Lwt: a Cooperative Thread Library

Jérôme Vouillon

Universite Paris Diderot - Paris 7, CNRS

Introduction

Library Lwt

- Cooperative threads
- Entirely written in Ocaml
- Monadic

Motivation : race conditions

Less opportunities for race conditions

- Explicit context switches
- Large portions of code executed atomically
- Modules `Hashtbl`, `Str` are thread-safe

Motivation : performance

Lightweight threads

Can use thousands of them

Context switches are cheap

Just a few function calls

Introduction to the library

Promises

Synchronous functions

```
input_char : in_channel -> char
Unix.sleep : int -> unit
```

Asynchronous functions

```
Lwt_chan.input_char : in_channel -> char t
Lwt_unix.sleep : int -> unit t
```

Promise: proxy for an eventual value

Combining promises

Asynchronous wait

Promise `p` : `'a t`
Function `f` : `'a -> 'b t`
$\Rightarrow$ result `v` : `'b t`

Combining promises

Asynchronous wait

Promise `p` : `'a t`
Function `f` : `'a -> 'b t`
$\Rightarrow$ result `v` : `'b t`

```
bind : 'a t -> ('a -> 'b t) -> 'b t
```

Combining promises

Asynchronous wait

Promise `p` : `'a t`
Function `f` : `'a -> 'b t`
$\Rightarrow$ result `v` : `'b t`

```
bind : 'a t -> ('a -> 'b t) -> 'b t
```

Trivial promise

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

Combining promises

Asynchronous wait

Promise `p` : `'a t`
Function `f` : `'a -> 'b t`
$\Rightarrow$ result `v` : `'b t`

```
bind : 'a t -> ('a -> 'b t) -> 'b t
```

Trivial promise

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

$\Rightarrow$ *this is a monad*

A first example

Sequential code

```
let forward in_fd out_fd buffer =
  let n = Unix.read in_fd buffer 0 1024 in
  Unix.sleep 3;
  let n' = Unix.write out_fd buffer 0 n in
  ()
```

Asynchronous code

```
let forward in_fd out_fd buffer =
  Lwt.bind (Lwt_unix.read in_fd buffer 0 1024) (fun n ->
  Lwt.bind (Lwt_unix.sleep 3) (fun () ->
  Lwt.bind (Lwt_unix.write out_fd buffer 0 n) (fun n' ->
  Lwt.return ())))
```

Improved syntax

Asynchronous code

```
let forward in_fd out_fd buffer =
  Lwt.bind (Lwt_unix.read in_fd buffer 0 1024) (fun n ->
  Lwt.bind (Lwt_unix.sleep 3) (fun () ->
  Lwt.bind (Lwt_unix.write out_fd buffer 0 n) (fun n' ->
  Lwt.return ())))
```

Asynchronous code, *improved syntax*

```
let forward in_fd out_fd buffer =
  Lwt_unix.read in_fd buffer 0 1024 >>= fun n ->
  Lwt_unix.sleep 3 >>= fun () ->
  Lwt_unix.write out_fd buffer 0 n >>= fun n' ->
  Lwt.return ()
```

Core of the library

The Lwt module

Core module Lwt

```
type 'a t

val return : 'a -> 'a t
val bind : 'a t -> ('a -> 'b t) -> 'b t

val fail : exn -> 'a t
val catch : (unit -> 'a t) -> (exn -> 'a t) -> 'a t

val wait : unit -> 'a t
val wakeup : 'a t -> 'a -> unit
val wakeup_exn : 'a t -> exn -> unit
val poll : 'a t -> 'a option
```

Dealing with exceptions

Raising an exception

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

Catching exceptions

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

Regular exceptions

Regular exception are also caught

```
Lwt.catch
  (fun () -> raise Exit)
  (fun e -> Lwt.return ())
```

... even when they do not happen immediately

```
Lwt.catch
  (fun () ->
     Lwt_unix.sleep 3 >>= fun () ->
     raise Exit)
  (fun e -> Lwt.return ())
```

Low-level primitives

A promise with no value yet

```
wait : unit -> 'a t
```

Assigning a value to a promise

```
wakeup : 'a t -> 'a -> unit
wakeup_exn : 'a t -> 'a -> unit
```

Current state of a promise

```
poll : 'a t -> 'a option
```

Structure of the library

`Lwt`	Core library
`Lwt_unix`	Unix system calls, scheduler
`Lwt_chan`	Buffered I/O
`Lwt_mutex`	Mutual exclusion locks
`Lwt_timeout`	Manage timers (for instance, for closing idle network connections)
`Lwt_preemptive`	Execute functions on preemptive threads

Using the library

A simple scheduler (1/2)

Queue of suspended threads

```
let queue = Queue.create ()
```

Yield function

```
let yield () =
  let res = Lwt.wait () in
  Queue.push res queue;
  res
```

A simple scheduler (2/2)

Scheduler

```
let rec run () =
  match
    try Some (Queue.take queue)
    with Queue.Empty -> None
  with
    None   -> ()
  | Some t -> Lwt.wakeup t (); run ()
```

Using the scheduler

```
let rec loop message n =
  if n > 0 then begin
    yield () >>= fun () ->
    Format.printf "%s@." message;
    loop message (n - 1)
  end else
    Lwt.return ()

let ta = loop "a" 6 in
let tb = loop "b" 5 in
run ()
```

Implementation

Promises

The type of promises (simplified)

```
type 'a t =
  { mutable state : 'a state }
and 'a state =
    Return of 'a
  | Sleep of (unit -> unit) list
```

Creating promises

```
let return v = { state = Return v }
let wait ()  = { state = Sleep [] }
```

Thread termination

Fulfilling a promise

```
let wakeup p v =
  match p.state with
    Sleep waiters ->
      p.state <- Return v;
      List.iter (fun f -> f ()) waiters
  | Return _ ->
      invalid_arg "wakeup"
```

Synchronization (1/2)

```
let rec bind p f =
  match p.state with
    Return v ->
      f v
  | Sleep waiters ->
      let result = wait () in
      let restart () = connect result (bind p f) in
      p.state <- Sleep (restart :: waiters);
      result
```

Synchronization (2/2)

```
let rec connect p p' =
  match p'.state with
    Return v ->
      wakeup p v
  | Sleep waiters' ->
      let restart () = connect p p' in
      p'.state <- Sleep (restart :: waiters')
```

Actual implementation : exceptions

Exceptions

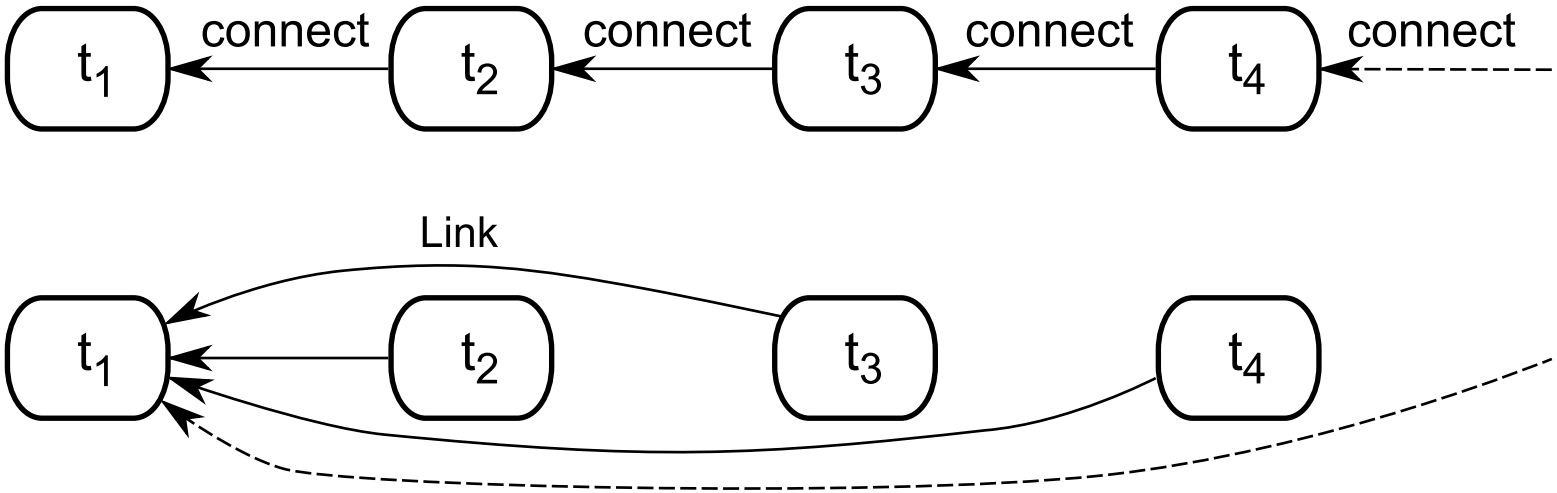
```
type 'a state =
    ...
  | Fail of exn
```

Actual implementation : memory-leaks

Avoiding memory leaks

Union-find datastructure

```
type 'a state =
    ...
  | Link of 'a t
```



Performances

Performances

From the Computer Language Shoutout Benchmarks

	Haskell	Lwt	System threads
thread-ring	5.5	5.5	34.0
chameneos-redux	4.3	14.9	430

Mostly measure context-switches costs...

Related works

CPS-based threads

- *A poor man's concurrency monad*, Claessen, 1999
- *Combining Events and Threads for Scalable Network Services*, Li and Zdancewic, 2007
- *An ocaml-based network services platform*, Waterson, 2007
- F#'s asynchronous workflows, Syme, 2007

Based on a monad of suspended computations

Applications

Successfully used in real-world applications

- Unison file synchronizer (since 2002)
- Ocsigen Web server

Download

Available at: `http://www.ocsigen.org/lwt`