Concurrence légère en OCaml : muthreads

Christophe Deleuze

LCIS – Laboratoire de Conception et d'Intégration des Systèmes
Grenoble-INP, Valence

JFLA 2013 – février 2013

Concurrence légère

- concurrence
  - plusieurs “fils d’exécution”
  - indépendance temporelle
- threads
  - système/plateforme
  - (relativement) coûteux
- événements
  - application
  - léger
  - ordonnancement coopératif
  - casse le flot de contrôle

Style trampoline

une opération bloquante est une fonction qui prend en paramètre sa *continuation*

```
let yield k = ...
let sleep d k = ...

let t () =
   print_string "je passe mon tour"; yield (fun () ->
   print_string "je dors un peu"; sleep 2. (fun () ->
   print_string "je me réveille"; ...))
```

Style trampoline

une opération bloquante est une fonction qui prend en paramètre sa *continuation*

```
let yield k = ...
let sleep d k = ...

let t () =
   print_string "je passe mon tour"; yield (fun () ->
   print_string "je dors un peu"; sleep 2. (fun () ->
   print_string "je me réveille"; ...))
```

Style trampoline

une opération bloquante est une fonction qui prend en paramètre sa *continuation*

```
let yield k = ...
let sleep d k = ...

let t () =
   print_string "je passe mon tour"; yield (fun () ->
   print_string "je dors un peu"; sleep 2. (fun () ->
   print_string "je me réveille"; ...))
```

Style trampoline

une opération bloquante est une fonction qui prend en paramètre sa *continuation*

```
let yield k = ...
let sleep d k = ...

let t () =
   print_string "je passe mon tour"; yield (fun () ->
   print_string "je dors un peu"; sleep 2. (fun () ->
   print_string "je me réveille"; ...))
```

- `>>=` “mise en séquence”
- `let (>>=) f k = f k`

Style trampoline

une opération bloquante est une fonction qui prend en paramètre sa *continuation*

```
let yield k = ...
let sleep d k = ...

let t () =
   print_string "je passe mon tour"; yield >>= fun () ->
   print_string "je dors un peu"; sleep 2. >>= fun () ->
   print_string "je me réveille"; ...
```

- `>>=` “mise en séquence”
- `let (>>=) f k = f k`

Muthreads

- bibliothèque OCaml basée sur une boucle d'événements
- style trampoline $\rightarrow$ notion de "thread"
- simple
  - quelques primitives
  - environ 1000 loc
- applications réseau
  - serveur FTP
  - résolveur DNS ("serveur récursif")

Plan

- aperçu des fonctionnalités
- aperçu de la réalisation
- extension au parallélisme
- conclusion/perspectives...

Entrées/sorties

Fonctions bloquantes en style trampoline :
`read`, `write`, `connect`, `accept`

```
let recv skt k =
  let s = String.create 512 in
  read skt s 0 512 >>= fun l ->
  k (parse_cmd s l)
```

Entrées/sorties

Fonctions bloquantes en style trampoline :
`read`, `write`, `connect`, `accept`

```
let recv skt k =
  let s = String.create 512 in
  read skt s 0 512 >>= fun l ->
  k (parse_cmd s l)
```

Entrées/sorties

Fonctions bloquantes en style trampoline :
`read`, `write`, `connect`, `accept`

```
let recv skt k =
  let s = String.create 512 in
  read skt s 0 512 >>= fun l ->
  k (parse_cmd s l)
```

```
...
recv skt >>= fun m -> match m with
| USER user -> ...
| QUIT -> ...
```

MVars

communication/synchronisation entre threads

mvar : variable mutable partagée (vide ou pleine)

- `make_mvar` – crée une mvar vide
- `take_mvar` – retire la valeur de la mvar
- `put_mvar` – place une valeur dans la mvar

MVars

communication/synchronisation entre threads

mvar : variable mutable partagée (vide ou pleine)

- make_mvar – crée une mvar vide
- take_mvar – retire la valeur de la mvar
- put_mvar – place une valeur dans la mvar

```
...
take_mvar click >>= fun (x,y) ->
put_mvar new_planet (new_pos x y) >>= fun () ->
...
```

Temporisation

Temporiser une opération : elle finit à temps ou est interrompue.

```
let recv skt k =
   let s = String.create 512 in
   timeout control_idle
     (read skt s 0 512)                 (* timed operation *)
     (fun () ->                         (* timer expired   *)
       send skt 421 "Timeout: closing connection" >>= fun () ->
       close skt; terminate ())
     (fun l -> k (parse_cmd s l))       (* read completed *)
```

Temporisation

Temporiser une opération : elle finit à temps ou est interrompue.

```
let recv skt k =
   let s = String.create 512 in
   timeout control_idle
     (read skt s 0 512)                        (* timed operation *)
     (fun () ->                                (* timer expired   *)
       send skt 421 "Timeout: closing connection" >>= fun () ->
       close skt; terminate ())
     (fun l -> k (parse_cmd s l))              (* read completed *)
```

Temporisation

Temporiser une opération : elle finit à temps ou est interrompue.

```
let recv skt k =
   let s = String.create 512 in
   timeout control_idle
     (read skt s 0 512)                    (* timed operation *)
     (fun () ->                            (* timer expired   *)
       send skt 421 "Timeout: closing connection" >>= fun () ->
       close skt; terminate ())
     (fun l -> k (parse_cmd s l))          (* read completed *)
```

Temporisation

Temporiser une opération : elle finit à temps ou est interrompue.

```
let recv skt k =
   let s = String.create 512 in
   timeout control_idle
     (read skt s 0 512)                 (* timed operation *)
     (fun () ->                         (* timer expired   *)
       send skt 421 "Timeout: closing connection" >>= fun () ->
       close skt; terminate ())
     (fun l -> k (parse_cmd s l))      (* read completed *)
```

Exceptions

La construction `try ... with` ne protège pas un thread :

```
try
  let f = do_open () in
  do_upload skt f >>= fun r ->
  close_out f; k r
with _ -> k 550     (* do_open may raise an exception *)
```

Exceptions

La construction `try ... with` ne protège pas un thread :

```
try
  let f = do_open () in
  do_upload skt f >>= fun r ->
  close_out f; k r
with _ -> k 550      (* do_open may raise an exception *)
```

utiliser la **fonction** `trywith`

```
trywith
  (fun k -> let f = do_open () in               (* protégée *)
            do_upload skt f >>= k)
  (fun _ k -> k 550)                     (* traite-exception *)
  (fun r -> close_out f; k r)                (* continuation *)
```

Exceptions

La construction `try ... with` ne protège pas un thread :

```
try
  let f = do_open () in
  do_upload skt f >>= fun r ->
  close_out f; k r
with _ -> k 550     (* do_open may raise an exception *)
```

utiliser la **fonction** `trywith`

```
trywith
  (fun k -> let f = do_open () in                (* protégée *)
            do_upload skt f >>= k)
  (fun _ k -> k 550)                   (* traite-exception *)
  (fun r -> close_out f; k r)              (* continuation *)
```

Exceptions

La construction `try ... with` ne protège pas un thread :

```
try
  let f = do_open () in
  do_upload skt f >>= fun r ->
  close_out f; k r
with _ -> k 550     (* do_open may raise an exception *)
```

utiliser la **fonction** `trywith`

```
trywith
  (fun k -> let f = do_open () in               (* protégée *)
            do_upload skt f >>= k)
  (fun _ k -> k 550)                   (* traite-exception *)
  (fun r -> close_out f; k r)              (* continuation *)
```

Exceptions

La construction `try ... with` ne protège pas un thread :

```
try
  let f = do_open () in
  do_upload skt f >>= fun r ->
  close_out f; k r
with _ -> k 550     (* do_open may raise an exception *)
```

utiliser la **fonction** `trywith`

```
trywith
  (fun k -> let f = do_open () in                 (* protégée *)
            do_upload skt f >>= k)
  (fun _ k -> k 550)                     (* traite-exception *)
  (fun r -> close_out f; k r)                (* continuation *)
```

Exceptions

La construction `try ... with` ne protège pas un thread :

```
try
  let f = do_open () in
  do_upload skt f >>= fun r ->
  close_out f; k r
with _ -> k 550      (* do_open may raise an exception *)
```

utiliser la **fonction** `trywith`

```
trywith
  (fun k -> let f = do_open () in
            do_upload skt f >>= k)
  (fun _ k -> k 550)
>>= fun r ->
close_out f; k r
```

Typage

thread = fonction de type `unit -> t`

```
type t
val spawn : (unit -> t) -> unit

type 'a mvar
val take_mvar : 'a mvar -> ('a -> t) -> t
val put_mvar : 'a mvar -> 'a -> (unit -> t) -> t
...
val recv : Unix.file_descr -> (Ftp.cmd -> t) -> t

val (>>=) : (('a -> t) -> t) -> ('a -> t) -> t
val terminate : unit -> t
```

Exécution d'un thread

structures de données

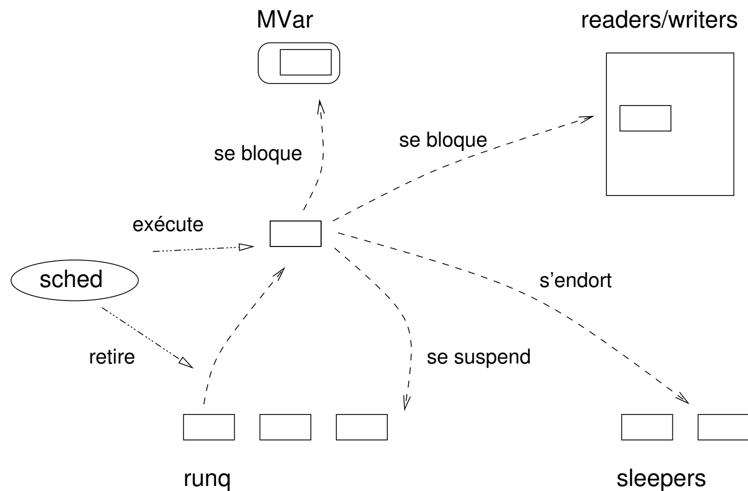
- `runq` file des threads prêts à exécuter
- `sleepers` liste triée des threads en sommeil (`sleep`)
- `readers/writers` tableau[fd] des threads bloqués sur E/S

thread prend en charge sa continuation

```
let yield k = queue_in_runq k; ...
```

sockets en mode non bloquant

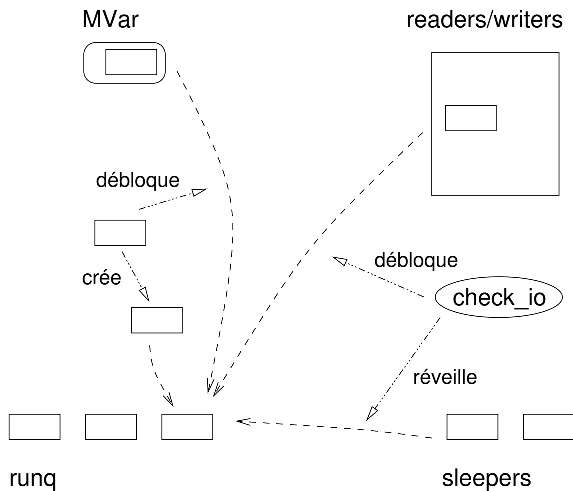
Exécution d'un thread



Réactivation d'un thread

- MVars : par le thread qui effectue l'opération attendue
- thread `check_io` exécuté régulièrement
  - appelle `Unix.select` pour les fd concernés
  - replace débloqués dans `runq`
  - examine `sleepers`

Réactivation d'un thread



Exceptions

La construction `try ... with` ne protège pas un thread :

```
try
  let f = do_open () in
  do_upload skt f >>= fun r ->
  close_out f; k r
with _ -> k 550     (* do_open may raise an exception *)
```

La pile n'est pas vide quand le thread se supend ! *(exception frame)*

→ associer au thread un contexte de récupération des exceptions

thread = continuation + pile de contexte

Exceptions

```
trywith :
(('a->t)->t) -> (exn->('a->t)->t) -> ('a->t) -> t
```

- `trywith f h k` pousse un élément de contexte `TryWith(h,k)` et poursuit l'exécution de `f`.
- `f` se suspend : le contexte est sauvé avec la continuation.
- `f` s'achève : ordonnanceur dépile et lance `k`.
- exception lancée : ordonnanceur "déroule" la pile à la recherche d'un `TryWith(h,k)` avec `h` "acceptant" l'exception.

Exceptions

```
trywith :
(('a->t)->t) -> (exn->('a->t)->t) -> ('a->t) -> t
```

- `trywith f h k` pousse un élément de contexte `TryWith(h,k)` et poursuit l'exécution de `f`.
- `f` se suspend `Susp` : le contexte est sauvé avec la continuation.
- `f` s'achève `Pop` : ordonnanceur dépile et lance `k`.
- exception lancée : ordonnanceur "déroule" la pile à la recherche d'un `TryWith(h,k)` avec `h` "acceptant" l'exception.

```
type t = Done | Susp | Pop
```

Exceptions

```
trywith :
(('a->t)->t) -> (exn->('a->t)->t) -> ('a->t) -> t
```

- `trywith f h k` pousse un élément de contexte `TryWith(h,k)` et poursuit l'exécution de `f`.
- `f` se suspend `Susp` : le contexte est sauvé avec la continuation.
- `f` s'achève `Pop` : ordonnanceur dépile et lance `k`.
- exception lancée : ordonnanceur "déroule" la pile à la recherche d'un `TryWith(h,k)` avec `h` "acceptant" l'exception.

```
type t = Done | Susp | Pop
```

petit problème de typage : type du résultat passé par `f` (ou `h`) à `k`.
→ cacher le `'a` dans une référence partagée

```
 | TryWith of (exn -> t) * (unit -> t)
```

Parallélisme

par partage de MVar entre processus

```
val rspawn : int -> ('a mvar -> t) -> 'a mvar -> unit
```

`rwpawn i thr mv`

- lance le thread `thr` dans le processus `i`
- l'argument `mv` est une MVar qui devient **partagée** avec ce thread

Mise en œuvre

une MVar partagée est

- numérotée et accessible depuis son numéro
- synchronisée par des messages entre processus
- libérée quand le thread "distant" termine

La “concurrence”

autres bibliothèques existantes

- Lwt
- Async

- basées sur la monade de promesse
- développées depuis des années

Conclusion

- bibliothèque légère pour concurrence légère
- formulation trampoline applicable
- importance du typage

Perspectives

- sucre syntaxique
- parallélisme/distribution
  - plus de souplesse
- traitements *"compute bound"*
  - ordonnancement préemptif par capture asynchrone de continuation