



# INTRODUCTION TO ASYNCHRONOUS CODE IN .NET

BILL DINGER

Solutions Architect | @adazlian

bill.dinger@vml.com

<https://github.com/BillDinger/AsyncTalk>

# WHAT PROBLEM ARE WE SOLVING HERE?

THROUGHPUT OF  
AN APPLICATION

RESPONSIVENESS  
OF THE  
APPLICATION

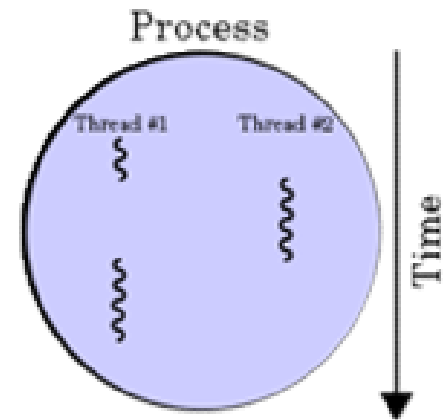
SPEED OF THE  
APPLICATION

# SPEED OF THE APPLICATION

- **TPL** (Task Parallel Library)
  - Introduced in .NET 4.0
- **Async/Await**
  - Introduced in .NET 4.5
- **PLINQ** (Parallel LINQ)
  - Introduced in .NET 4.0

# THE BASICS OF THREADING

- A **thread** is the smallest series of computer instructions that can be managed by a scheduler (typically the operating system).
- A **process** comprises one or more threads.
- A CPU core can typically execute only one thread simultaneously.



# DEMO OF A SINGLE THREADED APP VERSUS A MULTITHREAD APP

# TASK PARALLEL LIBRARY DESIGN GOALS

Make parallel programming easier on developers.

- Manually maintaining mutexes and thread primitives is not fun.

Make multithreading more efficient.

- Thread creation, destruction and synchronization are difficult to manage correctly and efficiently.

Give developers control to easily schedule task workflows.

- Allow continuation, result and task scheduling.

# TASK PARALLEL LIBRARY

- Introduced with .NET 4.0, uses tasks to schedule operations that are performed on a background thread pool managed by the .NET CLR
- Removes complexities of managing and synchronizing threads
- Supports exception handling, continuation and return values
- Most suitable to CPU-intensive operations that can be decomposed
- Automatically scales itself to meet processing demand and patterns

# .NET THREAD POOL BASICS

- When the CLR starts an application, it creates a pool of threads an application can use.
- Creating and destroying threads is computationally intensive. In addition, a thread has a minimum memory size of 1MB.
- .NET will manage this thread pool for you, queuing tasks as necessary and assigning threads to process them.



# TASKS

## BASICS

- Tasks are queued onto the .NET thread pool and executed in parallel.
- Tasks are not guaranteed to execute in any particular order.
- Tasks support exception handling, cancellation, workflow, returning results, child tasks and continuation.
- Task constructors take a `System.Action` (usually as a lambda expression) to execute.

# TASKS

## BASICS

A small unit of work takes a System.Action delegate. At its most basic:

```
public void RunATask()
{
    Task.Run(() => { Console.WriteLine("I'm a task!"); });
}
```

It can also be created as a Task to run later:

```
public void StartATask()
{
    int i = 0;
    var aTask = new Task(() =>
    {
        i = 17 + 23;
    });
    aTask.Start();
    Console.WriteLine(i);
}
```

# TASKS

## BASICS

What are Tasks?

- Tasks are small discrete blocks of code that execute asynchronously.
- They can be created by using `new Task<T>`, `Task.Run` or `Task.Factory.StartNew()`.
- Best practice uses the `Task.Factory.StartNew` pattern that creates and starts the task in one call, which is slightly more efficient.

```
public void RunATaskFactory()
{
    var someTask =
        Task.Factory.StartNew(
            () => { return "I'm a task factory!"; }
        );
    Console.WriteLine(someTask.Result);
}
```

# TASKS

## STARTING AND WAITING FOR MULTIPLE

Use WaitAll to start a group of tasks and block the calling thread until finished:

```
public void RunABunchOfTasks()
{
    var watch = Stopwatch.StartNew();
    var taskA = Task.Factory.StartNew(() => SpinWait(11));
    var taskB = Task.Factory.StartNew(() => SpinWait(4));
    var taskC = Task.Factory.StartNew(() => SpinWait(22));
    var taskD = Task.Factory.StartNew(() => SpinWait(7));
    var tasks = new[] { taskA, taskB, taskC, taskD };
    Task.WaitAll(tasks);
    watch.Stop();
    Console.WriteLine($"Finished in {watch.Elapsed.Seconds}");
}
```

# TASKS

## STARTING AND WAITING FOR MULTIPLE

Use WhenAll to start a group of tasks and await the result.

```
await Examples.AwaitRunABunchOfTasks();
```

```
public async Task AwaitRunABunchOfTasks()
{
    var taskA = Task.Factory.StartNew(() => SpinWait(11));
    var taskB = Task.Factory.StartNew(() => SpinWait(4));
    var taskC = Task.Factory.StartNew(() => SpinWait(22));
    var taskD = Task.Factory.StartNew(() => SpinWait(7));
    var tasks = new[] { taskA, taskB, taskC, taskD };
    await Task.WhenAll(tasks);
}
```

# TASKS

## RETURNING VALUES

Closures can be used to return values, but beware that this is not inherently thread-safe! The result below might return 117 or it might return 200.

```
public void UnsafeClosures()
{
    int i = 0;
    var taskA = new Task(() => i = 17 + 100);
    var taskB = new Task(() => i = 100 + 100);
    taskA.Start();
    taskB.Start();
    Console.WriteLine(i);
}
```

# TASKS

## RETURNING VALUES

Best practice — use the `Result<T>` task property to view the result.

```
public void ReturnAValue()
{
    var someTask = Task.Run(() => { return "a string"; });
    Console.WriteLine(someTask.Result);
}

public void ReturnTask()
{
    var taskA = new Task<int>(() => 17 + 100);
    var taskB = new Task<int>(() => 100 + 100);
    taskA.Start();
    taskB.Start();
    Console.WriteLine($"TaskA: {taskA} ");
    Console.WriteLine($"TaskB: {taskB} ");
}
```

# TASKS

## RETURNING VALUES

To return multiple results and process them, we can use `WhenAny`. This allows us to observe the first processed result off the stack.

```
public void ReturnResultsFromMany()
{
    var taskA = Task.Factory.StartNew(Somework);
    var taskB = Task.Factory.StartNew(Somework);
    var taskC = Task.Factory.StartNew(Somework);
    var taskD = Task.Factory.StartNew(Somework);

    var tasks = new List<Task<int>>() { taskA, taskB, taskC, taskD };
    while (tasks.Count > 0)
    {
        var completedIndex = Task.WaitAny(tasks.ToArray());
        var completedTask = tasks[completedIndex];
        Console.WriteLine(completedTask.Result);
        tasks.RemoveAt(completedIndex);
    }
}
```



# TASKS

## ADDING WORKFLOW

TPL natively supports continuation tasks that run after another task.

```
public void ContinueTask()
{
    var task =
        Task.Factory.StartNew(() => 100 + 30)
            .ContinueWith(antecedent => antecedent.Result + 17);
    Console.WriteLine(task.Result);
}
```

# TASKS

## ADDING WORKFLOW

ContinueWith has enumeration options (TaskContinuationOptions) that let you specify under what conditions to run it (i.e., the task has “faulted”).

```
public void ContinueTaskWithOptions()
{
    Task.Factory.StartNew(() => throw new Exception("Stuff"))
        .ContinueWith(
            antecedent =>
                Console.WriteLine(antecedent.Exception.Flatten().InnerException),
            TaskContinuationOptions.OnlyOnFaulted);
}
```

# TASKS

## HANDLING CANCELLATION

Cancellation is a cooperative model — the task being canceled must support it.

Tasks monitor tokens; if cancellation is detected, tasks clean up and throw exceptions. This will set the task status to canceled.

```
private void CheckIfCancel(CancellationToken ct)
{
    while (ct.IsCancellationRequested == false)
    {
        if (ct.IsCancellationRequested)
        {
            ct.ThrowIfCancellationRequested();
        }
        var waitTime = Rando.Next(0, 10);
        Console.WriteLine($"Not Cancelled waiting {waitTime}");
        Thread.Sleep(Rando.Next(0, 10));
    }
}
```

# TASKS

## HANDLING CANCELLATION

We can cancel multiple tasks by deriving from a single CancellationTokenSource.

```
public void TaskCancellation()
{
    var cts = new CancellationTokensource();
    var taskA = Task.Factory.StartNew(SomeWork, cts.Token);
    var taskB = Task.Factory.StartNew(SomeWork, cts.Token);
    Task.WaitAll(new Task[] { taskA, taskB });
    cts.Cancel();
}
```

# TASKS

## HANDLING ERRORS

In general, unhandled exceptions are propagated back to the calling thread when waiting on tasks or checking their results.

```
public void UnhandledExceptions()
{
    var taskA =
        Task.Factory.StartNew(() => throw new Exception("Stuff"));
    var taskB =
        Task.Factory.StartNew(() => throw new Exception("new"));
    var tasks = new []{taskA, taskB};

    // will bubble up exception here:
    Task.WaitAll(tasks);
}
```

# TASKS

## HANDLING ERRORS

Tasks will bubble up an AggregateException, which contains a list of all the exceptions.

```
public void UnhandledExceptions()
{
    var taskA =
        Task.Factory.StartNew(() => throw new Exception("Stuff"));
    var taskB =
        Task.Factory.StartNew(() => throw new Exception("new"));
    var tasks = new []{taskA, taskB};

    // will bubble up exception here:
    Task.WaitAll(tasks);
}
```

# DEADLOCKS OH MY!

## DEADLOCK

```
[Route("deadlock")]
```

```
[HttpGet]
```

0 references | Bill Dinger, 22 minutes ago | 1 author, 1 change

```
public string Deadlock()
{
    // The following code WILL deadlock.
    var result = GetHttpResultAsync().Result;
    return result;
}
```

# CONTEXT IS KING

## SYNCHRONIZATIONCONTEXT

Introduced in .NET 2.0 the `synchronizationContext` is a way for .NET to dispatch work to other threads and maintain a count of outstanding work. Once the work is complete the response will be returned to the user.

When a task is created it “captures” the `synchronizationContext` on creation and then passes control back to the original thread. When the task finishes it attempts to hand control back to the original thread however the controlling thread is waiting for the task to finish before it'll accept any additional work.



# AVOIDING DEADLOCKS

## AVOIDANCE

Deadlocks are possible – **never** wrap asynchronous code in synchronous. The only exception of note being the main thread of a console application.

- Use ConfigureAwait(False) in library code

2 references | Bill Dinger, 98 days ago | 1 author, 1 change

```
private async Task<string> GetHttpResultAsync()  
{  
    using (var httpClient = new HttpClient())  
    {  
        using (var result =  
            await httpClient.GetAsync(testUrl))  
        {  
            var content = await result.Content.ReadAsStringAsync().ConfigureAwait(false);  
            return content;  
        }  
    }  
}
```

- Use async all the way down.

# ASYNC/AWAIT

## BASICS

Use `async` to mark a method as awaitable. By convention, the method name should end in `async`.

- Awaiting a method yields control back to the calling thread
- Reliant upon tasks
- Built-in support on most popular frameworks (EntityFramework, StreamReader classes, HttpClient, Web API, etc.)
- Use for I/O heavy code

# ASYNCAWAIT

## HOW TO MAKE A METHOD *AWAITABLE*

Your method signature returns a `Task<T>` and you await one resource in your method body. You then return type `T`.

```
public async Task<int> GetAReturnCodeAsync()
{
    using (var httpclient = new HttpClient())
    {
        using (var result =
            await httpclient.GetAsync("https://detroitcode.amegala.com"))
        {
            return (int)result.StatusCode;
        }
    }
}
```

# ASYNCAWAIT

## HOW TO MAKE A METHOD *AWAITABLE*

Your method signature returns a task and you await one or more resources. You do not specify a return value.

```
public async Task GetAWebsiteContentAsync()
{
    using (var httpClient = new HttpClient())
    {
        using (var result =
            await httpClient.GetAsync("https://detroitcode.amegala.com"))
        {
            // implicitly returned:
            await result.Content.ReadAsStringAsync();
        }
    }
}
```

# ASYNC/AWAIT

## HOW TO MAKE A METHOD *AWAITABLE*

Your method signature returns void and you await a task. Only use for event handlers.

```
public async void GetAWebsiteOnClick(object sender, EventArgs e)
{
    await GetAWebsiteAsync();
}
```

# ASYNC/AWAIT

## BEHIND THE SCENES

- Async/Await doesn't cause allocation of a new thread; instead, work is handed off to a background thread while we wait for the results.
- Because of this, it's best to use Async/Await when dealing with I/O heavy tasks in client and server applications.
- Keep using TPL and PLINQ for CPU-intensive tasks.

# ASYNC/AWAIT

## DOWN THE ENTIRE STACK

Do not mix blocking and asynchronous code, because it can lead to deadlocks.

- Exception: console main windows

Avoid Async Void

- Exception: event handlers

Configure context: Use `ConfigureAwait(False)` when possible

- Exception: context is needed

# ASYNC/AWAIT: DEMO IN A WEB APP



# TECHNIQUE #3

## PARALLEL LINQ (PLINQ)

- Added as part of .NET 3.5
- As easy as calling `.AsParallel()` on the data source in LINQ methods:

```
public void PlinqExample()
{
    var range = Enumerable.Range(1, 10000);
    var avg =
        (from x in range.AsParallel() select x).Average();
    Console.WriteLine(avg);
}
```

# PLINQ

## BASICS

- Most functionality is contained in the `System.Linq.ParallelEnumerable` namespace.
- The opt-in model is only used when invoked on the data source using `.AsParallel` keywords.
- At runtime, PLINQ will analyze the source and decide if its safe to parallelize.
- By default, PLINQ will use as many threads as cores that exist on the machine. We can override this by using `WithDegreeOfParallelism()`.

```
public void DegreeOfParallelismExample()
{
    var range = Enumerable.Range(1, 10000);
    var avg =
        (from x in
            range.AsParallel().WithDegreeOfParallelism(10)
            select x).Average();
    Console.WriteLine(avg);
}
```

# PLINQ

## CANCELLATION

PLINQ uses cancellation tokens as do other parts of the threading namespace.

```
public void CancellationExample()
{
    var cts = new CancellationTokenSource();
    var token = cts.Token;
    var range = Enumerable.Range(1, 10000);
    var avg =
        (from x in
            range.AsParallel().WithCancellation(token)
            select x).Average();
    cts.Cancel();
    Console.WriteLine(avg);
}
```

# PLINQ

## PITFALLS

- Don't assume parallel is always faster.
- Avoid reading/writing to shared variables (or use concurrent collections).
- Avoid calls to things that aren't thread-safe.
- Prefer ForAll to ForEach whenever possible.

# PLINQ

## FORALL

ForAll iterates over a collection as each thread completes, instead of iterating sequentially like ForEach.

```
public void ForAllExample()
{
    var range = Enumerable.Range(1, 10000);
    var query = from num in range.AsParallel()
                where num % 10 == 0
                select num;

    var bag = new ConcurrentBag<int>();
    query.ForAll(e => bag.Add(e));
}
```

# CONCURRENT COLLECTIONS

Most .NET collections (arrays, lists, dictionarys) aren't thread-safe. Microsoft introduced the `System.Collections.Concurrent` namespace to offer thread-safe collections for use.

Five main collection types are provided:

- `ConcurrentBag<T>`
- `ConcurrentStack<T>`
- `ConcurrentQueue<T>`
- `ConcurrentDictionary<Tkey, TValue>`
- `BlockingCollection<T>`

QUESTIONS?

# RESOURCES

Patterns of Parallel Programming in C#/VB.NET

<https://www.microsoft.com/en-us/download/details.aspx?id=19222>

MSDN Documentation on TPL

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl>

Deadlocks in TPL & Async/Await

<https://everydaylifein.net/netframework/task-parallel-library-taskscheduler-deadlocks-threads.html>

Deadlocks in WPF

<https://blogs.msdn.microsoft.com/pfxteam/2011/01/13/await-and-ui-and-deadlocks-oh-my/>



# RESOURCES CONT.

Async Best Practices for C# & Visual Basic

<https://channel9.msdn.com/Events/TechEd/NorthAmerica/2014/DEV-B362>

Async/Await Best Practices

<https://msdn.microsoft.com/en-us/magazine/jj991977.aspx?f=255&MSPPError=-2147217396>

Lambda Expressions in PLINQ/TPL

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/lambda-expressions-in-plinq-and-tpl>

Async or Not?

<https://visualstudiomagazine.com/Blogs/Tool-Tracker/2014/07/To-Sync-or-Async.aspx>

Synchronization Context

<https://msdn.microsoft.com/en-us/magazine/gg598924.aspx>

THANK YOU.

