MdP : COURS 6

Modèles de Programmation

Emmanuel Chailloux

Plan du cours

- concurrence
  - modèle à mémoire partagée
  - modèle à mémoire répartie
- Processus légers (threads), préemption, coopération
- Canaux synchrones
- Internet et applications réparties

pourquoi la concurrence ?

séquentialité & concurrence :

- séquentialité (dépendence causale) : une instruction s'exécute après une autre ;
- la concurrence (indépendance causale) : plusieurs instructions s'exécutent en «même temps» ;

1. expressivité : facilité l'écriture d'algorithmes
   - séparation des tâches, explicitation de la communication, ...
2. efficacité : machines multicœurs et en réseau
   - différence entre puissances théorique et réelle

Modèles de parallélisme

2 grands modèles de programmation parallèle (ou simultanée) :

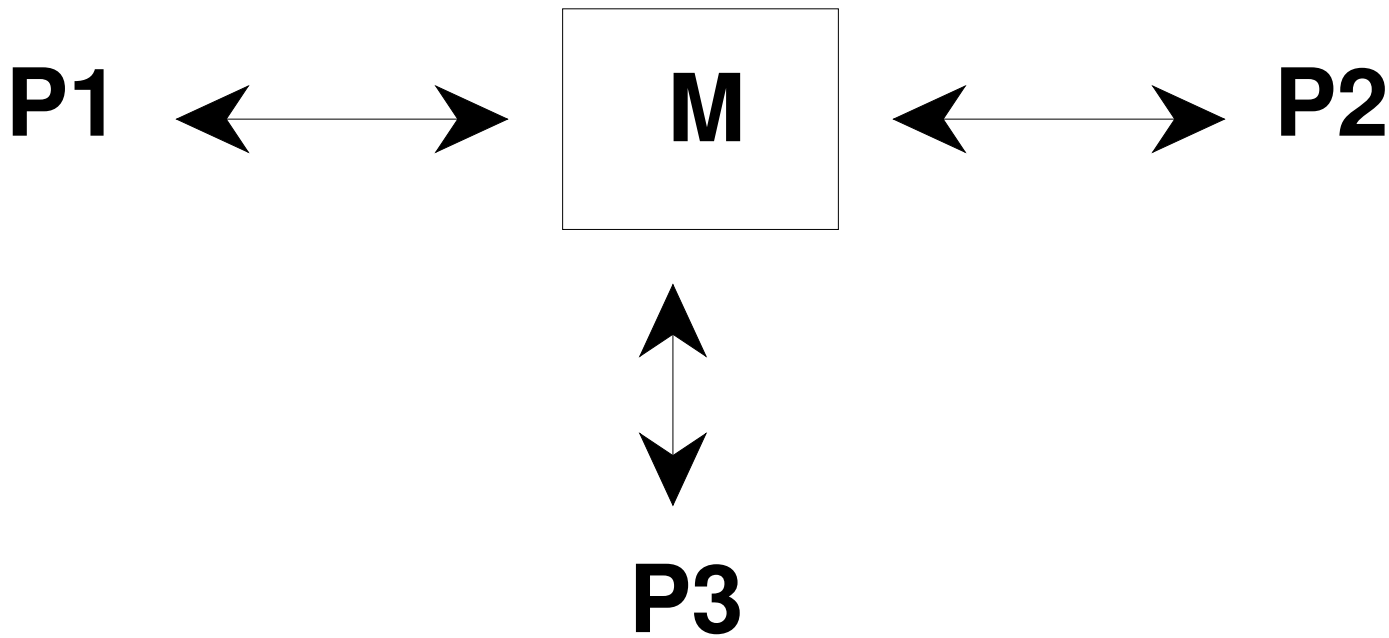
- systèmes à mémoire partagée
- systèmes répartis (à mémoire répartie)

Autres caractéristiques

- le non-déterminisme (une cause peut avoir plusieurs effets, mutuellement exclusifs) :
  un même programme ne termine pas ou termine en produisant des résultats différents ;
- la synchronisation (plusieurs causes indépendantes doivent s'être produites avant que l'effet puisse avoir lieu) :
  attente d'une condition sur plusieurs processus ;
- la communication (transfert d'informations) :
  envoi et réception d'informations d'un ou plusieurs processus à un ou plusieurs processus.

Systèmes à mémoire partagée (1)

On considère un ensemble S de processus séquentiels P_i interagissant sur une mémoire commune (ou partagée) que l'on note $S = [P_1 \,||...||\, P_n]$. Ces processus peuvent être aussi bien physiquement indépendants (un processus correspond à un processeur) que simulés logiquement par un unique processeur (comme les Threads en OCaml).



Systèmes à mémoire partagée (2)

La communication dans ce modèle est implicite.

L'information est transmise lors de l'écriture dans une zone de la mémoire partagée, puis quand un autre processus vient lire cette zone. Ce mécanisme est asynchrone, i.e. il ne nécessite pas que le récepteur soit prêt à écouter l'émetteur.

Par contre la synchronisation doit être explicite, en utilisant des instructions élémentaires.

Systèmes à mémoire partagée (3)

Sans synchronisation explicite, le résultat d'un programme est imprévisible. Par exemple, soit l'ensemble S de processus (avec x valant 0) défini ainsi : $S = [x := x + 1; x := x + 1 || x := 2 * x]$. Après l'exécution de S, x peut valoir 2,3, ou 4.

La synchronisation la plus simple est l'attente d'une condition. On la note *wait b*, où b est une expression booléenne. Un processus ne peut exécuter cette instruction que si b est vraie. En reprenant

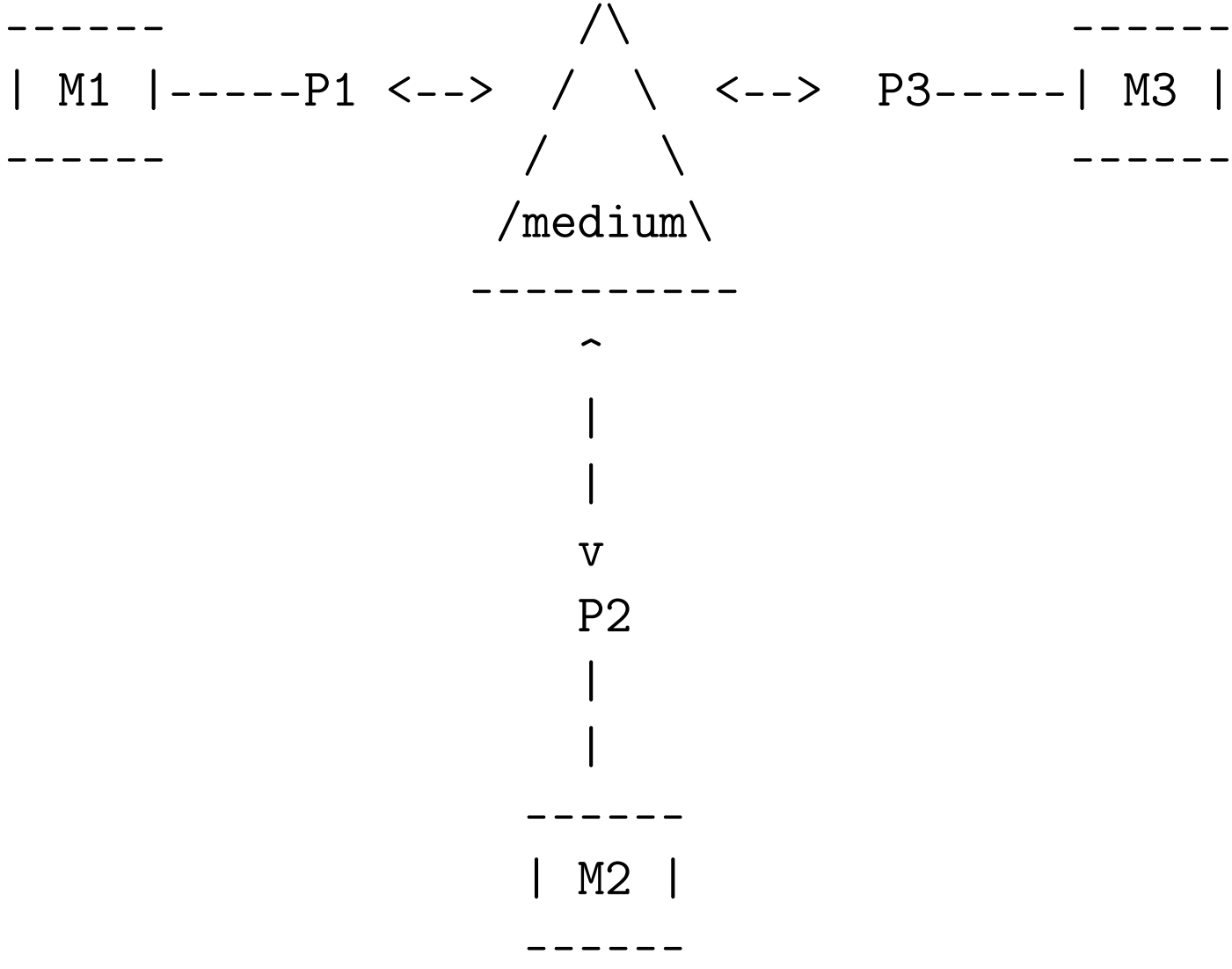
l'exemple précédent :
$S = [x := x + 1; x := x + 1 || wait(x = 1); x := 2 * x]$ on obtient comme valeur pour x que 3 ou 4. Par contre il est possible que le second processus reste bloqué s'il n'a testé x que pour les valeurs 0 ou 2.

Systèmes à mémoire partagée (4)

Cela amène le problème de l'atomicité. Il peut être utile de manipuler l'atomicité de manière explicite.

L'instruction *await b do P* attend que la condition b soit vraie pour exécuter les instructions de P de manière atomique dans le même état mémoire que le test de b.

Modèle à mémoire répartie (1)



La difficulté de ce modèle provient de l'implantation du medium.
Les programmes s'en chargeant s'appellent des *protocoles*.

Protocoles et communication

- **protocole** : organisés en couche. Les protocoles de haut niveau, implantant des services élaborés, utilisent les couches de plus bas niveaux (7 couches : modèle ISO).
- **parallélisme** : modèle valable dans le cas de parallélisme physique (réseau d'ordinateurs) ou logique (processus Unix communiquant par "pipes" ou threads O'Caml communiquant par canaux). Il n'y a pas de valeurs globales connues par tous les processus (comme un temps global). La seule contrainte sur le temps est l'impossibilité de recevoir un message avant son émission.
- **communication et synchronisation** : Dans ce modèle la communication est explicite alors que la synchronisation est implicite (elle est en fait produite par la communication). Ce modèle est le dual du précédent.

Modèles de communication

- **un-à-un** (point-à-point) : communication d'un processus à un autre ; les autres processus ignorent cette communication. Les deux primitives sont "envoi d'une valeur sur un canal" et "réception d'une valeur d'un canal".
- **un-à-tous** (diffusion) : communication d'un processus à tous les processus. Les primitives de communication sont : "envoi d'une valeur à tous" et "réception d'une valeur".
- **tous-à-tous** (diffusion) : communication de tous les processus à tous l es processus. La réception tient compte alors des différentes valeurs envoyées.

Types de communication

- **synchrone** : le transfert d'informations n'est possible que lors d'une synchronisation globale des processeurs émetteur et récepteur. L'émission et la réception peuvent être bloquantes.
- **asynchrone** : le medium peut stocker des messages en vue de leur achemineme nt futur. Il faut donc spécifier la capacité de stockage, l'ordre d'achemine ment, les délais de transmissions et la fiabilité de transmission. L'émission est non bloquante.
- **évanescent** : l'émission est non bloquante et le medium ne peut pas sto cker de messages. Le message émis est reçu par les processus prêt à le recevoir et perdu pour les autres.

Section critique, exclusion mutuelle

On appelle *section critique* une ressource qui ne doit être utilisée que par un processus au plus.

- Par exemple, on désire qu'un seul processus puisse utiliser une imprimante. C'est le cas du système Unix qui gère une queue d'impression sur les périphériques d'impression.

Pour cela les processus doivent s'exclure mutuellement de la section critique. On dit que l'activité A_1 du processus P_1 et l'activité A_2 du processus P_2 sont en *exclusion mutuelle* lorsque l'exécution de A_1 ne doit pas se produite en même temps que celle de A_2.

- algorithmes d'exclusion mutuelle
  - Dekker, Peterson, Lamport (Bakery)

Sémaphores (1)

Un sémaphore est une variable entière s ne pouvant prendre que des valeurs positives (ou nulles). Une fois s initialisé, les seules opérations admises sont : *wait*(s) et *signal*(s), notées respectivement $P(s)$ et $V(s)$. Elles sont définies ainsi :

- *wait*(s) : si $s > 0$ alors $s := s - 1$ (*await* s *do* $s := s - 1$), sinon l'exécution du processus ayant appelé *wait*(s) est suspendue.
- *signal*(s) : si un processus a été suspendu lors d'une exécution antérieure d'un *wait*(s) alors le réveiller, sinon $s := s + 1$.

s correspond au nombre de ressources d'un type donné.

Sémaphores (2)

remarques:

- Un sémaphore ne prenant que les valeurs 0 ou 1 est appelé *sémaphore binaire*.
- Les primitives *wait*(s) et *signal*(s) s'excluent mutuellement si elles portent sur le même sémaphore (l'ordre n'est donc pas connu).
- La définition de *signal* ne précise pas quel processus est réveillé s'il y en a plusieurs.

Sémaphores (3)

On peut utiliser les sémaphores pour l'exclusion mutuelle. Les deux processus `p 1` et `p 2` sont exécutés en parallèle grâce à la bibliothèque de `threads` d'OCaml.

```
1  let crit () = ...
2  let suite () = ...
3  let s = ref 1;;
4
5  let p i =
6    while true do
7    begin
8      wait(s);
9      crit();
10     signal(s);
11     suite()
12   end
13 ;;
14
15 Thread.create p 1;;
16 Thread.create p 2;;
```

Sémaphores (4)

Dans cet exemple, si un processus veut entrer en section critique, il entrera en section critique si :

- il n'y a que 2 processus (si P_1 est suspendu alors P_2 est en section critique) ;
- si aucun processus ne s'arrête en section critique (si P_2 est dans `crit` alors il exécutera $signal(s)$.

Cette vérification ne fonctionne plus à partir de 3 processus. Il peut y avoir privation si le choix du processus se fait toujours en faveur de certains processus.
Par exemple, si le choix s'effectue toujours en faveur du processus d'indice le plus bas, P_1 et P_2 pourraient se liguer pour se réveiller mutuellement, P_3 étant alors indéfiniment suspendu.

Le dîner des philosophes (1)

Le "dîner des philosophes", du à Dijkstra, illustre les différents pièges du modèle à mémoire partagée.

L'histoire se passe dans un monastère reculé où 5 moines se consacrent exclusivement à la philosophie. Ils passeraient bien tout leur temps à la réflexion s'ils ne devaient manger de temps en temps. La vie d'un philosophe se résume en une boucle infinie : penser - manger. Ils possèdent une table commune ronde. Au centre se trouve un plat de spaguettis qui est tou jours rempli.

Le dîner des philosophes (2)

Il y a 5 assiettes et 5 fourchettes. Le philosophe qui veut manger sort de sa cellule, s'assoit à table, mange et retourne ensuite à ses pensées. Les spaguettis sont si enchevétrés qu'il faut deux fourchettes pour pouvoir les manger. Un philosophe ne peut utiliser que les deux fourchettes autour de son assiette. Les problèmes

posés sont :

- l'interblocage : chaque philosophe tient une fourchette et attend qu'une autre se libère ;
- privation (ou famine) : un philosophe n'arrive jamais à obtenir 2 fourchettes.

Processus légers (Thread) en OCaml

Modèle de parallélisme à mémoire partagée

- contexte d'exécution sur une même machine virtuelle
- même zone mémoire ($\neq$ `fork`)
- ne va pas plus vite (distribution INRIA)
- sert à exprimer des algos concurrents

Bibliothèque OCaml

La bibliothèque sur les threads d'Objective CAML est découpée en cinq modules dont les quatre premiers définissent chacun des types abstraits :

- module Thread : création et exécution de processus légers (type Thread.t) ;
- module Mutex : création, pose et libération de verrous (type Mutex.t) ;
- module Condition : création de conditions (signaux), attente et réveil sur condition (type Condition.t) ;
- module Event : création de canaux de communication (type 'a Event.channel), des valeurs y transitant (type 'a Event.event), et des fonctions de communication.
- module ThreadUnix : fonctions d'entrées-sorties du module Unix non bloquantes. (déprécié)

Compilation avec threads

- basés sur les threads système (posix 1003) : byte-code et natif
- internes à la MV OCaml : byte-code
- ocamlc ou toplevel

```
1 $ ocamlc -thread -custom threads.cma  fichiers.ml -↩
    cclib -lthreads
2 $ ocamlmktop -thread -custom -o threadtop unix.cma ↩
    threads.cma -cclib -lthreads -cclib -lunix
3 $ ./threadtop -I +threads
```

- ocamlopt (+ threads posix)

```
1 $ ocamlc -thread -custom threads.cma fichiers.ml -↩
    cclib -lthreads \
2   -cclib -lunix -cclib -lpthread
3 $ ocamlmktop -thread -custom threads.cma fichiers.ml -↩
    cclib -lthreads \
4   -cclib -lunix -cclib -lpthread
5 $ ocamlopt -thread threads.cmxa fichiers.ml -cclib -↩
    lthreads \
6   -cclib -lunix -cclib -lpthread
```

Relations entre threads

1. sans relation
2. avec relation mais sans synchronisation
3. relation d'exclusion mutuelle
4. relation d'exclusion mutuelle avec communication

Module Thread

- `create f a` crée le processus de l'application de `f` sur `a` ;
- `self ()` retourne le processus courant et `id t` son identificateur ;
- `exit ()` termine le processus courant et `kill t` le processus indiqué ;
- `join th` suspend l'exécution de l'appelant jusqu'au moment où `th` est terminé ;
- `delay d` suspend l'exécution du thread pendant `d` secondes ;

```
1  # Thread.create;;
2  - : ('a -> 'b) -> 'a -> Thread.t = <fun>
3  # Thread.self;;
4  - : unit -> Thread.t = <fun>
5  # Thread.exit;;
6  - : unit -> unit = <fun>
7  # Thread.join;;
8  - : Thread.t -> unit = <fun>
9  # Thread.delay;;
10 - : float -> unit = <fun>
```

Module Mutex

- `create ()` crée un verrou d'exclusion mutuelle (mutex) ;
- `lock m` capture un "mutex",
- `try_lock m` capture si possible un verrou, retourne `true` si cela est fait et `false` sinon,
- et `unlock` libère un verrou.

```
1  # Mutex.create;;
2  - : unit -> Mutex.t = <fun>
3  # Mutex.lock;;
4  - : Mutex.t -> unit = <fun>
5  # Mutex.try_lock;;
6  - : Mutex.t -> bool = <fun>
7  # Mutex.unlock;;
8  - : Mutex.t -> unit = <fun>
```

Module Condition

- `create ()` : crée une variable condition ;
- `wait c m` : libère `m` et suspend le processus appelant sur la variable condition `c`, `signal c` réveille un des processus suspendus sur la variable condition `c`, et `broadcast c` réveille tous les processus suspendus sur `c`.

```
1 # Condition.wait;;
2 - : Condition.t -> Mutex.t -> unit = <fun>
3 # Condition.signal;;
4 - : Condition.t -> unit = <fun>
5 # Condition.broadcast;;
6 - : Condition.t -> unit = <fun>
```

Exemple 1 : Producteur/Consommateur (1)

```
1  # let f = Queue.create ()  and  m = Mutex.create () ;;
2  val f : '_a Queue.t = <abstr>
3  val m : Mutex.t = <abstr>
4
5  # let produire i p d =
6      incr p ;
7      Thread.delay d ;
8      Printf.printf "Le producteur (%d) a produit %d\n"
9                    i !p ;
10     flush stdout ;;
11 val produire : int -> int ref -> float -> unit = <fun>
12
13 # let stocker i p =
14     Mutex.lock m ;
15     Queue.add (i,!p) f ;
16     Printf.printf "Le producteur (%d) a ajout\'e son %d-i\'↩
           eme produit\n" i !p ;
17     flush stdout ;
18     Mutex.unlock m ;;
19 val stocker : int -> int ref -> unit = <fun>
```

Exemple 1 : Producteur/Consommateur (2)

```
# let producteur i =
    let p = ref 0 and d = Random.float 2.
    in while true do
         produire i p d ;
         stocker i p ;
         Thread.delay (Random.float 2.5)
       done ;;
val producteur : int -> unit = <fun>
```

Exemple 1 : Producteur/Consommateur (3)

```
# let consommateur i =
    while true do
     Mutex.lock m ;
     ( try
        let ip, p = Queue.take f
        in Printf.printf "Le consommateur(%d) " i ;
         Printf.printf "a retir\'e le produit (%d,%d)\n"
                       ip p ;
         flush stdout ;
       with
        Queue.Empty ->
         Printf.printf "Le consommateur(%d) " i ;
         Printf.printf "est reparti les mains vides\n" ) ;
     Mutex.unlock m ; Thread.delay (Random.float 2.5)
    done ;;
val consommateur : int -> unit = <fun>
```

Exemple 1 : Producteur/Consommateur (4)

Le programme de test suivant crée quatre producteurs et quatre consommateurs.

```
1 for i = 0 to 3 do
2   ignore (Thread.create producteur i);
3   ignore (Thread.create consommateur i)
4 done ;
5 while true do Thread.delay 5. done ;;
```

Exemple 2 : Producteur/Consommateur (1)

```
# let c = Condition.create () ;;
val c : Condition.t = <abstr>
```

On modifie la fonction de stockage du producteur en lui ajoutant l'émission d'un signal.

```
# let stocker2 i p =
    Mutex.lock m ;
    Queue.add (i,!p) f ;
    Printf.printf "Le producteur (%d) a ajout\'e son %d-↩
        ieme produit\n" i !p ;
    flush stdout ;
    Condition.signal c ;
    Mutex.unlock m ;;
val stocker2 : int -> int ref -> unit = <fun>
# let producteur2 i =
    let p = ref 0 in
    let d = Random.float 2.
    in while true do
         produire i p d;   stocker2 i p;
         Thread.delay (Random.float 2.5)
       done ;;
val producteur2 : int -> unit = <fun>
```

Exemple 2 : Producteur/Consommateur (2)

L'activité du consommateur se fait alors en deux temps :

- attendre qu'un produit soit disponible
- puis emporter le produit.

Le verrou est pris quand l'attente prend fin et il est rendu dès que le consommateur a emporté son produit. L'attente se fait sur la variable c.

```
# let attendre2 i =
    Mutex.lock m ;
    while Queue.length f = 0 do
      Printf.printf "Le consommateur (%d) attend\n" i ;
      Condition.wait c m
    done ;;
val attendre2 : int -> unit = <fun>
```

Exemple 2 : Producteur/Consommateur (3)

```
# let emporter2 i =
    let ip, p = Queue.take f in
      Printf.printf "Le consommateur (%d) " i ;
      Printf.printf "emporte le produit (%d, %d)\n" ip p ;
      flush stdout ;
      Mutex.unlock m ;;
val emporter2 : int -> unit = <fun>
# let consommateur2 i =
    while true do
      attendre2 i;
      emporter2 i;
      Thread.delay (Random.float 2.5)
  done ;;
val consommateur2 : int -> unit = <fun>
```

Notons qu'il n'est plus besoin de vérifier l'existence effective d'un produit puisqu'un consommateur doit attendre l'existence d'un produit dans la file d'attente pour que sa suspension prenne fin. Vu que la fin de son attente correspond à la prise du verrou, il ne court pas le risque de se faire dérober le nouveau produit avant de l'avoir emporté.

Extension pour le multi-cœur

Expérimentations:

- version 3.10.2
- pour AMD/Intel 64-bits
- nouveau runtime :
  - nouveau GC
  - nouvelle implantation de la bibliothèque de threads
- `http://www.algo-prog.info/ocmc/web/`

Modèle à mémoire distincte

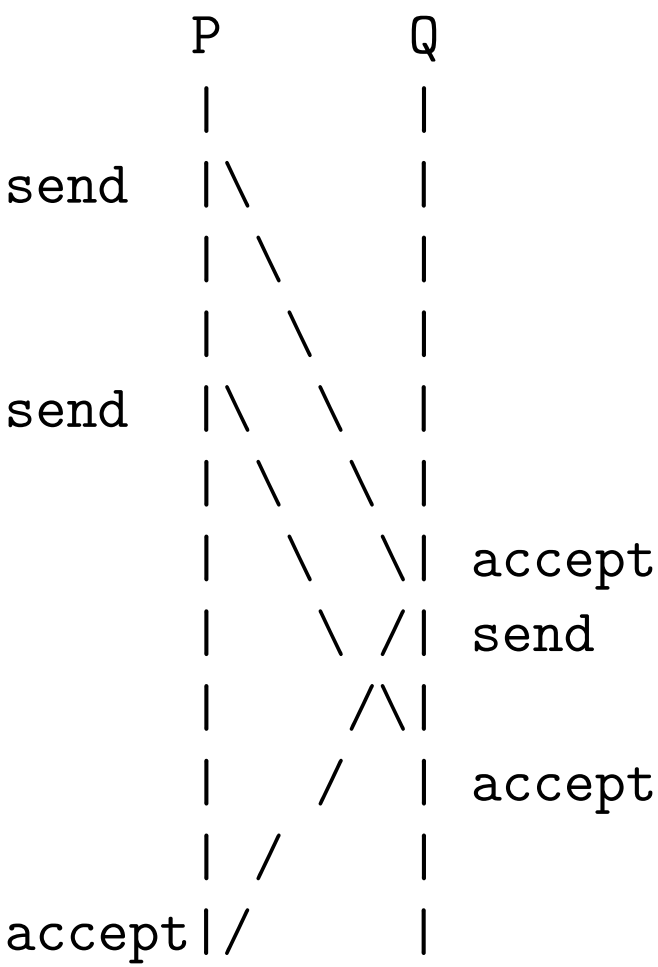
modèle à communication de messages (message passing)
2 primitives :

- “envoi un message” :
- “accepte un message”

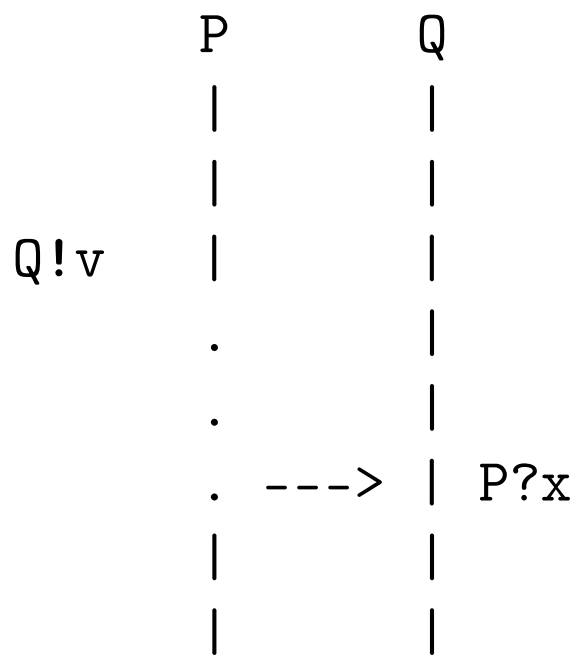
Caractéristiques

- envoi bloquant ou non
- réception bloquante ou non (*polling*)

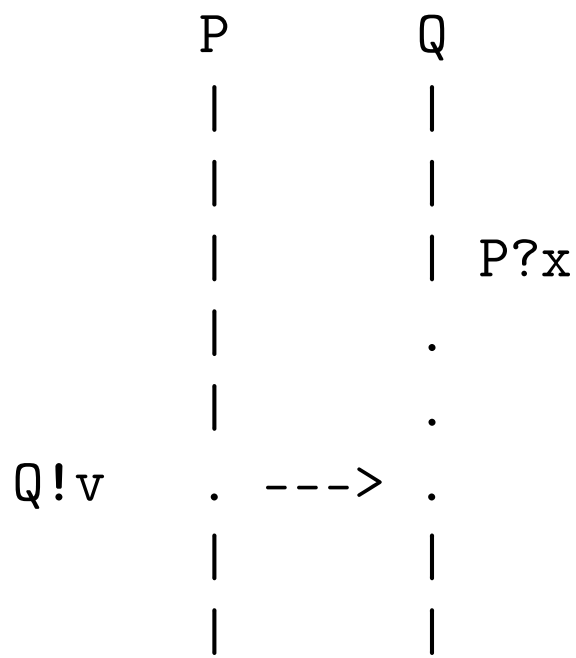
Communications asynchrones



Communications synchrones



Communications synchrones



Module Event - OCAML

- communication synchrone
- canaux fortement typés
- si synchronisation, réception bloquante ou non (*poll*)

event.mli

```
1  type 'a channel
2  val new_channel: unit -> 'a channel
3  type 'a event
4  val send: 'a channel -> 'a -> unit event
5  val receive: 'a channel -> 'a event
6  val always: 'a -> 'a event
7  val choose: 'a event list -> 'a event
8  val wrap: 'a event -> f:('a -> 'b) -> 'b event
9  val guard: (unit -> 'a event) -> 'a event
10 val sync: 'a event -> 'a
11 val select: 'a event list -> 'a
12 val poll: 'a event -> 'a option
```

Evénements, canaux et communication

- 2 types abstraits : `'a channel` et `'a event`
- `new_channel : unit -> 'a channel` : création d'un canal
- `send : 'a channel -> 'a -> unit event` : envoi une valeur `v` de type `'a` sur un canal `c` de type `'a channel`, retourne un événement dont la valeur est de type `unit` (valeur `()`)
- `receive : 'a channel -> 'a event`, retourne un événement de la valeur transmise.

`send` et `receive` ne sont pas bloquantes ! ! !

Synchronisation

- `sync : 'a event -> 'a` : fonction principale de synchronisation

  transforme un événement lié à une valeur en cette valeur.

Exemple 1 : partage de référence

```
1  let ch = Event.new_channel () ;;
2  let v = ref 0;;
3
4  let reader () = Event.sync (Event.receive ch);;
5  let writer () = Event.sync (Event.send ch ("S" ^ (↩
       string_of_int !v)));;
6
7  let loop_reader s d () =
8    for i=1 to 10 do
9      let r = reader() in
10       print_string (s ^ " " ^ r); print_newline();
11     Thread.delay  d
12   done ;;
13
14 let loop_writer d () =
15    for i=1 to 10 do incr v; writer(); Thread.delay d
16    done ;;
17
18 Thread.create (loop_reader "A" 1.1) ();;
19 Thread.create (loop_reader "B" 1.5) ();;
20 Thread.create (loop_reader "C" 1.9) ();;
21 Thread.delay 2.0;;
22 loop_writer 1. ();;
```

Exemple 1 : trace

```
% ocamlc -thread unix.cma threads.cma es1.ml
```

```
% ./a.out
C S1
A S2
B S3
C S4
A S5
B S6
C S7
A S8
B S9
C S10
```

```
% ./a.out
B S1
A S2
C S3
B S4
A S5
C S6
B S7
A S8
C S9
B S10
```

Exemple 2 : gensym (sans synchro)

```
1  type uid = UID of string Event.channel;;
2
3  let makeUidSrc () =
4    let ch = Event.new_channel () in
5    let rec loop i = begin
6       Event.send ch ("S"^(string_of_int i));
7       loop (i+1)
8    end in
9      Thread.create (fun () -> loop 0) () ;
10     UID ch
11 ;;
12
13 let getUid (UID ch) = Event.receive ch;;
```

Exemple 2 : gensym (avec synchro)

```
1  type uid = UID of string Event.channel;;
2
3  let makeUidSrc () =
4    let ch = Event.new_channel () in
5    let rec loop i = begin
6       Event.sync (Event.send ch ("S"^(string_of_int i)));
7       loop (i+1)
8    end in
9      Thread.create (fun () -> loop 0) () ;
10     UID ch
11 ;;
12
13 let getUid (UID ch) = Event.sync(Event.receive ch);;
```

Programme principal

```
1  let ch1 = makeUidSrc ();;
2
3  let main ti msg  () =
4    while (true) do
5      Thread.delay(ti);
6      let r = getUid ch1 in
7      print_string (msg); print_string " -- ";
8      print_string r;  print_newline();
9   done;;
10
11 Thread.create (main 1.1 "A") ();;
12
13 main 2.1 "B"  ();;
```

Trace

```
A -- S0
Src0
B -- S1
Src1
A -- S2
Src2
A -- S3
Src3
B -- S4
Src4
A -- S5
Src5
A -- S6
Src6
B -- S7
Src7
```

Polling

- `'a event -> 'a option` : version non bloquante de `sync`

  retourne `Some v` si un événement est présent, sinon `None`

Autres fonctions sur les événements

- `always : 'a -> 'a event` : crée un événement toujours prêt pour la synchronisation ;
- `wrap : 'a event -> ('a -> 'b) -> 'b event` applique une fonction sur la valeur de l'événement (fonction de post-processing)
- `wrap_abort : 'a event -> (unit -> unit) -> 'a event` applique la fonction en cas de non sélection de l'événement

Choix d'un événement dans une liste

- `choose : 'a event list -> 'a event`
- `select : 'a event list -> 'a`

```
1       let select x = sync(choose x);;
```

Exemple : accumulateur +/-

3 canaux : addCh, SubCh et readCh :

```
1  let rec accum sum =
2     print_int sum; print_newline();
3     Event.sync  (
4       Event.choose [
5           wrap (receive addCh) (fun x -> accum(sum + x));
6           wrap (receive subCh) ( fun x -> accum(sum - x));
7           wrap (send  readCh sum)  ( fun x -> accum(sum))
8        ]
9     );;
```

wrap associe des actions aux communications ! ! !

Requêtes

```
1 let clientCallEvt x =
2   wrap (send reqCh x) (fun () -> receive replyCh);;
```

Mémoire partagée synchronisée (1)

M-variable :

- une M-variable est soit vide, soit pleine
- opération `take` : prendre la valeur d'une M-variable si elle est pleine, bloquante sinon
- opération `put` : remplit une M-variable, provoque une erreur si elle est pleine

Interface

```
1 type 'a mvar
2 val mVar : unit -> 'a mvar
3 exception Put
4 val mTake : 'a mvar -> 'a Event.event
5 val mPut : 'a mvar -> 'a -> unit
```

Une M-variable est construite dans un état vide.

Mémoire partagée synchronisée (2)

```
1  type 'a mvar = MV of ('a Event.channel * 'a Event.channel
2                        * bool Event.channel);;
3
4  let mVar () =
5    let takeCh = Event.new_channel ()
6    and putCh = Event.new_channel ()
7    and ackCh = Event.new_channel () in
8    let rec empty () =
9      let x = Event.sync (Event.receive putCh) in
10       Event.sync (Event.send ackCh true);
11       full x
12   and full x =
13     Event.select
14       [Event.wrap (Event.send takeCh x) empty ;
15        Event.wrap (Event.receive putCh)
16                   (fun _ -> (Event.sync (Event.send ackCh
                                false); full x))]
17   in
18     ignore (Thread.create empty ());
19     MV (takeCh, putCh, ackCh) ;;
```

Mémoire partagée synchronisée (3)

```
1
2  let mTake ( mv : 'a mvar) = match mv with
3     MV (takechannel, _, _ ) -> Event.receive takechannel ;;
4
5  exception Put;;
6  let mPut mv x = match mv with
7    MV (takechannel, putchannel, ackchannel) ->
8       Event.sync (Event.send putchannel x);
9       if (Event.sync( Event.receive ackchannel)) then ()
10      else raise Put ;;
```

Internet

- réseau de réseaux
- 2 protocoles :
  - IPv4
  - et IPv6

Machines

- les passerelles

  opèrent le passage d'un réseau à un autre ;

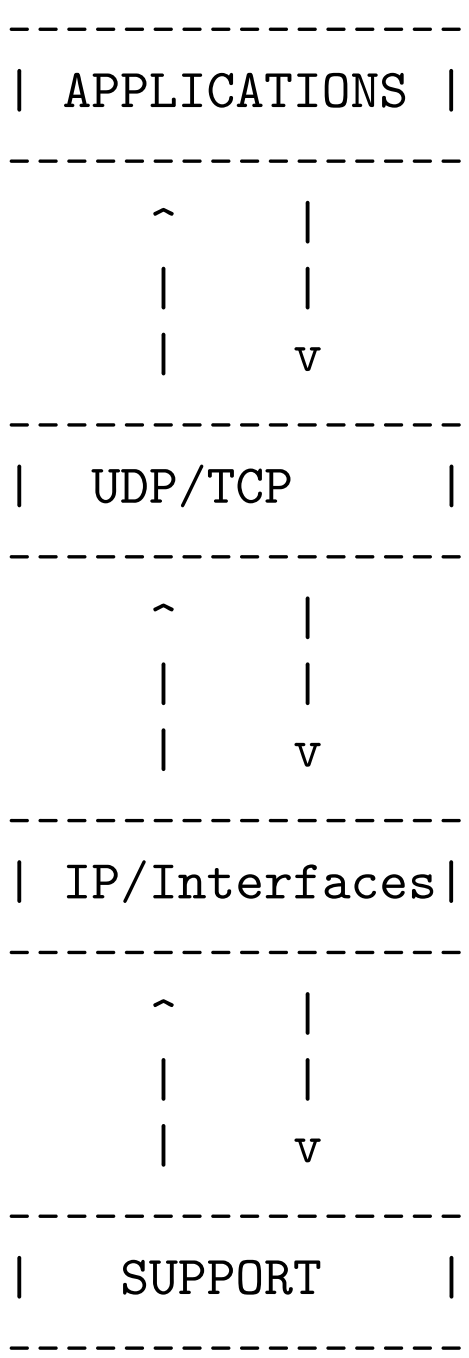
- les routeurs

  connaissent, en partie, la topologie d'Internet et opèrent l'acheminement des données ;

- les serveurs de noms

  connaissent la correspondance entre noms de machines et adresses réseau.

Couches



IP

- unité de transfert : datagramme (paquet)
  contient un entête et des données
- entête : contient les adresses du destinataire et du receveur
- non fiable : ni le bon ordre, ni le bon port, ni la non duplication des paquets transmis
- routage correct et signalisation d'erreur (si un paquet n'a pas pu arriver à destination)

UDP

User Datagram Protocol

- sans connexion non fiable

à la manière d'IP pour les interfaces

TCP

Transfert Control Protocol

- mode connecté et fiable
- gestion d'acquitement
- gestion de la retransmission
- gestion de l'ordonnacement des paquets

Optimisation de la transmission par des techniques de fenetrage

Services standard en mode client-serveur

- FTP (File Transfert Protocol)
- TELNET (Terminal Transfert Protocol)
- SMTP (Simple Mail Transfert Protocol)
- HTTP (Hyper Text Transfert Protocol)

D'autres services utilisent ce modèle client-serveur :

- NFS (Network File System)
- X-Window
- les services UNIX : rlogin, rwho ...

Communication par *sockets*

Module Unix (OCaml) - adressage IP

type abstrait : *inet_addr* pour les adresses IP

```
1 Unix.inet_addr_of_string ;;
2 Unix.string_of_inet_addr ;;
3 Unix.string_of_inet_addr (Unix.inet_addr_of_string "↩
      132.227.89.02") ;;
```

La structure des entrées de la base d'adresse est représentée par :

```
1 type host_entry =
2   { h_name : string;
3     h_aliases : string array;
4     h_addrtype : socket_domain;
5     h_addr_list : inet_addr array } ;;
```

Les deux premiers champs contiennent le nom de la machine et ses alias, le troisième, le type d'adresse et le dernier, la liste des adresses des interfaces de la machine.

nom ou adresse d'une interface

On obtient le nom de sa machine par la fonction :

```
1 Unix.gethostname ;;
2 let my_name = Unix.gethostname() ;;
```

Les fonctions d'interrogation de la base d'adresses nécessitent en entrée soit le nom, soit l'adresse de la machine.

```
1 Unix.gethostbyname ;;
2 Unix.gethostbyaddr ;;
3 let my_entrie_byname = Unix.gethostbyname my_name ;;
4 let my_addr = my_entrie_byname.Unix.h_addr_list.(0) ;;
5 let my_entrie_byaddr = Unix.gethostbyaddr my_addr ;;
6 let my_full_name = my_entrie_byaddr.Unix.h_name ;;
```

Ces fonctions déclenchent l'exception Not_found en cas d'échec de la requête.

base de services

La base de services contient la correspondance entre noms de service et numéros de port. La plupart des services standards d'Internet sont standardisés. La structure des entrées de la base de services est :

```
type service_entry =
  { s_name : string;
    s_aliases : string array;
    s_port : int;
    s_proto : string } ;;
```

Les premiers champs sont le nom du service et ses éventuels alias, le troisième contient le numéro de port du service et le dernier, le nom du protocole utilisé. Un service est en fait caractérisé par son numéro de port et son protocole. Les fonction d'interrogation sont :

```
Unix.getservbyname ;;
Unix.getservbyport ;;
Unix.getservbyport 80 "tcp" ;;
Unix.getservbyname "ftp" "tcp" ;;
```

Prises de communication

La métaphore classique est de comparer les sockets aux postes téléphoniques.

- Pour fonctionner ils doivent être raccordés au réseau (*socket).*
- *Pour recevoir un appel ils doivent posséder un numéro de type* sock_addr (*bind).*
- *Pendant un appel, il est possible de recevoir un autre appel si la configuration le permet (*listen).
- Il n'est pas nécessaire d'avoir soi-même un numéro pour appeler un autre poste (*connect) ; une fois la connexion établie la communication est dans les deux sens.*

Domaines

Domaine
Suivant qu'une prise est destinée à la communication interne ou externe elles appartiennent à un *domaine* différent. La bibliothèque Unix définit deux domaines possibles correspondant aux constructeurs du type :

```
1 type socket_domain = PF_UNIX | PF_INET;;
```

Le premier domaine correspond à la communication locale et le second, à la communication transitant sur le réseau Internet. Ce sont là les principaux domaines d'application des sockets.

Types et protocoles

Quelque soit leur domaine, les sockets définissent certaines propriétés de communications (fiabilité, ordonnancement, etc.) représentées par les constructeurs du type :

```
1 type socket_type = SOCK_STREAM | SOCK_DGRAM
2                  | SOCK_SEQPACKET | SOCK_RAW ;;
```

Suivant le type de socket utilisé, le protocole sous-jacent à la communication obéira aux caractéristiques définies. À chaque type de communication est associé un protocole par défaut.
En fait, nous n'utiliserons ici que le premier type de communication : SOCK_STREAM avec le protocole par défaut TCP. Il garantit fiabilité, ordonnancement et non duplication des messages échangés et fonctionne en mode connecté.

Création

La fonction de création d'une socket est :

```
1 Unix.socket ;;
2 - : Unix.socket_domain ->
3     Unix.socket_type -> int -> Unix.file_descr = <fun>
```

Le troisième argument (de type int) permet de préciser le protocole associé à la communication. La valeur 0 est interprétée comme « le protocole par défaut » associé au couple (domaine,type), arguments de la création de la socket. La valeur de retour de cette fonction est un descripteur de fichier. Ainsi les échanges pourront se faire en utilisant les fonctions standards, du module Unix, d'entrées-sorties.

Créons une socket TCP/IP :

```
1 let s_descr = Unix.socket Unix.PF_INET
2                           Unix.SOCK_STREAM 0 ;;
```

Adresse et connexion

Une socket que l'on vient de créer ne possède pas d'adresse. Pour qu'une connexion puisse s'établir entre deux sockets, l'appelant doit connaître l'adresse du récepteur.

L'adresse d'une socket (TCP/IP) est constituée d'une adresse IP et d'un numéro de port. correspondant au type suivant :

```
1 type sockaddr =
2   ADDR_UNIX of string | ADDR_INET of inet_addr * int ;;
```

L'entier qui fait partie de l'adresse des sockets du domaine Internet correspond au numéro de port.

Attribution d'une adresse

La première chose à faire, pour pouvoir recevoir des appels, après la création d'une socket est de lui attribuer, de la *lier* à, une adresse. C'est le rôle de la fonction :

```
#Unix.bind ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

En effet, nous avons déjà un descripteur de socket, mais l'adresse qui lui est attachée à la création ne peut guère nous être utile comme le montre l'exemple suivant :

```
let (addr_in, p_num) =
  match Unix.getsockname s_descr with
    Unix.ADDR_INET (a,n) -> (a,n)
  |  _ -> failwith "pas INET";;
val addr_in : Unix.inet_addr = <abstr>
val p_num : int = 0
Unix.string_of_inet_addr addr_in;;
- : string = "0.0.0.0"
```

Capacité d'écoute/réception (1)

Il faut encore procéder à deux opérations avant que notre socket soit complètement opérationnelle pour recevoir des appels : définir sa capacité d'écoute et se mettre effectivement à l'écoute. C'est le rôle respectif de :

```
1 # Unix.listen;;
2 - : Unix.file_descr -> int -> unit = <fun>
```

Le second argument (de type int) de la fonction listen indique le nombre maximal de connexions en attente toléré.

Capacité d'écoute/réception (2)

et de :

```
# Unix.accept;;
- : Unix.file_descr -> Unix.file_descr * Unix.sockaddr =
   <fun>
```

L'appel de la fonction accept attend une demande de connexion. Quand celle-ci survient la fonction accept se termine et retourne l'adresse de la socket ayant appelée ainsi qu'un nouveau descripteur de socket, (dite socket de service). Cette socket de service est automatiquement liée à une adresse. La fonction accept ne s'applique qu'aux sockets ayant exécuté un listen, c'est-à-dire aux sockets ayant paramètrées la file d'attente des demandes de connexion.

demande de connexion

La fonction duale de accept est ;

```
1 # Unix.connect;;
2 - : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

Un appel à `Unix.connect s_descr s_addr` établit un connexion entre la socket locale `s_descr` (qui est automatiquement liée) et la socket d'adresse `s_addr`.

Modèle client-serveur

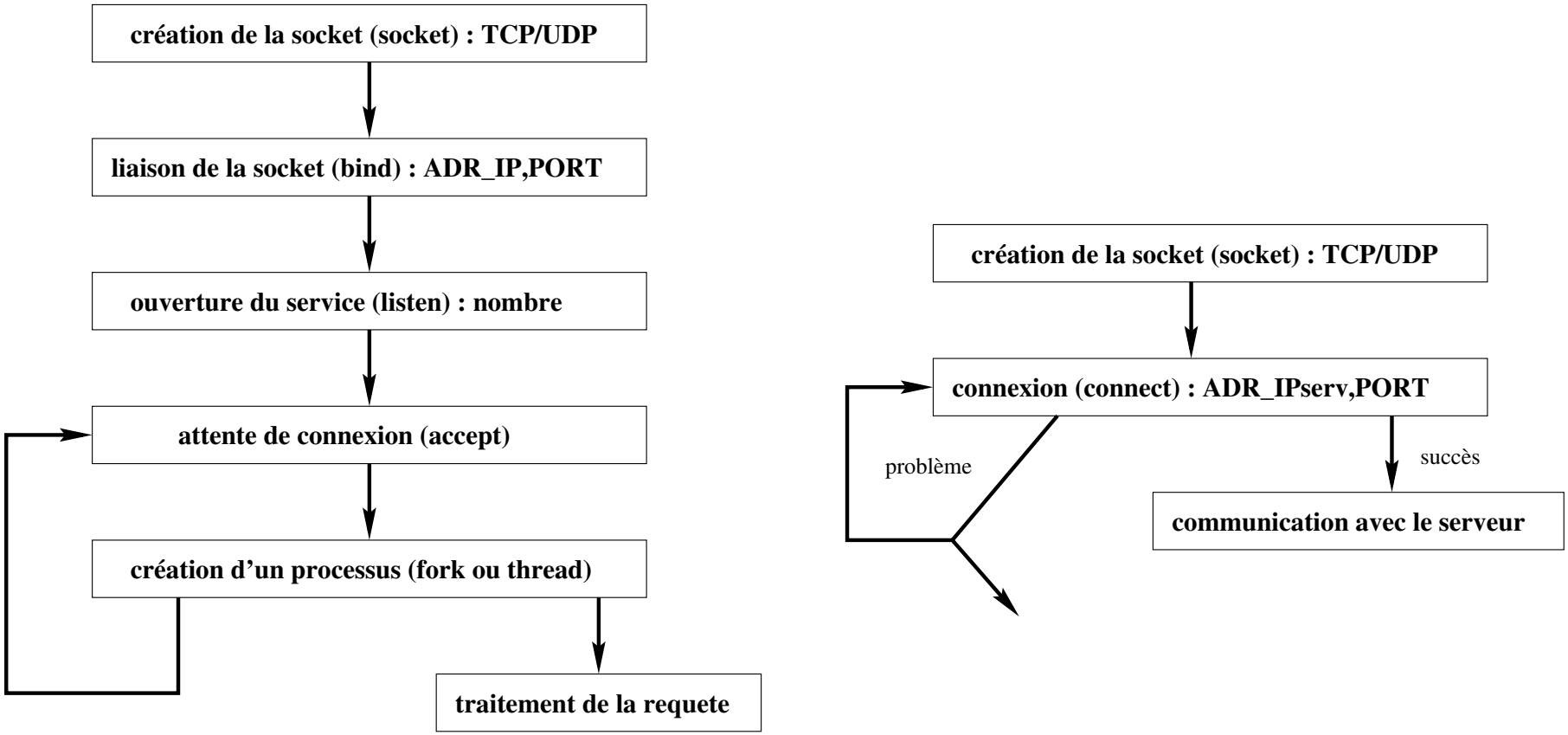
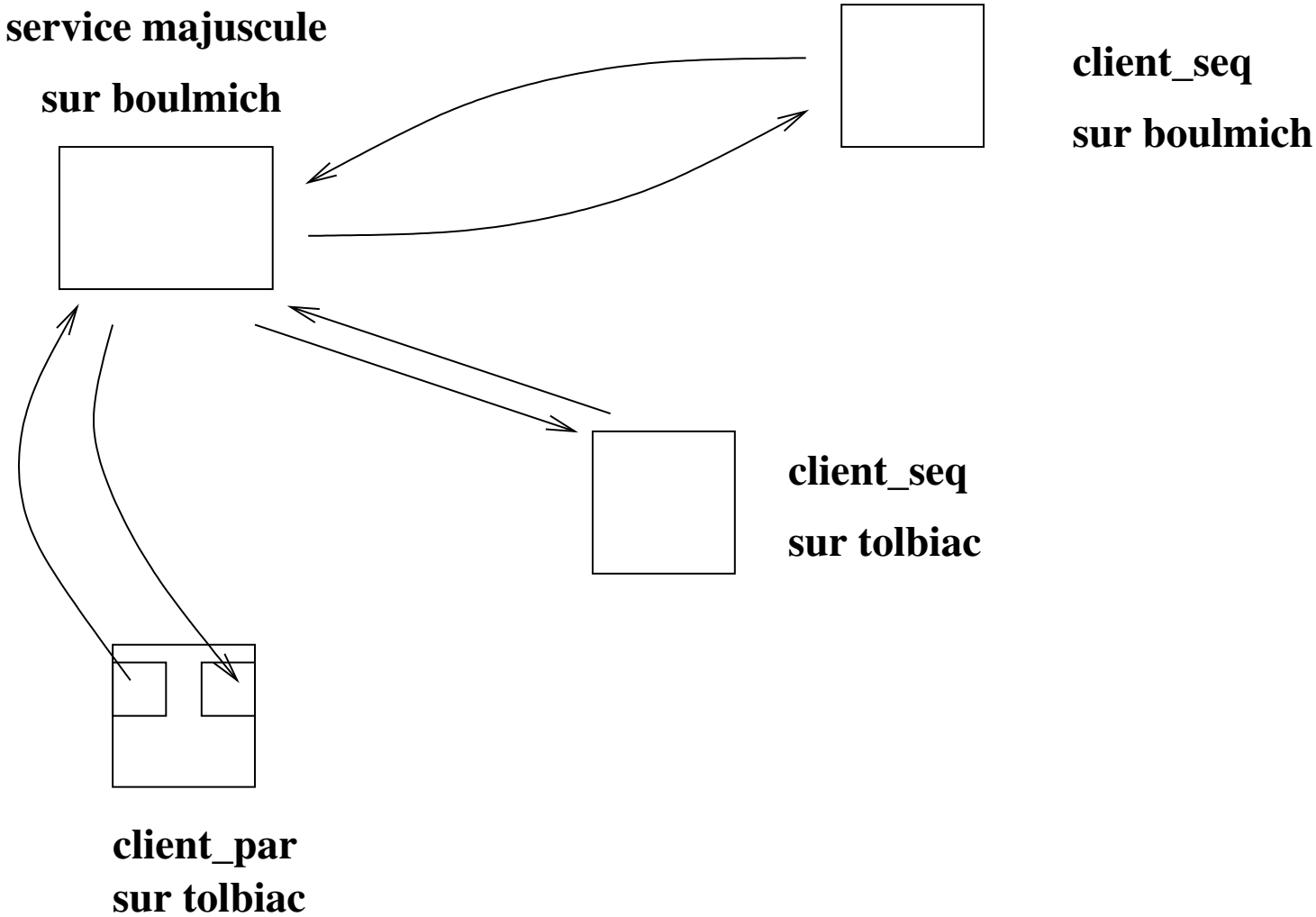


Figure : **Schéma des actions d'un serveur et d'un client**

Programmation d'un client-serveur



Service MAJUSCULE et des clients : certains tournent sur la même machine que le serveur et d'autres se trouvent sur des machines distantes.

server.ml (1)

```
1  (*  ocamlc -o serv.exe -thread -custom unix.cma threads.↩
       cma server.ml
2              -cclib -lthreads -cclib -lunix *)
3  class virtual  server  port n =
4  object (s)
5    val port_num = port
6    val nb_pending = n
7    val sock = ThreadUnix.socket Unix.PF_INET Unix.↩
         SOCK_STREAM 0
8    method start () =
9      let host = Unix.gethostbyname (Unix.gethostname()) in
10     let h_addr = host.Unix.h_addr_list.(0) in
11     let sock_addr = Unix.ADDR_INET(h_addr, port_num) in
12       Unix.bind sock sock_addr ;
13       Unix.listen sock nb_pending ;
14       while true do
15         let (service_sock, client_sock_addr) =
16             ThreadUnix.accept sock
17         in   s#treat service_sock client_sock_addr
18       done
19   method virtual treat : Unix.file_descr -> Unix.sockaddr ↩
         -> unit
20 end ;;
```

server.ml (2)

```
1  let gen_num = let c = ref 0 in (fun () -> incr c; !c) ;;
2  class virtual connexion sd (sa : Unix.sockaddr) b =
3  object (self)
4    val s_descr = sd
5    val s_addr = sa
6    val mutable numero = 0
7    val mutable debug = b
8    method set_debug b = debug <- b
9    initializer
10     numero <- gen_num();
11     if debug then (
12       Printf.printf "TRACE.connexion : objet traitant %d ↩
             cree\n" numero ;
13       print_newline())
14   method start () =  Thread.create (fun x -> self#run x ; ↩
         self#stop x) ()
15   method stop() =
16     if debug then (
17       Printf.printf "TRACE.connexion : fin objet traitant ↩
             %d\n" numero ;
18       print_newline () );
19       Unix.close s_descr
20   method virtual run : unit -> unit
21 end;;
```

server.ml (3)

```
1  exception Fin ;;
2  let my_input_line  fd =
3    let s = " "  and  r = ref "" in
4      while (ThreadUnix.read fd s 0 1 > 0) && s.[0] <> '\n'
5        do r := !r ^s done ;
6      !r ;;
7  class connexion_maj sd sa b =
8  object(self)
9    inherit connexion sd sa b
10   method run () =
11       try
12         while true do
13           let ligne =  my_input_line s_descr
14           in if (ligne = "") or (ligne = "\013") then ↩
                 raise Fin ;
15             let result = (String.uppercase ligne)^"\n"
16             in ignore (ThreadUnix.write s_descr result 0
17                                (String.length result))
18         done
19       with
20          Fin  -> ()
21     | exn  -> print_string (Printexc.to_string exn) ; ↩
         print_newline()
22 end ;;
```

server.ml (4)

```
1  class server_maj port n =
2  object(s)
3    inherit server port n
4    method treat s sa  =
5     ignore( (new connexion_maj s sa true)#start())
6  end;;
7
8  (*
9  val main : unit -> unit
10 *)
11
12 let main () =
13   if Array.length Sys.argv < 3
14   then Printf.printf "usage :  server port num\n"
15   else
16     let port = int_of_string(Sys.argv.(1))
17     and n = int_of_string(Sys.argv.(2)) in
18       (new server_maj port n )#start();;
19
20 main();;
```

client.ml (1)

```
1  (*    ocamlc -o client.exe -thread -custom unix.cma ↩
       threads.cma client.ml
2            -cclib -lthreads -cclib -lunix *)
3
4  class virtual client serv p =
5  object(s)
6    val sock = ThreadUnix.socket Unix.PF_INET Unix.↩
         SOCK_STREAM 0
7    val port_num = p
8    val server = serv
9
10   method start () =
11     let host = Unix.gethostbyname server in
12     let h_addr =  host.Unix.h_addr_list.(0) in
13     let sock_addr = Unix.ADDR_INET(h_addr,port_num) in
14       Unix.connect sock sock_addr;
15       s#treat sock sock_addr;
16       Unix.close sock
17   method virtual treat : Unix.file_descr -> Unix.sockaddr ↩
         -> unit
18 end;;
```

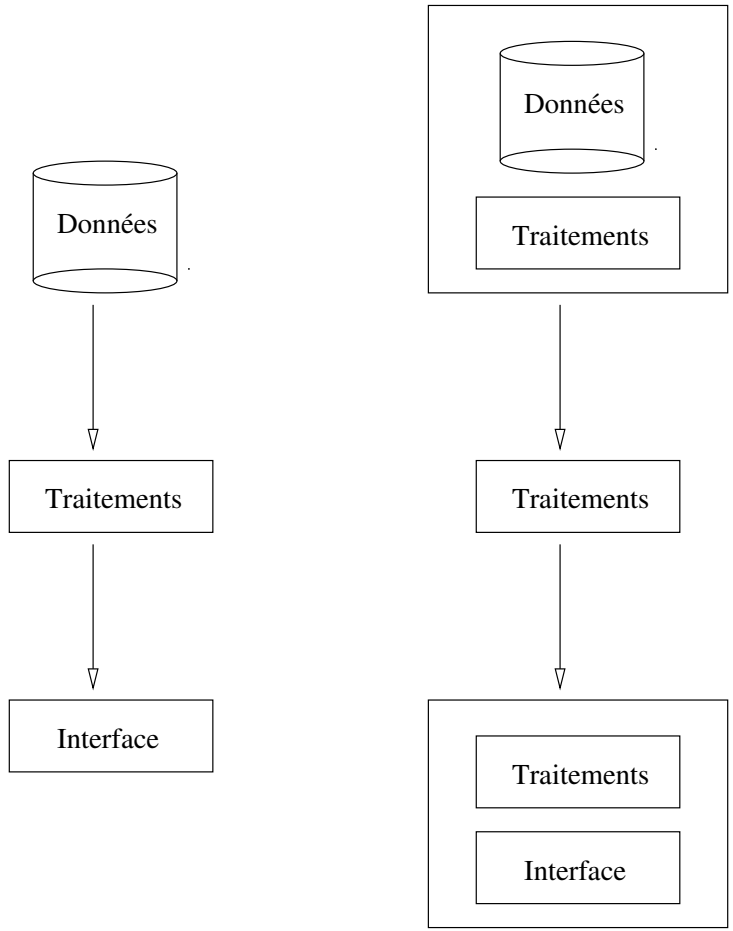
client.ml (2)

```
1
2  class client_maj s p =
3  object
4    inherit client s p
5    method treat s sa =
6      try
7        while true do
8          let si = (my_input_line Unix.stdin) ^ "\n"  in
9           ignore (ThreadUnix.write s si 0 (String.length si↩
               ));
10          let so = (my_input_line s) in
11          if so = "" then raise Fin
12          else (Printf.printf "%s\n" so; flush stdout)
13       done
14     with Fin -> ()
15 end;;
```

client.ml (3)

```
1
2  let main () =
3    if Array.length Sys.argv < 3
4    then Printf.printf "usage :  client server port\n"
5    else
6      let port = int_of_string(Sys.argv.(2))
7      and s = (Sys.argv.(1)) in
8        (new client_maj s  port  )#start();;
9
10 main();;
```

client-serveur à plusieurs niveaux



Différentes architectures de clients-serveurs :
Chaque niveau peut être implanté sur des machines différentes.
L'interface utilisateur s'exécute sur les machines des utilisateurs de l'application.

communications

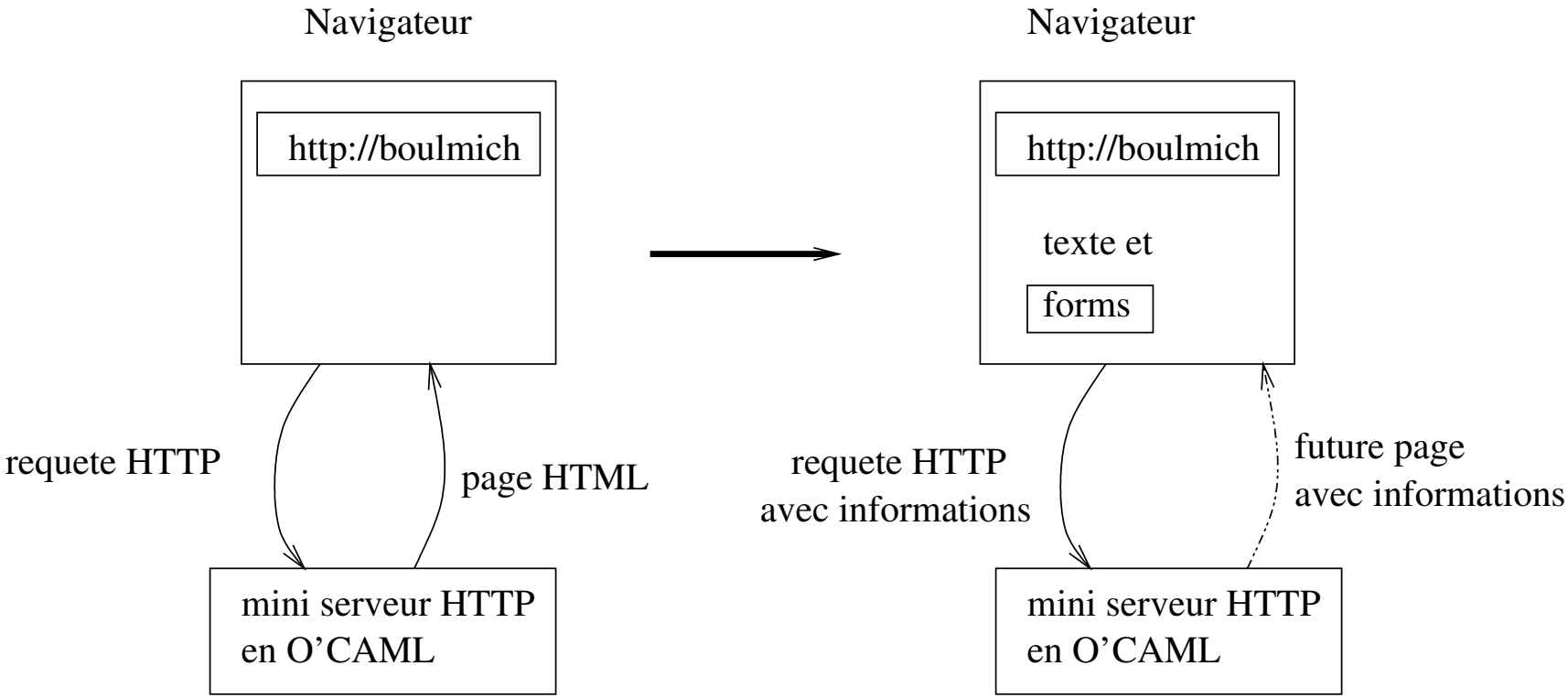


Figure : Communication entre un navigateur et un serveur OCaml