перечисленным состояниям нити добавляются еще два – новая (нить создается, но не готова) и завершено (состояние соответствующее окончанию работы нити).

В программах, содержащих несколько нитей, работа которых взаимосвязана, существует еще два состояния нити – приостановка выполнения нити и возобновление выполнения нити.

С учетом вех перечисленных состояний нити можно нарисовать следующую модель ее поведения.

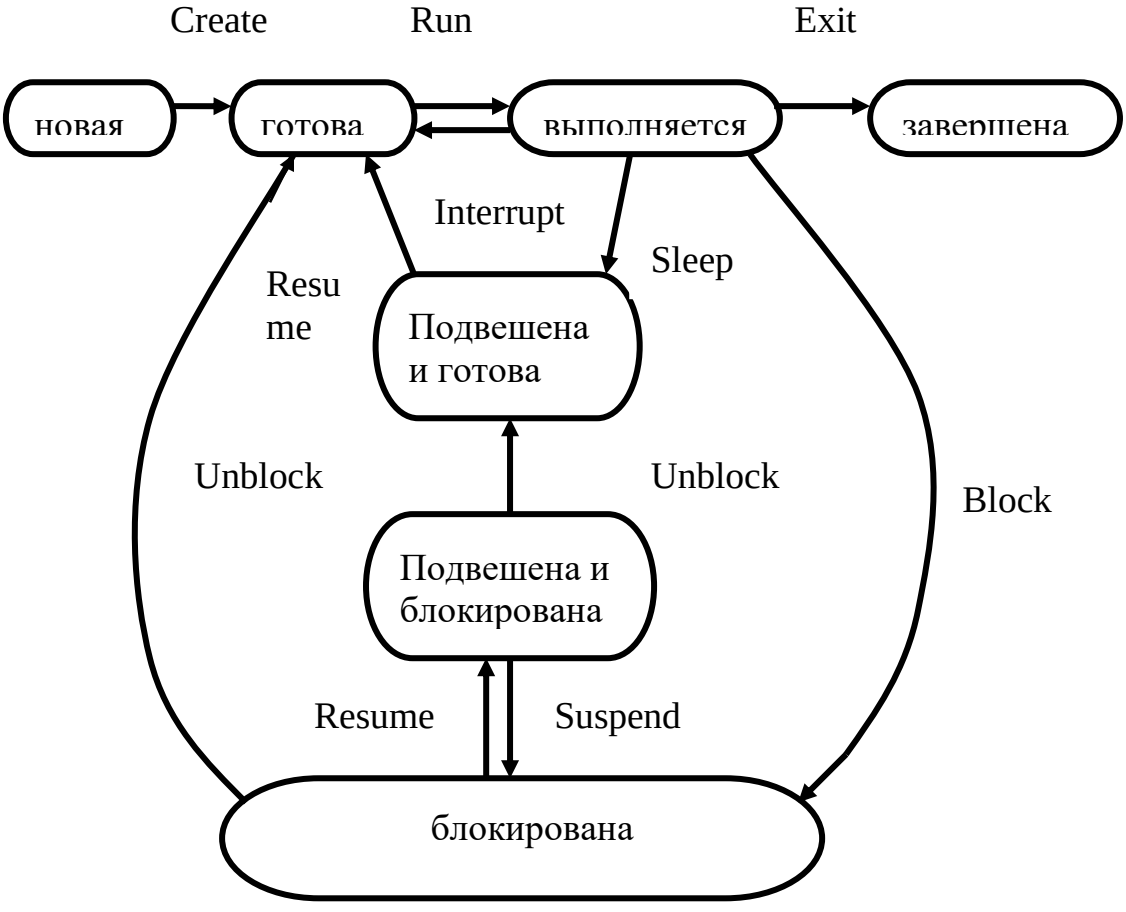


Рисунок 7.1 – Модель семи состояний нити.

Операция Create переводит нить из состояния «новая» в состояние «готова».

Операция Run переводит нить из состояния «готова» в состояние «выполняется» - нити выделяется процессорное время.

Операция Exit переводит нить из состояния «выполняется» в состояние «завершена».

Операция Interrupt переводит нить из состояния «выполняется» в состояние «готова» - обычно эта операция выполняется над нитью, когда истекло процессорное время, выделенное на исполнение нити.

Операция Resume возобновляет исполнение нити.

Операция Sleep приостанавливает исполнение нити.

Операция Unblock переводит нить из состояния «блокирована» в состояние «готова».

Операция Block переводит нить из состояния «выполняется» в состояние «блокирована» - обычно эта операция выполняется над нитью, когда она ждет наступление некоторого события, например, ввода данных или освобождение некоторого ресурса.

Операция Suspend приостанавливает исполнение нити.

7.4 Диспетчеризация и планирование нитей

Для пояснения принципов диспетчеризация и планирование нитей будем считать, что компьютер имеет только один процессор.

Для организации мультипрограммного режима работы процессора его время работы делится на кванты – интервалы, которые выделяются нитям для их работы (в Windows это примерно 20 – 30 миллисекунд). По истечении кванта времени исполнение нити прерывается и процессор назначается другой нити. Распределением квантов времени между нитями занимается специальная программа – менеджер нитей.

Когда менеджер нитей переключает процессор на исполнение другой нити, он должен выполнить следующие действия:

- сохранить контекст прерываемой нити;
- восстановить контекст запускаемой нити;
- передать управление запускаемой нити.

Если все обслуживаемые нити имеют одинаковый приоритет исполнения, то алгоритм управления нитями легко реализуется с помощью «очереди» по принципу FIFO (first in – first out) пришел первым- первым вышел, и прерванные нити становятся в конец очереди.

Схематично обслуживание нитей представлено на рисунке 7.2

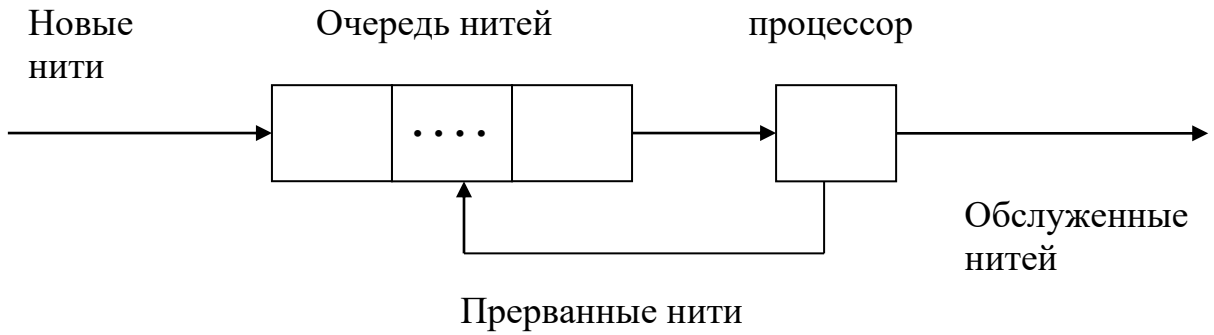


Рисунок 7.2 – Обслуживание нитей

Если нити имеют разные приоритеты, то формируются очереди нитей с одинаковыми приоритетами. Простейший алгоритм обслуживания нескольких очередей основан на обслуживании очередей с учетом приоритета их нитей. Схематично обслуживание нитей представлено на рисунке 7.3

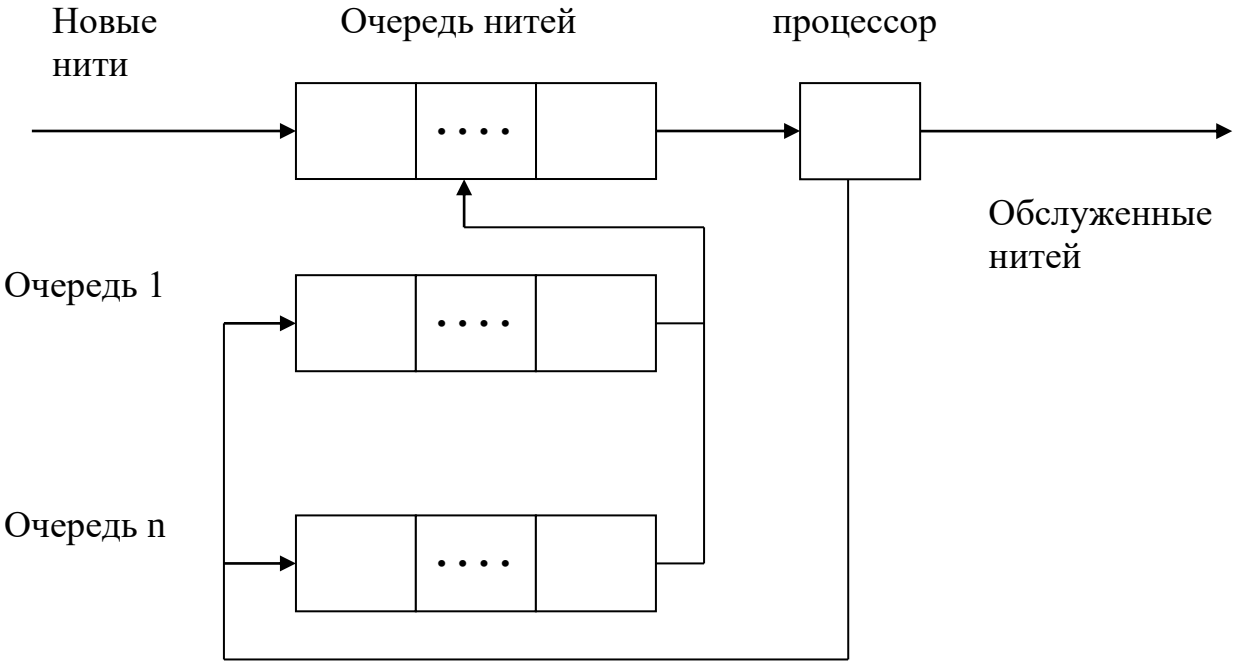


Рисунок 7.3 – Обслуживание нескольких очередей с разными приоритетами нитей

Алгоритмы управления нитями должны учитывать следующие параметры системы:

- время загрузки микропроцессора должно быть максимальным;
- время нахождения нити в системе должно быть минимальным;
- время ожидания нитей должно быть минимальным;
- пропускная способность системы должна быть максимальной;

- время реакции системы на обслуживание заявки должно быть минимальным.

На многопроцессорных компьютерах менеджер нитей использует как квантование времени работы нитей, так и параллельность работы процессоров, когда разные нити выполняют код на разных процессорах.

Необходимость квантования времени все равно остается, так как операционная система должна обслуживать как свои собственные (системные) нити, так и нити других приложений.

7.5 Определение нити

Приложение в Windows может состоять из одного или нескольких процессов, каждый из которых, может порождать одну или несколько нитей. Каждой нити выделяются некоторые ресурсы операционной системы, которые представляются в виде объекта.

Нитью в Windows называется объект, которому операционная система выделяет процессорное время для работы с системными ресурсами.

Каждой нити принадлежат следующие ресурсы:

- код исполняемой функции;
- набор регистров процессора;
- стек для работы приложения;
- стек для работы операционной системы;
- маркер доступа с информацией для системы безопасности.

Эти ресурсы образуют контекст нити.

Любому объекту в системе Windows присваивается свой регистрационный номер – дескриптор. Кроме дескриптора каждой нити в Windows присваивается свой идентификатор. Идентификаторы нитей используются служебными программами и позволяют отслеживать работу нитей.

В Windows различают нити двух типов – системные и пользовательские.

Системные нити запускаются ядром операционной системы и служат для выполнения различных системных задач.

Пользовательские нити запускаются приложениями и служат для решения задач пользователя.

В технологии .NET эта схема выделения ресурсов несколько изменена – каждый процесс приложения .NET может состоять из одного или нескольких доменов приложения, а в рамках домена может работать одна или несколько нитей (все свойства нити как объекта при этом сохраняются).

Ресурсы каждого домена полностью изолированы друг от друга. Различные домены приложения не могут совместно использовать никакие данные, будь то статические поля или глобальные переменные, которые могут использоваться разными нитями в рамках только общего домена.

Таким образом, домен это дополнительный уровень «изоляции» или «защиты» данных в .NET, который создается между процессом и нитью.

Необходимо отметить, что для некоторых приложений в .NET в рамках домена можно создавать пулы нитей – это специальные мини очередь для текущих нитей, которые создаются методом ThreadPool.QueueUserWorkItem вместо создания и запуска объекта Thread.

Как правило, диспетчер пула нитей устанавливает количество нитей в пуле автоматически (по умолчанию это значение равно 50). Можно самим установить верхний предел количества нитей в пуле методом ThreadPool.SetMaxThreads.

Обычно пулы применяются, когда необходимо организовать работу произвольного числа параллельно работающих приложений, например, на веб-серверах. При работе Web Services, ASP.NET, серверов WCF и т.п. пулы нитей выделяются автоматически.

7.6 Создание нити

При запуске любой программы на языке C# автоматически создается главная нить программы. В консольном приложении главная нить программы соответствует методу Main.

Для запуска **дополнительной нити** необходимо создать объект класса Thread. При создании объекта нити конструктору класса Thread передается делегат, в котором описан метод ThreadStart, позволяющий запускать в дополнительной нити метод, имя которого указано в качестве формального параметра метода делегата. Например,

```
Thread Pervij_dop = new Thread(new ThreadStart(Poick));
```

где Pervij_dop – имя первой дополнительной нити;

Poick – имя метода, который будет запущен в дополнительной нити.

Обычно делегат работает «по умолчанию» и создание объекта нити выглядит следующим образом:

```
Thread Pervij_dop = new Thread(Poick);.
Pervij_dop.Start();
```

Создавая консольное приложение, работающее с нитями, необходимо подключать пространство имен System.Threading.

Класс Thread имеет набор свойств и методов, некоторые из которых представлены следующим списком:

CurrentThread – статическое свойство, предназначенное только для чтения, которое возвращает ссылку на нить, выполняемую в настоящее время;

Sleep() – это статический метод, который приостанавливает выполнение текущей нити на указанное пользователем время;

IsAlive – это обычное свойство (требует создание объекта нити), возвращает **true** или **false** в зависимости от того, запущена нить или нет;

IsBackground – это обычное свойство, предназначенное для получения или установки значения, которое показывает, является ли эта нить фоновой;

Name – это обычное свойство для установки дружественного текстового имени нити;

Priority – обычное свойство, позволяющее получить/установить приоритет нити (используются значения из перечисления ThreadPrority);

ThreadState – обычное свойство, позволяющее получить информацию о состоянии нити (используются значения из перечисления ThreadState);

Interrupt() – обычный метод, прерывающий работу текущей нити;

Join()– обычный метод, который ждет появления другой нити или указанный промежуток времени;

Resume() – обычный метод, который продолжает работу после приостановки нити;

Start() – обычный метод, который начинает выполнение нити, определенной делегатом ThreadStart

Suspend() – обычный метод, который приостанавливает выполнение нити. Если выполнение нити уже приостановлено, то метод игнорируется.

Все свойства и методы класса Thread приведены в приложении данной лекции.

Рассмотрим чисто учебный пример создания объекта дополнительной нити и ее запуска из главной нити программы. Для визуального контроля работы главной и дополнительной нитей, традиционно в них используется цикл for с выводом некоторых символов или цифр. Не будем отступать от традиций:

```
using System;
using System.Threading;

namespace ConsoleApplication1
{
    class Thread1
    {
        static void WriteCimB()
        {
            for (int i = 1; i <= 100; i++) Console.Write("B");
        }
        static void Main()
        {
            Thread t_dop = new Thread(WriteCimB);
            t_dop.Start();
            for (int i = 1; i <= 100; i++) Console.Write("A");
            Console.ReadLine();
        }
    }
}
```

Работа программы:

```
AAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBB
BBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBB
```

BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBB

Главная нить предназначена для вывода в консольное окно 100 символов «A», одновременно она создает дополнительную нить, которая также выводит в консольное окно один символ 100 раз – «B». Из работы программы видно, что они работают одновременно – прерываются циклы **for** только квантами времени, выделяемыми нитям. На компьютере с одним процессором каждой нити для работы выделяется квант времени (в Windows это 20 мс.).

Можно исследовать состояние дополнительной нити, включив в программу следующие строки:

```
Console.WriteLine("состояние доп нити ={0}", t_dop.IsAlive);
Console.WriteLine("приоритет доп нити ={0}", t_dop.Priority);
```

Работа программы:

состояние доп нити =True
приоритет доп нити =Normal
AAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBB

Приложение лекции 7

Методы и свойства класса Thread

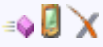
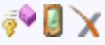
Конструкторы

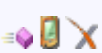
	Имя	Описание
	Thread	Перегружен. Инициализирует новый экземпляр класса Thread.

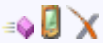
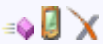
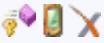
В начало страницы

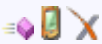
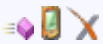
Методы

	Имя	Описание
	Abort	Перегружен. Вызывает исключение ThreadAbortException в вызвавшем его потоке для того, чтобы начать процесс завершения потока. Вызов данного метода обычно завершает поток.

	AllocateDataSlot	Выделяет неименованную область данных всем потокам. Для улучшения производительности используйте поля, отмеченные атрибутом ThreadStaticAttribute.
	AllocateNamedDataSlot	Выделяет именованную область данных всем потокам. Для улучшения производительности используйте поля, отмеченные атрибутом ThreadStaticAttribute.
	BeginCriticalRegion	Уведомляет хост, что выполнение близится ко входу к область кода, в которой эффекты прерывания выполнения или неуправляемого выполнения могут повлиять на другие задачи в домене приложения.
	BeginThreadAffinity	Уведомляет хост, что управляемый код близок к выполнению инструкций, зависящих от идентификации текущего потока операционной системы.
	EndCriticalRegion	Уведомляет хост, что выполнение близится ко входу к область кода, в которой эффекты прерывания выполнения или неуправляемой ошибки ограничены текущей задачей.
	EndThreadAffinity	Уведомляет хост об окончании выполнения кодом инструкций, которые зависят от идентификатора текущего потока в операционной системе.
	Equals	Определяет, равен ли заданный объект Object текущему объекту Object. (Унаследовано от Object.)
	Finalize	Освобождает все ресурсы, используемые классом

		CriticalFinalizerObject. (Унаследовано от CriticalFinalizerObject.) В .NET Compact Framework 3.5 этот член наследуется от Object.Finalize(). В XNA Framework 1.0 этот член наследуется от Object.Finalize().
	FreeNamedDataSlot	Удаляет связь между названием и областью для всех потоков в процессе. Для улучшения производительности используйте поля, отмеченные атрибутом ThreadStaticAttribute.
	GetApartmentState	Возвращает значение типа ApartmentState, показывающее состояние апартамента.
	GetCompressedStack	**Устаревшее.** Возвращает объект CompressedStack, который может быть использован для отслеживания стека текущего потока.
	GetData	Извлекает значение из заданной области текущего потока, внутри текущей области текущего потока. Для улучшения производительности используйте поля, отмеченные атрибутом ThreadStaticAttribute.
	GetDomain	Возвращает текущую область, в которой выполняется текущий поток.
	GetDomainID	Возвращает уникальный идентификатор домена приложения.
	GetHashCode	Возвращает хэш-код текущего потока. (Переопределяет Object.GetHashCode().)

		В .NET Compact Framework 3.5 этот член наследуется от Object.GetHashCode(). В XNA Framework 1.0 этот член наследуется от Object.GetHashCode().
	GetNamedDataSlot	Ищет именованную область данных. Для улучшения производительности используйте поля, отмеченные атрибутом ThreadStaticAttribute.
	GetType	Возвращает объект Type для текущего экземпляра. (Унаследовано от Object.)
	Interrupt	Прерывает работу потока, находящегося в состоянии **WaitSleepJoin**.
	Join	Перегружен. Блокирует вызывающий поток до завершения потока.
	MemberwiseClone	Создает неполную копию текущего объекта Object. (Унаследовано от Object.)
	MemoryBarrier	Синхронизирует доступ к памяти следующим образом: процессор, выполняющий текущий поток, не способен упорядочить инструкции в порядке обращения к памяти, до вызова MemoryBarrier выполняет после того, как память получит доступ после вызова MemoryBarrier.
	ResetAbort	Отменяет метод Abort, запрошенный для текущего потока.
	Resume	**Устаревшее.** Возобновляет приостановленную работу потока.
	SetApartmentState	Задает состояние апартамента

		потока до его запуска.
	SetCompressedStack	**Устаревшее.** Применяет записанное значение CompressedStack к текущему потоку.
	SetData	Задает данные в указанной области для текущей области потока, выполняющегося в данный момент. Для улучшения производительности используйте поля, отмеченные атрибутом ThreadStaticAttribute.
	SetProcessorAffinity	В .NET Compact Framework для Xbox 360, задает сходство процессоров для управляемого потока. Сходство процессоров определяет процессоры, на которых будет выполнен поток.
	Sleep	Перегружен. Блокирует текущий поток на заданное количество миллисекунд.
	SpinWait	Вынуждает поток ожидать количество отсчетов, определенное параметром *iterations*.
	Start	Перегружен. Позволяет планировать выполнение потока.
	Suspend	**Устаревшее.** Приостанавливает работу потока; если работа потока уже приостановлена, не оказывает влияния.
	ToString	Возвращает объект String, который представляет текущий объект Object. (Унаследовано от Object.)
	TrySetApartmentState	Задает состояние апартамента потока до его запуска.
	VolatileRead	Перегружен. Считывает значение поля. Это значение является

		последним, записанным каким-либо из процессоров компьютера, независимо от количества процессоров и от состояния кэш-буфера процессоров.
	VolatileWrite	Перегружен. Записывает значение непосредственно в поле, так что оно становится видимым для всех процессоров компьютера.

В начало страницы

Свойства

	Имя	Описание
	ApartmentState	**Устаревшее.** Получает или задает состояние апартамента для данного потока.
	CurrentContext	Возвращает текущий контекст, в котором выполняется поток.
	CurrentCulture	Получает или задает язык и региональные параметры для текущего потока.
	CurrentPrincipal	Получает или задает текущего участника потока (для безопасности на основе ролей).
	CurrentThread	Возвращает выполняющийся в данный момент поток.
	CurrentUICulture	Получает или задает текущие язык и региональные параметры, используемые диспетчером ресурсов для поиска ресурсов, связанных с языком и региональными параметрами, во время выполнения.
	ExecutionContext	Возвращает объект ExecutionContext, содержащий сведения о различных контекстах текущего потока.
	IsAlive	Возвращает значение, показывающее

		статус выполнения текущего потока.
	IsBackground	Получает или задает значение, показывающее, является ли поток фоновым.
	IsThreadPoolThread	Возвращает значение, показывающее, принадлежит ли поток к группе управляемых потоков.
	ManagedThreadId	Возвращает уникальный идентификатор текущего управляемого потока.
	Name	Получает или задает имя потока.
	Priority	Получает или задает значение, указывающее на планируемый приоритет потока.
	ThreadState	Возвращает значение, содержащее состояния текущего потока.

В начало страницы