**Interlocked vs Lock**

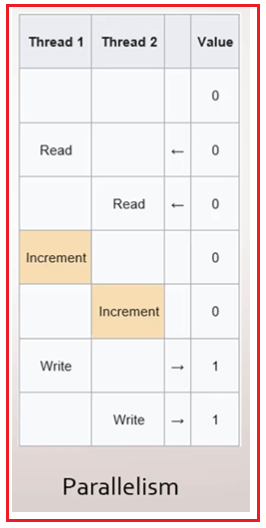
**Example to Understand Interlocked in C#:**

In C#, race conditions occur when we have a variable shared by several threads and these threads want to modify the variable simultaneously. The problem with this is that depending on the order of the sequence of operations done on a variable by different threads, the value of the variable will be different.

A variable is problematic if we access them in a multithreaded environment. Even increasing a variable by 1 or adding variables by 1 is problematic. This is because the operation is not Atomic. A simple variable incrementation is not an atomic operation.

In fact, it is divided into three parts reading, increasing, and writing. Given the fact that we have three operations, two threads can execute them in such a way that even if we increase the value of a variable twice, only one increase takes effect.

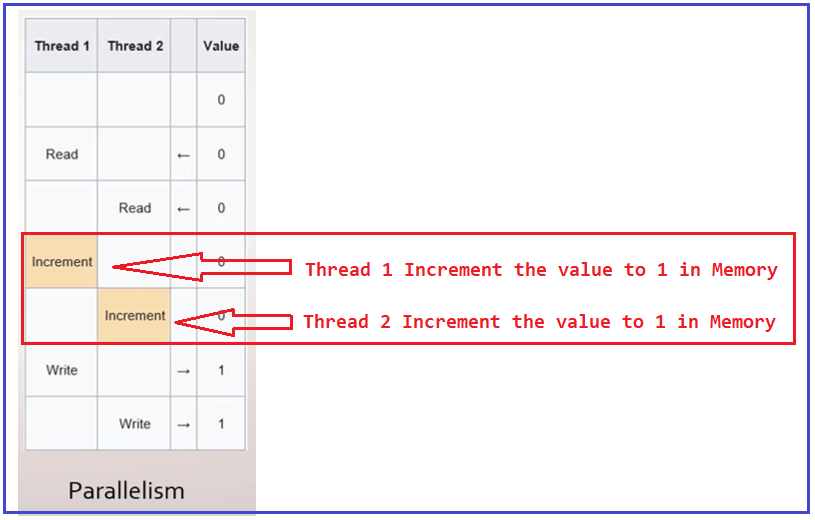
What happens if two threads sequentially try to increment a variable. Let us understand this with an example. Please have a look at the below table. Here, we have Thread 1 in column one and Thread 2 in column 2. And in the end, a value column represents the value of the variable. In this case, the result could be that the final value of the variable is either 1 or 2. Let’s see one possibility.



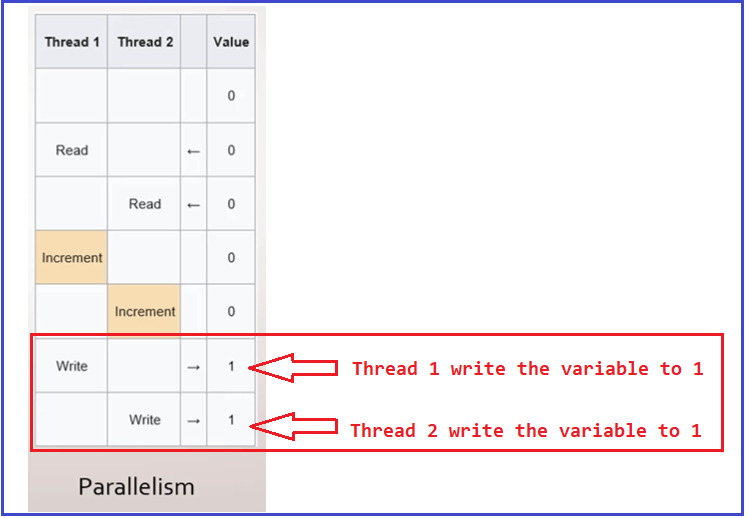
Now, Thread 1 and Thread 2 both read the values and so they both have the value of zero in memory. For a better understanding, please have a look at the below image.



Thread 1 increment the value, as well as Thread 2, also increment the value and both of them increment it to 1 in memory. For a better understanding, please have a look at the below image.



Once both the threads increment the value to 1 in memory. Then Thread 1 writes back to variable 1 and Thread 2 also writes back to variable 1, one more time. For a better understanding, please have a look at the below image.



This means that, as you can see, depending on the order of the execution of the methods, we are going to determine the value of the variable. Even though we increase the value twice in different threads because we were in a multithreaded environment, we had a Race condition, which means that now we don’t have a deterministic operation because sometimes it could be one, and sometimes it could be two.

##### How to solve the above problem?

There are many ways to resolve the above problem. The first mechanism that we’re going to look at to deal with the problems of having a variable edited by multiple threads is Interlocked.

##### Interlocked in C#:

The Interlocked Class in C# allows us to perform certain operations in an atomic way, which makes this operation safe to do from different threads on the same variable. That means Interlocked class gives us a few methods that allow us to perform certain operations safely or atomically, even if the code is going to be executed by several threads simultaneously.

##### Example to Understand Interlocked in C#:

First, we will see the example without using Interlocked and see the problem, and then we will rewrite the same example using Interlocked and will see how interlocked solve the thread safety problem.

Please have a look at the following example. In the below example, we have declared a variable and by using the Parallel For loop we are incrementing the value. As we know Parallel For loop uses multithreading so multiple threads trying to update (increment) the same IncrementValue variable. Here, as we are looping for 100000 times so we are expecting the value of the IncrementValue to be 100000.

**using** *System;*

**using** *System.Threading.Tasks;*

**namespace** *InterlockedDemo*

**{**

**class** Program

**{**

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

**{**

var IncrementValue = 0;

Parallel.For**(**0, 100000, \_ =**>**

**{**

//Incrementing the value

IncrementValue++;

**})**;

Console.WriteLine**(**"Expected Result: 100000"**)**;

Console.WriteLine**(**$"Actual Result: {IncrementValue}"**)**;

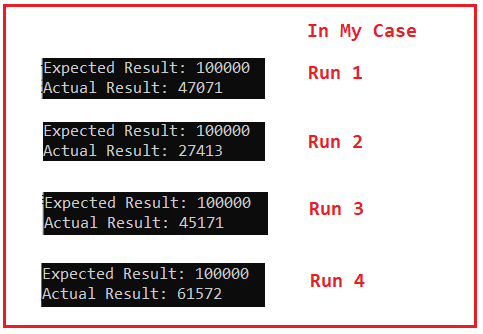
Console.ReadKey**()**;

**}**

**}**

**}**

Now, run the above code multiple times and you will get a different result each time, and you can see the difference between the Actual Result and the Expected Result as shown in the below image.



##### Example using Interlocked Class in C#:

The Interlocked Class in C# provides one static method called Increment. The Increment method increments a specified variable and stores the result, as an atomic operation. So, here we need to specify the variable with the ref keyword as shown in the below example.

**using** *System;*

**using** *System.Threading;*

**using** *System.Threading.Tasks;*

**namespace** *InterlockedDemo*

**{**

**class** Program

**{**

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

**{**

var IncrementValue = 0;

Parallel.For**(**0, 100000, \_ =**>**

**{**

//Incrementing the value

Interlocked.Increment**(ref** IncrementValue**)**;

**})**;

Console.WriteLine**(**"Expected Result: 100000"**)**;

Console.WriteLine**(**$"Actual Result: {IncrementValue}"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

###### **Output:**

Example using Interlocked Class in C#

Now, no matter how many times you execute the above code, you will get the same output. As you can see in the above output image, we are getting the Actual Result as the Expected Result. So, the Interlocked Class provides atomic operations for variables that are shared by multiple threads. That means the synchronization mechanism Interlocked allows us to avoid having race conditions by making the increment operation Atomic.

##### **What is Interlocked Class in C#?**

If you go to the definition of Interlocked class, you will see that this class provides many static methods such as Increment, Decrement, Add, Exchange, etc as shown in the below image to perform atomic operations on the variable. The Interlocked class belongs to the System.Threading namespace.

Following are the methods provided by the C# Interlocked class.

1. **Increment():** This method is used to increment a variable’s value and store its result. Int32 and Int64 integers are its legal parameters.
2. **Decrement():** This method is used to decrement a variable’s value and store its result. Int32 and Int64 integers are its legal parameters.
3. **Exchange():** This method is used to exchange values between variables. This method has seven overloaded versions based on the different types it can accept as its parameter.
4. **CompareExchange():** This method compares two variables and stores the result of the comparison in another variable. This method also has seven overloaded versions.
5. **Add():** This method is used to add two integer variables and update the result in the first integer variable. It is used to add integers of type Int32 as well as Int64.
6. **Read():** This method is used to reads an integer variable. It is used to read an integer of type Int64.

So, instead of addition, subtraction, and assignment operators we can use the Add, Increment, Decrement, Exchange, and CompareExchange methods. We have already seen the example of the Increment method. Now, let us see the examples of other static methods of the Interlocked class in C#.

##### **Interlocked.Add Method in C#:**

There are two overloaded versions of the Add method available in Interlocked Class. They are as follows:

1. **public static long Add(ref long location1, long value):** This method adds two 64-bit integers and replaces the first integer with the sum, as an atomic operation.
2. **public static int Add(ref int location1, int value):** This method adds two 32-bit integers and replaces the first integer with the sum, as an atomic operation. It returns the new value stored at the location1.

Following are the Parameters:

1. **location1:** A variable containing the first value to be added. The sum of the two values is stored in location1.
2. **value:** The value to be added to the location1 variable.

##### **Example to Understand Interlocked Add Method in C#:**

The following example shows the use of Add method of the Interlocked class.

using System;

using System.Threading;

using System.Threading.Tasks;

namespace InterlockedDemo

{

class Program

{

static void Main(string[] args)

{

long SumValueWithoutInterlocked = 0;

long SumValueWithInterlocked = 0;

Parallel.For(0, 100000, number =>

{

SumValueWithoutInterlocked = SumValueWithoutInterlocked + number;

Interlocked.Add(ref SumValueWithInterlocked, number);

});

Console.WriteLine($"Sum Value Without Interlocked: {SumValueWithoutInterlocked}");

Console.WriteLine($"Sum Value With Interlocked: {SumValueWithInterlocked}");

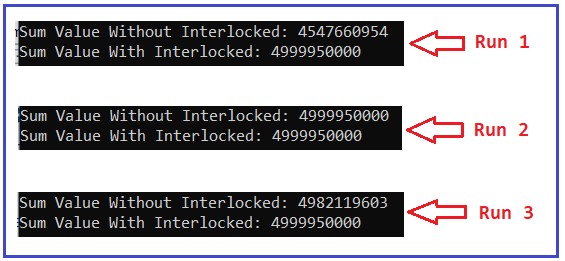
Console.ReadKey();

}

}

}

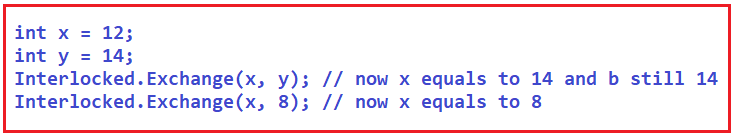
###### **Output:**



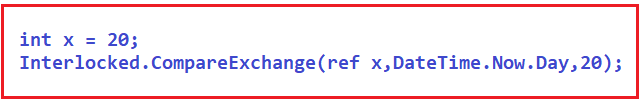
As you can see in the above image, Sum Value with interlocked always gives you the same result while the Sum value without Interlocked gives you a different result. That means Interlocked.Add method provides thread safety to the shared variable.

##### **Exchange and CompareExchange Method of Interlocked Class:**

The Exchange method of Interlocked Class in C# is atomically exchanging the values of the specified variables. The second value could be a hard-coded value or a variable. Only the first variable in the first parameter will be replaced by the second. For a better understanding, please have a look at the below image.



The CompareExchange method of Interlocked Class in C# is used to combine two operations. Comparing two values and storing the third value in one of the variables, based on the outcome of the comparison. If both are equal then replace the one used as the first parameter with the supplied value. For a better understanding, please have a look at the below image. Here, we create an integer variable and then assign the value 20 to it. Then we then call the Interlocked.CompareExchange method to compare the variable x with 20 and since they both are the same, so, it will replace x with DateTime. Now. Day, the current day of the month.



##### **Example to Understand Interlocked Exchange and CompareExchange Method in C#**

using System;

using System.Threading;

namespace InterlockedDemo

{

class Program

{

static long x;

static void Main(string[] args)

{

Thread thread1 = new Thread(new ThreadStart(SomeMethod));

thread1.Start();

thread1.Join();

// Written [20]

Console.WriteLine(Interlocked.Read(ref Program.x));

Console.ReadKey();

}

static void SomeMethod()

{

// Replace x with 20.

Interlocked.Exchange(ref Program.x, 20);

// CompareExchange: if x is 20, then change to current DateTime.Now.Day or any integer variable.

//long result = Interlocked.CompareExchange(ref Program.x, DateTime.Now.Day, 20);

long result = Interlocked.CompareExchange(ref Program.x, 50, 20);

// Returns original value from CompareExchange

Console.WriteLine(result);

}

}

}

**Output:**  
**20**  
**50**

##### **Interlocked vs Lock in C# from Performance Point of View:**

It is very easy to use the Interlocked methods in programs. But does it really perform faster than a lock? Let us see this with an example. In this benchmark, we have shown the 2 approaches in C#.

1. Version 1: We test a lock before an integer increment in the first loop. This code is longer and does not use Interlocked.
2. Version 2: This is the second version of the code. We test a call to Interlocked.Increment in the second loop.

using System;

using System.Diagnostics;

using System.Threading;

namespace InterlockedDemo

{

class Program

{

static object lockObject = new object();

static int \_test = 0;

const int \_max = 10000000;

static void Main()

{

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

// Version 1: use lock.

for (int i = 0; i < \_max; i++)

{

lock (lockObject)

{

\_test++;

}

}

stopwatch.Stop();

Console.WriteLine($"Result using Lock: {\_test}");

Console.WriteLine($"Lock took {stopwatch.ElapsedMilliseconds} Milliseconds");

//Reset the \_test value

\_test = 0;

stopwatch.Restart();

// Version 2: use Interlocked.

for (int i = 0; i < \_max; i++)

{

Interlocked.Increment(ref \_test);

}

stopwatch.Stop();

Console.WriteLine($"Result using Interlocked: {\_test}");

Console.WriteLine($"Interlocked took {stopwatch.ElapsedMilliseconds} Milliseconds");

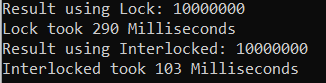
Console.ReadKey();

}

}

}

###### **Output:**



Here you can see the result is correct in both the approaches because the value printed is equal to the total number of increment operations. If you observe the Interlocked.Increment was several times faster, requiring only 103 Milliseconds versus 290 Milliseconds for the lock construct. The time may vary on your machine.

##### **When to use Lock over Interlocked in C#?**

So, if the same task is achieved using both lock and interlocked with thread-safety then it is recommended to use Interlocked in C#. However, in some situations is there where Interlocked will not work and, in those situations, we need to use the lock. Let us understand this with an example. Please have a look at the following code.

using System;

using System.Threading;

using System.Threading.Tasks;

namespace InterlockedDemo

{

class Program

{

static void Main(string[] args)

{

long IncrementValue= 0;

long SumValue = 0;

Parallel.For(0, 100000, number =>

{

Interlocked.Increment(ref IncrementValue);

Interlocked.Add(ref SumValue, IncrementValue);

});

Console.WriteLine($"Increment Value With Interlocked: {IncrementValue}");

Console.WriteLine($"Sum Value With Interlocked: {SumValue}");

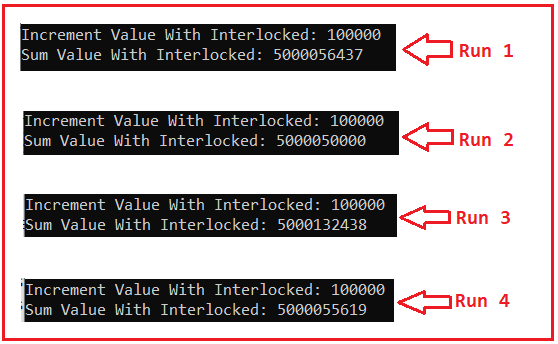
Console.ReadKey();

}

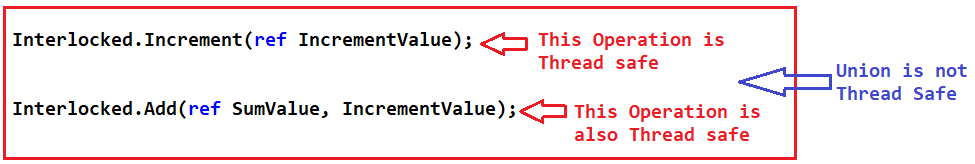
}

}

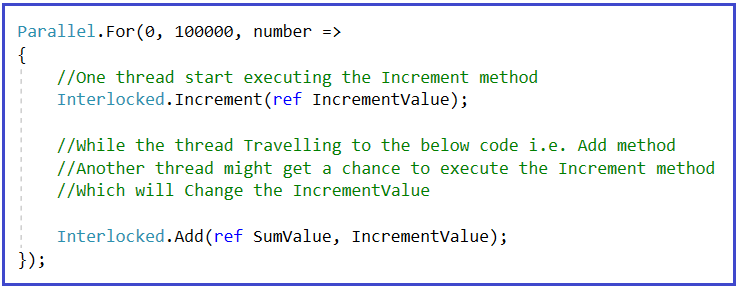
###### **Output:**



As you can see in the above output, we are getting different Sum Value even after using Interlocked. Why? This is because there is a Race condition. Then you might be thinking that we are using Interlocked.Add method and there should not be any race conditions. Right? But there is a Race condition because of the following.



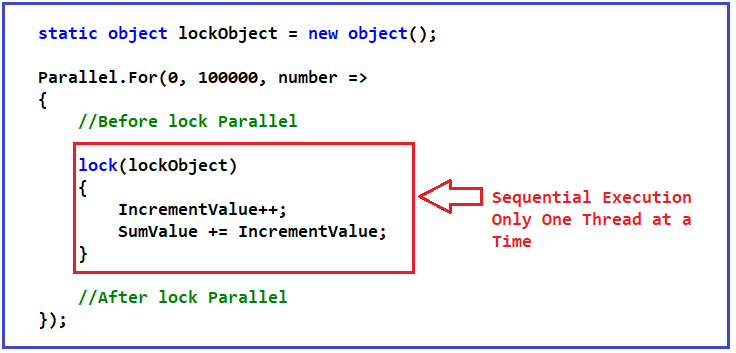
Individually Increment and Add methods are thread-safe but the union of these two methods is not thread-safe. For a better understanding, think of the code in the following way. One thread starts executing the Increment method. While the thread Travelling to the Add method, another thread might get a chance to execute the Increment method which will Change the IncrementValue again. And therefore, the IncrementValue variable value has already been incremented before the first threat had time to make that sum. So, this is the reason why there is a risk condition.



So, there is a Race condition in between these two operations i.e. Increment and Add. Individually, they both are thread-safe, together, they are not thread-safe because while Thread one is traveling from Increment Method to Add Method, multiple, multiple, multiple threads could execute the Increment method. And that is why there is a race condition.

##### **How to Solve the above Race Condition in C#?**

Since we have several operations and we want them to be executed only by one thread at a time, we can use the lock. In order to use the lock, we need to instantiate an object. It is recommended to have a dedicated object for the lock. The idea is that we make locks based on objects. For a better understanding please have a look at the below example. Whatever code is present before and after the lock block will be executed in parallel and the lock block code will be executed in sequential i.e. only one thread can access the lock block at a time.



So, if there are, say, two threads trying to access the lock block, only one thread will be able to enter while the order waits. And when thread one exits the lock block, then thread two will be able to enter the lock block and run the two lines of code. The following is the complete example code.

using System;

using System.Threading.Tasks;

namespace InterlockedDemo

{

class Program

{

static object lockObject = new object();

static void Main(string[] args)

{

long IncrementValue= 0;

long SumValue = 0;

Parallel.For(0, 10000, number =>

{

//Before lock Parallel

lock(lockObject)

{

IncrementValue++;

SumValue += IncrementValue;

}

//After lock Parallel

});

Console.WriteLine($"Increment Value With lock: {IncrementValue}");

Console.WriteLine($"Sum Value With lock: {SumValue}");

Console.ReadKey();

}

}

}

###### **Output:**

Interlocked vs Lock in C#

Every time we run the application, we get back the same result and we get back the same result because we are using a synchronization mechanism that allows us to make multiple operation threads safe.

We limit a part of our code to be sequential, even if several threads try to execute that code at the same time. We use locks when we need to perform several operations or an operation not covered by Interlocked.

**Note:** Be careful when using the lock. Always have a dedicated object for the Lock in C#. Do not try to reuse the objects and also try to keep it simple. Try to make the least amount of work inside of a lock because having too much work inside of a lock could have an impact on the performance of your application.