TD7-TME7 : Events et Canaux Synchrones en CaML

Exercice 1 – Mobilité : Vente en ligne

On se propose de modéliser un service de vente en ligne impliquant un intermédiaire. Chaque élément du système sera représenté par un thread et sera associé à un canal (il y aura donc autant de canaux que d'agents dans le système). Le système sera composé :

- De deux clients, qui envoient des requêtes pour un produit spécifique (par exemple ″`cafe`″ pour l'un, ″`the`″ pour l'autre) sur le canal de l'intermédiaire. Ils attendent ensuite de recevoir un nom canal frais sur leur propre canal, puis attendent de recevoir le produit sur ce nouveau canal. Ils recommencent un nombre fini de fois.
- D'un intermédiaire qui reçoit sur son canal des requêtes des deux clients. Lors de la réception d'une requete, i) il crée un nouveau canal, ii) il envoie sur le canal du client le nom du nouveau canal, iii) il envoie sur le canal du vendeur un couple composé du produit demandé et du nom du nouveau canal. Ensuite, il se relance.
- D'un vendeur qui contient un compteur interne et qui recoit une requete composé d'un produit et d'un canal. Il crée un "produit fini" composé du nom du produit demandé et de son compteur (comme ″`the 3`″, par exemple). Il envoie sur le canal reçu le produit fini.

Question 1

Décrypter la spécification. Décrire les agents du systèmes, les différents canaux utilisés, et le comportement de chacun des agents.

Question 2

Ecrire les fonctions récursives exécutées par le vendeur et l'intermédiaire, qui proposent des services (et sont donc non-terminantes). Celle du vendeur incrémentera son compteur à chaque requête traitée.

L'intermédiaire doit pouvoir distinguer quel client lui parle quand il reçoit une requête (afin de communiquer le nouveau canal à la bonne personne). Comment procéder ?

Question 3

Ecrire la fonction exécutée par les deux threads clients. Afin de vérifier que les clients récupèrent bien le produit demandé, on fera en sorte que chaque clients mette à jour une variable `log` qui enregistrera les produits reçus.

Question 4

Ecrire la fonction principale qui lance les quatre threads (deux clients, un intermédiaire, un vendeur), qui attend la terminaison des deux clients et qui affiche leurs logs.

Solution:

```
let c_sell = Event.new_channel ()
and c_brok = Event.new_channel ()
and c_buye1 = Event.new_channel ()
and c_buye2 = Event.new_channel ()
and log1 = ref ""
and log2 = ref ""

let rec seller n =
  let (y,z) = Event.sync (Event.receive c_sell) in
  Event.sync (Event.send y (z^" "^(string_of_int n)));
  seller (n+1)
```

```
let rec buyer arg =
  let (a,n,c_buye,log,varlog) = arg in
  if n == 0 then varlog:=log else
    begin
      Event.sync (Event.send c_brok (a,c_buye));
      let chan = Event.sync (Event.receive c_buye) in
      let prod = Event.sync (Event.receive chan) in
      buyer (a,n-1,c_buye,(log^"J'avais demande "^a^", j'ai recu "^prod^".\n"),varlog)
    end



let rec broker () =
  let (x,c_buye) = Event.sync (Event.receive c_brok)
  and nu_c = Event.new_channel () in
  Event.sync (Event.send c_sell (nu_c,x));
  Event.sync (Event.send c_buye nu_c);broker ()

let main =
  let _ = Thread.create seller 1
  and _ = Thread.create broker ()
  and tbuy1 = Thread.create buyer ("the",3,c_buye1,"",log1)
  and tbuy2 = Thread.create buyer ("cafe",2,c_buye2,"",log2)
  in
  Thread.join tbuy1;
  Thread.join tbuy2;
  print_endline !log1;
  print_endline !log2
```

Exercice 2 – Sélection

Question 1

Ecrire un programme qui lance trois threads fournisseurs qui mettent un temps aléatoire pour envoyer, chacun sur un canal différent, une produit différent (par exemple les chaines de caractères ″`pommme`″, ″`orange`″ et ″`banane`″) et un quatrième thread qui doit recevoir tous les produits. Les threads fournisseurs se lancent un nombre fini de fois.

Solution:

```
let c_b = Event.new_channel ()
and c_p = Event.new_channel ()
and c_o = Event.new_channel ()
and max = 7


let rec work (str,chan,n) =
  if (n < max) then
    begin
      Thread.delay ((float_of_int (3 + (Random.int 10)))/. 5.0);
```

```
      let _ = Event.sync (Event.send chan (str^" "^(string_of_int n))) in
      work (str,chan,(n+1))
    end
  else ()

let rec consumer () =
  let x = Event.select
    [Event.receive c_p;Event.receive c_b;Event.receive c_o]
  in print_endline x;consumer ()

let main () =
  let t1 = Thread.create work ("pomme",c_p,1)
  and t2 = Thread.create work ("orange",c_o,1)
  and t3 = Thread.create work ("banane",c_b,1)
  and _ = Thread.create consumer () in
  Thread.join t1;Thread.join t2;Thread.join t3

let _ = main ()
```

Exercice 3 – Broadcast

On se propose d'écrire un serveur de broadcast, créé avec comme argument une liste de canaux de sortie : a chaque fois qu'il recoit une information sur son canal d'entrée, il propage cette information sur tous les canaux de sortie.

On dispose, en outre, dans le système, de plusieurs threads écouteurs, chacun associé à un canal de sortie. Ils attendent un temps aléatoire puis recoivent l'information sur leur canal de sortie associé.

Question 1

Ecrire une fonction `broadcast_aux` qui prend une liste de canaux et une valeur et qui envoie une fois la valeur sur chaque canal de la liste. Attention : l'ordre dans lequel les synchronisations vont s'effectuer doit pouvoir varier ; on pourra utiliser wrap.

Question 2

Ecrire le programme entier. Utiliser un mutex pour afficher les sorties des threads écouteurs.

Solution:

```
let sortie = Mutex.create ()

let rec ecouteur (chan,id) =
  let x = let _ = Thread.delay ((float_of_int (3 + (Random.int 10)))/. 5.0) in
          Event.sync (Event.receive chan) in
  Mutex.lock sortie;
  print_endline ((string_of_int id)^" recoit "^x);
  Mutex.unlock sortie;
  ecouteur (chan,id)

let rec broadcast (l,bd) =
  let rec rem x l =
    let rec rem_a x acc = function
      | [] -> acc
      | t::q -> if x == t then rem_a x acc q else rem_a x (t::acc) q in
```

```
    rem_a x [] l in
  let rec broadcast_aux x l = match l with
      [] -> ()
    | _ -> Event.select (List.map
                           (fun ch -> Event.wrap (Event.send ch x)
                             (fun _ -> broadcast_aux x (rem ch l)))
                           l)   in
  let y = Event.sync (Event.receive bd) in
  broadcast_aux y l;broadcast (l,bd)

let creer_liste n =
  let rec creer_liste_aux n acc =
    if n = 0 then acc else creer_liste_aux (n-1) ((Event.new_channel ())::acc)
  in creer_liste_aux n []

let creer_threads l =
  let rec creer_threads_a n = function
    | [] -> ()
    | ch::q -> let _ = (Thread.create ecouteur (ch,n)) in
               creer_threads_a (n+1) q in
  creer_threads_a 1 l

let main () =
  let l = creer_liste 10 in
  let _ = creer_threads l in
  let bd = Event.new_channel () in
  let _ = Thread.create broadcast (l,bd) in
  let _ = Event.sync (Event.send bd "bonjour") in
  let _ = Event.sync (Event.send bd "ca va ?") in
  let _ = Event.sync (Event.send bd "au revoir") in
  let _ = Thread.delay 30.0 in ()

let _ = main ()
```

Exercice 4 – Monte-Carlo

Dans cet exercice on se propose de calculer π par la méthode de MonteCarlo !

On tire aléatoirement un point $M(x, y)$ avec $0 < x < 1$ et $0 < y < 1$.
Il appartient au disque (plus exactement au quart supérieur droit du disque) de centre $(0, 0)$ et de rayon 1 si et seulement si $x^2 + y^2 \leq 1$.
Tout se base sur le fait simple que la probabilité que ce point $M(x, y)$ appartient au disque est $\pi/4$.
Pour calculer π, il suffit donc de tirer aléatoirement des points $M(x, y)$ avec $-1 < x < 1$ et $-1 < y < 1$ et de calculer le rapport entre le nombre de points tirés appartenant au disque et le nombre total de points tirés.

Question 1

Ecrire un programme mettant en jeu :
- deux `channel`
- de ref sur des grands entiers

– 4 threads

On souhaite avoir deux threads qui testents des points et deux threads comptant respectivemnt les point dans le cercle et les points hors du cercle.

Solution:

```
(* dessin: un cercle de rayon 1 dans un carré de coté 2 *)

(***** A compiler avec

   ocamlc -thread unix.cma threads.cma nums.cma MonteCarlo_exit_1.ml \
        -o MonteCarlo_exit_1

*****)


open Printf;;
open Num;;
open Event;;


let voie      = new_channel();;
let pi        = ref 0.;;
let ok        = ref(num_of_int 0);;
let pasok     = ref(num_of_int 0);;
let clefok    = Mutex.create();;
let clefpasok = Mutex.create();;

let compteur (numero, channel) =
  while true do
    let x = (sync (receive channel)) in
      printf "Compteur %d " numero;
      if x then
        begin
          Mutex.lock clefok;
          incr_num ok;
          printf "Valeur OK = %d\n" (int_of_num !ok);
          Mutex.unlock clefok
        end
      else begin
             Mutex.lock clefpasok;
             incr_num pasok;
             printf "Valeur PAS_OK = %d\n" (int_of_num !pasok);
             Mutex.unlock clefpasok
           end
  done;;

let testeur (numero, nbop, channel) =
  for i = 1 to nbop do
    let x = (Random.float 2.0) -. 1.
        and y = (Random.float 2.0) -. 1. in
      printf "Testeur %d\n" numero;
      Thread.delay 0.1;
```

```
      (sync (send channel (x *. x +. y *. y < 1.0)))
  done;;

let main () =
  let th1 = Thread.create compteur (1, voie)
      and th2 = Thread.create compteur (2, voie)
      and th3 = Thread.create testeur (1, int_of_string Sys.argv.(1), voie)
      and th4 = Thread.create testeur (2, int_of_string Sys.argv.(1), voie) in

    Thread.delay 0.1;          (* Le temps pour les printf *)
    Thread.join th3;
    print_string "Fin th3\n";

    Thread.delay 0.1;          (* Le temps pour les printf *)
    Thread.join th4;
    print_string "Fin th4\n";

    Thread.delay 0.1;          (* Le temps pour les printf et compteur.
                                  Pas sur que les compteurs aient le temps
                                  de traiter le dernier OK ou PAS_OK.
                                  Meme avec un lock ou non.
                                  PAS TRES PROPRE *)
                               (* Un kill de th1 et/ou th2 est mauvais car
                                  on ne sait pas à ce niveau s'ils ont fini
                                  ou pas *)
    pi:=((float_of_num !ok)/.((float_of_num !pasok)+.(float_of_num !ok)))*.4.;
    print_float !pi;
    print_newline();

    print_string "j'ai fini\n";
    exit 0;;                   (* un peu violent *)

main ();;
```

Solution:

```
(* dessin: un cercle de rayon 1 dans un carré de coté 2 *)

(***** A compiler avec

   ocamlc -thread unix.cma threads.cma nums.cma MonteCarlo_exit_2.ml \
        -o MonteCarlo_exit_2

*****)


open Printf;;
open Num;;
open Event;;


let voie      = new_channel();;
```

```
let pi        = ref 0.;;
let ok        = ref(num_of_int 0);;
let pasok     = ref(num_of_int 0);;
let clefok    = Mutex.create();;
let clefpasok = Mutex.create();;

let compteur (numero, channel) =
  while true do
    let x = (sync (receive channel)) in
      printf "Compteur %d " numero;
      if x = 2.0 then
         Thread.exit ()
      else if x < 1. then
        begin
          Mutex.lock clefok;
          incr_num ok;
          printf "Valeur OK = %d\n" (int_of_num !ok);
          Mutex.unlock clefok
        end
      else begin
             Mutex.lock clefpasok;
             incr_num pasok;
             printf "Valeur PAS_OK = %d\n" (int_of_num !pasok);
             Mutex.unlock clefpasok
           end
  done;;

let testeur (numero, nbop, channel) =
  for i = 1 to nbop do
    let x = (Random.float 2.0) -. 1.
        and y = (Random.float 2.0) -. 1. in
      printf "Testeur %d\n" numero;
      Thread.delay 0.1;
      sync (send channel (x *. x +. y *. y))
  done;;

let main () =
  let th1 = Thread.create compteur (1, voie)
      and th2 = Thread.create compteur (2, voie)
      and th3 = Thread.create testeur (1, int_of_string Sys.argv.(1), voie)
      and th4 = Thread.create testeur (2, int_of_string Sys.argv.(1), voie) in

    Thread.delay 0.1;          (* le temps pour les printf *)
    Thread.join th3;
    print_string "Fin th3\n";

    Thread.delay 0.1;          (* le temps pour les printf *)
    Thread.join th4;
    print_string "Fin th4\n";

    sync (send voie 2.0);
    sync (send voie 2.0);
```

```
    Thread.join th1;
    print_string "Fin th1\n";
    Thread.join th2;
    print_string "Fin th2\n";

    pi:=((float_of_num !ok)/.((float_of_num !pasok)+.(float_of_num !ok)))*.4.;
    print_float !pi;
    print_newline();

    print_string "j'ai fini\n";
    exit 0;;                    (* un peu violent *)

main ();;
```

Solution:

```
(* dessin: un cercle de rayon 1 dans un carré de coté 2 *)

(***** A compiler avec

   ocamlc -thread unix.cma threads.cma nums.cma MonteCarlo_exit_3.ml \
        -o MonteCarlo_exit_3

*****)


open Printf;;
open Num;;
open Event;;


type info =
    De_main
  | Du_testeur of float;;

let voie:(info channel) = new_channel();;
let pi                  = ref 0.;;
let ok                  = ref(num_of_int 0);;
let pasok               = ref(num_of_int 0);;
let clefok              = Mutex.create();;
let clefpasok           = Mutex.create();;

let compteur (numero, channel) =
  while true do
    match sync (receive channel) with
        De_main -> Thread.exit()
      | Du_testeur x
          -> printf "Compteur %d " numero;
             if x < 1. then
                begin
                  Mutex.lock clefok;
```

```
                    incr_num ok;
                    printf "Valeur OK = %d\n" (int_of_num !ok);
                    Mutex.unlock clefok
                  end
               else begin
                      Mutex.lock clefpasok;
                      incr_num pasok;
                      printf "Valeur PAS_OK = %d\n" (int_of_num !pasok);
                      Mutex.unlock clefpasok
                    end
  done;;

let testeur (numero, nbop, channel) =
  for i = 1 to nbop do
    let x = (Random.float 2.0) -. 1.
        and y = (Random.float 2.0) -. 1. in
      printf "Testeur %d\n" numero;
      Thread.delay 0.1;
      sync (send channel (Du_testeur (x *. x +. y *. y)))
  done;;

let main () =
  let th1 = Thread.create compteur (1, voie)
      and th2 = Thread.create compteur (2, voie)
      and th3 = Thread.create testeur (1, int_of_string Sys.argv.(1), voie)
      and th4 = Thread.create testeur (2, int_of_string Sys.argv.(1), voie) in

    Thread.delay 0.1;          (* le temps pour les printf *)
    Thread.join th3;
    print_string "Fin th3\n";

    Thread.delay 0.1;          (* le temps pour les printf *)
    Thread.join th4;
    print_string "Fin th4\n";

    sync (send voie De_main);
    sync (send voie De_main);

    Thread.join th1;
    print_string "Fin th1\n";
    Thread.join th2;
    print_string "Fin th2\n";

    pi:=((float_of_num !ok)/.((float_of_num !pasok)+.(float_of_num !ok)))*.4.;
    print_float !pi;
    print_newline();

    print_string "j'ai fini\n";
    exit 0;;                   (* un peu violent *)

main ();;
```

Exercice 5 – Examen 2012

Les étudiants de l'UE MI019 PC2R jouent au jeu PPNU (plus petit nombre unique) qui consiste à deviner le plus petit nombre qui ne sera choisi par aucun autre joueur. En pratique, chaque étudiant écrit une fonction (de type `unit -> int` en OCaml) qui rend un nombre (les fonctions peuvent faire des opérations d'entrées/sorties avec le monde extérieur, mais on suppose qu'elles finissent toutes par terminer en temps raisonnable).

Soit `jVec` le vecteur qui contient toutes les fonctions des joueurs. En OCaml, `jVec` est de type `(unit -> int) array`.

Attention à bien gérer la synchronisation de manière à ne pas obtenir un « simple programme séquentiel » !

Question 1

Créer un canal d'événements nommé `cRes` qui vous servira à envoyer les résultats des fonctions des joueurs (donc des paires d'entiers : numéro du joueur $\times$ nombre choisi) a un thread jouant le rôle d'arbitre.

Question 2

Donner une définition du thread arbitre qui se charge de garder en mémoire le plus petit nombre unique qu'il a reçu ainsi que le numéro du joueur bien sûr. À la fin du jeu, le thread arbitre affiche le numéro du gagnant ainsi que le nombre qu'il a choisit, et meurt.

Question 3

Créer un canal d'événements nommé `cFun` qui vous servira à envoyer les fonctions des joueurs aux threads chargés de les appliquer. Chaque fonction sera envoyée avec son numéro.

Question 4

Créer un thread qui envoie toutes les fonctions du vecteur `jVec` dans le canal d'événement `cFun`.

Question 5

Créer un lot de `nTh` threads, chacun se chargeant, **tant qu'il est nécessaire de le faire**, de choisir une fonction dans le canal d'événements `cFun`, de l'exécuter, et d'envoyer le résultat de la fonction sur le canal `cRes`.

Question 6

Écrire le code qui manque pour lancer le programme et afficher le résultat (une seule fois, et si on veut rejouer, on relance le programme). Si aucun nombre choisi n'est unique, il n'y a aucun gagnant, et on pourra déclarer vainqueur le joueur `-1`.

Question 7

Expliquer (en le moins de mots possible) pourquoi vous n'avez pas eu besoin d'utiliser des *mutex*. (Toutefois, si vous avez vraiment dû en utiliser, expliquer pourquoi ils vous ont été indispensables.)

Question 8

Votre programme est-il robuste aux limitations des ressources du système ? Expliquer pourquoi (en le moins de mots possible). (Selon vos choix d'implantation, la réponse peut être oui comme elle peut être non, ce qui compte ici est donc l'explication.)

Question 9

Bonus : expliquer comment gérer une durée de vie limitée (*timeout*) pour chaque fonction des joueurs, ou bien implantez-le.

Voici les signatures des fonctions que vous pouvez utiliser si vous choisissez C en langage d'implantation.

C

```
typedef struct channel* channel_t;
typedef struct event* event_t;
channel_t new_channel ();
event_t send (channel_t, void*);
event_t receive (channel_t);
event_t always (void*);
event_t choose2 (event_t, event_t);
event_t choose3
  (event_t, event_t, event_t);
```

C

```
event_t chooseN (event_t*);
void* sync (event_t);
void* select2 (event_t, event_t);
void* select3 (event_t, event_t, event_t);
void* selectN (event_t*);
int poll (event_t, void**);
/* renvoie TRUE si une valeur est disponible,
auquel cas elle est stockée dans le second paramètre
 */
```

Solution:

```
(* Pour tester le programme : #!/bin/bash for i in {1..10} ; do
   ocamlopt -thread unix.cmxa threads.cmxa une_solution_pour_event.ml -o
   ppnu-event.exe && ./ppnu-event.exe ; done
*)

(* Par Philippe Wang *)

(** Creer quelques joueurs aleatoirement pour pouvoir tester ce mini
   programme. *)
let () =
  (* S'assurer du non determinisme ! *)
  Random.self_init ()

let jVec =
  (* Creation des joueurs. *)
  Array.init 300 (fun i -> fun () -> Random.int 50)

(* Q14. *)
let cRes : (int * int) Event.channel =
  (* Canal permettant de transmettre des paires d'entiers. *)
  Event.new_channel ()

(* Variable globale necessaire pour repondre a Q18 au vu de la suite
   du programme. (On peut bien sÃ»r faire autrement.)  *)
let continue : bool ref =
  ref true

(* Q15. *)
let arbitre : unit -> (int * int) = fun () ->
  (* Rend le plus petit nombre unique recu s'il existe, avec le numero
     du joueur; sinon, rend la paire (-1,-1). *)
  let module M =
    (* Module locale pour utiliser a moindre frais des arbres
       equilibres dont les noeuds sont indexes par des entiers. *)
    Map.Make(struct type t = int let compare = (-) end)
  in
  let memory : int M.t ref =
    (* La memoire pour stocker certains resultats des joueurs. *)
```

```
    ref M.empty
  in
    for i = 1 to Array.length jVec do
      (* Lire le bon nombre de valeurs dans le canal,
         ce nombre correspond a la taille du vecteur jVec. *)
      let (j : int), (v:int) =
        (* Recevoir une paire (valeur*joueur) sur le canal. On synchronise
           tout de suite car ca ne sert a rien de continuer sinon.*)
        Event.sync (Event.receive cRes)
      in
        if M.mem v !memory then
          (* Si la valeur existait dans la memoire, alors elle n'est
             pas unique, on la marque avec le joueur -1 (par convention
             arbitraire). *)
          memory := M.add v (-1) (M.remove v !memory)
        else
          (* Sinon, on ajoute la valeur et le joueur dans la
             memoire. *)
          memory := M.add v j !memory
    done;
    (* On peut signaler a tous qu'il est inutile de continuer.  Ce
       thread etant le seul a ecrire dans la reference 'continue', il
       n'est pas necessaire de proteger cette ecriture. *)
    continue := false;
    if M.is_empty !memory then
      (* Si la memoire est vide, alors c'est qu'il n'y avait aucun
         nombre unique. *)
      (-1, -1)
    else
      (* Sinon, il faut chercher le plus petit nombre unique. *)
      (let rec find () =
        if M.is_empty !memory then
          (-1, -1)
        else
          match M.min_binding !memory with
            | k, -1 -> memory := M.remove k !memory; find()
            | v, j -> v, j
        in find ())

        (* Plus court aurait ete les 2 lignes suivantes :
           try M.min_binding (M.filter (fun v j -> j <> -1) !memory)
           with Not_found -> -1, -1
           mais cela est moins efficace du point de vue des performances
           car on parcourt alors l'ensemble de l'arbre d'abord.
        *)


(* Q16. *)
let cFun : (int * (unit -> int)) Event.channel =
  (* Canal permettant de transmettre les numeros des joueurs
     avec leurs fonctions. *)
  Event.new_channel ()
```

```
(* Q17. La creation mÃªme du thread sera plus tard...
   mais elle aurait pu se faire tout de suite. *)
let send = fun () ->
  (* On cree un thread a chaque fois pour s'assurer que les envois
     peuvent se faire (presque) n'importe quand.  Pas obligatoire
     d'attendre la mort des threads (parce qu'on est en OCaml et qu'on
     beneficie d'un GC qui va se charger de faire cela et qu'on n'a
     pas besoin de le faire explicitement dans notre cas precis). *)
  Array.iteri
    (fun i j ->
      ignore (Thread.create
                (fun () -> Event.sync (Event.send cFun (i, j))) ()))
    jVec

(* Q18. *)
let make_threads = fun nTh ->
  (* Les nTh threads qui vont se charger d'executer les fonctions. *)
  for i = 1 to nTh do
    ignore
      (Thread.create
         (fun () ->
           while !continue do
              (* En fait, on pourrait mettre true a la place de
                 !continue car on ne fait qu'un seul tour. *)
             let ef = Event.receive cFun in
             let (j:int), (f:unit -> int) = Event.sync ef in
             Event.sync (Event.send cRes (j, f ()))
            (* On a un risque d'avoir des threads qui ne meurent
               pas... Mais d'apres les specifications du
               programme, comme ces threads ne sont pas attendus
               et qu'il ne s'agit pas du thread principal, ce
               n'est pas grave qu'ils restent passivement
               bloques sur une attente (Event.sync). *)
           done)
         ())
  done

let _ =
  (* lancement du programme... *)
  ignore (Thread.create send ());
  ignore (Thread.create make_threads 42);
  let nombre, gagnant = arbitre ()
  in Printf.printf "Le gagnant est %d avec le nombre %d\n%!" gagnant nombre
```