

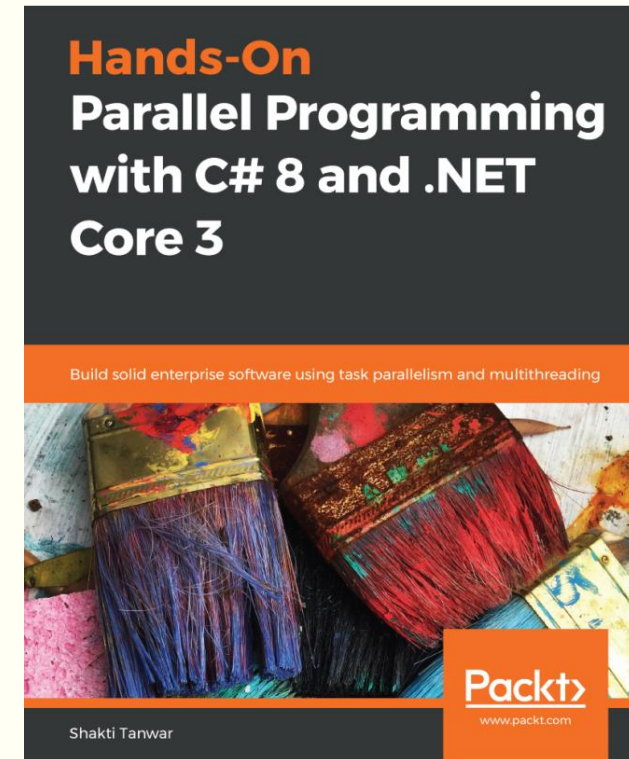


ДОДАТКОВІ ПИТАННЯ КОНКУРЕНТНОГО ВИКОНАННЯ КОДУ

Лекція 13
Об'єктно-орієнтоване програмування

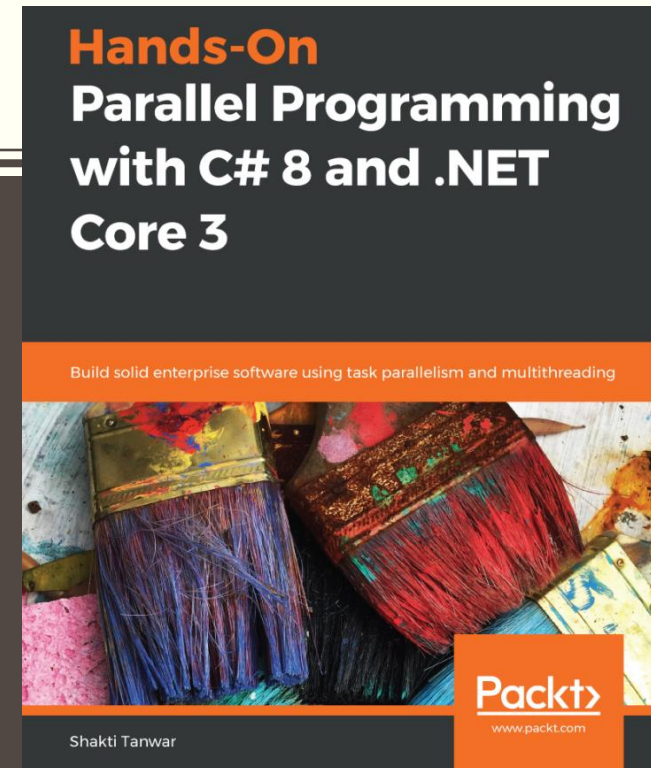
План лекції

- Потоки: синхронізація та планування.
- Конкурентні колекції даних.
- Асинхронні потоки.
- Основи реактивного програмування.



ПОТОКИ: СИНХРОНІЗАЦІЯ ТА ПЛАНУВАННЯ

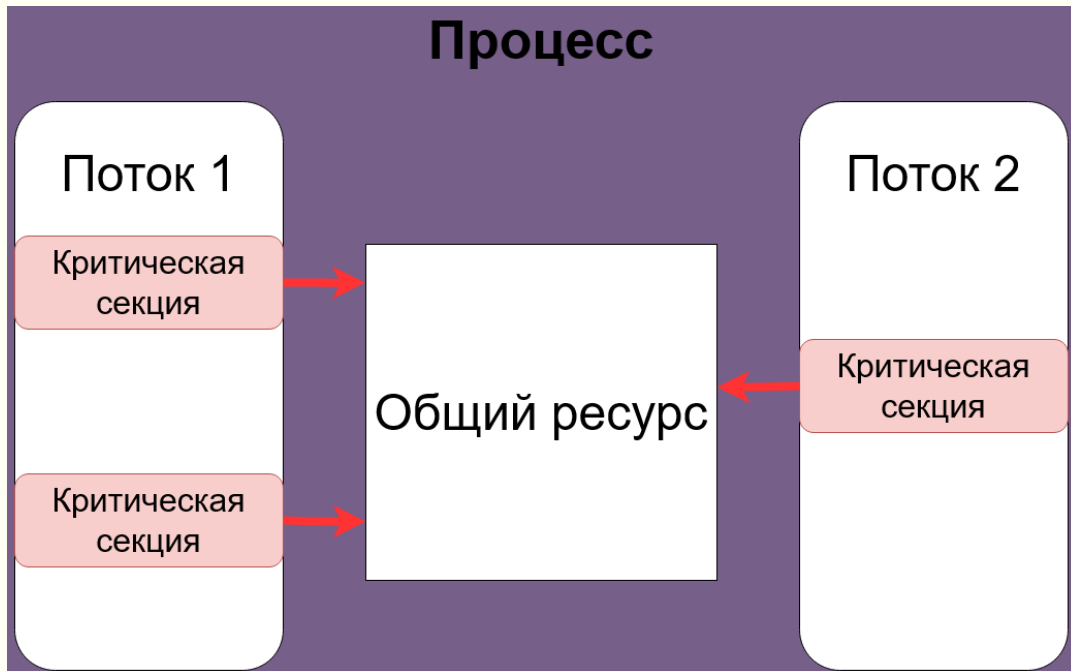
Питання 13.1. (глава 5)



Зміст запитання

- При розбитті роботи на задачі, які будуть виконуватись різними виконавчими елементами (work items) виникає потреба синхронізувати результати від кожного потоку.
 - Раніше розглядали поняття thread-local-storage та partition-local-storage, які можуть застосовуватись, щоб до деякої міри обійти проблему синхронізації.
 - Проте все ще потрібно синхронізувати потоки на запис даних у спільну пам'ять та щоб виконувати операції вводу-виводу.
- У питанні розглядатимемо:
 - Примітиви синхронізації
 - З'єднуючі (Interlocked) операції
 - Примітиви блокування (Locking primitives)
 - Примітиви сигналізування (Signaling primitives)
 - Легковагові (Lightweight) примітиви синхронізації
 - Бар'єри та події зі зворотним відліком (countdown events)

Примітиви синхронізації



- **Критична секція** – частина ходу виконання потоку, яка повинна бути захищена від конкурентного доступу, щоб підтримувати певні інваріанти.
 - Сама по собі вона не є примітивом синхронізації, проте базується на них.
- Примітиви синхронізації – прості програмні механізми, що забезпечуються операційною системою та впроваджують багатозадачність у ядрі ОС.
 - Зсередини вони використовують низькорівневі атомарні операції, а також бар'єри пам'яті (memory barriers).
 - Користувачу не потрібно власноруч реалізовувати замки (locks) та бар'єри пам'яті.
 - Популярні примітивами синхронізації: замки (блокування, locks), м'ютекси (mutexes), умовні змінні (conditional variables) та семафори.
 - Монітор (monitor) – високорівневий засіб синхронізації, який зсередини застосовує примітиви синхронізації.

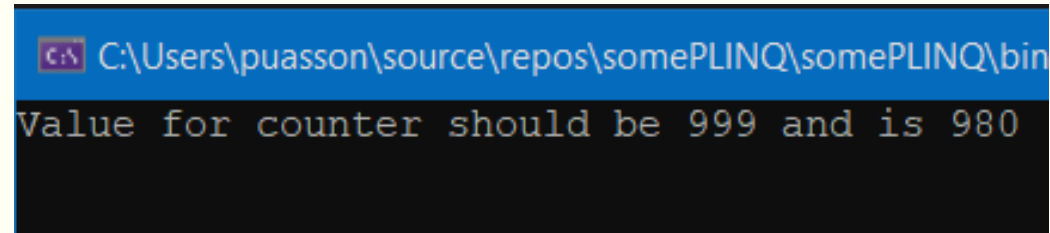
Синхронізаційні примітиви

- У .NET Framework присутній цілий ряд примітивів синхронізації, щоб мати справу зі взаємодією між потоками та уникати потенційних станів гонитви.
- Примітиви синхронізації в мові C# можна поділити на 5 категорій:
 - З'єднуючі (Interlocked) операції
 - Блокування (Locking)
 - Сигналізування (Signaling)
 - Легковагові (Lightweight) види синхронізації
 - SpinWait

З'єднуючі (Interlocked) операції

- Клас `Interlocked` інкапсулює примітиви синхронізації та використовується для здійснення атомарних операцій над змінними, спільними для потоків.
 - Він постачає методи на зразок `Increment`, `Decrement`, `Add`, `Exchange`, та `CompareExchange`.
 - Приклад без використання цього класу:

```
int _counter = 0;
Parallel.For(1, 1000, i =>
{
    Thread.Sleep(100);
    _counter++;
});
Console.WriteLine($"Value for counter should be 999 and is { _counter }");
```



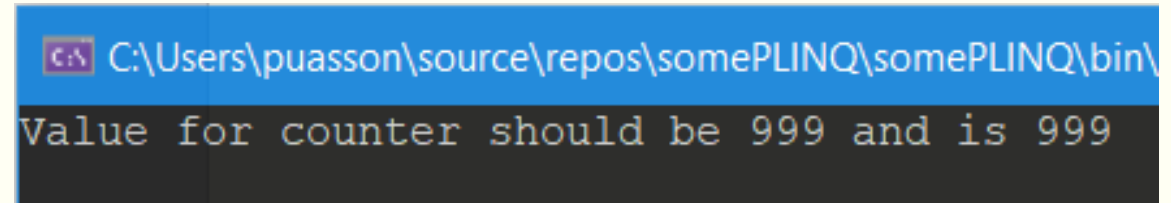
```
C:\Users\puasson\source\repos\somePLINQ\somePLINQ\bin
Value for counter should be 999 and is 980
```

- Відмінність результатів пояснюється станом гонитви між потоками, який виникає через бажання потоку зчитати значення зі змінної, значення якої було записано, проте ще не зафіксовано (committed).

З'єднуючі (Interlocked) операції

- Застосуємо клас Interlocked, щоб забезпечити потокобезпечність:

```
int _counter = 0;
Parallel.For(1, 1000, i =>
{
    Thread.Sleep(100);
    Interlocked.Increment(ref _counter);
});
Console.WriteLine($"Value for counter should be 999 and is { _counter }");
```



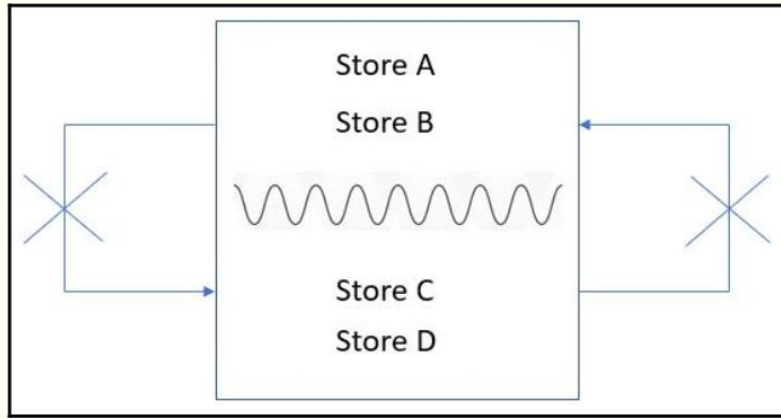
```
C:\Users\puasson\source\repos\somePLINQ\somePLINQ\bin\
Value for counter should be 999 and is 999
```

- Аналогічно, можемо застосувати Interlocked.Decrement(ref _counter), щоб потокобезпечно декрементувати значення.
- Повний список операцій:
 - Interlocked.Increment(ref _counter); // _counter стає 1
 - Interlocked.Decrement(ref _counter); // _counter стає 0
 - Interlocked.Add(ref _counter, 2); // Add: _counter стає 2
 - Interlocked.Add(ref _counter, -2); // Subtract: _counter стає 0
 - Console.WriteLine(Interlocked.Read(ref _counter)); // зчитує 64-бітне поле
 - Console.WriteLine(Interlocked.Exchange(ref _counter, 10)); // замінює значення _counter на 10
 - Console.WriteLine(Interlocked.CompareExchange(ref _counter, 100, 10));
//перевіряє, чи _counter = 10; якщо так, замінити на 100
 - Два методи було додано в .NET Framework 4.5: Interlocked.MemoryBarrier() та Interlocked.MemoryBarrierProcessWide().

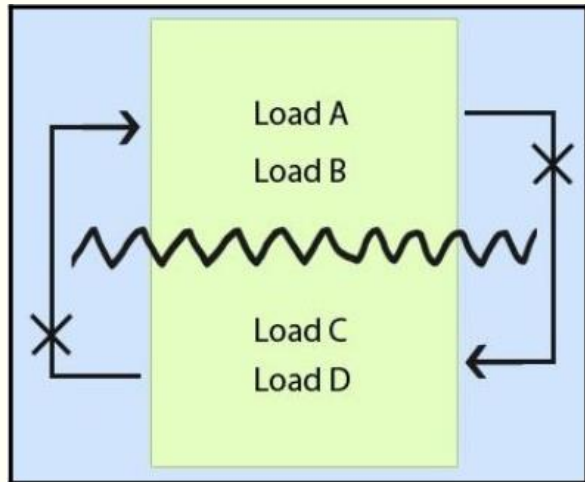
Бар'єри пам'яті в .NET

- Моделі багатопоточності по-різному працюють на одно- та багатоядерних системах.
 - На одноядерних системах лише 1 потік отримує ЦП, поки інші - чекатимуть. Це забезпечує правильний порядок доступу до пам'яті (для завантаження та зберігання) потоком. Дану модель також називають моделлю послідовної узгодженості (sequential consistency model).
 - В багатоядерних системах кілька потоків працюють конкурентно. Послідовна узгодженість не гарантується, оскільки або апаратне забезпечення, або JIT-компілятор можуть перевпорядкувати інструкції роботи з пам'яттю (memory instructions), щоб покращити продуктивність (кешування, спекулятивне завантаження чи затримки операцій збереження).
- Приклад load speculation: $a=b$; приклад операції збереження: $c=1$.
 - Інструкції (statements) для завантаження та зберігання, які виконує компілятор, не завжди працюють в порядку їх записування в коді.
 - Компілятор може обирати порядок виконання інструкцій для покращення продуктивності при роботі багатьох потоків над одним кодом.
 - Перевпорядкування (reordering) коду – проблема для багатоядерних процесорів зі слабкими моделями пам'яті; на одноядерні системи воно не впливає.
 - Код реструктурується так, щоб інший потік міг отримати переваги чи зберегти інструкцію, що вже в пам'яті.
 - Потрібні **бар'єри пам'яті (memory barrier)**, щоб гарантувати коректність перевпорядкування коду.

Бар'єри пам'яті забезпечують неможливість переходу бар'єру інструкціями до або після нього



Store (write) memory barrier



Load (read) memory barrier

15.03.2021

■ Існує 3 типи бар'єрів пам'яті:

- **Бар'єр на запис (store (write) memory barrier):** жодна операція збереження даних не може перейти бар'єр. Не впливає на операції завантаження, вони все ще можуть перевпорядковуватись. Еквівалентна ЦП-інструкція – SFENCE.
- **Бар'єр на зчитування (Load (read) memory barrier):** забороняє перехід бар'єру операціями завантаження (зчитування). Еквівалентна ЦП-інструкція – LFENCE.
- **Повний бар'єр (Full memory barrier):** забезпечує впорядкування, не пропускаючи операції зчитування/запису через бар'єр. ЦП-еквівалент – інструкція MFENCE. Поведінка такого бар'єру часто реалізується такими синхронізаційними конструкціями .NET:
 - Task.Start, Task.Wait та Task.Continuation
 - Thread.Sleep, Thread.Join, Thread.SpinWait,
 - Thread.VolatileRead та Thread.VolatileWrite
 - Thread.MemoryBarrier
 - Lock, Monitor.Enter та Monitor.Exit
 - Операції з класу Interlocked

■ Половинні бар'єри забезпечуються ключовим словом volatile та методами класу Volatile.

- .NET Framework постачає вбудовані паттерни, застосовуючи volatile-поля в класах, зокрема, Lazy<T> та LazyInitializer.

Уникнення перевогопрядкування коду за допомогою Thread.MemoryBarrier

```
static int a = 1, b = 2, c = 0;
private static void BarrierUsingTheadBarrier()
{
    b = c;
    Thread.MemoryBarrier();
    a = 1;
}
```

- Thread.MemoryBarrier створює повний бар'єр.
 - Він огортається всередині методу Interlocked.MemoryBarrier, тому можна переписати код так:

```
private static void BarrierUsingInterlockedBarrier()
{
    b = c;
    Interlocked.MemoryBarrier();
    a = 1;
}
```

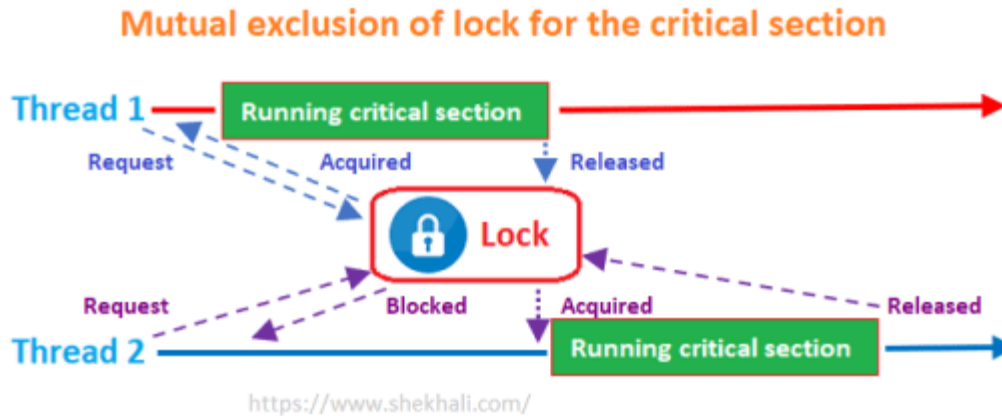
Уникнення переопрацювання коду

- За потреби створення бар'єру на рівні процесу чи системи можемо застосувати `Interlocked.MemoryBarrierProcessWide`, представлений у .NET Core 2.0.
 - Це обгортка над `FlushProcessWriteBuffer` Windows API або `sys_membarrier` з ядра Linux:

```
private static void BarrierUsingInterlockedProcessWideBarrier()  
{  
    b = c;  
    Interlocked.MemoryBarrierProcessWide();  
    a = 1;  
}
```

- Тут створено бар'єр на рівні процесу.

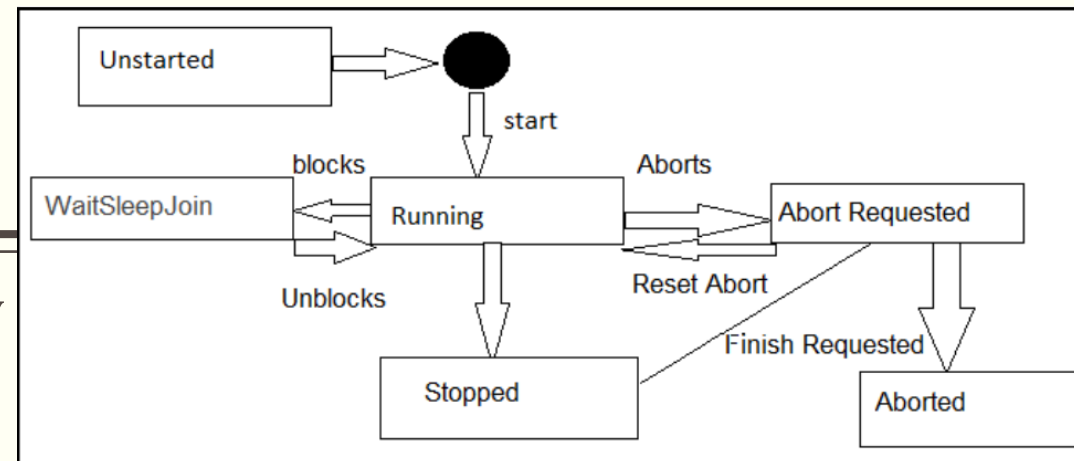
Блокуючі (locking) примітиви



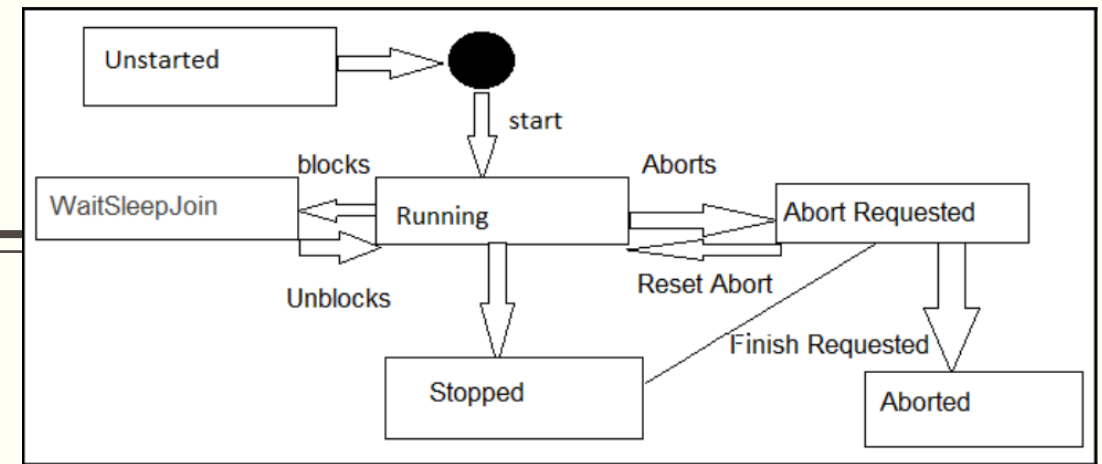
- Замки (Locks) можуть застосовуватись для обмеження доступу до захищеного ресурсу лише одним потоком чи групою потоків.
 - Для ефективної реалізації блокування потрібно ідентифікувати критичні секції, що потребують захисту.
- При застосуванні замка до спільного ресурсу виконуються такі кроки:
 - 1. Потік чи група потоків отримують доступ до спільного ресурсу, відкриваючи (acquire, отримуючи ключ) замок.
 - 2. Інші потоки, що не можуть отримати доступ, переходять у стан очікування.
 - 3. Як тільки замок буде закрито (ключ звільниться), він передається іншому потоку, який почне виконуватись.

Стани потоків

- У будь-який момент життєвого циклу потоку можна отримати його стан за допомогою властивості `ThreadState`.
- Потік може бути в одному з наступних станів:
 - **Незапущений (*Unstarted*)**: потік створено в CLR, проте метод `System.Threading.Thread.Start` ще не викликався.
 - **Запущений (*Running*)**: потік почав роботу в результаті виклику `Thread.Start()`. Він не очікує на заплановані операції.
 - ***WaitSleepJoin***: потік у заблокованому стані в результаті виклику потоком методів `Wait()`, `Sleep()` або `Join()`.
 - ***StopRequested***: потоку надійшов запит на зупинку виконання.
 - **Зупинений (*Stopped*)**: потік зупинив виконання.
 - ***AbortRequested***: метод `Abort()` викликано для потоку, проте переривання виконання ще не сталося, оскільки потік очікує на `ThreadAbortException`, який спробує здійснити переривання.
 - ***Aborted***: потік перервав виконання.
 - ***SuspendRequested***: потоку надійшов запит на призупинку (`suspend`) в результаті виклику методу `Suspend()`.
 - **Призупинений (*Suspended*)**: потік було призупинено.
 - **Фоновий (*Background*)**: потік виконується в фоні.



Стани потоків



- Коли CLR створює потік, він перебуває в стані Unstarted, а при виклику методу Thread.Start() переходить у запущений стан.
- Із запущеного стану потік може перейти в наступні стани: WaitSleepJoin; AbortRequested; Stopped.
 - Потік називають заблокованим, коли він знаходиться в стані WaitSleepJoin.
 - Виконання заблокованого потоку призупиняється (pause), оскільки він очікує на виконання певних зовнішніх умов, які можуть бути результатом виконання деякої CPU-bound операції вводу-виводу чи деякого іншого потоку.
 - Заблокований потік негайно звільняє відповідний ЦП-ресурс (time slice) та не використовує процесор, поки не буде задоволена блокуюча умова.
 - Блокування та розблокування накладає затрати продуктивності, оскільки вимагають перемикання контексту (context switching).
- Розблокування потоку може статись через такі події:
 - задоволена блокуюча умова;
 - викликано Thread.Interrupt() для заблокованого потоку;
 - виконання потоку перервано за допомогою методу Thread.Abort();
 - після закінчення встановленого часу (таймауту).

Блокування (Blocking) vs циклічне блокування (spinning)

- Зabloкований потік поступається (relinquish) процесорним ресурсом на певний період часу.
 - Це підвищує продуктивність, роблячи цей ресурс доступним іншим потокам, проте накладає затрати на перемикавання контексту.
 - Такий підхід корисний тоді, коли потік має блокуватись протягом суттєвого періоду часу.
 - Якщо час очікування невеликий, краще використовувати циклічне блокування без поступок процесорним часом.
- Наприклад, код для нескінченного циклу: `while(!done);`
 - Коли очікування завершується, змінній присвоюється значення `false`, і цикл переривається.
 - Хоч це і даремні витрати процесорного часу, можливе суттєве покращення продуктивності, якщо цикл триватиме відносно недовго.
 - .NET Framework постачає деякі спеціальні конструкції, зокрема `SpinWait`, `SpinLock` та ін.

Замки, м'ютекси та семафори

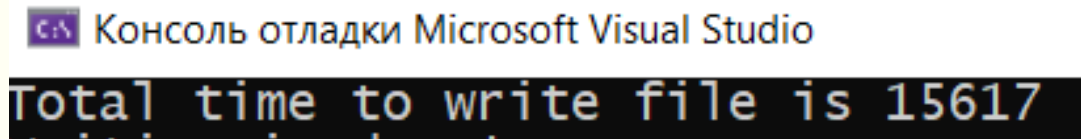
Synchronization Primitive	Allotted No. of Threads	Cross Process
Lock	1	×
Mutex	1	✓
Semaphore	Many	✓
SemaphoreSlim	Many	×

- **Замки та м'ютекси** – це блокуючі конструкції, які дозволяють лише 1 потоку мати доступ до захищеного ресурсу.
 - Замок – скорочена реалізація, яка використовує інший клас для високорівневої синхронізації – Monitor.
- **Семафор** – блокуюча конструкція, яка дозволяє заданій кількості потоків отримати доступ до захищеного ресурсу.
 - Замок може синхронізувати доступ тільки всередині процесу, проте для системного ресурсу чи спільної пам'яті необхідно синхронізувати доступ між кількома процесами.
 - М'ютекс дозволяє синхронізувати доступ до ресурсів між процесами, надаючи **замок рівня ядра (kernel-level lock)**.
- Класи Lock і Mutex дозволяють лише однопоточний доступ до спільних ресурсів, а Semaphore і SemaphoreSlim – багатопоточний.
 - Якщо Lock та SemaphoreSlim працюють лише всередині процесу, Mutex і Semaphore мають замок рівня процесу (process-wide lock).

Замок (Lock, блокування)

- Розглянемо код, який намагається записати число в текстовий файл:

```
var range = Enumerable.Range(1, 1000);
Stopwatch watch = Stopwatch.StartNew();
for (int i = 0; i < range.Count(); i++)
{
    Thread.Sleep(10);
    File.AppendAllText("test.txt", i.ToString());
}
watch.Stop();
Console.WriteLine($"Total time to write file is { watch.ElapsedMilliseconds}");
```



Консоль отладки Microsoft Visual Studio

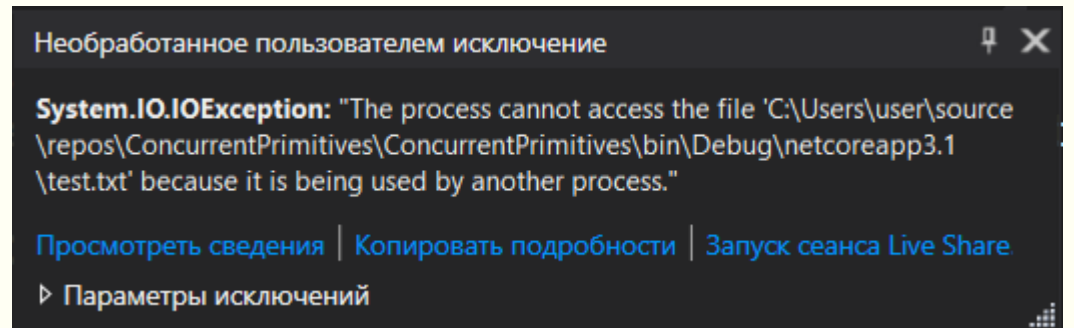
Total time to write file is 15617

- задача скомпонована з 1000 робочих елементів (work items) і кожний елемент виконується близько 10мс.
- Загальний час затримок досягає 10000мс.
- Також слід враховувати час на виконання вводу-виводу, тому загальний час виконання буде 15617.

Lock

- Спробуємо розпаралелити задачу за допомогою методів розширення `AsParallel()` і `AsOrdered()`:

```
var range = Enumerable.Range(1, 1000);
Stopwatch watch = Stopwatch.StartNew();
range.AsParallel().AsOrdered().ForAll(i =>
{
    Thread.Sleep(10);
    File.AppendAllText("test.txt", i.ToString());
});
watch.Stop();
Console.WriteLine($"Total time to write file is { watch.ElapsedMilliseconds}");
```



- Файл є спільним ресурсом з критичною секцією, тому дозволяє лише атомарні операції.
- У паралельному коді багато потоків намагаються записати в файл дані, що спричиняє виняткову ситуацію.
- Необхідно забезпечити як швидке виконання паралельного коду, так і підтримку атомарності запису.
- Додамо інструкцію `lock`.

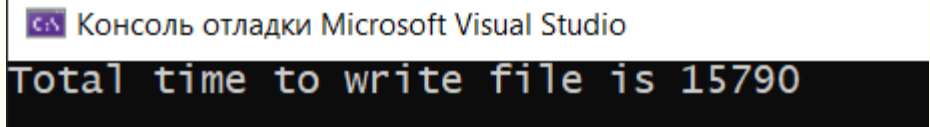
Lock

```
static object _locker = new object();
static void Main(string[] args)
{
    var range = Enumerable.Range(1, 1000);
    Stopwatch watch = Stopwatch.StartNew();

    range.AsParallel().AsOrdered().ForAll(i =>
    {
        lock (_locker)
        {
            Thread.Sleep(10);
            File.WriteAllText("test.txt", i.ToString());
        }
    });
    watch.Stop();
    Console.WriteLine($"Total time to write file is { watch.ElapsedMilliseconds}");

    Console.ReadKey();
}
```

- Спочатку оголосимо статичне поле типу object (потрібний посилальний тип, оскільки замок застосовується на пам'ять з кучі).
 - Далі включаємо замок у методі ForAll().
 - При запуску коду не отримуємо винятки, проте час виконання перевищує час послідовного виконання коду.
 - Замок забезпечує атомарність: лише одному потоку дозволено доступ до вразливого коду, проте маємо накладні витрати на блокування.
 - Такий замок називають дурним (dumb lock).



Консоль отладки Microsoft Visual Studio

Total time to write file is 15790

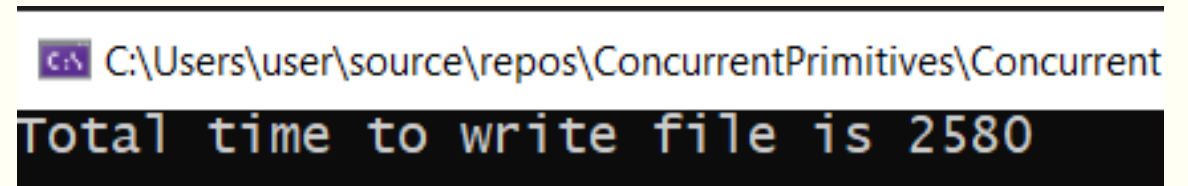
Lock

```
static object _locker = new object();
static void Main(string[] args)
{
    var range = Enumerable.Range(1, 1000);
    Stopwatch watch = Stopwatch.StartNew();

    range.AsParallel().AsOrdered().ForAll(i =>
    {
        Thread.Sleep(10);
        lock (_locker)
        {
            File.WriteAllText("test.txt", i.ToString());
        }
    });
    watch.Stop();
    Console.WriteLine($"Total time to write file is { watch.ElapsedMilliseconds}");

    Console.ReadKey();
}
```

- Внесемо зміни, блокуючи лише критичну секцію.
 - Отримуємо значну перевагу від комбінування синхронізації з паралелізмом.
 - Подібний результат можна отримати, використовуючи інші примітиви блокування (locking primitive), зокрема клас Monitor.



The screenshot shows a Windows command prompt window with a file path and a result line. The file path is C:\Users\user\source\repos\ConcurrentPrimitives\Concurrent. The result line, displayed in a large, bold, yellow font on a black background, states: Total time to write file is 2580.

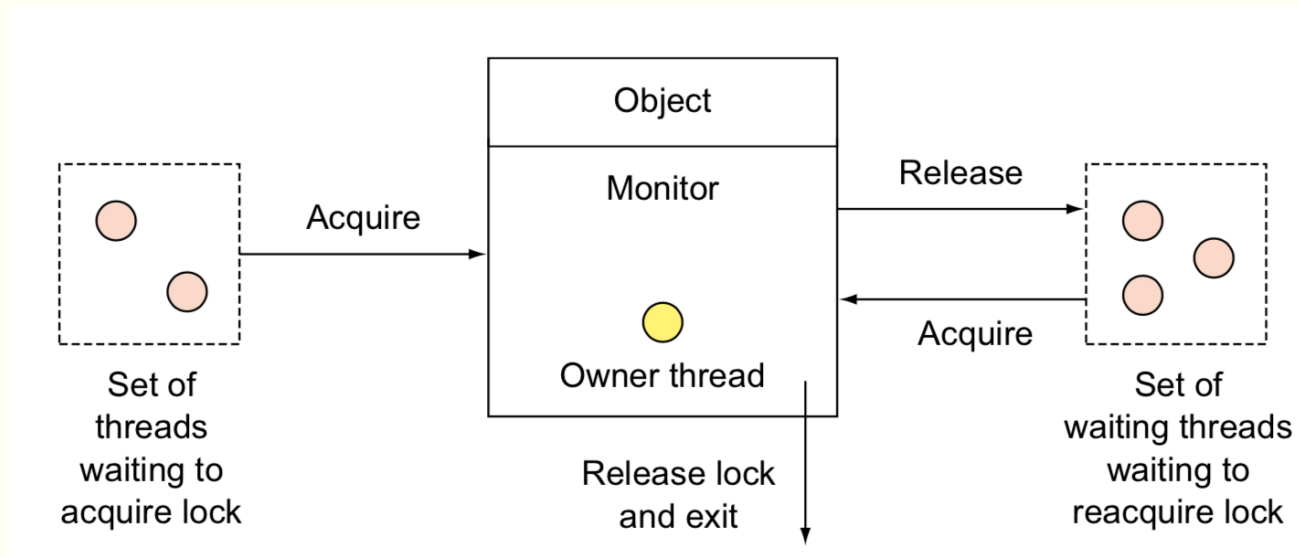
```
C:\Users\user\source\repos\ConcurrentPrimitives\Concurrent
Total time to write file is 2580
```

Lock

```
range.AsParallel().AsOrdered().ForAll(i =>
{
    Thread.Sleep(10);
    Monitor.Enter(_locker);
    try
    {
        File.WriteAllText("test.txt", i.ToString());
    }
    finally
    {
        Monitor.Exit(_locker);
    }
});
```

- Інструкція lock насправді є синтаксичним цукром для Monitor.Enter() та Monitor.Exit(), огорнутих блоком try-catch.

```
C:\Users\user\source\repos\ConcurrentPrimitives\Conc
Total time to write file is 2261
```



М'ютекс (Mutex)

- Попередній код добре працює для одного екземпляру додатку, оскільки задачі працюють всередині процесу, а замок блокує всередині процесу бар'єр пам'яті.
 - Якщо запустити кілька екземплярів програми, обидва додатки матимуть власну копію статичних даних, тобто блокуватимуть власні бар'єри пам'яті.
 - Це дозволяє одному потоку на процес входити в критичну секцію та намагатись записати дані в файл.
 - У результаті маємо виняток `System.IO.IOException: 'The process cannot access the file ...\test.txt' because it is being used by another process.'`
- Щоб мати змогу блокувати спільні ресурси, можна застосувати замок на рівні ядра (kernel level) за допомогою м'ютекса.
 - На відміну від замка, м'ютекс дозволяє лише 1 потоку на *систему* отримувати доступ до захищеного ресурсу, незалежно від кількості процесів, що виконуються.
 - М'ютекс може бути іменованим або неіменованим.
 - Неіменований м'ютекс працює подібно до замка та не може накладатись між процесами.

М'ютекс (Mutex)

```
private static Mutex mutex = new Mutex();

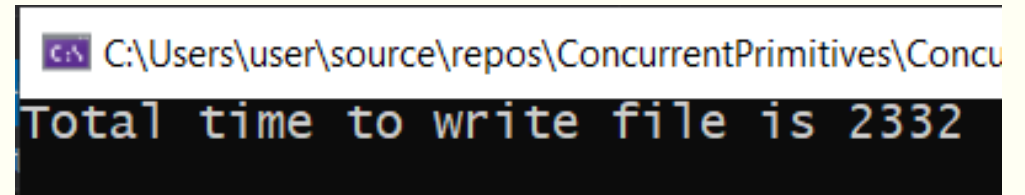
static void Main(string[] args)
{
    var range = Enumerable.Range(1, 1000);
    Stopwatch watch = Stopwatch.StartNew();

    range.AsParallel().AsOrdered().ForAll(i =>
    {
        Thread.Sleep(10);
        mutex.WaitOne();
        File.AppendAllText("test.txt", i.ToString());
        mutex.ReleaseMutex();
    });

    watch.Stop();
    Console.WriteLine($"Total time to write file is { watch.ElapsedMilliseconds }");

    Console.ReadKey();
}
```

- Для класу `Mutex` class можемо викликати метод `WaitHandle.WaitOne()`, щоб заблокувати критичну секцію, та `ReleaseMutex()`, щоб розблокувати її.
 - Знищення (disposing) м'ютекса автоматично звільнить доступ до секції.
 - Програма працює добре, проте при запуску багатьох екземплярів викидатиме `IOException`. Тому створюємо `namedMutex`:
 - `private static Mutex namedMutex = new Mutex(false, "ShaktiSinghTanwar");`
 - Опційно задаємо таймаут на виклик `WaitOne()` м'ютекса. У прикладі це можна зробити так:
 - `namedMutex.WaitOne(3000);`



```
C:\Users\user\source\repos\ConcurrentPrimitives\Concu
Total time to write file is 2332
```


Семафор (Semaphore)

- Замок, м'ютекс та монітор дозволяють лише одному потоку отримати доступ до захищеного ресурсу.
 - Проте інколи слід дозволяти кільком потокам отримувати такий доступ: у сценаріях пулінгу, троттлінгу тощо.
 - Семафор, на відміну від замка чи м'ютекса, дозволяє довільному потоку запускати вивільнення семафору.
 - Як і м'ютекс, семафор працює на рівні процесів.
- Типовий конструктор семафора приймає 2 параметри:
 - `initialCount` – задає початкову кількість потоків, яким можна заходити;
 - `maximumCount` – задає загальну кількість потоків, яким дозволено заходити.
- Нехай маємо віддалену службу, яка дозволяє лише 3 конкурентних підключення від клієнта та займає 1 секунду для обробки запиту:
 - `private static void DummyService(int i) { Thread.Sleep(1000); }`
 - Потрібно паралельно обробляти задачу, проте також переконатись, що немає більше 3 викликів сервісу в один момент. Це досягається створенням семафору з `maximumCount = 3`:
 - `Semaphore semaphore = new Semaphore(3, 3);`

Семафор (Semaphore)

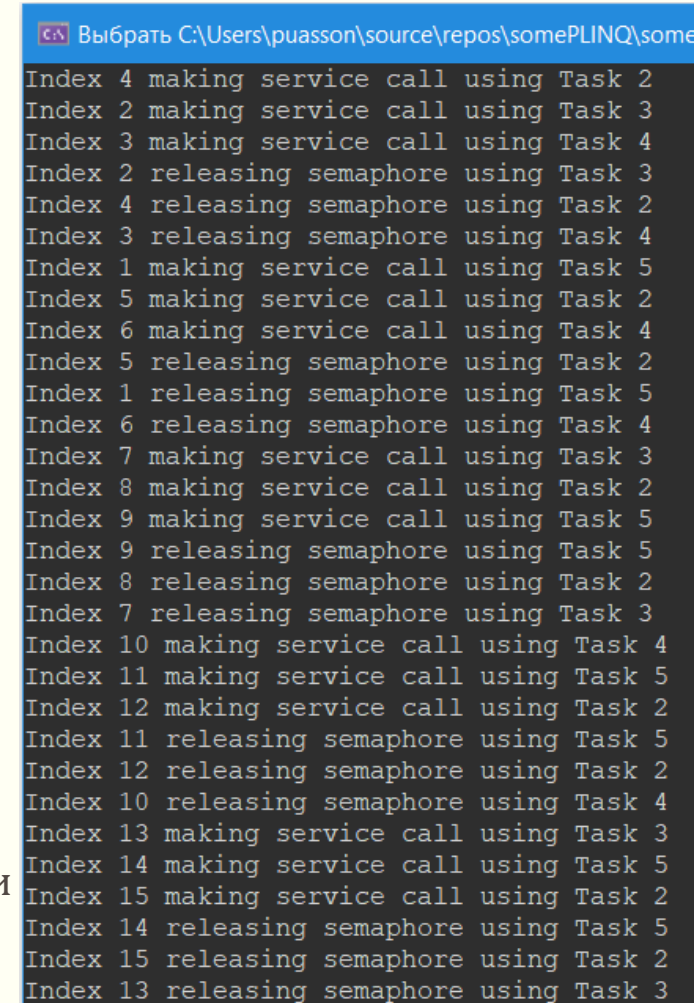
```
private static void CallService(int i) { Thread.Sleep(1000); }

static void Main(string[] args)
{
    Semaphore semaphore = new Semaphore(3, 3);
    var range = Enumerable.Range(1, 1000);
    range.AsParallel().AsOrdered().ForAll(i =>
    {
        semaphore.WaitOne();
        Console.WriteLine($"Index {i} making service call using Task { Task.CurrentId }" );

        CallService(i);    //Simulate Http call
        Console.WriteLine($"Index {i} releasing semaphore using Task { Task.CurrentId }");
        semaphore.Release();
    });

    Console.ReadKey();
}
```

- Запишемо код, який симулює 1000 паралельно зроблених звернень, проте з можливістю обробки лише трьох за раз.
 - Як тільки потік відкриває замок, інший потік входить, проте лише за умови наявності 3х потоків всередині критичної секції за один раз.



```
Выбрать C:\Users\puasson\source\repos\somePLINQ\some
Index 4 making service call using Task 2
Index 2 making service call using Task 3
Index 3 making service call using Task 4
Index 2 releasing semaphore using Task 3
Index 4 releasing semaphore using Task 2
Index 3 releasing semaphore using Task 4
Index 1 making service call using Task 5
Index 5 making service call using Task 2
Index 6 making service call using Task 4
Index 5 releasing semaphore using Task 2
Index 1 releasing semaphore using Task 5
Index 6 releasing semaphore using Task 4
Index 7 making service call using Task 3
Index 8 making service call using Task 2
Index 9 making service call using Task 5
Index 9 releasing semaphore using Task 5
Index 8 releasing semaphore using Task 2
Index 7 releasing semaphore using Task 3
Index 10 making service call using Task 4
Index 11 making service call using Task 5
Index 12 making service call using Task 2
Index 11 releasing semaphore using Task 5
Index 12 releasing semaphore using Task 2
Index 10 releasing semaphore using Task 4
Index 13 making service call using Task 3
Index 14 making service call using Task 5
Index 15 making service call using Task 2
Index 14 releasing semaphore using Task 5
Index 15 releasing semaphore using Task 2
Index 13 releasing semaphore using Task 3
```

Існує 2 види семафорів: локальний та глобальний

- Локальний семафор є локальним для додатку, в якому використовується.
 - Будь-який безіменний семафор створюватиметься локальним:
 - Semaphore semaphore = new Semaphore(1, 10);
- Глобальний семафор є глобальним для ОС, оскільки застосовує примітиви блокування на рівні ядра або системи:
 - Semaphore semaphore = new Semaphore(1, 10, "Globalsemaphore");
 - Якщо створювати семафор лише на один потік, він працюватиме, як замок.

Клас ReaderWriterLock

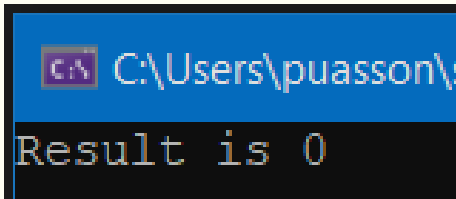
- Визначає замок, який підтримує багато читачів (readers) та лише одного записувача (writer) за раз.
 - Корисний у сценаріях, де спільний ресурс часто зчитується багатьма потоками, проте рідко оновлюється.
- Існує два класи для reader-writer-замків у .NET Framework: ReaderWriterLock та ReaderWriterLockSlim.
 - ReaderWriterLock практично застарілий, оскільки потенційно генерує взаємоблокування, знижує продуктивність, має обмежену підтримку рекурсії та оновлення/відкочування замків.
 - ReaderWriterLockSlim буде розглянуто пізніше.

Основи примітивів сигналювання (signaling primitives)

- Важливий аспект паралельного програмування – координація задач.
 - При створенні задач можна прийти до сценарію «виробник-споживач» (producer-consumer), коли потік-споживач очікує на спільний ресурс, який оновлюється іншим потоком (виробником).
 - Оскільки споживач не знає, коли виробник оновить спільний ресурс, він продовжує опитувати спільний ресурс, що може призвести до гонитви даних.
- Опитування (Polling) низькоефективне в таких сценаріях, кращезастосовувати примітиви сигналювання з .NET Framework.
 - За їх допомоги потік-споживач призупиняється, поки не отримає сигнал від потоку-виробника.
 - Поширені примітиви сигналювання: Thread.Join(), WaitHandle та EventWaitHandler.

Thread.Join

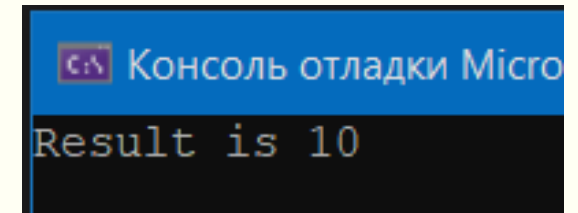
```
int result = 0;
Thread childThread = new Thread(() =>
{
    Thread.Sleep(5000);
    result = 10;
});
childThread.Start();
Console.WriteLine($"Result is {result}");
```



C:\Users\puasson\s
Result is 0

- Найпростіший спосіб змусити потік очікувати на сигнал від іншого потоку.
 - Має блокуючу природу, тобто викликаючий потік заблоковано, поки приєднаний (joined) потік не завершить роботу.
 - Опційно можна задати таймаут, який дозволить розблокувати потік із заблокованого стану після певного періоду часу.
- У коді очікуємо на результат 10, проте маємо 0.

```
int result = 0;
Thread childThread = new Thread(() =>
{
    Thread.Sleep(5000);
    result = 10;
});
childThread.Start();
childThread.Join();
Console.WriteLine($"Result is {result}");
```



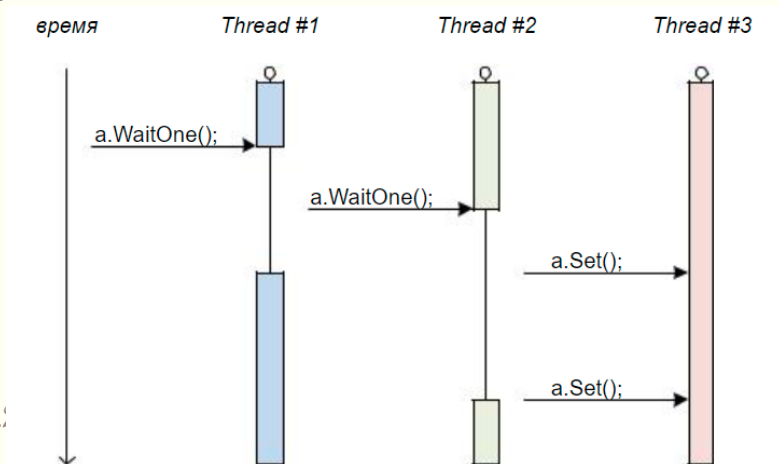
Консоль отладки Micro
Result is 10

Клас EventWaitHandle

```
AutoResetEvent autoResetEvent = new AutoResetEvent(false);
Task signallingTask = Task.Factory.StartNew(() => {
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(1000);
        autoResetEvent.Set();
    }
});

int sum = 0;
Parallel.For(1, 10, (i) => {
    Console.WriteLine($"Task with id {Task.CurrentId} waiting for signal to enter");
    autoResetEvent.WaitOne();
    Console.WriteLine($"Task with id {Task.CurrentId} received signal to enter");

    sum += i;
});
```



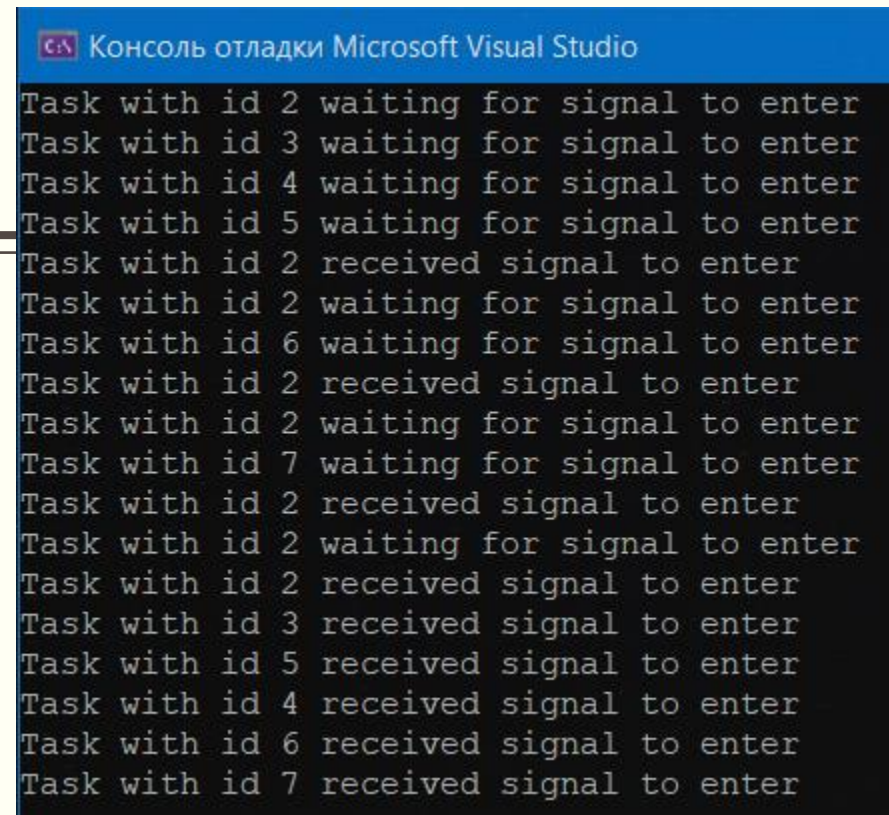
- Представляє подію синхронізації для потоку.
 - Є батьківським класом для класів `AutoResetEvent` та `ManualResetEvent`.
 - Можна сигналізувати `EventWaitHandle`, викликаючи `Set()` або `SignalAndWait()`.
 - Клас `EventWaitHandle` не має афінності потоку (thread affinity, прив'язка потоку до ядра), тому може отримувати сигнал від довільного потоку.
- `AutoResetEvent` відноситься до `WaitHandle`-класів, які автоматично скидаються (reset).
 - Як тільки стався reset, вони дозволяють одному потоку перейти створений бар'єр.
 - Після переходу решта потоків заново встановлюються (set), блокуючись до надходження нового сигналу.
 - In the following example, we are trying to find out the sum of 10 numbers in a thread-safe manner, without using locks.
- Спочатку створимо `AutoResetEvent` з початковим несигнальним станом (false): усі потоки мають чекати, поки не надійде сигнал.
 - Якщо задати початковий стан сигнальним (true), перший потік перейде бар'єр, а решта чекатимуть нового сигналу.

Приклад роботи з AutoResetEvent

```
AutoResetEvent autoResetEvent = new AutoResetEvent(false);
Task signallingTask = Task.Factory.StartNew(() => {
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep(1000);
        autoResetEvent.Set();
    }
});

int sum = 0;
Parallel.For(1, 10, (i) => {
    Console.WriteLine($"Task with id {Task.CurrentId} waiting for
        signal to enter");
    autoResetEvent.WaitOne();
    Console.WriteLine($"Task with id {Task.CurrentId} received
        signal to enter");
    sum += i;
});
```

- Далі створюємо сигнальну задачу (signaling task), яка 10 разів на секунду запускає сигнал за допомогою методу `autoResetEvent.Set()`; оголошуємо змінну `sum` та ініціалізуємо її нулем.
 - Запускаємо паралельний цикл, у якому створюється 10 задач.
 - Кожна задача негайно запускається та очікує на сигнал, блокуючись інструкцією `autoResetEvent.WaitOne()`.
 - Після кожної секунди надсилатиметься сигнал від сигнальної задачі, і один потік входить та змінюватиме суму.



```
Консоль отладки Microsoft Visual Studio
Task with id 2 waiting for signal to enter
Task with id 3 waiting for signal to enter
Task with id 4 waiting for signal to enter
Task with id 5 waiting for signal to enter
Task with id 2 received signal to enter
Task with id 2 waiting for signal to enter
Task with id 6 waiting for signal to enter
Task with id 2 received signal to enter
Task with id 2 waiting for signal to enter
Task with id 7 waiting for signal to enter
Task with id 2 received signal to enter
Task with id 2 waiting for signal to enter
Task with id 2 received signal to enter
Task with id 3 received signal to enter
Task with id 5 received signal to enter
Task with id 4 received signal to enter
Task with id 6 received signal to enter
Task with id 7 received signal to enter
```


Клас ManualResetEvent

```
private static void CallService() { Thread.Sleep(1000); }
```

```
static void Main(string[] args)
```

```
{
```

```
    ManualResetEvent manualResetEvent = new ManualResetEvent(false);
```

```
    Task signalOffTask = Task.Factory.StartNew(() => {
```

```
        while (true) {
```

```
            Thread.Sleep(2000);
```

```
            Console.WriteLine("Network is down");
```

```
            manualResetEvent.Reset();
```

```
        }
```

```
    });
```

```
    Task signalOnTask = Task.Factory.StartNew(() => {
```

```
        while (true) {
```

```
            Thread.Sleep(5000);
```

```
            Console.WriteLine("Network is Up");
```

```
            manualResetEvent.Set();
```

```
        }
```

```
    });
```

```
    for (int i = 0; i < 3; i++) {
```

```
        Parallel.For(0, 5, (j) => {
```

```
            Console.WriteLine($"Task with id {Task.CurrentId} waiting for network to be up");
```

```
            manualResetEvent.WaitOne();
```

```
            Console.WriteLine($"Task with id {Task.CurrentId} making service call");
```

```
            CallService();
```

```
        });
```

```
        Thread.Sleep(2000);
```

```
    }
```

```
    15.03.2021
```

```
}
```

- На відміну від AutoResetEvent, клас ManualResetEvent дозволяє потокам проходити бар'єр, поки його не буде заново задано.

- У коді потрібно здійснити 15 викликів сервісу паралельно пакетами по 5 та 2-секундною затримкою між пакетами (batch).
- Здійснюючи виклик сервісу, потрібно переконатись, що система з'єднана з мережею.
- Для імітації статусу мережі створимо 2 задачі: одна сигналізує, що зв'язку з мережею немає, а інша – що є.

- Спочатку створимо подію ручного reset-у з початковим станом «off».

- Далі створимо 2 задачі для симуляції перемикання статусу мережі, генеруючи
 - подію вимкнення мережі кожні 2 секунди (блокує всі мережеві виклики)
 - Подію ввімкнення кожні 5 секунд (дозволяє проходити мережевим викликам).

Клас ManualResetEvent

```
private static void CallService() { Thread.Sleep(1000); }
```

```
static void Main(string[] args)
```

```
{
```

```
    ManualResetEvent manualResetEvent = new ManualResetEvent(false);
```

```
    Task signalOffTask = Task.Factory.StartNew(() => {
```

```
        while (true) {
```

```
            Thread.Sleep(2000);
```

```
            Console.WriteLine("Network is down");
```

```
            manualResetEvent.Reset();
```

```
        }
```

```
    });
```

```
    Task signalOnTask = Task.Factory.StartNew(() => {
```

```
        while (true) {
```

```
            Thread.Sleep(5000);
```

```
            Console.WriteLine("Network is Up");
```

```
            manualResetEvent.Set();
```

```
        }
```

```
    });
```

```
    for (int i = 0; i < 3; i++) {
```

```
        Parallel.For(0, 5, (j) => {
```

```
            Console.WriteLine($"Task with id {Task.CurrentId} waiting for network to be up");
```

```
            manualResetEvent.WaitOne();
```

```
            Console.WriteLine($"Task with id {Task.CurrentId} making service call");
```

```
            CallService();
```

```
        });
```

```
        Thread.Sleep(2000);
```

```
    }
```

```
    15.03.2021
```

- Створено цикл for, який формує 5 задач у кожній ітерації зі sleep-інтервалом у 2 секунди між ітераціями.

- Перед здійсненням виклику сервісу ми очікуємо, що мережа працюватиме, викликаючи manualResetEvent.WaitOne();
- 5 задач запускаються та негайно блокуються, щоб очікувати, коли мережа запрацює.
- Після 5с мережа вмикається, і ми сигналізуємо за допомогою методу Set(), змушуючи всі 5 потоків здійснювати виклик сервісу.

- Це повторюється в кожній ітерації циклу.

```
Выбрать C:\Users\puasson\source\repos\somePLINQ\somePLINQ
Task with id 3 waiting for network to be up
Task with id 5 waiting for network to be up
Task with id 4 waiting for network to be up
Task with id 6 waiting for network to be up
Task with id 7 waiting for network to be up
Network is down
Network is down
Network is Up
Task with id 5 making service call
Task with id 4 making service call
Task with id 6 making service call
Task with id 3 making service call
Task with id 7 making service call
Network is down
Network is down
Task with id 14 waiting for network to be up
Task with id 15 waiting for network to be up
Task with id 16 waiting for network to be up
Task with id 17 waiting for network to be up
Task with id 18 waiting for network to be up
Network is Up
Task with id 17 making service call
Task with id 18 making service call
Task with id 16 making service call
Task with id 15 making service call
Task with id 14 making service call
Network is down
Network is down
Task with id 25 waiting for network to be up
Task with id 26 waiting for network to be up
```

Клас System.Threading.WaitHandle

- Успадковується від класу MarshalByRefObject та використовується для синхронізації потоків, запущених у додатку.
 - Блокування та сигналізування використовуються для синхронізації потоків, застосовуючи **дескриптори очікування (wait handles)**.
 - Потоки можуть блокуватись викликом будь-якого з методів класу WaitHandle та розблоковуватись залежно від типу обраної сигнальної конструкції.
- Методи класу WaitHandle:
 - **WaitOne**: блокує викликаючий потік, поки не надійде сигнал від дескриптора очікування.
 - **WaitAll**: блокує викликаючий потік , поки не надійде сигнал від усіх дескрипторів очікування.
 - **WaitAny**: блокує викликаючий потік , поки не надійде сигнал від довільного дескриптора очікування.
 - **SignalAndWait**: використовується для виклику Set() на дескрипторі очікування та викликає WaitOne() для іншого дескриптора очікування.
 - У багатопоточному середовищі цей метод може застосовуватись для розблокування одного потоку за раз, а потім **resets to wait for the next thread**:
public static bool SignalAndWait (System.Threading.WaitHandle toSignal, System.Threading.WaitHandle toWaitOn);

Приклад роботи WaitAll()

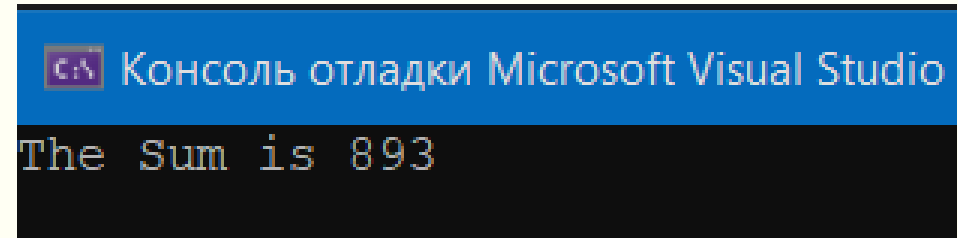
```
static int _dataFromService1 = 0;
static int _dataFromService2 = 0;

private static void FetchDataFromService1(object state) {
    Thread.Sleep(1000);
    _dataFromService1 = 890;
    var autoResetEvent = state as AutoResetEvent;
    autoResetEvent.Set();
}

private static void FetchDataFromService2(object state) {
    Thread.Sleep(1000);
    _dataFromService2 = 3;
    var autoResetEvent = state as AutoResetEvent;
    autoResetEvent.Set();
}

private static void WaitAll()
{
    List<WaitHandle> waitHandles = new List<WaitHandle> {
        new AutoResetEvent(false),
        new AutoResetEvent(false)
    };
    ThreadPool.QueueUserWorkItem(new WaitCallback (FetchDataFromService1), waitHandles.First());
    ThreadPool.QueueUserWorkItem(new WaitCallback (FetchDataFromService2), waitHandles.Last());
    //Waits for all the threads (waitHandles) to call the .Set() method
    //i.e. wait for data to be returned from both service
    WaitHandle.WaitAll(waitHandles.ToArray());
    Console.WriteLine($"The Sum is { _dataFromService1 + _dataFromService2 } ");
}
```

- Приклад використовує 2 потоки для імітації різних викликів сервісу.
 - Обидва потоки виконуватимуться паралельно, проте очікуватимуть `WaitHandle.WaitAll(waitHandles)` перед друком суми в консоль:



```

static int findIndex = -1;
static string winnerAlgo = string.Empty;
private static void BinarySearch(object state)
{
    dynamic data = state;
    int[] x = data.Range;
    int valueToFind = data.ItemToFind;
    AutoResetEvent autoResetEvent = data.WaitHandle as AutoResetEvent;
    //Search for item using .NET framework built in Binary Search
    int foundIndex = Array.BinarySearch(x, valueToFind);
    //store the result globally
    Interlocked.CompareExchange(ref findIndex, foundIndex, -1);
    Interlocked.CompareExchange(ref winnerAlgo, "BinarySearch", string.Empty);
    //Signal event
    autoResetEvent.Set();
}
public static void LinearSearch(object state)
{
    dynamic data = state;
    int[] x = data.Range;
    int valueToFind = data.ItemToFind;
    AutoResetEvent autoResetEvent = data.WaitHandle as AutoResetEvent;
    int foundIndex = -1;
    //Search for item linearly using for loop
    for (int i = 0; i < x.Length; i++)
        if (valueToFind == x[i])
            foundIndex = i;

    //store the result globally
    Interlocked.CompareExchange(ref findIndex, foundIndex, -1);
    Interlocked.CompareExchange(ref winnerAlgo, "LinearSearch", string.Empty);
    //Signal event
    autoResetEvent.Set();
}

```

Приклад роботи WaitAny()

- **WaitAny:** блокує викликаючий потік, поки не надійде сигнал від довільного дескриптора очікування.
 - У прикладі застосовуємо 2 потоки для пошуку елемента: вони виконуються паралельно, а програма очікує на завершення одного з них в методі `WaitHandle.WaitAny(waitHandles)` перед виводом індексу елемента в консоль.
 - Маємо 2 методи: бінарний та лінійний пошук.
 - Потрібно отримати результат якомога швидше: будемо сигналізувати про завершення пошуку за допомогою `AutoResetEvent`, а також збереження результатів у глобальні змінні `findIndex` та `winnerAlgo`.

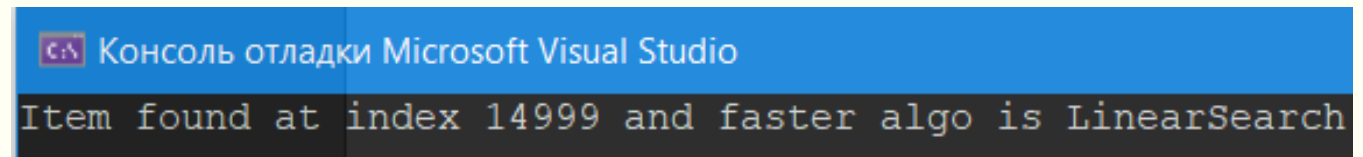
The following code calls both algorithms in parallel using ThreadPool

```
private static void AlgoSolverWaitAny()
{
    WaitHandle[] waitHandles = new WaitHandle[]
    {
        new AutoResetEvent(false),
        new AutoResetEvent(false)
    };
    var itemToSearch = 15000;
    var range = Enumerable.Range(1, 100000).ToArray();
    ThreadPool.QueueUserWorkItem(new WaitCallback (LinearSearch),
        new
        {
            Range = range,
            ItemToFind = itemToSearch,
            WaitHandle = waitHandles[0]
        });

    ThreadPool.QueueUserWorkItem(new WaitCallback(BinarySearch),
        new
        {
            Range = range,
            ItemToFind = itemToSearch,
            WaitHandle = waitHandles[1]
        });
    WaitHandle.WaitAny(waitHandles);
    Console.WriteLine($"Item found at index {findIndex} and faster algo is { winnerAlgo }");
}
```

15.03.2021

@Марченко С.В., ЧДБК, 2021



Консоль отладки Microsoft Visual Studio

Item found at index 14999 and faster algo is LinearSearch

38

Легковагові (Lightweight) синхронізаційні примітиви

- .NET Framework також надає примітиви легковагової синхронізації, які мають краще продуктивність, ніж їх відповідники.
 - Також там, де це можливо, уникається залежність від kernel-об'єктів, таких як дескриптори очікування.
 - Тому нові примітиви працюють лише всередині процесу та рекомендуються для застосування до потоків з коротким часом очікування.
 - Примітиви можна поділити на 2 категорії:

Legacy	Slim
ReaderWriterLock	ReaderWriterLockSlim
Semaphore	SemaphoreSlim
ManualResetEvent	ManualResetEventSlim

```
static ReaderWriterLockSlim _readerWriterLockSlim = new ReaderWriterLockSlim();
static List<int> _list = new List<int>();
private static void ReaderWriteLockSlim()
{
    Task writerTask = Task.Factory.StartNew(WriterTask);
    for (int i = 0; i < 3; i++)
        Task readerTask = Task.Factory.StartNew(ReaderTask);
}

static void WriterTask()
{
    for (int i = 0; i < 4; i++) {
        try {
            _readerWriterLockSlim.EnterWriteLock();
            Console.WriteLine($"Entered WriteLock on Task {Task.CurrentId}");
            int random = new Random().Next(1, 10);
            _list.Add(random);
            Console.WriteLine($"Added {random} to list on Task {Task.CurrentId}");
            Console.WriteLine($"Exiting WriteLock on Task {Task.CurrentId}");
        }
        finally {
            _readerWriterLockSlim.ExitWriteLock();
        }
        Thread.Sleep(1000);
    }
}

static void ReaderTask()
{
    for (int i = 0; i < 2; i++) {
        _readerWriterLockSlim.EnterReadLock();
        Console.WriteLine($"Entered ReadLock on Task {Task.CurrentId}");
        Console.WriteLine($"Items: {_list.Select(j => j.ToString()).Aggregate((a, b) => a + "," + b)} on Task { Task.CurrentId}");
        Console.WriteLine($"Exiting ReadLock on Task {Task.CurrentId}");
        _readerWriterLockSlim.ExitReadLock();
        Thread.Sleep(1000);
    }
}
```

ReaderWriterLockSlim

- Легковагова реалізація ReaderWriterLock.
- Представляє замок, який може застосовуватись для управління захищеними ресурсами таким чином, щоб дозволяти кільком потокам спільно читати, проте лише одному потоку записувати.

```
C:\Users\puasson\source\repos\somePLIT
Entered WriteLock on Task 1
Added 2 to list on Task 1
Exiting WriteLock on Task 1
Entered ReadLock on Task 3
Entered ReadLock on Task 2
Entered ReadLock on Task 4
Items: 2 on Task 3
Items: 2 on Task 4
Exiting ReadLock on Task 4
Exiting ReadLock on Task 3
Items: 2 on Task 2
Exiting ReadLock on Task 2
Entered WriteLock on Task 1
Added 5 to list on Task 1
Exiting WriteLock on Task 1
Entered ReadLock on Task 2
Entered ReadLock on Task 3
Items: 2,5 on Task 3
Entered ReadLock on Task 4
Items: 2,5 on Task 2
Exiting ReadLock on Task 3
Items: 2,5 on Task 4
Exiting ReadLock on Task 2
Exiting ReadLock on Task 4
Entered WriteLock on Task 1
Added 6 to list on Task 1
Exiting WriteLock on Task 1
Entered WriteLock on Task 1
Added 2 to list on Task 1
Exiting WriteLock on Task 1
```


SemaphoreSlim

```
private static void ThrottlerUsingSemaphoreSlim()
{
    var range = Enumerable.Range(1, 12);
    SemaphoreSlim semaphore = new SemaphoreSlim(3, 3);
    range.AsParallel().AsOrdered().ForAll(i =>
    {
        try
        {
            semaphore.Wait();
            Console.WriteLine($"Index {i} making service call using Task { Task.CurrentId}");
            //Simulate Http call
            CallService(i);
            Console.WriteLine($"Index {i} releasing semaphore using Task { Task.CurrentId}");
        }
        finally
        {
            semaphore.Release();
        }
    });
}

private static void CallService(int i) { Thread.Sleep(1000); }
```

```
C:\Users\puasson\source\repos\somePLINQ\somePLINQ\bin\
Index 4 making service call using Task 4
Index 3 making service call using Task 2
Index 2 making service call using Task 5
Index 3 releasing semaphore using Task 2
Index 4 releasing semaphore using Task 4
Index 2 releasing semaphore using Task 5
Index 1 making service call using Task 3
Index 5 making service call using Task 4
Index 7 making service call using Task 2
Index 5 releasing semaphore using Task 4
Index 1 releasing semaphore using Task 3
Index 7 releasing semaphore using Task 2
Index 6 making service call using Task 5
Index 9 making service call using Task 3
Index 10 making service call using Task 2
Index 10 releasing semaphore using Task 2
Index 9 releasing semaphore using Task 3
Index 6 releasing semaphore using Task 5
Index 8 making service call using Task 4
Index 12 making service call using Task 3
Index 11 making service call using Task 2
Index 12 releasing semaphore using Task 3
Index 11 releasing semaphore using Task 2
Index 8 releasing semaphore using Task 4
```

- Легковагова реалізація семафору.
 - Вона звужує (throttle) доступ до захищеного ресурсу згідно з визначеною кількістю потоків.
 - Тепер ми використовуємо метод Wait() замість WaitOne(), оскільки дозволяється перехід бар'єру більше, ніж одному потоку.
 - Ще одна відмінність: SemaphoreSlim завжди створюється як локальний семафор.

Клас ManualResetEventSlim

- Легковагова реалізація класу ManualResetEvent.
 - Має краще продуктивність та менші накладні витрати в порівнянні з ManualResetEvent.
- Створення об'єкту класу ManualResetEvent:
 - `ManualResetEventSlim manualResetEvent = new ManualResetEventSlim(false);`
 - Як і інші slim-відповідники, основна відмінність полягає в заміні методу `WaitOne()` на `Wait()`.

Бар'єр та події зі зворотним відліком (countdown events)

- .NET Framework має вбудовані примітиви сигналізування, які допомагають синхронізувати кілька потоків без значної логіки синхронізації.
 - Уся синхронізація обробляється зсередини відповідними структурами даних.
- Два важливих примітиви сигналізування: `CountDownEvent` та `Barrier`:
 - ***CountDownEvent***: клас `System.Threading.CountDownEvent` відноситься до подій, яким надходить сигнал, коли їх лічильник (`count`) набуває значення 0.
 - ***Barrier***: клас `Barrier` дозволяє кільком потокам працювати без контролюючого їх головного потоку. Він створює бар'єр для потоків-учасників, на якому вони переходять в стан очікування, поки всі не надійдуть. Бар'єр добре працює у випадках паралельної поетапної роботи.

Використання класів `Barrier` і `CountDownLatch`

- Нехай слід отримати (fetch) дані від 2 динамічно розміщених (hosted) сервісів.
 - Перед стягуванням даних з одного сервісу його слід розмістити (host).
 - Як тільки дані отримані, потрібно завершити (close down) роботу сервісу. Лише після цього можемо запустити другий сервіс та отримувати від нього дані.
 - Дані повинні стягуватись якомога швидше.
- Створимо бар'єр на 5 учасників: `static Barrier serviceBarrier = new Barrier(5);` і дві події зі зворотним відліком, які запускать/завершатимуть сервіси, коли 6 потоків перейдуть бар'єр.
 - 5 робочих задач братимуть участь разом із задачею, яка керуватиме запуском/завершенням роботи сервісів:
 - `static CountdownEvent serviceHost1CountdownEvent = new CountdownEvent(6);`
 - `static CountdownEvent serviceHost2CountdownEvent = new CountdownEvent(6);`
- Також створимо ще один `CountdownEvent` з `count = 5`.
 - `CountdownEvent` спрацює, коли всі робочі потоки завершать виконання:
 - `static CountdownEvent finishCountdownEvent = new CountdownEvent(5);`

Метод, який виконується робочими задачами

```
static string _serviceName = string.Empty;
static Barrier serviceBarrier = new Barrier(5);
static CountdownEvent serviceHost1CountdownEvent = new CountdownEvent(6);
static CountdownEvent serviceHost2CountdownEvent = new CountdownEvent(6);
static CountdownEvent finishCountdownEvent = new CountdownEvent(5);

private static void GetDataFromService1And2(int j)
{
    _serviceName = "Service1";
    serviceHost1CountdownEvent.Signal();
    Console.WriteLine($"Task with id {Task.CurrentId} signalled countdown event and waiting for service to start");
    //Waiting for service to start
    serviceHost1CountdownEvent.Wait();
    Console.WriteLine($"Task with id {Task.CurrentId} fetching data from service ");
    serviceBarrier.SignalAndWait();
    //change servicename
    _serviceName = "Service2";
    //Signal Countdown event
    serviceHost2CountdownEvent.Signal();
    Console.WriteLine($"Task with id {Task.CurrentId} signalled countdown event and waiting for service to start");
    serviceHost2CountdownEvent.Wait();
    Console.WriteLine($"Task with id {Task.CurrentId} fetching data from service ");
    serviceBarrier.SignalAndWait();
    //Signal Countdown event
    finishCountdownEvent.Signal();
}
```

Реалізація serviceManagerTask

```
private static void BarrierDemoWithStaticParticipants()
{
    Task[] tasks = new Task[5];

    Task serviceManager = Task.Factory.StartNew(() => {
        //Block until service name is set by any of thread
        while (string.IsNullOrEmpty(_serviceName))
            Thread.Sleep(1000);
        string serviceName = _serviceName;
        HostService(serviceName);
        //Now signal other threads to proceed making calls to service1
        serviceHost1CountdownEvent.Signal();
        //Wait for worker tasks to finish service1 calls
        serviceHost1CountdownEvent.Wait();
        //Block until service name is set by any of thread
        while (_serviceName != "Service2")
            Thread.Sleep(1000);
        Console.WriteLine($"All tasks completed for service {serviceName}.");
        //Close current service and start the other service
        CloseService(serviceName);
        HostService(_serviceName);
        //Now signal other threads to proceed making calls to service2
        serviceHost2CountdownEvent.Signal();
        serviceHost2CountdownEvent.Wait();
        //Wait for worker tasks to finish service2 calls
        finishCountdownEvent.Wait();
        CloseService(_serviceName);
        Console.WriteLine($"All tasks completed for service {_serviceName}.");
    });
    15.03.2021 @Марченко С.В., ЧДБК, 2021
```

```
//Finally make worker tasks
for (int i = 0; i < 5; ++i)
{
    int j = i;
    tasks[j] = Task.Factory.StartNew(() =>
    {
        GetDataFromService1And2(j);
    });
}
Task.WaitAll(tasks);
Console.WriteLine("Fetch completed");
46
```

Реалізація serviceManagerTask

```
private static void CloseService(string name)
{
    Console.WriteLine($"Service {name} closed");
}

private static void HostService(string name)
{
    Console.WriteLine($"Service {name} hosted");
}
```

- Блокування все ще коштує продуктивності, оскільки включає перемикання контексту.
 - Далі розглянемо деякі spinning-техніки, які можуть допомогти прибрати накладні витрати від перемикання контексту.

```
C:\Users\puasson\source\repos\somePLINQ\somePLINQ\bin\Debug\netcoreapp3.1\somePLINQ.exe
Task with id 3 signalled countdown event and waiting for service to start
Task with id 2 signalled countdown event and waiting for service to start
Task with id 4 signalled countdown event and waiting for service to start
Task with id 5 signalled countdown event and waiting for service to start
Service Service1 hosted
Task with id 5 fetching data from service
Task with id 2 fetching data from service
Task with id 4 fetching data from service
Task with id 6 signalled countdown event and waiting for service to start
Task with id 6 fetching data from service
Task with id 3 fetching data from service
Task with id 3 signalled countdown event and waiting for service to start
Task with id 4 signalled countdown event and waiting for service to start
Task with id 5 signalled countdown event and waiting for service to start
Task with id 2 signalled countdown event and waiting for service to start
Task with id 6 signalled countdown event and waiting for service to start
All tasks completed for service Service1.
Service Service1 closed
Service Service2 hosted
Task with id 3 fetching data from service
Task with id 4 fetching data from service
Task with id 5 fetching data from service
Task with id 2 fetching data from service
Task with id 6 fetching data from service
Service Service2 closed
All tasks completed for service Service2.
Fetch completed
```

Класи SpinWait та SpinLock

- Раніше зазначалось, що `spinning` набагато ефективніше за блокування для коротких очікувань, оскільки має знижені накладні витрати відносно перемикання контексту та переходу між станами.
 - Можемо створити `SpinWait`-об'єкт: `var spin = new SpinWait();`
 - При потребі зробити `spin` просто викликаємо `spin.SpinOnce();`
- Замки та `interlocking`-примітиви можуть суттєво сповільнити продуктивність, якщо період очікування на отримання замка дуже низький.
 - `SpinLock` постачає легковагову, низькорівневу альтернативу блокуванню.
 - `SpinLock` – значимий тип, тому для використання одного об'єкта в різних місцях необхідно передавати його за посиланням.
 - Враховуючи продуктивність коду, навіть коли `SpinLock` ще не отримав замок, шматок часу забирає оптимізація роботи збирача сміття.
 - За умовчанням `SpinLock` не підтримує трекінг потоків, тобто не слідкує за тим, який потік отримав замок.
 - Проте цю можливість можна ввімкнути: рекомендується лише для налагодження коду, оскільки сповільнює виконання.

Клас SpinLock

```
static SpinLock _spinLock = new SpinLock();
static List<int> _itemsList = new List<int>();

private static void SpinLock(int number)
{
    bool lockTaken = false;
    try
    {
        Console.WriteLine($"Task {Task.CurrentId} Waiting for lock");
        _spinLock.Enter(ref lockTaken);
        Console.WriteLine($"Task {Task.CurrentId} Updating list");
        _itemsList.Add(number);
    }
    finally
    {
        if (lockTaken)
        {
            Console.WriteLine($"Task {Task.CurrentId} Exiting Update");
            _spinLock.Exit(false);
        }
    }
}

static void Main(string[] args)
{
    Parallel.For(1, 5, (i) => SpinLock(i));
}
```

C:\Users\puasson\source\repos\

```
Task 3 Waiting for lock
Task 1 Waiting for lock
Task 2 Waiting for lock
Task 4 Waiting for lock
Task 2 Updating list
Task 2 Exiting Update
Task 3 Updating list
Task 3 Exiting Update
Task 4 Updating list
Task 4 Exiting Update
Task 1 Updating list
Task 1 Exiting Update
```

- Замок отримується за допомогою `_spinLock.Enter(ref lockTaken)` та звільняється з `_spinLock.Exit(false)`.
 - Між цими інструкціями код виконується як синхронізований між усіма потоками.



ДЯКУЮ ЗА УВАГУ!

Наступне питання: Конкурентні колекції даних