

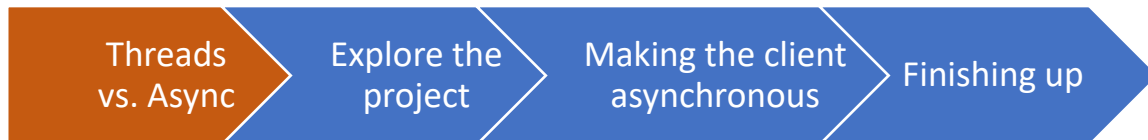
Dist. Sys.

Async and Tasks

In a previous session we looked at the concept of threads.

In this session we will look at the concept of Async and Await. These are keywords in the C# programming language which allow us to perform tasks without blocking the caller. At first glance this seems to be the same goal as threading, but we will explore the fundamental differences and where the two meet.

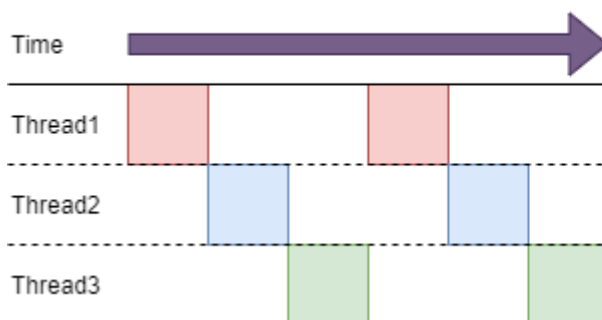
We will start off by following this flow:



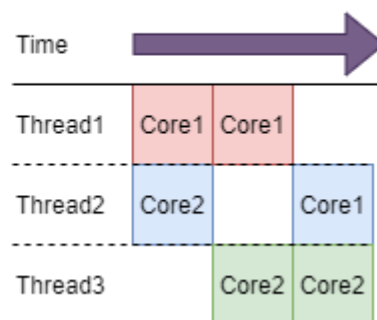
Let's recap threads:

- Threads allow us to execute commands concurrently, allowing our software to do more than one thing at any time
- Threads are managed by the runtime environment
- Threads are commonly used in servers to allow multiple clients to interact simultaneously
- The concept of multithreaded applications executing code 'at the same time' may be incorrect. Threaded code may swap processing time between threads very quickly to give the impression of concurrency. On the other hand, threads may operate on separate cores of a processor for real concurrency, although there are rarely enough cores for every application and thread to be running simultaneously

Single Core Processor



Dual Core Processor



We have seen previously that, due to work being done on different threads 'concurrently', we may run into problems where threads need to communicate between one another or share resources. We often need some assurances of what is happening and when.

Threads have another critical drawback: creating, managing and eventually destroying a thread are an extremely resource-intensive set of operations. Therefore, using threads to perform small tasks can be a waste of resources.

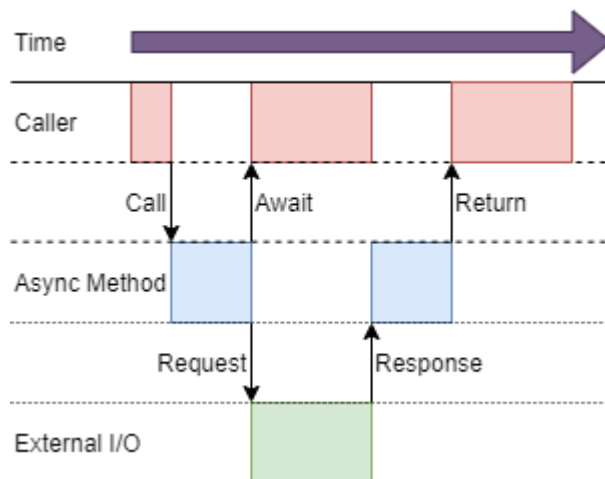
Fortunately, we have another option: asynchronous tasks. We considered the concept of an asynchronous callback in the workshop on threading, so we know that this type of operation allows us to run a specific method whenever the long-running operation finishes.

In this session we will go one step further and use async methods.

These methods allow us to indicate when they are waiting for some long-running operation to complete, allowing the calling code to continue to run whilst the method waits. This is usually how client user interfaces of distributed applications remain responsive whilst engaging in I/O.

Critically, the asynchronous method is ***not running concurrently with its caller***.

The flow looks more like this:



- The caller makes a call to the async method. This is a synchronous operation and identical to any other method call.
- The async method starts running and the caller is blocked.
- Then the async method gets to a call which requires some external function. This could be a call to a networked service endpoint. We don't know how long this will take and we don't want to block the caller from continuing while we wait so we mark this call with the ***await*** keyword.
- The await keyword allows the caller to continue to execute until a response has been passed from the external operation.

- At that point, a callback operation inside the execution environment halts the execution of the caller and gives it back to the async method to continue to do its job.
- The async method finishes and returns back to its caller as normal.

The critical feature here is the await keyword. This informs the runtime that work is being done asynchronously, outside the main application. This could be work on another thread, being done by the Operating System, or on another machine (e.g. a call to a server to do some work).

Once the await keyword is used, the execution of the method pauses until the response is received. In the meantime, an incomplete **Task** object is returned to the caller. The caller can now decide if it wants to block and wait for the Task to be complete or if it can continue.

This functionality gives us some crucial control over synchronisation. Only the long-running external I/O operation is happening concurrently with other parts of our application and we are able to check the status of the method that made the request for this I/O operation to see if it has completed yet. We can therefore remain much more in control, and this operation is extremely lightweight compared to starting a whole thread to make this call in the background.

Consider this scenario...

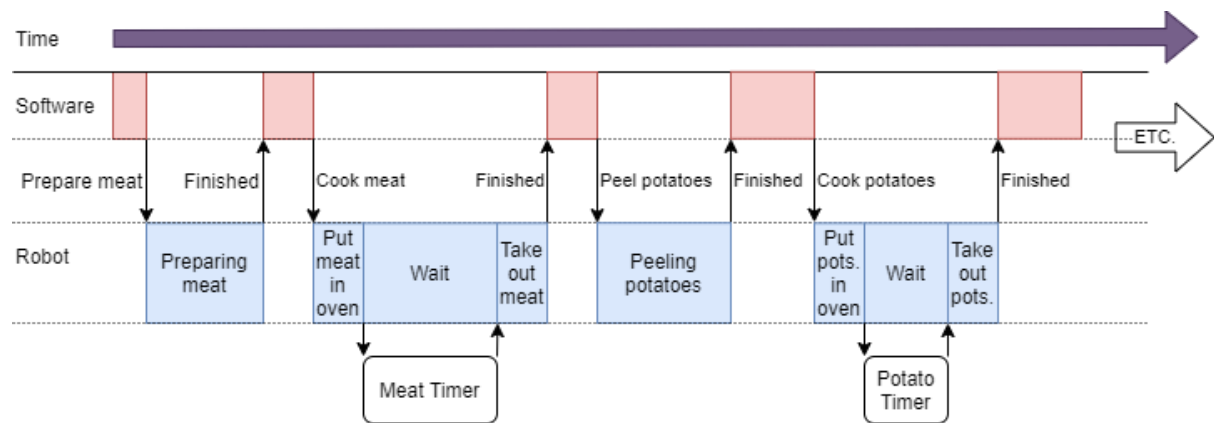
Making Lunch

We have a futuristic food-making robot (LUNCHO-9000) that we are able to interface with by giving it instructions. Depending on the instruction, some tasks take a while and others are very quick. We want a roast dinner for lunch, so there are essential items:

Roast Meat, Roast Potatoes, Yorkshire Pudding, Vegetables, Gravy, etc.

We only have one robot, and it can't do multiple things at once, so there is no option for 'threading' its behaviour. We could send it the instructions in the order we want the food to be prepared.

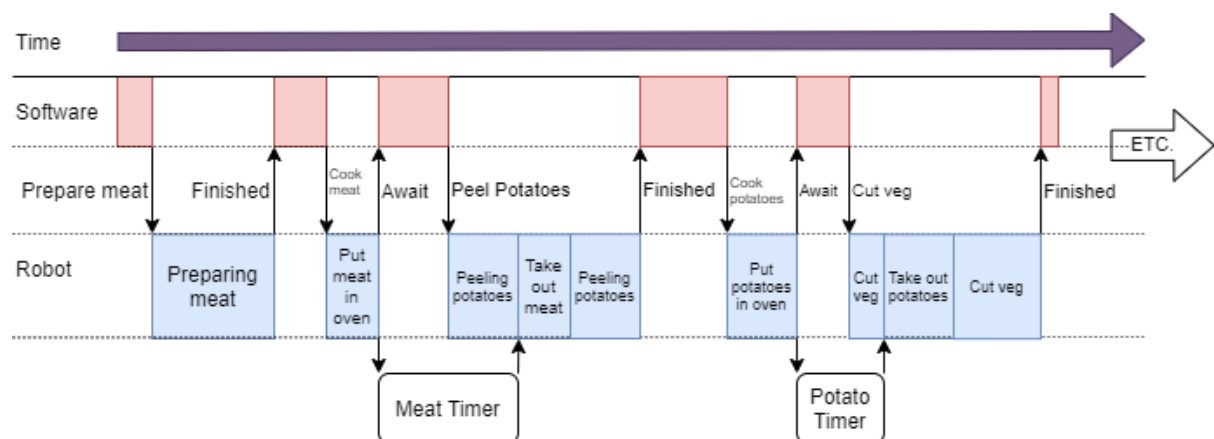
This might look something like this:



Immediately it is clear that we have a problem. The robot is waiting for one food item to cook before starting to work on the next. This means lots of waiting around doing nothing and our food will not be finished at the same time; we'll probably end up with many of our food items cold.

The waiting is blocking our ability to do other work.

Instead let's consider the scenario if we work asynchronously.




Now, our software and robot make use of the time whilst it would have been waiting for the external long-running task (the actual cooking) to get more done. Notice that the robot and software never actually do any work in parallel. They simply continue to do work whilst the external long-running task continues. This is extremely useful. With more efficient scheduling we could ensure that everything went into the oven at the right time to ensure that it all finished at the same moment.



We will now prepare an example project to explore asynchronous programming ourselves.

This project uses Sockets and Threads as we have explored in previous workshops, so ensure that you are familiar with these before continuing.



Download the lab file skeleton solution from Canvas and open it in Visual Studio.

Have a look through the projects and files that are included. There is a Server and a Client. The connection between the two is managed by sockets and the server uses threads to allow more than one concurrent connection. Work out what is going on in this code now.

There are copious comments, so read through these to get a good idea of what is going on.

The client currently does not call the server.



In the `EntryPoint` of the Client, make three calls to `SendRequest` targeting the endpoints `TaskOne`, `TaskTwo` and `TaskThree` in order.

If the solution still doesn't make any sense to you, put in some breakpoints to examine execution as it runs, remembering to remove them when you have a better idea.

Start the solution. It is set to start the Server and then the Client. If this is causing you any problems with the client requesting access to the server before the server has initialised, you may like to run the server individually and wait until you see "[!] Server Active" before starting the client.

You will need to wait for the tasks to complete once the client has started. Remember, task 1 takes ten seconds ... so be patient.

You should now be looking at two Console windows. A Server and a Client:

```
C:\Box\Box Sync\Projects\Async...
[!] Server Active
[!] New Connection: 125e7b0e
125e7b0e Called Endpoint: TaskOne
125e7b0e Sent Message: Hello World
[!] New Connection: 80ab169f
80ab169f Called Endpoint: TaskTwo
80ab169f Sent Message: Hello World
[!] New Connection: 393cca54
393cca54 Called Endpoint: TaskThree
393cca54 Sent Message: Hello World

Select C:\Box\Box Sync...
Task 1 Finished!
Task 2 Finished!
Task 3 Finished!
Execution finished
```

This all seems to be in order... so what is the problem?

Well, did you notice how long Task 1 took? In all of that time we simply waited for the server to send a response, which completely blocked the execution of any other code in our client. That was 10 seconds of wasted time.



There are two operations in the client which take time that could be spent doing something else. Both are I/O operations. These are `nStream.Write` in the `SendRequest` method and `nStream.Read` in the `RetrieveResponse` method (there are two reads in this method). By far the worst offender is the stream read, which simply waits for the number of expected bytes to be available in the stream before continuing. Theoretically, if these bytes never came, we could be waiting for ever.

These read and write operations are reliant on external I/O. The runtime environment and operating system are doing that work for you, so while they do their work we can continue to do ours!

Notice how each call to `SendRequest` creates a new TCP connection and `NetworkStream`? This is on purpose. The server is set up to only respond to one request on each thread so we need to open a new connection for each request. This is similar to how conventional HTTP webservers work.

Because of the multithreaded nature of the server we can have our client open more than one connection at the same time, therefore giving us the ability to make a second, third, etc. request before the first has returned. The server simply manages the new connection and request on a different thread.

Let's go ahead and make the **SendRequest** and **RetrieveResponse** methods **async** so they can perform their waiting without blocking execution:



Modify the **SendRequest** method signature from:

```
public static void SendRequest(string message, string endpoint)
```

to

```
public static async Task SendRequest(string message, string endpoint)
```

Notice that we have done two things here...

1. We have added the **async** keyword to mark this method as able to run asynchronously
2. We have changed the return type from **void** to **Task**. The **Task** tracks the completion of this method. If everything goes well and the method completes with no problems the task status is set to 'RanToCompletion'. If an exception is thrown, the task status will show 'faulted'. There are a handful of other statuses managed by the **TaskStatus** enum. The crucial thing about using **Task** is that it allows us to identify if the method has finished from outside the method.

Notice that we don't have to return a **Task** object. This is managed for us.

"But what about if our method returned something rather than **void**?" I hear you ask. Good question, you know that we can only have a single return type. Thankfully we can use the generic version of **Task** to return a type as the **Task.Result**.

If the method returned a **string**, we could change it to return **Task<string>** when we made the method **async**. We can then return a **string** as usual and everything is hooked up for us in the background to make this work.



Modify the **RetrieveResponse** method signature in the same way as you modified **SendRequest**.

Now our methods *can* act asynchronously but you should see green squiggly lines under your method names identifying the fact that they currently do not and therefore will run synchronously (like any other method).

Thankfully we know where the I/O is in these methods so we can make some changes.



Hover your cursor over the **nStream.Write** call in the **SendRequest** method to see its metadata.

Notice that it returns **void**? This is currently a synchronous call. Thankfully, the library developers assumed we may want to do this type of work asynchronously too, so they made a counterpart method: **WriteAsync**.

WriteAsync method uses an **await** keyword when it starts waiting for the external I/O work to be done. At this point, control is passed back to the caller. If we place our own **await** keyword, we can, in turn, return control to our caller.



Update the line in `SendRequest`: `nStream.Write(request, 0, request.Length);`

To: `await nStream.WriteAsync(request, 0, request.Length);`

Now update the two `Read` calls in `RetrieveResponse` in the same way.

The squiggly lines on the method names should have gone away because we are now using the `await` keyword to identify the line(s) which will cause the waiting and thus return control to the caller. Once the associated task completes, control will be passed back to this line.

You should be left with some more squiggly lines though, on calls to `SendRequest` and `RetrieveResponse`. These identify that the method is async and so execution will continue past this point when the method returns control. This is sort of what we want when we are calling `SendRequest` but we should consider the implications of this a little closer.

What do you think will happen when we run this application?

➤ Run both the server and client to find out.

Probably you're looking at something like the below images (remember that the thread `WriteLines` can overlap in any order so it might not be exactly this):

```
C:\Box\Box Sync\Projects\...
[!] Server Active
[!] New Connection: 864126c8
[!] New Connection: 68979971
[!] New Connection: cc223eb6
864126c8 Called Endpoint: TaskOne
864126c8 Sent Message: Hello World
68979971 Called Endpoint: TaskTwo
68979971 Sent Message: Hello World
cc223eb6 Called Endpoint: TaskThree
cc223eb6 Sent Message: Hello World

C:\Box\Box Sync...
Execution finished
```

So what happened? The client made the requests asynchronously but failed to do anything with the response because it finished executing before the responses came back.

Let's use those returned Task types to identify if **SendRequest** has finished before we continue to exit the client.



SendRequest now returns a Task so, for your calls to **SendRequest** in your **EntryPoint**, assign these returned objects to a local Task variable.

After these method calls, call the synchronous **Wait** method on each of these in turn. This will prevent the Client from continuing until the associated Task is complete.

Here is an example of this for **TaskOne**:

```
Task one = ClientBehaviours.SendRequest("Hello World", "TaskOne");

// Send Requests to TaskTwo and TaskThree here

one.Wait();

// Wait for two and three here

Console.WriteLine("Execution finished");

Console.ReadLine();
```

Note: There are other ways to achieve this same task waiting. Can you identify any? The further work from last week might help...

What will happen in this **EntryPoint** now?

- The first call to **SendRequest** will be made as usual
- When this **SendRequest** method starts waiting at its **await** keyword, control is passed back to the **EntryPoint**.
- The **EntryPoint** makes the next call to **SendRequest**, etc.
- When all three calls have been made to **SendRequest** the **EntryPoint** starts waiting for Task one to finish. When it does, it waits for Task two and then Task three to finish.
- If Task two has finished before Task one, the method does not need to wait.

This ensures that the three calls to **SendRequest** have finished executing before the program exits.

What do you think will happen when we run this application?

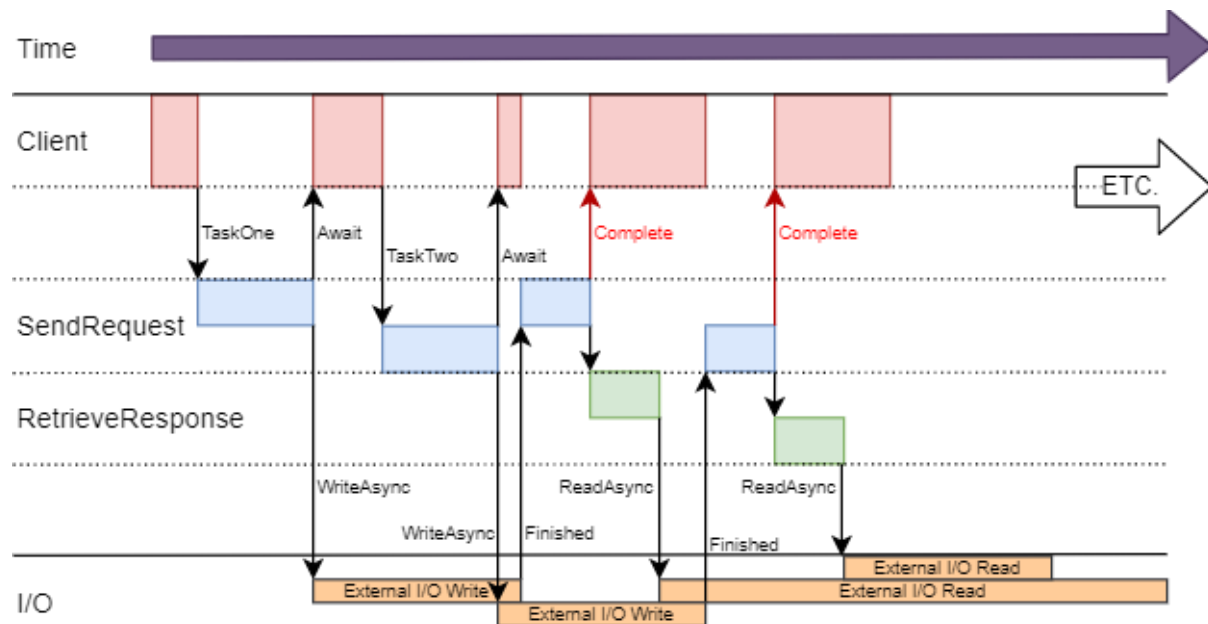
➤ Run both the server and client to find out.

Probably you're looking at something very similar to what you had before...

We are now waiting for the Task to be returned by **SendRequest** before continuing. The Task is only sent back once the **await** keyword is hit so these connections are now made one after another.

But our client still isn't working!

The diagram below shows an example start to the execution of the client. Look at the 'Complete' method returns marked in red. What is wrong about this?



Perhaps you have identified that a request and response should really be combined into one operation. **SendRequest** shouldn't be complete until it has received a response.

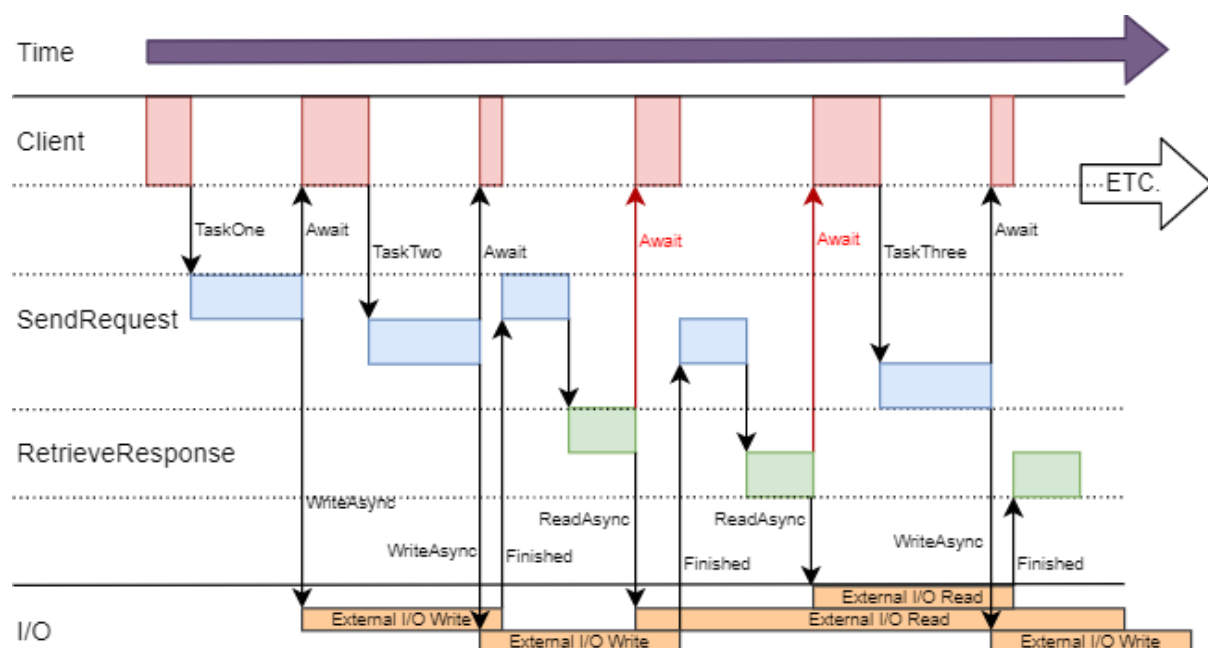
Visual Studio is probably highlighting this for you too. The call to **RetrieveResponse** should show a warning identifying that the **SendRequest** method will continue before the **RetrieveResponse** call returns. We don't want this!

Modify the line `RetrieveResponse(nStream);`

To

`await RetrieveResponse(nStream);`

Look at what this has done to our diagram...

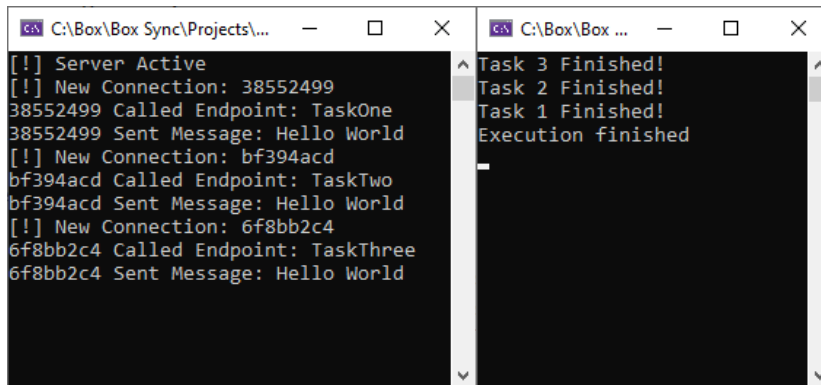


Now, rather than continuing past the **RetrieveResponse** call and thus marking the **SendRequest** Task as 'RanToCompletion', the Task continues to show as incomplete until its call to **RetrieveResponse** also completes (not shown on the diagram).

As our **EntryPoint** will continue to Wait until the **SendRequest** Task is complete (and this won't happen until the response is received), we ensure we get the response before continuing.

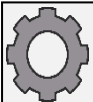
➤ Run both the server and client to see the outcome.

Hopefully you're looking at something like this:



The image shows two side-by-side console windows. The left window, titled 'C:\Box\Box Sync\Projects\...', displays server logs: '[!] Server Active', '[!] New Connection: 38552499', '38552499 Called Endpoint: TaskOne', '38552499 Sent Message: Hello World', '[!] New Connection: bf394acd', 'bf394acd Called Endpoint: TaskTwo', 'bf394acd Sent Message: Hello World', '[!] New Connection: 6f8bb2c4', '6f8bb2c4 Called Endpoint: TaskThree', and '6f8bb2c4 Sent Message: Hello World'. The right window, titled 'C:\Box\Box ...', displays client logs: 'Task 3 Finished!', 'Task 2 Finished!', 'Task 1 Finished!', and 'Execution finished'.

Notice the ordering. Task 3 finishes first, followed by task 2 and then, eventually task 1 finishes.



Download another copy of the skeleton solution from Canvas and set up the client to make the three calls without any of the async operators.

Put a stopwatch ...

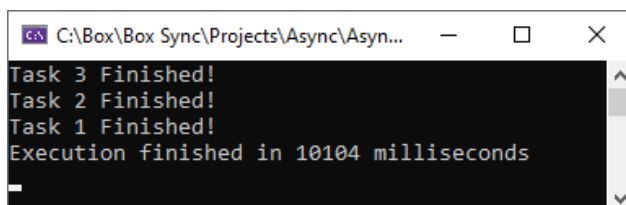
```
Stopwatch sw = new Stopwatch();
```

... starting at the beginning of the **EntryPoint** and ending just before the **Execution Finished** line.

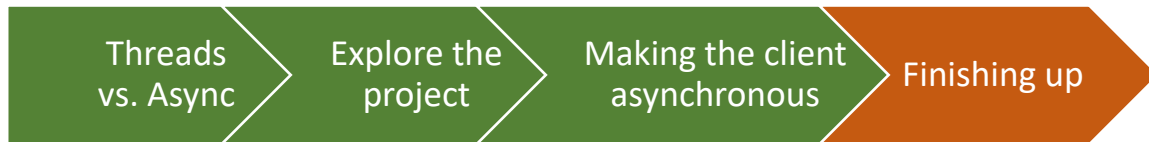
Modify the write out to show:

```
"Execution finished in " + sw.ElapsedMilliseconds + " milliseconds"
```

Do the same thing for your client with **async**. Compare the time taken.



The image shows a console window titled 'C:\Box\Box Sync\Projects\Async\Asyn...' with the following output: 'Task 3 Finished!', 'Task 2 Finished!', 'Task 1 Finished!', and 'Execution finished in 10104 milliseconds'.



Final Tasks

1. Run more than one client concurrently.
2. Add some new endpoints to the server. How about one that actually uses the message that the client sends?
3. What happens if your server runs out of threads? Can you improve the server?
4. In a previous session we looked at the concept of pipes and filters but we didn't apply these to a networked environment. Add some pipes and filters to the server.
5. Research `Task.Run`. What is this? What is a thread pool and why might you use `Task.Run` for short-running background work rather than a new `Thread`? Create an example project using `Task.Run`. Last week's further work may be able to help you with this!

Outcome

I would very, very much recommend that you continue to research asynchronous programming in C# as it is often critical for distributed system clients.

Here is a good starting point for independent research: <https://docs.microsoft.com/en-us/dotnet/csharp/async>

By the end of this you should have a good understanding of asynchronous programming and:

- the `async` keyword
- the `await` keyword
- `Task` objects