



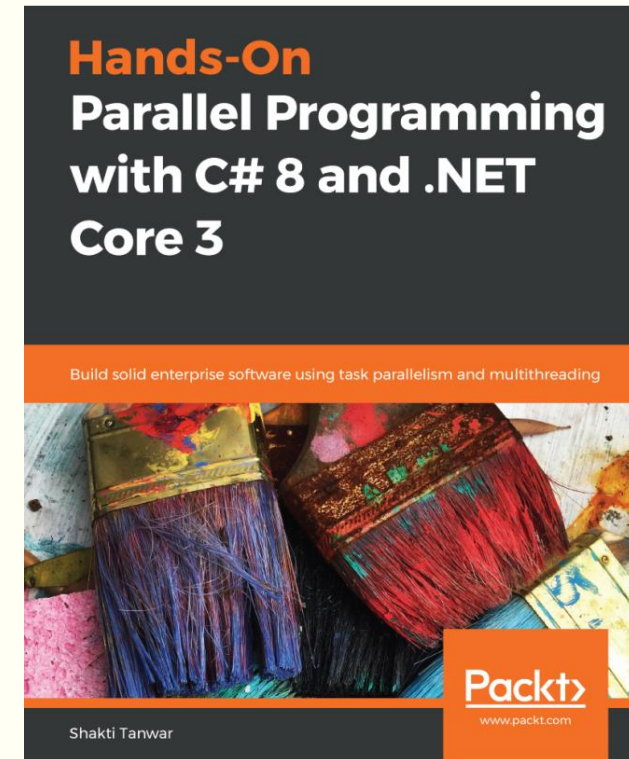
# ДОДАТКОВІ ПИТАННЯ КОНКУРЕНТНОГО ВИКОНАННЯ КОДУ

Лекція 13  
Об'єктно-орієнтоване програмування

# План лекції

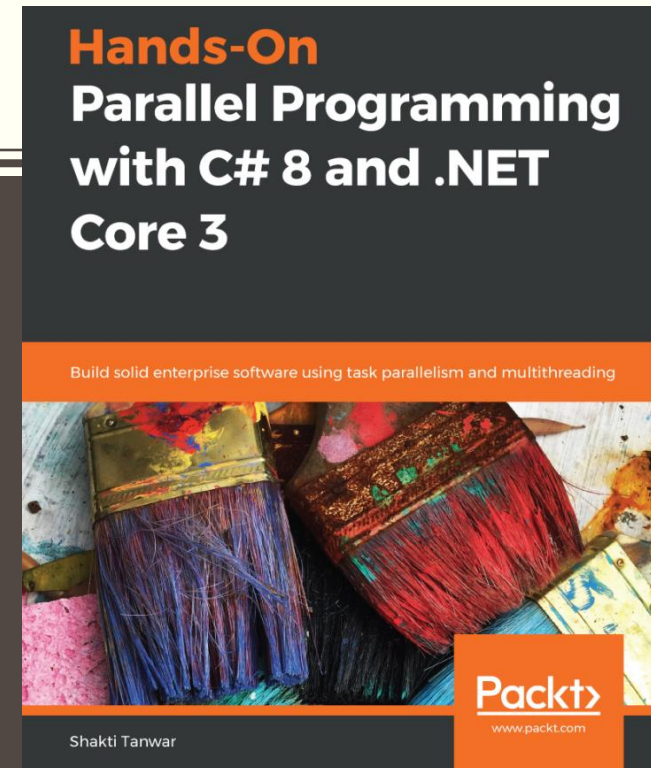
---

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



# ПОТОКИ: СИНХРОНІЗАЦІЯ ТА ПЛАНУВАННЯ

Питання 13.1. (глава 5)



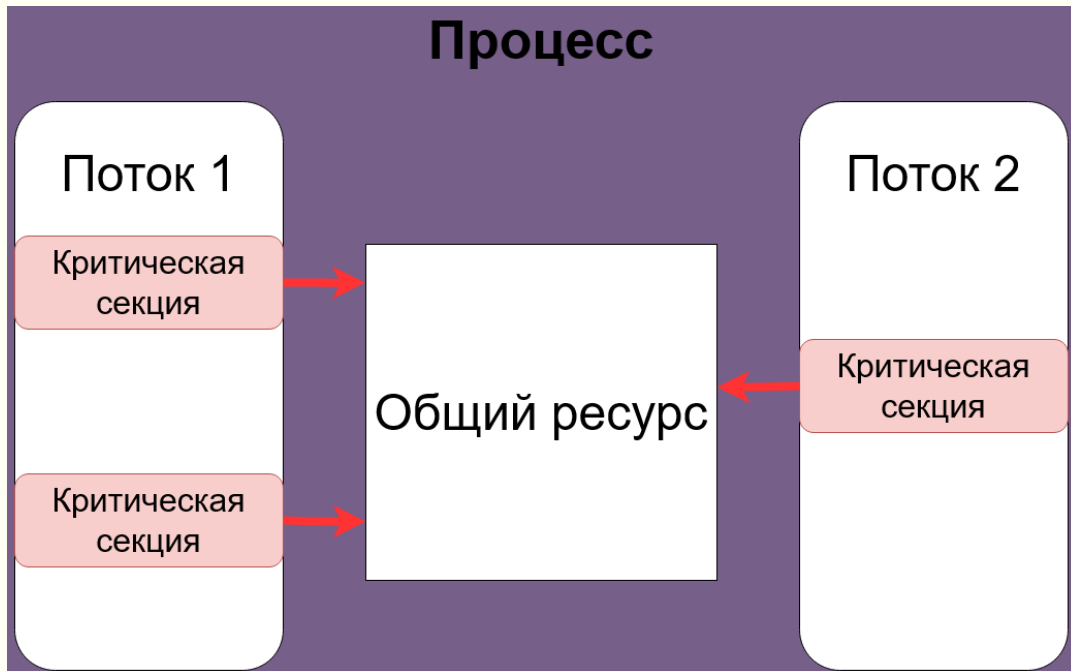
# Зміст запитання

---

- 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

---

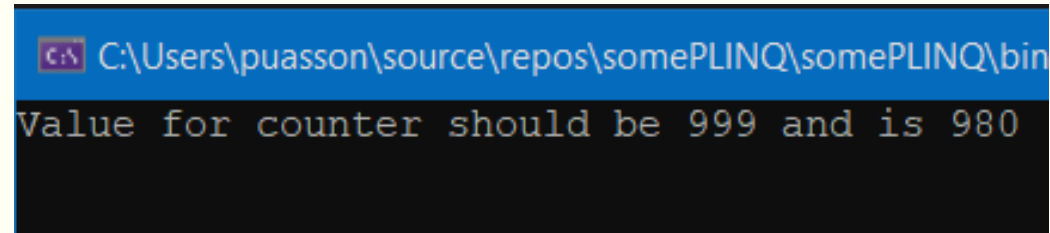
- У .NET Framework присутній цілий ряд примітивів синхронізації, щоб мати справу зі взаємодією між потоками та уникати потенційних станів гонитви.
- Примітиви синхронізації в мові C# можна поділити на 5 категорій:
  - З'єднуючі (Interlocked) операції
  - Блокування (Locking)
  - Сигналізування (Signaling)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

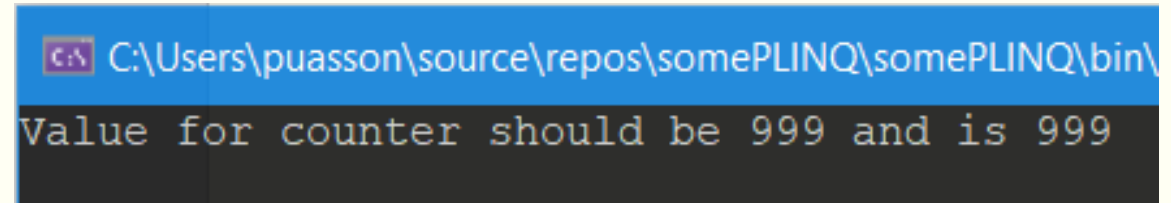
- Відмінність результатів пояснюється станом гонитви між потоками, який виникає через бажання потоку зчитати значення зі змінної, значення якої було записано, проте ще не зафіксовано (committed).

# 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().

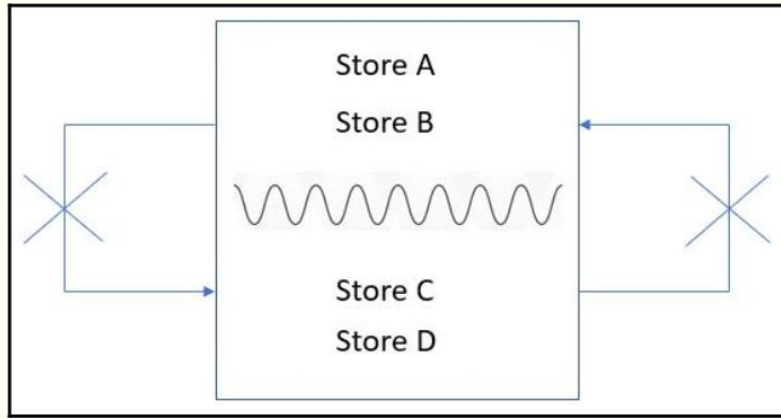


# Бар'єри пам'яті в .NET

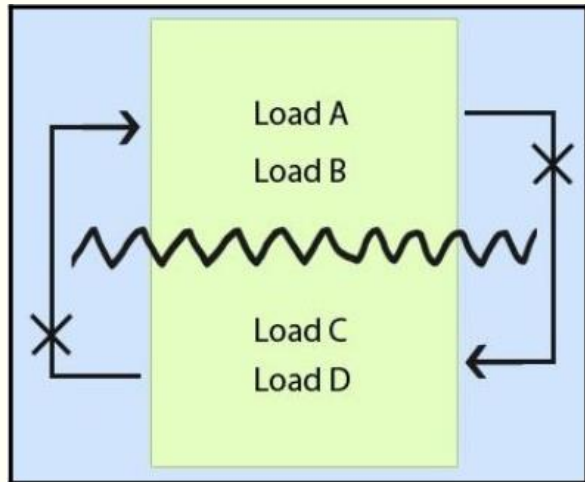
---

- 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 переопорядкування коду

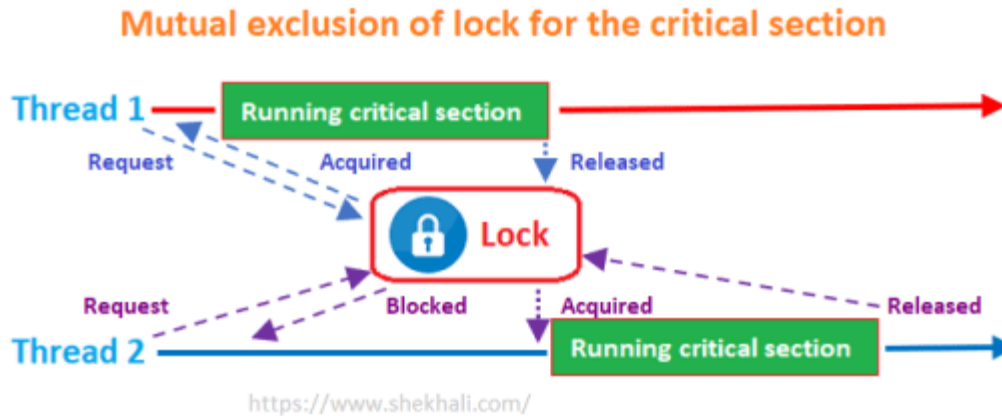
---

- 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.

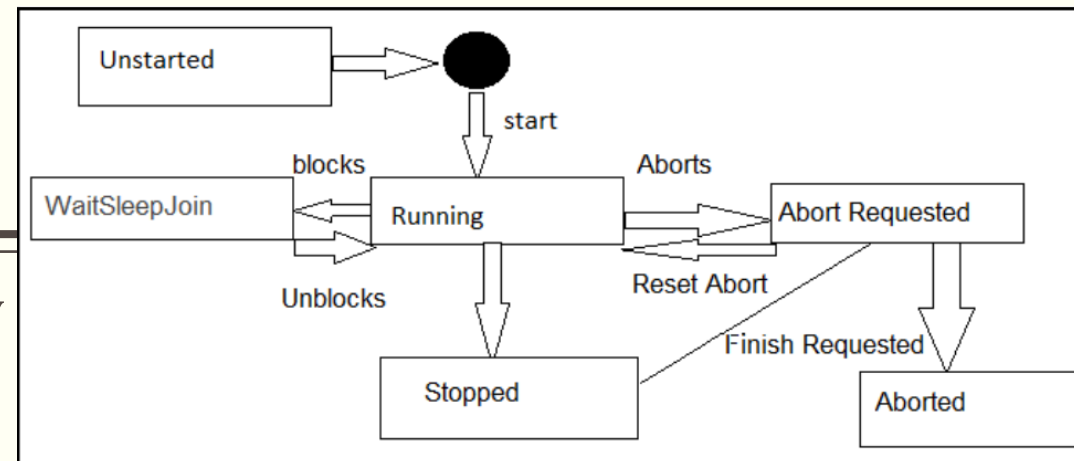
# Блокуючі (locking) примітиви



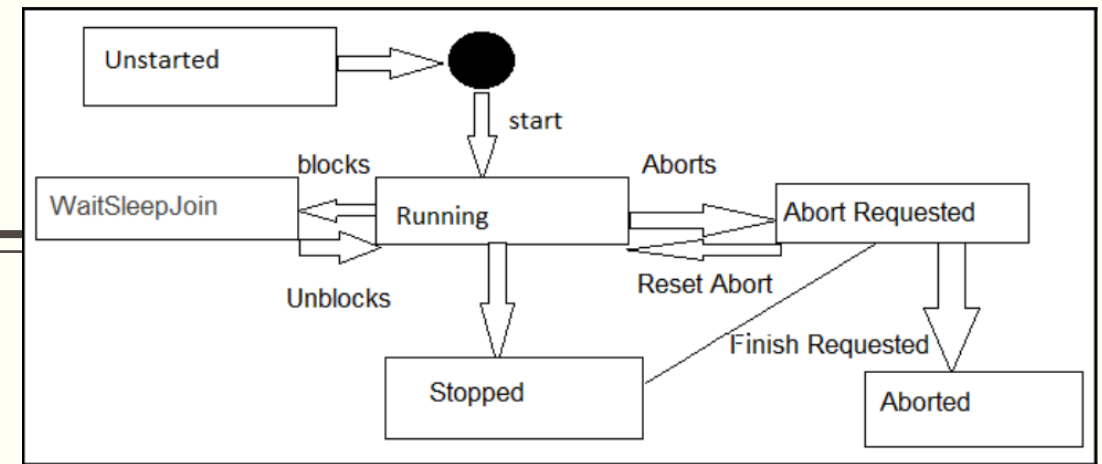
- Замки (Locks) можуть застосовуватись для обмеження доступу до захищеного ресурсу лише одним потоком чи групою потоків.
  - Для ефективної реалізації блокування потрібно ідентифікувати критичні секції, що потребують захисту.
- При застосуванні замка до спільного ресурсу виконуються такі кроки:
  - 1. 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 перемикання 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	×

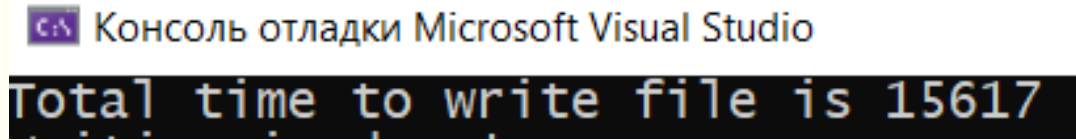
- **Замки та м'ютекси** – це блокуючі конструкції, які дозволяють лише 1 потоку мати доступ до захищеного ресурсу.
  - Замок – скорочена реалізація, яка використовує інший клас для високорівневої синхронізації – Monitor.
- **Семафор** – блокуюча конструкція, яка дозволяє заданій кількості потоків отримати 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).

# Lock

---

- Let's consider the following code, which tries to write a number to a text file:

```
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

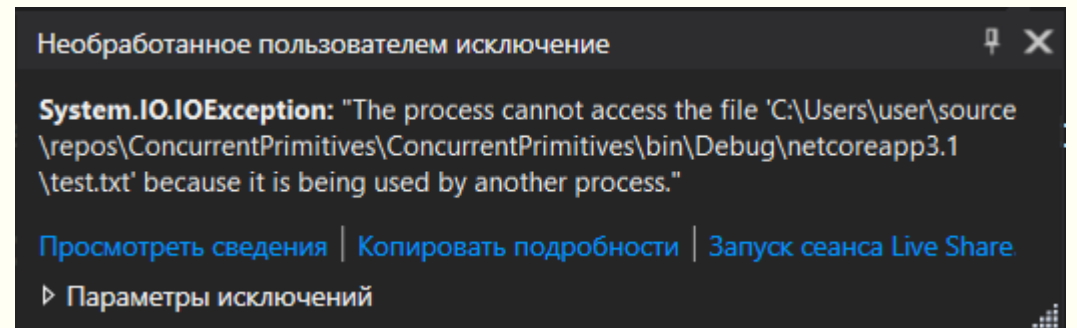
- the task is composed of 1,000 work items and each work item takes approximately 10 milliseconds to execute.
- The time that's taken by the task is 1,000 multiplied by 10, which is 10,000 milliseconds.
- We also have to take into consideration the time taken to perform I/O, so the total time turns out to be 15617.

# Lock

---

- Let's try to parallelize this task using the `AsParallel()` and `AsOrdered()` clauses

```
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}");
```



- What actually happened here is that the file is a shared resource with a critical section and therefore only allows atomic operations.
- With the parallel code, we have a situation where multiple threads are actually trying to write to the file and causing an exception.
- We need to make sure that the code runs in parallel as fast as possible but also maintains atomicity while writing to the file.
- We need to modify the preceding code using a lock statement.

# Lock

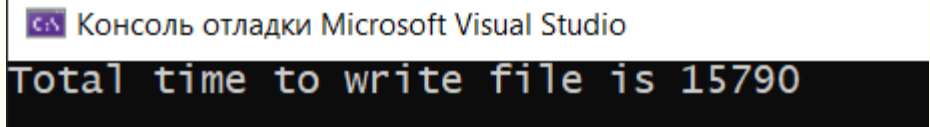
---

```
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();
}
```

- First, declare a static reference type variable. In our case, we take a variable of the object type.
  - We need a reference type variable since the lock can only be applied on the heap memory.
  - Next, we modify the code inside the ForAll() method to include a lock.
  - when we run this code, we won't get any exceptions, but the time that the task took was actually more than the sequential execution.
  - Lock ensures atomicity by making sure that only one thread is allowed to access the vulnerable code, but this comes with the overhead of blocking the thread that is waiting for the lock to be freed. We call this a dumb lock.



Консоль отладки Microsoft Visual Studio

Total time to write file is 15790

# Lock

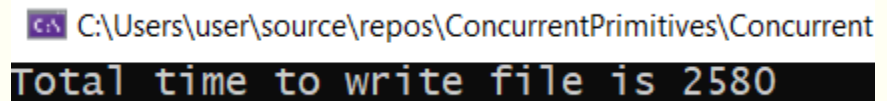
---

```
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();
}
```

- We can modify the program slightly to only lock the critical section to improve performance while maintaining atomicity.
  - we achieved significant gains by mixing synchronization along with parallelization.
  - We can achieve similar results using another locking primitive, that is, the Monitor class.



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

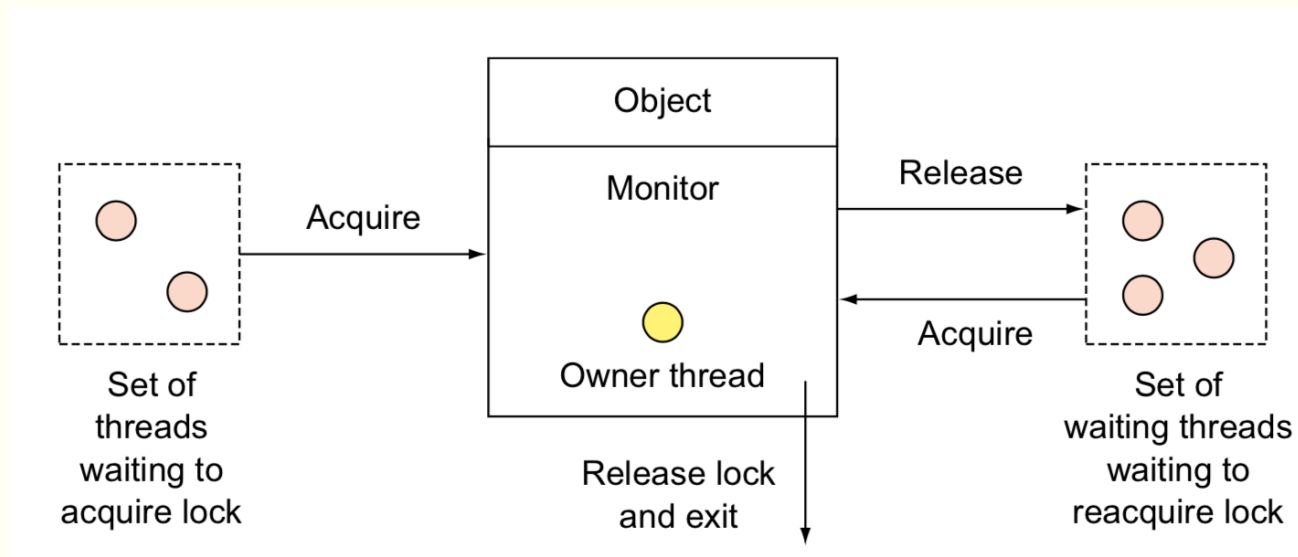
# Lock

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

- Lock is actually a shorthand syntax for achieving `Monitor.Enter()` and `Monitor.Exit()` wrapped inside a try-catch block.

- The same code can, therefore, be written as follows:

```
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 memory barrier.
  - If we run multiple instances of the application, both applications will have their own copy of the static data members and will, therefore, lock their own memory barriers.
  - This will allow one thread per process to actually enter the critical section and try to write the file.
  - This causes the following System.IO.IOException: 'The process cannot access the file ...\\test.txt' because it is being used by another process.'
- To be able to apply locks to shared resources, we can apply a lock at the kernel level using the mutex class.
  - Like lock, mutex allows only one thread to access a protected resource but can work across processes as well, thereby allowing only one thread per system to access a protected resource, irrespective of the number of processes that are executing.
  - A mutex can be named or unnamed. An unnamed mutex works like a lock and cannot work across processes.

# М'ютекс (Mutex)

---

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

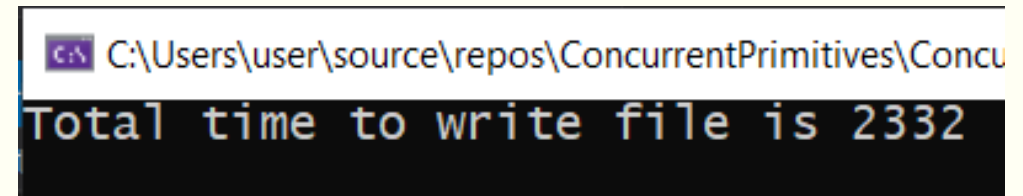
static void Main(string[] args)
{
    var range = Enumerable.Range(1, 1000);
    Stopwatch watch = Stopwatch.StartNew();

    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();
}
```

- With a Mutex class, we can call the `WaitHandle.WaitOne()` method to lock the critical section and `ReleaseMutex()` to unlock the critical sections.
  - Closing or disposing of a mutex automatically releases it.
  - program works well, but if we try to run it on multiple instances, it will throw an `IOException`. For this, we can create a `namedMutex`:
    - `private static Mutex namedMutex = new Mutex(false, "ShaktiSinghTanwar");`
    - Optionally, we can specify a timeout while calling `WaitOne()` on the mutex so that it waits for a signal for a specified amount of time before unblocking itself. This is shown in the following example:
      - `namedMutex.WaitOne(3000);`





# Семафор (Semaphore)

---

- Lock, mutex, and monitor allow only one thread to access a protected resource.
  - Sometimes, however, we need to allow multiple threads to be able to access a shared resource.
  - Examples of these include resource pooling scenarios and throttling scenarios.
  - A semaphore, unlike lock or mutex, is thread-agnostic, which means that any thread can call a release of semaphore. Just like a mutex, it works across processes as well.
- typical semaphore constructor accepts two parameters:
  - `initialCount`, which specifies how many threads are initially allowed to enter,
  - `maximumCount`, which specifies the total number of threads that can enter.
- Let's say we have a remote service that only allows three concurrent connections per client and takes one second to process a request, as follows:

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

- We have a method that has 1,000 work items that need to call the service with parameters.
- We need to process a task in parallel but also make sure that there are no more than three calls to the service at any time. We can achieve this by creating a semaphore with a max count of 3:
- `Semaphore semaphore = new Semaphore(3, 3);`

# Семафор (Semaphore)

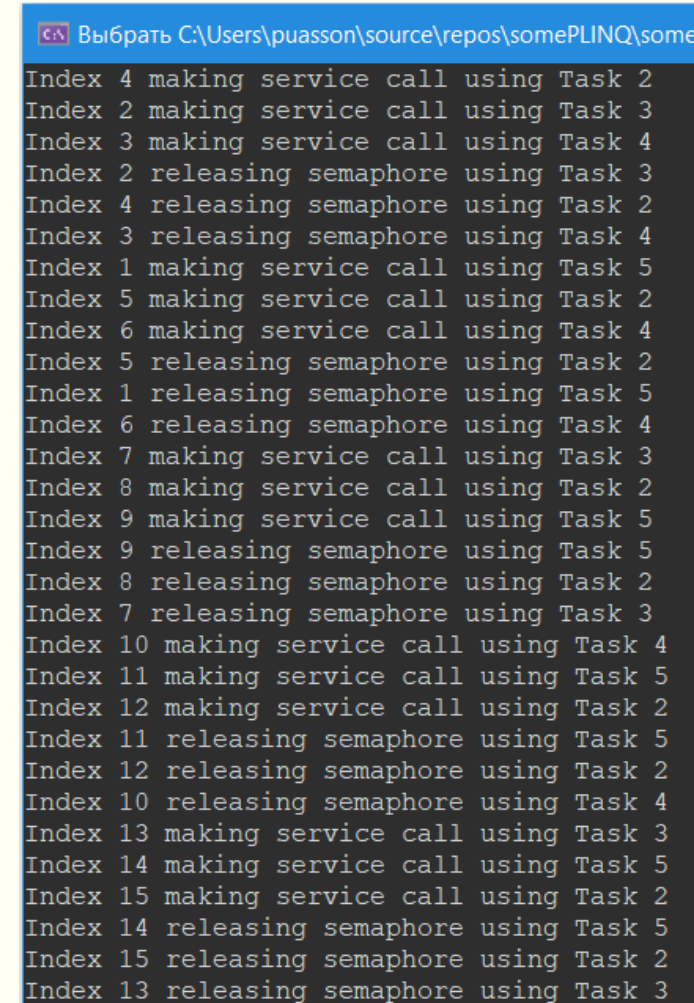
```
private static void CallService(int i) { Thread.Sleep(1000); }

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();
}
```

- we can write some code that can simulate making 1,000 requests in parallel, but only three at a time, using the following semaphore.
  - As you can see, three threads enter and call the service while other threads wait for the lock to be released.
  - As soon as a thread releases the lock, another thread enters but only if three threads are inside the critical section at any one time.



```
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
```

# There are two types of semaphores: local and global

---

- A local semaphore is local to the application where it's used.
  - Any semaphore that is created without a name will be created as a local semaphore, as follows:
  - Semaphore semaphore = new Semaphore(1,10);
- A global semaphore is global to the operating system as it applies kernel- or system-level locking primitives.
  - Any semaphore that is created with a name will be created as a global semaphore, as follows:.
  - Semaphore semaphore = new Semaphore(1,10,"Globalsemaphore");
  - If we create a semaphore with only one thread, it will act like a lock.

# ReaderWriterLock

---

- The ReaderWriterLock class defines a lock that supports multiple readers and a single writer at a time.
  - This is handy in scenarios where a shared resource is read frequently by many threads but updated infrequently.
  - There are two reader-writer lock classes that are provided by the .NET Framework: ReaderWriterLock and ReaderWriterLockSlim.
  - ReaderWriterLock is almost outdated now since it can incur potential deadlocks, reduced performance, complex recursion rules, and upgrading or downgrading of locks.
  - We will discuss ReaderWriterLockSlim in more detail later in this chapter.

# Introduction to signaling primitives

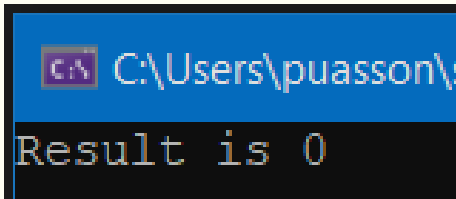
---

- An important aspect of parallel programming is task coordination.
  - While creating tasks, you may come across a producer/consumer scenario where a thread (the consumer) is waiting for a shared resource to be updated by another thread (the producer).
  - Since the consumer doesn't know when the producer is going to update the shared resource, it keeps on polling the shared resource, which can lead to race conditions.
  - Polling is highly inefficient in dealing with these scenarios.
  - It is better to use the signaling primitives that are provided by the .NET Framework.
  - With signaling primitives, the consumer thread is paused until it receives a signal from the producer thread.
  - Let's discuss some common signaling primitives, such as Thread.Join, WaitHandles, and EventWaitHandlers.

# Thread.Join

---

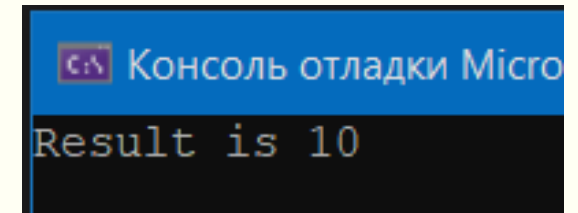
```
int result = 0;
Thread childThread = new Thread(() =>
{
    Thread.Sleep(5000);
    result = 10;
});
childThread.Start();
Console.WriteLine($"Result is {result}");
```



C:\Users\puasson\s  
Result is 0

- This is the simplest way in which we can make a thread wait for a signal from another thread.
  - Thread.Join is blocking in nature, which means that the caller thread is blocked until the joined thread is complete.
  - Optionally, we can specify a timeout that allows the blocked thread to come out of its blocking state once the timeout has been reached.
- We expected the result to be 10, but it has come out as 0.
  - This happened because the main thread that was supposed to write the value runs before the child thread has finished execution.
  - We can achieve the desired behavior by blocking the main thread until the child thread completes. This can be done by calling Join() on the child thread:

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

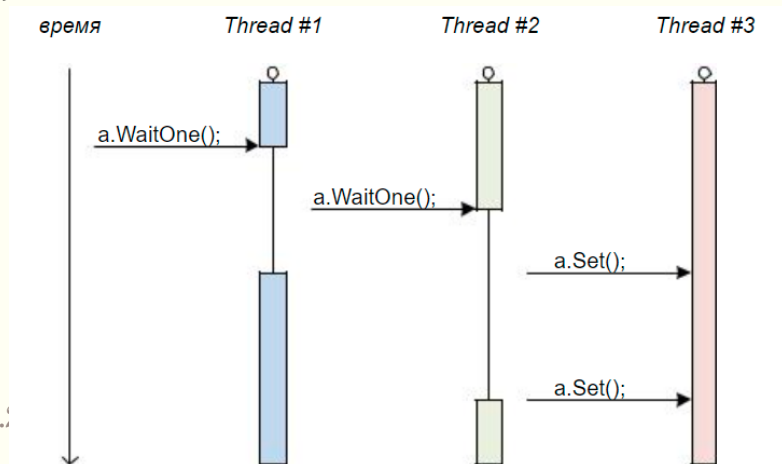


Консоль отладки Micro  
Result is 10

# 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;
});
```



- The `System.Threading.EventWaitHandle` class represents a synchronization event for a thread.
  - It serves as a base class for the `AutoResetEvent` and `ManualResetEvent` classes.
  - We can signal an `EventWaitHandle` by calling `Set()` or `SignalAndWait()`.
  - The `EventWaitHandle` class doesn't have any thread affinity, so it can be signaled by any thread.
- `AutoResetEvent` refers to `WaitHandle` classes that are automatically reset.
  - Once they are reset, they allow one thread to pass through the barrier that is created.
  - As soon as the thread is passed, they are set again, thereby blocking threads until the next signal.
  - In the following example, we are trying to find out the sum of 10 numbers in a thread-safe manner, without using locks.
- First, create an `AutoResetEvent` with the initial state as non-sigaled, or false.
  - This means that all the threads should wait until a signal is received.
  - If we set the initial state to signaled, or true, the first thread will go through while the others wait for a signal

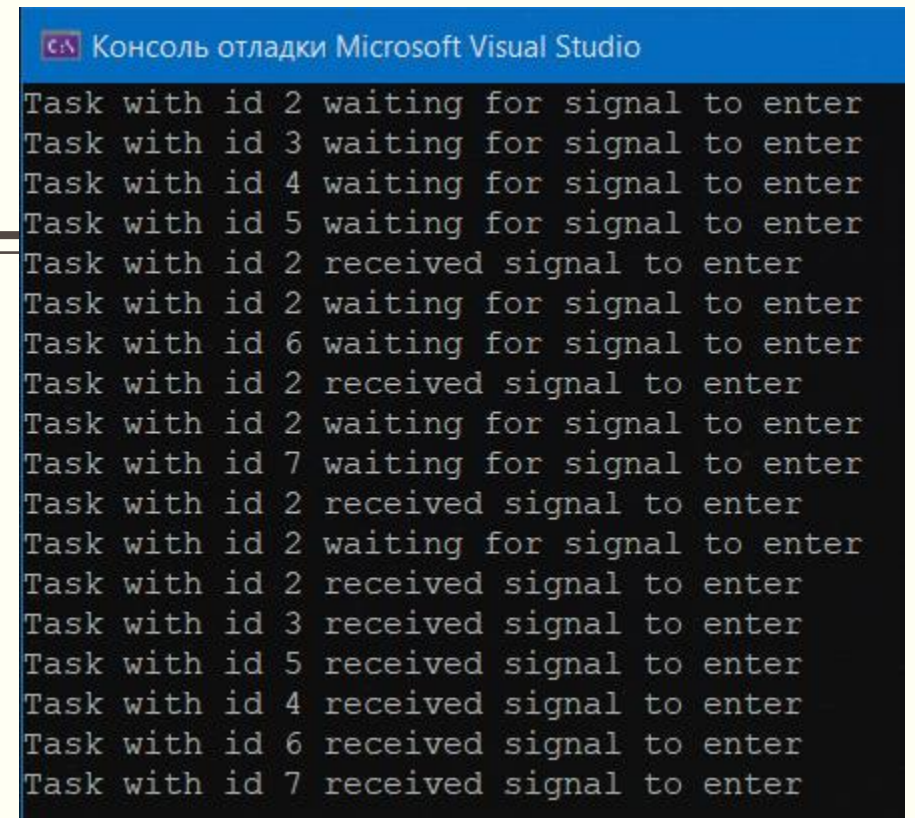


# Приклад роботи з AutoResetEvent

```
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;
});
```

- Next, create a signaling task that fires a signal 10 times per second using the `autoResetEvent.Set()` method; Declare a variable `sum` and initialize it to 0.
  - Create a parallel for loop that creates 10 tasks.
  - Each task will start immediately and wait for a signal to enter, thereby blocking at the `autoResetEvent.WaitOne()` statement.
  - After every second, a signal will be sent by the signaling task and one thread will enter and update the `sum`.



```
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
```



# 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

@Марченко С.В., ЧДБК, 2021

- Unlike AutoResetEvent, which only allows one thread to pass per signal, ManualResetEvent allows threads to keep passing through until it is set again.
  - In the following example, we need to make 15 service calls in batches of 5 in parallel, with a 2-second delay between each batch.
  - While making the service call, we need to make sure that the system is connected to the network.
  - To simulate the network status, we will create two tasks: one that signals the network off and one that signals the network on.
- First, we'll create a manual reset event with the initial state off.
  - Next, we'll create two tasks that simulate the network turning on and off by firing the network off event every two seconds (which blocks all the network calls) and the network on event every five seconds (which allows all the network calls to go through).

# 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
```

```
}
```

- we have created a for loop that creates five tasks in each iteration with a sleep interval of two seconds between iterations.

- Before making service calls, we wait for the network to be up by calling `manualResetEvent.WaitOne()`;
- As you can see, five tasks are started and blocked immediately to wait for the network to be
- up. After five seconds, when the network is up, we signal using the `Set()` method and all five threads pass through to make the service call.
- This is repeated with each iteration of the for loop.

```
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

---

- class that inherits from the MarshalByRefObject class and is used to synchronize threads that are running in an application.
  - Blocking and signaling are used to synchronize threads using wait handles.
  - Threads can be blocked by calling any of the methods of the WaitHandle class.
  - They are released, depending on the type of signaling construct that is selected.
- The methods of the WaitHandle class are as follows:
  - WaitOne: Blocks the calling thread until it receives a signal from the wait handles that it's waiting for.
  - WaitAll: Blocks the calling thread until it receives a signal from all of the wait handles it's waiting for.
  - WaitAny: Blocks the calling thread until it receives a signal from any of the wait handles it's waiting for.
  - SignalAndWait: This method is used to call Set() on a wait handle and calls WaitOne for another wait handle.
    - In a multithreaded environment, this method can be utilized to release one thread at a time and then resets to wait for the next thread:  
`public static bool SignalAndWait (System.Threading.WaitHandle toSignal, System.Threading.WaitHandle toWaitOn);`

# Приклад роботи WaitAll()

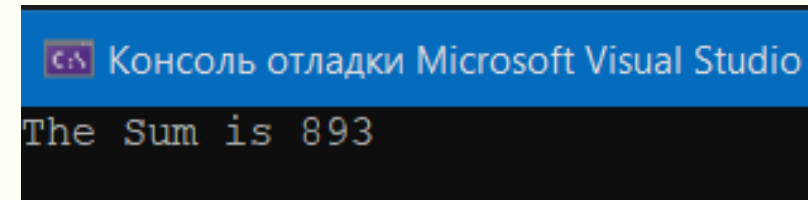
```
static int _dataFromService1 = 0;
static int _dataFromService2 = 0;

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 } ");
}
```

- Here is an example that makes use of two threads to simulate two different service calls.
  - Both threads will execute in parallel but will wait at `WaitHandle.WaitAll(waitHandles)` before printing the sum to the console:



```

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: Blocks the calling thread until it receives a signal from any of the wait handles it's waiting for.
  - Here is an example that makes use of two threads to perform an item search.
  - Both threads will execute in parallel and the program waits for any of the threads to finish execution at the WaitHandle.WaitAny(waitHandles) method before printing the item index to the console.
  - We have two methods, binary search and linear search, that perform a search using binary and linear algorithms.
  - We want to get a result as soon as possible from either of these methods.
  - We can achieve this via signaling using AutoResetEvent and store the results in the findIndex and winnerAlgo global variables

# The following code calls both algorithms in parallel using ThreadPool

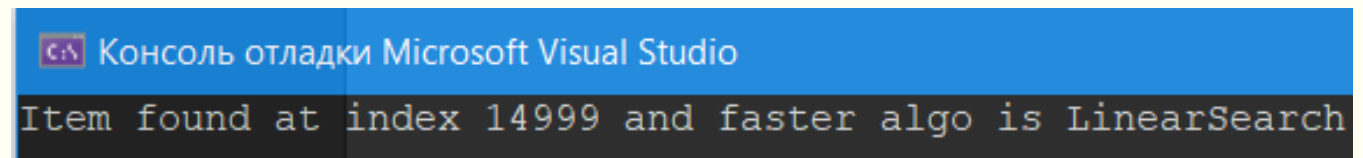
---

```
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

---

- The .NET Framework also provides lightweight synchronization primitives, which are better in performance than their counterparts.
  - They avoid dependency on kernel objects such as wait handles wherever possible, so they only work inside the process.
  - These primitives should be used when the thread's wait time is short.
  - We can divide them into two categories:

Legacy	Slim
ReaderWriterLock	ReaderWriterLockSlim
Semaphore	SemaphoreSlim
ManualResetEvent	ManualResetEventSlim

- Slim locks are slim implementations of legacy synchronization primitives that can improve performance by reducing overheads.

```
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().ForEach(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
```

- lightweight implementation of semaphore.
  - It throttles access to a protected resource to a number of threads.
  - The difference we can see here, apart from replacing the Semaphore class with SemaphoreSlim, is that we now have the Wait() method instead of WaitOne().
  - This makes much more sense as we are allowing more than one thread to pass through.
  - Another important difference is that SemaphoreSlim is always created as a local semaphore, unlike semaphore, which can be created globally as well.

# ManualResetEventSlim

---

- ManualResetEventSlim is a lightweight implementation of ManualResetEvent.
  - It has better performance and less overhead than ManualResetEvent.
- We can create an object using the following syntax, just like ManualResetEvent:
  - `ManualResetEventSlim manualResetEvent = new ManualResetEventSlim(false);`
- Just like other slim counterparts, one major difference here is that we have replaced the `WaitOne()` method with `Wait()`.
  - You can try running some ManualResetEvent demonstration code by making the preceding changes and see if it works.

# Barrier and countdown events

---

- The .NET Framework has some built-in signaling primitives that help us synchronize multiple threads without us having to write lots of synchronization logic.
  - All the synchronization is handled internally by the provided data structures.
- In this section, let's discuss two very important signaling primitives: `CountDownEvent` and `Barrier`:
  - ***CountDownEvent***: The `System.Threading.CountDownEvent` class refers to an event that's signaled when its count becomes 0.
  - ***Barrier***: The `Barrier` class allows multiple threads to run without having the master thread controlling them. It creates a barrier that participating threads must wait in until all the threads have arrived. `Barrier` works well for cases where work needs to be carried out in parallel and in phases.

# A case study using Barrier and CountdownEvent

---

- let's say we need to fetch data from two services that are dynamically hosted.
  - Before fetching the data from service one, we need to host it.
  - Once the data has been fetched, it needs to be closed down.
  - Only when service one has been closed down can we start service two and fetch data from it.
  - The data needs to be fetched as quickly as possible.
- Create a Barrier with 5 participants:
  - `static Barrier serviceBarrier = new Barrier(5);`
- Create two CountdownEvents that will trigger the start or close of services when six threads have passed through it.
  - Five worker tasks will participate, along with a task that will manage the start or close of services:
  - `static CountdownEvent serviceHost1CountdownEvent = new CountdownEvent(6);`
  - `static CountdownEvent serviceHost2CountdownEvent = new CountdownEvent(6);`
- Finally, create another CountdownEvent with a count of 5.
  - This refers to the number of threads that can pass through before the event is signaled.
  - CountdownEvent will trigger when all the worker tasks finish executing:
  - `static CountdownEvent finishCountdownEvent = new CountdownEvent(5);`

# Here is the method that is executed by the worker tasks

---

```
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();
}
```

# serviceManagerTask implementation

---

```
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
```

# serviceManagerTask implementation

---

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

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

- Blocking still comes at a performance cost as it involves context switching.
  - In the next section, we will look at some spinning techniques that can help remove that context switching overhead.

```
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
```

# SpinWait

---

- At the beginning of this chapter, we mentioned that spinning is much more efficient than blocking for smaller waits.
  - Spinning has fewer kernel overheads related to context switching and transitioning.
  - We can create a SpinWait object as follows:
  - `var spin = new SpinWait();`
  - Then, wherever we need to make a spin, we can just call the following command:
  - `spin.SpinOnce();`



# SpinLock

---

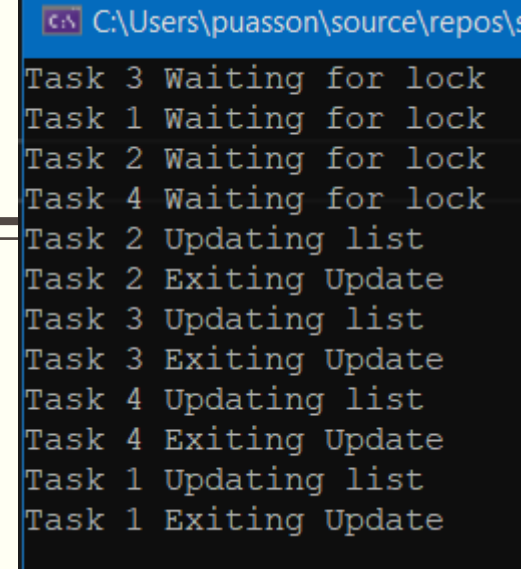
- Locks and interlocking primitives can significantly slow down performance if the wait time to get a lock is very low.
  - SpinLock provides a lightweight, low-level alternative to locking.
  - SpinLock is a value type, so if we want to use the same object in multiple places, we need to pass it by a reference.
  - For performance reasons, even when SpinLock hasn't even acquired the lock, it yields the time slice of the thread so that the garbage collector can work efficiently.
  - By default, SpinLock doesn't support thread tracking, which refers to determining which thread has acquired the lock.
  - However, this feature can be turned on. This is only recommended for debugging and not for production as it reduces performance.

# 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
```

- the lock is acquired using `_spinLock.Enter(ref lockTaken)` and released via `_spinLock.Exit(false)`.
  - Everything between these two statements will be executed as synchronized between all the threads.
  - As a rule of thumb, if we have small tasks, context switching can be completely avoided by using spinning.



---

# ДЯКУЮ ЗА УВАГУ!

**Наступна тема: Паралельне виконання коду**

---