Async programming

See https://tais993.github.io/Async-programming/ for website version

Terminology

Click to see all the terms we'll be talking about, you can always find the definition by pressing on their footnote.

CPU-Threads
: a thread that's managed by the CPU

OS-Threads
: a thread that's managed by the OS

Virtual threads
: a thread that's managed by the runtime, also known as a green thread, fiber or possibly coroutine

Threads
: just any form of thread, can be a CPU-Thread, OS-thread or a green-Thread

Concurrently
: multiple threads running on the same core (one by one)

Parallelism
: multiple threads running at the same time

Multithreaded
: multiple threads running concurrently (or in parallel)

Callback
: any executable code that is passed as an argument to other code

Asynchronous
: the occurrence of events independent of the main program flow and ways to deal with such events

Data-streams
: a sequence of elements made available over time

The issue, utilization of the CPU

CPU's have multiple cores, and multiple CPU-Threads[1] nowadays. Constantly, threads[2] block resulting in the underlying core doing nothing. Or the cores weren't doing anything in the first place, since there isn't a task for them.

And that's a waste, they should be used as much as possible, as efficiently as possible. So let's start at concurrent[3] programming, utilizing a thread as much as possible by switching tasks when one task blocks, which is where the await-async keywords come into the story.

Solution 1, Await-async keywords

Grab your coloring book, kids! We're going to start "coloring" our methods.

We have 2 types, async and sync methods. You make a method async by using the `async` keyword. This looks really simple, and like an amazing solution. There's a couple of caveats.

- You can only call async methods in async methods, you can't call an async method in a sync method, unless using a callback.[4]
- Sync functions return a value, async one's return a wrapper around the value (Future, Promise).
- You can't use try-catch, since the async one happens on another thread. (Often in callbacks)
- When using callbacks from code that you never wrote, you have to **trust** them that the callback actually runs.

To elaborate on this, take a view at this beautiful callback hell.

```
randomServer(function (server) {
    server.getRandomMember(function (member) {
        if (member.hasAcceptedRules) {
            member.giveRole(Role.STARTER, function (action) {
                if (action.hadSuccess) {
                    user.sendMessage("Gave you the starter role!");
                } else {
                    user.sendMessage("Something went wrong, please contact a
developer)");
                }
            });
        } else {
            user.sendMessage("You haven't accepted the rules yet, therefor I'll not
give you the starter role.");
        }
    })
});
```

There is a ton of callbacks in this code, which makes it really hard to read. It goes inside a method, inside a method, inside another method. Can't forget that debugging will be less enjoyable in this

case either.

Which is why people came up with the `await` keyword.

The `await` keyword

The `await` keyword allows people to wait for the method to finish, removing the need of callback.This fixes that sync methods can't call async methods (without a callback), but still numerous issues remain. An example of the `await` keyword can be found below.

```
var server = await randomServer();
var member = await server.getRandomMember();

if (member.hasAcceptedRules()) {
    var action = await member.giveRole(Role.STARTER);

    if (action.hadSuccess()) {
        user.sendMessage("Gave you the starter role!");
    } else {
        user.sendMessage("Something went wrong, please contact a developer)");
    }
} else {
    user.sendMessage("You haven't accepted the rules yet, therefor I'll not give you
the starter role.");
}
```

Notice how much easier it is to read, by all means it's still not perfect, it is a lot easier. The `await` keyword still is annoying, you have to constantly add it to your code``.

This removed 2 of our issues, callback hell is non-existent in this case, and you can call async methods from sync methods now. However, there still are numerous issues, so let's take a look at those.

- Sync functions return values, while async return a Task\<T\> (a wrapper of the value.
- Async methods only return the value if you use await, which means your method has to become `async` too
- Constant await gibberish, which floods the code
- Async methods, are still **async**

So yeah, is this **really** a good solution? I'd say no, it doesn't feel polished, it's not that pleasant to write either.

Benefits of async-await

I could say as much as I want, but I'd be lying if I said they were useless. They're really simple, a beginner programmer can easily adapt to this. And that's exactly why it fits so well with JS, a rather simple programming language. Depending on how many OS-threads your runtime has, it's either

concurrent, or both concurrent and parallel.

Solution 2, OS-Threads

So, how about instead of using concurrency we use parallelism?[5] Sure! OS-Threads[6], you can come in now.

Using OS-Threads we can let our processor run multiple things at the same time. You're loading a file? Just create a new thread and do it on there!

At least, that sounds like the most logic thing to do, but it's not that simple. Threads aren't lightweight, threads are heavy. Threads are designed with everything in mind, so they store a lot more data than they require.

So a solution, you use a pool, a ThreadPool. A thread pool re-uses threads, this way you don't constantly create new threads, and only create new ones when it's actually needed. So you achieve parallelism, but still it's not perfect, you know? It doesn't feel nice to constantly access a ThreadPool.

You made it run in parallel, but not (optimized) concurrent[3]. Everytime your thread gets blocked the OS needs to handle it, which happens rather inefficiently.

Also, there are security issues, a thread exposes its thread locals. A thread local is a variable local to the thread, once set they cannot be removed unless overwritten by a different value. While this makes sense, when in the context of a ThreadPool, this means that all tasks using the thread also get access to the locals.

And last, but not least, due to the fact it's an OS-thread, the cancellation program is rather complex and expensive. To elaborate on this, if a thread interrupts another thread, this thread sets the status to Interrupted, and throws an exception to all blocking methods. After which the status gets cleared.

The status gets cleared for 2 reasons. First, this allows ThreadPool's to re-use the thread. Second of all, one might require some of the thread's blocking methods after an error.

While the purpose of this behaviour makes sense, it's just rather error-prone. Preferably we'd want to create a new thread instead.

And one of the issues I personally notice myself, people don't know how to use threads. All tutorials teach how to manually create threads. In the example below, you also see me manually create a thread, but in reality you'll **rarely** if not **ever** do this. You want to re-use threads, they're too expensive to waste. This is simple to fix, by teaching people to use pools early.

To summarize this - Thread's expose thread locals to everything running in the thread - Complex cancellation program - No concurrency

```
public class Application {
    public static void main(String[] args) {
        new Thread(() -> {
            var server = randomServer();
            var member = server.getRandomMember();

            if (member.hasAcceptedRules()) {
                var action = member.giveRole(Role.STARTER);

                if (action.hadSuccess()) {
                    user.sendMessage("Gave you the starter role!");
                } else {
                    user.sendMessage("Something went wrong, please contact a
developer)");
                }
            } else {
                user.sendMessage("You haven't accepted the rules yet, therefor I'll
not give you the starter role.");
            }
        });
    }
}
```

But at least, the code is readable, no await, no playing around with Task<T>. This runs in parallel, the concurrency is still being handled by the OS, inefficiently.

It'd be awesome if we somehow, mixed those together in an effective manner? Which is where reactive comes in!

Benefits

No async-await mess, the virus doesn't spread throughout your codebase. Readability improved too.

Solution 3, (Functional) reactive programming

Reactive combines the best and worst of both, it results in a concurrent and parallel method of programming.

To take a look at the definition according to Wikipedia

> Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter).
>
> — en.wikipedia.org, Functional reactive programming

As you might notice, this sounds advanced. Unfortunately I'll have to tell you, reactive programming has a rather steep learning curve.

So to elaborate on what reactive programming is, I'll show you a simple example which we will break down together.

```
public class Application {
    public static void main(String[] args) {
        Stream.of("yeppers", "something", "buzz", "fuzz", "bar")
                .filter(str -> s.startsWith("b"))
                .map(str -> s.toUpperCase())
                .sorted()
                .forEach(str -> System.out.println(s));
    }
}
```

A Stream is a so called data-stream[4]. It depends on the kind of data-source how the stream behaves, in this case the stream instantly consists of all possible elements, and no new ones will be added later. Due to the size of the stream, there's no need for making it parallel. In this specific example, there's nothing concurrent (like streams combining) either. This is just, to show how a stream looks.

To refer to the example, you create a data-stream (Stream#of). The first thing that happens is a filter, a common function in reactive programming. This filter applies the given predicate to each element, and only if true is returned the element will stay in the data-stream. Following a map, a map is a function that applies the given function to the element(s), this is very common in reactive programming. A map allows you to, convert items. In this example it makes the existing string fully upper-case. It can also convert to something of a different type. Now we sort, this sort is the default sort of Java's Stream#sorted method, which sorts it in natural order. After this, we loop through all methods, you can compare this to a for loop. We print its value to the console.

So what you saw it the data-stream going through many functions, that change the elements of the data-stream. This can be compared to C#'s LINQ, this is a more basic version of Rx libraries in languages (RxJAva, RxPy, RxJS).

Before I'll show you a more Rx like example, I'll go over a few more terms relating to functional reactive programming.

You saw map before, but how about flatmap? Flatmap is an implementation that **maps** a data-stream into another data-stream. Instead of mapping, which will result in a data-stream in a data-stream, you flatmap and return the data-stream. This will also allow some things giving us nicer error handling, onErrorFlatMap, if an error comes. flatmap this into a data-stream (of the same type) An example can be seen below.

```
public class Application {
    public static void main(String[] args) {
        randomServer()
                .flatmap(server -> server.getRandomMember())
                .flatmap((member) -> {
                    if (member.hasAcceptedRules()) {
                        return member.giveRole(Role.STARTER)
                                .flatmap(action -> action.hadSuccess(), action -> {
                                    user.sendMessage("Gave you the starter role!");
                                }).onErrorFlatmap((action) -> {
                                    return user.sendMessage("Something went wrong,
please contact a developer)");
                                });
                    } else {
                        return user.sendMessage("You haven't accepted the rules yet,
therefor I'll not give you the starter role.");
                    }
                });
    }
}
```

If you're new to reactive programming, I'm sorry for the heart attack this might have given you. So as you see, concurrent, parallel, no callback hell, or well it's supposed to be no callback hell.

This looks awful, which is where something important in the programming world comes back. Methods, splitting logic. If Member#hasAcceptedRules is true, it should instead call a method that contains all the logic, this makes it a lot more readable. Just that change alone, will make a huge difference to the readability and maintainability of your reactive code.

```
public class Application {
    public static void main(String[] args) {
        randomServer()
                .flatmap(server -> server.randomMember())
                .flatmap((member) -> {
                    if (member.hasAcceptedRules()) {
                        return handleMemberRuleAccept(member);
                    } else {
                        return user.sendMessage("You haven't accepted the rules yet,
therefor I'll not give you the starter role.");
                    }
                });
    }

    private static RestAction handleMemberRuleAccept(Member member) {
        return member.giveRole(Role.STARTER)
                .flatmap(action -> action.hadSuccess(), action -> {
                    user.sendMessage("Gave you the starter role!");
                }).onErrorFlatmap((action) -> {
                    return user.sendMessage("Something went wrong, please contact a
developer)");
                });
    }
}
```

This still isn't something I'd show a beginner, someone needs decent/good programming knowledge to read this, and especially to write this. But before I keep complaining about the steep learning-curve, let's go over the benefits.

- Applies concurrency and parallelism
- It's good for maintainability and readability when using the correct methods (and when correctly splitting logic up in methods)

And the issues

- Extremely steep learning curve
- Hard to debug

So, is it really worth the hassle? It's steep learning-curve makes it something a beginner would be scared of. But when you did survive it, and you went through everything, you'll enjoy writing this kind of clean code.

Benefits

Well, this is a hard one. If your language itself doesn't support concurrency in any other way, you can't do a lot besides this. But if you can avoid it, I personally would. When used correctly, it can be an incredible tool, but due to its complexity it rarely meets its potential.

Solution 4, the final boss, fibers / coroutines

We'll separate this into 2 sub-topics. Self-handled fibers[7] and language maintained.

completely rewrite this tyvm idk how to explain it

Self-handled fibers

Self-handled fibers are handled by you, you need to use await-async

Language maintained fibers

Green threads, fibers, lightweight threads, virtual threads, it doesn't matter how you call them. They are the solution, in my eyes. You can code like it's sync, while the code runs in concurrency and parallelism.

Let's go over all the issues we found with all the other solutions, and check or these apply here.

Issues of solution 1, async keyword

- You can only call async methods in async methods, you can't call an async method in a sync method. (unless using a callback)
- Sync functions return a value, async one's return a wrapper around the value (Future, Promise)
- You can't use try-catch, since the async one happens on another thread. (Often in callbacks)
- When using callbacks from code that you never wrote, you have to trust them that the callback actually runs.

All issues, only apply to async methods. So you could say, fibers fixed them by removing async methods all together.

Issues of solution 2, OS-Threads

- Thread's expose thread locals to everything running in the thread

This issue came into existence once people started to re-use threads due to the fact that threads are rather expensive. But fibers, they aren't expensive, you **should** create a new fiber for each task. So it doesn't expose anything

- Complex cancellation program

Since fibers don't have to be re-used, the complexity is removed. One can just create a new fiber if someone goes wrong.

- No concurrency

Fibers are handled concurrently, so don't worry about blocking calls!

Issue of solution 3, (Functional) reactive programming

- Extremely steep learning curve

You don't need to know what fibers exactly do, only the basics, that they run concurrent and in parallel.While that's not something you know from birth, it isn't something you need months to study either.The most important part is that people shouldn't feel afraid to create a new fiber. And people shouldn't re-use fibers either.

- Hard to read and debug

Since it's sync code, it's easy to read.Since the same fiber always runs the code, it's also so much easier to debug.

So let's, just take a look at an example.

Note, the way the fiber is created doesn't exist, this is to show the idea, it's exact implementations might differ Java

```
public class Application {
    public static void main(String[] args) {
        new VirtualThread(() -> {
            var server = randomServer();
            var member = server.getRandomMember();

            if (member.hasAcceptedRules()) {
                var action = member.giveRole(Role.STARTER);

                if (action.hadSuccess()) {
                    user.sendMessage("Gave you the starter role!");
                } else {
                    user.sendMessage("Something went wrong, please contact a
developer)");
                }
            } else {
                user.sendMessage("You haven't accepted the rules yet, therefor I'll
not give you the starter role.");
            }
        });
    }
}
```

There, it looks exactly the same as the second solution, OS-Threads.Difference is that it's concurrent, and doesn't have the thread local issues, nor complex cancellation program.

Beautifully simple in the end, no major rewrites required.This, while it's still really performant, if not even more than reactive libraries.

In my eyes, it's perfect!But not everyone agrees, some people don't like the fact that the blocking calls are implicit concurrent. And it's upto you to decide whenever you mind that, I personally don't, I trust the implementation enough.

Benefits

Short summarization of what I mentioned before, it brings support to concurrent, parallel programming without being forced

[1] a thread that's managed by the CPU

[2] just any form of thread, can be a CPU-Thread, OS-thread or a green-Thread

[3] multiple threads running on the same core (one by one)

[4] any executable code that is passed as an argument to other code

[5] multiple threads running at the same time

[6] a thread that's managed by the OS

[7] a thread that's managed by the runtime, also known as a green thread, fiber, virtual thread or coroutine