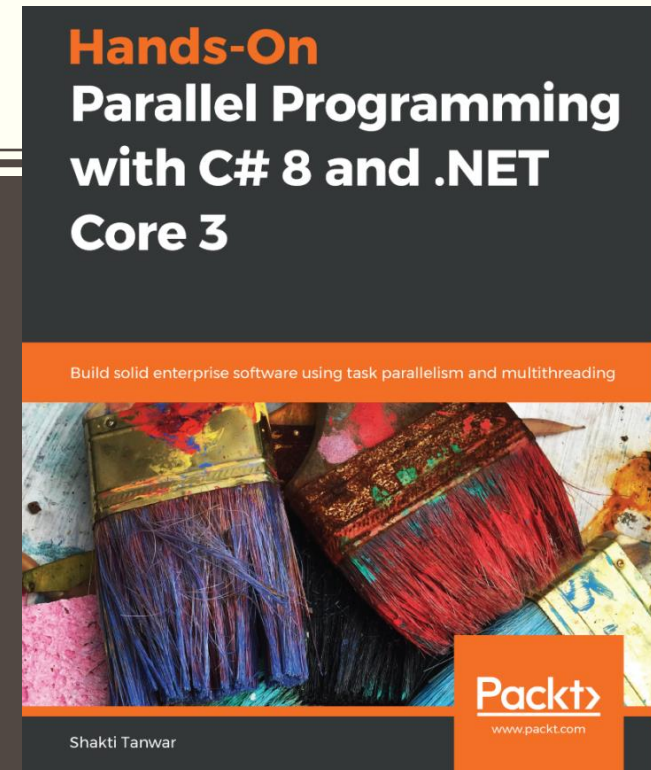


КОНКУРЕНТНІ КОЛЕКЦІЇ ДАНИХ

Питання 13.2. (глава 6)



Вступ до конкурентних колекцій

- Синхронізаційні примітиви складно реалізовувати.
 - Часто спільним ресурсом є колекція, яку зчитують/записують багато потоків.
 - Оскільки доступ до колекції може відбуватись багатьма способами (зокрема, за допомогою Enumerate, Read, Write, Sort або Filter), ускладнюється написання власної колекції з керованою синхронізацією за рахунок примітивів.
 - Звідси впливає потреба в потокобезпечних колекціях.
- Починаючи з .NET Framework 4, було додано багато потокобезпечних колекцій.
 - Було додано простір імен System.Threading.Concurrent, який містить такі конструкції:
 - IProducerConsumerCollection<T>
 - BlockingCollection<T>
 - ConcurrentDictionary<TKey,TValue>
- Використовуючи ці типи, не потрібно додаткової синхронізації, а зчитування/оновлення даних може здійснюватись автоматично.

Вступ до конкурентних колекцій

- Навіть для старих колекцій на зразок ArrayList чи Hashtable представлялась властивість Synchronized, яка дозволяла отримувати доступ до колекцій у потокобезпечній манері.
 - Проте це відбувалось із значною втратою продуктивності, оскільки для потокобезпечності вся колекція огорталась замком на кожну операцію зчитування/запису.
- Конкурентні колекції є легковаговими обгортками синхронізаційних примітивів, зокрема SpinLock, SpinWait, SemaphoreSlim, CountdownEvent тощо, менше навантажуючи процесорні ядра.
 - Spin-синхронізація набагато ефективніша за блокуючу синхронізацію при малих періодах очікування.
 - З наявністю вбудованих алгоритмів при зростанні періодів очікування більш легкі замки перетворюються в замки рівня ядра (kernel locks).

Колекція `IProducerConsumerCollection<T>`

- Колекції для задачі «виробник-споживач» надають ефективні альтернативи без замків у порівнянні з узагальненими відповідниками на зразок `Stack<T>` та `Queue<T>`.
 - Довільна колекція для виробництва чи споживання повинна дозволяти користувачу додавати/видаляти свої елементи.
 - .NET Framework забезпечує інтерфейс `IProducerConsumerCollection<T>`, який представляє потокобезпечні стеки, черги та множини (bags).
- Класи, які реалізують цей інтерфейс:
 - `ConcurrentQueue<T>`
 - `ConcurrentStack<T>`
 - `ConcurrentBag<T>`
- Метод протоколює 2 важливих методи: `TryAdd` і `TryTake`.
 - Синтаксис `TryAdd`: `bool TryAdd (T item);`
 - Метод додає елемент та повертає `true`. Якщо існують проблеми зі вставкою, метод поверне `false`.
 - Синтаксис `TryTake`: `bool TryTake (out T item);`
 - Метод видаляє елемент та повертає `true`. Якщо існують проблеми з видаленням, метод поверне `false`.

Колекція `ConcurrentQueue<T>`

- Конкурентні черги можуть використовуватись, щоб програмно реалізувати сценарії «виробник-споживач».
 - У даному шаблоні один або кілька потоків виробляють дані, а також один чи кілька потоків споживають їх.
 - Проблему гонитви даних зазвичай вирішують за допомогою:
 - Черг `Queue<T>`
 - Черг `ConcurrentQueue<T>`
- Враховуючи, який потік відповідає за додавання/споживання даних, шаблони «виробник-споживач» можна класифікувати як:
 - Чистий (Pure) шаблон «виробник-споживач», в якому потік може лише виробляти або лише споживати дані.
 - Змішаний (Mixed) шаблон «виробник-споживач», в якому довільний потік може бути одночасно і виробником, і споживачем.

Застосування черг для реалізації задачі «виробник-споживач»

- Буде кілька задач (tasks), які намагатимуться зчитувати чи записувати дані в чергу.
 - Потрібно забезпечити атомарність операцій:
 - 1. Створимо чергу та заповнимо її деякими даними:
 - 2. Оголосимо змінну, яка міститиме остаточний результат: `int sum = 0;`
 - 3. Далі створимо паралельний цикл, який зчитуватиме елемент з черги, виконуючи кілька задач (tasks) та додаючи результати в потокобезпечній манері і зберігаючи їх у змінній `sum`:

```
Queue<int> queue = new Queue<int>();  
for (int i = 0; i < 500; i++) {  
    queue.Enqueue(i);  
}
```



```
C:\Program Files\dotnet\dotnet.exe  
Calculated Sum is 125599 and should be 124750
```

```
Parallel.For(0, 500, (i) =>  
{  
    int localSum = 0;  
    int localValue;  
    while (queue.TryDequeue(out localValue))  
    {  
        Thread.Sleep(10);  
        localSum += localValue;  
    }  
    Interlocked.Add(ref sum, localSum);  
});  
Console.WriteLine($"Calculated Sum is {sum} and should be  
{Enumerable.Range(0, 500).Sum()}");
```

Застосування черг для реалізації задачі «виробник-споживач»

- Для забезпечення потокобезпечності введемо критичну секцію в паралельному циклі:

```
Parallel.For(0, 500, (i) =>
{
    int localSum = 0;
    int localValue;
    Monitor.Enter(_locker);
    while (cq.TryDequeue(out localValue))
    {
        Thread.Sleep(10);
        localSum += localValue;
    }
    Monitor.Exit(_locker);
    Interlocked.Add(ref sum, localSum);
});
```

- Аналогічно слід синхронізувати всі точки зчитування/запису в черзі для більш складних сценаріїв.
- Вивід:

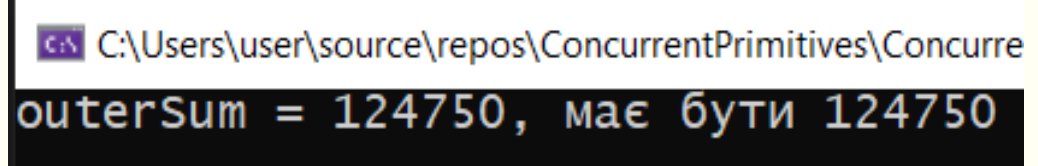


C:\Program Files\dotnet\dotnet.exe
Calculated Sum is 124750 and should be 124750

Розв'язання задач з використанням конкурентних черг

```
private static void ProducerConsumerUsingConcurrentQueues()
{
    // Create a Queue.
    ConcurrentQueue<int> cq = new ConcurrentQueue<int>();
    // Populate the queue.
    for (int i = 0; i < 500; i++)
    {
        cq.Enqueue(i);
    }
    int sum = 0;
    Parallel.For(0, 500, (i) =>
    {
        int localSum = 0;
        int localValue;
        while (cq.TryDequeue(out localValue))
        {
            Thread.Sleep(10);
            localSum += localValue;
        }
        Interlocked.Add(ref sum, localSum);
    });
    Console.WriteLine($"outerSum = {sum}, має бути {Enumerable.Range(0, 500).Sum()}");
}
```

- Застосуємо потокобезпечну реалізацію черги – клас `System.Collections.Concurrent.ConcurrentQueue`.



```
C:\Users\user\source\repos\ConcurrentPrimitives\Concurre
outerSum = 124750, має бути 124750
```

- Замінімо `Queue<int>` на `ConcurrentQueue<int>` в попередньому коді, який мав накладні витрати на синхронізацію.
- Використовуючи `ConcurrentQueue`, нам не потрібно переживати за інші примітиви синхронізації.

Продуктивність: Queue<T> vs ConcurrentQueue<T>

- Краще застосовувати ConcurrentQueue замість черг у таких сценаріях:
 - Чистий сценарій «виробник-споживач»: час обробки кожного елементу дуже низький.
 - Чистий сценарій «виробник-споживач»: існує лише один виділений потік-виробник та лише один потік-споживач.
 - Чистий чи змішаний сценарій «виробник-споживач»: продуктивність операції перевищує 500 FLOPS
- Слід застосовувати черги замість конкурентних черг у змішаних сценаріях «виробник-споживач», коли продуктивність операцій над кожним елементом не дуже велика, проте нижча 500 FLOPS.

Застосування ConcurrentStack<T>

- ConcurrentStack<T> – конкурентна версія Stack<T>, яка реалізує інтерфейс IProducerConsumerCollection<T>.
 - Можемо вставляти (push) чи видаляти (pop) елементи стеку в стилі Last In, First Out (LIFO).
 - Не включає блокування на рівні ядра (kernel-level locking), скоріше, покладається на spin-блокування та CAS-операції (compare-and-swap), щоб видалити будь-яку конкуренцію.
- Важливі методи класу:
 - **Clear**: очищає всі елементи колекції
 - **Count**: повертає кількість елементів у колекції
 - **IsEmpty**: повертає true, якщо колекція порожня
 - **Push (T item)**: додає елемент у колекцію
 - **TryPop (out T result)**: видаляє елемент з колекції та повертає true, якщо елемент успішно видалено; інакше – false
 - **PushRange (T[] items)**: додає перелік елементів у колекцію; операція здійснюється автоматично
 - **TryPopRange (T[] items)**: видаляє частину елементів колекції

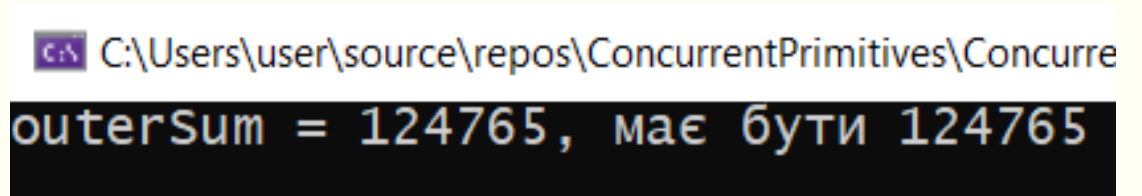
Створення конкурентного стеку

```
private static void ProducerConsumerUsingConcurrentStack()
{
    // Create a Queue.
    ConcurrentStack<int> concurrentStack = new ConcurrentStack<int>();
    // Populate the queue.
    for (int i = 0; i < 500; i++)
    {
        concurrentStack.Push(i);
    }
    concurrentStack.PushRange(new[] { 1, 2, 3, 4, 5 });
    int sum = 0;
    Parallel.For(0, 500, (i) =>
    {
        int localSum = 0;
        int localValue;
        while (concurrentStack.TryPop(out localValue))
        {
            Thread.Sleep(10);
            localSum += localValue;
        }
        Interlocked.Add(ref sum, localSum);
    });
    Console.WriteLine($"outerSum = {sum}, має бути 124765");
}
```

- Створимо конкурентний стек та заповнимо його елементами.

- Отримувати елементи стеку можна так:

- int localValue;
concurrentStack.TryPop(out localValue);
concurrentStack.TryPopRange (new[] { 1,2,3,4,5});



```
C:\Users\user\source\repos\ConcurrentPrimitives\ConcurrentPrimitives\bin\Debug\net6.0\ConcurrentPrimitives.exe
outerSum = 124765, має бути 124765
```

Використання ConcurrentBag<T>

- ConcurrentBag<T> - невідсортована колекція, оптимізована для ситуацій, у яких одні потоки виступають як виробниками, так і споживачами.
 - ConcurrentBag підтримує work-stealing-алгоритм та локальну чергу для кожного потоку.
 - Наступний код створює ConcurrentBag та вставляє/отримує його елементи:

```
ConcurrentBag<int> concurrentBag = new ConcurrentBag<int>();  
//Add item to bag  
concurrentBag.Add(10);  
int item;  
//Getting items from Bag  
concurrentBag.TryTake(out item);
```

```
Item is 3
Item is 2
Item is 1
Item is 4
Item is 5
Item is 6
```

Використання ConcurrentBag<T>

```
static ConcurrentBag<int> concurrentBag = new ConcurrentBag<int>();
private static void ConcurrentBagDemo()
{
    ManualResetEventSlim manualResetEvent = new ManualResetEventSlim(false);
    Task producerAndConsumerTask = Task.Factory.StartNew(() =>
    {
        for (int i = 1; i <= 3; ++i)
            concurrentBag.Add(i);
        //Allow second thread to add items
        manualResetEvent.Wait();
        while (concurrentBag.IsEmpty == false)
        {
            int item;
            if (concurrentBag.TryTake(out item))
                Console.WriteLine($"Item is {item}");
        }
    });
    Task producerTask = Task.Factory.StartNew(() =>
    {
        for (int i = 4; i <= 6; ++i)
            concurrentBag.Add(i);
        manualResetEvent.Set();
    });
}
```

■ Кожний потік має свою локальну чергу.

- Елементи 1, 2 та 3 додано в локальну чергу producerAndConsumerTask, а елементи 4, 5 і 6 – у локальну чергу producerTask.
- Коли producerAndConsumerTask вставив елементи, очікуємо на завершення вставки елементів producerTask.
- Як тільки всі елементи вставлено, producerAndConsumerTask починає отримувати їх.
- Оскільки спочатку в локальну чергу додавались елементи 1, 2, 3, він оброблятиме їх першими до переходу в локальну чергу producerTask.

Використання BlockingCollection<T>

- Потокобезпечна колекція, що реалізує інтерфейс `IProduceConsumerCollection<T>`.
 - Можемо додавати чи видаляти елементи з колекції конкурентно, не турбуючись про синхронізацію.
- Матимемо 2 потоки: виробника та споживача.
 - Потік-виробник формуватиме дані, причому ми можемо обмежувати максимальну вироблених кількість елементів до того, як потік-виробник перейде в сплячий режим та заблокується.
 - Потік-споживач буде споживати дані та блокуватись, коли колекція ставатиме порожньою.
 - Потік-виробник розблоковано, коли потік-споживач видаляє певні елементи з колекції.
 - Потік-споживач розблоковується, коли потік-виробник додає деякі елементи в колекцію.
- Важливі аспекти блокуючих колекцій:
 - **Обмеження (*Bounding*)**: можемо обмежити колекцію максимальним розміром, після якого жодних нових об'єктів не можна додати, а потік-виробник переходить у режим сну (sleep mode).
 - **Блокування (*Blocking*)**: можемо блокувати потік-споживач, коли колекція порожня.

Створення BlockingCollection<T>

```
BlockingCollection<int> blockingCollection = new BlockingCollection<int>(10);
Task producerTask = Task.Factory.StartNew(() =>
{
    for (int i = 0; i < 5; ++i)
    {
        blockingCollection.Add(i);
    }
    blockingCollection.CompleteAdding();
});
Task consumerTask = Task.Factory.StartNew(() =>
{
    while (!blockingCollection.IsCompleted)
    {
        int item = blockingCollection.Take();
        Console.WriteLine($"Item retrieved is {item}");
    }
});
Task.WaitAll(producerTask, consumerTask);
```

```
Item retrieved is 0
Item retrieved is 1
Item retrieved is 2
Item retrieved is 3
Item retrieved is 4
```

- У коді створюється `BlockingCollection` на максимум 10 елементів, після чого переходить у заблокований стан перед тим, як елементи споживаються потоками-споживачами.
- Додавати елементи в колекцію можна так:
 - `blockingCollection.Add(1);`
 - `blockingCollection.TryAdd(3, TimeSpan.FromSeconds(1));`
- Варіанти видалення елементів колекції:
 - `int item = blockingCollection.Take();`
 - `blockingCollection.TryTake(out item, TimeSpan.FromSeconds(1));`
- Потік-виробник викликає метод `CompleteAdding()`, коли більше немає елементів для вставки.
 - Даний метод задає властивості `IsAddingComplete` колекції значення `true`.
- Потік-споживач використовує властивість `IsCompleted`, коли колекція порожня, а `IsAddingComplete` також `true`.
 - Це вказує на те, що всі елементи було оброблено, а виробник не додаватиме нових.

Сценарій з багатьма виробниками і споживачами

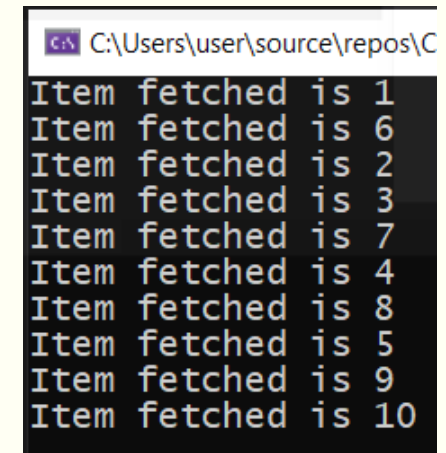
```
BlockingCollection<int>[] produceCollections = new BlockingCollection<int>[2];  
produceCollections[0] = new BlockingCollection<int>(5);  
produceCollections[1] = new BlockingCollection<int>(5);
```

```
Task producerTask1 = Task.Factory.StartNew(() => {  
    for (int i = 1; i <= 5; ++i) {  
        produceCollections[0].Add(i);  
        Thread.Sleep(100);  
    }  
    produceCollections[0].CompleteAdding();  
});  
Task producerTask2 = Task.Factory.StartNew(() => {  
    for (int i = 6; i <= 10; ++i) {  
        produceCollections[1].Add(i);  
        Thread.Sleep(200);  
    }  
    produceCollections[1].CompleteAdding();  
});
```

```
while (!produceCollections[0].IsCompleted || !produceCollections[1].IsCompleted)  
{  
    int item;  
    BlockingCollection<int>.TryTakeFromAny(produceCollections, out item, TimeSpan.FromSeconds(1));  
    if (item != default(int))  
        Console.WriteLine($"Item fetched is {item}");  
}
```

■ Для простоти створимо 2 виробників та одного споживача.

- Потоки-виробники формуватимуть елементи.
- Як тільки всі потоки-виробники викликали CompleteAdding(), споживач розпочне зчитування елементів колекції:
 - 1. Створюємо блокуючу колекцію з кількома виробниками.
 - 2. Далі створюємо 2 задачі-виробники, які додаватимуть елементи до виробників.
 - 3. Потім записуємо логіку споживача, який намагатиметься споживати елементи з обох колекцій виробників, як тільки такі елементи будуть доступними.



```
C:\Users\user\source\repos\C  
Item fetched is 1  
Item fetched is 6  
Item fetched is 2  
Item fetched is 3  
Item fetched is 7  
Item fetched is 4  
Item fetched is 8  
Item fetched is 5  
Item fetched is 9  
Item fetched is 10
```


Використання ConcurrentDictionary<TKey,TValue>

```
ConcurrentDictionary<int, string> concurrentDictionary = new ConcurrentDictionary<int, string>();
Task producerTask1 = Task.Factory.StartNew(() => {
    for (int i = 0; i < 20; i++) {
        Thread.Sleep(100);
        concurrentDictionary.TryAdd(i, (i * i).ToString());
    }
});

Task producerTask2 = Task.Factory.StartNew(() => {
    for (int i = 10; i < 25; i++) {
        concurrentDictionary.TryAdd(i, (i * i).ToString());
    }
});

Task producerTask3 = Task.Factory.StartNew(() => {
    for (int i = 15; i < 20; i++) {
        Thread.Sleep(100);
        concurrentDictionary.AddOrUpdate(i, (i * i).ToString(),
                                         (key, value)
                                         => (key * key).ToString());
    }
});

Task.WaitAll(producerTask1, producerTask2);
Console.WriteLine("Ключами є {0} ", string.Join(", ",
concurrentDictionary.Keys.Select(c => c.ToString()).ToArray()));
```

- ConcurrentDictionary<TKey,TValue> представляє потокобезпечний словник.
 - Використовується для вміщення пар «ключ-значення», які можна потокобезпечно зчитувати чи записувати.
- У коді створюємо 2 потоки-виробники, які додаватимуть елементи в словник.
 - Виробники створюватимуть дубльовані елементи, а словник забезпечуватиме їх потокобезпечну вставку без викидання помилок щодо повторюваних ключів.
 - Як тільки потік-виробник завершує роботу, споживач зчитуватиме всі елементи, використовуючи keys або values



ДЯКУЮ ЗА УВАГУ!

Наступне питання: Асинхронні потоки