Паралельне програмування з використанням бібліотеки TPL та PLINQ

Лекція 1. Бібліотека TPL. Паралелізм задач

1. Архітектура паралельного програмування .NET Framework 4

В .NET Framework 4 і Visual Studio 2010 з'явилися нові засоби підтримки паралельного програмування, які спрощують паралельну розробку. Загальний огляд архітектури паралельного програмування в .NET Framework 4 зображено на рис. 1.1. Звичайно, ці нові засоби базуються на засобах багатопоточності, але дозволяють спростити багатопоточне програмування.

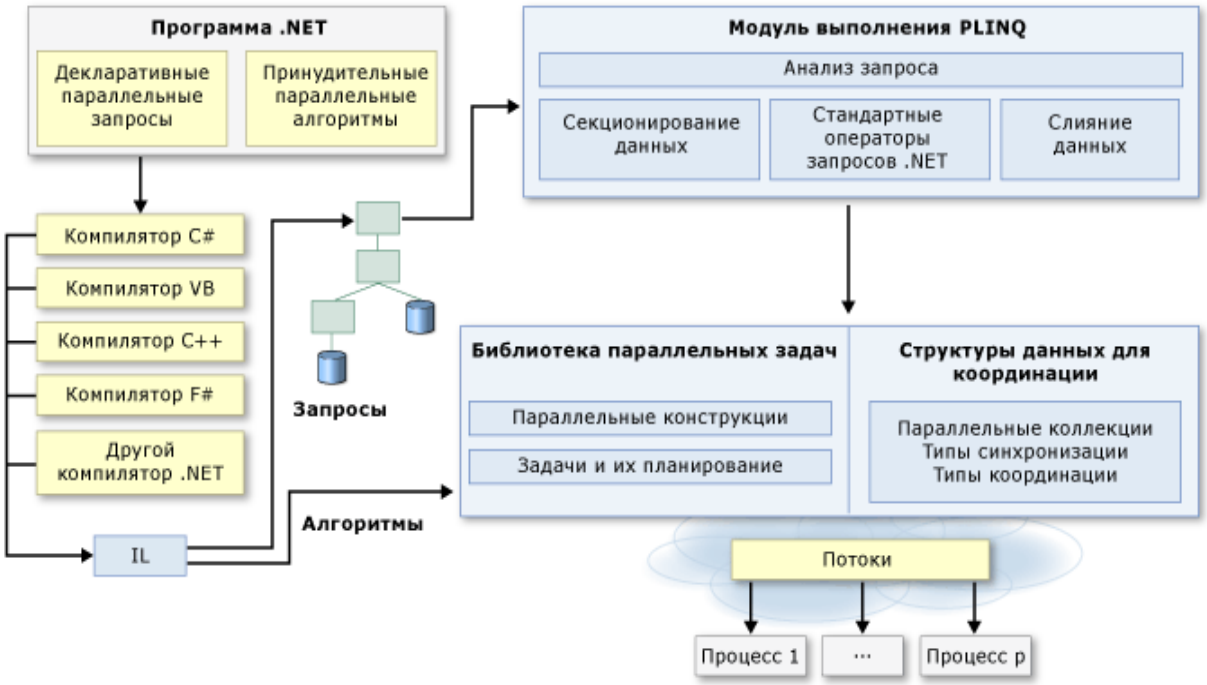


Рис. 1.1. Архітектура паралельного програмування .NET Framework 4

До засобів паралельного програмування в .NET Framework 4 належить, насамперед, *бібліотека паралельних задач TPL* (Task Parallel Library). Ця бібліотека удосконалює багатопотокове програмування двома основними способами.

По-перше, вона спрощує створення і застосування багатьох потоків.

По-друге, вона дозволяє автоматично використовувати декілька процесорів.

Інакше кажучи, TPL відкриває можливості для автоматичного масштабування програм з метою ефективного використання ряду доступних процесорів. Завдяки цим двом особливостям бібліотеки TPL

вона рекомендується до застосування для організації багатопотокової обробки.

Ще одним новим засобом паралельного програмування, який з'явився у версії 4.0 .NET Framework, є *паралельна мова інтегрованих запитів (PLINQ)*. Мова PLINQ дає можливість писати запити, для обробки яких автоматично використовується декілька процесорів, а також принцип паралелізму, коли це доречно.

2. Два підходи до паралельного програмування

Застосовуючи TPL, паралелізм в програму можна ввести двома основними способами.

Перший з них називається *паралелізмом даних*. При такому підході одна операція над сукупністю даних розбивається на два або більше паралельно виконуваних потоки, в кожному з яких обробляється частина даних. Так, якщо змінюється кожен елемент масиву, то, застосовуючи паралелізм даних, можна організувати паралельну обробку різних областей масиву в двох або більше потоках.

Не дивлячись на те, що паралелізм даних був завжди можливий і за допомогою класу Thread, побудова масштабованих рішень засобами цього класу вимагали чимало зусиль і часу. Використання бібліотеки TPL спрощує внесення такого паралелізму даних в програму.

Другий спосіб введення паралелізму називається *паралелізмом задач*. При такому підході дві або більше операції виконуються паралельно. Отже, паралелізмом задач є різновидом паралелізму, який досягався у минулому засобами класу Thread. А до переваг, які обіцяє застосування TPL, відноситься простота застосування і можливість автоматично масштабувати виконання коду на декілька процесорів.

Бібліотека TPL дозволяє автоматично розподіляти завантаження програм між доступними процесорами CLR. Це дозволяє підвищити продуктивність роботи програм без прямого використання потоків.

Бібліотека TPL визначена в просторі імен

System.Threading.Tasks

Але для роботи з нею потрібно також включати в програму клас System.Threading, оскільки він підтримує синхронізацію і інші засоби багатопоточності.

Структура простору імен **System.Threading.Tasks** зображена на рис. 1.2.

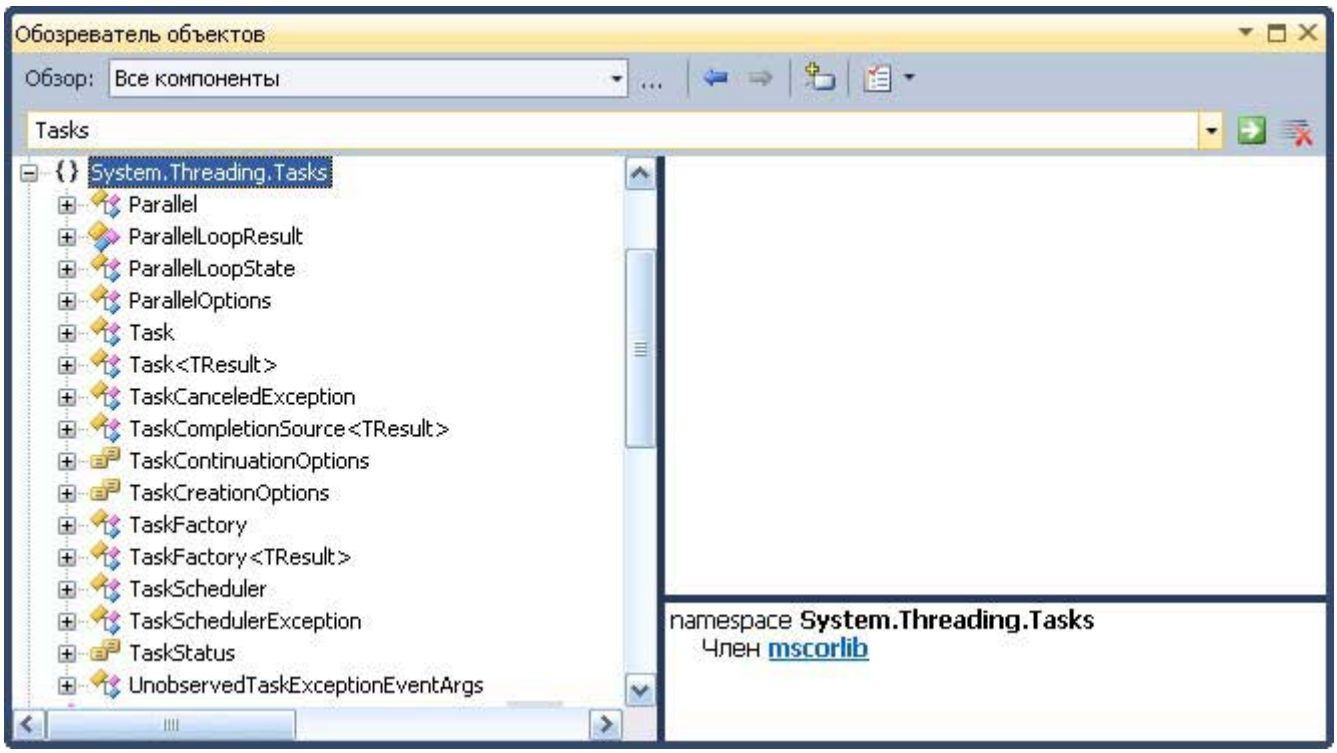


Рис. 1.2. Ієрархія класів простіру імен System.Threading.Tasks

3. Паралелізм задач. Клас Task

Паралелізм задач (task parallelism) – це самий низькорівневий підхід до розпаралелювання задач. Основні класи, які забезпечують паралелізм задач:

Клас	Призначення
Task	Для управління одиницею роботи
Task<TResult>	Для управління одиницею роботи, яка повертає значення
TaskFactory	Для створення задач
TaskFactory<TResult>	Для створення задач і продовжень з тим самим типом результата
TaskScheduler	Для управління планувальником завдань
TaskCompletionSource	Для ручного управління життєвим циклом завдання.

В основу TPL покладений клас **Task**. Цей клас відрізняється від класу Thread тим, що він є абстракцією, що представляє *асинхронну операцію*. А в класі Thread реалізується потік виконання. На системному рівні потік як і раніше залишається елементарною одиницею виконання,

яку можна планувати засобами операційної системи. Але відповідність екземпляра об'єкту класу Task і потоку виконання не обов'язково є взаємно-однозначною. Крім того, виконанням задач управляє планувальник задач, який працює з пулом потоків. Це, наприклад, означає, що декілька задач можуть розділяти один і той же потік.

Приклади:

Робота із задачами включає три основні операції: створення задачі, додавання задачі в чергу готових завдань, очікування завершення виконання задачі.

Робота з потоками:

```
Thread threadOne = new Thread(SomeWork);
threadOne.Start();
threadOne.Join();
```

Работа с задачами:

```
Task taskOne = new Task(SomeWork);
taskOne.Start();
taskOne.Wait();
```

Важлива відмінність полягає в тому, що виклик методу Start для задачі не створює новий потік, а поміщає задачу в чергу готових задач - пул потоків. Планувальник (TaskScheduler) відповідно до своїх правил розподіляє готові задачі по робочих потоках. Дії планувальника можна коригувати за допомогою параметрів задач. Момент фактичного запуску задачі в загальному випадку не визначений і залежить від завантаженості пулу потоків.

Створення задачі

Створити нову задачу у вигляді об'єкту класу Task і почати її виконання можна різними способами. Спочатку створимо об'єкт типу Task за допомогою конструктора і запустимо його, викликавши метод Start (). Для цього в класі Task визначено декілька конструкторів. Нижче наведено одну з форм конструктора:

public Task(Action дія)

де дія позначає точку входу в код, що представляє задачу, Action — делегат, визначений в просторі імен System. Форма делегата Action, якою ми будемо користуватися, виглядає таким чином.

public delegate void Action()

Таким чином, точкою входу повинен бути метод, що не приймає ніяких параметрів і не повертає ніяких значень.

Як тільки задача створена, її можна запустити на виконання, викликавши метод Start (). Нижче наведено одну з форм методу Start ().

public void Start()

Після виклику методу Start () планувальник запланує виконання задачі.

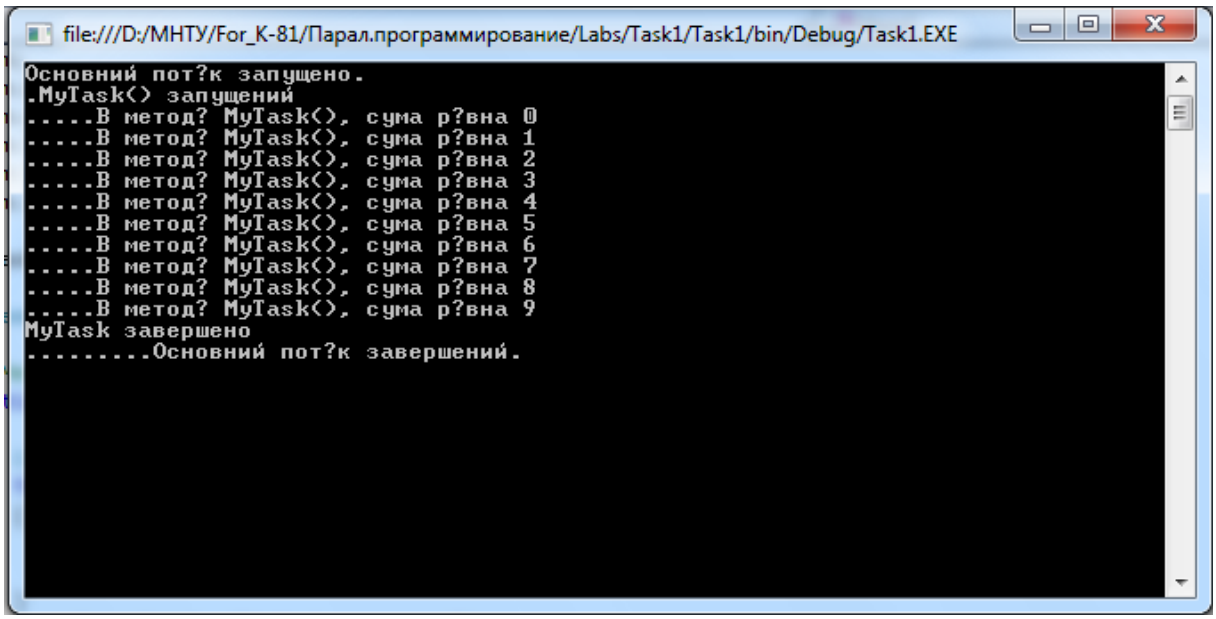
Приклад 1.1

В програмі створюється задача на основі методу MyTask (). Після того, як почне виконуватися метод Main (), задача фактично створюється і запускається на виконання. Обидва методи MyTask() і Main () виконуються паралельно.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Task1
{
    class DemoTask
    {
        // Метод, який виконується як задача
        static void MyTask()
        {
            Console.WriteLine("MyTask() запущений");
            for (int count = 0; count < 10; count++)
            {
                Thread.Sleep(500);
                Console.WriteLine("В методі MyTask(), сума рівна " + count);
            }
            Console.WriteLine("MyTask завершено");
        }
        static void Main()
        {
            Console.WriteLine("Основний потік запущено.");
            // Створити об'єкт задачі.
            Task tsk = new Task(MyTask);
            // Запустити задачу на виконання
            tsk.Start();
            // метод Main() активний до завершения методу MyTask().
            for (int i = 0; i < 60; i++)
            {
                Console.Write(".");
                Thread.Sleep(100);
            }
            Console.WriteLine("Основний потік завершений.");
            Console.ReadKey();
        }
    }
}
```

Результат виконання.



За замовчанням задача виконується у фоновому потоці. Отже, при завершенні основного потоку завершується і сама задача. Саме тому в прикладі 1.1 метод Thread.Sleep() використаний для збереження активним основного потоку до тих пір, поки не завершиться виконання методу MyTask ().

У наведеному вище прикладі програми задача, що призначалася для паралельного виконання, реалізована у вигляді статичного методу. Але така вимога до задачі не є обов'язковою. Наприклад, в наведеній нижче програмі 1.2, яка є перероблением варіантом попередньої, метод MyTask (), що виконує роль задачі, розміщений усередині класу.

Приклад 1.2

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Task1
{
    class MyClass
    {
       // Метод, виконуваний як задача
      public void MyTask()
      {
         Console.WriteLine("MyTask() запущений");
         for(int count = 0; count < 10; count++)
         {
           Thread.Sleep(100);
           Console.WriteLine("В методі MyTask(), сума рівна " + count);
         }
      Console.WriteLine("MyTask завершено ");
      }
    }
    class DemoTask
    {
```

```
        static void Main()
        {
            Console.WriteLine("Основний потік запущений.");
            // Створити об'єкт типа MyClass.
            MyClass mc = new MyClass();
            // Створити об'єкт задачі для методу mc.MyTask().
            Task tsk = new Task(mc.MyTask);
            // Запустити задачу на виконання,
            tsk.Start();
            // Зберегти метод Main() активним до завершення методу MyTask().
            for (int i = 0; i < 60; i++)
            {
                Console.Write(".");
                Thread.Sleep(100);
            }
            Console.WriteLine("Основной поток завершен.");
        }

    }
}
```

Результат виконання цієї програми таким самий, як і в прикладі 8.1. Єдина відмінність полягає в тому, що метод MyTask () викликається тепер для екземпляра об'єкту класу MyClass.

4. Застосування ідентифікатора задачі

На відміну від класу Thread; у класі Task не має властивості Name для зберігання імені задачі. Але замість цього є властивість **Id** для зберігання ідентифікатора задачі. Властивість **Id** доступна тільки для читання і відноситься до типу int. Вона оголошується таким чином.

public int Id { get; }

Кожна задача при створенні отримує ідентифікатор. Значення ідентифікаторів унікальні, але не впорядковані. Тому один ідентифікатор задачі може з'явитися перед іншим, хоча він може і не мати меншого значення.

Ідентифікатор виконуваної зараз задачі можна виявити за допомогою властивості **CurrentId**. Ця властивість доступна тільки для читання, відноситься до типу static і оголошується таким чином.

public static Nullable<int> CURRENTID { get; }

Властивість повертає виконувану зараз задачу або ж порожнє значення, якщо код не є задачею.

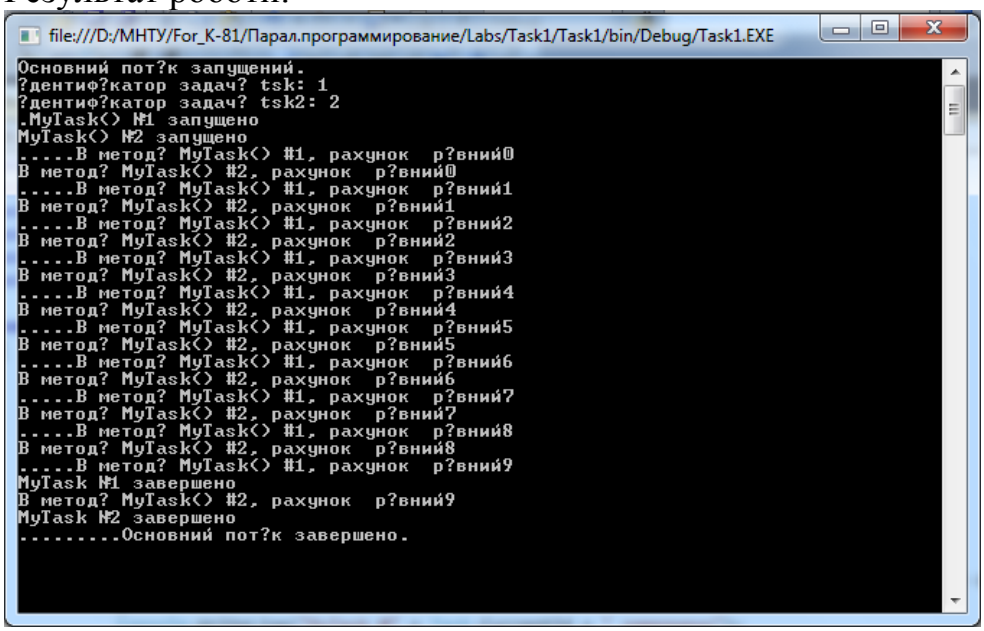
У прикладі 1.3. створюються дві задачі і показується, яка з них виконується.

Приклад 1.3

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

```
namespace Task1
{
    class DemoTask
    {
        // Метод, який виконується як задача,
        static void MyTask()
        {
            Console.WriteLine("MyTask() №" + Task.CurrentId + " запущено");
            for (int count = 0; count < 10; count++)
            {
                Thread.Sleep(500);
                Console.WriteLine("В методі MyTask() #" + Task.CurrentId + ", рахунок
рівний" + count);
            }
            Console.WriteLine("MyTask №" + Task.CurrentId + " завершено");
        }
        static void Main()
        {
            Console.WriteLine("Основний потік запущений.");
            // Створити об'єкти для двох задач.
            Task tsk1 = new Task(MyTask);
            Task tsk2 = new Task(MyTask);
            // Запустити задачі на виконання.
            tsk1.Start();
            tsk2.Start();
            Console.WriteLine("Ідентифікатор задачі tsk: " + tsk1.Id);
            Console.WriteLine("Ідентифікатор задачі tsk2: " + tsk2.Id);
            // Зберегти метод Main() активним до завершення решти задач
            for (int i = 0; i < 60; i++)
            {
                Console.Write(".");
                Thread.Sleep(100);
            }
            Console.WriteLine("Основний потік завершено.");
            Console.ReadKey();
        }
    }
}
```

Результат роботи:



5. Застосування методів очікування

У наведених вище прикладах основний потік виконання, а саме, метод Main(), завершувався тому, що такий результат гарантували виклики методу Thread.Sleep(). Але подібний підхід не можна вважати задовільним.

Організувати очікування завершення задач можна кращим способом, застосовуючи методи очікування, що спеціально надаються в класі Task. Найпростішим з них є метод Wait (), який припиняє виконання основного потоку до тих пір, поки не завершиться задача, що викликається. Нижче наведена проста форма оголошення цього методу.

public void Wait()

При виконанні цього методу можуть згенеруватися два виключення. Першим з них є виключення ObjectDisposedException. Воно генерується в тому випадку, якщо задача звільнена за допомогою виклику методу **Dispose ().** А друге виключення, AggregateException, генерується в тому випадку, якщо задача сама генерує виключення або ж відміняється. Як правило, відстежується і обробляється саме це виключення. У зв'язку з тим що завдання може згенерувати не одне виключення, якщо, наприклад, у неї є породжені завдання, всі подібні виключення збираються в єдине виключення типу AggregateException.

Нижче наведений варіант попередньої програми, змінений з метою продемонструвати застосування методу Wait(). Цей метод використовується усередині методу Main (), щоб припинити його виконання до тих пір, поки не завершаться обидві задачі tsk1 і tsk2.

Приклад 1.4

// Застосування методу Wait().

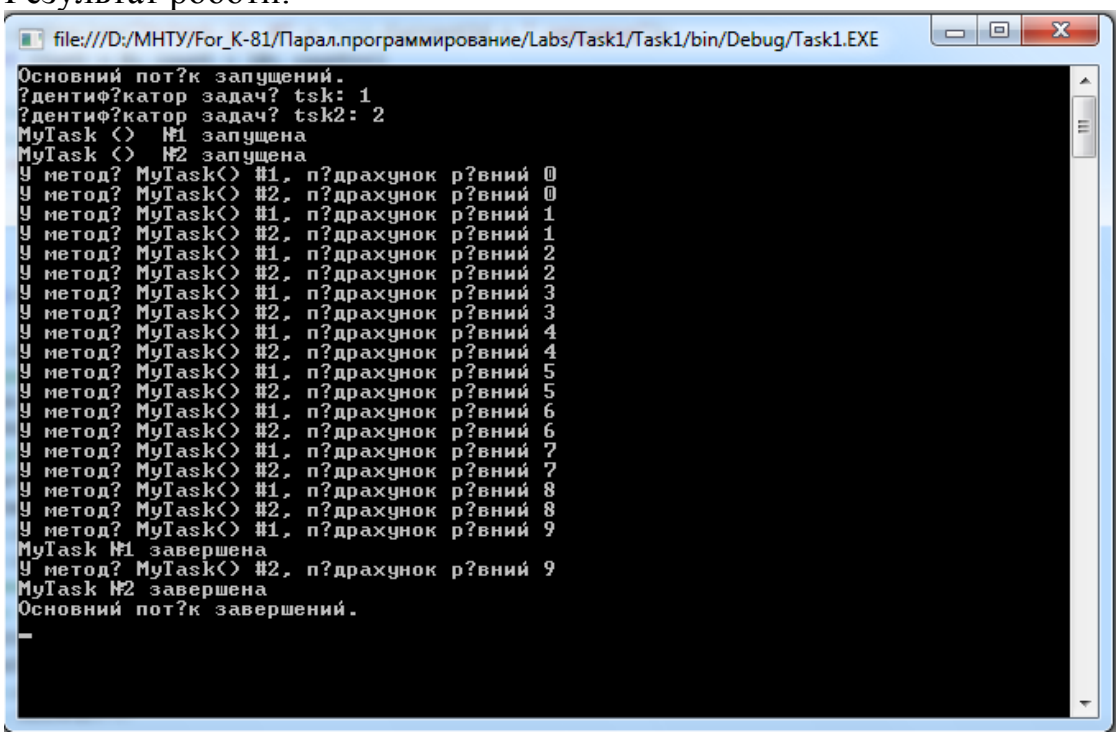
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Task1
{
    class DemoTask
    {
    // Метод, що виконується як задача
        static void MyTask()
        {
            Console.WriteLine("MyTask ()  №" + Task.CurrentId + " запущена");
            for (int count = 0; count < 10; count++)
            {
                Thread.Sleep(500);
                Console.WriteLine("У методі MyTask() #" + Task.CurrentId +
                ", підрахунок рівний " + count);
            }
            Console.WriteLine("MyTask №" + Task.CurrentId + " завершена");
        }

        static void Main()
```

```
        {
            Console.WriteLine("Основний потік запущений.");
            // Створити об'єкти двох задач
            Task tsk1 = new Task(MyTask);
            Task tsk2 = new Task(MyTask);
            // Запустити задачу на виконання.
            tsk1.Start();
            tsk2.Start();
            Console.WriteLine("Ідентифікатор задачі tsk: " + tsk1.Id);
            Console.WriteLine("Ідентифікатор задачі tsk2: " + tsk2.Id);
            // Зупинити виконання методу Main() доти
            // поки не завершаться обидва завдання tsk і tsk2
            tsk1.Wait();
            tsk2.Wait();
            Console.WriteLine("Основний потік завершений.");
            Console.ReadKey();
        }
    }
}
```

Результат роботи:



Як випливає з наведеного вище результату, виконання методу Main() припиняється до тих пір, поки не завершаться обидві задачі tsk1 і tsk2.

В нашому випадку виявляється достатньо двох викликів методу Wait(.), але такий самий результат можна отримати, скориставшись методом WaitAll(). Цей метод організує очікування завершення групи задач. Повернення з нього не відбудеться до тих пір, поки не завершаться всі задачі. Нижче наведена проста форма оголошення цього методу.

public static void WaitAll(params Task[] tasks)

Задачі, завершення яких потрібно чекати, передаються за допомогою параметра у вигляді масиву tasks. А оскільки цей параметр відноситься до

типу params, то даному методу можна окремо передати масив об'єктів типу Task або список задач. При цьому можуть згенеруватися різні виключення, включаючи і AggregateException.

Для того, щоб подивитися, як метод WaitAll() діє на практиці, замінимо в наведеному вище прикладі 1.3 послідовність викликів.

tsk1.Wait();
tsk2.Wait();
на
Task.WaitAll(tsk1, tsk2) ;

Програма працюватиме так само.

Організуючи очікування завершення декількох задач, слід бути особливо уважним, щоб уникнути взаємоблокувань. Так, якщо дві задачі чекають завершення одна одної, то виклик методу WaitAll() взагалі не призведе до повернення з нього.

Іноді потрібно організувати очікування до тих пір, поки не завершиться будь-яка з групи задач. Для цього призначений метод WaitAny(). Нижче наведена проста форма його оголошення.

public static int WaitAny(params Task[] tasks)

Задачі, завершення яких потрібно чекати, передаються за допомогою параметра у вигляді масиву tasks об'єктів типу Task або окремого списку аргументів типу Task. Цей метод повертає індекс задачі, яка завершується першою. При цьому можуть згенеруватися різні виключення.

Спробуйте застосувати метод WaitAny () на практиці, підставивши в попередній програмі наступний виклик.

Task.WaitAny(tsk1, tsk2);

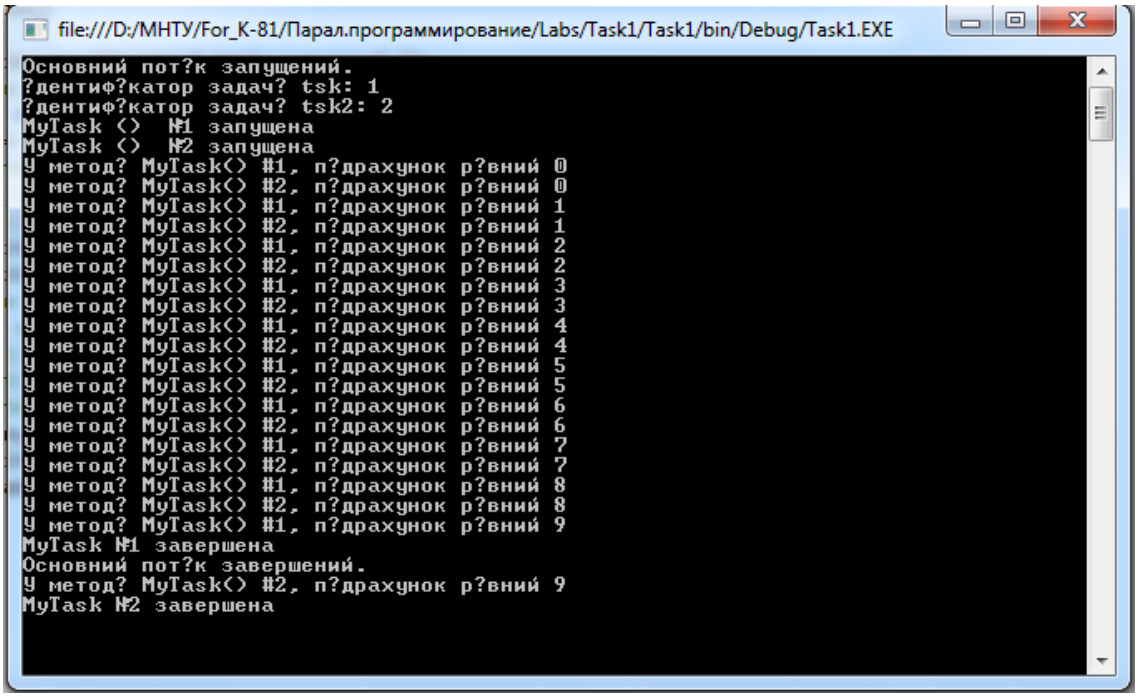
Отримаємо:

```
namespace Task1
{
    class DemoTask
    {
        // Метод, що виконується як задача
        static void MyTask()
        {
            Console.WriteLine("MyTask ()  №" + Task.CurrentId + " запущена");
            for (int count = 0; count < 10; count++)
            {
                Thread.Sleep(500);
                Console.WriteLine("У методі MyTask() #" + Task.CurrentId +
                ", підрахунок рівний " + count);
            }
            Console.WriteLine("MyTask №" + Task.CurrentId + " завершена");
        }

        static void Main()
        {
            Console.WriteLine("Основний потік запущений.");
            // Створити об'єкти двох задач
```

```
            Task tsk = new Task(MyTask);
            Task tsk2 = new Task(MyTask);
            // Запустити задачу на виконання.
            tsk.Start();
            tsk2.Start();
            Console.WriteLine("Ідентифікатор задачі tsk: " + tsk.Id);
            Console.WriteLine("Ідентифікатор задачі tsk2: " + tsk2.Id);
            // Зупинити виконання методу Main() доти
            // поки не завершаться обидва завдання tsk і tsk2
           // tsk.Wait();
          //  tsk2.Wait();
    //        Task.WaitAll(tsk, tsk2);
            Task.WaitAny(tsk, tsk2);
            Console.WriteLine("Основний потік завершений.");
            Console.ReadKey();
        }
    }
}
```

В результаті виконання методу Main () поновиться, а програма завершиться, як тільки завершиться одна з двох задач.



Крім форм методів Wait(), Wait All () і WaitAny () є інші їх варіанти, в яких можна указувати період простою або відстежувати ознаку відміни.

Приклад 1.5

http://msdn.microsoft.com/ru-ru/library/system.threading.tasks.task.aspx

Цей приклад демонструє використання методів класу Task для створення, запуску і призупинення задач.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading;
using System.Threading.Tasks;

namespace Task1
{
    class StartNewDemo
    {

        static void Main()
        {
            Action<object> action = (object obj) =>
            {
                Console.WriteLine("Task={0}, obj={1}, Thread={2}", Task.CurrentId,
obj.ToString(), Thread.CurrentThread.ManagedThreadId);
            };

            // Construct an unstarted task
            Task t1 = new Task(action, "alpha");

            // Cosntruct a started task
            Task t2 = Task.Factory.StartNew(action, "beta");

            // Block the main thread to demonstate that t2 is executing
            t2.Wait();

            // Launch t1
            t1.Start();

            Console.WriteLine("t1 has been launched. (Main Thread={0})",
Thread.CurrentThread.ManagedThreadId);

            // Wait for the task to finish.
            // You may optionally provide a timeout interval or a cancellation token
            // to mitigate situations when the task takes too long to finish.
            t1.Wait();

            // Construct an unstarted task
            Task t3 = new Task(action, "gamma");

            // Run it synchronously
            t3.RunSynchronously();

            // Although the task was run synchrounously, it is a good practice to wait
for it which observes for
            // exceptions potentially thrown by that task.
            t3.Wait();
            Console.ReadKey();
        }
    }
}
```

Висновки

В цій лекції ми розглянули нові засоби паралельного програмування, які реалізовані в бібліотеці TPL (Task Parallel Library). Ці засоби дозволяють спростити розробку паралельних програм. Замість потоків в цій бібліотеці використовується поняття задачі (Task). Бібліотека TPL удосконалює багатопотокове програмування двома основними способами.

По-перше, вона спрощує створення і застосування багатьох потоків.

По-друге, вона дозволяє автоматично використовувати декілька процесорів.

Контрольні запитання і завдання для самостійного виконання

1. Способи внесення паралелізму в програмування та їхня характеристика. Нові засоби паралельного програмування в .NET Framework 4.
2. Яка різниця між паралелізмом задач і паралелізмом даних?
3. Яка різниця між класами Task і Thread?
4. Що робить наступний фрагмент коду?

```
Task tsk = new Task(NewTask);
tsk.Start();
```

5. Як можна звернутися до задачі у разі потреби?
6. Що є результатом виконання наступної команди:

```
int IdTsk1=Task.CurrentId?
```

7. Що є результатом виконання наступної команди:

```
int IdTsk1=Task.CurrentId?
```

8. Яка різниця між методами WaitAll() і WaitAny()
9. Виконайте все наведені в лекції приклади та поясніть їхню роботу.
10. Виконайте приклад 1.5 та поясніть його роботу.