

# Многопоточное программирование



# Процесс

- ▶ Приложению в операционной системе соответствует - процесс. Процесс выделяет для приложения изолированное адресное пространство и поддерживает один или несколько потоков выполнения.
- ▶ `System.Diagnostics.Process`

```
Process current = Process.GetCurrentProcess();  
  
Console.WriteLine($"{current.Id} {current.ProcessName}  
                    { current.StartTime}");
```

```
9844 OOP_Lect03.12.2017 15:26:06
```

```
var allProcess = Process.GetProcesses(".");
```

*CurrentPriority*

*Id*

*StartAddress*

*StartTime*

...

## ► Управление процессами

```
Process calc = Process.Start("calc.exe");  
Thread.Sleep(5000);  
calc.Kill();
```

# домен приложения

- ▶ В .NET исполняемые файлы не обслуживаются прямо внутри процесса Windows. ОНИ обслуживаются в отдельном логическом разделе внутри процесса, который называется **доменом приложения** (*Application Domain — AppDomain*)
- ▶ В процессе может содержаться несколько доменов приложений
- ▶ **Класс System.AppDomain**

# Домен приложения

- ▶ 1) существуют внутри процессов
- ▶ 2) содержат загруженные сборки
- ▶ 3) процесс запускает при старте домен по умолчанию (**AppDomain.CurrentDomain**)
- ▶ 4) домены могут создаваться и уничтожаться в ходе работы в рамках процесса (менее затраты по сравн. с процессами)

```
AppDomain newD = AppDomain.CreateDomain("New");  
newD.Load("имя сборки");  
AppDomain.Unload(newD);
```

- ▶ 5) обеспечивают уровень изоляции кода

## Процесс Windows

### AppDomain #1 (Default)

MyApp.exe

TypeLib.dll

System.dll

### AppDomain #2

ExternLib.dll

System.dll

# Поток выполнения (thread)


- ▶ CLR поддерживает многопоточность  
опирается на многопот . ОС
- ▶ namespace System.Threading
- ▶ Класс Thread

делегат, для выполнения в потоке



```
public Thread(ThreadStart start);  
public Thread(ThreadStart start, int maxStackSize);
```

Максимальный размер стека,  
выделяемый потоку (1 МБ)



```
Thread th = new Thread((new Point()).Move);  
th.Start();
```

# Потоки



**локальное хранилище потоков  
(Thread Local Storage — TLS)**



# Класс Thread

```
Context ctx = Thread.CurrentContext;
```

получает контекст, в котором выполняется поток

```
var currt = Thread.CurrentThread;
```

получает ссылку на выполняемый поток

```
Console.Write(" " + currt.Name);
```

имя потока

```
if (currt.IsAlive)
```

работает ли поток в текущий момент

```
{
```

```
    Console.Write("Working");
```

```
}
```

```
if (!currt.IsBackground)
```

является ли поток фоновым

```
{
```

```
    Console.Write("not Background");
```

```
}
```

- ▶ CLR делит потоки: фоновые и основные
- ▶ Процесс не может завершиться, пока не завершены все его основные потоки.
- ▶ Завершение процесса автоматически прерывает все фоновые потоки

```
Console.Write(" " + currnt.ManagedThreadId);
```

уникальный числовой идентификатор  
управляемого потока

# Класс Thread

## ► Приоритеты

```
Console.Write(currt.Priority); //Normal
```

## ► перечисление **ThreadPriority**:

- **Lowest**
- **BelowNormal**
- **Normal (по умолчанию)**
- **AboveNormal**
- **Highest**

CLR считывает и анализирует значение приоритета и на их основании выделяет данному потоку то или иное количество времени.

```
Thread thrd = new Thread((new Point()).Move)
    { Name = "Point Move",
      Priority =
          ThreadPriority.BelowNormal,
      IsBackground = true,

    };
```

настройка свойств потока

# Класс Thread

## ► Статус потока

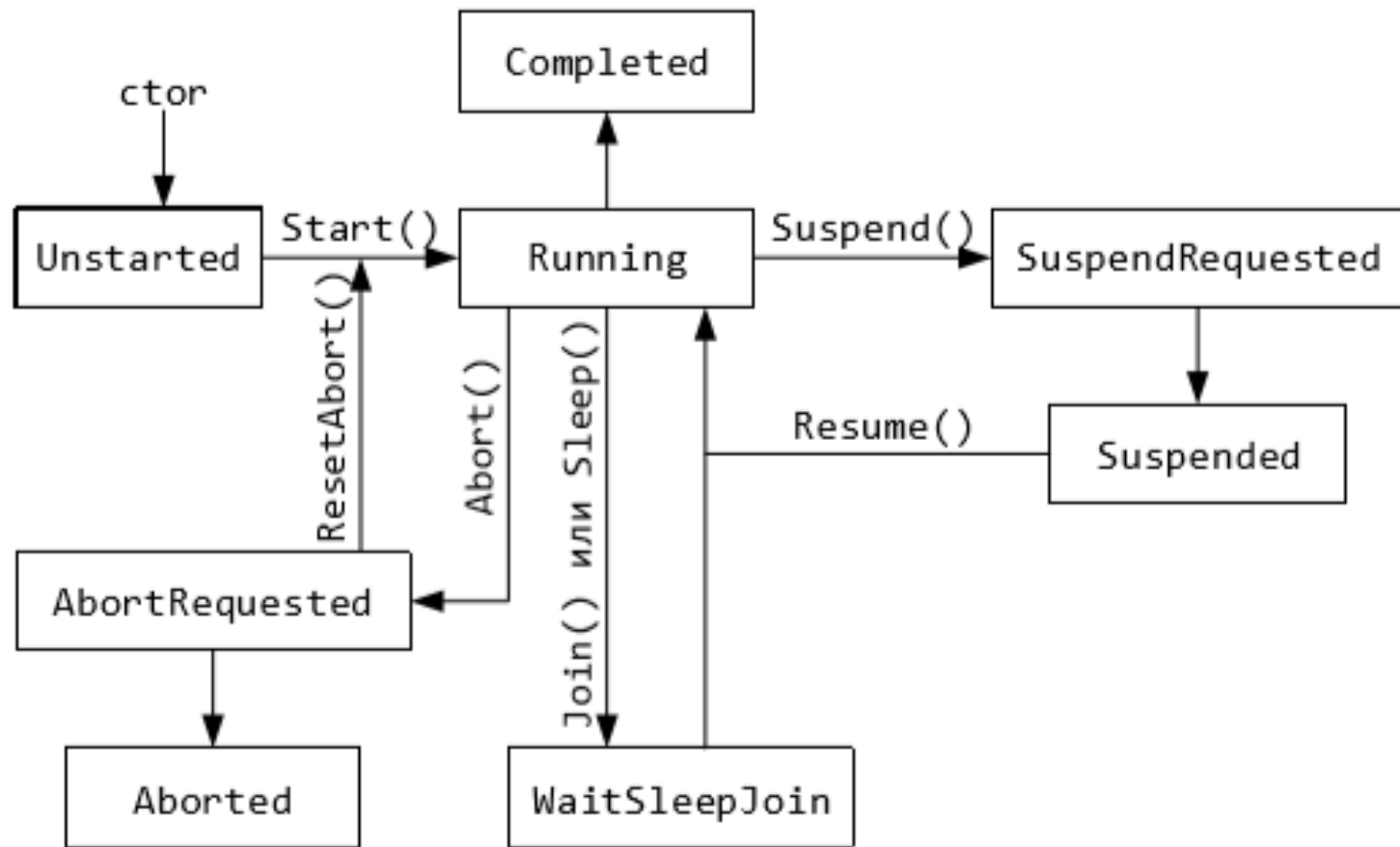
```
Console.WriteLine($"Статус потока: {currThread.ThreadState}");
```

```
Статус потока: Running
```

### Перечисление **ThreadState**:

- **Aborted**: поток остановлен, но пока еще окончательно не завершен
- **AbortRequested**: для потока вызван метод Abort, но остановка потока еще не произошла
- **Background**: поток выполняется в фоновом режиме
- **Running**: поток запущен и работает (не приостановлен)
- **Stopped**: поток завершен
- **StopRequested**: поток получил запрос на остановку
- **Suspended**: поток приостановлен
- **SuspendRequested**: поток получил запрос на приостановку
- **Unstarted**: поток еще не был запущен
- **WaitSleepJoin**: поток заблокирован в результате действия методов Sleep или Join

# Состояния потока



# Методы класса Thread:

- ▶ **GetDomain** - статический, возвращает ссылку домен приложения
- ▶ **GetDomainId** - статический, возвращает id домена приложения, в котором выполняется текущий поток
- ▶ **Sleep** – статический, останавливает поток на определенное количество миллисекунд
- ▶ **Abort** - уведомляет среду CLR о том, что надо прекратить поток (происходит не сразу)
- ▶ **Interrupt** - прерывает поток на некоторое время
- ▶ **Join** - блокирует выполнение вызвавшего его потока до тех пор, пока не завершится поток, для которого был вызван данный метод
- ▶ **Resume** - возобновляет работу приостановленного потока
- ▶ **Start** - запускает поток
- ▶ **Suspend** - приостанавливает поток
- ▶ **Yield** - передаёт управление следующему ожидающему потоку системы

## 1Workingnot Backgrou

[illegible]

```
Thread th2 = new Thread(ThreadClass.ThreadProc);
th2.Start();
Thread.Sleep(3000);
th2.Abort();
th2.Join();
```



# делегат ThreadStart

```
Thread thrd = new Thread(new ThreadStart((new Point()).Draw))
```



# Пул потоков

Для уменьшения издержек, связанных с созданием потоков, платформа .NET поддерживает специальный механизм, называемый пул потоков. Пул состоит из двух основных элементов:

очереди методов и  
рабочих потоков.

► ёмкость — максимальное число рабочих потоков

# Статический класс ThreadPool

- ▶ `SetMaxThreads()` - позволяет изменить ёмкость пула
- ▶ `SetMinThreads()` - устанавливает количество рабочих потоков, создаваемых без задержки
- ▶ `QueueUserWorkItem()` - помещение метода в очередь пула

```
ThreadPool.QueueUserWorkItem(Move);
```

# Синхронизация потоков

- ▶ координация действий для получения предсказуемого результата
- ▶ В потоках используются разделяемые ресурсы, общие для всей программы



```
class Program
{
    static int x = 0;
    static void Main(string[] args)
    {
        for (int i = 0; i < 5; i++)
        {
            Thread myThread = new Thread(Count);
            myThread.Name = "Поток " + i.ToString();
            myThread.Start();
        }

        Console.ReadLine();
    }
    public static void Count()
    {
        x++;
        Thread.Sleep(100 + x * x);
        x--;
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");

        Thread.Sleep(100+x*x);
    }
}
```

C:\Windows\system32\cmd.exe

Поток 0: 4  
Поток 2: 2  
Поток 1: 3  
Поток 4: 0  
Поток 3: 0

```
public class Unsafe
```

```
{
```

```
    private static int x, y;
```

```
    public void Div()
```

```
{
```

```
    if (y != 0) {
```

```
        Console.WriteLine(x / y);
```

```
    }
```

```
    y = 0;
```

```
}
```

```
}
```

критические секции

необходимо  
гарантировать  
выполнение операторов,  
только одним потоком  
в любой момент времени

# Оператор Lock

- ▶ определяет блок кода, внутри которого весь код блокируется и становится недоступным для других потоков до завершения работы текущего потока

```
class Program
```

```
{
```

```
    static int x = 0;
```

```
    static string objlocker = "null";
```

```
    static void Main(string[] args)
```

```
    {
```

```
        for (int i = 0; i < 5; i++)
```

```
        {
```

```
            Thread myThread = new Thread(Count);
```

```
            myThread.Name = "Поток " + i.ToString();
```

```
            myThread.Start();
```

```
        }
```

```
        Console.ReadLine();
```

```
    }
```

```
    public static void Count()
```

```
    {
```

```
        lock (objlocker)
```

```
        {
```

```
            x++;
```

```
            x--;
```

```
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
```

```
            Thread.Sleep(100 + x * x);
```

```
        }
```

```
    }
```

```
}
```



C:\Windows\system32\cmd.exe

Поток 0: 0

Поток 1: 0

Поток 2: 0

Поток 3: 0

Поток 4: 0

бъект-заглушка,

В операторе lock, объект objlocker блокируется, и на время его блокировки монопольный доступ к блоку кода имеет только один поток

Thread.Sleep(100 + x \* x);

выражение должно иметь ссылочный тип

# Monitor

- ▶ класс `System.Threading.Monitor`
- ▶ `Monitor.Enter()` - вход в критическую секцию,
- ▶ `Monitor.Exit()` – выход из секции.
- ▶ Вход и выход должны выполняться в одном и том же потоке.
- ▶ Аргументами методов является объект-идентификатор критической секции.



```
class Program
```

```
{
```

```
    static int x = 0;
```

```
    static string objlocker = "null";
```

идентификатором критической секции

```
...
```

```
public static void Count()
```

```
{
```

```
    try
```

```
    {
```

```
        Monitor.Enter(objlocker);
```

```
        {
```

```
            x++;
```

```
            Thread.Sleep(100 + x * x);
```

```
            x--;
```

```
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
```

```
            Thread.Sleep(100 + x * x);
```

```
        }
```

```
    }
```

```
    finally
```

```
    {
```

```
        Monitor.Exit(objlocker);
```

```
    }
```

```
}
```

Входит в критическую секцию  
блокирует объект objlocker

Выходит из критической секции  
освобождение объекта locker, и  
он становится доступным для  
других потоков.

# Мьютекс

- ▶ `System.Threading.Mutex`
- ▶ позволяет организовать критическую секцию для нескольких процессов
- ▶ `WaitOne()` - входа в критическую секцию,
- ▶ `ReleaseMutex()` – для выхода из неё (выход может быть произведён только в том же потоке выполнения, что и вход).

создаем объект мьютекса

```
static Mutex mutex = new Mutex();
```

Поток 0: 0  
Поток 1: 0  
Поток 2: 0  
Поток 3: 0  
Поток 4: 0

...

```
public static void Count()
```

```
{
```

```
    mutex.WaitOne();
```

```
    {
```

```
        x++;
```

```
        Thread.Sleep(100 + x * x);
```

```
        x--;
```

```
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
```

```
        Thread.Sleep(100 + x * x);
```

```
    }
```

```
    mutex.ReleaseMutex();
```

```
}
```

приостанавливает выполнение  
потока до тех пор, пока не буд  
получен мьютекс

поток освобождает его

# Семафор

- ▶ объект синхронизации, позволяющий войти в заданный участок кода не более чем  $N$  потокам ( $N$  – ёмкость семафора)
- ▶ получение и снятие блокировки в случае семафора может выполняться из разных потоков
- ▶ классы `System.Threading.Semaphore` (между процессами) и `SemaphoreSlim` (в рамках одного процесса)
- ▶ `Wait()` - получение блокировки,
- ▶ `Release()` – снятие блокировки

```

public class ThePool
{
    // ёмкость семафора равна 3
    private static SemaphoreSlim sema = new SemaphoreSlim(3);

    public static void Main() {
        for (var i = 1; i <= 10; i++)
            new Thread(Enter).Start(i);
    }
    private static void Enter(object id)
    {
        Console.WriteLine(id + " enter");
        sema.Wait();
        Console.WriteLine(id + " is swimming");
        Thread.Sleep(1000 * (int) id);
        Console.WriteLine(id + " is leaving");

        sema.Release();
    }
}

```

C:\Windows\system

```

2 is swimming
1 enter
1 is swimming
3 enter
3 is swimming
4 enter
5 enter
6 enter
7 enter
8 enter
9 enter
10 enter
1 is leaving
4 is swimming
2 is leaving
5 is swimming
3 is leaving
6 is swimming
4 is leaving
7 is swimming
5 is leaving
8 is swimming
6 is leaving
9 is swimming
7 is leaving
10 is swimming
8 is leaving
9 is leaving
10 is leaving
Для продолжения

```

# ReaderWriterLockSli

- ▶ ресурс нужно блокировать так, чтобы читать его могли несколько потоков, а записывать – только один
- ▶ EnterReadLock() и ExitReadLock() задают секцию чтения ресурса,
- ▶ EnterWriteLock() и ExitWriteLock() – секцию записи ресурса.

# Синхронизация на основе подачи сигналов

- ▶ на основе подачи сигналов
- ▶ AutoResetEvent
- ▶ ManualResetEvent
- ▶ ManualResetEventSlim
- ▶ CountdownEvent
- ▶ Barrier

# System.Threading.Timer

- ▶ позволяет запускать определенные действия по истечению некоторого периода времени

```
int num = 0;  
// устанавливаем метод обратного вызова  
TimerCallback tm = new TimerCallback(Enter);  
// создаем таймер  
Timer timer = new Timer(tm, num, 0, 2000);
```

- объект, передаваемый в качестве параметра в метод Count
- количество миллисекунд, через которое таймер будет запускаться. В данном случае таймер будет запускаться немедленно после создания, так как в качестве значения используется 0
- интервал между вызовами метода Count



# атрибут [ThreadStatic]

- ▶ применяется к статическим полям
- ▶ поле помечено таким атрибутом, то каждый поток будет содержать свой экземпляр поля

```
public class ClassThread
{
    public static int SharedField = 25;

    [ThreadStatic]
    public static int NonSharedField;
}
```

# ThreadLocal<T>

- Для создания неразделяемых статических полей

```
public class Slot
{
    private static readonly Random rnd = new Random();

    private static ThreadLocal<int> NonShared =
        new ThreadLocal<int>(() => rnd.Next(1, 20));

    ...
    NonShared.Value;
}
```

функция инициализации поля