17 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 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.

17.1 Async Basics

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

```
# open Core;;
# #show In_channel.read_all;;
val read_all : string -> string
# 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."
```

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 call is complete.

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;;
# #show Reader.file_contents;;
val file_contents : string -> string Deferred.t
```

We first load the Async package in the toplevel using `#require`, and then open the module. Async, like Core, is designed to be an extension to your basic programming environment, and is intended to be opened.

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`.

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

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."
```

Slightly confusingly, the type shown here is not the type of `contents`, which is `string Deferred.t`, but rather `string`, the type of the value contained within that deferred.

If we peek again, we'll see that the value of `contents` has been filled in.

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

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`. Here's the type-signature of `bind`.

```
# #show Deferred.bind;;
val bind : 'a Deferred.t -> f:('a -> 'b Deferred.t) -> 'b Deferred.t
```

bind is effectively a way of sequencing concurrent computations. In particular, Deferred.bind d ~f causes f to be called after the value of d has been 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)
      ~f:(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."
```

Again, bind is acting as a sequencing operator, causing the file to be saved via the call to Writer.save only after the contents of the file were first read via Reader.file_contents.

Writing out Deferred.bind explicitly can be rather verbose, and so Async 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>
```

Here, we've dropped the parentheses around the function on the right-hand 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 infix 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');;
Line 4, characters 5-45:
Error: This expression has type int but an expression was expected of
     type
         'a Deferred.t
```

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.t, but we've provided it with a function that returns the result directly. What we need is return, a function provided by Async that takes an ordinary value and wraps it up in a deferred.

```
# #show_val return;;
val return : 'a -> 'a Deferred.t
# let three = return 3;;
```

```
val three : int Deferred.t = <abstr>
# three;;
- : int = 3
```

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>
```

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 Chapter 8.1.3 (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:

```
# #show Deferred.map;;
val map : 'a Deferred.t -> f:('a -> 'b) -> 'b Deferred.t
```

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 = 10
```

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.

17.1.1 Using Let Syntax

As was discussed in Chapter 8.1.3 (bind and Other Error Handling Idioms), there is a special syntax, which we call *let syntax*, designed for working with monads, which we can enable by enabling ppx_let.

```
# #require "ppx_let";;
```

Here's what the bind-using version of count_lines looks like using that syntax.

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

And here's the map-based version of count_lines.

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

The difference here is just syntactic, with these examples compiling down to the same thing as the corresponding examples written using infix operators. What's nice about let syntax is that it highlights the analogy between monadic bind and OCaml's built-in let-binding, thereby making your code more uniform and more readable.

Let syntax works for any monad, and you decide which monad is in use by opening the appropriate `Let_syntax` module. Opening `Async` also implicitly opens `Deferred.Let_syntax`, but in some contexts you may want to do that explicitly.

For the most part, let syntax is easier to read and work with, and you should default to it when using Async, which is what we'll do for the remainder of the chapter.

17.1.2 Ivars and Upon

Deferreds are usually built using combinations of `bind`, `map` and `return`, but sometimes you want to construct a deferred where you can programmatically decide when it gets filled in. This can be 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 : '_weak1 Ivar.t =
  {Async_kernel__.Types.Ivar.cell = Async_kernel__Types.Cell.Empty}
# let def = Ivar.read ivar;;
val def : '_weak2 Deferred.t = <abstr>
# Deferred.peek def;;
- : '_weak3 option = None
# Ivar.fill ivar "Hello";;
- : unit = ()
# Deferred.peek def;;
- : string option = Some "Hello"
```

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 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 : Time.Span.t -> t
  val schedule : t -> (unit -> 'a Deferred.t) -> 'a Deferred.t
end
```

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:

```
# #show upon;;
val upon : 'a Deferred.t -> ('a -> unit) -> unit
```

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
```

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 the order they were scheduled, 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.

Understanding bind in Terms of Ivars and upon

Here's roughly what happens when you write `let d' = Deferred.bind d ~f`.

- A new ivar `i` is created to hold the final result of the computation. The corresponding deferred is returned
- A function is registered to be called when the deferred `d` becomes determined.
- That function, once run, calls `f` with the value that was determined for `d`.
- Another function is registered to be called when the deferred returned by `f` becomes determined.
- When that function is called, it uses it to fill `i`, causing the corresponding deferred it to become determined.

That sounds like a lot, but we can implement this relatively concisely.

```
# let my_bind d ~f =
    let i = Ivar.create () in
    upon d (fun x -> upon (f x) (fun y -> Ivar.fill i y));
    Ivar.read i;;
val my_bind : 'a Deferred.t -> f:('a -> 'b Deferred.t) -> 'b Deferred.t
    =
  <fun>
```

Async's real implementation has more optimizations and is therefore more complicated. But the above implementation is still a useful first-order mental model for how bind works under the covers. And it's another good example of how `upon` and ivars can be useful for building concurrency primitives.

17.2 Example: 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
open Async

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

Bind is used in the code to sequence the operations, with a `bind` marking each place we wait.

- First, we call `Reader.read` to get a block of input.

- 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, 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 `copy_blocks` is written is that it provides *pushback*, which is to say that if the process can't make progress writing, it will stop reading. If you don't implement pushback in your servers, then anything that prevents you from writing (e.g., a client that is unable to keep up) will cause your program to allocate unbounded amounts of memory, as it keeps track of all the data it intends to write but hasn't been able to yet.

Tail-Calls and Chains of Deferreds

There's another memory problem you might be concerned about, which is the allocation of deferreds. If you think about the execution of `copy_blocks`, you'll see it's creating a chain of deferreds, two per time through the loop. The length of this chain is unbounded, and so, naively, you'd think this would take up an unbounded amount of memory as the echo process continues.

Happily, this is a case that Async knows how to optimize. In particular, the whole chain of 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 does just this, and so there's no memory leak after all.

This is essentially a form of tail-call optimization, lifted to the Deferred monad. Indeed, you can tell that the bind in question doesn't lead to a memory leak in more or less the same way you can tell that the tail recursion optimization should apply, which is that the bind that creates the deferred is in tail-position. In other words, nothing is done to that deferred once it's created; it's simply returned as is.

`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.Where_to_listen.of_port 8765)
      (fun _addr r w ->
        let buffer = Bytes.create (16 * 1024) in
        copy_blocks buffer r w)
  in
  ignore
    (host_and_port
      : (Socket.Address.Inet.t, int) Tcp.Server.t Deferred.t)
```

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 ())
```

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 clients concurrently connecting and reading and writing data.

Now that we have the echo server, we can connect to the echo server using the netcat tool, which is invoked as `nc`. Note that we use `dune exec` to both build and run the executable. We use the double-dashes so that Dune's parsing of arguments doesn't interfere with argument parsing for the executed program.

```
$ dune exec -- ./echo.exe &
$ echo "This is an echo server" | nc 127.0.0.1 8765
This is an echo server
$ echo "It repeats whatever I write" | nc 127.0.0.1 8765
It repeats whatever I write
$ killall echo.exe
```

Functions that Never Return

The call to `never_returns` around the call to `Scheduler.go` is a little bit surprising, but it has a purpose: to make it clear to whoever invokes `Scheduler.go` that the function never returns.

By default, a function that doesn't return will have an inferred return type of `'a`:

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

This is a little odd, but it does make sense. After all, if a function never returns, we're free to impute any type at all to its non-existent return value. As a result, from a typing perspective, a function that never returns can fit into any context within your program.

But that itself can be problematic, especially with a function like `Scheduler.go`, where the fact that it never returns is perhaps not entirely obvious. The point of `never_returns` is to create an explicit marker so the user knows that the function in question doesn't return.

To do this, `Scheduler.go` is defined to have a return value of `Nothing.t`.

```
# #show Scheduler.go;;
val go : ?raise_unhandled_exn:bool -> unit -> never_returns
```

`never_returns` is just an alias of `Nothing.t`.

`Nothing.t` is *uninhabited*, which means there are no values of that type. As such, a function can't actually return a value of type `Nothing.t`, so only a function that never returns can have `Nothing.t` as its return type! And we can cause a function that never returns to have a return value of `Nothing.t` by just adding a type annotation.

```
# let rec loop_forever () : Nothing.t = loop_forever ();;
val loop_forever : unit -> never_returns = <fun>
```

The function `never_returns` consumes a value of type `Nothing.t` and returns an unconstrained type `'a`.

```
# #show_val never_returns;;
val never_returns : never_returns -> 'a
```

If you try to write a function that uses `Scheduler.go`, and just assumes that it returns `unit`, you'll get a helpful type error.

```
# let do_stuff n =
    let x = 3 in
    if n > 0 then Scheduler.go ();
    x + n;;
Line 3, characters 19-34:
Error: This expression has type never_returns
       but an expression was expected of type unit
       because it is in the result of a conditional with no else
    branch
```

We can fix this by inserting a call to `never_returns`, thus making the fact that `Scheduler.go` doesn't return apparent to the reader.

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

17.2.1 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
open Async

let run ~uppercase ~port =
  let host_and_port =
    Tcp.Server.create
      ~on_handler_error:`Raise
      (Tcp.Where_to_listen.of_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
    ~summary:"Start an echo server"
    (let%map_open.Command uppercase =
       flag
         "-uppercase"
         no_arg
         ~doc:" Convert to uppercase before echoing back"
     and port =
       flag
         "-port"
         (optional_with_default 8765 int)
         ~doc:" Port to listen on (default 8765)"
     in
     fun () -> run ~uppercase ~port)
  |> Command.run
```

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 : '_weak4 Pipe.Reader.t = <abstr>
val w : '_weak4 Pipe.Writer.t = <abstr>
```

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 Chapter 9 (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.
```

That's because a pipe has a certain amount of internal slack, a number of slots in the pipe to which something can be written before the write will block. By default, a pipe has zero slack, which means that the deferred returned by a write is determined only when the value is read out of the pipe.

```
# let (r,w) = Pipe.create ();;
val r : '_weak5 Pipe.Reader.t = <abstr>
val w : '_weak5 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 = ()
```

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>
```

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 16 (Command-Line Parsing). Opening Async, shadows the Command module with an extended version that contains the async call:

```
# #show Command.async_spec;;
val async_spec :
  ('a, unit Deferred.t) Async.Command.basic_spec_command
    Command.with_options
```

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`.

17.3 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 the installation instructions[1] 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 Handling Json Data[2].

`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.

17.3.1 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[3] 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
open Async

(* 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 ])
```

[1] http://dev.realworldocaml.org/install.html
[2] json.html#handling-json-data
[3] http://tools.ietf.org/html/rfc3986

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.

17.3.2 Parsing JSON Strings

The HTTP response from DuckDuckGo is in JSON, a common (and thankfully simple) format that is specified in RFC4627[4]. We'll parse the JSON data using the Yojson library, which was introduced in Chapter 19 (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 =
      match List.Assoc.find ~equal:String.equal kv_list key with
      | None | Some (`String "") -> None
      | Some s -> Some (Yojson.Safe.to_string s)
    in
    (match find "Abstract" with
    | Some _ as x -> x
    | None -> find "Definition")
  | _ -> None
```

17.3.3 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 =
  let%bind _, body = Cohttp_async.Client.get (query_uri word) in
  let%map string = Cohttp_async.Body.to_string body in
  word, get_definition_from_json string
```

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";;
# #show Cohttp_async.Client.get;;
val get :
  ?interrupt:unit Deferred.t ->
  ?ssl_config:Conduit_async.V2.Ssl.Config.t ->
  ?headers:Cohttp.Header.t ->
  Uri.t -> (Cohttp.Response.t * Cohttp_async.Body.t) Deferred.t
```

4 http://www.ietf.org/rfc/rfc4627.txt

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 streamed.

In this case, the HTTP body probably isn't very large, so we call `Cohttp_async.Body.to_string` to collect the data from the connection as a single deferred string, rather than consuming the data incrementally.

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))
```

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`. 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 =
  let%map results = Deferred.all (List.map words ~f:get_definition) in
  List.iter results ~f:print_result
```

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>
```

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. It also means that no printing occurs until all results arrive.

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))
```

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>
```

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

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

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

```
$ dune exec -- ./search.exe "Concurrent Programming" "OCaml"
Concurrent Programming
----------------------

"Concurrent computing is a form of computing in which several
computations are executed during overlapping time
periodsconcurrentlyinstead of sequentially. This is a property
of a systemthis may be an individual program, a computer, or a
networkand there is a separate execution point or \"thread of
control\" for each computation. A concurrent system is one where a
computation can advance without waiting for all other computations to
complete."

OCaml
-----

"OCaml, originally named Objective Caml, is the main implementation of
the programming language Caml, created by Xavier Leroy, Jérôme
Vouillon, Damien Doligez, Didier Rémy, Ascánder Suárez and others
in 1996. A member of the ML language family, OCaml extends the core
Caml language with object-oriented programming constructs."
```

17.4 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 Chapter 8.2 (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;
      let%map () = after (Time.Span.of_sec 0.5) in
      if will_fail then raise Exit else ();;
val maybe_raise : unit -> unit Deferred.t = <fun>
# maybe_raise ();;
- : unit = ()
# maybe_raise ();;
Exception: (monitor.ml.Error Exit ("Caught by monitor
    block_on_async"))
```

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` expression, but as you can see that doesn't quite do the trick:

```
# let handle_error () =
    try
      let%map () = maybe_raise () in
      "success"
    with _ -> return "failure";;
val handle_error : unit -> string Deferred.t = <fun>
# handle_error ();;
- : string = "success"
# handle_error ();;
Exception: (monitor.ml.Error Exit ("Caught by monitor
    block_on_async"))
```

This didn't work because `try/with` only captures exceptions that are thrown by the code executed synchronously within it, while `maybe_raise` schedules an Async job that will throw an exception in the future, after the `try/with` expression has exited.

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

Here's a trivial example of `try_with` in action.

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

17.4.1 Monitors

`try_with` is a useful tool for 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: (monitor.ml.Error Exit ("Caught by monitor blow up
    monitor"))
```

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.detach_and_get_error_stream` 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.detach_and_get_error_stream monitor)
      ~f:(fun _exn -> printf "an error happened\n");
    within' ~monitor (fun () ->
      let%bind () = after (Time.Span.of_sec 0.25) in
      failwith "Kaboom!");;
val swallow_error : unit -> 'a Deferred.t = <fun>
```

The deferred returned by this function is never determined, since the computation ends with an exception rather than a return value. That means that if we run this function in utop, we'll never get our prompt back.

We can fix this by using Deferred.any along with a timeout to get a deferred we know will become determined eventually. Deferred.any takes a list of deferreds, and returns a deferred which will become determined assuming any of its arguments becomes determined.

```
# Deferred.any [ after (Time.Span.of_sec 0.5)
               ; swallow_error () ];;
an error happened
- : unit = ()
```

As you can see, the message "an error happened" is printed out before the timeout expires.

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.detach_and_get_error_stream 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 () ->
      let%bind () = after (Time.Span.of_sec 0.25) in
      raise exn_to_raise);;
val swallow_some_errors : exn -> 'a Deferred.t = <fun>
```

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 if you want to depend on the details of the original exception thrown.

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:

```
# exception Another_exception;;
exception Another_exception
# Deferred.any [ after (Time.Span.of_sec 0.5)
               ; swallow_some_errors Another_exception ];;
```

```
Exception:
(monitor.ml.Error (Another_exception) ("Caught by monitor (id 69)")).
```

If instead we use `Ignore_me`, the exception will be ignored, and the computation will finish when the timeout expires.

```
# Deferred.any [ after (Time.Span.of_sec 0.5)
               ; swallow_some_errors Ignore_me ];;
ignoring exn
- : unit = ()
```

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.

17.4.2 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 ])
```

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 when we rebuild the application and run it on two servers, one of which won't respond to the query.

```
$ dune exec -- ./search.exe -servers localhost,api.duckduckgo.com
    "Concurrent Programming" "OCaml"
(monitor.ml.Error (Unix.Unix_error "Connection refused" connect
    127.0.0.1:80)
 ("Raised by primitive operation at file
    \"duniverse/async_unix/src/unix_syscalls.ml\", line 1046,
    characters 17-74"
```

```
  "Called from file \"duniverse/async_kernel/src/deferred1.ml\", line
    17, characters 40-45"
  "Called from file \"duniverse/async_kernel/src/job_queue.ml\", line
    170, characters 6-47"
  "Caught by monitor Tcp.close_sock_on_error"))
[1]
```

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 =
  match%map
    try_with (fun () ->
        let%bind _, body =
          Cohttp_async.Client.get (query_uri ~server word)
        in
        let%map string = Cohttp_async.Body.to_string body in
        word, get_definition_from_json string)
  with
  | Ok (word, result) -> word, Ok result
  | Error _ -> word, Error "Unexpected failure"
```

Here, we first use `try_with` to capture the exception, and then use `match%map` (another syntax provided by `ppx_let`) 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))
```

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

```
$ dune exec -- ./search.exe -servers localhost,api.duckduckgo.com
    "Concurrent Programming" OCaml
Concurrent Programming
----------------------

DuckDuckGo query failed: Unexpected failure

OCaml
-----
```

```
"OCaml, originally named Objective Caml, is the main implementation of
the programming language Caml, created by Xavier Leroy, Jérôme
Vouillon, Damien Doligez, Didier Rémy, Ascánder Suárez and others
in 1996. A member of the ML language family, OCaml extends the core
Caml language with object-oriented programming constructs."
```

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 Chapter 8.2.3 (Cleaning Up in the Presence of Exceptions).

17.5 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
      (let%map () = after (sec 0.5) in "A")
      (let%map () = after (sec 0.25) in 32.33);;
val string_and_float : (string * float) Deferred.t = <abstr>
# string_and_float;;
- : string * float = ("A", 32.33)
```

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
  [ (let%map () = after (sec 0.5) in "half a second")
  ; (let%map () = after (sec 1.0) in "one second")
  ; (let%map () = after (sec 4.0) in "four seconds")
  ];;
- : string = "half a second"
```

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
    [ (let%map () = after timeout in
        word, Error "Timed out")
    ; (match%map get_definition ~server word with
      | word, Error _ -> word, Error "Unexpected failure"
      | word, (Ok _ as x) -> word, x)
    ]
```

We use `let%map` 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 =
  match%map
    try_with (fun () ->
        let%bind _, body =
          Cohttp_async.Client.get ~interrupt (query_uri ~server word)
        in
        let%map string = Cohttp_async.Body.to_string body in
        word, get_definition_from_json string)
  with
  | Ok (word, result) -> word, Ok result
  | Error _ -> word, Error "Unexpected failure"
```

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 =
  match%map
    get_definition ~server ~interrupt:(after timeout) word
  with
  | word, (Ok _ as x) -> word, x
  | word, Error _ -> word, Error "Unexpected failure"
```

This 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>
```

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')
    ]
```

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

```
$ dune exec -- ./search.exe "concurrent programming" ocaml -timeout
    0.1s
concurrent programming
----------------------

"Concurrent computing is a form of computing in which several
computations are executed during overlapping time
periodsconcurrentlyinstead of sequentially. This is a property
of a systemthis may be an individual program, a computer, or a
networkand there is a separate execution point or \"thread of
control\" for each computation. A concurrent system is one where a
computation can advance without waiting for all other computations to
complete."

ocaml
-----

DuckDuckGo query failed: Timed out
```

17.6 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 the advantages of Async's cooperative threading model over 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 23 (Foreign Function Interface).

Multicore OCaml

OCaml doesn't support truly parallel threads today, but it will soon. The current development branch of OCaml, which is expected to be released in 2022 as OCaml 5.0, has a long awaited multicore-capable garbage collector, which is the result of years of research and hard implementation work.

We won't discuss the multicore gc here in part because it's not yet released, and in part because there's a lot of open questions about how OCaml programs should take advantage of multicore in a way that's safe, convenient, and performant. Given all that, we just don't know enough to write a chapter about multicore today.

In any case, while multicore OCaml isn't here yet, it's an exciting part of OCaml's near-term future.

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]
```

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;
    let%bind () = d in
    printf "\nFinished at: ";
    print_time ();
    printf "\n";
    Writer.flushed (force Writer.stdout);;
val log_delays : (unit -> unit Deferred.t) -> unit Deferred.t = <fun>
```

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));;
37.670135498046875us, 100.65722465515137ms, 201.19547843933105ms,
    301.85389518737793ms, 402.58693695068359ms,
Finished at: 500.67615509033203ms,
- : unit = ()
```

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

```
# let busy_loop () =
    let x = ref None in
    for i = 1 to 100_000_000 do x := Some i done;;
val busy_loop : unit -> unit = <fun>
# log_delays (fun () -> return (busy_loop ()));;
Finished at: 874.99594688415527ms,
- : unit = ()
```

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);;
31.709671020507812us, 107.50102996826172ms, 207.65542984008789ms,
    307.95812606811523ms, 458.15873146057129ms,
    608.44659805297852ms, 708.55593681335449ms, 808.81166458129883ms,
Finished at: 840.72136878967285ms,
- : unit = ()
```

Now `log_delays` does get a chance to run, but it’s no longer at clean 100 millisecond intervals. 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);;
32.186508178710938us, 116.56808853149414ms, 216.65477752685547ms,
    316.83063507080078ms, 417.13213920593262ms,
Finished at: 418.69187355041504ms,
- : unit = ()
```

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

```
$ dune exec -- native_code_log_delays.exe
197.41058349609375us,
Finished at: 1.2127914428710938s,
```

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.

17.6.1 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 will behave non-deterministically when accessed concurrently by multiple threads. The issues you can run into range from runtime exceptions to corrupted data structures. That means you should almost 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 behave in surprising ways 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's pretty simple if the following two conditions 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.

That said, you can 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.