**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()}");

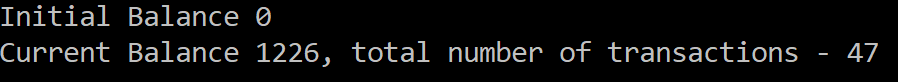
await bankAccount.AddMoneyToAccountAsync();

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)

{

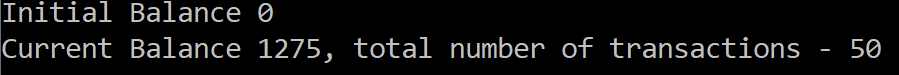
accountBalance = accountBalance + amount;

numberOfTransactions = numberOfTransactions + 1;

}

}

Once syncronization is implemented usin a lock here is the output of the



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

As you see output is wat 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. 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 AddBalanceToAcccount 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.

**Mutex (Exclusive)**

**SpinLock (Exclusive)**

**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**

**Blocking Methods**

**Sleep**

**Join**

**Task.Wait**