ТЕМА 1: ПАРАЛЕЛІЗМ ДАНИХ ТА ПАРАЛЕЛІЗМ ЗАДАЧ (Ч. 1)

Бібліотека TPL (Task Parallel Library) у .NET забезпечує потужний та зручний спосіб створення та управління паралельними задачами. Вона дозволяє створювати код з меншими зусиллями та більшою ефективністю. В основу TPL покладено клас `Task`. Елементарна одиниця виконання інкапсулюється в TPL засобами класу Task, а не Thread. Клас `Task` відрізняється від класу `Thread` тим, що він є абстракцією, що представляє асинхронну операцію.

Створення задачі. Для початку створимо об'єкт типу `Task` за допомогою конструктора

```
public Task(Action дія)
```

і запустимо його, викликавши метод `Start ()`.

```
//Сконструюввати об'єкт задачі
Task tsk=new Task(DemoTask.MyTask);

//Запустити задачу на виконання
tsk.Start();
```

Точкою входу повинен служити метод, що не приймає ніяких параметрів і не повертає ніяких значень.

```
public static void MyTask()
{
      Console.WriteLine("MyTask() is started.");
      for (int count=0;count<10;count++)
      {
            Thread.Sleep(500);
            Console.WriteLine("In the method MyTask() counter = " + count);
      }
      Console.WriteLine("MyTask() is done.");
}
```

У цій програмі окреме завдання створюється на основі методу `MyTask ()`. Після того як почне виконуватися метод `Main ()`, завдання фактично створюється і запускається на виконання. Обидва методи `MyTask ()` і `Main ()` виконуються паралельно.

```
namespace Lz4
{
      class DemoTask
      {
            //Метод, що виконується в якості задачі
            public static void MyTask()
            {
                  Console.WriteLine("MyTask() is started.");
                  for (int count=0;count<10;count++)
                  {
                        Thread.Sleep(500);
                        Console.WriteLine("In the method MyTask() counter = " + count);
                  }
                  Console.WriteLine("MyTask() is done.");
            }
      }
      class Program
      {
            static void Main(string[] args)
            {
                  Console.WriteLine("Main Thread is starting.");
```

```
            //Сконструюввати об'єкт задачі
            Task tsk=new Task(DemoTask.MyTask);

            //Запустити задачу на виконання
            tsk.Start();

            //Залишати метод Main() активним до завершення методу MyTask()
            for (int i=0;i<60;i++)
            {
                  Console.Write(".");
                  Thread.Sleep(100);
            }

            Console.WriteLine("Main() is done.");
            Console.ReadLine();
      }
   }
}
```

Результат виконання:

```
Main Thread is starting.
.MyTask() is started.
.....In the method MyTask() counter = 0
.....In the method MyTask() counter = 1
.....In the method MyTask() counter = 2
.....In the method MyTask() counter = 3
.....In the method MyTask() counter = 4
.....In the method MyTask() counter = 5
.....In the method MyTask() counter = 6
.....In the method MyTask() counter = 7
.....In the method MyTask() counter = 8
.....In the method MyTask() counter = 9
MyTask() is done.
.........Main() is done.
```

У наведеному прикладі програми задача, що призначалася для паралельного виконання, описана у вигляді статичного методу. Але така вимога до задачі не є обов'язковою. Наприклад, у наступній програмі метод `MyTask()`, що виконує роль задачі, інкапсульований всередині класу.

```
class MyClass
{
   //Метод, що виконується в якості задачі
   public void MyTask()
   {
      Console.WriteLine("MyTask() is started.");
      for (int count = 0; count < 10; count++)
      {
         Thread.Sleep(500);
         Console.WriteLine("In the method MyTask() counter = " + count);
      }
      Console.WriteLine("MyTask() is done.");
   }
}
```

Створення та запуск на виконання задачі:

```
//Сконструюввати об'єкт типу MyClass
MyClass mc=new MyClass();

//Сконструюввати об'єкт задачі
Task tsk=new Task(mc.MyTask);

//Запустити задачу на виконання
tsk.Start();
```

Результат виконання програми ідентичний до попереднього.

Застосування ідентифікатора задачі. У класі `Task` є властивість `Id` для зберігання ідентифікатора задачі, за яким можна розпізнавати задачі. Властивість `Id` доступна тільки для читання і відноситься до типу `int`. Вона оголошується наступним чином.

```
public int Id { get; }
```

Кожна задача отримує ідентифікатор, коли вона створюється. Значення ідентифікаторів унікальні, але не впорядковані.

Ідентифікатор задачі, що виконується в даний момент, можна виявити за допомогою властивості `CurrentId`. Це властивість доступна тільки для читання, відноситься до типу `static` і оголошується наступним чином.

```
public static Nullable<int> CurrentID { get; }
```

В наступному прикладі створюються дві задачі та показується, яка саме задача виконується в певний момент.

```
namespace Prog3 //використання ідентифікатора задачі
{
      class Program
      {
            //Метод, що виконується як задача
            static void MyTask()
            {
                  Console.WriteLine("MyTask №" + Task.CurrentId + " is started.");

                  for (int count = 0; count < 10; count++)
                  {
                        Thread.Sleep(500);
                        Console.WriteLine("In the method MyTask() №"+Task.CurrentId+"
counter = " + count);
                  }
                  Console.WriteLine("MyTask() №"+Task.CurrentId+ " is done.");
            }

            static void Main(string[] args)
            {
                  Console.WriteLine("Main Thread is starting.");

                  //Сконструюввати об'єкти двох задач
                  Task tsk1=new Task(MyTask);
                  Task tsk2=new Task(MyTask);

                  //Запустити задачі на виконання
                  tsk1.Start();
                  tsk2.Start();


                  Console.WriteLine("Id of task tsk1 = "+tsk1.Id);
```

```
                Console.WriteLine("Id of task tsk2 = " + tsk2.Id);

                for (int i = 0; i < 60; i++)
                {
                      Console.Write(".");
                      Thread.Sleep(100);
                }

                Console.WriteLine("Main() is done.");
                Console.ReadLine();
           }
      }
}
```

Результат виконання:

```
Main Thread is starting.
Id of task tsk1 = 1
Id of task tsk2 = 2
.MyTask №1 is started.
MyTask №2 is started.
....In the method MyTask() №1 counter = 0
.In the method MyTask() №2 counter = 0
....In the method MyTask() №1 counter = 1
.In the method MyTask() №2 counter = 1
....In the method MyTask() №1 counter = 2
In the method MyTask() №2 counter = 2
.....In the method MyTask() №1 counter = 3
In the method MyTask() №2 counter = 3
.....In the method MyTask() №1 counter = 4
.In the method MyTask() №2 counter = 4
....In the method MyTask() №1 counter = 5
In the method MyTask() №2 counter = 5
......In the method MyTask() №1 counter = 6
In the method MyTask() №2 counter = 6
....In the method MyTask() №1 counter = 7
.In the method MyTask() №2 counter = 7
....In the method MyTask() №1 counter = 8
.In the method MyTask() №2 counter = 8
....In the method MyTask() №1 counter = 9
MyTask() №1 is done.
.In the method MyTask() №2 counter = 9
MyTask() №2 is done.
.........Main() is done.
```

Для організації доцільніше використовувати метод `Wait ()`, який призупиняє виконання до тих пір, поки не завершиться задача, що викликається.

```
tsk1.Wait();
tsk2.Wait();
```

Можна скористатися іншим методом, який організовує очікування групи задач.

```
Task.WaitAll(tsk1, tsk2);
```

Метод, що очікує, поки не завершиться будь-яка з групи задач:

```
Task.WaitAny(tsk1, tsk2);
```

Метод `Dispose()` реалізується в класі `Task`, звільняючи ресурси, що використовуються цим класом. Це особливо важливо в тих програмах, де створюється

велика кількість задач. Слід мати на увазі, що метод `Dispose()` можна викликати для окремої задачі тільки після її завершення.

Застосування класу TaskFactory для запуску задачі. Задачу можна створити і відразу ж почати її виконання, викликавши метод `StartNew()`, визначений в класі `TaskFactory`.

```
//Сконструюввати та одразу запустити об'єкти двох задач
Task tsk1=Task.Factory.StartNew(MyTask);
Task tsk2=Task.Factory.StartNew(MyTask);
```

Метод `StartNew()` виявляється більш ефективним в тих випадках, коли задача створюється і відразу ж запускається на виконання.

Застосування лямбда-виразу в якості задачі. Крім використання звичайного методу в якості задачі, існує й інший, більш раціональний підхід: вказати лямбда-вираз як окремо вирішувану задачу. Лямбда-вираз є особливою формою анонімних функцій.

```
//Лямбда-вираз використовується для означення задачі
Task tsk1=Task.Factory.StartNew(()=>
{
      Console.WriteLine("Lambda_expr is started.");
      for (int count = 0; count < 10; count++)
      {
            Thread.Sleep(500);
            Console.WriteLine("In the Lambda_expr counter = " + count);
      }
      Console.WriteLine("Lambda_expr is done.");
});
```

Виклик методу `tsk.Dispose()` не відбувається до тих пір, поки не відбудеться повернення з методу `tsk.Wait()`.

```
//Очікуати виконання задачі Lambda_expr
tsk1.Wait();

//Вивільнити задачу Lambda_expr
tsk1.Dispose();
```

Створення продовження задачі. Продовження - це одна задача, яка автоматично починається після завершення другої задачі. Створити продовження можна, зокрема, за допомогою методу `ContinueWith()`, що визначений у класі `Task`.

```
public Task ContinueWith(Action<Task> дія_продовження)
```

де `дія_продовження` означає задачу, яка буде запущена на виконання після закінчення задачі, що викликає.

```
//Сконструюввати об'єкт першої задачі
Task tsk=new Task(MyTask);

//Сконструюввати продовження задачі
Task TaskCont=tsk.ContinueWith(ContTask);
```

В якості продовження задачі нерідко застосовується лямбда-вираз.

```
//Сконструюввати продовження задачі у вигляді Лямда-виразу
Task LambdaTaskCont = TaskCont.ContinueWith((first)=>
{
      Console.WriteLine("LambdaTaskCont is started.");
      for (int count = 0; count < 5; count++)
      {
            Thread.Sleep(500);
      Console.WriteLine("In the method LambdaTaskCont counter = " + count);
```

```
    }
    Console.WriteLine("LambdaTaskCont is done.");
});
```

У цьому фрагменті коду параметр `first` приймає попередню задачу (в даному випадку - `ContTask`).

Повернення значення із задачі. Задача може повертати значення. Це дуже зручно з двох причин. По-перше, це означає, що за допомогою задачі можна обчислити деякий результат. Подібним чином підтримуються паралельні обчислення. І по-друге, процес, що викликає, виявиться заблокованим до тих пір, поки не буде отримано результат. Це означає, що для організації очікування результату не потрібно ніякої особливої синхронізації.

Для того щоб повернути результат із задачі, досить створити цю задачу, використовуючи узагальнену форму `Task <TResult>` класу `Task`.

```
public Task(Func<TResult> функція)
public Task(Func<Object, TResult> функція, Object стан)
```

де функція означає делегат, що виконується. Тип `Func` використовується саме в тих випадках, коли задача повертає результат. У першому конструкторі створюється задача без аргументів, а в другому конструкторі - задача, що приймає аргумент типу `Object`, який передається як `стан`.

У наступній програмі створюються два методи. Перший із них, `MyTask()`, не приймає параметрів, а просто повертає логічне значення `true` типу `bool`. Другий метод, `Sumlt()`, приймає єдиний параметр, який приводиться до типу `int`, і повертає суму чисел, що менше заданого параметру.

```
namespace ReturnResult
{
    class Program
    {
        //Метод без аргументів, що повертає результат
        static bool MyTask()
        {
            return true;
        }

        //Метод, що повертає суму чисел, що менше заданого параметру
        static int SumIt(object v)
        {
            int x=(int) v;
            int sum=0;
            for (; x > 0; x--)
                sum += x;
            return sum;
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Main Thread is starting.");

            //Сконструюввати та запустити об'єкт першої задачі
            Task<bool> tsk1=Task<bool>.Factory.StartNew(MyTask);
            Console.WriteLine("The Result after running of MyTask =
"+tsk1.Result);

            //Сконструюввати та запустити об'єкт другої задачі
```

```
                Task<int> tsk2=Task<int>.Factory.StartNew(SumIt,5);
                Console.WriteLine("The Result after running of SumIt = " +
tsk2.Result);

                tsk1.Dispose();
                tsk2.Dispose();

                Console.WriteLine("Main() is done.");
                Console.ReadLine();
            }
      }
}
```

Результат виконання програми:

```
Main Thread is starting.
The Result after running of MyTask = True
The Result after running of SumIt = 15
Main() is done.
```

Розпаралелювання задач методом Invoke(). Метод `Invoke()`, визначений в класі `Parallel`, дозволяє виконувати один або кілька методів, що вказуються у вигляді його аргументів. Він також масштабує виконання коду, використовуючи доступні процесори, якщо є така можливість.

```
public static void Invoke(params Action[] actions)
```

Кожен метод, який передається методу `Invoke()` як аргумент, не повинен ні приймати параметрів, ні повертати значення. Завдяки тому що параметр `actions` даного методу відноситься до типу `params`, виконувані методи можуть бути вказані у вигляді змінного списку аргументів. Метод `Invoke()` спочатку ініціює виконання, а потім очікує завершення всіх переданих йому методів. Це, зокрема, позбавляє від необхідності (та й не дозволяє) викликати метод `Wait()`.

У наступній програмі два методи `MyMeth1()` і `MyMeth2()` виконуються паралельно за допомогою виклику методу `Invoke()`.

```
namespace Invoke_Par
{
      class Program
      {
            //Метод1, що виконується як задача
            static void MyMeth1()
            {
                Console.WriteLine("MyMeth1() is started.");
                for (int count = 0; count < 5; count++)
                {
                      Thread.Sleep(500);
                      Console.WriteLine("In the method MyMeth1() counter = " +
count);
                }
                Console.WriteLine("MyMeth1() is done.");
            }

            //Метод2, що виконується як задача
            static void MyMeth2()
            {
                Console.WriteLine("MyMeth2() is started.");
                for (int count = 0; count < 5; count++)
```

```
                {
                      Thread.Sleep(500);
                      Console.WriteLine("In the method MyMeth2() counter = " +
count);
                }
                Console.WriteLine("MyMeth2() is done.");
          }
          static void Main(string[] args)
          {
                Console.WriteLine("Main Thread is starting.");

                //Виконати паралельно два іменованих метода, метод Main
призупиняється, поки виконуються два методи
                Parallel.Invoke(MyMeth1, MyMeth2);

                Console.WriteLine("Main() is done.");
                Console.ReadLine();
          }
     }
}
```

Завдання

1. Створити програму, що створює дві задачі, які виконуються паралельно. Затримку методом `Sleep()` організувати на величину 200мс та пропорційно ідентифікатору задачі.
2. Організувати очікування виконання задач методом `WaitAll()`.
3. Визначити задачу для виконання у вигляді лямбда-виразу.
4. Створити програму паралельних обчислень за допомогою виклику методу `Invoke()`, де в якості аргументів застосовуються лямбда-вирази.