Разработка .NET приложений

Лекция 12

Многопоточное программирование
Асинхронное программирование

Процессы и потоки

- На заре развития ОС
  - 1 поток исполнения: кода ОС и кода программы
  - PC выполнял только одно действие в любой момент времени.
  - Ошибка в программе, например, бесконечный цикл, приводил к необходимости перезагрузки машины.

- Процесс – выполняющаяся программа (экземпляр). Содержит набор ресурсов.
  - Виртуальное адресное пространство
  - Информация о загруженных модулях
  - Один или несколько потоков. ……..
  - Изолированность (код, данные ядра ОС и других программ недоступны)

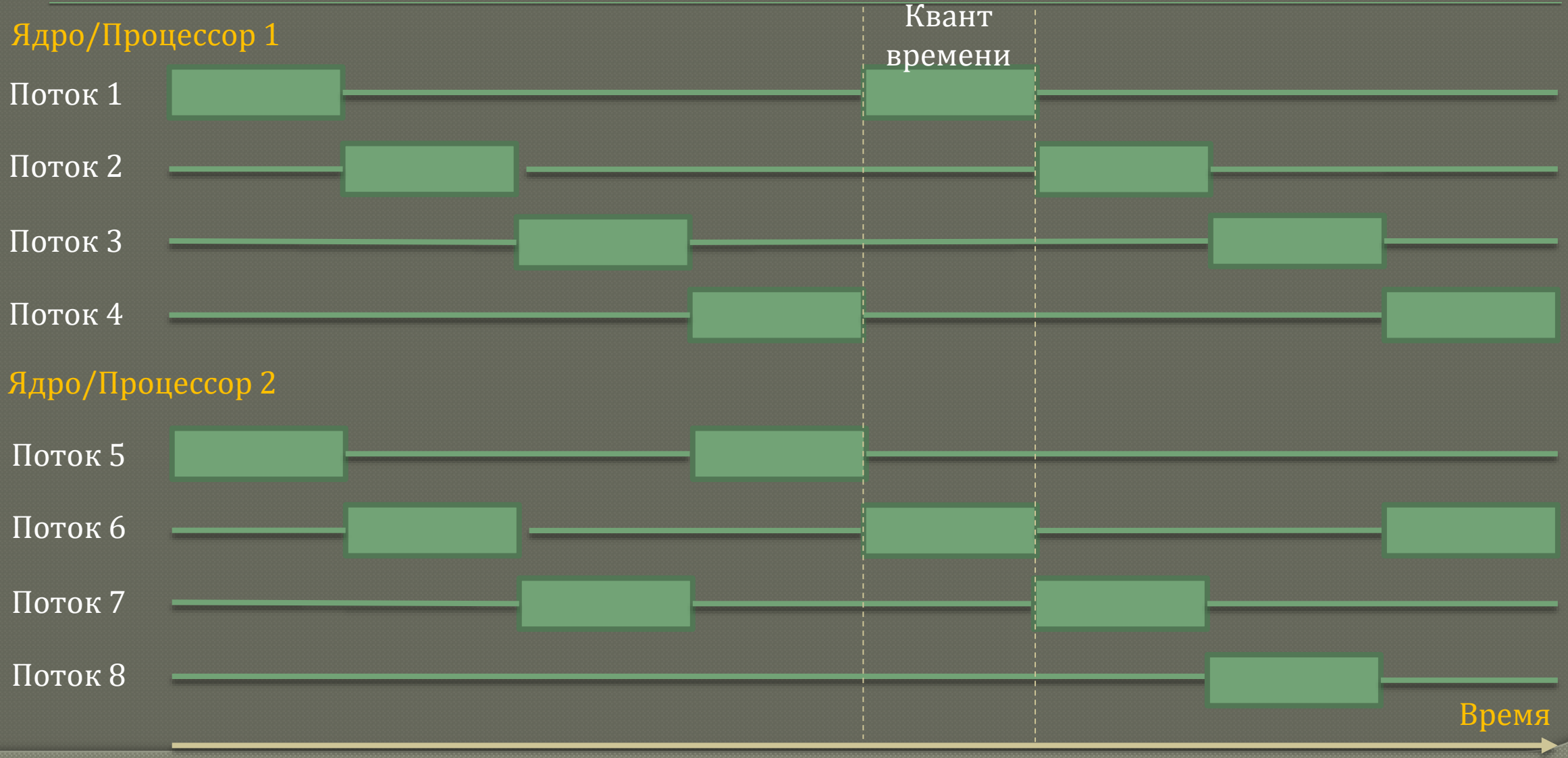
- Поток (нить, thread) – путь выполнения кода внутри исполняемого приложения (инструкция за инструкцией)
- При запуске приложения создается и запускается главный поток и ему в свое время передается управление
- Любой поток может запускать дополнительные потоки

- Завершение процесса – завершение всех его потоков

Процессы и потоки

- Современные ОС используют модель разделения времени для выполнения на одном процессоре нескольких потоков.
  - Поток получает квант времени в течении которого он выполняется на процессоре
  - По окончании кванта времени ОС
    - выбирает другой поток для выполнения, из потоков готовых к выполнения (с учетом их приоритетов)
    - сохраняет текущее состояние потока (контекст, регистры процессора и т.д.)
    - загружает состояние другого потока (контекст, регистры процессора и т.д.)
    - позволяет другому потоку выполнятся на процессоре. А старый поток ожидает своего нового кванта времени от ОС
    - Поток может досрочно освободить свой квант времени по различным причинам. Тогда ОС отдаст часть кванта другому потоку
    - Если ОС решает отдать новый квант времени уже выполняющемуся потоку, то переключение контекстов не происходит
- ОС планирует на процессоре потоки, не процессы.
- Потоки выполняются параллельно и независимо
- Нельзя точно определить, когда ОС выделит квант времени конкретному потоку (очень много факторов: несколько ядер/процессоров, неодинаковость ядер, гипертрединг, приоритеты потоков, аффинность потоков, синхронизации/блокировки потоков, временные повышения приоритетов потоков, изменение частот ядер и т.д.)
- Многопоточные приложения могут выполнятся и на однопроцессорном компьютере

Выполнение потоков



Создание потока в Windows

- Каждый обычный поток состоит
  - Объекта ядра потока (**thread kernal object**)
    - Содержит также контекст потока (блок с набором регистров процессора)
    - 700, 1240 байт (в зависимости от архитектуры ОС)
  - Блок окружения потока (**TEB, Thread Environment block**)
    - Содержит заголовки цепочки исключений
    - 4 кб
  - Стек пользовательского режима (**user-mode stack**)
    - Стек для хранения локальных переменных, адресов возврата из методов, структур
    - При создании 1 Мб
  - Стек режима ядра (**kernal-mode stack**)
    - Используется при вызове **Win API** функции режима ядра.
    - 12/24 кб (в зависимости от архитектуры ОС)
- При создании потока уведомляются все загруженные в процесс **dll** (вызывается функция **DllMain** из каждой **dll**)

- Создание и содержание потока достаточно затратно
- Почти все приложения имеют десятки и сотни потоков. В основном “спящих”

Переключение потока в Windows

- По окончании кванта времени ОС
  - выбирает другой поток для выполнения, из потоков готовых к выполнения (с учетом их приоритетов)
  - Сохраняет текущее состояние потока (контекст, регистры процессора и т.д.)
  - загружает состояние другого потока (контекст, регистры процессора и т.д.)
  - Если поток принадлежит другому процессу, то ОС переключает виртуальное адресное пространство для процессора
  - позволяет другому потоку выполнятся на процессоре. А старый поток ожидает своего нового кванта времени от ОС
- Поток может досрочно освободить свой квант времени по различным причинам. Тогда ОС отдаст часть кванта другому потоку
- Если ОС решает отдать новый квант времени уже выполняющемуся потоку, то переключение контекстов не происходит

- Никакого выигрыша в производительности или потреблении памяти переключение контекстов не дает.
- Новый поток работает с другими данными. Их тоже нужно загрузить в кэш процессора.
- Потеря конвейера выполнения на процессоре
- Чем больше потоков в процессе, тем дольше работать GC в CLR

- В тоже время простаивать дорогостоящим ресурсам может быть и не стоит

- В идеале должно выполняться столько потоков, сколько ядер/процессоров на машине

Планирование потоков. Приоритет потока

- Каждому потоку назначается приоритет от 0 до 31
- Операционная система планирует на выполнение потоки, не процессы. Оперирует приоритетами потоков
- ОС выбирает всегда поток для выполнения с наивысшим приоритетом. Потоки с более низкими приоритетами ожидают
- Если появился готовый к выполнению поток с более высоким приоритетом, чем выполняющийся, такт у выполняющегося потока прерывается и отдается более приоритетному потоку
- Возможно возникновение **starvation** (зависания потока)
- ОС может временно поднимать приоритеты определенных потоков по разными причинам (например, поток проигрывающий музыку, главный поток активного пользовательского приложения, поток обрабатывающей события мыши и т.д.)
- Некоторые **real-time** приоритеты зарезервированы за драйверами устройств

Относительный приоритет потока	**Класс приоритета процесса**					
	Idle	**Below Normal**	**Normal**	**Above Normal**	**High**	**Real-time**
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

Достоинства и недостатки многопоточной разработки

- При грамотном подходе может значительно ускорить работу приложения (только при многоядерной или много процессорной архитектуре)
- Позволяет повысить отзывчивость пользовательского интерфейса (даже при однопроцессорной архитектуре)
- Позволяет ускорить работу приложения за счет одновременного выполнения:
  - долгих удаленных операций (выполняющихся на других компьютерах)
    - Например, запрос к базе данных, к сервису или к интернет ресурсу
  - медленных, но мало затратных операций
    - Например, сохранение или чтение с диска

- Трудности разработки (дороговизна разработки)
  - Разбиение и оптимизация программы для многопоточной работы
  - Синхронизация потоков
  - Тестирование
- Трудности тестирования и отладки
  - Трудно обнаружимые ошибки
  - Невоспроизводимые ошибки
  - Непредсказуемые ошибки
- При неграмотном подходе может замедлить приложение
  - На создание и поддержание работы потоков тратятся ресурсы

Потоки в .NET

- Пространства имен
  - **System.Threading**
  - **System.Threading.Tasks**
  - **System.ComponentModel** (поток для **UI**, **BackgroundWorker**)
  - **System.Collections.Concurrent** (потокобезопасные коллекции)
- Класс **System.Threading.Thread**
  - Позволяет создать поток, запускать его на выполнение, и полностью его контролировать
  - Содержит методы для работы с потоками
  - Содержит, статические члены для работы с текущим выполняющимся потоком
  - **static Thread Thread.CurrentThread** – содержит текущий поток
- Единица кода для запуска в потоке – метод
  - В отдельном потоке всегда запускается какой-то метод

Запуск кода в отдельном потоке

- Необходимо написать метод, который будет выполнятся новым потоком
  - `public void ThreadMethod() {…}`
- Создать экземпляра делегата с ссылкой на метод
  - **ThreadStart** – для запуска метода без параметров
  - **ParameterizedThreadStart** – для запуска метода с одним параметром (но параметр **object**)
- Создать поток и передать ему делегат на метод
  - `Thread thread= new Thread(new ThreadStart(ThreadMethod));`
- Запустить поток
  - `thread.Start();`
- Реально создает поток в ОС

Передача параметров потоку

- Использование делегата **ParameterizedThreadStart** вместо **ThreadStart**
- Передача только 1 параметра, но параметра типа **object**

```
public static void ThreadMethod(object o){..}
Thread thread = new Thread(new ParameterizedThreadStart(ThreadMethod));
thread.Start(obj);
```

- Использования замыкания и лямбда выражения

```
int i = 5;
Thread thread = new Thread( () => ThreadMethodWithInt( i ) ));
```

- Методы для выполнения в потоке ничего не возвращают

Демонстрации

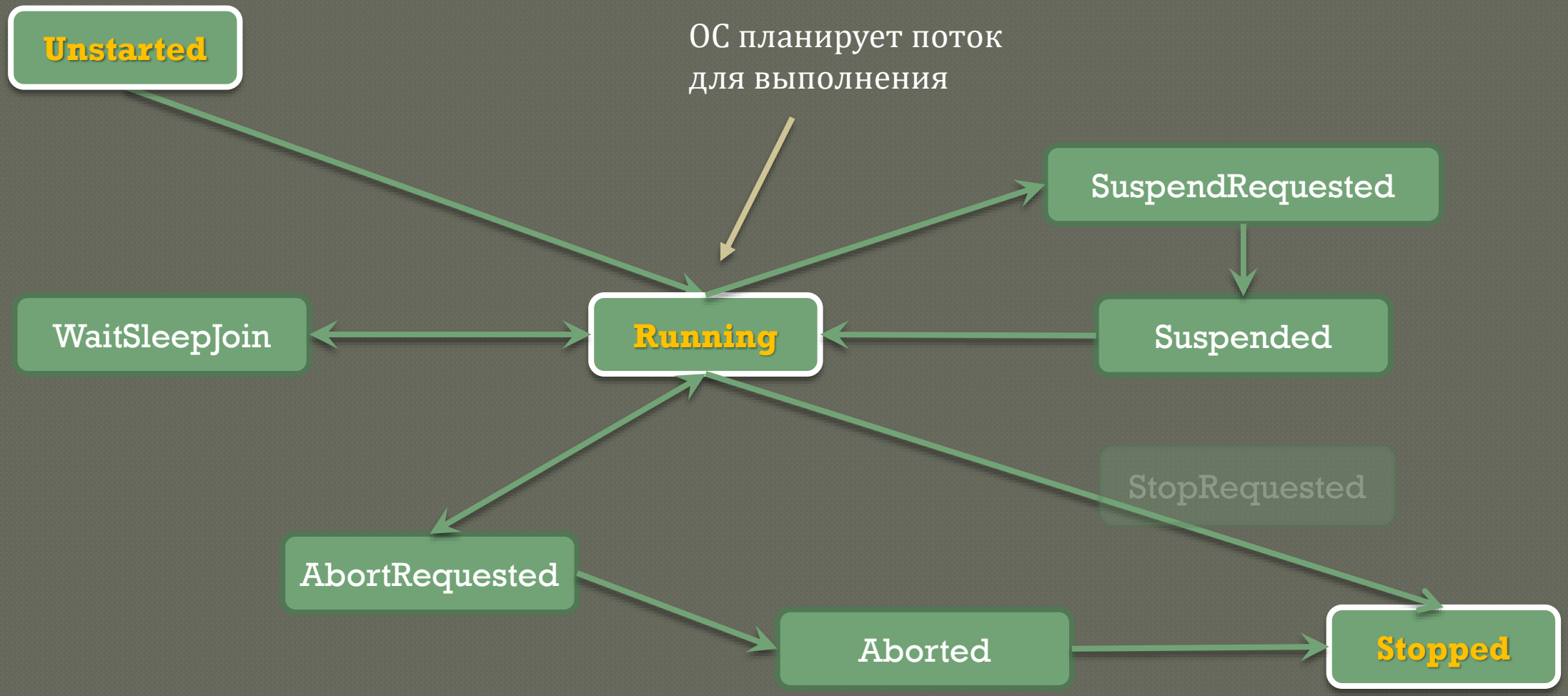
Thread

Передача параметров

Класс Thread

- Свойства потока
  - Name – имя потока (удобно использовать для отладки)
  - ManagedThreadId – уникальный ID потока
  - Priority – приоритет потока
  - IsAlive – поток запущен и не приостановлен
  - ThreadState – состояние потока
  - IsBackground – фоновый ли поток
  - IsThreadPoolThread – принадлежит ли поток пулу потоков CLR
- Полезные методы и свойства для работы с потоками
  - Thread.CurrentThread – ссылка на текущий поток (статическое вычислимое свойство)
  - Thread.Sleep() – заставляет поток ожидать указанное время (статический метод)
  - thread.Join() – заставляет ожидать текущий поток завершения указанного потока.
  - thread.Abort() – заставляет аварийно завершить поток

Состояния потоков



Завершение потока

- Поток завершится при выходе из метода
- thread.Abort() – аварийное завершение потока
  - При этом у прерываемого потока возникает исключение ThreadAbortedException
  - Прерываемый поток может обработать исключение ThreadAbortedException, но после этого исключение будут вызвано снова
- thread.AbortReset() – отмена прерывания потока (если успеть, пока поток еще аварийно не завершился)
- Завершенный поток нельзя запустить снова
- Приостановка потока
  - thread.Join() – блокировка текущего потока до завершения другого потока
  - Thread.Sleep() – заставляет текущий поток ожидать указанное время (статический метод). Квант времени отдается. Через указанный промежуток времени планировщик потоков начнет планировать текущий поток к выполнению

Фоновые потоки

- Потоки:
  - Потоки переднего плана (по умолчанию)
  - Фоновые потоки
- Процесс не завершится пока есть работающие потоки переднего плана
- Фоновые потоки при завершении основного потока получают исключение **ThreadAbortedException** и будут завершены
- Необходима реализация безопасного завершения фонового потока
- Установка потока как фонового

```
thread.IsBackground = true;
```

Демонстрации

Завершение фонового потока

Порядок выполнения потоков непредсказуем

- Потоки выполняются параллельно и независимо. Нельзя предсказать очередность выполнения блоков кода потоками.

```
static void Main()
{
    Thread t = new Thread(Write1);
   t.Start();
    while (true) Console.Write("-"); // Все время печатать '-'
}

static void Write1()
{
   while (true) Console.Write("1"); // Все время печатать '1'
}
```

Независимый стек локальных переменных

- У каждого потока свой стек локальных переменных. Они независимые.

```
static void Main()
{
   new Thread(Go).Start();      // Выполнить Go() в новом потоке
   Go();                        // Выполнить Go() в главном потоке
}

static void Go()
{
   // Определяем и используем локальную переменную 'cycles'
    for (int cycles = 0; cycles < 5; cycles++) Console.Write('+');
}
```

Общий Heap

- Вместе с тем потоки разделяют данные, относящиеся к тому же экземпляру объекта

```
class TestClass
{
      bool done = false;
      public void Go()
      {
         if (!done) {  done = true;   Console.WriteLine("Done"); }
      }
}

class ThreadTest
{
      static void Main()
     {
         TestClass testClass = new TestClass();
         new Thread(testClass.Go).Start();
         testClass.Go();
      }
}
```

Операции не являются атомарными

```
class Increment {
      decimal l = 0;
      public void inc()  {
         for (int i = 0; i < 100000; ++i) l = l +1;
                        Console.WriteLine(l);
     }   }

class Program {
      static void Main(string[] args)  {
           Increment i = new Increment ();
           for (int j = 0; j < 10; ++j)
            new Thread(i.inc).Start();
     } }
```

Присваивание ссылочных типов атомарно (при любой разрядности ОС)

Итого

- Потоки выполняются параллельно и независимо. Нельзя предсказать какой поток отработает быстрее.
- У каждого потока свой собственный стек. Собственные неразделяемые локальные переменные
- У потоков общий **Heap.** Потоки разделяют нелокальные переменные, доступные им по области видимости
- Операции неатомарные

- Операция присваивание ссылочных типов атомарно (при любой разрядности ОС)
- Операции присвоения типов **bool, char, byte, sbyte, short, ushort, int, uint, IntPtr, UIntPtr, float** происходит атомарно.

Синхронизация потоков

- Примитивы синхронизации пользовательского режима
  - Используются специальные директивы процессора
  - Поддержка на аппаратном уровне
  - Такие блокировки ОС не распознает и не отбирает процессорное время
  - Быстрые
  - Могут расходовать ресурсы впустую
- Примитивы синхронизации режима ядра ОС
  - Предоставляются ОС. В .NET обертки над объектами ОС
  - Требуют значительных затрат для общения с объектами ОС
  - ОС реально останавливает заблокированный поток, не дает ему выполняться, пока не снимется блокировка, не планирует поток для выполнения
- Гибридные примитивы синхронизации
  - Сочетают преимущества обоих подходов. Используются легкие примитивы синхронизации пользовательского режима при отсутствии блокировки и при блокировке на очень короткие промежутки времени. Если же блокировка не освобождается длительное время, то поток уходит в блокировку в режиме ядра.

Синхронизация потоков

- Примитивы синхронизации пользовательского режима
  - Класс **Volatile**
  - Класс **Interlocked**
  - **SpinLock**, **SpinWait**
- Гибридные примитивы синхронизации
  - Конструкция **lock**
  - Класс **Monitor**
  - Структура **SpinLock**
  - Классы **ReaderWriterLock**, **ReaderWriterLockSlim**
- Примитивы синхронизации режима ядра ОС
  - Класс **Mutex**
  - Семафоры
  - Наследники от **EventWaitHandle**

Примитивы синхронизации пользовательского режима

Оптимизация кода

```
private static bool _stopFflag;

static void Main(string[] args)
{
   Thread t = new Thread(Worker);
   t.Start();

   Thread.Sleep(2000);
   _stopFflag = true;
   Console.WriteLine("Stop");
   t.Join();
   Console.WriteLine(“Done");
}

private static void Worker()
{
   int x = 0;
   while (!_stopFflag) x++;
   Console.WriteLine(x);
}
```

- В **Debug** режиме и в режиме отладки проблем нет.
- В **Release** режиме и с включенной оптимизацией кода (по умолчанию включена для **Release**) оптимизатор заменяет код на **while(true)**

- Даже без оптимизации кода _stopFflag может попасть в кэш одного ядра процессора и не пошариться с другим ядром

*Идея Джеффри Рихтер. **CLR via C#**

Оптимизация кода

```
private static int _flag;
private static int _value;

static void Main(string[] args)
{
   Thread t = new Thread(Worker);
   Thread t2 = new Thread(Worker2);
   t.Start();
   t2.Start();

   t2.Join();
   Console.WriteLine("done");
}
```

```
private static void Worker()
{
   _value = 5;
   _flag = 1;
}


private static void Worker2()
{
   if (_flag == 1)
      Console.WriteLine(_value);
}
```

Поле **_value** может быть прочитано раньше **_flag**, поскольку они в методе не меняются

*Идея Джеффри Рихтер. **CLR via C#**

Volatile

- Класс **Volatile**
  - Volatile.Read() - считывает значение указанного поля. Добавляет барьер в памяти, предотвращая изменение порядка операций процессора с памятью: если после вызова этого метода следует операции чтения или записи, процессор не сможет выполнить их перед вызовом этого метода.
  - bool isRunning = Volatile.Read(ref _isBusy);
  - Volatile.Write() - записывает заданное значение в поле. Добавляет барьер в памяти, предотвращая изменение порядка операций процессора с памятью: если до вызова этого метода используются операции чтения или записи, процессор обязан будет выполнить их до вызова этого метода.
    - Volatile.Write(ref _isBusy, true);
- Модификатор поля **volatile** – все операции с полем будут выполнятся как **Volatile.Read()** и **Volatile.Write()**
  - public volatile bool _isBusy;
- Немедленная запись в память для шаринга значений между ядрами
- Не применим к **long** и **double**, поскольку чтение и запись их не атомарны

Класс Interlocked

- Атомарные операции. Статические члены
  - Interlocked.Increment(ref i); i – long или int
  - Interlocked.Decrement(ref i); i – long или int
  - Interlocked.Add(ref i1, i2); Переменные int, long

  - Interlocked.Exchange(ref i, value);
  - Interlocked.Exchange<T>(ref T i, T value);
    - Возвращает исходное значение

  - Interlocked.CompareExchange(ref i, value, compared);
    - Если i == compared, то i = value. Переменные типов: int, long, float, double, object.
    - Возвращает исходное значение
  - Interlocked.CompareExchange <T> (ref T i, T value, T compared) – для ссылочных типов

- Работает на уровне команд процессора
- Не блокирует потоки
- Ставят барьеры памяти, также как и volatile

SpinLock, SpinWait

- **SpinLock, SpinWait.** Примитивы пользовательского режима
- Не отдают процессорное время в процессе ожидания.

- **bool SpinWait.SpinUntil (Func<bool> condition, int millisecondsTimeout)**
  - выполняется до тех пор пока не будет выполнено условие и возвращает **true** или пока не выйдет таймаут, и возвращает **false**

- Структура **SpinLock** - примитив взаимно исключающей блокировки, в котором поток, пытающийся получить блокировку, ожидает в состоянии цикла, проверяя доступность блокировки.

```
// Должен быть общий для всех потоков
SpinLock sl = new SpinLock();


.....
bool gotLock = false;
  try
   {
      sl.Enter(ref gotLock);
      sb.Append((i % 10).ToString());
   }
   finally
   {
      // Only give up the lock if you actually acquired it
      if (gotLock) sl.Exit();
   }
```

Демонстрации

Interlocked

Гибридные примитивы синхронизации

Конструкция lock

- Гибридная синхронизация
- Необходимо определить **единую** доступную всем потокам **ссылочную** переменную (экземпляр объекта)
- Если объект в переменной не блокирован, то поток проходит беспрепятственно через оператор **lock**, блокируя объект
- Если объект в переменной блокирован, то поток остановится на операторе **lock** и будет ожидать пока другой поток не выйдет из конструкции **lock**. На короткое время заблокированный поток не отдает процессорное время, если ожидание превышает порог, то поток уходит в блокировку уровня ядра с освобождением процессора.
- При освобождении блокировки только 1 поток может взять блокировку и войти в **lock** блок
- Поток, взявший блокировку, может беспрепятственно входить в блокировку еще раз.

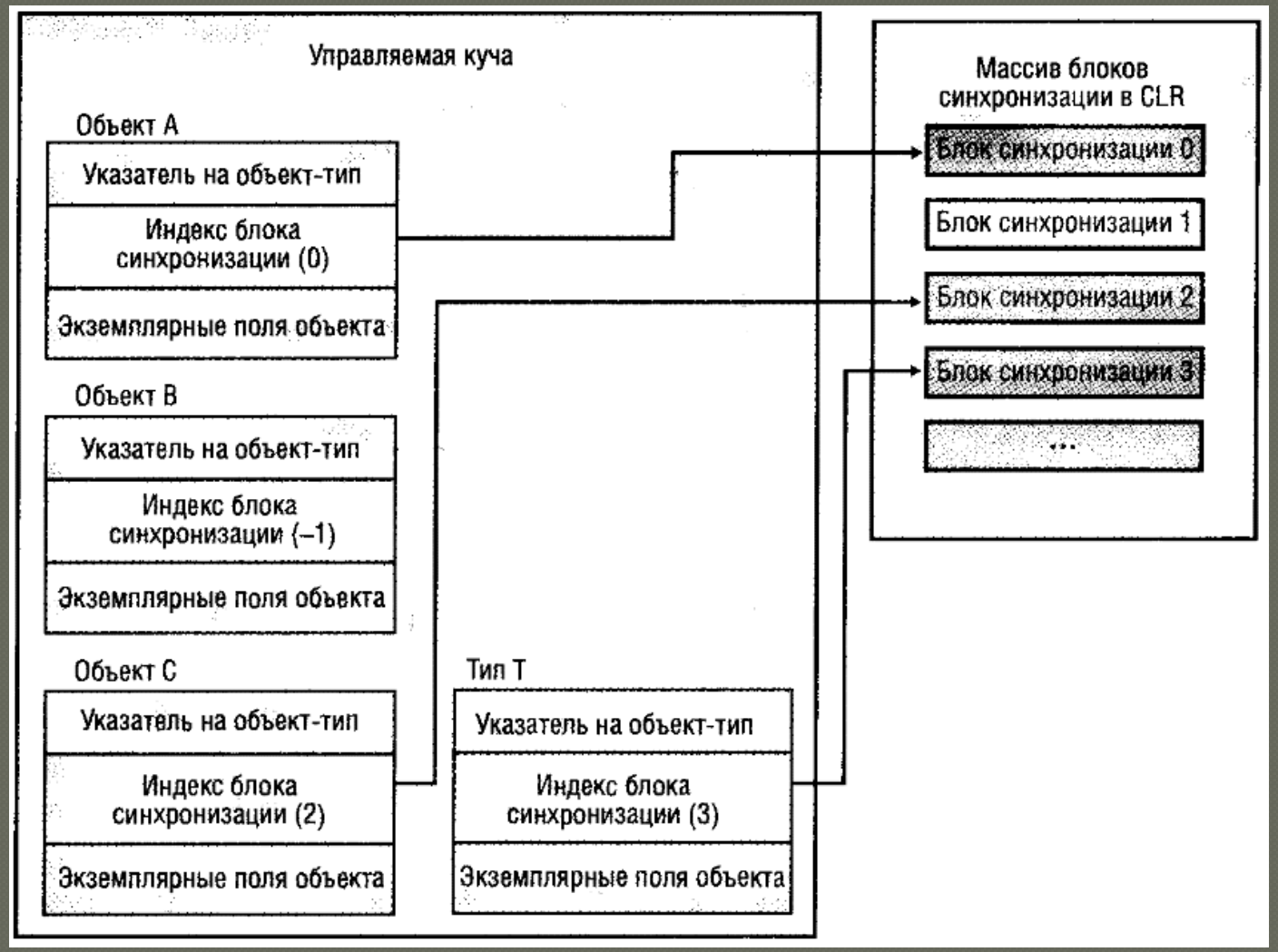
- Например:

```
public object lockObject = new object();

lock (lockObject)
{
        // Операции с разделяемыми ресурсами
}
```

- Каждый объект в куче имеет индекс блока синхронизации, который и используется для блокировок при синхронизации потоков
- Не используйте **string** из-за его неизменяемой структуры и интернирования

Конструкция lock



*См. Джеффри Рихтер. **CLR via C#**

Важно

- Как можно быстрее освобождать блокировку
- Избегать взаимоблокировок (deadlock)

```
lock (A)
{
    lock (B)
    {
        .....
    }
}
```

```
lock (B)
{
    lock (A)
    {
        .....
    }
}
```

- Блокировать только ссылочную переменную
- Экземпляр блокируемого объекта должен быть один и тот же для всех потоков

Демонстрации

lock

Класс Monitor

- Monitor.Enter(lockObject); - ожидание и вход потока в критическую секцию. Увеличение количества блокировок на 1.
- Monitor.Exit(lockObject); - выход из критической секции. Уменьшение количества блокировок на 1.
- Конструкция **lock** реализуется через класс **Monitor.**
- Необходимо самостоятельно следить за количеством установок / снятия блокировок.

- Конструкция **lock** эквивалентна конструкции

```
object _lockObj = new object();
bool _lockTaken = false;
try
{
    Monitor.Enter(_lockObj, ref _lockTaken);
    ......
}
finally
{
    if (_lockTaken) Monitor.Exit(_lockObj);
}
```

ReaderWriterLock

- Очереди читателей и писателей.
- Много потоков могут читать данные
- Только один поток может захватить объект для записи.

```
ReaderWriterLock rwl = new ReaderWriterLock();
rwl.AcquireReaderLock(timeout);
rwl.AcquireWriterLock(timeout);
rwl.UpgradeToWriterLock(timeout);
rwl.DowngradeFromWriterLock(ref cokie);
rwl.ReleaseReaderLock();
rwl.ReleaseWriterLock();
```

ReaderWriterLockSlim

- Аналогичен ReaderWriterLock
- Короткие блокировки реализуются как инструкции Spin
- Но имеет еще одно доп. Состояние:
  - Read mode
  - Write mode
  - Upgradeable mode

```
ReaderWriterLockSlim sl = new ReaderWriterLockSlim();
sl.EnterReadLock();        sl.TryEnterReadLock();
sl.ExitReadLock();
sl.EnterWriteLock();
sl.ExitWriteLock();
sl.EnterUpgradeableReadLock();
sl.ExitUpgradeableReadLock();
```

Примитивы синхронизации режима ядра

Mutex

- Тяжеловесный. Уровня ядра ОС. Оборачивает мьютекс ОС
- Реально блокирует поток. ОС перестает планировать заблокированный поток до снятия блокировки
- Может использоваться для синхронизации потоков в разных Процессах (именованные мьютексы).

Создание мьютекса - `Mutex mutex = new Mutex(false, "MyUniqueMutex");`
Взятие блокировки - `mutex.WaitOne();`
Обращение к ресурсам - .....
Снятие блокировки - `mutex.ReleaseMutex();`
Обязательно закрытие мьютекса, освобождение ресурсов ОС - `mutex.Close();`

- Есть перегруженные методы с ограниченным временем ожидания блокировки.
- Только поток вызвавший mutex.WaitOne() может освободить его - mutex.ReleaseMutex()
- Тяжелые последствия, если Mutex не закрыли при закрытии потока (следующий пользователь этого мьютекса получит исключение AbandonedMutexException)

Демонстрации

1. Приложение, допускающее только один запущенный экземпляр приложения
2. Синхронизация процессов

Семафоры

- Позволяют обеспечит доступ определенного числа потоков к разделяемым ресурсам
- Объект уровня ядра ОС. Тяжеловесный
- Реально блокирует поток. ОС перестает планировать заблокированный поток до снятия блокировки

- Может использоваться для синхронизации Процессов.
- Semaphore sem = new Semaphore(initBlocks, maxBlocks, “MySemaphore");
- sem.WaitOne(); - взятие блокироки
- sem.Release(); - освобождение блокировки (1 или несколько блокировок)
- sem.Close(); - освобожнение ресурсов ОС

- Release может делать и другой поток, нежели вызвавший WaitOne (в отличии от Mutex)
- Можно делать sem.Release(int) – освобождение нескольких блокировок
- Нет гарантии FIFO, или LIFO потоков. Не понятно какой поток из заблокированных сможет взять блокировку в случае ее отпускания

Демонстрации

Синхронизация процессов

Класс EventWaitHandle

- Наследники:
  - AutoResetEvent
  - ManualResetEvent
- Раздельно устанавливают блокировки и снимают.
- Один поток может ожидать, а другой по своей логике может его пропустить дальше
- Сообщает другому потоку, что событие произошло и тот может выполнять свои действия

- AutoResetEvent are = new AutoResetEvent(bool начальное состояние);
- are.Set(); - Снимает блокировку
- are.WaitOne(); - Ожидать снятия блокировки
- are.Reset(); - Устанавливает блокировку

- AutoResetEvent после прохода WaitOne автоматически устанавливает блокировку (Reset). ManualResetEvent – нет.

- ManualResetEvent.
  - Если один поток установил Set(), то все потоки проходят через WaitOne() без остановки.
  - Если какой-то поток установил вызвал Reset (), то все потоки при достижении WaitOne() встанут в кернеловую блокировку.

- Объекты ядра ОС.

Демонстрации

Синхронизация

Работа с коллекциями

- Некоторые коллекции содержат объект для синхронизации (для использования с **lock**) – **SyncRoot**

```
int[] col = new int[2];
.....
lock(col.SyncRoot)
{
        // работа с массивом
}
```

SyncRoot - практически не используется из-за наличия специальных потокобезопасных коллекций.

- Имеются специальные коллекции, доступ к которым из разных потоков не требует синхронизации, поскольку они содержат внутренние механизмы синхронизации
- **ConcurrentQueue<T>** - очередь
- **ConcurrentStack<T>** - стек
- **ConcurrentDictionary<TKey, TValue>** - словарь
- **ConcurrentBag<T>** - простой список

- **BlockingCollection<T>** - реализация **producer/consumer** паттерна

Демонстрации

Работа с коллекциями

Пул потоков

- В среде выполнения уже существует несколько запущенных потоков – пул потоков
- Количество потоков связано с количеством ядер и процессоров.
- При использовании потока из пула потоков нет накладных расходов на создание потока
- В пуле потоки фоновые
- Класс **ThreadPool** – позволяет получить доступ к пулу потоков **.NET**
- Постановка задания в очередь
  - Создание экземпляра делегата **void WaitCallback(object state)**
  - Постановка в очередь **ThreadPool.QueueUserWorkItem**
  - **(new WaitCallback(threadMethod), obj);**
- Переданное задание уже нельзя отменить
- Нет возможности узнать о завершении задания
- Нет возможности получить результат выполнения задания (в отличии от **Task<T>**)

- Обычно используют класс **Task** вместо класса **ThreadPool**

Демонстрации

Пул потоков

Асинхронный вызов делегатов

- Любой делегат имеет помимо метода для синхронного вызова – **Invoke**(), методы для асинхронного вызова **BeginInvoke**(), **EndInvoke**()

  Func <string, double, int> f = ….

  IAsyncResult f.BeginInvoke(string s, double d, AsyncCallback callback, object obj) – начинает вызов и передает параметры **string, double**

  int f.EndInvoke(IAsyncResult ires) – ожидает завершения и возвращает значение

- **AsyncCallback callback** – делегат будет вызван при окончании вычисления
- Выполнение в пуле потоков

- Интерфейс **IAsyncResult**
  - Свойство передавать параметры для последующей идентификации вызванного мето**bool IsComplated** – завершено ли вычисление
  - Свойство **object AsyncState** – позволяет да

- После появления **Task** практически перестал использоваться

Демонстрации

Асинхронный вызов делегата

Классы Task и Task<T>

- Простой запуск выполнения действия в **Thread Pool**
- **Thread Pool** содержит очередь тасок, и разбирает ее по мере выполнения тасок, по мере освобождения потоков
- При занятости всех потоков в **Thread Pool**, наличии тасок в очереди и наличии свободных процессорных ресурсов **Thread Pool** может создать новый поток пула и выполнять таски в нем.
- При простаивании потоков **Thread Pool** некоторые потоки могут быль удалены

- **Task** - класс для вызова метода, ничего не возвращающего, а **Task<TResult>** для возвращающего результат **TResult**
- Запуск таски – **Start()**. Конструктор – настройка таски
  - `void inc() { … }`
  - `Task t = new Task(inc);`
  - `t.Start();`

  - `Task<int> t = new Task<int>(GetInt);`
  - `t.Start();`

- Быстрый старт заданий **Task.Run**(Action method);
- Быстрый старт заданий **Task.Factory.StartNew**()
  - `Task  t = Task.Factory.StartNew(inc);`
  - `Task<int> t = Task<int>.Factory.StartNew(GetInt);`
  - `Task<int> t = Task<int>.Factory.StartNew(Add, new object[] { 5,7 } );` // но чаще используют лямбда выражение и замыкание

Получение результата

- Получение результата по окончании таски **Task<T>** - свойство **Result** (если результат не готов, то текущий поток приостановиться до получения готового результата)
  - **int res = t.Result;** // если **t – Task<int>**
- Ожидание завершения тасок
  - Ожидание завершения таски **Wait()**;
    - **t.Wait();**
  - Ожидание завершения всех тасок **Task.WaitAll()**
    - **Task.WaitAll(task1, task2, task3)**
  - Ожидание завершения хотя бы одной тасок **Task.WaitAny()**
    - **Task.WaitAny(task1, task2, task3)**

Exception при выполнении таски

- Если в процессе работы таски генерируется необработанное исключение, то оно сохраняется в коллекции.
- Поток пула возвращается в пул
- При вызове метода **Wait** у таски или свойства **Result** в текущем потоке будет сгенерировано исключение AggregateException
- **AggregateException** в коллекции **InnerExceptions** содержит набор исходных исключений.

- Обработка **AggregateException**
  - **AggregateException** содержит метод Handle
    - void Handle (Func<Exception,bool> predicate);
  - Если метод возвращает **true**, то данное внутреннее исключение считается обработанным
  - Если **Handle** вернет **false** хотя бы для одного исключения из **InnerExceptions**, исключение **AggregateException** сгенерируется заново, но с коллекцией только не обработанных исключений в **InnerExceptions**.

Скоординированная отмена заданий

- Разработчик при реализации асинхронной задачи явно добавить код проверки отмены задания
- Класс **CancellationTokenSource** – содержит состояния для скоординированной отмены
  - Метод **Cancel**() – нужно вызвать для отмены тасок
  - Свойство **IsCancellationRequested** – указывает запросили ли отмену
  - Свойство **Token** – токен для скоординированной отмены (структура **CancellationToken**). Передается в таски для определения запроса отмены

```
static void Main()
    {
        CancellationTokenSource cts = new CancellationTokenSource();
        CancellationToken token = cts.Token;
        Task.Factory.StartNew(() => LongMethod(100, token), token); // Токен передается в сам метод, для определения отмены
и параметром в метод StartNew(). Если таска не успела стартануть, и произошла отмена, то таска не будет стартовать совсем

        // выполнялась какая-то параллельная логика и решили остановить асинхронную задачу
       cts.Cancel();
     }
```

Проверка отмены задания

- Структура CancellationToken
  - Содержит приватную ссылку на свой CancellationTokenSource
  - Свойство IsCancellationRequested – указывает запросили ли отмену
  - Метод ThrowIfCancellationRequested() вызовет OperationCanceledException и таска будет завершена. Это нужно, чтобы таска вернула некоторый дополнительный результат к уже ожидаемому. Например Task<int> помимо int может вернуть исключение OperationCanceledException
  - Метод Register() – позволяет задать методы, которые будут выполняться в случае отмены задания.

- Разработчик при реализации метода таски должен проверять не запрошена ли отмена таски

```
private static void LongMethod(int count, CancellationToken cancellationToken)
{
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine(i);
        // проверка, не отменена ли задача
      if (cancellationToken.IsCancellationRequested) return; // Или ThrowIfCancellationRequested()
        Thread.Sleep(1000);
    }
}
```

- При использовании ThrowIfCancellationRequested, нужно обрабатывать AggregateException с исключением OperationCanceledException в InnerExceptions

Отмена задания

- CancellationTokenSource cts = new CancellationTokenSource();
- Если CancellationToken (cts.Token) передать в несколько тасок, то при вызове cts.Cancel() отменятся сразу все таски

```
static void Main()
{
    CancellationTokenSource cts = new CancellationTokenSource();
    CancellationToken token = cts.Token;
    Task.Factory.StartNew(() => LongMethod(100, token), token);
    Task.Factory.StartNew(() => OneMoreLongMethod(1, token), token);
    Task.Factory.StartNew(() => ThirdLongMethod("long method", token), token);
      …
      cts.Cancel(); // отменятся все 3 таски
}
```

- Отмена таски по истечению времени
  - Передача таймаута в конструктор CancellationTokenSource
    - CancellationTokenSource cts = new CancellationTokenSource(TimeSpan.FromSeconds(10));
  - Вызов CancelAfter у CancellationTokenSource
    - cts.CancelAfter(TimeSpan.FromSeconds(10));

Запуск нового задания по завершении предыдущего

- Продолжение выполнения **ContinueWith()**;
- Метод будет выполняться по завершении таски (сама таска пойдет на вход методу, так можно получить результат, полученный в предыдущей таске)
  - `Task<int> sumTask = Task.Run( () => Sum(100));`
  - `Task printTask = sumTask.ContinueWith(task => Console.WriteLine(rask.Result));`
- По завершении таски можно стартовать и не одну таску
  - `Task sendMainTask = sumTask.ContinueWith(task => SendMail(rask.Result));`
- Параметры запуска новой таски **TaskContinuationOptions**. Можно задать как параметр для **ContinueWith()**

TaskContinuationOptions

- Flag enum

None	По умолчанию
OnlyOnRanToCompletion / NotOnRanToCompletion	Указывает, что продолжение должно планироваться, только если предшествующая задача завершилась (не) успешно
OnlyOnCanceled / NotOnCanceled	(Не) планировать, предшествующая если задача была отменена
OnlyOnFaulted / NotOnFaulted	(Не) планировать, если задача сгенерировала исключение
OnlyOnCanceled	Планировать, только если задача была отменена
AttachedToParent / DenyChildAttach	Указывает, что задача (не) является дочерней задачей. Родительское задание завершается только после завершения всех его потомков.
ExecuteSynchronously / RunContinuationsAsynchronously	Запускать задачу синхронно или асинхронно
LongRunning	Указывает, что продолжение будет длительной подробной операцией. Просьба пулу потоков более активно создавать потоки
PreferFairness	Указание для планирования задач TaskScheduler в том порядке, в котором они были запланированы, т. е. задачи, запланированные ранее, будут выполняться ранее, а более поздние — позже.

Класс Parallel

- Parallel.For(initvalue, endvalue, Action<T>); - Выполнение цикла в максимально возможном числе потоков (ThreadPool). В цикле выполняется делегат Action<T> (который принимает 1 параметр T и, ничего не возвращает). Числом потоков управляет CLR
- Parallel.ForEach<T>(IEnumerable<T>, Action<T>); - Выполнение делегата Action<T> над всеми элементами перечисления в максимально возможном числе потоков. Числом потоков управляет CLR
  - List<int> l = new List<int>();
  - public void dec(int i) {}
  - Parallel.For(0, 10, dec);
  - Parallel.ForEach<int>(l, dec);
  - Parallel.ForEach(l, dec);
- Parallel.Invoke(params Action[] actions) – выполнение делегатов в отдельных потоках, если возможно
  - Parallel.Invoke(Print, PrintToScreen, SendToEmail, () => Console.WriteLine("Печатаем"));
- Класс ParallelOptions может использоваться для подстройки операций Parallel
  - *MaxDegreeOfParallelism* – ограничивает максимально число одновременно выполняющихся задач в классом Parallel.
  - *CancellationToken* – позволяет отменять задания, выполняющиеся классом Parallel

Демонстрации

Parallel

Task

Запуск процесса

- Класс **Process**
- Запуск процесса
  - **Process.Start**(…)

```
Process process = Process.Start(@"d:\Программка.exe");
```

  - process.**Start**();

```
Process process = new Process(@" d:\Программка.exe ");
process.Start();
```

- Ожидание завершения процесса **WaitForExit**()

```
process.WaitForExit();
```

- Получение информации о запущенных процессах **Process.GetProcesses**()
  - Process[] processes = Process.GetProcesses();
- Завершение процесса **Kill**()
  - process.Kill();

Демонстрации

Запуск и контроль другого процесса