Multithreading

Thread

- Thread
  - Объект ядра потока (thread kernel object). контекст потока, набор регистров процессора ~ 1KB
  - Блок окружения потока (Thread Environment Block, TEB). 4KB, содержит заголовок цепочки обработки исключений, локальное хранилище данных для потока и некоторые структуры данных, используемые интерфейсом графических устройств (GDI) и графикой OpenGL.
  - Стек пользовательского режима (user-mode stack). По умолчанию на каждый стек Windows выделяет 1 Мбайт памяти.
  - Стек режима ядра (kernel-mode stack). x86 - 12 KB, X64 — 24 Кбайт.

- System.Threading.Thread - соответствует потоку в ОС
- Самый низкоуровневый объект для работы с потоками
- Запрещен в приложениях для windows store

```
public static void Main()
{
    Console.WriteLine("Main thread");
    Thread dedicatedThread = new Thread(ComputeBoundOp);
    dedicatedThread.Start(5);
    Console.WriteLine("Main thread: Doing other work");
    Thread.Sleep(2000); // Имитация другой работы (10 секунд)
    dedicatedThread.Join(); // Ожидание завершения потока
    Console.WriteLine("Main thread: ending");
}

// Передаем делегат ParameterizedThreadStart в конструктор Thread
private static void ComputeBoundOp(Object state)
{
    Console.WriteLine("In ComputeBoundOp: state={0}", state);
    Thread.Sleep(1000); // Имитация другой работы (1 секунда)
}
```

Методы:

- Abort уведомить CLR, что надо прекратить поток (для проверки завершенности следует опрашивать свойство ThreadState)
- Interrupt прервать поток на время
- Join остановить вызвающий поток до завершения потока, экземляру которого был вызван данный метод
- Resume возобновить работу потока
- Start запустить поток (при этом поток непосредственно создается в ОС)
- Suspend приостановить
- static Sleep останавливить поток
- static GetDomain ссылка на домен приложения
- static GetDomainId id текущего домена приложения

Приоритет потоков:

```
dedicatedThread.Priority = ThreadPriority.AboveNormal;
```

- Columns - Process priority
- 17+ - драйвера устройств

CLR Priority	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
Idle	1	1	1	1	1	16

В CLR все потоки делятся на foreground / background

- При завершении активного потока:
  - Принудительно завершит все фоновые потоки
  - Все фоновые потоки завершатся немедленно и без появления исключений
- Thread - по-умолчанию foreground, ThreadPool - background
- IsBackground - можно изменять в процессе работы
- Общая рекомендация - лучше использовать фоновые потоки

```
public static void Main()
{
    Thread t = new Thread(Worker);
    t.IsBackground = true;
    t.Start();

    // Активный поток - приложение будет работать около 10 секунд
    // IsBackground = true - немедленно прекратит работу
    // В LINQPad5 работает криво, в студии работает нормально :)
    Console.WriteLine("Returning from Main");
}
private static void Worker()
{
    Thread.Sleep(10000);
    Console.WriteLine("Returning from Worker");
}
```

Thread Pool

- CLR умеет управлять собственным пулом потоков, чтобы не плодить лишние потоки
- На каждый объект CLR создается свой пул потоков, которые используют все AppDomain
- Пулл потоков динамически определяет количество реальных потоков, которые необходимы приложению

```
ThreadPool

static Boolean QueueUserWorkItem(WaitCallback callBack);
static Boolean QueueUserWorkItem(WaitCallback callBack, Object state);

delegate void WaitCallback(Object state);
```

```
public static void Main()
{
    Console.WriteLine("Main thread: starting");
    ThreadPool.QueueUserWorkItem(Compute, 5);

    Console.WriteLine("Main thread: 10 sec waiting");
    Thread.Sleep(10000);

    Console.WriteLine("Main thread: exit");
}

private static void Compute(Object state)
{
    Console.WriteLine($"Compute: state = {state}");
    Thread.Sleep(1000);
}
```

Не лишним будет упомянуть, что есть контекст выполнения потока:

- Параметры безопасности, Principal
- Контекстные данные логического вызова
- Копирование контекста занимает много ресурсов
- По-умолчанию для новых потоков копируется контекст безопасности
- Можно запретить копирование контекста

```
public sealed class ExecutionContext : IDisposable, ISerializable
{
    [SecurityCritical] public static AsyncFlowControl SuppressFlow();
    public static void RestoreFlow();
    public static Boolean IsFlowSuppressed();
    // Не показаны редко применяемые методы
}
```

Cancellation Token

- стандартный паттерн отмены операций
- скоординированная отмена (оба класса должны поддерживать явно отмену)

```
public sealed class CancellationTokenSource : IDisposable
{
    public CancellationTokenSource();
    public Boolean IsCancellationRequested { get; }
    public CancellationToken Token { get; }
    public void Cancel(); // Вызывает Cancel с аргументом false
    public void Cancel(Boolean throwOnFirstException);
    ...
}
```

- None - не хотим отменять, не связан ни с каким CancellationTokenSource

```
public struct CancellationToken // Значимый тип
{
    public static CancellationToken None { get; }
    Boolean IsCancellationRequested { get; } // Вызывается операциями, не
связанными с Task
    public void ThrowIfCancellationRequested(); // Вызван операциями, связанными с
Task
    public WaitHandle WaitHandle { get; } // WaitHandle устанавливается при отмене
CancellationTokenSource

    public Boolean CanBeCanceled { get; } // Редко используется
    public CancellationTokenRegistration Register(Action<Object> callback, Object
state, Boolean useSynchronizationContext);
    // Более простые варианты перегрузки не показаны, Члены GetHashCode, Equals, ==
и != не показаны
}
```

```
public static void Main()
{
    using (CancellationTokenSource cts = new CancellationTokenSource())
    {
        ThreadPool.QueueUserWorkItem(o => Count(cts.Token, 1000));
        Thread.Sleep(1000);
        cts.Cancel(); // Если метод Count уже вернул управления, Cancel не
оказывает никакого эффекта

        Thread.Sleep(1000);
        Console.WriteLine("Quit the programm");
    }
}

private static void Count(CancellationToken token, Int32 countTo)
{
    for (Int32 count = 0; count < countTo; count++)
    {
        if (token.IsCancellationRequested)
        {
            Console.WriteLine("Count is cancelled");
            break;
        }
```

```
            Console.WriteLine(count);
            Thread.Sleep(200);
        }
        Console.WriteLine("Count is done");
    }
```

```
public static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    CancellationToken token = cts.Token;

    var obj1 = new CancelableObject("1");
    var obj2 = new CancelableObject("2");

    token.Register(() => obj1.Cancel());
    token.Register(() => obj2.Cancel());

    cts.Cancel();
    cts.Dispose();
}

class CancelableObject
{
    public string id;

    public CancelableObject(string id)
    {
        this.id = id;
    }
```

```
    public void Cancel()
    {
        Console.WriteLine("Object {0} Cancel callback", id);
    }
}
```

```
var cts1 = new CancellationTokenSource();
cts1.Token.Register(() => Console.WriteLine("cts1 canceled"));
var cts2 = new CancellationTokenSource();
cts2.Token.Register(() => Console.WriteLine("cts2 canceled"));

var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(cts1.Token,
cts2.Token);
linkedCts.Token.Register(() => Console.WriteLine("linkedCts canceled"));

cts2.Cancel(); // Отмена любого из дочерних приводит к отмене общего

Console.WriteLine("cts1 canceled={0}, cts2 canceled={1}, linkedCts={2}",
    cts1.IsCancellationRequested, cts2.IsCancellationRequested,
linkedCts.IsCancellationRequested);
```

```
public sealed class CancellationTokenSource : IDisposable
{
    public CancellationTokenSource(Int32 millisecondsDelay);
    public CancellationTokenSource(TimeSpan delay);
    public void CancelAfter(Int32 millisecondsDelay);
    public void CancelAfter(TimeSpan delay);
}
```

TPL

System.Threading.Tasks

Класс Task

- с ThreadPool есть глобальные проблемы
  - возврат результатов из потока.
  - как узнать о завершении операции
- Упрощенно: Task - типизированная обертка над пуллом потоков с кучей удобных методов

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, 5);

new Task(ComputeBoundOp, 5).Start();

Task.Run(() => ComputeBoundOp(5));
```

```
Task taskA = Task.Run( () => Console.WriteLine("Hello from thread '{0}'.",
Thread.CurrentThread.ManagedThreadId ));

Console.WriteLine("Hello from thread '{0}'.", Thread.CurrentThread.ManagedThreadId
);
taskA.Wait();
```

Task<TResult>

```
// Создание задания Task (оно пока не выполняется)
Task<Int32> t = new Task<Int32>(n => Sum((Int32)n), 1000000000);

t.Start();  // Можно начать выполнение задания через некоторое время

t.Wait();   // Можно ожидать завершения задания в явном виде
// ПРИМЕЧАНИЕ. Существует перегруженная версия, принимающая тайм-
аут/CancellationToken

Console.WriteLine("The Sum is: " + t.Result); // Получение результата (свойство
Result вызывает метод Wait)
```

- .Wait - если метод еще не начал выполняться - может выполнять его прямо в текущем потоке, что потенциально может приводить к дедлокам
- Исключения, сделанные в методах задачи сохраняются в отедльную коллекцию и при вызове .Wait / .Result возвращаются исходному коду в виде AggregateException, которая будет содержать коллекцию со всеми исключениями
- Если не вызвать wait / result, то об ошибке и не узнать

```
Task[] tasks = new Task[3]
{
    new Task(() => Console.WriteLine("First")),
    new Task(() => Console.WriteLine("Second")),
    new Task(() => Console.WriteLine("Third"))
};
foreach (var t in tasks)
{
    t.Start();
}
Task.WaitAll(tasks);

Console.WriteLine("End");
```

ContinueWith

- Возвращает Task, который частенько не используется

```
public static void Main()
{
    Task task1 = new Task(()=>{ Console.WriteLine($"current: {Task.CurrentId}");
});

    // задача продолжения
    Task task2 = task1.ContinueWith(Display);

    Task task3 = task1.ContinueWith((Task t) => { Console.WriteLine($"current:
{Task.CurrentId}"); });
    Task task4 = task2.ContinueWith((Task t) => { Console.WriteLine($"current:
{Task.CurrentId}"); });

    task1.Start();
    task1.Wait();
    Console.WriteLine("After task1 wait");
    Thread.Sleep(5000);
    Console.WriteLine("End");
}
```

```
static void Display(Task t)
{
    Console.WriteLine($"current: {Task.CurrentId}, previous: {t.Id} ");
    Thread.Sleep(3000);
}
```

```
public static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    Task<int> t = new Task<int>(() => Sum(cts.Token, 2), cts.Token);

    t.Start();

    cts.Cancel(); // кстати задача уже может быть завершена
    try
    {
        // В случае отмены задания метод Result генерирует исключение
AggregateException
        Console.WriteLine("The sum is: " + t.Result);
    }
    catch (AggregateException x)
    {
        // Считаем обработанными все объекты OperationCanceledException,
        // все остальные исключения попадают в новый объект AggregateException,
состоящий только из необработанных исключений, который заново будет выкинут
        x.Handle(e => e is OperationCanceledException);

        Console.WriteLine("Sum was canceled");  // Строка выполняется, если все
исключения уже обработаны
```

```
    }
}

private static Int32 Sum(CancellationToken ct, int n)
{
    int sum = 0;
    for (; n > 0; n--)
    {
        ct.ThrowIfCancellationRequested(); // исключение OperationCanceledException
        checked { sum += n; }
        // исключение System.OverflowException
    }
    return sum;
}
```

async / await

async Main

SyncronizationContext