

Asynchronous Parallel Programming – Part I

Radu Nicolescu
Department of Computer Science
University of Auckland

12 March 2019

15 March 2019

- ① Goals
- ② Motivation: Sync vs Async
- ③ Asynchronous Workflows
- ④ Async ! Examples
- ⑤ Sample thread allocation to asyncs
- ⑥ Fetch Html Fork/Join
- ⑦ Async children
- ⑧ Switching threads
- ⑨ F# Summary
- ⑩ C# Summary

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary

Goals

- In a nutshell: `async = no blocking wait`

The `async` framework extends the `task` based parallel framework by providing “better” `no-thread-wait` support for `asynchronous` operations

- As a side effect, the `async` framework itself offers support for the `actor` programming model, which adds support for `interaction` between parallel `actors`, via `messaging`
- We focus at the `async` style offered by a functional language, F#, but similar patterns have recently become accessible in C# and all .NET languages: `async/await`, ... and practically all modern languages!

Goals

- In a nutshell: `async = no blocking wait`

The `async` framework extends the `task` based parallel framework by providing “better” `no-thread-wait` support for `asynchronous` operations

- As a side effect, the `async` framework itself offers support for the `actor` programming model, which adds support for `interaction` between parallel `actors`, via `messaging`
- We focus at the `async` style offered by a functional language, F#, but similar patterns have recently become accessible in C# and all .NET languages: `async/await`, ... and practically all modern languages!

Goals

- In a nutshell: `async = no blocking wait`

The `async` framework extends the `task` based parallel framework by providing “better” `no-thread-wait` support for `asynchronous` operations

- As a side effect, the `async` framework itself offers support for the `actor` programming model, which adds support for `interaction` between parallel `actors`, via `messaging`
- We focus at the `async` style offered by a functional language, F#, but similar patterns have recently become accessible in C# and all .NET languages: `async/await`, ... and practically all modern languages!

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary

synchronous vs asynchronous

- A **synchronous** call **waits** until the operation completes, by **blocking the current thread** (even if the result is not immediately needed)
- Scenarios – w/o parallelism:
 - wait until an external operation completes (file, network, web, database)
 - wait for a given time interval
- Scenarios – w/ parallelism:
 - wait for a message from another thread (call, response)
 - wait for a specific condition (signal, free resource)

synchronous vs asynchronous

- A **synchronous** call **waits** until the operation completes, by **blocking the current thread** (even if the result is not immediately needed)
- Scenarios – w/o parallelism:
 - wait until an external operation completes (file, network, web, database)
 - wait for a given time interval
- Scenarios – w/ parallelism:
 - wait for a message from another thread (call, response)
 - wait for a specific condition (signal, free resource)

synchronous vs asynchronous

- A **synchronous** call **waits** until the operation completes, by **blocking the current thread** (even if the result is not immediately needed)
- Scenarios – w/o parallelism:
 - wait until an external operation completes (file, network, web, database)
 - wait for a given time interval
- Scenarios – w/ parallelism:
 - wait for a message from another thread (call, response)
 - wait for a specific condition (signal, free resource)

synchronous vs asynchronous

- An **asynchronous** call does NOT wait until the operation completes, it returns immediately, without blocking the current thread
 - the returning thread may do some other useful stuff in your program or return to the thread pool
- The result of an **asynchronously** started operation can be picked up later
 - classical: by registering to receive **notifications** (callbacks, events) when the operation completes
 - modern: by **awaiting** (async waiting) for the completion of the operation (later, when the result is really needed)

synchronous vs asynchronous

- An **asynchronous** call does NOT wait until the operation completes, it returns immediately, without blocking the current thread
 - the returning thread may do some other useful stuff in your program or return to the thread pool
- The result of an **asynchronously** started operation can be picked up later
 - classical: by registering to receive **notifications** (callbacks, events) when the operation completes
 - modern: by **awaiting** (async waiting) for the completion of the operation (later, when the result is really needed)

A typical **synchronous** call

- A synchronous call follows a straightforward code pattern (in one single block):
 - prepare the call
 - initiate the call, **wait** for its completion, pick up results
 - process results or errors
- However, this pattern wastes threads... which are **blocked**, waiting for the call completion

A typical **synchronous** call

- A synchronous call follows a straightforward code pattern (in one single block):
 - prepare the call
 - initiate the call, **wait** for its completion, pick up results
 - process results or errors
- However, this pattern wastes threads... which are **blocked**, waiting for the call completion

A **synchronous** call in F#

```
1  let fetchSync (name:string , url:string) : string*int =
2      try
3          let uri = new System.Uri(url)
4          let webClient = new WebClient()
5
6          // blocking wait
7          let html = webClient.DownloadString(uri)
8
9          name, html.Length
10
11     with ex ->
12         printfn "*** %s" ex.Message
13         name, 0
```

```
1  (" Microsoft" , " http://www.microsoft.com/" )
2  ⇒ (" Microsoft" , 1020)
```

A traditional **asynchronous** call

- Complicated code sequence (two separate blocks):
 - ① first block (executed in the initial thread, possibly GUI)
 - prepare the call
 - initiate the call
 - ② second block ("callback", possibly executed in another thread)
 - pick up results
 - process results or errors
- Achieves efficiency (better thread management), but at the expense of logical clarity
- Typically, the thread executing the second block is triggered by the so-called IOCP (IO Port Completion) thread, a dedicated thread which watches the progress of IO operations

A traditional **asynchronous** call

- Complicated code sequence (two separate blocks):
 - ① first block (executed in the initial thread, possibly GUI)
 - prepare the call
 - initiate the call
 - ② second block (“callback”, possibly executed in another thread)
 - pick up results
 - process results or errors
- Achieves efficiency (better thread management), but at the expense of logical clarity
- Typically, the thread executing the second block is triggered by the so-called IOCP (IO Port Completion) thread, a dedicated thread which watches the progress of IO operations

A traditional **asynchronous** call

- Complicated code sequence (two separate blocks):
 - ① first block (executed in the initial thread, possibly GUI)
 - prepare the call
 - initiate the call
 - ② second block (“callback”, possibly executed in another thread)
 - pick up results
 - process results or errors
- Achieves efficiency (better thread management), but at the expense of logical clarity
- Typically, the thread executing the second block is triggered by the so-called IOCP (IO Port Completion) thread, a dedicated thread which watches the progress of IO operations

A traditional **asynchronous** call

- Complicated code sequence (two separate blocks):
 - ① first block (executed in the initial thread, possibly GUI)
 - prepare the call
 - initiate the call
 - ② second block (“callback”, possibly executed in another thread)
 - pick up results
 - process results or errors
- Achieves efficiency (better thread management), but at the expense of logical clarity
- Typically, the thread executing the second block is triggered by the so-called IOCP (IO Port Completion) thread, a dedicated thread which watches the progress of IO operations

A traditional **asynchronous** call in F#, part 1.2

```
1 let fetchAsyncOld (name:string , url:string) : unit =
2     let uri = new System.Uri(url)
3     let webClient = new WebClient()
4
5     // register part (2) as event handler
6     webClient.DownloadStringCompleted.AddHandler
7         (new DownloadStringCompletedEventHandler
8             (DownloadStringHandler))
9
10    // just start download, no wait
11    webClient.DownloadStringAsync (uri , name)
```

- Quiz: Why do we need **name** in `DownloadStringAsync`?
- Hint: See the sync version and next slide.

A traditional **asynchronous** call in F#, part 1.2

```
1 let fetchAsyncOld (name:string , url:string) : unit =
2     let uri = new System.Uri(url)
3     let webClient = new WebClient()
4
5     // register part (2) as event handler
6     webClient.DownloadStringCompleted.AddHandler
7         (new DownloadStringCompletedEventHandler
8             (DownloadStringHandler))
9
10    // just start download, no wait
11    webClient.DownloadStringAsync (uri , name)
```

- Quiz: Why do we need **name** in DownloadStringAsync?
- Hint: See the sync version and next slide.

A traditional **asynchronous** call in F#, part 2.2

```
1  let DownloadStringHandler
2      (sender: obj)
3      (e: DownloadStringCompletedEventArgs)
4      : unit =
5      let ex = e.Error
6
7      if ex = null then
8          let html = string e.Result
9          let name = string e.UserState
10
11         // event handlers must return unit = void , so
12         // can print the html or its length, but NOT return these
13
14     else
15         printfn " *** %s" ex.Message
```

- Messy way to obtain the **result** and recover **name**

Conclusions – callback hell

- Although efficient, this approach is not very encouraging...
- Will be difficult to scale on complex scenarios, requiring several parallel asynchronous downloads...
- Can we combine the logical clarity of the synchronous version with the efficiency of the asynchronous version?
- Solution: the **async** framework, available first in F#, then in C#, ... and now in almost all modern languages!

Conclusions – **callback hell**

- Although efficient, this approach is not very encouraging...
- Will be difficult to scale on complex scenarios, requiring several parallel asynchronous downloads...
- Can we combine the logical clarity of the synchronous version with the efficiency of the asynchronous version?
- Solution: the **async** framework, available first in F#, then in C#, ... and now in almost all modern languages!

F# async download

```
1 let fetchAsync (name:string, url:string) :  
2   Async<string*int> =  
3   async { // novel  
4     try  
5       let uri = new System.Uri(url)  
6       let webClient = new WebClient()  
7  
8       // no blocking wait — just 2 changes: let! Async...  
9       let! html = webClient.AsyncDownloadString(uri)  
10  
11      // printfn ...  
12      return name, html.Length  
13  
14      with ex ->  
15        printfn "*** %s" ex.Message  
16        return name, 0  
17    }
```

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows**
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary

F# async block in nutshell – typical **sugared monad**

```

1  async {
2      ...           // F# looking constructs
3      let! ... = ... // do!
4      ...           // F# looking constructs
5      return ...   // return!
6  }
```

- Obvious novelty: `async {...}` block: `async` is a class **instance!**
- Other novelties: `let!`, `do!`, `return`, `return!...` **only inside** `{ }...` implemented as **function calls** to `async`'s methods
- Less obvious: *F# looking constructs* (e.g. `if`, `while`, ...) are not exactly ordinary *F#...* also implemented as **function calls**
- Braces surround a **DSL** (Domain Specific Language)... cf. **LOP** (Language Oriented Programming)

F# async block in nutshell – typical **sugared monad**

```
1  async {  
2      ...           // F# looking constructs  
3      let! ... = ... // do!  
4      ...           // F# looking constructs  
5      return ...    // return!  
6  }
```

- Obvious novelty: **async** {...} block: **async** is a class **instance!**
- Other novelties: **let!**, **do!**, **return**, **return!...** **only inside** { }... implemented as **function calls** to **async's** methods
- Less obvious: F# looking constructs (e.g. **if**, **while**, ...) are not exactly ordinary F#... also implemented as **function calls**
- Braces surround a **DSL** (Domain Specific Language)... cf. **LOP** (Language Oriented Programming)

F# async block in nutshell – typical **sugared monad**

```

1  async {
2      ...           // F# looking constructs
3      let! ... = ... // do!
4      ...           // F# looking constructs
5      return ...    // return!
6  }
```

- Obvious novelty: **async** {...} block: **async** is a class **instance!**
- Other novelties: **let!**, **do!**, **return**, **return!**... **only inside** { }... implemented as **function calls** to **async**'s methods
- Less obvious: F# looking constructs (e.g. **if**, **while**, ...) are not exactly ordinary F#... also implemented as **function calls**
- Braces surround a **DSL** (Domain Specific Language)... cf. **LOP** (Language Oriented Programming)

F# async block in nutshell – typical **sugared monad**

```
1  async {  
2      ...           // F# looking constructs  
3      let! ... = ... // do!  
4      ...           // F# looking constructs  
5      return ...    // return!  
6  }
```

- Obvious novelty: **async** {...} block: **async** is a class **instance!**
- Other novelties: **let!**, **do!**, **return**, **return!**... **only inside** { }... implemented as **function calls** to **async**'s methods
- Less obvious: F# looking constructs (e.g. **if**, **while**, ...) are not exactly ordinary F#... also implemented as **function calls**
- Braces surround a **DSL** (Domain Specific Language)... cf. **LOP** (Language Oriented Programming)

F# async block in nutshell – typical **sugared monad**

```
1  async {  
2      ...           // F# looking constructs  
3      let! ... = ... // do!  
4      ...           // F# looking constructs  
5      return ...   // return!  
6  }
```

- Obvious novelty: **async** {...} block: **async** is a class **instance!**
- Other novelties: **let!**, **do!**, **return**, **return!**... **only inside** { }... implemented as **function calls** to **async**'s methods
- Less obvious: F# looking constructs (e.g. **if**, **while**, ...) are not exactly ordinary F#... also implemented as **function calls**
- Braces surround a **DSL** (Domain Specific Language)... cf. **LOP** (Language Oriented Programming)

F# async block in nutshell – w/ types explicated

```

1  async {           // Async<'T>, i.e. wraps a result ∈ 'T
2     let! y = Y      // y ∈ 'U, Y ∈ Async<'U>
3     do! Y           // Y ∈ Async<unit>... unit ≈ void
4     ...
5     return x       // end with a wrapped x ∈ 'T
6     ...
7     return! Z      // continue with Z ∈ Async<'T>
8                       // async's equivalent of tail call
9 }

```

- A good implementation of **return!** will use **trampolines** to optimise the tail calls!
- **trampolines** can be used to ad-hoc optimise tail calls in languages which do not properly support this!

F# async block in nutshell – w/ types explicit

```

1  async {           // Async<'T>, i.e. wraps a result ∈ 'T
2     let! y = Y      // y ∈ 'U, Y ∈ Async<'U>
3     do! Y           // Y ∈ Async<unit>... unit ≈ void
4     ...
5     return x       // end with a wrapped x ∈ 'T
6     ...
7     return! Z      // continue with Z ∈ Async<'T>
8                       // async's equivalent of tail call
9 }

```

- A good implementation of **return!** will use **trampolines** to optimise the tail calls!
- **trampolines** can be used to ad-hoc optimise tail calls in languages which do not properly support this!

Async Workflows

- The `Async<'T>` generic type : reifies async workflows (`<'T>` is the result type)
- The `Async` module : static functions which support async workflows, including conversions and utilities for classical types such as `Tasks`
- The `AsyncBuilder` generic class – singleton pattern
- The `async` keyword : the predefined instance of `AsyncBuilder`, for building user-friendly sugared monads (workflows)

```
1 let async = AsyncBuilder () // predefined
```

Async Workflows

- The **Async<'T> generic type** : reifies async workflows (<'T> is the result type)
- The **Async module** : static functions which support async workflows, including conversions and utilities for classical types such as Tasks
- The **AsyncBuilder generic class** – singleton pattern
- The **async keyword** : the predefined instance of AsyncBuilder, for building user-friendly sugared monads (workflows)

```
1 let async = AsyncBuilder () // predefined
```

Async Workflows

- The **Async<'T> generic type** : reifies async workflows (<'T> is the result type)
- The **Async module** : static functions which support async workflows, including conversions and utilities for classical types such as Tasks
- The **AsyncBuilder generic class** – singleton pattern
- The **async keyword** : **the** predefined instance of AsyncBuilder, for building user-friendly sugared monads (workflows)

```
1 let async = AsyncBuilder () // predefined
```

Async workflow (monad) – main topics

- How to define an async workflow?
- How to wrap a computation into an async workflow?
- How to start an async workflow?
- How to cancel an async workflow?
- How to timeout an async workflow?
- How to await for the result an async workflow (or its exception)?
- How to unwrap the result an async workflow?
- How to use these via sugared asyncs or directly via function calls?

Async workflow (monad) – readings

- Async module

`https:`

`//msdn.microsoft.com/en-us/visualfsharpdocs/
conceptual/control.async-class-%5Bfsharp%5D`

- AsyncBuilder class

`https://msdn.microsoft.com/en-us/
visualfsharpdocs/conceptual/control.
asyncbuilder-class-%5bfsharp%5d`

Async (return)

- The following expression constructs an `async` which, when invoked, will immediately return the given value

```
1 let m = async { return 3 }
```

- The above workflow is defined, but not yet started!
- This construct wraps a base value (of type `int`) into an `async` block (of type `Async<int>`)
- More general and **generic** scenario

```
1 let r = fun t -> async { return t }  
2 r: 'T -> Async<'T>
```

Async RunSynchronously

- How to start and then get the result value of an async?
- `Async.RunSynchronously` does exactly this, but ...
- ... but makes a **blocking wait**, until the async completes
- so only useful in specific contexts (e.g. top-level, fork/join)

```
1 // RunSynchronously : Async<'T> -> 'T
2
3 let t =
4     Async.RunSynchronously (async { return 3 })
5
6 // async { return 3 } -> 3
```

- Can use `Async.RunSynchronously` at the top-level (main), to transit from the sync to the async world
- Good practice: do not mix styles – **once async, always async**

Async let! – with return

- **let!** implements a **non-blocking wait + unwrap**
- ... but can only be used inside **async** blocks

```
1  let m = async { return 3 }
2
3  let r =
4      async {
5          let! t = m
6          return t + 1
7      }
```

- r is not started yet
- When started, r starts m, then **async** waits for its result (3)
- Finally, r wraps this result as an async result (4)

```
1  // async { return 3 } → async { return 4 }
```

Async let! – with return

- **let!** implements a **non-blocking wait + unwrap**
- ... but can only be used inside **async** blocks

```
1  let m = async { return 3 }
2
3  let r =
4      async {
5          let! t = m
6          return t + 1
7      }
```

- r is not started yet
- When started, r starts m, then **async** waits for its result (3)
- Finally, r wraps this result as an async result (4)

```
1  // async { return 3 } → async { return 4 }
```

Async let! – with return!

- Similar, more complex scenario

```
1  let m = async { return 3 }
2
3  let r =
4      async {
5          let! t = m
6          return! async { return t + 1 }
7      }
```

```
1  // async { return 3 } → async { return 4 }
```

- `return!` indicates an `async tail call`
- `return!` often used to implement `async tail recursions`
- `return!` typical functional `async` way to implement `loops`

Async let! – with return!

- Similar, more complex scenario

```
1 let m = async { return 3 }
2
3 let r =
4     async {
5         let! t = m
6         return! async { return t + 1 }
7     }
```

```
1 // async { return 3 } → async { return 4 }
```

- **return!** indicates an **async tail call**
- **return!** often used to implement **async tail recursions**
- **return!** typical functional async way to implement **loops**

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples**
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary

Example **async** block

- A most simple **async** block:

```
1 let async0 = async { return 10+10 }
```

- `async0` has type `Async<int>`
- Deferred evaluation : the **let** assignment does **not** run `async0`'s body

An awaited async run

- **Async.RunSynchronously** : Runs the asynchronous computation, **waits** for its completion (**blocking** call) and **unwraps** its result. (MSDN)

```
1 Async.RunSynchronously : Async<'T> -> 'T
```

```
1 let async0 = async { return 10+10 } // Async<int>
2
3 let res0 = Async.RunSynchronously async0 // 20
4
5 printfn "t0 %d" res0
```

- Although not shown here, `Async.RunSynchronously` has a couple of additional optional parameters, such as a **timeout** and a **cancellation token**

An awaited async run

- **Async.RunSynchronously** : Runs the asynchronous computation, **waits** for its completion (**blocking** call) and **unwraps** its result. (MSDN)

```
1 Async.RunSynchronously : Async<'T> -> 'T
```

```
1 let async0 = async { return 10+10 } // Async<int>
2
3 let res0 = Async.RunSynchronously async0 // 20
4
5 printfn "t0 %d" res0
```

- Although not shown here, `Async.RunSynchronously` has a couple of additional optional parameters, such as a **timeout** and a **cancellation token**

An **async** run with **continuations**

- **Async.StartWithContinuations** : Runs an asynchronous computation, starting immediately on the current operating system thread, but does not wait its completion. Calls one of the three continuations when the operation completes. (MSDN)

```
1 Async.StartWithContinuations
2   (async0 ,
3     (fun n -> printfn "success %d" n) ,
4     (fun x -> printfn "exception %s" x.Message) ,
5     (fun _ -> printfn "cancelled" ) )
```

- Quiz: what are the type and the value of `n` (the parameter of the 1st continuation)

An **async** run with **continuations**

- **Async.StartWithContinuations** : Runs an asynchronous computation, starting immediately on the current operating system thread, but does not wait its completion. Calls one of the three continuations when the operation completes. (MSDN)

```
1 Async.StartWithContinuations
2   (async0 ,
3     (fun n -> printfn "success %d" n) ,
4     (fun x -> printfn "exception %s" x.Message) ,
5     (fun _ -> printfn "cancelled" ) )
```

- Quiz: what are the type and the value of `n` (the parameter of the 1st continuation)

An `async` run as a `.NET Task`

- Essentially, F# `async`'s offer alternate views on classical `tasks`, with a different API: if you need, you can still “cast” and use them as tasks.

```
1 let task0 = Async.StartAsTask async0
2
3 printfn "Do some other work..."
4 ...
5
6 let r1 = task0.Result // task0.Wait()
7
8 printfn "done r1 %d" r1
```

- `Task.Result` : this is a `blocking` call, like `Task.Wait()`

An **async** run as a **.NET Task**

- Essentially, F# **async**'s offer alternate views on classical **tasks**, with a different API: if you need, you can still “cast” and use them as tasks.

```
1 let task0 = Async.StartAsTask async0
2
3 printfn "Do some other work..."
4 ...
5
6 let r1 = task0.Result // task0.Wait()
7
8 printfn "done r1 %d" r1
```

- Task.Result** : this is a **blocking** call, like **Task.Wait()**

A fork/join parallel async run - version 1.2

- **Async.Parallel** : Creates an asynchronous computation that executes in parallel a given sequence of asynchronous computations, initially queueing each as work items and using a fork/join pattern. (MSDN)

```
1 let async1 = async { return 10+10 } //Async<int>
2 let async2 = async { return 20+20 } //Async<int>
3
4 let res = // pipeline style
5     [ async1; async2 ]
6     |> Async.Parallel           // Async<int []>
7     |> Async.RunSynchronously // int []
8
9 printfn "res %A" res           // [|20; 40|]
```

- This is fork/join parallelism between siblings, all started at about the same time

Example **async !** block

- A simulatedJob has its job ID and its simulated sleeping period

```
1  let simulatedJob (id:int) (time:int): Async<int> =
2      async {
3          printfn "+++ [%d] Job %d starts"
4              (tid()) id
5
6          do! Async.Sleep time // ms
7
8          printfn "+++ [%d] Job %d ends %d ms"
9              (tid()) id timespan
10
11         return id
12     }
```

- `tid()` = current thread id; `timespan` = actual “sleeping” time
- `Async.Sleep` simulates a typical asynchronous operation, such as file download, **without blocking** any worker thread

Async.RunSynchronously

- Sync run:

```
1 Async.RunSynchronously (simulatedJob 1000 1000)
```

- Typical output:

```
1 +++ [11] Job 1000 starts  
2 +++ [12] Job 1000 ends 1003 ms
```

async run with continuations

- StartWithContinuations run:

```
1 Async.StartWithContinuations
2   ((simulatedJob 1000 1000),
3    (fun r -> printfn "*** Success result: %d" r),
4    (fun x -> printfn "*** Exception: %s" x.Message),
5    (fun _ -> printfn "*** Cancelled"))
```

- Quiz: what are the parameter of these continuations?
- Typical output:

```
1 +++ [11] Job 1000 starts
2 +++ [17] Job 1000 ends 1013 ms
3 *** [17] Success result: 1000
```

async run as a Task

- StartAsTask run:

```
1  let task =  
2      Async.StartAsTask (simulatedJob 2000 2000)  
3  
4  printfn "... Do some other work..."  
5  
6  let r = task.Result // task.Wait()  
7  printfn "... Task result: %d" r
```

- Quiz: what if simulatedJob throws an exception or is canceled?
- Typical output:

```
1  ... [11] Do some other work...  
2  +++ [18] Job 2000 starts  
3  +++ [9] Job 2000 ends 2004 ms  
4  ... [11] Task result: 2000
```

fork/join parallel async run

- Async.Parallel run:

```
1  let r =  
2    [ 3000; 2000; 1000]  
3    |> List.map (fun time ->  
4                  simulatedJob time time)  
5    |> Async.Parallel  
6    |> Async.RunSynchronously
```

- Typical output:

```
1  +++ [16] Job 1000 starts  
2  +++ [13] Job 3000 starts  
3  +++ [17] Job 2000 starts  
4  +++ [18] Job 1000 ends 1014 ms  
5  +++ [16] Job 2000 ends 2014 ms  
6  +++ [13] Job 3000 ends 3014 ms
```

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs**
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary

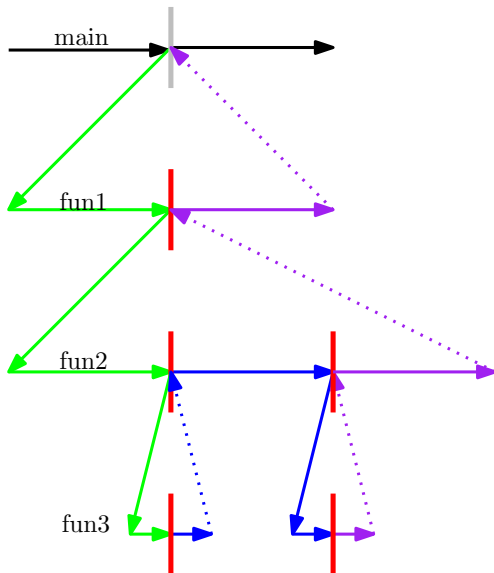
Thread allocation to asyncs – scenario

- Consider:

```
1  let fun3 =
2    async { .3.1. do! Async.Sleep 1000 .3.2. }
3
4  let fun2 =
5    async { .2.1. do! fun3 .2.2. do! fun3 .2.3. }
6
7  let fun1 =
8    async { .1.1. do! fun2 .1.2. }
9
10 do Async.RunSynchronously fun1
```

- No parallelism here, but insightful scenario
- F# and .NET will still use multiple thread-pool threads
- Many2many mapping of functions to threads

Thread allocation to asyncs – typical



Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join**
- 7 Async children
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary

URL List

- Consider a list of (site name, url) tuples:

```
1  let urlList =  
2      [ ("Microsoft", "http://www.microsoft.com/")  
3        ("MSDN", "http://msdn.microsoft.com/")  
4        ("Bing", "http://www.bing.com")  
5      ]
```

- Using the previously defined fetch functions, print a list of pairs, (**name**, **html.Length**)

How to run the sequential fetch

- Using fetchSync – **sequential**

```
1 // fetchSyncAll : unit -> list<string*int>
2 let fetchSyncAll () =
3     urlList // list<string*string>
4     |> List.map fetchSync // list<string*int>
5
6 let s = fetchSyncAll ()
7
8 printfn "... %A" s
```

- Typical output:

```
1 ... [("Microsoft", 1020); ("MSDN", 40698);
2      ("Bing", 88159)]
```

How to run the async fetch in fork/join parallel

- Using fetchSync – **parallel**

```

1 // fetchAsyncAll : unit -> (string*int)[]
2 let fetchAsyncAll () =
3     urlList // list<string*string>
4     |> List.map fetchAsync // list<Async<string*int>>
5     |> Async.Parallel // Async<string*int []>
6     |> Async.RunSynchronously // (string*int)[]
7
8 let z = fetchAsyncAll ()
9 printfn "... %A" z

```

- Typical output:

```

1 ... [| ("Microsoft", 1020); ("MSDN", 40698);
2      ("Bing", 88159) |]

```

- let!** calls in fetchAsync do **not block** – only the main thread waits for the join

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children**
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary

Async children

- Problem: how to start a child async from a parent async and run both in **parallel**
- One possibility: start the child async as a task, via `Async.StartAsTask`, then both parent and child run in parallel, and finally wait for its result using `Task.Result` (which includes `Task.Wait`)
- Let's see how we can do the same, while remaining within the structured F# async framework
- We look at a nice recursive design pattern, that also appears in other scenarios: to wrap the async in another async:

```
1 Async<'T> → Async<Async<'T>>
```

Async children

- Problem: how to start a child async from a parent async and run both in **parallel**
- One possibility: start the child async as a task, via `Async.StartAsTask`, then both parent and child run in parallel, and finally wait for its result using `Task.Result` (which includes `Task.Wait`)
- Let's see how we can do the same, while remaining within the structured F# async framework
- We look at a nice recursive design pattern, that also appears in other scenarios: to wrap the async in another async:

```
1 Async<'T> → Async<Async<'T>>
```

First, a bad solution

```
1 let child = async { // Async<int>
2     ...
3     return 10
4 }
```

```
1 let bad_parent = async {
2     ...
3     let! (res:int) = child // !
4     ...
5 }
```

- parent starts child by submitting it to the thread pool
- no thread is blocked, but the parent does NOT progress until child's result becomes available
- **no** parent/child parallelism

First, a bad solution

```
1 let child = async { // Async<int>
2     ...
3     return 10
4 }
```

```
1 let bad_parent = async {
2     ...
3     let! (res:int) = child // !
4     ...
5 }
```

- parent starts child by submitting it to the thread pool
- no thread is blocked, but the parent does NOT progress until child's result becomes available
- **no** parent/child parallelism

A mixed solution

```
1  let good_parent1 = async {  
2      ...  
3      let (task_child:Task<int>) =  
4          child |> Async.StartAsTask  
5  
6      ... run parent & child ...  
7  
8      let res = task_child.Result // Wait()  
9      ...  
10 }
```

- parent starts child as task
- parent/child parallelism
- but mixed API – and possible problems with exceptions

A mixed solution

```
1  let good_parent1 = async {  
2      ...  
3      let (task_child:Task<int>) =  
4          child |> Async.StartAsTask  
5  
6      ... run parent & child ...  
7  
8      let res = task_child.Result // Wait()  
9      ...  
10 }
```

- parent starts child as task
- parent/child parallelism
- but mixed API – and possible problems with exceptions

A good enough solution

```
1  let good_parent2 = async {  
2      ...  
3      let (async_child : Async<int>) =  
4          child |> Async.StartAsTask  
5              |> Async.AwaitTask  
6  
7      ... run parent & child ...  
8  
9      let! (res : int) = async_child  
10     ...  
11 }
```

- parent starts child as task, wraps it in another async,
- and gets a handle to the running (**hot**) async child
- parent/child parallelism with async API only
- still using task API and parent time for lines 3–6

A good enough solution

```
1  let good_parent2 = async {  
2      ...  
3      let (async_child : Async<int>) =  
4          child |> Async.StartAsTask  
5              |> Async.AwaitTask  
6  
7      ... run parent & child ...  
8  
9      let! (res : int) = async_child  
10     ...  
11 }
```

- parent starts child as task, wraps it in another async,
- and gets a handle to the running (**hot**) async child
- parent/child parallelism with async API only
- still using task API and parent time for lines 3–6

An even better solution

```
1 let good_parent3 = async {  
2     ...  
3     let! (async_child: Async<int>) =  
4         child |> Async.StartChild  
5  
6     ... run parent & child ...  
7  
8     let! (res: int) = async_child  
9     ...  
10 }
```

```
1 Async.StartChild : Async<'T> -> Async<Async<'T>>
```

- parent starts child as a inner async, wrapped in another async,
- and gets a handle to the running (hot) async child
- parent/child parallelism with async API only

An even better solution

```
1 let good_parent3 = async {  
2     ...  
3     let! (async_child : Async<int>) =  
4         child |> Async.StartChild  
5  
6     ... run parent & child ...  
7  
8     let! (res : int) = async_child  
9     ...  
10 }
```

```
1 Async.StartChild : Async<'T> -> Async<Async<'T>>
```

- parent starts child as a inner async, wrapped in another async,
- and gets a handle to the running (**hot**) async child
- parent/child parallelism with async API only

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads**
- 9 F# Summary
- 10 C# Summary

Switching threads

- Important problem, esp. when we need to update the GUI from a background thread
- Problem solved in various complicated ways
 - Java: `SwingWorker`, `SwingUtilities.Invoke...`, ...
 - C#: `Control.Invoke`, C# asynchronous events, ...

Switching threads in an F# form

```
1 let guiContext =
2   SynchronizationContext.Current // if on the GUI thread
3   WindowsFormsSynchronizationContext() // WinForms GUI thread
4
5 async {
6   while true do
7     do! Async.Sleep 1000 // simulate a long computation
8     ...
9     do! Async.SwitchToContext(guiContext)
10    ... // can now update the UI
11
12    do! Async.SwitchToThreadPool()
13    ... // revert to a thread pool thread
14
15 } |> Async.Start // start in a thread pool thread
```

Switching threads in an F# form

- The `SynchronizationContext` is a generalization of `ISynchronizeInvoke` pattern. More about this on MSDN, e.g.:
- `http://msdn.microsoft.com/en-us/magazine/gg598924.aspx`

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads
- 9 **F# Summary**
- 10 C# Summary

F# async summary

- To start an async:
 - `Async.RunSynchronously`, `Async.StartWithContinuations`, `Async.Start` (in thread pool), `Async.StartImmediate` (in current thread), `Async.StartAsTask`, `Async.StartChild`, ...
 - **let!**, **do!** (only from within another async)
 - `AsyncBuilder.Bind` (for cryptic monad fans)
- To continue an async with another async (tail call)
 - **return!** (only in the last position of the first async)
 - `AsyncBuilder.ReturnFrom` (for cryptic monad fans)
- To pick an async result
 - **let!**, **do!**, `AsyncBuilder.Bind`, `Async.StartWithContinuations`, `Async.StartAsTask+Task.Result`, `Async.StartChild+let!+let!`

F# async summary

- To start an async:
 - `Async.RunSynchronously`, `Async.StartWithContinuations`, `Async.Start` (in thread pool), `Async.StartImmediate` (in current thread), `Async.StartAsTask`, `Async.StartChild`, ...
 - **let!**, **do!** (only from within another async)
 - `AsyncBuilder.Bind` (for cryptic monad fans)
- To continue an async with another async (tail call)
 - **return!** (only in the last position of the first async)
 - `AsyncBuilder.ReturnFrom` (for cryptic monad fans)
- To pick an async result
 - **let!**, **do!**, `AsyncBuilder.Bind`, `Async.StartWithContinuations`, `Async.StartAsTask+Task.Result`, `Async.StartChild+let!+let!`

F# async summary

- To start an async:
 - `Async.RunSynchronously`, `Async.StartWithContinuations`, `Async.Start` (in thread pool), `Async.StartImmediate` (in current thread), `Async.StartAsTask`, `Async.StartChild`, ...
 - **let!**, **do!** (only from within another async)
 - `AsyncBuilder.Bind` (for cryptic monad fans)
- To continue an async with another async (tail call)
 - **return!** (only in the last position of the first async)
 - `AsyncBuilder.ReturnFrom` (for cryptic monad fans)
- To pick an async result
 - **let!**, **do!**, `AsyncBuilder.Bind`, `Async.StartWithContinuations`, `Async.StartAsTask+Task.Result`, `Async.StartChild+let!+let!`

F# async summary

- To change the synchronisation context (e.g. to/from GUI)
 - `Async.SwitchToContext`, `Async.SwitchToThreadPool`
 - **let!** and `Async.StartWithContinuations` automatically continue with a thread from the same context (e.g. thread pool or GUI)

F# async summary – caveat

- Async API unwraps all **results** and **exceptions** (and **cancellations**)
 - **let!**, **do!**, StartWithContinuations, RunSynchronously, ...
- Task API is less consistent and may return **exceptions** as **inner exceptions** in AggregateException,
 - Task.Result
- **Morale: prefer pure async styles and APIs**

F# async summary – caveat

- Async API unwraps all **results** and **exceptions** (and **cancellations**)
 - **let!**, **do!**, StartWithContinuations, RunSynchronously, ...
- Task API is less consistent and may return **exceptions** as **inner exceptions** in AggregateException,
 - Task.Result
- **Morale: prefer pure async styles and APIs**

F# async summary – caveat

- Async **cancellations** are triggered via instances of `OperationCanceledException`
- System async methods check for cancellations, but custom code should be **cooperative**
- Async API unwraps all **cancellations**
- Task API is less consistent and may return **cancellations** as either of:
 - an instance of `OperationCanceledException`
 - an instance of `TaskCanceledException`, possibly wrapped as a `AggregateException`
- **Morale:** prefer pure async styles and APIs
- Note: a child exception can escalate and cancel the parent async/task

F# async summary – caveat

- Async **cancellations** are triggered via instances of `OperationCanceledException`
- System async methods check for cancellations, but custom code should be **cooperative**
- Async API unwraps all **cancellations**
- Task API is less consistent and may return **cancellations** as either of:
 - an instance of `OperationCanceledException`
 - an instance of `TaskCanceledException`, possibly wrapped as a `AggregateException`
- **Morale: prefer pure async styles and APIs**
- Note: a child exception can escalate and cancel the parent async/task

F# async summary – caveat

- Async **cancellations** are triggered via instances of `OperationCanceledException`
- System async methods check for cancellations, but custom code should be **cooperative**
- Async API unwraps all **cancellations**
- Task API is less consistent and may return **cancellations** as either of:
 - an instance of `OperationCanceledException`
 - an instance of `TaskCanceledException`, possibly wrapped as a `AggregateException`
- **Morale: prefer pure async styles and APIs**
- Note: a child exception can escalate and cancel the parent `async/task`

Outline

- 1 Goals
- 2 Motivation: Sync vs Async
- 3 Asynchronous Workflows
- 4 Async ! Examples
- 5 Sample thread allocation to asyncs
- 6 Fetch Html Fork/Join
- 7 Async children
- 8 Switching threads
- 9 F# Summary
- 10 C# Summary**

C# .NET async methods – since C# 4.5, 5.0, ...

- The F# approach allows you all benefits of asynchronous programming with almost none of the pain. (MSDN)
- Two new C# keywords: **async** and **await**

```
1 async Task<int> Method (...) { // Async<int> monad
2     ...
3     var x = await FindAsync (...); // let! bind
4     ...
5     return 10;
6 }
```

- **Caveat:** **return int**, but the method returns Task<int>
- Cf. **C# seq iterators:** **yield return int**, but the method returns IEnumerable<int>
- In both cases, monad fingerprints...

C# .NET async methods – since C# 4.5, 5.0, ...

- The F# approach allows you all benefits of asynchronous programming with almost none of the pain. (MSDN)
- Two new C# keywords: **async** and **await**

```
1 async Task<int> Method (...) { // Async<int> monad
2     ...
3     var x = await FindAsync (...); // let! bind
4     ...
5     return 10;
6 }
```

- **Caveat:** **return int**, but the method returns Task<**int**>
- Cf. **C# seq iterators:** **yield return int**, but the method returns IEnumerable<**int**>
- In both cases, monad fingerprints...

A **synchronous** call in C#

```
1 (string, int) FetchSync (string name, string url) {
2     try {
3         Uri uri = new Uri (url);
4         WebClient webClient = new WebClient ();
5
6         // blocking wait!
7         string html = webClient.DownloadString (uri);
8
9         return (name, html.Length);
10
11     } catch (Exception ex) {
12         Console.WriteLine (" {0}", ex.Message);
13         return (name, 0);
14     }
15 }
```

A traditional **asynchronous** call in C#, part 1.2

```
1 void FetchAsyncOld (string name, string url) {
2     Uri uri = new Uri (url);
3     WebClient webClient = new WebClient ();
4
5     webClient.DownloadStringCompleted +=
6         DownloadStringHandler;
7         // register part (2) as event handler
8
9     webClient.DownloadStringAsync (uri , name);
10 }
```

- Quiz: Why do we need **name** in DownloadStringAsync?
- Hint: See the sync version and next slide.

A traditional **asynchronous** call in C#, part 2.2

```
1 void DownloadStringHandler (Object sender ,
2     DownloadStringCompletedEventArgs e) {
3     Exception ex = e.Error;
4
5     if (ex == null) {
6         string html = (string) e.Result;
7         string name = (string) e.UserState;
8
9         return; // (name, html.Length) ?
10
11    } else {
12        Console.WriteLine (" {0}", ex.Message);
13    }
14 }
```

- Observe how “easy” is to obtain the **result** and recover **name**

Download string methods

- `DownloadString`: sync
- `DownloadStringAsync`: traditional async, event-based
- `DownloadStringTaskAsync`: returns a task
 - for C# async
 - could be used by F# async, if combined with `AwaitTask`
- `AsyncDownloadString`: returns an async, for F# async

C# **async** download

```
1 async Task<(string , int)>
2 FetchAsync (string name, string url) {
3     try {
4         Uri uri = new Uri (url);
5         WebClient webClient = new WebClient ();
6
7         string html =
8             await webClient.DownloadStringTaskAsync (uri);
9
10        return (name, html.Length);
11
12    } catch (Exception ex) {
13        Console.WriteLine ("...", ex.Message);
14        return (name, 0);
15    }
16 }
```

Thread allocation to asyncs – similar scenario in C#

```
1  async Task fun3() {
2      ... await Task.Delay(1000); ... }
3
4  async Task fun2() {
5      ... await fun3(); ... await fun3(); ... }
6
7  async Task fun1() {
8      ... await fun2(); ... }
9
10 void Main() {
11     ...
12     Task.Run(() => fun1()).Wait(); // fun1().Wait();
13     ...
14 }
```