**Конструювання програмного забезпечення**

**Лабораторна робота№ 11**

Конструювання програм для роботи з потоками в середовищі програмування Visual C# 2010 Express Edition.

**Мета роботи:** засвоїти методику та виробити практичні навички в основах реалізації потоків в середовищі програмування Visual C# 2010 Express Edition.

**Теоретичні відомості.**

C# підтримує паралельне виконання коду через багатопоточність. Потік - це незалежний шлях виконання, здатний виконуватися одночасно з іншими потоками.

Програма на C # запускається як єдиний потік, автоматично створюваний CLR (Common Language Runtime — «загальне середовище виконання мов») і операційною системою ("головний" потік), і стає багатопоточною за допомогою створення додаткових потоків. Ось простий приклад і результат його виконання:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

namespace ThreadTest

{

class Program

{

static void Main(string[] args)

{

Thread t = new Thread(WriteY);

t.Start(); // Виконання WriteY в новому потоці

while (true)

Console.Write("x"); // Весь час друкує 'x'

}

static void WriteY()

{

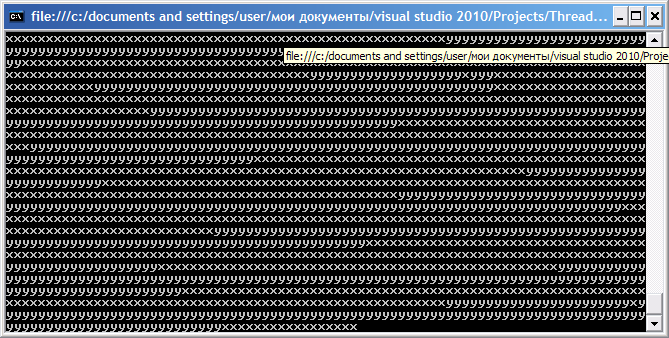
while (true)

Console.Write("y"); // Все Весь час друкує 'y'

}

}

}



У головному потоці створюється новий потік t, виконуючий метод, який безперервно друкує символ 'y'. Одночасно головний потік безперервно друкує символ 'x'. CLR призначає кожному потоку свій стек так, що локальні змінні зберігаються окремо.

Управління багатопоточністю здійснює планувальник потоків, цю функцію CLR зазвичай делегує операційній системі. Планувальник потоків гарантує, що активним потокам виділяється відповідний час на виконання, а потоки, які очікують або блоковані, приміром, на очікуванні ексклюзивної блокування, або для користувача введення - не споживають часу CPU.

На однопроцесорних комп'ютерах планувальник потоків використовує квантування часу - швидке перемикання між виконанням кожного з активних потоків. Це призводить до непередбачуваної поведінки, як в першому прикладі, де кожна послідовність символів 'X' і 'Y' відповідає кванту часу виділеного потоку. У Windows XP типове значення кванта часу - десятки мілісекунд - вибрано як набагато більше, ніж витрати CPU на перемикання контексту між потоками (кілька мікросекунд).

На багатопроцесорних комп'ютерах багатопоточність реалізована як суміш квантування часу і справжнього паралелізму, коли різні потоки виконують код на різних CPU. Необхідність квантування часу все одно залишається, тому що операційна система повинна обслуговувати як свої власні потоки, так і потоки інших програм.

Для створення потоків використовується конструктор класу Thread, який приймає як параметр делегат типу ThreadStart, який вказує метод, який потрібно виконати. Делегат ThreadStart визначається так:

public delegate void ThreadStart();

Виклик методу Start починає виконання потоку. Потік триває до виходу з виконуваного методу. Ось приклад, який використовує повний синтаксис C # для створення делегата ThreadStart:

class ThreadTest

{

static void Main()

{

Thread t = new Thread(new ThreadStart(Go));

t.Start();

Go();

}

static void Go() { Console.WriteLine("hello!"); }

У цьому прикладі потік виконує метод Go () одночасно з головним потоком. Результат - два майже одночасних «hello»:

hello!

hello!

Потік можна створити, використовуючи для присвоювання значень делегатам більш зручний скорочений синтаксис C #:

static void Main()

{

Thread t = new Thread(Go); // Без явного використання ThreadStart

t.Start();

...

}

static void Go() { ... }

У цьому випадку делегат ThreadStart виводиться компілятором автоматично. Інший варіант скороченого синтаксису використовує анонімний метод для створення потоку:

static void Main()

{

Thread t = new Thread(delegate() { Console.WriteLine("Hello!"); });

t.Start();

}

Потік, який закінчив виконання, не може бути початий заново.

Припустимо, що в розглянутому вище прикладі потрібно більш явно розрізняти результат кожного з потоків, наприклад, по регістру символів. Можна досягти цього, передаючи відповідний прапор в метод Go (), але в цьому випадку не можна використовувати делегат ThreadStart, тому що він не сприймає аргументів. .NET Framework визначає іншу версію делегата - ParameterizedThreadStart, яка може приймати один аргумент:

public delegate void ParameterizedThreadStart(object obj);

Попередній приклад можна переписати так:   
class ThreadTest

{

static void Main()

{

Thread t = new Thread(Go);

t.Start(true); // == Go(true)

Go(false);

}

static void Go(object upperCase)

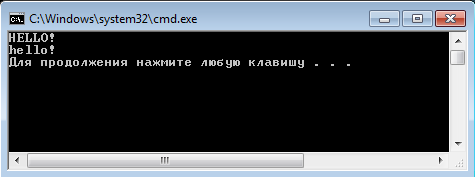
{

bool upper = (bool)upperCase;

Console.WriteLine(upper ? "HELLO!" : "hello!");

}

}



У цьому прикладі компілятор автоматично виводить делегат ParameterizedThreadStart, так як метод Go () приймає як параметр один object. З тим же успіхом можна було написати:

Thread t = new Thread(new ParameterizedThreadStart(Go));

t.Start(true);

Особливість використання ParameterizedThreadStart полягає в тому, що перед використанням потрібно привести аргумент з типу object до потрібного типу (в даному випадку bool). До того ж існує тільки версія, що приймає єдиний аргумент.

В якості альтернативи можна використовувати анонімний метод:

static void Main()

{

Thread t = new Thread(delegate(){ WriteText("Hello"); });

t.Start();

}

static void WriteText(string text) { Console.WriteLine(text); }

Зручність полягає в тому, що потрібний метод (в даному випадку WriteText) можна викликати з будь-якою кількістю аргументів і без жодного приведення типів. Однак треба взяти до уваги особливість семантики анонімних методів, пов'язану з зовнішньої змінної, яка стає очевидною в наступному прикладі:   
static void Main()

{

string text = "Before";

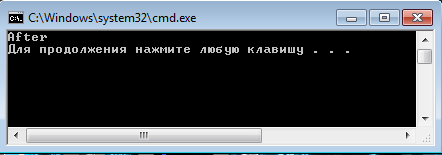
Thread t = new Thread(delegate() { WriteText(text); });

text = "After";

t.Start();

}

static void WriteText(string text) { Console.WriteLine(text); }



Потік можна перелічити, використовуючи властивість Name. Це надає велику зручність при налагодженні: імена потоків можна вивести в Console.WriteLine і побачити у вікні Debug - Threads в Microsoft Visual Studio.

Головному потоку програми також можна призначити ім'я - в наступному прикладі доступ до головного потоку здійснюється через статична властивість CurrentThread класу Thread:

sing System;

using System.Collections.Generic;

using System.Text;

using System.Threading;

namespace ConsoleApplication1

{

class ThreadNaming

{

static void Main()

{

Thread.CurrentThread.Name = "main";

Thread worker = new Thread(Go);

worker.Name = "worker";

worker.Start();

Go();

}

static void Go()

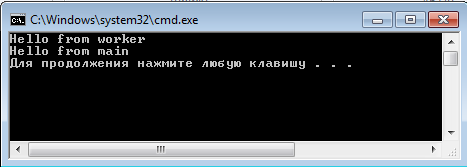
{

Console.WriteLine("Hello from " + Thread.CurrentThread.Name);

}

}

}



У наступному прикладі визначається метод з локальної змінної, а потім виконує його одночасно у головному і в новоствореному потоках:

using System;

using System.Collections.Generic;

using System.Text;

using System.Threading;

namespace ConsoleApplication1

{

class Program

{

static void Main()

{

new Thread(Go).Start(); // Виконати Go() в новому потоці

Go(); // Виконати Go() в головному потоці

}

static void Go()

{

// Определяем и используем локальную переменную 'cycles'

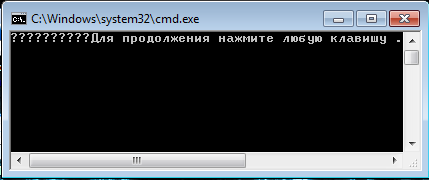
for (int cycles = 0; cycles < 5; cycles++)

Console.Write('?');

}

}

}



Виклик Thread.Sleep блокує поточний потік на вказаний час (або до переривання):

static void Main()

{

Thread.Sleep(0); // відмовляється від одного квата часу CPU

Thread.Sleep(1000); // заснути на 1000 мілісекунд

Thread.Sleep(TimeSpan.FromHours(1)); // заснути на 1 годину

}

Потік можна заблокувати до завершення іншого потоку викликом методу Join:

class JoinDemo

{

static void Main()

{

Thread t = new Thread(delegate() { Console.ReadLine(); });

t.Start();

t.Join(); // чекати поки потік незакінчиться

Console.WriteLine("Thread t's ReadLine complete!");

}

}

Метод Join може також приймати в якості аргументу timeout - у мілісекундах або як TimeSpan. Якщо вказаний час минув, а потік не завершився, Join повертає false. Join функціонує як Sleep - фактично наступні два рядки коду призводять до однакового результату:

Thread.Sleep (1000);

Thread.CurrentThread.Join (1000);

**Завдання**

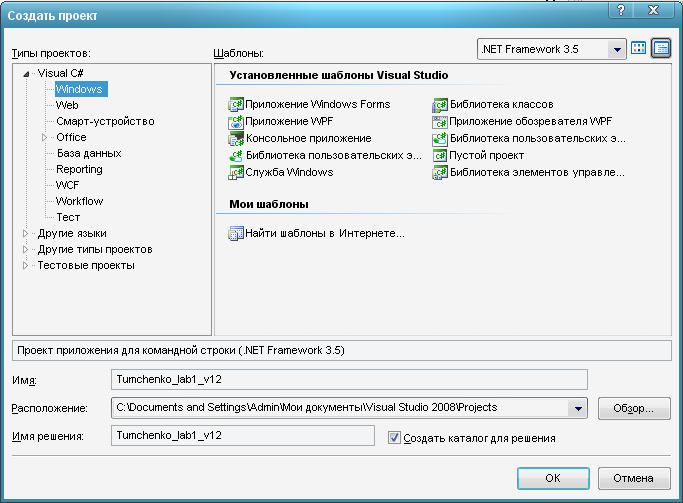
Реалізувати багато поточність в програмі, яка була розроблена в попередній лабораторній роботі на тему «Робота з ADO.NET та наборами в середовищі програмування Visual C# 2010 Express Edition» згідно варіантів.

Програма повинна містити:

* Реалізація прикладу з «Х» та «У»
* Кожна дія (вивід кількості елементів, зчитування, запис та знищення даних в БД) і пошук по всіх полях має виконуватися в окремому потоці
* Кожен потік має мати відповідне ім’я, яке потрібне виводити на консоль

**Хід роботи.**

1. Ознайомитися з теоретичними відомостями та прикладом виконання лабораторної роботи.
2. Розробити UML-діаграму програми.
3. Запустити Visual Studio 2003/2005/2008. Вибрати в меню **Файл/Создать/Проект,** у вікні, що відкриється вибрати тип проекту, шаблон консольної програми і ввести назву, яка відповідає прізвищу студента, його варіанту і номеру лабораторної роботи.



1. Реалізувати програму.
2. Розробити звіт. Звіт має включати номер і назву лабораторної роботи, мету, задачу, постановку задачі, розробку структури даних, блок-схему (UML-діаграму), код програми, результат виконання програми та висновок.
3. Захистити лабораторну роботу.