Programming the .NET Framework 4.0

Module 05 - Threading and Asynchronous Programming

Copyright © SELA Software & Education Labs, Ltd. | 14-18 Baruch Hirsch St., Bnei Brak 51202, Israel | www.selagroup.com

In This Chapter

- Multi-tasking, processes, threads, asynchrony, scheduling
- Asynchronous programming model (APM)
- Thread pool
- Manual threading
- Synchronization
- Task Parallel Library
- Async / Await
- Lab

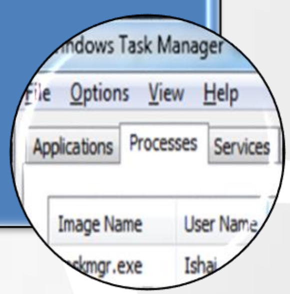
Multi-Tasking and Multi-Processing

- Executing **multiple tasks at once**
- Executing **tasks** on **multiple processors**

Processes and Threads

- **Virtual memory**
- Handle table
- Security token
- **Loaded binaries**

Process



- **Stack**
- Security token
- **Runs code**!

Thread

One Program, Multiple Threads

- Threads can execute at different points of the same code
- Threads can **execute** **simultaneously**

Why Threads?

- Deferred **background work**
- **Parallelization** of work
- Program structure

Why Not Threads?

- **Corruption** of **shared data**
- **Contention** for shared data
- Thread **affinitized** resources
- Program structure

Stack and Heap

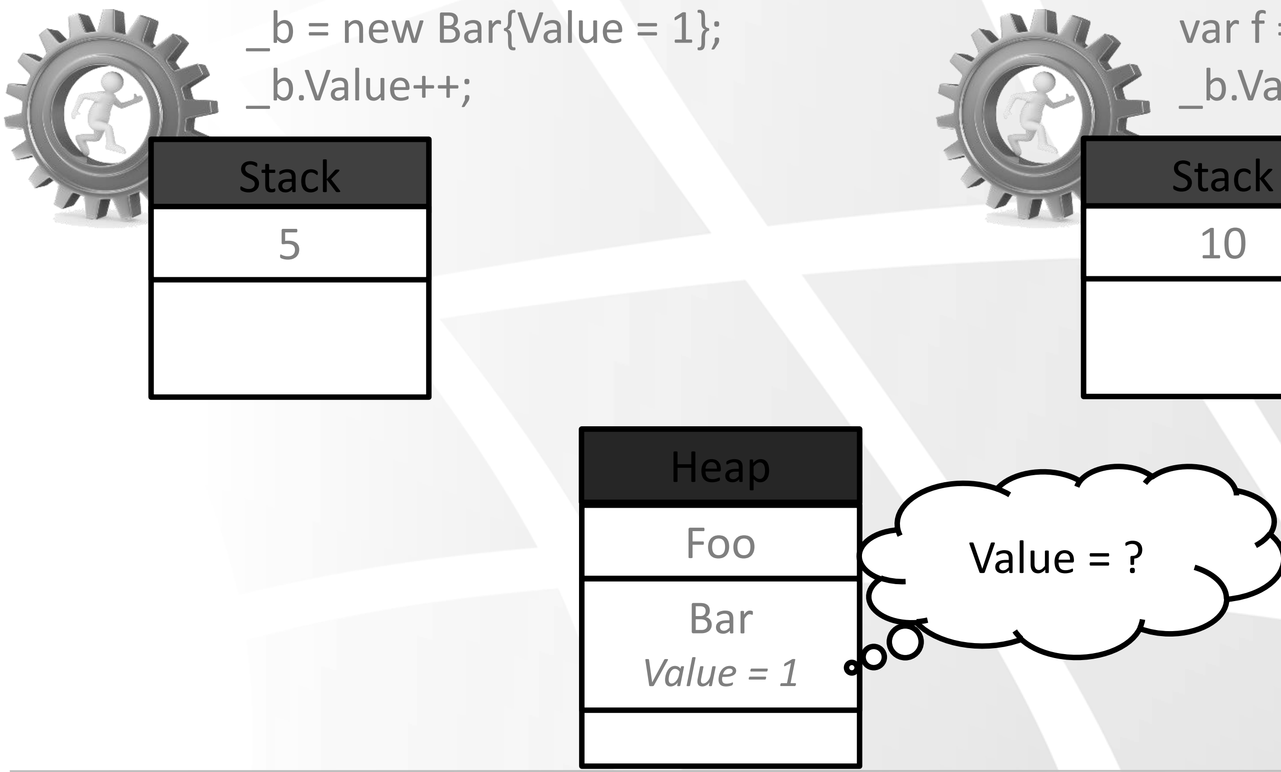
- Each **Thread** is having its own **stack**
- **Heap** is shared among **all threads**

Thread safety

- Stack and Heap

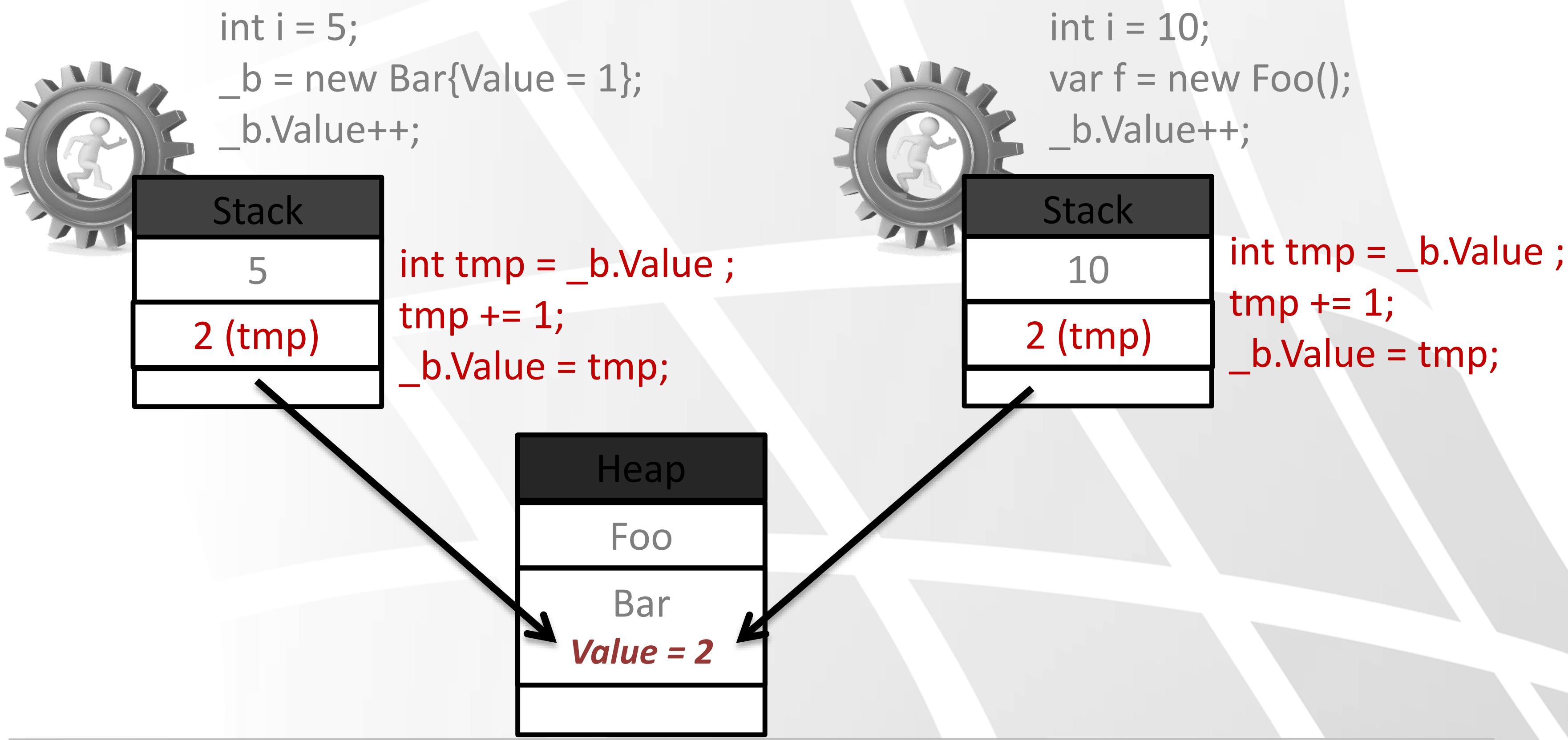
```
int i = 5;
_b = new Bar{Value = 1};
_b.Value++;
```

```
int i = 10;
var f = new Foo();
_b.Value++;
```



Thread safety

- Stack and Heap



Scheduling at a Glance

- A thread has a **priority**
- The **single highest-priority** ready thread always runs
- SMP **scheduling** is **very hard**

- **Starvation**
- **Synchronization** convoys

Amdahl's Law

- If P is the proportion of code that can be parallelized, then the maximum possible speedup (with N processors) is:

$$\boxed{S(N) = \frac{1}{(1-P) + \frac{P}{N}}}$$

- For P=90%, S=10
- For P=50%, S=2

Asynchronous Programming Model

- Do now and wait
- Blocked until operation completes

Synchronous

- Start now, check on later
- Continue working
- Can be notified through a callback

Asynchronous

APM and Files

```
while (true) {
    ar1 = reader.BeginRead(buf1, 0, 8192, null, null);
    while (!ar1.IsCompleted) ...
    if (ar2 != null)
        while (!ar2.IsCompleted) ...
    if ((read = reader.EndRead(ar1)) == 0)
        break;  //No more data to read
    if (ar2 != null)
        writer.EndWrite(ar2);
    Array.Copy(buf1, buf2, read);
   ar2 = writer.BeginWrite(buf2, 0, read, null, null);
}
```

tips

Specific example:

- BeginRead
- EndRead
- BeginWrite
- EndWrite

APM and Files

Demo

APM and Threads

- Threads provide the execution fabric
- .NET delegates provide the asynchrony

- `BeginInvoke, EndInvoke`
- `Invoke`

BeginInvoke and EndInvoke

```
var asyncSieve = new PrimeNumberCalculator(...);
Func<PrimeNumberCalculation, int> asyncSieveCalc =
         asyncSieve.Calculate;
IAsyncResult ar1 = asyncSieveCalc.BeginInvoke(
         PrimeNumberCalculation.Sieve, null, null);
var asyncStandard = new PrimeNumberCalculator(...);
Func<PrimeNumberCalculation, int> asyncStandardCalc =
         asyncStandard.Calculate;
IAsyncResult ar2 = asyncStandardCalc.BeginInvoke(
         PrimeNumberCalculation.Sieve, null, null);
while (!(ar1.IsCompleted && ar2.IsCompleted))
    Thread.Sleep(100);
Console.WriteLine("{0} primes using the sieve",
         asyncSieveCalc.EndInvoke(ar1));
Console.WriteLine("{0} primes using standard method",
         asyncStandardCalc.EndInvoke(ar2));
```

Various Ways to End

- `EndInvoke`
- Poll `IsCompleted`
- Wait for `AsyncWaitHandle`
- Register callback

Various Ways to End (contd.)

```
var calc = new PrimeNumberCalculator(5, 100000);
Func<PrimeNumberCalculation, int> invoker =
         calc.Calculate;

//Poll for IsCompleted property:
IAsyncResult ar = invoker.BeginInvoke(
         PrimeNumberCalculation.Sieve, null, null);
while (!ar.IsCompleted)
    Thread.Sleep(100);
int result = invoker.EndInvoke(ar);

//Wait for the wait handle:
ar = invoker.BeginInvoke(
         PrimeNumberCalculation.Sieve, null, null);
ar.AsyncWaitHandle.WaitOne();
result = invoker.EndInvoke(ar);
```

Maintain State With a Callback

- Easy with closures
  - Or could use AsyncState property

```
//Use callback
ar = invoker.BeginInvoke(
        PrimeNumberCalculation.Sieve, input =>
        {
                result = invoker.EndInvoke(input);
        }, null);
```

Maintain State with AsyncState

```
//Use callback without closures:
Printer printer = new Printer();
ar = invoker.BeginInvoke(PrimeNumberCalculation.Sieve,
         new AsyncCallback(CalculationEnded), printer);

private static void CalculationEnded(IAsyncResult ar)
{
    //We need to retrieve the delegate and the state:
    AsyncResult realAR = (AsyncResult)ar;
    Printer printer = (Printer)ar.AsyncState;
    var invoker =
(Func<PrimeNumberCalculation,int>)realAR.AsyncDelegate;
    //End the operation and print the result:
    int result = invoker.EndInvoke(ar);
    printer.Print(result);
}
```

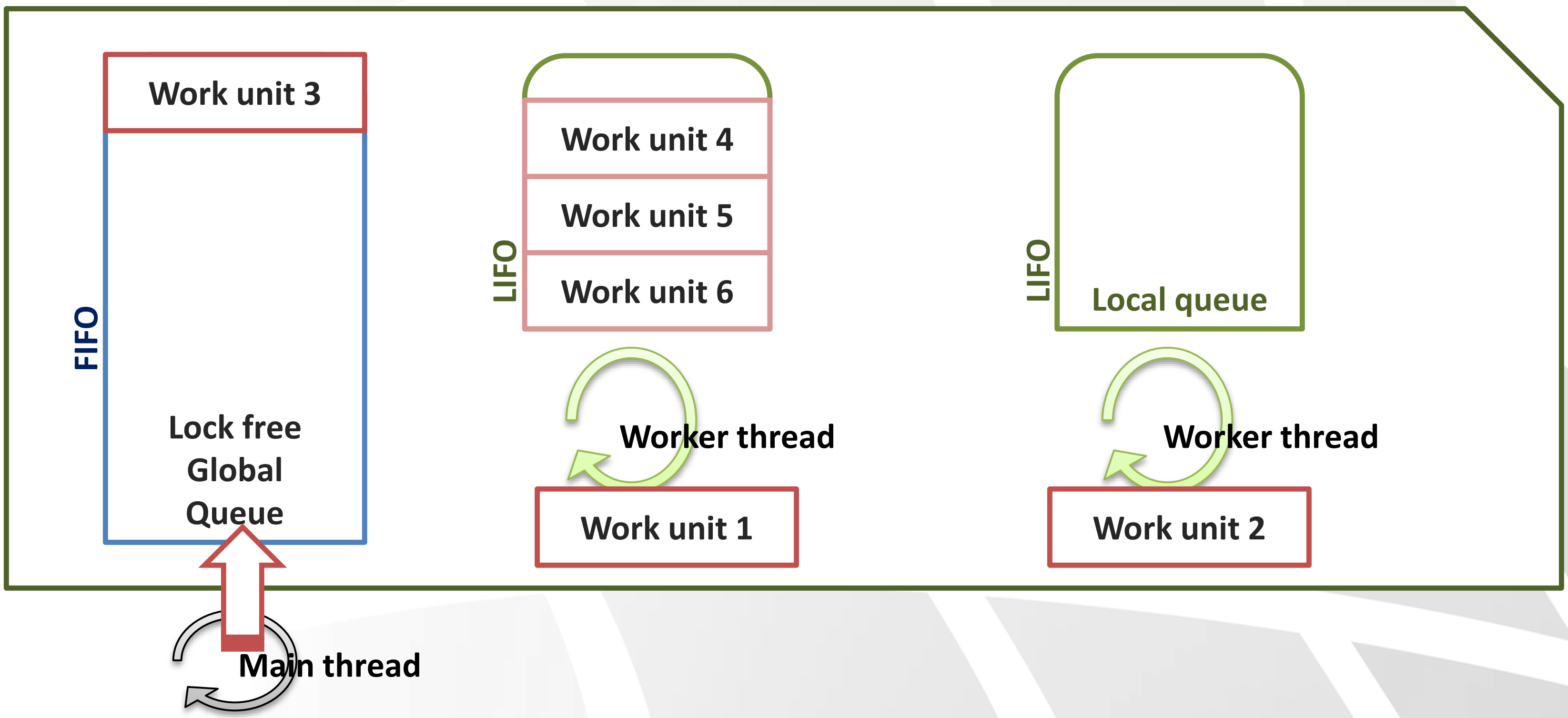
APM

Demo

Queuing Work for Execution

Which threads execute BeginInvoke?

Answer: Thread pool threads



ThreadPool.QueueUserWorkItem

```
class AsyncLogger {
    private readonly StreamWriter _writer;
    public AsyncLogger(string file) {
        _writer = new StreamWriter(file);
    }
    public void WriteLog(string message) {
        _writer.Write(message);
    }
    public void WriteLogAsync(string message) {
        ThreadPool.QueueUserWorkItem(delegate {
            WriteLog(message); });
    }
    public void Close() {
        _writer.Close();
    }
}
```

AsyncLogger

Demo

Manual Threading

- The thread pool manages threads

  So can we?

Not recommended!

- Thread creation overhead, management subtleties

Thread Class

- Thread.Start
- Thread states

```
Thread thread;
thread = new Thread(new ThreadStart(WriteThread));
thread.Start();

private void WriteThread()
{
    while (!stop)
    { ...
    }
}
```

When Does It End?

- Thread.Interrupt
- Thread.Abort
- Thread.Join
- it is not recommended to use Abort or Interrupt

```
stop = true;
thread.Join();
```

Abort vs. Interrupt

When to interrupt and when to abort?

- `ThreadInterruptedException`, `ThreadAbortedException`
- `Thread.ResetAbort`

Inter-Thread Communication

- "Global" variables
- Queues

AsyncLogger2

Demo

Shared Data → Synchronization

- Shared data may become corrupted

```
class Counter
{
    private int _value;

    public int Next() { return ++_value; }

    public int Current { get { return _value; } }
}
```

Shared State Corruption

Demo

Busy Synchronization

- Volatile variables
- Interlocked functions

```
class InterlockedCounter
{
    private int _value;

    public int Next()
    {
        return Interlocked.Increment(ref _value);
    }

    public int Current { get { return _value; } }
}
```

Critical Sections

- Enter critical section
- Do work
- Exit critical section
- Wait
- Wait
- Wait
- …

Thread 1

- Wait
- Wait
- Wait
- Enter critical section
- Do work
- Exit critical section
- …

Thread 2

Monitor and Lock

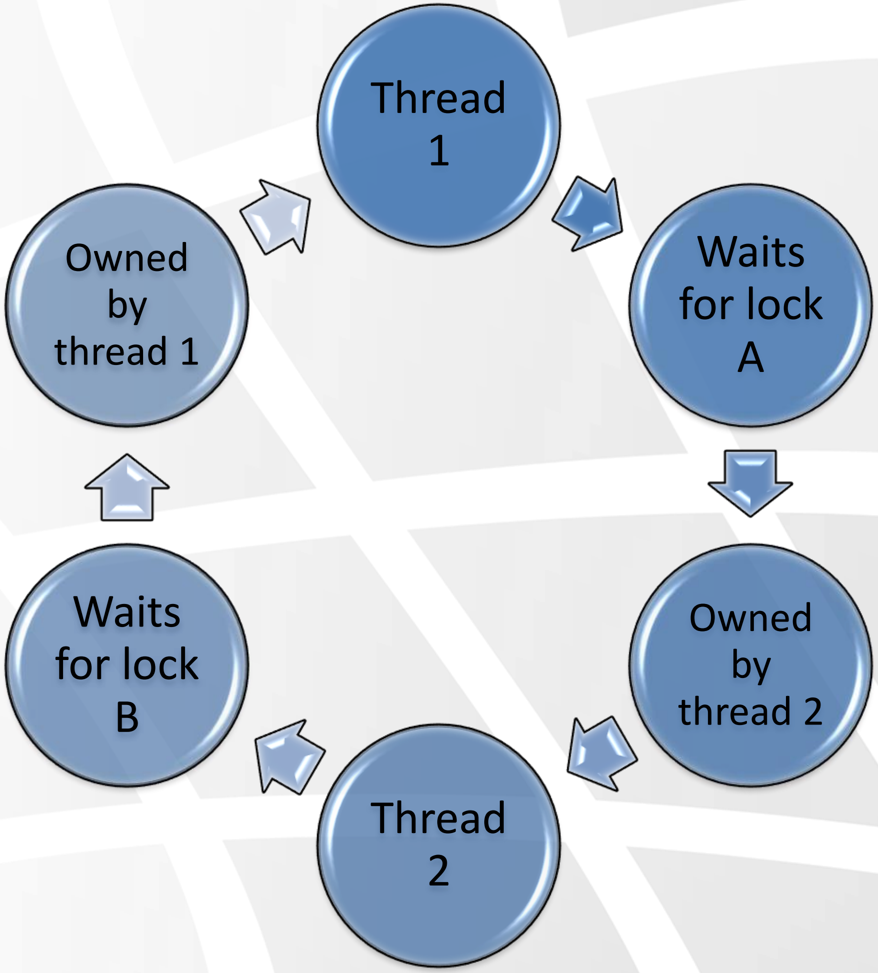
- Monitor.Enter and Monitor.Exit
- The lock keyword also performs the same function

```
class BankAccount {
    public decimal Balance { get; private set; }
    private readonly object _syncRoot = new object();
    public void Deposit(decimal amount) {
        lock (_syncRoot)
            Balance += amount;
    }
    public void Withdraw(decimal amount) {
        lock (_syncRoot)
             Balance -= amount;
     } }
```

Synchronization with *lock*

Demo

Deadlocks


Thread 1
Waits for lock A
Owned by thread 2
Thread 2
Waits for lock B
Owned by thread 1

Deadlocks in Code

```
Counter c1 = new Counter();
Counter c2 = new Counter();
ThreadPool.QueueUserWorkItem(delegate {
    lock (c1) {
        for (int i = 0; i < 100000; ++i) c1.Next();
        lock (c2) {
            for (int i = 0; i < 100000; ++i) c2.Next();
        }
    } });
ThreadPool.QueueUserWorkItem(delegate {
    lock (c2) {
        for (int i = 0; i < 100000; ++i) c2.Next();
        lock (c1) {
           for (int i = 0; i < 100000; ++i) c1.Next();
        }
    } });
```

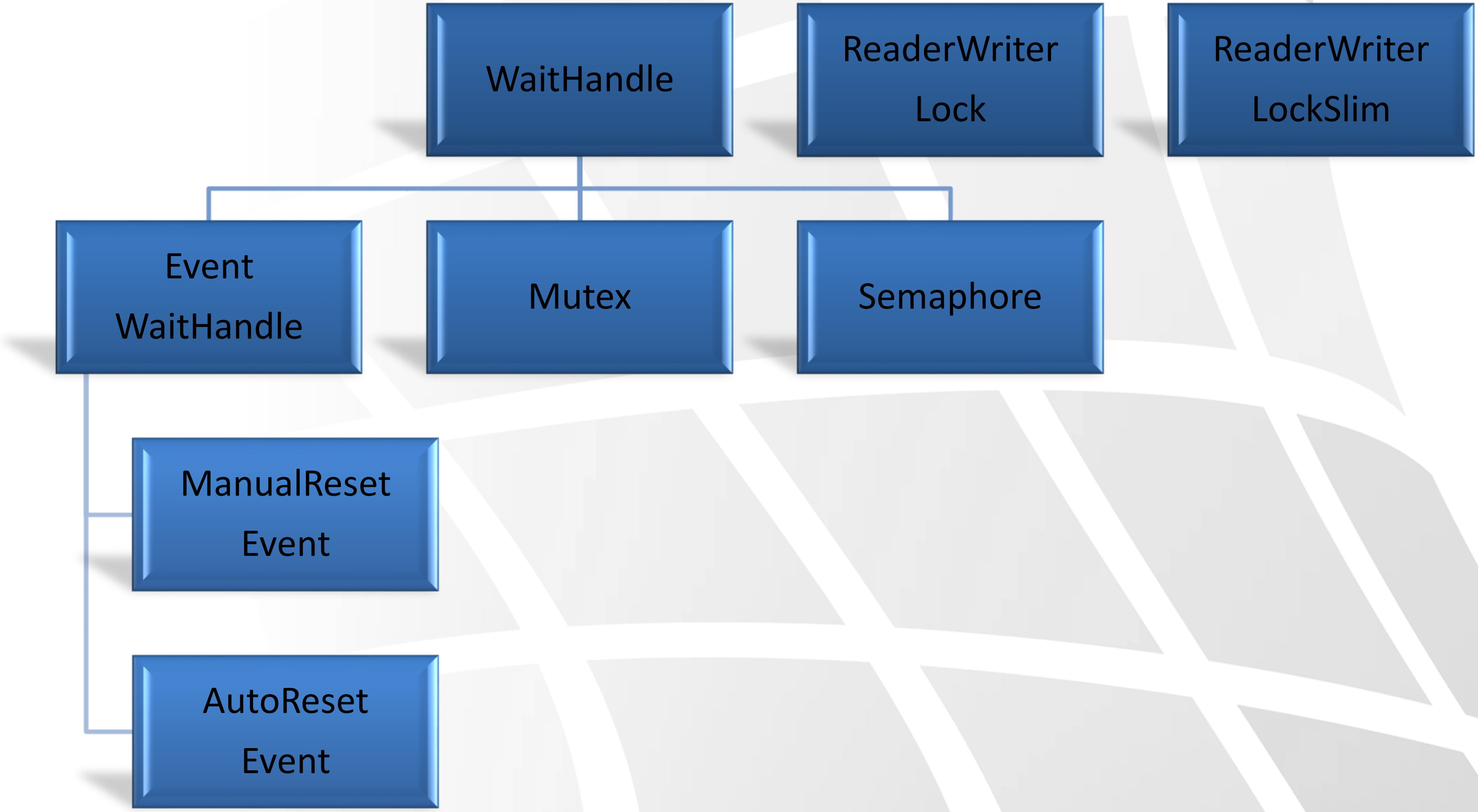
Deadlocks

Demo

Pulse, PulseAll, Wait, WaitAll

```
class WorkQueue<T> : Queue<T> {
    private readonly object _sync = new object();
    public new void Enqueue(T item) {
        Monitor.Enter(_sync);
        try {
            base.Enqueue(item);
            Monitor.Pulse(_sync);
        } finally { Monitor.Exit(_sync); }
    }
    public new T Dequeue() {
        Monitor.Enter(_sync);
        try {
           while (base.Count == 0) Monitor.Wait(_sync);
           return base.Dequeue();
         } finally { Monitor.Exit(_sync); }
    } }
```

WaitHandle Synchronization



Using Events for Synchronization

```
public static void DoAndLetMeKnow(
            Action action, EventWaitHandle @event)
{
    ThreadPool.QueueUserWorkItem(delegate
    {
        action();
        @event.Set();
    });
}
```

Advanced Synchronization

Demo

TPL – Task Parallel Library

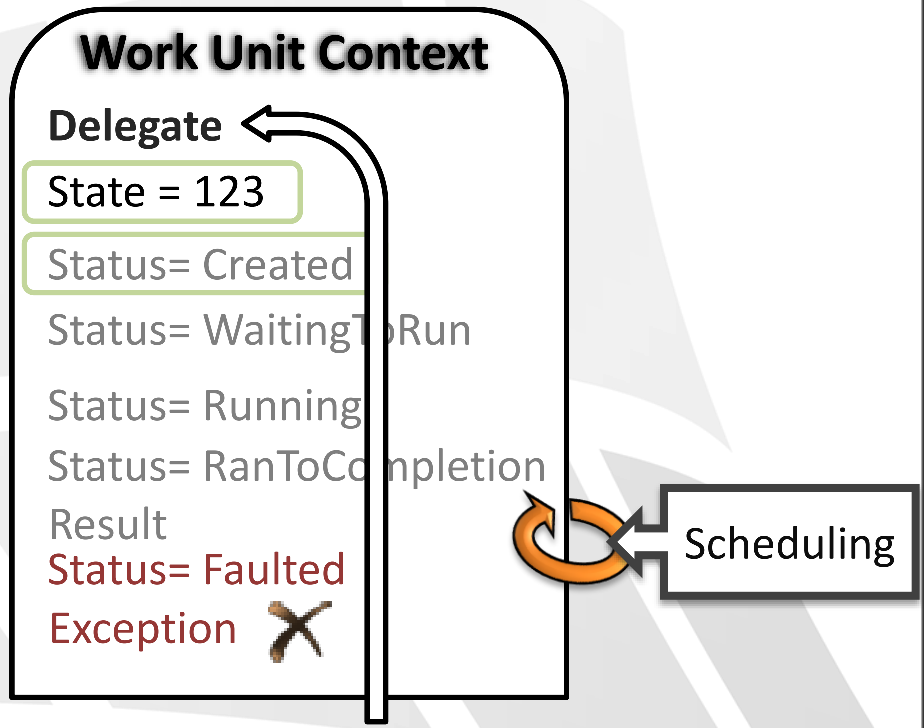
- By default execute on the Thread Pool
- Parallel execution APIs
- New synchronization types
- Lock-free (and thread-safe) collections
- Scheduling built around the Task type
  - Tasks represent an executing work item

Why Use the TPL?

- `ThreadPool.QueueUserWorkItem` was designed for simple fire and forget parallelism
- Tasks were designed to handle a broader scope which includes:
  - Cancellation
  - Wait
  - Result acquisition
  - Exception handling
  - …and more

Task

- A **task** encapsulates the work item data
  - **Execution status**
  - **State**
  - **Result**
  - **Exception**
  - Wait
  - Cancellation (covered later)
  - Continuation (covered later)




```
Task t = new Task ((state) => Console.WriteLine(“”), 123);
t.Start();
```

Scheduling Tasks

- Task.Factory.StartNew or Task.Run
- Wait for completion
- Obtain result

```
Task t = Task.Run(() => Console.WriteLine(42));
//Other work here...
t.Wait();

Task<int> t2 = Task.Run(() => { return 42; });
//Other work here...
Task.WriteLine(t.Result);
```

Exception Propagation

- Tasks may throw exceptions
- Rethrown when calling Task.Wait or Task.Result
- AggregateException

```
Task t = Task.Run(() => throw new ApplicationException());
try {
   t.Wait();
} catch (AggregateException ex) {
   foreach (Exception e in ex.InnerExceptions) ...
}
```

Decomposing Work With Tasks

Demo

Continuations

- Continuations chain tasks one after another
- Better than Wait()

```
Task<int> tskA = Task.Factory.StartNew(() => {return 42;});
Task tskB = tskA.ContinueWith(prevTask => {
  if (prevTask.Exception != null)
    Console.WriteLine("Failed");
  else
    Console.WriteLine("Returned {0}", prevTask.Result);
});
```

Continuations Can Get Messy...

```
private void orderButton_Click(...) {
  string filename = App.CatalogFileName;
  Task<Catalog> tc = OpenCatalogAsync(filename);
  tc.ContinueWith(_ => {
    Catalog c = tc.Result;
    Task<Inventory> ti = c.GetInventoryAsync();
    ti.ContinueWith(_ => {
      Inventory i = ti.Result;
      Task<bool> tb = i.IsInStockAsync(product: "Soap");
      tb.ContinueWith(...);
  });
}
```

C# 5.0 Async and Await

- async methods
- await keyword
- Compiler-generated continuations

```
async Task Execute()
{
  Console.WriteLine("run on calling thread");
  await Task.Factory.StartNew(() => Thread.Sleep(1000));
   Console.WriteLine("run on callback thread");
}
```

Thread Id = 1

Thread Id = 2

Thread Id = 3

Data Parallelism

What is the source of parallelism?

- Task (explicit) parallelism – **code**
- Data (implicit) parallelism – **data**

Parallel.For and Parallel.ForEach

```
// Process the range in parallel and wait for completion
Parallel.For(0, 1000, i =>
{
  socket[i].Send("Ping");
});

// Process the collection in parallel and wait for completion
Parallel.ForEach(blogUrls, url =>
{
  webClient.DownloadUrl(url);
});
```

Concurrent Collections

- `System.Collections.Concurrent`
- Thread-safe and efficient

- `ConcurrentBag`
- `ConcurrentQueue`
- `ConcurrentDictionary`

Parallel.For and Concurrent Collections

Demo

Thread-Safe Resource
Parallelizing Work

Lab

Summary

- Multi-tasking, processes, threads, asynchrony, scheduling
- Asynchronous programming model (APM)
- Thread pool
- Manual threading
- Synchronization
- Task Parallel Library
- Lab

Questions ?