Модуль 1. Багатопоточне програмування

Лекція 7. Інші методи синхронізації потоків

1. Атомарні оператори. Клас Interlocked

Бібліотека .NET 4.0 надає високоефективні *атомарні оператори*, які реалізовані як статичні методи класу **System.Threading.Interlocked.** Атомарні оператори призначені для *потокобезпечного неблокуючого виконання* операцій над даними, переважно цілочисельного типу.

Атомарність означає, що при виконанні оператора ніхто не втрутиться в роботу потоку. Функціонально, атомарні оператори рівносильні критичній секції, виділеній за допомогою lock, Monitor або інших засобів синхронізації.

Наприклад:

```
lock (sync_obj)
{
  counter++;
}
// можна виконати за допомогою атомарного оператора
Interlocked.Increment(ref counter);
```

Отже, клас Interlocked служить як альтернатива іншим засобам синхронізації, коли *потрібно тільки змінити значення спільної змінної*. Методи, доступні в класі Interlocked, гарантують, що їх дія виконуватиметься як одна безперервна операція. Це означає, що ніяка синхронізація в даному випадку взагалі не потрібна. У класі Interlocked надаються статичні методи для додавання двох цілих значень, інкрементування і декрементування цілого значення, порівняння і встановлення значень об'єкту, обміну об'єктами і отримання 64-розрядного значення. Всі ці операції виконуються без переривання.

Атомарні оператори є неблокуючими - потік не вивантажується і не чекає, тому вони забезпечують високу ефективність. Виконання операторів класу Interlocked займає удвічі менший час, ніж виконання критичної секції з lock –блокуванням без конкуренції.

Оператор	Метод	Типи даних
Збільшення лічильника на одиницю	Increment	Int32, Int64
Зменшення лічильника на одиницю	Decrement	Int32, Int64
Додавання	Add	Int32, Int64
Обмін значеннями	Exchange	Int32, Int64, double, single, object
Умовний обмін	CompareExchange	Int32, Int64, double, single, object

Читання 64-розрядного цілого	Read	Int64

У наведеному нижче прикладі програми демонструється застосування двох методів з класу Interlocked: **Increment ()** і **Decrement ().** При цьому використовуються наступні форми обох методів:

public static int Increment(ref int location)
public static int Decrement(ref int location)
де location — це змінна, яка підлягає збільшенню чи зменшенню.

Приклад 1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace Interlock
{
    class SharedRes
    {
        public static int Count = 0;
    }

    //У цьому потоці змінна SharedRes.Count збільшується.
    class IncThread
    {
        public Thread Thrd;
        public IncThread(string name)
        {
            Thrd = new Thread(this.Run);
            Thrd.Name = name;
            Thrd.Start();
        }
        // Точка входу в потік
        void Run()
        {
            for (int i = 0; i < 5; i++)
            {
                Interlocked.Increment(ref SharedRes.Count);
                Console.WriteLine(Thrd.Name + " Count = " + SharedRes.Count);
            }
        }
    }
    // У цьому потоці змінна SharedRes.Count зменшується.
    class DecThread
    {
        public Thread Thrd;
        public DecThread(string name)
        {
            Thrd = new Thread(this.Run);
            Thrd.Name = name;
            Thrd.Start();
        }
        // Точка входу в потік,
        void Run()
        {
            for (int i = 0; i < 5; i++)
            {
                Interlocked.Decrement(ref SharedRes.Count);
                Console.WriteLine(Thrd.Name + " Count = " + SharedRes.Count);
```

```
            }
        }
    }

    class InterlockedDemo
    {
        static void Main()
        {
            // Сконструювати два потоки.
            IncThread mt1 = new IncThread("Инкрементируюший Поток");
            DecThread mt2 = new DecThread("Декрементируюший Поток");
            mt1.Thrd.Join();
            mt2.Thrd.Join();
            Console.ReadKey();
        }
    }
}
```

Оператор Interlocked.CompareExchange дозволяє атомарно виконати конструкцію "перевірити-призначити" :

Приклад 2

```
lock(LockObj)
{
  if(x == curVal)
    x = newVal;
}
```

Критичну секцію можна замінити одним оператором:

```
oldVal = Interlocked.CompareExchange(ref x, newVal, curVal);
```

Якщо значення змінної **x** дорівнює значенню, що задається третім аргументом **curVal**, то змінній призначається значення другого аргументу **newVal**. Результат oldVal дозволяє встановити, чи здійснилася заміна значення.

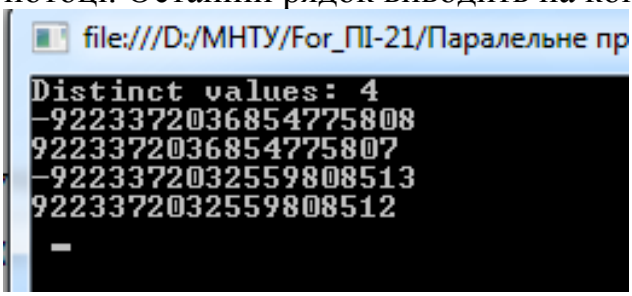
Атомарний оператор **Read** призначений для потокобезпечного читання 64-розрядних цілих чисел (Int64). Читання значень типу long (Int64) на 32-розрядній обчислювальній системі не є атомарною операцією на апаратному рівні. Тому багатопотокова робота з 64-розрядними змінними може призводити до некоректних результатів. У наступному фрагменті проілюстровано проблему читання змінних типу Int64:

Приклад 3.

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

namespace lab6
{
    class Program
    {
        static void Main(string[] args)
        {
           Int64 bigInt = Int64.MinValue;
           Thread t = new Thread(() =>
           {
               while (true)
               {
```

```
                    if (bigInt == Int64.MinValue)
                        bigInt = Int64.MaxValue;
                    else
                        bigInt = Int64.MinValue;
                }
            });
        t.Start();
        List<Int64> lstBig = new List<Int64>();
        for(int i=0; i < 100; i++)
         {
          Thread.Sleep(100);
          lstBig.Add(bigInt);
         }
       t.Abort();
         Console.WriteLine("Distinct values: " + lstBig.Distinct().Count());
         lstBig.Distinct().AsParallel().ForAll(Console.WriteLine);
         Console.ReadKey();
        }
    }
}
```

В цьому прикладі значення змінної **bigInt** змінюється тільки в одному потоці. Основний потік періодично читає поточні значення **bigInt**. Потік t циклічно змінює значення змінної **bigInt** з **MinValue** на **MaxValue** і з **MaxValue** на **MinValue**. Проте, результат виконання показує, що основний потік прочитав і інші значення. Ці "проміжні" значення з'явилися через не атомарність дій над 64-розрядними змінними - доки основний потік прочитав перші 32 біта числа, дочірній потік змінив наступні 32 біта. Передостанній рядок виводить число різних значень змінної bigInt, прочитаних в основному потоці. Останній рядок виводить на консоль усі різні значення.

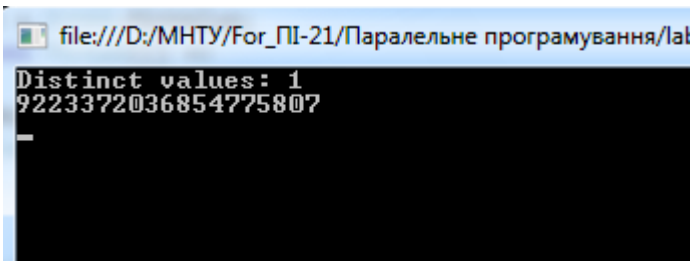


Для усунення проблеми необхідно зробити атомарним запис і читання змінної bigInt:

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

namespace lab6
{
    class Program
    {
        static void Main(string[] args)
        {
           Int64 bigInt = Int64.MinValue;
           Thread t = new Thread(() =>
           {
               Int64 oldValue = Interlocked.CompareExchange(ref bigInt,Int64.MinValue,
Int64.MaxValue);
```

```
                Interlocked.CompareExchange(ref bigInt,Int64.MaxValue, oldValue);

            });
         t.Start();
         List<Int64> lstBig = new List<Int64>();
         for(int i=0; i < 100; i++)
          {
           Thread.Sleep(100);
           lstBig.Add(Interlocked.Read(ref bigInt));
          }
        t.Abort();
          Console.WriteLine("Distinct values: " + lstBig.Distinct().Count());
          lstBig.Distinct().AsParallel().ForAll(Console.WriteLine);
          Console.ReadKey();
         }
    }
}
```

Зміна **bigInt** реалізована за допомогою двох операторів CompareExchange. Перший оператор намагається призначити **MinValue**, якщо поточне значення рівне **MaxValue**. Оператор повертає старе значення. Порівнюючи поточне із старим значенням, визначаємо, чи сталася зміна. Якщо зміни не було, то призначаємо максимальне значення **MaxValue**. Атомарне читання реалізоване за допомогою оператора **Interlocked.Read**. Виведення результатів свідчить про вирішення проблеми:



Операції над 64 розрядними цілими на 64-розрядній системі є атомарними на апаратному рівні, тому не вимагають засобів синхронізації при паралельному записі і читанні. Але при паралельному записі в декількох потоках, виникає проблема гонки даних.

2. Сигнальні повідомлення

Сигнальні повідомлення дозволяють реалізувати різні схеми синхронізації, як взаємне виключення, так і умовну синхронізацію. При умовній синхронізації потік блокується в очікуванні події, яка генерується в іншому потоці. Платформа .NET надає три типи сигнальних повідомлень: AutoResetEvent, ManualResetEvent і ManualResetEventSlim, а також шаблони синхронізації, побудовані на сигнальних повідомленнях (CountdownEvent, Barrier). Перші два типи побудовано на об'єкті ядра операційної системи. Третій тип ManualResetEventSlim є полегшеною версією об'єкту ManualResetEvent, є продуктивнішим.

У наступному фрагменті два потоки використовують один і той самий об'єкт типу ManualResetEvent. Перший потік виводить повідомлення від

другого потоку. Повідомлення записується у змінну, що розділяється. Виклик методу WaitOne блокує перший потік в очікуванні сигналу від другого потоку. Сигнал генерується при виклику методу Set.

Приклад 4.

```
void OneThread(object o)
{
  ManualResetEvent mre = (ManualResetEvent)o;
  mre.WaitOne();
  Console.WriteLine("Data from thread #2: " + data);
}
void SecondThread(object o)
{
  ManualResetEvent mre = (ManualResetEvent)o;
  Console.WriteLine("Writing data");
  data = "BBBBBB";
  mre.Set();
}
```

Результат:

```
Writing data..
Data from thread#2: BBBB
```

Різниця між AutoResetEvent і ManualResetEvent полягає в режимі скидання статусу сигнальної події: *автоматичне (auto reset)* або *ручне (manual reset)*. Сигнал з автоматичним скиданням знімається відразу ж після звільнення потоку, блокованого викликом WaitOne. Сигнал з ручним скиданням не знімається до тих пір, поки який-небудь потік не викличе метод **Reset**.

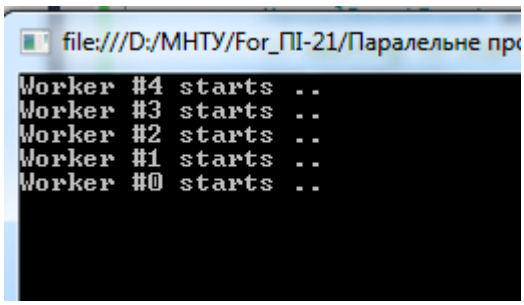
У наступному фрагменті розглядаються відмінності сигнальних повідомлень. Управляючий потік Manager запускає п'ять робочих потоків і кожному передає один і той самий сигнальний об'єкт. Робочі потоки чекають сигналу від управляючого потоку.

Приклад 5

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace lab7_1
{
  class Program
    {
     static void Worker(object initWorker)
       {
         string name = ((object[])initWorker)[0] as string;
         ManualResetEvent mre = ((object[])initWorker)[1] as ManualResetEvent;
         // Waiting to start work
         mre.WaitOne();
         Console.WriteLine("Worker {0} starts ..", name);
         // useful work
        }
        static void Manager()
        {
            int nWorkers = 5;
            Thread[] worker = new Thread[nWorkers];
            ManualResetEvent mre = new ManualResetEvent(false);
```

```
            for (int i = 0; i < nWorkers; i++)
            {
                worker[i] = new Thread(Worker);
                worker[i].Start(new object[] { "#" + i, mre });
            }
            // preparing data in shared variables for workers
            // let start work
            mre.Set();
        }
         static void Main(string[] args)
        {
            Manager();
            Console.ReadKey();
        }
    }
}
```

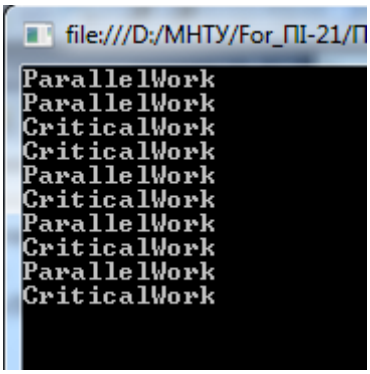
При встановленні події **mre** роботу почнуть усі очікуючі робочі потоки. При заміні об'єкту на AutoResetEvent подія скидатиметься автоматично і "упіймає" його тільки якийсь один потік. Таким чином, об'єкт AutoResetEvent можна використовувати для реалізації взаємно виключного доступу.



Приклад 6

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace lab6_1
{
  class Program
    {
      static void ThreadFunc(object o)
      {
          var lockEvent = o as AutoResetEvent;
          ParallelWork();

          lockEvent.WaitOne();
          CriticalWork();
          lockEvent.Set();
      }
      static void ParallelWork()
      {
          Console.WriteLine("ParallelWork");
      }
      static void CriticalWork()
      {
          Console.WriteLine("CriticalWork");
      }
      static void Main()
      {
```

```
            Thread[] workers = new Thread[5];
            for (int i = 0; i < 5; i++)
                workers[i] = new Thread(ThreadFunc);
            var lockEvent = new AutoResetEvent(true);
            for (int i = 0; i < 5; i++)
                workers[i].Start(lockEvent);
            Console.ReadKey();
        }
    }
}
```



В цьому прикладі п'ять робочих потоків частину роботи можуть виконувати паралельно, але якийсь фрагмент повинні виконувати послідовно (критична секція). Повідомлення типу AutoResetEvent використовується для організації взаємно-виключного доступу до критичної секції. Об'єкт ініціалізувався зі встановленим сигналом для того, щоб спочатку роботи один з потоків увійшов до критичної секції. При завершенні виконання критичної секції потік дає сигнал одному з очікуючих потоків. Порядок входження потоків в критичну секцію, також як і при використанні об'єкту Monitor, не визначений.

Об'єкт ManualResetEventSlim функціонально відповідає сигнальній події з ручним скиданням. Застосування гібридного блокування підвищує продуктивність в сценаріях з малим часом очікування. Виклик методу Wait протягом обмеженого проміжку часу зберігає потік в активному стані (циклічна перевірка статусу сигналу), якщо сигнал не поступив, то здійснюється виклик дескриптора очікування ядра операційної системи.

Об'єкти AutoResetEvent, ManualResetEvent, а також об'єкти Semaphore, Mutex походять від об'єкту, що інкапсулює дескриптор очікування ядра WaitHandle. Тип WaitHandle містить корисні статичні методи очікування декількох об'єктів синхронізації ядра:

```
var ev1 = new ManualResetEvent(false);
var ev2 = new ManualResetEvent(false);
new Thread(SomeFunc).Start(ev1);
new Thread(SomeFunc).Start(ev2);
// Чекаємо усі сигнали
WaitHandle.WaitAll(new ManualResetEvent[] {ev1, ev2});
ev1.Reset(); ev2.Reset();
// Чекаємо хоч б один сигнал
int iFirst = WaitHandle.WaitAny(new ManualResetEvent[]
{ev1, ev2});
```

3. Шаблони синхронізації - Barrier, CountdownEvent

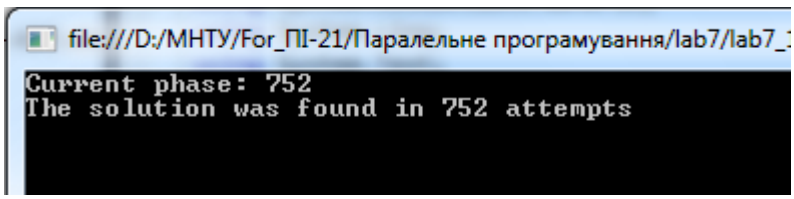
Бар'єр - це примітив синхронізації, який визначається користувачем і дозволяє декільком потокам (які називаються учасниками) паралельно здійснювати поетапну роботу з алгоритмом. Кожен учасник виконується до досягнення точки бар'єру в коді. Бар'єр означає закінчення одного етапу роботи. Коли учасник досягає бар'єру, він блокується до тих пір, поки усі учасники не досягнуть цього бар'єру. Коли усі учасники досягли бар'єру, при необхідності можна викликати дію наступного етапу. Це дія може використовуватися для виконання дій одним потоком, поки усі інші потоки все ще залишаються блокованими. Після виконання дії усі учасники розблоковуються.

Мета наступної програми в підрахунку кількості ітерацій (чи етапів), необхідних для двох потоків, щоб кожен знайшов свою половину рішення на одному і тому ж етапі, використовуючи алгоритм перемішування слів випадковим чином. Після того, як кожен потік перемішає свої слова, операція наступного етапу бар'єру порівнює два результати для перевірки, чи стоять в правильному порядку слова в отриманому реченні.

Приклад 7. Barrier

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
namespace lab7_1
{
    class Program
    {
        static string[] words1 = new string[] { "brown", "jumped", "the", "fox", "quick" };
        static string[] words2 = new string[] { "dog", "lazy", "the", "over" };
        static string solution = "the quick brown fox jumped over the lazy dog.";

        static bool success = false;
        static Barrier barrier = new Barrier(2, (b) =>
        {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < words1.Length; i++)
            {
                sb.Append(words1[i]);
                sb.Append(" ");
            }
            for (int i = 0; i < words2.Length; i++)
            {
                sb.Append(words2[i]);

                if (i < words2.Length - 1)
                    sb.Append(" ");
            }
            sb.Append(".");
#if TRACE
            System.Diagnostics.Trace.WriteLine(sb.ToString());
#endif
            Console.CursorLeft = 0;
            Console.Write("Current phase: {0}", barrier.CurrentPhaseNumber);
            if (String.CompareOrdinal(solution, sb.ToString()) == 0)
            {
                success = true;
```

```
                Console.WriteLine("\r\nThe solution was found in {0} attempts",
barrier.CurrentPhaseNumber);
            }
        });

        static void Main(string[] args)
        {

            Thread t1 = new Thread(() => Solve(words1));
            Thread t2 = new Thread(() => Solve(words2));
            t1.Start();
            t2.Start();

            // Keep the console window open.
            Console.ReadLine();
        }

        // Use Knuth-Fisher-Yates shuffle to randomly reorder each array.
        // For simplicity, we require that both wordArrays be solved in the same phase.
        // Success of right or left side only is not stored and does not count.
        static void Solve(string[] wordArray)
        {
            while (success == false)
            {
                Random random = new Random();
                for (int i = wordArray.Length - 1; i > 0; i--)
                {
                    int swapIndex = random.Next(i + 1);
                    string temp = wordArray[i];
                    wordArray[i] = wordArray[swapIndex];
                    wordArray[swapIndex] = temp;
                }

                // We need to stop here to examine results
                // of all thread activity. This is done in the post-phase
                // delegate that is defined in the Barrier constructor.
                barrier.SignalAndWait();
            }
        }
    }

}
```



Бар' єр є об'єктом, що запобігає продовженню виконання окремих завдань в паралельному режимі, поки усі завдання не досягнуть бар'єру.Така функція корисна при виникненні паралельного виконання на етапах і необхідності синхронізації кожного етапу між завданнями. В даному прикладі на операцію відводиться два етапи. На першому етапі кожне завдання заповнює даними свою частину буфера. Після закінчення заповнення своєї частини завдання сигналізує бар'єру, що готова до продовження роботи, і переходить в режим очікування. Після того, як усі завдання відправили сигнали бар'єру, вони розблоковуються, і починається другий етап. Бар'єр потрібний, оскільки для другого етапу вимагається, щоб кожне завдання мало доступ до усіх даних, створених до цього моменту. Без бар'єру перші виконувані завдання можуть

спробувати виконати читання з буферів, які ще не заповнені іншими завданнями. Таким чином можна синхронізувати будь-яку кількість етапів.

CountdownEvent

System.Threading.CountdownEvent - це примітив синхронізації, що розблоковує очікуючі потоки після отримання певного числа сигналів. CountdownEvent призначений для сценаріїв, в яких без нього довелося б використовувати ManualResetEvent або ManualResetEventSlim і вручну зменшувати значення змінної-лічильника перед сигналізацією події. Наприклад, в сценарії розгалуження і з'єднання можна просто створити CountdownEvent з лічильником сигналів, рівним 5, а потім запустити п'ять робочих елементів в пулі потоків, викликаючи після завершення роботи в кожному робочому елементі метод Signal. Кожен виклик методу Signal зменшує значення лічильника на 1. У головному потоці виклик методу Wait призведе до блокування, яке триватиме до тих пір, поки значення лічильника не дорівнюватиме нулю.

Приклад 8. CountdownEvent

https://msdn.microsoft.com/ru-ru/library/dd997365(v=vs.110).aspx

```
IEnumerable<Data> source = GetData();
using (CountdownEvent e = new CountdownEvent(1))
{
    // fork work:
    foreach (Data element in source)
    {
        // Dynamically increment signal count.
        e.AddCount();
        ThreadPool.QueueUserWorkItem(delegate(object state)
         {
             try
             {
                 ProcessData(state);
             }
             finally
             {
                 e.Signal();
             }
         },
         element);
    }
    e.Signal();

    // The first element could be run on this thread.

    // Join with work.
    e.Wait();
}
// .,.
```

4. Взаємодія процесів

Багатозадачність на основі потоків найчастіше організовується при програмуванні на C#. Але там, де це доречно, можна організувати і багатозадачність на основі *процесів*. В цьому випадку замість запуску іншого потоку в одній і тій самій програмі одна програма починає виконання іншої.

При програмуванні на C# це робиться за допомогою класу **Process**, визначеного в просторі імен System.Diagnostics.

Простий спосіб запустити інший процес — скористатися методом **Start(),** визначеним в класі Process. Нижче наведена одна з найпростіших форм цього методу:

public static Process Start(string і'мя_файла)

де і'мя_файла позначає конкретне ім'я файлу, який повинен виконуватися або ж пов'язаний з виконуваним файлом.

Коли створений процес завершується, слід викликати метод Close (), щоб звільнити пам'ять, виділену для цього процесу. Нижче наведена форма оголошення методу Close ().

public void Close ()

Процес може бути перерваний двома способами. Якщо процес є програмою Windows з графічним інтерфейсом, то для переривання такого процесу викликається метод CloseMainWindow (), форма якого наведена нижче.

public bool CloseMainWindow()

Цей метод посилає процесу повідомлення, яке наказує йому зупинитися. Він повертає логічне значення true, якщо повідомлення отримане, і логічне значення false, якщо програма не має графічного інтерфейсу або головного вікна. Слід враховувати, що метод CloseMainWindow () служить тільки для запиту зупинки процесу. Якщо програма проігнорує такий запит, то вона не буде перервана як процес.

Для безумовного переривання процесу слід викликати метод Kill ():

public void Kill()

Але методом Kill () слід користуватися акуратно, оскільки він приводить до неконтрольованого переривання процесу. Будь-які незбережені дані, пов'язані з процесом, що переривається, будуть втрачені.

Для того, щоб організувати очікування завершення процесу, можна скористатися методом WaitForExit (). Нижче наведені дві його форми.

public void WaitForExit()

public bool WaitForExit(int мілісекунд)

У першій формі очікування продовжується до тих пір, поки процес не завершиться, а в другій формі — тільки протягом вказаної кількості мілісекунд. У останньому випадку метод WaitForExit () повертає логічне значення **true**, якщо процес завершився, і логічне значення **false**, якщо він все ще виконується.

У наведеному нижче прикладі програми демонструється створення, очікування і закриття процесу. У цій програмі спочатку запускається стандартна програма Windows: текстовий редактор WordPad.exe, а потім організовується очікування завершення програми WordPad як процесу.

Приклад 9.

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

```
using System.Diagnostics;

namespace Proc
{
    class StartProcess
    {
      static void Main()
      {
      Process newProc = Process.Start("wordpad.exe");
      Console.WriteLine("Новий процес запущений.");
      newProc.WaitForExit();
      newProc.Close(); // звільнити виділені ресурси
      Console.WriteLine("Новий процес завершений.");
      Console.ReadKey();
      }
    }
}
```

При виконанні цієї програми запускається стандартний застосунок WordPad, і на екрані з'являється повідомлення "Новий процес запущений. ". Потім програма чекає закриття WordPad. Після закінчення роботи WordPad на екрані з'являється завершальне повідомлення "Новий процес завершений.".

Висновки

В лекціях 5, 6 і 7 ми розглянули різні механізми організації взаємодії між потоками, які реалізовані в .NET. Найпростішим способом синхронізації є використання блокування (lock).

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.

Альтернативою розглянутим засобам синхронізації є клас **Interlocked**, який застосовується коли *потрібно тільки змінити значення спільної змінної*.

Крім цих засобів синхронізації в C# є інші, простіші засоби: сигнальні повідомлення, примітиви синхронізації. У багатьох випадках їх використання спрощує взаємодію потоків.

В C# є також *багатозадачність* на основі *процесів*. В цьому випадку замість запуску іншого потоку в одній і тій самій програмі, одна програма починає виконання іншої. При програмуванні на C# це робиться за допомогою класу **Process**, визначеного в просторі імен System.Diagnostics.

Контрольні запитання і завдання для самостійного виконання

1. Яке призначення класу Interlocked? Які методи збільшують і зменшують спільні змінні?
2. Який метод класу Interlocked реалізує умовний обмін значеннями?
3. Яке призначення сигнальних повідомлень? Як вони працюють?
4. В якому випадку краще використовувати сигнальні повідомлення замість інших засобів синхронізації?
5. Яка різниця між сигнальними повідомленням AutoResetEvent, ManualResetEvent і ManualResetEventSlim?
6. Що таке Бар’єр? Як його створити?

7. Яка різниця між такими шаблонами синхронізації як Barrier і CountdownEvent ? В якому випадку який з них краще використовувати?
8. Яке призначення класу Process? Визначити за допомогою msdn основні члени цього класу і їхнє призначення.
9. Як запустити і закрити процес?
10. Яке призначення методу Kill ()?
11. Як організувати очікування завершення процесу, а не потоку?

Вправи:

1. Дослідіть ефективність полегшених засобів синхронізації в порівнянні з аналогічними об'єктами ядра операційної системи: SemaphoreSlim - Semaphore, ManualResetEvent - ManualResetEventSlim.
Для аналізу можна використовувати завдання звернення до лічильника, що розділяється :

```
voidThreadFunc() {
// Вхід в критичну секцію з допомогою того або іншого засобу синхронізації
totalCount++;
// Вихід з критичної секції
}
```

2. Дослідіть ефективність атомарних операторів в порівнянні із засобами організації критичної секції (lock, Monitor, Mutex).