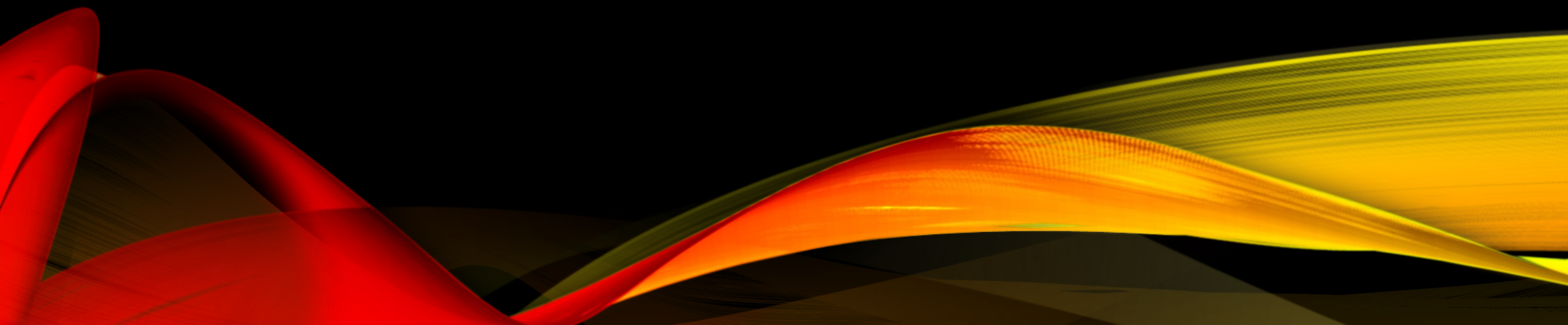
ЛЕКЦИЯ 17

- Модуль 3
- 15.03.2023
- Процессы и потоки

ЦЕЛИ ЛЕКЦИИ

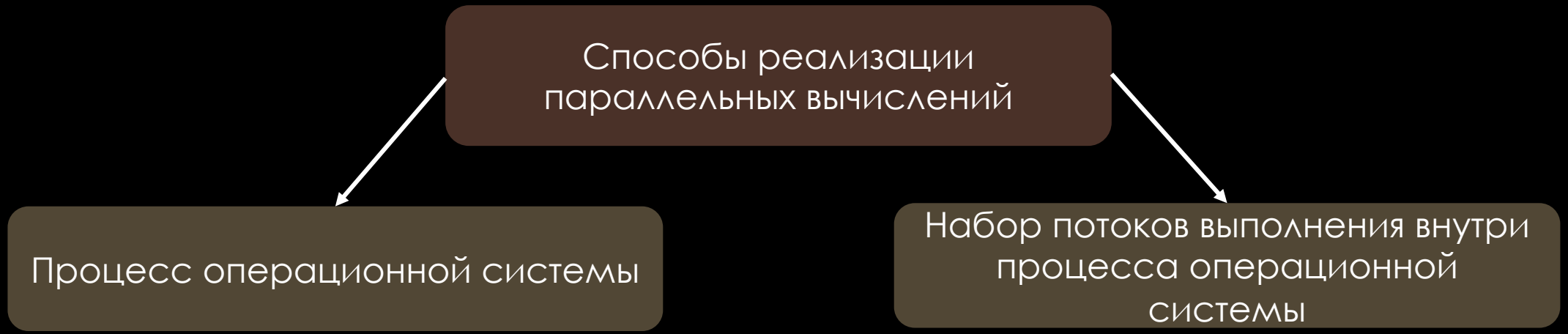
- Познакомится с основами параллельного программирования
- Разобраться с концепциями процесса и потока
- Познакомиться с потоками (Thread) в C#

ПРОЦЕССЫ И ПОТОКИ



ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ И СПОСОБЫ ИХ РЕАЛИЗАЦИИ

Параллельные вычисления – способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающий параллельно (одновременно)



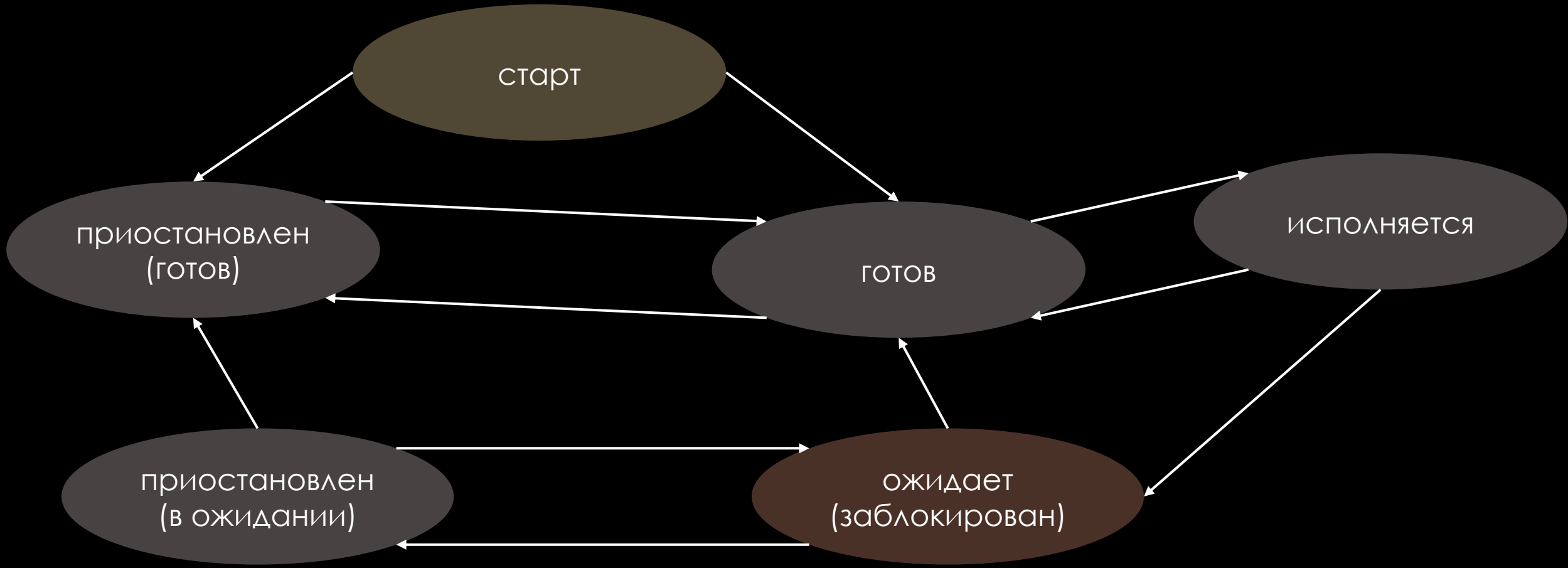
ПРОЦЕСС

Процесс – совокупность взаимосвязанных и взаимодействующих действий, преобразующих входные данные в выходные

- ISO 9000:2000 Definitions

Процесс – выполняющаяся программа и всё ее ресурсы: адресное пространство, глобальные переменные, регистры, стек, открытые файлы и т.д.

СТАТУСЫ ПРОЦЕССА ОС



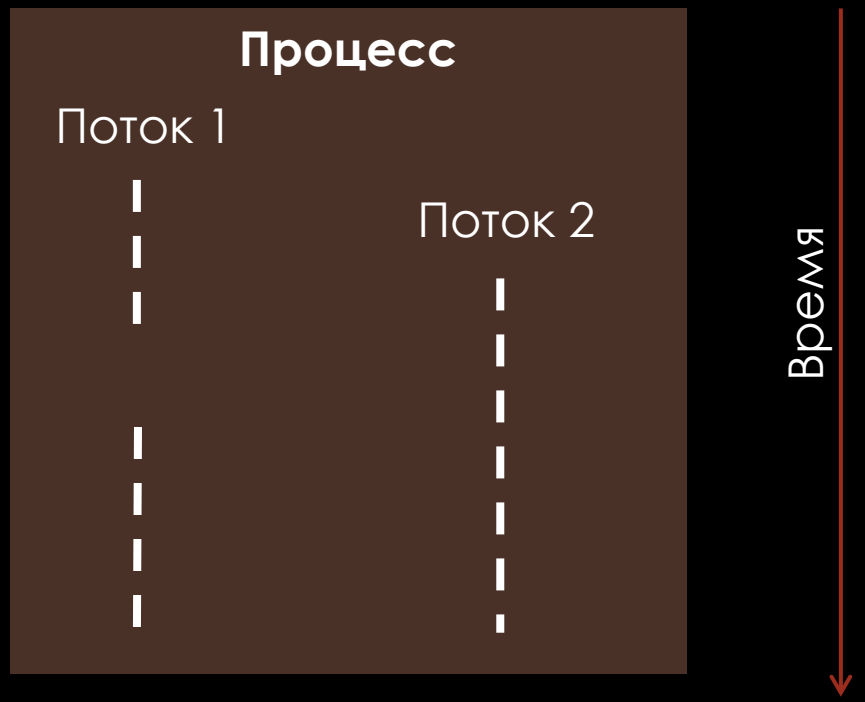
ВИДЫ ПАРАЛЛЕЛЬНОГО ВЗАИМОДЕЙСТВИЯ

- Взаимодействие через разделяемую память (C#)
  - Захват управления для координации потоков между собой (мьютексы, семафоры, мониторы)
- Взаимодействие с помощью передачи сообщений

ПОТОК ВЫПОЛНЕНИЯ

Поток (thread) или **поток выполнения** (thread of execution) – наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы

- Приоритет планирования
- Способы сохранения контекста потока

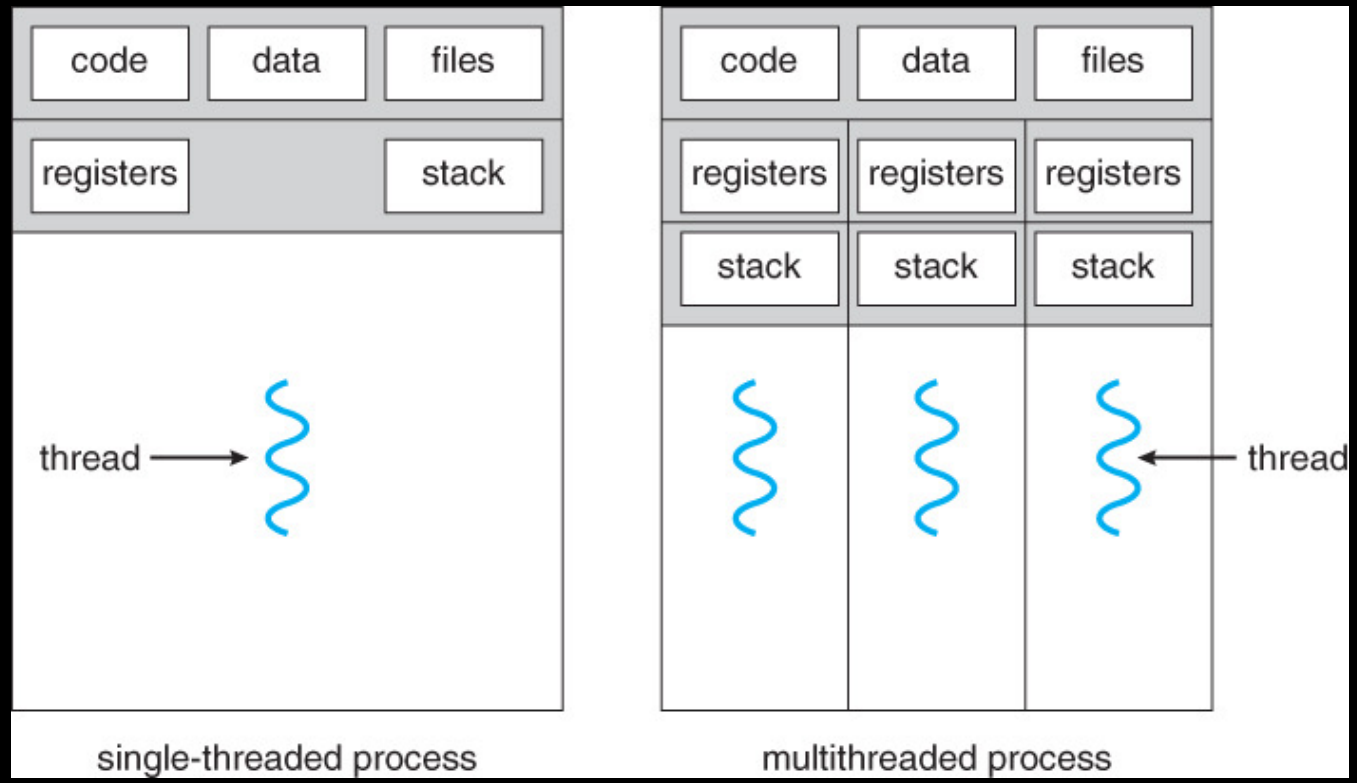


- По умолчанию программа на C# запускается в одном потоке – основной поток
- Програмно (то есть в коде) могут быть созданы дополнительные потоки для параллельного выполнения с основным потоком – рабочие потоки

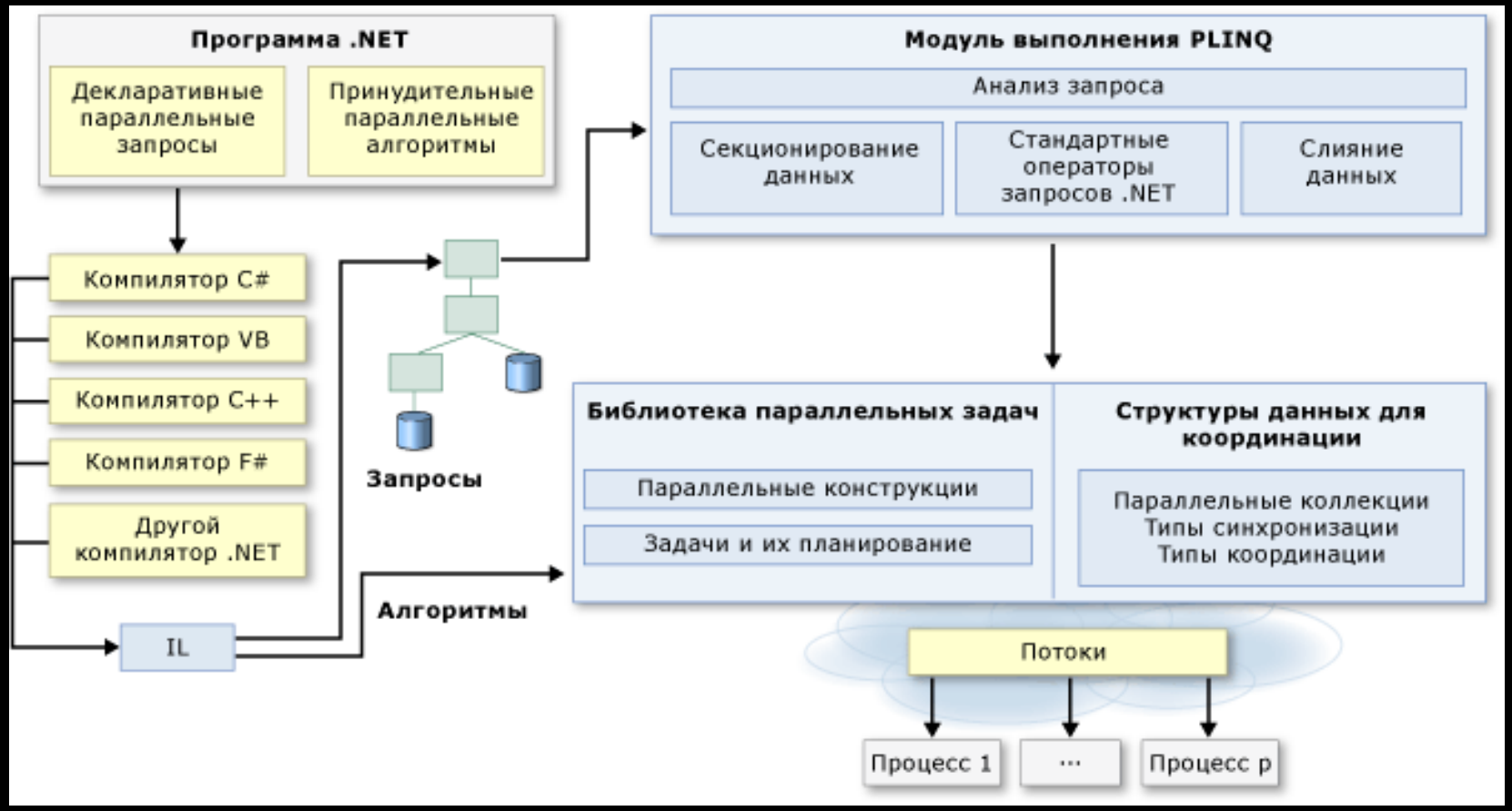
СИНХРОННОСТЬ И АСИНХРОННОСТЬ

Синхронное выполнение программы – выполнение в единственном потоке

Асинхронное выполнение – выполнение программы в нескольких потоках



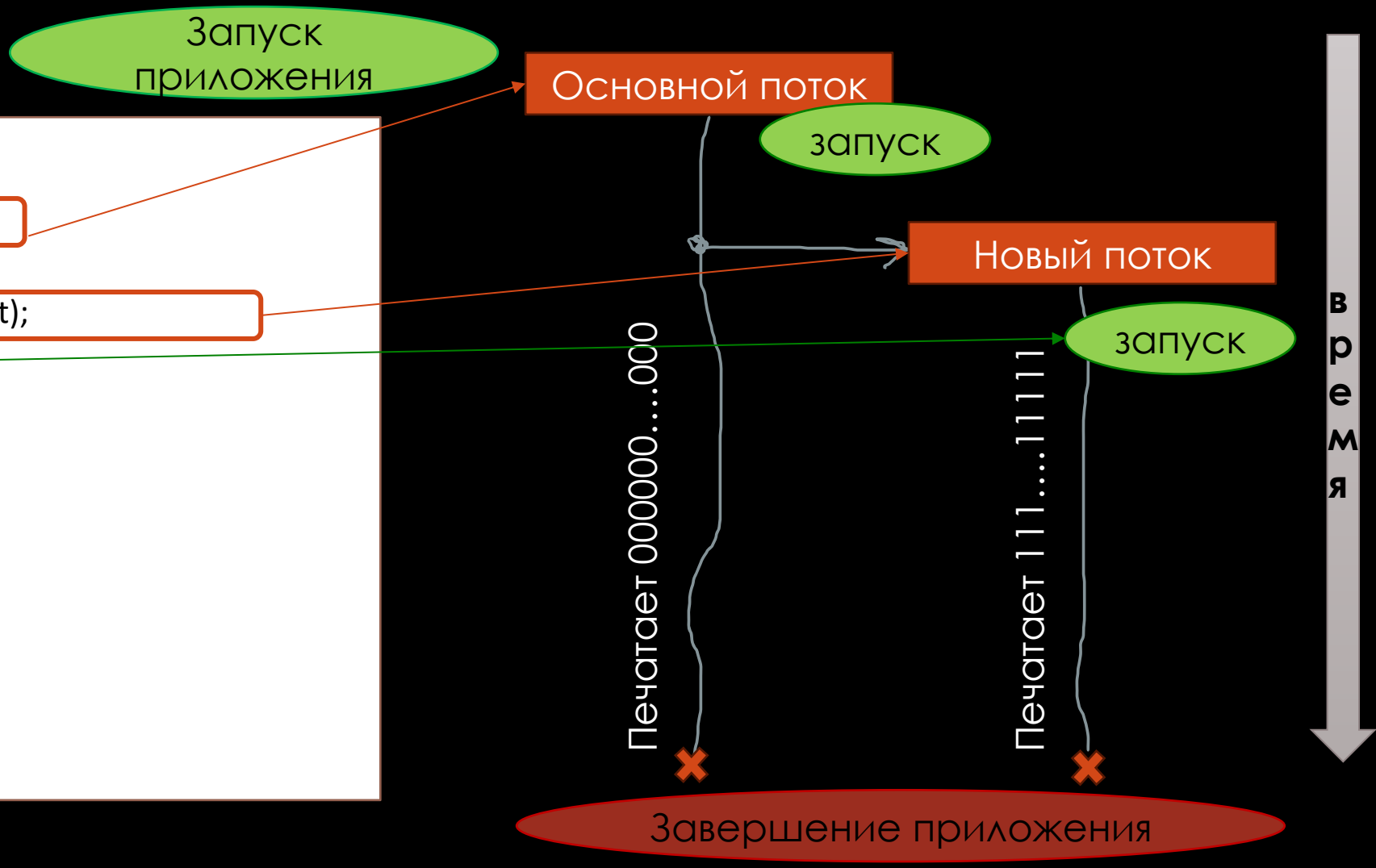
АРХИТЕКТУРА ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ .NET



ПРИМЕР СОЗДАНИЯ ПОТОКА

```
class Program
{
   public static void Main()
   {
      Thread funcThread = new Thread(Print);
      funcThread.Start();

      for (int i = 0; i < 50; i++)
         Console.Write(0);
   }
   static void Print()
   {
      for (int i = 0; i < 50; i++)
         Console.Write(1);
   }
}
```



ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ И ПОТОКИ

```
public static void Main()
{
   // Новый поток.
   new Thread (Print).Start();
        // Основной поток.
        Print();
}

static void Print()
{
   for (int i = 0; i < 5; i++)
      Console.Write(1);
}
```

Вывод будет разным. Во втором случае потоки «разделяют» общие данные объекта

```
public static void Main()
{
   A a = new A(); // Один объект.
   // Новый поток.
   new Thread(a.Print).Start();
   // Основной поток.
   a.Print();
}
class A
{
   char c = 'a'; // Общий ресурс для потоков.
   public void Print()
   {
      c++;
      for (int i = 0; i < 5; i++)
         Console.Write(c);
   }
}
```

ВЗАИМОДЕЙСТВИЕ ПОТОКОВ

Методы взаимодействия потоков (захваты управления):

- Взаимоисключения (мьютексы [mutually exclusive access])
- Семафоры
- Критические секции
- События

КРИТИЧЕСКИЕ СЕКЦИИ: ОБЪЕКТ-ЗАГЛУШКА

- `lock` – использование объекта-заглушки для обеспечения эксклюзивного доступа к критической секции кода в одном процессе

Шаблон использования объекта-заглушки

```
class ThreadSafe
{
   private readonly object _locker = new object();

   public void Method()
   {
      // Этот код может выполняться несколькими потоками.
      lock (_locker)
      {
         // Этот код может выполняться только одним потоком.
      }
      // Этот код может выполняться несколькими потоками.
   }
}
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ ОБЪЕКТА-ЗАГЛУШКИ

```
static char ch = 'a';
public static void Main()
{
   // Новый поток.
   new Thread(Print).Start();
   // Основной поток.
   Print();
}

static void Print()
{
   ch++;
   for (int i = 0; i < 5; i++)
      Console.Write(ch);
}
```

```
static readonly object _locker = new object();
static char ch = 'a';
public static void Main()
{
   // Новый поток.
   new Thread(Print).Start();
   // Основной поток.
   Print();
}
static void Print()
{
   lock (_locker)
   {
      ch++;
      for (int i = 0; i < 5; i++)
         Console.Write(ch);
   }

}
```

КРИТИЧЕСКИЕ СЕКЦИИ: МОНИТОР

- `Monitor` – использование объекта-заглушки для обеспечения эксклюзивного доступа к критической секции кода в одном процессе

Шаблон использования класса `Monitor`

Если в критической секции произойдёт исключение, то `finally` гарантирует снятие блокировки

```
private static object _locker = new object();
private static  int _counter = 0;

public static void CounterInc()
{
   Monitor.Enter(_locker); // Блокировка захвачена.
   try
   {
     _counter++;
   }
   finally
   {
     Monitor.Exit(_locker); // Блокировка освобождена.
   }
}
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ КЛАССА MONITOR

```
static readonly object _locker = new object();
private static int _count = 0;
static char ch = 'a';
public static void Main()
{
   // Новый поток.
   new Thread(Print).Start();
   // Основной поток.
   Print();
}
```

```
static void Print()
{
   Monitor.Enter(_locker);
   try
   {
      _count++;
      ch++;
      for (int i = 0; i < 5; i++)
         Console.Write(ch);
   }
   finally
   {
      Monitor.Exit(_locker);
   }
}
```

МЬЮТЕКС

Мьютекс - это особый вид двоичного семафора, который используется для обеспечения механизма блокировки

Преимущества:
- Не возникает условий гонки, поскольку в критической секции в один момент времени находится только один процесс
- Обеспечивается целостность данных
- Это простой механизм блокировки, который активируется процессом перед входом в критическую секцию и освобождается при выходе из нее

Недостатки:
- Если после входа в критическую секцию поток заснет или будет вытеснен высокоприоритетным процессом, ни один другой поток не сможет войти в критическую секцию
- Когда предыдущий поток покидает критическую секцию, в нее могут войти только другие процессы
- Реализация мьютекса может привести к занятому ожиданию, что приводит к трате процессорного времени

ВЗАИМОИСКЛЮЧЕНИЯ: МЬЮТЕКС

- `Mutex` – частный случай семафора, применяющийся для ограничения доступа к ресурсу, при этом освободить ресурс может только занявший его поток

Шаблон использования класса `Mutext`

```
private static Mutex mtx = new Mutex();

public static void Method()
{
   mtx.WaitOne();
   try
   {
      // Критическая секция, выполняющаяся одним потоком.
   }
   finally
   {
      mtx.ReleaseMutex(); // Блокировка освобождена.
   }
}
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ МЬЮТЕКСА

Создали один мьютекс

```
private static Mutex mtx = new Mutex();

private static int _count = 0;
static char ch = 'a';
public static void Main()
{
   // Новый поток.
   new Thread(Print).Start();
   // Основной поток.
   Print();
}
```

Выполнение потока приостановлено до получение мьютекса

```
static void Print()
{
   mtx.WaitOne();
   try
   {
      _count++;
      ch++;
      for (int i = 0; i < 5; i++)
         Console.Write(ch);
   }
   finally
   {
      mtx.ReleaseMutex();
   }
}
```

СЕМАФОР

Семафор - это обычная целочисленная неотрицательная переменная, с которой можно работать только с помощью двух специальных неделимых (атомарных) операций Ожидание (Wait, P) и Сингал (Signal, V)

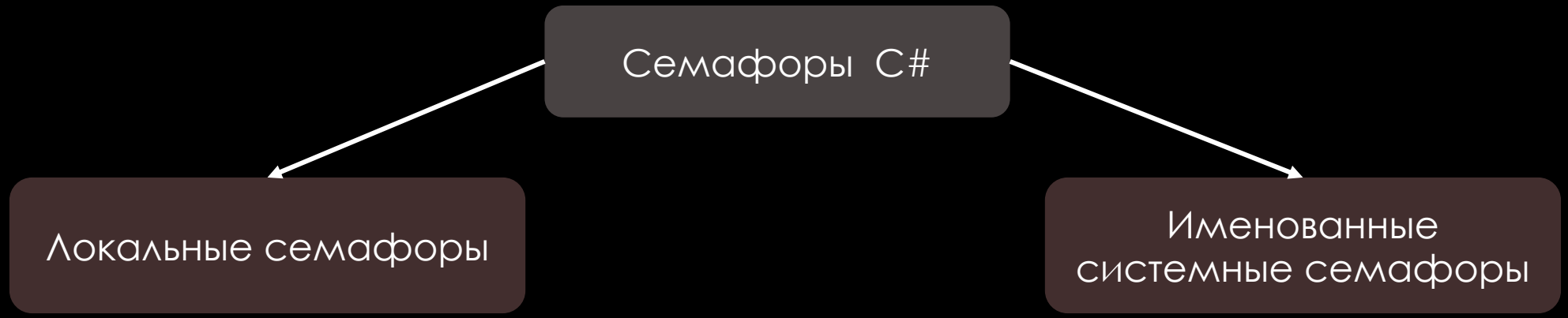
Преимущества:

- Несколько потоков могут получить доступ к критической секции одновременно
- Одновременно к критической секции будет обращаться только один процесс, однако допускается работа нескольких потоков
- Семафоры являются машинно-независимыми, поэтому их следует запускать через микроядро
- Гибкое управление ресурсами

Недостатки:

- Содержит инверсию приоритета
- Операции семафора (ожидание, сигнал) должны быть реализованы корректно, чтобы избежать дедлоков, что приводит к потере модульности, поэтому семафоры нельзя использовать в крупномасштабных системах
- Семафор чувствителен к ошибкам при программировании, что может провоцировать дедлоки и нарушению свойства взаимного исключения
- ОС должна отслеживать все вызовы операций ожидания и сигналов

СЕМАФОР C#


Семафоры C#
Локальные семафоры
Именованные системные семафоры

СЕМАФОР

- `Semaphore` – служит для ограничения количества потоков, получающих доступ к ресурсу. Подходит для контроля пулов ресурсов, позволяет получить именованный системный семафор

Шаблон использования класса `Semaphore`

```
class Program
{
   private static Semaphore smph = new Semaphore(2,2);
   public static void Method()
   {
      smph.WaitOne();
      try
      {
         // Критическая секция, выполняющаяся одним потоком.
      }
      finally
      {
         smph.Release(); // Блокировка освобождена.
      }
   }
}
```

УПРОЩЁННЫЙ СЕМАФОР

- `SemaphoreSlim` – служит для ограничения количества потоков, получающих доступ к ресурсу. Рекомендован к использованию для синхронизации на уровне приложения

Шаблон использования класса `SemaphoreSlim`

```
class Program
{
   private static SemaphoreSlim smph = new SemaphoreSlim(2,2);
   public static void Method()
   {
      smph.Wait();
      try
      {
         // Секция, ограниченная для 2 потоков.
      }
      finally
      {
         smph.Release(); // Блокировка освобождена.
      }
   }
}
```

СОБЫТИЯ

- `AutoResetEvent, ManualResetEvent` – служит для управления потоками с помощью сигналов и событий, т.е. позволяет составить систему оповещений между потоками

Шаблон использования класса `AutoResetEvent`

```
class Program
{
   private static AutoResetEvent arv = new(false);
   public static void Method()
   {
      // В этом потоке оповещаем о событии
      arv.Set();
   }
   public static void Method_1()
   {
      // В этом потоке ожидаем события
      arv.WaitOne();
   }
}
```

ПУЛ ПОТОКОВ

Пул потоков – специальный набор рабочих потоков приложения, управляемых системой

Для чего нужны пулы потоков?

- Абстракция над созданием потока (поток создаётся без явного указания программистом)
- В уже созданных потоках исполняются разные задачи (это способствует снижению затрат на создание потока и балансировки загрузки процессора)
- Управление пропускной способностью (ограничения или снятие ограничений на использование ядер ЦП)

ОСОБЕННОСТИ РЕАЛИЗАЦИИ ПУЛА ПОТОКОВ

- Статический класс `System.Threading.ThreadPool`
- Реализация на основе очереди делегатов
  - Используется коллекция `ConcurrentQueue<>`
- Количество потоков в пуле лимитировано только объёмом доступной памяти
- Необработанные исключения в потока пула приводят к завершению процессов, но есть исключения, которые используются для управления потоком выполнения программы:
  - `ThreadAbortException` (только .NET Framework приложения)
  - `AppDomainUnloadedException` из-за выгрузки домена приложения, в котором выполняется поток
  - CLR или ведущий процесс прерывает выполнение потока путем создания внутреннего исключения
  - https://learn.microsoft.com/ru-ru/dotnet/standard/threading/exceptions-in-managed-threads

СОЗДАНИЕ И ЗАПУСК ПОТОКА THREAD

```
static void Print()  {
    for (int k = 0; k < 5; k++) {
        Console.WriteLine("\t\t\tПоток_2");
    }
}
```

```
/// <summary>
/// запустить несколько раз, чтобы увидеть отличия
/// </summary>
public static void Main() {
    Thread tr = new Thread(Print);
    tr.Start();
    for (int k = 0; k < 5; k++) {
        Console.WriteLine("Поток_1");
        Thread.Sleep(0);    // Внимание, МОЖЕТ передаться управление
    }
    Console.WriteLine("Нажмите любую клавишу!");
    Console.ReadKey(true); // можно убрать
}
```

Основные конструкторы :
Thread(ThreadStart start)
Thread(ParameterizedThreadStart start)

Типы параметров:
delegate void ThreadStart()
delegate void ParameterizedThreadStart(Object obj)

ЗАДЕРЖКИ В ПОТОКАХ (МЕТОД SLEEP)

```
static void Print2() {
    for (int k = 0; k < 5; k++)  {
        Thread.Sleep(100);
        Console.WriteLine("\t\t\tПоток_2");
    }
}
```

```
public static void Main()  {
    Thread tr = new Thread(Print2);
    tr.Start();
    for (int k = 0; k < 5; k++)  {
        Thread.Sleep(100);
        Console.WriteLine("Поток_1");
    }
    Console.WriteLine("Нажмите любую клавишу!");
    Console.ReadKey(true); // можно убрать
}
```

СИНХРОНИЗАЦИЯ ПОТОКОВ И ИСПОЛЬЗОВАНИЕ JOIN

```
static void Print3()
{
    for (int k = 0; k < 5; k++)
        Console.WriteLine("\t\t\tПоток_2");
}
```

```
/// <summary>
/// Синхронизация потоков
/// </summary>
static void Main()
{
    Thread tr = new Thread(Print3);
    tr.Start();
    tr.Join();  // Ожидание окончания потока tr
    for (int k = 0; k < 5; k++)
        Console.WriteLine("Поток_1");
    Console.WriteLine("Нажмите любую клавишу!");
    Console.ReadKey(true);
}
```

НЕКОТОРЫЕ ЧЛЕНЫ КЛАССА THREAD

`Thread.CurrentThread` – статическое свойство (тип Thread)

`IsAlive` – свойство (тип bool, поток запущен и не завешен)
`Name` – свойство (тип string, имя потока, write-once)

`Start()` – запуск потока (перевод в состояние *running*)
`Start(аргумент)` – запуск потока с передачей аргумента

`Join()` – синхронизация (блокирует выполнение вызывающего потока до завершения потока, представленного экземпляром)
`Join(аргумент)` – синхронизация с таймаутом

`Sleep(time)` – приостановить исполнение потока на *time* мс
`Thread.Sleep(0)` – возможность переключиться на исполнение другого потока

ИМЕНОВАНИЕ ПОТОКОВ И ПЕРЕДАЧА ДАННЫХ В ПОТОК

```
static void Print4(object par) {
    Console.WriteLine("\t\t" + Thread.CurrentThread.Name + " запущен из " + (string)par);
}

public static void Main() {
    Thread.CurrentThread.Name = "Main";
    // Thread.CurrentThread.Name = "Main2"; // System.InvalidOperationException: Это
свойство уже назначено и не может изменяться
    Thread tr = new Thread(Print4);
    tr.Name = "Вторичный";
    tr.Start("Main"); // передаем в поток данные
    Console.WriteLine("Поток_1 = " + Thread.CurrentThread.Name);
    Console.WriteLine("Нажмите любую клавишу!");
    Console.ReadKey(true);
}
```

ПЕРЕДАЧА РЕЗУЛЬТАТОВ РАБОТЫ ИЗ ПОТОКА

```
static void Fib(object par) {
    int[] ar = (int[])par;
    ar[0] = ar[1] = 1;
    for (int j = 2; j < ar.Length; j++)
        ar[j] = ar[j - 1] + ar[j - 2];
}
```

```
public static void Main05() {
    Thread tr = new Thread(Fib);
    int[] row = new int[8];
    tr.Start(row);
    tr.Join();
    foreach (int e in row)
        Console.Write(e + "  ");
    Console.WriteLine("\r\nНажмите любую клавишу!");
    Console.ReadKey(true);
}
```

ПЕРЕДАЧА ДАННЫХ МЕЖДУ ПОТОКАМИ

```
static void Fib6(int n, out int[] par) {
    par = new int[n];
    par[0] = par[1] = 1;
    for (int j = 2; j < n; j++)
        par[j] = par[j - 1] + par[j - 2];
}

public static void Main06() {
    int[] row = null;
    Thread tr = new Thread(() => Fib6(6, out row));
    tr.Start();
    tr.Join();
    foreach (int e in row)
        Console.Write(e + "  ");
    Console.WriteLine("\r\nНажмите любую клавишу!");
    Console.ReadKey(true);
}
```

ЛОКАЛЬНАЯ ПЕРЕМЕННАЯ ПЕРЕДАЕТСЯ В ДВА ПОТОКА

```
static void Print7(string mes) {
    Console.WriteLine(mes);
}

public static void Main07() {
    string line = "Tr_1";
    Thread tr1 = new Thread(() => { Print7(line); });
    tr1.Start();
    // tr1.Join();
    line = "Tr_2";
    Thread tr2 = new Thread(() => { Print7(line); });
    tr2.Start();
    Console.WriteLine("\r\nНажмите любую клавишу!");
    Console.ReadKey(true);
}
```

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ МЕТОДА ПОТОКА

```
static void Factorial()
{
    int r = 1;
    for (int i = 0; i < 5; i++)
    r *= i + 1;
    Console.WriteLine(r);
}

public static void Main08()
{
    Thread tr = new Thread(Factorial);
    tr.Start();
    Factorial();
    Console.WriteLine("\r\nНажмите любую клавишу!");
    Console.ReadKey(true);
}
```

ССЫЛКИ

- https://alexeykalina.github.io/technologies/concurrency.html
- https://habr.com/ru/articles/784134/
- https://learn.microsoft.com/en-us/dotnet/standard/threading/overview-of-synchronization-primitives
- https://learn.microsoft.com/ru-ru/dotnet/standard/threading/mutexes
- https://professorweb.ru/my/csharp/thread_and_files/1/1_1.php
- https://dic.academic.ru/dic.nsf/ruwiki/1085882
- https://dic.academic.ru/dic.nsf/ruwiki/17043
- https://habr.com/ru/articles/793634/
- https://www.tstu.ru/book/elib3/mm/2016/evdokimov/site/page30.30.html
- https://learn.microsoft.com/ru-ru/dotnet/standard/threading/threads-and-threading
- https://learn.microsoft.com/en-us/windows/win32/procthread/processes-and-threads
- https://www.geeksforgeeks.org/mutex-vs-semaphore/
- https://habr.com/ru/articles/654101/
- https://learn.microsoft.com/ru-ru/dotnet/standard/threading/the-managed-thread-pool
- https://habr.com/ru/articles/654111/