Lwt user manual

Jérémie Dimino

January 26, 2011

Contents

1 Introduction

When writing a program, a common developer's task is to handle IO operations. Indeed most software interact with several different resources, such as:

- the kernel, by doing system calls
- the user, by reading the keyboard, the mouse, or any input device
- a graphical server, to build graphical user interface
- other computers, by using the network
- ...

When this list contains only one item, it is pretty easy to handle. However as this list grows it becomes harder and harder to make everything works together. Several choices have been proposed to solve this problem:

- using a main loop, and integrate all components we are interacting with into this main loop.
- using preemptive system threads

Both solution have their advantages and their drawbacks. For the first one, it may works, but it becomes very complicated to write some piece of asynchronous sequential code. The typical example being with graphical user interfaces freezing and not redrawing themselves because they are waiting for some blocking part of the code to complete.

If you already wrote code using preemptive threads, you shall know that doing it right with threads is a hard job. Moreover system threads consume non negligible resources, and so you can only launch a limited number of threads at the same time. Thus this is not a real solution.

`Lwt` offers a new alternative. It provides very light-weight cooperative threads; "launching" a thread is a very quick operation, it does not require a new stack, a new process, or anything else. Moreover context switches are very fast. In fact, it is so easy that we will launch a thread for every system call. And composing cooperative threads will allow us to write highly asynchronous programs.

In a first part, we will explain the concepts of `Lwt`, then we will describe the many sub-libraries of `Lwt`.

2 The Lwt core library

In this section we describe the basics of `Lwt`. It is advised to start an ocaml toplevel and try the given code examples. To start, launch `ocaml` in a terminal or in emacs with the tuareg mode, and type:

```
# #use "topfind";;
# #require "lwt";;
```

`Lwt` is also shipped with an improved toplevel, which supports line edition and completion. If it has been correctly installed, you should be able to start it with the following command:

```
$ lwt-toplevel
```

2.1 Lwt concepts

Let's take a classical function of the `Pervasives` module:

```
# Pervasives.input_char;
- : in_channel -> char = <fun>
```

This function will wait for a character to come on the given input channel, then return it. The problem with this function is that it is blocking: while it is being executed, the whole program will be blocked, and other events will not be handled until it returns.

Now let's look at the lwt equivalent:

```
# Lwt_io.read_char;;
- : Lwt_io.input_channel -> char Lwt.t = <fun>
```

As you can see, it does not returns a character but something of type `char Lwt.t`. The type `'a Lwt.t` is the type of threads returning a value of type `'a`. Actually the `Lwt_io.read_char` will try to read a character from the given input channel and *immediatly* returns a light-weight thread.

Now, let's see what we can do with a `Lwt` thread. The following code create a pipe, and launch a thread reading on the input side:

```
# let ic, oc = Lwt_io.pipe ();;
val ic : Lwt_io.input_channel = <abstr>
val oc : Lwt_io.output_channel = <abstr>
# let t = Lwt_io.read_char ic;;
val t : char Lwt.t = <abstr>
```

We can now look at the state of our newly created thread:

```
# Lwt.state t;;
- : char Lwt.state = Sleep
```

A thread may be in one of the following states:

- `Return x`, which means that the thread has terminated successfully and returned the value `x`
- `Fail exn`, which means that the thread has terminated, but instead of returning a value, it failed with the exception `exn`
- `Sleep`, which means that the thread is currently sleeping and have not yet returned a value or an exception

The thread `t` is sleeping because there is currently nothing to read on the pipe. Let's write something:

```
# Lwt_io.write_char oc 'a';;
- : unit Lwt.t = <abstr>
# Lwt.state t;;
- : char Lwt.state = Return 'a'
```

So, after we write something, the reading thread has been wakeup and has returned the value `'a'`.

2.1.1 Primitives for thread creation

There are several primitives for creating `lwt` threads. These functions are located in the module `Lwt`.

Here are the main primitives:

- `Lwt.return : 'a -> 'a Lwt.t`

  creates a thread which has already terminated and returned a value

- `Lwt.fail : exn -> 'a Lwt.t`

  creates a thread which has already terminated and failed with an exception

- `Lwt.wait : unit -> 'a Lwt.t * 'a Lwt.u`

  creates a sleeping thread and returns this thread plus a wakener (of type `'a Lwt.u`) which must be used to wakeup the sleeping thread.

To wake up a sleeping thread, you must use one of the following functions:

- `Lwt.wakeup : 'a Lwt.u -> 'a -> unit`
  wakes up the thread with a value.
- `Lwt.wakeup_exn : 'a Lwt.u -> exn -> unit`
  wakes up the thread with an exception.

Note that this is an error to wakeup two times the same threads. `Lwt` will raise `Invalid_argument` if you try to do so.

With these informations, try to guess the result of each of the following expression:

```
# Lwt.state (Lwt.return 42);;
# Lwt.state (fail Exit);;
# let waiter, wakener = Lwt.wait ();;
# Lwt.state waiter;;
# Lwt.wakeup wakener 42;;
# Lwt.state waiter;;
# let waiter, wakener = Lwt.wait ();;
# Lwt.state waiter;;
# Lwt.wakeup_exn wakener Exit;;
# Lwt.state waiter;;
```

2.1.2 Primitives for thread composition

The most important operation you need to know is `bind`:

```
val bind : 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t
```

`bind t f` creates a thread which waits for `t` to terminates, then pass the result to `f`. If `t` is a sleeping thread, then `bind t f` will be a sleeping thread too, until `t` terminates. If `t` fails, then the resulting thread will fail with the same exception. For example, consider the following expression:

```
Lwt.bind
  (Lwt_io.read_line Lwt_io.stdin)
  (fun str -> Lwt_io.printlf "You typed %S" str)
```

This code will first wait for the user to enter a line of text, then print a message on the standard output.

Similarly to `bind`, there is a function to handle the case when `t` fails:

```
val catch : (unit -> 'a Lwt.t) -> (exn -> 'a Lwt.t) -> 'a Lwt.t
```

`catch f g` will call `f ()`, then waits for its termination, and if it fails with an exception `exn`, calls `g exn` to handle it. Note that both exceptions raised with `Pervasives.raise` and `Lwt.fail` are caught by `catch`.

2.1.3 Cancelable threads

In some case, we may want to cancel a thread. For example, because it has not terminated after a timeout. This can be done with cancelable threads. To create a cancelable thread, you must use the `Lwt.task` function:

```
val task : unit -> 'a Lwt.t * 'a Lwt.u
```

It has the same semantic as `Lwt.wait` except that the sleeping thread can be canceled with `Lwt.cancel`:

```
val cancel : 'a Lwt.t -> unit
```

The thread will then fails with the exception `Lwt.Canceled`. To execute a function when the thread is canceled, you must use `Lwt.on_cancel`:

```
val on_cancel : 'a Lwt.t -> (unit -> unit) -> unit
```

Note that it is also possible to cancel a thread which has not been created with `Lwt.task`. In this case, the deepest cancelable thread connected with the given thread will be cancelled.

For example, consider the following code:

```
# let waiter, wakener = Lwt.task ();;
val waiter : '_a Lwt.t = <abstr>
val wakener : '_a Lwt.u = <abstr>
# let t = bind waiter (fun x -> return (x + 1));;
val t : int Lwt.t = <abstr>
```

Here, cancelling `t` will in fact cancel `waiter`. `t` will then fails with the exception `Lwt.Canceled`:

```
# Lwt.cancel t;;
- : unit = ()
# Lwt.state waiter;;
- : int Lwt.state = Fail Lwt.Canceled
# Lwt.state t;;
- : int Lwt.state = Fail Lwt.Canceled
```

By the way, it is possible to prevent a thread from being canceled by using one of the function `Lwt.protected`:

```
val protected : 'a Lwt.t -> 'a Lwt.t
```

Canceling `(proctected t)` will have no effect on `t`.

2.1.4 Primitives for multi-thread composition

We now show how to compose several threads at the same time. The main functions for this are in the `Lwt` module: `join`, `choose` and `pick`.

The first one, `join` takes a list of threads and wait for all of them to terminate:

```
val join : unit Lwt.t list -> unit Lwt.t
```

Moreover, if at least one thread fails, `join l` will fails with the same exception as the first to fail, after all threads threads terminated.

On the contrary `choose` waits for at least one thread to terminate, then returns the same value or exception:

```
val choose : 'a Lwt.t list -> 'a Lwt.t
```

For example:

```
# let waiter1, wakener1 = Lwt.wait ();;
val waiter1 : '_a Lwt.t = <abstr>
val wakener1 : '_a Lwt.u = <abstr>
# let waiter2, wakener2 = Lwt.wait ();;
val waiter2 : '_a Lwt.t = <abstr>
val wakener : '_a Lwt.u = <abstr>
```

```
# let t = Lwt.choose [waiter1; waiter2];;
val t : '_a Lwt.t = <abstr>
# Lwt.state t;;
- : '_a Lwt.state = Sleep
# Lwt.wakeup wakener2 42;;
- : unit = ()
# Lwt.state t;;
- : int Lwt.state = Return 42
```

Thel last one, `pick`, is the same as `join` except that it cancels all other threads when one terminates.

2.1.5 Threads local storage

Lwt can stores variables with different values on different threads. This is called threads local storage. For example, this can be used to store contexts or thread identifiers. The contents of a variable can be read with:

```
val Lwt.get : 'a Lwt.key -> 'a option
```

which takes a key to identify the variable we want to read and returns either `None` if the variable is not set, or `Some x` if it is. The value returned is the value of the variable in the current thread.

New keys can be created with:

```
val Lwt.new_key : unit -> 'a Lwt.key
```

To set a variable, you must use:

```
val Lwt.with_value : 'a Lwt.key -> 'a option -> (unit -> 'b) -> 'b
```

`with_value key value f` will executes `f` with the binding `key -> value`. The old value associated to `key` is restored after `f` terminates.

For example, you can use local storage to store thread identifiers and use them in logs:

```
let id_key = Lwt.new_key ()

let log msg =
  let thread_id =
    match Lwt.get id_key with
      | Some id -> id
      | None -> "main"
  in
  Lwt_io.printlf "%s: %s" thread_id msg

lwt () =
  Lwt.join [
    Lwt.with_value id_key (Some "thread 1") (fun () -> log "foo");
    Lwt.with_value id_key (Some "thread 2") (fun () -> log "bar");
  ]
```

2.1.6 Rules

`Lwt` will always try to execute the more it can before yielding and switching to another cooperative thread. In order to make it works well, you must follow the following rules:

- do not write function that may takes time to complete without using `Lwt`,

- do not do IOs that may block, otherwise the whole program will hang. You must instead use asynchronous IOs operations.

2.2 The syntax extension

`Lwt` offers a syntax extension which increases code readability and makes coding using `Lwt` easier. To use it add the "lwt.syntax" package when compiling:

```
$ ocamlfind ocamlc -syntax camlp4o -package lwt.syntax -linkpkg -o foo foo.ml
```

Or in the toplevel (after loading topfind):

```
# #camlp4o;;
# #require "lwt.syntax";;
```

The following construction are added to the language:

- `lwt` *pattern*$_1$ `=` *expr*$_1$ [`and` *pattern*$_2$ `=` *expr*$_2$ ...] `in` *expr*

  which is a parallel let-binding construction. For example in the following code:

  ```
  lwt x = f () and y = g () in
  expr
  ```

  the thread `f ()` and `g ()` are launched in parallel and their result are then bound to `x` and `y` in the expression *expr*.

  Of course you can also launch the two threads sequentially by writing your code like that:

  ```
  lwt x = f () in
  lwt y = g () in
  expr
  ```

- `try_lwt` *expr* [`with` *pattern*$_1$ $\rightarrow$ *expr*$_1$ ...] [`finally` *expr*$'$]

  which is the equivalent of the standard `try-with` construction but for `Lwt`. Both exception raised by `Pervasives.raise` and `Lwt.fail` are caught.

- `for_lwt` *ident* `=` *expr*$_{init}$ (`to` | `downto`) *expr*$_{final}$ `do` *expr* `done`

  which is the equivalent of the standard `for` construction but for `Lwt`.

- `raise_lwt` *exn*

  which is the same as `Lwt.fail` *exn* but with backtrace support.

2.2.1 Correspondence table

You can keep in mind the following table to write code using lwt:

without `Lwt`	**with** `Lwt`
`let` $pattern_1$ `=` $expr_1$ `and` $pattern_2$ `=` $expr_2$ ... `and` $pattern_n$ `=` $expr_n$ `in` $expr$	`lwt` $pattern_1$ `=` $expr_1$ `and` $pattern_2$ `=` $expr_2$ ... `and` $pattern_n$ `=` $expr_n$ `in` $expr$

`try`
  $expr$
`with`
  | $pattern_1$ → $expr_1$
  | $pattern_2$ → $expr_2$
  ...
  | $pattern_n$ → $expr_n$
	`try_lwt`
	  $expr$
	`with`
	  | $pattern_1$ → $expr_1$
	  | $pattern_2$ → $expr_2$
	  ...
	  | $pattern_n$ → $expr_n$

`for` $ident$ `=` $expr_{\text{init}}$ `to` $expr_{\text{final}}$ `do`
  $expr$
`done`
	`for_lwt` $ident$ `=` $expr_{\text{init}}$ `to` $expr_{\text{final}}$ `do`
	  $expr$
	`done`

`for` $ident$ `=` $expr_{\text{init}}$ `downto` $expr_{\text{final}}$ `do`
  $expr$
`done`
	`for_lwt` $ident$ `=` $expr_{\text{init}}$ `downto` $expr_{\text{final}}$ `do`
	  $expr$
	`done`

`raise` exn	`raise_lwt` exn

2.3 Backtrace support

When using `Lwt`, exceptions are not recorded by the ocaml runtime, and so you can not get backtraces. However it is possible to get them when using the syntax extension. All you have to do is to pass the `-lwt-debug` switch to `camlp4`:

```
$ ocamlfind ocamlc -syntax camlp4o -package lwt.syntax \
    -ppopt -lwt-debug -linkpkg -o foo foo.ml
```

2.4 Other modules of the core library

The core library contains several modules that depend only on `Lwt`. The following naming convention is used in `Lwt`: when a function takes as argument a function returning a thread that is going to be executed sequentially, it is suffixed with "`_s`". And when it is going to be executed in parallel, it is suffixed with "`_p`". For example, in the `Lwt_list` module we have:

```
val map_s : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t
val map_p : ('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t
```

2.4.1 Mutexes

`Lwt_mutex` provides mutexes for `Lwt`. Its use is almost the same as the `Mutex` module of the thread library shipped with OCaml. In general, programs using `Lwt` do not need a lot of mutexes. They are only usefull for serialising operations.

2.4.2 Lists

The `Lwt_list` module defines iteration and scanning functions over lists, similar to the ones of the `List` module, but using functions that return a thread. For example:

```
val iter_s : ('a -> unit Lwt.t) -> 'a list -> unit Lwt.t
val iter_p : ('a -> unit Lwt.t) -> 'a list -> unit Lwt.t
```

In `iter_s f l`, `iter_s` will call f on each elements of `l`, waiting for completion between each elements. On the contrary, in `iter_p f l`, `iter_p` will call f on all elements of `l`, then wait for all the threads to terminate.

2.4.3 Data streams

`Lwt` streams are used in a lot of places in `Lwt` and its sub libraries. They offer a high-level interface to manipulate data flows.

A stream is an object which returns elements sequentially and lazily. Lazily means that the source of the stream is guessed for new elements only when needed. This module contains a lot of stream transformation, iteration, and scanning functions.

The common way of creating a stream is by using `Lwt_stream.from` or by using `Lwt_stream.create`:

```
val from : (unit -> 'a option Lwt.t) -> 'a Lwt_stream.t
val create : unit -> 'a Lwt_stream.t * ('a option -> unit)
```

As for streams of the standard library, `from` takes as argument a function which is used to create new elements.

`create` returns a function used to push new elements into the stream and the stream which will receive them.

For example:

```
# let stream, push = Lwt_stream.create ();;
val stream : '_a Lwt_stream.t = <abstr>
val push : ''a option -> unit = <fun>
# push (Some 1);;
- : unit = ()
# push (Some 2);;
- : unit = ()
# push (Some 3);;
- : unit = ()
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Return 1
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Return 2
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Return 3
# Lwt.state (Lwt_stream.next stream);;
- : int Lwt.state = Sleep
```

Note that streams are consumable. Once you take an element from a stream, it is removed from it. So, if you want to iterates two times over a stream, you may consider "clonning" it, with `Lwt_stream.clone`. Cloned stream will returns the same elements in the same order. Consuming one will not consume the other. For example:

```
# let s = Lwt_stream.of_list [1; 2];;
val s : int Lwt_stream.t = <abstr>
# let s' = Lwt_stream.clone s;;
val s' : int Lwt_stream.t = <abstr>
# Lwt.state (Lwt_stream.next s);;
- : int Lwt.state = Return 1
# Lwt.state (Lwt_stream.next s);;
- : int Lwt.state = Return 2
# Lwt.state (Lwt_stream.next s');;
- : int Lwt.state = Return 1
# Lwt.state (Lwt_stream.next s');;
```

```
- : int Lwt.state = Return 2
```

2.4.4 Mailbox variables

The `Lwt_mvar` module provides mailbox variables. A mailbox variable, also called a "mvar", is a cell which may contains 0 or 1 element. If it contains no elements, we say that the mvar is empty, if it contains one, we say that it is full. Adding an element to a full mvar will block until one is taken. Taking an element from an empty mvar will block until one is added.

Mailbox variables are commonly used to pass messages between threads.

Note that a mailbox variable can be seen as a pushable stream with a limited memory.

3 The lwt.unix library

The package `lwt.unix` contains all `unix` dependant modules of `Lwt`. Among all its features, it implements cooperative versions of functions of the standard library and the unix library.

3.1 Unix primitives

The `Lwt_unix` provides cooperative system calls. For example, the `Lwt` counterpart of `Unix.read` is:

```
val read : file_descr -> string -> int -> int -> int Lwt.t
```

`Lwt_io` provides features similar to buffered channels of the standard library (of type `in_channel` or `out_channel`) but cooperatively.

`Lwt_gc` allow you to register finaliser that return a thread. At the end of the program, `Lwt` will wait for all the finaliser to terminates.

3.2 The lwt scheduler

The `Lwt_main` contains the `Lwt` *main loop*. It can be customized by adding filters, and/or by replacing the `select` function.

Filters are responsible to collect sources to monitor before entering the blocking `select`, then to react and wakeup threads waiting for sources to become ready.

3.3 The logging facility

The package `lwt.unix` contains a module `Lwt_log` providing loggers. It support logging to a file, a channel, or to the syslog daemon. You can also defines your own logger by providing the appropriate functions (function `Lwt_log.make`).

Several loggers can be merged into one. Sending logs on the merged logger will send these logs to all its components.

For example to redirect all logs to `stderr` and to the syslog daemon:

```
# Lwt_log.default_logger :=
    Lwt_log.broadcast [
      Lwt_log.channel ~close_mode:`Keep ~channel:Lwt_io.stderr ();
      Lwt_log.syslog ~facility:`User ();
    ]
;;
```

`Lwt` also provides a syntax extension, in the package `lwt.syntax.log`. It does not modify the language but it replaces log statement of the form:

```
Lwt_log.info_f ~section "something happened: %s" msg
```

by:

```
if Lwt_log.Section.level section <= Lwt_log.Info then
  Lwt_log.info_f ~section "somethign happend: %s" msg
else
  Lwt.return ()
```

The advantages of using the syntax extension are the following:

- it check the log level before calling the logging function, so arguments are not computed if not needed
- debugging logs can be removed at parsing time

By default, the syntax extension remove all logs with the level `debug`. To keep them pass the command line option `-lwt-debug` to camlp4.

4 The Lwt.react library

The `Lwt_event` and `Lwt_signal` modules provide helpers for using the `react` library with `Lwt`. Among them we have `Lwt_event.next`, which takes an event and returns a thread which will wait until the next occurence of this event. For example:

```
# let event, push = React.E.create ();;
val event : '_a React.event = <abstr>
val push : '_a -> unit = <fun>
# let t = Lwt_event.next event;;
val t : '_a Lwt.t = <abstr>
# Lwt.state t;;
- : '_a Lwt.state = Sleep
# push 42;;
- : unit = ()
# Lwt.state t;;
- : int Lwt.state = Return 42
```

Another interesting feature is the ability to limit events (resp. signals) to occurs (resp. to changes) too often. For example, suppose you are doing a program which displays something on the screeen each time a signal changes. If at some point the signal changes 1000 times per second, you probably want not to render it 1000 times per second. For that you use `Lwt_signal.limit`:

```
val limit : (unit -> unit Lwt.t) -> 'a React.signal -> 'a React.signal
```

`Lwt_signal.limit f signal` returns a signal which varies as `signal` except that two consecutive updates are separeted by a call to `f`. For example if `f` returns a thread which sleep for 0.1 seconds, then there will be no more than 10 changes per second. For example:

```
let draw x =
  (* Draw the screen *)
  ...

let () =
  (* The signal we are interested in: *)
  let signal = ... in

  (* The limited signal: *)
  let signal' = Lwt_signal.limit (fun () -> Lwt_unix.sleep 0.1) signal in
```

```
  (* Redraw the screen each time the limited signal change: *)
  Lwt_signal.always_notify_p draw signal'
```

5 The lwt.text library

The `lwt.text` library provides functions to deal with text mode (in a terminal). It is composed of the three following modules:

- `Lwt_text`, which is the equivalent of `Lwt_io` but for unicode text channels
- `Lwt_term`, providing various terminal utilities, such as reading a key from the terminal
- `Lwt_read_line`, which provides functions to input text from the user with line editing support

5.1 Text channels

A text channel is basically a byte channel plus an encoding. Input (resp. output) text channels decode (resp. encode) unicode characters on the fly. By default, output text channels use transliteration, so they will not fails because text you want to print cannot be encoded in the system encoding.

For example, with you locale sets to "C", and the variable `name` set to "Jérémie", you got:

```
# lwt () = Lwt_text.printlf "My name is %s" name;;
My name is J?r?mie
```

5.2 Terminal utilities

The `Lwt_term` allow you to put the terminal in *raw mode*, meanings that input is not buffered and character are returned as the user type them. For example, you can read a key with:

```
# lwt key = Lwt_term.read_key ();;
val key : Lwt_term.key = Lwt_term.Key_control 'j'
```

The second main feature of `Lwt_term` is the ability to prints text with styles. For example, to print text in bold and blue:

```
# open Lwt_term;;
# lwt () = printlc [fg blue; bold; text "foo"];;
foo
```

If the output is not a terminal, then `printlc` will drop styles, and act as `Lwt_text.printl`.

5.3 Read-line

`Lwt_read_line` provides a full featured and fully customisable read-line implementation. You can either use the high-level and easy to use `read_*` functions, or use the advanced `Lwt_read_line.Control.read_*` functions.

For example:

```
# open Lwt_term;;
# lwt l = Lwt_read_line.read_line ~prompt:[text "foo> "] ();;
foo> Hello, world!
val l : Text.t = "Hello, world!"
```

The second class of functions is a bit more complicated to use, but allow to control a running read-line instances. For example you can temporary hide it to draw something, you can send it commands, fake input, and the prompt is a signal so it can change dynamically.

6 Other libraries

6.1 Detaching computation to preemptive threads

It may happen that you want to run a function which will take time to compute or that you want to use a blocking function that cannot be used in a non-blocking way. For these situations, `Lwt` allow you to *detach* the computation to a preemptive thread.

This is done by the module `Lwt_preemptive` of the `lwt.preemptive` package which maintains a spool of system threads. The main function is:

```
val detach : ('a -> 'b) -> 'a -> 'b Lwt.t
```

`detach f x` will execute `f x` in another thread and asynchronously wait for the result.

The `lwt.extra` package provides wrappers for a few blocking functions of the standard C library like `gethostbyname` (in the module `Lwt_lib`).

6.2 SSL support

The package `lwt.ssl` provides the module `Lwt_ssl` which allow to use SSL asynchronously

6.3 Glib integration

The `lwt.glib` embed the `glib` main loop into the `Lwt` one. This allow you to write GTK application using `Lwt`. The one thing you have to do is to call `Lwt_glib.install` at the beginning of you program.

7 Writing stubs using `Lwt`

`Lwt` internally uses `libev`. Sometimes C library need to watch file-descriptors and/or to use timeouts. It is possible to use them with `Lwt`; all you need to do is to integrate the library with `libev`. You can find all informations for that in the `libev` manual.

However, if you are calling OCaml code in libev callbacks, you must add the macro `LWT_UNIX_CHECK` before calling the OCaml code. This macro is defined in the header file `lwt_unix.h` installed by `Lwt`. This macro takes care of acquiring back the runtime system mutex, which is released before entering the libev main loop.

7.1 Thread-safe notifications

If you want to notify the main thread from another thread, you can use the `Lwt` thread safe notification system. First you need to create a notification identifier (which is just an integer) from the OCaml side using the `Lwt_unix.make_notification` function, then you can send it from either the OCaml code with `Lwt_unix.send_notification` function, or from the C code using the function `lwt_unix_send_notification` (defined in `lwt_unix_.h`).

Notification are received and processed asynchronously by the main thread.

7.2 Jobs

For operations that can not be executed asynchronously, `Lwt` uses a system of jobs that can be executed in a different threads. A job is composed of four functions:

- A function to create the job, which creates a job structure info and stores parameters in it. This function is executed in the main thread.
- A function which execute the job. This one may be executed asynchronously in another thread.
- A function which read the result of the job. This function is executed in the main thread.
- And finally a function that free resources allocated for the job, which is also executed in the main thread.

We show as example the implementation of `Lwt_unix.mkdir`. On the C side we have:

```
/* The job info structure */
struct job_mkdir {
  /* Informations required by lwt.
     It must be the first field of the structure. */
  struct lwt_unix_job job;

  /* The name of the directory to create. */
  char *name;

  /* Permissions for the directory. */
  int perms;

  /* The result of the call to mkdir. */
  int result;

  /* The errno value after the call. */
  int error_code;
};

/* Convenient macro for retrieving a mkdir job info structure from an
   ocaml custom value. */
#define Job_mkdir_val(v) *(struct job_mkdir**)Data_custom_val(v)

/* The function that effectively executes the job. */
static void worker_mkdir(struct job_mkdir *job)
{
  /* Call mkdir and save its result. */
  job->result = mkdir(job->name, job->perms);

  /* Save the contents of [errno]. */
  job->error_code = errno;
}

/* The stub that create the job. */
CAMLprim value lwt_unix_mkdir_job(value val_name, value val_perms)
{
  struct job_mkdir *job = lwt_unix_new(struct job_mkdir);

  /* Sets the worker for this job. */
  job->job.worker = (lwt_unix_job_worker)worker_mkdir;

  /* Copy the name of the directory into the C memory. */
  job->name = lwt_unix_strdup(String_val(val_name));

  /* Copy the perms parameter. */
  job->perms = Int_val(val_perms);

  /* Put the job into an ocaml custom value and returns it. */
  return lwt_unix_alloc_job(&(job->job));
}

/* The stub that read the result of the job. */
CAMLprim value lwt_unix_mkdir_result(value val_job)
{
```

```
  struct job_mkdir *job = Job_mkdir_val(val_job);

  /* If mkdir failed, raise the unix error now. */
  if (job->result < 0) unix_error(job->error_code, "mkdir", Nothing);

  return Val_unit;
}

/* The stub that free resources. */
CAMLprim value lwt_unix_mkdir_free(value val_job)
{
  struct job_mkdir *job = Job_mkdir_val(val_job);

  /* Free the name of the directory. */
  free(job->name);

  /* Free resources allocated by lwt_unix for this job. */
  lwt_unix_free_job(&job->job);

  return Val_unit;
}
```

and on the ocaml side:

```
(* The stub for creating the job. *)
external mkdir_job : string -> int -> [ `unix_mkdir ] job = "lwt_unix_mkdir_job"

(* The stub for reading the result of the job. *)
external mkdir_result : [ `unix_mkdir ] job -> unit = "lwt_unix_mkdir_result"

(* The stub reading the result of the job. *)
external mkdir_free : [ `unix_mkdir ] job -> unit = "lwt_unix_mkdir_free"

(* And finally the ocaml function. *)
let mkdir name perms =
  Lwt_unix.execute_job (mkdir_job name perms) mkdir_result mkdir_free
```