Concurrency

Asynchrony, Coroutines, Threads

What is concurrency

- Intuitive definition: starting one thing before other things have finished
- Humans can work concurrent:
  - I start making slides
  - I switch to answering email
  - I finish my slides
  - I finish my emails
- Note: I was only ever working on one thing at a time!
- Concurrency != Parallelism

Concurrency is good, even with a single “thread”

- Concurrency is super useful when one task requires waiting for something to happen... do something else while you wait
- Concurrency improves interactivity. Stop what you're doing to handle user input
- Concurrency an benefit from parallelism. If 2 tasks take lots of devoted work (CPU time), you can do them concurrently in parallel and reduce the total time each of them take

Concurrency is hard

- How do we keep track of everything that's in flight?
- Can I be sure that data I'm using will persist as long as I need it?
- Can I be sure that data I'm using won't be changed by another task?
- Who's job is it to switch between tasks?
- How do errors from one task get propagated to tasks that need to know about them?

Approaches to Concurrency

OS Threads

- Concurrent tasks run in different OS threads
- The OS is responsible for switching between threads (preemptive multitasking), so any given task might get yanked off the CPU at any point
- Note: OS Threads work even on a single core (ie, without parallelism)
- Shared pointers means shared data. Mutexes required for safety and is not enforced by any system
- Threads are pretty much independent and aside from waiting for another, any other information passing (return values/errors) basically has to be done manually
- GC languages help make sure lifetimes are handled properly

Concurrency approach: manual concurrency

- Manually track what's in flight, what tasks are “blocked” waiting for something, what's ready, etc
- Often designs are built around an “event loop” and a system call like `select`
- Each time through the event loop, we query what tasks are “ready” to run a little bit, then pass data to the tasks via callbacks
- Callbacks need to be non blocking, the API is all callback based... no fun for anyone

Slight improvement: User threads (AKA green threads, fibers)

- Similar idea to OS threads except that threads are responsible for “yielding” when they want to let another thread run
- For example, a user thread might send a network request, and then yield, so that another thread can run while it waits for a response
- Multiple green threads can share a single OS thread. OS threads are “expensive” so it's actually cheap to have many green threads and switch between them
- If there's a single OS thread, then sharing data is relatively easy. No other threads can touch our data unless we `yield`
- We can add parallelism (called M by N threading) by having multiple OS threads, each running multiple green threads

Green thread issues

- Threads must yield to let other threads run
- Often, library support provides “yielding” wrappers for slow system calls to, for example, send a network request and yield automatically, instead of waiting for the response
- Scheduling can get tricky, especially with parallelism
- Still no built in way to handle return values or errors across threads

Manual async with callbacks: Promises/Futures

- Promises (AKA futures) is another approach for managing concurrency with slow operations
- Functions like `sendHttpRequest` don't return the HTTP response, they immediately return a `Promise` object
- A `Promise` is a reference to the “work” that's in flight and depending on the language/library, you can do different things with one
- The most common operations are `then` or `wait`
- `then` takes a callback to run (potentially in another thread) when the operation completes
- `wait` stops the current thread (by yielding) and resuming once the operation completes

Neat promise tricks

- You can define functions that take multiple promises and combine them in different ways
- Example: `waitAll([promises...])` waits for all promises before continuing
- `waitAny([promises])`, etc
- Some promise APIs let you attach `onError` callbacks, or will throw an exception if there's an error when you call `wait()`
- This model is quite powerful, but can result in really ugly nesting of callbacks

Promise syntax sugar: async/await

- Dealing with promises directly is sort of annoying, so many languages have a way of hiding that from programmers
- Functions declared as `async` implicitly return a promise, not a value
- When we need to wait for an async function to finish before continuing, we use the `await` keyword which yields until the promise is ready
- async/await makes code pretty easy to follow (ie, it's straight line code, with limited callbacks), and you can use try/catch naturally
- Unfortunately, you have to track which functions are “normal” and which are “async” ... normal functions can't call async functions naively because they might yield

Problem: lifetimes

- All these approaches have a sort of ad-hoc approach to managing lifetimes/scope
- You can pass a promsie around since it's just an object, or forget about it entirely
- Where should reported errors go? In a language without a GC, when should it get cleaned up?
- How do we handle cancellation?
- Working threads is sort of like using `goto` for control flow... it's too powerful and can make spaghetti
- We need something like loops and `if` statements for concurrency

Kotlin's solution (and C++ actually!): Coroutines

- There's many different ways of describing coroutines. Depending on the language, some are “more correct” than others
- I like to think of them as “resumable functions”
- With a normal function, when you call it, you allocate a stack frame, jump to its code, and get a value back when it returns
- With a coroutine, it can be “suspended” (similar to a function return), and then you can “resume” it and it keeps running
- In some languages, a coroutine can “yield” multiple return values and be resumed by the caller after each one (not in Kotlin!)
- Eventually the coroutine completes for good, which is like a normal function returning

Coroutines in Kotlin

- Since coroutines are like functions they have a call stack
- In a GC langauge, we could just allocate those in the heap, and let the GC clean the up whenver the coroutine actually finishes. In C++, for example, that would NOT be an option
- Even though we COULD just say “let the GC handle it” it actually makes code harder to reason about when we don't understand the “scope” of our coroutines
- In Kotlin, coroutines run in a `scope` which means that lifetimes are “nested”
- This is called “structured concurrency”

Coroutines example

```
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one is delayed
}
```

- The 2 sets of {} are coroutine “scopes,” and are nested. The scope from launch is entirely inside the scope from runBlocking
- Delay is a suspending function which pauses the coroutine and then resumes it one second later

Coroutines

- When a coroutine is running, it could run on any thread. Coroutines can even be switched to different threads in between suspending/resuming them
- In Android, we'll sometime need to specify that we want to run on the main UI thread, or in a background thread

LiveData analog for coroutines: Flow<t>

- You can sort of think of a `LiveData` wrapper as a “sequence” of values (each time we call `setValue` we add another element to the sequence)
- We iterate over the sequence asynchronously (ie, as things come in)
- We can do something very similar using coroutines, which the Kotlin standard lib (not Android specific!) put into a class named `Flow`

Flow example

```
fun simple(): Flow<int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
}    }
fun main() = runBlocking<unit> {
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // Collect the flow
    simple().collect { value -> println(value) }
}
```

Cool flow stuff

- Flows can be easily converted to lists or live data
- There's also tons of neat functions like map/filter, and many others
- There's also many functions for combining data from different flows, etc

Details for us

- Like with async/await, coroutines are declared as `suspend` meaning they might pause in the middle
- We can't call a `suspend` function directly from a normal function and have to use a coroutine scope like `runBlocking`
- Often, our coroutine scopes will be tied to things like `lifecycleOwner`, which we'll see when we talk about the `Room` database wrapper library