Asynchronous Programming Techniques & Best Practices

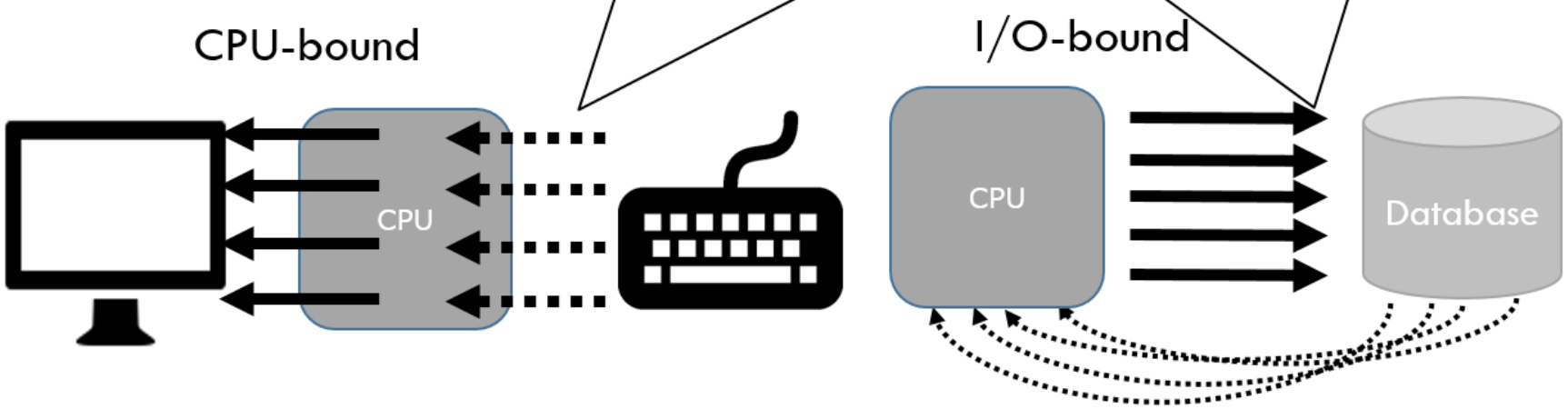
BY RICCARDO TERRELL - @RIKACE

Is it CPU-bound, or I/O-bound?

Asynchronous for I/O Bound

The CPU-bound computations receive input from the keyboard to do some work, and then print the result to the screen. In a single core machine, each computation mast be completed before proceed to the next one.

The I/O-bound computations are executed independently from the CPU, and the operation is done elsewhere. In this case several asynchronous database calls are executing simultaneously. Later a notification will inform the caller when the operation complete *(Callback)*.



Threadpool

Principles

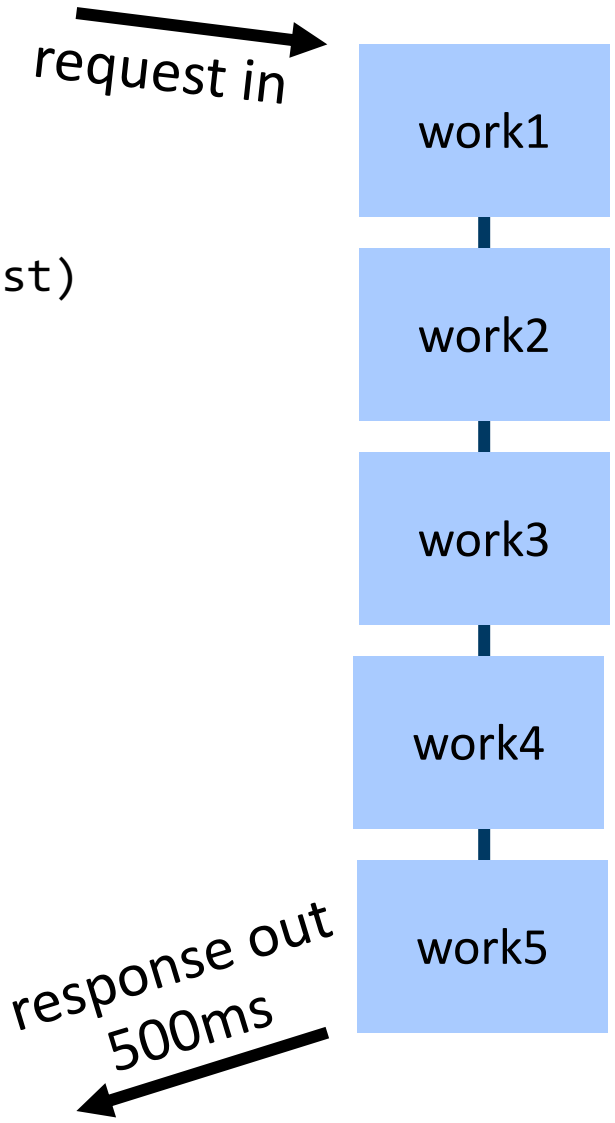
- CPU-bound work means things like: LINQ-over-objects, or big iterations, or computational inner loops.
- Parallel.ForEach and Task.Run are a good way to put CPU-bound work onto the thread pool.
- Thread pool will gradually feel out how many threads are needed to make best progress.
- Use of threads will *never* increase throughput on a machine that's under load.

Guidance

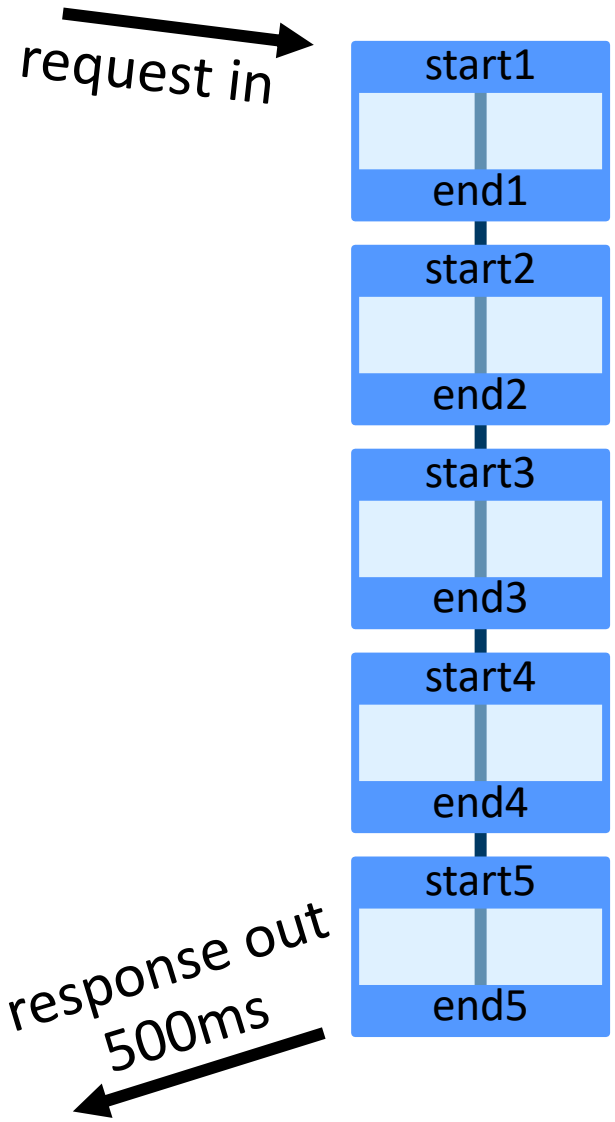
- For IO-bound "work", use await rather than background threads.
- For CPU-bound work, consider using background threads via Parallel.ForEach or Task.Run, unless you're writing a library, or scalable server-side code.

Threadpool

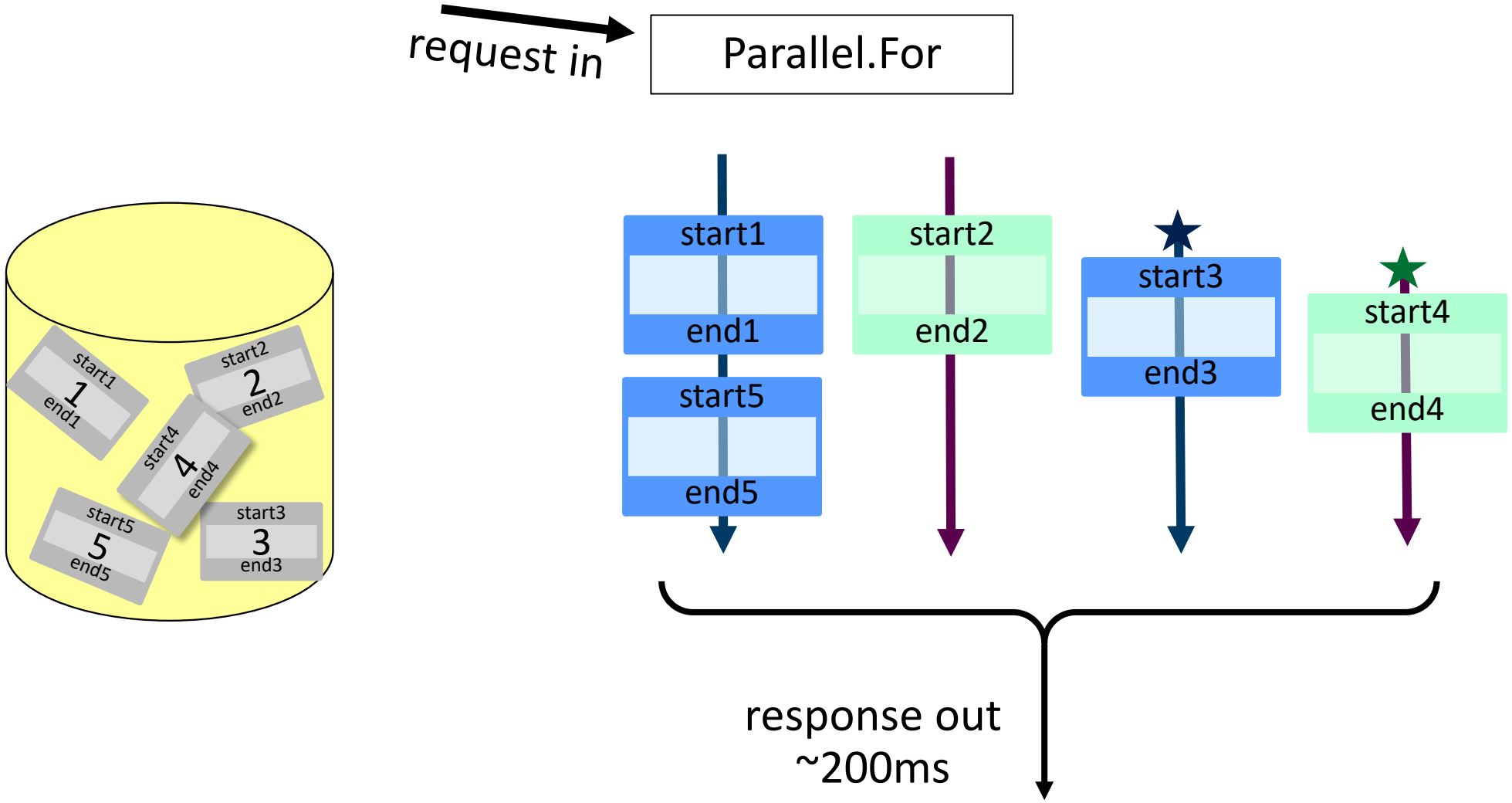
```
// table1.DataSource = LoadHousesSequentially(1,5);
// table1.DataBind();

public List<House> LoadHousesSequentially(int first, int last)
{
    var loadedHouses = new List<House>();

    for (int i = first; i <= last; i++) {
        House house = House.Deserialize(i);
        loadedHouses.Add(house);
    }

    return loadedHouses;
}
```



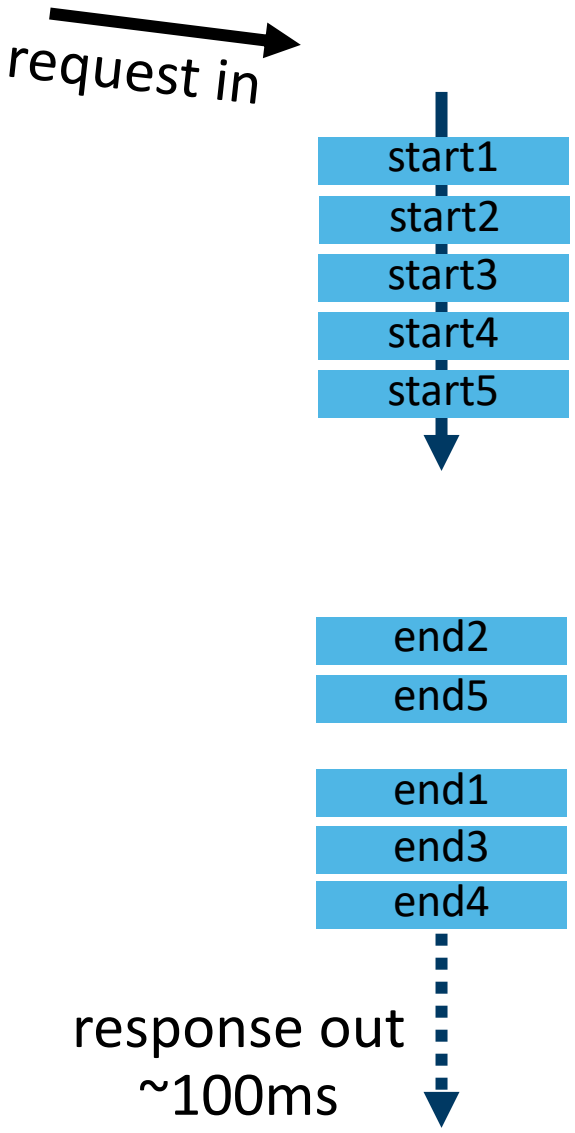
Threadpool


request in
start1
end1
start2
end2
start3
end3
start4
end4
start5
end5
response out
500ms

Threadpool


request in
Parallel.For
start1
1
end1
start2
2
end2
start4
4
end4
start5
5
end5
start3
3
end3
start1
end1
start5
end5
start2
end2
start3
end3
start4
end4
response out
~200ms

Threadpool


request in
start1
start2
start3
start4
start5
end2
end5
end1
end3
end4
response out
~100ms

Threadpool

```
// table1.DataSource = await LoadHousesAsync(1,5);
// table1.DataBind();

public async Task<List<House>> LoadHousesAsync(int first, int last)
{
    var tasks = new List<Task<House>>();

    for (int i = first; i <= last; i++)
    {
        Task<House> t = House.LoadFromDatabaseAsync(i);
        tasks.Add(t);
    }

    House[] loadedHouses = await Task.WhenAll(tasks);
    return loadedHouses.ToList();
}
```

Synchronous code - What is the problem?

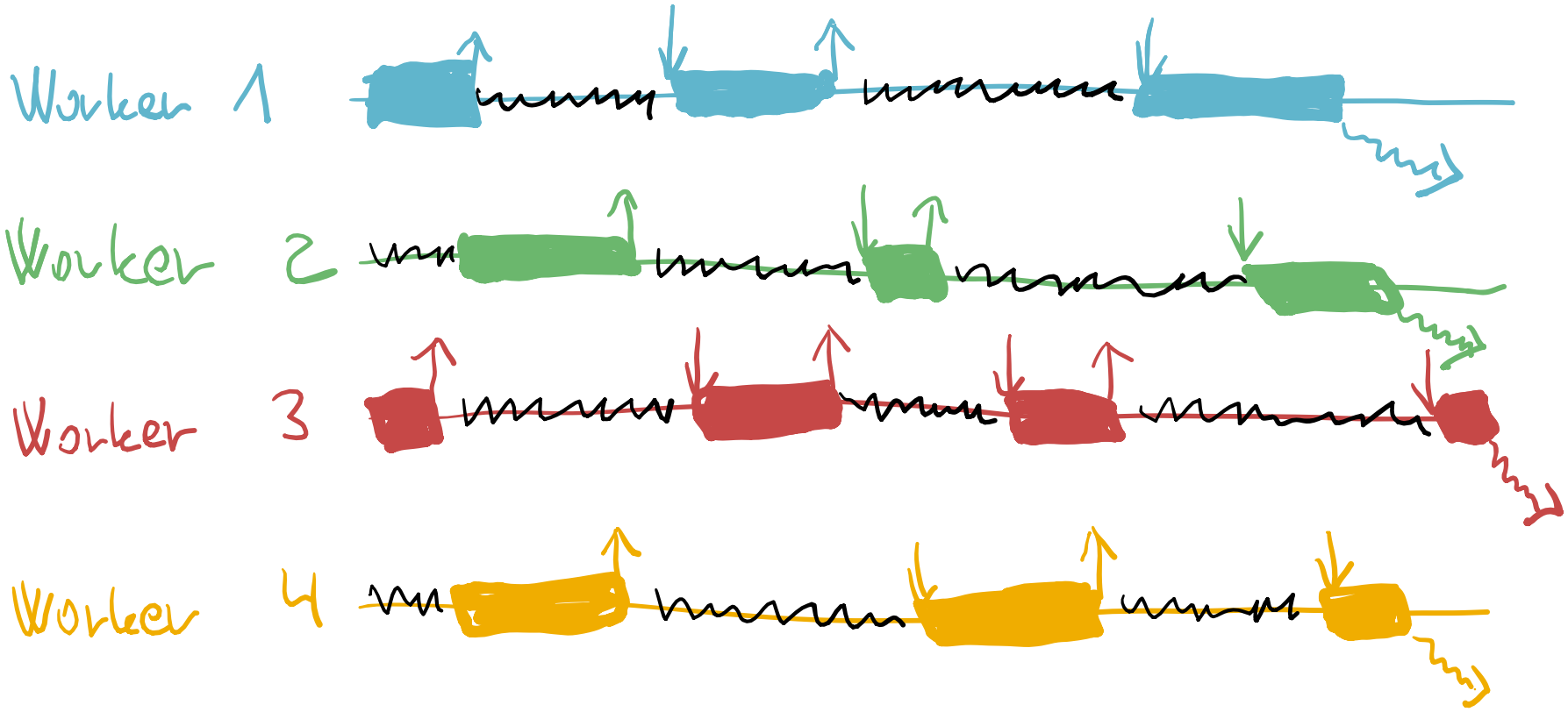
Synchronous I/O or user-interface code

```
let wc = new WebClient()
let data = wc.DownloadData(url)
outputStream.Write(data, 0, data.Length)
```

Blocks thread while waiting

- Does not scale – cannot create 10000 threads
- Blocking user interface – when run on GUI thread
- Simple to write – loops, exception handling etc.

Synchronous



(A)synchronous code

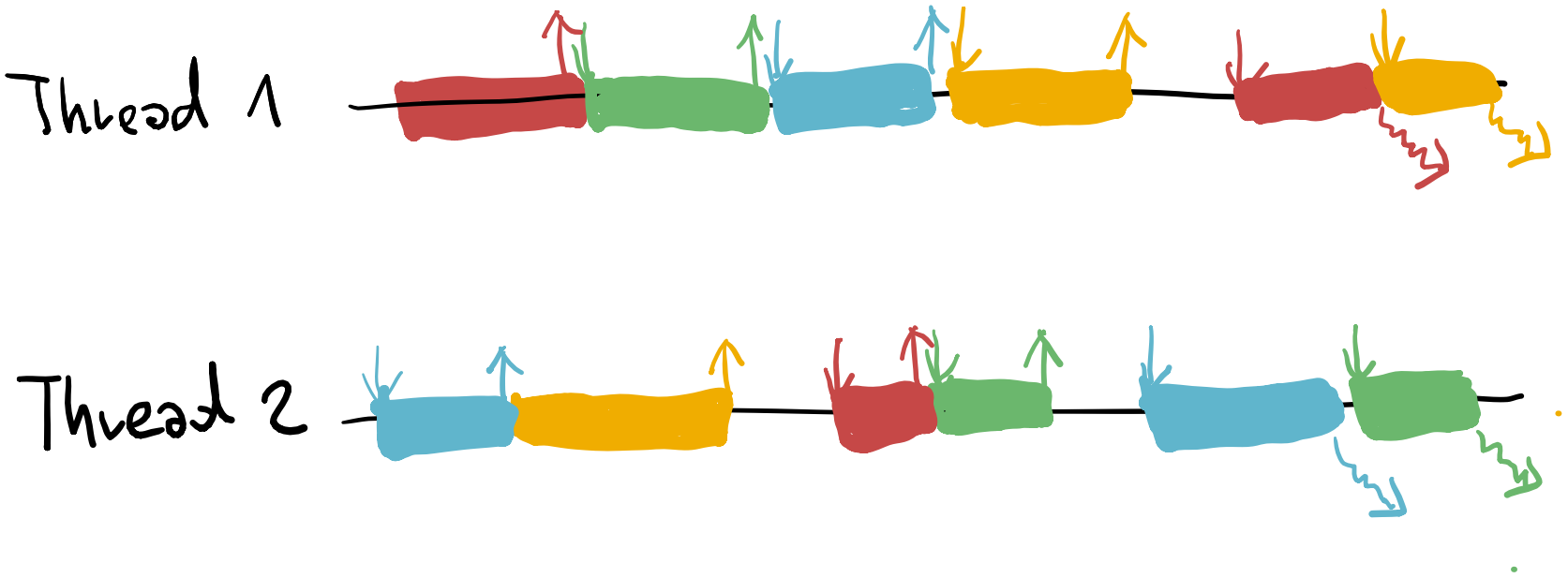
```
var wc = new WebClient();
var html = await wc.DownloadDataTaskAsync(url);
await outputStream.WriteAsync()
```

Easy to **change**, easy to **write**

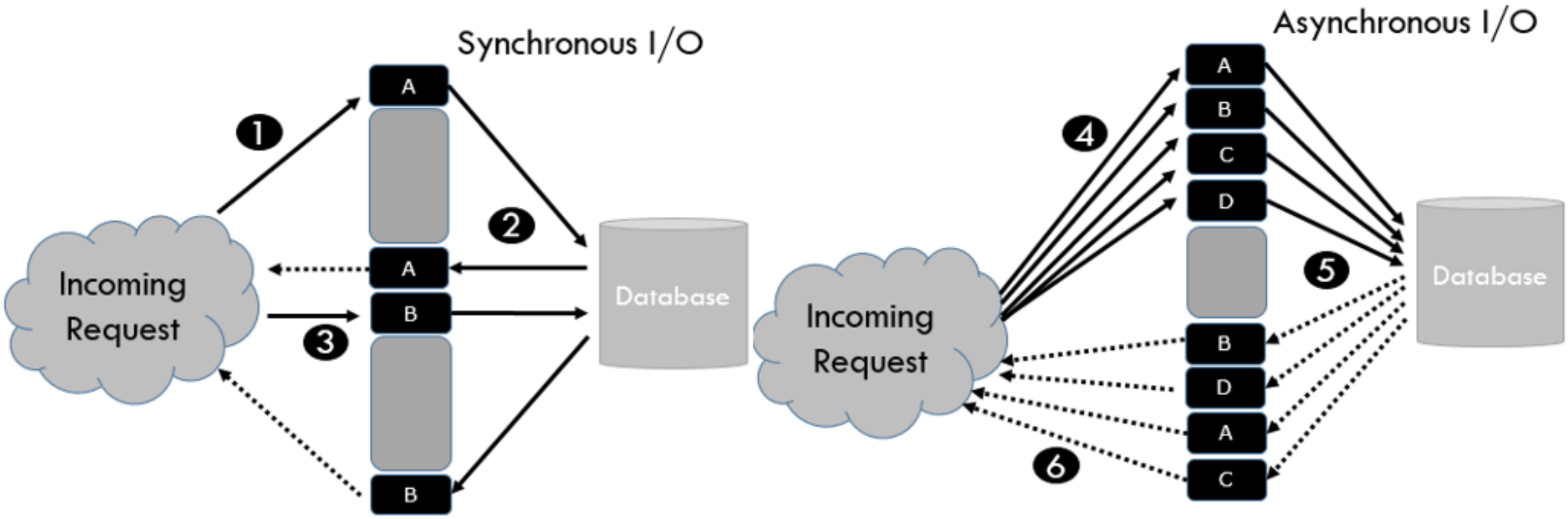
Can use **loops** and **exception handling**

Scalable – no blocking of threads

Asynchronous



Synchronous vs Asynchronous



async & await are about joins, not forks

```
public static Task
PausePrintAsync()
{
    // Requires a thread for all
    // of its processing
    return Task.Run(() => {
        Thread.Sleep(10000);
        Console.WriteLine("Hello");
    });
}
```

```
public static Task
PausePrintAsync()
{
    // Don't consume threads
    // when no threads are needed
    var t = Task.Delay(10000);
    return t.ContinueWith(_ => {

Consol WriteLine("Hello"));
    });
}
```

Task

Synchronous join

```
task.Wait();
```

Asynchronous join

```
await task;
```

Copy stream example

```
async Task CopyStream(Stream input, Stream output, IProgress<int> progress)
{
    byte[][] buffers = new byte[2][] { new byte[BUFFER_SIZE], new byte[BUFFER_SIZE] };
    int filledBufferNum = 0;
    int readBytes = await input.ReadAsync(buffers[filledBufferNum], 0,
                                                        buffers[filledBufferNum].Length);

    while (readBytes > 0)
    {
        progress.Report(readBytes);

        Task writeTask = output.WriteAsync(buffers[filledBufferNum], 0, readBytes);
        filledBufferNum ^= 1;
        Task<int> readTask = input.ReadAsync(buffers[filledBufferNum], 0,
                                                        buffers[filledBufferNum].Length);
        await Task.WhenAll(readTask, writeTask);
        readBytes = readTask.Result;
    }
}
```

Its all about Scalability

```
let httpAsync (url : string) = async {
    let req = WebRequest.Create(url)
    let! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    return! reader.ReadToEndAsync() }

let sites =
    [   "http://www.live.com";"    “http://www.fsharp.org";
        "http://news.live.com";    "http://www.digg.com";
        "http://www.yahoo.com";    "http://www.amazon.com"
        "http://www.google.com";   "http://www.netflix.com";
        "http://www.facebook.com"; "http://www.docs.google.com";
        "http://www.youtube.com";  "http://www.gmail.com";
        "http://www.reddit.com";   "http://www.twitter.com"; ]

sites
|> Seq.map httpAsync
|> Async.Parallel
|> Async.Start
```

Asynchronous Caching

```
static Func<T, Task<R>> MemoizeLazyTaskdSafe<T, R>(Func<T, Task<R>> func)
{
    ConcurrentDictionary<T, Lazy<Task<R>>> cache = new ConcurrentDictionary<T, Lazy<Task<R>>>();
    return arg => cache.GetOrAdd(arg, a => new Lazy<Task<R>>(() => func(a))).Value;
}

var client = new HttpClient();
var getData = MemoizeLazyTaskdSafe<string, string>(url =>
{
  Console.WriteLine($"Get String Async {url} - {DateTime.Now.ToString("hh:mm:ss.fff")}");
  return client.GetStringAsync(url);
});

var result_1 = await getData("http://www.google.com");
var result_2 = await getData("http://www.microsoft.com");
var result_3 = await getData("http://www.google.com");
var result_4 = await getData("http://www.microsoft.com");
```

Scalability

is the ability to cope and perform under an increasing workload

Its all about Asynchronicity !

Concurrent Server Side programming

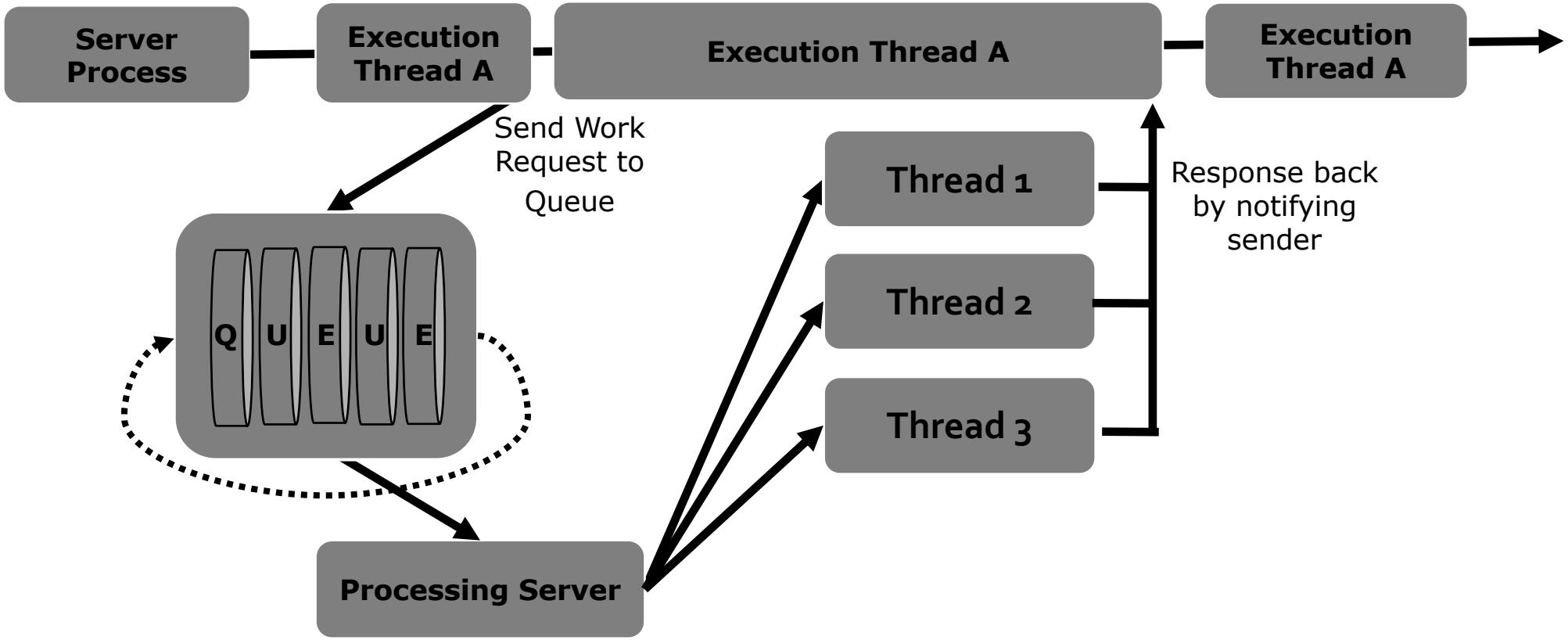
Reliability and scalability : effort is needed to ensure that applications don't stop working. The throughput needs to scale as new clients start using the service.

Concurrent requests : server-side applications must be designed to enable handling multiple requests concurrently. A program. With a single sequential control flow is bad design.

I/O bound operations : server-side applications exploit large number of I/O operations, which may occupy the largest portion of the request-processing time. This applies to receiving input from the client and sending a response back.

Shared state : servers often exist to enable state sharing among a large number of client. This means that concurrently running processes in a server-side application need to be able to safely access shared states.

A different asynchronous pattern
Queuing work for later execution



Brain Teasers

Symptom: Not Running Asynchronously

```
async void button1_Click(…)
{
    …
    await Task.Run(() => work());
    …
}
```

Problem:
Thinking ‘async’ forks.

Solution:
Use Task.Run around a synchronous method if you need to offload… but don’t expose public APIs like this.

Symptom: Completing Too Quickly

▲ 1 of 12 ▼ ParallelLoopResult Parallel.For(**int fromInclusive**, int toExclusive, Action<int> body)

Executes a for (For in Visual Basic) loop in which iterations may run in parallel.

fromInclusive: *The start index, inclusive.*

```
Parallel.For(0, 10, async i => {
    await Task.Delay(1000);
});
```

Problem:
Async void lambda.

Solution:
Use extreme caution (& knowledge) when passing around async lambdas as void-returning delegates.

▲ 2 of 16 ▼ (awaitable) Task<Task> TaskFactory.StartNew<Task>(**Func<Task> function**)

Creates and starts a System.Threading.Tasks.Task<TResult>.

Usage:
Task x = await StartNew(...);

function: *A function delegate that returns the future result to be available through the S*

```
await Task.Run(
    async () => { await Task.Delay(1000);
});
```

Problem:
Task<Task> instead of Task.

Solution:
Make sure you're unwrapping your Tasks, explicitly (Unwrap) or implicitly.

Symptom: Method Not Completing

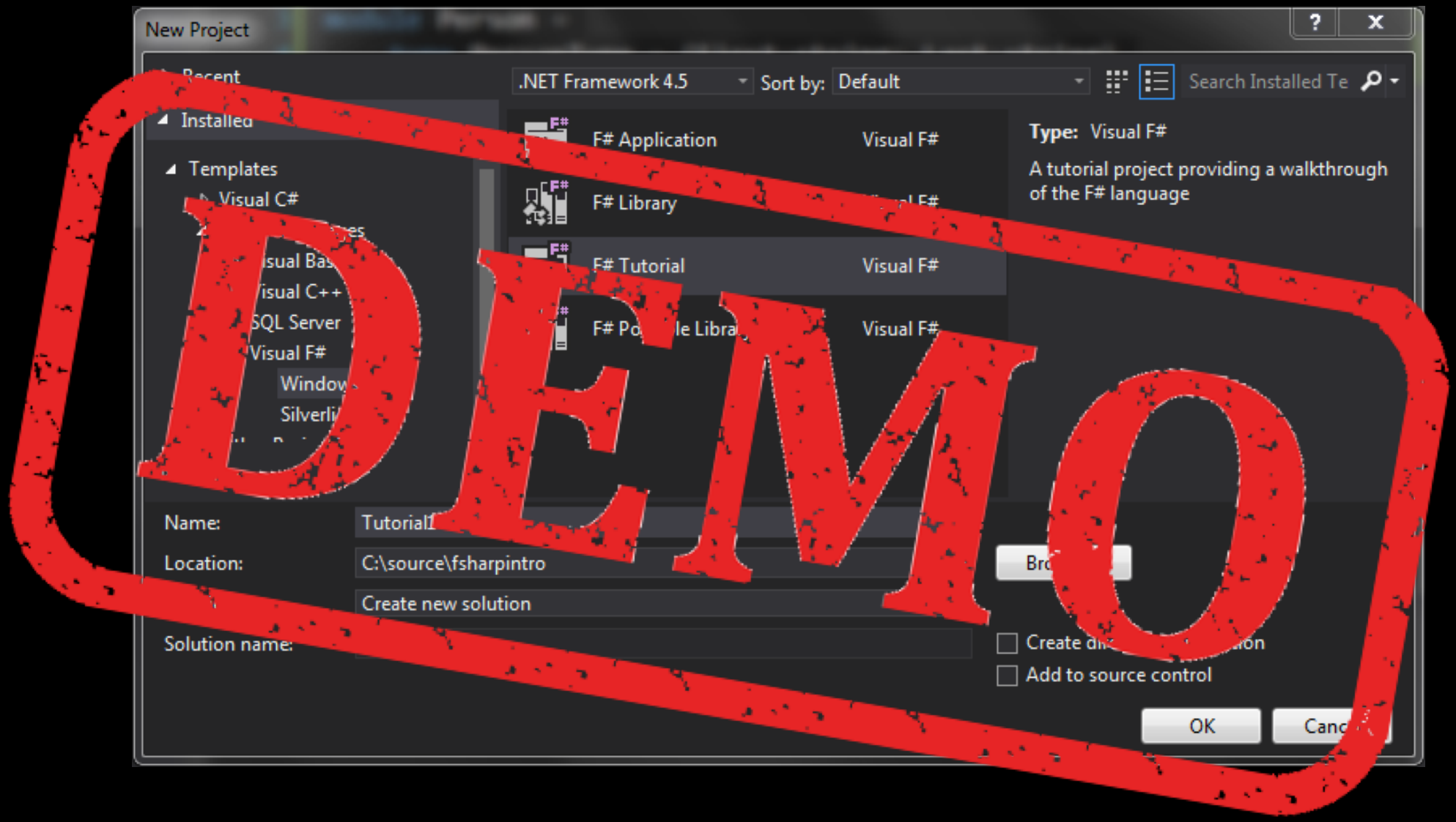
```
async void button1_Click(…) {
    await FooAsync();
}
async Task FooAsync()  {
    await
Task.Delay(1000).ConfigureAwait(false);
}
```

Problem:
Deadlocking UI thread.

Solution:
Don’t synchronously wait on the UI thread.

Problem:
Deadlocking UI thread.

Solution:
Use ConfigureAwait(false) whenever possible.


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Some Task combinators

Reliable Tasks with Retry

```
async Task<T> Retry<T>(Func<Task<T>> task, int retries,
                       TimeSpan delay, CancellationToken? cts = null) =>
    await task().ContinueWith(async innerTask =>
    {
        cts?.ThrowIfCancellationRequested();
        if (innerTask.Status != TaskStatus.Faulted)
            return innerTask.Result;
        if (retries == 0)
            throw innerTask.Exception;
        await Task.Delay(delay, cts.Value);
        return await Retry(task, retries - 1, delay, cts.Value);
    }).Unwrap();
```

Retry in action

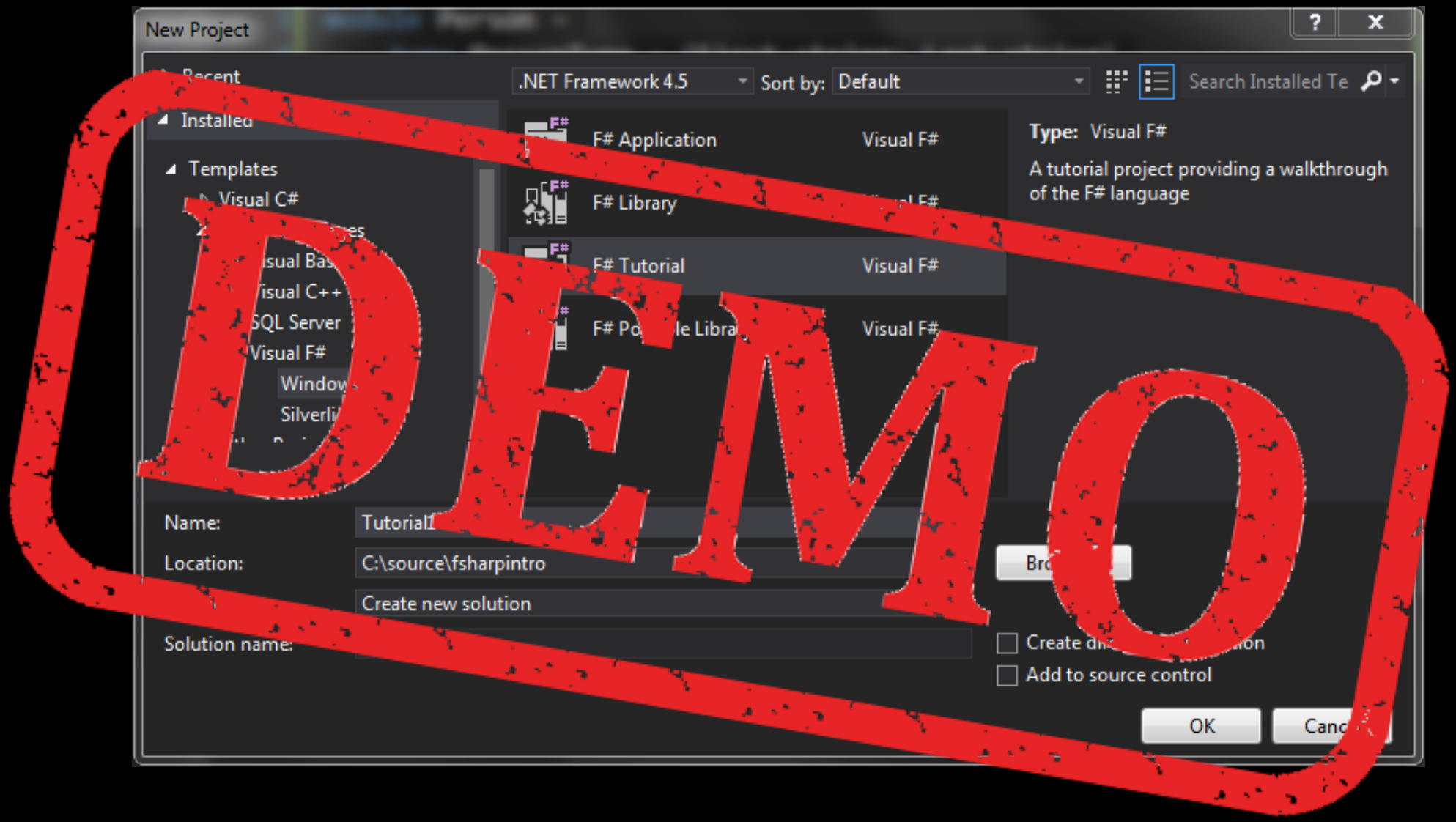
```
Image image = await Async.Retry(async () =>
                        await DownloadImageAsync("Bugghina001.jpg")
                    , 5, TimeSpan.FromSeconds(2));
```

Reliable Tasks with Otherwise

```
async Task<T> Otherwise<T>(this Task<T> task, Func<Task<T>> orTask) =>
    await task.ContinueWith(async innerTask =>
    {
        if (innerTask.Status == TaskStatus.Faulted) return await orTask();
        return await Task.FromResult<T>(innerTask.Result);
    }).Unwrap();
```

Otherwise in action

```
Image image = await Async.Retry(async () =>
                    await DownloadImageAsync("Bugghina001.jpg")
                .Otherwise(async () =>
                    await DownloadImageAsync("Bugghina002.jpg")),
            5, TimeSpan.FromSeconds(2));
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO