Kenny R N Wilson

# Threading

*Multiple concurrently executing threads*

---

**THIS DOCUMENT COVERS**

# Threading Basics

At the OS level a process consists of a process kernel object and an address space. The process itself does not execute anything. Instead each process consists of one or more threads each of which represent a single sequential flow of control within the processes single address space. At the OS level a thread is made up of a thread kernel object and a call stack. Because each thread has its own call stack it also has its own local variables. All threads in the same app domain read and write to the same heap address space. It is the programmer's responsibility to ensure that shared memory is accessed and written correctly.

A process is a heavier weight concept than a thread because a process requires a virtual address space and a set of DLLs which take up file and memory resources in the operating system

### Thread Definition

- Single sequential flow of control within a single address space
- Each thread has a separate call stack and its own local variables
- At OS level a thread has a thread kernel object and a stack
- Multiple threads read/write to the same heap
- Threads share heap memory with other threads (running in same app domain)
- Programmers must ensure that shared memory is accessed correctly

We utilise threads to improve performance when carrying out I/O, to take advantage of multiple cores and to maintain a responsive user interface when performing long running operations.

### Advantages of threads

- Better performance than multiple processes communicating across address spaces
- Improve performance when driving slow devices
- Maintain responsive user interface
- Take advantage of multiple cores
- Simplify UI – spell checking in background prevents need for "check spelling" option
- One process in loop can be killed by another process

Overuse of multi-threaded programming can, however, reduce performance. Context switching between threads is a relatively expensive operation and large amounts of threads are known to reduce throughout. Multi-threaded programming increases code complexity and can introduce non-deterministic behaviour. Bugs are harder to find as often there are few clues at the point of failure as to the root cause of the problem.

### Disadvantages of multi-threaded programming

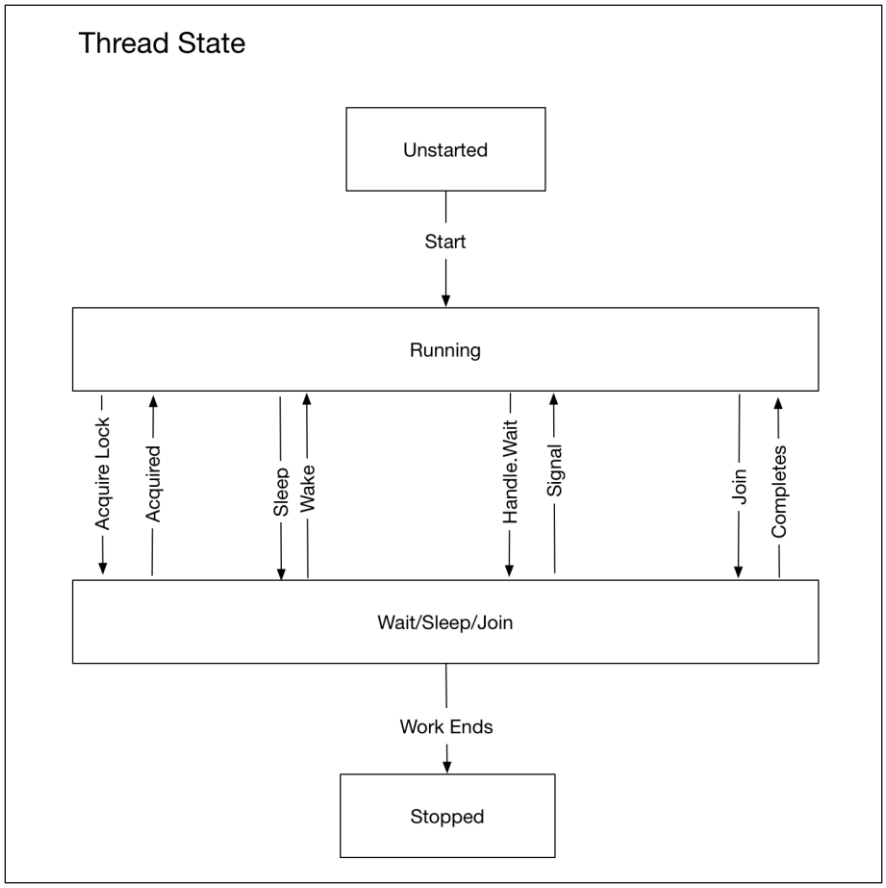
- Context switching between threads is expensive
- Too many threads can reduce throughput and hence performance.
- Non-deterministic behaviour
- Rarely enough information at the point of failure to debug the application
- Only guarantee is that bugs will never show until in production

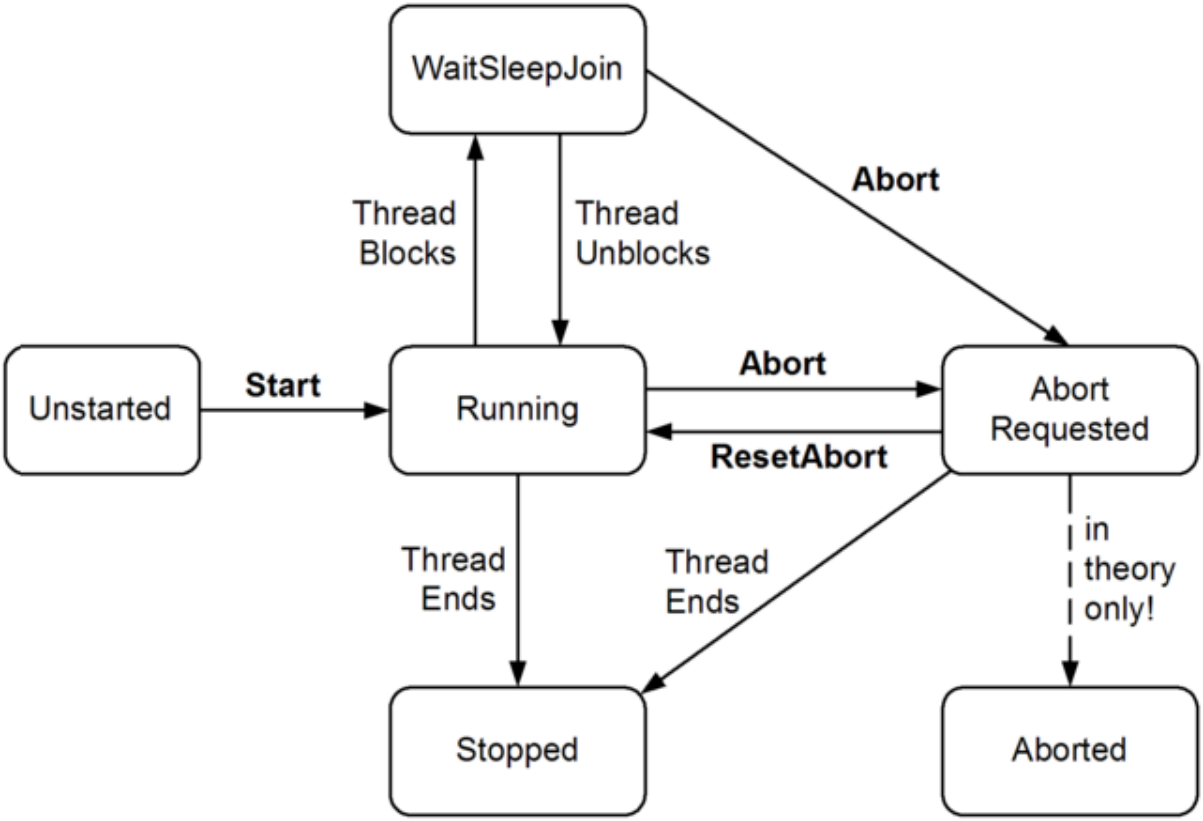
## Thread State

A thread state can essentially be in one of four states

- Unstarted
- Running
- Wait/Sleep/Join - blocked
- Stopped

Blocked threads consume almost no processor time




WaitSleepJoin
Abort
Thread
Blocks
Thread
Unblocks
Unstarted
Start
Running
Abort
ResetAbort
Abort
Requested
Thread
Ends
Thread
Ends
in
theory
only!
Stopped
Aborted

# Questions - Threading basics

**What is a thread?**

*Single sequential flow of control within a single address space*

*Each thread has separate call stack with own local variables*

*Multiple threads read/write same memory locations*

*Programmer ensures shared memory accessed in correct way*

**Why would we use multi-threaded programming?**

1. *Maintain responsive user interface*
2. *Scalability – taking advantage of multiple CPUs*
3. *Performing RPC to remote server*
4. *Driving slow devices such as disks, terminals, printers*

**Why are multiple threads preferable to multiple processes?**

*Multiple separate processes in separate address spaces are expensive to set up*

*Cost of inter-process communication is high*

**What threading facilities does .NET provide?**

*Thread creation*

*Join*

*Mutual exclusion*

**What are the main problems with multi-threaded programming?**

1. *Increased complexity*
2. *Non-deterministic behaviour*
3. *Cost in allocating and switching threads*
4. *Non-reproducible bugs*
5. *Only guarantee is that bugs will never show until in production*
6. *Rarely enough information at the point of failure to debug the application*

**Compare and contrast a thread and a process?**

*Processes run in parallel on a computer and threads run in parallel in a process*

*Processes are fully isolated from each other and threads only have limited isolation*

*Threads share heap memory with other threads running in the same app domain*

**What is meant by the term preemption?**

*A threads execution is interspersed with execution of code on another thread*

**What happens when a C# client application starts?**

*A single main thread in automatically created by the CLR and OS*

**What is the primary cause of complexity and obscure errors in multithreading?**

*Shared data*

**When can multiple threads improve performance?**

*Only if the application is I/O bound OR*

*Computer has multiple CPU's*

**Why limit the number of threads in a process?**

*Creating a thread is not cheap*

*Destroying a thread is not cheap*

*Context switching is expensive*

**Why is creating a thread expensive?**

*1MB of address space for the user-mode stack*

*12K of address space for the kernel-mode stack*

*Ever .DLL needs notification of thread creation*

**Why is destroying a thread expensive?**

*Every .DLL needs notification*

*Kernel object and stack need to be freed*

**Why is context switching expensive?**

*Enter kernel mode*

*Save CPU registers into current threads kernel object*

*Determine next thread to schedule*

*Load CPU registers from about to run threads kernel object*

*Leave kernel mode*

# Creating Threads and Scheduling Work

## New Threads

If we want to start a new thread to execute a piece of code, we create an instance of type Thread and pass in a delegate containing the instructions to be executed by the thread.

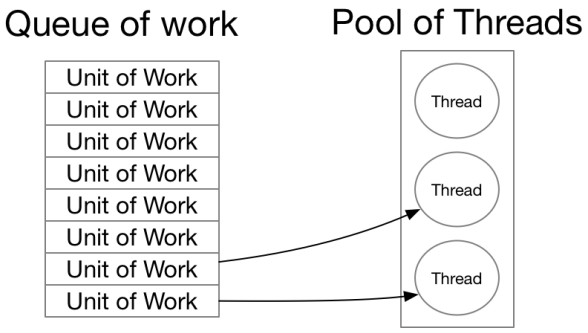
```
void Main()
{
    Thread t1 = new Thread(this.ThreadBodyA);
    Thread t2 = new Thread(this.ThreadBodyB);

    t1.Start();
    t2.Start("Test Message");
}

public void ThreadBodyA() { }
public void ThreadBodyB(object message) { WriteLine(message); }
```

Creating a thread creates a thread kernel object and allocates space from the process address space for the stack. Each thread can access the process kernel objects and any part of the process address space. This last point is key for enabling thread communication.

The thread's kernel object has a context containing the copy of a set of CPU registers including the stack pointer and instruction pointer. In addition it contains flags that indicate if the thread is active and whether or not it has been signalled.

## Thread Pooling

One problem with creating a new Thread every time we want to execute code is that thread creation and destruction are expensive operations that introduce a lot of overhead. A thread pool combines a queue with a number of worker threads. The worker threads take work items off the queue and process them. Each .NET application has only one thread pool.




This pooling and recycling of threads is more efficient than continually creating and destroying threads explicitly. An intelligent thread pool will typically reduce the number of threads when the CPU is 100% and increase the number of threads when all threads on the pool are busy and the CPU is less than 50%. We can schedule work to be executed on a thread pool as follows.

## New thread versus thread pool

Some places where we might use a dedicated background thread rather than the thread pool

- Code Editor compiling the code in the background to provide syntax highlighting
- Spreadsheet background calculations

## Scheduling

Windows is a pre-emptive multithreading OS which means it needs to determine when to schedule different threads. After every time slice the OS saves the values in the CPU registers into the currently executing thread's context object. It then decides which thread from the available schedulable threads to switch to. The context switch involves copying the context from the thread kernel object into the CPU registers.

### Priority Classes

Each application has a priority class chosen from the below

- Idle
- Below Normal
- Normal
- Above Normal
- High
- Real Time

Typically, one should avoid the High priority as much as possible and hardly ever use the Real Time category. The real time category is higher than most OS threads and can hence starve essential disk and network traffic. Task manager runs at High so it can kill badly behaved processes.

### Thread Priorities

- Idle
- Lowest
- Below normal
- Normal
- Above normal
- Highest
- Time critical

The OS maps the combination of a process's priority class and a threads priority to a priority level between 0 and 31 with 31 being highest. When deciding which schedulable thread to run the OS first considers threads of highest priority. Only if no thread of the highest priority level is schedulable is the second highest priority thread considered and so on. This can lead to starvation of lower priority threads.

It is good practice for high priority threads to do small amount of work infrequently and low priority tasks can remain schedulable a lot and do a lot of work when they have the CPU as they are only executing when nothing higher needs the cycle.

The nature of a pre-emptive operating system means one cannot know for sure a particular thread will running at any given time or that it will not be pre-empted at any moment in time.

## Sleep

Threads support the Sleep function. Sleep essentially performs two functions

1. Threads gives up its current time slice
2. Tells the OS not to make the thread schedulable for a specified number of milliseconds

If we pass 0 milliseconds this means the thread relinquishes its current time slice but is immediately available for rescheduling.

**Listing 1Scheduling work on the ThreadPool**

```
ThreadPool.QueueUserWorkItem(Console.WriteLine, "Hello World");
```

As of .NET 4.0, a more common and powerful way to schedule work on the thread pool is via Tasks. We will cover tasks in more detail later but for now the following code executes a simple piece of work on a background thread.

**Listing 2Using tasks with the thread pool**

```
Task.Run(() => Console.WriteLine("Hello World"));
```

**Uses of the thread pool**

- Tasks on default task scheduler
- User code
- ASP.NET and Web Services
- WCF
- System.Timer and System.Threading.Timer
- Asynchronous delegates
- BackgroundWorker class

## Timers

ToDo:

# Questions – Creating Threads and Scheduling Work

**What are the disadvantages of creating and starting threads for every asynchronous task?**

*Creating and destroying threads is expensive*

**Write code to create and start a new thread**

```
void Main()
{
    Thread t1 = new Thread(this.ThreadBodyA);
    Thread t2 = new Thread(this.ThreadBodyB);

    t1.Start();
    t2.Start("Test Message");
}

public void ThreadBodyA() { }
public void ThreadBodyB(object message) { WriteLine(message); }
```

**What is the difference between foreground and background threads?**

*Foreground  threads keep the application alive so long as one of them is running*

*Background threads abruptly terminate as soon as all foreground threads end*

**What is the only situation where multiple threads share local variables?**

*Local variables captured in an anonymous method passed as the delegate to a thread constructor.*

**What is returned by a threads isAlive property?**

*Once started returns true until the threads ends*

**When does a thread end?**

*When the delegate passed to the threads constructor finishes executing*

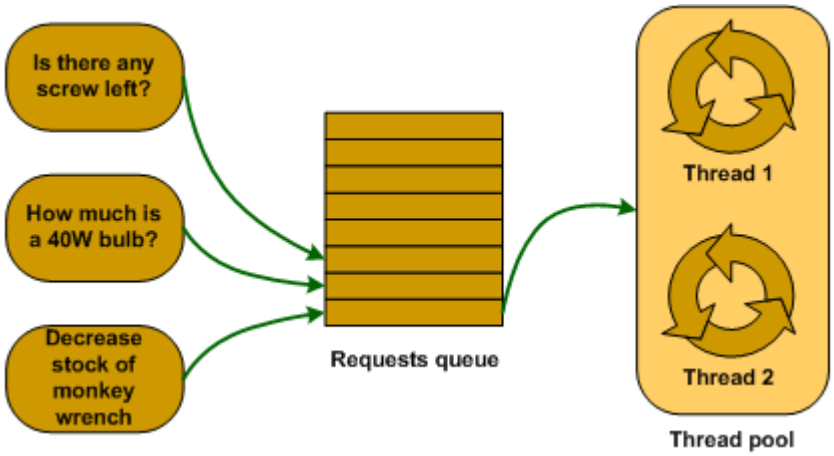
**Can you restart a thread once it ends?**

- No

**What is a thread pool?**

*Combines queue with number of worker threads*

*Worker threads process requests off the queue*




**What are the benefits of a thread pool?**

*Reduces overhead of creating and destroying threads*

**What are the features of an intelligent thread pool?**

*Worker threads process requests off the queue*

*Remove threads when CPU is 100% to prevent thrashing*

*Add threads where all are busy and CPU < 50% as most likely existing work items are I/O bound*

**What are the features of the .NET thread pool?**

*One per process*

*All methods are static*

**Write code to execute work on the thread pool?**

```
        ThreadPool.QueueUserWorkItem(new WaitCallback(this.DoWork));
    }

    public void DoWork(object state) {}
```

**What is the signature of WaitCallback?**

```
public delegate WaitCallback( object state ) ;
```

## TIMERS

**What are the available types of Timer**

*System.Threading.Timer*

*System.Timers.Timer*

*System.Windows.Forms.Timer (Windows Form)*

*System.Windows.Threading.DispatcherTimer (WPF)*

**What are the main differences between the first two and last two?**

*The first two are general purpose multi-threaded timers*

*The second two are special purpose single threaded timers used when updating Windows forms controls or WPF elements*

**Characteristics of System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer?**

*Used when updating Windows form controls or WPF Elements*

*Doesn't use thread pool*

*Always fires on same thread that creates it*

**Advantages of System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer?**

*Thread-safe*

*New tick will never fire until previous one runs*

*Can update UI controls directly from Tick event handler*

**Disadvantages of System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer?**

*Can impact UI responsiveness if tick handler isn't lightweight*

*Less accurate because can be delayed while other UI requests are processed*

**When to use System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer**

*Small job that involves updating some aspect of user interface*

*E.g. clock or countdown display*

**Characteristics of System.Threading.Timer**

*Simplest multi-threaded timer*

*Uses the thread pool*

*Callback may fire on any different threads*

*Callbacks or event handlers must be thread safe*

**What are the characteristics of System.Timers.Timer?**

*Wrapper around System.Threading.Timer*

*Adds convenience methods and uses the underlying implementation*

**Who use .NET thread pool?**

*Tasks using default task scheduler*

*User code*

*WCF,*

*ASP.NET and Web Services*

*System.Timer and System.Threading.Timer*

*Asynchronous delegates*

*BackgroundWorker helper class*

**Does each app domain have its own thread pool?**

*No there is only one thread pool per process*

**What two types of thread does the .NET thread pool use?**

*Worker threads – execute user functions and timers*

*Completion Port threads – Used for IO operations where possible*

**What are worker threads?**

*Managed by .NET framework*

*Execute user functions and timers*

**Write code to queue a method for asynchronous execution on a thread pool?**

**What advantage do asynchronous delegates have over thread pools queue work items?**

Typed method arguments

# Synchronization, Communication and Thread Safety

In an ideal world multiple threads inside a process would not communicate, applications would run faster and there would be less incorrect code. This is not, however, realistic. Threads need to synchronize their execution.

## Thread Safety

A thread safe piece of code is one that behaves deterministically in the presence of multiple threads of execution. Windows is a pre-emptive multithreaded operating system and as such it gives no guarantees about when our threads will run and when they will be pre-empted. In general, most types are not designed to be thread safe out of the box for the following reasons

- Increases complexity of development
- More inefficient in single threaded scenarios
- Making a type thread safe does not make the code calling it thread safe

Any operation which executes as a single atomic instruction cannot be pre-empted. The CLR guarantees that reads and writes to object references and primitive types of size 32 bit or less will be atomic. Combinations of reads and write such as decrements and more complex sequences are never guaranteed to be atomic.

In addition to non-atomic sequences the compiler and the CPU can carry out statement re-ordering and read/write caching optimisations that can cause code that is correct in a single threaded environment to have bugs in a multi-threaded environment. In order to ensure correctness a programmer should ensure all memory falls into one of the following three buckets

- Thread exclusive
- Read only
- Lock Protected

The .NET framework provides the **ContexBoundObject** and Synchronization attribute to support automatic locking schemes. These are rarely a good idea as they provide overly course grained concurrency that can introduce unintended deadlocks. The only .NET framework types that have built in thread safety are those in the **System.Collections.Concurrent** namespace. Although most framework types do not have thread safety out of the box they do follow the convention that static methods are thread safe and instance methods are not.

In the following section on synchronization and communication we show many constructs that can be used to create code that behaves correctly in a multi-threaded environment.

## Synchronization Constructs

In the previous section we mentioned several causes of incorrect multi-threaded behaviour. The .NET framework provides several different thread communication and synchronization primitives. We can categorize the primitives based on whether they execute as user mode operation, kernel mode operations or a hybrid mixture of both.

User mode constructs execute using special hardware instructions meaning they are very fast. They block for a tiny amount of time and indeed the OS never even knows the thread is blocked. This means they never cause a thread pool to create new threads to compensate for blocked threads. The downside of this is that as the OS does not know the thread is blocked it can continually reschedule the same thread causing spinning which wastes CPU cycles.

Kernel mode constructs are expensive as they require moving from user mode to kernel mode and back which is expensive. The only upside is that when kernel mode constructs block, they can prevent the CPU from wasting cycles.

Synchronization is only required when there is a chance that multiple threads will access the data at the same time.  If possible, it is better to structure one's code to prevent the possibility of multiple threads accessing shared data than fixing the accesses using synchronization constructs. The use of synchronization constructs is tedious, error prone and incurs a significant performance overhead.

The .NET FCL provides several types and constructs that facilitate thread communication and synchronization. Typical uses of these constructs are

- Restrict the number of threads that can access a section of code concurrently
- Ensure correctness which multiple threads access shared memory
- Turn off compiler and CPU reordering and read/write caching optimisations
- Enable threads to notify each other.

### Shared Memory / Critical Sections

One way for threads to communicate is via shared memory. In .NET shared memory is likely to be objects allocated on the heap. For example, several worker threads might take units of work off a queue, execute them and update some shared memory. Another dedicated thread might read the results and perform calculations on them. In a pre-emptive multithreading environment, we have no guarantees around when our thread will run and when it will be pre-empted. This can lead to nondeterminism where the result depends on the order in which threads are pre-empted and scheduled. Such problems are known as race conditions. A race condition occurs when four conditions are met.

- Memory locations are accessible from more than one thread
- Invariant is associated with shared memory
- Invariant does not hold for some part of update
- Another thread accesses the memory the while invariant is broken

Typically, to uphold the invariant an operation or sequence of operations must be performed atomically. The CLR guarantees that reads and writes to object references and primitive types of size 32bits or less will be atomic. Single reads or writes to fields larger than 32 bits such as longs or doubles are not guaranteed to be atomic and hence can be susceptible to **torn reads**. Combinations of reads and writes are never guaranteed to be atomic. In order to prevent race conditions, we need to use thread synchronization. Synchronization constructs are used to restrict the number of threads that can concurrently access critical sections of code and by extension restrict the number of threads that can concurrently access the shared memory read/written inside those critical sections.

### Signalling

Often one thread needs to inform another thread or threads that is has completed some task. Signalling constructs perform this kind of communication. One or more threads can block waiting for one or more threads to let them know that a piece of work has been completed.

## User Mode Constructs

When a thread pool thread blocks the thread pool will create new threads to compensate. Creating, destroying and context switching between threads is expensive and wastes resources. One way to keep our applications responsive it to prevent the worker threads we have from blocking. When they are not blocked, they are free to process more tasks and keep the application responsive and scalable.

### Interlocked

The interlocked keyword is a powerful non-blocking user mode construct which provides the following benefits.

The interlocked keyword provides the following benefits.

1. Extends atomic read/write semantics to 64-bit longs and doubles
2. Provides atomic read and write combinations for 32-bit and 64-bit fields and references

We can use the interlocked keyword for several scenarios. One scenario is to implement our own SpinLock. Notice how this works. We have super-fast access to the lock when uncontended. But if the lock is contended, we spin. We can reduce the impact of this spinning by yielding our current time slice in the hope the thread with the resource will finish its work and release it.

**Spin Lock**

```
public class MySpinLock
{
    private int taken;

    public void Enter()
    {
          while (true)
          {
                if (Interlocked.Exchange(ref taken, 1) == 0) return;

                // Yield the current thread in an attempt to lessen the
                // impact of spinning and allow the thread that holds the
                // lock to finish what it is doing
                Thread.Yield();
          }
    }

    public void Leave() => Volatile.Write(ref taken,0);
}
```

The interlocked keyword only supports some simple operations such as add (and subtract via negative arguments), increment and decrement. We can do other operations using the following pattern

### Other Interlocked Via Pattern

```
public double Multiply(ref double field, double y)
{
    while (true)
    {
          // The value in the field before we do the
          // exchange
          double fieldVal1 = field;

          // AttemptedProduct
          double product = fieldVal1 * y;

          // We only update field if the value in the field is
          // equal to snapShot1. In this case the value of
          // product was calculated from the current value of
          // field
          double fieldVal2 =
                CompareExchange(ref field, product, fieldVal1);

          // We can now compare the value in snapshot2 to see if
          // we actually did the update. If we did the update return it
          if (fieldVal1 == fieldVal2) return product;

          // Otherwise we need to spin around again
          Thread.Yield();
    }
}
```

### Kernel Mode Constructs

Kernel model constructs tend to be slower than user mode constructs as they require passing from managed code to unmanaged user code to unmanaged kernel code and back again. They do however come with several advantages

- A thread trying to obtain a contended resource will block preventing spinning
- Kernel mode constructs can synchronize managed and unmanaged code
- Kernel mode constructs can synchronize across processes

All kernel mode constructs are based on two types; the **Event** and the **Semaphore**. The event is a kernel object based around a Boolean value which indicates whether or not the event is in a signalled state and the Semaphore is a wrapper around an integer count that indicates how many more threads can be allowed into a section of code. The .NET types that are based on the kernel synchronization constructs are all subclasses of the **WaitHandle** base class which wraps a native kernel object handle. The WaitHandle base class defines a common instance method **WaitOne** which enables threads to block on a WaitHandle and sub-classes define different signalling protocols.

- WaitHandle
  - Event
    - AutoResetEvent
    - ManualResetEvent
- Semaphore
- Mutex

### Semaphore

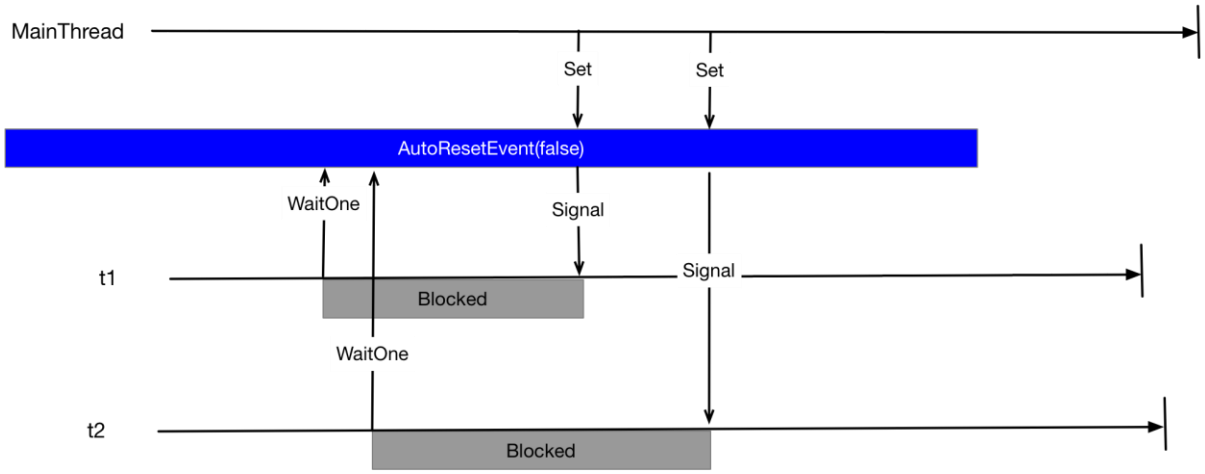
A Semaphore is essentially a set of methods that act upon a kernel level integer known as the count. Threads calling WaitOne block if the count is zero and unblock if the count is above zero. Any time a thread waiting on a semaphore unblocks the count is decremented.

A semaphore is a non-exclusive locking construct. It allows a specified number of threads to concurrently enter a critical section. Once the specified number of threads are inside the critical section other threads will block on **WaitOne** until **Release** is called on the Semaphore. One possible use would be to limit concurrent access to a resource such as a disk.

```
Semaphore e = new Semaphore(0,3);

for (int i = 0; i < 10; i++)
{
    int j = i;
    new Thread(() =>
    {
          e.WaitOne();
          Thread.Sleep(1000);
          e.Release();
    }).Start();
}

e.Release(3);
```

### AutoResetEvent

AutoResetEvent supports turnstile behaviour. When in a nonsignaled state any threads that call WaitOne on it will block. If multiple threads call WaitOne on a non-signalled AutoResetEvent a queue will form. Calling Set releases at most one waiting thread and Resets the state back to non-signalled keeping any other waiting threads blocked.




```
AutoResetEvent e = new AutoResetEvent(initialState: false);

Action work = () =>
{
    // Simulate some work this thread is doing
    Thread.Sleep(1000);
    logger.Info($"Calling WaitOne");

    // Now wait for notification from another thread
    // that we can complete
    e.WaitOne();

    // Simulate some more work
    logger.Info("Running");
    Thread.Sleep(5000);
};

new Thread(()=> work()).Start();
new Thread(()=> work()).Start();

// Main thread does some work
Thread.Sleep(5000);
logger.Info("Calling Set to Signal ARE");

// Main thread notifies one background thread
e.Set();

// Main thread does some work
Thread.Sleep(5000);
Thread.Sleep(2000);

// Main thread notifies one background thread
logger.Info("Calling Set to Signal ARE");
e.Set();
```

Calling WaitOne on a signalled AutoResetEvent calls it to become non-signalled preventing other threads from entering it. Knowing this we can use an AutoResetEvent to implement a simple blocking construct as follows.

```
public class BlockingLock
{
    private AutoResetEvent _event = new AutoResetEvent(true);

    public void Enter() => _event.WaitOne();

    public void Leave() => _event.Set();
}
```

### ManualResetEvent

This object behaves like a latch. If multiple threads are queuing having called WaitOne then all will be allowed into the critical section Set is invoked. We can think of Set as opening the latch and Reset as closing the latch.

```
ManualResetEvent e = new ManualResetEvent(false);

new Thread(() =>
{
    e.WaitOne();
    Console.WriteLine("t1 in");
}).Start();

new Thread(() =>
{
    e.WaitOne();
    Console.WriteLine("t2 in");
}).Start();

Thread.Sleep(1000);
e.Set();
Console.WriteLine("Signalled");

// > Signalled
// > t2 in
// > t1 in
```

### Mutex

A Mutex is a kernel mode exclusive locking construct. In some ways it is similar to an AutoResetEvent or a Semaphore (1) in that all three constructs release one thread at a time. The Mutex is however a more complex type as it has thread affinity and supports re-entry.

The downside of including support for re-entry and thread affinity is all users of the type incur the size and performance overhead whether or not they need this functionality.

A thread calls **WaitOne** to obtain the Mutex and **ReleaseMutex** to release it.

```
Mutex e = new Mutex();

  new Thread(() =>
  {
      e.WaitOne(); // Block on obtaining Mutex

      // Critical Section

      e.ReleaseMutex(); // Release Mutex
  }).Start();
```

## Managed Hybrid Constructs

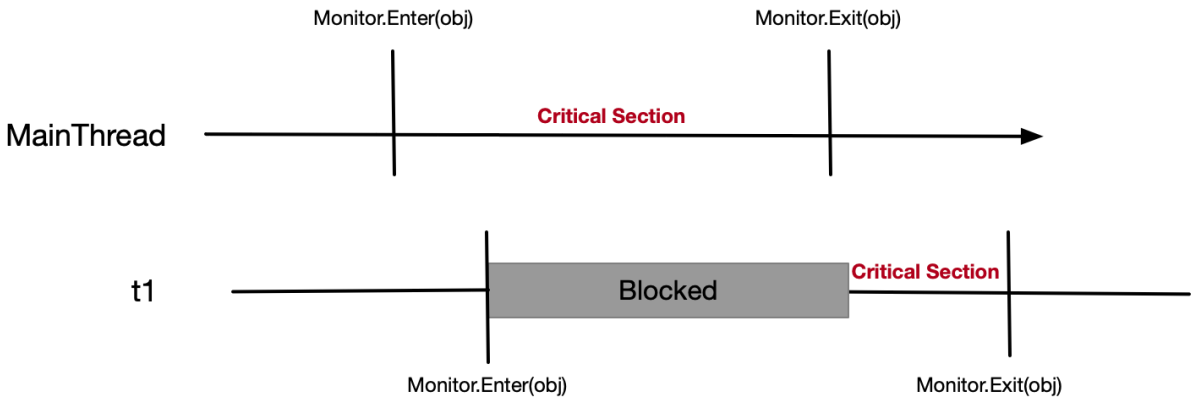
In the previous section we looked at user mode constructs and kernel mode constructs. In practice most of the .NET locking constructs are implemented in managed code and use a mixture of user mode and kernel mode constructs. Often the idea is to use the speed and efficiency of user mode constructs when there is no contention and to fall back on kernel mode constructs when there is contention to prevent excessing spinning from wasting CPU cycles.

### Monitor

Monitor is a managed mutual exclusion synchronization primitive that provides has thread affinity and re-entry support.

```
void Main()
{
    MyExtensions.SetupLog4Net();
    var l = new Object();

    new Thread( () =>
    {
          Thread.Sleep(500);
          logger.Info("Acquiring lock");
          lock (l)
          {
                logger.Info("Executing Critical Section");
                Thread.Sleep(1000);
                logger.Info("Leaving Critical Section");
          }
    }).Start();

    logger.Info("Acquiring lock");
    lock (l)
    {
          logger.Info("Executing Critical Section");
          Thread.Sleep(5000);
          logger.Info("Leaving Critical Section");
    }
}
```

Then what is happening is as follows



When the compiler sees code like this

```
lock(l)
{
    WriteLine("Inside the critical section");
}
```

It translates it to code like this

```
bool gotLock = false;
try
{
    Monitor.TryEnter(l, ref gotLock);
    WriteLine("Inside the critical section");
}
finally
{
    if (gotLock) Monitor.Exit(l);
}
```

There are however problems with Monitor. Strings should not be used as locks. Because of interning separate pieces of code could inadvertently lock on the same string causing deadlock. Passing a value type to Monitor.Enter would lead to boxing meaning each call to enter and exit would have a separate heap object and hence there would be no mutual exclusion at all.

### Summary of Monitor

- Faster and more convenient than Mutex
- Only reference objects can be used as locks
- Acquiring an uncontended lock is faster acquiring contended locks
- Most important methods are Monitor.Enter and Monitor.Exit
- TryEnter take a timespan and returns true if lock is obtained, false if it times out

### SemaphoreSlim

SemaphoreSlim is a lightweight managed synchronisation primitive that restricts the number of threads that can concurrently access a critical section of code. Unlike Semaphore it does not use Windows kernel semaphore. As such it only supports local Semaphores and not named system Semaphores.

SemaphoreSlim is constructed with two numbers; a count and a maximum count. When a thread enters the SemaphoreSlim the count is decremented and when a thread releases the SemaphoreSlim the count is incremented. Once the count is zero any subsequent calls to Wait will block until another thread calls Release. A thread can release the SemaphoreSlim multiple times by calling Release multiple times or by calling Release(int n)

```
var s = new System.Threading.SemaphoreSlim(0, 2);

for (int i = 0; i < 4; i++)
{
    int j = i;
    new Thread(() =>
   {
       WriteLine($"{j} Acquiring Sempahore");
       s.Wait();
       WriteLine($"{j} In Critical Section");
       Thread.Sleep(3000);
       WriteLine($"{j} Releasing SemaphoreSlim");
       s.Release();
   }).Start();
}

WriteLine("Main releasing semaphore twice");
s.Release(2);
```

### ManualResetEventSlim

The ManualResetEventSlim was introduced in .NET Framework 4. It does not subclass WaitHandle and instead has an efficient managed implementation that supports cancellation. The downside is it cannot be used for IPC like ManualResetEvent. Although it does not subclass WaitHandle it can provide a WaitHandle via a property of the same name.

### ReaderWriterLockSlim

The .NET framework defines two Reader/Writer lock types. ReaderWriterLock and ReaderWriterLockSlim. The Slim version should always be used as it is newer, faster and avoids many causes of deadlock. ReaderWriterLock allows concurrent reads and exclusive writes. Its most important methods are

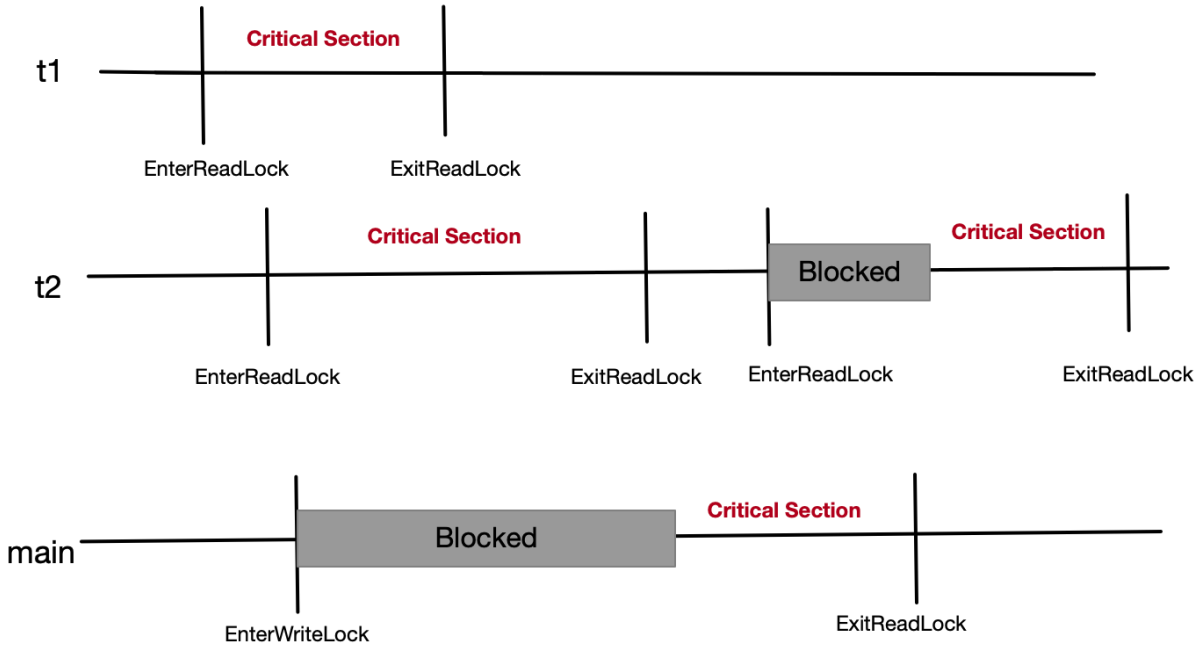
## Main Methods

- EnterReadLock
- ExitReadLock
- EnterWriteLock
- ExitwriteLock

## Key Features

- Thread holding write lock blocks all other threads trying to read or write
- If no thread contains a write lock any number of threads can obtain read lock
- A write lock is universally exclusive
- A read lock is compatible with other read locks

The following shows multiple reads blocking until a write finishes



### CountdownEvent

Enables one to wait on multiple threads. It is constructed with an integer value. Each time a thread calls Signal the count is decremented and when the count goes to zero. Anything waiting on Wait is allowed through when the count goes to zero.

PLINQ and Parallel are often better at solving the type of problem where we would use CountdownEvent

```
CountdownEvent e = new CountdownEvent(2);

new Thread(() =>
{
    e.Wait();
    Console.WriteLine("t1 in");
}).Start();

new Thread(() =>
{
    e.Wait();
    Console.WriteLine("t2 in");
}).Start();

Thread.Sleep(1000);
e.Signal();
Console.WriteLine("Signalled");
e.Signal();
Console.WriteLine("Signalled");

// > Signalled
// > Signalled
// > t1 in
// >t2 in
```

### Barrier

The barrier class enables multiple threads to work concurrently on a multi-phase algorithm. The same barrier can be used for multiple phases. One use of this type is a GC algorithm running in server mode. One thread per core start up with each thread working on a different thread stack as part of the mark. As each thread completes it must wait for all threads to complete mark. Then each thread compacts a different part of the heap again wating for the others to complete.

### Spinlock

Specialized advanced exclusive locking construct that can improve performance in highly contended environments.

## Lazy Initialization

## Thread Local Storage

### WAIT AND PULSE

#### The Condition Variable Pattern

One thread wants to execute code when some other condition is true.

### LIST OF SYNCHRONIZATION CONSTRUCTS

| | |
|---|---|
| Monitor/Lock | Mixed mode exclusive locking with thread affinity and rentry |
| SpinLock | Uses a loop implementation of Wait |
| SemaphoreSlim | Lighweight managed non exclusive primitive that limits the number of threads that can concurrently access a critical section |
| ReaderWriterLockSlim | Lighweight managed that supports concurrent reads and exclusive writes. |
| ManualResetEventSlim | A lightweight managed latch primitive |
| CountdownEvent | Enable one thread to wait on multiple threads. Only when signalled a specified number of times do threads blocking on Wait get through. |
| Barrier | Enables multiple tasks to cooperatively work on an algorithm in parallel through multiple phases |
| Mutex | **`WaitHandle`** based exclusive locking |
| ManualResetEvent | WaitHandle based latch |
| AutoResetEvent | WaitHandle based turnstyle. |
| Semaphore | Waithandle based primitive that allows a number of threads into critical section a |

## Fences and Barriers

Compiler and CPU statement re-ordering and caching optimisations can also lead to code that is correct in a single threaded environment behaving incorrectly in a pre-emptive multithreading environment.

#### Fences/Barriers

Fences and barriers are used to prevent statement re-ordering and read/write caching optimisations by the compiler and CPU. The following code is not thread safe even though the two fields are read and written atomically.

```
public int value;
public bool isRead = false;

public void Write()
{
    value = 100;  ❶
    isRead = true;❷
}

public void Read()
{
    if (isRead)                      ❸
          Console.WriteLine(value);  ❹
```

```
}
```

The reason for this is that the Compiler and CPU can re-order instructions if the meaning of the code does not change from a single threaded perspective. The order of ❶ and ❷ could be switched. Similarly, with ❸ or ❹ .

A full fence is achieved by using **Thread.MemoryBarrier**. The effect is that read/write caching is prevented and the CPU cannot reorder instructions such that memory accesses before the call to MemoryBarrier occur after the call to MemoryBarrier

```
public void Write()
{
    value = 100;
    Thread.MemoryBarrier();
    isRead = true;
}

public void Read()
{
    if (isRead)
    {
          Thread.MemoryBarrier();
          Console.WriteLine(value);
    }
}
```

### Volatile.Read/Write

The static method **Volatile.Read** utilises a memory barrier to ensure

1. No reads/writes after Volatile.Read execute before Volatile.Read
2. The value read is the latest written by any processor, irrespective of caching

The static method **Volatile.Write** utilises a memory barrier to ensure

1. No reads/writes before Volatile.Write execute after Volatile.Write
2. The value written is immediately available to all CPU's

### volatile keyword

Marking a field with the volatile keyword ensures all reads are performed by Volatile.Read and all writes are performed by Volatile.Write. This can be a bad idea for several reasons

1. It is unlikely all reads and writes to a field need to use Volatile.Read/Write and hence marking a field volatile incur needless performance overhead
2. Volatile fields cannot be passed by reference
3. A volatile write followed by a volatile read can still be reordered.

Volatile can only be applied to reference types and primitive types whose size is less than or equal to 32 bits. The volatile keyword cannot be applied fields of type double and long.

## Concurrent Collections

There are four Concurrent collections

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentBag<T>
- ConcurrentDictionary<T>

They are considerably slower than the non-concurrent versions. One must also remember that a collection being thread safe does not make client code using that collection thread safe. The stack, queue and bag are internally backed by linked lists which are more appropriate for low lock multithreaded implementation. This of course has repercussions on the memory usage.

# Locking issues

## Deadlock

Acquiring locks incurs a performance cost. Perhaps more importantly though injudicious use of locking can lead to deadlock. The following code shows how

**Figure 1Deadlock**

Deadlock

```
public void DeadlockCreator()
{
    lock (_lockA)  ← 1
    {
        new Thread(() =>
        {
            lock (_lockB)  ← 2
            {
                lock (_lockA)  ← 3
                {

                }
            }
        }).Start();

        Thread.Sleep(100);

        lock (_lockB)  ← 4
        {
        }

        Console.WriteLine("Finished");
    }
}
```

The following points give recommendation on reducing the likelihood of deadlock

### REDUCING THE LIKELIHOOD OF DEADLOCK

- Deadlock can occur if a thread pool thread blocks waiting for async call to complete
- Do not ever block a thread executed on the thread pool waiting for another function on the pool
- Do not create any class whose synchronous methods wait for asynchronous functions, since this class could be called from a thread in a pool
- Do not ever block a thread executed on the pool waiting for another function on the pool
- Do not use any class inside an asynchronous function if the class block waiting for asynchronous functions

### Limitations

- Protecting an object with a lock only works if all concurrent threads obtain the lock
- Particularly applicable to widely scoped usage e.g. static members
- In general .NET makes all static members thread safe whereas instance methods are not

### Rich Clients

- Only the thread that instantiates a UI object can call any of its members
- In windows forms call Invoke or BeginInvoke

## Race Condition Example

Consider the following piece of code which is not thread safe. The problem occurs when multiple threads call MakeDeposit. If one thread gets pre-empted after executing line one but before it can execute line two and a second thread executes MakeDeposit before the first thread gets re-scheduled, then effectively one of deposits gets lost.

**Figure 2 Non thread-safe code**

```
public class Account
 {
     public void MakeDeposit(double amount)
     {
         double newBalance = _balance + amount;   <-- 1
         _balance = newBalance;                   <-- 2
     }

     public double CurrentBalance => _balance;

     private double _balance;
 }
```

In order to prevent this kind of indeterminacy we need to implement a synchronization strategy. Synchronization is the topic of the next chapter but for now we show how to use a Monitor via the lock keyword to remove the race condition from this simple code.

**Listing 3Locking Fixes Race condition**

```
public class Account2
{
    public void MakeDeposit(double amount)
    {
        lock (this)   <-- 1
        {
            double newBalance = _balance +amount;
            _balance = newBalance;
        }
    }

    public double CurrentBalance => _balance;

    private double _balance;
}
```

# Questions – Synchronization, Communication and thread safety

## Thread Safety

**Describe thread safety?**

*A thread safe piece of code is one that behaves deterministically in the presence of multiple threads of execution*

**What guarantees does Windows give around when a thread will run?**

*Windows is a pre-emptive multithreaded operating system.*

*No guarantees around when our thread will run and when it will be pre-empted*

**Why are most types not threads safe?**

*Increases complexity of development*

*More inefficient in single threaded scenarios*

*Make type thread safe does not make the code calling it thread safe*

**Which instructions cannot be pre-empted?**

*Any operation which executes as a single atomic instruction cannot be pre-empted*

**Which instructions does the CLR guarantee to be atomic?**

*Reads and writes to object references*

*Reads and writes to primitive types of size 32bits or less*

**Which instructions are not guaranteed to be atomic?**

*Single reads or writes to fields larger than 32 bits such as longs or double*

*Combinations of reads and writes such as decrements and increments*

**What four conditions must be met needed for a race codition to be possible?**

*Memory locations accessible from more than one thread*

*Invariant is associated with shared memory needed*

*Invariant does not hold for some part of update*

*Another thread accesses the memory while invariant is broken*

**Other than non-atomic code sequences what else can cause to code that is correct in a single threaded environment execute incorrectly in multithreaded environments?**

*Compiler and CPU statement re-ordering and caching optimisations*

**What is needed to ensure program correctness?**

*A correct program should ensure all memory falls into three buckets*

*Thread exclusive*

*Read only*

*Lock Protected*

**How can one simply make large and complex types behave safely in multi threaded environments?**

*Protect large swathes of code with single exclusive lock*

*Minimise shared state such as stateless web server*

**When is this approach 1 essential?**

*Using third party types not thread safe*

**What support does .Net have for automatic locking?**

*1. Subclass ContextBoundObject*

*2. Apply Synchronization attribute to class*

**Why is this rarely a good idea?**

*The context bound object can introduce unintended deadlocks and overly course grained concurrency.*

**What are the only .net framework types that have built in thread safety?**

*Types in System.Collections.Concurrent*

*Primitive types*

**By convention, which members of .Net Framework types are thread safe?**

*Static members are thread safe and instance members are not*

**Why are enumerator and enumerable separate?**

*to enable read only thread safety. Two threads can enumerate the same collection simultaneously as each gets their own enumerator*

## Synchronization Constructs

**Give a broad categorisation of synchronization primitives?**

*Those that execute in user mode*

*Those that execute in kernel mode*

*Those that combine user mode and kernel mode operations*

**What are the advantages of user mode constructs?**

*They execute special hardware instructions so are very fast*

*They block for such a small period the OS doesn't even know there is blocking*

**Where is this particularly useful?**

*A thread pool will create new threads if it detects blocking. Creating, destroying and context switching between threads is expensive. If the thread pool does not detect blocking it cannot create new threads and this problem is avoided.*

**What is the downside of the user mode constructs?**

*As the OS does not know the thread is blocked it will continually reschedule the same thread cause spinning and wasted CPU cycles*

**What are the disadvantages of the kernel mode constructs?**

*They require transitioning from managed code to unmanaged user code to unmanaged kernel code and back which is expensive*

**What are the advantages of the kernel mode constructs?**

*As they block, they prevent spinning from wasting CPU cycles.*

**What if possible is a better alternative to using synchronization primitives?**

*Structure ones code to prevent the possible of multiple threads accessing shared data*

**What is the general problem of synchronization constructs?**

*The use of synchronization constructs is tedious, error prone and incurs a significant performance overhead.*

**What are the general uses of synchronization constructs?**

*Restrict the number of threads that can access a section of code concurrently*

*Ensure correctness which multiple threads access shared memory*

*Turn off compiler and CPU reordering and read/write caching optimisations*

*Enable threads to notify each other.*

**What guarantees does Windows give around when our threads will run?**

*In a pre-emptive multithreading environment, we have no guarantees around when our thread will run and when it will be pre-empted*

**What is the result of this?**

*This can lead to nondeterminism where the result depends on the order in which threads are pre-empted and scheduled*

**What is this known as?**

*Race condition*

**What four conditions are needed for a race condition?**

*Memory locations are accessible from more than one thread*

*Invariant is associated with shared memory*

*Invariant does not hold for some part of update*

*Another thread accesses the memory the while invariant is broken*

**What operations does the CLR guarantee to be atomic?**

*Reads and writes to object references and primitive types of size 32bits or less*

**What operations does the CLR not guarantee to be atomic?**

*Single reads or writes to fields larger than 32 bits such as longs or doubles*

*Combinations of reads and writes*

**How are race conditions prevented?**

*Synchronization constructs are used to restrict the number of threads that can concurrently access critical sections of code and by extension restrict the number of threads that can concurrently access the shared memory read/written inside those critical sections.*

**What are signalling constructs?**

*Enable threads to notify each other*

### User Mode Constructs

**What is the key to scalable and responsive applications?**

*Don't block the threads you have so they can be used and reused to execute other tasks*

**What benefits do the methods in Interlocked provide?**

*Extends atomic read/write semantics to 64-bit longs and doubles*

*Provides atomic read and write combinations for 32-bit and 64-bit fields and references*

**Implement a SpinLock without using kernel mode objects?**

```
public class MySpinLock
{
    private int taken;

    public void Enter()
    {
          while (true)
          {
                if (Interlocked.Exchange(ref taken, 1) == 0) return;

                // Yield the current thread in an attempt to lessen the
                // impact of spinning and allow the thread that holds the
                // lock to finish what it is doing
                Thread.Yield();
          }
    }

    public void Leave() => Volatile.Write(ref taken,0);
}
```

**Use Interlocked to create a thread safe multiply**

```
public double Multiply(ref double field, double y)
{
    while (true)
    {
          // The value in the field before we do the
          // exchange
          double fieldVal1 = field;

          // AttemptedProduct
          double product = fieldVal1 * y;

          // We only update field if the value in the field is
          // equal to snapShot1. In this case the value of
          // product was calculated from the current value of
          // field
          double fieldVal2 =
                CompareExchange(ref field, product, fieldVal1);

          // We can now compare the value in snapshot2 to see if
          // we actually did the update. If we did the update return it
          if (fieldVal1 == fieldVal2) return product;

          // Otherwise we need to spin around again
          Thread.Yield();
    }
}
```

## KERNEL MODE CONSTRUCTS

**Compared to user mode constructs what are the disadvantages of kernel mode constructs?**

*Managed Code -> Unmanaged user code -> Unmanaged kernel code and back*

*Slow*

**What are the benefits of the kernel mode constructs?**

*A thread trying to obtain a contended resource will block preventing spinning*

*Kernel mode constructs can synchronize managed and unmanaged code*

*Kernel mode constructs can synchronize across processes*

**What two key objects are kernel mode synchronization constructs based on?**

*Event and Semaphore*

**What is an Event?**

*Kernel based methods that act on a kernel Boolean that indicates if the event is signalled.*

**What is a Semaphore?**

*Kernel based methods that act on a kernel int called count*

*The count indicates how many more threads can be released.*

*Non-exclusive locking construct*

*Inter-process support*

*WaitHandle based*

**Describe the operation of a Semaphore?**

*A Semaphore is constructed with two numbers; a count and a max count. If the count is zero any thread calling WaitOne blocks. The count is decremented each time a thread enters the semaphore and incremented each time Release is called.*

**What are the two types of Semaphore?**

*Named system semaphores*

*Local semaphores*

**How does one create a named system semaphore?**

*By passing in a name to the constructor*

**How does one acquire and release a semaphore?**

*Semaphore.WaitOne*

*Semaphore.Release(int)*

**Compare Semaphore with Mutex**

*Mutex has thread affinity*

*Mutex is exclusive locking construct*

**When is a Semaphore useful?**

*Limiting concurrent execution of a critical section*

**How does a Semaphore with capacity one differ from a Mutex?**

*Semaphore does not have thread affinity.*

**If the Semaphore is full what happens when Release() is called?**

*One thread blocking on WaitOne gets access.*

**What is the .NET base class of kernel mode Synchronization objects?**

*WaitHandle which wraps a kernel object handle*

**What is method defined in WaitHandle that enables threads to block?**

*WaitOne*

**What are the subclasses of WaitHandle?**

- WaitHandle
  - Event
    - AutoResetEvent
    - ManualResetEvent
  - Semaphore
  - Mutex

**What is the behaviour of AutoResetEvent?**

*Turnstile*

*Set releases one thread at a time*

**What happens if multiple threads call WaitOne on a non-signalled ARE?**

*They form a queue of blocked threads*

**What happens if multiple threads call Set on a non-signalled ARE?**

*One thread from the queue is released and the ARE is automatically Reset to nonsignaled.*

**Write code to perform a very simple kernel blocking based lock using ARE?**

```
public class BlockingLock
{
    private AutoResetEvent _event = new AutoResetEvent(true);

    public void Enter() => _event.WaitOne();

    public void Leave() => _event.Set();
}
```

**What are the key methods of ARE?**

*WaitOne*

*Set*

**What is the core behaviour of a ManualResetEvent?**

*Latch*

*Set releases all threads*

**What is a Mutex and how does it differ from a monitor?**

*Mutual exclusive locking construct*

*WaitHandle based*

*Slower than Monitor*

*Supports inter-process locking by naming*

*Thread affinity*

*Might be used to ensure only one instance of a process runs at a time*

*Acquiring/Releasing Mutex 20 times slower than a Monitor*

**When might one use a Mutex?**

*Ensuring only one instance of an application runs*

**How does one create a cross process Mutex?**

*pass in a string name in the constructor*

**What is the purpose of the exclusive locking constructs?**

*Restrict access to critical sections of code to one thread at a time*

**Compare the core behaviour of AutoResetEvent, ManualResetEvent, Semaphore?**

*AutoResetEvent releases one thread when Set is invoked*

*ManualResetEvent releases all threads when Set is invoked*

*Semaphore releases n threads when Release (int n) is invoked*

**What is the core behaviour of a semaphore?**

*Threads calling WaitOne block if the count is zero*

*Threads unlock when count is above zero*

*Any time a thread waiting on a semaphore unblocks the count is decremented*

**What is a Semaphore used for?**

*Restrict the number of threads that can access a critical section*

*Non-exclusive kernel object-based synchronization construct.*

**What might be a use?**

*to limit concurrent access to a resource such as a disk*

**What is a Mutex?**

*Kernel based mutual exclusion construct.*

**What is a Mutex similar too?**

*Semaphore(1) and AutoResetEvent because is only releases one thread at a time.*

**What are the extra features of a Mutex over these constructs?**

*Thread affinity*

*Re-entrant*

**What are the downsides of these extra features?**

*Increase memory and reduce performance in situations when re-entry and affinity not needed.*

**Use an AutoResetEvent to implement your own Semaphore?**

```
public class BlockingLock
{
    private AutoResetEvent _event = new AutoResetEvent(true);
    private int _rentryCount = 0;
    private int _owningThread = 0;

    public void Enter()
    {
          int callingThread = Thread.CurrentThread.ManagedThreadId;

          if (callingThread == _owningThread)
          {
                _rentryCount++;
                return;
          }

          // Calling thread not the owning thread.
          // If the ARE is signalled this will return immediately
          // and fall through to the below. If the ARE is
          // non signalled then we will block here until the
          // owning thread calls exit
           _event.WaitOne();

           // Now we are the owing thread
           _owningThread = callingThread;
           _rentryCount = 1;
    }

    public void Exit()
    {
          if (Thread.CurrentThread.ManagedThreadId != _owningThread)
                throw new InvalidOperationException();

          if (--_rentryCount == 0)
          {
                _owningThread =0;
                _event.Set();
          }
    }
}
```

**Why is this likely to perform better than Semaphore?**

*The thread affinity and re-entry is implemented in user code rather than kernel code*

**What is meant by the term re-entrant?**

*A single thread can repeatedly obtain the same synchronization object. The object is released when the same number of exit calls have been made as enter calls.*

***What are the exclusive locking constructs?***

*Monitor*

*Mutex*

*Spinlock*

**What are the non-exclusive locking constructs?**

*Semaphore*

*SemaphoreSlim*

*ReaderWriterLockSlim*

**What are the signalling constructs?**

*ManualResetEvent*

*AutoResetEvent*

*AutoResetEventSlim*

*CountdownEvent*

*Barrier*

**Which of the signalling constructs are EventWaitHandles?**

*AutoResetEvent*

*ManualResetEvent*

**What is the purpose of signalling constructs?**

*Allow one or more threads to block until one or more other threads notify them*

**What is synchronization?**

*Coordinating the way multiple separate threads of execution interact such that program correctness is maintained.*

**What are the three exclusive locking constructs in .NET?**

*Monitor*

*Mutex*

*Spinlock*

**What is the difference between them?**

*Mutex is kernel based mutual exclusion construct.*

*Monitor is a managed code hybrid mutual exclusion construct*

*Spinlock is specialised for situations where there is little contention*

**Which one of the these back the lock construct?**

*Monitor*

**Is the code totalRequests++ thread safe and why?**

*No. The compiler will generate three assembler instructions to load the value into a register, increment it and write it back*

## Managed Hybrid Constructs

**What is a Monitor?**

*Managed exclusive locking construct*

*Simple resource scheduling mechanism*

*Policy is one thread at a time*

**What advantages does a Monitor gain by being a hybrid construct?**

*the idea is to use the speed and efficiency of user mode constructs when there is no contention and to fall back on kernel mode constructs when there is contention to prevent excessing spinning from wasting CPU cycles.*

**What is the eternal trade of when determining granularity of locking schemes?**

*Course provides correctness and sequential performance*

*Fine provides parallel scalability*

**How is a lock implemented in C#?**

*In C# implemented by Monitor class*

*Monitor.Enter causes all other threads to wait until Monitor.Exit called*

**What happens if one more thread contends a taken lock?**

*They are held on queue and given access to the lock on FIFO basis*

**When should one lock?**

*When accessing writable fields accessible from multiple threads?*

**When is the atomicity provided by a lock broken?**

*If an exception is thrown inside the block protected by the lock*

**What is the cost of acquiring and releasing a lock?**

*less than 50 nanoseconds if uncontended*

*a microsecond if contended and may be longer before the thread is rescheduled.*

**What is a microsecond?**

*10^-6 second*

**What is a nano second?**

*I thousandth of a microsecond or 10^-9 seconds*

**Why should string not be used as locks?**

*Because of interning separate pieces of code could inadvertently lock on the same string causing deadlock*

**Why should value types not be used as locks?**

*They would be boxed meaning each call to enter and exit would have a separate heap object and hence there would be no mutual exclusion at all.*

**Compare semaphore and SemaphoreSlim?**

*Semaphore can be used inter-process*

*Semaphore is a WaitHandle*

*Slim is 10 times faster*

*Slim supports cancellation*

**When would one use ReaderWriterLockSlim?**

*A type is safe for concurrent reads but not concurrent writes*

*There are many readers and only occasional updates*

**Given an example?**

*A server-side cache*

**Describe the behaviour?**

*A thread holding a write lock blocks all other threads obtaining a read or write lock*

*A thread holding a read lock prevents a thread obtaining a write lock*

*Many threads can have a read lock concurrently if no thread has a write lock*

**When would we use ReaderWriterLock over slim?**

*Never. ReaderWriterLockSlim is improved version*

**What are the methods of the ReaderWriterLockSlim?**

*EnterReadLock()*

*ExitReadLock()*

*EnterWriteLock()*

*ExitWriteLock()*

**Compare ReaderWriterLockSlim to lock**

*Allows more concurrent read than a lock would allow*

**What is a ManualResetEventSlim?**

*Lightweight latch*

*Set releases all threads*

**What are the key methods of ManualResetEventSlim?**

*Wait*

*Set*

**What is a CountDownEvent?**

*Enables thread(s) to wait for multiple threads to complete*

*Signalled when count goes to zero*

**What are the key methods of CountDownEvent?**

*Wait*

*Signal*

**Compare CountDownEvent and Semaphore?**

*A semaphore releases multiple threads when signalled*

*A CountDownEvent becomes signalled when count goes to zero*

**What is a Barrier?**

*Enables multiple threads to cooperate through multiple phases of an algorithm*

**Describe a use of the Barrier?**

*GC algorithm running in server mode. One thread per core start up with each thread working on a different thread stack as part of the mark. As each thread completes it must wait for all threads to complete mark. Then each thread compacts a different part of the heap again wating for the others to complete.*

## Lazy Initialization

**Why would one use the generic System.Lazy type?**

*To lazily initialize a field in a thread safe manner*

**Why would one use instead the static LazyInitializer class?**

*Memory is constrained and one does not want an instance of Lazy<T>*

*Share the same lock across multiple lazily initialized fields*

## Thread Local Storage

**What is thread local storage?**

*Storage that each thread has its own copy of*

**How does one set up thread local storage?**

*Use the ThreadStatic attribute*

*Use the generic ThreadLocal<T> type*

*Use Thread.GetData/Thread.SetData methods*

**What are the limitations of ThreadStatic?**

*Only applies to static fields*

*Field initializers only execute once*

**What are better?**

*Using ThreadLocal<T> which works for instance and static fields*

*Initializer executes on first call by each thread*

**What is a third option?**

*Thread.GetData/Thread.SetData*

*Allows ones to set/get slots in each threads local storage*

**When might one use TLS?**

*User session information or transaction information that is kept on per thread basis, used by a load of methods and we don't want to pass the whole thing in the stack.*

## WAIT AND PULSE

## FENCES AND BARRIERS

**What are the benefits of fences/barriers?**

*Prevent statement re-ordering and read/write caching optimisations by the compiler/CPU*

**Wht is this code not thread safe?**

```
public int value;
public bool isRead = false;

public void Write()
{
    value = 100;  ❶
    isRead = true;❷
}

public void Read()
{
    if (isRead)                      ❸
          Console.WriteLine(value);  ❹
```

*The order of ❶ and ❷ could be switched. Similarly, with ❸ or ❹.*

**How does one carry out a full fence in .NET?**

*by using Thread.MemoryBarrier*

**What is the effect of Thread.MemoryBarrier?**

*Read/write caching is prevented*

*CPU cannot reorder instructions such that memory accesses before the call to MemoryBarrier occur after the call to MemoryBarrier*

**What is the effect of Volatile.Read?**

*No reads/writes after Volatile.Read execute before Volatile.Read*

*The value read is the latest written by any processor, irrespective of caching*

**What is the effect of Volatile.Write?**

*No reads/writes before Volatile.Write execute after Volatile.Write*

*The value written is immediately available to all CPU's*

**What is the effect of volatile keyword?**

*All reads are performed by Volatile.Read*

*All writes are performed by Volatile.Write*

**Why is this a bad idea?**

*Unlikely all reads and writes to a field need Volatile.Read/Write and hence marking a field volatile incur needless performance overhead*

*Volatile fields cannot be passed by reference*

*A volatile write followed by a volatile read can still be reordered.*

**When can volatile be used?**

*reference types and primitive types whose size is less than or equal to 32 bits*

**Which types are not supported by volatile?**

*type double and long*

## Synchronous Collections

**What are the concurrent collections?**

- ConcurrentStack<T>
- ConcurrentQueue<T>
- ConcurrentBag<T>
- ConcurrentDictionary<T>

**When would one use them?**

*In highly concurrent scenarios?*

**What are the disadvantages of using them?**

*Considerable slower*

**What must one remember when using them?**

*A collection being thread sage does not make the code using it thread safe*

**How are the Stack, Queue and Bag implemented internally?**

*As linked lists*

**Why?**

*More appropriate for low lock multithreaded implementation.*

**What is the implication of this design choice?**

*Increased memory usage*

**Are concurrent Queue and concurrent stack completely lock free?**

*Concurrent queue and concurrent stack are completely lock free*

*They use compare and swap and memory barriers to prevent*

**What are the disadvantages of this?**

*This can spin when there is contention*

**When would one use concurrent bag?**

*Very efficient when multiple threads are producing and consuming values*

**Is ConcurrentDictionary lock free?**

*It is completely lock free for reads but uses locks for writes*

*It is optimized for reads*

## LOCKING ISSUES

**What is deadlock?**

*Two threads block waiting for a resource held by the other*

**What four conditions must hold for deadlock to occur?**

*Mutual Exclusion – a thread owns resource, other can't acquire*

*Holder of resource allowed to perform unbounded wait*

*Resources cannot be forcibly removed from current owners*

*Circular wait condition  exists*

**What is the result of deadlock?**

*The locked threads block indefintely*

**How can deadlock be prevented?**

*Have few enough locks that its never necessary to take more than one at a time*

*Have a convention on order in which locks are take ( levels )*

**What strategies can one employ to minimise deadlocks?**

*Avoid calling methods you don't own while holding a lock*

**What can reduce the need for locking?**

*Tasks and continuations*

*Immutable objects*

**Describe a deadlock specific to WPF?**

*If you call Dispatcher.Invoke from a thread holding a lock that a method currently executing on the UI thread needs there will be an immediate deadlock*

*The deadlock can occur also if the method being queued requires a lock the method calling invoke holds*

**How can these problems be reduced?**

*Use Dispatcher.BeginInvoke rather than Dispatcher.Invoke*

*Release locks before calling BeginInvoke or Invoke*

# WPF Threading Model

Consider the following piece of code which instantiates a minimal WPF user interface consisting of a single Window. This simple fragment highlights some important aspects of threading in WPF

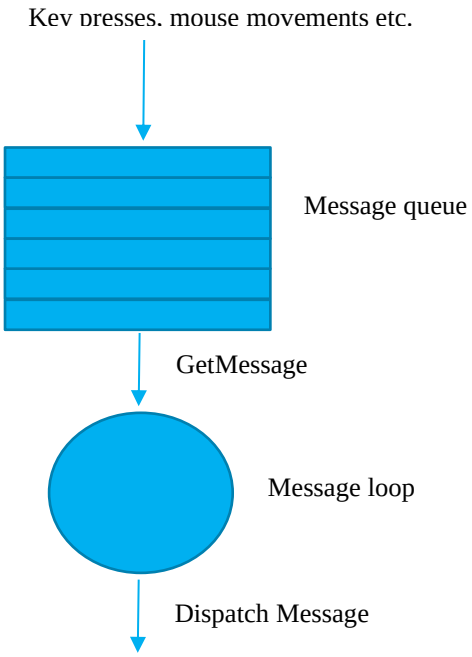
**Listing 4 Minimal WPF Interface**

```
[STAThread] ❶
public static void Main()
{
  Window w = new Window();
  w.Show();

  Dispatcher.Run(); ❷
}
```

The first thing of note is that we require a Window. Every WPF user interface must consists of at least one window. This is in fact just a Win32 window. Like a Win32 window a WPF window must run inside a thread whose COM apartment state is `ApartmentState.STA` We achieve by marking with the correct attribute ❶.

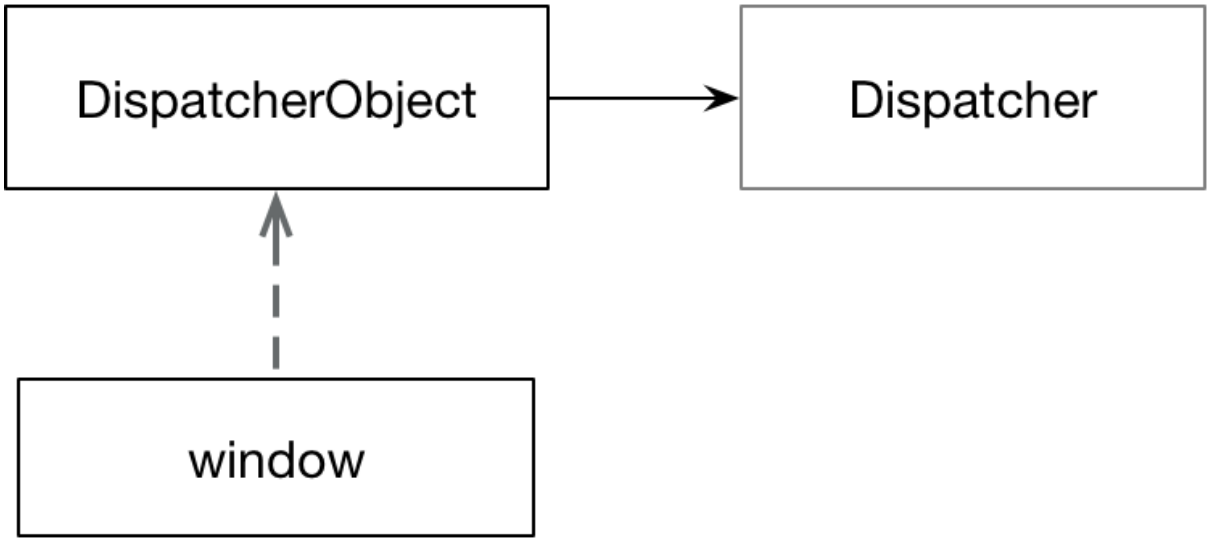
The second important point is that for the thread to be a UI thread it needs to have an associated `Dispatcher` which encapsulates a message loop. We set up the dispatcher by calling `Dispatcher.Run`❷ Inside this method WPF will setup the message loop. It is the message loop that makes the main thread a user interface thread. The message loop sits inside a loop, pulling messages from a message queue and dispatching them.

**Figure 3Windows Message Loop**




All WPF objects that inherit from `DispatcherObject` are associated with the same Dispatcher as their parent window. These objects are said to have thread affinity. Furthermore, these objects can only be safely updated on the thread with which they have affinity - the thread associated with their dispatcher. Any input messages from a window or its child `DispatcherObjects` such as keystrokes and mouse presses are pushed onto the message queue by the operating system. It is then the message loop's job to pull these from the queue, process them and dispatch them such that handlers can respond to them.

**Figure 4DispatcherObject**



If any thread other than the user interface thread wants to interact with DispatcherObjects' much do so by first obtaining the DependencyObject's dispatcher and then calling the Dispatcher's ❸ `BeginInvoke` method. The following piece of code highlights the approach

**Listing 5 Dispatcher**

```
[STAThread]
public static void Main()
{
    Window w = new Window();
    w.Loaded += WOnLoaded;
    w.Show();
    Dispatcher.Run();
}
private static void WOnLoaded(object sender, RoutedEventArgs
routedEventArgs)
{
    Window w = sender as Window;
    w.Background = new SolidColorBrush(Colors.Green);

    Task.Run(() =>
    {
          Thread.Sleep(2000);

          // Update the window on the thread it has affinity with
          Action d = () => w.Background = Brushes.NavajoWhite;
          w.Dispatcher.BeginInvoke(d, null); ❸
    });
}
```

> If we don't use the Dispatcher and instead try and directly set the background from the background thread then we get a runtime exception saying **System.InvalidOperationException: The calling thread cannot access this object because a different thread owns it**

## SynchronizationContext

SynchronizationContext is abstract class defined in System.ComponentModel. WPF and WinForms define subclasses which you can get a handle to be calling SynchronizationContext.Current from a UI thread.

If one holds a reference to a SynchronizationContext one can post to it with similar effect to calling BeginInvoke on a Dispatcher. The advantage of the Synchronisation context is that it is not dependent on any framework. So, code that works with SynchronizationContext can be utilised in WPF and WinForms applications. Calling Post is equivalent to directly calling BeginInvoke on a Dispatcher and calling Send is equivalent to calling Invoke directly on a dispatcher

One key use of SynchronizationContext is when executing code in the background and then using the results to update a user control using Tasks

The following code shows how to update the UI thread safely using a synchronisation context rather than directly accessing the dispatcher.

**Listing 6 SynchronizationContext**

```
private static void WOnLoaded2(object s, RoutedEventArgs e)
{
    Window w = s as Window;
    w.Background = new SolidColorBrush(Colors.Green);

    SynchronizationContext synchronizationContext =
SynchronizationContext.Current;

    Task.Run(() =>
    {
          Thread.Sleep(2000);
          synchronizationContext.Post(state => w.Background =
Brushes.NavajoWhite, null);
    });
}
```

### Listing 7SynchronizationContext Implementation

```
public class SingleThreadedSynchronizationContext : SynchronizationContext
{
    public SingleThreadedSynchronizationContext()
    {
        new Thread(() =>
                {
                    while (true)
                    {
                        try
                        {
                            _callbacks.Take()();
                        }
                        catch (Exception e)
                        {
                            Console.WriteLine(e);
                        }
                    }
                })
        { Name = "SingleThreadedSynchronizationContext" }
            .Start();
    }

    public override void Post(SendOrPostCallback d, object state)
    {
        _callbacks.Add(() => d(state));
    }

    public override void Send(SendOrPostCallback d, object state)
    {
        var tcs = new TaskCompletionSource<object>();

        _callbacks.Add(() =>
        {
            d(state);
            tcs.SetResult(null);
        });

        tcs.Task.Wait();
    }

    private readonly BlockingCollection<Action> _callbacks =
        new BlockingCollection<Action>();
}
```

# Questions - WPF Threading model

**What does a thread need to be a WPF UI thread?**

*for the thread to be a UI thread it needs to have an associated* `Dispatcher` *which encapsulates a message loop. We set up the dispatcher by calling* `Dispatcher.Run`

**What is thread infinity?**

*WPF objects which extend DispatcherObject can only be safely updated on the thread with which they have affinity - the thread associated with their dispatcher*

**How does one safely update a DispatcherObject from a thread other than the one on which it has infinity?**

*By obtaining its Dispatcher and calling Dispatcher.Invoke on it w.Dispatcher.BeginInvoke(d, null); Even better use SynchronizationContext.Post or SynchronizationContext.Send*

**What is the difference between BeginInvoke and Invoke?**

*Invoke blocks the calling thread until the passed delegate completes it work*

**why shouldn't one use Dispatcher.Invoke?**

*Dispatcher.Invoke blocks the calling thread until the Dispatchers associated thread executes the given delegate. The following examples show when using invoke causes a deadlock that is fixed by using BeginInvoke*

**What is a SynchronizationContext?**

*An abstraction for thread safe update of UI components. In WPF delegates to an underlying Dispatcher*

**What method on Synchronization is equivalent to Dispatcher.Invoke and which to Dispatcher.BeginInvoke?**

*Post is equivalent to BeginInvoke and Send is equivalent to Invoke.*

**Given an example of an application using multiple UI threads?**

*Windows explorer uses a different UI thread for each folder window.*

# Timers

# Questions Timers

**What is a Timer?**

*Executes a method at regular intervals many times*

**What are the different timers?**

*System.Threading.Timer*

*System.Timers.Timer*

*System.Windows.Forms.Timer*

*System.Windows.Threading.DispatcherTimer*

**Which is the simplest Multi-threaded timer?**

*System.Threading.Timer*

**What is System.Timers.Timer?**

*A wrapper around System.Threading.Timer*

**What does System.Timers.Timer add?**

*Synchronization object*

# Tasks

Tasks separate the specification and co-ordination of units of work from the details of how they are scheduled. Unlike threads, it is easy to get a return value from a task. Tasks can be chained together by specifying that one task continues when another completes. Large operations can be formed by combining smaller ones.

## Starting Tasks

**Listing 8 Starting Tasks**

```
❶
var t1 = Task.Factory.StartNew(() =>
Console.WriteLine("TaskFactory.StartNext"));

❷
var t2 = Task.Run(() => Console.WriteLine("Task.Run"));

❷
var t3 = new Task(() => Console.WriteLine("new Task()"));
t3.Start();
```

## Unit of Work definition separate from scheduling

The following piece of code creates two separate but identical units of work and then schedules each one differently

**Listing 9 Separating Scheduling and UOW**

```
Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

void Function()
{
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
}

Task t1 = new Task(Function);
Task t2 = new Task(Function);

t1.Start(TaskScheduler.Default);
t2.Start(TaskScheduler.Current);
```

## Chaining Tasks

Tasks can be chained together using Continuations. Continuations are set up using the `ContinueWith` method. The following code shows how to chain two tasks together. We have two very basic methods ❶,❷that carry out the work we want to do. We want ❷to execute once ❶is complete. We create a task ❸and pass in an Action delegate that invokes TaskOneFunction. We then add a continuation that will execute our second method. Note however we must add a wrapper method ❺ that takes the antecedent task and calls TaskTwoFunction

**Listing 10Chaining One**

```
void Main()
{
    MyExtensions.SetupLog4Net();
    logger.Info(nameof(Main));

    Task taskOne = ❸ new Task( new Action(TaskOneFunction));

    ❹
    taskOne.ContinueWith( new Action<Task>(TaskTwoFunctionWrapper));

    taskOne.Start();
}

❶
public void TaskOneFunction()
{
    logger.Info(nameof(TaskOneFunction));
}

❷
public void TaskTwoFunction()
{
    logger.Info(nameof(TaskTwoFunction));
}

❺
public void TaskTwoFunctionWrapper(Task antecedent)
{
    TaskTwoFunction();
}
```

### CHAINING VALUE RETURNING TASKS

Now we move on to consider how to chain value returning tasks together. If our first task is of type `Task<int>` then the continuations delegate function must take an argument of type `Task<int>`

**Listing 11 Chaining Two**

```
void Main()
{
    MyExtensions.SetupLog4Net();

    Task<int> taskOne = new Task<int>( new Func<int>(TaskOneFunction));

    Task<int> continuation =taskOne.ContinueWith( new
Func<Task<int>,int>(TaskTwoFunctionWrapper));

    taskOne.Start();
    logger.Info(continuation.Result);
}


public int TaskOneFunction()
{
    logger.Info(nameof(TaskOneFunction));
    return 5;
}

public int TaskTwoFunction(int x)
{
    logger.Info(nameof(TaskTwoFunction));
    return x * 2;
}

public int TaskTwoFunctionWrapper(Task<int> antecedent)
{
    return TaskTwoFunction(antecedent.Result);
}
```

### SUB-TASKS AND UNWRAP

Sometimes one task will want to use another task within its body. Where the tasks return values this can lead to a type of ❶`Task<Task<Tresult>>` which is rather inconvenient. We can get around this using the static method ❷Unwrap which creates a proxy task that only completes when the outer and inner tasks complete. It achieves this internally without blocking using callbacks.

**Listing 12Unwrapping**

```
Task<double> GetSpot() => Task.Run(() => 100.0);

Task<double> GetForward(double spot) => Task.Run(() =>spot
*Math.Exp(0.1));

❶Task<Task<double>> forward = GetSpot()
.ContinueWith(x => GetForward(x.Result));

forward
❷.Unwrap()
.ContinueWith(f => log.Info(f.Result));
```

The following example show how Unwrap and ContinueWith can be used to chain together value returing tasks where the output from one function forms the input to the next function

**Listing 13Unwrap and ContinueWith**

```
Task<int> Increment(int x)
{
      return Task.Run(() =>
    {
            log.Info(nameof(Increment));
            return ++x;
    });
}

var result = Increment(0)
.ContinueWith(f => Increment(f.Result))
.Unwrap().ContinueWith(f => Increment(f.Result))
.Unwrap().ContinueWith(f => Increment(f.Result))
.Unwrap().ContinueWith(f => Increment(f.Result))
.Unwrap();

result.ContinueWith(r => log.Info(r.Result));
```

## Scheduling Tasks

The examples of creating and starting tasks in the previous section all utilised default scheduling which schedules tasks on a thread pool. If we want to specify the scheduler, we can use the following call.

**Listing 14 Explicit Scheduling**

```
TaskScheduler scheduler = TaskScheduler.Default;
Task t3 = new Task(() => Console.WriteLine("new Task()"));
t3.Start(scheduler);
```

Again, we use the default scheduler but this time we explicitly specify it. The following built in options are supported

**Listing 15 Scheduling Options**

```
TaskScheduler s1 = TaskScheduler.Default;

TaskScheduler s2 = TaskScheduler.Current;

TaskScheduler s3 = TaskScheduler.FromCurrentSynchronizationContext()
```

It is often useful to add a level of indirection to the schedulers used. In this way we can switch in different schedulers as suits our needs. I.e. we might want to use a single threaded scheduler rather than thread pool when unit testing. Often a team will create something along the lines of `ISchedulerProvider` to provide this extra level of indirection.

If the out of the box .NET schedulers don't do what you want you can always implement your own by sub classing TaskScheduler.

**Listing 16 Custom Task Scheduler**

```
class CurrentThreadScheduler : TaskScheduler
{
    protected override void QueueTask(Task t) =>
          TryExecuteTask(t);

    protected override bool TryExecuteTaskInline(Task t, bool b) =>
          TryExecuteTask(t);

    protected override IEnumerable<Task> GetScheduledTasks() =>
          Enumerable.Empty<Task>();
}
```

### HOW SCHEDULERS WORK

Consider the following scheduler

### CHAINING AND SCHEDULING EXAMPLE

**Listing 17 Chaining and Scheduling Example**

```
private void ButtonBase_OnClickAsyncAwait(object sender, RoutedEventArgs
e)
{
      ❶
    int Id() => Thread.CurrentThread.ManagedThreadId;

    Console.WriteLine($@"Handler {Id()}");

    var rateTask = new Task<double>(() =>
    {
          Thread.Sleep(1000);
          Console.WriteLine($@"Task1 {Id()}");
          return 0.1;
    });

    var fwdTask = rateTask.ContinueWith(task =>
    {
          Thread.Sleep(1000);
          Console.WriteLine($@"Task2 {Id()}");
          return 100 * Math.Exp(task.Result);
    }, TaskScheduler.Default); ❸

    fwdTask.ContinueWith(task =>
    {
          Console.WriteLine($@"Task3 {Id()}");
          TextBlock.Text = task.Result.ToString();
    }, TaskScheduler.FromCurrentSynchronizationContext()); ❹

    rateTask.Start(TaskScheduler.Default); ❷
}
```

❶ The button click itself is executed by the UI thread and as such has an associated SynchronizationContext associated with it. ❷The first task in the chain is started and explicitly scheduled on the default thread pool scheduler. ❸The first continuation is also explicitly scheduled on the default scheduler. ❹The final continuation uses the special `TaskScheduler.FromCurrentSynchronizationContext()` to execute the task on the SynchronizationContext captured at the point the final task is created.

If we want the second background task to execute on the same thread pool thread as the first background task we can use the `TaskContinuationOptions.ExecuteSynchronously` ❺option as follows

**Listing 18 TaskContinuationOptions.ExecuteSynchronously**

```
var fwdTask = rateTask.ContinueWith(task =>
{
    Thread.Sleep(1000);
    Console.WriteLine($@"Task2 {Id()}");
    return 100 * Math.Exp(task.Result);
}, TaskContinuationOptions.ExecuteSynchronously); ❺
```

## Exception handling

Another benefit of tasks is that any uncaught exception raised from task code is re-thrown such that it can be caught by the caller when it invokes Task.Wait or Task.Result

**Listing 19 Exception Handling**

```
public Task GetExceptionTask()
{
    return Task.Run(() =>
    {
          MyLogger.Log("exception raised");
          throw new ArgumentException();
    });
}

[Test]
public void TaskExceptionTest()
{
    try
    {
          var exceptionTask = GetExceptionTask();
          exceptionTask.Wait();
    }
    catch (Exception e)
    {
          MyLogger.Log($"Exception {e.GetType()}");
          MyLogger.Log($"Inner exception {e.InnerException.GetType()}");
    }
}
```

Because a task can cause multiple exceptions calling Wait or GetResult results in any exceptions being wrapped by an AggregateException.

## Cancelling Tasks

The threading API provides two types that standardise cancellation

- CancellationTokenSource
- CancellationToken

By using two types the API seeks to separate the responsibilities of listening to and responding to cancellation events from the raising of cancellations. The following code shows a simple use case .

**Listing 20 Cancelling Tasks**

```
void Main()
{
    ❶
    var source = new CancellationTokenSource();

    ❷
    Task t = LongRunningAsyncMethod(source.Token);

    ❸
    source.CancelAfter(2);

    try
    {
          ❹
          t.Wait();
    }
    catch (Exception ex)
    {
          Console.WriteLine($"Caught {ex.InnerException}");
    }

    Console.WriteLine($"Final status: {t.Status}");
}

static async Task LongRunningAsyncMethod(CancellationToken token)
{                                                    ❷
    await Task.Delay(TimeSpan.FromSeconds(20), token);
}
```

❶ We start off by creating a CancellationTokenSource which we then pass into a long running operation ❷ that will take about 20 seconds to run. We then instruct the CancellationTokenSource to carry out a cancellation after 2 seconds ❸ and then carry out a wait on the task ❹. The output from executing this fragment is

```
Caught System.Threading.Tasks.TaskCanceledException: A task was canceled.
Final status: Canceled
```

# Questions - Tasks

For threading programming questions see

`linqpad\Queries\InterviewQuestions\Threading\Tasks`

**What do tasks do?**

Separates specification and co-ordination of units of work from the scheduling details

**Why use tasks?**

1. *Separate the specification and co-ordination of units of work from their scheduling.*
2. *Easy to get a return value from a task.*
3. *Tasks can be chained together*
4. *Exceptions throws by unit of work re-thrown when Wait/Result called*
5. *Standardised cancellation protocol*
6. *Large operations can be formed by combining smaller ones.*

**What is a Task<TResult> in computer science terms?**

*A future. A promise of a value (or exception) at some time in the future.*

**How can one start a task?**

```
Task.Factory.StartNew()

Task.Run()

Task t = new Task(); t.Start();
```

**Sometimes one task will want to use another task within its body. Where the tasks return values this can lead to a type of `Task<Task<Tresult>>` which is rather inconvenient. We can get around this using the static method Unwrap. What does Unwrap do?**

```
Task<double> GetSpot() => Task.Run(() => 100.0);

Task<double> GetForward(double spot) => Task.Run(() =>spot
*Math.Exp(0.1));

Task<Task<double>> forward = GetSpot()
.ContinueWith(x => GetForward(x.Result));

forward
.Unwrap()
```

```
  .ContinueWith(f => log.Info(f.Result));
```

*Unwrap creates a proxy task that only completes when the outer and inner tasks complete. It achieves this internally without blocking by using callbacks,*

**How can one define how a task is scheduled?**

*By passing in a scheduler to* `Task.Start`

**How are tasks chained?**

*A continuation is specified using the* `Task.ContinueWith` *method*

**Sometimes one task will want to use another task within its body. Where the tasks return values this can lead to a type of Task<Task<TResult>>. How would you deal with this?**

*By calling* `Task.Unwrap`

**What are the default scheduling options?**

`TaskScheduler.Current` *– Use the thread the current task is executing on.*

`TaskScheduler.Default` *– Use the thread pool to execute the task*

`TaskScheduler.FromCurrentSynchronizationContext`

**What might we do when testing?**

*Add an extra level of indirection such as* `ISchedulerProvider`

**If we want a continuation task to execute on same thread as the antecedent what do we do?**

*Use* `TaskContinuationOptions.ExecuteSynchronously`

**How do tasks with deal with the possibility of multiple exceptions?**

*Exceptions are wrapped in an* `AggregateException`

**Describe how cancellation works with Tasks?**

*The cancellation part of the API consists of two types* `CancellationTokenSource` *and* `CancellationToken`

*The purpose being to separate the responsibilities of listening to and responding to cancellation events from the raising of cancellations*

**What methods must one implement to write one's own scheduler?**

```
IEnumerable<Task> GetScheduledTasks();
void QueueTask(Task task);
bool TryExecuteTaskInline(Task task, bool taskWasPreviouslyQueued);
```

**Why should one be careful using Task.Result and Task.Wait?**

*They are blocking operations. If they are called on a thread that is needed for continuations, they can cause deadlock.*

# Awaiters

Awaiters provide another way of specifying a continuation. In the following code the task inside the OnCompleted handler is a continuation which is this example is executed on the same thread that the task was executing on

**Listing 21 Basic Awaiter Example**

```
void Main()
{
    ❶
    MyExtensions.SetupLog4Net();
    logger.Info("Main() started");

    Task<double> task = GetSpotPrice();

    TaskAwaiter<double> awaiter = task.GetAwaiter();

    awaiter.OnCompleted(() =>
    {
          ❸
          logger.Info(awaiter.GetResult());
    });
}


age
```

The code at points ❷and ❸execute on the same thread which is different from the main thread that ❶executes on.

Awaiters are types that meet the following criteria.

1. Implement `InotifyCompletion`
2. Contain a property `public bool IsCompleted`
3. Contain a method `public TResult GetResult()`

So we could create our own noddy awaiter for our own noddy type as follows.

**Listing 22 Implementing Awaiters**

```
async void Main()
{
    MyExtensions.SetupLog4Net();
    logger.Info("Main() started");

    AnAwaitableType<double> awaitable =
          new AnAwaitableType<double>(100.0);

    AwaiterImplementation<double> awaiter = awaitable.GetAwaiter();
    awaiter.OnCompleted(() =>
    {
          logger.Info($"{awaiter.GetResult()}");
    });
}

public class AnAwaitableType<T>
{
    public AnAwaitableType(T value) => _value = value;

    public AwaiterImplementation<T> GetAwaiter() =>
          new AwaiterImplementation<T>(_value);

    private T _value;
}

public class AwaiterImplementation<TResult> : INotifyCompletion
{
    public AwaiterImplementation(TResult value) => _value = value;

    public bool IsCompleted => true;

    public void OnCompleted(Action continuation) => continuation();

    public TResult GetResult() => _value;

    private TResult _value;
}
```

The default task awaiters have two interesting properties. First if a synchronization context is present at the point the continuation is the registered it will be captured and the continuation will be posted to that context. In the following code the onloaded handler is running on the UI Thread

**Listing 23Capturing SynchronizationContext**

```
private static void WOnLoaded(object s, RoutedEventArgs e)
{
    Task<double> task = GetSpotPrice();

    TaskAwaiter<double> awaiter = task.GetAwaiter();

    awaiter.OnCompleted(() => logger.Info(awaiter.GetResult()));
}

public static Task<double> GetSpotPrice()
{
    return Task.Run(() =>
    {
          logger.Info("Task running");
          return 110.0;
    });
}
```

Because the `OnLoaded` handler is running on the UI Thread the output is as follows.

```
[Main Query Thread] Main() started
[5] Task running
[Main Query Thread] 110
```

The second important point regards exceptions. If a task throws an exception, then the awaiter unwraps the exception from the AggregateException. Of course, this means if the AggregateException had more than one inner exception only the first is returned and the others are lost.

**Listing 24 TaskAwaiter unwraps exceptions**

```
      awaiter.OnCompleted(() =>
      {
            try
            {
                  logger.Info(awaiter.GetResult());
            }
            catch (Exception ex)
            {
                  logger.Info($"{ex.GetType().Name}");
            }

      });

…

public Task<double> GetSpotPrice()
{
      return Task.Run(() =>
      {
            throw new Exception("Something happened");
            return 0.0;
      });
}
```

If we want the code to give us the original AggregateException we can write our own awaiter to replace the use of the default Task awaiter. X

# Questions - Awaiters

**What are Awaiters?**

*Awaiters are another way of specifying continuations.*

**What are the two ways of specifying continuations in .NET?**

1. *Task.ContinueWith*
2. *Awaiters*

**How does one define an awaiter?**

*A type that*

- *Implements* `INotifyCompletion`
- *Contains a* `bool IsCompleted` *property*
- *Contains a* `TResult GetResult()` *method*

*Calling* `task.GetAwaiter` *returns a* `TaskAwaiter` *and the Awaiter's* `OnCompleted` *method can be used to schedule a continuation*

**What are the key features of task awaiters?**

*If a synchronisation context is present at the point a continuation is registered then it is captured and used for the continuation*

*If an exception is throws in the task the awaiter unwraps it from the aggregate exception*

# Async Await

C #5 added support for asynchronous functions which are methods or anonymous functions whose declaration contains the `async` keyword. The code inside the async method can utilize the `await` keyword (asynchronous wait) to make asynchronous code look like synchronous code. The compiler generated state machine allows one to write much cleaner and simpler code that does away with the need to write explicit callbacks and specialized multithreaded exception handling.

The basic idea is as follows. Imagine a piece of code that calls a long running function that returns a task. We can define the method with the async keyword and then internally we can use the `await` keyword as follows.

**Listing 25 Simple Async Await**

```
private async void ButtonBase_OnClick(object sender, RoutedEventArgs e)
{
Console.WriteLine($@"{nameof(ButtonBase_OnClick)}:
{Thread.CurrentThread.ManagedThreadId}");

Task<int> task = LongRunningTaskAsync();
task.Start(TaskScheduler.Default); ❶

int result = await task;❷

Console.WriteLine($@"{nameof(ButtonBase_OnClick)}:
{Thread.CurrentThread.ManagedThreadId}");❸
}

public Task<int> LongRunningTaskAsync()
{
  return new Task<int>(() =>
  {
    Console.WriteLine($@"LongRunningTaskAsync:
{Thread.CurrentThread.ManagedThreadId}");
    return 100;
  });
}
```

The button handler is already running on a UI thread. When we kick of the long running task on a background thread ❶we immediately call await on it. ❷The code looks like the method will block at this stage but what happens is that the method returns and a continuation is setup to execute the remainder of the method once the long running task completes. The async/await mechanism has logic to capture the synchronization context if one exists and hence the continuation is posted to the correct UI thread ❸

## How Async/Await works

Consider a very basic async method called `GetDoubleAsync` that awaits a task returned as the result of the method `GetLongRunningTask`. LongRunningTask represents an operation whose result will sometimes be available immediately and sometime will require an expensive background calculation. For educational purpuses we pass in a flag to show which path is being executed. We want to observe the call stack and thread of execution. In the general case the code after an await is a continuation and hence it does not have the caller of the async method in its stack trace.

**Listing 26 Long Running Task Async**

```
public Task<double> LongRunningTaskAsync(bool completeImmediately)
{
    if (completeImmediately)
    {
          Log(nameof(LongRunningTaskAsync));
          return Task<double>.FromResult(100.0);
    }

    return Task<double>.Run(() =>
    {
          Log(nameof(LongRunningTaskAsync));
          return 100.0;
    });
}
```

### TASK EXECUTES ON POOL AS LONG RUNNING

Consider now the case where the LongRunningTask requires to calculate its value on the ThreadPool

```
private async Task<double> GetDoubleAsync()
{
    ❶Log(nameof(GetDoubleAsync));
     double result = await LongRunningTaskAsync(false);
    ❷Log(nameof(GetDoubleAsync));
     return result;
}
```

The output for a given run is then as follows Notice the two parts of the method; before and after the await call both execute on the thread associated with the synchronization context and the long running task executes on a background thread

```
GetDoubleAsync 20
LongRunningTaskAsync 21
GetDoubleAsync 20
```

The stack trace at point ❶ looks as follows. Notice how it still has the caller frame.

```
UserQuery.GetDoubleAsync()
UserQuery.<<Main>b__0_0>d.MoveNext()
```

At point❷however the calling method has disappeared from the stack trace. We just have compiler generated code on the stack. This is the code the compiler will have generated to implement async await

```
  UserQuery.<GetDoubleAsync>d__1.MoveNext()
AsyncMethodBuilderCore.MoveNextRunner.InvokeMoveNext(Object stateMachine)
```

### TASK EXECUTES IMMEDIATELY

Now consider the case where the long running task result is available immediately.

```
private async Task<double> GetDoubleAsync()
{
    ❶Log(nameof(GetDoubleAsync));
     double result = await LongRunningTaskAsync(true);
    ❷Log(nameof(GetDoubleAsync));
     return result;
}
```

The output becomes as follows showing that everting executes on the same synchronization context thread.

```
GetDoubleAsync 12
LongRunningTaskAsync 12
GetDoubleAsync 12
```

And our stack trace then become as follows. First, before the async await ❶ note the caller Main is still on the stack

```
UserQuery.<GetDoubleAsync>d__1.MoveNext()
CompilerServices.AsyncTaskMethodBuilder`1.Start[TStateMachine](TStateMach
ine& stateMachine)
UserQuery.GetDoubleAsync()
UserQuery.<Main>b__0_0()
```

And after the async the caller is still on the stack because no continuation was needed due to the ❷long running task completing immediately

```
UserQuery.<GetDoubleAsync>d__1.MoveNext()
CompilerServices.AsyncTaskMethodBuilder`1.Start[TStateMachine](TStateMach
ine& stateMachine)
UserQuery.GetDoubleAsync()
UserQuery.<Main>b__0_0()
```

### COMBINING AWAIT STATEMENTS

Consider the following piece of code which needlessly reduced concurrency

```
public async Task SomeEventHandler()
{
    Log(1, $"{nameof(SomeEventHandler)} Before await");
    double rate = await GetPresentValue( 1.0);
}

public async Task<double> GetPresentValue(double t)
{
    Log(2, $"{nameof(GetPresentValue)} Before await");
    var pv = await GetFutureValue() * Math.Exp(await GetRate() * t);
    Log(5,$"{nameof(GetPresentValue)} After await");
    return pv;
}

public Task<double> GetRate()
{
    return Task<double>.Run(() =>
    {
          Log(4, $"{nameof(GetRate)}");
          return 0.1;
    });
}

public Task<double> GetFutureValue()
{
    return Task<double>.Run(() =>
    {
          Log(3, $"{nameof(GetFutureValue)}");
          return 100.0;
    });
}

public void Log(int num, String s)
{
    var onSc = sc == null ? false : sc.OnSynchThread();
    Thread t = Thread.CurrentThread;
    Console.WriteLine($"{num} ({t.ManagedThreadId},{onSc}) {s}");
}
```

The logging statements show the thread they are executing on. We see the output as follows.

```
1 (23,True) SomeEventHandler Before await
2 (23,True) GetPresentValue Before await
3 (5,False) GetFutureValue
4 (5,False) GetRate
5 (23,True) GetPresentValue After await
```

### IMPLEMENTING ASYNC/AWAIT WITH AWAITERS

We can show how we might go about writing our own async await implementation using awaiters. Note this is easy for very simple methods but quickly becomes very hard in complex methods hence the beauty of having the compiler generate state machines for us

```
private Task<double> GetDoubleAwaiter()
{
    Log(nameof(GetDoubleAwaiter));

    // Get the awaiter
    TaskAwaiter<double> awaiter =
LongRunningTaskAsync(true).GetAwaiter();

    // Special case when the result is already available
    if (awaiter.IsCompleted)
    {
          Log(nameof(GetDoubleAwaiter));
          return Task<double>.FromResult(awaiter.GetResult());
    }

    // General case requires us to register a continuation
    TaskCompletionSource<double> tcs = new
TaskCompletionSource<double>();
    awaiter.OnCompleted(() =>
    {
          Log(nameof(GetDoubleAwaiter));
          tcs.SetResult(awaiter.GetResult());
    });

    return tcs.Task;
}
```

### WRITING OUR OWN AWAITERS

Tasks come with an implementation of awaiters but we can create our own awaiters for types. For a type to be used on the rhs of an await expression it must have a method called GetAwaiter (or have a suitable extension method) that returns a type that implements a specific pattern

# Questions - Async Await

**What is a continuation?**

*A piece of code that will be executed once an asynchronous operation completes. The continuation knows where the calling method was before the asynchronous operation was started so it can continue from the correct place. When one starts an asynchronous operation we can tell it what we want to be executed as a continuation when that operation completes*

**How are continuations implemented in .NET?**

*As action delegates that are invoked with the result of the asynchronous operation*

**Why are continuations better than blocking?**

*They do force valuable threads to just wait around doing nothing until the asynchronous operation completes.*

**What is a SynchronizationContext?**

*Abstracts the concept of invoking delegates on particular threads*

**What are the methods of SynchronizationContext?**

*Post – asynchronous, similar to BeginInvoke*

*Send – synchronous, similar to invoke*

**What is async/await?**

*Language support, layered on top of TPL, for writing asynchronous code*

**What is the benefit of having language support?**

*Frees the developer from writing boilerplate code to perform explicit callbacks and have disparate error handling*

**What is the result?**

*The resultant asynchronous code looks superficially like synchronous code*

**What is an asynchronous function?**

*A method or an anonymous function declared with the async modifier*

**What is an anonymous function?**

*A lambda expression or an anonymous method*

**What are the valid return types of an async method**

*Void*

*Task*

*Task<TResult>*

**Why is it preferable to return Task rather than void?**

*Enables the caller to attach continuations, check for completion and failure.*

**What restrictions are there on the parameters of an async method?**

*Cannot be out or ref as the caller is no longer on the stack for the continuation part of the method*

**What is await?**

*Asynchronous wait. It prevents having to block when waiting for long running operations to complete. If the task being awaited is of type TResult it unwraps to a value of TResult*

**What does await do?**

*Instructs the compiler to build a continuation*

*Consume asynchronous operation*

*If the task is of type Task<TResult> it unwraps a value of TResult*

**What is the benefit of the wrapping and unwrapping with async/await**

*Makes is easy for async methods to consume the results of other async methods*

**Where can await be used?**

*Inside an async method*

*Not inside unsafe code*

*Not inside a lock*

**why not inside a lock?**

*A monitor can only be released by the thread that acquired it and the continuation could be on another thread*

**What are the advantages?**

*The compiler generated state machine allows one to write much cleaner and simpler code that does away with the need to write explicit callbacks and specialized multithreaded exception handling.*

**Can you see any problem with the following code?**

```
public async Task<double> GetPresentValueInefficient(double t)
{
    var pv = await GetFutureValue() * Math.Exp(await GetRate() * t);
    return pv;
}
```

*The await calls are needlessly synchronized. One will wait for the other to complete*

**Improve the code from the following question?**

```
public async Task<double> GetPresentValueEfficient(double t)
{
    var fvTask = GetFutureValue();
    var rateTask = GetRate();

    var pv = await fvTask * Math.Exp(await rateTask* t);
    return pv;
}
```

**What happens when this code is executed?**

```
    double forward = GetForward(await GetSpot(), await GetRate());
    logger.Info(forward);
```

*Answer*

```
    Task<double> spotTask = GetSpot();
    double spot = await spotTask;
    Task<double> rateTask = GetRate();
    double rate = await rateTask;
    double forward2 = GetForward(spot,rate);
    logger.Info(forward2);
```

**When will the caller call stack be present at a line after an await?**

*If the task has completed at the point the await is called*

**When will an async method not return on encountering an await expression?**

*If the expression being awaited has already completed*

**Write code using an awaiter to achieve the same result as this code?**

```
public async Task<double> GetPresentValue(double t)
{
    double rate = await GetRate();
    return rate;
}

public Task<double> GetRateAwaiter()
{
    Task<double>  rateTask = GetRate();
    TaskAwaiter<double> rateAwaiter = rateTask.GetAwaiter();
    TaskCompletionSource<double> tcs = new
TaskCompletionSource<double>();

    if ( rateAwaiter.IsCompleted)
    {
          return Task<double>.FromResult(rateAwaiter.GetResult());
    }
    else
    {
          rateAwaiter.OnCompleted(() =>
        {
                tcs.SetResult(rateAwaiter.GetResult());
        });
    }

    return tcs.Task;
}
```

# Thread communication

Interaction

Communication

Synchronization

Coordination

Thread can communicate using simple blocking methods or signalling constructs and event wait handles

### SIMPLE BLOCKING METHODS

- Waiting for another thread to finish using Join
- Waiting for period to elapse

### SIGNALLING CONSTRUCTS AND EVENT WAIT HANDLES

- Allow a thread to pause until receiving notification from another
- Avoids the need for inefficient polling
- Event Wait Handles

## Questions - Thread Communication

**What is the most common method of communication between thread?**

*Shared memory*

**What is the effect of the join method?**

*Causes the calling thread to block until the given thread ends*

**Why is it rarely used in practice?**

*Because most threads are daemons, have no result or communicate using some synchronization construct*

# Patterns

## Obsolete patterns

### Asynchronous Delegates

We can invoke a delegate asynchronously using its begininvoke method. The delegate will execute on the thread pool. Invoking EndInvoke frees any resources and retrives its return value.

```
Func<int, int> squareDelegate = (x) => x * x;
IAsyncResult ias = squareDelegate.BeginInvoke(2, null, null);
int result = squareDelegate.EndInvoke(ias);
Console.WriteLine(result);
```

Asynchronous delegates introduce significant overhead and one should wherever possible use tasks instead.

### Asynchronous Programming Model

### BackgroundWorker

### Asynchronous delegates