FPV Tutorübung

Woche 13

Threading

Manuel Lerchner

19.07.2023

Threads

```
(* Threads: *)

type t

Thread.create : ('a → 'b) → 'a → t
Thread.join : t → unit
Thread.self : unit → t
Thread.id : t → int


(* Channel: *)

Event.new_channel : unit → 'a channel

sync(send <chan> <val>) : unit
sync(recieve <chan>) : 'a
```

Threads in utop:

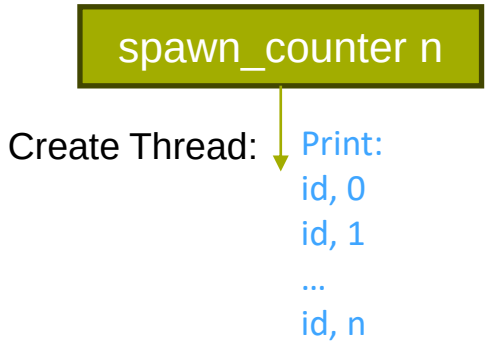
- utop -I +threads
- #use “src/file.ml“

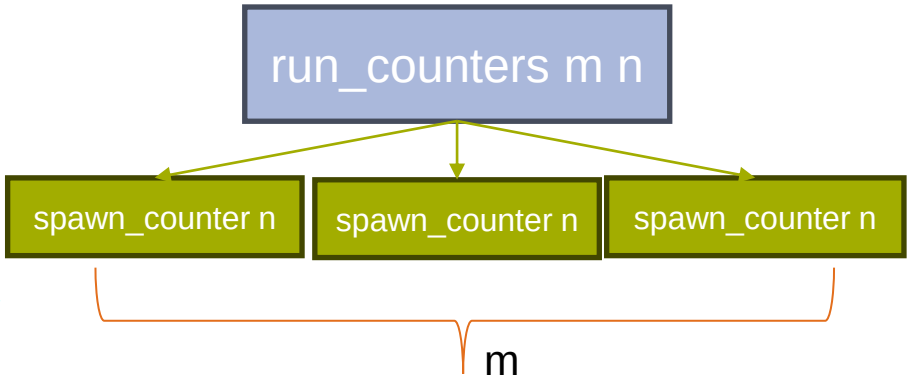
T01: Hellish Counting

1. **spawn_counter**
   As a first step, implement a function `spawn_counter : int -> Thread.t` that spawns a new thread. This thread should then print all numbers from 0 to the passed argument to the standard output. Print the thread's id in addition to the current number, so that you can identify who is responsible for the output.
2. **run_counters**
   Write a function `run_counters : int -> int -> unit` that, when called with `run_counters m n`, spawns `m` counters, each counting to `n`. Make sure `run_counters` does not return before all the counters have stopped.
3. **What happens?**
   Discuss the output you expect for calls of `run_counters m n` with different values of `m` and `n`. Then, check it out!

As a next step, the threads shall now be synchronized, such that all threads take turns with their output. First all threads print 0, then all threads print 1 and so on. Use channels for communication between the threads. Make sure they shutdown correctly and are joined by the main thread.

4. **Threads taking turns**
   Implement a new version of `spawn_counter` and `run_counters` such that the counters take turns counting.

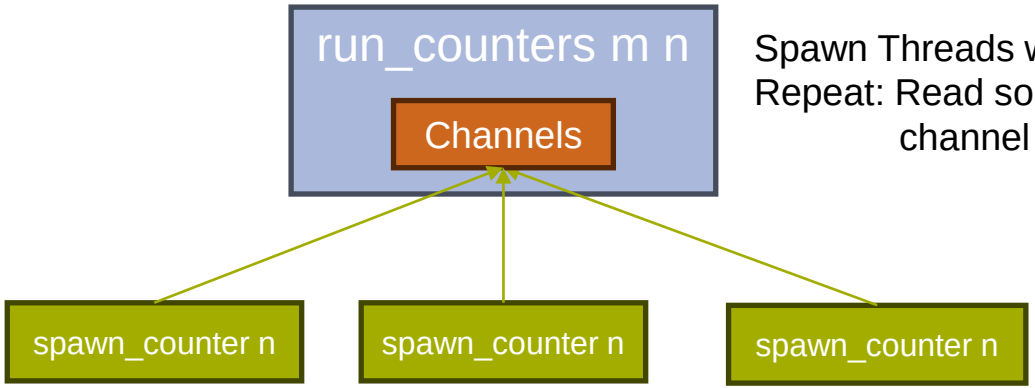




T01: Hellish Counting

As a next step, the threads shall now be synchronized, such that all threads take turns with their output. First all threads print 0, then all threads print 1 and so on. Use channels for communication between the threads. Make sure they shutdown correctly and are joined by the main thread.

1. Threads write in their channel after they printed somehting and start to **wait**!
2. Main Thread repeatetly reads from all channels, to „unblock“ them
3. Threads proceed to print next value
4. Repeat until all counters are finished



Spawn Threads with an individual channel.
Repeat: Read something from every channel to „unblock“ them

Print a value, and send something in the channel!
Wait for completion!

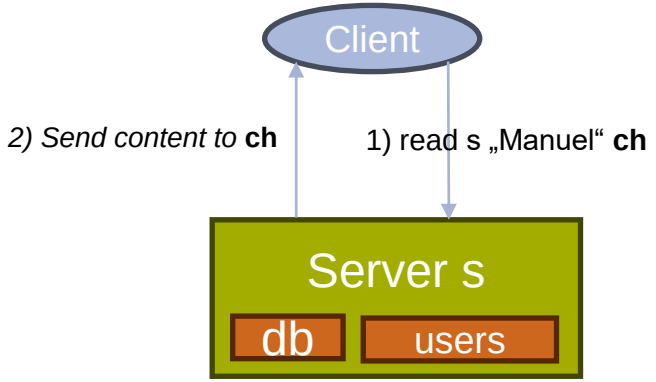
T02: Blog Server

Clients communicate with the server using messages through a single channel:

- A user can publish a new post on her block by sending a Post message to the server. The message has to contain the user's name, password and text to be published. If username and password are correct, the server appends the text to the user's blog. Messages with incorrect credentials or non-existing users are simply ignored.
- To read a user's blog, a Read message with the corresponding user has to be sent to the server. Furthermore, the message has to contain a channel on which the server sends the requested blog or an empty list if no such user exists.

Implement these functions:

1. **start_server** 1 of 1 tests passing
   `start_server : (user * pass) list -> t` starts a server on its own thread. As an argument the function receives the list of registered users and their corresponding passwords.
2. **post** 1 of 1 tests passing
   `post : t -> user -> pass -> string -> unit` publishes a new blog post (last argument) in the given users blog.
3. **read** 2 of 2 tests passing
   `read : t -> user -> blog` requests a user's blog from the server.



Server main loop:
while true:
    sync read message from channel
    parse message
    perfom action (update db)

T03: How about the Future

A *promise* represents the result of an asynchronous computation. Imagine a time-consuming operation, that is relocated to another thread, then the main thread keeps some kind of "handle" to check whether the operation in the other thread has finished, to query the result or as a means to do other operations with the result. This "handle" is what we call a *promise*.

Implement a `module Promise` with a type `'a t` that represents a promise object. Furthermore, perform these tasks:

1. **`promise`**
   Implement `promise : ('a -> 'b) -> 'a -> 'b t`, that applies the function given as the first argument to the data given as second argument in a separate thread. A promise for the result of this operation is returned.
2. **`await`**
   Implement `await : 'a t -> 'a` that waits for the asynchronous operation to finish and returns the result. It should be possible to call `await` multiple times on the same promise, so adapt your implementation of `promise` accordingly.
3. **Exception Support**
   Extend your implementation with exception support, such that if a function running in an asynchronous operation throws, this exception is stored, then raised again when a call to `await` is made on the promise.
4. **`map`**
   Implement `map : ('a -> 'b) -> 'a t -> 'b t` such that a call `map f p` returns a promise that represents the result of applying `f` to the result of the promise `p`. The application of `f` must again be asynchronous, so `map` must not block! The blocking must only happen once `await` is called.
5. **`bind`**
   Implement `bind : ('a -> 'b t) -> 'a t -> 'b t`. A call `bind f p` is like `map`, except that the function `f` returns a promise, which then in turn becomes the result of the promise returned by `bind`. Once again, the call to `bind` must not block.
6. **`any`**
   Implement `any : 'a t list -> 'a t` that constructs a promise that provides its result once any of the given promises has finished its computation, either normally or by raising an exception. The result of `any` is the result of that promise. Make sure that `any` does not block!
7. **`all`**
   Implement `all : 'a t list -> 'a list t` that constructs a promise that corresponds to a list of all the results of the given promises. If any promise raises an exception, the result of `all` should be an exception. Make sure that `all` does not block!
8. **More Future**
   Find additional useful functions for the module `Promise` and implement them.