**Chapter 7: Synchronization Constructs**

**Overview**

When we design a highly scalable application accessed by many concurrent users, there is a very high possibility that same data is read/write my multiple users at the same time. If the write operation on the shared data across threads is not handled correctly it will lead to unexpected output. Let’s see this with an example of transactions in a bank account

1. Initial amount in bank account 1000 units.
2. A withdraw request of 500 units is placed through an ATM
3. Sametime another with draw request of 600 units is placed through internet banking

Assuming both transactions are initiated exactly at the same time both would see a balance of 1000 units and will allow both the transactions to pass successfully, however, this will lead to inconsistent state with data. If handled correctly one of the steps should fail with an exception like “Insufficient balance”. This handling of data across threads is done using synchronization and will help getting a predictable outcome. Let’s see this with an example in which we add money to bank account through multiple concurrent transactions. We start creating BankAccount Class and add methods to increase available balance, will start with creating class and 2 private variables accountBalance and numberOfTransactions

public class BankAccount

{

private long accountBalance;

private int numberOfTransactions;

public int NumberOfTransactions

{

get

{

return numberOfTransactions;

}

}

public BankAccount(long initialAccountBalance)

{

this.accountBalance = initialAccountBalance;

numberOfTransactions = 0;

}

public long ShowBalance()

{

return this.accountBalance;

}

}

Now add a private method AddBalanceToAcccount as below to BankAccount class that takes amount as a parameter and increments account Balance and numberOfTransactions

async Task AddBalanceToAcccount(long amount)

{

await Task.Delay(1);

accountBalance = accountBalance + amount;

numberOfTransactions = numberOfTransactions + 1;

}

Create another public async method AddMoneyToAccountAsync which will run a loop and call AddBalanceToAcccount, basically what we are doing here is parallelly simulating 50 transactions. AddMoneyToAccountAsync will look like below

/// <summary>

/// Add money to account through multiple transactions

/// </summary>

public async Task AddMoneyToAccountAsync()

{

var tasks = new Task[50];

for (int i = 1; i <= tasks.Length; i++)

{

tasks[i - 1] = AddBalanceToAcccount(i);

}

await Task.WhenAll(tasks);

}

Calling it through a console application expected value of variable accountBalance for 50 iterations should be 1275

static async Task Main(string[] args)

{

BankAccount bankAccount = new BankAccount(0);

Console.WriteLine($"Initial Balance {bankAccount.ShowBalance()}");

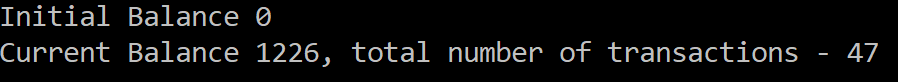
await bankAccount.AddMoneyToAccountAsync();

Console.WriteLine($"Current Balance {bankAccount.ShowBalance()}, total number of transactions - {bankAccount.NumberOfTransactions}");

Console.Read();

}

Here is the output on executing above code



**Figure 7.1 – Output of application without synchronization**

We can clearly see that it’s lesser than wat is expected and in reality what has happened here is since multiple threads are parallelly accessing same variable at the same time and there is no restriction on overwriting values and at some point few of the threads have overwritten value of variable accountBalance and hence unpredicted outcome. Same has happened with variable numberOfTransactions.

To overcome this we need a mechanism to stop multiple threads parallely accessing shared resources which is what synchronization is about.Hence to fix above code we can use one of the synchronization construct, in this case locks. With that implemented at any given point in time only one thread can access the resources in another words only one thread can enter critical section and all other threads that need access to cricitcal section shall wait till lock is released by owning thread.

So we create a locking object and lock critical section using that as and our method will look like this

//Lock

object locker = new object();

async Task AddBalanceToAcccount(long amount)

{

await Task.Delay(1);

lock (locker)

{

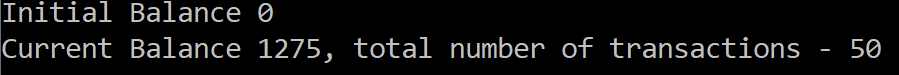
accountBalance = accountBalance + amount;

numberOfTransactions = numberOfTransactions + 1;

}

}

Once syncronization is implemented using a lock, here is the output of the sample



**Figure 7.2 – Output of application with synchronization**

As you see output is what was predicted, we can clearly see that if Synchronization is not implemented for a shared resources in multi thread environment there is a high possibility of data getting corrupted and that’s when it becomes really critical that we implement proper synchronization constructs to achieve predictable results.

To do – Explain what is thread safety

Synchronization can be achieved through various constructs provided by .net

1. Blocking Methods
2. Locking Constructs
3. Signaling Constructs
4. Non-Blocking Synchronization Constructs

We will deep dive into each of these in next sections

**Locking constructs**

Locking constructs are types in .NET that help in synchronization for a shared resource between threads or coordinating insert/updates/overwrites among threads. They are primarily categorized into

* Exclusive – Exclusive locks are the types which allow to lock a resource and resource cannot be modified until lock is released, while an object is exclusively locked no other thread can read/update that object. Exclusive locks are always acquired by one single thread at any point in and all other threads must wait till the acquiring thread release the lock. Exclusive locks are supported in .NET through
  + lock (Monitor.Enter/Monitor.Exit)
  + Mutex
  + SpinLock
* Non – exclusive locks - These are the types which allows limited number of threads to access a shared resource i.e. if 10 threads are trying to access a resource using a non-exclusive lock shared resource access can be restricted to say threads. Usually it is like multiple reads can be performed however shared resource cannot be modified until the read lock is released. .NET supports non-exclusive locks through
  + Semaphore (Non - Exclusive)
  + SemaphoreSlim (Non - Exclusive)
  + Reader/Writer locks (Non - Exclusive)

**Lock or Monitor.Enter/Monitor.Exit (Exclusive)**

Lock statement is the easiest way to achieve synchronization in multi-threaded code where any shared resource within the scope of lock can be accessed using only one thread at point in time. To lock a shared resource using lock statement we need a create an object and wrap it inside lock keyword just like below –

object locker = new object(); //Declare lock object

async Task AddBalanceToAcccount(long amount)

{

await Task.Delay(1);

lock (locker) //Locking accountBalance variable

{

accountBalance = accountBalance + 10;

Console.WriteLine("balance updated");

} //Un-Locking accountBalance variable

}

In this example if multiple threads parallelly call AddBalanceToAcccount only one thread is allowed to access code block inside lock statement so only one thread can modify variable accountBalance at any point in time based on first come first serve basis. All the other threads will continue to wait until lock is released by the thread that acquired it, what this actually means no matter the number of threads parallelly call AddBalanceToAcccount method, code from lock(locker) will always execute sequentially hence preventing data corruption.

Lock statement is in-fact syntactic sugar for Monitor.Enter and Monitor.Exit so here’s how compiler converts preceding code-

bool lockAcquired = false;

try

{

Monitor.Enter(locker, ref lockAcquired);

accountBalance = accountBalance + amount;

numberOfTransactions = numberOfTransactions + 1;

}

finally

{

if (lockAcquired)

{

Monitor.Exit(locker);

}

}

Output will remain same in either case and it upto the developer to use whichever syntax they are comfortable with. However, for advanced thread coordination Monitor class is helpful as it has other methods like Monitor.Wait/Monitor.Pulse/Monitor.Pulseall that can be used for signaling off course these methods can be used in tandem with lock but using same construct across makes it more readable. There are certain things that needs to be remembered for using locks

* We should always lock on a reference type - The reason behind that is since Enter method expects an object and if a value type is passed to it boxing would occur which will create a copy of the type passed and hence when Exit method is called it will be a different copy again which means that they are operating on different objects. If we change locker to a value type like int we will get a run time exception - System.Threading.SynchronizationLockException: 'Object synchronization method was called from an unsynchronized block of code.'
* Double check acquiring lock as it helps in improving performance specially in cases where code block inside lock needs to be executed only once. For example – Singleton Class or any instantiation code which needs to occur only if object is null.
* Exception handling in locks is nothing different that a typical try catch block in calling method, it is very important that unhandled exceptions are handled through a try catch block or less any exception with in the code block of a lock can cause application to crash.

One last point is to avoid locks if possible, as such locking is not time consuming or going to degrade performance however pausing threads and then resuming do results in some lag. So, unless and absolutely necessary avoid locks, there are types available in .Net that can be used instead of using locks like instead of Dictionary use ConcurrentDictionary

**Mutex (Exclusive)**

Mutex is just like lock (full form mutually exclusive lock), however scope of locking spawns across processes i.e. if multiple instances of same process is running mutex can be used to execute a code block by a single thread across processes. In .Net mutex can be created by creating object of System.Threading.Mutex class, the following example will show on how to create and use Mutex to achieve synchronization.

This example is a simple file create (or file upload class) class where we are a writing a file to a disk, So we will create a simple class called and add a method WriteTextAsync that takes filename as input and writes some data into that file. Class and method implementation will look like below

public class FileUpload

{

private async Task WriteTextAsync(string fileName)

{

string text = $"Mutex is just like lock (full form mutually exclusive lock), however scope of locking spawns across processes i.e. " +

"if multiple instances of same process running mutex can be used to execute a code block by a single thread across processes.";

byte[] encoding = Encoding.Unicode.GetBytes(text);

await Task.Delay(1);

using (var mutex = new Mutex(false, fileName))

{

mutex.WaitOne();

using (FileStream fs = new FileStream(fileName, FileMode.Append, FileAccess.Write, FileShare.None, bufferSize: 64, useAsync: true))

{

fs.Write(encoding, 0, encoding.Length);

}

mutex.ReleaseMutex();

}

}

}

Now we will call this method through another async method that will simulate parallel calls through tasks. That method will look like below and will be added as public method in the class.

public async Task CreateorUpdateFiles()

{

var tasks = new Task[50];

for (int i = 1; i <= tasks.Length; i++)

{

tasks[i - 1] = WriteTextAsync($"File{i % 5}.txt");

}

Stopwatch timer = new Stopwatch();

timer.Start();

await Task.WhenAll(tasks);

Console.WriteLine($"Time ellapsed {timer.ElapsedMilliseconds}");

}

So we are simulating 50 parallel calls in this method and after every 5th iteration writing into the same file, this method also has timer to calculate the time taken for this opertaion. Now we will use this class in main mthod of simple console application, so create a console appliaction and this class to that console application. Create an object of class FileUpload and call CreateorUpdateFiles method. Our main method will look like below

static async Task Main(string[] args)

{

Console.WriteLine("Writing file to disk");

FileUpload fileupload = new FileUpload();

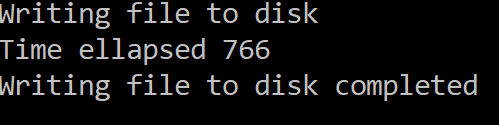
await fileupload.CreateorUpdateFiles();

Console.WriteLine("Writing file to disk completed");

Console.Read();

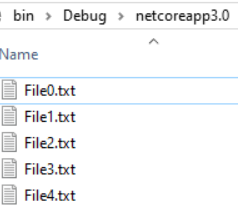
}

Once we run this application we can see 5 files getting created and each will have the text 10 times (as we are looping for 50 times and writing to same file after every 5th iteration). Output will look like below



**Figure 7.3 – Output of FileUpload application with synchronization using Mutex**

If we go to the debug folder we can see 5 files are created and content of the file would be the string that we passed.



**Figure 7.4 – Files created in debug folder**

We can clearly see that there is no loss of data i.e. each file has 10 copies of the string that we passed and there is no run time exception. To see the benfit of Mutex let’s remove the mutex and run the applciation, our WriteTextAsync will look like below

private async Task WriteTextAsync(string fileName)

{

string text = $"Mutex is just like lock (full form mutually exclusive lock), however scope of locking spawns across processes i.e. " +

"if multiple instances of same process running mutex can be used to execute a code block by a single thread across processes.";

byte[] encoding = Encoding.Unicode.GetBytes(text);

await Task.Delay(1);

using (FileStream fs = new FileStream(fileName, FileMode.Append, FileAccess.Write, FileShare.None, bufferSize: 64, useAsync: true))

{

fs.Write(encoding, 0, encoding.Length);

}

}

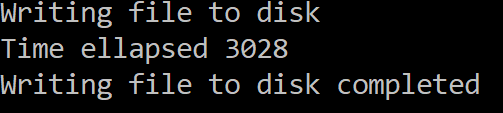
Once we run the application now we will see below exception which is expected because there file is locked by one of the Thread for adding data and another thread parallelly tries to do same thing and raises an access exception.

System.IO.IOException

HResult=0x80070020

Message=The process cannot access the file '..\netcoreapp3.0\File0.txt' because it is being used by another process.

In this scenario we can use lock as well and will get same output, however it doesn’t makes sense to lock writing into a different file as lock will allow to writing into any file sequentially i.e. if currently a thread is writing into file1, lock will block writing into any other file also and that’s why a named mutex would be better here, considering the performance impact as code is blocked only for specific files. For testing purpose removing mutex and adding a lock would result in significant dip in performance which we can see in below output



**Figure 7.4 – Output of FileUpload application with synchronization using lock**

So with this we can say lock and Mutex can be used to achive synchronization however to lock a block of code or a resource across process named Mutex can be used.

Note - In above example we can use WriteAsync instead of Write however that will result in an exception as Mutex has thread affinity which means thread calling waitone needs to call release method and since code after await would run on a different thread it would give an exception – “[Object synchronization method was called from an unsynchronized block of code. Exception on Mutex.Release()](https://stackoverflow.com/questions/9017521/object-synchronization-method-was-called-from-an-unsynchronized-block-of-code-e)”

To avoid this exception we need to use advanced synchrnization construct called AutoResetEvent which we will see later in this chapter.

**Some important facts about Mutex**

* Mutrex has thread affinity,so the thread locking a resource needs to unlock the reource

**SpinLock (Exclusive)**

Spinlock is another form of exclusive lock which will synchronize access to shared resource however there is no thread context switching. So going back to all other locking techniques whenever a thread is blocked to access a shared resource it stops consuming any CPU cycles by giving up its processor time slice and causing a context switching in thread, same thing happens when thread is unblocked from blocked state. Although this context switching leads only to a few milliseconds delay but at large scale this is still an overhead.

So if there is a shared resource and needs locking for a very few milliseconds it would be better to not block all the threads that needs to access shared resources, but to just continue spinning something like calling a while loop until the shared resource is unblocked. This can be achieved in .net using a SpinLock class , let’s take AddBalanceToAccount method of BankAccount class this time we will use a SpinLock to synchronize access to accountBalance variable. We will first declare an object of SpinLock class that looks like this

SpinLock spinLock = new SpinLock();

Then use this lock to protect variables accountBalance and numberOfTransactions by calling Enter method of SpinLock class, this method accepts a Boolean variable which needs to be false before calling this method as once lock is acquired spinlock set this variable to true. This variable helps if there is any exception after lock is acquire, lock can be released safely. This is how our AddBalanceToAccount method will look like

async Task AddBalanceToAcccount(long amount)

{

await Task.Delay(1);

bool lockAcquired = false;

try

{

spinLock.Enter(ref lockAcquired);

accountBalance = accountBalance + amount;

numberOfTransactions = numberOfTransactions + 1;

}

finally

{

if (lockAcquired)

{

spinLock.Exit();

}

}

}

Once we run this code with a timer through AddMoneyToAccountAsync output will look like below

public async Task AddMoneyToAccountAsync()

{

Stopwatch timer = new Stopwatch();

timer.Start();

var tasks = new Task[99999];

for (int i = 1; i <= tasks.Length; i++)

{

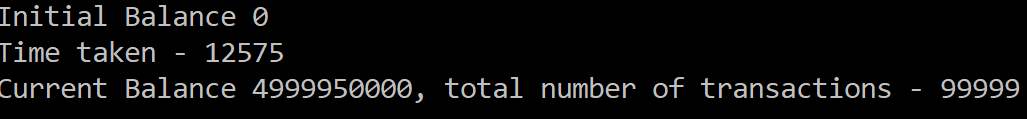
tasks[i - 1] = AddBalanceToAcccount(i);

}

await Task.WhenAll(tasks);

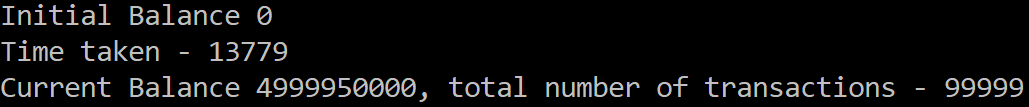
Console.WriteLine($"Time taken - {timer.ElapsedMilliseconds}");

}



**Figure 7.5 – Output of using spinlock**

Just for testing purpose replacing spinlock with a lock and checking output we get below and we can see that time taken has slightly goe up and that’s because there isn’t any context switching with spin lock



**Figure 7.5 – Output of using lock**

**Some important facts about spinlock**

* Use spinlock only for locking code that executes realy fast perhaps few microseconds
* Always prefer locks over spinlock as spinlock although consumes less times consume lot of CPU

**Semaphore (Non - Exclusive)**

**SemaphoreSlim (Non - Exclusive)**

**Reader/Writer locks (Non - Exclusive)**

**Signaling constructs**

**AutoResetEvent**

**ManualResetEvent**

**ManualResetEventSlim**

**CountdownEvent**

**Barrier classes**

**Wait and Pulse**

**Nonblocking synchronization constructs**

**Thread.MemoryBarrier**

**Thread.VolatileRead**

**Thread.VolatileWrite**

**The volatile keyword**

**The Interlocked class**

**SpinWait**

**Blocking Methods**

**Sleep**

**Join**

**Task.Wait**