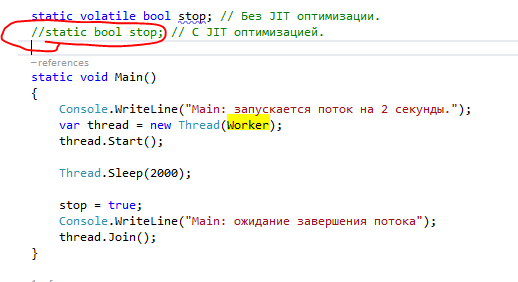
Volatile

Поля, объявленные как volatile, не проходят оптимизацию компилятором, которая предусматривает доступ посредством отдельного потока.

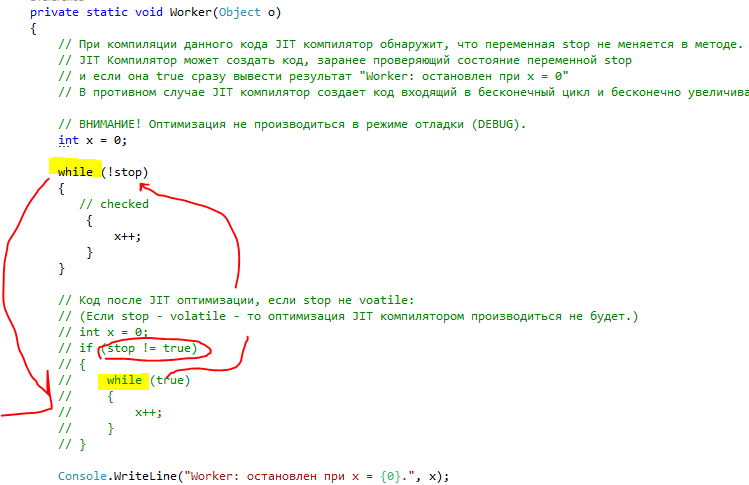
Это гарантирует наличие наиболее актуального значения в поле в любое время.

Ключевое слово гарантирует что при чтении и записи манипуляция будет происходить непосредственно с памятью а не со значениями, которые кэшированы в регистры процессора.

Т.е. если в нашем классе мы создадим статическое поле и это поле будет применено в методе который будет запускаться в отдельном потоке, при запуске на Release поле перед входом в метод закэшируется в регистре процессора



Компилятор обнаружил что переменная не меняется в методе, Проверил поле один раз



Ключевое слово volatile можно применять к полям следующих типов:

1. Ссылочные типы.

2. Типы, такие как sbyte, byte, short, ushort, int, uint, char, float и bool.

3. Тип перечисления с одним из следующих базовых типов: byte, sbyte, short, ushort, int или uint.

4. Параметры универсальных типов, являющиеся ссылочными типами.

Ключевое слово volatile можно применить только к полям класса или структуры.

Локальные переменные не могут быть объявлены как volatile.

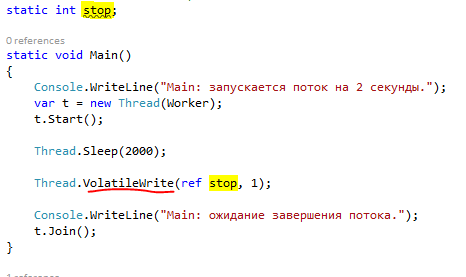
Thread.VolatileWrite() и Thread.VolatileRead()

Если нет возможности изменить поле (добавив ключевое слово volatile), то неоходимо применить статические методы класса Thread.VolatileRead() и Thread.VolatileWrite()

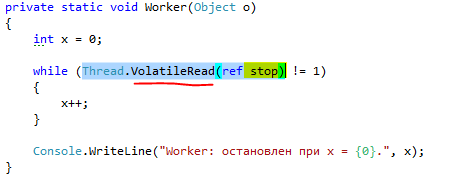
Thread.VolatileWrite(ref stop, 1); - записывает значение в переменную

Thread.VolatileRead(ref stop) – считывает значение из переменной

Запись

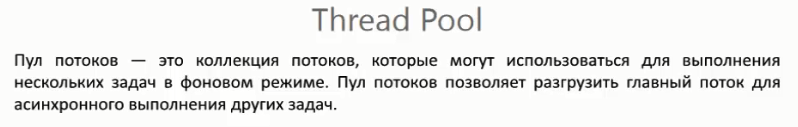


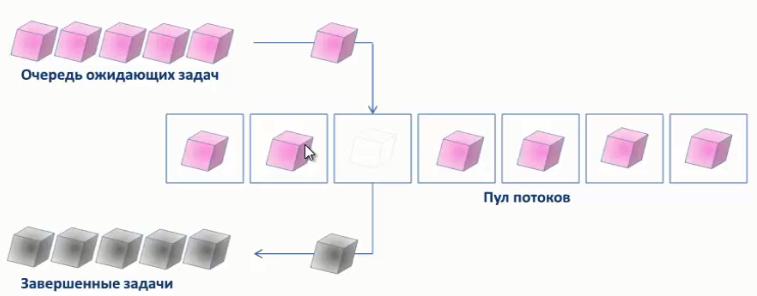
Чтение



ThreadPool

static class ThreadPool





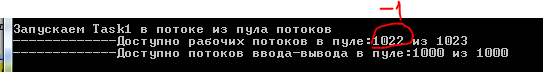
ThreadPool.QueueUserWorkItem(new WaitCallback(Task1)); - поставить метод Task1 на выполнение в пул потоков (new WaitCallback – класс делегат)

ThreadPool.GetAvailableThreads(out availableWorkThreads, out availableIOThreads); - GetAvailableThreads возвращает значение, переменная, указанная параметром workerThreads(1-я), содержит число дополнительных рабочих потоков, которые могут быть запущены, а переменная, указанная параметром completionPortThreads (2-я), содержит число дополнительных потоков асинхронного ввода/вывода, которые могут быть запущены

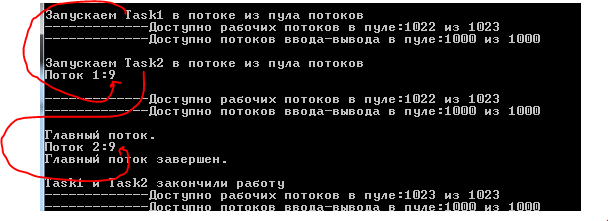
ThreadPool.GetMaxThreads(out maxWorkThreads, out maxIOThreads); - 1й Максимальное количество рабочих потоков в пуле потоков. 2й Максимальное количество потоков асинхронного ввода/вывода в пуле потоков

Изначально результат этих двух методов



После запуска в пуле потоков нового потока

Два метода работали в одном потоке Thread.CurrentThread.ManagedThreadId



!!!!!!!!!!!!!!!!!!

ЕЛСИ ПОТОКИ НЕ ВЫПОЛНЯЮТСЯ ОДНОВРЕМЕННО ТО ОНИ БУДУТ ВЫПОЛНЕНЫ В ОДНОМ ПОТОКЕ И МЕНЕДЖЕРУ ПОТОКОВ НЕ НУЖНО СОЗДАВАТЬ ДВА ПОТОКА (трудоемкая задача)

Пул потока предлагает механизм вызова метода

ThreadPool.RegisterWaitForSingleObject(auto, callback, null, 1000, false);

auto - от кого ждать сингнал

callback - что выполнять

null - 1-й аргумент Callback метода

1000 - интервал между вызовами Callback метода

если true - вызвать Callback метод один раз. Если false - вызывать Callback метод с интервалом.

AutoResetEvent auto = new AutoResetEvent(false);

**WaitOrTimerCallback** callback = new WaitOrTimerCallback(CallbackMethod);

ThreadPool.RegisterWaitForSingleObject(auto, callback, null, Timeout.Infinite, true);

waitHandle.Unregister(auto); - снятие с регистрации сигнала

1 аргумент переданный в метод

2 метод вызван в результате таймаута или сигналом



Примитивы синхранизации доступа к разделяемым ресурсам

Mutex

Mutual Exclusion (Взаимное Исключение).

Использование Mutex для синхронизации доступа к защищенным ресурсам.

Mutex - Примитив синхронизации, который также может использоваться в межпроцессорной синхронизации.

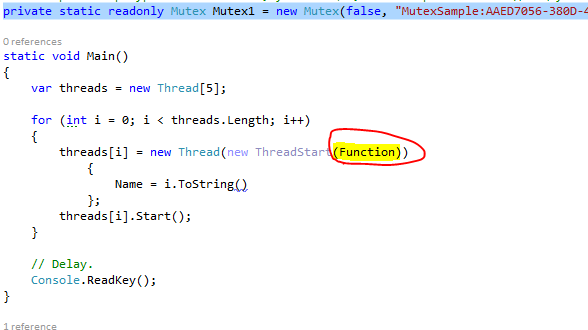
функционирует аналогично AutoResetEvent но снабжен дополнительной логикой:

1. Запоминает какой поток им владеет. ReleaseMutex не может вызвать поток, который не владеет мьютексом.

2. Управляет рекурсивным счетчиком, указывающим, сколько раз поток-владелец уже владел объектом.

private static readonly Mutex Mutex1 = new Mutex(false, "MutexSample:AAED7056-380D-412E-9608-763495211EA8"); - 1й аргумент, требуется ли занять наш Mutex в момент создания (передаем false говоря о том что хотим создать экземпляр не занимая его, т.е. не заходя в критическую секцию) 2й код, его имя для управлния (Mutex синхронизация ядра, в связи с этим мы можем управлять им из нескольких процессов, а в классе Monitor только из нескольких потоков)

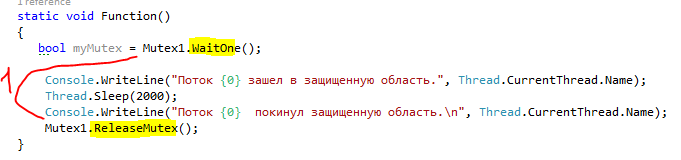
В каждом потоке запускаем метод Function



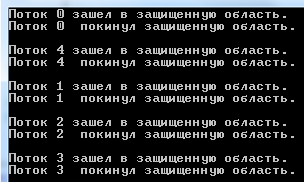
Mutex1.WaitOne(); - говорит о том что может работать только один поток, зашел в критическую секцию (занял разделяемый ресурс)

Mutex1.ReleaseMutex(); - вышел из критической секции, освободил ресурс

1 – критическая секция

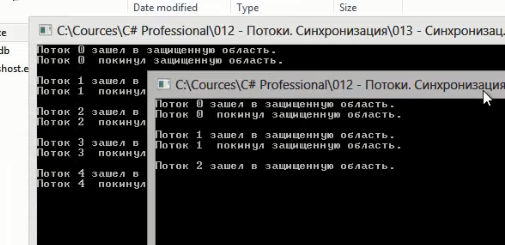


В один момент времени в критической секции находился только один поток



!!!!!!!!!!!!!!!!!!!!!!!!!

Из нескольких процессов заходят по очереди

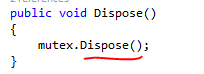


ОБЯЗАТЕЛЬНО

сколько вызываем методов Mutex1.WaitOne();

Столько мы должны вызвать методов Mutex1.ReleaseMutex();

В конце нужно освобождать



Semaphore – использует объект ядра для синхронизации

pool = new Semaphore(2, 4, "MySemafore65487563487"); - Первый аргумент:

Задаем количество слотов для использования в данный момент (не более максимального количества).

Второй аргумент:

Задаем максимальное количество слотов для данного семафора.

pool.WaitOne(); - занял слот семафора

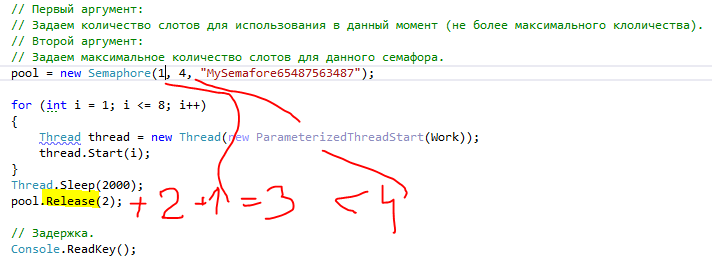
Console.WriteLine("Поток {0} занял слот семафора.", number);

Thread.Sleep(1000);

Console.WriteLine("Поток {0} -----> освободил слот.", number);

pool.Release(); - освободил слот

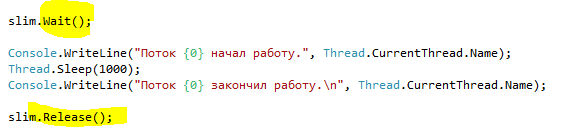
pool.Release(2); - сколько освободить слотов но не более максимума (второго параметра), так же учитываются не вызванные pool.Release() лучше не использовать т.к. другие процессы так же могут освободить и будет перебор максимального значения слотов (ошибка)



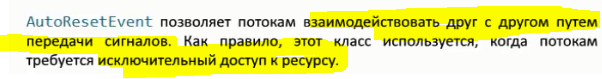
SemaphoreSlim –

когда не требуются междоменные или межпроцессорные организации

точно такой же по использованию как и предыдущий



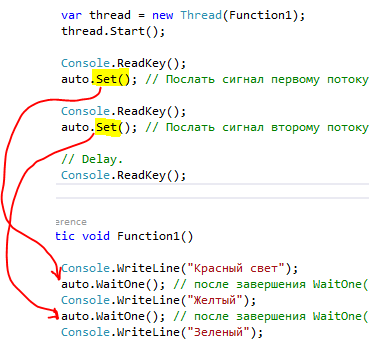
AutoResetEvent



static readonly AutoResetEvent auto = new AutoResetEvent(false); - Уведомляет ожидающий поток о том, что произошло событие., 2-й параметр говорит что он находится в несигнальном состоянии

auto.WaitOne(); - после завершения WaitOne() AutoResetEvent автоматически переходит в несигнальное состояние.

auto.Set(); - Послать сигнал потоку

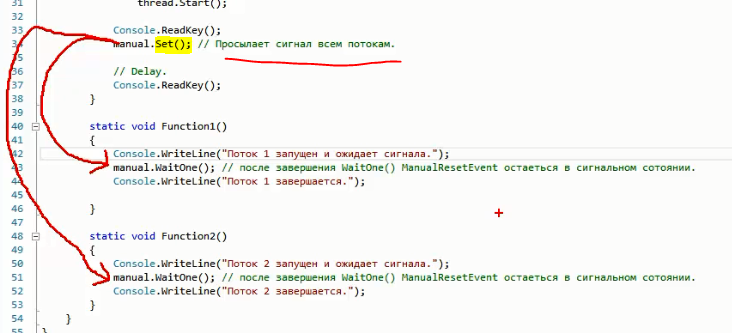


ManualResetEvent



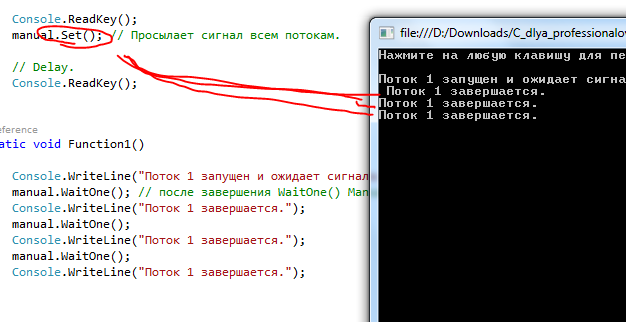
ГЛАВНАЯ ОТЛИЧИТЕЛЬНАЯ ЧЕРТА

ПОСЫЛАЕТСЯ СИГНАЛ ВСЕМ ПОТОКАМ (а в AutoResetEvent только одному)



И ГЛАВНОЕ

После вызова Set() остается в сигнальном состоянии (как будто того что ему постоянно шлют сигнал и на методы WaitOne() реагировать не будет)



**Нужно делать** manual.Reset(); - Сбрасывает в несигнальное состояние.

ManualResetEventSlim

происходит синхронизация с помощью объекта ядра

устанавливает и сбрасывает сигнальное состояние объекта на много быстрей чем обычный

если не нужна междоменное или межпроцессорное взаимодействие

EventWaitHandle

междоменное или межпроцессорное взаимодействие с применением ManualResetEvent или AutoResetEvent

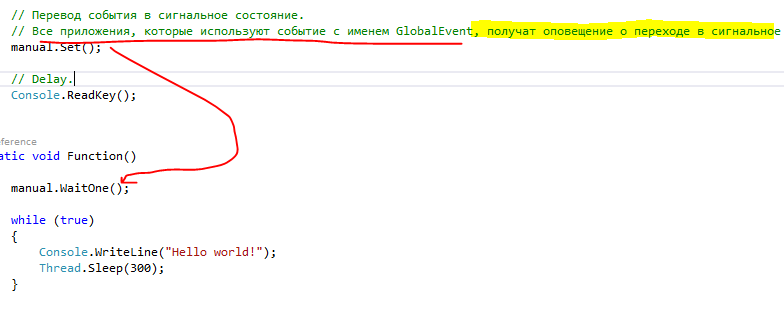
Если объект ядра с именем GlobalEvent уже существует будет получена ссылка на него.

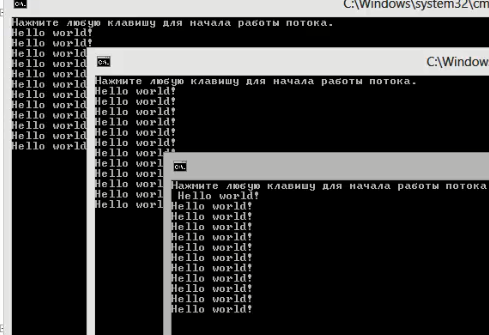
false - несигнальное состояние.

ManualReset - тип событиия.

GlobalEvent - имя по которому все приложения будут слушать событие.

manual = new EventWaitHandle(false, EventResetMode.ManualReset, "GlobalEvent::GUID");





WaitHandle

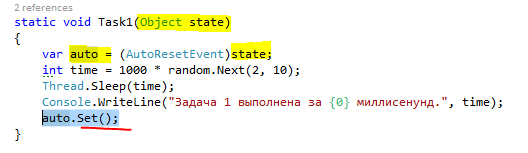
static WaitHandle[] events = new WaitHandle[] { new AutoResetEvent(false), new AutoResetEvent(false) };

Очередь для двух задач в двух разных потоках.

ThreadPool.QueueUserWorkItem(new WaitCallback(Task1), events[0]); - передается экземпляры WaitHandle

ThreadPool.QueueUserWorkItem(Task2, events[1]);

WaitHandle.WaitAll(events);- Ожидание пока все задачи завершаться. (в Task1 и Task2 должны вызваться auto.Set();)



int index = WaitHandle.WaitAny(events); - Ожидание пока одна из задач завершится.

