

Модуль 1. Багатопоточне програмування

Лекція 6. Синхронізація потоків з використанням м'ютекса і семафора

1. Взаємоблокування і стан гонки

При розробці багатопоточних програм слід бути уважним, щоб уникнути взаємоблокувань і станів гонок. *Взаємоблокування* — це ситуація, в якій один потік чекає певних дій від іншого потоку, а інший потік, у свою чергу, чекає чогось від першого потоку. У результаті обидва потоки припиняються, чекаючи один одного, і жоден з них не виконується. Це призведе до зависання програми.

Часто причину взаємоблокування не просто з'ясувати, аналізуючи початковий код програми, оскільки процеси, що паралельно виконуються, можуть взаємодіяти досить складним чином під час виконання.

Для виключення взаємоблокування потрібне уважне програмування і ретельне тестування. В цілому, якщо багатопоточна програма періодично "зависає", то найбільш імовірною причиною цього є взаємоблокування.

Стан *гонки* виникає у тому випадку, коли два або більше потоків намагаються одночасно дістати доступ до спільного ресурсу без належної синхронізації. Так, в одному потоці може зберігатися значення в змінній, а в іншому — змінюватися поточне значення цієї ж змінної. У відсутності синхронізації кінцевий результат залежатиме від того, в якому саме порядку виконуються потоки: чи змінюється значення змінної в другому потоці або ж воно зберігається в першому. Про подібну ситуацію говорять, що потоки "*ганяються один за одним*", причому кінцевий результат залежить від того, який з потоків завершиться першим.

Виникнення стану гонок, як і взаємоблокування, виявити складно. Тому його краще запобігти, синхронізуючи належним чином доступ до спільних ресурсів при програмуванні.

2. Синхронізація потоків за допомогою м'ютекса

В більшості випадків, коли потрібна синхронізація, виявляється достатньо оператора **lock**. Проте в деяких випадках, як, наприклад, при обмеженні доступу до спільних ресурсів, зручнішими виявляються механізми синхронізації, які вбудовані в .NET Framework. Це об'єкти **м'ютекс** і **семафор**.

М'ютекс (mutual exclusion — взаємовиключення) призначений для тих ситуацій, в яких спільний ресурс може бути одночасно використаний тільки в **одному потоці**, тобто для надання монопольного доступу до ресурсу.

Наприклад, системний журнал спільно використовується в декількох процесах, але тільки в одному з них дані можуть записуватися у файл цього журналу у будь-який момент часу. Для синхронізації процесів в такій ситуації ідеально підходить м'ютекс.

М'ютекс підтримується в класі `System.Threading.Mutex`. У нього є декілька конструкторів. Нижче наведені два найбільш використовуваних конструктори.

```
public Mutex()  
public Mutex(bool initiallyOwned)
```

У першій формі конструктора створюється м'ютекс, яким спочатку ніхто не володіє. М'ютекс без параметрів ще називають *локальним*.

А в другій формі початковим станом м'ютекса заволодіває викликаючий потік, якщо параметр `initiallyOwned` має логічне значення **true**. У іншому випадку м'ютексом ніхто не володіє.

Для того, щоб отримати м'ютекс, в коді програми слід викликати метод **WaitOne()** для цього м'ютекса. Метод `WaitOne()` успадковується класом `Mutex` від класу `Thread.WaitHandle`. Нижче наведена його проста форма.

```
public bool WaitOne();
```

Метод `WaitOne()` чекає до тих пір, поки не буде отриманий м'ютекс, для якого він був викликаний. Отже, цей метод блокує виконання викликаючого потоку до тих пір, поки не стане доступним вказаний м'ютекс. Він завжди повертає логічне значення **true**.

Коли ж в коді більше не потрібно володіти м'ютексом, він звільняється за допомогою виклику методу `ReleaseMutex()`, форма якого наведена нижче.

```
public void ReleaseMutex();
```

У цій формі метод `ReleaseMutex()` звільняє м'ютекс, для якого він був викликаний, що дає можливість іншому потоку отримати даний м'ютекс. Для застосування м'ютекса з метою синхронізації доступу до спільного ресурсу методи `WaitOne()` і `ReleaseMutex()` використовуються так, як показано в наведеному нижче прикладі.

```
Mutex myMtx = new Mutex();  
// ...  
myMtx.WaitOne(); // чекати отримання м'ютекса  
// Дістати доступ до спільного ресурсу.  
myMtx.ReleaseMutex(); // звільнити м'ютекс
```

При виклику методу **WaitOne()** виконання відповідного потоку припиняється до тих пір, поки не буде отриманий м'ютекс. А при виклику методу **ReleaseMutex()** м'ютекс звільняється і потім може бути отриманий іншим потоком. Завдяки такому підходу до синхронізації одночасний доступ до спільного ресурсу обмежується тільки одним потоком.

У наведеному нижче прикладі створюються два потоки у вигляді класів `IncThread` і `DecThread`, яким потрібний доступ до спільного ресурсу: до змінної `SharedRes.Count`. У потоці `IncThread` змінна `SharedRes.Count` збільшується на 1, а в потоці `DecThread` — зменшується. Щоб уникнути одночасного доступу обох потоків до спільного ресурсу `SharedRes.Count` цей доступ синхронізується м'ютексом `Mtx`, членом класу `SharedRes`.

Приклад 1

```
// Застосувати мьютекс
using System;
using System.Threading;
namespace Lab_5_Mutex1
{
    // У цьому класі міститься спільний ресурс(змінна Count)
    //а також мьютекс (Mtx), який керує доступом до неї.
    class SharedRes
    {
        public static int Count = 0;
        public static Mutex Mtx = new Mutex();
    }
    // У цьому потоці змінна SharedRes.Count збільшується.
    class IncThread
    {
        int num;
        public Thread Thrd;
        public IncThread(string name, int n)
        {
            Thrd = new Thread(this.Run);
            num = n;
            Thrd.Name = name;
            Thrd.Start();
        }
        // Точка входу в потік,
        void Run()
        {
            Console.WriteLine(Thrd.Name + " очікує мьютекс.");
            // Отримати мьютекс.
            SharedRes.Mtx.WaitOne();
            Console.WriteLine(Thrd.Name + " отримує мьютекс.");
            do
            {
                Thread.Sleep(100);
                SharedRes.Count++;
                Console.WriteLine("В потік " + Thrd.Name +
                    ", SharedRes.Count = " + SharedRes.Count);
                num--;
            } while (num > 0);
            Console.WriteLine(Thrd.Name + " звільняє мьютекс.");
            // Звільнити мьютекс.
            SharedRes.Mtx.ReleaseMutex();
        }
    }

    //У цьому потоці змінна SharedRes.Count зменшується.
    class DecThread
    {
        int num;
        public Thread Thrd;
        public DecThread(string name, int n)
        {
            Thrd = new Thread(new ThreadStart(this.Run));
            num = n;
            Thrd.Name = name;
        }
    }
}
```

```

        Thrd.Start();
    }
    // Точка входу в потік,
    void Run()
    {
        Console.WriteLine(Thrd.Name + " очікує мьютекс.");
        // Отримати мьютекс.
        SharedRes.Mtx.WaitOne();
        Console.WriteLine(Thrd.Name + " отримує мьютекс");
        do
        {
            Thread.Sleep(100);
            SharedRes.Count--;
            Console.WriteLine("В потоці " + Thrd.Name + ", SharedRes.Count = " +
SharedRes.Count);
            num--;
        } while (num > 0);
        Console.WriteLine(Thrd.Name + " звільняє мьютекс.");
        // Звільнити мьютекс.
        SharedRes.Mtx.ReleaseMutex();
    }
}
class MutexDemo
{
    static void Main()
    {
        // Створити два потоки.
        IncThread mt1 = new IncThread("Збільшуючий Потік", 5);
        Thread.Sleep(1); // дозволити збільшуючому потоку початися
        DecThread mt2 = new DecThread("Зменшуючий Потік", 5);
        mt1.Thrd.Join();
        mt2.Thrd.Join();
        Console.ReadKey();
    }
}
}
}

```

Результат роботи програми

```

file:///D:/MHTY/For_K-81/Парал.программирование/Labs/Lab_3/Lab_3/bin/Debug/Lab_3.EXE
Збільшуючий Потік очікує мьютекс.
Зменшуючий Потік очікує мьютекс.
Збільшуючий Потік отримує мьютекс.
В потоці Збільшуючий Потік, SharedRes.Count = 1
В потоці Збільшуючий Потік, SharedRes.Count = 2
В потоці Збільшуючий Потік, SharedRes.Count = 3
В потоці Збільшуючий Потік, SharedRes.Count = 4
В потоці Збільшуючий Потік, SharedRes.Count = 5
Збільшуючий Потік звільняє мьютекс.
Зменшуючий Потік отримує мьютекс
В потоці Зменшуючий Потік, SharedRes.Count = 4
В потоці Зменшуючий Потік, SharedRes.Count = 3
В потоці Зменшуючий Потік, SharedRes.Count = 2
В потоці Зменшуючий Потік, SharedRes.Count = 1
В потоці Зменшуючий Потік, SharedRes.Count = 0
Зменшуючий Потік звільняє мьютекс.

```

Як впливає з наведеного вище результату, доступ до спільного ресурсу (змінної `SharedRes.Count`) синхронізований, і тому значення цієї змінної може бути одночасно змінене тільки в одному потоці.

Для того, щоб переконатися в тому, що м'ютекс необхідний для отримання наведеного вище результату, закоментуємо виклики методів `WaitOne()` і `ReleaseMutex()` у коді цієї програми.

Приклад 2

```
// Застосувати м'ютекс
using System;
using System.Threading;
namespace Lab_5_Mutex2
{
    // У цьому класі міститься спільний ресурс(змінна Count)
    //а також м'ютекс (Mtx), який керує доступом до неї.
    class SharedRes
    {
        public static int Count = 0;
        public static Mutex Mtx = new Mutex();
    }
    // У цьому потоці змінна SharedRes.Count збільшується.
    class IncThread
    {
        int num;
        public Thread Thrd;
        public IncThread(string name, int n)
        {
            Thrd = new Thread(this.Run);
            num = n;
            Thrd.Name = name;
            Thrd.Start();
        }
        // Точка входу в потік,
        void Run()
        {
            Console.WriteLine(Thrd.Name + " очікує м'ютекс.");
            // Отримати м'ютекс.
            // SharedRes.Mtx.WaitOne();
            Console.WriteLine(Thrd.Name + " отримує м'ютекс.");
            do
            {
                Thread.Sleep(100);
                SharedRes.Count++;
                Console.WriteLine("В потоці " + Thrd.Name +
                    ", SharedRes.Count = " + SharedRes.Count);
                num--;
            } while (num > 0);
            Console.WriteLine(Thrd.Name + " звільняє м'ютекс.");
            // Звільнити м'ютекс.
            // SharedRes.Mtx.ReleaseMutex();
        }
    }

    //У цьому потоці змінна SharedRes.Count зменшується.
    class DecThread
    {
        int num;
        public Thread Thrd;
        public DecThread(string name, int n)
        {
            Thrd = new Thread(new ThreadStart(this.Run));
            num = n;
            Thrd.Name = name;
            Thrd.Start();
        }
        // Точка входу в потік,
        void Run()
```

```

{
    Console.WriteLine(Thrd.Name + " очікує мьютекс.");
    // Отримати мьютекс.
    // SharedRes.Mtx.WaitOne();
    Console.WriteLine(Thrd.Name + " отримує мьютекс");
    do
    {
        Thread.Sleep(100);
        SharedRes.Count--;
        Console.WriteLine("В потоці " + Thrd.Name + ", SharedRes.Count = " +
SharedRes.Count);
        num--;
    } while (num > 0);
    Console.WriteLine(Thrd.Name + " звільняє мьютекс.");
    // Звільнити мьютекс.
    // SharedRes.Mtx.ReleaseMutex();
}
class MutexDemo
{
    static void Main()
    {
        // Створити два потоки.
        IncThread mt1 = new IncThread("Збільшуючий Потік", 5);
        Thread.Sleep(1); // дозволити збільшуючому потоку початися
        DecThread mt2 = new DecThread("Зменшуючий Потік", 5);
        mt1.Thrd.Join();
        mt2.Thrd.Join();
        Console.ReadKey();
    }
}
}
}

```

При її подальшому виконанні ви отримаєте наступний результат, хоча у вас він може бути дещо іншим.

```

Збільшуючий Потік очікує мьютекс.
Збільшуючий Потік отримує мьютекс.
Зменшуючий Потік очікує мьютекс.
Зменшуючий Потік отримує мьютекс
В потік? Збільшуючий Потік, SharedRes.Count = 1
В потік? Зменшуючий Потік, SharedRes.Count = 0
В потік? Збільшуючий Потік, SharedRes.Count = 1
В потік? Зменшуючий Потік, SharedRes.Count = 0
В потік? Збільшуючий Потік, SharedRes.Count = 1
В потік? Зменшуючий Потік, SharedRes.Count = 0
В потік? Збільшуючий Потік, SharedRes.Count = 1
В потік? Зменшуючий Потік, SharedRes.Count = 0
В потік? Збільшуючий Потік, SharedRes.Count = 1
Збільшуючий Потік звільняє мьютекс.
В потік? Зменшуючий Потік, SharedRes.Count = 0
Зменшуючий Потік звільняє мьютекс.

```

Як впливає з наведеного вище результату, без м'ютекса збільшення і зменшення змінної SharedRes.Count відбувається безладно, а не послідовно.

Приклад 3

```
// This example shows how a Mutex is used to synchronize access
// to a protected resource. Unlike Monitor, Mutex can be used with
// WaitHandle.WaitAll and WaitAny, and can be passed across
// AppDomain boundaries.

using System;
using System.Threading;
namespace Lab_5_Mutex3

class Test
{
    // Create a new Mutex. The creating thread does not own the
    // Mutex.
    private static Mutex mut = new Mutex();
    private const int numIterations = 1;
    private const int numThreads = 3;

    static void Main()
    {
        // Create the threads that will use the protected resource.
        for(int i = 0; i < numThreads; i++)
        {
            Thread myThread = new Thread(new ThreadStart(MyThreadProc));
            myThread.Name = String.Format("Thread{0}", i + 1);
            myThread.Start();
        }

        // The main thread exits, but the application continues to
        // run until all foreground threads have exited.
    }

    private static void MyThreadProc()
    {
        for(int i = 0; i < numIterations; i++)
        {
            UseResource();
        }
    }

    // This method represents a resource that must be synchronized
    // so that only one thread at a time can enter.
    private static void UseResource()
    {
        // Wait until it is safe to enter.
        mut.WaitOne();

        Console.WriteLine("{0} has entered the protected area",
            Thread.CurrentThread.Name);

        // Place code to access non-reentrant resources here.

        // Simulate some work.
        Thread.Sleep(500);

        Console.WriteLine("{0} is leaving the protected area\r\n",
            Thread.CurrentThread.Name);

        // Release the Mutex.
        mut.ReleaseMutex();
    }
}
```

```

    }
}

```

Приклад 4. Знаходження суми чисел. Використання м'ютекса

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Diagnostics;
namespace Lab5Mutex
{
    class Program
    {
        static int[] Data; // масив чисел
        static int count1, count2;
        public static Mutex Mtx = new Mutex(); //створення об'єкту
        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");
        }

        static void RunMas(object obj)
        {
            //використовується круговий алгоритм (карусель)
            //перший потік знаходить суму усіх парних елементів масиву,
            //другий потік - суму всіх непарних
            // num - номер першого елементу масиву 0 або 1
            // Зверніть увагу на те, що в цій формі методу RunMas()
            // вказується параметр типа object.
            Mtx.WaitOne();
            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);
            }
            Mtx.ReleaseMutex();
        }

        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();
    }
}
}

```

М'ютекс, створений в попередньому прикладі, відомий тільки тому процесу, який його створив. Але м'ютекс можна створити і так, щоб він був відомий де-небудь ще. Для цього він повинен бути іменованим. Нижче наведені форми конструктора, призначені для створення такого м'ютекса.

public Mutex(bool initiallyOwned, string имя)
public Mutex(bool initiallyOwned, string имя, out bool createdNew)

В обох формах конструктора ім'я позначає конкретне ім'я м'ютекса. Якщо в першій формі конструктора параметр `initiallyOwned` має логічне значення **true**, то володіння м'ютексом запитується. Але оскільки м'ютекс може належати іншому процесу на системному рівні, то для цього параметра краще вказати логічне значення **false**. А після повернення з другої форми конструктора параметр `createdNew` буде мати логічне значення **true**, якщо володіння м'ютексом було запитане і отримане, і логічне значення **false**, якщо запит на володіння був відхилений. Існує і третя форма конструктора типу `Mutex`, в якій допускається вказувати об'єкт типу `MutexSecurity`. За допомогою іменованих м'ютексів можна синхронізувати взаємодію процесів. Наступний приклад демонструє використання іменованого м'ютекса.

Приклад 5. <http://msdn.microsoft.com/ru-ru/library/f55ddskf.aspx>

```

// This example shows how a named mutex is used to signal between
// processes or threads.
// Run this program from two (or more) command windows. Each process
// creates a Mutex object that represents the named mutex "MyMutex".
// The named mutex is a system object whose lifetime is bounded by the
// lifetimes of the Mutex objects that represent it. The named mutex
// is created when the first process creates its local Mutex; in this

```

```
// example, the named mutex is owned by the first process. The named
// mutex is destroyed when all the Mutex objects that represent it
// have been released.
// The constructor overload shown here cannot tell the calling thread
// whether initial ownership of the named mutex was granted. Therefore,
// do not request initial ownership unless you are certain that the
// thread will create the named mutex.
```

```
using System;
using System.Threading;

public class Test
{
    public static void Main()
    {
        // Create the named mutex. Only one system object named
        // "MyMutex" can exist; the local Mutex object represents
        // this system object, regardless of which process or thread
        // caused "MyMutex" to be created.
        Mutex m = new Mutex(false, "MyMutex");

        // Try to gain control of the named mutex. If the mutex is
        // controlled by another thread, wait for it to be released.
        Console.WriteLine("Waiting for the Mutex.");
        m.WaitOne();

        // Keep control of the mutex until the user presses
        // ENTER.
        Console.WriteLine("This application owns the mutex. " +
            "Press ENTER to release the mutex and exit.");
        Console.ReadLine();

        m.ReleaseMutex();
    }
}
```

3. Синхронізація потоків за допомогою Семафору

Семафор подібний до мьютекса, за винятком того, що він надає *одночасний доступ до спільного ресурсу не одному, а декільком потокам*. Тому семафор придатний для синхронізації цілого ряду ресурсів. Семафор управляє доступом до спільного ресурсу, використовуючи для цієї мети **лічильник**. Якщо значення лічильника більше нуля, то доступ до ресурсу дозволений. А якщо це значення рівне нулю, то доступ до ресурсу заборонений. За допомогою лічильника ведеться підрахунок кількості дозволів. Отже, для доступу до ресурсу потік повинен отримати дозвіл від семафора.

У .NET 4 пропонується два класи з функціональністю семафора: *Semaphore* і *SemaphoreSlim*. Клас *Semaphore* може бути іменований, використовувати ресурси в масштабі усієї системи і забезпечувати синхронізацію між різними процесами. Клас *SemaphoreSlim* є полегшеною версією класу *Semaphore*, яка оптимізована для забезпечення коротшого часу очікування.

Зазвичай потік, якому потрібний доступ до спільного ресурсу, намагається отримати дозвіл від семафора. Якщо значення лічильника семафора більше нуля, то потік отримує дозвіл, а лічильник семафора зменшується. Інакше потік блокується до тих пір, поки не отримає дозвіл. Коли

ж потоку більше не потрібний доступ до спільного ресурсу, він вивільняє дозвіл, а лічильник семафора збільшується. Якщо дозволу чекає інший потік, то він отримує його у цей момент. Кількість одночасно дозволених доступів вказується при створенні семафора. Так, якщо створити семафор, що одночасно дозволяє тільки один доступ, то такий семафор діятиме як мьютекс.

Семафори особливо корисні в тих випадках, коли спільний ресурс складається з групи або пулу ресурсів. Наприклад, пул ресурсів може складатися з цілого ряду мережових з'єднань, кожне з яких служить для передачі даних. Тому потоку, якому потрібне мережове з'єднання, все одно, яке саме з'єднання він отримає.

В цьому випадку семафор забезпечує зручний механізм управління доступом до мережових з'єднань.

Семафор реалізується в класі **System.Threading.Semaphore**, у якого є декілька конструкторів. Нижче наведена проста форма конструктора даного класу:

```
public Semaphore(int initialCount, int maximumCount)
```

де `initialCount` — це первинне значення для лічильника дозволів семафора, тобто початкова кількість доступних дозволів; `maximumCount` — максимальне значення лічильника, тобто максимальна кількість дозволів, які може дати семафор.

Семафор застосовується так само, як і мьютекс. Для отримання доступу до ресурсу в коді програми викликається метод **WaitOne ()** для семафора. Цей метод успадковується класом `Semaphore` від класу `WaitHandle`.

Метод `WaitOne ()` чекає до тих пір, поки не буде отриманий семафор, для якого він викликається. Таким чином, він блокує виконання потоку, який викликається, до тих пір, поки вказаний семафор не дасть дозвіл на доступ до ресурсу.

Якщо коду більше не потрібно володіти семафором, він звільняє його, викликаючи метод `Release ()`. Нижче наведено дві форми цього методу.

```
public int Release()
```

```
public int Release(int releaseCount)
```

У першій формі метод `Release ()` вивільняє тільки один дозвіл, а в другій формі — кількість дозволів визначається параметром `releaseCount`.

В обох формах цей метод повертає підраховану кількість дозволів, що існували до вивільнення.

Метод **WaitOne ()** допускається викликати в потоці кілька разів перед викликом методу **Release ()**. Але кількість викликів методу **WaitOne ()** повинна дорівнювати кількості викликів методу **Release ()** перед вивільненням дозволу. З іншого боку, можна скористатися формою виклику методу `Release(int num)`, щоб передати кількість дозволів, що вивільняються, рівну кількості викликів методу `WaitOne ()`.

Нижче наведений приклад програми, в якій демонструється застосування семафора. У цій програмі семафор використовується в класі `MyThread` для

одночасного виконання тільки двох потоків типу `MyThread`. Отже, ресурсом, що розділяється, в даному випадку є ЦП.

Приклад 6

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace Semaphore
{
    // Цей потік дозволяє одночасне виконання
    // тільки двох своїх екземплярів
    class MyThread
    {
        public Thread Thrd;
        // Тут створюється семафор, що дає тільки два дозволи з двох початкових
        static Semaphore sem = new Semaphore(2, 2);
        public MyThread(string name)
        {
            Thrd = new Thread(this.Run);
            Thrd.Name = name;
            Thrd.Start();
        }
        // Точка входу в потік
        void Run()
        {
            Console.WriteLine(Thrd.Name + " чекає дозволу.");
            sem.WaitOne();
            Console.WriteLine(Thrd.Name + " отримує дозвіл.");
            for (char ch = 'A'; ch < 'D'; ch++)
            {
                Console.WriteLine(Thrd.Name + " : " + ch + " ");
                Thread.Sleep(500);
            }
            Console.WriteLine(Thrd.Name + " вивільняє дозвіл.");
            // Звільнити семафор
            sem.Release();
        }
    }
    class SemaphoreDemo
    {
        static void Main()
        {
            // Створити три потоки
            MyThread mt1 = new MyThread("Потік #1");
            MyThread mt2 = new MyThread("Потік #2");
            MyThread mt3 = new MyThread("Потік #3");
            mt1.Thrd.Join();
            mt2.Thrd.Join();
            mt3.Thrd.Join();
            Console.ReadKey();
        }
    }
}
```

```

Выбрать D:\МНТУ\For_КПИ-41\Паралельне г
Пот?к #1 чекає дозволу.
Пот?к #1 отримує дозв?л.
Пот?к #1 : А
Пот?к #2 чекає дозволу.
Пот?к #2 отримує дозв?л.
Пот?к #2 : А
Пот?к #3 чекає дозволу.
Пот?к #2 : В
Пот?к #1 : В
Пот?к #2 : С
Пот?к #1 : С
Пот?к #2 вив?льняє дозв?л.
Пот?к #3 отримує дозв?л.
Пот?к #3 : А
Пот?к #1 вив?льняє дозв?л.
Пот?к #3 : В
Пот?к #3 : С
Пот?к #3 вив?льняє дозв?л.

```

У класі MyThread оголошується семафор **sem**, як показано нижче.

```
static Semaphore sem = new Semaphore (2, 2);
```

При цьому створюється семафор, здатний дати не більше двох дозволів на доступ до ресурсу з двох спочатку наявних дозволів.

Зверніть увагу на те, що виконання методу MyThread. Run () не може бути продовжено до тих пір, поки семафор **sem** не дасть відповідний дозвіл. Якщо дозвіл відсутній, то виконання потоку припиняється. Коли ж дозвіл з'являється, виконання потоку поновлюється. У методі Main () створюються три потоки. Але виконуватися можуть тільки два перші потоки, а третій повинен чекати закінчення одного з цих двох потоків.

Семафор, створений в попередньому прикладі, відомий тільки тому процесу, який його створив. Але семафор можна створити і так, щоб він був відомий де-небудь ще. Для цього він повинен бути іменованим. Нижче наведені форми конструктора класу Semaphore, призначені для створення такого семафора.

```
public Semaphore(int initialCount, int maximumCount, string ім'я)  
public Semaphore(int initialCount, int maximumCount, string ім'я  
out bool createdNew)
```

У обох формах ім'я позначає конкретне ім'я, яке передається конструктору. Якщо в першій формі семафор, на який указує ім'я, ще не існує, то він створюється за допомогою значень, визначуваних параметрами initialCount і maximumCount.

А якщо він вже існує, то значення параметрів initialCount і maximumCount ігноруються. Після повернення з другої форми конструктора параметр createdNew матиме логічне значення true, якщо семафор був створений. В цьому випадку значення параметрів initialCount і maximumCount використовуються для створення семафора.

Якщо ж параметр createdNew матиме логічне значення **false**, значить, семафор вже існує і значення параметрів initialCount і maximumCount

ігноруються. Існує і третя форма конструктора класу Semaphore, в якій допускається указувати об'єкт керування доступом типу Semaphore Security. За допомогою іменованих семафорів можна синхронізувати взаємодію **процесів**.

Клас **SemaphoreSlim** є полегшеною версією класу Semaphore. SemaphoreSlim працює у рамках *одного процесу* і не використовує повний функціонал класу Semaphore, за рахунок цього працює він швидше.

Приклад 7. У наступному прикладі розглядається застосування об'єкту класу SemaphoreSlim замість Semaphore.

У методі Main ініціалізується семафор SemaphoreSlim. Початкове значення внутрішнього лічильника дорівнює 0, максимальне значення - 5. Робочі потоки блокуються, оскільки лічильник семафора дорівнює нулю. Головний потік збільшує лічильник на три одиниці, тим самим звільняючи три потоки. Після невеликої паузи головний потік звільняє ще два потоки.

```
namespace Lab5_SemaphorSlim
{
    class Program
    {
        private static SemaphoreSlim sem;
        private static void Worker(object num)
        {
            // Чекаємо сигналу
            sem.Wait();
            // Починаємо роботу
            Console.WriteLine("Worker {0} starting", num);
        }
        private static void Main()
        {
            // Максимальна місткість семафора : 5
            // Початковий стан: 0 (усі блокуються)
            sem = new SemaphoreSlim(0, 5);
            Thread[] workers = new Thread[10];
            for (int i = 0; i < workers.Length; i++)
            {
                workers[i] = new Thread(Worker);
                workers[i].Start(i);
            }
            Thread.Sleep(300);
            Console.WriteLine("Дозволяємо роботу трьом потокам");
            sem.Release(3);
            Thread.Sleep(200);
            Console.WriteLine("Дозволяємо роботу ще двом потокам");
            sem.Release(2);
        }
    }
}
```

```

file:///D:/MHTY/For_K-01/Lab5_SemaphorSlim/Lab5_SemaphorSlim/bin/Deb
Позволяємо роботу трьом потокам
Worker 4 starting
Worker 2 starting
Worker 0 starting
Позволяємо роботу ще двом потокам
Worker 6 starting
Worker 8 starting

```

Приклад 8. <http://inrecolan.ru/blog/viewpost/421>

У прикладі завантажуються п'ять сторінок з Інтернету і виводиться на екран розмір кожної з них. Для завантаження сторінок використовується не більше двох потоків одночасно. Реалізація наступна: створюємо семафор з вказівкою максимально можливої кількості потоків, між викликами методів Wait() і Release() розташовуємо виконуваний код, для якого нами вводилися обмеження (в даному випадку це отримання сторінки і відображення інформації на екрані після закінчення завантаження), в основному потоці відразу запускаємо усі допоміжні.

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Net;

namespace Semapher
{
    class Program
    {
        static readonly SemaphoreSlim s = new SemaphoreSlim(2);
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.Default;
            new List<string>()
            {
                "http://inrecolan.ru",
                "http://inrecolan.com",
                "http://google.com",
                "http://istu.edu.ua",
                "http://lib.istu.edu.ua"
            }.ForEach(url => new Thread(() => DownloadPage(url)).Start());
            Console.ReadKey();
        }

        static void DownloadPage(string url)
        {
            Console.WriteLine("Запит на завантаження сайту " + url);
            try
            {
                s.Wait();
                Console.WriteLine("Завантажується " + url);
                using (var wc = new WebClient())
                {
                    var content = wc.DownloadString(url);
                    Console.WriteLine("Завантажений сайт " + url + "Розмір " +
content.Length);
                }
            }
        }
    }
}

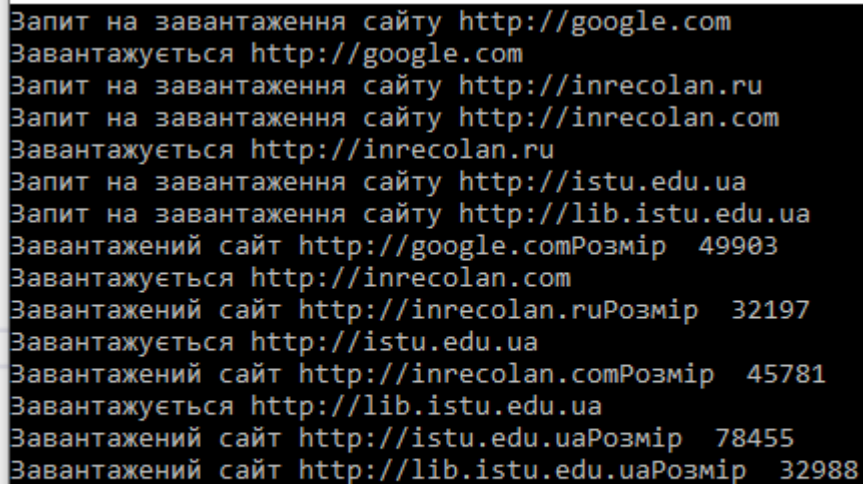
```

```

    }
    finally
    {
        s.Release();
    }
}
}
}

```

Результат



```

Запит на завантаження сайту http://google.com
Завантажується http://google.com
Запит на завантаження сайту http://inrecolan.ru
Запит на завантаження сайту http://inrecolan.com
Завантажується http://inrecolan.ru
Запит на завантаження сайту http://istu.edu.ua
Запит на завантаження сайту http://lib.istu.edu.ua
Завантажений сайт http://google.comРозмір 49903
Завантажується http://inrecolan.com
Завантажений сайт http://inrecolan.ruРозмір 32197
Завантажується http://istu.edu.ua
Завантажений сайт http://inrecolan.comРозмір 45781
Завантажується http://lib.istu.edu.ua
Завантажений сайт http://istu.edu.uaРозмір 78455
Завантажений сайт http://lib.istu.edu.uaРозмір 32988

```

Висновки

Класи **Monitor**, **Мьютекс**, **Семафор** призначені для синхронізації потоків.

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.

З моніторами пов'язані незвичайні аспекти. Наприклад, метод `Monitor.Wait()` дозволяє потоку тимчасово поступитися блокуванням іншому потоку, а потім отримати його назад. Система сигналів, які називають пульсаціями, служить для повідомлення початкового потоку про те, що блокування звільнене.

Схожим механізмом синхронізації доступу до спільних ресурсів є м'ютекси. М'ютекси відрізняються від моніторів тим, що можуть використовуватися з дескрипторами очікування. Вони можуть також блокуватися кілька разів. В цьому випадку вони мають бути розблоковані стільки ж разів, щоб блокування звільнилося.

М'ютекс призначений для тих ситуацій, в яких спільний ресурс може бути одночасно використаний тільки в **одному потоці**, тобто для надання монопольного доступу до ресурсу.

М'ютекси бувають двох типів: *локальні м'ютекси* (без імені), і *іменовані системні м'ютекси*. Локальний м'ютекс існує тільки усередині одного процесу. Він може використовуватися будь-яким потоком в процесі, який містить посилання на об'єкт **Mutex**, що представляє м'ютекс. Кожний неіменований об'єкт **Mutex** представляє окремий локальний м'ютекс.

Іменовані системні м'ютекси доступні в межах всієї операційної системи і можуть бути використані для синхронізації дій процесів. Можна створити об'єкт **Mutex**, що представляє іменований системний м'ютекс, використовуючи конструктор з підтримкою імен. Об'єкт операційної системи може бути створений в той же час, або існувати до створення об'єкту **Mutex**.

Крім монітору і м'ютексу для синхронізації потоків використовується семафор, який надає одночасний доступ до спільного ресурсу не одному, а декільком потокам. Тому семафор придатний для синхронізації цілого ряду ресурсів.

Контрольні запитання і завдання для самостійного виконання

1. Яке призначення м'ютекса? В яких випадках його слід використовувати?
2. Яке призначення методів WaitOne() і ReleaseMutex() ?
3. Яка різниця між іменованим і неіменованим м'ютексом?
4. Коли слід використовувати іменований м'ютекс?
5. Запустіть на виконання 2 екземпляри прикладу з прикладу 5 і проаналізуйте результати.
6. Які методи синхронізації реалізовані в класі Semaphore?
7. Для чого в семафорі використовується лічильник?
8. Яка різниця між такими примітивами синхронізації як семафор і м'ютекс?
9. Який метод класу Semaphore дозволяє отримати доступ до спільного ресурсу?
10. Який метод класу Semaphore дозволяє звільнити ресурс?

Деякі приклади взяті з ресурсу msdn.microsoft.com/ru-ru/:

<http://msdn.microsoft.com/ru-ru/library/system.threading.thread.isbackground.aspx>

<http://msdn.microsoft.com/ru-ru/library/4tssbxcw.aspx>

<http://msdn.microsoft.com/ru-ru/library/f55ddskf.aspx>

<http://inrecolan.ru/blog/viewpost/421>