# C# ile Paralel Programlama-1 (Task Sınıfı)

Paralel programlama genellikle sektöre yeni girmiş arkadaşlar için göz korkutan bir konu oluyor, ancak .net framework ile genel paralel programla problemlerinden soyutlanmış bir şekilde “business logic” e odaklanarak paralel uygulama geliştirmek artık kolay bir hal almış durumda. Ben de çok derinlere inmeden paralel programlanın temelleri, implementasyonları ve nasıl çalıştıkları ile ilgili bilgi vermeye çalışacağım.

Örnek kodların çıktısını anlaşılır bir şekilde göstermek için örneklerin tamamını Console uygulamasında geliştirdim ancak örnekler .net ortamının sağladığı diğer templateler üzerinde de çalışacaktır.

Kapsam oldukça geniş olduğu için konuları kendimce ayırıp “seri” formatında paylaşacağım.

**Task sınıfı ve kullanımı**

* [**Taskler arası veri paylaşımı**](https://ilkererhalim.medium.com/c-ile-paralel-programlama-2-task-senkronizasyonu-ve-veri-payla%C5%9F%C4%B1m%C4%B1-1631162b0c1d)
* [**Birden fazla uygulamadan aynı kaynağa erişme örneği**](https://ilkererhalim.medium.com/c-ile-bir-dosyaya-birden-fazla-uygulama-%C3%BCzerinden-e%C5%9F-zamanl%C4%B1-olarak-eri%C5%9Fmek-c5404e18e53c)
* [**Eş zamanlı koleksiyonlar**](https://ilkererhalim.medium.com/c-ile-paralel-programlama-3-e%C5%9F-zamanl%C4%B1-koleksiyonlar-24ec8e40790a)
* [**Producer Consumer Pattern**](https://ilkererhalim.medium.com/c-ile-paralel-programlama-4-producer-consumer-pattern-a8eac67ab94)
* [**Task koordinasyonu**](https://ilkererhalim.medium.com/c-ile-paralel-programlama-5-task-koordinasyonu-a81131d5d04b)
* [**Paralel döngüler Ve Paralel Linq**](https://ilkererhalim.medium.com/c-ile-paralel-programlama-6-koleksiyonlar-i%CC%87%C3%A7erisinde-paralel-i%CC%87terasyonlar-fda8c8cb2024)
* [**Web ortamında async/await nasıl çalışır ?**](https://ilkererhalim.medium.com/web-ortam%C4%B1nda-async-await-nas%C4%B1l-%C3%A7al%C4%B1%C5%9F%C4%B1r-76ee4712122f)

Başlamadan önce paralel programlamanın amacının performans iyileştirmek olmadığını söylemek istiyorum.**Paralel programlamanın amacı uygulamanın aynı anda birden fazla işi yapabilmesidir.**Bunun sonucunda eğer performans artıyorsa bu ek bir faydadır.

.net ortamında paralel programlama denilince bir yazılımcının aklına 3 terim ve bu terimlerin karşılığı olan 3 sınıf gelir.

* Thread
* ThreadPool
* Task

**Thread**

İşletim sistemi seviyesindeki gerçek threadi temsil eder, temel problemi maliyetidir, her threadin kendi kaynakları olduğu için belleği ve işlemciyi meşgul eder.

**ThreadPool**

Threadlerin bulunduğu koleksiyondur, yapması için iş göndermek ve alabileceği thread sayısını belirtmekten başla bir kontrol sağlamaz.

**Task**

Yapılması gereken işleri temsil eder, bir işi alt bir programda yapıp, tamamlanıp tamamlanmadığını ve eğer varsa dönen sonuçları ana programa söyleyebilir. İşletim sistemi seviyesinde bir thread oluşturmaz, yönetmesi ve kullanması kolaydır. Paralel programlama ile ilgili bir çok API task üzerine inşa edilmiştir. Bu yüzden yazıda Task sınıfı ve beraberinde kullanabileceğimiz özelliklerinden bahsedeceğim.

Kaynak: <https://blog.slaks.net/2013-10-11/threads-vs-tasks/>

# ****Task Nasıl Kullanılır ?****

.net ortamında System.Threading.Tasks (System.Runtime assembly) namespace üzerinden bir Task oluşturmanın 3 basit yolu vardır.

* Task.Factory.StartNew methodu
* Yeni bir Task instance oluşturmak
* Task.Run methodu

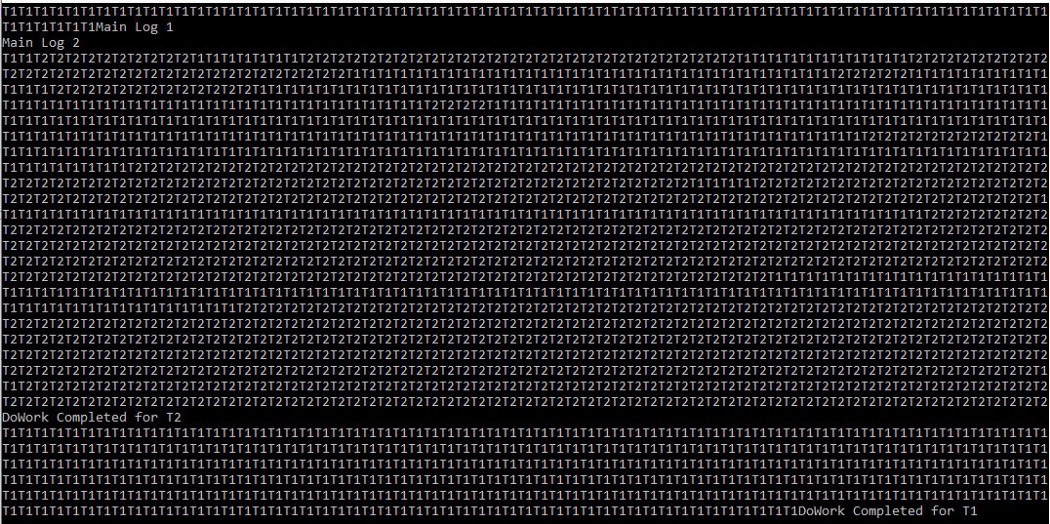
|  |
| --- |
| using System; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace CreatingAndStartingTask |
|  | { |
|  | class Program |
|  | { |
|  | static void DoWork(string message) |
|  | { |
|  | for (int i = 0; i < 1000; i++) |
|  | { |
|  | Console.Write(message); |
|  | } |
|  | } |
|  |  |
|  | static void Main(string[] args) |
|  | { |
|  | // Yeni bir task instance ı oluşturup. Action parametresine gönderdiğimiz DoWork methodunu anında çalıştırır. |
|  | // StartNew methodu aslında iki iş yapar hem bir Task instance ı oluşturup geri döndürür hemde oluşturduğu instance üzerindeki Start methodunu çalıştırır. |
|  | Task.Factory.StartNew(() => |
|  | { |
|  | DoWork("T1"); |
|  | Console.WriteLine("DoWork Completed for T1"); |
|  | }); |
|  | // Instance ı kendimiz oluşturduğumuzda, action parametresindeki methodu çalıştırmaz. |
|  | // Oluşturduğumuz instancetaki Start methodunu tetiklediğimizde DoWork çalışacaktır. |
|  | var task = new Task(() => |
|  | { |
|  | DoWork("T2"); |
|  | Console.WriteLine("DoWork Completed for T2"); |
|  | }); |
|  | Console.WriteLine("Main Thread Log 1"); |
|  | // Task constructor' ı içerisinde tanımladığımız alttaki satırdan sonra çalışacaktır. |
|  | task.Start(); |
|  | Console.WriteLine("Main Thread Log 2"); |
|  | Console.ReadKey(); |
|  | } |
|  | } |
|  | } |

Daha önce hiç Task kullanmadıysanız sonuçlar sizi şaşırtabilir, normal şartlarda beklediğimiz sonuç;

Terminal üzerinde sıra ile :

* 1000 kere T1
* DoWork Completed for T1
* Main Thread Log 1
* 1000 kere T2
* DoWork Completed for T2

Yazmasını bekleriz, ancak örnek kodun çıktısına baktığımızda öyle olmadığını görüyoruz.



Resimde de gördüğünüz gibi terminal çıktısındaki sıralama beklenenden oldukça farklı, bunun sebebi işlemlerin(Tasklerin) çalışmak için birbirini beklememesi.

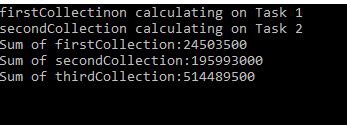
# **Task İçerisinden Sonuçları Okumak**

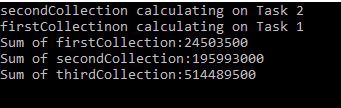
Task sınıfının generic bir child sınıfı mevcut, bu sınıf ile instance aldığınızda ya da StartNew/Run metodlarını generic overloadları üzerinden çalıştırdığınızda taskin verdiğiniz tipte bir dönüş yapmasını bekler. Geri dönen değere Result propertysi üzerinden erişilebilinir.

var task = new Task<int>(...);  
var task = Task.Factory.StartNew<int>(...);  
var task = Task.Run<int>(...);

|  |
| --- |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Linq; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace CreatingAndStartingTask |
|  | { |
|  | class Program |
|  | { |
|  |  |
|  | public static int Sum(IEnumerable<int> source) |
|  | { |
|  | int sum = 0; |
|  | foreach (var item in source) |
|  | { |
|  | sum += item; |
|  | } |
|  | return sum; |
|  | } |
|  |  |
|  | static void Main(string[] args) |
|  | { |
|  | var firstCollection = Enumerable.Range(1, 7000); |
|  | var secondCollection = Enumerable.Range(7000, 14000); |
|  | var thirdCollection = Enumerable.Range(14000, 21000); |
|  | // StartNew methodu int tipi ile çalıştığında method içerideki action' ın geriye integer bir değer dönmesini bekler. |
|  | var task1 = Task.Factory.StartNew<int>(() => |
|  | { |
|  | Console.WriteLine("firstCollectinon calculating on Task {0}", Task.CurrentId); |
|  | return Sum(firstCollection); |
|  | }); |
|  | // Action geriye zaten bir integer değer döndürdüğü için int tipini tanımlamak zorunda değiliz. |
|  | var task2 = Task.Factory.StartNew(() => |
|  | { |
|  | Console.WriteLine("secondCollection calculating on Task {0}", Task.CurrentId); |
|  | return Sum(secondCollection); |
|  | }); |
|  | // Lambda ile methodları aşağıdaki gibi çalıştırıp sonucunu alabiliriz. |
|  | var task3 = Task.Factory.StartNew(() => Sum(thirdCollection)); |
|  |  |
|  | Console.WriteLine("Sum of firstCollection:{0}", task1.Result); |
|  | Console.WriteLine("Sum of secondCollection:{0}", task2.Result); |
|  | Console.WriteLine("Sum of thirdCollection:{0}", task3.Result); |
|  | Console.ReadKey(); |
|  | } |
|  | } |
|  | } |

Çalışma sırası burada da birinci örnekteki gibidir, hangi taskin önce çalışacağı koda bakılarak tahmin edilemez. Tahmin edilebilecek tek çıktı sonuçların ekrana yazdırıldığı sıralamadır.





Aynı kodun çalışmasına rağmen çalışma sıraları birbirinden farklıdır.

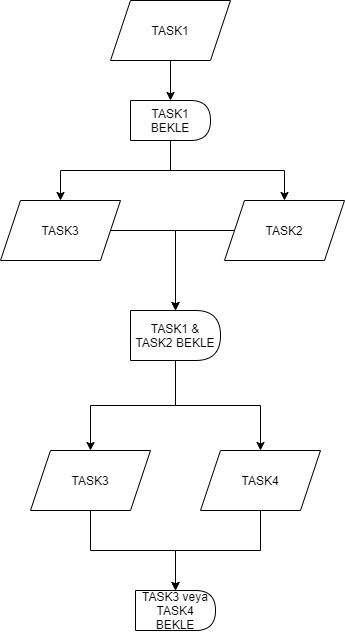
# Taskin veya Tasklerin Tamamlanmasını Beklemek

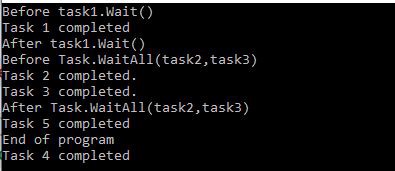
Bir veya birden fazla taskin tamamlanmasını beklemek için çeşitli ihtiyaçlara göre birden fazla yol mevcut, bunların en çok kullanılan 3 tanesi;

* **Task örneği üzerinden Wait metodunu çalıştırmak:** İlgili task tamamlanana kadar bir sonraki satıra geçilmez.
* **Task sınıfı üzerinden WaitAll metodunu çalıştırmak:**Parametre olarak gönderilen tüm task örnekleri tamamlanmadan bir sonraki satıra geçilmez.
* **Task sınıfı üzerinden WaitAny metodunu çalıştırmak:**Parametre olarak gönderilen task örneklerinden herhangi biri tamamlanmadan bir sonraki satıra geçilmez.

|  |
| --- |
| using System; |
|  | using System.Threading; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace CreatingAndStartingTask |
|  | { |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | var task1 = Task.Factory.StartNew(() => |
|  | { |
|  | Thread.Sleep(2000); |
|  | Console.WriteLine("Task 1 completed"); |
|  | }); |
|  | Console.WriteLine("Before task1.Wait()"); |
|  | // task1 tamamlanana kadar main thread bekleyecek. |
|  | task1.Wait(); |
|  | Console.WriteLine("After task1.Wait()"); |
|  | var task2 = Task.Factory.StartNew(() => |
|  | { |
|  | Thread.Sleep(3000); |
|  | Console.WriteLine("Task 2 completed."); |
|  | }); |
|  | var task3 = Task.Factory.StartNew(() => |
|  | { |
|  | Thread.Sleep(5000); |
|  | Console.WriteLine("Task 3 completed."); |
|  | }); |
|  | Console.WriteLine("Before Task.WaitAll(task2,task3)"); |
|  | // task2 ve task3 tamamlanana kadar main thread bekleyecek. |
|  | Task.WaitAll(task2, task3); |
|  | Console.WriteLine("After Task.WaitAll(task2,task3)"); |
|  |  |
|  | var task4 = Task.Factory.StartNew(() => |
|  | { |
|  | Thread.Sleep(1500); |
|  | Console.WriteLine("Task 4 completed"); |
|  | }); |
|  | var task5 = Task.Factory.StartNew(() => |
|  | { |
|  | Thread.Sleep(1000); |
|  | Console.WriteLine("Task 5 completed"); |
|  | }); |
|  | // task4 veya task5 ten herhangi biri tamamlanana kadar main thread bekleyecek. |
|  | Task.WaitAny(task4, task5); |
|  | Console.WriteLine("End of program"); |
|  | Console.ReadKey(); |
|  | } |
|  | } |
|  | } |

Yukarıdaki örnek çalıştırıldığında akış ve çıktı aşağıdaki gibi olacaktır.





Bir Taskin Çalışmasını Durdurmak(İptal Etmek)

Bir task içerisinde yapılan iş gerektiğinde iptal edilebilinmelidir, bu yüzden Task içerisinde çalışacak bir geliştirme yapıyorken, işin iptal edilmek istenilip istenilmediğini sürekli kontrol edilmeli ve iptal isteğinden en kısa süre sonra iş iptal edilmelidir.

System.Threading namespace teki **CancellationTokenSource**sınıfı ile bir token oluşturulup, bu token üzerinden Task iptal etme işlemi yapılır.

|  |
| --- |
| using System; |
|  | using System.Threading; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace CreatingAndStartingTask |
|  | { |
|  | class Program |
|  | { |
|  | static void Counter(CancellationToken token) |
|  | { |
|  | int count = 1; |
|  | while (true) |
|  | { |
|  | /\* |
|  | if (token.IsCancellationRequested) |
|  | { token.ThrowIfCancellationRequested() ile aynı işi yapar. |
|  | throw new OperationCanceledException(); |
|  | } |
|  | \*/ |
|  | // IsCancellationRequested propertysini kontrol edip döngünün bozulması mümkün ancak OperationCanceledException fırlatmadığımız sürece Task tamamlanmış olarak kabul edilir. |
|  | token.ThrowIfCancellationRequested(); |
|  | Console.WriteLine(count++); |
|  | } |
|  | } |
|  |  |
|  | static void Main(string[] args) |
|  | { |
|  | var cancellationTokenSource = new CancellationTokenSource(); |
|  | var task = new Task(() => |
|  | { |
|  | Counter(cancellationTokenSource.Token); |
|  | }, cancellationTokenSource.Token); |
|  | task.Start(); |
|  |  |
|  | Console.ReadKey(); |
|  | // Bu satırdan sonra token cancel edilip task içerisinden OperationCanceledException fırlatılacak, ancak exception Main thread üzerinde fırlamadığı için kullanıcıya henüz birşey yansımayacak. |
|  | cancellationTokenSource.Cancel(); |
|  |  |
|  | try |
|  | { |
|  | // Task içerisinde fırlayan exceptionlar .Wait methodu çalıştığında Main threade yansır. Ve bu exceptionlar AggregateException içerisine toplanır. |
|  | task.Wait(); |
|  | } |
|  | catch (AggregateException ex) |
|  | { |
|  | // AggregateException içerisindeki Handle ile her exception durumu ayrı ayrı yönetilebilinir. |
|  | // OperationCanceledException olması durumunda geriye handle edildi anlamına gelen true döndürüyoruz. |
|  | // OperationCanceledException fırlaması durumunda kullanıcıya birşey yansımayacak ancak diğer exceptionları handle etmiyoruz. |
|  | ex.Handle((exeption) => |
|  | { |
|  | if (exeption is OperationCanceledException) |
|  | { |
|  | Console.WriteLine("İşlem iptal edildi."); |
|  | return true; |
|  | } |
|  | return false; |
|  | }); |
|  | } |
|  | Console.ReadKey(); |
|  | } |
|  |  |
|  | } |
|  | } |

### Örnek çıktı: Belirli bir süre çalışan uygulama herhangi bir tuşa basıldıktan sonra terminale İşlem iptal edildi. yazar

# Yukarıdaki örnek çalıştırıldığında Counter methodu ayrı bir task içerisinde çalışıp terminale sayıları yazmaya başlayacak, ancak bir tuşa basıldığında cancellationTokenSource.Cancel metodu çalıp taski iptal edecek.

# C# ile Paralel Programlama-2 (Task Senkronizasyonu ve Veri Paylaşımı)

Tasklerin eş zamansız çalıştığını ve hangi taskin daha önce tamamlanacağını bilmiyoruz, bir geliştirici için bu senaryo başlı başına bir problemken bir de ne zaman çalışacağını bilmediğiniz iki ayrı task içerisinde aynı veriye erişme ihtiyacını düşünün.

Başlamadan önce bahsetmek istediğim iki terim var.

* Atomic Operation
* Critical Section

## **Atomic Operation**

Antik Yunanca’daki ‘atomos’ kelimesinden gelir. Kesilemez/bölünemez anlamına gelen bu sözcük yazılımda da tam olarak bu durumu ifade etmek için kullanılır.

Bir işlem yapılırken araya başka bir işlemin girmesi mümkün değilse bu bir **atomik operasyon**dur.

value = 5; // Bir atomik operasyondur, bu tanım yapılırken araya başka bir işlemin girmesi mümkün değildir.value++; value+= 5; // işlemleri **atomik operasyon değildir.** Bunun sebebi atama işleminin iki adımda yapılmasıdır.{   
 int temp = value+5;  
 value = temp;  
}  
// Bu iki satır arasında başka bir işlem yapılabilinir.

Senkron bir programda çalışıyorken value +=5 atamasının detayı önemsizdir, ancak aynı anda aynı değişkene erişen birden fazla iş parçacığı olduğunda yukarıdaki kadar basit bir operasyonda dahi karmaşa çıkar.

**Atomik Operasyonlar**

* Referans atamaları
* 32 bit sistemlerde 32 bit veya daha küçük değer okuma/yazma işlemleri
* 64 bit sistemlerde 64 bit veya daha küçük değer okuma/yazma işlemleri

Farklı taskler içerisinde, aynı değişkene, atomik operasyon harici bir işlem uygulanması programın sağlıklı çalışmasını engeller.

|  |
| --- |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace DataSharing |
|  | { |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | int number = 0; |
|  | var tasks = new List<Task>(); |
|  | for (int i = 0; i < 15; i++) |
|  | { |
|  | tasks.Add(Task.Factory.StartNew(() => |
|  | { |
|  | for (int j = 0; j < 20; j++) |
|  | { |
|  | // Atomik olmayan işlem. |
|  | number++; |
|  | } |
|  | })); |
|  | tasks.Add(Task.Factory.StartNew(() => |
|  | { |
|  | for (int j = 0; j < 20; j++) |
|  | { |
|  | // Atomik olmayan işlem. |
|  | number--; |
|  | } |
|  | })); |
|  | } |
|  | Task.WaitAll(tasks.ToArray()); |
|  | Console.WriteLine(number); |
|  | Console.ReadKey(); |
|  | } |
|  |  |
|  | } |
|  | } |

Yukarıdaki örnekte ana program üzerinde tanımlanmış olan bir değişkene (number) iki farklı task içerisinde atomik olmayan değer atamaları yapıyoruz.

15 farklı task içerisinde 20 kere sayının değeri 1 arttırıldı, 20 kere 1 azaltıldı. Ancak uygulama her çalıştırıldığında çıkan sonuçlar farklı olabilir.

Ben denediğimde aldığım çıktılar 24, -3, -22, -11 oldu.

Bunun sebebi number++ işleminin atomik olmamasıdır. Değişkenin yeni değeri hesaplanıp geçici bir değişkene atanırken diğer task tarafından değeri değişitirilmiş olabilir.

**Critical Section**

Farklı taskler içerisinde aynı kaynaklara erişilen alanlara critical section adı verilir. Critical section implementasyonlarının genel mantığı bir task bir kaynağa erişirken diğer task(ler) o kaynağa erişmek için bekler.

Taskleri eş zamanlı çalıştırmanın ihtiyaca göre kullanılabilinecek birden fazla yolu vardır.

**Lock/Monitoring**

Bu yöntemde, bir değer farklı taskler içerisinde kullanılırken, dummy bir obje oluşturulup lock scope’u içerisinde işlem yapılır.

lock(dummyObject)  
{  
...  
}

|  |
| --- |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace DataSharing |
|  | { |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | int number = 0; |
|  | var tasks = new List<Task>(); |
|  | object locker = new object(); |
|  | for (int i = 0; i < 15; i++) |
|  | { |
|  | tasks.Add(Task.Factory.StartNew(() => |
|  | { |
|  | for (int j = 0; j < 20; j++) |
|  | { |
|  | lock (locker) |
|  | { |
|  | number++; |
|  | } |
|  | } |
|  | })); |
|  | tasks.Add(Task.Factory.StartNew(() => |
|  | { |
|  | for (int j = 0; j < 20; j++) |
|  | { |
|  | lock (locker) |
|  | { |
|  | number--; |
|  | } |
|  | } |
|  | })); |
|  | } |
|  | Task.WaitAll(tasks.ToArray()); |
|  | Console.WriteLine(number); |
|  | Console.ReadKey(); |
|  | } |
|  | } |
|  | } |

Bir önceki örneği yukarıdaki gibi düzenlediğinizde sonucun her zaman 0 olduğunu göreceksiniz. Bunun sebebi, task’lerden biri locker isimli objeyi kilitlediğinde diğer task işleme devam etmek için kilidin açılmasını bekler.

Lock keywordü Monitor implementasyonunun basitleştirilmiş halidir. Lock kullanımının karşılığı aşağıdaki gibidir.

try  
{  
 Monitor.Enter(locker);  
}  
finally  
{  
 Monitor.Exit(locker);  
}

Monitor sınıfı ile çalışıldığında maximum bekleme süresi(timeout) tanımlanabilinir. Critical section içerisindeki iş yükü azaldığında diğer tasklere bilgi verip daha hızlı çalışmaları sağlanabilir.

**SpinLock (BusyWaiting)**

Standart bir critical section implementasyonunda işletim sistemi threadi wait-state moduna alır ve o çekirdek üzerinde başka bir threadin koşmasına izin verir, eğer bekleme süresi çok kısa ise bu işlem verimliliğini olumsuz etkiler çünkü çalışmaya devam etmesi gereken threadin tekrar ayağa kalkması gerekecektir.

SpinLock bekleme işlemini bir döngü ile yapar ve çekirdek başka bir threadi çalıştıramaz, ancak sizi [preemtion](https://en.wikipedia.org/wiki/Preemption_(computing)" \t "_blank) maliyetinden kurtarır.

*Modern işletim sistemlerinde Monitor implemantasyonu hybrid olarak yapılır, bir süre busy waiting yapıldıktan sonra thread core waiting e geçer.*

|  |
| --- |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Threading; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace DataSharing |
|  | { |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | int number = 0; |
|  | var tasks = new List<Task>(); |
|  | SpinLock spinLock = new SpinLock(); |
|  | for (int i = 0; i < 15; i++) |
|  | { |
|  | tasks.Add(Task.Factory.StartNew(() => |
|  | { |
|  | for (int j = 0; j < 20; j++) |
|  | { |
|  | bool lockTaken = false; |
|  | try |
|  | { |
|  | spinLock.Enter(ref lockTaken); |
|  | number++; |
|  | } |
|  | finally |
|  | { |
|  | if (lockTaken) |
|  | { |
|  | spinLock.Exit(); |
|  | } |
|  | } |
|  | } |
|  | })); |
|  | tasks.Add(Task.Factory.StartNew(() => |
|  | { |
|  | for (int j = 0; j < 20; j++) |
|  | { |
|  | bool lockTaken = false; |
|  | try |
|  | { |
|  | spinLock.Enter(ref lockTaken); |
|  | number--; |
|  | } |
|  | finally |
|  | { |
|  | if (lockTaken) |
|  | { |
|  | spinLock.Exit(); |
|  | } |
|  | } |
|  |  |
|  | } |
|  | })); |
|  | } |
|  | Task.WaitAll(tasks.ToArray()); |
|  | Console.WriteLine(number); |
|  | Console.ReadKey(); |
|  | } |
|  |  |
|  | } |
|  | } |

İlk örneğin SpinLock kullanarak implementasyonu…

**ReaderWriterLock**

Bir kaynağa yalnızca okuma ya da yazma işlemi için erişilirken kullanılır, Reader locklar birbirilerini kilitlemezler ancak writer lock diğerlerini bekletir. Olduçka basit bir implementasyonu vardır.

|  |
| --- |
| using System; |
|  | using System.Threading; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace DataSharing |
|  | { |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | var readerWriterLock = new ReaderWriterLockSlim(); |
|  | int value = 10; |
|  | Console.WriteLine("Press a key to increase the number"); |
|  | var task = Task.Run(() => |
|  | { |
|  | while (true) |
|  | { |
|  | try |
|  | { |
|  | readerWriterLock.EnterReadLock(); |
|  | Console.WriteLine(value); |
|  | Thread.Sleep(5000); |
|  | } |
|  | finally |
|  | { |
|  | readerWriterLock.ExitReadLock(); |
|  | } |
|  | } |
|  | }); |
|  |  |
|  | while (true) |
|  | { |
|  | Console.ReadKey(); |
|  | readerWriterLock.EnterWriteLock(); |
|  | value++; |
|  | readerWriterLock.ExitWriteLock(); |
|  | } |
|  |  |
|  | } |
|  | } |
|  | }  Bir read locktan çıkış yapmadan bir write lock oluşturulamaz, ancak genel olarak okuma işlemi yapacak olan ve belli koşullarda yazma işlemi yapacak olan bir task için UpgradeableReadLock oluşturulabilinir.   |  | | --- | | var task = Task.Run(() => | |  | { | |  | Random rnd = new Random(); | |  | while (true) | |  | { | |  | try | |  | { | |  | readerWriterLock.EnterUpgradeableReadLock(); | |  | if (rnd.Next(5) == 2) | |  | { | |  | try | |  | { | |  | readerWriterLock.EnterWriteLock(); | |  | value += rnd.Next(10); | |  | } | |  | finally | |  | { | |  | readerWriterLock.ExitWriteLock(); | |  | } | |  | } | |  | Console.WriteLine(value); | |  | Thread.Sleep(5000); | |  | } | |  | finally | |  | { | |  | readerWriterLock.ExitUpgradeableReadLock(); | |  | } | |  | } | |  | }); | |

**Interlocked**

System.Threading namespace te yer alan bu sınıf yardımıyla herhangi bir lock kullanmadan primitive tiplere müdahale edebiliriz.

lock örneğindeki lock(locker){...} kısımlarını silip;

// Sayıynın arttırılmak istenildiği yeri   
**Interlocked.Increment(ref number);**// Sayının azaltılmak istenildiği yeri  
**Interlocked.Decrement(ref number);**

Olacak şekilde güncellediğinizde sonuçta bir değişiklik olmadığını göreceksiniz. Ancak program artık **lock-free**bir program. Interlocked içerisindeki metodlar yaptıkları işleri atomik seviyelerde yaparlar ve herhangi bir şekilde taski teorik olarak kilitlemezler.

*Ancak pratikte işler değişebilir; lock-free cezbedici bir kavram olsa da arka planda işlem yapıldıktan sonra verilen değerin değişip değişmediği kontrol edilir ve eğer bir değişiklik varsa işlem tekrar yapılır.*

Interlocked metotları ile yapabileceğiniz atomik operasyonlar.

* Add: Referansı verilen sayıya ikinci parametrede verilen sayıyı ekler, çıkarma işlemi yapmak için negatif değer verilebilinir.
* CompareExchange: İkinci ve üçüncü parametre eşit ise referansın değerini eşit olan parametre olarak olarak günceller
* Increment: Referansı verilen sayıyı bir arttırır.
* Decrement: Referansı verilen sayıyı bir azaltır.
* Exchange: Referansı verilen değişkeni ya da instance ı ikinci parametredeki değer ya da instance olarak değiştirir.

**Mutex(Mutual Exclution)**

Bazı ek özellikleri dışında Monitör ile benzer bir yapıdadır. Monitor process bazlı çalışırken Mutex processler arası çalışabilir (birden fazla uygulamanın aynı kaynağa eriştiği senaryolarda) ve birden fazla mutex objesi mergelenip eş zamanlı çalıştırılabilinir.

Mutex mutex = new Mutex();  
Task.Run(() =>  
{  
 bool lockTaken = mutex.WaitOne();  
 try  
 {  
 // İşlem  
 }finally  
 {  
 if (lockTaken)  
 {  
 mutex.ReleaseMutex();  
 }  
 }  
});

**Birden fazla Mutex’in mergelenmesi.**

İki mutex ile iki ayrı kaynak üzerine erişim sağyalan birden fazla taskimiz olduğunu hayal edelim ve bu iki kaynak üzerinde hesaplama yapan başka bir taskimiz daha olsun. Bunun gibi senaryolarda WaitHandle sınıfındaki WaitAll metodu ile iki kaynak için bir lock oluşturabiliriz.

bool lockTaken = WaitHandle.WaitAll(new[] { mutex1, mutex2 });

|  |
| --- |
| using System; |
|  | using System.Collections.Generic; |
|  | using System.Threading; |
|  | using System.Threading.Tasks; |
|  |  |
|  | namespace DataSharingAndSynchronization |
|  | { |
|  |  |
|  | class Player |
|  | { |
|  | public int Coin { get; set; } |
|  | } |
|  |  |
|  | class PlayerService |
|  | { |
|  | public void AddCoin(Player player, int coin) |
|  | { |
|  | player.Coin += coin; |
|  | } |
|  |  |
|  | public void TransferCoin(Player from, Player to, int coin) |
|  | { |
|  | from.Coin -= coin; |
|  | to.Coin += coin; |
|  | } |
|  | } |
|  |  |
|  | class Program |
|  | { |
|  | static void Main(string[] args) |
|  | { |
|  | Player player1 = new Player(); |
|  | Player player2 = new Player(); |
|  | PlayerService playerService = new PlayerService(); |
|  | Mutex player1Mutex = new Mutex(); |
|  | Mutex player2Mutex = new Mutex(); |
|  | List<Task> taskList = new List<Task>(); |
|  | for (int i = 0; i < 10; i++) |
|  | { |
|  | taskList.Add(Task.Run(() => |
|  | { |
|  | for (int j = 0; j < 1000; j++) |
|  | { |
|  | bool lockTaken = player1Mutex.WaitOne(); |
|  | try |
|  | { |
|  | playerService.AddCoin(player1, 1); |
|  | } |
|  | finally |
|  | { |
|  | if (lockTaken) |
|  | { |
|  | player1Mutex.ReleaseMutex(); |
|  | } |
|  | } |
|  | } |
|  | })); |
|  | taskList.Add(Task.Run(() => |
|  | { |
|  | for (int j = 0; j < 500; j++) |
|  | { |
|  | bool lockTaken = player2Mutex.WaitOne(); |
|  | try |
|  | { |
|  | playerService.AddCoin(player2, 1); |
|  | } |
|  | finally |
|  | { |
|  | if (lockTaken) |
|  | { |
|  | player2Mutex.ReleaseMutex(); |
|  | } |
|  | } |
|  | } |
|  | })); |
|  | taskList.Add(Task.Run(() => |
|  | { |
|  | for (int j = 0; j < 500; j++) |
|  | { |
|  | // WaitHandle.WaitAll ile ortak bir lock oluşturabiliyoruz. |
|  | bool lockTaken = WaitHandle.WaitAll(new[] { player1Mutex, player2Mutex }); |
|  | try |
|  | { |
|  | playerService.TransferCoin(player1, player2, 1); |
|  | } |
|  | finally |
|  | { |
|  | if (lockTaken) |
|  | { |
|  | player1Mutex.ReleaseMutex(); |
|  | player2Mutex.ReleaseMutex(); |
|  | } |
|  | } |
|  | } |
|  | })); |
|  | } |
|  | Task.WaitAll(taskList.ToArray()); |
|  | Console.WriteLine("Player 1:{0}", player1.Coin); |
|  | Console.WriteLine("Player 2:{0}", player2.Coin); |
|  | Console.ReadKey(); |
|  | } |
|  |  |
|  | } |
|  | } |

Yukarıdaki örnekte;  
On farklı task içerisinde  
player1 e bin defa 1 Coin ekleniyor.  
player2 ye beş yüz defa 1 Coin ekleniyor.  
player1 den player2 ye beş yüz defa 1 Coin transfer ediliyor.

Bu tasklerin tamamı aynı anda anda başlayacaktır, ancak player1Mutex ve player2Mutexleri kilitli olduğu sürece transfer işlemi bekleyecektir. Yani bir player a coin ekleme işlemi yapılırken aynı anda aktarma işlemi yapılmayacaktır.

Mutexin processler arası paylaşım özelliği kullanılarak uygulamanın cihaz üzerinde iki kere çalışması engellenebilinir. Bunun için Mutex sınıfı içerisinde static tanımlanmış olan **OpenExisting** methodu kullanılır, mutex daha önce oluşturulmadıysa **WaitHandleCannotBeOpennedException**fırlatacaktır, bu exceptionı yakaladığımızda mutexin diğer uygulamada zaten tanımlanmış olduğunu anlayabiliriz.

|  |
| --- |
| using System; |
|  | using System.Threading; |
|  |  |
|  | namespace DataSharing |
|  | { |
|  | class Program |
|  | { |
|  | const string appName = "ExampleApp"; |
|  | static void Main(string[] args) |
|  | { |
|  | Mutex appMutex; |
|  | bool enableToRun = false; |
|  | try |
|  | { |
|  | Mutex.OpenExisting(appName); |
|  | } |
|  | catch (WaitHandleCannotBeOpenedException) |
|  | { |
|  | enableToRun = true; |
|  | appMutex = new Mutex(false, appName); |
|  | } |
|  | if (!enableToRun) |
|  | { |
|  | Console.WriteLine("{0} is running in another window.", appName); |
|  | Console.ReadKey(); |
|  | } |
|  | else |
|  | { |
|  | Console.WriteLine("Program..."); |
|  | // Logic. |
|  | Console.ReadKey(); |
|  | } |
|  |  |
|  | } |
|  |  |
|  | } |
|  | } |

Yukarıdaki örnek üzerinden build aldıktan sonra oluşan .exe yi iki kere çalıştırdığınızda ilk exe için “Program…” çıktısını alacaksınız, ancak ikinci .exe de “ExampleApp is running in another windows.” çıktısı alınacaktır.

Mutex ile aynı dosyaya farklı uygulamalardan erişme örneği için [bir sonraki yazımı](https://ilkererhalim.medium.com/c-ile-bir-dosyaya-birden-fazla-uygulama-%C3%BCzerinden-e%C5%9F-zamanl%C4%B1-olarak-eri%C5%9Fmek-c5404e18e53c) okuyabilirsiniz.