Jacques Garrigue, 2009 年 1 月 28 日

部屋 2 室での予約システム

```
let period_matrix n l =
  let m = Array.create_matrix n n 0 in
  List.iter (fun (b,e) -> m.(b).(e) <- m.(b).(e)+1) l;
  m

let update arr j m =
  if arr.(j) < m then arr.(j) <- m

let optimize n req =
  let req = period_matrix (n+1) req in                (* 申込のリストを行列に *)
  let mat = Array.create_matrix (n+1) (n+1) 0 in
  (* mat.(i).(j) は部屋 1 を i 日目まで，部屋 2 を j 日目まで使ったときの最大値 *)
  for i = 0 to (n-1) do
    for j = i to (n-1) do                          (* 対称性を使い，三角行列にする *)
      let m = mat.(i).(j) in
      update mat.(i) (j+1) m;                          (* 部屋 2 を 1 日空ける *)
      update mat.(i+1) j m;                            (* 部屋 1 を 1 日空ける *)
      if i = j then                                   (* 両部屋がともに i 日目 *)
for k = (i+1) to n do
  (* 部屋 2 だけに予約を入れる *)
  if req.(i+1).(k) = 1 then update mat.(i) k (m+k-i)
  (* 両部屋に予約を入れる *)
  if req.(i+1).(k) >= 2 then update mat.(k) k (m+2*(k-i))
done
      else
for k = (j+1) to n do
  if req.(i+1).(k) >= 1 then begin
            (* (三角行列を保つために) 部屋 1 と 2 を交換し，部屋 2 に予約を入れる *)
    update mat.(j) k (m+k-i);
    (* 部屋 1 にも丁度いい申込があれば，そちらも同時に予約 *)
    if req.(j+1).(k) >= 1 then update mat.(k) k (m+k-i+k-j)
  end;
         (* 部屋 2 だけを予約した場合 *)
  if req.(j+1).(k) >= 1 then update mat.(i) k (m+k-j)
done
    done
  done;
  mat
val optimize : int -> (int * int) list -> int array array

let opt = optimize 7 [(1,1);(1,3);(1,3);(2,6);(4,6); (7,7);(7,7)]
- : int array array =
[|[|0; 1; 1; 1; 1; 1; 6; 7|]; [|0; 1; 1; 4; 4; 4; 7; 8|];
  [|0; 1; 1; 4; 4; 4; 7; 8|]; [|0; 0; 1; 6; 6; 6; 9; 10|];
  [|0; 0; 0; 6; 6; 6; 9; 10|]; [|0; 0; 0; 0; 6; 6; 9; 10|];
  [|0; 0; 0; 0; 0; 6; 12; 12|]; [|0; 0; 0; 0; 0; 0; 12; 14|]|]
```

9　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
```

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
```

哲学者の晩餐会

哲学者 $2n$ 人が円いテーブルの回りに座っている．そこには料理を盛った $2n$ 枚の皿と n 本のフォークと n 本のナイフがある．フォークとナイフが足りないので，隣同士で共有している．各哲学者をプロセスとして定義し，食事をさせる．

```
(* fork と knife はフォークとナイフがテーブルにあるかどうかを表す *)
(* myfork と myknife はフォークとナイフが手に持っているかどうかを表す *)
let philosopher c fork knife name =
  let myfork = ref false and myknife = ref false in
  while !c do
    if !myfork && !myknife then begin
```

```
      (* フォークとナイフを持っているときは食べて，それらを返す *)
      print_endline (name ^ " eating");
      Thread.delay 1.;
      myfork := false; myknife := false;
      fork := true; knife := true;
      prerr_endline (name ^ " put fork and knife")
    end else if not !myfork && !fork then begin
      (* フォークがテーブルにあればそれを取る *)
      fork := false; myfork := true;
      prerr_endline (name ^ " got fork")
    end else if not !myknife && !knife then begin
      (* ナイフがテーブルにあればそれを取る *)
      knife := false; myknife := true;
      prerr_endline (name ^ " got knife")
    end;
    Thread.delay 1.
  done;
  print_endline (name ^ " finished")
val philosopher : bool ref -> bool ref -> bool ref -> string -> unit

let f1 = ref true and f2 = ref true
and k1 = ref true and k2 = ref true
val f1 : bool ref = {contents = true}
...
let c = ref true
val c : bool ref = {contents = true}

let plato = Thread.create (philosopher c f1 k1) "Plato"
and descartes = Thread.create (philosopher c f2 k1) "Descartes"
and kant = Thread.create (philosopher c f2 k2) "Kant"
and leibniz = Thread.create (philosopher c f1 k2) "Leibniz"
val plato : Thread.t = <abstr>
val descartes : Thread.t = <abstr>
val kant : Thread.t = <abstr>
val leibniz : Thread.t = <abstr>
Plato got fork
Descartes got fork
Kant got knife
Plato got knife
Plato eating
Plato put fork and knife
Descartes got knife
Plato got fork
Descartes eating
Descartes put fork and knife
Leibniz got forkPlato got knife

Plato eating
Plato put fork and knife
Descartes got knife
Plato got fork                                          (* なぜか出力が止まる *)
c:=false;;
- : unit = ()
Kant finished
```

```
Leibniz finished
Descartes finished
Plato finished
```

さて，なぜ止まったかを調べてみると，4 人の哲学者がそれぞれフォークがナイフのどちらかを手に持って，もう一方がテーブルにないので，食べられないでいる．それを改善する簡単な方法は必ずフォークを先に取るようにする．

```
let philosopher c fork knife name =
    ...
    (* ナイフを取るために，フォークを既に持っていなければならない *)
    end else if !myfork && not !myknife && !knife then begin
      knife := false; myknife := true;
      prerr_endline (name ^ " got knife")
    end;
    ...
  print_endline (name ^ " finished")
val philosopher : bool ref -> bool ref -> bool ref -> string -> unit
```

修正されたプログラムでは晩餐会がいつまでも続く……

練習問題 9.1 *1.* 任意の数の哲学者が参加する晩餐会を関数で定義せよ．

2. プリント文を各スレッドの中で実行するのではなく，プリントサーバーを定義して，それを使うようにせよ．フォークとナイフ同様，プリントサーバーがに値を渡すのに printing 参照を true にしてから，phrase をセットしなければならない．

```
let printing = ref false
let phrase = ref ""
let print s =
  while !printing do Thread.delay 0.1 done;
  printing := true; phrase := s
```