**Chapter 7: Synchronization Constructs**

**Overview**

When we design a high 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 it will lead to unexpected output. Let’s see this with an example of money 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, it will look something like this

public class BankAccount

{

private long accountBalance;

private int numberOfTransactions;

public int NumberOfTransactions

{

get

{

return numberOfTransactions;

}

}

public BankAccount(long initialAccountBalance)

{

this.accountBalance = initialAccountBalance;

numberOfTransactions = 0;

}

/// <summary>

/// Add money to account through multiple transactions

/// </summary>

public async Task AddMoneyToAccount()

{

var tasks = new Task[50];

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

{

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

}

await Task.WhenAll(tasks);

}

async Task AddBalanceToAcccount(long amount)

{

await Task.Delay(1);

accountBalance = accountBalance + amount;

numberOfTransactions = numberOfTransactions + 1;

}

public long ShowBalance()

{

return this.accountBalance;

}

}

Here we are increament accountBalance by calling metod AddBalanceToAcccount in a loop.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);

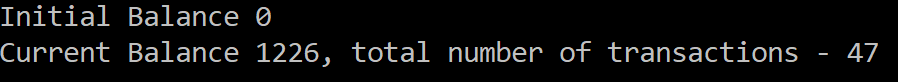
Console.WriteLine($"Initial Balance {bankAccount.ShowBalance()}");

await bankAccount.AddMoneyToAccount();

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

Console.Read();

}



**Fig 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 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 accessin shared resources which is what synchronization is about.So to fix above code we can use synchronization tecnique in this case locks so that 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 lockin 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)

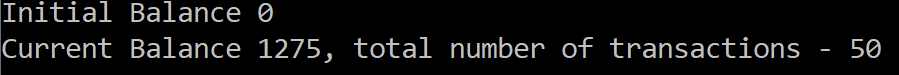
{

accountBalance = accountBalance + amount;

numberOfTransactions = numberOfTransactions + 1;

}

}



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

With this example we can clearly see why there is a need of Synchronization. Synchronization can be achieved through various strategies

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

**Blocking Methods**

**Sleep**

**Join**

**Task.Wait**

**Locking constructs**

**Exclusive**

**lock (Monitor.Enter/Monitor.Exit)**

**Mutex**

**SpinLock**

**Nonexclusive locking constructs**

**Semaphore**

**SemaphoreSlim**

**reader/writer locks**

**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**