# Лекція 5. Синхронізація потоків

## 1. Засоби синхронізації потоків

При використанні декількох потоків буває потрібно координувати їхні дії. Процес досягнення такої координації називається **синхронізацією**. Найпоширенішою причиною використання синхронізації є необхідність розділяти між потоками спільний ресурс, який може бути одночасно доступний лише одному потоку. Наприклад, коли в одному потоці виконується запис інформації у файл (базу даних), другому потоку слід заборонити робити це в той самий момент часу. Синхронізація потрібна і в тому випадку, якщо один потік чекає подію, що викликається іншим потоком. Тобто, потоки не є незалежними. У подібній ситуації потрібні якісь засоби, що дозволяють призупинити один з потоків до тих пір, поки не станеться подія в іншому потоці. Після цього очікуючий потік може відновити своє виконання.

Середовище .NET Framework надає широкий набір засобів синхронізації. Більшість з них, зокрема блокування, монітор, м’ютекс, семафр, є об’єктною оболонкою над одноіменними засобами операційної системи.

**Таблиця 1. Засоби синхронізації**

| Тип синхронізації | Реалізація |
|---|---|
| Блокування | Join, Sleep, SpinWait |
| Взаємовиключний доступ | Lock, Monitor, Mutex, SpinLock |
| Сигнальні повідомлення | AutoResetEvent, ManualResetEvent, ManualResetEventSlim |
| Семафори | Semaphore, SemaphoreSlim |
| Атомарні оператори | Interlocked |
| Конкурентні колекції | ConcurrentBag, ConcurrentQueue, ConcurrentDictionary, ConcurrentStack, BlockedCollection |
| Блокування введення-виведення | ReaderWriterLock, ReaderWriterLockSlim |
| Шаблони синхронізації | Barrier, CountdownEvent |

Розглянемо ці засоби більш детально.

## 2. Блокування

В основу синхронізації покладено поняття **блокування**, за допомогою якого організується управління доступом до блоку коду в об'єкті. Коли об'єкт заблокований одним потоком, інші потоки не можуть дістати доступ до заблокованого блоку коду. Коли ж блокування знімається одним потоком, об'єкт стає доступним для використання в іншому потоці.

Очікування може бути активним або пасивним. При активному "очікуванні" потік циклічно перевіряє стан очікуваної події.

```
Thread thr = new Thread(SomeWork);
thr.Start();
```

```
while(thr.IsAlive) ;
```

Таке блокування називається *активним*, оскільки фактично потік не припиняє своєї роботи і не звільняє процесорний час для інших потоків. Активне очікування ефективне тільки при незначному часі очікування.

*Пасивне очікування* реалізується за допомогою операційної системи, яка зберігає контекст потоку і вивантажує його, надаючи можливість виконуватися іншим потокам. При настанні очікуваної події операційна система "будить" потік - завантажує контекст потоку і виділяє йому процесорний час. Пасивне очікування вимагає часу на збереження контексту потоку при блокуванні і завантаження контексту при відновленні роботи потоку, але дозволяє використовувати обчислювальні ресурси під час очікування для виконання інших завдань. Пасивне очікування реалізується за допомогою методів **Sleep() і Join().**

У наступному фрагменті використовуються два типи очікування (активне і пасивне). У першому випадку застосовується циклічна перевірка стану потоку. У другому випадку використовується метод Join.

**Приклад 1. Активне і пасивне очікування**

```
using System;
using System.Threading;

namespace Lab4_Parallel
{
    class Program
    {
        static bool b;
        static double res;
        static void SomeWork()
        {
            for (int i = 0; i < 100000; i++)
                for (int j = 0; j < 20; j++)
                    res += Math.Pow(i, 1.33);
            b = true;
        }
        static void Main()
        {
            Thread thr1 = new Thread(SomeWork);
            thr1.Start();
            // Активне очікування в циклі
            while (!b);
            Console.WriteLine("Result = " + res);
            res = 0;
            Thread thr2 = new Thread(SomeWork);
            thr2.Start();
            // Пасивне очікування
            thr2.Join();
            Console.ReadKey();
        }
      }
```

Існують гібридні засоби синхронізації, що поєднують в собі переваги активного і пасивного очікування. Гібридне блокування використовують об'єкти синхронізації, введені в .NET 4.0: SpinWait, SpinLock, SemaphoreSlim, ManualResetEventSlim, ReaderWriteLockSlim та ін. Потоки, що блокуються за допомогою гібридних засобів синхронізації, на початку фази очікування знаходяться в активному стані - не вивантажуються, циклічно перевіряють статус очікуваної події. Якщо очікування затягується, то активне блокування стає не ефективним і операційна система вивантажує очікуючий потік.

## 3. Взаємовиключний доступ до блоку коду за допомогою оператора lock

### Поняття про критичну секцію

Одне з основних призначень засобів синхронізації полягає в організації взаємовиключного доступу до ресурсу, що розділяється. Зміни спільних даних одним потоком не повинні перериватися іншими потоками. Фрагмент коду, в якому здійснюється робота з ресурсом, що розділяється, і який повинен виконуватися тільки в одному потоці одночасно, називається **критичною секцією**. Це може бути весь метод, або його частина.

Така синхронізація організовується за допомогою ключового слова **lock** і також називається *блокуванням*, але стосується синхронізації даних.

Коли один потік входить в **критичну секцію** (захоплює об'єкт синхронізації), інший потік чекає завершення усього блоку (звільнення об'єкту синхронізації). Якщо заблоковано було декілька потоків, то при звільненні критичної секції тільки один потік розблоковується і входить у критичну секцію.

Для виділення критичної секції за допомогою оператора **lock** необхідно вказати об'єкт синхронізації, який виступає ідентифікатором блокування.

Нижче наведена загальна форма блокування:

```
lock(lockObj) {
// оператори, що синхронізуються
}
```

де **lockObj** позначає посилання на об'єкт, що синхронізується. Якщо ж потрібно синхронізувати лише один оператор, то фігурні дужки не потрібні. Оператор **lock** гарантує, що фрагмент коду, захищений для цього об'єкту, буде використовуватися лише в потоці, який одержав це блокування. А всі інші потоки блокуються до тих пір, поки блокування не буде знято. Блокування знімається після закінчення фрагмента коду, що захищається ним.

Заблокованим вважається такий об'єкт, який представляє ресурс, що синхронізується. В деяких випадках ним є екземпляр самого ресурсу або довільний екземпляр об'єкту, використовуваного для синхронізації.

При цьому, об'єкт, що блокується, не може бути відкритим (public), оскільки він може бути заблокований з іншого, неконтрольованого в програмі фрагмента коду і надалі взагалі не розблокується.

Розглянемо приклад управління доступом до методу Sumlt(), що підсумовує елементи масиву цілих чисел.

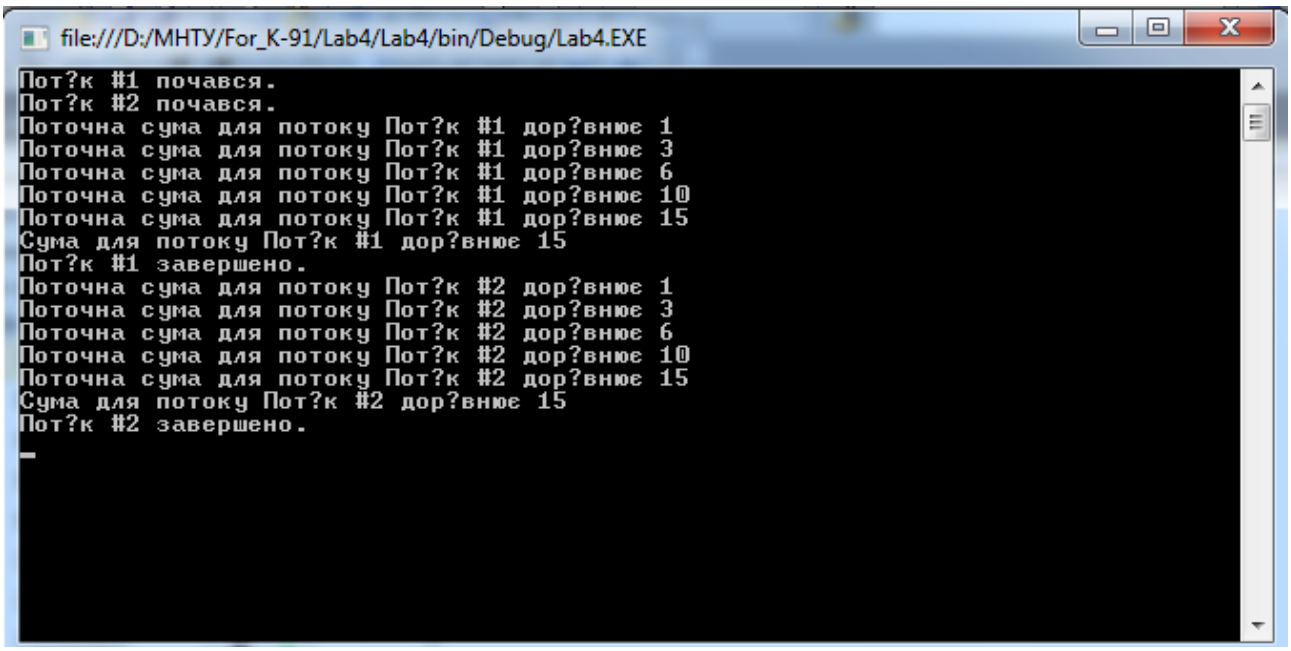
**Приклад 2.**

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

namespace Lab4_1
{
    class SumArray
    {
        int sum;
        object lockOn = new object(); // закритий об'єкт, доступний
        // для подальшого блокування
        public int Sumlt(int[] nums)
        {
            lock (lockOn)
            { // заблокувати весь метод
                sum = 0; // встановити початкове значення суми
                for (int i = 0; i < nums.Length; i++)
                {
                    sum += nums[i];
                    Console.WriteLine("Поточна сума для потоку " +
                    Thread.CurrentThread.Name + " равна " + sum);
                    Thread.Sleep(10); // дозволити перемикання завдань
                }
                return sum;
            }
        }
    }
    //---------------
    class MyThread
    {
        public Thread Thrd;
        int[] a;
        int answer;
        // Створити один об'єкт типа SumArray для всіх
        // екземплярів класу MyThread.
        static SumArray sa = new SumArray();
        // Сконструювати новий потік.
        public MyThread(string name, int[] nums)
        {
            a = nums;
            Thrd = new Thread(this.Run);
            Thrd.Name = name;
            Thrd.Start(); // почати потік
        }
        // Почати виконання нового потоку
        void Run()
        {
            Console.WriteLine(Thrd.Name + " почався.");
            answer = sa.Sumlt(a);
            Console.WriteLine("Сума для потоку " + Thrd.Name + " дорівнює " +
answer);
            Console.WriteLine(Thrd.Name + " завершено.");
        }
    }
```

```
    //---
    class Sync
    {
        static void Main()
        {
            int[] a = { 1, 2, 3, 4, 5 };
            MyThread mt1 = new MyThread("Потік #1", a);
            MyThread mt2 = new MyThread("Потік #2", a);
            mt1.Thrd.Join();
            mt2.Thrd.Join();
            Console.ReadKey();
        }

    }
}
```




**Правильні результати роботи потоків**

В обох потоках правильно підраховується сума, рівна 15.

Розглянемо цю програму детальніше. Спочатку в ній створюються три класи. Першим з них є клас SumArray, в якому визначається метод Sumlt (), що підсумовує елементи масиву. Другим створюється клас MyThread, в якому використовується статичний об'єкт **sa** типу SumArray. Отже, єдиний об'єкт типу SumArray використовується всіма об'єктами типа MyThread. За допомогою цього об'єкту виходить сума елементів цілочисельного масиву. Зверніть увагу на те, що поточна сума запам'ятовується в полі **sum** об'єкту типа SumArray. Тому якщо метод Sumlt () використовується паралельно в двох потоках, то обидва потоки спробують звернутися до поля sum, щоб зберегти в ньому поточну суму. А оскільки це може призвести до помилок, доступ до методу Sumlt () має бути синхронізований. І нарешті, в третьому класі, Sync, створюються два потоки, в яких підраховується сума елементів масиву.

Оператор **lock** в методі Sumlt () перешкоджає одночасному використанню даного методу в різних потоках. Зверніть увагу на те, що в операторі **lock**

синхронізується об'єкт lockOn. Це закритий об'єкт, призначений виключно для синхронізації. Метод Sleep () навмисно викликається для того, щоб сталося перемикання завдань, хоча в даному випадку це неможливо. Код в методі Sumlt() заблокований, і тому він може бути одночасно використаний лише в одному потоці. Таким чином, коли починає виконуватися другий породжений потік, він не зможе увійти до методу Sumlt () до тих пір, поки з нього не вийде перший породжений потік. Завдяки цьому гарантується отримання правильного результату.

Для того, щоб повністю з'ясувати принцип дії взаємовиключного блокування, спробуйте видалити з програми в методі Sumlt () блокування. У результаті метод Sumlt () перестане бути синхронізованим, а отже, зможе паралельно використовуватися в будь-яких потоках для одного і того ж об'єкту.

Оскільки поточна сума зберігається в полі sum, вона може бути змінена в кожному потоці, що викликає метод Sumlt (). Це означає, що якщо два потоки одночасно викликають метод Sumlt () для одного і того ж об'єкту, то кінцевий результат буде невірним, оскільки вміст поля sum відображає змішаний результат підсумовування в обох потоках.

Як приклад нижче наведений результат виконання цієї програми після зняття блокування з методу Sumlt ().

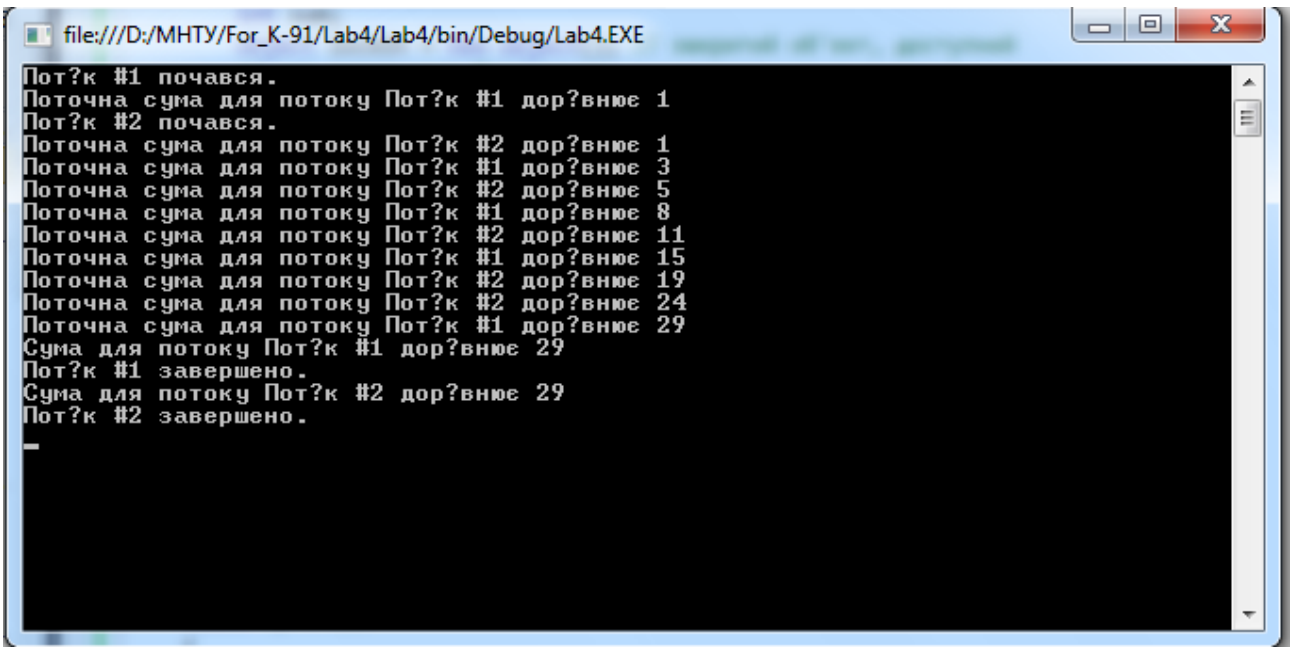
**Приклад 3.**

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

namespace Lab4_1
{
    class SumArray
    {
        int sum;
  //       object lockOn = new object(); // закритий об'єкт, доступний
        // для подальшого блокування
        public int Sumlt(int[] nums)
        {
 //           lock (lockOn)
            { // заблокувати весь метод
                sum = 0; // встановити початкова значення суми
                for (int i = 0; i < nums.Length; i++)
                {
                    sum += nums[i];
                    Console.WriteLine("Поточна сума для потоку " +
                    Thread.CurrentThread.Name + " равна " + sum);
                    Thread.Sleep(10); // дозволити перемикання завдань
                }
                return sum;
            }
        }
    }
    //---------------
    class MyThread
    {
        public Thread Thrd;
        int[] a;
```

```
        int answer;
        // Створити один об'єкт типа SumArray для всіх
        // екземплярів класу MyThread.
        static SumArray sa = new SumArray();
        // Сконструювати новий потік.
        public MyThread(string name, int[] nums)
        {
            a = nums;
            Thrd = new Thread(this.Run);
            Thrd.Name = name;
            Thrd.Start(); // почати потік
        }
        // Почати виконання нового потоку
        void Run()
        {
            Console.WriteLine(Thrd.Name + " почався.");
            answer = sa.Sumlt(a);
            Console.WriteLine("Сума для потоку " + Thrd.Name + " дорівнює " +
answer);
            Console.WriteLine(Thrd.Name + " завершено.");
        }
    }
    //---
    class Sync
    {
        static void Main()
        {
            int[] a = { 1, 2, 3, 4, 5 };
            MyThread mtl = new MyThread("Потік #1", a);
            MyThread mt2 = new MyThread("Потік #2", a);
            mtl.Thrd.Join();
            mt2.Thrd.Join();
            Console.ReadKey();
        }

    }
}
```




**Потоки не синхронізовані. Результат обчислення неправильний.**

Отже, як видно з цього прикладу, в обох породжених потоках метод Sumlt() використовується одночасно для одного і того ж об'єкту, а це призводить до спотворення значення в полі sum.

**Приклад 4. Використанням оператора lock для обчислення суми елементів масиву з декомпозицією за даними (круговий алгоритм)**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Diagnostics;
namespace Lab4
{
    class Program
    {
        static int[] Data; // масив чисел
        static int count1, count2;
        static void GenMas(int n)
        {
            // заповнення масиву. Не розпаралелюється
            Data = new int[n]; // масив чисел
            for (int i = 0; i < Data.Length; i++)
                Data[i] += i + 1;
            Console.WriteLine("Генерація чисел і заповнення масиву");
 //           for (int i = 0; i < Data.Length; i++)
 //               Console.WriteLine("Data[" + i + "] = " + Data[i]);
        }

        static void RunMas(object obj)
        {
            //використовується круговий алгоритм (карусель)
            //перший потік знаходить суму усіх парних елементів масиву,
            //другий потік - суму всіх непарних
            // num - номер першрго елементу масиву 0 або 1
            // Зверніть увагу на те, що в цій формі методу RunMas()
            // вказується параметр типа object.
            int min = (int)obj;  //номер першого елементу масиву
            object lockOn = new object(); // закритий об'єкт, доступний
            lock (lockOn)
            {
               int count = 0;
               int i = min;
               while (i < Data.Length)
                {
                    count += Data[i];
                    i += 2;
                }
                if (min == 0)
                {
                    count1 = count;
                   Console.WriteLine("Сума чисел, підрахована потоком = " +
Thread.CurrentThread.Name + "" + count1);
                }
                else
                {
                    count2 = count;
                    Console.WriteLine("Сума чисел, підрахована потоком = " +
Thread.CurrentThread.Name + "" + count2);
                }
            }
        }
        static void Main()
```

```
        {
            //паралелізм даних - розбиття масиву на 2 частини
            Stopwatch sw = new Stopwatch();
            sw.Start();
            Console.WriteLine("Паралельна версія");
            GenMas(100);
            Thread thrd1 = new Thread(RunMas);
            thrd1.Name = "t0";
            thrd1.Start(0);  //0- парні номери елементів масиву
            Thread thrd2 = new Thread(RunMas);
            thrd2.Name = "t1";
            thrd2.Start(1);  //1- непарні номери елементів масиву
            // створення потоків
            // Тут змінна num передається методу Start ()
            //як аргумент.
            thrd1.Join();
            thrd2.Join();
            int TotalSumma = count1 + count2;
            Console.WriteLine("Сума першої частини масиву  =" + count1);
            Console.WriteLine("Сума другої частини масиву  =" + count2);
            Console.WriteLine("Загальна сума. " + TotalSumma);
            Console.WriteLine("Основний потік завершено.");
            sw.Stop();
            TimeSpan ts = sw.Elapsed;
            Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
            Console.ReadKey();
        }
    }

}
```

**Висновки щодо використання блокування.**

• Якщо блокування будь-якого заданого об'єкту отримане в одному потоці, то після блокування об'єкту воно не може бути отримане в іншому потоці.

• Іншим потокам, що намагаються отримати блокування того ж самого об'єкту, доведеться чекати до тих пір, поки об'єкт не виявиться в розблокованому стані.

• Коли потік виходить із заблокованого фрагмента коду, відповідний об'єкт розблоковується.

**3. Зовнішнє блокування потоків**

Синхронізація потоків за допомогою оператора **lock** для блоку не завжди може використовуватися. Припустимо, що потрібно синхронізувати доступ до методу класу, який був створений кимось іншим і сам не синхронізований. Подібна ситуація можлива при використанні чужого класу, код якого недоступний (наприклад, бібліотечний клас), і оператор **lock** не можна вставити у відповідний метод чужого класу.

В цьому випадку доступ до об'єкту може бути заблокований з коду, зовнішнього по відношенню до даного об'єкту, для чого досить вказати цей об'єкт в операторі lock.

**Приклад 5.**

Розглянемо як можна по-іншому реалізувати той самий приклад обчислення суми. Код в методі Sumlt () вже не є заблокованим, а об'єкт lockOn більше не створюється. Замість цього виклики методу Sumlt () блокуються в класі MyThread.

// Інший спосіб блокування для синхронізації доступу до об'єкту.

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

namespace Lab2_1
{
    class SumArray
    {
        int sum;
        public int Sumlt(int[] nums)
        {
            sum = 0; // встановити початкове значення суми
            for (int i = 0; i < nums.Length; i++)
            {
                sum += nums[i];
                Console.WriteLine("Поточна сума для потоку " +
                Thread.CurrentThread.Name + " дорівнює " + sum);
                Thread.Sleep(10); // дозволити перемикання завдань
            }
            return sum;
        }
    }

    class MyThread
    {
        public Thread Thrd;
        int[] a;
        int answer;
        /* Створити один об'єкт типа SumArray для всіх
        екземплярів класу MyThread. */
        static SumArray sa = new SumArray();
        // Сконструювати новий потік.
        public MyThread(string name, int[] nums)
        {
            a = nums;
            Thrd = new Thread(this.Run);
            Thrd.Name = name;
            Thrd.Start(); // почати потік
        }
        // Почати виконання нового потоку
        void Run()
        {
            Console.WriteLine(Thrd.Name + " начат.");
            // Заблокувати виклики методу Sumlt().
            lock (sa) answer = sa.Sumlt(a);
            Console.WriteLine("Сума для потоку " + Thrd.Name +
            " дорівнює " + answer);
            Console.WriteLine(Thrd.Name + " завершено.");
        }
    }
    class Sync
    {
        static void Main()
        {
```

```
            int[] a = { 1, 2, 3, 4, 5 };
            MyThread mt1 = new MyThread("Потомок #1", a);
            MyThread mt2 = new MyThread("Потомок #2", a);
            mt1.Thrd.Join();
            mt2.Thrd.Join();
            Console.ReadKey();
        }
    }

    }
```

В цьому прикладі блокується виклик методу sa.Sumlt (), а не сам метод Sumlt(). Нижче наведений відповідний рядок коду, в якому здійснюється подібне блокування.

// Заблокувати виклики методу Sumlt().

lock(sa) answer = sa.Sumlt(a);

Об'єкт **sa** є закритим, і тому він може бути заблокований. При такому підході до синхронізації потоків програма дає такий самий правильний результат, як і при попередньому підході.

### 4. Взаємовиключний доступ за допомогою класу Monitor

#### 4.1. Використання методів Monitor.Enter (), Monitor.Exit()

Ключове слово **lock** насправді служить в C# швидким способом доступу до засобів синхронізації, які визначені в класі **Monitor** з простору імен System. Threading. У цьому класі визначено методи для управління синхронізацією. Наприклад, для блокування об'єкту викликається метод Enter (), а для зняття блокування — метод Exit (). Нижче наведені загальні форми цих методів:

**public static void Enter(object obj)**
**public static void Exit(object obj)**

де obj позначає об'єкт, що синхронізується. Якщо ж об'єкт недоступний, то після виклику методу Enter () потік, що викликається, чекає до тих пір, поки об'єкт не стане доступним.

Оператор **lock** в мові C# перетворюється компілятором у використання класу **Monitor**. Наприклад, наступний оператор lock

```
lock(sync_obj)
{
  // Critical section
}
```

перетвориться у виклик методу Enter () класу Monitor, який чекає до тих пір, поки потік отримає дозвіл на блокування об'єкту. Тільки один потік одночасно може бути власником блокування об'єкту. Як тільки блокування дозволяється, потік може входити у критичну секцію. Метод Exit () класу Monitor знімає блокування. Оскільки блокування повинне зніматися у будь-якому випадку (навіть у разі видачі якого-небудь виключення), метод Exit () поміщається в останній обробник блоку try.

```
try
{
 Monitor.Enter(sync_obj);
 // Critical section
}
finally
{
 Monitor.Exit(sync_obj);
}
```

**Приклад 6. Простий приклад застосування методів класу Monitor.** В цьому прикладі в методі main () створюється масив об’єктів потоків `threads`. Потім масив a[] заповнюється числами від 1 до 100. Далі створюються потоки для виклику методу `CheckSum`. Монітор блокує доступ потокам до масиву (об’єктом синхронізації є масив a). Після цього викликається метод Mix(), в якому елементи масиву міняються місцями.

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

namespace Lab4_Monitor2
{
    class Program
    {
        public static int[] a = new int[100];
        public static bool stop = false;
        public static bool incorrect = false;

        static void Main(string[] args)
        {
            int i;
            Thread[] threads = new Thread[10]; //створюється об'єкти для 10 потоків

            // заповнюється масив числами від 1 до 100
            for (i = 0; i < 100; i++)
                a[i] = i + 1;

            for (i = 0; i < 10; i++)
            {
                // створюється 10 потоків для виклику 1 метода CheckSum
                threads[i] = new Thread(new ThreadStart(CheckSum));
                threads[i].Name = "Thread " + (i + 1).ToString();
                threads[i].Start();  //запуск отоків
            }
            Mix();  //
            stop = true;
            if (!incorrect)
                Console.WriteLine("Sum is correct!");
            Console.ReadKey();

        }

        public static void Mix()
        {
            Console.WriteLine("mixing...");
```

```
            Random rnd = new Random();
            DateTime start = DateTime.Now;
            while (!stop && (DateTime.Now - start).Seconds < 10)
            {
                int i = rnd.Next(0, 100);
                int j = rnd.Next(0, 100);

                Monitor.Enter(a);
                try
                {
                    // Console.WriteLine("swapping ({0}, {1})", i, j);
                    int tmp = a[i];
                    a[i] = a[j];
                    a[j] = tmp;
                }
                finally
                {
                    Monitor.Exit(a);
                }
            }
        }

        public static void CheckSum()
        {
            while (!stop)
            {
                int s = 0;

                Monitor.Enter(a);
                try
                {
                    for (int i = 0; i < 100; i++)
                    {
                        s += a[i];
                    }
                }
                finally
                {
                    Monitor.Exit(a);
                }

                if (s != 5050)
                {
                    Console.WriteLine(
                        "{0}: Sum = {1} -- incorrect!",
                        Thread.CurrentThread.Name, s
                    );
                    incorrect = true;
                    stop = true;
                }
            }
        }

    }
}
```

При роботі синхронізованої програми помилки не відбувається, ось її видача:

```
mixing...
Sum is correct!
```

**Приклад 7. Версія обчислення суми елементів масиву з розділенням на частини з використанням монітора**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Diagnostics;
namespace Lab4Monitor
{
    class Program
    {
        static int[] Data; // масив чисел
        static int count1, count2;
        static void GenMas(int n)
        {
            // заповнення масиву. Не розпаралелюється
            Data = new int[n]; // масив чисел
            for (int i = 0; i < Data.Length; i++)
                Data[i] += i + 1;
            Console.WriteLine("Генерація чисел і заповнення масиву");
            //           for (int i = 0; i < Data.Length; i++)
            //               Console.WriteLine("Data[" + i + "] = " + Data[i]);
        }

        static void RunMas(object obj)
        {
            //використовується круговий алгоритм (карусель)
            //перший потік знаходить суму усіх парних елементів масиву,
            //другий потік - суму всіх непарних
            // num - номер першрго елементу масиву 0 або 1
            // Зверніть увагу на те, що в цій формі методу RunMas()
            // вказується параметр типа object.
            try
               {
                Monitor.Enter(Data);
                int min = (int)obj;  //номер першого елементу масиву
                int count = 0;
                int i = min;
                while (i < Data.Length)
                {
                    count += Data[i];
                    i += 2;
                }
                if (min == 0)
                {
                    count1 = count;
                    Console.WriteLine("Сума чисел, підрахована потоком = " +
Thread.CurrentThread.Name + "" + count1);
                }
                else
                {
                    count2 = count;
                    Console.WriteLine("Сума чисел, підрахована потоком = " +
Thread.CurrentThread.Name + "" + count2);
                }
            }
            finally
            {
                Monitor.Exit(Data);
            }
         }
        static void Main()
        {
```

```
            //паралелізм даних - розбиття масиву на 2 частини
            Stopwatch sw = new Stopwatch();
            sw.Start();
            Console.WriteLine("Паралельна версія");
            GenMas(100);
            Thread thrd1 = new Thread(RunMas);
            thrd1.Name = "t0";
            thrd1.Start(0);  //0- парні номери елементів масиву
            Thread thrd2 = new Thread(RunMas);
            thrd2.Name = "t1";
            thrd2.Start(1);  //1- непарні номери елементів масиву
            // створення потоків
            // Тут змінна num передається методу Start ()
            //як аргумент.
            thrd1.Join();
            thrd2.Join();
            int TotalSumma = count1 + count2;
            Console.WriteLine("Сума першої частини масиву  =" + count1);
            Console.WriteLine("Сума другої частини масиву  =" + count2);
            Console.WriteLine("Загальна сума. " + TotalSumma);
            Console.WriteLine("Основний потік завершено.");
            sw.Stop();
            TimeSpan ts = sw.Elapsed;
            Console.WriteLine("Total time: {0}", ts.TotalMilliseconds);
            Console.ReadKey();
        }
    }
}
```

### 4.2 Використання методу Monitor.TryEnter ()

Клас Monitor в C# має одну велику перевагу в порівнянні з оператором lock: він дозволяє додавати значення тайм-ауту для очікування отримання блокування. Тобто замість того, щоб до нескінченності чекати отримання блокування, ми можемо скористатися методом **TryEnter** і передати в ньому значення тайм-ауту, що вказує, скільки часу слід чекати отримання блокування. У разі отримання блокування для **obj** метод TryEnter повинен повертати значення true і виконувати синхронізований доступ до стану, що охороняється об'єктом **obj**. У випадку якщо **obj** заблокований іншим потоком протягом більше 500 мілісекунд, метод TryEnter повинен повертати значення false, що вказує потоку перервати очікування і робити що-небудь інше. Можливо, пізніше потік зможе спробувати отримати блокування ще раз.

```
if (Monitor.TryEnter(obj, 500))
{
try
// отримання блокування
// синхронізована область для obj
finally Monitor.Exit(obj);
else
// блокування отримати не вдалося, робимо що-небудь інше
}
```

Крім того, в класі Monitor визначені методи **Wait (), Pulse () і PulseAll (),** за допомогою яких можна організувати взаємодію потоків.

### 4.3. Синхронізація потоків за допомогою методів Wait (), Pulse () і PulseAll () класу Monitor

Розглянемо наступну ситуацію. Потік **1** виконується в блоці коду **lock** і йому потрібний доступ до ресурсу R, який тимчасово недоступний. Що ж тоді робити потоку **2**? Якщо потік **1** увійде до організованого циклу опитування, чекаючи звільнення ресурсу R, то тим самим він зв'яже відповідний об'єкт, блокуючи доступ до нього інших потоків. Це не найкраще рішення, оскільки воно позбавляє переваг багатопоточного програмування.

Краще рішення полягає в тому, щоб тимчасово звільнити об'єкт і тим самим дати можливість виконуватися іншим потокам. Такий підхід грунтується на деякій формі взаємодії між потоками, завдяки якій один потік може повідомляти інший про те, що він заблокований і що інший потік може відновити своє виконання. Взаємодія між потоками організується в C# за допомогою методів Wait (), Pulse () і PulseAll (), які визначені в класі Monitor.

Методи Wait (), Pulse () і PulseAll () можуть викликатися лише із *заблокованого фрагмента блоку*. Вони застосовуються таким чином. Коли виконання потоку тимчасово заблоковане, він викликає метод **Wait ().** У результаті потік переходить в стан очікування, а блокування з відповідного об'єкту знімається, що дає можливість використовувати цей об'єкт в іншому потоці. Надалі очікуючий потік активізується, коли інший потік увійде в аналогічний стан блокування, і викликає метод Pulse () або PulseAll (). При виклику методу Pulse () поновлюється виконання першого потоку, який чекає своєї черги на отримання блокування. А виклик методу PulseAll () сигналізує про зняття блокування всім чекаючим потокам.

Нижче наведені дві часто використовувані форми методу Wait ().

**public static bool Wait(object obj)**
**public static bool Wait(object obj, int мілісекунд_простою)**

У першій формі очікування триває аж до повідомлення про звільнення об'єкту, а в другій формі — як до повідомлення про звільнення об'єкту, так і до виділення періоду часу, на який вказує кількість мілісекунд_простою.

В обох формах obj позначає об'єкт, звільнення якого очікується. Нижче наведені загальні форми методів Pulse () і PulseAll ():

**public static void Pulse(object obj)**
**public static void PulseAll(object obj)**

де obj позначає об'єкт, що звільняється.

Якщо методи Wait (), Pulse () і PulseAll () викликаються з коду, що знаходиться за межами синхронізованого коду, наприклад поза блоком lock, то генерується виключення SynchronizationLockException.

**Приклад 8. Використання методів Wait () і Pulse ().**

Програма імітує роботу годинника і відображає цей процес на екрані словами "тік" і так". Для цієї мети в програмі створюється клас TickTock, що містить два наступні методи: Tick () і Tock(). Метод Tick () виводить на екран слово "тік", а метод Tock() — слово "так". Для запуску годинника далі в програмі створюються два потоки: один з них викликає метод Tick (), а інший — метод Tock ().

Мета прикладу полягає в тому, щоб обидва потоки виконувалися по черзі виводячи на екран слова "тік" і так".

// Використання методів Wait() і Pulse () для імітації роботи годинника

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

namespace Lab3
{
    class TickTock
    {
        object lockOn = new object();
        public void Tick(bool running)
        {
            lock (lockOn)
            {
                if (!running)
                { // остановить часы
                    Monitor.Pulse(lockOn); // уведомить любые ожидающие потоки
                    return;
                }
                Console.Write("тик ");
                Monitor.Pulse(lockOn); // разрешить выполнение метода Tock()
                Monitor.Wait(lockOn); // ожидать завершения метода Tock()
            }
        }
        public void Tock(bool running)
        {
            lock (lockOn)
            {
                if (!running)
                { // остановить часы
                    Monitor.Pulse(lockOn); // уведомить любые ожидающие потоки
                    return;
                }
                Console.WriteLine("так");
                Monitor.Pulse(lockOn); // разрешить выполнение метода Tick()
                Monitor.Wait(lockOn); // ожидать завершения метода Tick()
            }
        }
    }

    class MyThread
    {
        public Thread Thrd;
```

```
        TickTock ttOb;
        // Сконструировать новый поток.
        public MyThread(string name, TickTock tt)
        {
            Thrd = new Thread(this.Run);
            ttOb = tt;
            Thrd.Name = name;
            Thrd.Start();
        }
        // Начать выполнение нового потока,
        void Run()
        {
            if (Thrd.Name == "Tick")
            {
                for (int i = 0; i < 5; i++) ttOb.Tick(true);
                ttOb.Tick(false);
            }
            else
            {
                for (int i = 0; i < 5; i++) ttOb.Tock(true);
                ttOb.Tock(false);
            }
        }
    }

    class TickingClock
    {
        static void Main()
        {
            TickTock tt = new TickTock();
            MyThread mt1 = new MyThread("Tick", tt);
            MyThread mt2 = new MyThread("Tock", tt);
            mt1.Thrd.Join();
            mt2.Thrd.Join();
            Console.WriteLine("Часы остановлены");
            Console.ReadKey();
        }
    }
}
```

Результат виконання програми
тик так
тик так
тик так
тик так
тик так
Часы остановлены

Розглянемо цю програму детальніше. У методі Main () створюється об'єкт tt типу TickTock, який використовується для запуску двох потоків на виконання. Якщо у методі Run () з класу MyThread виявляється ім'я потоку Tick, яке відповідає ходу годинника "тік", то викликається метод Tick (). А якщо це ім'я потоку Tock, який відповідає ходу годинника "так", то

викликається метод Tock (). Кожен з цих методів викликається п'ять разів підряд з передачею логічного значення **true** в якості аргументу. Годинник йде до тих пір, поки цим методам передається логічне значення true і зупиняються, як тільки передається логічне значення false.

Найважливіша частина програми, що розглядається тут, знаходиться в методах Tick () і Tock (). Почнемо з методу Tick (), код якого для зручності наведено нижче.

```
public void Tick(bool running) {
lock(lockOn) {
if(!running) { // остановить часы
   Monitor.Pulse(lockOn); // уведомить любые ожидающие потоки
return;
}
Console.Write("тик ");
Monitor.Pulse(lockOn); // разрешить выполнение метода Tock()
Monitor.Wait(lockOn); // ожидать завершения метода Tock()
}
}
```

Перш за все зверніть увагу на код методу Tick () у блоці **lock**. Нагадаємо, що методи Wait () і Pulse () можуть використовуватися лише в синхронізованих блоках коду. На початку методу Tick () перевіряється значення поточного параметра, яке служить явною ознакою зупинки годинника. Якщо це логічне значення **false**, то годинник зупинений. В цьому випадку викликається метод Pulse (), що дозволяє виконання будь-якого потоку, чекаючого своєї черги.

Якщо ж годинник йде при виконанні методу Tick (), то на екран виводиться слово "тик" з пропуском, потім викликається метод Pulse (), а після нього — метод Wait (). При виклику методу Pulse () дозволяється виконання потоку для того ж самого об'єкту, а при виклику методу Wait () виконання методу Tick () припиняється до тих пір, поки метод Pulse () не буде викликаний з іншого потоку. Таким чином, коли викликається метод Tick (), відображається одне слово "тик" з пробілом, дозволяється виконання іншого потоку, а потім виконання даного методу припиняється.

Метод Tock() є точною копією методу Tick (), за винятком того, що він виводить на екран слово "так". Таким чином, при вході в метод Tock () на екран виводиться слово "так", викликається метод Pulse (), а потім виконання методу Tock() припиняється. Методи Tick () і Tock() взаємно синхронізовані.

Коли годинник зупинений, метод Pulse () викликається для того, щоб забезпечити успішний виклик методу Wait (). Нагадаємо, що метод Wait () викликається в обох методах, Tick () і Tock (), після виведення відповідного слова на екран. Але справа в тому, що коли годинник зупинений, один з цих методів все ще знаходиться в стані очікування. Тому завершуючий виклик методу Pulse () потрібний, щоб виконати очікуючий метод до кінця.

**Висновки.**

При використанні декількох потоків буває потрібно *координувати їхні дії*. Процес досягнення такої координації називається **синхронізацією**.

В основу синхронізації покладено поняття **блокування**, за допомогою якого організовується управління доступом до коду блоку в об'єкті. Синхронізація організовується за допомогою ключового слова **lock**.

Контрольований блок коду часто називають **критичним блоком**, або **критичною секцією**.

Ключове слово **lock** насправді служить в C# швидким способом доступу до засобів синхронізації, які визначені в класі **Monitor** з простору імен System. Threading. У цьому класі визначено методи для управління синхронізацією.

Клас **Monitor** контролює доступ до об'єктів, надаючи блокування об'єкта одному потоку. Можна також використовувати **Monitor** для того, щоб переконатися, що жоден потік не має доступу до секції коду програми, блокування, що виконується власником, поки інший потік не виконуватиме код, використовуючи інший об'єкт з блокуванням.

**Контрольні запитання**

1. В яких випадках потрібно синхронізувати виконання потоків?
2. Як заблокувати не сам метод, а виклик методу, який синхронізується?
3. Коли не можна використовувати синхронізацію потоків за допомогою блокування? Що робити в цьому випадку?
4. В чому суть ідеї синхронізації потоків за допомогою блокування? Приклад.
5. Що буде, якщо паралельні потоки, які використовують спільні дані, не будуть синхронізовані?
6. Яке призначення класу Monitor?
7. Яке призначення методів Wait (), Pulse () і Pulse All? Яка між ними різниця?
8. За допомогою довідкової системи **msdn** визначити, які ще методи і властивості входять у клас Monitor.
9. Чи можуть методи Wait(), Pulse () і PulseAll () викликатися з незаблокованого блоку коду?
10. Поясніть механізм синхронізації за допомогою метода Wait()?