CSC353

More about Synchronization

Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Beyond Monitor – implement other synchronization strategies
2. Work with Synchronized Collections

Implement other synchronization strategies

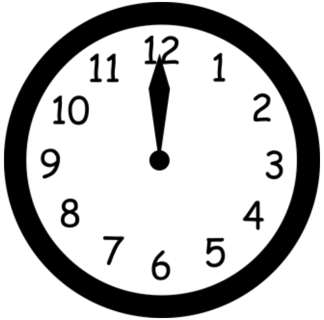
Xamarin University

Tasks

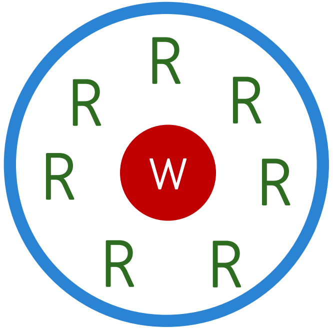
1. Examine Synchronization strategies
2. Use Wait Handles to control execution of multiple threads
3. Control multiple thread entry to a resource
4. Coordinate thread activity

Synchronization strategies

- Monitors are the primary strategy to synchronize data access, but .NET provides other synchronization mechanisms for specific cases



Efficient waiting



High reader vs. writer ratio

Allow **n** accesses simultaneously

Coordinate activity

Reminder: multiple lock coordination

- Be careful about waiting on multiple monitors – this will almost always produce race conditions and deadlock scenarios in your code

```
public class Account
{
    object accountLock = new object();

    public static void Transfer(Account accountFrom, Account accountTo, decimal amount)
    {
        lock (accountFrom.accountLock)
        {
            lock (accountTo.accountLock)
            {
               ... // Transfer money
            }
        }
    }
}
```

What if some other thread owns this lock and is waiting on the first one?

Locking with Mutex

Xamarin University

- A **`Mutex`** is a locking object that can be used across processes – although this extra power is not typically useful in mobile applications

```
public class Account
{
    Mutex accountLock = new Mutex();

    public static void Transfer(Account accountFrom, Account accountTo, decimal amount)
    {
        accountFrom.accountLock.WaitOne();
        accountTo.accountLock.WaitOne();

        ... // Transfer money

        accountFrom.accountLock.ReleaseMutex();
        accountTo.accountLock.ReleaseMutex();
    }
}
```

`WaitOne` methods acquire the lock

The **`ReleaseMutex`** method releases the lock

WaitHandle coordination

- **`Mutex`** derives from **`WaitHandle`** so it can be used with the **`WaitHandle`** static synchronization methods such as **`WaitAll`** and **`WaitAny`**

```
public class Account
{
    Mutex accountLock = new Mutex();

    public static void Transfer(Account accountFrom, Account accountTo, decimal amount)
    {
        WaitHandle.WaitAll(new[] { accountFrom.accountLock, accountTo.accountLock });
        try {
            ... // Transfer money
        }
        finally {
            accountFrom.accountLock.ReleaseMutex();
            accountTo.accountLock.ReleaseMutex();
        }
    }
}
```

The operating system ensures that no deadlock will occur here and that this thread will own both mutexes before returning

Xamarin University

Release Mutex

- Always call **`ReleaseMutex`** when you are finished – this releases the lock and allows other clients to acquire the lock and access the resource

```
public class Account
{
    Mutex accountLock = new Mutex();

    public static void Transfer(Account accountFrom, Account accountTo, decimal amount)
    {
        WaitHandle.WaitAll(new[] { accountFrom.accountLock, accountTo.accountLock });
        try {
            ... // Transfer money
        }
        finally {
            accountFrom.accountLock.ReleaseMutex();
            accountTo.accountLock.ReleaseMutex();
        }
    }
}
```

Must always release the mutexes when you are finished accessing the data

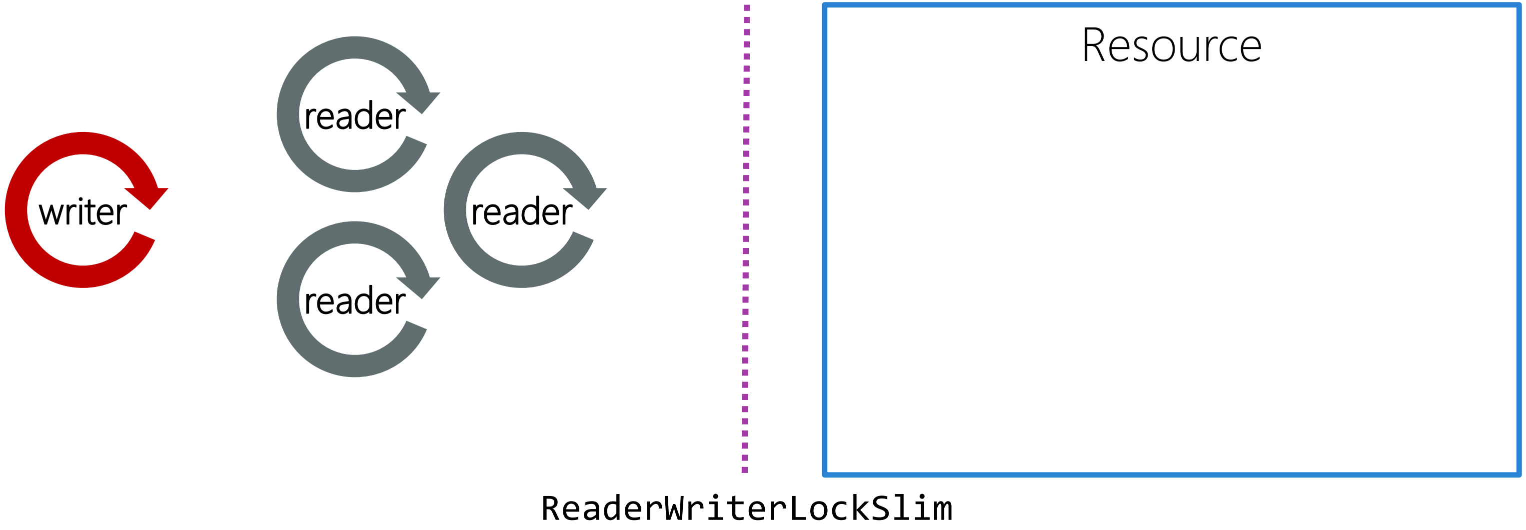
Reader vs. writer ratio

- Monitors block all thread access to a block of code while the monitor is owned; even if all the thread wants to do is *read* the data

- In these cases, it would be safe to allow multiple threads to access the resource as long as they do not modify any data

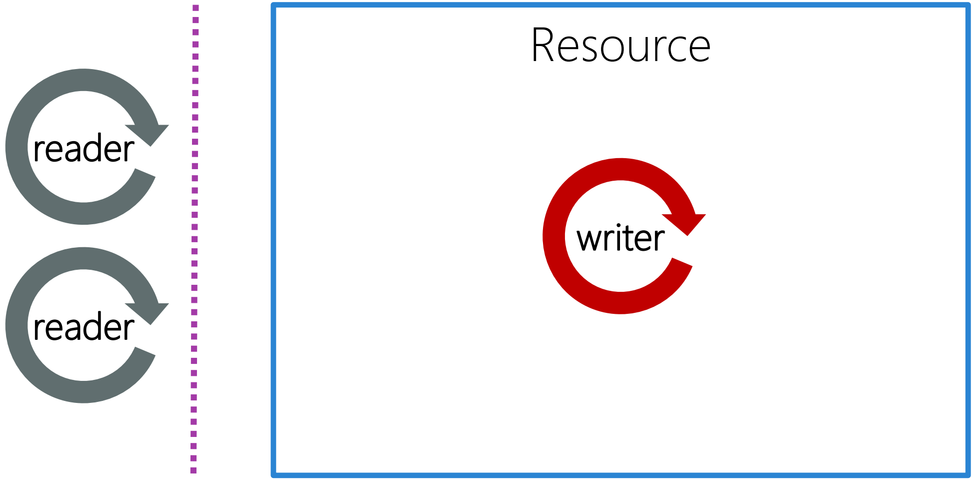
Introducing ReaderWriterLockSlim

- `ReaderWriterLockSlim` is a locking mechanism that allows multiple readers simultaneous access to a resource, but only a single writer at a time



Introducing ReaderWriterLockSlim

- `ReaderWriterLockSlim` is a locking mechanism that allows multiple readers simultaneous access to a resource, but only a single writer at a time



`ReaderWriterLockSlim`

Using ReaderWriterLockSlim

```
Dictionary<string, string> cache = new Dictionary<string, string>();
ReaderWriterLockSlim rwls = new ReaderWriterLockSlim();

string GetCachedValue(string key) {
    rwls.EnterReadLock();
    try { return cache[key]; }
    finally { rwls.ExitReadLock(); }
}

void SetCacheValue(string key, stri
    rwls.EnterWriteLock();
    try { cache.Add(key,value); }
    finally { rwls.ExitWriteLock(); }
```

Multiple threads are allowed to enter and exit the read lock simultaneously

.. but only one thread (at a time) can enter the write lock, all read lock requests are blocked while the write lock is owned

Be careful: There are several older classes with similar names that are either deprecated or more expensive to use – should generally prefer types that end with `Slim`

Should I prefer RWLS?

- Always start with Monitor; it is, by far, the most efficient lock in .NET

- For infrequent update cases, try replacing with RWLS *and then profile the application*

- For certain scenarios it may be more efficient, for others it might not be; must test it to find out

Demonstration

Using ReaderWriterLockSlim

Xamarin University

Allowing multiple threads

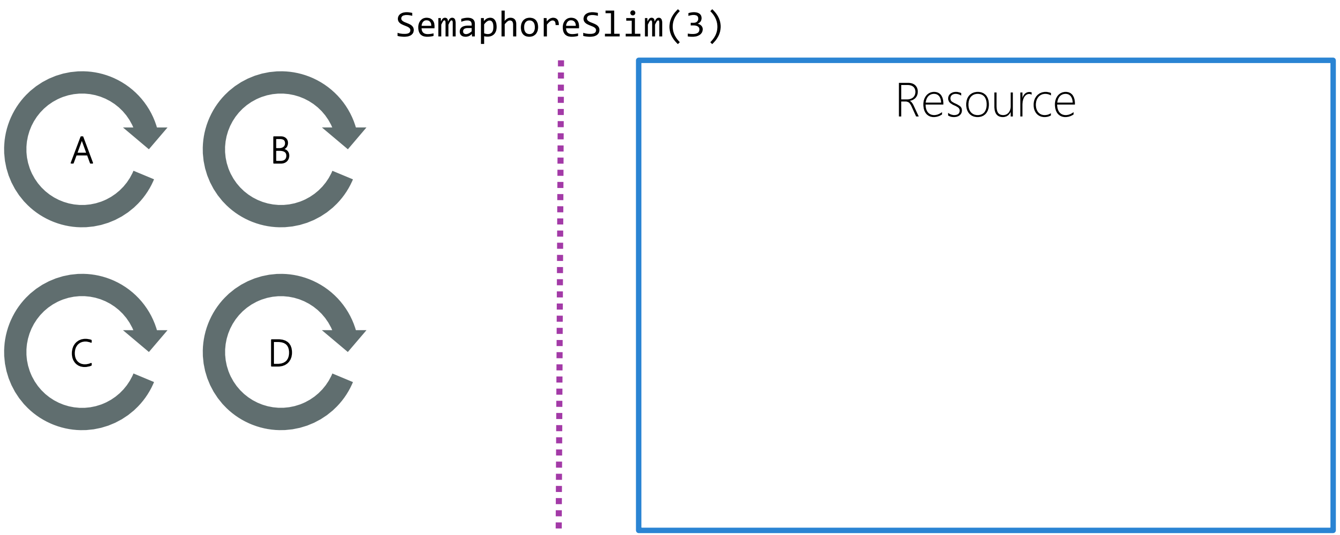
- Monitors allow one thread at a time; but sometimes we'd like to allow for more than one thread, up to a known limit

- This is done with a *semaphore* which allows **N** number of simultaneous accesses

- New threads cannot acquire semaphore until an existing owner releases it

Introducing SemaphoreSlim

- `SemaphoreSlim` allows for a known number of simultaneous acquisitions



Using SemaphoreSlim

```
public class LicenseServer
{
    SemaphoreSlim guard = new SemaphoreSlim(3);  // 3 simultaneous requests allowed

    public async Task<string> RequestLi                    ng key) {
        await guard.WaitAsync();

        try {
            return await InternalRequest(key);
        }
        finally {
            guard.Release();
        }
    }
    ...
}
```

Initialize with the allowed count

Using SemaphoreSlim

```
public class LicenseServer
{
    SemaphoreSlim guard = new SemaphoreSlim(3);  // 3 simultaneous requests allowed

    public async Task<string> RequestLi
        await guard.WaitAsync();

        try {
            return await InternalReques
        }
        finally {
            guard.Release();
        }
    }
    ...
}
```

Attempt to acquire the semaphore – class includes both async and blocking versions of **`Wait`** method; be aware that the lock is *not* reentrant, even for the same thread

Using SemaphoreSlim

```
public class LicenseServer
{
    SemaphoreSlim guard = new SemaphoreSlim(3);  // 3 simultaneous requests allowed

    public async Task<string> RequestLicenseCodeAsync(string key) {
        await guard.WaitAsync();

        try {
            return await InternalRequest(key);
        }
        finally {
            guard.Release();
        }
    }
    ...
```

Must make sure to pair every **`Wait`** with a **`Release`** method call

There are several variations of **`Wait`** available, including ones that take **`CancellationToken`**s to be able to cancel the wait from an external request

Demonstration

Using Semaphore

Coordinating activity

- Sometimes the synchronization required is to *wait* for an event to occur to coordinate two or more threads

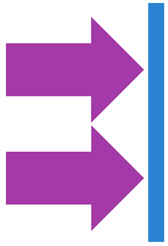
Something happened

Countdown coordination

Rendezvous coordination

Coordination terminology

- Several common terms / methods are used when using coordination objects



Wait

Threads call Wait to block until the coordination event occurs (is "signaled")

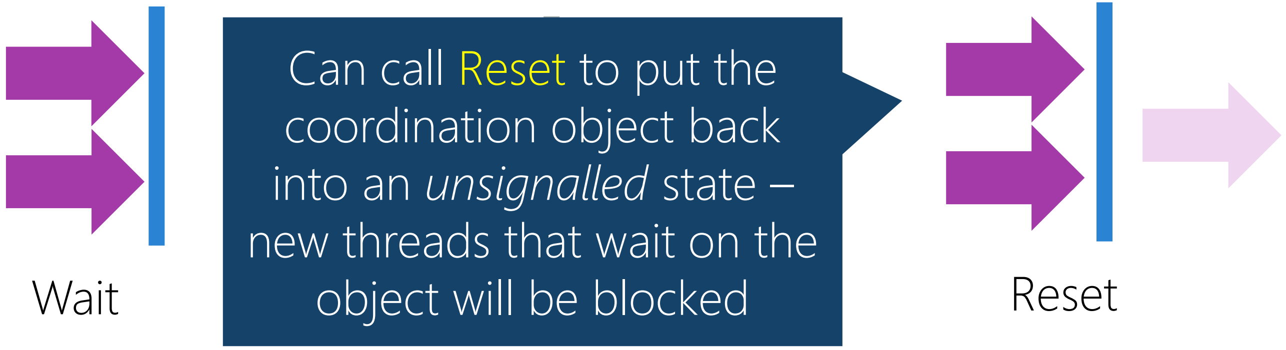
Coordination terminology

- Several common terms / methods are used when using coordination objects

When the event *does* happen, a thread calls Signal (or sometimes Set) to tell any waiters

Coordination terminology

- Several common terms / methods are used when using coordination objects

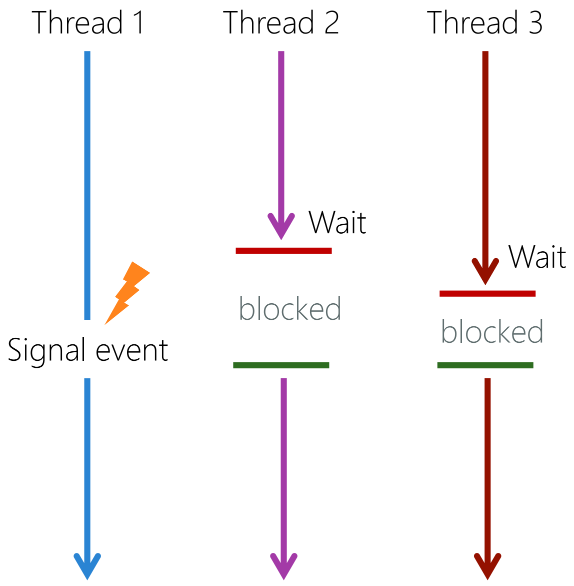


Event-based coordination

- Sometimes, two or more background threads need to coordinate on a specific action occurring

- This can be achieved using a `ManualResetEventSlim` object

- Particularly useful when *multiple* threads must be coordinated to a specific action

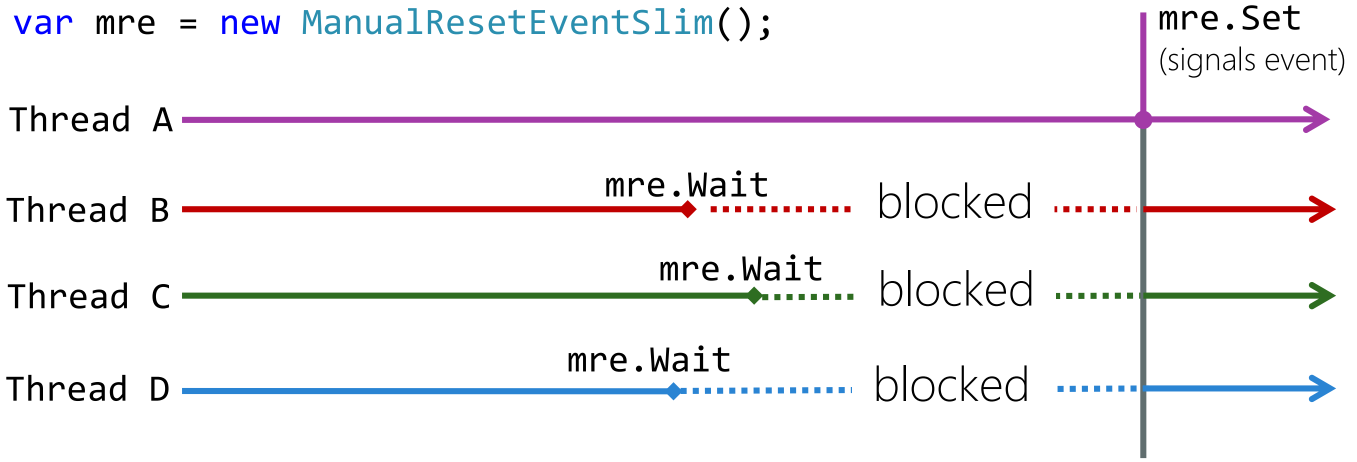


All waiting threads are scheduled when event is signaled

Event-based coordination

- `ManualResetEventSlim` releases all waiting threads when it is signaled; it is used to indicate a specific action has occurred in a thread

```
var mre = new ManualResetEventSlim();
```



Using ManualResetEventSlim

- We want to build a thread-safe, blocking queue

```
BlockingQueue<string> queue = new BlockingQueue<string>();
```

```
string newValue = queue.Dequeue();
```

```
queue.Enque
```

Multiple threads can call **Dequeue**, each will block until data is available, and only one thread will see the data

Using ManualResetEventSlim

❖ We want to build a thread-safe, blocking queue

```
BlockingQueue<string> queue = new BlockingQueue<string>();
```

```
string newValue = queue.Dequeu
```

Multiple threads can call **Enqueue**, and if threads are waiting on data, one will process it

```
queue.Enqueue("Pass to a thread!");
```

Event-based coordination

- Blocking queue will utilize a **`Queue<T>`** and a **`ManualResetEventSlim`** to provide notification about data availability

```
public class BlockingQueue<T>
{
    Queue<T> queue = new Queue<T>();
    ManualResetEventSlim hasData = new ManualResetEventSlim(false);
    ...
}
```

Event starts in *un-signaled* state by default, can specify initial state in constructor or call **`Reset`** to change it back to un-signaled later

Event-based coordination

- **Enqueue** method will lock queue, add our item and then signal the event if this is the only data element available to wake up any waiters

```
public class BlockingQueue<T>
{
    public void Enqueue(T obj) {
        lock (queue) {
            queue.Enqueue(obj);
            if (queue.Count == 1)
                hasData.Set();
        }
    }
    ...
```

To unblock waiting threads, must signal the event by calling **Set**; any waiters are then made ready for scheduling and have the opportunity to run

Event-based coordination

- **Dequeue** will lock queue and then release the lock and wait if no data is available; once data arrives, we lock queue and try to get the data

```
public T Dequeue() {
   T value;
   bool taken = true; Monitor.Enter(queue);
   try {
      while (queue.Count == 0) {
         taken = false; Monitor.Exit(queue);
         hasData.Wait();
         Monitor.Enter(queue); taken = true;
      }
      value = queue.Dequeue();
      if (queue.Count == 0)
         hasData.Reset();
   }
   finally { if (taken) Monitor.Exit(queue); }
   return value;
}
```

Wait causes thread to *block* until event is signaled; can pass optional timeout, or **CancellationToken** to provide for cancellation

Event-based coordination

- **Dequeue** must also reset the event once we have no more data available

```
public T Dequeue() {
   T value;
   bool taken = true; Monitor.Enter(queue);
   try {
      while (queue.Count == 0) {
         taken = false; Monitor.Exit(queue);
         hasData.Wait();
         Monitor.Enter(queue); taken = true;
      }
      value = queue.Dequeue();
      if (queue.Count == 0)
         hasData.Reset();
   }
   finally { if (taken) Monitor.Exit(queue); }
   return value;
}
```

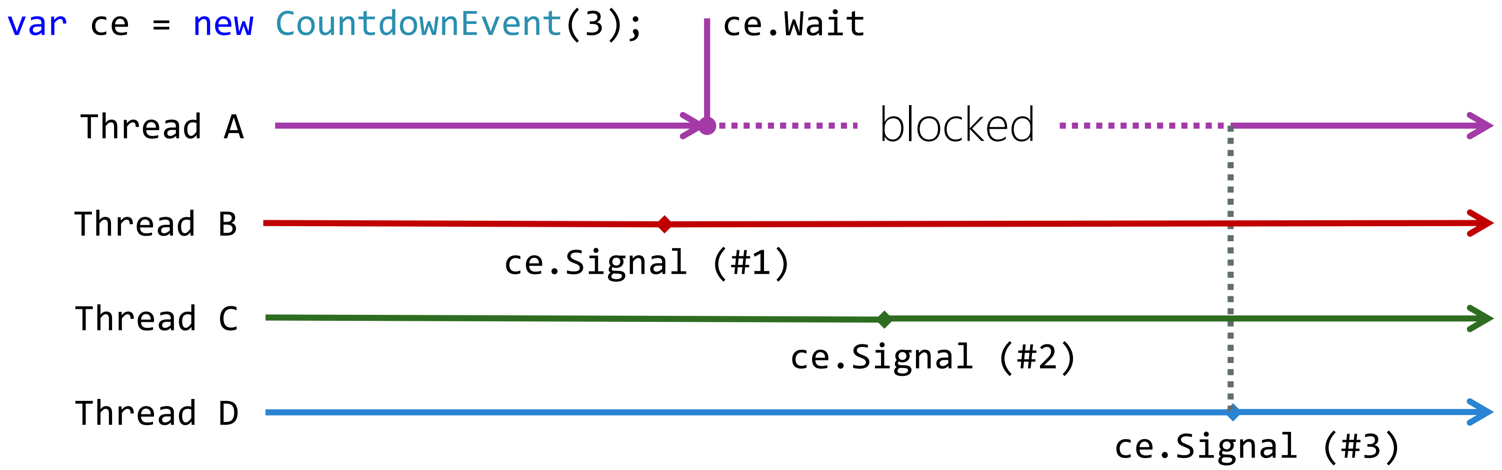
Reset causes the event to move back to the *unsignaled* state so that new threads that call **Wait** will be blocked on the event

Countdown coordination

❖ A common coordination style is to wait until a set number of activities have occurred or completed using a `CountdownEvent`

Countdown coordination

- `CountdownEvent` unblocks waiters when it has been signaled a specific number of times



Countdown coordination

- `CountdownEvent` allows one thread to wait for N operations to occur

```
CountdownEvent cde = new CountdownEvent(5);
for (int i = 0; i < cde.InitialCount; i++)
{
    Task.Run(() => {
        // Do work
        cde.Signal();
    });
}

cde.Wait(1000);
int doneAfter1Second = cde.InitialCount - cde.CurrentCount;
cde.Wait();
```

Must indicate how many times we will be signaled

Countdown coordination

Xamarin University

- `CountdownEvent` allows one thread to wait for N operations to occur

```
CountdownEvent cde = new CountdownEvent(5);
for (int i = 0; i < cde.InitialCount; i++)
{
    Task.Run(() => {
        // Do work
        cde.Signal();
    });
}

cde.Wait(1000);
int doneAfter1Second = cde.InitialCount - cde.CurrentCount;
cde.Wait();
```

Each subordinate task/thread then signals the `CountdownEvent`

Countdown coordination

- `CountdownEvent` allows one thread to wait for N operations to occur

```
CountdownEvent cde = new CountdownEvent(5);
for (int i = 0; i < cde.InitialCount; i++)
{
    Task.Run(() => {
        // Do work
        cde.Signal();
    });
}

cde.Wait(1000);
int doneAfter1Second = cde.InitialCount - cde.CurrentCount;
cde.Wait();
```

Can block waiting for all the operations to occur; supplying an optional timeout or `CancellationToken`

Countdown coordination

- `CountdownEvent` allows one thread to wait for N operations to occur

```
CountdownEvent cde = new CountdownEvent(5);
for (int i = 0; i < cde.InitialCount; i++)
{
    Task.Run(() => {
        // Do work
        cde.Signal();
    });
}

cde.Wait(1000);
int doneAfter1Second = cde.InitialCount - cde.CurrentCount;
cde.Wait();
```

Can look at properties to determine how many operations have been counted so far

Rendezvous coordination

- Sometimes useful to know when a set of threads have hit a certain point in their processing

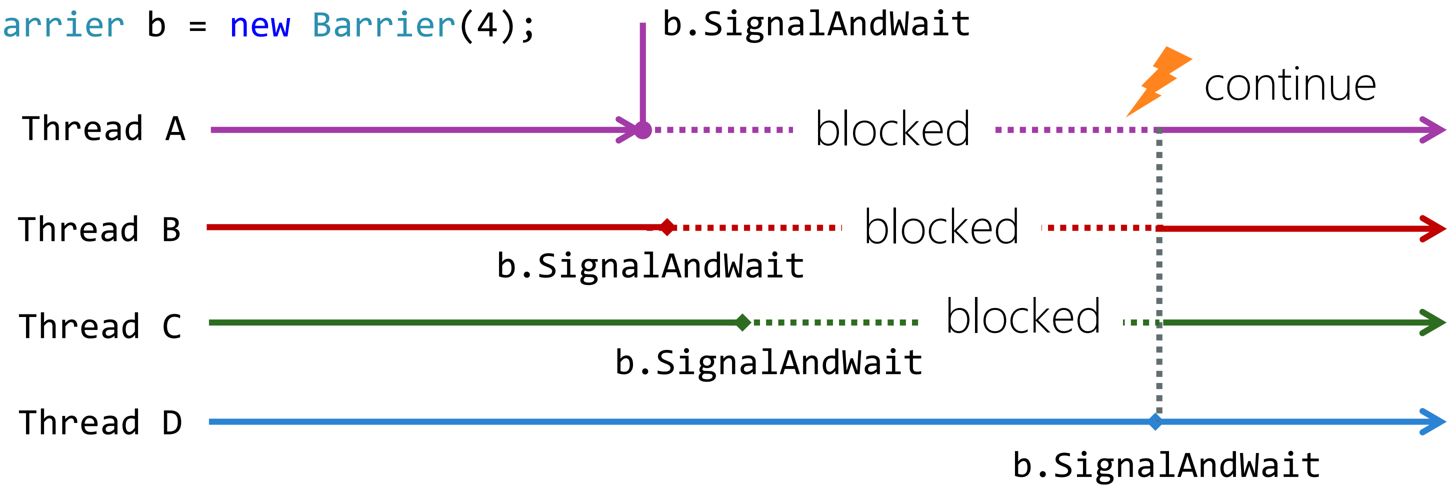
- Then have each thread *wait* until all other threads have also hit that point; once all are in phase, we let the threads continue

- The **`Barrier`** class allows for this sort of "rendezvous" coordination

Rendezvous coordination

- **`Barrier`** class provides for "rendezvous" style coordination where our work is done in *phases* and must be synchronized between each phase

```
Barrier b = new Barrier(4);
```



Rendezvous coordination

- **`Barrier`** class provides for "rendezvous" style coordination where our work is done in *phases* and must be synchronized between each phase

```
Barrier b = new Barrier(3);
void RunAHorse() {
    ... // Initialize the horse
    // Wait on the gate (barrier)
    b.SignalAndWait();
    // All horses ready, gate is open - GO!
}
...
for (int i = 0; i < 3; i++) {
   Task.Run(RunAHorse);
}
```

Must specify how many signals to require before unblocking callers

Xamarin University

Rendezvous coordination

- **`Barrier`** class provides for "rendezvous" style coordination where our work is done in *phases* and must be synchronized between each phase

```
Barrier b = new Barrier(3);
void RunAHorse() {
    ... // Initialize the horse
    // Wait on the gate (bar
    b.SignalAndWait();
    // All horses ready, ga
}
...
for (int i = 0; i < 3; i++) {
   Task.Run(RunAHorse);
}
```

Threads will block until **`SignalAndWait`** is called 3 times

Rendezvous coordination

- **`Barrier`** class provides for "rendezvous" style coordination where our work is done in *phases* and must be synchronized between each phase

```
Barrier b = new Barrier(3);
void RunAHorse() {
    ... // Initialize the horse
    // Wait on the gate (barrier)
    b.SignalAndWait();
    // All horses ready, gate is open - GO!
}
...
for (int i = 0; i < 3; i++) {
   Task.Run(RunAHorse);
}
```

Once we hit 3 calls to **`SignalAndWait`**, all threads are released and the barrier should be disposed

Dealing with dynamic participants

- Can add and remove additional counts from the **`Barrier`** at runtime

```
Barrier b = new Barrier(1);
...
b.AddParticipant(); // Now 2
...
b.RemoveParticipant(); // Back to one
```

Barrier Phasing

- Can supply a *phase action* to be executed after all threads have reached the barrier, but *before* the threads are released to run

```
Barrier barrier = new Barrier(NumThreads, new Action(VerifyBalances));

void DoTransfers(List<TransferRequest> requests) {
    ... // Transfer money
     barrier.SignalAndWait();
    ...
}

void VerifyBalances() { ... }
```

Method is called once all threads are blocked on the barrier

Summary

1. Examine Synchronization strategies
2. Use Wait Handles to control execution of multiple threads
3. Control multiple thread entry to a resource
4. Coordinate thread activity

Working with Synchronized Collections

Xamarin University

Tasks

1. Explore thread-safe collections
2. Use concurrent collections
3. Understand the Producer/Consumer pattern

Collections in .NET

- Standard collections in .NET are not thread safe and must always be synchronized in your code if they are accessed in a read/write fashion by multiple threads

```
object collectionGuard = new object();
List<string> data = new List<string>();

public void Add(string text)
{
    lock (collectionGuard) {
        data.Add(text);
    }
}
```

Thread-safe collections

- .NET 4.5 introduced a set of collections in the namespace `System.Collections.Concurrent` which ensure thread-safety; either by using locks internally, or through the use of lock-free, thread-safe algorithms

`ConcurrentQueue<T>`

Provides a first-in-first-out (FIFO) collection of data

Thread-safe collections

- .NET 4.5 introduced a set of collections in the namespace **`System.Collections.Concurrent`** which ensure thread-safety; either by using locks internally, or through the use of lock-free, thread-safe algorithms

ConcurrentQueue<T>

ConcurrentStack<T>

Provides a last-in-first-out (LIFO) collection of data

Thread-safe collections

- .NET 4.5 introduced a set of collections in the namespace `System.Collections.Concurrent` which ensure thread-safety; either by using locks internally, or through the use of lock-free, thread-safe algorithms

`ConcurrentQueue<T>`

`ConcurrentStack<T>`

`ConcurrentDictionary<K,V>`

Provides a collection of random-access key-value pairs

Thread-safe collections

- .NET 4.5 introduced a set of collections in the namespace **System.Collections.Concurrent** which ensure thread-safety; either by using locks internally, or through the use of lock-free, thread-safe algorithms

ConcurrentQueue<T>

ConcurrentStack<T>

Conc

ConcurrentBag<T>

Provides an unordered collection of objects

Using the concurrent collections

- Because multiple threads might be changing the collection, all retrieval operations are "non-deterministic"

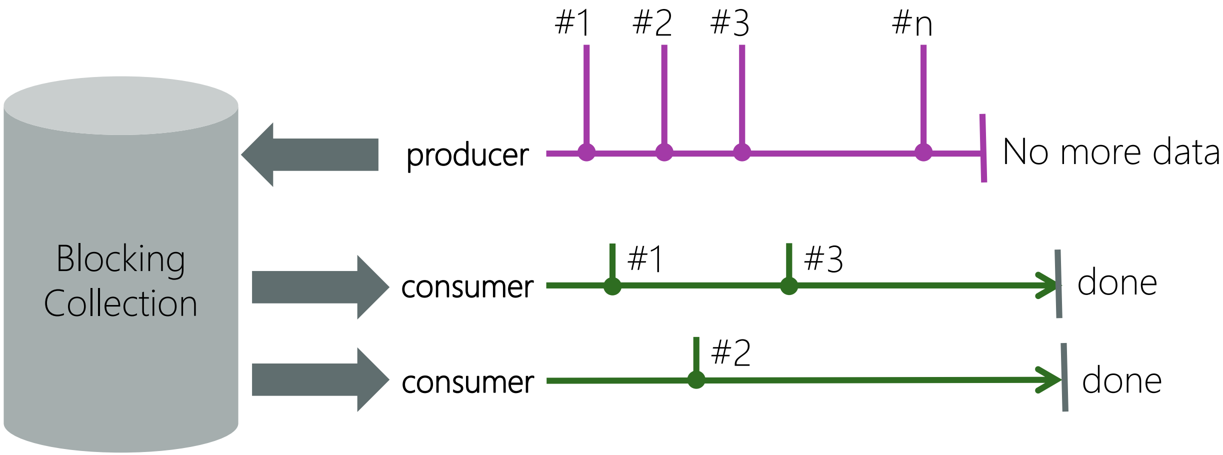
```
ConcurrentQueue<string> data = new ConcurrentQueue<string>();

public string GetNextItem()
{
    string rc = null;
    return (data.TryDequeue(out rc)) ? rc : null;
}
```

No need to take a lock, just try to dequeue value

Producer / Consumer pattern

- New collections also provide support for **producer / consumer** pattern through a wrapper `BlockingCollection<T>`



Producer / Consumer pattern

- **`BlockingCollection`** provides **`IEnumerable`** which automatically terminates when the producer indicates no more values are forthcoming

```
ConcurrentStack<int> stack = new ConcurrentStack<int>();
BlockingCollection<int> work = new BlockingCollection<int>(stack);

void Producer() {
    for (int i = 0; i < 100; i++)
        work.Add(i);
    work.CompleteAdding();
}

void Consumer() {
    foreach (int value in work.GetConsumingEnumerable()) {
        // Process value
    }
}
```

Constructor takes the type of collection to use as the backing storage, defaults to queue

Producer / Consumer pattern

- **`BlockingCollection`** provides **`IEnumerable`** which automatically terminates when the producer indicates no more values are forthcoming

```
ConcurrentStack<int> stack = new ConcurrentStack<int>();
BlockingCollection<int> work = new BlockingCollection<int>(stack);

void Producer() {
    for (int i = 0; i < 100; i++)
        work.Add(i);
    work.CompleteAdding();
}

void Consumer() {
    foreach (int value in work.GetConsumingEnumerable()) {
        // Process value
    }
}
```

Producer adds values into the blocking collection

Producer / Consumer pattern

- **`BlockingCollection`** provides **`IEnumerable`** which automatically terminates when the producer indicates no more values are forthcoming

```
ConcurrentStack<int> stack = new ConcurrentStack<int>();
BlockingCollection<int> work = new BlockingCollection<int>(stack);

void Producer() {
    for (int i = 0; i < 100; i++)
        work.Add(i);
    work.CompleteAdding();
}

void Consumer() {
    foreach (int value in work.GetConsumingEnumerable()) {
        // Process value
    }
}
```

Consumer(s) read from blocking **`IEnumerable`**

Producer / Consumer pattern

- **`BlockingCollection`** provides **`IEnumerable`** which automatically terminates when the producer indicates no more values are forthcoming

```
ConcurrentStack<int> stack = new ConcurrentStack<int>();
BlockingCollection<int> work = new BlockingCollection<int>(stack);

void Producer() {
    for (int i = 0; i < 100;
        work.Add(i);
    work.CompleteAdding();
}

void Consumer() {
    foreach (int value in work.GetConsumingEnumerable()) {
        // Process value
    }
}
```

Producer then indicates when no more data is available, this ends the **`IEnumerable`** loop

Demonstration

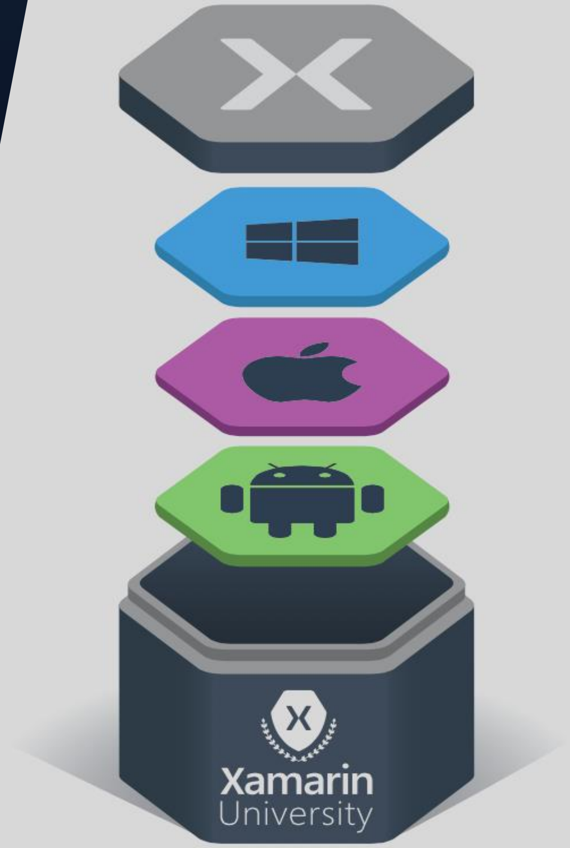
Blocking Collection Demo

Summary

1. Explore thread-safe collections
2. Use concurrent collections
3. Understand the Producer/Consumer pattern

Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile



Microsoft