Lec 19: Asynchronous Computation

CS220: Programming Principles

Sang Kil Cha

Asynchronous Computation

Asynchrony

Asynchrony means that one or more computations can execute ***independently*** of the main program flow. In other words, the main program flow does not wait for the completion of the asynchronous computations.

Concurrency, Parallelism, and Asynchrony

- ***Concurrency*** is when multiple computations execute in overlapping time periods.
- ***Parallelism*** is when multiple computations run at exactly the same time.
- ***Asynchrony*** is when one or more computations execute separately from the main program flow.

See `https://learn.microsoft.com/en-us/dotnet/fsharp/tutorials/async` for more discussion about the terms.

Synchronous vs. Asynchronous

Synchronous.

```
download "http://example.com/"
// Above computation may take long
handleUserInput ()
```

Asynchronous.

```
downloadAsync "http://example.com/"
// Above immediately returns
handleUserInput ()
```

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don’t accept late responses.

Why Asynchronous?

Maximize the ability of our computation resource. You don't get blocked by a single task.

Suppose we are downloading 10 files $(F_1, F_2, \cdots, F_{10})$ from 10 distinct URLs where each web server has different network throughput.

Asynchronous Computation Everywhere

1. Web browser allows you to navigate web pages while downloading a file.
2. Visual studio captures compilation errors while you are typing your code.
3. Copilot can assist you while you are writing your code.
4. And many more.

How to Detach a Computation from the Main Program?

1. Create an asynchronous computation: `async { expression }`.
2. Run the asynchronous computation with `Async.Start` or similar functions.
3. Then you can continue with the main program flow while the asynchronous computation is running.

Async<'T> type.

An asynchronous computation is represented by the Async<'T> type, which will be evaluated as a value of type 'T at some point in the future.

```
async {
  let ones = Seq.unfold (fun _ -> Some (1, ())) ()
  ones |> Seq.item 1000000 |> printfn "%d"
} |> Async.Start
printfn "This message may appear before the above."
```

Synchronous Wait

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

We can wait for the result of an asynchronous computation using `Async.RunSynchronously`.

```
async {
  let ones = Seq.unfold (fun _ -> Some (1, ())) ()
  ones |> Seq.item 1000000 |> printfn "%d"
} |> Async.RunSynchronously
printfn "This message will appear after the above."
```

Achieving Parallelism

`val Async.Parallel:  seq<Async<’T» -> Async<’T []>`

We can run multiple asynchronous computations in parallel using `Async.Parallel`.

```
[ async { Console.WriteLine "A" }
  async { Console.WriteLine "B" } ]
|> Async.Parallel
|> Async.RunSynchronously
```

Example: Asynchronous Web Browsing

All existing web browsers can download and render multiple web pages in a concurrent manner. The download process is roughly as follows.

1. Send a page request to the target web server.
2. Wait for the server's response.
3. Get the response.

> The entire process should run in the order, although each of the steps is asynchronous!

Let the download process be an async computation (WebClient.AsyncDownloadString). Then we can create another asynchronous computation that internally runs the download computation.

```
open System
open System.Net

let download url = async {
  let uri = Uri (url)
  use webclient = new WebClient ()
  let! html = webclient.AsyncDownloadString (uri)
  Console.WriteLine $"{url} Read {html.Length} chars."
}

[ download "https://www.google.com"
  download "https://github.com"
  download "https://fsharp.org" ]
|> Async.Parallel
|> Async.RunSynchronously
```

`let!`

"`let! name = expr`" in an `async` block means: "perform the asynchronous computation `expr` and bind the result to `name` when the operation completes.".

This way, we can easily have nested asynchronous operations.

`return` and `return!`

We use `return` or `return!` at the end of an `async` computation (although it can be omitted unless there is a recursion). In the above example, The last line of the `async` computation simply omitted the `return` keyword.

`return!` is similar to `return`, but it first evaluates an `async` computation and waits for it to finish (as in `let!`), and returns the result wrapper with an `async` computation.

```
async {
  let! x = SomeAsyncComp
  return x
}
```

```
async {
  return! SomeAsyncComp
}
```

Example: Parallelism and Performance

Suppose we create a list of random strings. And we want to count the number of ’a’s in each string, and then sum them up.

```
let chars = "abcdefghijklmnopqrstuvwxyz" |> Seq.toArray
let r = Random ()
let strings =
  List.init 10000 (fun _ ->
    let len = r.Next 10000
    String.init len (fun _ ->
      chars[r.Next(chars.Length)] |> string))

let countA (s: string) =
  s
  |> Seq.filter (fun ch -> ch = 'a') |> Seq.length

let countA' s = async { return countA s }
```

```
// Nonparallel
strings
|> List.map countA
|> List.reduce (+)
|> printfn "%d"

// Parallel (10x faster on a 8-core machine)
strings
|> List.map countA'
|> Async.Parallel
|> Async.RunSynchronously
|> Array.reduce (+)
|> printfn "%d"
```

Task Expressions

Similar to `Async<'T>`, `Task<'T>` is a type that represents an asynchronous computation that will produce a value of type `'T` at some point in the future.

While `Async<'T>` is specific to F#, `Task<'T>` is a part of the .NET framework and is compatible with other languages. Another important difference is that `Task<'T>` expression will be evaluated immediately when it is created.

Example Task Expression

```
open System
open System.Threading
open System.Threading.Tasks

let taskLoop cnt =
  task {
    for i in 1 .. cnt do
      Console.WriteLine $"Count = {i}"
      do! Task.Delay 1000
  }
let t = taskLoop 10
t.Wait ()
```

Actor Model

Actor Model

The actor model in computer science is a mathematical model of concurrent computation that treats "actors" as the universal ***primitives*** of concurrent computation[1].

[1] https://en.wikipedia.org/wiki/Actor_model

Actor

An actor, which is a primitive unit of concurrent computation, receives a ***message*** and do some computation based on it in a completely ***isolated environment***. Actors can communicate with each other by sending messages (mails), and each actor has its own ***mailbox***, which stores all the messages that are yet processed. Actors may modify their own state, but they can only affect each other through messages. This way, we can avoid using locks!

F#'s Native Support for Actor Model

`MailboxProcessor<'Msg>` class represents an actor which executes an asynchronous computation.

```
type MailboxProcessor<'Msg> =
class
    interface IDisposable
    ...
    member this.Post : 'Msg -> unit
    member this.Start : unit -> unit
    static member Start : (MailboxProcessor<'Msg> -> Async<
        unit>) -> MailboxProcessor<'Msg>
    ...
```

Example: Echo Agent

```
let echoAgent = MailboxProcessor.Start (fun inbox ->
  let rec loop () = async {
    let! msg = inbox.Receive ()
    printfn "Got message: %s" msg
    return! loop ()
  }
  loop ())

echoAgent.Post "foo"
echoAgent.Post "bar"
```

So, Why Actor Model?

1. No shared states: no locking!
2. Easy to write concurrent programs.

Eliminating Mutable States (Bank Account)

```fsharp
type BankMessage = Withdraw of int

let account initialBalance =
  MailboxProcessor.Start (fun inbox ->
    let rec loop balance = async {
      let! msg = inbox.Receive ()
      match msg with
      | Withdraw amount ->
        let balance' = balance - amount
        printfn "Balance: %d" balance'
        return! loop balance'
    }
    loop initialBalance
  )
```

In-Class Activity #19

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.

   ```
   - git clone https://github.com/KAIST-CS220/CS220-Main.git
   ```

2. Move in to the directory `CS220-Main/Activities`

   ```
   - cd CS220-Main
   - cd Activities
   ```

Problem

Finalize the bank account actor model and test It. Explain why this program does not require any locks.

Conclusion

Further Readings

- https://fsharpforfunandprofit.com/posts/concurrency-async-and-parallel/
- https://fsharpforfunandprofit.com/posts/concurrency-actor-model/

Question?