Asynchronous programming in depth

Synchronous Versus Asynchronous Operation

- A synchronous operation does its work before returning to the caller.
- An asynchronous operation can do (most or all of) its work after returning to the caller.
    - Asynchronous methods are less common and initiate **concurrency**, because work continues in parallel to the caller.
    - Asynchronous methods typically return quickly (or immediately) to the caller; thus, they are also called nonblocking methods.
- Thread.Start
- Task.Run
- Methods that attach continuations to tasks

What Is Asynchronous Programming?

- Principle of asynchronous programming is that you write long-running (or potentially long-running) functions asynchronously.
- in contrast to the conventional approach of writing long-running functions synchronously, and then calling those functions from a new thread or task to introduce concurrency as required.

Two distinct uses (benefits):

- writing applications that deal efficiently with a lot of concurrent **I/O**. The challenge here is not thread *safety* (because there's usually minimal shared state) but thread *efficiency*;
- simplify thread-safety in rich-client applications.

Asynchronous Programming Patterns

Asynchronous Programming Model (APM)

Not recommended

- `Begin` and `End` methods (for example, `BeginWrite` and `EndWrite`)
- `IAsyncResult`

Event-based Asynchronous Pattern (EAP)

Not recommended

- Methods with `Async` suffix
- One or more events

Task-based Asynchronous Pattern (TAP)

Recommended

- `Task`s and `async`/`await` keywords.

Tasks

Tasks are constructs used to implement what is known as the ***Promise Model of Concurrency***. In short, they offer you a "promise" that work will be completed at a later point, letting you coordinate with the promise with a clean API.

- **Task** represents a single operation which does not return a value.
- **Task<T>** represents a single operation which returns a value of type **T**.

Tasks expose an API protocol for *monitoring*, *waiting upon* and *accessing the result value* (in the case of **Task<T>**) of a task.

Tasks Continuation. Awaiter

```
Task<int> primeNumberTask = Task.Run (() =>
   Enumerable.Range (2, 3000000).Count (n =>
     Enumerable.Range (2, (int) Math.Sqrt(n)-1).All (i => n % i > 0)));

var awaiter = primeNumberTask.GetAwaiter();
// var awaiter = primeNumberTask.ConfigureAwait (false).GetAwaiter();
awaiter.OnCompleted (() =>
{
    int result = awaiter.GetResult(); // Instead of .Result (direct exception)
    Console.WriteLine (result); // Writes result
});
```

Asynchronous functions

Language integration, with the **`async`/`await`** keywords, provides a higher-level abstraction for using **`Task`**s.

- Method declaration should be marked with **`async`** keyword.
- Method should return **`Task`** or **`Task<T>`**

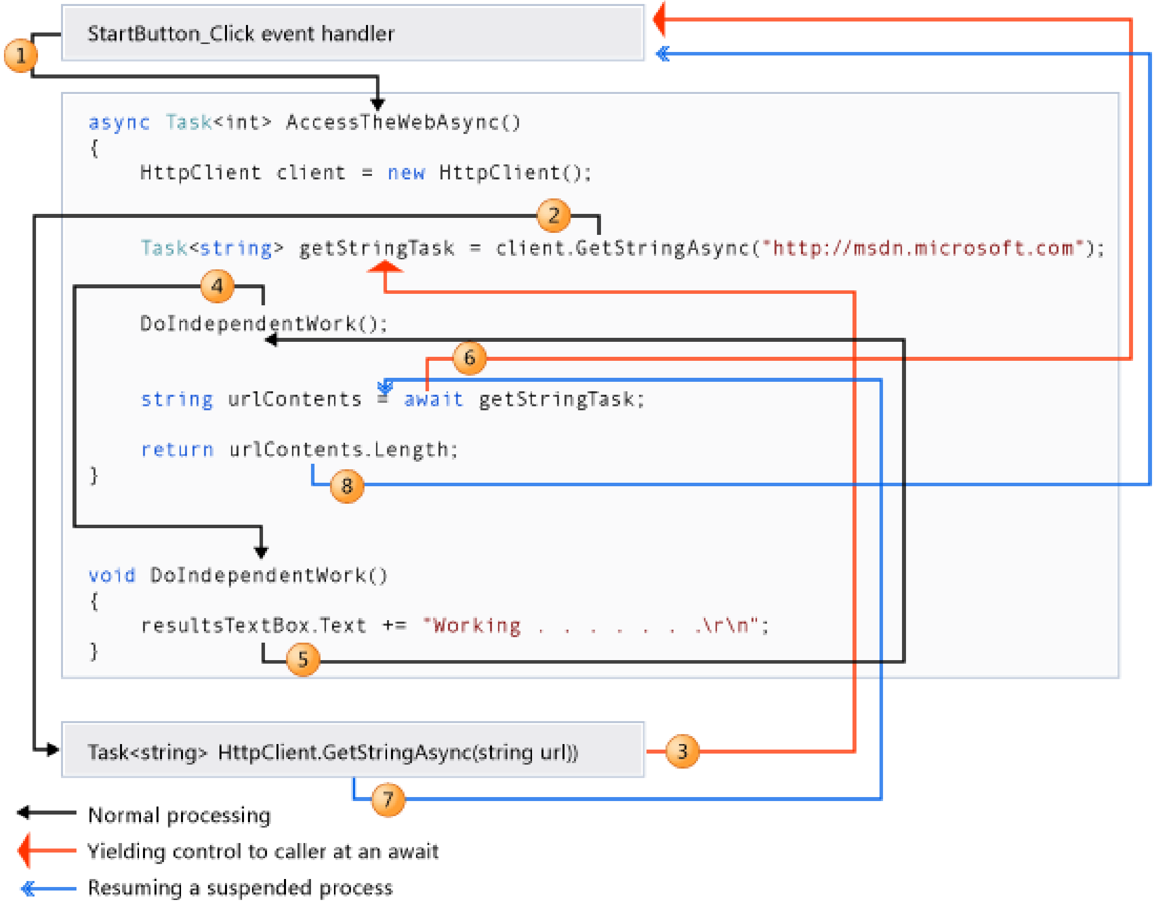
```
async Task<string> DoSomeWork() {
    // ...
    return await sr.ReadToEndAsync();
}
```

Without **async/await**:

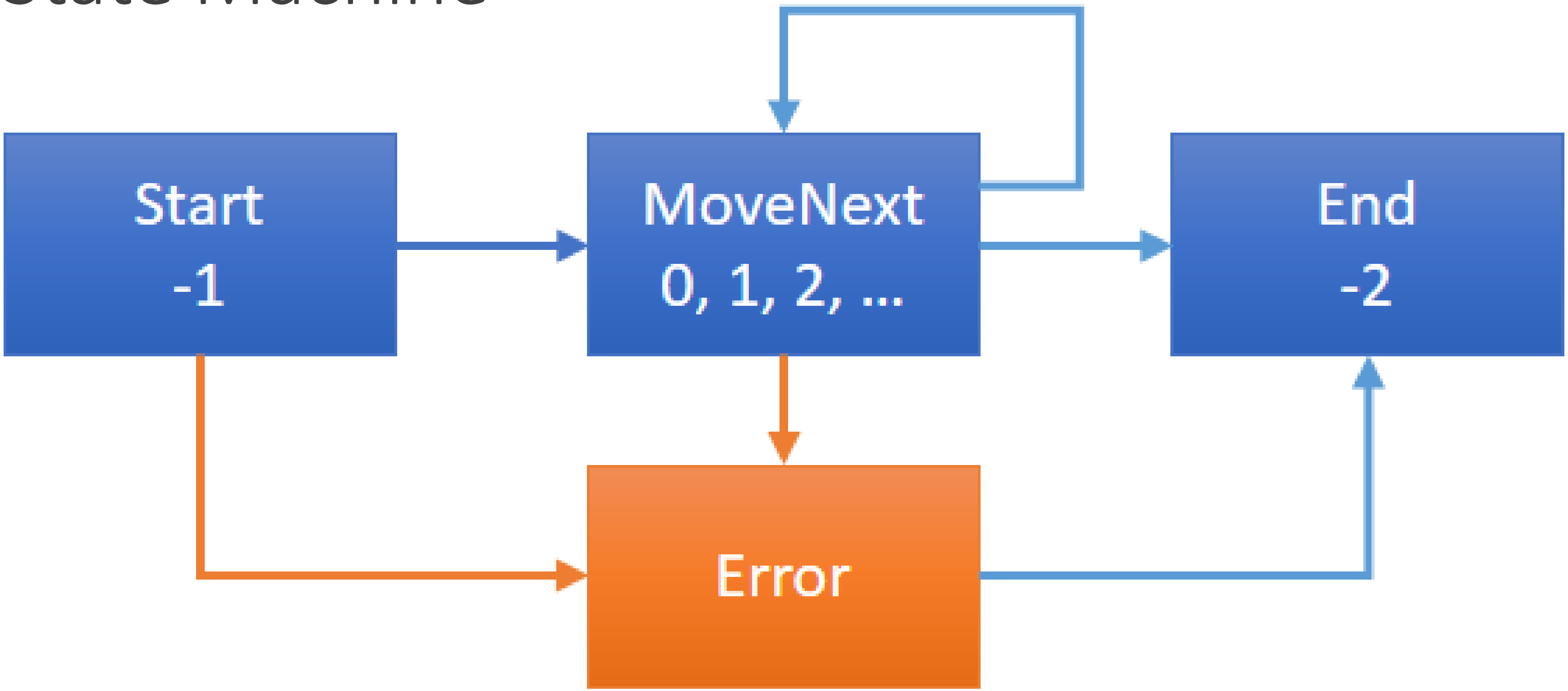
```
Task.Run(() => ComputeHeavyOp(1))
  .ContinueWith(_ => ComputeHeavyOp(2))
  .ContinueWith(_ => ComputeHeavyOp(3));
```

With **async/await**:

```
await ComputeHeavyOpAsync(1);
await ComputeHeavyOpAsync(2);
await ComputeHeavyOpAsync(3);
```


StartButton_Click event handler
async Task<int> AccessTheWebAsync()
{
HttpClient client = new HttpClient();
Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");
DoIndependentWork();
string urlContents = await getStringTask;
return urlContents.Length;
}
void DoIndependentWork()
{
resultsTextBox.Text += "Working . . . .\r\n";
}
Task<string> HttpClient.GetStringAsync(string url))
1
2
3
4
5
6
7
8
Normal processing
Yielding control to caller at an await
Resuming a suspended process

State Machine



Asynchronous functions

- With any asynchronous function, you can replace the void return type with a Task to make the method itself usefully asynchronous (and awaitable). No further changes are required.
- C# also supports **void** return type for **async** functions. It can be helpful when you want to create an **async** event handler.

```
async void Form_Load(object sender, EventArgs e) {
    await OnFormLoadAsync(sender, e);
}
```

- You can also use **async** lambdas:

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

Asynchronous functions

Basic principle of how to design with asynchronous functions in C#:

1. Write your methods synchronously.
2. Replace synchronous method calls with asynchronous method calls, and await them.
3. Except for “top-level” methods (typically event handlers for UI controls), upgrade your asynchronous methods’ return types to Task or Task<TResult> so that they’re awaitable.

Task.WhenAll and Task.WhenAny

`Task.WhenAll` waits for all tasks to complete:

```
var tasks = from i in Enumerable.Range(0,100)
    select SomeTask(i);
TResult[] resultsArray = await Task.WhenAll(tasks);
```

Task.WhenAny waits for the first task to complete:

```
var tasks = from i in Enumerable.Range(0,100)
    select SomeTask(i);
Task<TResult> firstFinished = await Task.WhenAny(tasks);
```

async/await limitations

Asynchronous functions have some limitations:

- Constructors, property accessors and event accessors cannot be **`async`**.
- Cannot have any **`out`** or **`ref`** parameters.
- Cannot use **`await`** inside **`unsafe`** block.
- You cannot obtain a **`lock`** before **`await`**, and release it after **`await`**.

Asynchronous Streams

```
static async Task<IEnumerable<int>> RangeTaskAsync (int start, int count, int delay)
{
    List<int> data = new List<int>();
    for (int i = start; i < start + count; i++)
    {
        await Task.Delay (delay);
        data.Add (i);
    }
    return data;
}
```

```
foreach (var data in await RangeTaskAsync(0, 10, 500))
    Console.WriteLine (data);
```

Asynchronous Streams

```
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator (...);
}

public interface IAsyncEnumerator<out T>:
IAsyncDisposable
{
    T Current { get; }
    ValueTask<bool> MoveNextAsync();
}
```

```
async IAsyncEnumerable<int> RangeAsync (
int start, int count, int delay)
{
    for (int i = start; i < start + count; i++)
    {
        await Task.Delay (delay);
        yield return i;
    }
}
```

```
// for Linq queries System.Linq.Async
```

```
await foreach (var number in RangeAsync (0, 10, 500))
    Console.WriteLine (number);
```

The Task-Based Asynchronous Pattern

- Returns a “hot” (running) Task or Task<TResult>
- Has an “Async” suffix (except for special cases such as task combinators)
- Is overloaded to accept a cancellation token and/or IProgress<T> if it supports cancellation and/or progress reporting
- Returns quickly to the caller (has only a small initial synchronous phase)
- Does not tie up a thread if I/O-bound

Questions

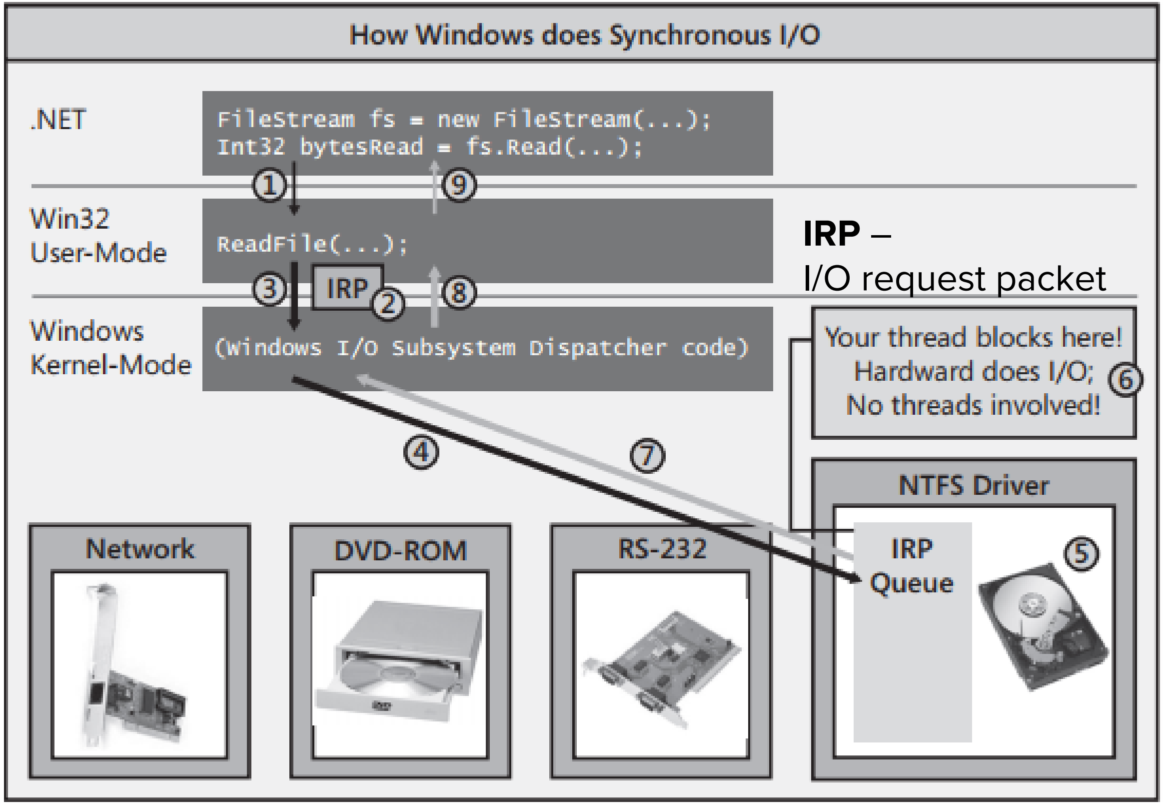
- Does using the “**`async`**” keyword on a method force all invocations of that method to be asynchronous?
- Can I mark any method as “**`async`**”?
- In which cases can we use **`void`** return type in **`async`** functions?
- Is “**`await task;`**” the same thing as “**`task.Wait()`**”?
- What is Asynchronous Streams? When should you use it?

Asynchronous operations

There are two major types of asynchronous operations:

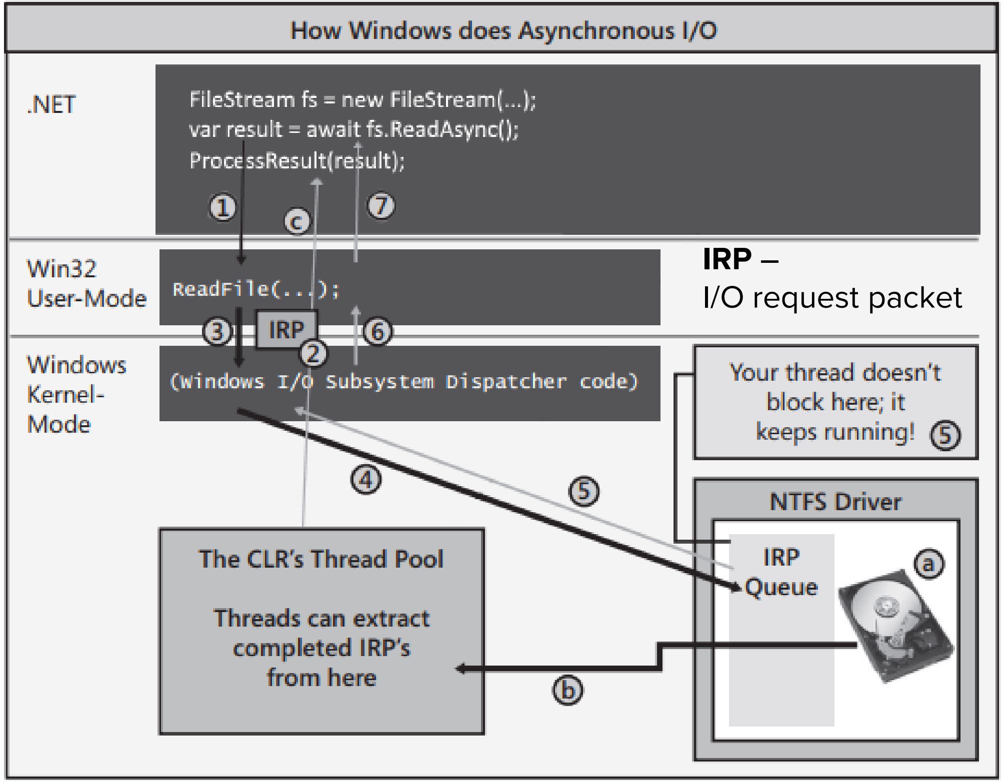
- I/O-bound operations
- CPU-bound operations

I/O-Bound Operations



IRP –
I/O request packet

I/O-Bound Operations



IRP –
I/O request packet

I/O result is
awaited with IOCP

Benefits

Because there are no threads dedicated to blocking on unfinished tasks, the server thread pool can service a much higher volume of requests.

Because I/O-bound work spends virtually no time on the CPU, dedicating an entire CPU thread to perform barely any useful work would be a poor use of resources.

CPU-Bound Operation

CPU-bound async code is a bit different than I/O-bound async code. Because the work is done on the CPU, there's **no way to get around dedicating a thread** to the computation.

The use of **`async`/`await`** provides you with a clean way to interact with a background thread and keep the caller of the async method responsive.

Note that this does not provide any protection for shared data. If you are using shared data, you will still need to **apply an appropriate synchronization strategy**.

Question

- What are the benefit of using **`async`/`await`** with:
  - I/O-bound operations?
  - CPU-bound operations?

Awaitable/Awaiter pattern

Requirements to the awaitable types:

- It has a **`GetAwaiter()`** method (instance method or extension method);
- Its **`GetAwaiter()`** method returns an awaiter. An object is an awaiter if:
  - It implements **`INotifyCompletion`** or **`ICriticalNotifyCompletion`** interface;
  - It has an **`IsCompleted`**, which has a getter and returns a Boolean;
  - it has a **`GetResult()`** method, which returns void, or a result.

C# supports both **`GetAwaiter()`** *instance* method and *extension* method.

Async ASP.NET MVC actions

When a request arrives, a thread from the pool is dispatched to process that request.

If the request is processed synchronously, the thread that processes the request is busy while the request is being processed, and that thread cannot service another request.

An asynchronous request takes the **same amount of time** to process as a synchronous request. However, a **thread is not blocked** from responding to other requests.

Asynchronous requests prevent **request queuing** and **thread pool growth** when there are many concurrent requests that invoke long-running operations.

Async ASP.NET MVC actions

```
public async Task<ActionResult> GizmosAsync()
{
    ViewBag.SyncOrAsync = "Asynchronous";
    var gizmoService = new GizmoService();
    return View("Gizmos", await gizmoService.GetGizmosAsync());
}
```

Use asynchronous actions for the following conditions:

- The operations are **network-bound** or **I/O-bound** instead of CPU-bound.
- Parallelism is more important than simplicity of code.

Questions

- How **async** actions helps to increase throughput of the system?