1. Что такое процесс, домен, поток? Как они связаны между собой?

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

* существуют внутри процессов
* Процесс выделяет для приложения изолированное адресное пространство и поддерживает один или несколько потоков выполнения.
* **Поток** - используемый внутри процесса путь выполнения

Процесс — это **запущенная программа**. Это единица работы в операционной системе, которая управляет выполнением программы. Когда вы запускаете программу на своём компьютере (например, текстовый редактор, браузер или игра), операционная система создаёт процесс для этой программы.

1. Как получить информацию о процессах?

Process.GetProcesses() возвращает все процессы, запущенные на компьютере.

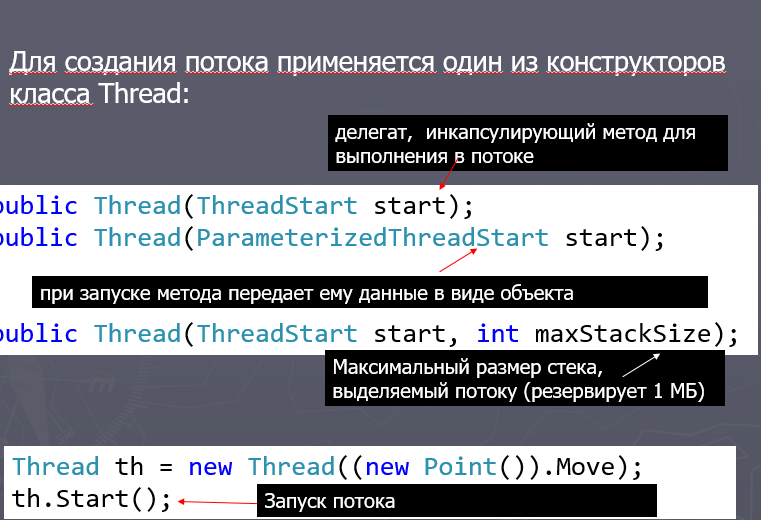
using System.Diagnostics;

1. Как создать и настроить домен?

**Класс System.AppDomain**

1. Как создать и настроить поток?

В C# для создания потока используется класс Thread из пространства имен System.Threading



1. В каких состояниях может быть поток?

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

1. Какие методы управления потоками вы знаете, для чего и как их использовать?

**GetDomain - с**татический,  возвращает ссылку домен приложения

**GetDomainId** - **с**татический, возвращает id домена приложения, в котором выполняется текущий поток

**Sleep – с**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

1. Какие приоритеты потока вы знаете?

Класс Thread предоставляет перечисление ThreadPriority с уровнями приоритета:

1. **Lowest** (Самый низкий).
2. **BelowNormal** (Ниже нормального).
3. **Normal** (Нормальный, значение по умолчанию).
4. **AboveNormal** (Выше нормального).
5. **Highest** (Самый высокий).
6. Что такое пул потоков и для чего он используется?

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

очереди методов

рабочих потоков.

1. Что такое критическая секция? Поясните использование.

**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.

1. Что такое мьютекс? Поясните использование

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).

1. Что такое семафор? Поясните использование

объект синхронизации, позволяющий войти в заданный участок кода не более чем N потокам (N – ёмкость семафора)

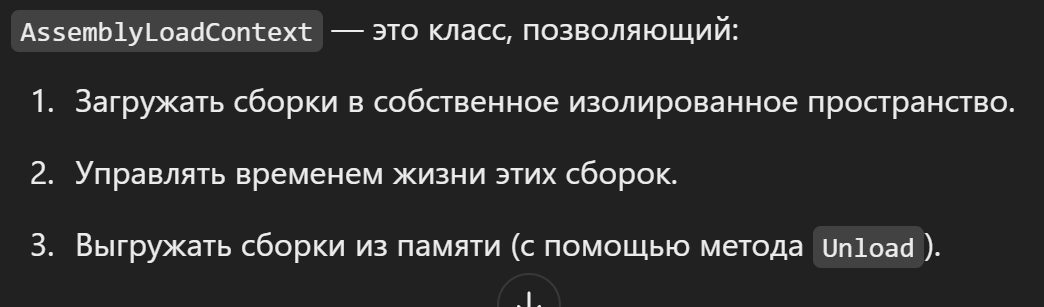
получение и снятие блокировки в случае семафора может выполняться из разных потоках

1. Что такое неблокирующие средства синхронизации?

**Неблокирующие средства синхронизации** — это механизмы, которые позволяют потокам синхронизировать свои действия, не блокируя выполнение других потоков.

13. Для чего можно использовать класс Timer?

Создает событие после заданного интервала с возможностью создания повторяющихся событий.



Создание нового **контекста сборки** (AssemblyLoadContext) и создание нового **домена приложения** (AppDomain) выполняют схожие задачи — изоляция кода и сборок,

Когда в вашем коде несколько потоков пытаются выполнить одни и те же операции с общими данными или ресурсами, например, с файлом, консолью или какой-то переменной, может возникнуть ситуация, когда один поток начнет изменять данные, а другой — одновременно. Чтобы избежать таких ошибок, используется механизм **синхронизации**, и в данном случае — это lock.  lockObject — это объект, который служит в качестве **замка** для кода, помещенного в блок lock.

 Когда поток входит в блок lock, он "запирает" этот объект, и другие потоки, пытающиеся войти в блок lock с этим же объектом, будут **ждать своей очереди**, пока первый поток не закончит работу и не "откроет замок".

 Таким образом, **только один поток может работать с защищенным кодом в этот момент времени**, что предотвращает возможные ошибки и конфликты.

Генерирует событие через заданный интервал времени с возможностью генерации повторяющихся событий.

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.IO;

using System.Linq;

using System.Reflection;

using System.Runtime.Loader;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

namespace laba14

{

public static class ProcessManager

{

public static void ListProcesses()

{

Process[] processes = Process.GetProcesses();//возвращает массив объектов Process, где каждый объект представляет один запущенный процесс. Получить информацию о обо всех процессах системы

using (StreamWriter file = new StreamWriter("C:\\Users\\User\\Documents\\2курс1сем\\лабыООП\\lab14\\lab14\\processes.txt"))//Открывается файл для записи с помощью класса StreamWriter. Блок using гарантирует, что файл будет закрыт после завершения записи, даже если возникнет ошибка.

{

foreach (Process process in processes)//Для каждого процесса (объекта Process) в массиве processes:

{

try

{

file.WriteLine($"ID: {process.Id}, Имя: {process.ProcessName}, Приоритет: {process.BasePriority}, Время начала: {process.StartTime}, Сколько времени использовался: {process.TotalProcessorTime}");

Console.WriteLine($"ID: {process.Id}, Имя: {process.ProcessName}, Приоритет: {process.BasePriority}, Время начала: {process.StartTime} Сколько времени использовался: {process.TotalProcessorTime}");

}

catch (Exception ex)

{

file.WriteLine($"Процесс {process.Id} ({process.ProcessName}) вызвал ошибку: {ex.Message}");

Console.WriteLine($"Не удалось получить подробную информацию о процессе {process.ProcessName}: {ex.Message}");

}

}

}

}

}

public static class AppDomainManager

{

public static void ManipulateAppDomain()

{

Console.WriteLine($"Имя текущего домена: {AppDomain.CurrentDomain.FriendlyName}"); //Получает понятное имя этого домена приложения.

Console.WriteLine("Загруженные сборки в текущий домен:");

foreach (Assembly asm in AppDomain.CurrentDomain.GetAssemblies()) //GetAssemblies(): Возвращает список всех сборок, загруженных в текущий домен.

{

Console.WriteLine($"- {asm.FullName}");

}

// Создаём новый контекст загрузки сборок. Создание нового контекста загрузки сборки означает создание отдельного пространства, в котором сборки могут быть загружены и изолированы от других сборок.

Console.WriteLine("\nСоздаём новый контекст загрузки сборок...");

var context = new AssemblyLoadContext("NewAssemblyLoadContext", isCollectible: true);//isCollectible: true Это означает, что все сборки, загруженные в этот контекст, и сам контекст могут быть выгружены из памяти, освобождая ресурсы.

try

{

// Загружаем сборку System.Text.Json в новый контекст

Console.WriteLine("\nЗагружаем сборку System.Text.Json...");

//Метод LoadFromAssemblyName используется для загрузки сборки в указанный контекст загрузки.

var assembly = context.LoadFromAssemblyName(new AssemblyName("System.Text.Json"));//Мы создаём новый объект AssemblyName, передавая строку "System.Text.Json", которая указывает на имя сборки, которую мы хотим загрузить.

Console.WriteLine($"Сборка {assembly.FullName} успешно загружена в новый контекст.");

//assembly — это переменная, которая будет хранить результат работы метода LoadFromAssemblyName. Этот результат — это объект типа Assembly, который представляет загруженную сборку в памяти

}

catch (Exception ex)

{

Console.WriteLine($"Ошибка при загрузке сборки: {ex.Message}");

}

finally

{

// Выгружаем контекст загрузки

Console.WriteLine("\nВыгружаем контекст загрузки...");//Выгрузить сборку — это процесс освобождения памяти и ресурсов, связанных с загруженной сборкой, после того как она больше не требуется или не используется

context.Unload();

Console.WriteLine("Контекст загрузки успешно выгружен.");

}

}

}

public static class PrimeCalculator

{

private static Thread primeThread; //это объект, представляющий поток выполнения, который используется для вычисления простых чисел.Поток создается, но еще не запущен.

private static bool isPaused = false; // это булевая переменная, которая управляет состоянием паузы для потока. Если true, поток приостановлен.

public static void CalculateInThread()

{

Console.Write("Введите n: ");

int n = int.Parse(Console.ReadLine());//Console.ReadLine() — это метод, который читает строку, введенную пользователем в консоль.преобразует введенную строку в целое число n.

primeThread = new Thread(() => CalculatePrimes(n));//создается новый поток, который будет выполнять задачу, определенную в переданной ему функции.В данном случае, это анонимная функция () => CalculatePrimes(n), которая будет вызывать метод CalculatePrimes(n) с параметром n,

primeThread.Start(); //запускает поток, который мы только что создали. Когда этот метод вызывается, поток начинает выполнять код, переданный в нем. В данном случае, это вызов метода CalculatePrimes(n).

//Поток теперь будет вычислять все простые числа от 2 до n в фоновом режиме, параллельно с выполнением основного потока программы.

Thread.Sleep(3000);

PauseThread();

Thread.Sleep(2000);//то приостановка текущего потока (основного потока) на 2000 миллисекунд (или 2 секунды).

ResumeThread();

}

private static void CalculatePrimes(int n) //Этот метод выполняет вычисления простых чисел от 2 до заданного числа n.

{

// Открытие файла для записи простых чисел

using (StreamWriter writer = new StreamWriter("C:\\Users\\User\\Documents\\2курс1сем\\лабыООП\\lab14\\lab14\\primes.txt"))

{

for (int i = 2; i <= n; i++)

{

if (IsPrime(i))

{

writer.WriteLine(i);

Console.WriteLine(i);

}

}

}

}

private static bool IsPrime(int number)

{

for (int i = 2; i <= Math.Sqrt(number); i++)

{

if (number % i == 0) return false;

}

return true;

}

private static void PauseThread()

{

isPaused = true;

Console.WriteLine("Поток приостановлен.");

}

private static void ResumeThread()

{

isPaused = false;

Console.WriteLine("Поток возобновлен.");

}

}

public static class EvenOddThreads

{

private static object lockObject = new object(); //это объект, который используется для синхронизации потоков и предотвращения их конфликтов при одновременном доступе к общим ресурсам.

public static void RunThreads()

{

Console.Write("Введите n: ");

int n = int.Parse(Console.ReadLine());

//Создаются два потока: evenThread и oddThread.

// Поток evenThread будет заниматься печатью четных чисел, а поток oddThread — нечетных.

Thread evenThread = new Thread(() => PrintEvenNumbers(n));

Thread oddThread = new Thread(() => PrintOddNumbers(n));

evenThread.Priority = ThreadPriority.Highest;//Потоку evenThread присваивается самый высокий приоритет.CLR считывает и анализирует значение приоритета и на их основании выделяет данному потоку то или иное количество времени.

oddThread.Priority = ThreadPriority.Lowest;

oddThread.Start();

oddThread.Join();//// Ожидаем завершения потока с четными числами. он гарантирует, что один поток завершится перед тем, как начнется другой.

evenThread.Start();

evenThread.Join();

//Запускаем потоки

//evenThread.Start();

//oddThread.Start();

//Ждем завершения работы потоков

//evenThread.Join();

//oddThread.Join();

}

private static void PrintEvenNumbers(int n)

{

using (StreamWriter writer = new StreamWriter("C:\\Users\\User\\Documents\\2курс1сем\\лабыООП\\lab14\\lab14\\even.txt"))

{

for (int i = 2; i <= n; i += 2)

{

lock (lockObject)

{

writer.WriteLine(i);

Console.WriteLine(i);

}

Thread.Sleep(1000);

}

}

}

private static void PrintOddNumbers(int n)

{

using (StreamWriter writer = new StreamWriter("C:\\Users\\User\\Documents\\2курс1сем\\лабыООП\\lab14\\lab14\\odd.txt"))

{

for (int i = 1; i <= n; i += 2)

{

lock (lockObject)

{

writer.WriteLine(i);

Console.WriteLine(i);

}

Thread.Sleep(100);

}

}

}

}

public static class RepeatingTask

{

private static Timer timer;

private static int countdownValue;//хранит текущее значение обратного отсчета в секундах.

public static void StartCountdown(int seconds)

{

countdownValue = seconds; // сохраняем значение переданного аргумента seconds в переменную countdownValue, чтобы начать отсчет с этого значения.

timer = new Timer(Countdown, null, 0, 1000);

//Создаем новый объект Timer. Этот объект будет вызывать метод Countdown каждую секунду (1000 миллисекунд).

// Countdown — это метод, который будет выполняться при каждом срабатывании таймера.

//Параметр null означает, что мы не передаем никаких дополнительных данных в метод Countdown.

//0 — это начальная задержка в миллисекундах, то есть таймер начнется сразу.

//1000 — это интервал в миллисекундах, через который таймер будет повторно вызывать метод Countdown.

Console.WriteLine($"Обратный отсчет начался с {countdownValue} секунд. Нажмите Enter, чтобы остановить.");

Console.ReadLine();

timer.Dispose(); //Освобождает все ресурсы, используемые компонентом

}

private static void Countdown(object state)

{

if (countdownValue > 0)

{

Console.WriteLine(countdownValue);

countdownValue--;

}

else

{

Console.WriteLine("Обратный отсчет завершен!");

timer.Dispose();

}

}

}

class Programm

{

static void Main(string[] args)

{

Console.WriteLine("Выберите вариант:");

Console.WriteLine("1. Список всех запущенных процессов.");

Console.WriteLine("2. Исследование доменов приложений и манипулирование ими.");

Console.WriteLine("3. Вычисление простых чисел в отдельном потоке.");

Console.WriteLine("4. Два потока для четных и нечетных чисел.");

Console.WriteLine("5. Повторение задачи с использованием таймера.");

Console.WriteLine("0. Выход.");

while (true)

{

Console.WriteLine("Ваш выбор: ");

switch (Console.ReadLine())

{

case "1":

ProcessManager.ListProcesses();

break;

case "2":

AppDomainManager.ManipulateAppDomain();

break;

case "3":

PrimeCalculator.CalculateInThread();

break;

case "4":

EvenOddThreads.RunThreads();

break;

case "5":

{

Console.Write("Введите значение n: ");

string input = Console.ReadLine();

if (int.TryParse(input, out int num))

{

RepeatingTask.StartCountdown(num);

}

else

{

Console.WriteLine("Введите число!");

}

break;

}

case "0":

return;

default:

Console.WriteLine("Неправильный выбор!");

break;

}

}

}

}

}