Jacques Garrigue, 2006 年 2 月 1 日

14　並行計算とスレッド

スレッドの基礎

まず，スレッドを使うために，以下のコマンドで ocaml を起動させる．

```
$ ocaml -I +threads unix.cma threads.cma
```

OCamlbrowser でも，-I +threads を付けなければならない．
以下の関数が使われる．

```
Thread.create : ('a -> 'b) -> 'a -> Thread.t
                    (* Thread.create f x は f x を実行する新しいスレッドを作る *)
Thread.delay : float -> unit
                (* 現在のスレッドを x 秒 (以上) 休ませる．他のスレッドが実行される *)
```

その関数で，Hello を定期的に表示するスレッドを定義する．

```
let hello_thread () =
  while true do print_endline "Hello"; Thread.delay 5. done ;;
val hello_thread : unit -> unit
# Thread.create hello_thread ();;
- : Thread.t = <abstr>
# Hello
Hello              (* 5 秒おきに Hello が表示されるが，その合間に普通の処理ができる *)
```

場合によって，Ctrl-C でスレッドが止まらないこともある．emacs なら buffer を消せばいい．kterm なら，ウィンドウを閉じる．

```
let continue = ref true                          (* スレッドを止められるようにする *)
val continue : bool ref = {contents = true}
let hello_thread2 () =
  while !continue do print_endline "Hello"; Thread.delay 5. done;
  print_endline "Finished" ;;
val hello_thread2 : unit -> unit
# Thread.create hello_thread2 ();;
- : Thread.t = <abstr>
# Hello
Hello
continue :=Hello                                 (* 入力と出力がまざってしまう *)
 false;;
- : unit = ()
# Finished
```

イベントによる通信

イベントを処理するこんな関数を使う．

```
Event.new_channel : unit -> 'a Event.channel     (* 新しい通信チャンネルを作る *)
Event.send : 'a Event.channel -> 'a -> unit Event.event
                                (* 通信チャンネルにメッセージを送るイベントを作る *)
Event.receive : 'a Event.channel -> 'a Event.event
                            (* 通信チャンネルからメッセージをもらうイベントを作る *)
Event.sync : 'a Event.event -> 'a
```

```
                                    (* イベントが実行できるまで待って，それを実行する *)
Event.select : 'a Event.event list -> 'a    (* 複数のイベントのどれかを実行する *)
Event.poll : 'a Event.event -> 'a option  (* すぐに実行できるイベントを実行する *)
Event.wrap : 'a Event.event -> ('a -> 'b) -> 'b Event.event
                                              (* イベントのデータに関数を適用する *)
```

まず，send と receive を直ちに行う関数を定義する．

```
let send ch x = Event.sync (Event.send ch x)
val send : 'a Event.channel -> 'a -> unit
let receive ch = Event.sync (Event.receive ch)
val receive : 'a Event.channel -> 'a
```

イベントを使った Hello サーバ．

```
let rec hello_server ch =
  let b = receive ch in
  if b then (print_endline "Hello"; hello_server ch)
  else print_endline "Finished"
val hello_server : bool Event.channel -> unit
# let ch = Event.new_channel ();;
val ch : '_a Event.channel = <abstr>
# Thread.create hello_server ch;;
- : Thread.t = <abstr>
# send ch true;;                                          (* 一回送ってみる *)
- : unit = ()
# Hello

let rec timer_thread time ch =
  send ch true;
  Thread.delay time;
  timer_thread time ch ;;
val timer_thread : float -> bool Event.channel -> 'a
# Thread.create (timer_thread 5.) ch ;;
- : Thread.t = <abstr>
# Hello
Hello                                                     (* 止める方法がない *)

let rec timer_thread ch1 time ch2= (* ch1 から命令をもらい，ch2 でサーバに伝える *)
  let ch3 = Event.new_channel () in
  ignore (Thread.create (fun () -> Thread.delay time; send ch3 true) ());
  let b = Event.select [Event.receive ch1; Event.receive ch3] in
  send ch2 b;
  if b then timer_thread ch1 time ch2 else begin
    print_endline "Finish timer";
    ignore (receive ch3)
  end ;;
val timer_thread : bool Event.channel -> float -> bool Event.channel -> unit
# let ch' = Event.new_channel () ;;
val ch' : '_a Event.channel = <abstr>
# Thread.create (timer_thread ch' 3.) ch;;
- : Thread.t = <abstr>
# Hello
Hello
send ch' Hello
false;;
- : unit = ()
# Finish timer
Finished
```

検索問題への応用

もしもスレッドは全て並行に走ってくれるのなら，NP が実現できるはずだ．(実際には，処理系が並行実行をやっているではない)

まず，大量のスレッドが発生するので，同時に実行させる数を制限しないとシステムエラーが起きる．

```
let token = Event.new_channel ()                          (* スレッド終了時に送る *)
let threads = Event.new_channel ()                    (* 新しいスレッドを作らせる *)
let finished = Event.new_channel ()     (* スレッドが全て終了したときに送られる *)
let count = ref 100                  (* 実際の実行されるスレッドは 100 までとする *)
let waiting = ref []
let rec scheduler () =
  Event.select
    [Event.wrap (Event.receive token) (fun n -> incr count);
     Event.wrap (Event.receive threads) (fun f -> waiting := f :: !waiting)];
  while !count > 0 &&
    match !waiting with
      [] -> false
    | f :: l ->
        waiting := l;
        ignore (Thread.create (fun f -> f (); send token ()) f);
        true
  do decr count done;
  if !count = 100 then
    ignore (Thread.create
              (fun () -> while !count = 100 do send finished () done) ());
  scheduler ()
;;
val scheduler : unit -> 'a
# Thread.create scheduler ();;
- : Thread.t = <abstr>
let create f x =                              (* スレッドをスケジューラに作らせる *)
  send threads (fun () -> f x) ;;
val create : ('a -> '_b) -> 'a -> unit = <fun>
```

セールスマン問題をそれで解いてみる．

```
let random_matrix n =                                 (* 適当なグラフを生成する *)
  let m = Array.create_matrix n n 0 in
  for i = 1 to n-1 do
    for j = 0 to i-1 do
      let d = Random.int (2*n) - n in
      if d > 0 then begin
        m.(i).(j) <- d;
        m.(j).(i) <- d;
      end
    done
  done;
  m
val random_matrix : int -> int array array

# let m = random_matrix 10 ;;
val m : int array array =
  [|[|0; 0; 0; 0; 0; 4; 0; 8; 0|]; ... |]

let rec last l =
```

```
  match l with
    [] -> invalid_arg "last"
  | [a] -> a
  | _ :: l -> last l
val last : 'a list -> 'a

let rec search ch graph (visited : int list) next dist =
  if List.length visited = Array.length graph then
    if next = last visited then send ch (dist, List.rev visited)
    else ()
  else if List.mem next visited then () else
  let search' = search ch graph (next :: visited) in
  Array.iteri
    (fun next d ->
      if d > 0 then ignore (create (search' next) (dist+d)))
    graph.(next)
val search :
  (int * int list) Event.channel ->
  int array array -> int list -> int -> int -> unit

# let ch = Event.new_channel () ;;
val ch : '_a Event.channel = <abstr>
# search ch m [] 0 0 ;;
- : unit = ()

let next timeout ch =
  Event.select
    [Event.wrap (Event.receive finished) (fun () -> failwith "finished");
     Event.wrap (Event.receive timeout) (fun () -> failwith "timeout");
     Event.receive ch]
val next : unit Event.channel -> 'a Event.channel -> 'a
let best = ref (100000, [])
val best : (int * '_a list) ref = contents = (100000, [])
let better () =
  let timeout = Event.new_channel () in
  ignore (Thread.create
            (fun () -> Thread.delay 10.; Event.poll (Event.send timeout ()))
            ());
  let rec better () =
    let r = next timeout ch in
    if fst r < fst !best then (best := r; r) else better ()
  in better ()
val better : unit -> int * int list

# better();;
- : int * int list = (67, [0; 1; 4; 3; 7; 2; 9; 6; 8; 5])
# better();;
Exception: Failure "finished".
```

実習課題 (14回目)

1. 大きさ10から15までのグラフで実験してみなさい. betterを使った後に, countやList.length !waitingをみると処理の進み具合が分かる.

2. チャンネルとイベントを使って, 値が一個入るバッファーを定義せよ.