Race Condition C#

A race condition occurs when two or more threads are able to access shared data and they try to change it at the same time. To fully understand a race condition we will first talk about shared resources and than discuss about what is a race condition in threading.

Shared Resources

Not all resources are meant to be used concurrently. Resources like integers and collection must be handled carefully when accessed through multiple threads, resources that are accessed and updated within multiple threads are known as Shared Resources. Let’s see an example,

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _01_Shared_Resources
{
    class Program
    {
        private static int sum;

        static void Main(string[] args)
    {
        //create thread t1 using anonymous method
        Thread t1 = new Thread(() => {
            for (int i = 0; i < 10000000; i++)
            {
                //increment sum value
                sum++;
            }
        });

        //create thread t2 using anonymous method
        Thread t2 = new Thread(() => {
            for (int i = 0; i < 10000000; i++)
            {
                //increment sum value
                sum++;
            }
        });


        //start thread t1 and t2
```

```
            t1.Start();

            t2.Start();


            //wait for thread t1 and t2 to finish their execution

            t1.Join();

            t2.Join();


            //write final sum on screen

            Console.WriteLine("sum: " + sum);


            Console.WriteLine("Press enter to terminate!");

            Console.ReadLine();

        }

    }

}
```

However,

There is really some problem with the code above because every time we run it we see different output. To truly understand the problem we must first understand what is a Race condition.

What Is Race Condition?

Race Condition is a scenario where the outcome of the program is affected because of timing. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don’t know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are “racing” to access/change the data.

In our case, the line which is causing race condition is sum++, though this line seems to single line code and must not affect with concurrency but this single line of code gets transformed into multiline processor level instructions by JIT at the time of execution, below is the example

```
mov eax, dword ptr [sum]

inc eax

mov dword ptr [sum], eax
```

So what happens when our multiple threads execute this part of the code.

Let’s assume there is this thread X and thread Y.

Suppose thread X reads the value of some variable and store in register X.eax for increment but after doing increment from value 0 to 1, X thread got suspended by Thread scheduler and Y thread start executing this part of the code where Y thread also reads the value of variable sum in register Y.eax and does the increment from value 0 to 1 and now after doing this increment both thread will update sum variable to 1 thus its value will be 1 even though both the threads incremented the value.

So in simple words, it’s just the race between threads X and Y to read and update the value of variable sum and thus cause the race condition.

But we can overcome this kind of problems using some of the thread synchronization techniques that are,

- Atomic Update
- Data Partitioning
- Wait-Based Technique

C# Thread Synchronization

Thread synchronization refers to the act of shielding against multithreading issues such as data races, deadlocks and starvation.

C# Interlocked

So as we saw processor increments the variable, written in a single line of C# code in three steps (three line of code) in processor-specific language, that is read, increment and update the variable

One way to tackle this problem is to carry out all this three operation in one single atomic operation. This can be done only on data that is word-sized. Here, by atomic I mean uninterruptable and word-sized means value that can fit in a register for the update, which is a single integer in our case.

However,

Today's processors already provide lock feature to carry out an atomic update on word-sized data. However, we can't use this processor specific instruction directly in C# code but there is Interlocked Class in DotNet Framework that is a wrapper around this processor-level instruction that can be used to carry out atomic operations like increment and decrement on a word-sized data.

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

```

```
namespace _03_Atomic_Update

{

    class Program

    {

        private static int sum;



        static void Main(string[] args)

        {

            //create thread t1 using anonymous method

            Thread t1 = new Thread(() => {

                for (int i = 0; i < 10000000; i++)

                {

                    //use threading Interlocked class for atomic update

                    Interlocked.Increment(ref sum);

                }

            });



            //create thread t2 using anonymous method

            Thread t2 = new Thread(() => {

                for (int i = 0; i < 10000000; i++)

                {
```

```
                //use threading Interlocked class for atomic update
                Interlocked.Increment(ref sum);
            }
        });


        //start thread t1 and t2
        t1.Start();
        t2.Start();

        //wait for thread t1 and t2 to finish their execution
        t1.Join();
        t2.Join();

        //write final sum on screen
        Console.WriteLine("sum: " + sum);

        Console.WriteLine("Press enter to terminate!");
        Console.ReadLine();
    }
}
```

```
}
```

Data Partitioning

Data Partitioning is actually kind of strategy where you decide to process data by partitioning it for multiples threads. Its kind of “you do that and I will do that” strategy.

To use data partitioning you must have some domain-specific knowledge of data (such as an array or multiple files manipulation), where you decide that one thread will process just one slice of data while other thread will work on another slice. Let’s see an example,

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _04_Data_Partitioning
{
    /// <summary>
    /// This program calculate the sum of array elements using
    /// data partition technique. Here we split the data into
    /// two halves and calculate the sum1 and sum2 for each half
    /// in thread t1 and t2 respectively, then finally we print
/// the final sum on screen by adding sum1 and sum2.
/// </summary>
class Program
{
    private static int[] array;
    private static int sum1;
    private static int sum2;

    static void Main(string[] args)
    {
        //set length for the array size
        int length = 1000000;
        //create new array of size lenght
        array = new int[length];

        //initialize array element with value of their respective index
        for (int i = 0; i < length; i++)
        {
            array[i] = i;
        }

        //index to split on
        int dataSplitAt = length / 2;

        //create thread t1 using anonymous method
        Thread t1 = new Thread(() =>
        {
            //calculate sum1
            for (int i = 0; i < dataSplitAt; i++)
            {
                sum1 = sum1 + array[i];
            }
        });


        //create thread t2 using anonymous method
        Thread t2 = new Thread(() =>
        {
            //calculate sum2
            for (int i = dataSplitAt; i < length; i++)
            {
                sum2 = sum2 + array[i];
            }
        });


        //start thread t1 and t2
        t1.Start();
        t2.Start();

        //wait for thread t1 and t2 to finish their execution
        t1.Join();
        t2.Join();

        //calculate final sum
        int sum = sum1 + sum2;

        //write final sum on screen
        Console.WriteLine("Sum:" + sum);

        Console.WriteLine("Press enter to terminate!");
        Console.ReadLine();
    }
    }
}
```

However,

This technique can’t be adapted for every scenario there may be a situation where one slice of data depends on the output of the previous slice of data one example of this scenario is Fibonacci series where, data[n]=data[n-1]+data[n-2] in such a situation data partitioning can’t be adopted.

Wait Based Synchronization

The third technique is a Wait-Based technique which is a very sophisticated way to handle the race condition, used in a situation where above two methods can’t be adopted that easily. In this technique, a thread is blocked until someone decides its safe for them to proceed.

Suppose there are two threads namely X and Y and both want to access some resource R

Now to protect this resource we choose some lock primitive or synchronization primitive as LR (primitive here is some primitive type like int or array)

Now when thread X want to access resource R it will first acquire the lock ownership of LR, once this thread got ownership of LR it can access the resource R safely. As long as thread X has this ownership no other thread can access the LR ownership

While X has ownership if Y request to acquire the ownership of lock LR it requests will get block until thread X releases its ownership.

Wait Based Primitives in CLR

.Net has following Wait Based Primitives that you can use to apply Wait-Based technique.

They all share the same basic usage

- Access the lock ownership
- Manipulate the protected resource
- Release the lock ownership

C# Monitor Class

The Monitor class allows you to synchronize access to a region of code by taking and releasing a lock on a particular object by calling the Monitor.Enter, Monitor.TryEnter, and Monitor.Exit methods. Object locks provide the ability to restrict access to a block of code, commonly called a critical section. While a thread owns the lock for an object, no other thread can acquire that lock. You can also use the Monitor class to ensure that no other thread is allowed to access a section of application code being executed by the lock owner unless the other thread is executing the code using a different locked object.

C# Mutex Class

You can use a Mutex object to provide exclusive access to a resource. The Mutex class uses more system resources than the Monitor class, but it can be marshaled across application domain boundaries, it can be used with multiple waits, and it can be used to synchronize threads in different processes. For a comparison of managed synchronization mechanisms, see Overview of Synchronization Primitives.

C# ReaderWriterLock Class

The ReaderWriterLockSlim class addresses the case where a thread that changes data, the writer, must have exclusive access to a resource. When the writer is not active, any number of readers can access the resource. When a thread requests exclusive access, subsequent reader requests block until all existing readers have exited the lock, and the writer has entered and exited the lock.

C# Monitor

What is Monitor In C#?

Monitor class is one of the wait based synchronization primitive that provides gated access to the resource. It gates or throttles the access to the shared resource.

So, Monitor assure that thread access the shared resource one thread at a time

Here is the code to use Monitor for shared resources to avoid the race condition

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _05_Monitor_Class_Usage
{
    class Program
    {
```

```
private static int sum;
private static object _lock = new object();

static void Main(string[] args)
{

    //create thread t1 using anonymous method
    Thread t1 = new Thread(() => {
        for (int i = 0; i < 10000000; i++)
        {
            //acquire lock ownership
            Monitor.Enter(_lock);

            //increment sum value
            sum++;

            //release lock ownership
            Monitor.Exit(_lock);
        }
    });

    //create thread t2 using anonymous method
    Thread t2 = new Thread(() => {
        for (int i = 0; i < 10000000; i++)
        {
            //acquire lock ownership
            Monitor.Enter(_lock);

            //increment sum value
            sum++;

            //release lock ownership
            Monitor.Exit(_lock);
        }
    });


    //start thread t1 and t2
    t1.Start();
    t2.Start();

    //wait for thread t1 and t2 to finish their execution
```

```
            t1.Join();

            t2.Join();


            //write final sum on screen

            Console.WriteLine("sum: " + sum);


            Console.WriteLine("Press enter to terminate!");

            Console.ReadLine();

        }

    }

}
```

Here, sum++ is considered as the critical section, as this operation should be done in a thread-safe manner we use the monitor to carry out this operation as one thread at a time.

As you saw Monitor use Enter and Exit method which accepts an object to associate with lock primitive

Why?

locks are created by CLR only when you use monitor API to get acquisition of a lock (for performance reasons), so basically they are maintained as a table of the lock by CLR.

So,

When you pass this object in monitor method this object stores the index of the lock object created by CLR in their header as information to use this lock for gated access to the resource.

In short, this object is not the actual lock but stores reference to the lock object use by monitor class to access the resource in wait based manner.

Basically, you can use any type of object to associate with a lock. However, the recommended method is to use private objects and always avoid string as lock objects as the issue they cause due to their implementation method in CLR

Monitor Class Usage In C#

So Monitor has the same wait based technique usage. When you call the Monitor.Enter method you get the ownership of the lock, then you perform your thread-safe operation and then release the lock using Monitor.Exit.

Always remember this is the programmer understanding to use wait based technique thoughtfully at the required places as there are no physical restrictions on how you access the resource or implement the model. It's just you deciding how to implement the flow of multithreading program and shared resources.

Exception Aware Monitor Usage

Now,

Consider the same example code above and there are two threads trying to acquire the lock X and Y now thread X got the ownership and Y got blocked until X releases the ownership.

However,

Before releasing the lock thread X threw some runtime exception error hence it will exit the code before releasing the lock, as a result, thread Y will get blocked forever.

We want to throw the exception but also want to release the lock.

So,

To overcome this problem we have to use proper try-finally construct to manage the exception (not handle it). Let's see the code how to do it,

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace _06_Exception_Aware_Monitor
{
    class Program
    {
        private static int sum;
        private static object _lock = new object();

        static void Main(string[] args)
        {

            //create thread t1 using anonymous method
            Thread t1 = new Thread(() => {
                for (int i = 0; i < 10000000; i++)
                {
```

```
                //acquire lock ownership
                Monitor.Enter(_lock);
                try
                {
                    //increment sum value
                    sum++;
                }
                finally
                {
                    //release lock ownership
                    Monitor.Exit(_lock);
                }
            }
        });

        //create thread t2 using anonymous method
        Thread t2 = new Thread(() => {
            for (int i = 0; i < 10000000; i++)
            {
                //acquire lock ownership
                Monitor.Enter(_lock);
                try
                {
                    //increment sum value
                    sum++;
                }
                finally
                {
                    //release lock ownership
                    Monitor.Exit(_lock);
                }
            }
        });


        //start thread t1 and t2
        t1.Start();
        t2.Start();

        //wait for thread t1 and t2 to finish their execution
        t1.Join();
        t2.Join();
```

```

            //write final sum on screen
            Console.WriteLine("sum: " + sum);

            Console.WriteLine("Press enter to terminate!");
            Console.ReadLine();
        }
    }
}
```

C# Lock Keyword

Some high level languages have syntactic sugar which reduces the amount of code that must be written in some common situation like above.

C# has this lock syntax for the same code we wrote above. Here is the code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

```csharp

namespace _07_Lock_Keyword
{
    class Program
    {
        private static int sum;
        private static object _lock = new object();

        static void Main(string[] args)
        {

            //create thread t1 using anonymous method
            Thread t1 = new Thread(() => {
                for (int i = 0; i < 10000000; i++)
                {
                    lock (_lock)
                    {
                        //increment sum value
                        sum++;
                    }
                }
            });

            //create thread t2 using anonymous method
            Thread t2 = new Thread(() => {
                for (int i = 0; i < 10000000; i++)
                {
                    lock(_lock)
                    {
                        //increment sum value
                        sum++;
                    }
                }
            });


            //start thread t1 and t2
            t1.Start();
            t2.Start();

            //wait for thread t1 and t2 to finish their execution
            t1.Join();
```

```
            t2.Join();

            //write final sum on screen
            Console.WriteLine("sum: " + sum);

            Console.WriteLine("Press enter to terminate!");
            Console.ReadLine();
        }
    }
}
```

so we simply use the lock keyword syntax and write critical section code in its body and compiler will generate the Exception Aware Monitor code for us.