Programmation système en OCaml : introduction et cas d'usage.

Carine Morel et Lucas Pluvinage

20 juillet 2022

Tarides

Tarides

- on fait des logiciels et du service en OCaml
- 70 employés (France, UK, USA, Inde etc..)
- tout en open source !

Plan

Programmation système en OCaml

➙ autour de l'écriture d'un "mini" shell :

- ❒ le module Unix
- ❒ les processus Unix
- ❒ et plus

➙ les algos d'exclusion mutuelle en OCaml

OCaml dans la vraie vie

- Mirage : **un** système en OCaml
- QCheck : un vérificateur de propriétés automatiques
- OCaml 5.0
- Écosystème OCaml

Programmation système en OCaml

Introduction : programmation système en OCaml

- Des modules bas niveau (cf `ocaml.org/api`)
  - `Sys` : fonctions communes à Unix et aux autres OS sous lesquels tourne OCaml,
  - `Unix` : fonctions spécifiques à Unix.
- Des sur-couches sur le module `Unix` :
  - certaines fonctions de la `Sdtlib`,
  - `Lwt.Unix`
- Des modules utilitaires comme `Filename`

Introduction : le mini-shell

Objectif : programmer un shell en utilisant le module `Unix`

Les fonctionnalités et commandes que l'on va implémenter :

- Manipulation des fichiers et répertoires : `ls`, `mkdir`, `ln`, `cat`
- Modification du répertoire courant : `cd`
- Re-directions de flux : $>$, $<$
- Tube : $|$

Manipulation de fichiers et répertoires

Manipulation de fichiers et répertoire : interface du mini-shell

Première version de l'AST des commandes :

```
type command =
  | Cat of string list                (* cat files *)
  | Ln of string * string * bool      (* ln source dest [-s] *)
  | Mv of string * string             (* mv source dest *)
  | Rm of string * bool               (* rm filename [-r] *)
  | Mkdir of string * int option      (* mkdir dir [-m int] *)
  | Ls of string option               (* ls [filename] *)
```

Manipulation de fichiers et répertoire : interface du mini-shell

Première version de l'AST des commandes :

```
type command =
  | Cat of string list                (* cat files *)
  | Ln of string * string * bool      (* ln source dest [-s] *)
  | Mv of string * string             (* mv source dest *)
  | Rm of string * bool               (* rm filename [-r] *)
  | Mkdir of string * int option      (* mkdir dir [-m int] *)
  | Ls of string option               (* ls [filename] *)
```

Parser, et exécuteur de commandes :

```
val parse : string -> command
val exec_cmd : command -> unit
```

Parseur et exécuteur de commande

- Parseur : plusieurs solutions (`Args`, `Angstrom` etc..)
- Exécuteur de commandes :

```
let exec_cmd cmd =
    match cmd with
    | Cat filename                -> failwith "todo"
    | Ln (source, dest, symb)     -> failwith "todo"
    | Mv (source, dest)           -> failwith "todo"
    | Mkdir (dirname, perm_opt)   -> failwith "todo"
    | Rm (filename, recursive)    -> failwith "todo"
    | Ls name_opt                 -> failwith "todo"
```

Lecture et écriture de fichier (`cat`)

`cat f1 f2 f3 ...` : concatène le contenu des fichiers en entrée et les écrit dans la sortie standard.

Exemple

```
lyrm@carine:~$ echo -n "Bon" > f1
lyrm@carine:~$ echo "jour !" > f2
lyrm@carine:~$ cat f1 f2
Bonjour !
lyrm@carine:~$
```

Lecture et écriture de fichier (`cat`)

`cat f1 f2 f3 ...` : concatène le contenu des fichiers en entrée et les écrit dans la sortie standard.

Exemple

```
lyrm@carine:~$ echo -n "Bon" > f1
lyrm@carine:~$ echo "jour !" > f2
lyrm@carine:~$ cat f1 f2
Bonjour !
lyrm@carine:~$
```

Pseudo-code

Pour chaque fichier en entrée :

- ouvrir le fichier
- le lire et l'écrire sur la sortie standard
- fermer le fichier

Lecture et écriture de fichier (cat)

```
type file_descr          (* descripteurs de fichiers *)
val stdin : file_descr   (* entree standard *)
val stdout : file_descr  (* sortie standard *)
val stderr : file_descr  (* sortie d'erreur standard *)

type open_flag = (* modes d'ouverture *)
    | O_RDONLY   | O_WRONLY | O_RDWR
    | O_CREAT    | O_TRUNC
 (* |.. et tous les autres  *)
type file_perm = int (* droits (Ex : 0o777) *)

val openfile :
    string -> open_flag list -> file_perm -> file_descr
val close : file_descr -> unit
val read : file_descr -> bytes -> int -> int -> int
val single_write : file_descr -> bytes -> int -> int ->
    int
```

OCaml vs C : `write`

```
val single_write : file_descr -> bytes -> int -> int ->
    int
```

NAME top

write - write to a file descriptor

SYNOPSIS top

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

OCaml vs C : openfile

```
val openfile :
      string -> open_flag list -> file_perm -> file_descr
```

NAME top

open, openat, creat - open and possibly create a file

SYNOPSIS top

```
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);

int openat(int dirfd, const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags, mode_t mode);

/* Documented separately, in openat2(2): */
int openat2(int dirfd, const char *pathname,
            const struct open_how *how, size_t size);
```

Lecture et écriture de fichier (`cat`)

- pour lire un fichier

```
Unix.openfile filename [O_RDONLY] 0
```

- écrire en créant ou en effaçant un fichier existant

```
Unix.openfile filename
    [O_WRONLY; O_TRUNC; O_CREAT] 0o666
```

- écrire du code exécutable

```
Unix.openfile filename
    [O_WRONLY; O_TRUNC; O_CREAT] 0o777
```

- ajouter des données à la fin d'un fichier existant ou le créer vide sinon

```
Unix.openfile filename
    [O_WRONLY; O_APPEND; O_CREAT] 0o666
```

Lecture et écriture de fichier (`cat`)

Lecture et écriture de fichier (`cat`)

```
let exec_cmd cmd = match cmd with
  | Cat files ->
    List.iter (fun file ->
      let fd_in = (* file_perm inutile ici *)
        Unix.(openfile file [ O_RDONLY ] 0) in
      write_fd_out fd_in;
      Unix.close fd_in) files
  | ... -> ...
```

Lecture et écriture de fichier (cat)

```
let exec_cmd cmd = match cmd with
  | Cat files ->
    List.iter (fun file ->
      let fd_in = (* file_perm inutile ici *)
        Unix.(openfile file [ O_RDONLY ] 0) in
      write_fd_out fd_in;
      Unix.close fd_in) files
  | ... -> ...

let write_fd_stdout fd_in =
  let buffer_size = 8192 in
  let buffer = Bytes.create buffer_size in
  let rec loop () =
    match Unix.read fd_in buffer 0 buffer_size with
    | 0 -> ()
    | r -> ignore (Unix.(single_write stdout buffer 0 r));
           loop () in
  loop ()
```

Opérations sur les noms de fichiers (`ln (-s)`, `rm`, `mv`)

`ln f (-s)` : création d'un lien physique (symbolique).

`rm f -f` : efface le fichier de nom f

`mv f1 f2` : renomme le fichier de nom f1 en f2

Exemple

```
lyrm@carine:~/Tests$ ls
fichier_ordinaire
lyrm@carine:~/Tests$ ln fichier_ordinaire lien_physique
lyrm@carine:~/Tests$ ln -s fichier_ordinaire lien_symbolique
lyrm@carine:~/Tests$ ls
fichier_ordinaire  lien_physique  lien_symbolique
lyrm@carine:~/Tests$ rm -f lien_symbolique
lyrm@carine:~/Tests$ ls
fichier_ordinaire  lien_physique
lyrm@carine:~/Tests$ mv lien_physique fichier_ordinaire_2
lyrm@carine:~/Tests$ ls
fichier_ordinaire  fichier_ordinaire_2
```

Opérations sur les noms de fichiers (`ln (-s)`, `rm`, `mv`)

`ln f (-s)` : création d'un lien physique (symbolique).

`rm f -f` : efface le fichier de nom f

`mv f1 f2` : renomme le fichier de nom f1 en f2

Exemple

```
lyrm@carine:~/Tests$ ls
fichier_ordinaire
lyrm@carine:~/Tests$ ln fichier_ordinaire lien_physique
lyrm@carine:~/Tests$ ln -s fichier_ordinaire lien_symbolique
lyrm@carine:~/Tests$ ls
fichier_ordinaire  lien_physique  lien_symbolique
lyrm@carine:~/Tests$ rm -f lien_symbolique
lyrm@carine:~/Tests$ ls
fichier_ordinaire  lien_physique
lyrm@carine:~/Tests$ mv lien_physique fichier_ordinaire_2
lyrm@carine:~/Tests$ ls
fichier_ordinaire  fichier_ordinaire_2
```

Facile : ce sont des fonctions `Unix` existantes.

Opérations sur les noms de fichiers (`ln (-s)`, `rm`, `mv`)

```
(* Efface un fichier *)
val unlink : string -> unit

(* Renomme un fichier *)
val rename : string -> string -> unit

(* Creer un lien physique *)
val link : ?follow:bool -> string -> string -> unit

(* Creer un lien symbolique *)
val symlink : string -> string -> unit

(* Lit le contenu d'un lien symbolique *)
val readlink : string -> string
```

Opérations sur les noms de fichiers (`ln (-s)`, `rm`, `mv`)

```
let exec_cmd cmd = match cmd with
  | Cat files -> ...

  | Ln (source, dest, symbolic) ->
    if symbolic then Unix.symlink source dest
    else Unix.link source dest

  | Mv (source, dest) -> Unix.rename source dest

  | Rm (filename, recursive) ->
    if recursive then failwith "todo"
    else Unix.unlink filename

  | ... -> ...
```

Opérations sur les répertoires (`mkdir (-m int)`, `rm -r`, `ls`)

`mkdir dir (-m int)` : créer un répertoire avec les permissions définies par l'option `-m`.

`rm -r dir` : efface le répertoire nommé `dir` et son contenu .

`ls dir` : liste le contenu du répertoire `dir` ou du répertoire courant par défault sur la sortie standard.

Exemple

```
lyrm@carine:~$ ls Tests/
fichier_ordinaire  fichier_ordinaire_2
lyrm@carine:~$ cd Tests/
lyrm@carine:~/Tests$ mkdir nouveau_repertoire
lyrm@carine:~/Tests$ ls
fichier_ordinaire  fichier_ordinaire_2  nouveau_repertoire
lyrm@carine:~/Tests$ rm -r nouveau_repertoire
lyrm@carine:~/Tests$ ls
fichier_ordinaire  fichier_ordinaire_2
```

Opérations sur les répertoires : `mkdir (-m int)`, `rm -r`

Fonctions Unix :

```
type file_perm = int (* ex : 0o777 *)
val mkdir : string -> file_perm -> unit
val rmdir : string -> unit
```

Opérations sur les répertoires : `mkdir (-m int), rm -r`

Fonctions Unix :

```
type file_perm = int (* ex : 0o777 *)
val mkdir : string -> file_perm -> unit
val rmdir : string -> unit
```

Implémentation :

```
let exec_cmd cmd = match cmd with
  | Rm (filename, recursive) ->
    if recursive then Unix.rmdir filename
    else Unix.unlink filename
  | Mkdir (filename, perm_opt) -> (* Note : Le parseur
      ajoute "0o" devant la perm (666 ici) ([mkdir name -
      m 666]) *)
    let perm = match perm_opt with None -> 0o775 | Some
        p -> p in
    Unix.mkdir filename perm
  |  ... -> ...
```

Opérations sur les répertoires : `ls`

Fonctions Unix :

```
type dir_handle                  (* descripteur de lecture *)
val opendir : string ->  dir_handle
val readdir : dir_handle -> string  (* lit une entree *)
val closedir : dir_handle -> unit
```

Boucle de lecture des entrées du répertoire

```
let read_dir dirname =
  let rec loop dir_handle acc =
    try
      let nfile = Unix.readdir dir_handle in
        loop dir_handle (nfile :: acc)
    with End_of_file -> acc in

  let dir_handle = Unix.opendir dirname in (* Ouverture *)
  let files = loop dir_handle [] in          (* Lecture *)
  Unix.closedir direname;                  (* Fermeture *)
  files
```

Opérations sur les répertoires : `ls`

```
let exec_cmd cmd = match cmd with
    | Ls diropt ->
      let dirname = match diropt with
        | None -> Filename.current_dir_name
        | Some d -> d  in
      (* Lecture *)
      let files = read_dir dirname in
      (* Filtrage *)
      let files =  List.filter
          (fun file ->
            not (file = Filename.parent_dir_name ||
                 file = Filename.current_dir_name))
          all_files in
      (* Concatenation *)
      let files = String.concat "\t" files in
      (* Ecriture sur la sortie standard *)
      write_stdout (files ^ "\n")
    | ... -> ...
```

Opérations sur les répertoires : `ls`

La fonction d'écriture sur la sortie standard ressemble beaucoup à celle vue précédemment :

```
let write_stdout text =
  (* Conversion en bytes *)
  let text = Bytes.of_string text in
  (* Taille max du bloc d'ecriture *)
  let max_len = 8192 in
  (* boucle d'ecriture *)
  let rec loop ind to_write =
    let len = min max_len to_write in
    let written = Unix.single_write Unix.stdout text ind
        len in
    if written = max_len then
      loop (ind + written) (to_write - written)
  in
  loop 0 (Bytes.length text)
```

Opérations sur les répertoires : `ls`

Ajouter l'option `-i` pour `ls`

```
lyrm@carine:~/Tests$ ls -i
12976378 file1  12976443 file2
```

```
type stats = {
dev : int; (* Numéro de périphérique *)
ino : int; (* Numéro d'Inode *)
kind : file_kind; (* Type de fichiers (S_REG, S_DIR) *)
perm : file_perm; (* Droits *)
 }

(* Information sur le fichier *)
val stat : string -> stats
(* Comme [stat] avec un descripteur de fichier *)
val fstat : file_descr -> stats
(* Donne l'information sur un lien symbolique *)
val lstat : string -> stats
```

Gestion des erreurs Unix et boucle principale de lecture

Gestion des erreurs

En C, pour chaque appel système :

```
if (unappel() == -1) {
  printf("unappel() a échoué\n");
  if (errno == ...) { ... }}
```

Gestion des erreurs

En C, pour chaque appel système :

```
if (unappel() == -1) {
  printf("unappel() a échoué\n");
  if (errno == ...) { ... }}
```

Avec OCaml, une fois à la fin du programme

```
type error = | E2BIG | EACCES | EAGAIN | EBADF | ...
val handle_unix_error : ('a -> 'b) -> 'a -> 'b
```

Gestion des erreurs

En C, pour chaque appel système :

```
if (unappel() == -1) {
  printf("unappel() a échoué\n");
  if (errno == ...) { ... }}
```

Avec OCaml, une fois à la fin du programme

```
type error = | E2BIG | EACCES | EAGAIN | EBADF | ...
val handle_unix_error : ('a -> 'b) -> 'a -> 'b
```

- ❒ applique le premier argument (notre programme) au second et retourne le résultat.
- ❒ Si une exception `Unix.Unix_error` est levée :
  - ❖ affiche une message décrivant l'erreur
  - ❖ le programme termine avec le code d'erreur 2

Boucle principale de lecture : premier essai

```
let minishell () =
```

```
let () = Unix.handle_unix_error minishell ()
```

Boucle principale de lecture : premier essai

```
let minishell () =

    try
      while true do
       (* Lire stdin *)
        let cmd_line = Stdlib.input_line Stdlib.
            stdin in
        try
          (* Parser la commande *)
          let cmd = parse cmd_line in
          (* Exécuter la commande *)
          exec_cmd cmd
        with
        | Parsing_error err -> print_error err
        | Empty_line -> ()
      done
    with End_of_file -> ()

let () = Unix.handle_unix_error minishell ()
```

Processus Unix

Processus `Unix`

Un processus Unix :

- est un programme en train de s'exécuter
- se compose de :
  - un texte de programme
  - un état du programme (point de contrôle courante, valeur de variables, descripteurs de fichiers ouverts etc..)
- se crée avec

```
val fork : unit -> int
```

- le fils est une copie du père (duplique les fd par exemple)
- la seule différence est la valeur de sortie de `fork` :
  - le fils : 0
  - le père : l'identifiant du fils (pid)

Boucle de lecture

```
let minishell () =
   try
     while true do
       (* Lecture de l'entree standard *)
       let cmd = input_line stdin in
       match Unix.fork () with
         | 0 -> (* Processus fils *)
           let cmd = parse cmd_line in
           exec_cmd cmd;
           exit 0
         | pid_son -> (* Processus père *)
           let pid, status = Unix.wait() in
           (* Affiche le status de sortie du fils *)
            print_status "Program" status
     done
   with End_of_file -> ()

let () = Unix.handle_unix_error minishell ()
```

Redirection et tubes

Encore plus de fonctionnalités

- des commandes externes (pas implémenté en OCaml (!!))
- des redirections (< >)
- des tubes (|)
- les combinateurs AND et OR (&& et ||)

AST évolué

```
type cmd_kind =
    | Internal of command  (* les commandes precedentes *)
    | External of string list  (* la triche avec execvp *)
    | Cd of string
type redirection =
    | In of string (* > *)
    | Out of string (* < *)
type t =
    | Command of cmd_kind * redirection list
    | Pipe of t * t (* | *)
    | And of t * t  (* && *)
    | Or of t * t   (* || *)
val execute : t -> unit
```

Nouvelle boucle de lecture

```
let minishell () =
 try
     prompt None;
     while true do
       let cmd = input_line stdin in
       try
         let cmd = Ast.parse cmd in
         let code = interprete cmd in
         prompt (Some code)
       with
       | Parser.Parsing_error err ->
            Parser.print_error err
       | Parser.Empty_line -> ()
     done
   with End_of_file -> ()
```

Dossier courant

- pas de `fork`
- `Unix.chdir : string -> unit`
- `Unix.getcwd : unit -> string`

Implémentation :

```
let prompt = function
    | None -> Printf.printf "%s␣>␣" (Unix.getcwd ())
    | Some code -> Printf.printf "%s␣(%d)>" (Unix.getcwd
        ()) code

let rec interprete : Ast.t -> int = function
    | Ast.Command (Cd target, _) -> (
      match Unix.chdir target with
      | exception Unix.Unix_error _ -> 1
      | _ -> 0)
    | ...
```

Programmes externes

`Unix.execvp : string -> string array -> 'a`
exécute le programme donné en argument en cherchant dans le `PATH`.

- ne retourne pas
- conserve l'environnement et les descripteurs ouverts

Implémentation

```
let exec_cmd = function
  | Ast.Internal cmd -> Command.exec_cmd cmd
  | Ast.External (program :: _ as cmd) ->
      Unix.execvp program (Array.of_list cmd)
```

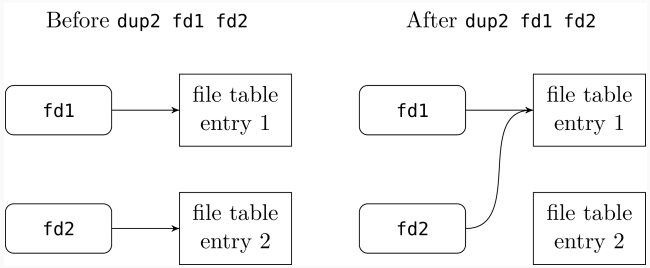
Mise à jour de `interprete`

```
let fork_and_wait fn =
  match Unix.fork () with
  | 0 -> fn () |> exit
  | pid_son -> (
      let _pid, status = Unix.waitpid [] pid_son in
      match status with WEXITED i -> i | WSIGNALED i ->
          i | WSTOPPED i -> i)

let rec interprete : Ast.t -> int = function
  | Ast.Command (Cd target, _) -> (
      match Unix.chdir target with
      | exception Unix.Unix_error _ -> 1
      | _ -> 0)
  | Command (cmd, redirections) ->
      fork_and_wait (fun () -> exec_cmd cmd; 0)
  | ...
```

Redirections

```
let rec interprete : Ast.t -> int = function
  | Ast.Command (Cd target, _) -> (
      match Unix.chdir target with
      | exception Unix.Unix_error _ -> 1
      | _ -> 0)
  | Command (cmd, redirections) ->
      fork_and_wait (fun () ->
          (* mise en place des redirections avant d'
             exécuter la commande*)
          List.iter setup_redirection redirections;
          exec_cmd cmd;
          0)
  | ...
```

Redirections



Fonction Unix :

```
val dup2 : file_descr ->
    file_descr -> unit
```

Implémentation :

```
let redirection_entree target =
  let fd = Unix.openfile target [ Unix.O_RDONLY ] 0 in
  Unix.dup2 fd Unix.stdin;
  Unix.close fd

let redirection_sortie target =
  let fd = Unix.openfile target [ Unix.O_CREAT ;
                                  Unix.O_TRUNC ;
                                  Unix.O_WRONLY ] 0o660
  in
  Unix.dup2 fd Unix.stdout;
  Unix.close fd
```

Tubes

Fonction Unix :

```
val pipe : unit -> file_descr * file_descr
```

Implémentation :

```
let pipe_cmd (a: Command.t) (b: Command.t) =
  let fd_in, fd_out = Unix.pipe () in
  match Unix.fork () with
  | 0 ->
      Unix.dup2 fd_out Unix.stdout;
      Unix.close fd_out;
      Unix.close fd_in;
      interprete a
  | _ ->
      Unix.dup2 fd_in Unix.stdin;
      Unix.close fd_out;
      Unix.close fd_in;
      interprete b
```

AND / OR

```
let rec interprete : Ast.t -> int = function
    ...
    | And (a, b) ->
        let code_a = interprete a in
        if code_a <> 0 then code_a else interprete b
    | Or (a, b) ->
        let code_a = interprete a in
        if code_a == 0 then 0 else interprete b
```

Interlude réseau

Réseau - côté client

```
type host_entry = {
^^Ih_addr_list : inet_addr array;
    ...
}
val gethostbyname : string -> host_entry

type socket_domain = PF_UNIX | PF_INET | PF_INET6
type socket_type =
    | SOCK_STREAM (* TCP *)
    | SOCK_DGRAM  (* UDP *)
val socket : ?cloexec:bool ->
       socket_domain -> socket_type -> int -> file_descr

type sockaddr =
    | ADDR_UNIX of string
    | ADDR_INET of inet_addr * int
val connect : file_descr -> sockaddr -> unit
```

Réseau - côté client - exemple

```
open Unix

bytes
let requete_http = Bytes.of_string "GET / HTTP/1.1\r\nHost: perdu.com\r\n\r\n"
bytes
let reponse = Bytes.create 128
host_entry
let hote = gethostbyname "perdu.com"
file_descr
let socket = socket PF_INET SOCK_STREAM 0
sockaddr
let addresse = ADDR_INET (hote.h_addr_list.(0), 80)

let () = connect socket addresse

let _ = write socket requete_http 0 (Bytes.length requete_http)

let () =
  let rec r () =
    let n = read socket reponse 0 128 in
    String.sub (Bytes.to_string reponse) 0 n ▷ print_endline ;
    if n = 128 then
      r ()
  in
  r ()
```

Réseau - côté serveur

```
(* attache la socket à une adresse *)
val bind : file_descr -> sockaddr -> unit
(* met la socket en mode d'écoute *)
val listen : file_descr -> int -> unit
(* accepte une nouvelle connexion *)
val accept :
    ?cloexec:bool -> file_descr -> file_descr * sockaddr
```

Réseau - côté serveur - exemple

```
open Unix

file_perm
let buffer_size = 128
bytes
let buffer = Bytes.create buffer_size
file_descr
let socket = socket PF_INET SOCK_STREAM 0

let () =
  bind socket (ADDR_INET (inet_addr_of_string "0.0.0.0", 1025));
  listen socket 1;
  let rec loop () =
    let (connection, src) = accept socket in
    let (ip, port) = match src with
      | ADDR_INET (ip, port) → (ip, port)
      | _ → failwith ""
    in
    Printf.printf "Nouveau client: %s:%d\n%!" (string_of_inet_addr ip) port;
    let rec read_loop () =
      let n = read connection buffer 0 buffer_size in
      write connection buffer 0 n ▷ ignore;
      if n > 0 then read_loop ()
    in
    read_loop ();
    Printf.printf "Client parti.\n%!";
    loop ()
  in
  loop ()
```

Programmation système haut niveau

Interactions haut niveau avec le système

Le module `Stdlib` contient des primitives haut niveau pour la manipulation de fichiers

```
type in_channel
type out_channel

val open_in : string -> in_channel
val input_line : in_channel -> string
val close_in : in_channel -> unit

val open_out : string -> out_channel
val output_string : out_channel -> string -> unit
val close_out : in_channel -> unit
```

Programmation asynchrone

Le problème des appels en système de lecture/écriture : c'est bloquant.

Plusieurs stratégies :

- Utiliser des fils d'exécution (threads)
- `Lwt` : bibliothèque pour fils d'exécutions légers et coopératifs

Exemple : cp asynchrone avec Lwt_unix

```
open Lwt.Syntax

let rec perform_copy_lwt src dst =
  let* n = Lwt_unix.read src buffer 0 buf_size in
  if n = buf_size then
    let* _ = Lwt_unix.write dst buffer 0 n in
    perform_copy_lwt src dst
  else
    let* _ = Lwt_unix.write dst buffer 0 n in
    Lwt.return (`Ok ())

let cp_lwt src dest =
  Lwt_main.run @@
  let* fd_src = Lwt_unix.openfile src [O_RDONLY] 0 in
  let* fd_dst = Lwt_unix.openfile dest [O_RDWR; O_CREAT;
       O_TRUNC] 0o640 in
  perform_copy_lwt fd_src fd_dst
```

Concurrence en OCaml

Fil d'exécution : introduction

Processus Unix : espaces mémoire totalement disjoints

Fils d'exécution :

- le même espace d'adressage
- les différences entre eux :
  - leur identité,
  - leur pile dexécution,
  - quelques informations pour le système (masque des signaux, état des verrous et conditions, etc.)

Le partage de donnée

```
let main () =
  let compteur = ref 0 in
  let max = 1_000_000 in

  let increment () =
    (*let list =
        ref (List.init 50 (fun i -> i)) in*)
    for _ = 1 to max do
      let tmp = !compteur in
      (*list := List.rev !list;*)
      compteur := tmp + 1
    done
  in
  let t1 = Thread.create increment () in
  let t2 = Thread.create increment () in
  Thread.join t1;
  Thread.join t2;
  print_int !compteur
```

Le partage de donnée

```
let main () =
  let compteur = ref 0 in
  let max = 1_000_000 in

  let increment () =
    (*let list =
        ref (List.init 50 (fun i -> i)) in*)
    for _ = 1 to max do
      let tmp = !compteur in
      (*list := List.rev !list;*)
      compteur := tmp + 1
    done
  in
  let t1 = Thread.create increment () in
  let t2 = Thread.create increment () in
  Thread.join t1;
  Thread.join t2;
  print_int !compteur
```

```
1698861
1301916
1692602
1667393
1346085
1410027
1485207
1512540
1578875
1303801
```

Les solutions de synchronisation :

- les opérations atomiques (module `Atomic`) : une opération qui s'exécute sans être interrompus
- les mutex (algorithme de Peterson et de la boulangerie)
- les sémaphores

Peterson

```
let b1, b2 = Atomic.make false, Atomic.make false
let tour = Atomic.make 0
```

```
1  let rec p1 () =
2   Atomic.set b1 true;
3   Atomic.set tour 2;
4   while (Atomic.get b2
5    && Atomic.get tour = 2)
        do ()
6   done;
7   (* SC *)
8   Atomic.set b1 false;
9   p1 ()
```

```
let rec p2 () =
 Atomic.set b2 true;
 Atomic.set tour 1;
 while (Atomic.get b1
   && Atomic.get tour = 1)
      do ()
 done;
 (* SC *)
 Atomic.set b2 false;
 p2 ()
```

Propriétés de Peterson

- un seul thread peut être dans la section critique en même temps (preuve par l'absurde)
- pas d'interblocage : un thread peut toujours avancer
- pas de famine : un thread ne peut pas ne jamais avoir accès à la SC

Peterson

```
module Mutex = struct
  type 'a t =
    { flag : bool Atomic.t Array.t;
      turn : int Atomic.t }

  let init () =
    { flag = Array.make 2 (Atomic.make false);
      turn = Atomic.make 0 }

  let lock t id =
    Atomic.set t.flag.(id) true;
    Atomic.set t.turn (1 - id);
    while Atomic.get t.flag.(1 - id) && Atomic.get t.
        turn = 1 - id do
      ()
    done

  let unlock t id = Atomic.set t.flag.(id) false
end
```

Algorithme de la boulangerie

Ses avantages :

- plus que 2 fils d'exécution
- ne nécessite pas d'opérations atomiques
- fonctionne sur le principe d'une file d'attente avec tickets numérotés

Algorithme de la boulangerie

```
type t =
  { compteur : int Array.t; choix : bool Array.t; size : int }

let init size =
  { compteur = Array.make size 0; choix = Array.make size false;
      size}

let lock t id =
  t.choix.(id) <- true;
  let max = ref 0 in
  Array.iteri
      (fun i elt -> if i <> id then max := Int.max elt !max)
      t.compteur;
  t.compteur.(id) <- !max + 1;
  t.choix.(id) <- false;
  for j = 0 to t.size -1 do
    while t.choix.(j) do () done;
    while t.compteur.(j) > 0 &&
          (t.compteur.(j) < t.compteur.(id) ||
          (t.compteur.(j) == t.compteur.(id) && j < id)) do ()
              done;
  done
```

Fin de la partie 1

Questions

OCaml dans la vraie vie

OCaml IRL

- MirageOS : un système d'exploitation modulaire écrit en OCaml
- OCaml 5 : programmation parallèle (et effets)
- QCheck : vérification de propriétés automatique
- Écosystème et environnements de programmation

Mirage

MirageOS

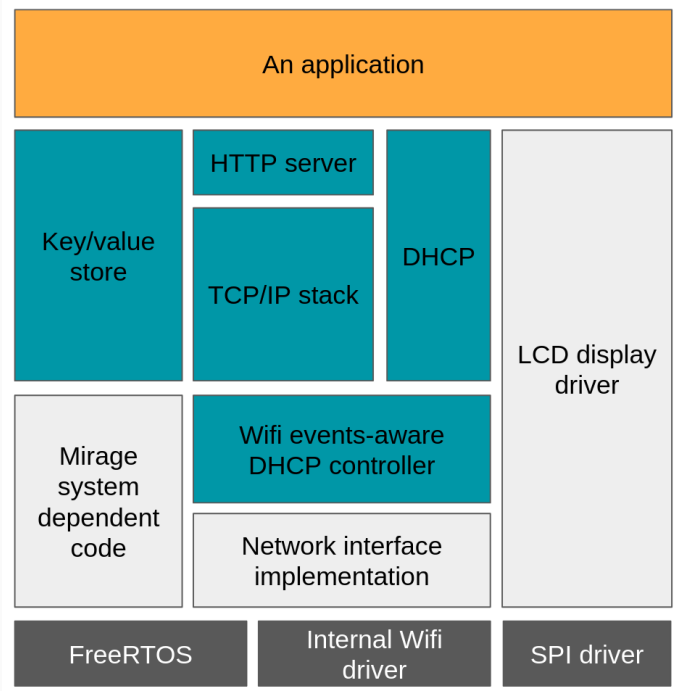


Figure 1 – Une application modulaire

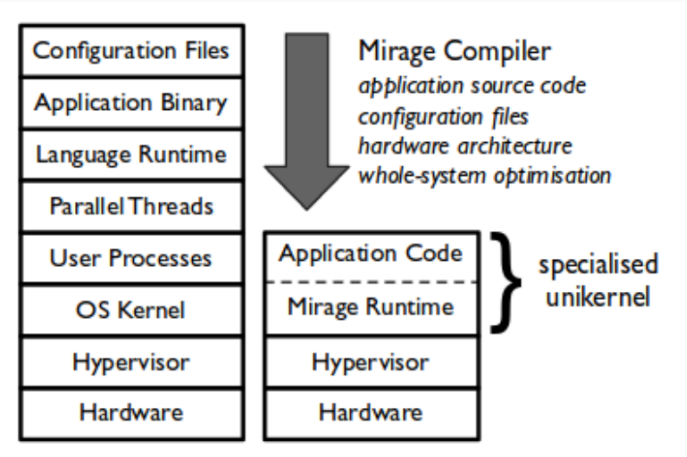


Figure 2 – MirageOS compile à la fois l'application et le système dans un seul exécutable

Abstraction : signatures de modules

```
module type Read_only_store = sig
    type t
    type key = string

    val get : t -> key -> string Lwt.t
end

module type HTTP = sig
    type t
    module Request : sig
        type t

        val path : t -> string
    end
    type reponse = string

    val listen : t -> (HTTP.Request.t -> response Lwt.t)
         -> unit Lwt.t
end
```

Implémentation

```
(* Les dépendances de notre application sont abstraites
   via le foncteur Make *)
module Make (FS : Read_only_store) (Server : HTTP) =
struct
    let start fs http =
        Server.listen http (fun request ->
            let path = Server.Request.path request in
            FS.get fs path)
end

(* pour les tests *)
module Application =
    Make (In_memory_store) (Mock_http_server)

(* en production *)
module Application = Make (EXT4) (Cohttp.Server)
```

Modules disponibles

- Socket réseau : TCP/IP de l'OS ou en OCaml
- Stockage : en mémoire, Irmin, FAT
- Protocoles : Git, HTTP, SSH, DNS, DHCP
- Cryptographie / compression

OCaml 5

OCaml 5

Sortie de OCaml 5 à la rentrée

- OCaml multicore
- les effets

Sortie de OCaml 5 à la rentrée

- OCaml multicore
- les effets

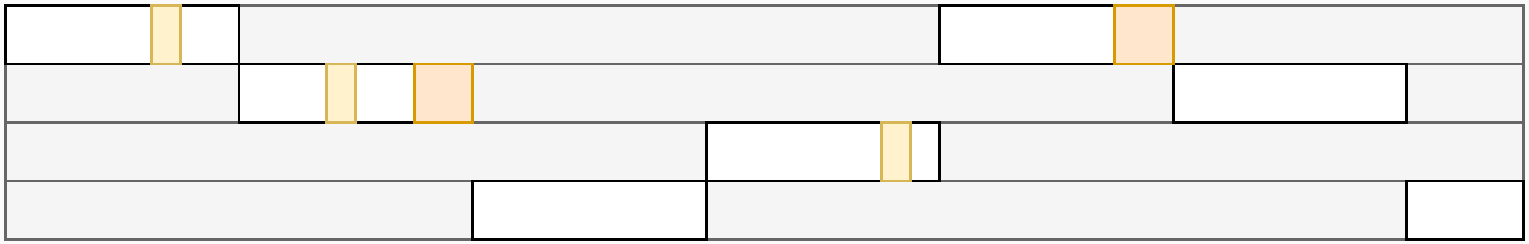


Figure 3 – Concurrence

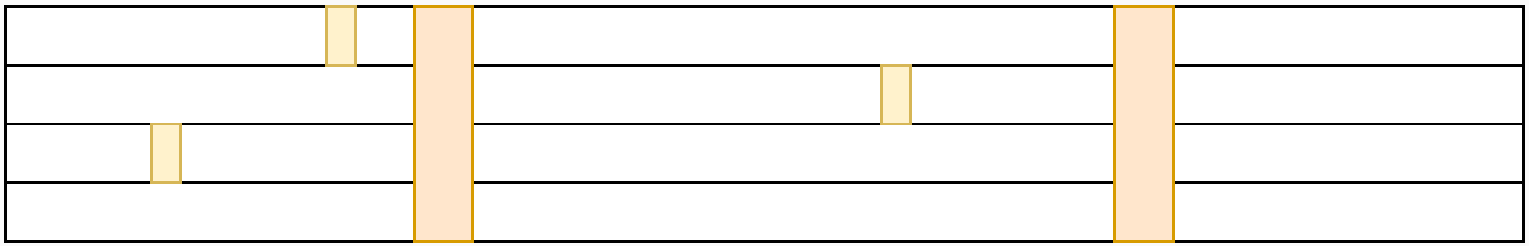


Figure 4 – Parallélisme

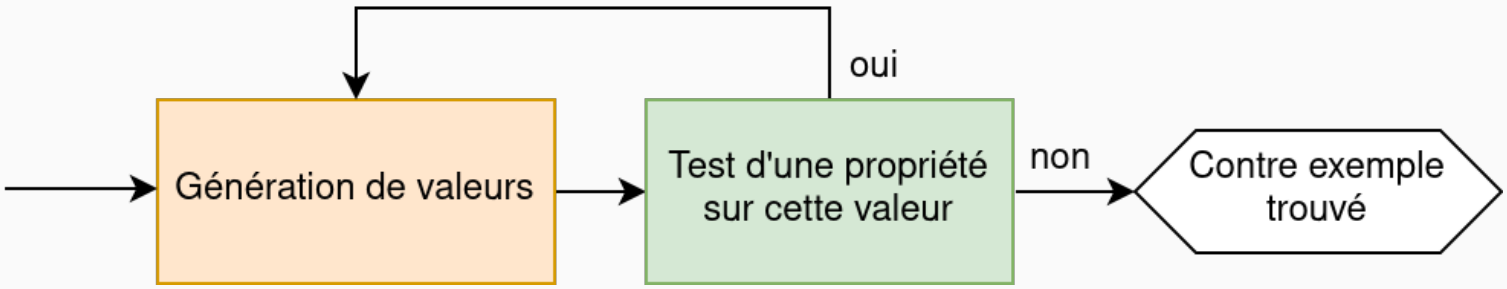
Effets

```
effect Compter : int

let programme () =
    let somme = ref 0 in
    for i = 0 to 9 do
        somme := !somme + perform Compter
    done;
    print_int !somme

let _ =
  let compteur = ref 0 in
  try programme () with
  | effect Compter k ->
    incr compteur;
    Printf.printf "+1: %d\n" !compteur;
    continue k (!compteur)
```

Vérification de propriétés avec QCheck

QCheck

Vérification de propriétés



QCheck cherche un contre exemple minimal

```
type 'a generator
val small_int : int generator
val small_list : 'a generator -> 'a list generator
val Test.make : 'a generator -> ('a -> bool) -> Test.t
```

Plus d'informations :

`https://github.com/c-cube/qcheck`

Exemple de test

```
let generateur = QCheck.(small_list small_int)
```

Exemple 1 :

```
(* forall l, List.rev (List.rev l) = l *)
Test.make generateur
    (fun l ->  List.rev (List.rev l) = l)
```

Exemple de test

```
let generateur = QCheck.(small_list small_int)
```

Exemple 1 :

```
(* forall l, List.rev (List.rev l) = l *)
Test.make generateur
    (fun l ->  List.rev (List.rev l) = l)
```

Exemple 2 :

```
(* forall l, int list, tri_fusion l est trié *)
Test.make generateur
    (fun l ->  List.sort compare entree = tri_fusion
        entree)
```

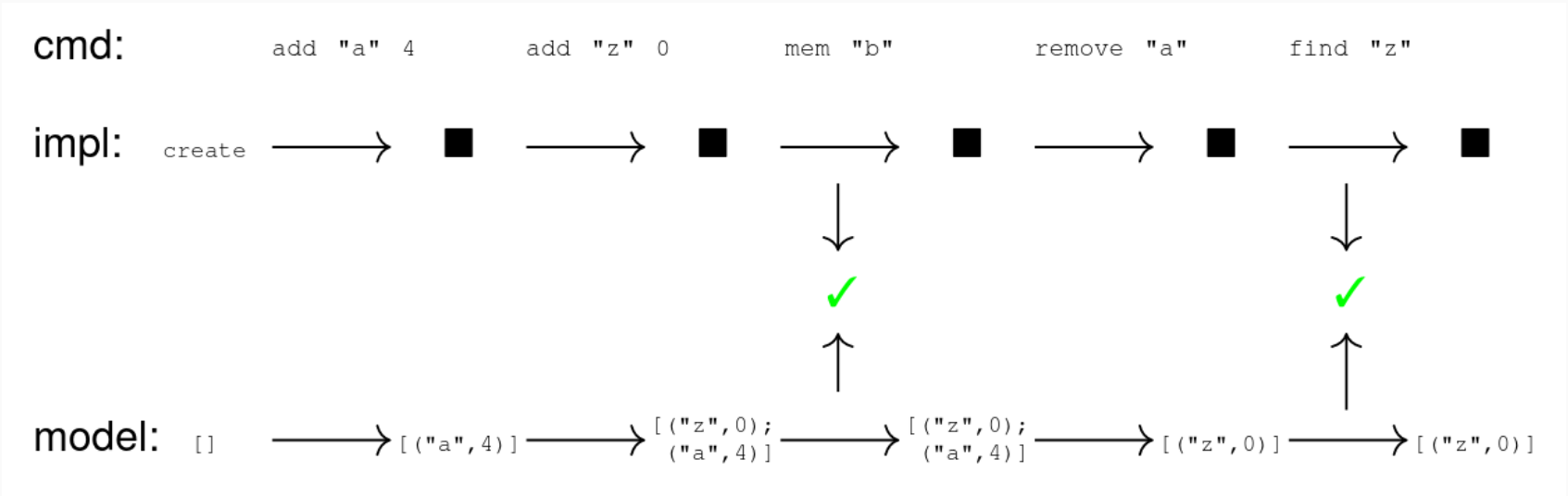
Réduction

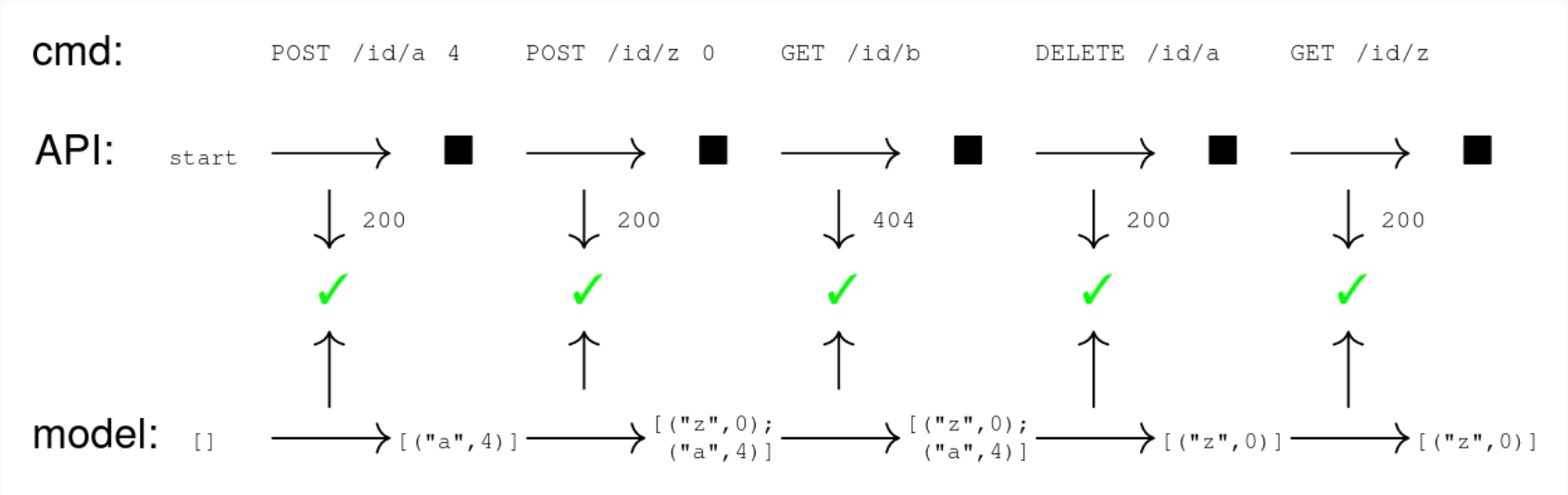
```
let rev lst =
  if List.length lst >= 5 then lst
  else List.rev lst
```

Réduction du contre exemple

```
0   => [31766076390300787l9; -70277791713580719l;
    1243225506173352439; -168141035741141589;
    447859169338941937 8; -290848208446581001 1;
    -447160499359612583 6; 209704831468549078 2;
    -777119667999583759]
1   => [1243225506173352439; -168141035741141589;
    447859169338941937 8; -290848208446581001 1;
    -447160499359612583 6; 209704831468549078 2;
    -777119667999583759]
200 => [0; 0; 0; 409579748962010 0; -777119667999583759]
255 => [0; 0; 0; 0; -194279916999895940]
299 => [0; 0; 0; 0; -11044]
313 => [0; 0; 0; 0; -1]
```

Simulation d'un état





Écosystème

Écosystème

Ressources en ligne

Site ocaml.org

Documentation des paquets (ex : ocaml.org/p/bos)

Outils de programmation

- Gestionnaire de paquets : `opam`
- Build system : `dune`
- Formatteur : `ocamlformat`
- Extension VS Code : OCaml Platform / ocaml-lsp-server
- Installateur (Linux) : `ocaml-platform-installer`
- Installateur (Windows) : Diskuv OCaml

Environnement pré-configuré

Github Codespace : un environnement OCaml pré-configuré (nécessite un compte github éducation)

Industrie

Les entreprises qui utilise ent OCaml, en vrac

- Facebook
- ahrefs
- Jane Street
- Hyper
- LexiFi
- etc..

(Cf ocaml.org/industrial-users)

The end

Questions