Многопоточное программирование

Процесс

- Приложению в операционной системе соответствует – процесс (концепция уровня ОС). Процесс выделяет для приложения изолированное адресное пространство и поддерживает один или несколько потоков выполнения.

О процессе

- 1) для каждого загружаемого в память файла *.exe в операционной системе создается отдельный изолированный процесс, который используется на протяжении всего времени его существования
- 2) выход из строя одного процесса никак не сказывается на работе других процессов
- 3) доступ напрямую к данным в одном процессе из другого процесса невозможен (API - распределенных вычислений Windows Communication Foundation)
- 4) каждый процесс Windows получает уникальный *идентификатор процесса (Process ID — PID)*
- 5) может независимо загружаться и выгружаться операционной системой (в том числе программно)

Process

- позволяет управлять уже запущенными процессами, а также запускать новые.
- System.Diagnostics

```
Process current = Process.GetCurrentProcess();

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

Свойство Handle: возвращает дескриптор процесса

Свойство Id: получает уникальный идентификатор процесса в рамках текущего сеанса ОС

Свойство MachineName: возвращает имя компьютера, на котором запущен процесс

Свойство MainModule: представляет основной модуль - исполняемый файл программы, представлен объектом типа ProcessModule

Свойство Modules: получает доступ к коллекции ProcessModuleCollection, которая в виде объектов ProcessModule хранит набор модулей (например, файлов dll и exe), загруженных в рамках данного процесса

Свойство ProcessName: возвращает имя процесса, которое нередко совпадает с именем приложения

Свойство StartTime: возвращает время, когда процесс был запущен

Свойство PageMemorySize64: возвращает объем памяти, который выделен для данного процесса

Свойство VirtualMemorySize64: возвращает объем виртуальной памяти, который выделен для данного процесса

Метод CloseMainWindow(): закрывает окно процесса, который имеет графический интерфейс

Метод GetProcesses(): возвращает массив всех запущенных процессов

Метод GetProcessesByName(): возвращает процессы по его имени. Так как можно запустить несколько копий одного приложения, то возвращает массив

Метод GetProcessById(): возвращает процесс по Id. Так как можно запустить несколько копий одного приложения, то возвращает массив

Метод Kill(): останавливает процесс

Метод Start(): запускает новый процесс

- Получить информацию о обо всех процессах системы

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

```
using System.Diagnostics;

var process = Process.GetCurrentProcess();
Console.WriteLine($"Id: {process.Id}");
Console.WriteLine($"Name: {process.ProcessName}");
Console.WriteLine($"VirtualMemory: {process.VirtualMemorySize64}");
```

- Управление процессами

```
Process calc = Process.Start("calc.exe");
Thread.Sleep(4000);
 calc.Kill();
```

Получим id процессов, который представляют запущенные экземпляры Visual Studio

```
using System.Diagnostics;

Process[] vsProcs = Process.GetProcessesByName("devenv");    // ДЛЯ Windows
// Process[] vsProcs = Process.GetProcessesByName("VisualStudio"); //  ДЛЯ MacO
foreach (var proc in vsProcs)
    Console.WriteLine($"ID: {proc.Id}");
```

ProcessModule

класс Prosess имеет свойство Modules, которое представляет объект ProcessModuleCollection

свойства:

BaseAddress: адрес модуля в памяти

FileName: полный путь к файлу модуля

EntryPointAddress: адрес функции в памяти, которая запустила модуль

ModuleName: название модуля (краткое имя файла)

ModuleMemorySize: возвращает объем памяти, необходимый для загрузки модуля

Получим все модули

```
using System.Diagnostics;

Process proc = Process.GetProcessesByName("devenv")[0]; // ДЛЯ Windows
// Process proc = Process.GetProcessesByName("VisualStudio")[0]; // ДЛЯ MacOS
ProcessModuleCollection modules = proc.Modules;

foreach(ProcessModule module in modules)
{
    Console.WriteLine($"Name: {module.ModuleName}  FileName: {module.FileName}"
}
```

Запуск нового процесса. Process.Start()

При обращении к исполняемому файлу .NET запускает приложение

```
// обращение к исполняемой программе
Process.Start(@"C:\Program Files\Google\Chrome\Application\chrome");

// Process.Start("/Applications/Google Chrome.app/Contents/MacOS/Google Chrome")
```

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**

О домене 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);
```

Выгрузить сборки из домена нельзя, можно выгрузить весь домен

- 5) обеспечивают уровень изоляции кода

Свойство BaseDirectory: базовый каталог, который используется для получения сборок (как правило, каталог самого приложения)

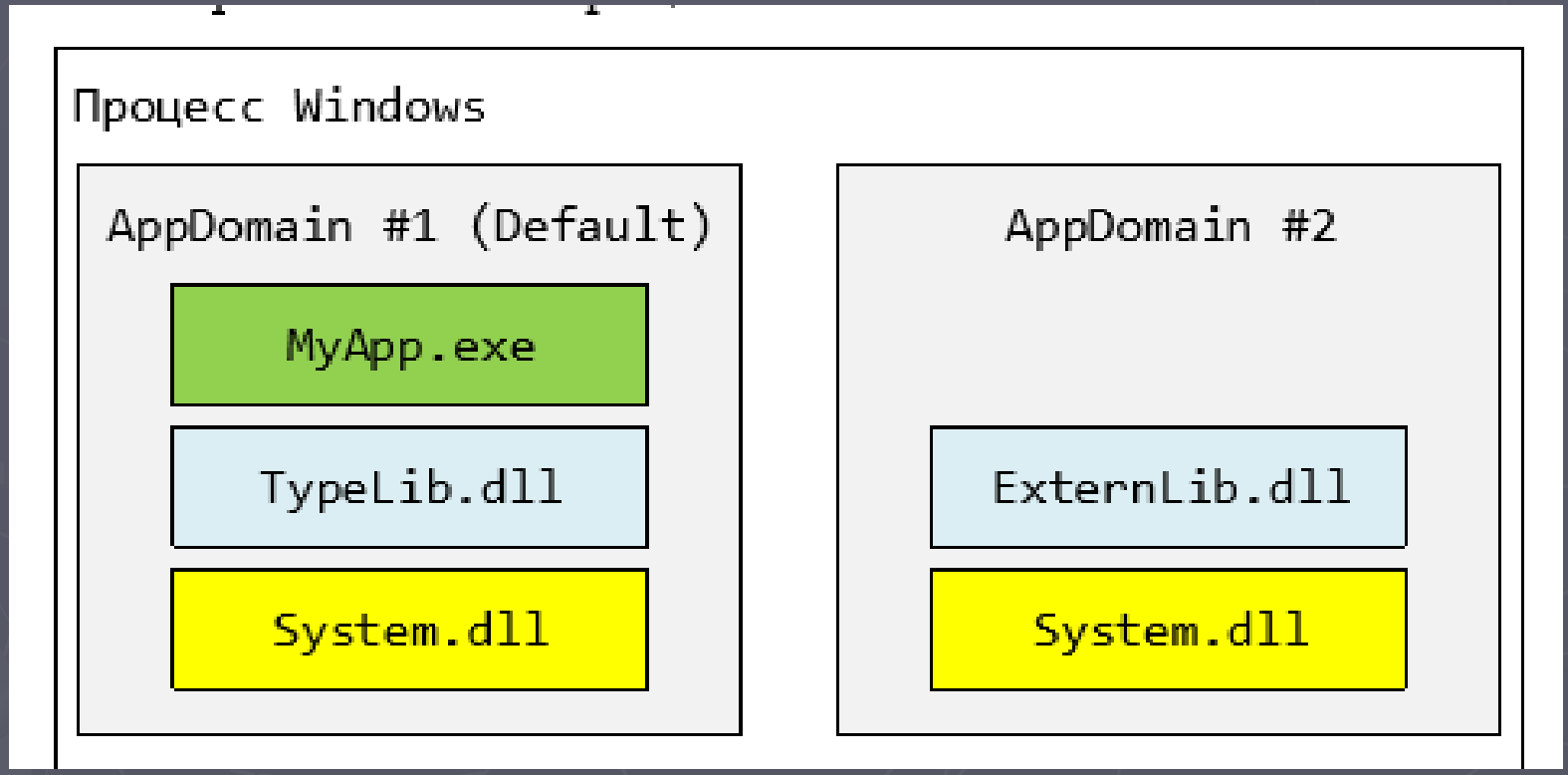
Свойство CurrentDomain: домен текущего приложения

Свойство FriendlyName: имя домена приложения

Свойство SetupInformation: представляет объект AppDomainSetup и хранит конфигурацию домена приложения

Метод ExecuteAssembly(): запускает сборку exe в рамках текущего домена приложения

Метод GetAssemblies(): получает набор сборок .NET, загруженных в домен приложения


Процесс Windows
AppDomain #1 (Default)
MyApp.exe
TypeLib.dll
System.dll
AppDomain #2
ExternLib.dll
System.dll

```
using System.Reflection;

AppDomain domain = AppDomain.CurrentDomain;
Console.WriteLine($"Name: {domain.FriendlyName}");
Console.WriteLine($"Base Directory: {domain.BaseDirectory}")
Console.WriteLine();

Assembly[] assemblies = domain.GetAssemblies();
foreach (Assembly asm in assemblies)
    Console.WriteLine(asm.GetName().Name);
```

```
Name: HelloApp
Base Directory: /Users/eugene/Projects/HelloApp/HelloApp/bin/Debug/net6.0/

System.Private.CoreLib
HelloApp
System.Runtime
System.Console
System.Threading
Microsoft.Win32.Primitives
System.Collections
System.Memory
```

О Потоках

- **Поток** - используемый внутри процесса путь выполнения
  - CLR поддерживает многопоточность опирается на многопот . ОС
  - В каждом процессе Windows содержится первоначальный "поток", который является входной точкой для приложения (метод Main())
  - поток, который создается первым во входной точке процесса, называется *главным потоком (primary thread)*.
  - Главный поток создается автоматически
  - Процессы, в которых содержится единственный главный поток выполнения, изначально являются *безопасными потоками (thread safe)*,

два типа потоков:

- основной
- фоновый

- если первым завершится основной поток, то фоновые потоки в его процессе будут также принудительно остановлены
- если же первым завершится фоновый поток, то это не повлияет на остановку основного потока — тот будет продолжать функционировать до тех пор, пока не выполнит всю работу и самостоятельно не остановится

Обычно при создании потока ему по-умолчанию присваивается основной тип.

Потоки

локальное хранилище потоков
(Thread Local Storage – TLS)

Чтобы поток не забывал, на чем он работал перед тем, как его выполнение было приостановлено, каждому потоку предоставляется возможность записывать данные в *локальное хранилище потоков (Thread Local Storage — TLS)* и выделяется отдельный стек вызовов,

Многопоточное приложение

отдельные компоненты работают одновременно (псевдоодновременно), не мешая друг другу.

Случаи использования многопоточности:

- выполнение длительных процедур, ходом выполнения которых надо управлять;
- функциональное разделение программного кода: пользовательский интерфейс – функции обработки информации;
- обращение к серверам и службам Интернета, базам данных, передача данных по сети;
- одновременное выполнение нескольких задач, имеющих различный приоритет.

- 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("   " + currt.ManagedThreadId);
```

уникальный числовой идентификатор управляемого потока

Виды многопоточности

- Переключательная многопоточность.

Основа – резидентные программы.

Программа размещалась в памяти компьютера вплоть до перезагрузки системы, и управление ей передавалось каким-либо заранее согласованным способом (предопределенной комбинацией клавиш на клавиатуре).

- Совместная многопоточность.

Передача управления от одной программы другой. При этом возвращение управления – это проблема выполняемой программы. Возможность блокировки, при которой аварийно завершаются ВСЕ программы.

- Вытесняющая многопоточность.

ОС централизованно выделяет всем запущенным приложениям определенный квант времени для выполнения в соответствии с приоритетом приложения. Реальная возможность работы нескольких приложений в ПСЕВДОПАРАЛЛЕЛЬНОМ режиме.

"Зависание" одного приложения не является крахом для всей системы и оставшихся приложений.

Класс Thread

- System.Threading
- представляет управляемые потоки.

Статические члены класса Thread	Назначение
CurrentThread	Свойство. Только для чтения. Возвращает ссылку на поток, выполняемый в настоящее время.
GetData() SetData()	Обслуживание слота текущего потока.
GetDomain() GetDomainID()	Получение ссылки на домен приложения (на ID домена), в рамках которого работает указанный поток.
Sleep()	Блокировка выполнения потока на определенное время.

Нестатические члены	Назначение
IsAlive	Свойство. Если поток запущен, то true
IsBackground	Свойство. Работа в фоновом режиме. GC работает как фоновый поток.
Name	Свойство. Дружественное текстовое имя потока. Если поток никак не назван – значение свойства установлено в null. Поток может быть поименован единожды. Попытка переименования потока возбуждает исключение.
Priority	Свойство. Значение приоритета потока. Область значений – значения перечисления ThreadPriority.
ThreadState	Свойство. Состояние потока. Область значений – значения перечисления ThreadState.

Класс Thread

```
Context ctx = Thread.CurrentContext;


var currt = Thread.CurrentThread;

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


if (currt.IsAlive)
     {
         Console.Write("Working");
      }

if (!currt.IsBackground)
             {
                 Console.Write("not Background");
             }
```

получает контекст, в котором выполняется поток

получает ссылку на выполняемый поток

имя потока

работает ли поток в текущий момент

является ли поток фоновым

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: {currt.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

Для создания потока применяется один из конструкторов класса Thread:

делегат, инкапсулирующий метод для выполнения в потоке

```
public Thread(ThreadStart start);
public Thread(ParameterizedThreadStart start);
```

при запуске метода передает ему данные в виде объекта

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

Максимальный размер стека, выделяемый потоку (резервирует 1 МБ)

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

Запуск потока

Делегат ThreadStart

```
public delegate void ThreadStart();
```

```
using System.Threading;

// создаем новый поток
Thread myThread1 = new Thread(Print);
Thread myThread2 = new Thread(new ThreadStart(Print));
Thread myThread3 = new Thread(()=>Console.WriteLine("Hello Threads"));

myThread1.Start();  // запускаем поток myThread1
myThread2.Start();  // запускаем поток myThread2
myThread3.Start();  // запускаем поток myThread3

void Print() => Console.WriteLine("Hello Threads");
```

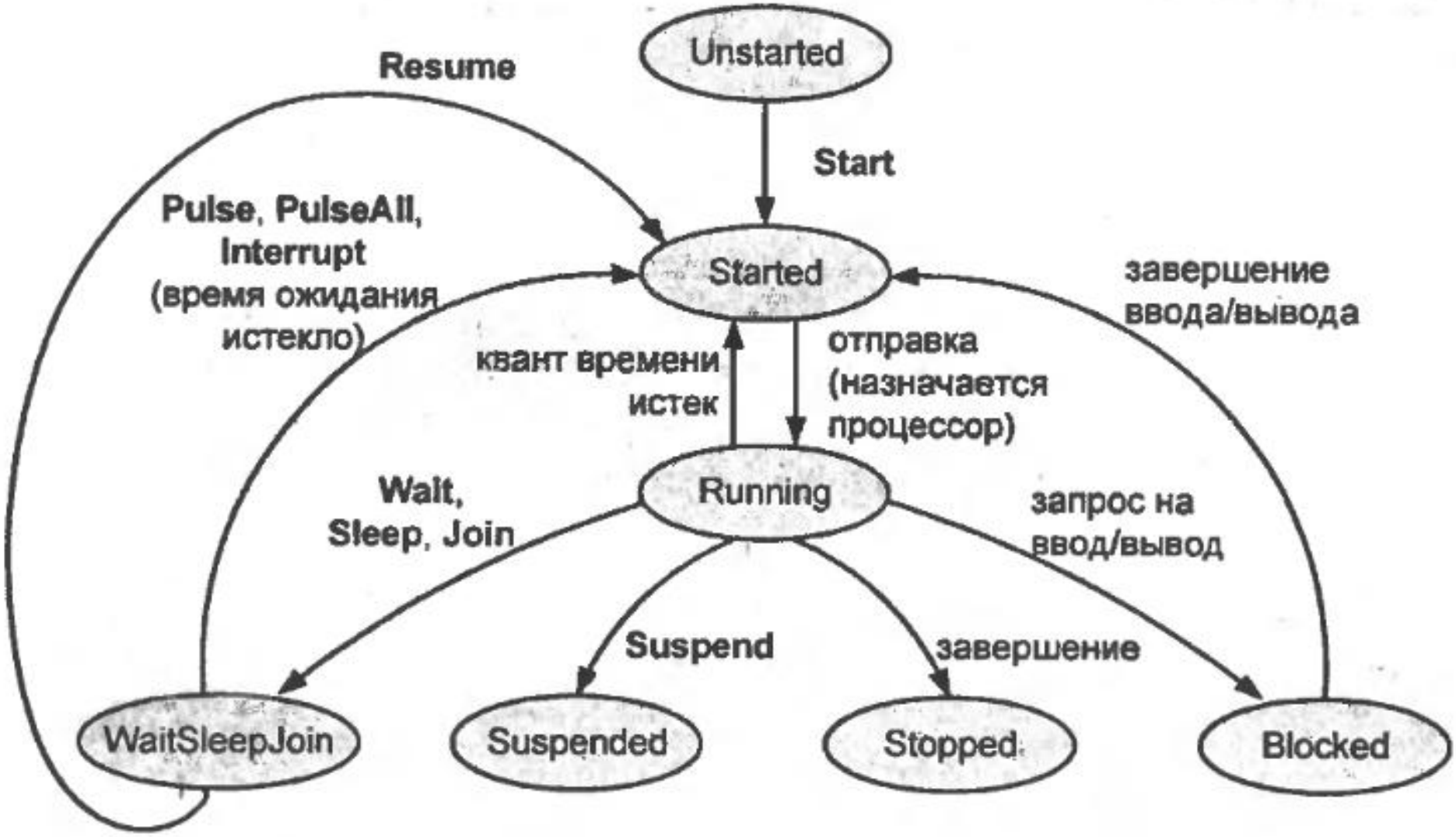
```
using System.Threading;

// создаем новый поток
Thread myThread = new Thread(Print);
// запускаем поток myThread
myThread.Start();

// действия, выполняемые в главном потоке
for (int i = 0; i < 5; i++)
{
    Console.WriteLine($"Главный поток: {i}");
    Thread.Sleep(300);
}

// действия, выполняемые во втором потокке
void Print()
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine($"Второй поток: {i}");
        Thread.Sleep(400);
    }
}
```

```
Главный поток: 0
Второй поток: 0
Главный поток: 1
Второй поток: 1
Главный поток: 2
Второй поток: 2
Главный поток: 3
Второй поток: 3
Главный поток: 4
Второй поток: 4
```

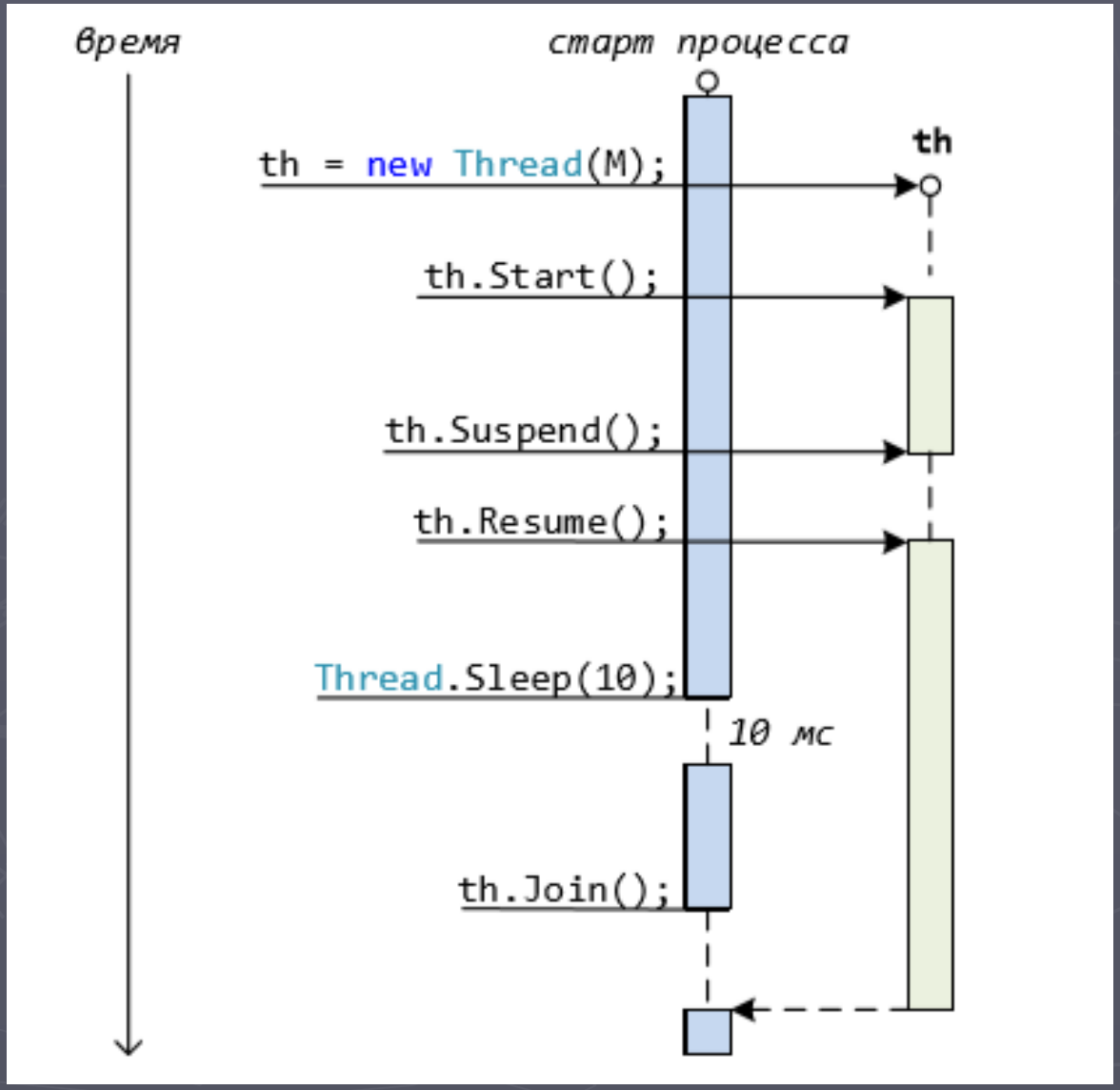
Методы класса Thread:

- **GetDomain - c**татический, возвращает ссылку домен приложения
- **GetDomainId - c**татический, возвращает id домена приложения, в котором выполняется текущий поток
- **Sleep – c**татический, 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

Жизненный цикл потока



Временная диаграмма работы потоков



метод Abort()

прерывание потока до его нормального завершения

- генерирует исключение ThreadAbortException

- если поток требуется остановить перед тем, как продолжить выполнение программы, то после метода Abort() следует сразу же вызвать метод Join().
- Обычно поток должен завершаться естественным образом.

public void Abort(object stateInfo)

где stateInfo обозначает любую информацию, которую требуется передать потоку, когда он останавливается.

```
public static class ThreadClass
        {
            public static void ThreadProc()
            {
                while (true)
                {
                    try
                    {
                        Console.WriteLine("Работаю ...");
                        Thread.Sleep(1000);
                    }
                    catch (ThreadAbortException e)
                    {
                        Console.WriteLine("Запрос Abort!");
                        Thread.ResetAbort();
                    }
                }
            }
        }
```

исключается повторное генерирование исключения по завершении обработчика исключения

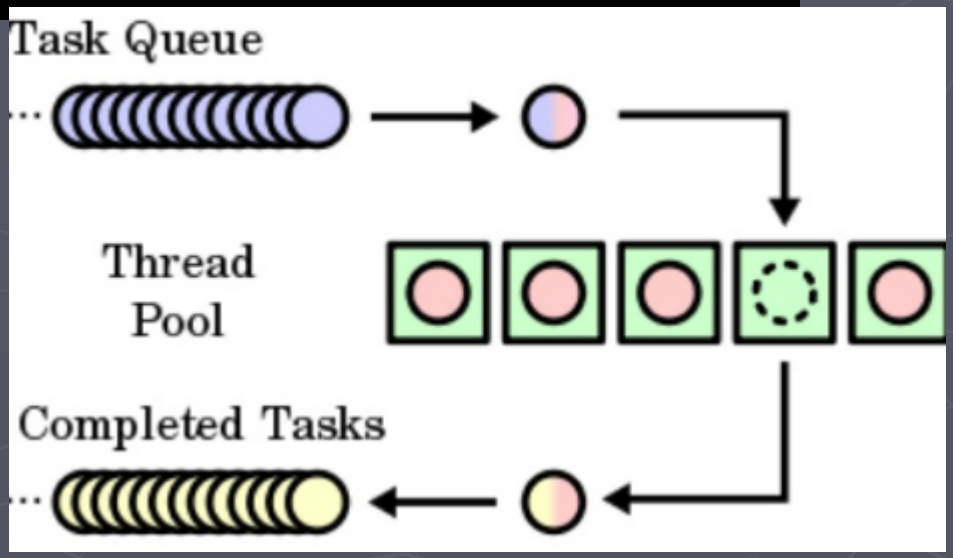
```
Thread th2 = new Thread(ThreadClass.ThreadProc);
th2.Start();
Thread.Sleep(3000);
th2.Abort();
th2.Join();
```

```
    1Workingnot Backgrour
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Запрос Abort!
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
Работаю ...
```

Пул потоков

Для уменьшения издержек, связанных с созданием потоков, платформа .NET поддерживает специальный механизм, называемый пул потоков. Пул состоит из двух основных элементов:
1) очереди методов
2) рабочих потоков.

- ёмкость – максимальное число рабочих потоков



Статический класс ThreadPool

- ThreadCount - возвращает текущее количество потоков в пуле потоков
- SetMaxThreads() - позволяет изменить ёмкость пула
- SetMinThreads() - устанавливает количество рабочих потоков, создаваемых без задержки
- QueueUserWorkItem() - помещение метода в очередь пула

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

```
public static void Main()
{
    // Queue the task.
    ThreadPool.QueueUserWorkItem(ThreadProc);
    Console.WriteLine("Main thread does some work, then sleeps.");
    Thread.Sleep(1000);

    Console.WriteLine("Main thread exits.");
}

// This thread procedure performs the task.
static void ThreadProc(Object stateInfo)
{
    // No state object was passed to QueueUserWorkItem, so stateInfo is null.
    Console.WriteLine("Hello from the thread pool.");
}
```

Если закомментировать вызов Thread.Sleep метода, основной поток завершает работу перед выполнением метода в потоке пула потоков.

Когда не следует использовать потоки из пула потоков

Вариант 1	Необходим не фоновый поток
Вариант 2	Поток должен иметь определенный приоритет
Вариант 3	Имеются задачи, которые приводят к блокировке потока на длительное время. Для пула потоков определено максимальное количество потоков, поэтому большое число заблокированных потоков в пуле может препятствовать запуску задач.

предполагаем, что метод выведет все значения x
от 1 до 5. И так для каждого потока.

```
int x = 0;

// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";    // устанавливаем имя для каждо
    myThread.Start();
}

void Print()
{
    x = 1;
    for (int i = 1; i < 6; i++)
    {
        Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
        x++;
        Thread.Sleep(100);
    }
}
```

в реальности в процессе работы будет происходить переключение между потоками, и значение переменной x становится непредсказуемым.

Решение проблемы состоит в том, чтобы синхронизировать потоки и ограничить доступ к разделяемым ресурсам на время их использования каким-нибудь потоком

```
Поток 1: 1
Поток 5: 1
Поток 4: 1
Поток 2: 1
Поток 3: 1
Поток 1: 6
Поток 5: 7
Поток 3: 7
Поток 2: 7
Поток 4: 9
Поток 1: 11
Поток 4: 11
Поток 2: 11
Поток 3: 14
Поток 5: 11
Поток 1: 16
Поток 2: 16
Поток 3: 16
Поток 5: 18
```

Синхронизация потоков

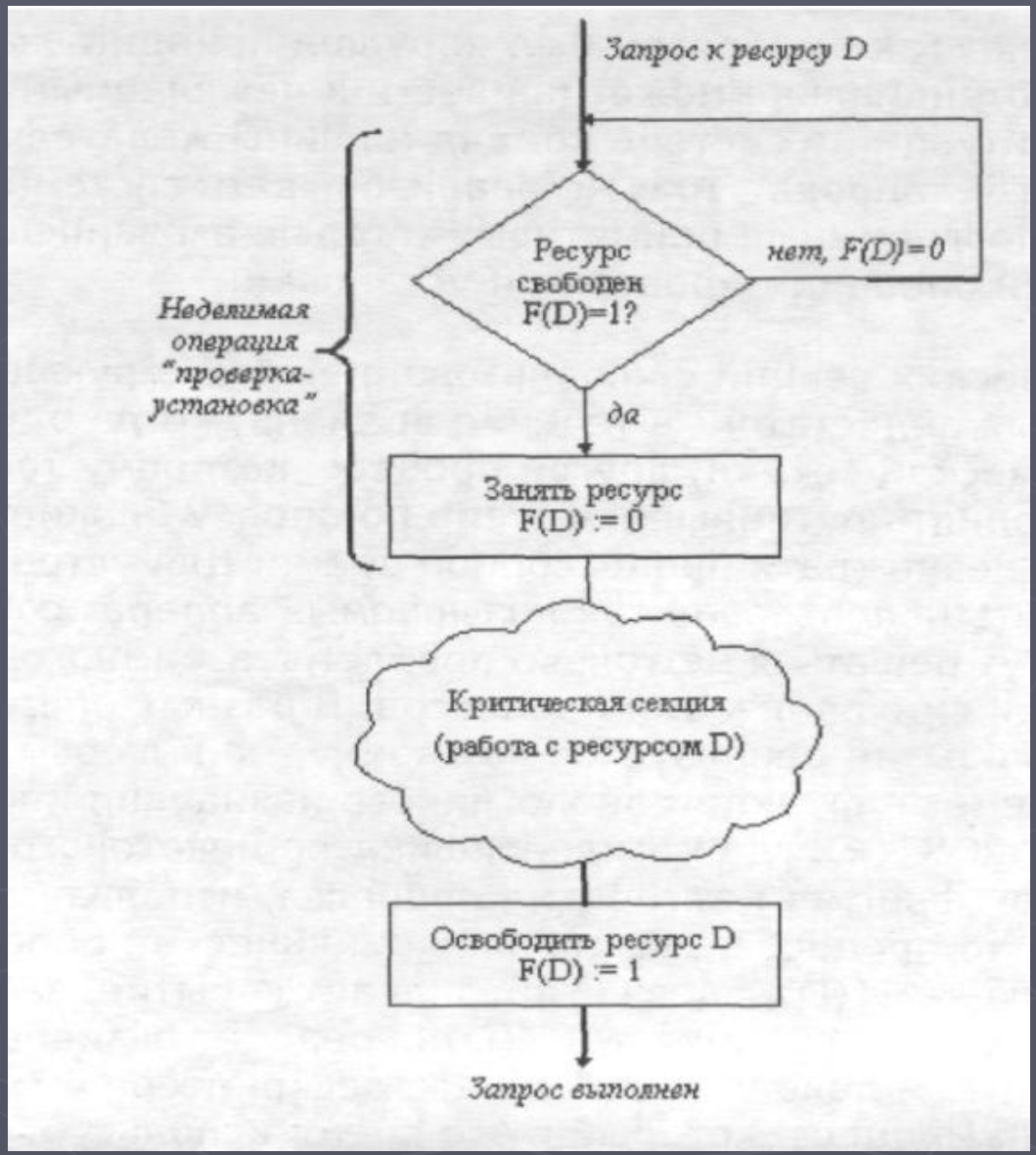
- координация действий для получения предсказуемого результат
- В потоках используются разделяемые ресурсы, общие для всей программы

Способы синхронизации потоков

- Монитор (Monitor)
- AutoResetEvent
- Мьютекс (Mutex)
- Семафор (Semaphore)

Критическая секция — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком исполнения.

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.
- Остальный потоки помещаются в очередь ожидания и ждут, пока текущий поток не освободит данный блок кода.

```
lock (переменная)
{
    ...
}
```

Необходимо

- Как можно быстрее освобождать блокировку
- Избегать взаимоблокировок

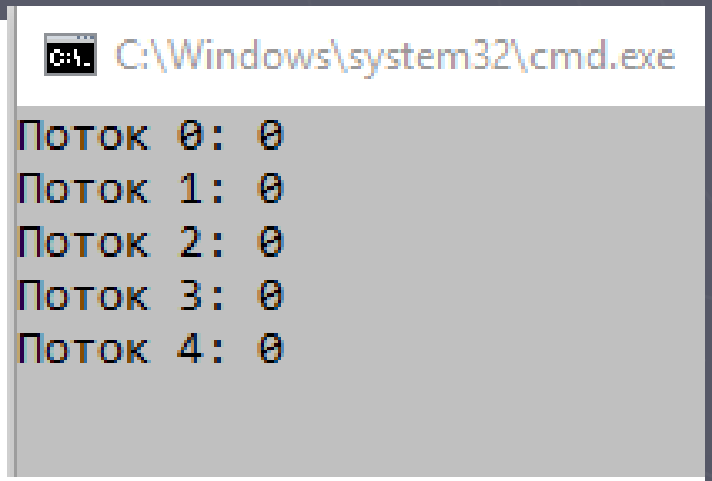
```
lock (A)
{
    lock (B)
    {
    }
}
```

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

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

Оператор Lock

```
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);
                }
            }
            }
        }
```



Объект-заглушка,

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

выражение должно иметь ссылочный тип

```
int x = 0;
object locker = new();  // объект-заглушка
// запускаем пять потоков
for (int i = 1; i < 6; i++)
{
    Thread myThread = new(Print);
    myThread.Name = $"Поток {i}";
    myThread.Start();
}


void Print()
{
    lock (locker)
    {
        x = 1;
        for (int i = 1; i < 6; i++)
        {
            Console.WriteLine($"{Thread.CurrentThread.Name}: {x}");
            x++;
            Thread.Sleep(100);
```

```
Поток 1: 1
Поток 1: 2
Поток 1: 3
Поток 1: 4
Поток 1: 5
Поток 5: 1
Поток 5: 2
Поток 5: 3
Поток 5: 4
Поток 5: 5
Поток 3: 1
Поток 3: 2
Поток 3: 3
Поток 3: 4
Поток 3: 5
Поток 2: 1
```

Monitor

механизм взаимодействия и **синхронизации** процессов, обеспечивающий доступ к неразделяемым ресурсам.

- Monitor.Enter() - вход в критическую секцию, увеличение блокировок на 1
- Monitor.Exit() – выход из секции (-1 блок-ка)
- Вход и выход должны выполняться в одном и том же потоке.
- Аргументами методов является объект-идентификатор критической секции.

```
Monitor.Enter (переменная)
...
Monitor.Exit (переменная)
```

void Pulse (object obj): уведомляет поток из очереди ожидания, что текущий поток освободил объект obj

void PulseAll(object obj): уведомляет все потоки из очереди ожидания, что текущий поток освободил объект obj. После чего один из потоков из очереди ожидания захватывает объект obj.

bool TryEnter (object obj): пытается захватить объект obj. Если владение над объектом успешно получено, то возвращается значение true

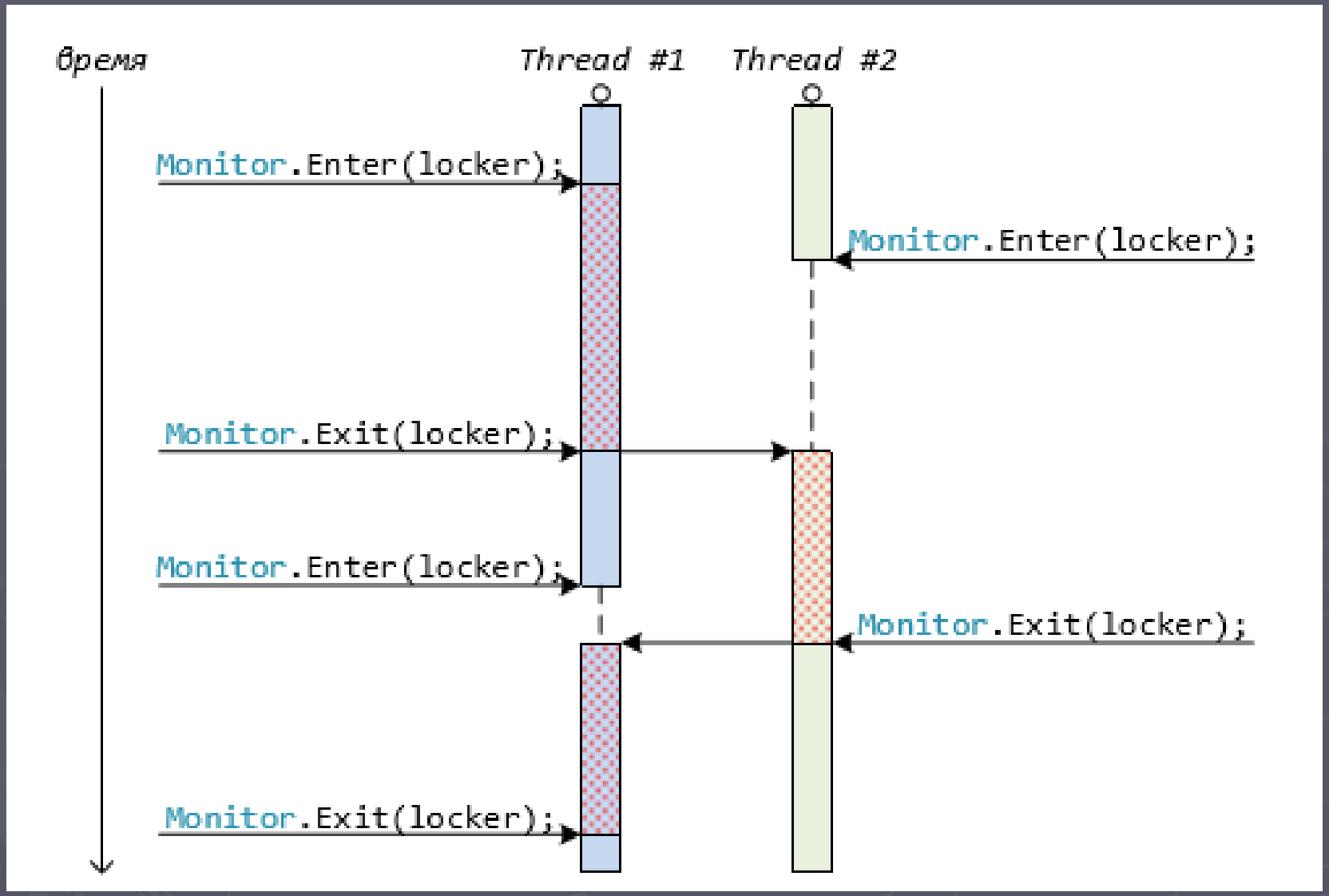
bool Wait (object obj): освобождает блокировку объекта и переводит поток в очередь ожидания объекта. Следующий поток в очереди готовности объекта блокирует данный объект. А все потоки, которые вызвали метод Wait, остаются в очереди ожидания, пока не получат сигнала от метода Monitor.Pulse или Monitor.PulseAll, посланного владельцем блокировки.

```
 class Program
        {
            static int x = 0;
           static  string objlocker = "null";
...

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);
                }
            }
```

идентификатор критической секции

Входит в критическую секцию блокирует объект objlocker

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


Время
Thread #1
Thread #2
Monitor.Enter(locker);
Monitor.Enter(locker);
Monitor.Exit(locker);
Monitor.Enter(locker);
Monitor.Exit(locker);
Monitor.Exit(locker);

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).

```
Mutex myMutex = new Mutex();
   ...
myMutex.WaitOne();
   ...
myMutex.ReleaseMutex();
```

создаем объект мьютекса

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

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


...
            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. мьютек получает один из ожидающих потоков.

Изначально мьютекс свободен, поэтому его получает один из потоков.

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 потоках
- классы System.Threading.Semaphore (между процессами) и SemaphoreSlim (в рамках одного процесса)
- Wait() - получение блокировки,
- Release() – снятие блокировки

```
Semaphore mySem = new Semaphore(3, 3);
    ...
mySem.WaitOne();
    ...
mySem.Release();
```

Конструкторы класса Semaphore

- Semaphore (int initialCount, int maximumCount):

initialCount задает начальное количество потоков, maximumCount - максимальное количество потоков, которые имеют доступ к общим ресурсам

- Semaphore (int initialCount, int maximumCount, string? name): в дополнение задает имя семафора

- Semaphore (int initialCount, int maximumCount, string? name, out bool createdNew):

createdNew при значении true указывает, что новый семафор был успешно создан. Если этот параметр равен false, то семафор с указанным именем уже существует

Например, у нас такая задача: есть некоторое число читателей, которые приходят в библиотеку три раза в день и что-то там читают. И пусть у нас будет ограничение, что единовременно в библиотеке не может находиться больше трех читателей.

```
class Reader
{
    // создаем семафор
    static Semaphore mySem = new Semaphore(3, 3);

    Thread myThread;
    int count = 3;// счетчик чтения

    public Reader(int i)
    {
        myThread = new Thread(Read);
        myThread.Name = "Читатель " + i.ToString();
        myThread.Start();
    }

    public void Read()
    {
        while (count > 0)
        {
            mySem.WaitOne();   освобождаем место

            Console.WriteLine("{0} входит в библиотеку", Thread.Current
Thread.Name);

            Console.WriteLine("{0} читает", Thread.CurrentThread.Name);
            Thread.Sleep(1000);

            Console.WriteLine("{0} покидает библиотеку", Thread.Curren
tThread.Name);

            mySem.Release();

            count--;
            Thread.Sleep(1000);
        }
```

```
static void Main(string[] args)
{
    for (int i = 1; i < 6; i++)
    {
        Reader reader = new Reader(i);
    }

    Console.ReadLine();
}
```

```
Читатель 5 входит в библиотеку
Читатель 5 читает
Читатель 4 входит в библиотеку
Читатель 4 читает
Читатель 1 входит в библиотеку
Читатель 1 читает
Читатель 5 покидает библиотеку
Читатель 1 покидает библиотеку
Читатель 4 покидает библиотеку
Читатель 3 входит в библиотеку
Читатель 3 читает
Читатель 2 входит в библиотеку
Читатель 2 читает
Читатель 4 входит в библиотеку
Читатель 3 покидает библиотеку
Читатель 2 покидает библиотеку
Читатель 5 входит в библиотеку
```

ReaderWriterLockSlim

- ресурс нужно блокировать так, чтобы читать его могли несколько потоков, а записывать – только один

два вида замков:

- Чтение блокировки
- Блокировка записи

- EnterReadLock() и ExitReadLock() задают секцию чтения ресурса,
- EnterWriteLock() и ExitWriteLock() – секцию записи ресурса.

Синхронизация на основе подачи сигналов

при этом один поток получает уведомления от другого потока (для возобновления работы заблокированного потока)

- AutoResetEvent
- ManualResetEvent
- ManualResetEventSlim
- CountdownEvent
- Barrier

AutoResetEvent

позволяет при получении сигнала переключить данный объект-событие из сигнального в несигнальное состояние.

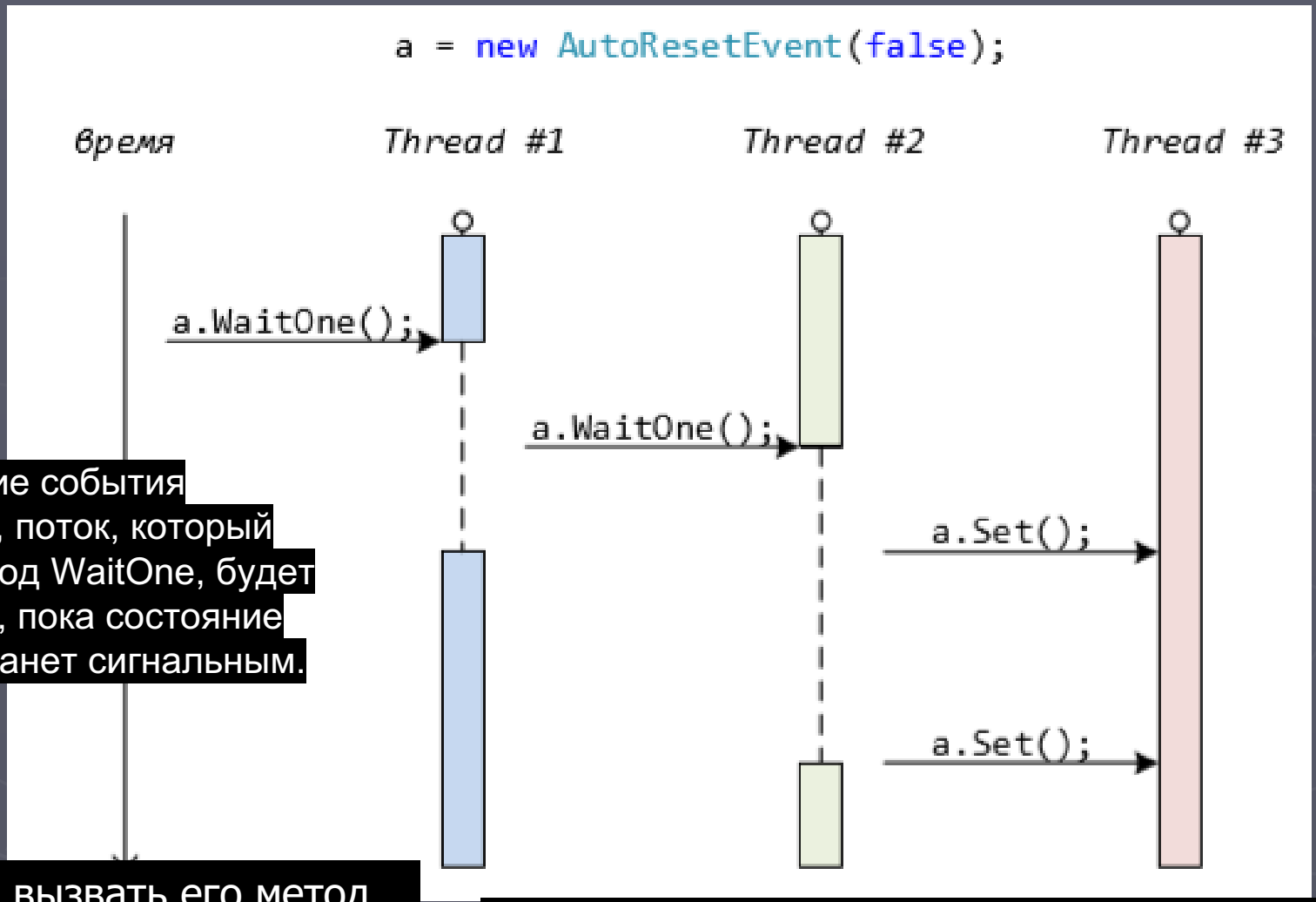
Reset(): задает несигнальное состояние объекта, блокируя потоки.

Set();: задает сигнальное состояние объекта, позволяя одному или нескольким ожидающим потокам продолжить работу.

WaitOne(): задает несигнальное состояние и блокирует текущий поток, пока текущий объект AutoResetEvent не получит сигнал.

```
AutoResetEvent myEvent = new AutoResetEvent(true);
// Передавая в конструктор значение true, мы тем самым указы
ваем, что создаваемый объект изначально будет в сигнальном
состоянии.
    ...
myEvent.WaitOne();
    ...
myEvent.Set();
```

AutoResetEvent



Если состояние события несигнальное, поток, который вызывает метод WaitOne, будет заблокирован, пока состояние события не станет сигнальным.

поток может вызвать его метод WaitOne(), чтобы остановиться и ждать сигнала. Для отправки сигнала применяется вызов метода Set().

ожидающие потоки освобождаются и запускаются последовательно, на манер очереди

```
static int x = 0;

// переменную будем использовать для синхронизаци
static AutoResetEvent myEvent = new AutoResetEvent(true);

// главный метод программы
static void Main(string[] args)
{
    // создаем потоки
    for (int i = 0; i < 5; i++)
    {
        Thread myThread = new Thread(Count);
        myThread.Name = "Поток " + i.ToString();
        myThread.Start();
    }
}

// поток
public static void Count()
{
    myEvent.WaitOne();
    x = 1;
    for (int i = 1; i < 9; i++)
    {
        Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, x);
        x++;
        Thread.Sleep(100);
    }
    myEvent.Set();
}
```

ожидаем сигнала

сигнализируем, что waitHandler в сигнальном состоянии

Barrier

- организует для нескольких потоков точку встречи во времени

```
class Program
    {
            private static readonly Barrier _barrier = new Barrier(3);
            public static void Main()
            {
                new Thread(Print).Start();
                new Thread(Print).Start();
                new Thread(Print).Start();
            // вывод: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
                 Console.ReadLine();
            }
            private static void Print() {
                for (var i = 0; i < 5; i++) {
                   Console.Write(i + " ");
                 _barrier.SignalAndWait();
            }
        }
```

количество участников

System.Threading.**Timer**

позволяет запускать определенные действия по истечению некоторого периода времени

Принимает метод, который должен в качестве параметра принимать объект типа object.

```
int num = 0;
// устанавливаем метод обратного вызова
TimerCallback tm = new TimerCallback(Count);
// 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 запускать немедленно после создания, так как в качестве значения используется 0
- интервал между вызовами метода Count

после запуска программы каждые две секунды будет срабатывать метод Count.

```
class Program
{
    static void Main(string[] args)
    {
        int num = 0;
        // устанавливаем метод обратного вызова
        TimerCallback tm = new TimerCallback(Count);
        // создаем таймер
        Timer timer = new Timer(tm, num, 0, 2000);

        Console.ReadLine();
    }
    public static void Count(object obj)
    {
        int x = (int)obj;
        for (int i = 1; i < 9; i++, x++)
        {
            Console.WriteLine($"{x * i}");
        }
    }
```

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;
            }
```

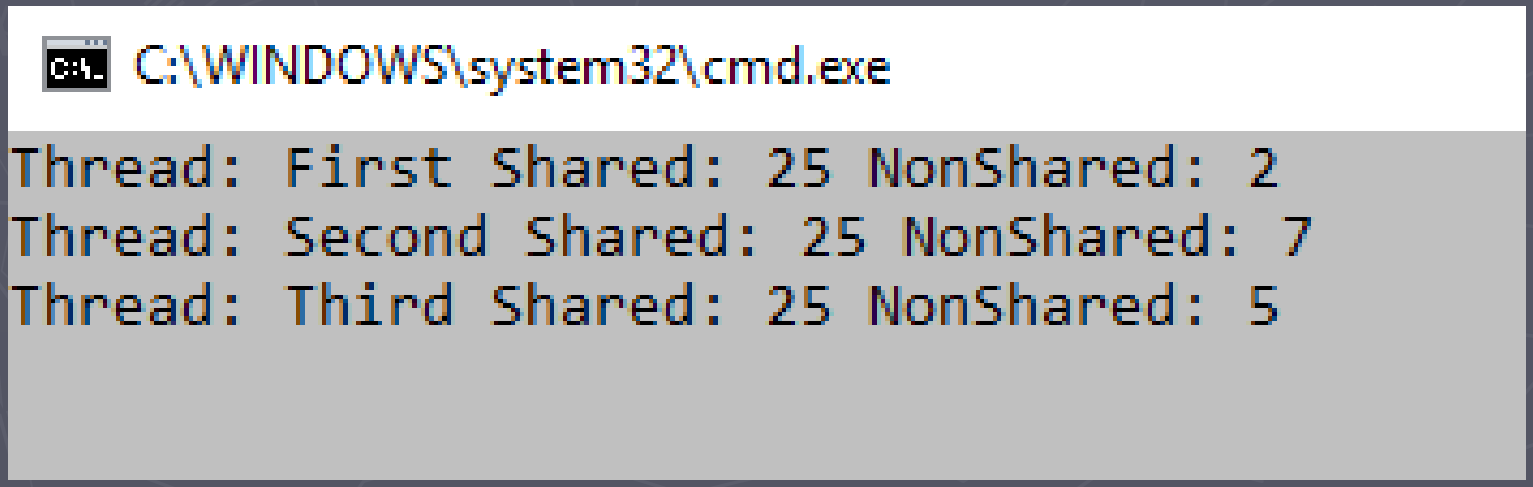
не рекомендуется делать инициализацию при объявлении, так как код инициализации выполнится только в одном потоке

ThreadLocal<T>

- Для создания неразделяемых статических полей.
- предоставляет локальное хранилище потока и для статических полей, и для полей экземпляра, и позволит Вам задать значения по умолчанию.

```
public class Slot
    {
        private static readonly Random rnd = new Random();
        private static int Shared = 25;
        private static ThreadLocal<int> NonShared =
               new ThreadLocal<int>(() => rnd.Next(1, 20));

        public static void PrintData()
        {
            Console.WriteLine($"Thread: {Thread.CurrentThread.Name} " +
                $"Shared: {Shared} NonShared: {NonShared.Value}");
        }
```

```
public class MainClass
        {
            public static void Main()
            {           // для тестирования запускаем три потока
                new Thread(Slot.PrintData) { Name = "First" }.Start();
                new Thread(Slot.PrintData) { Name = "Second" }.Start();
                new Thread(Slot.PrintData) { Name = "Third" }.Start();

                Console.ReadLine();
            }
        }
```



Потоки

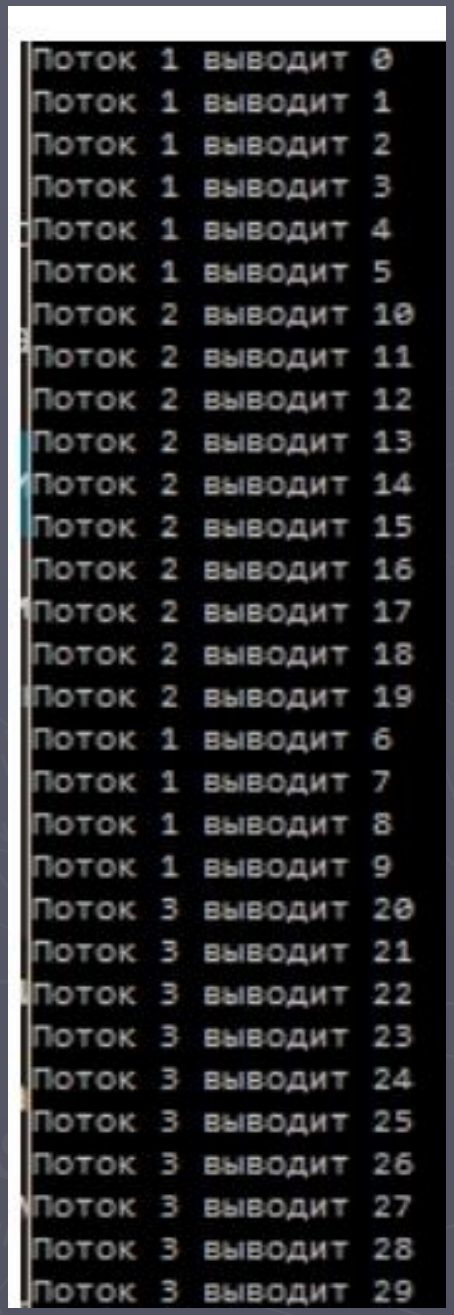
- Поток (Thread) – это низкоуровневый инструмент для организации параллельной работы
- Ограничения:
  - 1) отсутствует механизм продолжений (после завершения метода, работающего в потоке, в этом же потоке автоматически запускается другой заданный метод)
  - 2) Затруднено получение значения функции, выполняющейся в отдельном потоке
  - 3) создание множества потоков ведёт к повышенному расходу памяти и замедлению работы приложения

на что влияет приоритетность потоков

пример:

програма с тремя потоками, каждый из которых будет выводить в консоль цифры от 0 до 9, от 10 до 19 и от 20 до 29 соответственно.

Поставим перед собой задачу вывести в консоль все эти числа последовательно от 0 до 29.



у всех трёх потоков одинаковый приоритет,
процессору, по сути, будет всё равно, какой за каким потоки выводить

```
static void mythread1()
{
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine("Поток 1 выводит " + i);
    }
}

static void mythread2()
{

    for (int i = 10; i < 20; i++)
    {
        Console.WriteLine("Поток 2 выводит " + i);
    }
}

static void mythread3()
{
    for (int i = 20; i < 30; i++)
    {
        Console.WriteLine("Поток 3 выводит " + i);
    }
}

static void Main(string[] args)
{
    Thread thread1 = new Thread(mythread1);
    Thread thread2 = new Thread(mythread2);
    Thread thread3 = new Thread(mythread3);
    thread1.Priority = ThreadPriority.Highest;
    thread2.Priority = ThreadPriority.AboveNormal;
    thread3.Priority = ThreadPriority.Lowest;

    thread1.Start();
    thread2.Start();
    thread3.Start();

    Console.ReadLine();
}
```

```
Поток 3 выводит 0
Поток 3 выводит 1
Поток 3 выводит 2
Поток 3 выводит 3
Поток 3 выводит 4
Поток 3 выводит 5
Поток 3 выводит 6
Поток 3 выводит 7
Поток 3 выводит 8
Поток 3 выводит 9
Поток 2 выводит 10
Поток 2 выводит 11
Поток 2 выводит 12
Поток 2 выводит 13
Поток 2 выводит 14
Поток 2 выводит 15
Поток 2 выводит 16
Поток 2 выводит 17
Поток 2 выводит 18
Поток 2 выводит 19
Поток 1 выводит 20
Поток 1 выводит 21
Поток 1 выводит 22
Поток 1 выводит 23
Поток 1 выводит 24
Поток 1 выводит 25
Поток 1 выводит 26
Поток 1 выводит 27
Поток 1 выводит 28
Поток 1 выводит 29
```

```
thread1.Priority = ThreadPriority.Lowest;
thread2.Priority = ThreadPriority.BelowNormal;
thread3.Priority = ThreadPriority.Highest;
```

```
оток 3 выводит 20
оток 3 выводит 21
оток 3 выводит 22
оток 3 выводит 23
оток 3 выводит 24
оток 3 выводит 25
оток 3 выводит 26
оток 3 выводит 27
оток 3 выводит 28
оток 3 выводит 29
оток 2 выводит 10
оток 2 выводит 11
оток 2 выводит 12
оток 2 выводит 13
оток 2 выводит 14
оток 2 выводит 15
оток 2 выводит 16
оток 2 выводит 17
оток 2 выводит 18
оток 2 выводит 19
оток 1 выводит 0
оток 1 выводит 1
оток 1 выводит 2
оток 1 выводит 3
оток 1 выводит 4
оток 1 выводит 5
оток 1 выводит 6
оток 1 выводит 7
оток 1 выводит 8
оток 1 выводит 9
```

отличие фоновых потоков в C# от основных.

имеется два потока — thread1 и поток в из метода Main. Изначально потоки работают независимо друг от друга, и, пока не закончится выполняться один поток, второй поток нельзя будет закончить принудительно

```
        static void mythread1()
        {
            for (int i = 0; i < 1000000; i++)
            {
                Console.WriteLine("Поток 1 выводит " + i);
            }
        }

        static void Main(string[] args)
        {
            Thread thread1 = new Thread(mythread1);
            thread1.IsBackground = true;

            thread1.Start();

            Thread.Sleep(100);
        }
    }
```

Но сделали его фоновый

значит он будет полностью зависеть от потока в этом методе.

приостанавливаем приоритетный поток исключительно для того, чтобы успеть сделать скриншот вывода

```
Поток 1 выводит 161
Поток 1 выводит 162
Поток 1 выводит 163
Поток 1 выводит 164
Поток 1 выводит 165
Поток 1 выводит 166
Поток 1 выводит 167
Поток 1 выводит 168
Поток 1 выводит 169
Поток 1 выводит 170
Поток 1 выводит 171
Поток 1 выводит 172
Поток 1 выводит 173
Поток 1 выводит 174
Поток 1 выводит 175
Поток 1 выводит 176
Поток 1 выводит 177
Поток 1 выводит 178
Поток 1 выводит 179
Поток 1 выводит 180
Поток 1 выводит 181
Поток 1 выводит 182
Поток 1 выводит 183
Поток 1 выводит 184
```