**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()}");

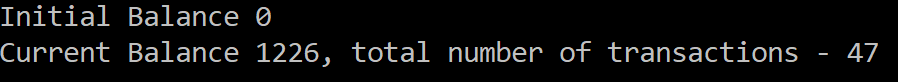
await bankAccount.AddMoneyToAccountAsync();

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)

{

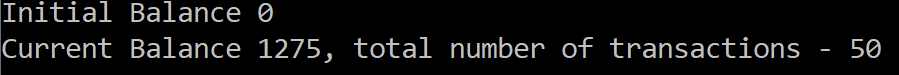
accountBalance = accountBalance + amount;

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 5 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)

Taking an analogy here:

Say there a travel website that allows you to book a seat in train with a restricted capacity of 50, however seat is allotted inside the train

* Here when we book a ticket it is guaranteed that we will get a seat, however since maximum number of seats are 50 exactly 50 people(threads) would be allowed to book a seat which means 50 people have an non - exclusive lock. Once a person exits from the train entry allowed for people in waiting list.
* Now inside train the seat occupied by person cannot be shared which means an exclusive lock is applied on the seat. Seat cannot be used until person releases it.

**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 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();

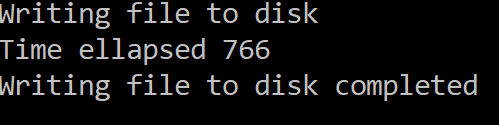
await fileupload.CreateorUpdateFiles();

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

Console.Read();

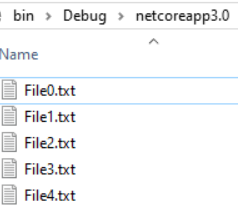
}

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);

}

}

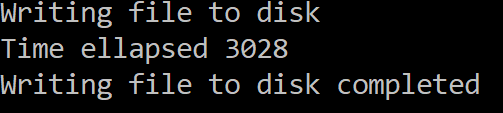
Once we run the application now we will see below exception which is expected because there file is locked by one of the Thread for adding data and another thread parallelly tries to do same thing and raises an access exception.

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 isn’t 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 which is 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++)

{

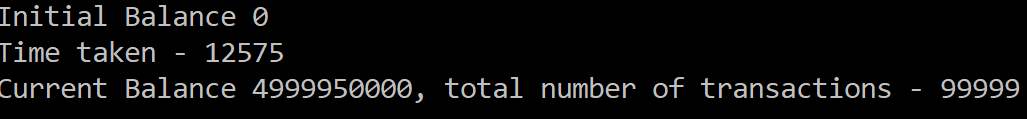
tasks[i - 1] = AddBalanceToAcccount(i);

}

await Task.WhenAll(tasks);

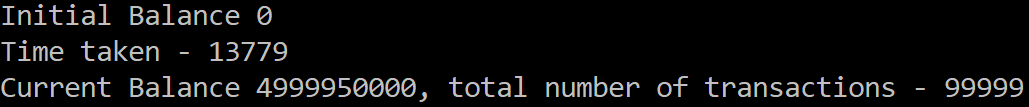
Console.WriteLine($"Time taken - {timer.ElapsedMilliseconds}");

}



**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 7.6 – Output of using lock**

**Some important facts about spinlock**

* Use spinlock only for locking code that executes realy fast perhaps few microseconds i.e. a briefnning is preferred over blocking
* Always prefer locks over spinlock as spinlock although consumes less times in some scenarios consume lot of CPU

**Semaphore (Non - Exclusive)**

Semaphore is a non-exclusive lock that supports synchronization by limiting the access to limited number of threads. So unlike mutex which allows only one thread to enter critical section semaphore allows set of threads to enter critical section, the number of threads that have access to shared resources is defined which creating semaphore. Semaphore is a non-exclusive lock and hence it should be used in situations where we need to lock a pool of resources for example in a client server scenario (where you own both client app and server APIs) say you want restrict number of calls your client app can make to your API concurrently from within a single instance, or something like thread pool or database connection semaphores are ideal fit.

In .Net semaphores can be created using [System.Threading.Semaphore](https://docs.microsoft.com/en-us/dotnet/api/system.threading.semaphore) class, an object of Semaphore class needs to be instantiated which has multiple constructors but 2 important parameters needs to be passed always

1. Initial number of entries
2. Maximum number of concurrent entries

So, a typically initialization will look like below

Semaphore semaphore = new Semaphore(0,3);

In this case we are telling initial concurrent requests allowed is 0 and will allow upto 3 concurrent threads after release i.e. in this case we are initializing semaphore however program has to wait until release is called at least once. If semaphore needs to enter semaphore immediately and allow maximum concurrency ideally both the parameters should have same value something like below

Semaphore semaphore = new Semaphore(3,3);

To acquire a semaphore we need to call WaitOne method of semaphore class and to release need to call Release method which accepts an optional integer to release semaphore that many number of times, if nothing is passed semaphore releases one thread. We will see this with a simple example of building water where water needs two Hydrogen threads and one Oxygen thread in this sequence. Restriction is that all the threads from one molecule bond before subsequent molecules from any other thread. We will take an input of series of strings a combination of ‘H’ and ‘O’ and handle accordingly. So, if input is HHHHOO output would be HHOHHO i.e. each character is processed by a thread and after second H program needs to wait for next O in the sequence to complete water molecule before processing other characters in sequence.

Will start with first creating Water class and we need two semaphores one for Hydrogen and another for Oxygen. Now since for every water molecule two Hydrogen threads are required we will initialize Hydrogen Semaphore with a maximum concurrent thread count of 2 and since Hydrogen thread can be processed as soon as it is created initial number of entries can be 0, so this will look like below

Semaphore semaphoreH = new Semaphore(2, 2);

Then we will create Oxygen semaphore which will have maximum concurrent thread count of 1 and since Oxygen thread always needs to be processed/released only after two Hydrogen threads initial concurrent requests should be 0 that is Oxygen thread needs to wait till any two Hydrogen threads are processed. With this Oxygen semaphore will look like below

Semaphore semaphoreO = new Semaphore(0, 1);

So, our class will look like below

public class Water

{

Semaphore semaphoreO = new Semaphore(0, 1);

Semaphore semaphoreH = new Semaphore(2, 2);

}

Now add two private methods to print Hydrogen and Oxygen like below

void ReleaseHydrogen()

{

Console.WriteLine("H");

}

void ReleaseOxygen()

{

Console.WriteLine("O");

}

Now we need two more methods to process Hydrogen and Oxygen threads and release each other accordingly, this is where we will use our semaphores -

* Hydrogen method will allow to enter two threads in critical section and will release Oxygen semaphore when two Hydrogen threads are processed, but won’t allow more than two Hydrogen threads to enter critical section until one Oxygen thread is processed
* Oxygen method will wait to Release Oxygen if two Hydrogen threads are processed and release Hydrogen semaphore twice or else will wait on Oxygen semaphore.

Both these methods will look like below

int hCount = 0;

public async Task HThread(Action releaseH)

{

await Task.Delay(1);

//Wait on Hydrogen thread, code after this will be blocked after processing two Hydrogen threads until one Oxygen thread is processed

semaphoreH.WaitOne();

releaseH();

hCount++;

if (hCount % 2 == 0) //For every two Hydrogen threads releasing Oxygen semaphore to process Oxygen method.

{

semaphoreO.Release();

}

}

public async Task OThread(Action releaseO)

{

await Task.Delay(1);

//Locking on Oxygen semaphore, this will allow to be processed only when 2 Hydrogen threads are processed or else will wait.

//Code after this is blocked until two Hydrogen threads are processed as initial concurrent threads for Oxygen semaphore is 0 (first parameter)

semaphoreO.WaitOne();

releaseO();

semaphoreH.Release(2); //Exiting Hydrogen semaphore twice, allowing two Hydrogen to be processed

}

Now let’s define method to build water in which we will loop through input sequence and initiate Hydrogen thread if input character is H and initiate Oxygen thread if input character is O

public async Task BuildWaterAsync(string input)

{

List<Task> tasks = new List<Task>();

foreach (char c in input)

{

switch (c)

{

case 'O':

tasks.Add(OThread(ReleaseOxygen));

break;

case 'H':

tasks.Add(HThread(ReleaseHydrogen));

break;

default:

break;

}

}

await Task.WhenAll(tasks);

}

Let’s consume this class in a simple console application and call BuildWaterAsync method, so create a console application add Water class to the app and replace main method with below code

static async Task Main(string[] args)

{

Water water = new Water();

while (true)

{

Console.WriteLine("Please enter sequence of Hydrogen and Oxygen molecules or e to Exit");

string input = Console.ReadLine();

if (input == "e")

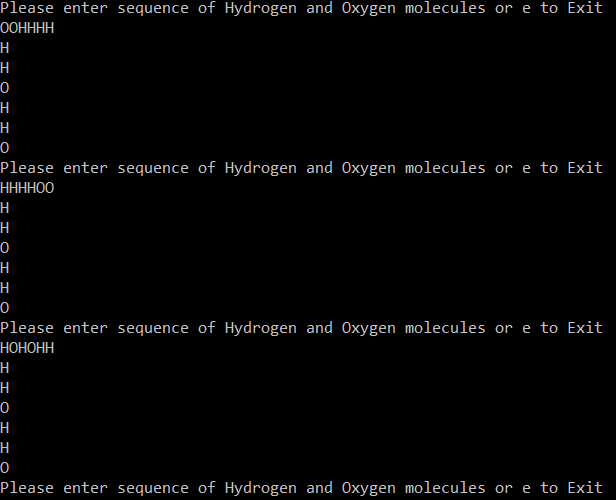
break;

await water.BuildWaterAsync(input);

}

}

Once we run this code output will look below



**Figure 7.7 – Output of build water application using semaphore**

In this sample we can clearly see how semaphore is helping to lock critical section with more than one threads and signaling on the availability of resources. Similarly, many classic synchronization problems like Dining Philosopher problem, producer-consumer problem, reader-writer problem can be solved using semaphores.

Note – The above example is built with assumption that user will always enter a combination that can be converted into one water molecule as intent is here is to understand semaphore. Further validation can be added to handle scenarios where there aren’t enough Hydrogen threads or Oxygen threads.

**Some important facts about semaphore**

* Semaphore doesn’t have thread affinity, so any thread call Release method, it’s application responsibility to release semaphore appropriately.
* Semaphore are usually used for signaling of resource availability, like a thread is available in thread pool.
* Semaphores can be named semaphores; these can be used to support across process synchronization.
* A typical scenario of Semaphore usage is a requirement where we want to limit concurrent database connections or in a multi core scenario to limit the number of concurrent threads executing a specific operation.

**SemaphoreSlim (Non - Exclusive)**

SemaphoreSlim is another class in System.Threading and is lightweight version to create Semaphores in C#. When we use System.Threading.Semaphore class to create a semaphore it internally uses Windows kernel semaphores which involves blocking , context switching of threads and also expensive kernel transition, however SemaphoreSlim implements spinning through spinwait and if it cannot acquire lock after spinning for a while(Microseconds) then it uses blocking to acquire lock. As discussed earlier that spinning for a very brief period of time is less expensive as compared to blocking hence SemaphoreSlim is good fit for such scenarios where the wait time to acquire critical section is less. Couple of other properties that SemaphoreSlim supports are

* To acquire a slot in SemaphoareSlim we need to call Wait method or WaitAsync method
* SemaphoreSlim doesn’t support named Semaphores so by default it’s always local Semaphore
* SemaphoreSlim has support for asyn methods like WaitAsync
* Since async methods are available SemaphoreSlim also allows cancellation token which means cancellation is allowed and at times can be useful to come out of deadlock.
* SemaphoreSlim has a constructor that supports initializing it with one parameter which is initial available slots and no upper limit. In such semaphores release method can be called any number of times and there won’t be any exception(SemaphoreFullException) thrown in such cases, it’s developer’s responsibility to call wait and release methods appropriately.
* SemaphoreSlim has CurrentCount property which tells the number of threads that can get a slot.
* Just like Semaphore, SemaphoreSlim is also thread agnostic.

Going back to our example if we use SemaphoreSlim we need to change initialization as below

SemaphoreSlim semaphoreH = new SemaphoreSlim(2, 2);

SemaphoreSlim semaphoreO = new SemaphoreSlim(0, 1);

Instead of WaitOne we will call WaitAsync and making use of CurrentCount our code will look like below

public async Task HThread(Action releaseH)

{

if (semaphoreH.CurrentCount == 0 && semaphoreO.CurrentCount == 1)

{

Console.WriteLine("Hydrogen is ready, waiting for Oxygen");

}

//Wait on Hydrogen thread, code after this will be blocked after processing two Hydrogen threads until one Oxygen thread is processed

await semaphoreH.WaitAsync();

releaseH();

hCount++;

if (hCount % 2 == 0) //For every two Hydrogen threads releasing Oxygen semaphore to process Oxygen method.

{

semaphoreO.Release();

}

}

public async Task OThread(Action releaseO)

{

if (semaphoreH.CurrentCount > 0 && semaphoreO.CurrentCount == 0)

{

Console.WriteLine("Oxygen is ready, waiting for Hydrogen");

}

//Locking on Oxygen semaphore, this will allow to be processed only when 2 Hydrogen threads are processed or else will wait.

//Code after this is blocked until two Hydrogen threads are processed as initial concurrent threads for Oxygen semaphore is 0 (first parameter)

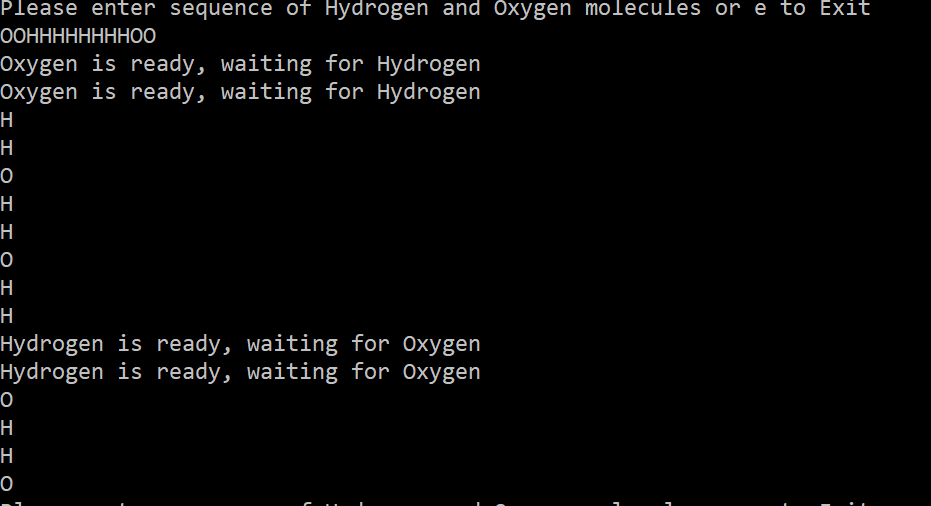
await semaphoreO.WaitAsync();

releaseO();

semaphoreH.Release(2); //Exiting Hydrogen semaphore twice, allowing two Hydrogen to be processed

}

Output for with SemaphoreSlim will have additional information as we now can tell when Hydrogen is waiting on Oxygen or vice-versa



**Figure 7.8 – Output of build water application using SemaphoreSlim**

As such there is no hard and fast rule on what to use when but mostly by rule of elimination where if we want cross process semaphore use Semaphore class, if we want lightweight semaphore for synchronization of a resource that is held for a very shorter period of time go for SemaphoreSlim.

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

Often, it’s a case where we use exclusive locks for a shared resource which is recommended practice and also guarantees of proper synchronization of data, however there could be scenarios where a resource just red multiple times with periodic updates. Using lock in such scenarios will ensure synchronization during concurrent access however it will slow down the application as two threads that just wanted to read data will be processed sequentially. Such cases can be better handled using ReaderWriter locks which allows a shared resource to be accessed by multiple threads that wanted to perform read operation and allows single thread for write operation.

Taking an analogy of a teacher writing on blackboard and students copying it

* While teacher is writing on the blackboard none of the students can see what is on the blackboard (Single write)
* Teacher won’t erase content on black board until last student finishes copying the content (Multiple reads and write thread in queue until last read thread has released the lock)

A ReaderWriter lock can be achieved in C# either by creating an object of System.Threading.ReaderWriterLock or System.Threading.ReaderWriterLockSlim class. ReaderWriterLockSlim is a thinner version of ReaderWriterLock which lesser memory footprint and better performing. Both these classes have methods -

* To acquire read lock which can be called by multiple threads – AcquireReaderLock /EnterReadLock
* To release read lock acquired by a thread (ReadWriteLocks have thread affinity so same thread that acquired lock needs to release the lock, this is applicable for both read and write locks) – ReleaseReaderLock/ExitReadLock
* To acquire write lock – AcquireWriterLock/EnterWriteLock
* To release write lock – ReleaseWriteLock/ExitWriteLock
* To acquire an upgradeable lock. An upgradable lock helps in acquiring a read lock and then upgrade to write lock based on a condition for example an upsert scenario. This can be normally achieved by acquiring a read lock, check if data is present, if not release read lock and then acquire a write lock. However, state of the shared resource may not remain same between releasing read lock and acquiring write lock hence it is preferable to use upgradeable lock.

Let’s see this with a simple example of writing data into file where we simulate around 30 threads with multiple reads and periodic update. Let’s create a console application and create a new public class FileWrite, create an instance of ReaderWriterLockSlim and also add a timer that will be used to get the execution time, our class will look like below

public class FileWrite

{

Stopwatch timer; //To compare performance with Monitor

public FileWrite()

{

timer = new Stopwatch();

timer.Start();

}

const string fileName = "SampleReadLock.txt";

ReaderWriterLockSlim readerWriterLockSlim = new ReaderWriterLockSlim();

}

Add two methods

private void ReadFile() – A method that reads data from file, we will lock the read operation using EnterReadLock and ExitReadLock method of ReaderWriteLock so as to avoid trying to read file when it is open for writing data (If we do not lock the read operation FileStream will throw System.IO.IOException -The process cannot access the file) . This method will look like below

private void ReadFile()

{

if (File.Exists(fileName))

{

readerWriterLockSlim.EnterReadLock();

using (FileStream fs = new FileStream(fileName, FileMode.Open, FileAccess.Read, FileShare.Read, 2048, useAsync: true))

{

using (System.IO.StreamReader rdr = new System.IO.StreamReader(fs))

{

Thread.Sleep(500); //Used to perform timer calculation,

Console.WriteLine(rdr.ReadToEnd());

}

}

readerWriterLockSlim.ExitReadLock();

}

}

private void WriteFile(int lineNumber) – A method that writes into file but before writing into file acquires a ReaderWriterLock using EnterWriteLock and ExitWriteLock. This method implementation will look below.

private void WriteFile(int lineNumber)

{

readerWriterLockSlim.EnterWriteLock();

string text = $"Line {lineNumber} ReadWriteLock" + Environment.NewLine;

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

using (FileStream fs = new FileStream(fileName, FileMode.Append, FileAccess.Write, FileShare.Write, 2048, useAsync: true))

{

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

}

readerWriterLockSlim.ExitWriteLock();

}

Now add method that will write into file if particular text is present else display so primarily an upsert operation. This method will make use of EnterUpgradeableReadLock which can be used to upgrade to write lock conditionally. This method implementation will look below

private void ReadorUpdateFile()

{

string fileContent = String.Empty;

if (File.Exists(fileName))

{

readerWriterLockSlim.EnterUpgradeableReadLock();

//First read the contents and if specific content exists then print on console else write into file

using (FileStream fs = new FileStream(fileName, FileMode.Open, FileAccess.Read, FileShare.Read, 2048, useAsync: true))

{

using (System.IO.StreamReader rdr = new System.IO.StreamReader(fs))

{

fileContent = rdr.ReadToEnd();

}

}

if (!(fileContent.Contains("Line 15")))

{

readerWriterLockSlim.EnterWriteLock();

using (FileStream fswrite = new FileStream(fileName, FileMode.Append, FileAccess.Write, FileShare.Write, 2048, useAsync: true))

{

byte[] encoding = Encoding.ASCII.GetBytes($"Line 15 ReadWriteLock" + Environment.NewLine);

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

}

readerWriterLockSlim.ExitWriteLock();

}

else

{

Thread.Sleep(500); //Used to perform timer calculation,

Console.WriteLine(fileContent);

}

readerWriterLockSlim.ExitUpgradeableReadLock();

}

}

Let’s a add another method to call these methods, let’s call it PerformFileOperation. The purpose of this method is to simulate around 30 parallel requests and primarily calling ReadFile and conditionally calls WriteFile, ReadorUpdateFile methods

public async Task PerformFileOperation()

{

var tasks = new Task[31];

for (int i = 0; i < tasks.Length; i++)

{

if (i % 10 == 0) //Calling write every tenth time

{

tasks[i] = Task.Run(() => WriteFile(i + 1));

Thread.Sleep(1000); //Used to perform timer calculation

}

else if (i == 15 || i == 21) //Calling upsert twice

{

tasks[i] = Task.Run(() => ReadorUpdateFile());

}

else //Calling read most of the time

{

tasks[i] = Task.Run(() => ReadFile());

}

}

await Task.WhenAll(tasks);

Console.WriteLine($"Time ellapsed {timer.ElapsedMilliseconds}"); //Displaying time taken for execution

readerWriterLockSlim.Dispose();

}

Let’s instantiate this class in our main method and call PerformFileOperation method to read from file and write into file. Main method will look below

static async Task Main(string[] args)

{

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

FileWrite fileupload = new FileWrite();

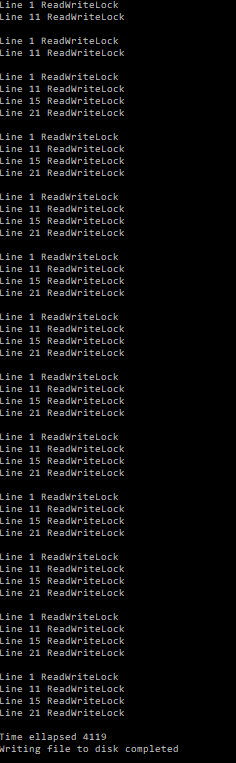
await fileupload.PerformFileOperation();

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

Console.Read();

}

Once we run this application



**Figure 7.9 – Output of file read application using ReaderWriteLockSlim**

Now let’s remove ReaderWriteLock and replace synchronization mechanism with Lock and run the application we noticed that output remains same however execution time is much higher (almost 3 times) and we know that is expected because Lock won’t allow multiple threads concurrently. So, ReaderWriteLock is an effective synchronization mechanism for a shared resource when there are many reads but periodic updates.

**Some important facts about reader/write locks**

* ReaderWriteLocks allows multiple read, one exclusive write lock.
* Among multiple read one can be upgraded to write lock
* ReaderWriteLock has thread affinity

**Signaling constructs**

**AutoResetEvent 3**

**ManualResetEvent/ ManualResetEventSlim 4**

**CountdownEvent 3**

**Barrier classes 3**

May be multiple API calls

**Wait and Pulse 3**

**Nonblocking synchronization constructs (10 pages)**

**Thread.MemoryBarrier**

**Thread.VolatileRead**

**Thread.VolatileWrite**

**The volatile keyword**

**The Interlocked class**

**SpinWait**