UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports Techniques

N°147

*Programme 2*

*Calcul symbolique, Programmation et Génie logiciel*

# PROGRAMMATION DU SYSTÈME UNIX EN CAML LIGHT

Xavier LEROY

Décembre 1992

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. :(1)39 63 55 11

# Programmation du système Unix en Caml Light

**Xavier Leroy**[1]

### Résumé

Ce rapport est un cours d'introduction à la programmation du système Unix, mettant l'accent sur la communication entre les processus. La principale nouveauté de ce travail est l'utilisation du langage Caml Light, un dialecte du langage ML, à la place du langage C qui est d'ordinaire associé à la programmation système. Ceci donne des points de vue nouveaux à la fois sur la programmation système et sur le langage ML.

# Unix system programming in Caml Light

**Xavier Leroy**

### Abstract

This report is an introductory course on Unix system programming, with an emphasis on communications between processes. The main novelty of this work is the use of the Caml Light language, a dialect of the ML language, instead of the C language that is customary in systems programming. This gives an unusual perspective on systems programming and on the ML language.

[1] École Normale Supérieure et INRIA Rocquencourt, projet Formel

# Introduction

Ce rapport est issu des notes d'un cours de programmation système que j'ai enseigné en première année du Magistère de Mathématiques Fondamentales et Appliquées et d'Informatique de l'École Normale Supérieure.

La tradition veut que la programmation du système Unix se fasse dans le langage C. Dans le cadre de ce cours, il m'a semblé plus intéressant d'utiliser un langage de plus haut niveau, Caml en l'occurrence, pour expliquer la programmation du système Unix.

La présentation Caml des appels systèmes est plus abstraite, utilisant toute la puissance de l'algèbre de types de ML pour représenter de manière claire les arguments et les résultats, au lieu de devoir tout coder en termes d'entiers et de champs de bits comme en C. En conséquence, il est plus facile d'expliquer la sémantique des appels systèmes, sans avoir à se perdre dans les détails de l'encodage des arguments et des résultats. (Voir par exemple la présentation de l'appel `wait`, page 29.)

De plus, Caml apporte une plus grande sécurité de programmation que C, en particulier grâce au typage statique et à la clarté de ses constructions de base. Ces traits, qui peuvent apparaître au programmeur C chevronné comme de simples éléments de confort, se révèlent cruciaux pour les programmeurs inexpérimentés comme ceux auxquels ce cours s'adresse.

Un deuxième but de cette présentation de la programmation système en Caml est de montrer le langage Caml à l'œuvre dans un domaine qui sort nettement de ses applications usuelles, à savoir la démonstration automatique, la compilation, et le calcul symbolique. Caml se tire plutôt bien de l'expérience, essentiellement grâce à son solide noyau impératif, complété ponctuellement par les autres traits plus novateurs du langage (polymorphisme, fonctions d'ordre supérieur, exceptions). Le fait que Caml combine programmation applicative et programmation impérative, au lieu de les exclure mutuellement, rend possible l'intégration dans un même programme de calculs symboliques compliqués et d'une bonne interface avec le système.

Ces notes supposent le lecteur familier avec le système Caml Light, et avec l'utilisation des commandes Unix, en particulier du shell. On se reportera à la documentation du système Caml Light [1] pour toutes les questions relatives au langage, et à la section 1 du manuel Unix ou à un livre d'introduction à Unix [3, 4] pour toutes les questions relatives à l'utilisation d'Unix.

On décrit ici uniquement l'interface "programmatique" du système Unix, et non son implémentation ni son architecture interne. L'architecture interne de BSD 4.3 est décrite dans [6] ; celle de

System V, dans [7]. Les livres de Tanenbaum [8, 9] donnent une vue d'ensemble des architectures de systèmes et de réseaux.

Le système Caml Light et la bibliothèque d'interface avec Unix présentée ici sont en accès libre par FTP anonyme sur la machine `nuri.inria.fr` (128.93.1.26), dans le répertoire `lang/caml-light`.

# Chapitre 1

# Généralités

## 1.1 Les modules sys et unix

Les fonctions qui donnent accès au système depuis Caml Light sont regroupées dans deux modules. Le premier module, `sys`, contient les quelques fonctions communes à Unix et aux autres systèmes d'exploitation sous lesquels tourne Caml Light. Le second module, `unix`, contient tout ce qui est spécifique à Unix. On trouvera dans l'annexe C l'interface du module `unix`.

Par la suite, on fait référence aux identificateurs des modules `sys` et `unix` sans préciser de quel module ils proviennent. Autrement dit, on suppose qu'on est dans la portée des directives `#open "sys";;` et `#open "unix";;`. Dans les exemples complets (ceux dont les lignes sont numérotées), on met explicitement les `#open`, afin d'être vraiment complet.

Pour compiler un programme Caml Light qui utilise la bibliothèque Unix, il faut faire :

```
camlc unix.zo mod1.ml mod2.ml mod3.ml -o myprog -custom -lunix
```

en supposant que le programme est composé des trois modules `mod1`, `mod2` et `mod3`. On peut aussi compiler séparément les modules :

```
camlc -c mod1.ml
camlc -c mod2.ml
camlc -c mod3.ml
```

puis faire pour l'édition de lien :

```
camlc unix.zo mod1.zo mod2.zo mymod3.zo -o myprog -custom -lunix
```

Dans les deux cas, l'argument `unix.zo` représente la partie de la bibliothèque `unix` écrite en Caml Light ; l'argument `-lunix`, la partie écrite en C. L'option `-custom` signale qu'on va étendre l'exécutant (*runtime system*) standard de Caml Light par des primitives C supplémentaires.

On peut aussi accéder au système Unix depuis le système interactif (le "toplevel"). Il faut d'abord créer un système interactif contenant les fonctions systèmes préchargées :

```
camlmktop -custom -o camlunix unix.zo -lunix
```

Ce système se lance ensuite par :

```
camllight camlunix
```

## 1.2 Interface avec le programme appelant

Lorsqu'on lance un programme depuis un shell (interpréteur de commandes), le shell transmet au programme des *arguments* et un *environnement*. Les arguments sont les mots de la ligne de commande qui suivent le nom de la commande. L'environnement est un ensemble de chaînes de la forme `variable=valeur`, représentant les liaisons globales de variables d'environnements : les liaisons faites avec `setenv var=val` dans le cas du shell `csh`, ou bien avec `var=val ; export var` dans le cas du shell `sh`.

Les arguments passés au programme sont placés dans le vecteur de chaînes `command_line` :

```
command_line : string vect
```

L'environnement du programme tout entier s'obtient par la fonction `environment` :

```
environment : unit -> string vect
```

Une manière plus commode de consulter l'environnement est par la fonction `getenv` :

```
getenv : string -> string
```

`getenv` $v$ renvoie la valeur associée à la variable de nom $v$ dans l'environnement, et déclenche l'exception `Not_found` si cette variable n'est pas liée.

Un programme termine normalement après avoir exécuté toutes les phrases qui le composent. Le code de retour renvoyé au programme appelant est toujours zéro dans ce cas. Un programme peut terminer prématurément par l'appel `exit` :

```
exit : int -> 'a
```

L'argument est le code de retour à renvoyer au programme appelant. La convention est de renvoyer zéro comme code de retour quand tout s'est bien passé, et un code de retour non-nul pour signaler une erreur. Le shell `sh`, dans les constructions conditionnelles, interprète le code de retour 0 comme le booléen "vrai" et tout code de retour non-nul comme le booléen "faux".

Il y a en fait deux fonctions `exit`, l'une dans le module `sys`, l'autre dans le module `io`. La seule différence entre ces deux fonctions est que `sys__exit` ne termine pas les écritures en attente sur la sortie standard et sur la sortie d'erreur du programme, alors que `io__exit` exécute `io__flush` sur la sortie standard et sur la sortie d'erreur, terminant ainsi les écritures en attente, avant d'appeler `sys__exit`.

## 1.3 Traitement des erreurs

Sauf mention du contraire, toutes les fonctions du module `unix` déclenchent l'exception `Unix_error` en cas d'erreur.

```
exception Unix_error of error * string * string
```

Le deuxième argument de l'exception `Unix_error` est le nom de l'appel système qui a déclenché l'erreur. Le troisième argument identifie, si possible, l'objet sur lequel l'erreur s'est produite ; par exemple, pour un appel système qui prend en argument un nom de fichier, c'est ce nom qui se retrouve en troisième position dans `Unix_error`. Enfin, le premier argument de l'exception est un code d'erreur, indiquant la nature de l'erreur. Il appartient au type concret énuméré `error` :

```
type error = EPERM | ENOENT | EINTR | ...
```

Les constructeurs de ce type reprennent les mêmes noms et les mêmes significations que ceux employés dans la partie 2 du manuel Unix (voir par exemple la page `intro(2)`).

Étant donné la sémantique des exceptions, une erreur qui n'est pas spécialement prévue et interceptée par un `try` se propage jusqu'au sommet du programme, et le termine prématurément. Qu'une erreur imprévue soit fatale, c'est généralement la bonne sémantique. Il convient néanmoins de l'afficher de manière claire. Pour ce faire, le module `unix` fournit la fonctionnelle `handle_unix_error` :

```
handle_unix_error : ('a -> 'b) -> 'a -> 'b
```

L'appel `handle_unix_error` $f$ $x$ applique la fonction $f$ à l'argument $x$. Si cette application déclenche l'exception `Unix_error`, un message décrivant l'erreur est affiché, et on sort par `exit 2`. L'utilisation typique est

```
handle_unix_error main ();;
```

où la fonction `main : unit -> unit` exécute le corps du programme.

Pour référence, voici comment est implémentée `handle_unix_error`.

```
#open "unix";;
let handle_unix_error f arg =
  try
    f arg
  with Unix_error(err, fun_name, arg) ->
    prerr_string sys__command_line.(0);
    prerr_string " : '";
    prerr_string fun_name;
    prerr_string "' failed";
    if string_length arg > 0 then begin
      prerr_string " on '";
      prerr_string arg;
      prerr_string "'"
    end;
```

```
    prerr_string ": ";
    prerr_endline (error_message err);
    io__exit 2
;;
```

La primitive `error_message`, de type `error -> string`, renvoie un message décrivant l'erreur donnée en argument. L'argument numéro zéro de la commande, `sys__command_line.(0)`, contient le nom de commande utilisé pour invoquer le programme.

# Chapitre 2

# Les fichiers

Le terme “fichier” en Unix recouvre plusieurs types d'objets :

- les fichiers normaux : les suites finies d'octets contenant du texte ou des informations binaires qu'on appelle d'ordinaire fichiers
- les répertoires
- les liens symboliques
- les fichiers spéciaux (*devices*), qui donnent en particulier accès aux périphériques de la machine
- les tuyaux nommés (*named pipes*)
- les prises (*sockets*) nommées dans le domaine Unix.

## 2.1 Noms de fichiers, descripteurs de fichiers

Il y a deux manières d'accéder à un fichier. La première est par son *nom*, ou *chemin d'accès* à l'intérieur de la hiérarchie de fichiers. (Un même fichier peut avoir plusieurs noms différents, du fait des liens durs.) Les noms sont représentés par des chaînes de caractères (type `string`). Voici quelques exemples d'appels système qui opèrent au niveau des noms de fichiers :

| | |
|---|---|
| `unlink` $f$ | efface le fichier de nom $f$ (comme la commande `rm -f` $f$) |
| `link` $f_1$ $f_2$ | crée un lien dur nommé $f_2$ sur le fichier de nom $f_1$ (comme la commande `ln` $f_1$ $f_2$) |
| `symlink` $f_1$ $f_2$ | crée un lien symbolique nommé $f_2$ sur le fichier de nom $f_1$ (comme la commande `ln -s` $f_1$ $f_2$) |
| `rename` $f_1$ $f_2$ | renomme en $f_2$ le fichier de nom $f_1$ (comme la commande `mv` $f_1$ $f_2$). |

L'autre manière d'accéder à un fichier est par l'intermédiaire d'un descripteur. Un descripteur représente un pointeur vers un fichier, plus des informations comme la position courante de lecture/écriture dans ce fichier, des permissions sur ce fichier (peut-on lire ? peut-on écrire ?), et des drapeaux gouvernant le comportement des lectures et des écritures (écritures en ajout ou en écrasement, lectures bloquantes ou non). Les descripteurs sont représentés par des valeurs du type abstrait `file_descr`.

Les accès à travers un descripteur sont en grande partie indépendants des accès via le nom du fichier. En particulier, lorsqu'on a obtenu un descripteur sur un fichier, le fichier peut être détruit ou renommé, le descripteur pointera toujours sur le fichier d'origine.

Au lancement d'un programme, trois descripteurs ont été préalloués et liés aux variables `stdin`, `stdout` et `stderr` du module `unix` :

| | | |
|---|---|---|
| `stdin : file_descr` | | l'entrée standard du processus |
| `stdout : file_descr` | | la sortie standard du processus |
| `stderr : file_descr` | | la sortie d'erreur standard du processus |

Lorsque le programme est lancé depuis un interpréteur de commandes interactif et sans redirections, les trois descripteurs font référence au terminal. Mais si, par exemple, l'entrée a été redirigée par la notation $< f$, alors le descripteur `stdin` fait référence au fichier de nom $f$. De même pour les autres possibilités de redirection.

## 2.2 Ouverture d'un fichier

L'appel système `open` permet d'obtenir un descripteur sur un fichier d'un certain nom.

```
open : string -> open_flag list -> int -> file_descr
```

Le premier argument est le nom du fichier à ouvrir. Le deuxième argument est une liste de drapeaux pris dans le type énuméré `open_flag`, et décrivant dans quel mode le fichier doit être ouvert, et que faire s'il n'existe pas. Le troisième argument indique avec quels droits d'accès créer le fichier, le cas échéant. Le résultat est un descripteur de fichier pointant vers le fichier indiqué. La position de lecture/écriture est initialement fixée au début du fichier.

La liste des modes d'ouverture (deuxième argument) doit contenir exactement un des trois drapeaux suivants :

| | |
|---|---|
| `O_RDONLY` | ouverture en lecture seule |
| `O_WRONLY` | ouverture en lecture seule |
| `O_RDWR` | ouverture en lecture et en écriture |

Ces drapeaux conditionnent la possibilité de faire par la suite des opérations de lecture ou d'écriture à travers le descripteur. L'appel `open` échoue si on demande à ouvrir en écriture un fichier sur lequel le processus n'a pas le droit d'écrire, ou si on demande à ouvrir en lecture un fichier que le processus n'a pas le droit de lire. C'est pourquoi il ne faut pas ouvrir systématiquement en mode `O_RDWR`.

La liste des modes d'ouverture peut contenir en plus un ou plusieurs des drapeaux suivants :

| | |
|---|---|
| `O_APPEND` | ouverture en ajout |
| `O_CREAT` | créer le fichier s'il n'existe pas |
| `O_TRUNC` | tronquer le fichier à zéro s'il existe déjà |
| `O_EXCL` | échouer si le fichier existe déjà |

Si `O_APPEND` est fourni, le pointeur de lecture/écriture sera positionné à la fin du fichier avant chaque écriture. En conséquence, toutes les écritures s'ajouteront à la fin du fichier. Au contraire, sans `O_APPEND`, les écritures se font à la position courante (initialement, le début du fichier).

Si `O_TRUNC` est fourni, le fichier est tronqué au moment de l'ouverture : la longueur du fichier est ramenée à zéro, et les octets contenus dans le fichier sont perdus. Les écritures repartent donc d'un fichier vide. Au contraire, sans `O_TRUNC`, les écritures se font par-dessus les octets déjà présents, ou à la suite.

Si `O_CREAT` est fourni, le fichier est créé s'il n'existe pas déjà. Le fichier est créé avec une taille nulle, et avec pour droits d'accès les droits indiqués par le troisième argument, modifiés par le masque de création du processus. (Le masque de création est consultable et modifiable par la commande `umask`, et par l'appel système de même nom). Les droits sont codés sous forme de bits dans un entier (cf. la commande `chmod`) :

| Bit (octal) | Notation `ls -l` | Droit |
|---|---|---|
| `0o100` | `--x------` | exécution, pour le propriétaire |
| `0o200` | `-w-------` | écriture, pour le propriétaire |
| `0o400` | `r--------` | lecture, pour le propriétaire |
| `0o10` | `-----x---` | exécution, pour les membres des groupes du propriétaire |
| `0o20` | `----w----` | écriture, pour les membres des groupes du propriétaire |
| `0o40` | `---r----` | lecture, pour les membres des groupes du propriétaire |
| `0o1` | `--------x` | exécution, pour les autres utilisateurs |
| `0o2` | `-------w-` | écriture, pour les autres utilisateurs |
| `0o4` | `------r--` | lecture, pour les autres utilisateurs |
| `0o4000` | `--s------` | le bit `s` sur l'utilisateur |
| `0o2000` | `-----s---` | le bit `s` sur le groupe |

Le masque de création est codé de même. Tous les bits à 1 dans le masque de création sont mis à zéro dans les droits du fichier créé.

**Exemple :** la plupart des programmes prennent `0o666` comme troisième argument de `open`, c'est-à-dire `rw-rw-rw-` en notation symbolique. Avec le masque de création standard de `0o022`, le fichier est donc créé avec les droits `rw-r--r--`. Avec un masque plus confiant de `0o002`, le fichier est créé avec les droits `rw-rw-r--`.

Si `O_EXCL` est fourni, `open` échoue si le fichier existe déjà. Ce drapeau, employé en conjonction avec `O_CREAT`, permet d'utiliser des fichiers comme verrous (*locks*).[1] Un processus qui veut prendre

[1] Ce n'est pas possible si le fichier verrou réside sur une partition NFS, car NFS n'implémente pas correctement l'option `O_CREAT` de open.

le verrou appelle `open` sur le fichier avec les modes `O_EXCL` et `O_CREAT`. Si le fichier existe déjà, cela signifie qu'un autre processus détient le verrou. Dans ce cas, `open` déclenche une erreur, et il faut attendre un peu, puis réessayer. Si le fichier n'existe pas, `open` retourne sans erreur et le fichier est créé, empêchant les autres processus de prendre le verrou. Pour libérer le verrou, le processus qui le détient fait `unlink` dessus.

**Exemple :** pour se préparer à lire un fichier :

```
open filename [O_RDONLY] 0
```

Le troisième argument peut être quelconque, puisque `O_CREAT` n'est pas spécifié. On prend conventionnellement `0`. Pour écrire un fichier à partir de rien, sans se préoccuper de ce qu'il contenait éventuellement :

```
open filename [O_WRONLY; O_TRUNC; O_CREAT] 0o666
```

Si le fichier qu'on écrit va contenir du code exécutable (cas des fichiers créés par `ld`), ou un script de commandes, on ajoute les droits d'exécution dans le troisième argument :

```
open filename [O_WRONLY; O_TRUNC; O_CREAT] 0o777
```

Si le fichier qu'on écrit est confidentiel, comme par exemple les fichiers "boîte aux lettres" dans lesquels `mail` stocke les messages lus, on le crée en restreignant la lecture et l'écriture au propriétaire uniquement :

```
open filename [O_WRONLY; O_TRUNC; O_CREAT] 0o600
```

Pour se préparer à ajouter des données à la fin d'un fichier existant, et le créer vide s'il n'existe pas :

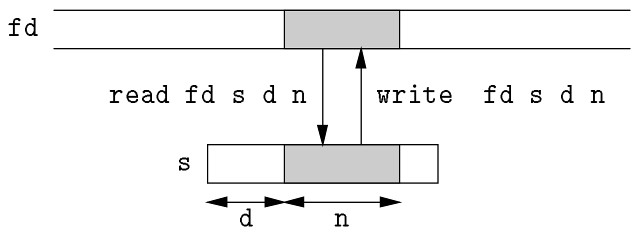
```
open filename [O_WRONLY; O_APPEND; O_CREAT] 0o666
```

## 2.3 Lecture et écriture

Les appels systèmes `read` et `write` permettent de lire et d'écrire les octets d'un fichier.

```
read  : file_descr -> string -> int -> int -> int
write : file_descr -> string -> int -> int -> int
```

Les deux appels ont la même interface. Le premier argument est le descripteur sur lequel la lecture ou l'écriture doit avoir lieu. Le deuxième argument est une chaîne de caractère contenant les octets à écrire (cas de `write`), ou dans laquelle vont être stockés les octets lus (cas de `read`). Le troisième argument est la position, dans la chaîne de caractères, du premier octet à écrire ou à lire. Le quatrième argument est le nombre d'octets à lire ou à écrire. Les troisième et quatrième arguments désignent donc une sous-chaîne de la chaîne passée en deuxième argument. (Cette sous-chaîne ne doit pas déborder de la chaîne d'origine ; `read` et `write` ne vérifient pas ce fait.)




L'entier renvoyé par `read` ou `write` est le nombre d'octets réellement lus ou écrits.

Les lectures et les écritures ont lieu à partir de la position courante de lecture/écriture. (Si le fichier a été ouvert en mode `O_APPEND`, cette position est placée à la fin du fichier avant toute écriture.) Cette position est avancée du nombre d'octets lus ou écrits.

Dans le cas d'une écriture, le nombre d'octets effectivement écrits est toujours le nombre d'octets demandés. Dans le cas où il n'est pas possible d'écrire les octets (si le disque est plein, par exemple), `write` déclenche une erreur. L'entier renvoyé par `write` peut donc être ignoré.[2]

**Exemple :** supposant `fd` lié à un descripteur ouvert en écriture,

```
write fd "Hello world!" 3 7
```

écrit les caractères “`lo worl`” dans le fichier correspondant, et renvoie 7.

Dans le cas d'une lecture, il se peut que le nombre d'octets effectivement lus soit strictement inférieur au nombre d'octets demandés. Premier cas : lorsque la fin du fichier est proche, c'est-à-dire lorsque le nombre d'octets entre la position courante et la fin du fichier est inférieur au nombre d'octets requis. En particulier, lorsque la position courante est sur la fin du fichier, `read` renvoie zéro. Cette convention “zéro égal fin de fichier” s'applique aussi aux lectures depuis des fichiers spéciaux ou des dispositifs de communication. Par exemple, `read` sur le terminal renvoie zéro si on frappe `ctrl-D` en début de ligne.

Deuxième cas où le nombre d'octets lus peut être inférieur au nombre d'octets demandés : lorsqu'on lit depuis un fichier spécial tel qu'un terminal, ou depuis un dispositif de communication comme un tuyau ou une prise. Par exemple, lorsqu'on lit depuis le terminal, `read` bloque jusqu'à ce qu'une ligne entière soit disponible. Si la longueur de la ligne dépasse le nombre d'octets requis, `read` retourne le nombre d'octets requis. Sinon, `read` retourne immédiatement avec la ligne lue, sans forcer la lecture d'autres lignes pour atteindre le nombre d'octets requis. (J'ai décrit ici le comportement par défaut du terminal ; on peut aussi mettre le terminal dans un mode de lecture caractère par caractère au lieu de ligne à ligne. Faire `man 4 termio` pour avoir tous les détails.)

**Exemple :** l'expression suivante lit au plus 100 caractères depuis l'entrée standard, et renvoie la chaîne des caractères lus.

```
let buffer = create_string 100 in
let n = read stdin buffer 0 100 in
  sub_string buffer 0 n
```

[2] La seule exception est lorsqu'on écrit sur un descripteur de fichiers qui référence un tuyau ou une prise, et qui est placé dans le mode entrées/sorties non bloquantes. Dans ce cas, les écritures peuvent être partielles. On ne parlera pas d'entrées/sorties non bloquantes dans ce cours.

**Exemple :** la fonction `really_read` ci-dessous a la même interface que `read`, mais fait plusieurs tentatives de lecture si nécessaire pour essayer de lire le nombre d'octets requis. Si, ce faisant, elle rencontre une fin de fichier, elle déclenche l'exception `End_of_file`.

```
let rec really_read fd buffer start length =
  if length <= 0 then () else
    match read fd buffer start length with
      0 -> raise End_of_file
    | r -> really_read fd buffer (start+r) (length-r);;
```

## 2.4 Fermeture d'un descripteur

L'appel système `close` ferme le descripteur passé en argument.

```
close : file_descr -> unit
```

Après qu'un descripteur a été fermé, toute tentative de lire, d'écrire, ou de faire quoi que ce soit avec ce descripteur échoue. Il est recommandé de fermer les descripteurs dès qu'ils ne sont plus utilisés. Ce n'est pas obligatoire ; en particulier, contrairement à ce qui se passe avec la bibliothèque standard `io`, il n'est pas nécessaire de fermer les descripteurs pour être certain que les écritures en attente ont été effectuées : les écritures faites avec `write` sont immédiatement transmises au noyau. D'un autre côté, le nombre de descripteurs qu'un processus peut allouer est limité par le noyau (à quelques dizaines par processus). Faire `close` sur un descripteur inutile permet de le désallouer, et donc d'éviter de tomber à court de descripteurs.

## 2.5 Exemple complet : copie de fichiers

On va programmer une commande `file_copy`, à deux arguments $f_1$ et $f_2$, qui recopie dans le fichier de nom $f_2$ les octets contenus dans le fichier de nom $f_1$.

```
#open "unix";;

let buffer_size = 8192;;
let buffer = create_string buffer_size;;

let file_copy input_name output_name =
  let fd_in = open input_name [O_RDONLY] 0 in
  let fd_out = open output_name [O_WRONLY; O_CREAT; O_TRUNC] 0o666 in
  let rec copy_loop () =
    match read fd_in buffer 0 buffer_size with
      0 -> ()
    | r -> write fd_out buffer 0 r; copy_loop () in
  copy_loop ();
  close fd_in;
  close fd_out;;

```

```
17 let main () =
18 if vect_length command_line = 3 then begin
19   file_copy command_line.(1) command_line.(2);
20   exit 0
21 end else begin
22   prerr_endline "Usage: file_copy <input_file> <output_file>";
23   exit 1
24 end;;
25
26 handle_unix_error main ();;
```

L'essentiel du travail est fait par la fonction `file_copy` des lignes 6–15. On commence par ouvrir un descripteur en lecture seule sur le fichier d'entrée (ligne 7), et un descripteur en écriture seule sur le fichier de sortie (ligne 8). Le fichier de sortie est tronqué s'il existe déjà (option `O_TRUNC`), et créé s'il n'existe pas (option `O_CREAT`), avec les droits `rw-rw-rw-` modifiés par le masque de création. (Ceci n'est pas satisfaisant : si on copie un fichier exécutable, on voudrait que la copie soit également exécutable. On verra plus loin comment attribuer à la copie les mêmes droits d'accès qu'à l'original.) Dans les lignes 9–13, on effectue la copie par blocs de `buffer_size` caractères. On demande à lire `buffer_size` caractères (lignes 10). Si `read` renvoie zéro, c'est qu'on a atteint la fin du fichier d'entrée, et la copie est terminée (ligne 11). Sinon (ligne 12), on écrit les `r` octets qu'on vient de lire sur le fichier de destination, et on recommence. Finalement, on ferme les deux descripteurs. Le programme principal (lignes 17–24) vérifie que la commande a reçu deux arguments, et les passe à la fonction `file_copy`.

Toute erreur pendant la copie, comme par exemple l'impossibilité d'ouvrir le fichier d'entrée, parce qu'il n'existe pas ou parce qu'il n'est pas permis de le lire, ou encore l'échec d'une écriture par manque de place sur le disque, se traduit par une exception `Unix_error` qui se propage jusqu'au niveau le plus externe du programme, où elle est interceptée et affichée par `handle_unix_error`.

**Exercice 1** *Ajouter une option* `-a` *au programme, telle que* `file_copy -a` $f_1$ $f_2$ *ajoute le contenu de* $f_1$ *à la fin de* $f_2$ *si* $f_2$ *existe déjà.*

## 2.6 Coût des appels système. Les tampons

Dans l'exemple `file_copy`, les lectures se font par blocs de 8192 octets. Pourquoi pas octet par octet ? ou mégaoctet par mégaoctet ? Pour des raisons d'efficacité. La figure 2.1 montre la vitesse de copie, en octets par seconde, du programme `file_copy`, quand on fait varier la taille des blocs (la variable `buffer_size`) de 1 octet a 1 mégaoctet, en doublant à chaque fois.

Pour de petites tailles de blocs, la vitesse de copie est à peu près proportionnelle à la taille des blocs. Cependant, la quantité de données transférées est la même quelle que soit la taille des blocs. L'essentiel du temps ne passe donc pas dans le transfert de données proprement dit, mais dans la gestion de la boucle `copy_loop`, et dans les appels `read` et `write`. En mesurant plus finement, on voit que ce sont les appels `read` et `write` qui prennent l'essentiel du temps. On en conclut donc qu'un appel système, même lorsqu'il n'a pas grand chose à faire (`read` d'un caractère), prend un

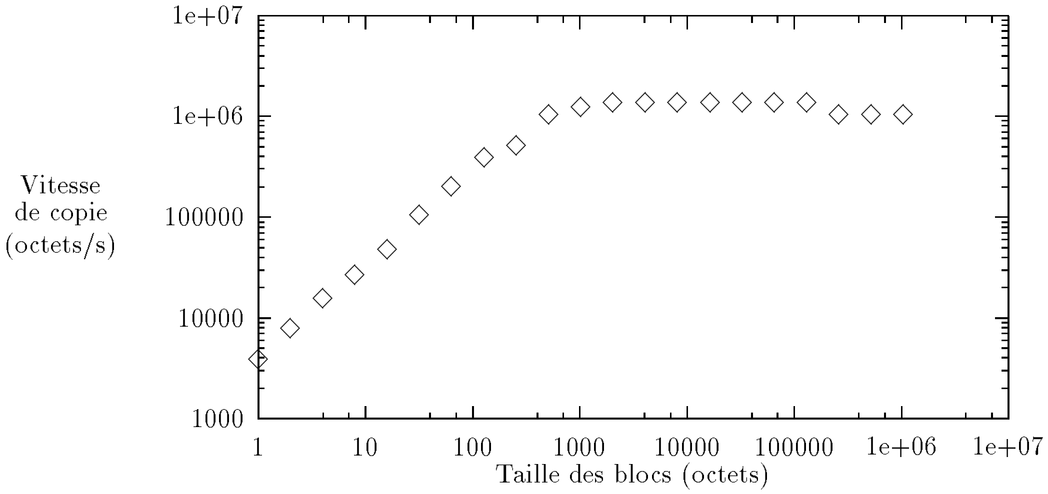



Figure 2.1 : Vitesse de copie en fonction de la taille des blocs

temps minimum d'environ 200 microsecondes (sur la machine employée pour faire le test). Pour des blocs d'entrée/sortie de petite taille, c'est ce temps d'appel système qui prédomine.

Pour des blocs plus gros, entre 1K et 128K, la vitesse est constante et maximale. Ici, le temps lié aux appels systèmes et à la boucle de copie est petit devant le temps de transfert des données, ce dernier étant indépendant de la taille des blocs.

Enfin, pour de très gros blocs (256K ou plus), la vitesse est toujours indépendante de la taille des blocs, mais elle est légèrement en-dessous du maximum. Entre en jeu ici le temps nécessaire pour allouer le bloc et lui attribuer des pages de mémoire réelle au fur et à mesure qu'il se remplit.

Moralité : un appel système, même s'il fait très peu de travail, coûte cher — beaucoup plus cher qu'un appel de fonction normale : en gros, de 100 microsecondes à 1 milliseconde par appel système, suivant les architectures. Il est donc important d'éviter de faire des appels système trop fréquents. En particulier, les opérations de lecture et d'écriture doivent se faire par blocs de taille suffisante, et non caractère par caractère.

Dans des exemples comme `file_copy`, il n'est pas difficile de faire les entrées/sorties par gros blocs. En revanche, d'autres types de programmes s'écrivent naturellement avec des entrées caractère par caractère (exemples : lecture d'une ligne depuis un fichier, analyse lexicale), et des sorties de quelques caractères à la fois (exemple : affichage d'un nombre). Pour répondre aux besoins de ces programmes, la plupart des systèmes fournissent des bibliothèques d'entrées-sorties, qui intercalent une couche de logiciel supplémentaire entre l'application et le système d'exploitation. Par exemple, en Caml Light, on dispose du module `io` de la bibliothèque standard, qui fournit deux types abstraits `in_channel` et `out_channel`, analogues aux descripteurs de fichiers, et des opérations sur ces types, comme `input_char`, `input_line`, `output_char`, ou `output_string`. Cette couche supplémentaire utilise des tampons (*buffers*) pour transformer des suites de lectures ou d'écritures caractère par caractère en une lecture ou une écriture d'un bloc. On obtient donc de bien meilleures performances pour les programmes qui procèdent caractère par caractère. De plus, cette couche supplémentaire permet une plus grande portabilité des programmes : il suffit d'adapter cette bibliothèque aux appels système fournis par un autre système d'exploitation, et tous les programmes qui utilisent la bibliothèque sont immédiatement portables vers cet autre système d'exploitation.

## 2.7 Exemple complet : une petite bibliothèque d'entrées-sorties

Pour illustrer les techniques de lecture/écriture par tampon, voici une implémentation simple d'un fragment de la bibliothèque `io` de Caml Light. L'interface est la suivante :

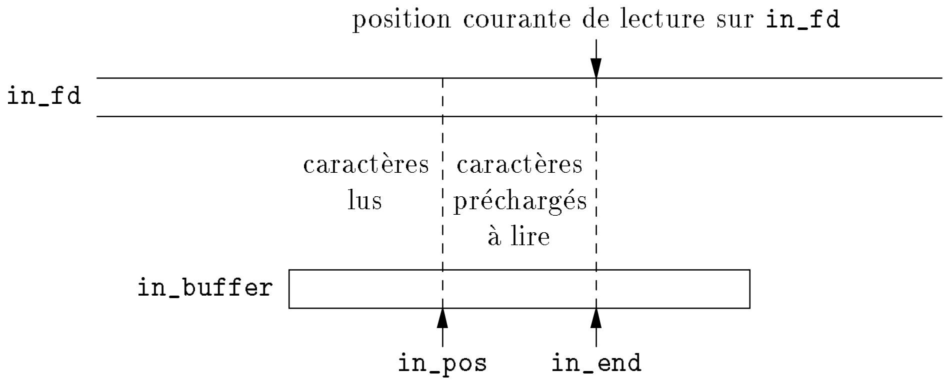
```
type in_channel;;
exception End_of_file;;
value open_in : string -> in_channel
  and input_char : in_channel -> string
  and close_in : in_channel -> unit;;
type out_channel;;
value open_out : string -> out_channel
  and output_char : out_channel -> string -> unit
  and close_out : out_channel -> unit;;
```

Commençons par la partie "lecture". Le type abstrait `in_channel` est implémenté comme suit :

```
#open "unix";;

type in_channel =
  { in_buffer: string;
    in_fd: file_descr;
    mutable in_pos: int;
    mutable in_end: int };;
```

La chaîne de caractères du champ `in_buffer` est le tampon proprement dit. Le champ `in_fd` est un descripteur de fichier (Unix), ouvert sur le fichier en cours de lecture. Le champ `in_pos` est la position courante de lecture dans le tampon. Le champ `in_end` est le nombre de caractères valides dans le tampon.



Les champs `in_pos` et `in_end` vont être modifiés en place à l'occasion des opérations de lecture ; on les déclare donc `mutable`.

```
let buffer_size = 8192;;
let open_in filename =
  { in_buffer = create_string buffer_size;
    in_fd = open filename [O_RDONLY] 0;
    in_pos = 0;
    in_end = 0 };;
```

À l'ouverture d'un fichier en lecture, on crée le tampon avec une taille raisonnable (suffisamment grande pour ne pas faire d'appels système trop souvent ; suffisamment petite pour ne pas gâcher de mémoire), et on initialise le champ `in_fd` par un descripteur de fichier Unix ouvert en lecture seule sur le fichier en question. Le tampon est initialement vide (il ne contient aucun caractère du fichier) ; le champ `in_end` est donc initialisé à zéro.

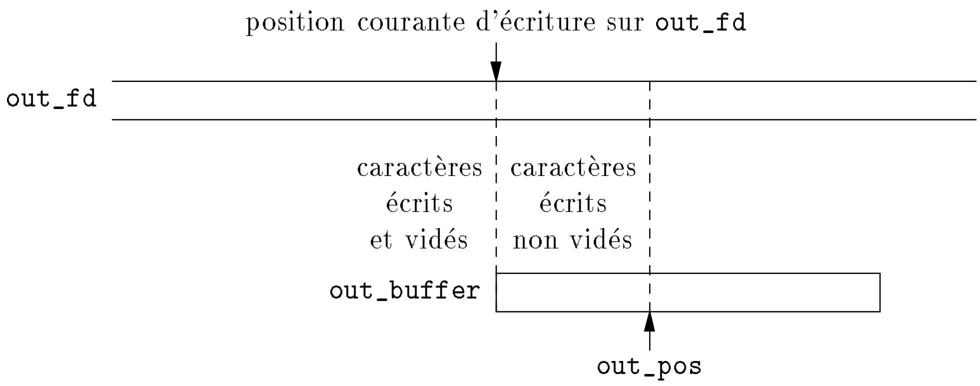
```
let input_char chan =
  if chan.in_pos < chan.in_end then begin
    let c = nth_char chan.in_buffer chan.in_pos in
      chan.in_pos <- chan.in_pos + 1;
      c
  end else begin
    match read chan.in_fd chan.in_buffer 0 buffer_size
    with 0 -> raise End_of_file
       | r -> chan.in_end <- r;
              chan.in_pos <- 1;
              nth_char chan.in_buffer 0
  end;;
```

Pour lire un caractère depuis un `in_channel`, de deux choses l'une. Ou bien il reste au moins un caractère dans le tampon ; c'est-à-dire, le champ `in_pos` est strictement inférieur au champ `in_end`. Alors on renvoie le prochain caractère du tampon, celui à la position `in_pos`, et on incrémente `in_pos`. Ou bien le tampon est vide. On fait alors un appel système `read` pour remplir le tampon. Si `read` rentourne zéro, c'est que la fin du fichier a été atteinte ; on déclenche alors l'exception `End_of_file`. Sinon, on place le nombre de caractères lus dans le champ `in_end`. (On peut avoir obtenu moins de caractères que demandé, et donc le tampon peut être partiellement rempli.) Et on renvoie le premier des caractères lus.

```
let close_in chan =
  close chan.in_fd;;
```

La fermeture d'un `in_channel` se réduit à la fermeture du descripteur Unix sous-jacent.

La partie "écriture" est très proche de la partie "lecture". La seule dissymétrie est que le tampon contient maintenant des écritures en retard, et non plus des lectures en avance.




```
let out_channel =
  { out_buffer: string;
    out_fd: file_descr;
    mutable out_pos: int };;

let open_out filename =
  { out_buffer = create_string 8192;
    out_fd = open filename [O_WRONLY; O_TRUNC; O_CREAT] 0o666;
    out_pos = 0 };;

let output_char chan c =
  if chan.out_pos < string_length chan.out_buffer then begin
    set_nth_char chan.out_buffer chan.out_pos c;
    chan.out_pos <- chan.out_pos + 1
  end else begin
    write chan.out_fd chan.out_buffer 0 chan.out_pos;
    set_nth_char chan.out_buffer 0 c;
    chan.out_pos <- 1
  end;;

let close_out chan =
  write chan.out_fd chan.out_buffer 0 chan.out_pos;
  close chan.out_fd;;
```

Pour écrire un caractère sur un `out_channel`, ou bien le tampon n'est pas plein, et on se contente de stocker le caractère dans le tampon à la position `out_pos`, et d'avancer `out_pos` ; ou bien le tampon est plein, et dans ce cas on le vide dans le fichier par un appel `write`, puis on stocke le caractère à écrire au début du tampon.

Quand on ferme un `out_channel`, il ne faut pas oublier de vider le contenu du tampon (les caractères entre les positions 0 incluse et `out_pos` exclue) dans le fichier. Autrement, les écritures effectuées depuis la dernière vidange seraient perdues.

**Exercice 2** *Implémenter une fonction*

```
output_string : out_channel -> string -> unit
```

*qui se comporte comme une série de* `output_char` *sur chaque caractère de la chaîne, mais est plus efficace.*

## 2.8 Positionnement

L'appel système `lseek` permet de changer la position courante de lecture et d'écriture.

```
lseek : file_descr -> int -> seek_command -> int
```

Le premier argument est le descripteur qu'on veut positionner. Le deuxième argument est la position désirée. Il est interprété différemment suivant la valeur du troisième argument, qui indique le type de positionnement désiré :

| | |
|---|---|
| `SEEK_SET` | Positionnement absolu. Le deuxième argument est le numéro du caractère où se placer. Le premier caractère d'un fichier est à la position zéro. |
| `SEEK_CUR` | Positionnement relatif à la position courante. Le deuxième argument est un déplacement par-rapport à la position courante. Il peut être négatif aussi bien que positif. |
| `SEEK_END` | Positionnement relatif à la fin du fichier. Le deuxième argument est un déplacement par-rapport à la fin du fichier. Il peut être négatif aussi bien que positif. |

L'entier renvoyé par `lseek` est la position absolue du pointeur de lecture/écriture (après que le positionnement a été effectué).

Une erreur se déclenche si la position absolue demandée est négative. En revanche, la position demandée peut très bien être située après la fin du fichier. Juste après un tel positionnement, un `read` renvoie zéro (fin de fichier atteinte) ; un `write` étend le fichier par des zéros jusqu'à la position demandée, puis écrit les données fournies.

**Exemple :** pour se placer sur le millième caractère d'un fichier :

```
lseek fd 1000 SEEK_SET
```

Pour reculer d'un caractère :

```
lseek fd (-1) SEEK_CUR
```

Pour connaître la taille d'un fichier :

```
let file_size = lseek fd 0 SEEK_END in ...
```

Pour les descripteurs ouverts en mode `O_APPEND`, le pointeur de lecture/écriture est automatiquement placé à la fin du fichier avant chaque écriture. L'appel `lseek` ne sert donc à rien pour écrire sur un tel descripteur ; en revanche, il est bien pris en compte pour la lecture.

Le comportement de `lseek` est indéterminé sur certains types de fichiers pour lesquels l'accès direct est absurde : les dispositifs de communication (tuyaux, prises), mais aussi la plupart des fichiers spéciaux (*devices*), comme par exemple le terminal. Dans la plupart des implémentations d'Unix, un `lseek` sur de tels fichiers est simplement ignoré : le pointeur de lecture/écriture est positionné, mais les opérations de lecture et d'écriture l'ignorent. Sur certaines implémentations, `lseek` sur un tuyau ou sur une prise déclenche une erreur.

**Exercice 3** *La commande* `tail` *affiche les N dernières lignes d'un fichier. Comment l'implémenter efficacement si le fichier en question est un fichier normal ? Comment faire face aux autres types de fichiers ? Comment ajouter l'option* `-f` *? (cf.* `man tail`*).*

## 2.9 Exemple complet : lire un message de Yaf

(Pour ceux qui l'ignorent, Yaf est une ré-implémentation, par Damien Doligez et l'auteur, du toujours populaire Forum de Roland Dirlewanger, lui-même ré-implémentation du Forum de Harald Wertz, lui-même inspiré d'un machin qui tournait sur la machine Multics de l'INRIA, et dont l'origine se perd dans la nuit des temps. Un programme "maturité et valeurs sûres", donc.)

Les messages de Yaf sont stockés sous le format suivant. Au groupe `foo` sont associés deux fichiers : le fichier d'index `/.../index/foo` et le fichier des textes `/.../texts/foo`. (Ici, `/...` représente un chemin qui dépend des installations). Le fichier de textes contient les textes des messages, mis bout à bout sans séparation. Le fichier d'index contient une suite d'enregistrements de taille fixe, qui décrivent les messages du groupe, par ordre croissant des numéros de messages. Ces enregistrements font 48 octets, et sont décrits par la structure C suivante :

```
typedef struct {   /* Structure d'un descripteur de message: */
  long ptr;        /* position du début du texte */
  long date;       /* date où le message a été placardé */
  long rep;        /* si non nul, numéro du message auquel celui-ci répond */
  long length;     /* longueur du texte */
  char author[30]; /* nom de l'auteur du message */
} mesg_desc;
```

Le champ `ptr` est un pointeur vers le fichier des textes : c'est la position du premier caractère du message décrit. Cette organisation en deux fichiers permet d'accéder en temps constant à un message par son numéro, sans pour autant gâcher de la place ou limiter la taille des messages. De plus, on peut facilement faire du compactage sur le fichier des textes, pour récupérer la place occupée par les messages enlevés, par exemple.

Le programme `read_yaf` qui suit prend en argument un nom de groupe et un numéro de message, et affiche le message sur sa sortie standard. On commence par définir un type concret des descripteurs de messages.

```
#open "sys";;
#open "unix";;
type mesg_desc =
  { ptr: int;
```

```
    date: int;
    rep: int;
    length: int;
    author: string };;
let sizeof_mesg_desc = 48;;
let yaf_dir = "/usr/local/games/lib/yafdir";;
```

La fonction `read_desc` ci-dessous lit le descripteur d'un message, étant donné le nom du groupe et le numéro du message dans ce groupe.

```
let read_desc group num =
  let index_fd =
    try
      open (yaf_dir ^ "/index/" ^ group) [O_RDONLY] 0
    with
      Unix_error(ENOENT,_,_) ->
        prerr_endline ("Le groupe " ^ group ^ " n'existe pas.");
        exit 1 in
  let last_msg = (lseek index_fd 0 SEEK_END) / sizeof_mesg_desc in
  if num < 1 or num > last_msg then begin
    prerr_endline ("Pas de message " ^ string_of_int num ^
                   " dans le groupe " ^ group ^ ".");
    prerr_endline ("(Le dernier message a le numéro " ^
                   string_of_int last_msg ^ ".)");
    exit 1
  end;
  lseek index_fd ((num-1) * sizeof_mesg_desc) SEEK_SET
  let buffer = create_string sizeof_mesg_desc in
  read index_fd buffer 0 sizeof_mesg_desc;
  close index_fd;
  { ptr = peek__long buffer 0;
    date = peek__long buffer 4;
    rep = peek__long buffer 8;
    length = peek__long buffer 12;
    author = peek__cstring buffer 16 }
;;
```

On ouvre le fichier d'index en lecture. S'il n'existe pas, c'est que le groupe demandé n'existe pas. Ensuite, on détermine le nombre d'enregistrements contenus dans l'index, c'est-à-dire le numéro du dernier message (le premier message a, bêtement, le numéro 1) : c'est la taille du fichier, comme déterminée par `lseek index_fd 0 SEEK_END`, divisée par la taille des enregistrements. Ceci permet de savoir si le message demandé existe ou non. Si oui, on positionne `index_fd` sur le premier octet du descripteur du message, et on lit l'enregistrement le décrivant dans la chaîne de caractères `buffer`. (Comme le fichier d'index est un fichier normal et comme il est supposé contenir des enregistrements non tronqués, `read` doit lire en un coup les `sizeof_mesg_desc` octets demandés.) Pour extraire les valeurs des différents champs, on utilise les fonctions du module `peek`, qui permettent de lire des objets C à partir de leur représentation binaire dans une chaîne de caractères.

La fonction `print_msg` ci-dessous affiche un message à partir de son descripteur.

```
let print_msg group desc =
  print_string ("Auteur: " ^ desc.author); print_newline();
  let date = localtime desc.date in
    print_string "Date: ";
    print_int date.tm_mday; print_string "/";
    print_int (date.tm_mon + 1); print_string "/";
    print_int date.tm_year; print_string " ";
    print_int date.tm_hour; print_string ":";
    print_int date.tm_min; print_newline();
  if desc.rep <> 0 then begin
    print_string "Réponse a [";
    print_int desc.rep;
    print_string "]"; print_newline()
  end;
  let text_fd = open (yaf_dir ^ "/texts/" ^ group) [O_RDONLY] 0 in
  lseek text_fd desc.ptr SEEK_SET;
  let buffer = create_string desc.length in
  read text_fd buffer 0 desc.length;
  print_string buffer;
  close text_fd
;;
```

La première partie de la fonction affiche les informations contenues dans le descripteur. La fonction `localtime` convertit une date exprimée en nombre de secondes depuis le 1$^{\text{ier}}$ janvier 1970 minuit (telle que renvoyée par la fonction `time`) en une date au format année, mois, jour, heure, minutes, secondes. La deuxième partie de la fonction ouvre le fichier des textes en lecture, se positionne sur le premier caractère du message, le lit dans la chaîne `buffer`, et l'affiche.

Voici enfin le programme principal :

```
let usage() =
  prerr_endline "Syntaxe: read_yaf <nom du groupe> <numero du message>";
  io__exit 2;;

let main() =
  if vect_length command_line <> 3 then usage();
  let group = command_line.(1)
  and num = try int_of_string command_line.(2) with Failure _ -> usage() in
  print_msg group (read_desc group num);
  io__exit 0;;

handle_unix_error main ();;
```

## 2.10 Détermination du type d'un fichier

Les appels système `stat`, `lstat` et `fstat` permettent de déterminer le type d'un fichier, plus un certain nombre d'informations supplémentaires.

```
stat  : string -> stats
lstat : string -> stats
fstat : file_descr -> stats
```

Les appels `stat` et `lstat` prennent un nom de fichier en argument. L'appel `fstat` prend en argument un descripteur ouvert sur le fichier dont on veut les informations. La différence entre `stat` et `lstat` se voit sur les liens symboliques : `lstat` renvoie les informations sur le lien symbolique lui-même, alors que `stat` renvoie les informations sur le fichier vers lequel pointe le lien symbolique.

Le résultat de ces trois appels est un objet enregistrement (*record*) avec les champs suivants :

| | |
|---|---|
| `st_dev : int` | Un identificateur de la partition disque où se trouve le fichier |
| `st_ino : int` | Un identificateur du fichier à l'intérieur de sa partition. Le couple (`st_dev`, `st_ino`) identifie de manière unique un fichier dans le système de fichier. |
| `st_kind : file_kind` | Le type du fichier. Le type `file_kind` est un type concret énuméré, de constructeurs :<br>`S_REG` fichier normal<br>`S_DIR` répertoire<br>`S_CHR` fichier spécial de type caractère<br>`S_BLK` fichier spécial de type bloc<br>`S_LNK` lien symbolique<br>`S_FIFO` tuyau<br>`S_SOCK` prise |
| `st_perm : int` | Les droits d'accès au fichier |
| `st_nlink : int` | Pour un répertoire : le nombre d'entrées dans le répertoire. Pour les autres : le nombre de liens durs sur ce fichier. |
| `st_uid : int` | Le numéro de l'utilisateur propriétaire du fichier. |
| `st_gid : int` | Le numéro du groupe propriétaire du fichier. |
| `st_rdev : int` | L'identificateur du "device" associé (pour les fichiers spéciaux). |
| `st_size : int` | La taille du fichier, en octets. |
| `st_atime : int` | La date du dernier accès au contenu du fichier. (En secondes depuis le $1^{\text{ier}}$ janvier 1970, minuit). |
| `st_mtime : int` | La date de la dernière modification du contenu du fichier. (Idem.) |
| `st_ctime : int` | La date du dernier changement de l'état du fichier : ou bien écriture dans le fichier, ou bien changement des droits d'accès, du propriétaire, du groupe propriétaire, du nombre de liens. |

## 2.11 Opérations spécifiques à certains types de fichiers

### 2.11.1 Fichiers normaux

On peut raccourcir un fichier normal par les appels suivants :

```
truncate  : string -> int -> unit
ftruncate : file_descr -> int -> unit
```

Le premier argument désigne le fichier à tronquer (par son nom, ou via un descripteur ouvert sur ce fichier). Le deuxième argument est la taille désirée. Toutes les données situées à partir de cette position sont perdues.

### 2.11.2 Répertoires

Seul le noyau écrit dans les répertoires (lorsque des fichiers sont créés). Il est donc interdit d'ouvrir un répertoire en écriture. On peut ouvrir un répertoire en lecture, et le lire avec `read`. Cependant, le format des entrées des répertoires varie suivant les versions d'Unix, et il est souvent complexe. Les fonctions suivantes permettent de lire séquentiellement un répertoire de manière portable :

```
opendir  : string -> dir_handle
readdir  : dir_handle -> string
closedir : dir_handle -> unit
```

La fonction `opendir` renvoie un descripteur de lecture sur un répertoire. La fonction `readdir` lit la prochaine entrée d'un répertoire (ou déclenche l'exception `End_of_file` si la fin du répertoire est atteinte). La chaîne renvoyée est un nom de fichier relatif au répertoire lu.

Pour créer un répertoire, ou détruire un répertoire vide, on dispose de :

```
mkdir : string -> int -> unit
rmdir : string -> unit
```

Le deuxième argument de `mkdir` encode les droits d'accès donnés au nouveau répertoire.

### 2.11.3 Liens symboliques

La plupart des opérations sur fichiers "suivent" les liens symboliques : c'est-à-dire, elles s'appliquent au fichier vers lequel pointe le lien symbolique, et non pas au lien symbolique lui-même. Exemples : `open`, `stat`, `truncate`, `opendir`. On dispose de deux opérations sur les liens symboliques :

```
symlink  : string -> string -> unit
readlink : string -> string
```

L'appel `symlink` $f_1$ $f_2$ créé le fichier $f_2$ comme étant un lien symbolique vers $f_1$. (Comme la commande `ln -s` $f_1$ $f_2$.) L'appel `readlink` renvoie le contenu d'un lien symbolique, c'est-à-dire le nom du fichier vers lequel il pointe.

### 2.11.4 Fichiers spéciaux

Les fichiers spéciaux ont des comportements assez variables en réponse aux appels système généraux sur fichiers. La plupart des fichiers spéciaux (terminaux, lecteurs de bandes, disques, ...) obéissent à `read` et `write` de la manière évidente (mais parfois avec des restrictions sur le nombre d'octets écrits ou lus). Beaucoup de fichiers spéciaux ignorent `lseek`.

L'appel système `ioctl` donne accès aux possibilités spécifiques à un fichier spécial. Exemple de telles possibilités : pour un dérouleur de bande, le rembobinage ou l'avance rapide ; pour un terminal, le choix du mode d'édition de ligne, des caractères spéciaux, des paramètres de la liaison série (vitesse, parité, etc). L'appel `ioctl` prend en argument un descripteur de fichier ouvert sur le fichier spécial en question, un entier qui code la commande à exécuter sur ce fichier spécial, et un troisième argument qui est l'argument de cette commande. Le type et la signification de cet argument dépendent de la commande : ce peut être un paramètre entier, ou bien un (pointeur vers un) enregistrement ; certaines commandes modifient l'enregistrement pour y stocker des résultats. L'appel `ioctl` ne peut donc pas être correctement typé en ML, puisque le type de son troisième argument dépend de la valeur de son deuxième argument (le code de la commande). Dans la bibliothèque `unix`, il se présente sous la forme de deux appels :

```
ioctl_int : file_descr -> int -> int -> int
ioctl_ptr : file_descr -> int -> string -> int
```

La première forme passe un argument entier. Le deuxième forme passe un pointeur vers une chaîne de caractère, qui est la représentation binaire d'un enregistrement d'arguments ou de résultats. (La chaîne peut être modifiée en place par la commande.) Il faut utiliser les fonctions des modules `peek` et `poke` pour coder et décoder le *record*. On trouvera en annexe B un exemple d'utilisation de `ioctl_ptr` pour donner accès aux paramètres d'un terminal. L'utilisation de `ioctl_ptr` est peu sûre : la structure des *records* et les codes de commandes sont hautement dépendants de la machine.

## 2.12 Exemple complet : copie récursive de fichiers

On va étendre la commande `file_copy` pour copier, en plus des fichiers normaux, les liens symboliques et les répertoires. Pour les répertoires, on copie récursivement leur contenu.

On commence par une fonctionnelle d'intérêt général : un itérateur sur les entrées d'un répertoire.

```
#open "unix";;

let do_dir f dirname =
  let d = opendir dirname in
  try
    while true do
      f(readdir d)
    done
  with End_of_file ->
    closedir d
```

```
;;
```

La fonction `set_infos` ci-dessous modifie le propriétaire, les droits d'accès et les dates de dernier accès/dernière modification d'un fichier. Son but est de préserver ces informations pendant la copie.

```
let set_infos filename infos =
  utime filename infos.st_atime infos.st_mtime;
  chmod filename infos.st_perm;
  try
    chown filename infos.st_uid infos.st_gid
  with Unix_error(EPERM,_,_) ->
    ()
;;
```

L'appel système `utime` modifie les dates d'accès et de modification. L'appel `chmod` modifie les droits d'accès. Enfin, l'appel `chown` modifie le propriétaire (deuxième argument) et le groupe propriétaire (troisième argument). Seul le super-utilisateur a le droit de changer arbitrairement ces informations. Pour les utilisateurs normaux, il y a un certain nombres de cas où `chown` va échouer avec une erreur "permission denied". On rattrape donc cette erreur-là et on l'ignore.

On réutilise la fonction `file_copy` de l'exemple du même nom pour copier les fichiers normaux.

```
let file_copy input_name output_name =
  ...
```

Voici la fonction récursive principale.

```
let rec copy source dest =
  let infos = lstat source in
  match infos.st_kind with
    S_REG ->
      file_copy source dest;
      set_infos dest infos
  | S_LNK ->
      let link = readlink source in
      symlink link dest
  | S_DIR ->
      mkdir dest 0o200;
      do_dir
        (fun file ->
          if file = "." or file = ".." then () else
            copy (source ^ "/" ^ file) (dest ^ "/" ^ file))
        source;
      set_infos dest infos
  | _ ->
      prerr_endline ("Can't cope with special file " ^ source)
;;
```

On commence par lire les informations du fichier source. Si c'est un fichier normal, on copie son contenu avec `file_copy`, puis ses informations avec `set_infos`. Si c'est un lien symbolique, on lit ce vers quoi il pointe, et on crée un lien qui pointe vers la même chose. Si c'est un répertoire, on crée un répertoire comme destination, puis on lit les entrées du répertoire source (en ignorant les entrées `.` et `..`, qu'il ne faut certainement pas copier), et on appelle récursivement `copy` pour chaque entrée. Les autres types de fichiers sont ignorés, avec un message d'avertissement.

Le programme principal est sans surprise :

```
let main () =
  if vect_length command_line <> 3 then begin
    prerr_endline "Usage: copy_rec <source> <destination>";
    io__exit 2
  end else begin
    copy command_line.(1) command_line.(2);
    io__exit 0
  end
;;
handle_unix_error main ();;
```

**Exercice 4** *Copier intelligemment les liens durs. Tel que présenté ci-dessus,* `file_copy` *duplique N fois un même fichier qui apparaît sous N noms différents dans la hiérarchie de fichiers à copier. Essayer de détecter cette situation, de ne copier qu'une fois le fichier, et de faire des liens durs dans la hiérarchie de destination.*

# Chapitre 3

# Les processus

Un processus est un programme en train de s'exécuter. Un processus se compose d'un texte de programme (du code machine) et d'un état du programme (point de contrôle courant, valeur des variables, pile des retours de fonctions en attente, descripteurs de fichiers ouverts, etc.)

Cette partie présente les appels systèmes Unix permettant de créer de nouveaux processus et de leur faire exécuter d'autres programmes.

## 3.1 Création de processus

L'appel système `fork` permet de créer un processus.

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

Le nouveau processus (appelé "le processus fils") est un clone presque parfait du processus qui a appelé `fork` (dit "le processus père" [1]) : les deux processus (père et fils) exécutent le même texte de programme, sont initialement au même point de contrôle (le retour de `fork`), attribuent les mêmes valeurs aux variables, ont des piles de retours de fonctions identiques, et détiennent les mêmes descripteurs de fichiers ouverts sur les mêmes fichiers. Ce qui distingue les deux processus, c'est la valeur renvoyée par `fork` : zéro dans le processus fils, un entier non nul dans le processus père. En testant la valeur de retour de `fork`, un programme peut donc déterminer s'il est dans le processus père ou dans le processus fils, et se comporter différemment dans les deux processus :

```
match fork() with
  0 -> (* code execute uniquement par le fils *)
| _ -> (* code execute uniquement par le pere *)
```

L'entier non nul renvoyé par `fork` dans le processus père est l'identificateur du processus fils. Chaque processus est identifié dans le noyau par un numéro, l'identificateur du processus (*process id*). Un processus peut obtenir son numéro d'identification par l'appel `getpid()`.

[1] Les dénominations politiquement correctes sont "processus parent" et "processus enfant". Il est aussi accepté d'alterner avec "incarnation mère" et "incarnation fille".

Le processus fils est initialement dans le même état que le processus père (mêmes valeurs des variables, mêmes descripteurs de fichiers ouverts). Cet état n'est pas partagé entre le père et le fils, mais seulement dupliqué au moment du `fork`. Par exemple, si une variable est liée à une référence avant le `fork`, une copie de cette référence et de son contenu courant est faite au moment du `fork` ; après le `fork`, chaque processus modifie indépendamment "sa" référence, sans que cela se répercute sur l'autre processus. De même, les descripteurs de fichiers ouverts sont dupliqués au moment du `fork` : les descripteurs font dont référence aux mêmes fichiers dans les deux processus, mais chaque processus a sa position courante de lecture/écriture dans le fichier ; un processus peut donc faire `lseek` sur un de ces descripteurs sans affecter l'autre processus.

## 3.2 Exemple complet : la commande `leave`

La commande `leave` *hhmm* rend la main immédiatement, mais crée un processus en tâche de fond qui, à l'heure *hhmm*, rappelle qu'il est temps de partir.

```
1  #open "sys";;
2  #open "unix";;
3
4  let main () =
5    let hh = int_of_string (sub_string command_line.(1) 0 2)
6    and mm = int_of_string (sub_string command_line.(1) 2 2) in
7    let now = localtime(time()) in
8    let delay = (hh - now.tm_hour) * 3600 + (mm - now.tm_min) * 60 in
9    if delay <= 0 then begin
10     print_string "Hey! That time has already passed!";
11     print_newline();
12     exit 0
13   end;
14   if fork() <> 0 then exit 0;
15   sleep delay;
16   print_string "\007\007\007Time to leave!";
17   print_newline();
18   exit 0;;
19
20 handle_unix_error main ();;
```

On commence par un parsing rudimentaire de la ligne de commande, pour extraire l'heure voulue. On calcule ensuite la durée d'attente, en secondes (ligne 8). (L'appel `time` renvoie la date courante, en secondes depuis le premier janvier 1970, minuit. La fonction `localtime` transforme ça en année/mois/jour/heures/minutes/secondes.) On crée alors un nouveau processus par `fork`. Le processus père (celui pour lequel `fork` renvoie un entier non nul) termine immédiatement. Le shell qui a lancé `leave` rend donc aussitôt la main à l'utilisateur. Le processus fils (celui pour lequel `fork` renvoie zéro) continue à tourner. Il ne fait rien pendant la durée indiquée (appel `sleep`), puis affiche son message et termine.

## 3.3 Attente de la terminaison d'un processus

L'appel système `wait` attend qu'un des processus fils créés par `fork` ait terminé, et renvoie des informations sur la manière dont ce processus a terminé. Il permet la synchronisation père-fils, ainsi qu'une forme très rudimentaire de communication du fils vers le père.

```
wait : unit -> int * process_status
```

Le premier résultat est le numéro du processus fils intercepté par `wait`. Le deuxième résultat peut être :

| | |
|---|---|
| `WEXITED`($r$) | le processus fils a terminé normalement (par `exit` ou en arrivant au bout du programme) ; $r$ est le code de retour (l'argument passé à `exit`) |
| `WSIGNALED`($sig, core$) | le processus fils a été tué par un signal (ctrl-C, `kill`, etc. ; voir plus bas pour les signaux) ; *sig* identifie le type du signal, et le booléen *core* indique si le processus a fait "core dumped" ou non |
| `WSTOPPED`($sig$) | le processus fils a été stoppé par le signal *sig* ; ne se produit que dans le cas très particulier où un processus (typiquement un debugger) est en train de surveiller l'exécution d'un autre (par l'appel `ptrace`). |

Si un des processus fils a déjà terminé au moment où le père exécute `wait`, l'appel `wait` retourne immédiatement. Sinon, `wait` bloque le père jusqu'à ce qu'un des fils termine (comportement dit "de rendez-vous"). Pour attendre $N$ fils, il faut répéter $N$ fois `wait`.

**Exemple :** la fonction `fork_search` ci-dessous fait une recherche linéaire dans un vecteur, en deux processus. Elle s'appuie sur la fonction `simple_search`, qui fait la recherche linéaire simple.

```
exception Found;;

let simple_search cond v =
  try
    for i = 0 to vect_length v - 1 do
      if cond v.(i) then raise Found
    done;
    false
  with Found -> true;;

let fork_search cond v =
  let n = vect_length v in
  match fork() with
    0 ->
      let found = simple_search cond (sub_vect v (n/2) (n-n/2)) in
      exit (if found then 1 else 0)
  | _ ->
      let found = simple_search cond (sub_vect v 0 (n/2)) in
```

```
19        match wait() with
20          (pid, WEXITED retcode) -> found or (retcode <> 0)
21        | (pid, _)               -> failwith "fork_search"
22    ;;
```

Après le `fork`, le processus fils parcourt la moitié haute du tableau, et sort avec le code de retour 1 s'il a trouvé un élément satisfaisant le prédicat `cond`, ou 0 sinon (lignes 15 et 16). Le processus père parcourt la moitié basse du tableau, puis appelle `wait` pour se synchroniser avec le processus fils (lignes 18 et 19). Si le fils a terminé normalement, on combine son code de retour avec le booléen résultat de la recherche dans la moitié basse du tableau. Sinon, quelque chose d'horrible s'est produit, et la fonction `fork_search` échoue.

En plus de la synchronisation entre processus, l'appel `wait` assure aussi la récupération de toutes les ressources utilisées par le processus fils. Quand un processus termine, il passe dans un état dit "zombie", où la plupart des ressources qu'il utilise (espace mémoire, etc) ont été désallouées, mais pas toutes : il continue à occuper un emplacement dans la table des processus, afin de pouvoir transmettre son code de retour au père via l'appel `wait`. Ce n'est que lorsque le père a exécuté `wait` que le processus zombie disparaît de la table des processus. Cette table étant de taille fixe, il importe, pour éviter le débordement, de faire `wait` sur les processus qu'on lance.

Si le processus père termine avant le processus fils, le fils se voit attribuer le processus numéro 1 (`init`) comme père. Ce processus contient une boucle infinie de `wait`, et va donc faire disparaître complètement le processus fils dès qu'il termine. Ceci débouche sur une technique utile dans le cas où on ne peut pas facilement appeler `wait` sur chaque processus qu'on a créé (parce qu'on ne peut pas se permettre de bloquer en attendant la terminaison des fils, par exemple) : la technique "du double `fork`".

```
match fork() with
  0 -> if fork() <> 0 then exit 0;
       (* faire ce que le fils doit faire *)
| _ -> wait();
       (* faire ce que le pere doit faire *)
```

Le fils termine par `exit` juste après le deuxième `fork`. Le petit-fils se retrouve donc orphelin, et est adopté par le processus `init`. Il ne laissera donc pas de processus zombie. Le père exécute `wait` aussitôt pour récupérer le fils. Ce `wait` ne bloque pas longtemps puisque le fils termine très vite.

## 3.4 Lancement d'un programme

Les appels `execve`, `execv` et `execvp` lancent l'exécution d'un programme à l'intérieur du processus courant. Sauf en cas d'erreur, ces appels ne retournent jamais : ils arrêtent le déroulement du programme courant et se branchent au début du nouveau programme.

```
execve : string -> string vect -> string vect -> unit
execv  : string -> string vect -> unit
execvp : string -> string vect -> unit
```

Le premier argument est le nom du fichier contenant le code du programme à exécuter. Dans le cas de `execvp`, ce nom est également recherché dans les répertoires du path d'exécution (la valeur de la variable d'environnement `PATH`).

Le deuxième argument est la ligne de commande à transmettre au programme exécuté ; ce vecteur de chaînes va se retrouver dans `sys__command_line` du programme exécuté.

Dans le cas de `execve`, le troisième argument est l'environnement à transmettre au programme exécuté ; `execv` et `execvp` transmettent inchangé l'environnement courant.

Les appels `execve`, `execv` et `execvp` ne retournent jamais de résultat : ou bien tout se passe sans erreurs, et le processus se met à exécuter un autre programme ; ou bien une erreur se produit (fichier non trouvé, etc.), et l'appel déclenche l'exception `Unix_error`.

**Exemple :** les trois formes ci-dessous sont équivalentes :

```
execve "/bin/ls" [|"ls"; "-l"; "/tmp"|] (environment())
execv  "/bin/ls" [|"ls"; "-l"; "/tmp"|]
execvp "ls"      [|"ls"; "-l"; "/tmp"|]
```

**Exemple :** voici un "wrapper" autour de la commande `grep`, qui ajoute l'option `-i` (confondre majuscules et minuscules) à la liste d'arguments :

```
#open "sys";;
#open "unix";;
let main () =
  execvp "grep"
         (concat_vect [|"grep"; "-i"|]
                      (sub_vect command_line 1
                                (vect_length command_line - 1)))
;;
handle_unix_error main ();;
```

**Exemple :** voici un "wrapper" autour de la commande `emacs`, qui change le type du terminal :

```
#open "sys";;
#open "unix";;
let main () =
  execve "/usr/local/bin/emacs"
         command_line
         (concat_vect [|"TERM=hacked-xterm"|] (environment()));;
handle_unix_error main ();;
```

C'est le même processus qui a fait `exec` qui exécute le nouveau programme. En conséquence, le nouveau programme hérite de certains morceaux de l'environnement d'exécution du programme qui a fait `exec` :

- même numéro de processus, même processus père, même comportement vis-à-vis du processus père qui fait `wait`
- même entrée standard, même sortie standard, même sortie d'erreur standard
- même signaux ignorés (cf. la partie sur les signaux)

## 3.5 Exemple complet : un mini-shell

Le programme qui suit est un interprète de commandes simplifié : il lit des lignes sur l'entrée standard, les coupe en mots, lance la commande correspondante, et recommence jusqu'à une fin de fichier sur l'entrée standard. On commence par la fonction qui coupe une chaîne de caractères en une liste de mots. Pas de commentaires pour cette horreur.

```
#open "unix";;

let split_words s =
  let rec skip_blanks i =
    if i < string_length s & nth_char s i = ` `
    then skip_blanks (i+1)
    else i in
  let rec split start i =
    if i >= string_length s then
      [sub_string s start (i-start)]
    else if nth_char s i = ` ` then
      let j = skip_blanks i in
      sub_string s start (i-start) :: split j j
    else
      split start (i+1) in
  vect_of_list (split 0 0);;
```

On passe maintenant à la boucle principale de l'interpréteur.

```
let main () =
  try
    while true do
      let cmd = input_line std_in in
      let words = split_words cmd in
      match fork() with
        0 ->
          begin try
            execvp words.(0) words
          with Unix_error(err, _, _) ->
            print_string
              ("Cannot execute " ^ words.(0) ^ " : " ^ error_message err);
            print_newline();
            exit 255
          end
      | pid_son ->
          match wait() with
            (pid, WEXITED 255) ->
              ()
          | (pid, WEXITED status) ->
              print_string "Program exited with code ";
```

```
       print_int status;
       print_newline()
     | (pid, WSIGNALED(sig, coredumped)) ->
       print_string "Program killed by signal ";
       print_int sig;
       print_newline();
       if coredumped then begin
         print_string "Yo mama! Core dumped";
         print_newline()
       end
     | (pid, WSTOPPED sig) ->
       print_string "Program stopped (???)";
       print_newline()
  done
 with End_of_file ->
  ()
;;
handle_unix_error main ();;
```

À chaque tour de boucle, on lit une ligne sur l'entrée standard, via la fonction `input_line` de la bibliothèque standard `io`. (Cette fonction déclenche l'exception `End_of_file` quand la fin de fichier est atteinte, faisant sortir de la boucle.) On coupe la ligne en mots, puis on fait `fork`. Le processus fils fait `exec` pour lancer la commande, avec récupération des erreurs. Le processus père appelle `wait` pour attendre que la commande termine, puis décode et affiche l'information d'état renvoyée par `wait`. Le code de retour 255 est traité spécialement et indique que le fils n'a pas réussi à exécuter `execvp`. (Ce n'est pas une convention standard ; on espère que peu de commandes renvoient le code de retour 255.)

**Exercice 5** *Ajouter la possibilité d'exécuter des commandes en tâche de fond, si elles sont suivies par* `&`.

# Chapitre 4

# Communications inter-processus classiques

On a vu jusqu'ici comment manipuler des processus et les faire communiquer avec l'extérieur par l'intermédiaire de fichiers. Le reste de ce cours est consacré au problème de faire communiquer entre eux des processus s'exécutant en parallèle, pour leur permettre de coopérer.

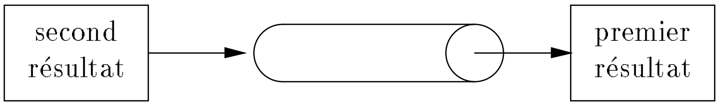
## 4.1 Les tuyaux

Les fichiers normaux ne fournissent pas un moyen satisfaisant de communication entre processus parallèles. Exemple : dans une situation écrivain/lecteur (un processus écrit des informations, l'autre les lit), si on utilise un fichier comme médium, le lecteur peut constater que le fichier ne grossit plus (`read` renvoie zéro), mais il ne peut pas savoir si c'est dû au fait que le processus écrivain a terminé, ou bien si l'écrivain est simplement en train de calculer la prochaine information. De plus, le fichier garde trace de toutes les informations transmises, ce qui pose des problèmes, en particulier de place disque.

Les tuyaux fournissent un mécanisme adapté à ce style de communication. Un tuyau se présente sous la forme de deux descripteurs de fichiers. L'un, ouvert en écriture, représente l'entrée du tuyau. L'autre, ouvert en lecture, représente la sortie du tuyau. On crée un tuyau par l'appel système `pipe` :

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

Le premier résultat est un descripteur ouvert en lecture sur la sortie du tuyau ; le deuxième résultat est un descripteur ouvert en écriture sur l'entrée du tuyau. Le tuyau proprement dit est un objet interne au noyau, accessible uniquement via ces deux descripteurs. En particulier, le tuyau crée n'a pas de nom dans le système de fichiers.



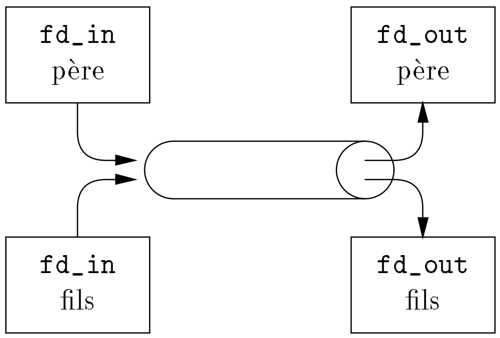
Un tuyau se comporte comme un tampon géré en file d'attente (*first-in, first-out*) : ce qu'on lit sur la sortie du tuyau, c'est ce qu'on a écrit sur l'entrée, dans le même ordre. Les écritures (`write` sur le descripteur d'entrée) remplissent le tuyau, et lorsqu'il est plein, bloquent en attendant qu'un autre processus vide le tuyau en lisant depuis l'autre extrémité ; ce processus est répété plusieurs fois, si nécessaire, jusqu'à ce que toutes les données fournies à `write` ait été transmises. Les lectures (`read` sur le descripteur de sortie) vident le tuyau. Si le tuyau est entièrement vide, la lecture bloque jusqu'à ce qu'un octet au moins soit écrit dedans. Les lectures retournent dès qu'il y a quelque chose dans le tuyau, sans attendre que le nombre d'octets demandés à `read` soit atteint.

Les tuyaux ne présentent donc aucun intérêt si c'est le même processus qui écrit et qui lit dedans. (Ce processus risque fort de se bloquer à tout jamais sur une écriture trop grosse, ou sur une lecture depuis un tuyau vide.) Ce sont donc généralement des processus différents qui écrivent et qui lisent dans un tuyau. Comme le tuyau n'est pas nommé, il faut que ces processus aient été créés par `fork` à partir du processus qui a alloué le tuyau. En effet, les descripteurs sur les deux extrémités du tuyau, comme tous les descripteurs, sont dupliqués au moment du `fork`, et font donc référence au même tuyau dans le processus père et dans le processus fils.

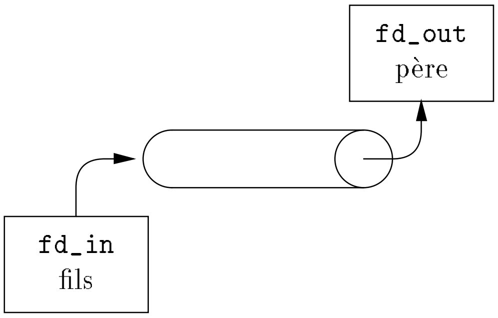
**Exemple :** le fragment de code ci-dessous est typique.

```
let (fd_out, fd_in) = pipe() in
match fork() with
  0 ->
    close fd_out;
    ... write fd_in buffer1 offset1 count1 ...
  _ ->
    close fd_in;
    ... read fd_out buffer2 offset2 count2 ...
```

Après le `fork`, il y a deux descripteurs ouverts sur l'entrée du tuyau (un dans le père, un dans le fils), et de même pour la sortie.




On a choisi de faire du fils l'écrivain, et du père le lecteur. Le fils ferme donc son descripteur sur la sortie du tuyau (pour économiser les descripteurs, et pour éviter certaines erreurs de programmation). Le père ferme de même son descripteur sur l'entrée du tuyau. La situation est donc la suivante :

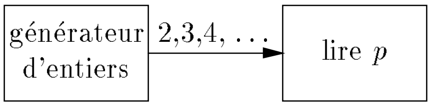



Les données que le fils écrit sur `fd_in` arrivent donc bien jusqu'au descripteur `fd_out` du père.

Lorsqu'on a fermé tous les descripteurs sur l'entrée d'un tuyau, `read` sur la sortie du tuyau renvoie zéro : fin de fichier. Lorsqu'on a fermé tous les descripteurs sur la sortie d'un tuyau, `write` sur l'entrée du tuyau provoque la mort du processus qui fait `write`. (Plus précisément : le noyau envoie un signal `SIGPIPE` au processus qui fait `write`, et le comportement par défaut de ce signal est de tuer le processus. Voir la partie "signaux" ci-dessous.)

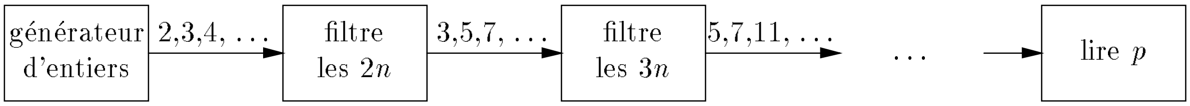
## 4.2 Exemple complet : le crible d'Eratosthène parallèle

L'exemple qui suit est un grand classique de la programmation par processus communicants. Le but du programme est d'énumérer les nombres premiers et de les afficher au fur et à mesure. L'idée est la suivante : initialement, on connecte un processus qui énumère les entiers à partir de 2 sur sa sortie avec un processus "filtre". Le processus filtre commence par lire un entier $p$ sur son entrée, et l'affiche à l'écran.




Le premier processus filtre lit donc $p = 2$. Ensuite, il crée un nouveau processus filtre, connecté à sa sortie, et il se met à filtrer les multiples de $p$ depuis son entrée ; tous les nombres lus qui ne sont pas multiples de $p$ sont réémis sur sa sortie.


Le processus suivant lit donc $p = 3$, l'affiche, et se met à filtrer les multiples de 3. Et ainsi de suite.




Cet algorithme n'est pas directement implémentable en Unix, parce qu'il crée trop de processus (le nombre d'entiers premiers déjà trouvés, plus un). La plupart des systèmes Unix limitent le nombre de processus à quelques dizaines. De plus, dix processus actifs simultanément suffisent à effondrer la plupart des machines monoprocesseurs, en raison des coûts élevés de commutation de contextes pour passer d'un processus à un autre. Dans l'implémentation qui suit, chaque processus attend d'avoir lu un certain nombre d'entiers premiers $p_1, \ldots, p_k$ sur son entrée avant de se transformer en filtre à éliminer les multiples de $p_1, \ldots, p_k$. En pratique, $k = 1000$ ralentit raisonnablement la création de nouveaux processus.

On commence par le processus qui produit les nombres entiers à partir de 2.

```
#open "unix";;

let generate output =
  let rec gen m =
    output_binary_int output m;
    gen (m+1)
  in gen 2;;
```

La fonction `output_binary_int` est une fonction de la bibliothèque standard `io` qui écrit la représentation d'un entier (sous forme de quatre octets) sur un `out_channel`. L'entier peut ensuite être relu par la fonction `input_binary_int`. L'intérêt d'employer ici la bibliothèque `io` est double : premièrement, il n'y a pas à faire soi-même les fonctions de conversion entiers/chaînes de quatre caractères ; deuxièmement, on fait beaucoup moins d'appels système, d'où de meilleures performances. Les deux fonctions ci-dessous construisent un `in_channel` ou un `out_channel` qui lit ou écrit sur le descripteur Unix donné en argument :

```
in_channel_of_descr  : file_descr -> in_channel
out_channel_of_descr : file_descr -> out_channel
```

On passe maintenant au processus filtre. Il utilise la fonction auxiliaire `read_first_primes`. L'appel `read_first_primes input n` lit `n` nombres sur `input` (un `in_channel`), en éliminant les multiples des nombres déjà lus. On affiche ces `n` nombres, et on en renvoie la liste.

```
let read_first_primes input count =
  let rec read_primes first_primes count =
    if count <= 0 then
      first_primes
    else
      let n = input_binary_int input in
      if exists (fun m -> n mod m = 0) first_primes then
        read_primes first_primes count
      else begin
        print_int n; print_newline();
        read_primes (n :: first_primes) (count - 1)
      end in
  read_primes [] count
;;
```

Voici la fonction filtre proprement dite.

```
let rec filter input =
```

```
    let first_primes = read_first_primes input 1000 in
    let (fd_out, fd_in) = pipe() in
    match fork() with
      0 ->
        close fd_in;
        filter (in_channel_of_descr fd_out)
    | _ ->
        close fd_out;
        let output = out_channel_of_descr fd_in in
        while true do
          let n = input_binary_int input in
          if exists (fun m -> n mod m = 0) first_primes then
            ()
          else begin
            print_int n; print_newline();
            output_binary_int output n
          end
        done
;;
```

Le filtre commence par appeler `read_first_primes` pour lire les 1000 premiers nombres premiers sur son entrée (le paramètre `input`, de type `in_channel`). Ensuite, on crée un tuyau et on clone le processus par `fork`. Le processus fils se met à filtrer de même ce qui sort du tuyau. Le processus père lit des nombres sur son entrée, et les envoie dans le tuyau s'ils ne sont pas multiples d'un des 1000 nombres premiers lus initialement.

Le programme principal se réduit à connecter par un tuyau le générateur d'entiers avec un premier processus filtre.

```
let main () =
  let (fd_out, fd_in) = pipe () in
  match fork() with
    0 ->
      close fd_in;
      filter (in_channel_of_descr fd_out)
  | _ ->
      close fd_out;
      generate (out_channel_of_descr fd_in)
;;
printexc__f main ();;
```

## 4.3 Les tuyaux nommés

Certaines variantes d'Unix (System V, SunOS, Ultrix) fournissent la possibilité de créer des tuyaux associés à un nom dans le système de fichiers. Ces tuyaux nommés permettent la communication entre des processus sans liens de famille particuliers (au contraire des tuyaux normaux, qui limitent la communication au créateur du tuyau et à ses descendants).

L'appel permettant de créer un tuyau nommé est :

```
mkfifo : string -> int -> unit
```

Le premier argument est le nom du tuyau ; le deuxième, les droits d'accès voulus.

On ouvre un tuyau nommé par `open`, comme pour un fichier normal. Lectures et écritures sur un tuyau nommé ont la même sémantique que sur un tuyau simple.

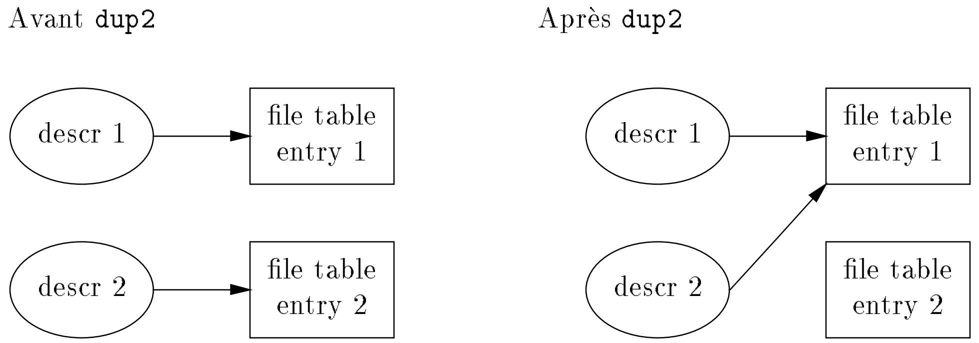
## 4.4 Redirections de descripteurs

Avec ce qu'on a vu jusqu'ici, on ne sait toujours pas connecter par un tuyau des processus via leurs entrées et sorties standard, comme le fait le shell pour exécuter des commandes de la forme `cmd1 | cmd2`. En effet, les descripteurs sur les extrémités d'un tuyau qu'on obtient avec `pipe` (ou avec `open` sur un tuyau nommé) sont de "nouveaux" descripteurs, distincts des descripteurs `stdin`, `stdout` et `stderr`.

Pour ce genre de problèmes, Unix fournit un appel système, `dup2` (lire : "DUPlicate a descriptor TO another descriptor"), qui permet de rediriger un descripteur vers un autre. Ceci est rendu possible par le fait qu'il y a un niveau d'indirection entre un descripteur de fichier (un objet de type `file_descr`) et l'objet du noyau, qu'on appelle une *file table entry*, qui contient effectivement le pointeur vers le fichier ou le tuyau associé, et la position courante de lecture/écriture.

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

L'appel `dup2 fd1 fd2` a pour effet de faire pointer le descripteur `fd2` vers la même *file table entry* que celle pointée par `fd1`. Après exécution de cet appel, `fd2` est donc un duplicata de `fd1` : les deux descripteurs font référence au même fichier ou tuyau, à la même position de lecture/écriture.




**Exemple :** redirection de l'entrée standard.

```
let fd = open "foo" [O_RDONLY] 0 in
dup2 fd stdin;
close fd;
execvp "bar" [|"bar"|]
```

Après le dup2, le descripteur stdin fait référence au fichier foo. Toute lecture sur stdin va donc lire depuis le fichier foo. (Et aussi toute lecture depuis fd ; mais on ne va plus utiliser fd par la suite, et c'est pourquoi on le ferme immédiatement.) De plus, cet état de choses est préservé par execvp. Et donc, le programme bar va s'exécuter avec son entrée standard reliée au fichier foo. C'est ainsi que le shell exécute des commandes de la forme bar < foo.

**Exemple :** redirection de la sortie standard.

```
let fd = open "foo" [O_WRONLY; O_TRUNC; O_CREAT] 0o666 in
dup2 fd stdout;
close fd;
execvp "bar" [|"bar"|]
```

Après le dup2, le descripteur stdout fait référence au fichier foo. Toute écriture sur stdout va donc écrire sur le fichier foo. Le programme bar va donc s'exécuter avec sa sortie standard reliée au fichier foo. C'est ainsi que le shell exécute des commandes de la forme bar > foo.

**Exemple :** relier l'entrée d'un programme à la sortie d'un autre.

```
let (fd_out, fd_in) = pipe() in
match fork() with
  0 -> dup2 fd_out stdin;
       close fd_in;
       close fd_out;
       execvp "cmd2" [|"cmd2"|]
| _ -> dup2 fd_in stdout;
       close fd_in;
       close fd_out;
       execvp "cmd1" [|"cmd1"|]
```

Le programme cmd2 est exécuté avec son entrée standard reliée à la sortie du tuyau. En parallèle, le programme cmd1 est exécuté avec sa sortie standard reliée à l'entrée du tuyau. Et donc, tout ce que cmd1 écrit sur sa sortie standard est lu par cmd2 sur son entrée standard.

Que se passe-t'il si cmd1 termine avant cmd2 ? Au moment du exit dans cmd1, les descripteurs ouverts par cmd1 sont fermés. Il n'y a donc plus de descripteur ouvert sur l'entrée du tuyau. Lorsque cmd2 aura vidé les données en attente dans le tuyau, la prochaine lecture renverra une fin de fichier ; cmd2 fait alors ce qu'elle est censée faire quand elle atteint la fin de son entrée standard — par exemple, terminer. Exemple de cette situation :

```
cat foo bar gee | grep buz
```

D'un autre côté, si cmd2 termine avant cmd1, le dernier descripteur sur la sortie du tuyau est prématurément fermé, et donc cdm1 va recevoir un signal (qui, par défaut, termine le processus) la prochaine fois qu'il essaye d'écrire sur sa sortie standard. Exemple de cette situation :

```
grep buz gee | more
```

et on quitte more avant la fin en appuyant sur q. À ce moment-là, grep est prématurément arrêté, sans qu'il aille jusqu'à la fin du fichier gee.

**Exercice 6** *Comment implémenter quelques-unes des autres redirections du shell* `sh`*, à savoir :*

`>>` `2>` `2>>` `2>1` `<<`

## 4.5 Exemple complet : composer $N$ commandes

On va construire une commande `compose`, telle que

$$\texttt{compose}\ cmd_1\ cmd_2\ \dots\ cmd_n$$

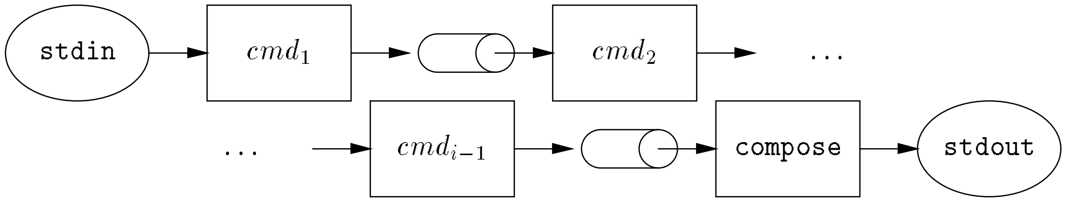
se comporte comme la commande shell

$$cmd_1 \mid cmd_2 \mid \cdots \mid cmd_n.$$

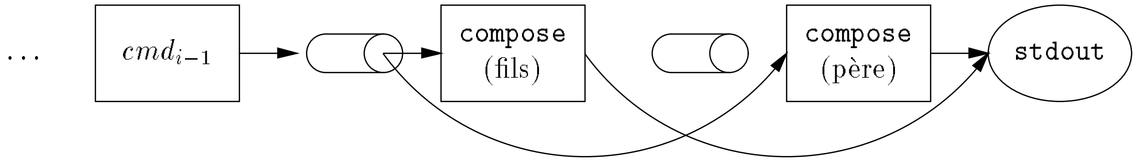
```
#open "sys";;
#open "unix";;

let main () =
  let n = vect_length command_line - 1 in
  for i = 1 to n - 1 do
    let (fd_out, fd_in) = pipe() in
    match fork() with
      0 ->
        dup2 fd_in std_out;
        close fd_in;
        close fd_out;
        execv "/bin/sh" [| "/bin/sh"; "-c"; command_line.(i) |]
    | _ ->
        dup2 fd_out std_in;
        close fd_in;
        close fd_out
  done;
  match fork() with
    0 ->
      execv "/bin/sh" [|"/bin/sh"; "-c"; command_line.(n) |]
  | _ ->
      let rec wait_for_children retcode =
        try
          match wait() with
            (pid, WEXITED n) -> wait_for_children (retcode lor n)
          | (pid, _)         -> wait_for_children 127
        with
            Unix_error(ECHILD, _, _) -> () in
      exit (wait_for_children 0)
;;
handle_unix_error main ();;
```

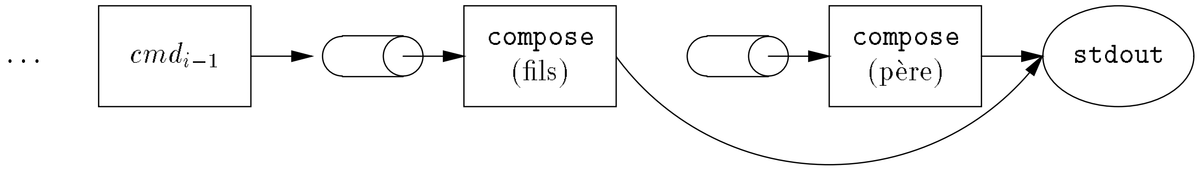
L'essentiel du travail est fait par la boucle `for` des lignes 6–18. Pour chaque commande sauf la dernière, on crée un nouveau tuyau, puis un nouveau processus. Le nouveau processus (le fils) relie l'entrée du tuyau à sa sortie standard, et exécute la commande. Il hérite l'entrée standard du processus père au moment du `fork`. Le processus principal (le père) relie la sortie du tuyau à son entrée standard, et continue la boucle. Supposons (hypothèse de récurrence) que, au début du $i^{\text{ème}}$ tour de boucle, la situation est la suivante :



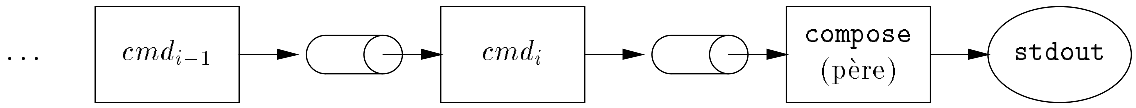

(Les carrés représentent des processus, avec leur entrée standard à gauche et leur sortie standard à droite. Les deux ellipses représentent l'entrée standard et la sortie standard initiales du processus `compose`.) Après `pipe` et `fork`, la situation est la suivante :




Le père appelle `dup2` ; on obtient :




Le fils appelle `dup2` puis `exec` ; on obtient :




Tout est prêt pour le prochain tour de boucle.

L'exécution de la dernière commande est faite en dehors de la boucle, parce qu'il n'y a pas besoin de créer un nouveau tuyau : le processus `compose` a déjà la bonne entrée standard (la sortie de l'avant-dernière commande) et la bonne sortie standard (celle fournie initialement à la commande `compose`) ; il suffit donc de faire `fork` et `exec` du côté du processus fils. Le processus père se met alors à attendre que les fils terminent : on fait `wait` de manière répétée jusqu'à ce que l'erreur `ECHILD` (plus de fils à attendre) se déclenche. On combine entre eux les codes de retour des processus fils par un "ou" bit-à-bit (opérateur `lor`) pour fabriquer un code de retour raisonnable pour `compose` : zéro si tous les fils ont renvoyé zéro, non-nul sinon.

Remarque : si les exécutions des commandes $cmd_i$ passent par l'intermédiaire de `/bin/sh`, c'est pour le cas où l'on fait par exemple

```
compose "grep foo" "wc -l"
```

L'appel à `/bin/sh` assure le découpage de chaque commande complexe en mots. (On aurait pu le faire soi-même comme dans l'exemple du mini-shell, mais c'est pénible.)

## 4.6 Multiplexage d'entrées-sorties

Dans les exemples qu'on a vu jusqu'ici, les processus communiquent de façon “linéaire”. En particulier, un processus lit des données en provenance d'au plus un autre processus. Les situations où un processus doit lire des données en provenance de plusieurs processus posent des problèmes supplémentaires, comme on va le voir maintenant.

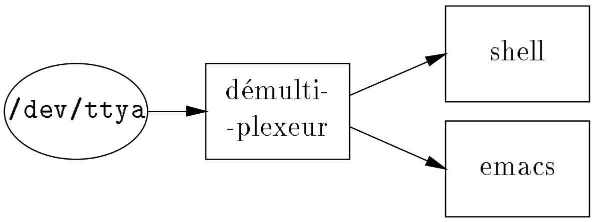
On considère l'exemple d'un émulateur de terminal multi-fenêtres sur un micro-ordinateur. C'est-à-dire, on a un micro-ordinateur relié à une machine Unix par une liaison série, et on cherche à faire émuler par le micro-ordinateur plusieurs terminaux, affichant chacun dans une fenêtre différente sur l'écran du micro, et reliés à des processus différents sur la machine Unix. Par exemple, une fenêtre peut être reliée à un shell, et une autre à un éditeur de textes. Les sorties du shell s'affichent sur la première fenêtre, les sorties de l'éditeur, sur la seconde ; les caractères entrés au clavier du micro sont envoyés sur l'entrée standard du shell si la première fenêtre est active, ou sur l'entrée standard de l'éditeur si la seconde est active.

Comme il n'y a qu'une liaison physique entre le micro et la machine Unix, il va falloir multiplexer les liaisons virtuelles fenêtre/processus, c'est-à-dire entrelacer les transmissions de données. Voici le protocole qu'on va utiliser. Transitent sur la liaison série des paquets de la forme suivante :

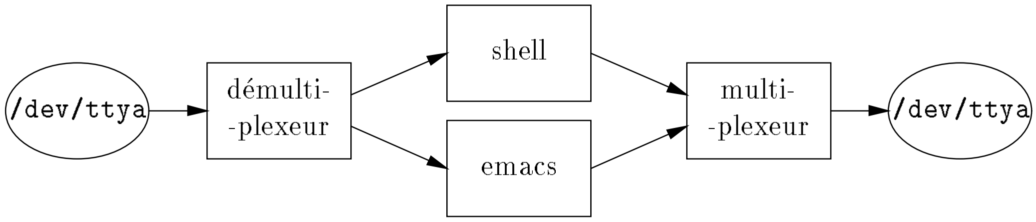
- un octet indiquant le numéro du processus ou de la fenêtre destinataire
- un octet indiquant le nombre $N$ d'octets de données qui suivent
- $N$ octets représentant les données à transmettre au destinataire.

Voici comment les choses se passent du côté de la machine Unix. Les processus utilisateur (shell, éditeur, etc.) vont être reliés par des tuyaux à un ou plusieurs processus auxiliaires, qui lisent et écrivent sur la liaison série, et effectuent le multiplexage et le démultiplexage. La liaison série se présente sous la forme d'un fichier spécial (`/dev/ttya`, par exemple), sur lequel on fait `read` et `write` pour recevoir ou envoyer des octets au micro-ordinateur.

Le démultiplexage (transmission dans le sens micro vers processus) ne pose pas de difficultés. Il suffit d'avoir un processus qui lit des paquets depuis la liaison série, puis écrit les données lues sur le tuyau relié à l'entrée standard du processus utilisateur destinataire.

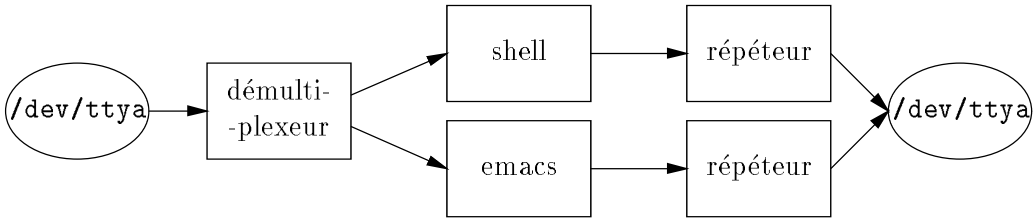


Le multiplexage (transmission dans le sens processus vers micro) est plus délicat. Essayons l'approche symétrique du démultiplexage : un processus lit successivement sur les tuyaux reliés aux sorties standard des processus utilisateur, puis met les données lues sous forme de paquets (en ajoutant le numéro de la fenêtre destinataire et la taille du bloc de données lu) et les écrit sur la liaison série.




Cette approche ne marche pas, car les lectures sur tuyaux sont bloquantes. Par exemple, si on décide de lire sur la sortie du shell, mais que le shell n'affiche rien pendant ce temps, le processus multiplexeur reste bloqué ; et si pendant ce temps l'éditeur affiche des caractères, ces caractères ne seront pas transmis au micro-ordinateur. Il n'y a aucun moyen de savoir à l'avance quels sont les tuyaux sur lesquels il y a effectivement des données en attente d'affichage. (En algorithmique parallèle, cette situation où un processus se voit refuser indéfiniment l'accès à une ressource partagée est connue sous le nom de "situation de famine".)

Essayons une deuxième approche : à chaque processus utilisateur est associé un processus "répéteur", qui lit la sortie standard du processus utilisateur par l'intermédiaire d'un tuyau, la met sous forme de paquet, et écrit directement le paquet sur la liaison série. (Chaque processus répéteur a ouvert `/dev/ttya` en écriture.)




Les situations de blocage ne sont plus à craindre, puisque chaque processus utilisateur a sa sortie retransmise indépendamment des autres processus utilisateur. En revanche, le protocole n'est pas forcément respecté. Deux répéteurs peuvent en effet écrire deux paquets au même instant (ou presque). Or, le noyau Unix ne garantit pas que les écritures sont atomiques, c'est-à-dire effectuées en une seule opération ininterruptible. Le noyau peut très bien transmettre à `/dev/ttya` une partie du paquet écrit par le premier répéteur, puis le paquet écrit par le second répéteur, puis le reste du premier paquet. Ceci va plonger le démultiplexeur qui est à l'autre bout de la liaison dans la plus extrême confusion : il va prendre le deuxième paquet pour une partie des données du premier paquet, puis il va essayer d'interpréter le reste des données du premier paquet comme un en-tête de paquet.

Pour éviter cette situation, il faut que les processus répéteurs se synchronisent pour assurer que, à tout instant, au plus un répéteur est en train d'écrire sur la liaison série. (En algorithmique parallèle, on dit qu'il faut assurer l'exclusion mutuelle entre les processus qui accèdent à la ressource partagée.) On peut le faire avec les moyens qu'on a vu jusqu'ici : les répéteurs créent un fichier donné (le "verrou") avec l'option `O_EXCL` de `open` avant d'écrire un paquet, et le détruisent après avoir écrit le paquet. Cette méthode n'est pas très efficace, en raison du coût de création et de destruction du fichier verrou.

Une troisième approche consiste à reprendre la première approche (un seul processus multiplexeur), en mettant en mode "non bloquant" les descripteurs ouverts sur les tuyaux reliés aux sorties standard des processus utilisateur. (Le passage en mode "non bloquant" se fait par une des options de l'appel système `fcntl`, qu'on ne décrira pas dans ce cours.) En mode "non bloquant", les opérations de lecture sur un tuyau vide ne bloquent pas, mais retournent immédiatement avec une erreur. Il suffit d'ignorer cette erreur et de passer à la lecture sur le prochain processus utilisateur. Cette approche empêche la famine et ne pose pas de problèmes d'exclusion mutuelle, mais se révèle très inefficace. En effet, le processus multiplexeur fait ce qu'on appelle de l'attente active : il consomme du temps de calcul même s'il n'y a rien à faire — par exemple, si les processus utilisateur n'envoient rien sur leur sortie pendant un certain temps. Bien entendu, on peut diminuer la fréquence des tentatives de lecture en introduisant par exemple des appels `sleep` dans la boucle de lecture ; mais il est très difficile de trouver la bonne fréquence : celle qui charge suffisamment peu la machine lorsqu'il n'y a rien à retransmettre, mais qui n'introduit pas de délai perceptible dans la transmission lorsqu'il y a beaucoup à retransmettre.

On le voit, il y a là un problème sérieux. Pour le résoudre, les concepteurs de BSD Unix ont introduit un nouvel appel système, `select`, qui se trouve maintenant sur la plupart des variantes d'Unix. L'appel `select` permet à un processus de se mettre en attente (passive) d'un ou plusieurs événements d'entrée-sortie. Les événements possibles sont :

- événements en lecture : "il y a des données à lire sur tel descripteur"
- événements en écriture : "on peut écrire sur tel descripteur sans être bloqué"
- événements exceptionnels : "une condition exceptionnelle est vraie sur tel descripteur". Par exemple, sur certaines connexions réseau, on peut envoyer des données prioritaires (*out-of-band data*), qui court-circuitent les données normales en attente de transmission. La réception de telles données prioritaires constitue une condition exceptionnelle.

L'appel système `select` a l'interface suivante :

```
select : file_descr list -> file_descr list -> file_descr list -> float ->
            file_descr list * file_descr list * file_descr list
```

Les trois premiers arguments sont des ensembles de descripteurs (représentés par des listes) : le premier argument est l'ensemble des descripteurs à surveiller en lecture ; le deuxième argument est l'ensemble des descripteurs à surveiller en écriture ; le troisième argument est l'ensemble des descripteurs à surveiller en exception. Le quatrième argument est un délai maximal d'attente, exprimé en secondes. S'il est positif ou nul, l'appel `select` va rendre la main au bout de cet temps, même si aucun événement ne s'est produit. S'il est strictement négatif, l'appel `select` attend indéfiniment qu'un des événements demandés se produise.

L'appel `select` renvoie un triplet de listes de descripteurs : les descripteurs effectivement prêts en lecture (première composante), en écriture (deuxième composante), ou sur lesquels une condition exceptionnelle s'est produite (troisième composante). Si le délai maximal s'est écoulé sans aucun événement, les trois listes sont vides.

**Exemple :** le fragment de code ci-dessous surveille en lecture les deux descripteurs `fd1` et `fd2`, et reprendre la main au bout de 0,5 secondes.

```
match select [fd1;fd2] [] [] 0.5 with
  [],[],[] -> (* le delai de 0,5s est ecoule *)
| fdl,[],[] ->
    if mem fd1 fdl then
         (* lire depuis fd1 *);
    if mem fd2 fdl then
         (* lire depuis fd2 *)
```

**Exemple :** la fonction `multiplex` ci-dessous est le coeur du multiplexeur/démultiplexeur pour l'émulateur de terminaux virtuels décrit plus haut. Elle prend un descripteur ouvert sur la liaison série, et deux tableaux de descripteurs, l'un contenant les tuyaux reliés aux entrées standard des processus utilisateur, l'autre les tuyaux reliés aux sorties.

```
#open "unix";;

let rec really_read fd buff start length =
  if length <= 0 then () else
    match read fd buff start length with
      0 -> raise End_of_file
    | n -> really_read fd buff (start+n) (length-n);;

let buffer = create_string 258;;

let multiplex channel inputs outputs =
  let input_fds = channel :: list_of_vect inputs in
  try
    while true do
      let (ready_fds, _, _) = select input_fds [] [] (-1.0) in
      for i = 0 to vect_length inputs - 1 do
        if mem inputs.(i) ready_fds then begin
          let n = read inputs.(i) buffer 2 255 in
          set_nth_char buffer 0 (char_of_int i);
          set_nth_char buffer 1 (char_of_int n);
          write channel buffer 0 (n+2);
          ()
        end
      done;
      if mem channel ready_fds then begin
        really_read channel buffer 0 2;
        let i = int_of_char(nth_char buffer 0)
```

```
    and n = int_of_char(nth_char buffer 1) in
    if n = 0 then
     close outputs.(i)
    else begin
     really_read channel buffer 0 n;
     write outputs.(i) buffer 0 n;
     ()
    end
   end
  done
 with End_of_file ->
  ()
;;
```

La fonction `multiplex` commence par construire un ensemble de descripteurs (`input_fds`) contenant les descripteurs d'entrée (ceux qui sont connectés aux sorties standard des processus utilisateur), plus le descripteur sur la liaison série. À chaque tour de la boucle `while true`, on appelle `select` pour surveiller en lecture une copie de `alive_fds`. On ne surveille aucun descripteur en écriture ni en exception, et on ne borne pas le temps d'attente. Lorsque `select` retourne, on teste s'il y a des données en attente sur un des descripteurs d'entrée, ou sur le canal de communication.

S'il y a des données sur une des entrées, on fait `read` sur cette entrée, on ajoute un en-tête aux données lues, et on écrit le paquet obtenu sur le canal. Si `read` renvoie zéro, ceci indique que le tuyau correspondant a été fermé. Le programme à l'autre bout de la liaison série va recevoir un paquet contenant zéro octets de données, ce qu'il va interpréter comme “le processus utilisateur numéro tant est mort” ; il peut alors fermer la fenêtre correspondante.

S'il y a des données sur le canal, on lit d'abord l'en-tête, ce qui donne le numéro `i` de la sortie destinataire et le nombre `n` d'octets à lire ; puis on lit les `n` octets sur le canal, et on les écrit sur la sortie numéro `i`. Cas particulier : si `n` vaut 0, on ferme la sortie correspondante. L'idée est que l'émulateur de terminal à l'autre bout de la liaison va envoyer un paquet avec `n` = 0 pour signifier une fin de fichier sur l'entrée standard du processus correspondant.

On sort de la boucle quand `really_read` déclenche une exception `End_of_file`, c'est-à-dire quand la fin de fichier est atteinte sur la liaison série.

## 4.7 Les signaux

Les signaux, ou interruptions logicielles, sont des événements externes qui changent le déroulement d'un programme de manière asynchrone, c'est-à-dire à n'importe quel instant lors de l'exécution du programme. En ceci les signaux s'opposent aux autres formes de communications vues jusqu'ici, où les programmes doivent explicitement demander à recevoir les messages externes en attente, par exemple en faisant `read` sur un tuyau.

Lorsqu'un processus reçoit un signal, plusieurs comportements sont possibles.

- Le signal termine l'exécution du processus. Dans certain cas, le système écrit une image de l'état du processus dans le fichier `core`, qu'on peut examiner plus tard avec un *debugger*; c'est ce qu'on appelle un *core dump*.

- Le signal suspend l'exécution du processus, mais le garde en mémoire. Le processus père (typiquement, un shell) est prévenu de ce fait, et peut choisir de terminer le processus, ou de le redémarrer en tâche de fond, en lui envoyant d'autres signaux.

- Rien ne se passe. Le signal est complètement ignoré.

- Le signal provoque l'exécution d'une fonction qui lui a été associée dans le programme. L'exécution normale du programme reprend lorsque la fonction retourne.

Il y a plusieurs types de signaux, indiquant chacun une condition particulière. Le type énuméré `signal` en donne la liste. En voici quelques-uns, avec le comportement par défaut associé :

| Nom | Signification | Comportement |
|---|---|---|
| `SIGHUP` | Hang-up (fin de connexion) | Terminaison |
| `SIGINT` | Interruption (`ctrl-C`) | Terminaison |
| `SIGQUIT` | Interruption forte (`ctrl-\`) | Terminaison + core dump |
| `SIGFPE` | Erreur arithmétique (division par zéro) | Terminaison + core dump |
| `SIGKILL` | Interruption très forte (ne peut être ignorée) | Terminaison |
| `SIGSEGV` | Violation des protections mémoire | Terminaison + core dump |
| `SIGPIPE` | Écriture sur un tuyau sans lecteurs | Terminaison |
| `SIGALRM` | Interruption d'horloge | Ignoré |
| `SIGTSTP` | Arrêt temporaire d'un processus (`ctrl-Z`) | Suspension |
| `SIGCONT` | Redémarrage d'un processus arrêté | Ignoré |
| `SIGCHLD` | Un des processus fils est mort ou a été arrêté | Ignoré |

Les signaux reçus par un programme proviennent de plusieurs sources possibles :

- L'utilisateur, au clavier. Par exemple, le *driver* de terminal envoie le signal `SIGINT` à tous les processus lancés depuis ce terminal quand l'utilisateur tape le caractère d'interruption `ctrl-C`. De même, il envoie `SIGQUIT` quand l'utilisateur tape `ctrl-\`. Et il envoie `SIGHUP` lorsque la connexion avec le terminal est fermée, ou bien parce que l'utilisateur s'est déconnecté, ou bien, dans le cas d'une connexion à travers un modem, parce que la liaison téléphonique a été coupée.

- L'utilisateur, par la commande `kill`. Cette commande permet d'envoyer un signal quelconque à un processus quelconque. Par exemple, `kill -KILL 194` termine à coup sûr le processus numéro 194.

- Le système, pour des processus qui se comportent mal. Par exemple, un processus qui effectue une division par zéro reçoit un signal `SIGFPE`.

- Le système, pour prévenir un processus que son environnement a changé. Par exemple, lorsqu'un processus meurt, son père reçoit le signal `SIGCHLD`.

## 4.8 Produire des signaux

L'appel système `kill` permet d'envoyer un signal à un processus.

```
kill : signal -> int -> unit
```

Le paramètre entier est le numéro du processus auquel le signal est destiné. Une erreur se produit si on envoie un signal à un processus n'appartenant pas au même utilisateur que le processus envoyeur. Un processus peut s'envoyer des signaux à lui-même.

L'appel système `alarm` permet de produire des interruptions d'horloge.

```
alarm: int -> int
```

L'appel `alarm`($n$) retourne immédiatement, mais fait envoyer au processus le signal `SIGALRM` $n$ secondes plus tard.

## 4.9 Changer l'effet d'un signal

La fonction `signal` permet de changer le comportement du processus lorsqu'il reçoit un signal d'un certain type.

```
signal : signal -> signal_handler -> unit
```

Le deuxième argument indique le comportement désiré. Si le deuxième argument est `Signal_ignore`, le signal est ignoré. Si le deuxième argument est `Signal_default`, le comportement par défaut est restauré. Si le deuxième argument est `Signal_handle`($f$), où $f$ est une fonction de type `unit->unit`, la fonction $f$ sera appelée à chaque fois qu'on reçoit le signal.

L'appel `fork` préserve les comportements des signaux : les comportements initiaux pour le fils sont ceux pour le père au moment du `fork`. Les appels `exec` remettent les comportements à `Signal_default`, à une exception près : les signaux ignorés avant le `exec` restent ignorés après.

**Exemple :** On veut parfois se déconnecter en laissant tourner des tâches de fond (gros calculs, programmes "espions", etc). Pour ce faire, il faut éviter que les processus qu'on veut laisser tourner ne terminent lorsqu'ils reçoivent le signal `SIGHUP` envoyé au momemt où l'on se déconnecte. Il existe une commande `nohup` qui fait précisément cela :

`nohup` $cmd\ arg_1\ \ldots\ arg_n$

exécute la commande $cmd\ arg_1\ \ldots\ arg_n$ en la rendant insensible au signal `SIGHUP`. (Certains shells font automatiquement `nohup` sur tous les processus lancés en tâche de fond.) Voici comment l'implémenter en trois lignes :

```
signal SIGHUP Signal_ignore;
execvp command_line.(1)
       (sub_vect command_line 1 (vect_length command_line - 1))
```

L'appel `execvp` préserve le fait que `SIGHUP` est ignoré.

Voici maintenant quelques exemples d'interception de signaux.

**Exemple :** Pour sortir en douceur quand le programme s'est mal comporté. Par exemple, un programme comme `yaf` ou `forum` peut essayer de sauver la liste des messages déjà lus dans le fichier correspondant avant de s'arrêter. Il suffit d'exécuter, au début du programme :

```
signal SIGQUIT (Signal_handle quit);
signal SIGSEGV (Signal_handle quit);
signal SIGFPE  (Signal_handle quit);
```

où la fonction `quit` est de la forme :

```
let quit() =
  (* Essayer de sauver les messages lus dans le fichier *);
  exit 100;;
```

**Exemple :** Pour récupérer les interruptions de l'utilisateur. Certains programmes interactif peuvent par exemple vouloir revenir dans la boucle de commande lorsque l'utilisateur frappe `ctrl-C`. Il suffit de déclencher une exception lorsqu'on reçoit le signal `SIGINT`.

```
exception Break;;
let break () = raise Break;;
...
let main_loop() =
  signal SIGINT (Signal_handle break);
  while true do
    try
      (* lire et exécuter une commande *)
    with Break ->
      (* afficher "Interrompu" *)
  done;;
```

**Exemple :** Pour exécuter des tâches périodiques (animations, etc) entrelacées avec l'exécution du programme principal. Par exemple, voici comment faire "bip" toutes les 30 secondes, quel que soit l'activité du programme (calculs, entrées-sorties).

```
let beep () = output_char std_out `\007`; flush std_out; alarm 30; ();;
...
signal SIGALRM (Signal_handle beep); alarm 30;;
```

## 4.10 Problèmes avec les signaux

L'utilisation des signaux, en particulier comme mécanisme de communication asynchrone inter-processus, se heurte à un certain nombre de limitations :

- Les signaux transportent peu d'information : un type de signal, et rien d'autre.

- Les signaux sont complètement asynchrones, et peuvent donc se produire n'importe quand au cours de l'exécution du programme. En conséquence, une fonction appelée sur un signal ne doit pas en général modifier les structures de données du programme principal, parce que le programme principal peut avoir été interrompu juste au milieu d'une modification de la structure.

- Sur de nombreuses versions d'Unix (en particulier System V), le comportement d'un signal est automatiquement remis à `Signal_default` lorsqu'il est intercepté. La fonction associée peut bien sûr rétablir le bon comportement elle-même ; ainsi, dans l'exemple du "bip" toutes les 30 secondes, il faudrait écrire :

  ```
  let rec beep () =
    set_signal SIGALRM (Signal_handle beep);
    output_char std_out `\007`; flush std_out;
    alarm 30; ();;
  ```

  Le problème est que les signaux qui arrivent entre le moment où le comportement est automatiquement remis à `Signal_default` et le moment où le `set_signal` est exécuté ne sont pas traités correctement : suivant le type du signal, ils peuvent être ignorés, ou causer la mort du processus, au lieu d'exécuter la fonction associée.

  D'autres versions d'Unix (BSD) traitent les signaux de manière plus satisfaisante : le comportement associé à un signal n'est pas changé lorsqu'on le reçoit ; et lorsqu'un signal est en cours de traitement, les autres signaux du même type sont mis en attente.

# Chapitre 5

# Communications modernes : les prises

## 5.1 Les prises

Les prises (traduction libre de *sockets*) sont une généralisation des tuyaux permettant, en particulier, la communication à travers le réseau entre processus tournant sur des machines différentes. Le mécanisme des prises a été introduit en BSD 4.2, et se retrouve maintenant sur presque toutes les machines Unix connectées à un réseau (Ethernet ou autre).

La communication par tuyaux présente certaines insuffisances. Tout d'abord, la communication est locale à une machine : les processus qui communiquent doivent tourner sur la même machine (cas des tuyaux nommés), voire tourner sur la même machine et avoir le créateur du tuyau comme ancêtre commun (cas des tuyaux normaux). D'autre part, les tuyaux ne sont pas bien adaptés à un modèle particulièrement utile de communication, le modèle dit par connexions, ou modèle client-serveur. Dans ce modèle, un seul programme (le serveur) accède directement à une ressource partagée ; les autres programmes (les clients) accèdent à la ressource par l'intermédiaire d'une connexion avec le serveur ; le serveur sérialise et réglemente les accès à la ressource partagée. (Exemple : le système de fenêtrage X-windows — les ressources partagées étant ici l'écran, le clavier, la souris et le haut-parleur.) Le modèle client-serveur est difficile à implémenter avec des tuyaux. La grande difficulté, ici, est l'établissement de la connexion entre un client et le serveur. Avec des tuyaux non nommés, c'est impossible : il faudrait que les clients et le serveur aient un ancêtre commun ayant alloué à l'avance un nombre arbitrairement grand de tuyaux. Avec des tuyaux nommés, on peut imaginer que le serveur lise des demandes de connexions sur un tuyau nommé particulier, ces demandes de connexions pouvant contenir le nom d'un autre tuyau nommé, créé par le client, à utiliser pour communiquer directement avec le client. Le problème est d'assurer l'exclusion mutuelle entre les demandes de connexion provenant de plusieurs clients en même temps.

Les prises sont une abstraction un peu plus générale que les tuyaux qui pallie ces faiblesses. Tout d'abord, des appels systèmes spéciaux sont fournis pour établir des connexions suivant le modèle client-serveur. Ensuite, plusieurs domaines de communication sont pris en compte. Le domaine de communication associé à une prise indique avec qui on peut communiquer via cette prise ; il conditionne le format des adresses employées pour désigner le correspondant. Deux exemples de domaines :

- le domaine Unix : les adresses sont des noms dans la hiérarchie de fichiers d'une machine. La communication est limitée aux processus tournant sur cette machine (comme dans le cas des tuyaux nommés).
- le domaine Internet : les adresses sont constituées de l'adresse d'une machine dans le réseau Internet (adresse de la forme `129.199.129.1`, par exemple), plus un numéro de service à l'intérieur de la machine. La communication est possible entre processus tournant sur deux machines quelconques reliées au réseau Internet.[1]

Enfin, plusieurs sémantiques de communication sont prises en compte. La sémantique indique en particulier si la communication est fiable (pas de pertes ni de duplication de données), et sous quelle forme les données se présentent au récepteur (flot d'octet, ou flot de paquets — petits blocs d'octets délimités). La sémantique conditionne le protocole utilisé pour la transmission des données. Voici trois exemples de sémantiques possibles :

| | Stream | Datagrams | Sequenced packets |
|---|---|---|---|
| Fiable | oui | non | oui |
| Forme des données | flot d'octets | paquets | paquets |

La sémantique "stream" est très proche de celle de la communication par tuyaux. C'est la plus employée, en particulier lorsqu'il s'agit de retransmettre des suites d'octets sans structure particulière (exemple : `rsh`). La sémantique "sequenced packets" structure les données transmises en paquets : chaque écriture délimite un paquet, chaque lecture lit au plus un paquet. Elle est donc bien adaptée à la communication par messages. La sémantique "datagrams" correspond au plus près aux possibilités hardware d'un réseau de type Ethernet : les transmissions se font par paquets, et il n'est pas garanti qu'un paquet arrive à son destinataire. C'est la sémantique la plus économique en termes d'occupation du réseau. Certains programmes l'utilisent pour transmettre des données sans importance cruciale (exemple : `biff`) ; d'autres, pour tirer plus de performances du réseau, étant entendu qu'ils doivent gérer eux-même les pertes.

## 5.2 Création d'une prise

L'appel système `socket` permet de créer une nouvelle prise.

```
socket : socket_domain -> socket_type -> int -> file_descr
```

Le résultat est un descripteur de fichier qui représente la nouvelle prise. Ce descripteur est initialement dans l'état dit "non connecté" ; en particulier, on ne peut pas encore faire `read` ou `write` dessus.

[1] Le réseau Internet se compose de réseaux locaux, généralement du type Ethernet, reliés par des lignes téléphoniques spécialisées. Il relie quelques centaines de milliers de machines dans le monde entier, avec une densité particulièrement forte aux États-Unis. À l'intérieur du domaine Internet, il n'y a pas de différence au niveau des programmes entre communiquer avec la machine voisine, branchée sur le même câble Ethernet, et communiquer avec une machine à l'autre bout du monde, à travers une dizaine de répéteurs et une liaison satellite.

Le premier argument indique le domaine de communication auquel la prise appartient :

| | |
|---|---|
| PF_UNIX | le domaine Unix |
| PF_INET | le domaine Internet |

Le deuxième argument indique la sémantique de communication désirée :

| | |
|---|---|
| SOCK_STREAM | flot d'octets, fiable |
| SOCK_DGRAM | paquets, non fiable |
| SOCK_SEQPACKET | paquets, fiable |
| SOCK_RAW | accès direct au couches basses du réseau |

Le troisième argument est le numéro du protocole de communication à utiliser. Il vaut généralement 0, ce qui désigne le protocole par défaut. D'autres valeurs donnent accès à des protocoles spéciaux. Exemple typique : le protocole ICMP (Internet Control Message Protocol), qui est le protocole utilisé par la commande `ping` pour envoyer des paquets avec retour automatique à l'envoyeur. Les numéros des protocoles spéciaux se trouvent dans le fichier `/etc/protocols` (ou dans la table `protocols` des yellow pages, le cas échéant). La fonction de bibliothèque `getprotobyname` permet de consulter cette table de manière portable :

```
getprotobyname : string -> protocol_entry
```

L'argument est le nom du protocole désiré. Le résultat est un type *record* comprenant, entre autres, un champ `p_proto` qui est le numéro du protocole.

## 5.3 Adresses

Un certain nombre d'opérations sur les prises utilisent des adresses de prises. Ce sont des valeurs du type concret `sockaddr` :

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

L'adresse `ADDR_UNIX`($f$) est une adresse dans le domaine Unix. La chaîne $f$ est le nom du fichier correspondant dans la hiérarchie de fichiers de la machine.

L'adresse `ADDR_INET`($a, p$) est une adresse dans le domaine Internet. Le premier argument, $a$, est l'adresse Internet d'une machine ; le deuxième argument, $p$, est un numéro de service (*port number*) à l'intérieur de cette machine.

Les adresses Internet sont représentées par le type abstrait `inet_addr`. Deux fonctions permettent de convertir des chaînes de caractères de la forme `128.93.8.2` en valeurs du type `inet_addr`, et réciproquement :

```
inet_addr_of_string : string -> inet_addr
string_of_inet_addr : inet_addr -> string
```

Une autre manière d'obtenir des adresses Internet est par consultation de la table `/etc/hosts`, qui associe des adresses Internet aux noms de machines. La fonction de bibliothèque `gethostbyname` permet de consulter cette table (et, sur les machines modernes, interroge les "name servers" en cas d'échec).

```
gethostbyname : string -> host_entry
```

L'argument est le nom de la machine désirée. Le résultat est un type *record* comprenant, entre autres, un champ `h_addr_list`, qui est un vecteur d'adresses Internet : les adresses de la machine. (Une même machine peut être reliée à plusieurs réseaux, sous des adresses différentes.)

Pour ce qui est des numéros de services (*port numbers*), les services les plus courants sont répertoriés dans la table `/etc/services`. On peut la consulter par la fonction

```
getservbyname : string -> string -> service_entry
```

Le premier argument est le nom du service (par exemple, `ftp` pour le serveur FTP, `smtp` pour le courrier, `nntp` pour le serveur de News, `talk` et `ntalk` pour les commandes du même nom, etc.). Le deuxième argument est le nom du protocole : généralement, `tcp` si le service utilise des connexions avec la sémantique "stream", ou `udp` si le service utilise des connexions avec la sémantique "datagrams". Le résultat de `getservbyname` est un type *record* dont le champ `s_port` contient le numéro désiré.

**Exemple :** pour obtenir l'adresse du serveur FTP de `nuri.inria.fr` :

```
ADDR_INET((gethostbyname "nuri.inria.fr").h_addr_list.(0),
          (getservbyname "ftp" "tcp").s_port)
```

## 5.4 Connexion à un serveur

L'appel système `connect` permet d'établir une connexion avec un serveur à une adresse donnée.

```
connect : file_descr -> sockaddr -> unit
```

Le premier argument est un descripteur de prise. Le deuxième argument est l'adresse du serveur auquel on veut se connecter.

Une fois `connect` effectué, on peut envoyer des données au serveur en faisant `write` sur le descripteur de la prise, et lire les données en provenance du serveur en faisant `read` sur le descripteur de la prise. Lectures et écritures sur une prise se comportent comme sur un tuyau : `read` bloque tant qu'aucun octet n'est disponible, et peut renvoyer moins d'octets que demandé ; et si le serveur a fermé la connexion, `read` renvoie zéro et `write` déclenche un signal `SIGPIPE`.

## 5.5 Établissement d'un service

On vient de voir comment un client se connecte à un serveur ; voici maintenant comment les choses se passent du côté du serveur. La première étape est d'associer une adresse à une prise, la rendant ainsi accessible depuis l'extérieur. C'est le rôle de l'appel `bind` :

```
bind : file_descr -> sockaddr -> unit
```

Le premier argument est le descripteur de la prise ; le second, l'adresse à lui attribuer.

Dans un deuxième temps, on déclare que la prise peut accepter les connexions avec l'appel `listen` :

```
listen : file_descr -> int -> unit
```

Le premier argument est le descripteur de la prise. Le second indique combien de demandes de connexion peuvent être mises en attente. Une valeur de 2 ou 3 est habituelle ; le maximum est souvent 5. Lorsque ce nombre est dépassé, les demandes de connexion excédentaires échouent.

Enfin, on reçoit les demandes de connexion par l'appel `accept` :

```
accept : file_descr -> file_descr * sockaddr
```

L'argument est le descripteur de la prise. Le premier résultat est un descripteur sur une prise nouvellement créée, qui est reliée au client : tout ce qu'on écrit sur ce descripteur peut être lu sur la prise que le client a donné en argument à `connect`, et réciproquement. Du côté du serveur, la prise donnée en argument à `accept` reste libre et peut accepter d'autres demandes de connexion. Le second résultat est l'adresse du client qui se connecte. (Il peut servir à vérifier que le client est bien autorisé à se connecter ; c'est ce que fait le serveur X avec `xhost`, par exemple.)

**Exemple :** la plupart des serveurs appellent `fork` immédiatement après le `accept`. Le processus fils traite la connexion. Le processus père recommence aussitôt à faire `accept`. C'est ce qu'on peut appeler le modèle du standard téléphonique. Voici donc à quoi ressemble un serveur classique :

```
let s = socket SOCK_INET SOCK_STREAM 0 in
bind s my_address;
listen s 3;
while true do
  let (d,_) = accept s in
  match fork() with
    0 -> close s;
         (* Communiquer avec le client sur le descripteur d *)
         (* Quand c'est fini: *)
         close d; exit 0
  | _ -> close d
done
```

## 5.6 Déconnexion

Il y a deux manières d'interrompre une connexion. La première est de faire `close` sur la prise. Ceci ferme la connexion en écriture et en lecture, et désalloue la prise. Ce comportement est parfois trop brutal. Par exemple, un client peut vouloir fermer la connexion dans le sens client vers serveur, pour transmettre une fin de fichier au serveur, mais laisser la connexion ouverte dans le sens serveur vers client, pour que le serveur puisse finir d'envoyer les données en attente. L'appel système `shutdown` permet ce genre de coupure progressive de connexions.

```
shutdown : file_descr -> shutdown_command -> unit
```

Le premier argument est le descripteur de la prise à fermer. Le deuxième argument peut être :

| | |
|---|---|
| `SHUTDOWN_SEND` | ferme la prise en écriture ; `read` sur l'autre bout de la connexion va renvoyer une marque de fin de fichier |
| `SHUTDOWN_RECEIVE` | ferme la prise en lecture ; `write` sur l'autre bout de la connexion va déclencher un signal `SIGPIPE` |
| `SHUTDOWN_ALL` | ferme la prise en lecture et en écriture ; à la différence de `close`, la prise elle-même n'est pas désallouée |

## 5.7 Exemple complet : le client-serveur universel

On va définir deux commandes, `client` et `server`, qui ont le comportement suivant :

- `client` *host port* établit une connexion avec le service de numéro *port* sur la machine de nom *host*, puis envoie sur la connexion tout ce qu'il lit sur son entrée standard, et écrit sur sa sortie standard tout ce qu'il reçoit sur la connexion

- `server` *port cmd* $arg_1$ ... $arg_n$ reçoit les demandes de connexion au numéro *port*, et à chaque connexion lance la commande *cmd* avec $arg_1$ ... $arg_n$ comme arguments, et la connexion comme entrée et sortie standard.

Par exemple, si on lance

```
server 8500 grep foo
```

sur la machine `pomerol`, on peut ensuite faire depuis n'importe quelle machine

```
client pomerol 8500 < /etc/passwd
```

et il s'affiche la même chose que si on avait fait

```
grep foo < /etc/passwd
```

sauf que `grep` est exécuté sur `pomerol`, et non pas sur la machine locale.

Ces deux commandes `client` et `server` constituent une application client-serveur "universel", dans la mesure où ils regroupent le code d'établissement de service et de connexions qui est commun à beaucoup de serveurs, tout en déléguant la partie implémentation du service et du protocole de communication, propre à chaque application, à d'autres programmes : le programme lancé par `server`, pour la partie serveur ; le programme qui appelle `client`, pour la partie client.

On commence par la commande `client`. La première fonction, `retransmit`, se contente de recopier les données arrivant d'un descripteur sur un autre descripteur. Elle termine lorsque la fin de fichier est atteinte sur le descripteur d'entrée.

```
#open "sys";;
#open "unix";;

```

```
exception Finished;;

let retransmit input output =
  let buffer = create_string 4096 in
  try
    while true do
      match read input buffer 0 4096 with
        0 -> raise Finished
      | n -> write output buffer 0 n
    done
  with Finished ->
    ()
;;
```

Les choses sérieuses commencent ici.

```
let main () =
  let server_name = command_line.(1)
  and port_number = int_of_string command_line.(2) in
  let server_addr =
    try
      inet_addr_of_string server_name
    with Failure _ ->
      try
        (gethostbyname server_name).h_addr_list.(0)
      with Not_found ->
        prerr_endline ("Unknown host " ^ server_name);
        exit 2 in
  let sock =
    socket PF_INET SOCK_STREAM 0 in
  connect sock (ADDR_INET(server_addr, port_number));
  match fork() with
    0 -> retransmit stdin sock;
         shutdown sock SHUTDOWN_SEND;
         exit 0
  | _ -> retransmit sock stdout;
         close stdout;
         wait()
;;
handle_unix_error main ();;
```

On commence par déterminer l'adresse Internet du serveur auquel se connecter. Elle peut être donnée (dans le premier argument de la commande) ou bien sous forme numérique, ou bien sous forme d'un nom de machine. On essaye donc d'abord de convertir `command_line.(1)` en adresse Internet par `inet_addr_of_string` ; si ça échoue, on interroge la base `/etc/hosts`, et on prend la première des adresses obtenues.

Ensuite, on crée une prise dans le domaine Internet, avec la sémantique “stream” et le protocole par défaut, et on la connecte à l'adresse indiquée.

On clone alors le processus par `fork`. Le processus fils recopie les données de l'entrée standard vers la prise ; lorsque la fin de fichier est atteinte sur l'entrée standard, il ferme la connexion en écriture, transmettant ainsi une fin de fichier au serveur, et termine. Le processus père recopie sur la sortie standard les données lues sur la prise. Lorsqu'une fin de fichier est détectée sur la prise, il se synchronise avec le processus fils par `wait`, et termine.

On passe maintenant à la commande `server`.

```
#open "sys";;
#open "unix";;

let main () =
  let port_number = int_of_string command_line.(1)
  and args = sub_vect command_line 2 (vect_length command_line - 2) in
  let sock =
    socket PF_INET SOCK_STREAM 0 in
  let my_host_addr =
    (gethostbyname(gethostname())).h_addr_list.(0) in
  bind sock (ADDR_INET(my_host_addr, port_number));
  listen sock 3;
  while true do
    let (s, calleraddr) = accept sock in
      begin match calleraddr with
        ADDR_INET(host,_) ->
          print_string ("Connection from " ^ string_of_inet_addr host);
          print_newline()
      | ADDR_UNIX _ ->
          print_string "Connection from the Unix domain (???)";
          print_newline()
      end;
      match fork() with
        0 -> if fork() <> 0 then exit 0;
             dup2 s stdin;
             dup2 s stdout;
             dup2 s stderr;
             close s;
             execvp args.(0) args
      | _ -> wait();
             close s
  done
;;
handle_unix_error main ();;
```

On commence par créer une prise dans le domaine Internet, avec la sémantique "stream" et le protocole par défaut. On la prépare à recevoir des demandes de connexion sur le *port* indiqué sur la ligne de commande par les appels `bind` et `listen` des lignes 11 et 12. L'adresse fournie à `bind` contient l'adresse Internet de la machine qui fait tourner le programme ; la manière habituelle de l'obtenir (lignes 9 et 10) est de chercher le nom de la machine (renvoyé par l'appel `gethostname`) dans la table `/etc/hosts`.

Le serveur entre ensuite dans une boucle infinie, où il attend une demande de connexion (`accept`, ligne 14), affiche l'adresse du client (lignes 15–22), et crée un processus par `fork` pour traiter la connexion. Le processus fils fait de la prise connectée au client son entrée standard et ses sorties standard, puis exécute la commande fournie en argument à `server`. (On a employé la technique du "double `fork`" de la partie 3.3 pour éviter de laisser traîner des processus zombies.)

Remarque : la fermeture de la connexion se fait sans intervention du programme `serveur`. Premier cas : le client reçoit une fin de fichier sur son entrée standard. Il ferme alors la connexion dans le sens client vers serveur. La commande lancée par le serveur reçoit une fin de fichier sur son entrée standard. Elle finit ce qu'elle a à faire, puis appelle `exit`. Ceci ferme ses sorties standard, qui sont les derniers descripteurs ouverts en écriture sur la connexion. Le client reçoit donc une fin de fichier sur la connexion, et termine à son tour. Le cas où le client termine prématurément (signal) est similaire, sauf que la connexion est complètement fermée du côté du client, ce qui peut provoquer la mort anticipée par signal `SIGPIPE` de la commande du côté serveur ; ça convient parfaitement, vu que plus personne n'est là pour lire les sorties de cette commande. Enfin, la commande côté serveur peut terminer (de son plein gré ou par signal) avant d'avoir reçu une fin de fichier. Les deux répéteurs côté client reçoivent l'un une fin de fichier, l'autre un signal `SIGPIPE`, ce qui provoque la terminaison du processus client, puis des répéteurs.

## 5.8 Exemples de protocoles

Dans les cas simples (`rsh`, `rlogin`, ...), les données à transmettre entre le client et le serveur se présentent naturellement comme deux flux d'octets, l'un du client vers le serveur, l'autre du serveur vers le client. Dans ces cas-là, le protocole de communication est évident. Dans d'autres cas, les données à transmettre sont plus complexes, et nécessitent un codage avant de pouvoir être transmises sous forme de suite d'octets sur une prise. Il faut alors que le client et le serveur se mettent d'accord sur un protocole de transmission précis, qui spécifie le format des requêtes et des réponses échangées sur la prise. La plupart des protocoles employés par les commandes Unix sont décrits dans des documents appelés "RFC" (request for comments) : au début simple propositions ouvertes à la discussion, ces documents acquièrent valeur de norme au cours du temps, au fur et à mesure que les utilisateurs adoptent le protocole décrit.[2]

### 5.8.1 Protocoles "binaires"

La première famille de protocoles vise à transmettre les données sous une forme compacte la plus proche possible de leur représentation en mémoire, afin de minimiser le travail de conversion nécessaire, et de tirer parti au mieux de la bande passante du réseau. Exemples typiques de protocoles

[2] Les RFC sont disponibles par FTP anonyme sur de nombreux sites. En France : `nuri.inria.fr`, dans le répertoire `rfc`.

de ce type : le protocole X-windows, qui régit les échanges entre le serveur X et les applications X, et le protocole NFS (RFC 1094).

Les nombre entiers ou flottants sont généralement transmis comme les 1, 2, 4 ou 8 octets qui constituent leur représentation binaire. Pour les chaînes de caractères, on envoie d'abord la longueur de la chaîne, sous forme d'un entier, puis les octets contenus dans la chaîne. Pour des objets structurés (n-uplets, records), on envoie leurs champs dans l'ordre, concaténant simplement leurs représentations. Pour des objets structurés de taille variable (tableaux), on envoie d'abord le nombre d'éléments qui suivent. Le récepteur peut alors facilement reconstituer en mémoire la structure transmise, à condition de connaître exactement son type. Lorsque plusieurs types de données sont susceptibles d'être échangés sur une prise, on convient souvent d'envoyer en premier lieu un entier indiquant le type des données qui va suivre.

**Exemple :** L'appel `XFillPolygon` de la bibliothèque X, qui dessine et colorie un polygone, provoque l'envoi au serveur X d'un message de la forme suivante :

- l'octet 69 (le code de la commande `FillPoly`)
- un octet quelconque de remplissage
- un entier sur deux octets indiquant le nombre de sommets $n$ du polygone
- un entier sur quatre octets identifiant la fenêtre où tracer
- un entier sur quatre octets identifiant le “graphic context”
- un octet de “forme”, indiquant si le polygone est convexe, etc.
- un octet indiquant si les coordonnées des sommets sont absolus ou relatifs
- $4n$ octets codant les coordonnées des sommets du polygone, en deux entiers de 16 bits pour chaque sommet.

Dans ce type de protocole, il faut prendre garde aux différences d'architecture entre les machines qui communiquent. En particulier, dans le cas d'entiers sur plusieurs octets, certaines machines mettent l'octet de poids fort en premier (c'est-à-dire, en mémoire, à l'adresse basse) (architectures dites *big-endian*), et d'autres mettent l'octet de poids faible en premier (architectures dites *little-endian*). Par exemple, l'entier 16 bits $12345 = 48 \times 256 + 57$ est représenté par l'octet 48 à l'adresse $n$ et l'octet 57 à l'adresse $n + 1$ sur une architecture big-endian, et par l'octet 57 à l'adresse $n$ et l'octet 48 à l'adresse $n + 1$ sur une architecture little-endian. Le protocole doit donc spécifier que tous les entiers multi-octets sont transmis en mode big-endian, par exemple. Une autre possibilité est de laisser l'émetteur choisir librement entre big-endian et little-endian, mais de signaler dans l'en-tête du message quelle convention il utilise par la suite.

Le système Caml Light facilite grandement ce travail de mise en forme des données (travail souvent appelé *marshaling* dans la littérature) en fournissant deux prmitives générales de transformation d'une valeur Caml Light quelconque en suite d'octets, et réciproquement :

```
output_value : out_channel -> 'a -> unit
input_value : in_channel -> 'a
```

Le but premier de ces deux primitives est de pouvoir sauvegarder n'importe quel objet structuré dans un fichier disque, puis de le recharger ensuite ; mais elles s'appliquent également bien à la transmission de n'importe quel objet le long d'un tuyau ou d'une prise. Ces primitives savent faire face à tous les types de données Caml Light à l'exception des fonctions ; elles préservent le partage et les circularités à l'intérieur des objets transmis ; et elles savent communiquer entre des architectures d'endianness différente.

**Exemple :** Si X-windows était écrit en Caml Light, on aurait un type concret `request` des requêtes pouvant être envoyées au serveur, et un type concret `reply` des réponses éventuelles du serveur :

```
type request =
    ...
  | FillPolyReq of (int * int) vect * drawable * graphic_context
                                   * poly_shape * coord_mode
  | GetAtomNameReq of atom
  | ...
and reply =
    ...
  | GetAtomNameReply of string
  | ...
```

et de même un type concret Le coeur du serveur serait une boucle de lecture et décodage des requêtes de la forme suivante :

```
(* Recueillir une demande de connexion sur le descripteur s *)
let requests = in_channel_of_descr s
and replies  = out_channel_of_descr s in
try
  while true do
    match input_value requests with
        ...
      | FillPoly(vertices, drawable, gc, shape, mode) ->
          fill_poly vertices drawable gc shape mode
      | GetAtomNameReq atom ->
          output_value replies (GetAtomNameReply(get_atom_name atom))
      | ...
  done
with End_of_file ->
  (* fin de la connexion *)
```

La bibliothèque X, liée avec chaque application, serait de la forme :

```
(* Établir une connexion avec le serveur sur le descripteur s *)
let requests = out_channel_of_descr s
and replies  = in_channel_of_descr s;;
let fill_poly vertices drawable gc shape mode =
  output_value requests
```

```
               (FillPolyReq(vertices, drawable, gc, shape, mode));;
let get_atom_name atom =
  output_value request (GetAtomNameReq atom);
  match input_value replies with
    GetAtomNameReply name -> name
  | _ -> fatal_protocol_error "get_atom_name";;
```

Il faut remarquer que le type de `input_value` donné ci-dessus est sémantiquement incorrect, car beaucoup trop général : il n'est pas vrai que le résultat de `input_value` a le type `'a` pour tout type `'a`. La valeur renvoyée par `input_value` appartient à un type bien précis, et non pas à tous les types possibles ; mais le type de cette valeur ne peut pas être déterminé au moment de la compilation, puisqu'il dépend du contenu du fichier qui va être lu à l'exécution. Le typage correct de `input_value` nécessite une extension du langage ML connue sous le nom d'objets dynamiques : ce sont des valeurs appariées avec une représentation de leur type, permettant ainsi des vérifications de types à l'exécution. On se reportera à [10] pour une présentation plus détaillée.

### 5.8.2 Protocoles "texte"

Les services réseaux où l'efficacité du protocole n'est pas cruciale utilisent souvent un autre type de protocoles : les protocoles "texte", qui sont en fait un petit langage de commandes. Les requêtes sont exprimées sous forme de lignes de commandes, avec le premier mot identifiant le type de requête, et les mots suivants les arguments éventuels. Les réponses sont elles aussi sous forme d'une ou plusieurs lignes de texte, commençant souvent par un code numérique, pour faciliter le décodage de la réponse. Quelques protocoles de ce type :

| | | |
|---|---|---|
| SMTP (simple mail transfert protocol) | RFC 821 | courrier électronique |
| FTP (file transfert protocol) | RFC 959 | transferts de fichiers |
| NNTP (network news transfert protocol) | RFC 977 | lecture des News |

Le grand avantage de ce type de protocoles est que les échanges entre le client et le serveur sont immédiatement lisibles par un être humain. En particulier, on peut utiliser `telnet` pour dialoguer "en direct" avec un serveur de ce type[3] : on tape les requêtes comme le ferait un client, et on voit s'afficher les réponses. Ceci facilite grandement la mise au point. Bien sûr, le travail de codage et de décodage des requêtes et des réponses est plus important que dans le cas des protocoles binaires ; la taille des messages est également un peu plus grande ; d'où une moins bonne efficacité.

**Exemple :** Voici un exemple de dialogue interactif avec un serveur SMTP. Les lignes précédées par `>` vont dans le sens client → serveur, et sont donc tapées par l'utilisateur. Les lignes précédées par `<` vont dans le sens serveur → client.

```
    pom: telnet margaux smtp
    Trying 128.93.8.2 ...
    Connected to margaux.inria.fr.
    Escape character is '^]'.
<   220 margaux.inria.fr Sendmail 5.64+/AFUU-3 ready at Wed, 15 Apr 92 17:40:59
```

[3] Il suffit de lancer `telnet` *machine service*, où *machine* est le nom de la machine sur laquelle tourne le serveur, et *service* est le nom du service (`smtp`, `nntp`, etc.).

```
>   HELO pomerol.inria.fr
<   250 Hello pomerol.inria.fr, pleased to meet you
>   MAIL From:<god@heavens.sky.com>
<   250 <god@heavens.sky.com>... Sender ok
>   RCPT To:<xleroy@margaux.inria.fr>
<   250 <xleroy@margaux.inria.fr>... Recipient ok
>   DATA
<   354 Enter mail, end with "." on a line by itself
>   From: god@heavens.sky.com (Himself)
>   To: xleroy@margaux.inria.fr
>   Subject: salut!
>
>   Ca se passe bien, en bas?
>   .
<   250 Ok
>   QUIT
<   221 margaux.inria.fr closing connection
    Connection closed by foreign host.
```

Les commandes HELO, MAIL et RCPT transmettent le nom de la machine expéditrice, l'adresse de l'expéditeur, et l'adresse du destinataire. La commande DATA permet d'envoyer le texte du message proprement dit. Elle est suivie par un certain nombre de lignes (le texte du message), terminées par une ligne contenant le seul caractère "point". Pour éviter l'ambiguïté, toutes les lignes du message qui commencent par un point sont transmises en doublant le point initial ; le point supplémentaire est supprimé par le serveur.

Les réponses sont toutes de la forme un code numérique en trois chiffres plus un commentaire. Quand le client est un programme, il interprète uniquement le code numérique ; le commentaire est à l'usage de la personne qui met au point le système de courrier. Les réponses en 5xx indiquent une erreur ; celles en 2xx, que tout s'est bien passé.

## 5.9 Exemple complet : un vérificateur d'adresses de courrier

Le protocole SMTP contient des requêtes (VRFY et EXPN) pour vérifier si une adresse de courrier est reconnue par le serveur, obtenir la liste de tous les destinataires d'une liste de distribution, et même compléter les adresses incomplètes. On va définir une commande verify-addr, prenant en argument le nom d'une machine et une adresse de courrier, qui se connecte au serveur SMTP de la machine, envoie la requête VRFY, et imprime les résultats.

```
#open "unix";;

let rec read_response inchan =
  let resp = input_line inchan in
  let i = ref 0 in
  while !i < string_length resp
      & int_of_char(nth_char resp !i) >= int_of_char `0`
```

```
      & int_of_char(nth_char resp !i) <= int_of_char `9`
  do incr i done;
  let code = int_of_string (sub_string resp 0 !i) in
  let sep = nth_char resp !i in
  let text = sub_string resp (!i + 1) (string_length resp - !i - 2) in
  match sep with
    `-` ->
      let (code', text') = read_response inchan in (code, text ^ "\n" ^ text')
  | _  ->
      (code, text)
;;
```

La fonction `read_response` ci-dessus lit et décode une réponse du serveur. Elle sépare et renvoie la partie code de retour et la partie commentaire. Le serveur peut aussi envoyer des réponses multi-lignes, signalées par un signe `-` séparant le code de retour du commentaire, au lieu de l'espace pour les réponses mono-lignes. Si un `-` est détecté après le code de retour, la fonction se rappelle récursivement pour lire les lignes suivantes du message, et concatène les commentaires obtenus.

Le programme principal est simple et devrait se passer de commentaires.

```
let main() =
  if vect_length sys__command_line != 3 then begin
    prerr_endline "Usage: verify-addr <hostname> <address>";
    exit 2
  end;
  let server_name = sys__command_line.(1)
  and mail_address = sys__command_line.(2) in
  let server_addr =
    try
      inet_addr_of_string server_name
    with Failure _ ->
      try
        (gethostbyname server_name).h_addr_list.(0)
      with Not_found ->
        prerr_endline ("Unknown host " ^ server_name);
        exit 2 in
  let smtp_port =
    try
      (getservbyname "smtp" "tcp").s_port
    with Not_found ->
      prerr_endline "Unknown service 'smtp'";
      exit 2 in
  let sock =
    socket PF_INET SOCK_STREAM 0 in
  connect sock (ADDR_INET(server_addr, smtp_port));
  let requests = out_channel_of_descr sock
```

```
  and replies = in_channel_of_descr sock in
  let (code, msg) = read_response replies in
  if code != 220 then begin
    prerr_endline msg;
    exit 2
  end;
  output_string requests ("VRFY " ^ mail_address ^ "\r\n");
  flush requests;
  let (code, msg) = read_response replies in
  if code >= 500 then begin
    print_string "Error: "; print_string msg; print_newline()
  end else begin
    print_string msg; print_newline()
  end;
  output_string requests "QUIT\r\n";
  flush requests;
  let (code, msg) = read_response replies in
  close sock
;;
handle_unix_error main ();;
```

Exemples d'utilisation :

```
verify-addr margaux.inria.fr leroy
        Error: leroy... User unknown
verify-addr margaux.inria.fr xleroy
        Xavier Leroy <xleroy@pomerol.inria.fr>
verify-addr dmi.ens.fr etudiants
        Vincent Maillot <vmaillot>
        Albert Shih <shih@clipper>
        [...]
        Laurent Alonso <alonso@loria.loria.fr>
```

# Bibliographie


**Caml Light**

[1] Xavier Leroy, Michel Mauny. *The Caml Light system, release 0.5. Documentation and user's manual.* Logiciel L-5, distribué par l'INRIA.

**Programmation du système Unix**

[2] Le manuel Unix, sections 2 et 3.

[3] Brian Kernighan, Rob Pike. *The Unix programming environment*, Addison-Wesley. Traduction française : *L'environnement de la programmation Unix*, Interéditions.

[4] Jean-Marie Rifflet. *La programmation sous Unix.* McGraw-Hill.

[5] Jean-Marie Rifflet. *La communication sous Unix.* McGraw-Hill.

**Architecture du noyau Unix**

[6] Samuel Leffler, Marshall McKusick, Michael Karels, John Quarterman. *The design and implementation of the 4.3 BSD Unix operating system*, Addison-Wesley.

[7] Maurice Bach. *The design and implementation of the Unix operating system*, Prentice-Hall.

**Culture générale sur les systèmes**

[8] Andrew Tanenbaum. *Operating systems, design and implementation*, Prentice-Hall. Traduction française : *Les systèmes d'exploitation : conception et mise en œuvre*, Interéditions.

[9] Andrew Tanenbaum. *Computer Networks*, Prentice-Hall. Traduction française : *Réseaux : architectures, protocoles, applications*, Interéditions.

**Typage des communications d'objets structurés**

[10] Xavier Leroy, Michel Mauny. *Dynamics in ML.* Actes de FPCA 91, LNCS 523, Springer-Verlag.

# Annexe A

# Corrigés des exercices

### Exercice 1

Si l'option -a est fournie, il faut faire

```
open output_name [O_WRONLY; O_CREAT; O_APPEND] 0o666
```

à la place de

```
open output_name [O_WRONLY; O_CREAT; O_TRUNC] 0o666
```

Le parsing de l'option est laissé au lecteur.

### Exercice 2

L'idée est de recopier la chaîne à sortir dans le tampon. Il faut tenir compte du cas où il ne reste pas assez de place dans le tampon (auquel cas il faut vider le tampon), et aussi du cas où la chaîne est plus longue que le tampon (auquel cas il faut l'écrire directement). Voici une possibilité.

```
let output_string chan s =
  let avail = string_length chan.out_buffer - chan.out_pos in
  if string_length s <= avail then begin
    blit_string s 0 chan.out_buffer chan.out_pos (string_length s);
    chan.out_pos <- chan.out_pos + string_length s
  end
  else if chan.out_pos = 0 then begin
    write chan.out_fd s 0 (string_length s)
  end
  else begin
    blit_string s 0 chan.out_buffer chan.out_pos avail;
    write chan.out_fd chan.out_buffer 0 (string_length chan.out_buffer);
    let remaining = string_length s - avail in
    if remaining < string_length chan.out_buffer then begin
```

```
    blit_string s avail chan.out_buffer 0 remaining;
    chan.out_pos <- remaining
  end else begin
    write chan.out_fd s avail remaining;
    chan.out_pos <- 0
  end
end;;
```

### Exercice 3

L'implémentation naïve de `tail` est de lire le fichier séquentiellement, depuis le début, en gardant dans un tampon circulaire les $N$ dernières lignes lues. Quand on atteint la fin du fichier, on affiche le tampon. Il n'y a rien de mieux à faire quand les données proviennent d'un tuyau ou d'un fichier spécial qui n'implémente pas `lseek`. Si les données proviennent d'un fichier normal, il vaut mieux lire le fichier en partant de la fin : avec `lseek`, on lit les 4096 derniers caractères ; on les balaye pour compter les retours à la ligne ; s'il y en a au moins $N$, on affiche les $N$ lignes correspondantes et on sort ; sinon, on recommence en ajoutant les 4096 caractères précédents, etc.

Pour ajouter l'option `-f`, il suffit, une fois qu'on a affiché les $N$ dernières lignes, de se positionner à la fin du fichier, et d'essayer de lire (par `read`) à partir de là. Si `read` réussit à lire quelque chose, on l'affiche aussitôt et on recommence. Si `read` renvoie 0, on attend un peu (`sleep 1`), et on recommence.

### Exercice 4

Il faut garder une table des fichiers source déjà copiés, qui au couple (`st_dev`, `st_ino`) d'un fichier source fait correspondre le nom de son fichier destination. À chaque copie, on consulte cette table pour voir si un fichier source avec le même couple (`st_dev`, `st_ino`) a déjà été copié ; si oui, on fait un lien dur vers le fichier destination, au lieu de refaire la copie. Pour diminuer la taille de cette table, on peut n'y mettre que des fichiers qui ont plusieurs noms, c'est-à-dire tels que `st_nlink` $> 1$.

```
let copied_files = (hashtbl__new 53 : ((int * int), string) hashtbl__t);;

let rec copy source dest =
  let infos = lstat source in
  match infos.st_kind with
    S_REG ->
      if infos.st_nlink > 1 then begin
        try
          let dest' = hashtbl__find copied_files (infos.st_dev, infos.st_ino)
          in link dest' dest
        with Not_found ->
          hashtbl__add copied_files (infos.st_dev, infos.st_ino) dest;
          file_copy source dest;
          set_infos dest infos
      end else begin
```

```
      file_copy source dest;
      set_infos dest infos
    end
| S_LNK -> ...
```

### Exercice 5

Dans le cas où la ligne de commande se termine par `&`, il suffit que le processus père n'appelle pas `wait`, et passe immédiatement au prochain tour de la boucle. Seule difficulté : le père peut maintenant avoir plusieurs fils qui s'exécutent en même temps (les commandes en arrière-plan qui n'ont pas encore terminé, plus la dernière commande synchrone), et `wait` peut se synchroniser sur l'un quelconque de ces fils. Dans le cas d'une commande synchrone, il faut donc répéter `wait` jusqu'à ce que le fils récupéré soit bien celui qui exécute la commande.

```
while true do
  let cmd = input_line std_in in
  let (words, ampersand) = parse_command_line cmd in
  match fork() with
    0 -> (* ... *)
  | pid_son ->
      if ampersand then () else
        let rec wait_for_son() =
          let (pid, status) = wait() in
            if pid = pid_son then status else wait_for_son() in
        match wait_for_son() with
            WEXITED 255 -> (* ... *)
          | WEXITED status -> (* ... *)
          | (* ... *)
done
```

### Exercice 6

Pour `>>`, on procède comme pour `>`, sauf que le fichier est ouvert avec les options

```
[O_WRONLY; O_APPEND; O_CREAT]
```

Pour `2>`, on procède comme pour `>`, sauf que

```
dup2 fd stderr
```

est exécuté en lieu et place de

```
dup2 fd stdout
```

Pour `2>1`, il suffit d'appeler

```
dup2 stderr stdout
```

avant d'exécuter la commande. Enfin, pour `<<`, le shell `sh` crée un fichier temporaire dans `/tmp`, contenant les lignes qui suivent `<<`, puis exécute la commande en redirigeant son entrée standard sur ce fichier. Une autre méthode serait de connecter par un tuyau l'entrée standard de la commande à un processus fils qui envoie les lignes suivant `<<` sur ce tuyau.

# Annexe B

# Un exemple d'utilisation de ioctl

En guise d'exemple, nous allons définir deux fonctions `get_term_params` et `set_term_params` donnant accès aux paramètres d'un pilote de terminal. Ces paramètres sont représentés par le type concret suivant :

```
type term_params =
  { input_modes: int;
    output_modes: int;
    serial_line_modes: int;
    line_discipline_modes: int;
    control_chars: char vect };;
```

Les quatre champs entiers sont des tableaux de bits (faire `man termio` pour avoir la signification de chaque bit). Le champ `input_modes` code le traitement des caractères à l'entrée (vérification de parité ou non, traductions CR/LF, etc.). Le champ `output_modes` code le traitement des caractères à la sortie (traductions CR/LF, expansion des tabs, insertions de délais entre caractères, etc.). Le champ `serial_line_modes` code les paramètres de la liaison série (vitesse, type de contrôle de flux, etc.). Le champ `line_discipline_modes` contrôle l'édition des lignes : entrée ligne par ligne ou caractère par caractère, reconnaissance des caractères d'effacement ou non, reconnaissance des caractères d'interruption (et production de signaux) ou non, etc. Enfin, le tableau `control_chars` indique quels sont les caractères de fin de ligne, d'effacement, d'interruption, etc.

```
let TCGETS = 0x40245408;; (* en SunOS 4.1; confer /usr/include/sys/termios.h *)
let get_term_params fd =
  let buffer = create_string 34 in
  ioctl_ptr fd TCGETS buffer;
  let chars = make_vect 17 `x` in
  for i = 0 to 16 do chars.(i) <- nth_char buffer (i+17) done;
  { input_modes = peek__long buffer 0;
    output_modes = peek__long buffer 4;
    serial_line_modes = peek__long buffer 8;
    line_discipline_modes = peek__long buffer 12;
```

```
    control_chars = chars }
;;

let TCSETSF = 0x8024540b;;  (* confer /usr/include/sys/termios.h *)
let set_term_params fd p =
  let buffer = create_string 34 in
  poke__long buffer 0 p.input_modes;
  poke__long buffer 4 p.output_modes;
  poke__long buffer 8 p.serial_line_modes;
  poke__long buffer 12 p.line_discipline_modes;
  for i = 0 to 16 do set_nth_char buffer (i+17) chars.(i) done;
  ioctl_ptr fd TCSETSF buffer;
  ()
;;
```

# Annexe C

# Interfaces

## C.1 unix: Interface to the Unix system

### Error report

```
type error =
    ENOERR
  | EPERM              (* Not owner *)
  | ENOENT             (* No such file or directory *)
  | ESRCH              (* No such process *)
  | EINTR              (* Interrupted system call *)
  | EIO                (* I/O error *)
  | ENXIO              (* No such device or address *)
  | E2BIG              (* Arg list too long *)
  | ENOEXEC            (* Exec format error *)
  | EBADF              (* Bad file number *)
  | ECHILD             (* No children *)
  | EAGAIN             (* No more processes *)
  | ENOMEM             (* Not enough core *)
  | EACCES             (* Permission denied *)
  | EFAULT             (* Bad address *)
  | ENOTBLK            (* Block device required *)
  | EBUSY              (* Mount device busy *)
  | EEXIST             (* File exists *)
  | EXDEV              (* Cross-device link *)
  | ENODEV             (* No such device *)
  | ENOTDIR            (* Not a directory*)
  | EISDIR             (* Is a directory *)
  | EINVAL             (* Invalid argument *)
  | ENFILE             (* File table overflow *)
  | EMFILE             (* Too many open files *)
  | ENOTTY             (* Not a typewriter *)
```

```
| ETXTBSY           (* Text file busy *)
| EFBIG             (* File too large *)
| ENOSPC            (* No space left on device *)
| ESPIPE            (* Illegal seek *)
| EROFS             (* Read-only file system *)
| EMLINK            (* Too many links *)
| EPIPE             (* Broken pipe *)
| EDOM              (* Argument too large *)
| ERANGE            (* Result too large *)
| EWOULDBLOCK       (* Operation would block *)
| EINPROGRESS       (* Operation now in progress *)
| EALREADY          (* Operation already in progress *)
| ENOTSOCK          (* Socket operation on non-socket *)
| EDESTADDRREQ      (* Destination address required *)
| EMSGSIZE          (* Message too long *)
| EPROTOTYPE        (* Protocol wrong type for socket *)
| ENOPROTOOPT       (* Protocol not available *)
| EPROTONOSUPPORT   (* Protocol not supported *)
| ESOCKTNOSUPPORT   (* Socket type not supported *)
| EOPNOTSUPP        (* Operation not supported on socket *)
| EPFNOSUPPORT      (* Protocol family not supported *)
| EAFNOSUPPORT      (* Address family not supported by protocol family *)
| EADDRINUSE        (* Address already in use *)
| EADDRNOTAVAIL     (* Can't assign requested address *)
| ENETDOWN          (* Network is down *)
| ENETUNREACH       (* Network is unreachable *)
| ENETRESET         (* Network dropped connection on reset *)
| ECONNABORTED      (* Software caused connection abort *)
| ECONNRESET        (* Connection reset by peer *)
| ENOBUFS           (* No buffer space available *)
| EISCONN           (* Socket is already connected *)
| ENOTCONN          (* Socket is not connected *)
| ESHUTDOWN         (* Can't send after socket shutdown *)
| ETOOMANYREFS      (* Too many references: can't splice *)
| ETIMEDOUT         (* Connection timed out *)
| ECONNREFUSED      (* Connection refused *)
| ELOOP             (* Too many levels of symbolic links *)
| ENAMETOOLONG      (* File name too long *)
| EHOSTDOWN         (* Host is down *)
| EHOSTUNREACH      (* No route to host *)
| ENOTEMPTY         (* Directory not empty *)
| EPROCLIM          (* Too many processes *)
| EUSERS            (* Too many users *)
| EDQUOT            (* Disc quota exceeded *)
| ESTALE            (* Stale NFS file handle *)
```

```
  | EREMOTE             (* Too many levels of remote in path *)
  | EIDRM               (* Identifier removed *)
  | EDEADLK             (* Deadlock condition. *)
  | ENOLCK              (* No record locks available. *)
  | ENOSYS              (* Function not implemented *)
  | EUNKNOWNERR
```

The type of error codes.

`exception Unix_error of error * string * string`

Raised by the system calls below when an error is encountered. The first component is the error code; the second component is the function name; the third component is the string parameter to the function, if it has one, or the empty string otherwise.

`value error_message : error -> string`

Return a string describing the given error code.

`value handle_unix_error : ('a -> 'b) -> 'a -> 'b`

`handle_unix_error f x` applies `f` to `x` and returns the result. If the exception `Unix_error` is raised, it prints a message describing the error and exits with code 2.

### Interface with the parent process

`value environment : unit -> string vect`

Return the process environment, as an array of strings with the format "variable=value". See also `sys__getenv`.

### Process handling

```
type process_status =
    WEXITED of int
  | WSIGNALED of int * bool
  | WSTOPPED of int
```

The termination status of a process. `WEXITED` means that the process terminated normally by `exit`; the argument is the return code. `WSIGNALED` means that the process was killed by a signal; the first argument is the signal number, the second argument indicates whether a "core dump" was performed. `WSTOPPED` means that the process was stopped by a signal; the argument is the signal number.

```
type wait_flag =
    WNOHANG
  | WUNTRACED
```

Flags for `waitopt`. `WNOHANG` means do not block if no child has died yet, but immediately return with a pid equal to 0. `WUNTRACED` means report also the children that receive stop signals.

`value execv : string -> string vect -> unit`

`execv prog args` execute the program in file `prog`, with the arguments `args`, and the current process environment.

`value execve : string -> string vect -> string vect -> unit`

Same as `execv`, except that the third argument provides the environment to the program executed.

`value execvp : string -> string vect -> unit`

Same as `execv`, except that the program is searched in the path.

`value fork : unit -> int`

Fork a new process. The returned integer is 0 for the child process, the pid of the child process for the parent process.

`value wait : unit -> int * process_status`

Wait until one of the children processes die, and return its pid and termination status.

`value waitopt : wait_flag list -> int * process_status`

Same as `wait`, but takes a list of options to avoid blocking, or report also stopped children.

`value getpid : unit -> int`

Return the pid of the process.

`value getppid : unit -> int`

Return the pid of the parent process.

`value nice : int -> int`

Change the process priority. The integer argument is added to the "nice" value. (Higher values mean lower priority.) Return the new nice value.

## Basic file input/output

`type file_descr`

The abstract type of file descriptors.

```
value stdin : file_descr
value stdout : file_descr
value stderr : file_descr
```

File descriptors for standard input, standard output and standard error.

```
type open_flag =
    O_RDONLY                        (* Open for reading *)
  | O_WRONLY                        (* Open for writing *)
  | O_RDWR                          (* Open for reading and writing *)
  | O_NDELAY                        (* Open in non-blocking mode *)
  | O_APPEND                        (* Open for append *)
  | O_CREAT                         (* Create if nonexistent *)
  | O_TRUNC                         (* Truncate to 0 length if existing *)
  | O_EXCL                          (* Fail if existing *)
```

The flags to `open`.

`type file_perm == int`

The type of file access rights.

`value open : string -> open_flag list -> file_perm -> file_descr`

Open the named file with the given flags. Third argument is the permissions to give to the file if it is created. Return a file descriptor on the named file.

`value close : file_descr -> unit`

Close a file descriptor.

`value read : file_descr -> string -> int -> int -> int`

`read fd buff start len` reads `len` characters from descriptor `fd`, storing them in string `buff`, starting at position `ofs` in string `buff`. Return the number of characters actually read.

`value write : file_descr -> string -> int -> int -> int`

`write fd buff start len` writes `len` characters to descriptor `fd`, taking them from string `buff`, starting at position `ofs` in string `buff`. Return the number of characters actually written.

### Interfacing with the standard input/output library (module io).

`value in_channel_of_descr : file_descr -> in_channel`

Create an input channel reading from the given descriptor.

`value out_channel_of_descr : file_descr -> out_channel`

Create an output channel writing on the given descriptor.

`value descr_of_in_channel : in_channel -> file_descr`

Return the descriptor corresponding to an input channel.

`value descr_of_out_channel : out_channel -> file_descr`

Return the descriptor corresponding to an output channel.

### Seeking and truncating

```
type seek_command =
    SEEK_SET
  | SEEK_CUR
  | SEEK_END
```

Positioning modes for `lseek`. `SEEK_SET` indicates positions relative to the beginning of the file, `SEEK_CUR` relative to the current position, `SEEK_END` relative to the end of the file.

`value lseek : file_descr -> int -> seek_command -> int`

Set the current position for a file descriptor

`value truncate : string -> int -> unit`

Truncates the named file to the given size.

`value ftruncate : file_descr -> int -> unit`

Truncates the file corresponding to the given descriptor to the given size.

### File statistics

```
type file_kind =
    S_REG                               (* Regular file *)
  | S_DIR                               (* Directory *)
  | S_CHR                               (* Character device *)
  | S_BLK                               (* Block device *)
  | S_LNK                               (* Symbolic link *)
```

```
  | S_FIFO                              (* Named pipe *)
  | S_SOCK                              (* Socket *)
type stats =
  { st_dev : int;                       (* Device number *)
    st_ino : int;                       (* Inode number *)
    st_kind : file_kind;                (* Kind of the file *)
    st_perm : file_perm;                (* Access rights *)
    st_nlink : int;                     (* Number of links *)
    st_uid : int;                       (* User id of the owner *)
    st_gid : int;                       (* Group id of the owner *)
    st_rdev : int;                      (* Device minor number *)
    st_size : int;                      (* Size in bytes *)
    st_atime : int;                     (* Last access time *)
    st_mtime : int;                     (* Last modification time *)
    st_ctime : int }                    (* Last status change time *)
```

The informations returned by the `stat` calls.

`value stat : string -> stats`

Return the information for the named file.

`value lstat : string -> stats`

Same as `stat`, but in case the file is a symbolic link, return the information for the link itself.

`value fstat : file_descr -> stats`

Return the information for the file associated with the given descriptor.

### Operations on file names

`value unlink : string -> unit`

Removes the named file

`value rename : string -> string -> unit`

`rename old new` changes the name of a file from `old` to `new`.

`value link : string -> string -> unit`

`link source dest` creates a hard link named `dest` to the file named `new`.

### File permissions and ownership

```
type access_permission =
    R_OK                                (* Read permission *)
  | W_OK                                (* Write permission *)
  | X_OK                                (* Execution permission *)
  | F_OK                                (* File exists *)
```

Flags for the `access` call.

`value chmod : string -> file_perm -> unit`

Change the permissions of the named file.

`value fchmod : file_descr -> file_perm -> unit`

Change the permissions of an opened file.

`value chown : string -> int -> int -> unit`

Change the owner uid and owner gid of the named file.

`value fchown : file_descr -> int -> int -> unit`

Change the owner uid and owner gid of an opened file.

`value umask : int -> int`

Set the process creation mask, and return the previous mask.

`value access : string -> access_permission list -> unit`

Check that the process has the given permissions over the named file. Raise `Unix_error` otherwise.

### File descriptor hacking

`value fcntl_int : file_descr -> int -> int -> int`

Interface to `fcntl` in the case where the argument is an integer. The first integer argument is the command code; the second is the integer parameter.

`value fcntl_ptr : file_descr -> int -> string -> int`

Interface to `fcntl` in the case where the argument is a pointer. The integer argument is the command code. A pointer to the string argument is passed as argument to the command. The string argument is usually set up with the functions from modules `peek` and `poke`.

### Directories

`value mkdir : string -> file_perm -> unit`

Create a directory with the given permissions.

`value chdir : string -> unit`

Change the process working directory.

`value rmdir : string -> unit`

Remove an empty directory.

`type dir_handle`

The type of descriptors over opened directories.

`value opendir : string -> dir_handle`

Open a descriptor on a directory

`value readdir : dir_handle -> string`

Return the next entry in a directory

`value rewinddir : dir_handle -> unit`

Reposition the descriptor to the beginning of the directory

`value closedir : dir_handle -> unit`

Close a directory descriptor.

### Pipes and redirections

`value pipe : unit -> file_descr * file_descr`

Create a pipe. The first component of the result is opened for reading, that's the exit to the pipe. The second component is opened for writing, that's the entrace to the pipe.

`value dup : file_descr -> file_descr`

Duplicate a descriptor.

`value dup2 : file_descr -> file_descr -> unit`

`dup2 fd1 fd2` duplicates `fd1` to `fd2`, closing `fd2` if already opened.

### Symbolic links

`value symlink : string -> string -> unit`

`symlink source dest` creates the file `dest` as a symbolic link to the file `source`.

`value readlink : string -> string`

Read the contents of a link.

### Named pipes

`value mkfifo : string -> file_perm -> unit`

Create a named pipe with the given permissions.

### Special files

`value ioctl_int : file_descr -> int -> int -> int`

Interface to `ioctl` in the case where the argument is an integer. The first integer argument is the command code; the second is the integer parameter.

`value ioctl_ptr : file_descr -> int -> string -> int`

Interface to `ioctl` in the case where the argument is a pointer. The integer argument is the command code. A pointer to the string argument is passed as argument to the command. The string argument is usually set up with the functions from modules `peek` and `poke`.

### Polling

```
value select :
  file_descr list -> file_descr list -> file_descr list -> float ->
        file_descr list * file_descr list * file_descr list
```

Wait until some input/output operations become possible on some channels. The three list arguments are, respectively, a set of descriptors to check for reading (first argument), for writing (second argument), or for exceptional conditions (third argument). The fourth argument is the maximal timeout, in seconds; a negative fourth argument means no timeout (unbounded wait). The result is composed of three sets of descriptors: those ready for reading (first component), ready for writing (second component), and over which an exceptional condition is pending (third component).

### Locking

```
type lock_command =
    F_ULOCK               (* Unlock a region *)
  | F_LOCK                (* Lock a region, and block if already locked *)
  | F_TLOCK               (* Lock a region, or fail if already locked *)
  | F_TEST                (* Test a region for other process' locks *)
```

Commands for `lockf`.

```
value lockf : file_descr -> lock_command -> int -> unit
```

`lockf fd cmd size` puts a lock on a region of the file opened as `fd`. The region starts at the current read/write position for `fd` (as set by `lseek`), and extends `size` bytes forward if `size` is positive, `size` bytes backwards if `size` is negative, or to the end of the file if `size` is zero.

### Signals

```
type signal =
    SIGHUP              (* hangup *)
  | SIGINT              (* interrupt *)
  | SIGQUIT             (* quit *)
  | SIGILL              (* illegal instruction (not reset when caught) *)
  | SIGTRAP             (* trace trap (not reset when caught) *)
  | SIGABRT             (* used by abort *)
  | SIGEMT              (* EMT instruction *)
  | SIGFPE              (* floating point exception *)
  | SIGKILL             (* kill (cannot be caught or ignored) *)
  | SIGBUS              (* bus error *)
  | SIGSEGV             (* segmentation violation *)
  | SIGSYS              (* bad argument to system call *)
  | SIGPIPE             (* write on a pipe with no one to read it *)
  | SIGALRM             (* alarm clock *)
  | SIGTERM             (* software termination signal from kill *)
  | SIGURG              (* urgent condition on IO channel *)
  | SIGSTOP             (* sendable stop signal not from tty *)
  | SIGTSTP             (* stop signal from tty *)
  | SIGCONT             (* continue a stopped process *)
  | SIGCHLD             (* to parent on child stop or exit *)
  | SIGIO               (* input/output possible signal *)
  | SIGXCPU             (* exceeded CPU time limit *)
  | SIGXFSZ             (* exceeded file size limit *)
  | SIGVTALRM           (* virtual time alarm *)
  | SIGPROF             (* profiling time alarm *)
  | SIGWINCH            (* window changed *)
  | SIGLOST             (* resource lost (eg, record-lock lost) *)
  | SIGUSR1             (* user defined signal 1 *)
  | SIGUSR2             (* user defined signal 2 *)
```

The type of signals.

```
type signal_handler =
    Signal_default                        (* Default behavior for the signal *)
  | Signal_ignore                         (* Ignore the signal *)
  | Signal_handle of (unit -> unit)       (* Call the given function
                                             when the signal occurs. *)
```

The behavior on receipt of a signal

```
value kill : int -> signal -> unit
```

Send a signal to the process with the given process id.

```
value signal : signal -> signal_handler -> unit
```

Set the behavior to be taken on receipt of the given signal.

```
value pause : unit -> unit
```

Wait until a non-ignored signal is delivered.

## Time functions

```
type process_times =
  { tms_utime : float;          (* User time for the process *)
    tms_stime : float;          (* System time for the process *)
    tms_cutime : float;         (* User time for the children processes *)
    tms_cstime : float }        (* System time for the children processes *)
```

The execution times (CPU times) of a process.

```
type tm =
  { tm_sec : int;                         (* Seconds 0..59 *)
    tm_min : int;                         (* Minutes 0..59 *)
    tm_hour : int;                        (* Hours 0..23 *)
    tm_mday : int;                        (* Day of month 1..31 *)
    tm_mon : int;                         (* Month of year 0..11 *)
    tm_year : int;                        (* Year - 1900 *)
    tm_wday : int;                        (* Day of week (Sunday is 0) *)
    tm_yday : int;                        (* Day of year 0..365 *)
    tm_isdst : bool }                     (* Daylight time savings in effect *)
```

The type representing wallclock time and calendar date.

```
value time : unit -> int
```

Return the current time since 00:00:00 GMT, Jan. 1, 1970, in seconds.

```
value gmtime : int -> tm
```

Convert a time in seconds, as returned by `time`, into a date and a time. Assumes Greenwich meridian time zone.

```
value localtime : int -> tm
```

Convert a time in seconds, as returned by `time`, into a date and a time. Assumes the local time zone.

```
value alarm : int -> int
```

Schedule a `SIGALRM` signals after the given number of seconds.

`value sleep : int -> unit`

Stop execution for the given number of seconds.

`value times : unit -> process_times`

Return the execution times of the process.

`value utimes : string -> int -> int -> unit`

Set the last access time (second arg) and last modification time (third arg) for a file. Times are expressed in seconds from 00:00:00 GMT, Jan. 1, 1970.

### User id, group id

`value getuid : unit -> int`

Return the user id of the user executing the process.

`value geteuid : unit -> int`

Return the effective user id under which the process runs.

`value setuid : int -> unit`

Set the real user id and effective user id for the process.

`value getgid : unit -> int`

Return the group id of the user executing the process.

`value getegid : unit -> int`

Return the effective group id under which the process runs.

`value setgid : int -> unit`

Set the real group id and effective group id for the process.

`value getgroups : unit -> int list`

Return the list of groups to which the user executing the process belongs.

```
type passwd_entry =
  { pw_name : string;
    pw_passwd : string;
    pw_uid : int;
    pw_gid : int;
    pw_gecos : string;
    pw_dir : string;
    pw_shell : string }
```

Structure of entries in the `passwd` database.

```
type group_entry =
  { gr_name : string;
    gr_passwd : string;
    gr_gid : int;
    gr_mem : string vect }
```

Structure of entries in the `groups` database.

`value getlogin : unit -> string`

Return the login name of the user executing the process.

`value getpwnam : string -> passwd_entry`

Find an entry in `passwd` with the given name, or raise `Not_found`.

```
value getgrnam : string -> group_entry
```

Find an entry in `group` with the given name, or raise `Not_found`.

```
value getpwuid : int -> passwd_entry
```

Find an entry in `passwd` with the given user id, or raise `Not_found`.

```
value getgrgid : int -> group_entry
```

Find an entry in `group` with the given group id, or raise `Not_found`.

### Internet addresses

```
type inet_addr
```

The abstract type of Internet addresses.

```
value inet_addr_of_string : string -> inet_addr
value string_of_inet_addr : inet_addr -> string
```

Conversions between string with the format `XXX.YYY.ZZZ.TTT` and Internet addresses. `inet_addr_of_string` raises `Failure` when given a string that does not match this format.

### Sockets

```
type socket_domain =
    PF_UNIX                        (* Unix domain *)
  | PF_INET                        (* Internet domain *)
```

The type of socket domains.

```
type socket_type =
    SOCK_STREAM                    (* Stream socket *)
  | SOCK_DGRAM                     (* Datagram socket *)
  | SOCK_RAW                       (* Raw socket *)
  | SOCK_SEQPACKET                 (* Sequenced packets socket *)
```

The type of socket kinds, specifying the semantics of communications.

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

The type of socket addresses. `ADDR_UNIX name` is a socket address in the Unix domain; `name` is a file name in the file system. `ADDR_INET(addr,port)` is a socket address in the Internet domain; `addr` is the Internet address of the machine, and `port` is the port number.

```
type shutdown_command =
    SHUTDOWN_RECEIVE               (* Close for receiving *)
  | SHUTDOWN_SEND                  (* Close for sending *)
  | SHUTDOWN_ALL                   (* Close both *)
```

The type of commands for `shutdown`.

```
type msg_flag =
    MSG_OOB
  | MSG_DONTROUTE
  | MSG_PEEK
```

The flags for `recv`, `recvfrom`, `send` and `sendto`.

```
value socket : socket_domain -> socket_type -> int -> file_descr
```

Create a new socket in the given domain, and with the given kind. The third argument is the protocol type; 0 selects the default protocol for that kind of sockets.

```
value socketpair :
      socket_domain -> socket_type -> int -> file_descr * file_descr
```

Create a pair of unnamed sockets, connected together.

```
value accept : file_descr -> file_descr * sockaddr
```

Accept connections on the given socket. The returned descriptor is a socket connected to the client; the returned address is the address of the connecting client.

```
value bind : file_descr -> sockaddr -> unit
```

Bind a socket to an address.

```
value connect : file_descr -> sockaddr -> unit
```

Connect a socket to an address.

```
value listen : file_descr -> int -> unit
```

Set up a socket for receiving connection requests. The integer argument is the maximal number of pending requests.

```
value recv : file_descr -> string -> int -> int -> msg_flag list -> int
value recvfrom :
      file_descr -> string -> int -> int -> msg_flag list -> int * sockaddr
```

Receive data from an unconnected socket.

```
value send : file_descr -> string -> int -> int -> msg_flag list -> int
```

Send data over an unconnected socket.

```
value sendto :
      file_descr -> string -> int -> int -> msg_flag list -> sockaddr -> int
value shutdown : file_descr -> shutdown_command -> int
```

### Host and protocol databases

```
type host_entry =
  { h_name : string;
    h_aliases : string vect;
    h_addrtype : socket_domain;
    h_addr_list : inet_addr vect }
```

Structure of entries in the `hosts` database.

```
type protocol_entry =
  { p_name : string;
    p_aliases : string vect;
    p_proto : int }
```

Structure of entries in the `protocols` database.

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

Structure of entries in the `services` database.

`value gethostname : unit -> string`

Return the name of the local host.

`value gethostbyname : string -> host_entry`

Find an entry in `hosts` with the given name, or raise `Not_found`.

`value gethostbyaddr : inet_addr -> host_entry`

Find an entry in `hosts` with the given address, or raise `Not_found`.

`value getprotobyname : string -> protocol_entry`

Find an entry in `protocols` with the given name, or raise `Not_found`.

`value getprotobynumber : int -> protocol_entry`

Find an entry in `protocols` with the given protocol number, or raise `Not_found`.

`value getservbyname : string -> string -> service_entry`

Find an entry in `services` with the given name, or raise `Not_found`.

`value getservbyport : int -> string -> service_entry`

Find an entry in `services` with the given service number, or raise `Not_found`.

## C.2 `peek`: To fetch various objects in C format from a string

Arguments: a string and an offset into that string. Result: as follows.

`value short : string -> int -> int`

A signed short

`value ushort : string -> int -> int`

An unsigned short

`value int : string -> int -> int`

A signed int

`value uint : string -> int -> int`

An unsigned int

`value long : string -> int -> int`

A signed long (currenty truncated to 31 bits)

`value ulong : string -> int -> int`

An unsigned long (currently trucated to 31 bits)

`value nshort : string -> int -> int`

An unsigned short in network byte order

`value nlong : string -> int -> int`

An unsigned long in network byte order

`value cstring : string -> int -> string`

A null-terminated string

## C.3 `poke`: To store various objects in a string, in C format

Arguments: the destination string, the offset into that string, and the object to poke.
Result: none.

`value short : string -> int -> int -> unit`

A short

`value int : string -> int -> int -> unit`

An int

`value long : string -> int -> int -> unit`

A long

`value nshort : string -> int -> int -> unit`

An unsigned short in network byte order

`value nlong : string -> int -> int -> unit`

An unsigned long in network byte order