Asynchronous Programming

"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism."

— *Edward A. Lee* *(The Problem with Threads, Berkeley 2006)*

Objectives

- Increase speed of code without unwanted side effects
- Control the Immutability and Isolation are your best friends to write concurrent application

Async Programming motivation

- OS threads are **expensive**, while lightweight threading alone is less interoperable.
- Asynchronous programming **using callbacks is difficult.**
- **Unbounded** parallelism – no hardware constraints

Is 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
```

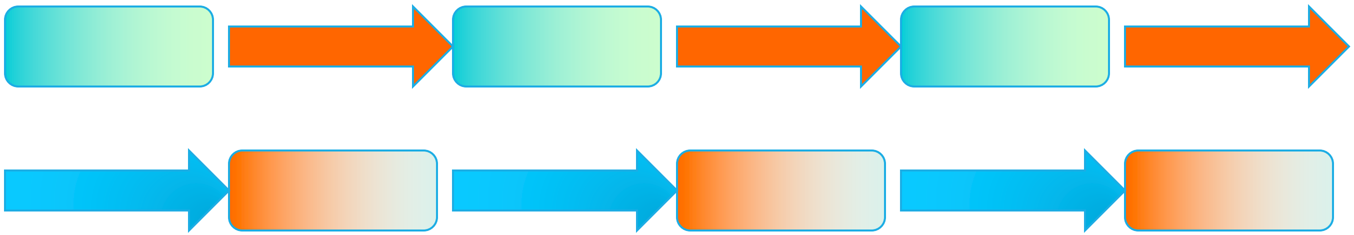
```
Func<string, Task<byte[]>> downloadSiteIcone = async domain =>
{
        var response = await new
                HttpClient().GetAsync($"http://{domain}/favicon.ico");
        return await response.Content.ReadAsByteArrayAsync();
}
```

Classic Synchronous programming

- We are used to writing code linearly

Job 1

Job 2

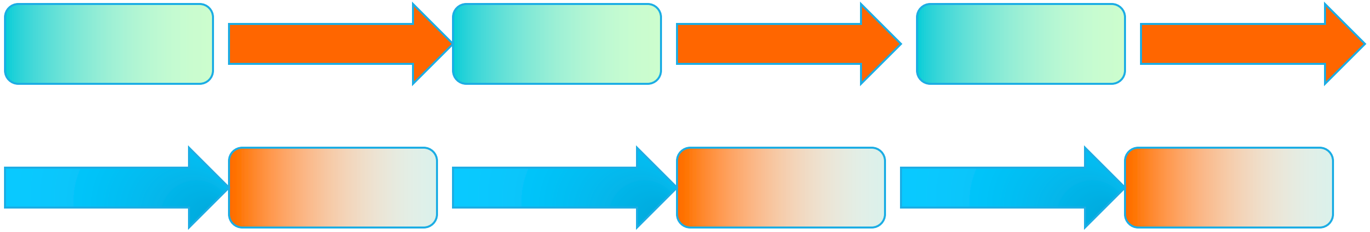


Classic Synchronous programming

- We are used to writing code linearly

Job 1

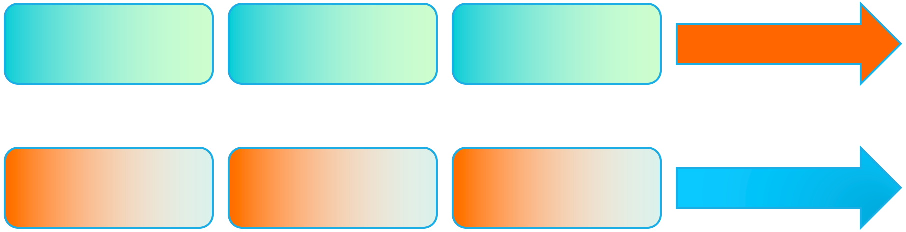
Job 2



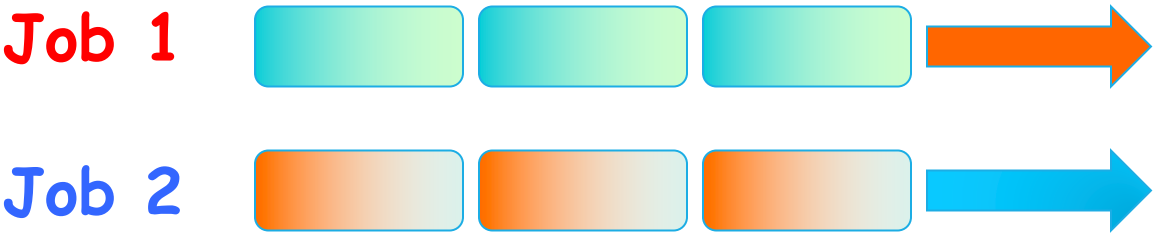
- Jobs executing in parallel

Job 1

Job 2

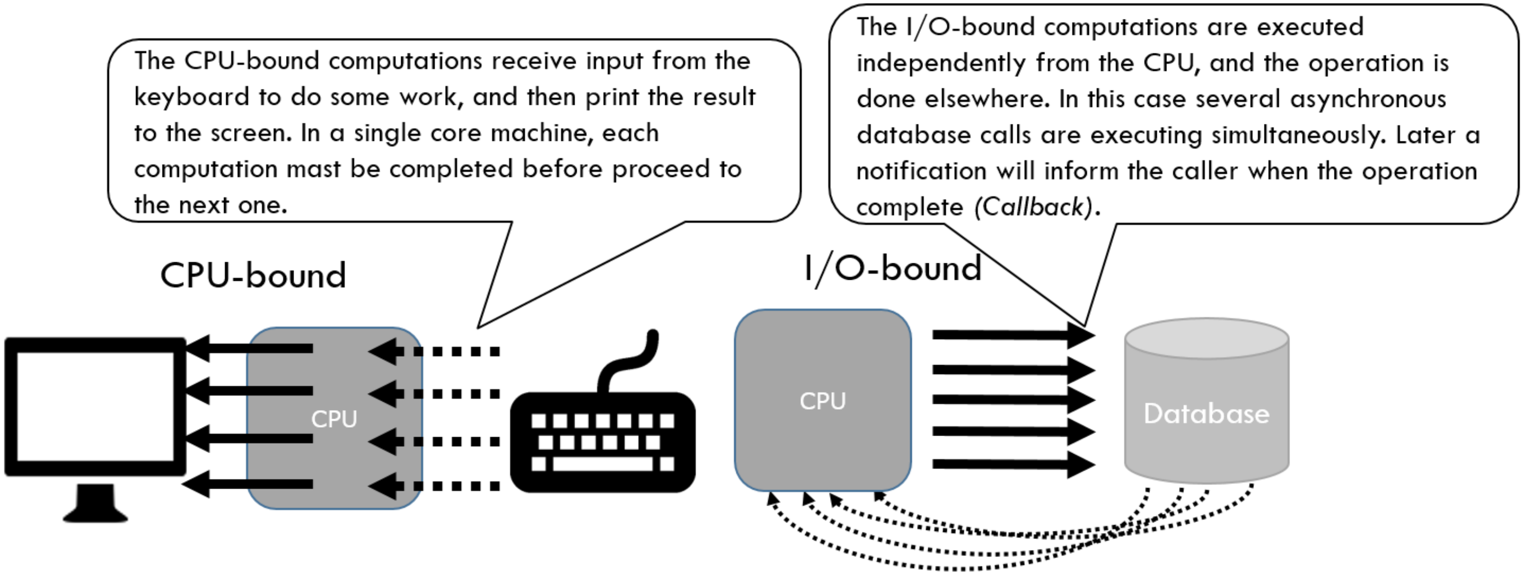


Classic Asynchronous programming



- Classic Asynchronous programming requires decoupling Begin from End
- Very difficult to:
  - Combine multiple asynchronous operations
  - Deal with exceptions, cancellation and transaction

Is all about Scalability



Parallel asynchronous programming

- Parallel vs. asynchronous programming
  - Two different problems
  - C# **Tasks** are good for both
  - F# **Async** mainly for asynchronous programming

- Declarative parallelism
  - Compose tasks to run in parallel

- Task-based parallelism
  - Create tasks and wait for them later

Asynchronous Workflows

- *Software is often I/O-bound, it provides notable performance benefits*
  - Connecting to the Database
  - Leveraging web services
  - Working with data on disk
- *Network and disk speeds increasing slower*
- *Not Easy to predict when the operation will complete* (no-deterministic)
- ***IO bound functions can scale regardless of threads***
  - ***IO bound computations can often "overlap"***
  - ***This can even work for huge numbers of computations***

async/await

C#

```
public async Task PrintThenSleepThenPrint()
{
    Console.WriteLine("before sleep");
    await Task.Delay(5000);
    Console.WriteLine("wake up");
}

await PrintThenSleepThenPrint()
Console.WriteLine("continuing");
```

async workflow

F#

```
let printThenSleepThenPrint = async {
        printfn "before sleep"
        do! Async.Sleep 5000
        printfn "wake up"
    }

Async.StartImmediate printThenSleepThenPrint
printfn "continuing"
```

Async workflows in F# and C#

C# – Return Task and add async and await

```
async Task<int> PageLength(Uri url) {
  var wc = new WebClient();
  var html = await wc.DownloadStringTaskAsync(url);
  return html.Length;
}
```

F# – Wrap in async and use let! keyword

```
let pageLength (url:Uri) = async {
  let wc = new WebClient();
 let! html = wc.AsyncDownloadString(url)
  return html.Length }
```

async workflow

- **let!** – like **`await`** on Task<T> in C#
- **do!** - like **`await`** on Task in C#
- **return!** – like **`return await`** on Task<T> C#

Synchronous Programming

```
let readData path =
  let stream = File.OpenRead(path)
  let data = Array.zeroCreate<byte> stream.Length

let bytesRead = stream.Read(data, 0, data.Length)
```

- Blocks thread while waiting
  - Does not scale
  - Blocking user interface – when run on GUI thread
  - Simple to write – loops, exception handling etc

Anatomy of Async Workflows

```
let readData path = async {
   let stream = File.OpenRead(path)
   let! data = stream.AsyncRead(stream.Length)
   return data }
```

- Easy transition from synchronous
  - Wrap in asynchronous workflow with the ***async*** keyword, use **let!** for async calls and add **return**
  - No need of explicit callback
  - Easy to debug
- Supports loops, recursion, exceptions, cancellation, resource management
- **Operation complete in the same scope**

Anatomy of Async Workflows

```
let readData path : Async<byte[]> = async {
   let stream = File.OpenRead(path)
   let! data = stream.AsyncRead(stream.Length)
   return data }
```

- Async defines a block of code which execute on demand
- Easy to compose

Async Composition

```
let readData path = async {  // string -> Async<byte[]>
      let stream = File.OpenRead(path)
      return! stream.AsyncRead(stream.Length) }
```

```
string -> byte[] -> Async<unit>
let writeData path data = async {
      let stream = File.OpenWrite(path)
      do! stream.AsyncWrite(data, 0, data.Length) }
```

```
// string -> string -> Async<unit>
let copyFile source destination = async {
      let! data = readData source
      do!  writeData destination data }
```

Async Composition

```
let readData path = async {  // string -> Async<byte[]>
      let stream = File.OpenRead(path)
      return! stream.AsyncRead(stream.Length) }
```

```
string -> Async<byte[]> -> Async<unit>
let writeData path getData = async {
      let! data = getData
      let stream = File.OpenWrite(path)
      do! stream.AsyncWrite(data, 0, data.Length) }
```

```
// string -> string -> Async<unit>
let copyFile source destination : Async<unit> =
   readData source |> writeData destination // Async.Start
```

Parallelizing Async

Parallel composition of workflows

```
let! docs = [ for url in urls -> downloadPage url ]
            |> Async.Parallel
```

Task-based parallelism

```
async {
  let! dp1 = Async.StartChild(downloadPage(url1))
  let! dp2 = Async.StartChild(downloadPage(url2))
  let! page1 = dp1
  let! page2 = dp2 } |> Async.Start
```

Async and parallel programming

F#

```
let parallel2 (job1, job2) =
    async {
        let! task1 = Async.StartChild job1
        let! task2 = Async.StartChild job2
        let! res1 = task1
        let! res2 = task2
        return (res1, res2) }
```

Asynchronous Workflows - Continuations

```
let token = new CancellationTokenSource()
let continuation result =
    printfn "Async operation completed: %A" result
let exceptionContintuaion (ex:exn) =
    printfn "Exception thrown: %s" ex.Message
let cancellationContinuation (cancel:OperationCanceledException) =
    printfn "Async operation cancelled"
```

```
Async.StartWithContinuations(readFileAsynchronous,
        continuation, // ‘a -> unit
        exceptionContinuation, // exn -> unit
        cancellationContinuation,  // opc -> unit
        token.Token)
```

Exceptions & Parallel

Creates an asynchronous computation that executes all the given asynchronous computations queueing each as work items and using a fork/join pattern.

```
let rec fib x = if x <= 2 then 1 else fib(x-1) + fib(x-2)

let fibs =
  Async.Parallel[for i in 0..40 -> async { return fib(i) }]
   |> Async.Catch
   |> Async.RunSynchronously
   |> function
     | Choice1Of2 result > printfn "Successfully %A" result
     | Choice2Of2 exn -> printfn "Exception occured %s" exn.Message
```

Declarative parallelism

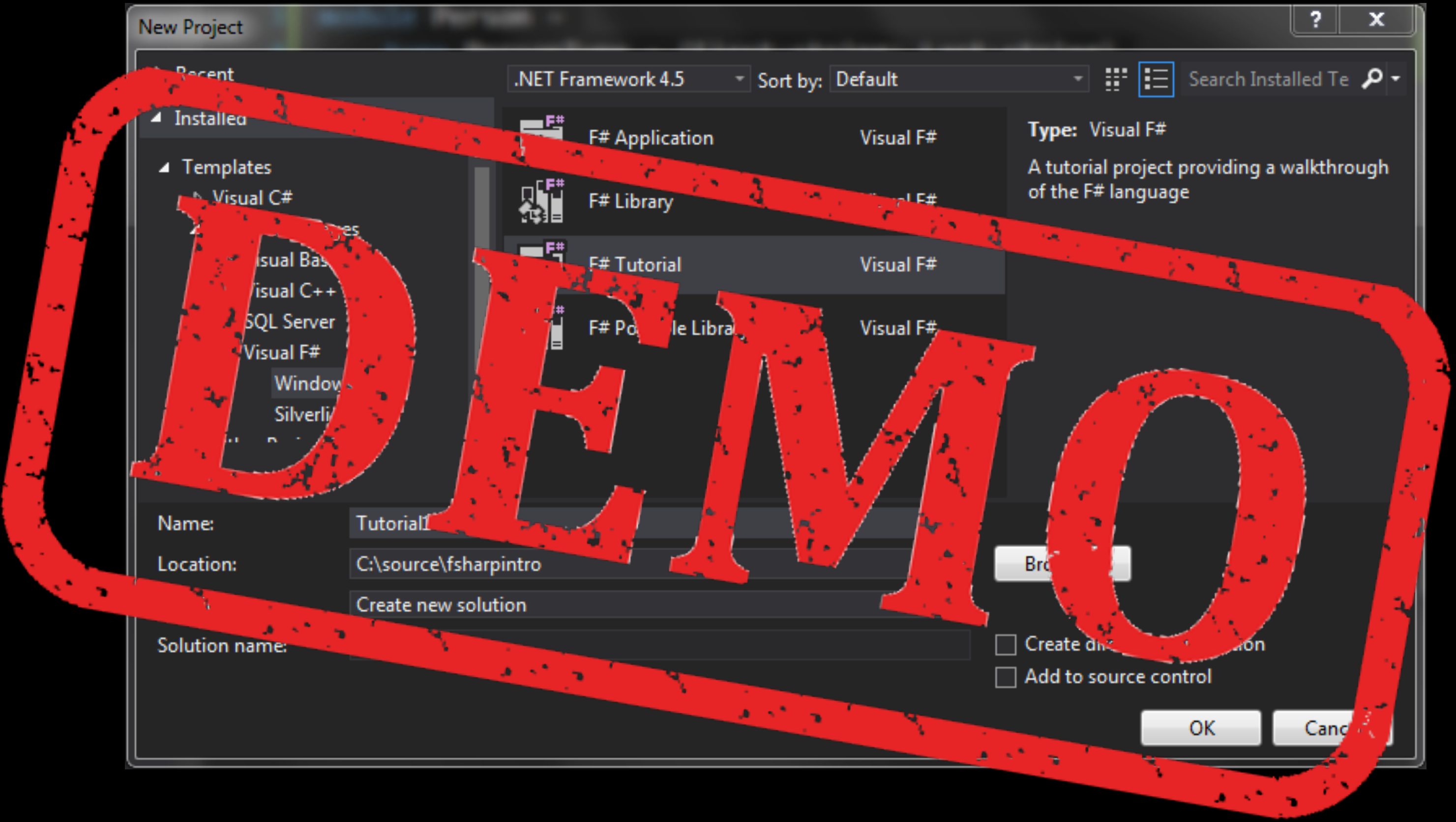
Run in parallel and wait for completion

```
var docs = await Task.WhenAll
  (from url in pages select DownloadPage(url));
```

```
let! docs =
 Async.Parallel [ for url in urls -> downloadPage url ]
```

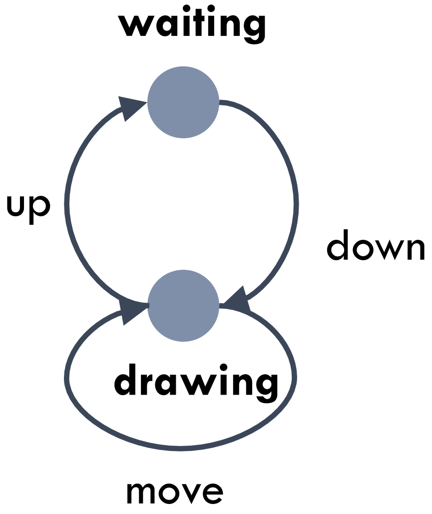
Functional approach

- Works nicely with F# sequences and LINQ


New Project
Recent
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
.NET Framework 4.5
Sort by:
Default
Search Installed Te
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
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

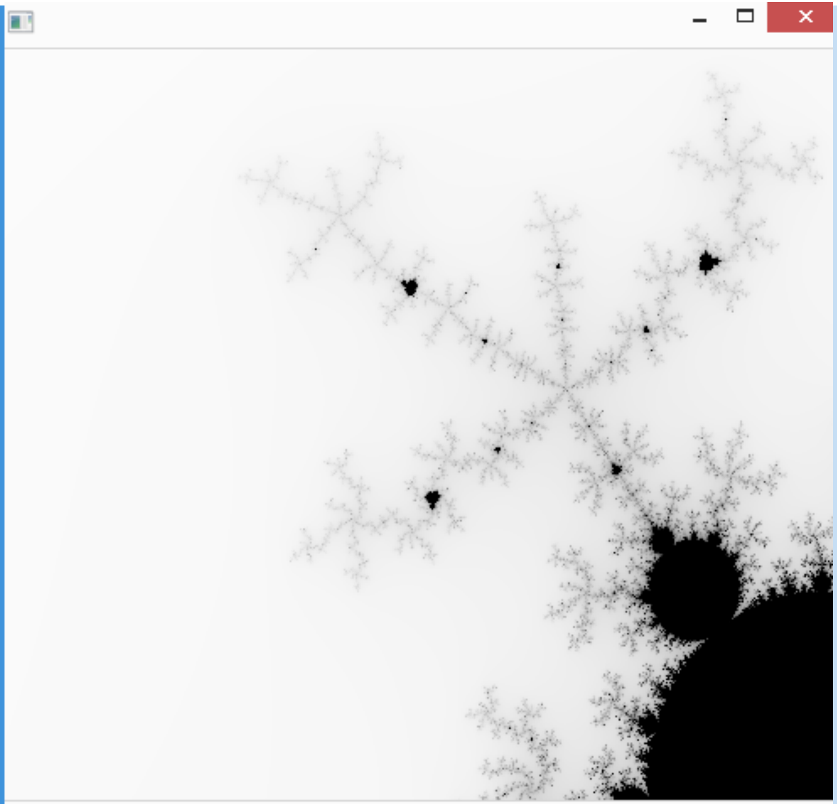
Fractal-Zoom

Fractal-Zoom




```
let rec waiting() = async {
    let! md = Async.AwaitObservable(self.MouseDown)
    // ...
   do! drawing(rc, md.GetPosition(canvas)) }
and drawing(rc:Canvas, pos) = async {
   let! evt = Async.AwaitObservable(canvas.MouseUp,
                                     canvas.MouseMove)

   match evt with
   | Choice1Of2(up) ->
          // ...
          do! waiting()
   | Choice2Of2(move) ->
          // ...
          do! drawing(rc, pos) }
do  waiting() |> Async.StartImmediate
```

Fractal-Zoom



Tasks

1. Convert current functionality to run in parallel
2. Implement Zoom Out

Task async/await is a monad

Task async/await is a monad

```
static Task<T> Return<T>(T task)=> Task.FromResult(task);


static async Task<R> Bind<T, R>(this Task<T> task, Func<T, Task<R>> cont)
    => await cont(await task.ConfigureAwait(false)).ConfigureAwait(false);


static async Task<R> Map<T, R>(this Task<T> task, Func<T, R> map)
=> map(await task.ConfigureAwait(false));
```

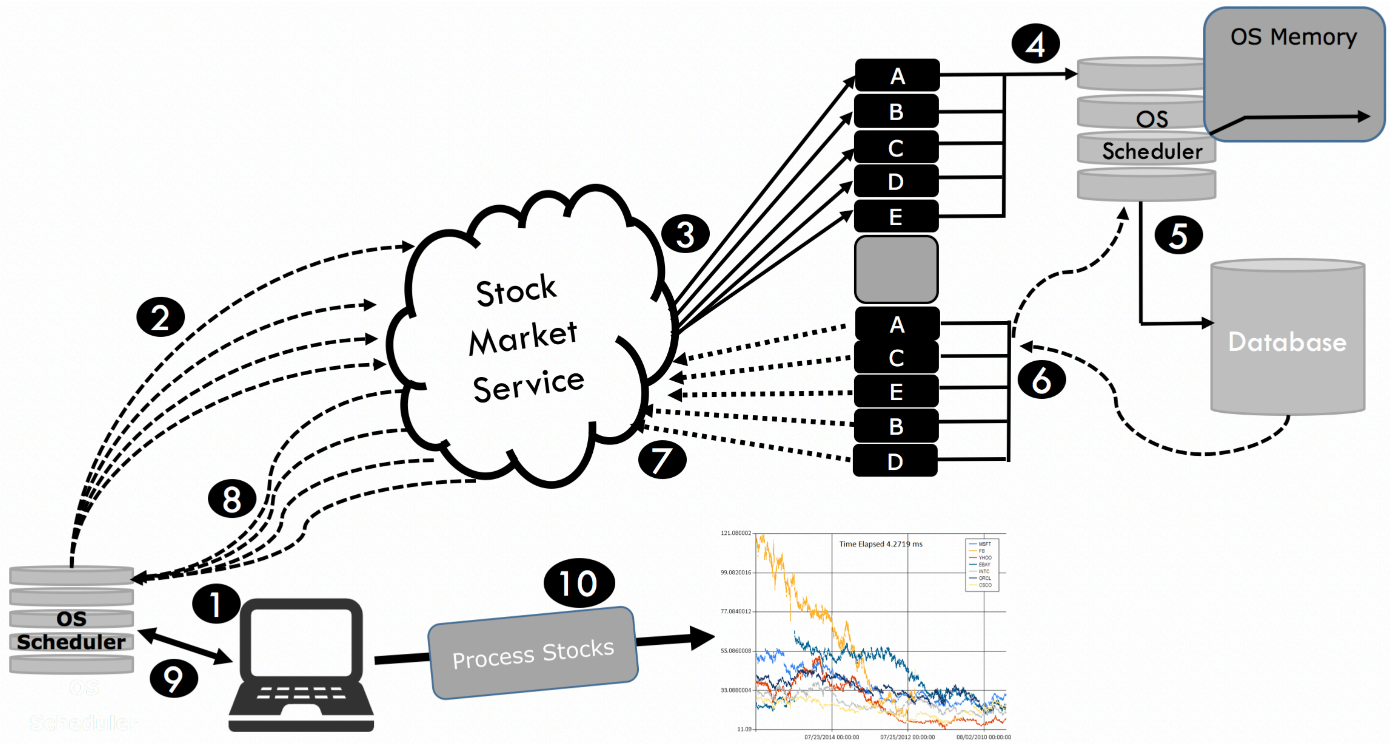
Task async/await is a monad

```
static async Task<R> SelectMany<T, R>(this Task<T> task,
Func<T, Task<R>> then) => await Bind(await task);

static async Task<R> SelectMany<T1, T2, R>(this Task<T1> task,
Func<T1, Task<T2>> bind, Func<T1, T2, R> project)
    {
        T taskResult = await task;
        return project(taskResult, await bind(taskResult));
    }
static async Task<R> Select<T, R>(this Task<T> task, Func<T, R> project)
        => await Map(task, project);

static async Task<R> Return<R>(R value) => Task.FromResult(value);
```

Lab

- run multiple asynchronous operations in parallel
- process the result as complete

Asynchronous Parallel Stock-Tickers



Error handling in functional way

Error handling in functional way

```
module Option =
    let ofChoice choice =
        match choice with
        | Choice1Of2 value -> Some value
        | Choice2Of2 _ -> None

module AsyncOption =
        let handler (operation:Async<'a>) : AsyncOption<'a> = async {
            let! result = Async.Catch operation
            return (Option.ofChoice result)
        }
```

Error handling in functional way

```
let downloadAsyncImage(blobReference:string) : Async<Image> = async {
        let! container = Helpers.getCloudBlobContainerAsync()
        let blockBlob = container.GetBlockBlobReference(blobReference)
        use memStream = new MemoryStream()
        do! blockBlob.DownloadToStreamAsync(memStream)
        return Bitmap.FromStream(memStream)
    }


downloadAsyncImage "Bugghina001.jpg"
|> AsyncOption.handler
|> Async.map(fun imageOpt ->
    match imageOpt with
    | Some(image) -> image.Save("ImageFolder\Bugghina.jpg")
    | None -> log "There was a problem downloading the image")
|> Async.Start
```

Preserving Exception semantic - Result Type

```
type Result<'TSuccess,'TFailure> =
 | Success of 'TSuccess
 | Failure of 'Tfailure

module Result =
        let ofChoice value =
            match value with
            | Choice1Of2 value -> Success value
            | Choice2Of2 e -> Failure e

module AsyncResult =
             let handler (operation:Async<'a>) : AsyncResult<'a> = async {
             let! result = Async.Catch operation
             return (Result.ofChoice result)     }
```

AsyncResult monadic operators

```
module AsyncResult =
    let retn (value:'a) : AsyncResult<'a> =  value |> Ok |> async.Return

    let map (selector : 'a -> 'b) (asyncResult : AsyncResult<'a>) = async {
        let! result = asyncResult
        match result with
        | Ok x -> return selector x |> handler
        | Error err -> return (Error err)     }

    let bind (selector : 'a -> AsyncResult<'b>) (asyncResult : AsyncResult<'a>) = async {
        let! result = asyncResult
        match result with
        | Ok x -> return! selector x |> handler
        | Error err -> return Error err    }
```

AsyncResult in action

```
let processImage(blobReference:string) (destinationImage:string) : AsyncResult<unit> =
        async {
            let storageAccount = CloudStorageAccount.Parse("< Azure Coonnection >")
            let blobClient = storageAccount.CreateCloudBlobClient()
            let container = blobClient.GetContainerReference("Media")
            let! _ = container.CreateIfNotExistsAsync()
            let blockBlob = container.GetBlockBlobReference(blobReference)
            use memStream = new MemoryStream()
            do! blockBlob.DownloadToStreamAsync(memStream)
            return Bitmap.FromStream(memStream) }
            |> AsyncResult.handler
            |> AsyncResult.bind(fun image -> toThumbnail(image))
            |> AsyncResult.map(fun image -> toByteArray(image))
            |> AsyncResult.bimap
                        (fun bytes -> FileEx.WriteAllBytesAsync(destinationImage, bytes))
                         (fun ex -> logger.Error(ex) |> AsyncResult.retn)
```

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra