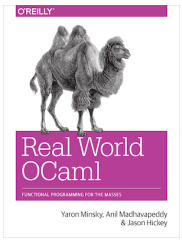


Table of Contents

Chapter 18. Concurrent Programming with Async

The logic of building programs that interact with the outside world is often dominated by waiting: waiting for the click of a mouse, or for data to be fetched from disk, or for space to be available on an outgoing network buffer. Even mildly sophisticated interactive applications are typically *concurrent*: needing to wait for multiple different events at the same time, responding immediately to whatever event happens first.

One approach to concurrency is to use preemptive system threads, which is the dominant approach in languages like Java or C#. In this model, each task that may require simultaneous waiting is given an operating system thread of its own so it can block without stopping the entire program.

Another approach is to have a single-threaded program, where that single thread runs an *event loop* whose job is to react to external events like timeouts or mouse clicks by invoking a callback function that has been registered for that purpose. This approach shows up in languages like JavaScript that have single-threaded runtimes, as well as in many GUI toolkits.

Each of these mechanisms has its own trade-offs. System threads require significant memory and other resources per thread. Also, the operating system can arbitrarily interleave the execution of system threads, requiring the programmer to carefully protect shared resources with locks and condition variables, which is exceedingly error-prone.

Single-threaded event-driven systems, on the other hand, execute a single task at a time and do not require the same kind of complex synchronization that preemptive threads do. However, the inverted control structure of an event-driven program often means that your own control flow has to be threaded awkwardly through the system's event loop, leading to a maze of event callbacks.

This chapter covers the Async library, which offers a hybrid model that aims to provide the best of both worlds, avoiding the performance compromises and synchronization woes of preemptive threads without the confusing inversion of control that usually comes with event-driven systems.

ASYNC BASICS

Recall how I/O is typically done in Core. Here's a simple example:

```
# In_channel.read_all;;
 - : string -> string = <fun>
# Out_channel.write_all "test.txt" ~data:"This is only a test.";;
 - : unit = ()
# In_channel.read_all "test.txt";;
 - : string = "This is only a test."
```

OCaml Utop ⁎ async/main.topscript , continued (part 1) ⁎ all code

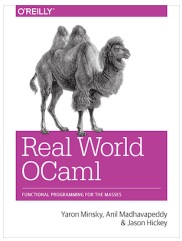
From the type of `In_channel.read_all`, you can see that it must be a blocking operation. In particular, the fact that it returns a concrete string means it can't return until the read has completed. The blocking nature of the call means that no progress can be made on anything else until the read is completed.

In Async, well-behaved functions never block. Instead, they return a value of type `Deferred.t` that acts as a placeholder that will eventually be filled in with the result. As an example, consider the signature of the Async equivalent of `In_channel.read_all`:

```
# #require "async";;

# open Async.Std;;

# Reader.file_contents;;
 - : string -> string Deferred.t = <fun>
```

OCaml Utop ⁎ async/main.topscript , continued (part 3) ⁎ all code

We first load the Async package in the toplevel using `#require`, and then open `Async.Std`, which adds a number of new identifiers and modules into our environment that make using Async



more convenient. Opening `Async.Std` is standard practice for writing programs using Async, much like opening `Core.Std` is for using Core.

A deferred is essentially a handle to a value that may be computed in the future. As such, if we call `Reader.file_contents`, the resulting deferred will initially be empty, as you can see by calling `Deferred.peek` on the resulting deferred:

```
# let contents = Reader.file_contents "test.txt";;
 val contents : string Deferred.t = <abstr>
# Deferred.peek contents;;
 - : string option = None
```

OCaml Utop ⋆ async/main.topscript , continued (part 4) ⋆ all code

The value in `contents` isn't yet determined partly because nothing running could do the necessary I/O. When using Async, processing of I/O and other events is handled by the Async scheduler. When writing a standalone program, you need to start the scheduler explicitly, but **utop** knows about Async and can start the scheduler automatically. More than that, **utop** knows about deferred values, and when you type in an expression of type `Deferred.t`, it will make sure the scheduler is running and block until the deferred is determined. Thus, we can write:

```
# contents;;
 - : string = "This is only a test."
```

OCaml Utop ⋆ async/main.topscript , continued (part 5) ⋆ all code

If we peek again, we'll see that the value of `contents` has been determined:

```
# Deferred.peek contents;;
 - : string option = Some "This is only a test."
```

OCaml Utop ⋆ async/main.topscript , continued (part 6) ⋆ all code

In order to do real work with deferreds, we need a way of waiting for a deferred computation to finish, which we do using `Deferred.bind`. First, let's consider the type-signature of `bind`:

```
# Deferred.bind ;;
 - : 'a Deferred.t -> ('a -> 'b Deferred.t) -> 'b Deferred.t = <fun>
```

OCaml Utop ⋆ async/main.topscript , continued (part 7) ⋆ all code

`Deferred.bind d f` takes a deferred value `d` and a function `f` that is to be run with the value of `d` once it's determined. You can think of `Deferred.bind` as a kind of sequencing operator, and what we're doing is essentially taking an asynchronous computation `d` and tacking on another stage comprised by the actions of the function `f`.

At a more concrete level, the call to `Deferred.bind` returns a new deferred that becomes determined when the deferred returned by `f` is determined. It also implicitly registers with the scheduler an *Async job* that is responsible for running `f` once `d` is determined.

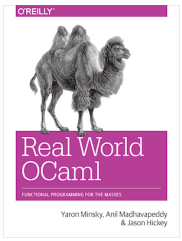
Here's a simple use of `bind` for a function that replaces a file with an uppercase version of its contents:

```
# let uppercase_file filename =
    Deferred.bind (Reader.file_contents filename)
     (fun text ->
        Writer.save filename ~contents:(String.uppercase text))
  ;;
 val uppercase_file : string -> unit Deferred.t = <fun>
# uppercase_file "test.txt";;
 - : unit = ()
# Reader.file_contents "test.txt";;
 - : string = "THIS IS ONLY A TEST."
```

OCaml Utop ⋆ async/main.topscript , continued (part 8) ⋆ all code

Writing out `Deferred.bind` explicitly can be rather verbose, and so `Async.Std` includes an infix operator for it: `>>=`. Using this operator, we can rewrite `uppercase_file` as follows:

```
# let uppercase_file filename =
    Reader.file_contents filename
    >>= fun text ->
    Writer.save filename ~contents:(String.uppercase text)
  ;;
```




```
 val uppercase_file : string -> unit Deferred.t = <fun>
```

OCaml Utop ⁎ async/main.topscript , continued (part 9) ⁎ all code

In the preceding code, we've dropped the parentheses around the function on the righthand side of the bind, and we didn't add a level of indentation for the contents of that function. This is standard practice for using the `bind` operator.

Now let's look at another potential use of `bind`. In this case, we'll write a function that counts the number of lines in a file:

```
# let count_lines filename =
    Reader.file_contents filename
    >>= fun text ->
    List.length (String.split text ~on:'\n')
  ;;
 Characters 85-125:
 Error: This expression has type int but an expression was expected of type
          'a Deferred.t
```

OCaml Utop ⁎ async/main.topscript , continued (part 10) ⁎ all code

This looks reasonable enough, but as you can see, the compiler is unhappy. The issue here is that `bind` expects a function that returns a deferred, but we've provided it a function that returns the nondeferred result directly. To make these signatures match, we need a function for taking an ordinary value and wrapping it in a deferred. This function is a standard part of Async and is called `return`:

```
# return;;
 - : 'a -> 'a Deferred.t = <fun>
# let three = return 3;;
 val three : int Deferred.t = <abstr>
# three;;
 - : int = 3
```

OCaml Utop ⁎ async/main.topscript , continued (part 11) ⁎ all code

Using `return`, we can make `count_lines` compile:

```
# let count_lines filename =
    Reader.file_contents filename
    >>= fun text ->
    return (List.length (String.split text ~on:'\n'))
  ;;
 val count_lines : string -> int Deferred.t = <fun>
```

OCaml Utop ⁎ async/main.topscript , continued (part 12) ⁎ all code

Together, `bind` and `return` form a design pattern in functional programming known as a *monad*. You'll run across this signature in many applications beyond just threads. Indeed, we already ran across monads in the section called “bind and Other Error Handling Idioms”.

Calling `bind` and `return` together is a fairly common pattern, and as such there is a standard shortcut for it called `Deferred.map`, which has the following signature:

```
# Deferred.map;;
 - : 'a Deferred.t -> f:('a -> 'b) -> 'b Deferred.t = <fun>
```

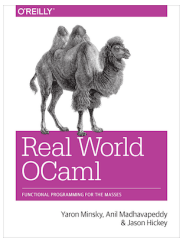
OCaml Utop ⁎ async/main.topscript , continued (part 13) ⁎ all code

and comes with its own infix equivalent, `>>|`. Using it, we can rewrite `count_lines` again a bit more succinctly:

```
# let count_lines filename =
    Reader.file_contents filename
    >>| fun text ->
    List.length (String.split text ~on:'\n')
  ;;
 val count_lines : string -> int Deferred.t = <fun>
# count_lines "/etc/hosts";;
 - : int = 11
```

OCaml Utop ⁎ async/main.topscript , continued (part 14) ⁎ all code

Note that `count_lines` returns a deferred, but **utop** waits for that deferred to become determined, and shows us the contents of the deferred instead.



Ivars and Upon

Deferreds are usually built using combinations of `bind`, `map` and `return`, but sometimes you want to construct a deferred that you can determine explicitly with usercode. This is done using an *ivar*. (The term ivar dates back to a language called Concurrent ML that was developed by John Reppy in the early '90s. The "i" in ivar stands for incremental.)

There are three fundamental operations for working with an ivar: you can create one, using `Ivar.create`; you can read off the deferred that corresponds to the ivar in question, using `Ivar.read`; and you can fill an ivar, thus causing the corresponding deferred to become determined, using `Ivar.fill`. These operations are illustrated below:

```
# let ivar = Ivar.create ();;
 val ivar : '_a Ivar.t = <abstr>
# let def = Ivar.read ivar;;
 val def : '_a Deferred.t = <abstr>
# Deferred.peek def;;
 - : '_a option = None
# Ivar.fill ivar "Hello";;
 - : unit = ()
# Deferred.peek def;;
 - : string option = Some "Hello"
```

OCaml Utop ⁎ async/main.topscript , continued (part 15) ⁎ all code

Ivars are something of a low-level feature; operators like `map`, `bind` and `return` are typically easier to use and think about. But ivars can be useful when you want to build a synchronization pattern that isn't already well supported.

As an example, imagine we wanted a way of scheduling a sequence of actions that would run after a fixed delay. In addition, we'd like to guarantee that these delayed actions are executed in the same order they were scheduled in. Here's a reasonable signature that captures this idea:

```
# module type Delayer_intf = sig
    type t
    val create : Time.Span.t -> t
    val schedule : t -> (unit -> 'a Deferred.t) -> 'a Deferred.t
  end;;
 module type Delayer_intf =
   sig
     type t
     val create : Core.Span.t -> t
     val schedule : t -> (unit -> 'a Deferred.t) -> 'a Deferred.t
   end
```

OCaml Utop ⁎ async/main.topscript , continued (part 16) ⁎ all code

An action is handed to `schedule` in the form of a deferred-returning thunk (a thunk is a function whose argument is of type `unit`). A deferred is handed back to the caller of `schedule` that will eventually be filled with the contents of the deferred value returned by the thunk. To implement this, we'll use an operator called `upon`, which has the following signature:

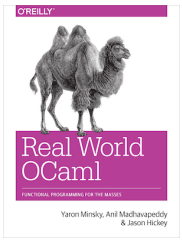
```
# upon;;
 - : 'a Deferred.t -> ('a -> unit) -> unit = <fun>
```

OCaml Utop ⁎ async/main.topscript , continued (part 17) ⁎ all code

Like `bind` and `return`, `upon` schedules a callback to be executed when the deferred it is passed is determined; but unlike those calls, it doesn't create a new deferred for this callback to fill.

Our delayer implementation is organized around a queue of thunks, where every call to `schedule` adds a thunk to the queue and also schedules a job in the future to grab a thunk off the queue and run it. The waiting will be done using the function `after`, which takes a time span and returns a deferred which becomes determined after that time span elapses:

```
# module Delayer : Delayer_intf = struct
    type t = { delay: Time.Span.t;
               jobs: (unit -> unit) Queue.t;
             }

    let create delay =
      { delay; jobs = Queue.create () }

    let schedule t thunk =
      let ivar = Ivar.create () in
```




```
    Queue.enqueue t.jobs (fun () ->
      upon (thunk ()) (fun x -> Ivar.fill ivar x));
    upon (after t.delay) (fun () ->
      let job = Queue.dequeue_exn t.jobs in
      job ());
    Ivar.read ivar
end;;
module Delayer : Delayer_intf
```

OCaml Utop ∗ async/main.topscript , continued (part 18) ∗ all code

This code isn't particularly long, but it is subtle. In particular, note how the queue of thunks is used to ensure that the enqueued actions are run in order, even if the thunks scheduled by `upon` are run out of order. This kind of subtlety is typical of code that involves ivars and `upon`, and because of this, you should stick to the simpler map/bind/return style of working with deferreds when you can.

EXAMPLES: AN ECHO SERVER

Now that we have the basics of Async under our belt, let's look at a small standalone Async program. In particular, we'll write an echo server, i.e., a program that accepts connections from clients and spits back whatever is sent to it.

The first step is to create a function that can copy data from an input to an output. Here, we'll use Async's `Reader` and `Writer` modules, which provide a convenient abstraction for working with input and output channels:

```
open Core.Std
open Async.Std

(* Copy data from the reader to the writer, using the provided buffer
   as scratch space *)
let rec copy_blocks buffer r w =
  Reader.read r buffer
  >>= function
  | `Eof -> return ()
  | `Ok bytes_read ->
    Writer.write w buffer ~len:bytes_read;
    Writer.flushed w
    >>= fun () ->
    copy_blocks buffer r w
```

OCaml ∗ async/echo.ml ∗ all code

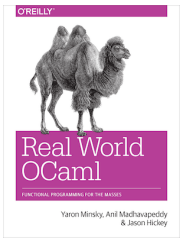
Bind is used in the code to sequence the operations: first, we call `Reader.read` to get a block of input. Then, when that's complete and if a new block was returned, we write that block to the writer. Finally, we wait until the writer's buffers are flushed, waiting on the deferred returned by `Writer.flushed`, at which point we recurse. If we hit an end-of-file condition, the loop is ended. The deferred returned by a call to `copy_blocks` becomes determined only once the end-of-file condition is hit.

One important aspect of how this is written is that it uses *pushback*, which is to say that if the writer can't make progress writing, the reader will stop reading. If you don't implement pushback in your servers, then a stopped client can cause your program to leak memory, since you'll need to allocate space for the data that's been read in but not yet written out.

You might also be concerned that the chain of deferreds that is built up as you go through the loop would lead to a memory leak. After all, this code constructs an ever-growing chain of binds, each of which creates a deferred. In this case, however, all of the deferreds should become determined precisely when the final deferred in the chain is determined, in this case, when the `Eof` condition is hit. Because of this, we could safely replace all of these deferreds with a single deferred. Async has logic to do just this, and so there's no memory leak after all. This is essentially a form of tail-call optimization, lifted to the Async monad.

`copy_blocks` provides the logic for handling a client connection, but we still need to set up a server to receive such connections and dispatch to `copy_blocks`. For this, we'll use Async's `Tcp` module, which has a collection of utilities for creating TCP clients and servers:

```
(** Starts a TCP server, which listens on the specified port, invoking
    copy_blocks every time a client connects. *)
let run () =
  let host_and_port =
    Tcp.Server.create
      ~on_handler_error:`Raise
      (Tcp.on_port 8765)
```




```
      (fun _addr r w ->
         let buffer = String.create (16 * 1024) in
         copy_blocks buffer r w)
  in
  ignore (host_and_port : (Socket.Address.Inet.t, int) Tcp.Server.t Deferred.t)
```

OCaml ∗ async/echo.ml , continued (part 1) ∗ all code

The result of calling `Tcp.Server.create` is a `Tcp.Server.t`, which is a handle to the server that lets you shut the server down. We don't use that functionality here, so we explicitly ignore `server` to suppress the unused-variables error. We put in a type annotation around the ignored value to make the nature of the value we're ignoring explicit.

The most important argument to `Tcp.Server.create` is the final one, which is the client connection handler. Notably, the preceding code does nothing explicit to close down the client connections when the communication is done. That's because the server will automatically shut down the connection once the deferred returned by the handler becomes determined.

Finally, we need to initiate the server and start the Async scheduler:

```
(* Call [run], and then start the scheduler *)
let () =
  run ();
  never_returns (Scheduler.go ())
```

OCaml ∗ async/echo.ml , continued (part 2) ∗ all code

One of the most common newbie errors with Async is to forget to run the scheduler. It can be a bewildering mistake, because without the scheduler, your program won't do anything at all; even calls to `printf` won't reach the terminal.

It's worth noting that even though we didn't spend much explicit effort on thinking about multiple clients, this server is able to handle many concurrent clients without further modification.

Now that we have the echo server, we can connect to the echo server using the netcat tool, which is invoked as `nc`:

```
$ ./echo.native &
$ nc 127.0.0.1 8765
 This is an echo server
 This is an echo server
 It repeats whatever I write.
 It repeats whatever I write.
```

Terminal ∗ async/run_echo.out ∗ all code

Functions that Never Return

You might wonder what's going on with the call to `never_returns`. `never_returns` is an idiom that comes from Core that is used to mark functions that don't return. Typically, a function that doesn't return is inferred as having return type `'a`:

```
# let rec loop_forever () = loop_forever ();;
 val loop_forever : unit -> 'a = <fun>
# let always_fail () = assert false;;
 val always_fail : unit -> 'a = <fun>
```

OCaml Utop ∗ async/main.topscript , continued (part 19) ∗ all code

This can be surprising when you call a function like this expecting it to return `unit`. The type-checker won't necessarily complain in such a case:

```
# let do_stuff n =
    let x = 3 in
    if n > 0 then loop_forever ();
    x + n
  ;;
 val do_stuff : int -> int = <fun>
```

OCaml Utop ∗ async/main.topscript , continued (part 20) ∗ all code

With a name like `loop_forever`, the meaning is clear enough. But with something like `Scheduler.go`, the fact that it never returns is less clear, and so we use the type system to make it more explicit by giving it a return type of `never_returns`. Let's do the same trick with `loop_forever`:

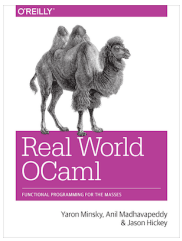


Table of Contents


```
# let rec loop_forever () : never_returns = loop_forever ();;
 val loop_forever : unit -> never_returns = <fun>
```

OCaml Utop ⋆ async/main.topscript , continued (part 21) ⋆ all code

The type `never_returns` is uninhabited, so a function can't return a value of type `never_returns`, which means only a function that never returns can have `never_returns` as its return type! Now, if we rewrite our `do_stuff` function, we'll get a helpful type error:

```
# let do_stuff n =
    let x = 3 in
    if n > 0 then loop_forever ();
    x + n
  ;;
 Characters 38-67:
 Error: This expression has type unit but an expression was expected of type
          never_returns
```

OCaml Utop ⋆ async/main.topscript , continued (part 22) ⋆ all code

We can resolve the error by calling the function `never_returns`:

```
# never_returns;;
 - : never_returns -> 'a = <fun>
# let do_stuff n =
    let x = 3 in
    if n > 0 then never_returns (loop_forever ());
    x + n
  ;;
 val do_stuff : int -> int = <fun>
```

OCaml Utop ⋆ async/main.topscript , continued (part 23) ⋆ all code

Thus, we got the compilation to go through by explicitly marking in the source that the call to `loop_forever` never returns.

Improving the Echo Server

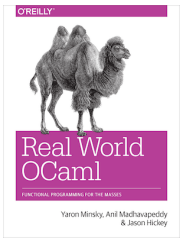
Let's try to go a little bit farther with our echo server by walking through a few improvements. In particular, we will:

- Add a proper command-line interface with `Command`
- Add a flag to specify the port to listen on and a flag to make the server echo back the capitalized version of whatever was sent to it
- Simplify the code using Async's `Pipe` interface

The following code does all of this:

```
open Core.Std
open Async.Std

let run ~uppercase ~port =
  let host_and_port =
    Tcp.Server.create
      ~on_handler_error:`Raise
      (Tcp.on_port port)
      (fun _addr r w ->
        Pipe.transfer (Reader.pipe r) (Writer.pipe w)
          ~f:(if uppercase then String.uppercase else Fn.id))
  in
  ignore (host_and_port : (Socket.Address.Inet.t, int) Tcp.Server.t Deferred.t);
  Deferred.never ()

let () =
  Command.async_basic
    ~summary:"Start an echo server"
    Command.Spec.(
      empty
      +> flag "-uppercase" no_arg
        ~doc:" Convert to uppercase before echoing back"
      +> flag "-port" (optional_with_default 8765 int)
        ~doc:" Port to listen on (default 8765)"
    )
    (fun uppercase port () -> run ~uppercase ~port)
  |> Command.run
```

OCaml ⋆ async/better_echo.ml ⋆ all code



Note the use of `Deferred.never` in the `run` function. As you might guess from the name, `Deferred.never` returns a deferred that is never determined. In this case, that indicates that the echo server doesn't ever shut down.

The biggest change in the preceding code is the use of Async's `Pipe`. A `Pipe` is an asynchronous communication channel that's used for connecting different parts of your program. You can think of it as a consumer/producer queue that uses deferreds for communicating when the pipe is ready to be read from or written to. Our use of pipes is fairly minimal here, but they are an important part of Async, so it's worth discussing them in some detail.

Pipes are created in connected read/write pairs:

```
# let (r,w) = Pipe.create ();;
 val r : '_a Pipe.Reader.t = <abstr>
 val w : '_a Pipe.Writer.t = <abstr>
```

OCaml Utop ⋆ async/main.topscript , continued (part 24) ⋆ all code

`r` and `w` are really just read and write handles to the same underlying object. Note that `r` and `w` have weakly polymorphic types, as discussed in the section called "Imperative Programming", and so can only contain values of a single, yet-to-be-determined type.

If we just try and write to the writer, we'll see that we block indefinitely in **utop**. You can break out of the wait by hitting **`Control-C`**:

```
# Pipe.write w "Hello World!";;
 Interrupted.
```

OCaml Utop ⋆ async/pipe_write_break.rawscript ⋆ all code

The deferred returned by write completes on its own once the value written into the pipe has been read out:

```
# let (r,w) = Pipe.create ();;
 val r : '_a Pipe.Reader.t = <abstr>
 val w : '_a Pipe.Writer.t = <abstr>
# let write_complete = Pipe.write w "Hello World!";;
 val write_complete : unit Deferred.t = <abstr>
# Pipe.read r;;
 - : [ `Eof | `Ok of string ] = `Ok "Hello World!"
# write_complete;;
 - : unit = ()
```

OCaml Utop ⋆ async/main.topscript , continued (part 25) ⋆ all code

In the function `run`, we're taking advantage of one of the many utility functions provided for pipes in the `Pipe` module. In particular, we're using `Pipe.transfer` to set up a process that takes data from a reader-pipe and moves it to a writer-pipe. Here's the type of `Pipe.transfer`:

```
# Pipe.transfer;;
 - : 'a Pipe.Reader.t -> 'b Pipe.Writer.t -> f:('a -> 'b) -> unit Deferred.t =
 <fun>
```

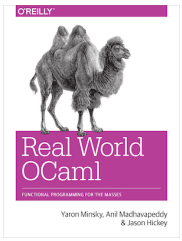
OCaml Utop ⋆ async/main.topscript , continued (part 26) ⋆ all code

The two pipes being connected are generated by the `Reader.pipe` and `Writer.pipe` call respectively. Note that pushback is preserved throughout the process, so that if the writer gets blocked, the writer's pipe will stop pulling data from the reader's pipe, which will prevent the reader from reading in more data.

Importantly, the deferred returned by `Pipe.transfer` becomes determined once the reader has been closed and the last element is transferred from the reader to the writer. Once that deferred becomes determined, the server will shut down that client connection. So, when a client disconnects, the rest of the shutdown happens transparently.

The command-line parsing for this program is based on the Command library that we introduced in Chapter 14, *Command-Line Parsing*. Opening `Async.Std`, shadows the `Command` module with an extended version that contains the `async_basic` call:

```
# Command.async_basic;;
 - : summary:string ->
     ?readme:(unit -> string) ->
```




```
    ('a, unit -> unit Deferred.t) Command.Spec.t -> 'a -> Command.t
= <fun>
```

OCaml Utop ⁎ async/main.topscript , continued (part 27) ⁎ all code

This differs from the ordinary `Command.basic` call in that the main function must return a `Deferred.t`, and that the running of the command (using `Command.run`) automatically starts the Async scheduler, without requiring an explicit call to `Scheduler.go`.

EXAMPLE: SEARCHING DEFINITIONS WITH DUCKDUCKGO

DuckDuckGo is a search engine with a freely available search interface. In this section, we'll use Async to write a small command-line utility for querying DuckDuckGo to extract definitions for a collection of terms.

Our code is going to rely on a number of other libraries, all of which can be installed using OPAM. Refer to this Real World OCaml page if you need help on the installation. Here's the list of libraries we'll need:

`textwrap`
A library for wrapping long lines. We'll use this for printing out our results.

`uri`
A library for handling URIs, or "Uniform Resource Identifiers," of which HTTP URLs are an example.

`yojson`
A JSON parsing library that was described in Chapter 15, *Handling JSON Data*.

`cohttp`
A library for creating HTTP clients and servers. We need Async support, which comes with the `cohttp.async` package.

Now let's dive into the implementation.

URI Handling

HTTP URLs, which identify endpoints across the Web, are actually part of a more general family known as Uniform Resource Identifiers (URIs). The full URI specification is defined in RFC3986 and is rather complicated. Luckily, the `uri` library provides a strongly typed interface that takes care of much of the hassle.

We'll need a function for generating the URIs that we're going to use to query the DuckDuckGo servers:

```
open Core.Std
open Async.Std

(* Generate a DuckDuckGo search URI from a query string *)
let query_uri query =
  let base_uri = Uri.of_string "http://api.duckduckgo.com/?format=json" in
  Uri.add_query_param base_uri ("q", [query])
```

OCaml ⁎ async/search.ml ⁎ all code

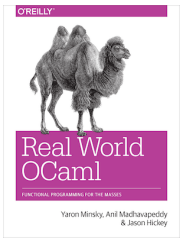
A `Uri.t` is constructed from the `Uri.of_string` function, and a query parameter `q` is added with the desired search query. The library takes care of encoding the URI correctly when outputting it in the network protocol.

Parsing JSON Strings

The HTTP response from DuckDuckGo is in JSON, a common (and thankfully simple) format that is specified in RFC4627. We'll parse the JSON data using the Yojson library, which was introduced in Chapter 15, *Handling JSON Data*.

We expect the response from DuckDuckGo to come across as a JSON record, which is represented by the `Assoc` tag in Yojson's JSON variant. We expect the definition itself to come across under either the key "Abstract" or "Definition," and so the following code looks under both keys, returning the first one for which a nonempty value is defined:

```
(* Extract the "Definition" or "Abstract" field from the DuckDuckGo results *)
let get_definition_from_json json =
  match Yojson.Safe.from_string json with
  | `Assoc kv_list ->
```




```
    let find key =
      begin match List.Assoc.find kv_list key with
      | None | Some (`String "") -> None
      | Some s -> Some (Yojson.Safe.to_string s)
      end
    in
    begin match find "Abstract" with
    | Some _ as x -> x
    | None -> find "Definition"
    end
  | _ -> None
```

OCaml ∗ async/search.ml , continued (part 1) ∗ all code

Executing an HTTP Client Query

Now let's look at the code for dispatching the search queries over HTTP, using the Cohttp library:

```
(* Execute the DuckDuckGo search *)
let get_definition word =
  Cohttp_async.Client.get (query_uri word)
  >>= fun (_, body) ->
  Pipe.to_list body
  >>| fun strings ->
  (word, get_definition_from_json (String.concat strings))
```

OCaml ∗ async/search.ml , continued (part 2) ∗ all code

To better understand what's going on, it's useful to look at the type for `Cohttp_async.Client.get`, which we can do in **utop**:

```
# #require "cohttp.async";;

# Cohttp_async.Client.get;;
 - : ?interrupt:unit Deferred.t ->
     ?headers:Cohttp.Header.t ->
     Uri.t -> (Cohttp.Response.t * string Pipe.Reader.t) Deferred.t
 = <fun>
```

OCaml Utop ∗ async/main.topscript , continued (part 28) ∗ all code

The `get` call takes as a required argument a URI and returns a deferred value containing a `Cohttp.Response.t` (which we ignore) and a pipe reader to which the body of the request will be written.

In this case, the HTTP body probably isn't very large, so we call `Pipe.to_list` to collect the strings from the pipe as a single deferred list of strings. We then join those strings using `String.concat` and pass the result through our parsing function.

Running a single search isn't that interesting from a concurrency perspective, so let's write code for dispatching multiple searches in parallel. First, we need code for formatting and printing out the search result:

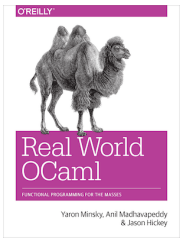
```
(* Print out a word/definition pair *)
let print_result (word,definition) =
  printf "%s\n%s\n\n%s\n\n"
    word
    (String.init (String.length word) ~f:(fun _ -> '-'))
    (match definition with
    | None -> "No definition found"
    | Some def ->
      String.concat ~sep:"\n"
        (Wrapper.wrap (Wrapper.make 70) def))
```

OCaml ∗ async/search.ml , continued (part 3) ∗ all code

We use the `Wrapper` module from the `textwrap` package to do the line wrapping. It may not be obvious that this routine is using Async, but it does: the version of `printf` that's called here is actually Async's specialized `printf` that goes through the Async scheduler rather than printing directly. The original definition of `printf` is shadowed by this new one when you open `Async.Std`. An important side effect of this is that if you write an Async program and forget to start the scheduler, calls like `printf` won't actually generate any output!

The next function dispatches the searches in parallel, waits for the results, and then prints:

```
(* Run many searches in parallel, printing out the results after they're all
   done. *)
let search_and_print words =
```




```
  Deferred.all (List.map words ~f:get_definition)
  >>| fun results ->
  List.iter results ~f:print_result
```

OCaml ∗ async/search.ml , continued (part 4) ∗ all code

We used `List.map` to call `get_definition` on each word, and `Deferred.all` to wait for all the results. Here's the type of `Deferred.all`:

```
# Deferred.all;;
 - : 'a Deferred.t list -> 'a list Deferred.t = <fun>
```

OCaml Utop ∗ async/main.topscript , continued (part 29) ∗ all code

Note that the list returned by `Deferred.all` reflects the order of the deferreds passed to it. As such, the definitions will be printed out in the same order that the search words are passed in, no matter what order the queries return in. We could rewrite this code to print out the results as they're received (and thus potentially out of order) as follows:

```
(* Run many searches in parallel, printing out the results as you go *)
let search_and_print words =
  Deferred.all_unit (List.map words ~f:(fun word ->
    get_definition word >>| print_result))
```

OCaml ∗ async/search_out_of_order.ml , continued (part 1) ∗ all code

The difference is that we both dispatch the query and print out the result in the closure passed to `map`, rather than wait for all of the results to get back and then print them out together. We use `Deferred.all_unit`, which takes a list of `unit` deferreds and returns a single `unit` deferred that becomes determined when every deferred on the input list is determined. We can see the type of this function in **utop**:

```
# Deferred.all_unit;;
 - : unit Deferred.t list -> unit Deferred.t = <fun>
```

OCaml Utop ∗ async/main.topscript , continued (part 30) ∗ all code

Finally, we create a command-line interface using `Command.async_basic`:

```
let () =
  Command.async_basic
    ~summary:"Retrieve definitions from duckduckgo search engine"
    Command.Spec.(
      empty
      +> anon (sequence ("word" %: string))
    )
    (fun words () -> search_and_print words)
  |> Command.run
```

OCaml ∗ async/search.ml , continued (part 5) ∗ all code

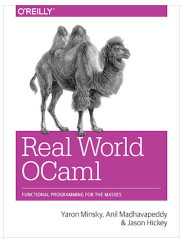
And that's all we need for a simple but usable definition searcher:

```
$ corebuild -pkg cohttp.async,yojson,textwrap search.native
$ ./search.native "Concurrent Programming" "OCaml"
 Concurrent Programming
 ----------------------

 "Concurrent computing is a form of computing in which programs are
 designed as collections of interacting computational processes that
 may be executed in parallel."

 OCaml
 -----

 "OCaml, originally known as Objective Caml, is the main implementation
 of the Caml programming language, created by Xavier Leroy, Jérôme
 Vouillon, Damien Doligez, Didier Rémy and others in 1996."
```

Terminal ∗ async/run_search.out ∗ all code

EXCEPTION HANDLING

When programming with external resources, errors are everywhere: everything from a flaky server to a network outage to exhausting of local resources can lead to a runtime error. When



programming in OCaml, some of these errors will show up explicitly in a function's return type, and some of them will show up as exceptions. We covered exception handling in OCaml in the section called "Exceptions", but as we'll see, exception handling in a concurrent program presents some new challenges.

Let's get a better sense of how exceptions work in Async by creating an asynchronous computation that (sometimes) fails with an exception. The function `maybe_raise` blocks for half a second, and then either throws an exception or returns `unit`, alternating between the two behaviors on subsequent calls:

```
# let maybe_raise =
    let should_fail = ref false in
    fun () ->
      let will_fail = !should_fail in
      should_fail := not will_fail;
      after (Time.Span.of_sec 0.5)
      >>= fun () ->
      if will_fail then raise Exit else return ()
 ;;
 val maybe_raise : unit -> unit Deferred.t = <fun>
# maybe_raise ();;
 - : unit = ()
# maybe_raise ();;
 Exception:
 (lib/monitor.ml.Error_
  ((exn Exit) (backtrace ""))
   (monitor
    (((name block_on_async) (here ()) (id 55) (has_seen_error true)
      (someone_is_listening true) (kill_index 0))
     ((name main) (here ()) (id 1) (has_seen_error false)
      (someone_is_listening false) (kill_index 0))))).
```

OCaml Utop ⋆ async/main.topscript , continued (part 31) ⋆ all code

In **utop**, the exception thrown by `maybe_raise ()` terminates the evaluation of just that expression, but in a standalone program, an uncaught exception would bring down the entire process.

So, how could we capture and handle such an exception? You might try to do this using OCaml's built-in `try/with` statement, but as you can see that doesn't quite do the trick:

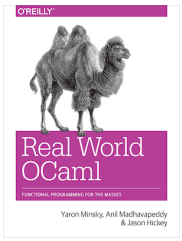
```
# let handle_error () =
    try
      maybe_raise ()
      >>| fun () -> "success"
    with _ -> return "failure"
  ;;
 val handle_error : unit -> string Deferred.t = <fun>
# handle_error ();;
 - : string = "success"
# handle_error ();;
 Exception:
 (lib/monitor.ml.Error_
  ((exn Exit) (backtrace ""))
   (monitor
    (((name block_on_async) (here ()) (id 58) (has_seen_error true)
      (someone_is_listening true) (kill_index 0))
     ((name main) (here ()) (id 1) (has_seen_error false)
      (someone_is_listening false) (kill_index 0))))).
```

OCaml Utop ⋆ async/main.topscript , continued (part 32) ⋆ all code

This didn't work because `try/with` only captures exceptions that are thrown in the code directly executed within it, while `maybe_raise` schedules an Async job to run in the future, and it's that job that throws an exception.

We can capture this kind of asynchronous error using the `try_with` function provided by Async:

```
# let handle_error () =
    try_with (fun () -> maybe_raise ())
    >>| function
    | Ok ()   -> "success"
    | Error _ -> "failure"
  ;;
 val handle_error : unit -> string Deferred.t = <fun>
# handle_error ();;
 - : string = "success"
```




```
# handle_error ();;
 - : string = "failure"
```

OCaml Utop ⁎ async/main.topscript , continued (part 33) ⁎ all code

`try_with f` takes as its argument a deferred-returning thunk `f` and returns a deferred that becomes determined either as `Ok` of whatever `f` returned, or `Error exn` if `f` threw an exception before its return value became determined.

Monitors

`try_with` is a great way of handling exceptions in Async, but it's not the whole story. All of Async's exception-handling mechanisms, `try_with` included, are built on top of Async's system of *monitors*, which are inspired by the error-handling mechanism in Erlang of the same name. Monitors are fairly low-level and are only occasionally used directly, but it's nonetheless worth understanding how they work.

In Async, a monitor is a context that determines what to do when there is an unhandled exception. Every Async job runs within the context of some monitor, which, when the job is running, is referred to as the current monitor. When a new Async job is scheduled, say, using `bind` or `map`, it inherits the current monitor of the job that spawned it.

Monitors are arranged in a tree—when a new monitor is created (say, using `Monitor.create`), it is a child of the current monitor. You can explicitly run jobs within a monitor using `within`, which takes a thunk that returns a nondeferred value, or `within'`, which takes a thunk that returns a deferred. Here's an example:

```
# let blow_up () =
    let monitor = Monitor.create ~name:"blow up monitor" () in
    within' ~monitor maybe_raise
  ;;
 val blow_up : unit -> unit Deferred.t = <fun>
# blow_up ();;
 - : unit = ()
# blow_up ();;
 Exception:
 (lib/monitor.ml.Error_
  ((exn Exit) (backtrace ""))
   (monitor
    (((name "blow up monitor") (here ()) (id 69) (has_seen_error true)
      (someone_is_listening false) (kill_index 0))
     ((name block_on_async) (here ()) (id 68) (has_seen_error false)
      (someone_is_listening true) (kill_index 0))
     ((name main) (here ()) (id 1) (has_seen_error false)
      (someone_is_listening false) (kill_index 0))))))).
```

OCaml Utop ⁎ async/main.topscript , continued (part 34) ⁎ all code

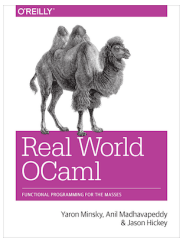
In addition to the ordinary stack-trace, the exception displays the trace of monitors through which the exception traveled, starting at the one we created, called "blow up monitor." The other monitors you see come from **utop**'s special handling of deferreds.

Monitors can do more than just augment the error-trace of an exception. You can also use a monitor to explicitly handle errors delivered to that monitor. The `Monitor.errors` call is a particularly important one. It detaches the monitor from its parent, handing back the stream of errors that would otherwise have been delivered to the parent monitor. This allows one to do custom handling of errors, which may include reraising errors to the parent. Here is a very simple example of a function that captures and ignores errors in the processes it spawns:

```
# let swallow_error () =
    let monitor = Monitor.create () in
    Stream.iter (Monitor.errors monitor) ~f:(fun _exn ->
      printf "an error happened\n");
    within' ~monitor (fun () ->
      after (Time.Span.of_sec 0.5) >>= fun () -> failwith "Kaboom!")
  ;;
 val swallow_error : unit -> 'a Deferred.t = <fun>
# swallow_error ();;
 an error happened
```

OCaml Utop ⁎ async/main-35.rawscript ⁎ all code

The message "an error happened" is printed out, but the deferred returned by `swallow_error` is never determined. This makes sense, since the calculation never actually completes, so there's no value to return. You can break out of this in **utop** by hitting **`Control+C`**.



Here's an example of a monitor that passes some exceptions through to the parent and handles others. Exceptions are sent to the parent using `Monitor.send_exn`, with `Monitor.current` being called to find the current monitor, which is the parent of the newly created monitor:

```
# exception Ignore_me;;
 exception Ignore_me
# let swallow_some_errors exn_to_raise =
    let child_monitor  = Monitor.create  () in
    let parent_monitor = Monitor.current () in
    Stream.iter (Monitor.errors child_monitor) ~f:(fun error ->
      match Monitor.extract_exn error with
      | Ignore_me -> printf "ignoring exn\n"
      | _ -> Monitor.send_exn parent_monitor error);
    within' ~monitor:child_monitor (fun () ->
       after (Time.Span.of_sec 0.5)
       >>= fun () -> raise exn_to_raise)
  ;;
 val swallow_some_errors : exn -> 'a Deferred.t = <fun>
```

OCaml Utop ⋆ async/main.topscript , continued (part 36) ⋆ all code

Note that we use `Monitor.extract_exn` to grab the underlying exception that was thrown. Async wraps exceptions it catches with extra information, including the monitor trace, so you need to grab the underlying exception to match on it.

If we pass in an exception other than `Ignore_me`, like, say, the built-in exception `Not_found`, then the exception will be passed to the parent monitor and delivered as usual:

```
# swallow_some_errors Not_found;;
 Exception:
 (lib/monitor.ml.Error_
  ((exn Not_found) (backtrace ""))
   (monitor
    (((name (id 72)) (here ()) (id 72) (has_seen_error true)
      (someone_is_listening true) (kill_index 0))
     ((name block_on_async) (here ()) (id 71) (has_seen_error true)
      (someone_is_listening true) (kill_index 0))
     ((name main) (here ()) (id 1) (has_seen_error false)
      (someone_is_listening false) (kill_index 0))))).
```

OCaml Utop ⋆ async/main.topscript , continued (part 37) ⋆ all code

If instead we use `Ignore_me`, the exception will be ignorred, and the deferred never becomes determined:

```
# swallow_some_errors Ignore_me;;
 ignoring exn
```

OCaml Utop ⋆ async/main-38.rawscript ⋆ all code

In practice, you should rarely use monitors directly, and instead use functions like `try_with` and `Monitor.protect` that are built on top of monitors. One example of a library that uses monitors directly is `Tcp.Server.create`, which tracks both exceptions thrown by the logic that handles the network connection and by the callback for responding to an individual request, in either case responding to an exception by closing the connection. It is for building this kind of custom error handling that monitors can be helpful.

Example: Handling Exceptions with DuckDuckGo

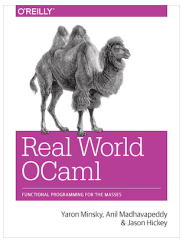
Let's now go back and improve the exception handling of our DuckDuckGo client. In particular, we'll change it so that any query that fails is reported without preventing other queries from completing.

The search code as it is fails rarely, so let's make a change that allows us to trigger failures more predictably. We'll do this by making it possible to distribute the requests over multiple servers. Then, we'll handle the errors that occur when one of those servers is misspecified.

First we'll need to change `query_uri` to take an argument specifying the server to connect to:

```
(* Generate a DuckDuckGo search URI from a query string *)
let query_uri ~server query =
  let base_uri =
    Uri.of_string (String.concat ["http://";server;"/?format=json"])
  in
  Uri.add_query_param base_uri ("q", [query])
```

OCaml ⋆ async/search_with_configurable_server.ml , continued (part 1) ⋆ all code



In addition, we'll make the necessary changes to get the list of servers on the command-line, and to distribute the search queries round-robin across the list of servers. Now, let's see what happens if we rebuild the application and run it giving it a list of servers, some of which won't respond to the query:

```
$ corebuild -pkg cohttp.async,yojson,textwrap \
    search_with_configurable_server.native
$ ./search_with_configurable_server.native \
    -servers localhost,api.duckduckgo.com \
    "Concurrent Programming" OCaml
 ("unhandled exception"
  ((lib/monitor.ml.Error_
    ((exn (Unix.Unix_error "Connection refused" connect 127.0.0.1:80))
     (backtrace
      ("Raised by primitive operation at file \"lib/unix_syscalls.ml\", line 797, ch
       "Called from file \"lib/deferred.ml\", line 20, characters 62-65"
       "Called from file \"lib/scheduler.ml\", line 125, characters 6-17"
       "Called from file \"lib/jobs.ml\", line 65, characters 8-13" ""))
     (monitor
      (((name Tcp.close_sock_on_error) (here ()) (id 5) (has_seen_error true)
        (someone_is_listening true) (kill_index 0))
       ((name main) (here ()) (id 1) (has_seen_error true)
        (someone_is_listening false) (kill_index 0))))))
   (Pid 15971)))
```

Terminal ∗ async/run_search_with_configurable_server.out ∗ all code

As you can see, we got a "Connection refused" failure, which ends the entire program, even though one of the two queries would have gone through successfully on its own. We can handle the failures of individual connections separately by using the `try_with` function within each call to `get_definition`, as follows:

```
(* Execute the DuckDuckGo search *)
let get_definition ~server word =
  try_with (fun () ->
    Cohttp_async.Client.get (query_uri ~server word)
    >>= fun  (_, body) ->
    Pipe.to_list body
    >>| fun strings ->
    (word, get_definition_from_json (String.concat strings)))
  >>| function
  | Ok (word,result) -> (word, Ok result)
  | Error _          -> (word, Error "Unexpected failure")
```

OCaml ∗ async/search_with_error_handling.ml , continued (part 1) ∗ all code

Here, we first use `try_with` to capture the exception, and then use map (the `>>|` operator) to convert the error into the form we want: a pair whose first element is the word being searched for, and the second element is the (possibly erroneous) result.

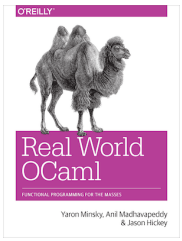
Now we just need to change the code for `print_result` so that it can handle the new type:

```
(* Print out a word/definition pair *)
let print_result (word,definition) =
  printf "%s\n%s\n\n%s\n\n"
    word
    (String.init (String.length word) ~f:(fun _ -> '-'))
    (match definition with
     | Error s -> "DuckDuckGo query failed: " ^ s
     | Ok None -> "No definition found"
     | Ok (Some def) ->
       String.concat ~sep:"\n"
         (Wrapper.wrap (Wrapper.make 70) def))
```

OCaml ∗ async/search_with_error_handling.ml , continued (part 2) ∗ all code

Now, if we run that same query, we'll get individualized handling of the connection failures:

```
$ corebuild -pkg cohttp.async,yojson,textwrap \
    search_with_error_handling.native
$ ./search_with_error_handling.native \
    -servers localhost,api.duckduckgo.com \
    "Concurrent Programming" OCaml
```




```
Concurrent Programming
----------------------

DuckDuckGo query failed: Unexpected failure

OCaml
-----

"OCaml, originally known as Objective Caml, is the main implementation
of the Caml programming language, created by Xavier Leroy, Jérôme
Vouillon, Damien Doligez, Didier Rémy and others in 1996."
```

Terminal ∗ async/run_search_with_error_handling.out ∗ all code

Now, only the query that went to `localhost` failed.

Note that in this code, we're relying on the fact that `Cohttp_async.Client.get` will clean up after itself after an exception, in particular by closing its file descriptors. If you need to implement such functionality directly, you may want to use the `Monitor.protect` call, which is analogous to the `protect` call described in the section called "Cleaning Up in the Presence of Exceptions".

TIMEOUTS, CANCELLATION, AND CHOICES

In a concurrent program, one often needs to combine results from multiple, distinct concurrent subcomputations going on in the same program. We already saw this in our DuckDuckGo example, where we used `Deferred.all` and `Deferred.all_unit` to wait for a list of deferreds to become determined. Another useful primitive is `Deferred.both`, which lets you wait until two deferreds of different types have returned, returning both values as a tuple. Here, we use the function `sec`, which is shorthand for creating a time-span equal to a given number of seconds:

```
# let string_and_float = Deferred.both
   (after (sec 0.5)  >>| fun () -> "A")
   (after (sec 0.25) >>| fun () -> 32.33);;
 val string_and_float : (string * float) Deferred.t = <abstr>
# string_and_float;;
 - : string * float = ("A", 32.33)
```

OCaml Utop ∗ async/main.topscript , continued (part 39) ∗ all code

Sometimes, however, we want to wait only for the first of multiple events to occur. This happens particularly when dealing with timeouts. In that case, we can use the call `Deferred.any`, which, given a list of deferreds, returns a single deferred that will become determined once any of the values on the list is determined:

```
# Deferred.any [ (after (sec 0.5) >>| fun () -> "half a second")
               ; (after (sec 10.) >>| fun () -> "ten seconds") ] ;;
 - : string = "half a second"
```

OCaml Utop ∗ async/main.topscript , continued (part 40) ∗ all code

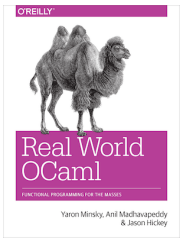
Let's use this to add timeouts to our DuckDuckGo searches. The following code is a wrapper for `get_definition` that takes a timeout (in the form of a `Time.Span.t`) and returns either the definition, or, if that takes too long, an error:

```
let get_definition_with_timeout ~server ~timeout word =
  Deferred.any
    [ (after timeout >>| fun () -> (word,Error "Timed out"))
    ; (get_definition ~server word
       >>| fun (word,result) ->
       let result' = match result with
         | Ok _ as x -> x
         | Error _ -> Error "Unexpected failure"
       in
       (word,result')
      )
    ]
```

OCaml ∗ async/search_with_timeout.ml , continued (part 1) ∗ all code

We use `>>|` above to transform the deferred values we're waiting for so that `Deferred.any` can choose between values of the same type.

A problem with this code is that the HTTP query kicked off by `get_definition` is not actually shut down when the timeout fires. As such, `get_definition_with_timeout` can leak an open



connection. Happily, Cohttp does provide a way of shutting down a client. You can pass a deferred under the label `interrupt` to `Cohttp_async.Client.get`. Once `interrupt` is determined, the client connection will be shut down.

The following code shows how you can change `get_definition` and `get_definition_with_timeout` to cancel the `get` call if the timeout expires:

```
(* Execute the DuckDuckGo search *)
let get_definition ~server ~interrupt word =
  try_with (fun () ->
    Cohttp_async.Client.get ~interrupt (query_uri ~server word)
    >>= fun  (_, body) ->
    Pipe.to_list body
    >>| fun strings ->
    (word, get_definition_from_json (String.concat strings)))
  >>| function
  | Ok (word,result) -> (word, Ok result)
  | Error exn        -> (word, Error exn)
```

OCaml ∗ async/search_with_timeout_no_leak_simple.ml , continued (part 1) ∗ all code

Next, we'll modify `get_definition_with_timeout` to create a deferred to pass in to `get_definition`, which will become determined when our timeout expires:

```
let get_definition_with_timeout ~server ~timeout word =
  get_definition ~server ~interrupt:(after timeout) word
  >>| fun (word,result) ->
  let result' = match result with
    | Ok _ as x -> x
    | Error _ -> Error "Unexpected failure"
  in
  (word,result')
```

OCaml ∗ async/search_with_timeout_no_leak_simple.ml , continued (part 2) ∗ all code

This will work and will cause the connection to shutdown cleanly when we time out; but our code no longer explicitly knows whether or not the timeout has kicked in. In particular, the error message on a timeout will now be `"Unexpected failure"` rather than `"Timed out"`, which it was in our previous implementation.

We can get more precise handling of timeouts using Async's `choose` function. `choose` lets you pick among a collection of different deferreds, reacting to exactly one of them. Each deferred is paired, using the function `choice`, with a function that is called if and only if that deferred is chosen. Here's the type signature of `choice` and `choose`:

```
# choice;;
 - : 'a Deferred.t -> ('a -> 'b) -> 'b Deferred.choice = <fun>
# choose;;
 - : 'a Deferred.choice list -> 'a Deferred.t = <fun>
```

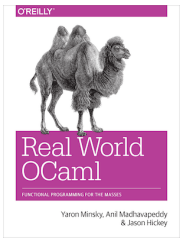
OCaml Utop ∗ async/main.topscript , continued (part 41) ∗ all code

Note that there's no guarantee that the winning deferred will be the one that becomes determined first. But `choose` does guarantee that only one `choice` will be chosen, and only the chosen `choice` will execute the attached function.

In the following example, we use `choose` to ensure that the `interrupt` deferred becomes determined if and only if the timeout deferred is chosen. Here's the code:

```
let get_definition_with_timeout ~server ~timeout word =
  let interrupt = Ivar.create () in
  choose
    [ choice (after timeout) (fun () ->
       Ivar.fill interrupt ();
       (word,Error "Timed out"))
    ; choice (get_definition ~server ~interrupt:(Ivar.read interrupt) word)
        (fun (word,result) ->
           let result' = match result with
             | Ok _ as x -> x
             | Error _ -> Error "Unexpected failure"
           in
           (word,result')
        )
    ]
```

OCaml ∗ async/search_with_timeout_no_leak.ml , continued (part 2) ∗ all code



Now, if we run this with a suitably small timeout, we'll see that one query succeeds and the other fails reporting a timeout:

```
$ corebuild -pkg cohttp.async,yojson,textwrap \
      search_with_timeout_no_leak.native
$ ./search_with_timeout_no_leak.native \
      "concurrent programming" ocaml -timeout 0.2s
 concurrent programming
 ----------------------

 DuckDuckGo query failed: Timed out

 ocaml
 -----

 "OCaml or Objective Caml, is the main implementation of the Caml
 programming language, created by Xavier Leroy, Jérôme Vouillon,
 Damien Doligez, Didier Rémy and others in 1996."
```

Terminal ∗ async/run_search_with_timeout_no_leak.out ∗ all code

WORKING WITH SYSTEM THREADS

Although we haven't worked with them yet, OCaml does have built-in support for true system threads, i.e., kernel-level threads whose interleaving is controlled by the operating system. We discussed in the beginning of the chapter why Async is generally a better choice than system threads, but even if you mostly use Async, OCaml's system threads are sometimes necessary, and it's worth understanding them.

The most surprising aspect of OCaml's system threads is that they don't afford you any access to physical parallelism. That's because OCaml's runtime has a single runtime lock that at most one thread can be holding at a time.

Given that threads don't provide physical parallelism, why are they useful at all?

The most common reason for using system threads is that there are some operating system calls that have no nonblocking alternative, which means that you can't run them directly in a system like Async without blocking your entire program. For this reason, Async maintains a thread pool for running such calls. Most of the time, as a user of Async you don't need to think about this, but it is happening under the covers.

Another reason to have multiple threads is to deal with non-OCaml libraries that have their own event loop or for another reason need their own threads. In that case, it's sometimes useful to run some OCaml code on the foreign thread as part of the communication to your main program. OCaml's foreign function interface is discussed in more detail in Chapter 19, *Foreign Function Interface*.

Another occasional use for system threads is to better interoperate with compute-intensive OCaml code. In Async, if you have a long-running computation that never calls `bind` or `map`, then that computation will block out the Async runtime until it completes.

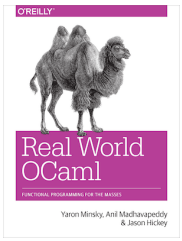
One way of dealing with this is to explicitly break up the calculation into smaller pieces that are separated by binds. But sometimes this explicit yielding is impractical, since it may involve intrusive changes to an existing codebase. Another solution is to run the code in question in a separate thread. Async's `In_thread` module provides multiple facilities for doing just this, `In_thread.run` being the simplest. We can simply write:

```
# let def = In_thread.run (fun () -> List.range 1 10);;
 val def : int list Deferred.t = <abstr>
# def;;
 - : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9]
```

OCaml Utop ∗ async/main.topscript , continued (part 42) ∗ all code

to cause `List.range 1 10` to be run on one of Async's worker threads. When the computation is complete, the result is placed in the deferred, where it can be used in the ordinary way from Async.

Interoperability between Async and system threads can be quite tricky. Consider the following function for testing how responsive Async is. The function takes a deferred-returning thunk, and it first runs that thunk, and then uses `Clock.every` to wake up every 100 milliseconds and print



out a timestamp, until the returned deferred becomes determined, at which point it prints out one last timestamp:

```
# let log_delays thunk =
    let start = Time.now () in
    let print_time () =
      let diff = Time.diff (Time.now ()) start in
      printf "%s, " (Time.Span.to_string diff)
    in
    let d = thunk () in
    Clock.every (sec 0.1) ~stop:d print_time;
    d >>| fun () -> print_time (); printf "\n"
  ;;
 val log_delays : (unit -> unit Deferred.t) -> unit Deferred.t = <fun>
```

OCaml Utop ∗ async/main.topscript , continued (part 43) ∗ all code

If we feed this function a simple timeout deferred, it works as you might expect, waking up roughly every 100 milliseconds:

```
# log_delays (fun () -> after (sec 0.5));;
 0.154972ms, 102.126ms, 203.658ms, 305.73ms, 407.903ms, 501.563ms,
 - : unit = ()
```

OCaml Utop ∗ async/main-44.rawscript ∗ all code

Now see what happens if, instead of waiting on a clock event, we wait for a busy loop to finish running:

```
# let busy_loop n =
    let x = ref None in
    for i = 1 to 100_000_000 do x := Some i done
  ;;
 val busy_loop : 'a -> unit = <fun>
# log_delays (fun () -> return (busy_loop ()));;
 19.2185s,
 - : unit = ()
```

OCaml Utop ∗ async/main-45.rawscript ∗ all code

As you can see, instead of waking up 10 times a second, `log_delays` is blocked out entirely while `busy_loop` churns away.

If, on the other hand, we use `In_thread.run` to offload this to a different system thread, the behavior will be different:

```
# log_delays (fun () -> In_thread.run busy_loop);;
 0.332117ms, 16.6319s, 18.8722s,
 - : unit = ()
```

OCaml Utop ∗ async/main-46.rawscript ∗ all code

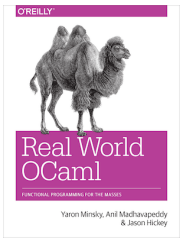
Now `log_delays` does get a chance to run, but not nearly as often as every 100 milliseconds. The reason is that now that we're using system threads, we are at the mercy of the operating system to decide when each thread gets scheduled. The behavior of threads is very much dependent on the operating system and how it is configured.

Another tricky aspect of dealing with OCaml threads has to do with allocation. When compiling to native code, OCaml's threads only get a chance to give up the runtime lock when they interact with the allocator, so if there's a piece of code that doesn't allocate at all, then it will never allow another OCaml thread to run. Bytecode doesn't have this behavior, so if we run a nonallocating loop in bytecode, our timer process will get to run:

```
# let noalloc_busy_loop () =
    for i = 0 to 100_000_000 do () done
;;
 val noalloc_busy_loop : unit -> unit = <fun>
# log_delays (fun () -> In_thread.run noalloc_busy_loop);;
 0.169039ms, 4.58345s, 4.77866s, 4.87957s, 12.4723s, 15.0134s,
 - : unit = ()
```

OCaml Utop ∗ async/main-47.rawscript ∗ all code

But if we compile this to a native-code executable, then the nonallocating busy loop will block anything else from running:




```
$ corebuild -pkg async native_code_log_delays.native
$ ./native_code_log_delays.native
 15.5686s,
$
```

Terminal ∗ async/run_native_code_log_delays.out ∗ all code

The takeaway from these examples is that predicting thread interleavings is a subtle business. Staying within the bounds of Async has its limitations, but it leads to more predictable behavior.

Thread-Safety and Locking

Once you start working with system threads, you'll need to be careful about mutable data structures. Most mutable OCaml data structures do not have well-defined semantics when accessed concurrently by multiple threads. The issues you can run into range from runtime exceptions to corrupted data structures to, in some rare cases, segfaults. That means you should always use mutexes when sharing mutable data between different systems threads. Even data structures that seem like they should be safe but are mutable under the covers, like lazy values, can have undefined behavior when accessed from multiple threads.

There are two commonly available mutex packages for OCaml: the `Mutex` module that's part of the standard library, which is just a wrapper over OS-level mutexes and `Nano_mutex`, a more efficient alternative that takes advantage of some of the locking done by the OCaml runtime to avoid needing to create an OS-level mutex much of the time. As a result, creating a `Nano_mutex.t` is 20 times faster than creating a `Mutex.t`, and acquiring the mutex is about 40 percent faster.

Overall, combining Async and threads is quite tricky, but it can be done safely if the following hold:

- There is no shared mutable state between the various threads involved.
- The computations executed by `In_thread.run` do not make any calls to the Async library.

It is possible to safely use threads in ways that violate these constraints. In particular, foreign threads can acquire the Async lock using calls from the `Thread_safe` module in Async, and thereby run Async computations safely. This is a very flexible way of connecting threads to the Async world, but it's a complex use case that is beyond the scope of this chapter.

< Previous Next >