Concurrency

Concurrency vs. Parallelism

The definitions of "concurrency" and "parallelism" sometimes get mixed up, but they are not the same.

A concurrent system is one that can be in charge of many tasks, although not necessarily executing them at the same time. You can think of yourself being in the kitchen cooking: you chop an onion, put it to fry, and while it's being fried you chop a tomato, but you are not doing all of those things at the same time: you distribute your time between those tasks. Parallelism would be to stir fry onions with one hand while with the other one you chop a tomato.

At the moment of this writing, Crystal has concurrency support but not parallelism: several tasks can be executed, and a bit of time will be spent on each of these, but two code paths are never executed at the same exact time.

A Crystal program by default executes in a single operating system thread, except for the garbage collector (currently Boehm GC). Parallelism is supported, but it is currently considered experimental. Check out this Crystal Blog post about parallelism for more information.

Fibers

To achieve concurrency, Crystal has fibers. A fiber is in a way similar to an operating system thread except that it's much more lightweight and its execution is managed internally by the process. So, a program will spawn multiple fibers and Crystal will make sure to execute them when the time is right.

Event loop

For everything I/O related there's an event loop. Some time-consuming operations are delegated to it, and while the event loop waits for that operation to finish the program can continue executing other fibers. A simple example of this is waiting for data to come through a socket.

Channels

Crystal has Channels inspired by CSP. They allow communicating data between fibers without sharing memory and without having to worry about locks, semaphores or other special structures.

Execution of a program

When a program starts, it fires up a main fiber that will execute your top-level code. There, one can spawn many other fibers. The components of a program are:

- The Runtime Scheduler, in charge of executing all fibers when the time is right.
- The Event Loop, which is just another fiber, being in charge of async tasks, like for example files, sockets, pipes, signals and timers (like doing a `sleep`).
- Channels, to communicate data between fibers. The Runtime Scheduler will coordinate fibers and channels for their communication.
- Garbage Collector: to clean up "no longer used" memory.

A Fiber

A fiber is an execution unit that is more lightweight than a thread. It's a small object that has an associated stack of 8MB, which is what is usually assigned to an operating system thread.

Fibers, unlike threads, are cooperative. Threads are pre-emptive: the operating system might interrupt a thread at any time and start executing another one. A fiber must explicitly tell the Runtime Scheduler to switch to another fiber. For example if there's I/O to be waited on, a fiber will tell the scheduler "Look, I have to wait for this I/O to be available, you continue executing other fibers and come back to me when that I/O is ready".

The advantage of being cooperative is that a lot of the overhead of doing a context switch (switching between threads) is gone.

A Fiber is much more lightweight than a thread: even though it's assigned 8MB, it starts with a small stack of 4KB.

On a 64-bit machine it lets us spawn millions and millions of fibers. In a 32-bit machine we can only spawn 512 fibers, which is not a lot. But because 32-bit machines are starting to become obsolete, we'll bet on the future and focus more on 64-bit machines.

The Runtime Scheduler

The scheduler has a queue of:

- Fibers ready to be executed: for example when you spawn a fiber, it's ready to be executed.
- The event loop: which is another fiber. When there are no other fibers ready to be executed, the event loop checks if there is any async operation that is ready, and then executes the fiber waiting for that operation. The event loop is currently implemented with `libevent`, which is an abstraction of other event mechanisms like `epoll` and `kqueue`.
- Fibers that voluntarily asked to wait: this is done with `Fiber.yield`, which means "I can continue executing, but I'll give you some time to execute other fibers if you want".

Communicating data

Because at this moment there's only a single thread executing your code, accessing and modifying a class variable in different fibers will work just fine. However, once multiple threads (parallelism) is introduced in the language, it might break. That's why the recommended mechanism to communicate data is using channels and sending messages between them. Internally, a channel implements all the locking mechanisms to avoid data races, but from the outside you use them as communication primitives, so you (the user) don't have to use locks.

Sample code

Spawning a fiber

To spawn a fiber you use `spawn` with a block:

```
spawn do
  # ...
  socket.gets
  # ...
end

spawn do
  # ...
  sleep 5.seconds
  #  ...
end
```

Here we have two fibers: one reads from a socket and the other does a `sleep`. When the first fiber reaches the `socket.gets` line, it gets suspended, the Event Loop is told to continue executing this fiber when there's data in the socket, and the program continues with the second fiber. This fiber wants to sleep for 5 seconds, so the Event Loop is told to continue with this fiber in 5 seconds. If there aren't other fibers to execute, the Event Loop will wait until either of these events happen, without consuming CPU time.

The reason why `socket.gets` and `sleep` behave like this is because their implementations talk directly with the Runtime Scheduler and the Event Loop, there's nothing magical about it. In general, the standard library already takes care of doing all of this so you don't have to.

Note, however, that fibers don't get executed right away. For example:

```
spawn do
  loop do
    puts "Hello!"
  end
end
```

Running the above code will produce no output and exit immediately.

The reason for this is that a fiber is not executed as soon as it is spawned. So, the main fiber, the one that spawns the above fiber, finishes its execution and the program exits.

One way to solve it is to do a `sleep`:

```
spawn do
  loop do
    puts "Hello!"
  end
end

sleep 1.second
```

This program will now print "Hello!" for one second and then exit. This is because the `sleep` call will schedule the main fiber to be executed in a second, and then executes another "ready to execute" fiber, which in this case is the one above.

Another way is this:

```
spawn do
  loop do
    puts "Hello!"
  end
end

Fiber.yield
```

This time `Fiber.yield` will tell the scheduler to execute the other fiber. This will print "Hello!" until the standard output blocks (the system call will tell us we have to wait until the output is ready), and then execution continues with the main fiber and the program exits. Here the standard output *might* never block so the program will continue executing forever.

If we want to execute the spawned fiber for ever, we can use `sleep` without arguments:

```
spawn do
  loop do
    puts "Hello!"
  end
end

sleep
```

Of course the above program can be written without `spawn` at all, just with a loop. `sleep` is more useful when spawning more than one fiber.

Spawning a call

You can also spawn by passing a method call instead of a block. To understand why this is useful, let's look at this example:

```
i = 0
while i < 10
  spawn do
    puts(i)
  end
  i += 1
end

Fiber.yield
```

The above program prints "10" ten times. The problem is that there's only one variable `i` that all spawned fibers refer to, and when `Fiber.yield` is executed its value is 10.

To solve this, we can do this:

```
i = 0
while i < 10
  proc = ->(x : Int32) do
    spawn do
      puts(x)
    end
  end
  proc.call(i)
  i += 1
end

Fiber.yield
```

Now it works because we are creating a Proc and we invoke it passing `i`, so the value gets copied and now the spawned fiber receives a copy.

To avoid all this boilerplate, the standard library provides a `spawn` macro that accepts a call expression and basically rewrites it to do the above. Using it, we end up with:

```
i = 0
while i < 10
  spawn puts(i)
  i += 1
end

Fiber.yield
```

This is mostly useful with local variables that change at iterations. This doesn't happen with block arguments. For example, this works as expected:

```
10.times do |i|
  spawn do
    puts i
  end
end

Fiber.yield
```

Spawning a fiber and waiting for it to complete

We can use a channel for this:

```
channel = Channel(Nil).new

spawn do
  puts "Before send"
  channel.send(nil)
  puts "After send"
end

puts "Before receive"
channel.receive
puts "After receive"
```

This prints:

```
Before receive
Before send
```

```
After send
After receive
```

First, the program spawns a fiber but doesn't execute it yet. When we invoke `channel.receive`, the main fiber blocks and execution continues with the spawned fiber. Then `channel.send(nil)` is invoked. Note that this `send` does not occupy space in the channel because there is a `receive` invoked prior to the first `send`, `send` is not blocked. Fibers only switch out when blocked or executing to completion. So the spawned fiber will continue after `send`, and execution will switch back to main fiber once `puts "After send"` is executed.

The main fiber then resumes at `channel.receive`, which was waiting for a value. Then the main fiber continues executing and finishes.

In the above example we used `nil` just to communicate that the fiber ended. We can also use channels to communicate values between fibers:

```
channel = Channel(Int32).new

spawn do
  puts "Before first send"
  channel.send(1)
  puts "Before second send"
  channel.send(2)
end

puts "Before first receive"
value = channel.receive
puts value # => 1

puts "Before second receive"
value = channel.receive
puts value # => 2
```

Output:

```
Before first receive
Before first send
Before second send
1
Before second receive
2
```

Note that when the program executes a `receive`, the current fiber blocks and execution continues with the other fiber. When `channel.send(1)` is executed, execution continues because `send` is non-blocking if the channel is not yet full. However, `channel.send(2)` does cause the fiber to block

because the channel (which has a size of 1 by default) is full, so execution continues with the fiber that was waiting on that channel.

Here we are sending literal values, but the spawned fiber might compute this value by, for example, reading a file, or getting it from a socket. When this fiber will have to wait for I/O, other fibers will be able to continue executing code until I/O is ready, and finally when the value is ready and sent through the channel, the main fiber will receive it. For example:

```
require "socket"

channel = Channel(String).new

spawn do
  server = TCPServer.new("0.0.0.0", 8080)
  socket = server.accept
  while line = socket.gets
    channel.send(line)
  end
end

spawn do
  while line = gets
    channel.send(line)
  end
end

3.times do
  puts channel.receive
end
```

The above program spawns two fibers. The first one creates a TCPServer, accepts one connection and reads lines from it, sending them to the channel. There's a second fiber reading lines from standard input. The main fiber reads the first 3 messages sent to the channel, either from the socket or stdin, then the program exits. The `gets` calls will block the fibers and tell the Event Loop to continue from there if data comes.

Likewise, we can wait for multiple fibers to complete execution, and gather their values:

```
channel = Channel(Int32).new

10.times do |i|
  spawn do
    channel.send(i * 2)
  end
end

sum = 0
10.times do
```

```
  sum += channel.receive
end
puts sum # => 90
```

You can, of course, use `receive` inside a spawned fiber:

```
channel = Channel(Int32).new

spawn do
  puts "Before send"
  channel.send(1)
  puts "After send"
end

spawn do
  puts "Before receive"
  puts channel.receive
  puts "After receive"
end

puts "Before yield"
Fiber.yield
puts "After yield"
```

Output:

```
Before yield
Before send
Before receive
1
After receive
After send
After yield
```

Here `channel.send` is executed first, but since there's no one waiting for a value (yet), execution continues in other fibers. The second fiber is executed, there's a value on the channel, it's obtained, and execution continues, first with the first fiber, then with the main fiber, because `Fiber.yield` puts a fiber at the end of the execution queue.

Buffered channels

The above examples use unbuffered channels: when sending a value, if a fiber is waiting on that channel then execution continues on that fiber.

With a buffered channel, invoking `send` won't switch to another fiber unless the buffer is full:

```
# A buffered channel of capacity 2
channel = Channel(Int32).new(2)

spawn do
  puts "Before send 1"
  channel.send(1)
  puts "Before send 2"
  channel.send(2)
  puts "Before send 3"
  channel.send(3)
  puts "After send"
end

3.times do |i|
  puts channel.receive
end
```

Output:

```
Before send 1
Before send 2
Before send 3
After send
1
2
3
```

Note that the first `send` does not occupy space in the channel. This is because there is a `receive` invoked prior to the first `send` whereas the other 2 `send` invocations take place before their respective `receive`. The number of `send` calls do not exceed the bounds of the buffer and so the send fiber runs uninterrupted to completion.

Here's an example where all space in the buffer gets occupied:

```
# A buffered channel of capacity 1
channel = Channel(Int32).new(1)

spawn do
  puts "Before send 1"
  channel.send(1)
  puts "Before send 2"
  channel.send(2)
  puts "Before send 3"
  channel.send(3)
  puts "End of send fiber"
end

3.times do |i|
```

```
  puts channel.receive
end
```

Output:

```
Before send 1
Before send 2
Before send 3
1
2
3
```

Note that "End of send fiber" does not appear in the output because we `receive` the 3 `send` calls which means `3.times` runs to completion and in turn unblocks the main fiber which executes to completion.

Here's the same snippet as the one we just saw - with the addition of a `Fiber.yield` call at the very bottom:

```
# A buffered channel of capacity 1
channel = Channel(Int32).new(1)

spawn do
  puts "Before send 1"
  channel.send(1)
  puts "Before send 2"
  channel.send(2)
  puts "Before send 3"
  channel.send(3)
  puts "End of send fiber"
end

3.times do |i|
  puts channel.receive
end

Fiber.yield
```

Output:

```
Before send 1
Before send 2
Before send 3
1
2
3
End of send fiber
```

With the addition of a `Fiber.yield` call at the end of the snippet we see the "End of send fiber" message in the output which would have otherwise been missed due to the main fiber executing to completion.