Лекція 6

БАГАТОПОТОКОВЕ, ПАРАЛЕЛЬНЕ ТА АСИНХРОННЕ ПРОГРАМУВАННЯ

Лекція 6. Багатопотокове, паралельне та асинхронне програмування

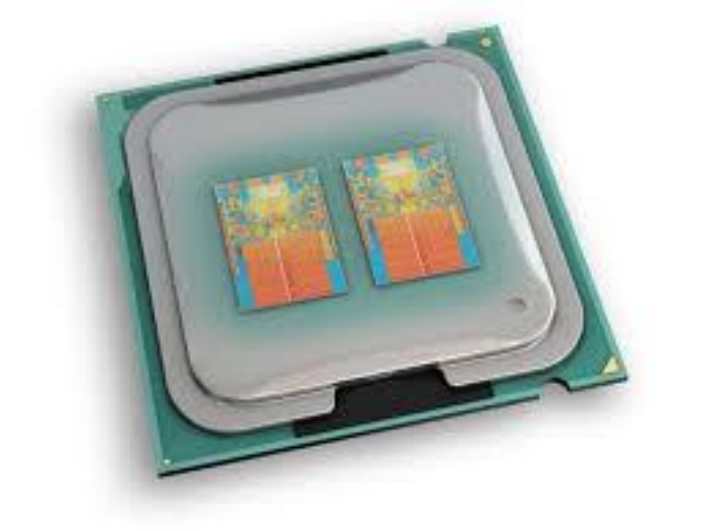
План

1. Загальне поняття багатопоточності та паралелізму.
2. Проблеми паралелізму.
3. Асинхронні виклики.
4. Простір імен System.Threading.
5. Клас Thread.
6. Синхронізація потоків.

1. Загальне поняття багатопоточності та паралелізму

Під **паралелізмом** розуміється одночасне виконання двох чи більше операцій. У контексті комп'ютера це означає, що система виконує кілька незалежних операцій **паралельно**, тобто одночасно, а не послідовно.

Створення і поширення комп'ютерів, обладнаних кількома процесорами чи кількома ядрами на одному кристалі (**багатоядерними процесорами**), призвело до необхідності розробки програм, які підтримують паралелізм.



На сьогодні виробники апаратного забезпечення вважають за краще не підвищувати тактову частоту процесора (його швидкодію), а збільшувати кількість його ядер. Тобто **для підвищення швидкодії потрібно створювати багатопоточне програмне забезпечення.**

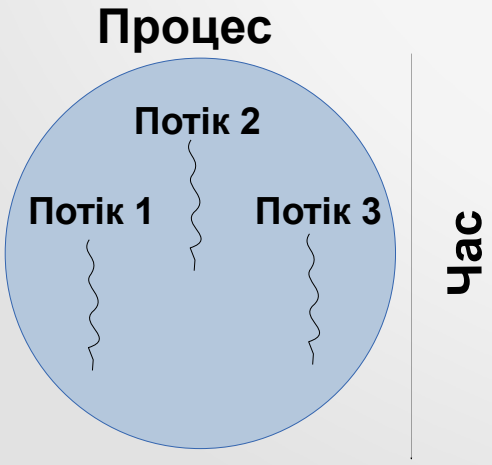
ВАЖЛИВО! Сучасні процесори навіть за наявності одного ядра можуть одночасно виконувати кілька команд. Це називається апаратним паралелізмом

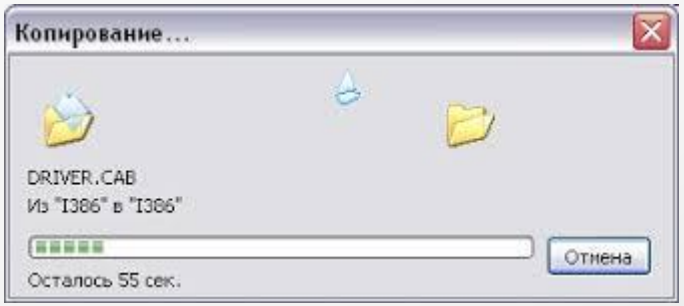
1. Загальне поняття багатопоточності та паралелізму

Будь-який **обчислювальний процес** у Windows складається щонайменше з одного потоку виконання.

Потік (**thread**) – це одна з дій усередині процесу, що є найменшою одиницею обробки, виконання якої може бути призначено **планувальником** операційної системи. Інакше кажучи, потік – це окремий шлях виконання програмного коду всередині застосунку (процесу), що виконується.

Будь-який процес може бути як **одно-** , так і **багатопотоковим**.



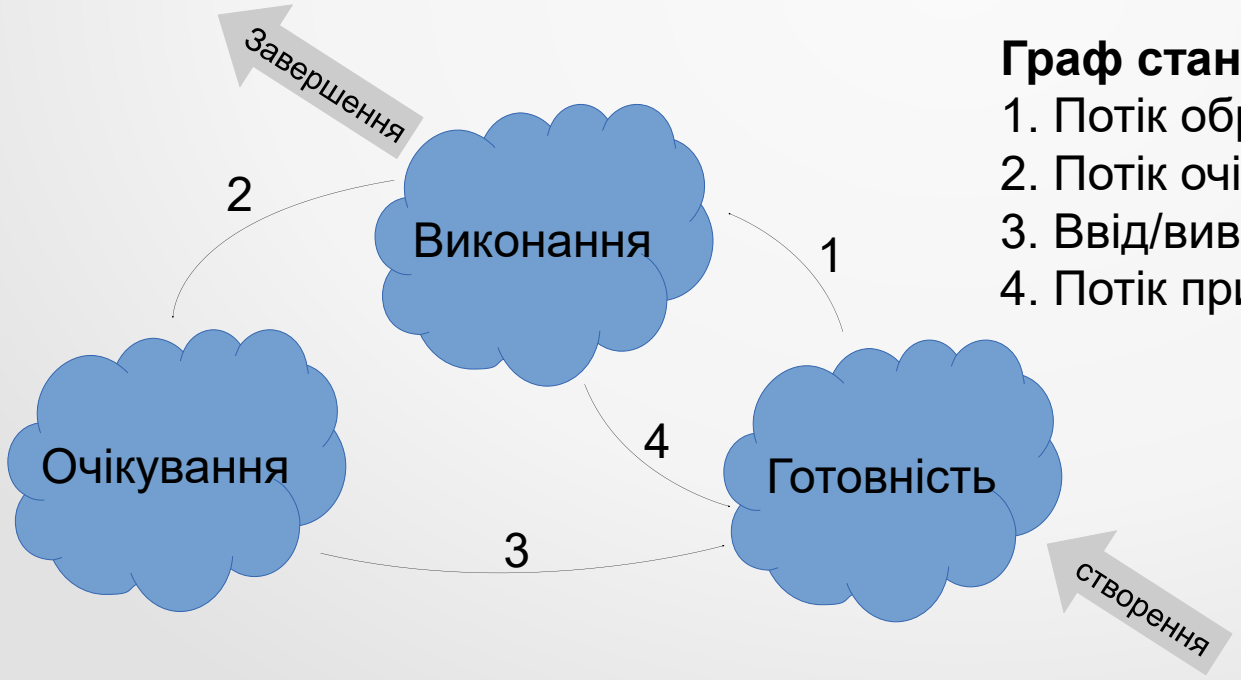


Приклад багатопотокової програми: один потік копіює, інший виконує анімацію.

1. Загальне поняття багатопоточності та паралелізму

Створення багатопотокових програм дозволяє досягти реального **паралелізму** в роботі комп'ютера, обладнаного кількома **процесорами** (або **багатоядерним** процесором).

Навіть на **однопроцесорному** комп'ютері використання багатопотокових програм дозволяє підвищити загальну **реактивність** системи за рахунок можливості обходу **блокувань** (додаток, що очікує ввід/вивід, блокується системою, але використання кількох потоків у ньому дозволяє реагувати на команди користувача в інших незаблокованих потоках).



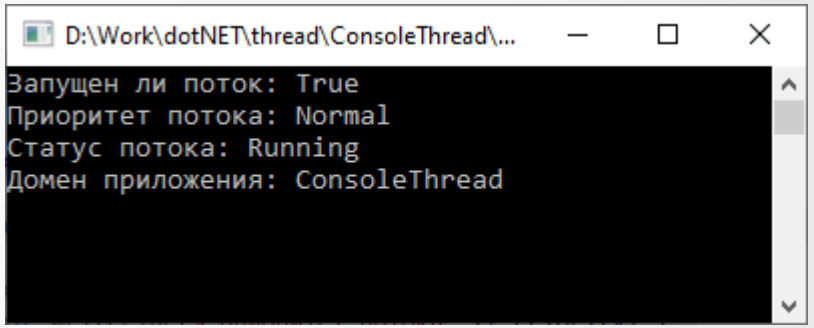
Граф станів потоку:

1. Потік обрано для виконання.
2. Потік очікує на ввід/вивід.
3. Ввід/вивід завершено.
4. Потік призупинено планувальником.

1. Загальне поняття багатопоточності та паралелізму

У .NET **первинний потік виконання** (створений середовищем CLR під час виконання методу Main()) у будь-який час може створити необхідну кількість **вторинних потоків** для виконання додаткових одиниць роботи. Щоб отримати посилання на поточний потік, можна звернутися до статичної властивості **Thread.CurrentThread** класу **Thread**:

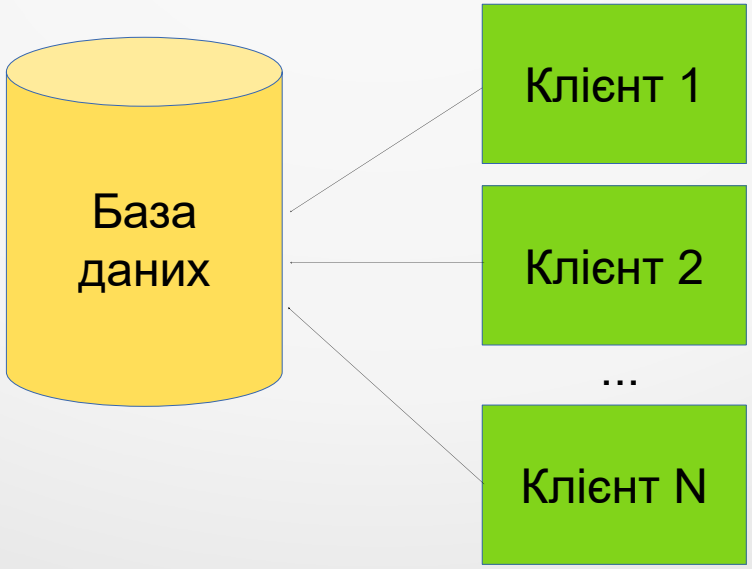
```
using System;
використовуючи System.Threading;

namespace ConsoleThread
{
   class Program
   {
      static void Main(string[] args)
      {
         Thread t = Thread.CurrentThread; // Отримання поточного потоку
         Console.WriteLine($"Чи запущений потік: {t.IsAlive}");
         Console.WriteLine($"Пріоритет потоку: {t.Priority}");
         Console.WriteLine($"Статус потоку: {t.ThreadState}");
         Console.WriteLine($"Домен програми: {Thread.GetDomain().FriendlyName}");
         Console.ReadLine();
      }
   }
}
```



2. Проблеми паралелізму

Конкурентністю називається можливість виконання кількох потоків у періоди часу, які перекриваються, що може призводити до **стану гонитви** (race condition) – проблеми **синхронізації** доступу поттоків до деяких спільних ресурсів, наприклад даних (гонитва за даними).

Наприклад, якщо потік А ще не завершився, а потік В вже намагається оперувати його даними (які ще можливо не готові), то в цьому випадку поведінка потоків (і процесу в цілому) може стати непередбачуваною і може призводити до дуже **нетривіальних помилок** .



2. Проблеми паралелізму

Розглянемо такий приклад. Нехай у потоці виконується оператор інкременту мови C#:

x++; // Збільшення значення цілочислової змінної x на 1

Байт-код , що реалізує даний оператор, можна описати, наприклад, так:

load x into register
add 1 to register
store register in x

При виконанні цих операцій може виникнути гонитва. Нехай, x = 5.

Крок	Потік 1	Потік 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3	store register in x		6	6
4		load x into register	6	6
5		add 1 to register	6	7
6		store register in x	7	7

2. Проблеми паралелізму

Так також припустимо:

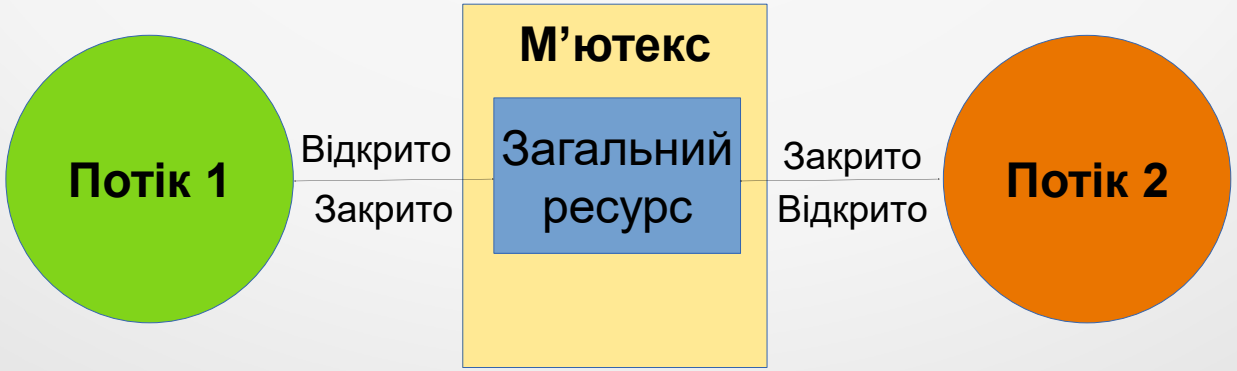
Крок	Потік 1	Потік 2	x	register
1		load x into register	5	5
2		add 1 to register	5	6
3		store register in x	6	6
4	load x into register		6	6
5	add 1 to register		6	7
6	store register in x		7	7

А ось так уже ні:

Крок	Потік 1	Потік 2	x	register
1	load x into register		5	5
2	add 1 to register		5	6
3		load x into register	5	5
4	store register in x		5	5
5		add 1 to register	5	6
6	store register in x		6	6

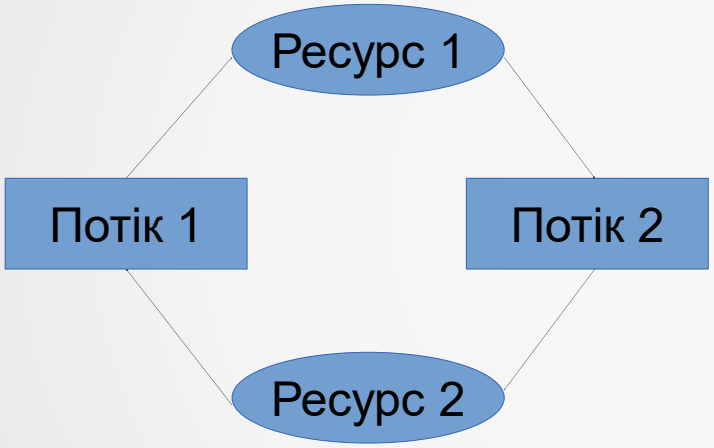
2. Проблеми паралелізму

Для усунення цієї проблеми (тобто синхронізації) потоки використовують так звані **м’ютекси**, під якими розуміються спеціальні об'єкти (**семафори**), що мають можливість приймати два значення (наприклад, відкрито/закрито). Якщо потік звертається до м’ютексу і останній має значення «відкрито», він встановлює його значення у «закрито» і може монопольно використовувати **критичну область коду**. Всі інші потоки, які звертаються до цієї області коду, блокуються. Тобто одночасно лише один потік може володіти м’ютексом. Після завершення своєї роботи з критичною областю, потік, що заблокував м’ютекс, встановлює його значення у «відкрито», після чого інші потоки можуть отримати до нього (а, відповідно, і до критичної області) доступ.



2. Проблеми паралелізму

На жаль, використання блокувань не завжди вирішує проблему синхронізації потоків, тому що на практиці при реалізації багатопотокових додатків можуть виникати так звані **взаємні блокування** (**клінчі**).



Взаємне блокування двох потоків, які потребують двох ресурсів

Взаємне блокування – це ситуація, коли два потоки очікують на закінчення роботи один одного і, таким чином, жоден з них не може закінчитися. За наявності м'ютексів взаємне блокування відбувається, коли потоку А потрібен м'ютекс, яким володіє потік В, і навпаки.

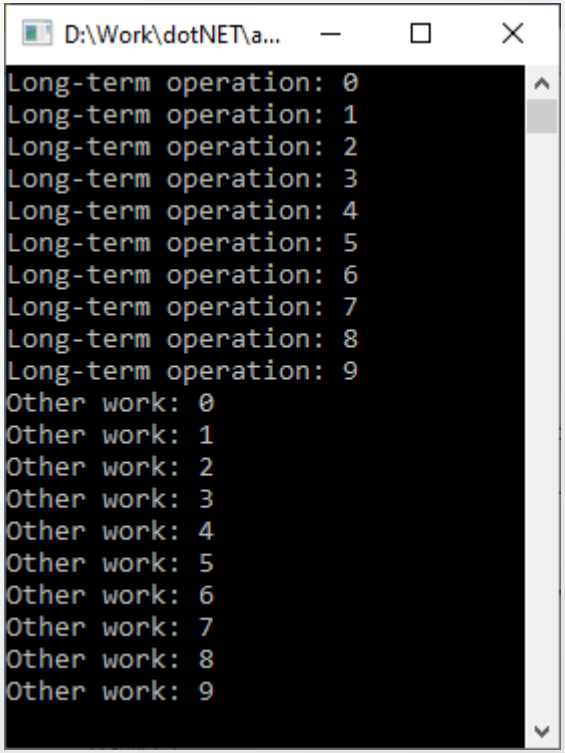
ВАЖЛИВО! Для профілактики клінчів потрібно правильно проектувати послідовність роботи потоків програми.

3. Асинхронні виклики

Якщо однопотокова програма виконує якусь тривалу операцію, то доки вона не закінчиться, програма не зможе виконувати жодних інших завдань (оновлювати свій інтерфейс, реагувати на команди користувача і т.п.).

```
using System;
namespace ConsoleThread
{
   class Program
   {
      static void Main(string[] args)
      {
         DoLongWork(); // Викликати тривалу процедуру
         // Виконати іншу роботу в первинному потоці...
         for (int i = 0; i < 10; i++)
            Console.WriteLine("Other work: {0}", i);
         Console.ReadLine();
      }
      static void DoLongWork()
      {
         for (int i = 0; i < 10; i++)
            Console.WriteLine("Long-term operation: {0}", i);
      }
   }
}
```

Поки не закінчиться тривала операція, програма не може виконувати іншу роботу



3. Асинхронні виклики

Для виправлення цієї ситуації використовується механізм **асинхронності** – можливість незалежного (асинхронного) виконання різних частин програмного коду.

Для асинхронного запуску методу в .NET використовуються дві функції – **BeginInvoke()** та **EndInvoke()**.

Асинхронний виклик ініціюється за допомогою BeginInvoke(). Він має ті самі параметри, що й метод, який потрібно виконати асинхронно, а також два додаткові необов'язкові параметри. Перший параметр є делегатом **AsyncCallback**, який посилається на метод, що викликається після завершення асинхронного виклику. Другий параметр – об'єкт, що визначається користувачем, який передає дані в метод зворотного виклику.

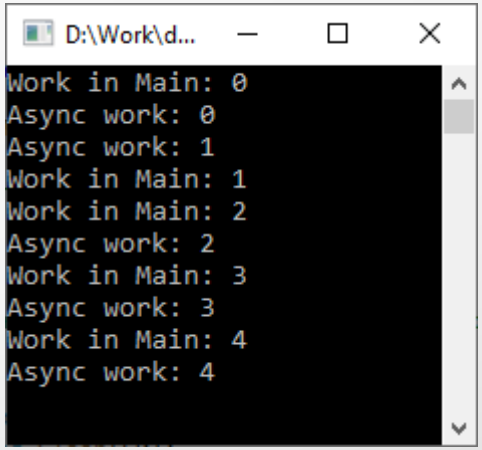
Метод EndInvoke() отримує результати асинхронного виклику. Його можна викликати будь-коли після виклику методу BeginInvoke(). Якщо асинхронний виклик не завершено, метод EndInvoke() блокує потік, що викликає, до завершення виклику.

Примітка. Асинхронні виклики делегатів, зокрема методи BeginInvoke() та EndInvoke(), не підтримуються у платформі .NET Compact Framework.

3. Асинхронні виклики

Приклад реалізації асинхронності:

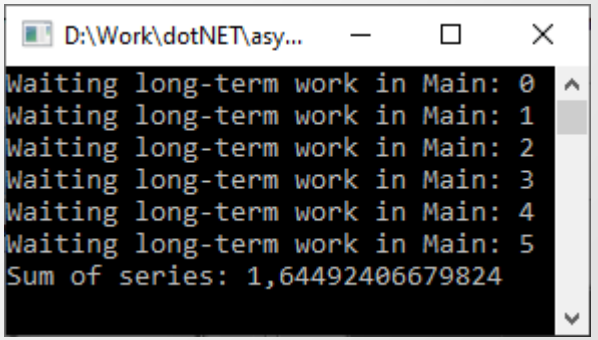
```
using System;
using System.Threading;

class Program
{
   public delegate void AsyncDelegate();
   static void Main(string[] args)
   {
      // Створення делегата...
      var pd = new AsyncDelegate(AsyncWork);
      // Асинхронний виклик методу
      var ar = pd.BeginInvoke(null, null);
      // Виконати іншу роботу в первинному потоці...
      for (int i = 0; i < 5; i++)
      {
         Console.WriteLine("Work in Main: {0}", i);
         Thread.Sleep(50);
      }
      // Обробка асинхронного результату
      pd.EndInvoke(ar);
      Console.ReadLine();
   }

   static void AsyncWork()
   {
      for (int i = 0; i < 5; i++)
      {
         Console.WriteLine("Async work: {0}", i);
         Thread.Sleep(50);
      }
   }
}
```



3. Асинхронні виклики

Приклад очікування на результат асинхронної процедури:

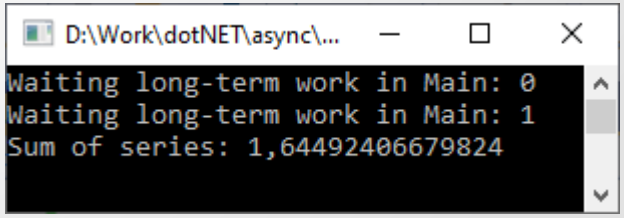
```
using System;
using System.Threading;

namespace ConsoleThread
{
   class Program
   {
      public delegate double AsyncDelegate(int n);
      static void Main(string[] args)
      {
         // Викликати Add() у вторинному потоці.
         var pd = new AsyncDelegate(CalcSeries);
         var ar = pd.BeginInvoke(100000, null, null);
         int i = 0;

         // Виконувати іншу роботу у первинному потоці,
         // поки не завершиться тривала асинхронна процедура
         while (!ar.IsCompleted)
            Console.WriteLine("Waiting long-term work in Main: {0}", i++);
         // Виведення результату асинхронної процедури
         Console.WriteLine("Sum of series: {0}", pd.EndInvoke(ar));
         Console.ReadLine();
      }

      static double CalcSeries(int n)
      {
         double Sum = 0;

         for (double i = 1; i <(double)n; i++)
            Sum += 1.0/(i*i);
         return Sum;
      }
   }
}
```



3. Асинхронні виклики

Ще один приклад очікування завершення асинхронної процедури:

```
using System;

class Program
{
   private static bool IsAsyncDone = false;
   public delegate double AsyncDelegate(int n);
   static void Main(string[] args)
   {
      // Викликати Add() у вторинному потоці.
      var pd = new AsyncDelegate(CalcSeries);
      var ar = pd.BeginInvoke(100000,
                  new AsyncCallback (WorkDone), null);
      int i = 0;

      // Виконати іншу роботу в первинному потоці...
      while (! IsAsyncDone)
         Console.WriteLine("Waiting long-term work in Main: {0}", i++);
      Console.WriteLine("Sum of series: {0}", pd.EndInvoke(ar));
      Console.ReadLine();
   }
   static double CalcSeries(int n)
   {
      double Sum = 0;

      for (double i = 1; i <(double)n; i++)
         Sum += 1.0/(i*i);
      return Sum;
   }
   static void WorkDone(IAsyncResult ar)
   {
      IsAsyncDone = true;
   }
}
```



4. Простір імен System.Threading

У просторі імен **System.Threading** платформи .NET реалізовано низку типів, призначених для розробки багатопотокових додатків.

Таблиця 1. Ключові типи простору імен System.Threading

Тип	Опис
Interlocked	Реалізує атомарні операції для змінних, що поділяються між кількома потоками
Monitor	Забезпечує синхронізацію потокових об'єктів із використанням блокування та очікування/сигналів
Mutex	Реалізація м'ютексу
ParameterizedThreadStart	Делегат, що дозволяє потоку викликати методи, які приймають довільну кількість аргументів.
Semaphore	Дозволяє обмежити кількість потоків, що мають доступ до ресурсів
Thread	Інкапсулює потік, що виконується середовищем CLR

4. Простір імен System.Threading

Таблиця 1 (продовження)

Тип	Опис
ThreadPool	Дозволяє взаємодіяти з підтримуваним CLR пулом потоків усередині заданого процесу
ThreadPriority	Перелік, що визначає пріоритети потоків (Highest, Normal тощо)
ThreadStart	Делегат, що дозволяє встановити метод для виклику в заданому потоці (на відміну від ParametrizedThreadStart, методи ThreadStart завжди повинні мати один і той же прототип)
ThreadState	Перелік, що задає допустимі стани потоку (Running, Aborted, …)
Timer	Надає механізм виконання методу через вказані інтервали
TimerCallback	Делегат, що використовується у поєднанні з Timer

5. Клас Thread

Для створення та управління потоками використовується клас **Thread**.

Таблиця 2. Основні властивості та методи класу Thread

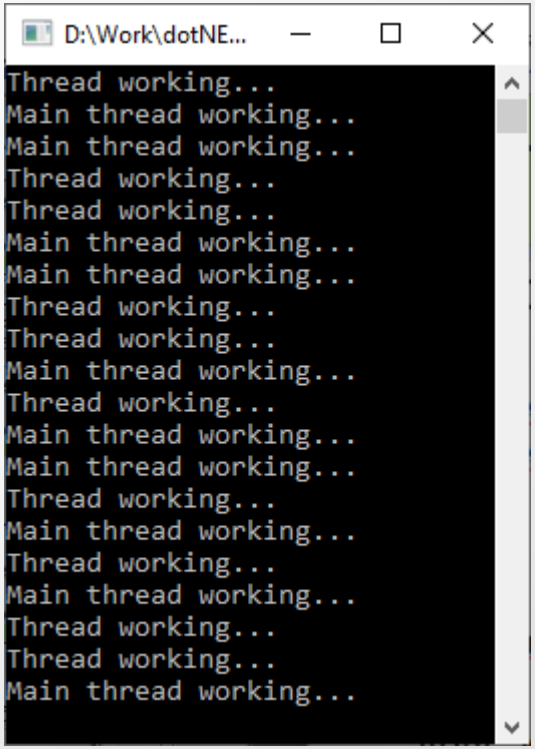
Член класу	Призначення
CurrentThread	Посилання на поточний потік
IsAlive	Ознака, чи запущено потік
IsBackground	Ознака, чи потік є фоновим
Priority	Пріоритет потоку (визначається у перерахуванні ThreadPriority)
ThreadState	Стан потоку (визначається у ThreadState)
Sleep()	Зупиняє поточний потік на вказаний час
Abort()	Припиняє виконання потоку, як тільки це буде можливо
Interrupt()	Зупиняє потік на заданий період очікування
Join()	Блокує батьківський потік до завершення зазначеного потоку
Resume()	Відновлює раніше зупинений потік
Start()	Запускає потік
Suspend()	Припиняє потік

5. Клас Thread

Приклад створення потоку:

```
using System;
using System.Threading;

namespace ConsoleThread
{
   class Program
   {
      static void Main(string[] args)
      {
         Thread thr = new
                      Thread(new ThreadStart(doWork));

         thr.Start();
         for (int i = 0; i < 10; i++)
         {
            Console.WriteLine("Main thread working...");
            Thread.Sleep(50);
         }
         Console.ReadKey();
      }
   }
}
```

```
public static void doWork()
{
   for (int i = 0; i < 10; i++)
   {
      Console.WriteLine("Thread working...");
      Thread.Sleep(50);
   }
}
```



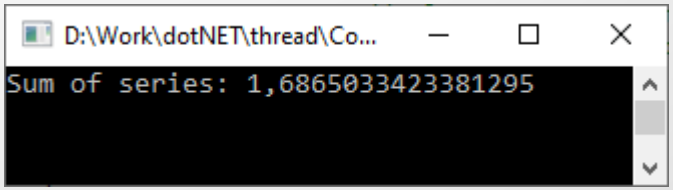
5. Клас Thread

Приклад багатопотокового обчислення суми ряду:

```
using System;
using System.Threading;
namespace ConsoleThread
{
   // Клас передачі параметрів в потік
   class SeriesParams
   {
      public int begin, end;
      public SeriesParams(int b, int e)
      {
         begin = b;
         end = e;
      }
   }
   class Program
   {
      public static int NumThread = 8; // Кількість потоків
      public static double Sum = 0; // Підсумкова сума
      static void Main(string[] args)
      {
         Thread [] thr = new Thread [NumThread];
         var MaxIter = 1000000; // Загальна кількість ітерацій
         var step = MaxIter/NumThread; // Кількість ітерацій у потоці
```

5. Клас Thread

```
        for (int i = 0; i < NumThread; i++) // Запуск потоків
        {
            thr[i] = new Thread(new ParameterizedThreadStart(CalcSeries)); // Створення і-го потоку
            // Запуск потоку та передача в нього параметрів
            thr[i].Start(new SeriesParams(i * step, (i == NumThread - 1) ? MaxIter : (i + 1) * step));
        }
        for (int i = 0; i < NumThread; i++) // Очікування завершення потоків
            thr [i].Join ();
        Console.WriteLine("Sum of series: {0}", Sum);
        Console.ReadKey();
    }
    // Обчислення суми ряду для заданого діапазону ітерацій
    public static void CalcSeries(object param)
    {
        double sum = 0;
        if (param is SeriesParams)
        {
            for (double i = ((SeriesParams)param).begin; i < ((SeriesParams)param).end; i++)
                sum + = (1.0 / (1 + i * i * i));
            Sum + = sum;
        }
    }
  }
}
```



5. Клас Thread

Для визначення часу роботи програми був використаний клас **Stopwatch** із простору імен **System.Diagnostics**.

Визначення часу обчислення ряду в єдиному потоці має такий вигляд:

```
Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();
CalcSeries(new SeriesParams(0, MaxIter));
stopwatch.Stop();

Console.WriteLine("Milliseconds: {0}",
stopwatch.ElapsedMilliseconds);
```

Залежність часу обчислення суми ряду від кількості використаних потоків

6. Синхронізація потоків

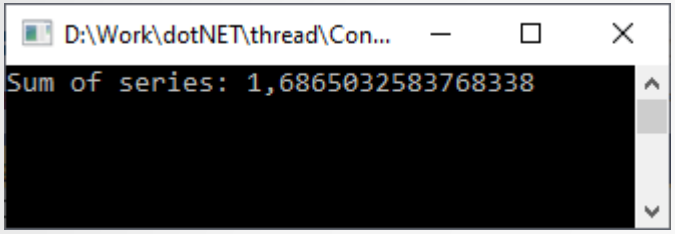
Для запобігання **гонкам** і **синхронізації** потоків у .NET найчастіше використовують **м'ютекси**. Розглянемо наступний приклад. Змінимо код методу CalcSeries() наступним чином:

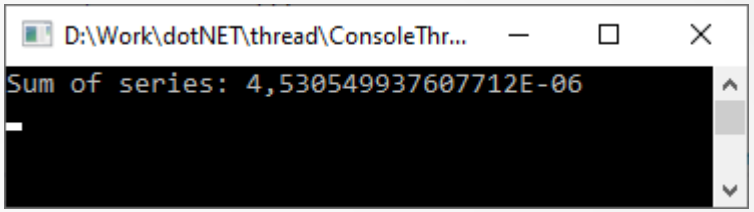
```
public static void CalcSeries(object param)
{
   double sum = 0; // Локальна змінна потоку
   if (param is SeriesParams)
   {
      for (double i = ((SeriesParams)param).begin; i < ((SeriesParams)param).end; i++)
         sum + = (1.0 / (1 + i * i * i)); // На кожній ітерації змінюється локальна змінна
      Sum + = sum;
   }
}
```

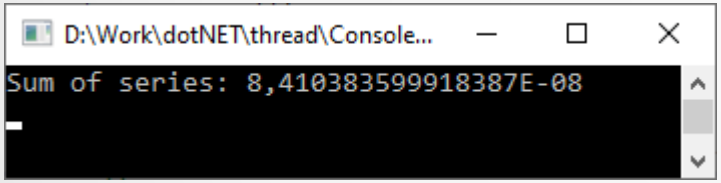
```
public static void CalcSeries(object param)
{
   if (param is SeriesParams)
   {
      for (double i = ((SeriesParams)param).begin; i < ((SeriesParams)param).end; i++)
         Sum + = (1.0 / (1 + i * i * i)); // На кожній ітерації в потоці змінюється властивість класу
   }
}
```

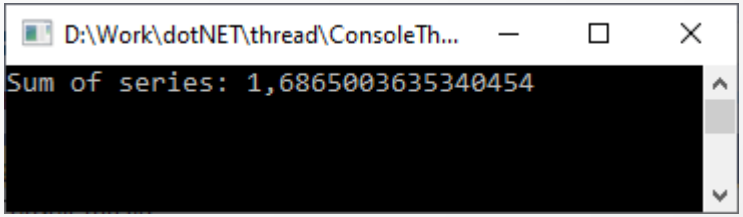
6. Синхронізація потоків

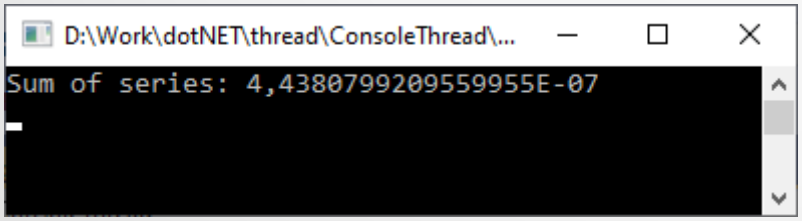
Серія запусків програми призводить до різних результатів:

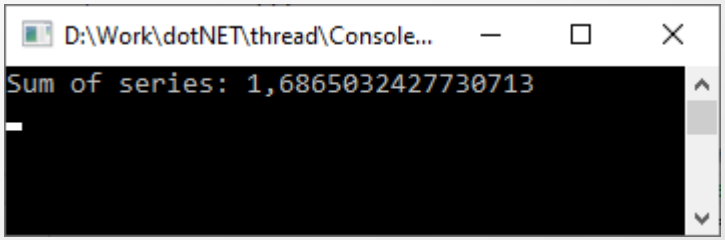












6. Синхронізація потоків

Ясно, що до такого розкиду результатів призводить гонка за ресурс – підсумкову суму ряду **Sum** (властивість класу **Program**).

Для усунення цієї проблеми використаємо м'ютекс наступним чином:

```
class Program
{
   public static Mutex mutex = new Mutex (); // Створення мьютексу
   public static int NumThread = 10; // Кількість потоків
   // …
   public static void CalcSeries(object param)
   {
      double sum = 0;

      if (param is SeriesParams)
      {
         for (double i = ((SeriesParams)param).begin; i < ((SeriesParams)param).end; i++)
            sum + = (1.0 / (1 + i * i * i));
         mutex.WaitOne(); // Закрити м'ютекс
         Sum + = sum; // Змінити ресурс, за який може виникнути гонка
         mutex.ReleaseMutex(); // Звільнити м'ютекс
      }
   }
}
```