Programmation du système Unix en Objective Caml

Xavier Leroy et Didier Rémy[1]

1. INRIA Rocquencourt

Résumé

Ce document 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 Objective Caml, 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 Objective Caml

This document 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 Objective Caml 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.

Table des matières

Introduction

Ces notes sont issues d'un cours de programmation système que Xavier Leroy a enseigné en première année du Magistère de Mathématiques Fondamentales et Appliquées et d'Informatique de l'École Normale Supérieure en 1994. Cette première version utilisait le langage Caml-Light [1]. Didier Rémy en a fait une traduction pour le langage OCaml [2] pour un cours enseigné en Majeure d'Informatique à l'École Polytechnique de 2003 à 2006. À cette occasion, Gilles Roussel, Fabrice Le Fessant et Maxence Guesdon qui ont aidé à ce cours ont également contribué à améliorer ces notes. Cette version comporte des ajouts et quelques mises à jour : en presque une décennie certains ordres de grandeur ont décalé leur virgule d'un chiffre ; aussi, la toile était seulement en train d'être tissée et l'exemple, aujourd'hui classique, du relais HTTP aurait presqu'eut un coté précurseur en 1994. Mais surtout le langage OCaml a gagné en maturité depuis et a été utilisé dans de véritables applications système, telles que Unison [16].

La tradition veut que la programmation du système Unix se fasse dans le langage C. Dans le cadre de ce cours, il 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 44.)

De plus, OCaml 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 OCaml est de montrer le langage OCaml à l'œuvre dans un domaine qui sort nettement de ses applications usuelles, à savoir la démonstration automatique, la compilation, et le calcul symbolique. OCaml 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 OCaml 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 OCaml, et avec l'utilisation des commandes Unix, en particulier du shell. On se reportera à la documentation du système OCaml [2] pour toutes les questions relatives au langage, et à la section 1 du manuel Unix ou à un livre d'introduction à Unix [5, 6] 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 [8] ; celle de System V, dans [9]. Les livres de Tanenbaum [11, 12] donnent une vue d'ensemble des

architectures de systèmes et de réseaux.

Le système Objective OCaml dont la bibliothèque d'interface avec Unix présentée ici fait partie intégrante est en accès libre à l'URL `http://caml.inria.fr/ocaml/`.

Chapitre 1

Généralités

1.1 Les modules `Sys` et `Unix`

Les fonctions qui donnent accès au système depuis OCaml 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 OCaml. 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.

Les modules `Sys` et `Unix` peuvent redéfinir certains identificateurs du module `Pervasives` et cacher leur anciennes définitions. Par exemple, `Pervasives.stdin` est différent de `Unix.stdin`. Les anciennes définitions peuvent toujours être obtenues en les préfixant.

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

```
ocamlc -o prog unix.cma mod1.ml mod2.ml mod3.ml
```

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

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

puis faire pour l'édition de lien :

```
ocamlc -o prog unix.cma mod1.cmo mod2.cmo mod3.cmo
```

Dans les deux cas, l'argument `unix.cma` représente la bibliothèque `Unix` écrite en OCaml.

Pour utiliser le compilateur natif plutôt que le bytecode, on remplace `ocamlc` par `ocamlopt` et `unix.cma` par `unix.cmxa`.

On peut aussi accéder au système Unix depuis le système interactif (le "toplevel"). Si le lien dynamique des bibliothèques C est possible sur votre plate-forme, il suffit de lancer le toplevel `ocaml` et de taper la directive

```
#load "unix.cma";;
```

Sinon, il faut d'abord créer un système interactif contenant les fonctions systèmes pré-chargées :

```
ocamlmktop -o ocamlunix unix.cma
```

Ce système se lance ensuite par :

```
./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 `argv` :

```
Sys.argv : string array
```

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

```
Unix.environment : unit -> string array
```

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

```
Unix.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.

Exemple: Comme premier exemple, voici le programme `echo` qui affiche la liste de ses arguments, comme le fait la commande Unix de même nom.

```
1  let echo() =
2    let len = Array.length Sys.argv in
3    if len > 1 then
4      begin
5        print_string Sys.argv.(1);
6        for i = 2 to len - 1 do
7          print_char ' ';
8          print_string Sys.argv.(i);
9        done;
10       print_newline();
11     end;;
12 echo();;
```

Un programme peut terminer prématurément par l'appel `exit` :

```
val 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". Lorsqu'un programme termine normalement après avoir exécuté toutes les phrases qui le composent, il effectue un appel implicite à `exit 0`. Lorsqu'un programme termine prématurément parce qu'une exception levée n'a pas été rattrapée, il effectue un appel implicite à `exit 2`. La fonction `exit` vide toujours les tampons des canaux ouverts en écriture. La fonction `at_exit` permet d'enregistrer d'autres actions à effectuer au moment de la terminaison du programme.

```
val at_exit : (unit -> unit) -> unit
```

La fonction enregistrée la dernière est appelée en premier. L'enregistrement d'une fonction avec `at_exit` ne peut pas être ultérieurement annulé. Cependant, ceci n'est pas une véritable restriction, car on peut facilement obtenir cet effet en enregistrant une fonction dont l'exécution dépend d'une variable globale.

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` (voir page 138 pour une description complète) :

type `error = E2BIG | EACCES | EAGAIN | ... | EUNKNOWNERR` **of** `int`

Les constructeurs de ce type reprennent les mêmes noms et les mêmes significations que ceux employés dans la norme POSIX plus certaines erreurs de UNIX98 et BSD. Toutes les autres erreurs sont rapportées avec le constructeur `EUNKNOWNERR`.

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 pour des petites applications. Il convient néanmoins de l'afficher de manière claire. Pour ce faire, le module `Unix` fournit la fonctionnelle `handle_unix_error`.

val `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 prog ();;
```

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

Pour référence, voici comment est implémentée `handle_unix_error`.

```
1  open Unix;;
2  let handle_unix_error f arg =
3    try
4      f arg
5    with Unix_error(err, fun_name, arg) ->
6      prerr_string Sys.argv.(0);
7      prerr_string ": \"";
8      prerr_string fun_name;
9      prerr_string "\" failed";
10     if String.length arg > 0 then begin
11       prerr_string " on \"";
12       prerr_string arg;
13       prerr_string "\""
14     end;
15     prerr_string ": ";
16     prerr_endline (error_message err);
17     exit 2;;
```

Les fonctions de la forme `prerr_xxx` sont comporte comme les fonction `print_xxx` mais à la différence qu'elles écrivent dans le flux d'erreur `stderr` au lieu d'écrire dans le flux standard `stdout`. De plus `prerr_endline` vide le tampon `stderr` (alors que `print_endline` ne le fait pas).

La primitive `error_message`, de type `error -> string`, renvoie un message décrivant l'erreur donnée en argument (ligne 16). L'argument numéro zéro de la commande, `Sys.argv.(0)`, contient le nom de commande utilisé pour invoquer le programme (ligne 6).

La fonction `handle_unix_error` traite des erreurs fatales, *i.e.* des erreurs qui arrêtent le programme. C'est un avantage de OCaml d'obliger les erreurs à être prises en compte, ne serait-ce qu'au niveau le plus haut provoquant l'arrêt du programme. En effet, toute erreur dans un appel système lève une exception, et le fil d'exécution en cours est interrompu jusqu'à un niveau où elle est explicitement rattrapée et donc traitée. Cela évite de continuer le programme dans une situation incohérente.

Les erreurs de type `Unix_error` peuvent aussi, bien sûr, être filtrée sélectivement. Par exemple, on retrouvera souvent plus loin la fonction suivante

```
let rec restart_on_EINTR f x =
  try f x with Unix_error (EINTR, _, _) -> restart_on_EINTR f x
```

qui est utilisé pour exécuter une fonction et la relancer automatiquement lorsque elle est interrompue par un appel système (voir 4.5).

1.4 Fonctions de bibliothèque

Nous verrons au travers d'exemples que la programmation système reproduit souvent les mêmes motifs. Nous seront donc souvent tentés de définir des fonctions de bibliothèque permettant de factoriser les parties communes et ainsi de réduire le code de chaque application à sa partie essentielle.

Alors que dans un programme complet on connaît précisément les erreurs qui peuvent être levées et celles-ci sont souvent fatales (on arrête le programme), on ne connaît pas en général le contexte d'exécution d'une fonction de bibliothèque. On ne peut pas supposer que les erreurs sont fatales. Il faut donc laisser l'erreur retourner à l'appelant qui pourra décider d'une action appropriée (arrêter le programme, traiter ou ignorer l'erreur). Cependant, la fonction de libraire ne va pas en général pas se contenter de regarder l'erreur passer, elle doit maintenir le système dans un état cohérent. Par exemple, une fonction de bibliothèque qui ouvre un fichier puis applique une opération sur le descripteur associé à ce fichier devra prendre soin de refermer le descripteur dans tous les cas de figure, y compris lorsque le traitement du fichier provoque une erreur. Ceci afin d'éviter une fuite mémoire conduisant à l'épuisement des descripteurs de fichiers.

De plus le traitement appliqué au fichier peut être donné par une fonction reçu en argument et on ne sait donc pas précisément quand ni comment le traitement peut échouer (mais l'appelant en général le sait). On sera donc souvent amené à protéger le corps du traitement par un code dit de «finalisation» qui devra être exécuté juste avant le retour de la fonction que celui-ci soit normal ou exceptionnel.

Il n'y a pas de construction primitive de finalisation `try ... finalize` dans le langage OCaml mais on peut facilement la définir [1] :

```
let try_finalize f x finally y =
  let res = try f x with exn -> finally y; raise exn in
  finally y;
  res
```

Cette fonction reçoit le corps principal `f` et le traitement de finalisation `finally`, chacun sous la forme d'une fonction, et deux paramètres `x` et `y` à passer respectivement à chacune des deux fonctions pour les lancer. Le corps du programme `f x` est exécuté en premier et son résultat est

1. Une construction primitive n'en serait pas moins avantageuse.

gardé de coté pour être retourné après l'exécution du code de finalisation `finally y`. Lorsque le corps du programme échoue, *i.e.* lève une exception `exn`, alors le code de finalisation est exécuté puis l'exception `exn` est relancée. Si à la fois le code principal et le code de finalisation échouent, l'exception lancée est celle du code de finalisation (on pourrait faire le choix inverse).

Note Dans le reste du cours, nous utiliserons une bibliothèque auxiliaire `Misc` qui regroupe quelques fonctions d'usage général, telle que `try_finalize`, souvent utilisées dans les exemples et que nous introduirons au besoin. L'interface du module `Misc` est donnée en appendice B.7. Pour compiler les exemples du cours, il faut donc dans un premier temps rassembler les définitions du module `Misc` et le compiler.

Le module `Misc` contient également certaines fonctions ajoutées à titre d'illustration qui ne sont pas utilisées directement dans le cours. Elles enrichissent simplement la bibliothèque `Unix` ou en redéfinissent le comportement de certaines fonctions. Le module `Misc` doit prendre priorité sur le module `Unix`.

Exemples Le cours comporte de nombreux exemples. Ceux-ci ont été compilés avec OCaml, version 3.10). Certains programmes doivent être légèrement modifiés pour être adaptés à une version plus ancienne.

Les exemples sont essentiellement de deux types : soit ce sont des fonctions réutilisables d'usage assez général, dites «fonctions de bibliothèque», soit ce sont de petites applications. Il est important de faire la différence entre ces deux types d'exemples. Dans le premier cas, on voudra laisser le contexte d'utilisation de la fonction le plus large possible, et on prendra donc soin de bien spécifier son interface et de bien traiter tous les cas particuliers. Dans le second cas, une erreur est souvent fatale et entraîne l'arrêt du programme en cours. Il suffit alors de rapporter correctement la cause de l'erreur, sans qu'il soit besoin de revenir à un état cohérent, puisque le programme sera arrêté immédiatement après le report de l'erreur.

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.

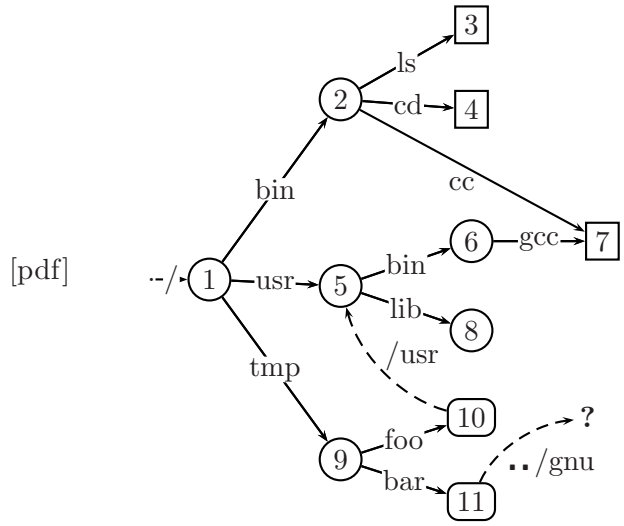
La représentation d'un fichier contient à la fois les données contenues dans le fichier et des informations sur le fichier (aussi appelées méta-données) telles que son type, les droits d'accès, les dernières dates d'accès, *etc.*

2.1 Le système de fichiers

En première approximation, le système de fichier est un arbre. La racine est notée `'/'`. Les arcs sont étiquetés par des noms (de fichiers), formés d'une chaîne de caractères quelconques à l'exception des seuls caractères `'\000'` et `'/'`, mais il est de bon usage d'éviter également les caractères non imprimables ainsi que les espaces. Les nœuds non terminaux du système de fichiers sont appelés *répertoires* : il contiennent toujours deux arcs `.` et `..` qui désignent respectivement le répertoire lui-même et le répertoire parent. Les autres nœuds sont parfois appelés fichiers, par opposition aux répertoires, mais cela reste ambigu, car on peut aussi désigner par fichier un nœud quelconque. Pour éviter toute ambiguïté, on pourra parler de ≪ fichiers non répertoires ≫.

Les nœuds du système de fichiers sont désignés par des chemins. Ceux-ci peuvent se référer à l'origine de la hiérarchie et on parlera de chemins absolus, ou à un répertoire (en général le répertoire de travail). Un chemin relatif est une suite de noms de fichiers séparés par le caractère `'/'` ; un chemin absolu est un chemin relatif précédé par le caractère `'/'` (notez le double usage de ce caractère comme séparateur de chemin et comme le nom de la racine).

La bibliothèque `Filename` permet de manipuler les chemins de façon portable. Notamment `Filename.concat` permet de concaténer des chemins sans faire référence au caractère `'/'`, ce qui permettra au code de fonctionner également sur d'autres architectures (par exemple le caractère de séparation des chemins est `'\'` sous Windows). De même, le module `Filename` donne des noms `currentdir` et `parentdir` pour désigner les arcs `.` et `..`. Les fonctions `Filename.basename` et `Filename.dirname` extraient d'un chemin p un préfixe d et un suffixe b tel que les chemins p et d/b désignent le même fichier, d désigne le répertoire dans lequel se trouve le fichier et b le



– **Liens inverses omis**

– **Liens durs**
7 a deux antécédents 2 et 6

– **Liens symboliques**
10 désigne 5
11 ne désigne aucun nœud

– **Chemins équivalents** de 9 à 8 ?
../usr/lib
./../usr/lib, *etc.*
foo/lib

FIGURE 2.1 – Un petit exemple de hiérarchie de fichiers

nom du fichier dans ce répertoire. Les opérations définies dans `Filename` opèrent uniquement sur les chemins indépendemment de leur existence dans la hiérarchie.

En fait, la hiérarchie n'est pas un arbre. D'abord les répertoires conventionnels **.** et **..** permettent de s'auto-référencer et de remonter dans la hiérarchie, donc de créer des chemins menant d'un répertoire à lui-même. D'autre part les fichiers non répertoires peuvent avoir plusieurs antécédents. On dit alors qu'il a plusieurs ≪liens durs≫. Enfin, il existe aussi des ≪liens symboliques≫ qui se prêtent à une double interprétation. Un lien symbolique est un fichier non répertoire dont le contenu est un chemin. On peut donc interpréter un lien symbolique comme un fichier ordinaire et simplement lire son contenu, un lien. Mais on peut aussi suivre le lien symbolique de façon transparente et ne voir que le fichier cible. Cette dernière est la seule interprétation possible lorsque le lien apparaît au milieu d'un chemin : Si s est un lien symbolique dont la valeur est le chemin ℓ, alors le chemin $p/s/q$ désigne le fichier ℓ/q si ℓ est un lien absolu ou le fichier ou $p/\ell/q$ si ℓ est un lien relatif.

La figure 2.1 donne un exemple de hiérarchie de fichiers. Le lien symbolique 11 désigné par le chemin `/tmp/bar`, dont la valeur est le chemin relatif `../gnu`, ne désigne aucun fichier existant dans la hiérarchie (à cet instant).

En général un parcours récursif de la hiérarchie effectue une lecture arborescente de la hiérarchie :

– les répertoires `currentdir` et `parentdir` sont ignorés.
– les liens symboliques ne sont pas suivis.

Si l'on veut suivre les liens symboliques, on est alors ramené à un parcourt de graphe et il faut garder trace des nœuds déjà visités et des nœuds en cours de visite.

Chaque processus a un répertoire de travail. Celui-ci peut être consulté par la commande `getcwd` et changé par la commande `chdir`. Il est possible de restreindre la vision de la hiérarchie. L'appel `chroot` p fait du nœud p, qui doit être un répertoire, la racine de la hiérarchie. Les chemins absolus sont alors interprétés par rapport à la nouvelle racine (et le chemin **..** appliqué à la nouvelle racine reste bien entendu à la racine).

2.2 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 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 *cmd* `<` f, alors le descripteur `stdin` fait référence au fichier de nom f pendant l'exécition de la commande *cmd*. De même *cmd* `>` f (respectivement *cmd* `2>` f) fait en sorte que le descripteur `stdout` (respectivement `stderr`) fasse reférence au fichier f pendant l'exécution de la commande *cmd*.

2.3 Méta-données, types et permissions

Les appels système `stat`, `lstat` et `fstat` retournent les méta-données sur un fichier, c'est-à-dire les informations portant sur le nœud lui-même plutôt que son contenu. Entre autres, ces informations décrivent l'identité du fichier, son type du fichier, les droits d'accès, les dates des derniers d'accès, plus un certain nombre d'informations supplémentaires.

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

Les appels `stat` et `lstat` prennent un nom de fichier en argument. L'appel `fstat` prend en argument un descripteur déjà ouvert et donne les informations sur le fichier qu'il désigne. La différence entre `stat` et `lstat` se voit sur les liens symboliques : `lstat` renvoie les informations

`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 : `S_REG` fichier normal `S_DIR` répertoire `S_CHR` fichier spécial de type caractère `S_BLK` fichier spécial de type bloc `S_LNK` lien symbolique `S_FIFO` tuyau `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 périphérique 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^{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.

TABLE 2.1 – Champs de la structure `stats`

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*) de type `stats` décrit dans la table 2.1.

Identification

Un fichier est identifié de façon unique par la paire composé de son numéro de périphérique (typiquement la partition sur laquelle il se trouve) `st_dev` et de son numéro d'inode `st_ino`.

Propriétaires

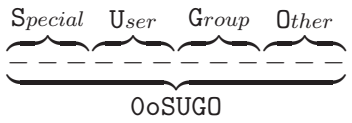
Un fichier a un propriétaire `st_uid` et un groupe propriétaire `st_gid`. L'ensemble des utilisateurs et des groupes d'utilisateurs sur la machine est habituellement décrit dans les fichiers `/etc/passwd` et `/etc/groups`. On peut les interroger de façon portable par nom à l'aide des commandes `getpwnam` et `getgrnam` ou par numéro à l'aide des commandes `getpwuid` et `getgrgid`.

Le nom de l'utilisateur d'un processus en train de tourner et l'ensemble des groupes auxquels il appartient peuvent être récupérés par les commandes `getlogin` et `getgroups`.

L'appel `chown` modifie le propriétaire (deuxième argument) et le groupe propriétaire (troisième argument) d'un fichier (premier argument). Seul le super utilisateur a le droit de changer arbitrairement ces informations. Lorsque le fichier est tenu par un descripteur, on utilisera `fchown` en passant les descripteur au lieu du nom de fichier.

Droits

Les droits sont codés sous forme de bits dans un entier et le type `file_perm` est simplement une abréviation pour le type `int` : Les droits comportent une information en lecture, écriture et exécution pour l'utilisateur, le groupe et les autres, plus des bits spéciaux. Les droits sont donc représentés par un vecteur de bits :



où pour chacun des champs user, group et other on indique dans l'ordre les droits en lecture (`r`), écriture (`w`) et exécution (`x`). Les permissions sur un fichier sont l'union des permissions individuelles :

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
0o1000	--------t	le bit `t` sur le groupe (sticky bit)
0o2000	-----s---	le bit `s` sur le groupe (`set-gid`)
0o4000	--s------	le bit `s` sur l'utilisateur (`set-uid`)

Le sens des droits de lecture et d'écrire est évident ainsi que le droit d'exécution pour un fichier. Pour un répertoire, le droit d'exécution signifie le droit de se placer sur le répertoire (faire `chdir` sur ce répertoire). Le droit de lecture sur un répertoire est nécessaire pour en lister son contenu mais pas pour en lire ses fichiers ou sous-répertoires (mais il faut alors en connaître le nom).

Les bits spéciaux ne prennent de sens qu'en présence du bit `x` (lorsqu'il sont présents sans le bit `x`, ils ne donnent pas de droits supplémentaires). C'est pour cela que leur représentation se superpose à celle du bit `x` et on utilise les lettres `S` et `T` au lieu de `s` et `t` lorsque le bit `x` n'est pas simultanément présent. Le bit `t` permet aux sous-répertoires créés d'hériter des droits du répertoire parent. Pour un répertoire, le bit `s` permet d'utiliser le `uid` ou le `gid` de propriétaire du répertoire plutôt que de l'utilisateur à la création des répertoires. Pour un fichier exécutable, le bit `s` permet de changer au lancement l'*identité effective* de l'utilisateur (`setuid`) ou du groupe (`setgid`). Le processus conserve également ses identités d'origine, à moins qu'il ait les privilèges du super utilisateur, auquel cas, `setuid` et `setgid` changent à la fois son identité effective et son identité d'origine. L'identité effective est celle sous laquelle le processus s'exécute. L'identité d'origine est maintenue pour permettre au processus de reprendre ultérieurement celle-ci comme

effective sans avoir besoin de privilèges. Les appels système `getuid` et `getgid` retournent les identités d'origine et `geteuid` et `getegid` retournent les identités effectives.

Un processus possède également un masque de création de fichiers représenté de la même façon. Comme son nom l'indique, le masque est spécifie des interdictions (droits à masquer) : lors de la création d'un fichier tous les bits à 1 dans le masque de création sont mis à zéro dans les droits du fichier créé. Le masque peut être consulté et changé par la fonction

```
val umask : int -> int
```

Comme pour de nombreux appels système qui modifient une variable système, l'ancienne valeur de la variable est retournée par la fonction de modification. Pour simplement consulter la valeur, il faut donc la modifier deux fois, une fois avec une valeur arbitraire, puis remettre l'ancienne valeur en place. Par exemple, en faisant :

```
let m = umask 0 in ignore (umask m); m
```

Les droits d'accès peuvent être modifiés avec l'appel `chmod`.

On peut également tester les droits d'accès «dynamiquement» avec l'appel système `access`

```
type access_permission = R_OK | W_OK | X_OK | F_OK

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

où les accès demandés sont représentés pas le type `access_permission` dont le sens est immédiat sauf pour `F_OK` qui signifie seulement que le fichier existe (éventuellement sans que le processus ait les droits correspondants).

Notez que `access` peut retourner une information plus restrictive que celle calculée à partir de l'information statique retournée par `lstat` car une hiérarchie de fichiers peut être montrée avec des droits restreints, par exemple en lecture seule. Dans ce cas, `access` refusera le droit d'écrire alors que l'information contenue dans les méta-données relative au fichier peut l'autoriser. C'est pour cela qu'on parle d'information «dynamique» (ce que le processus peut réellement faire) par opposition à «statique» (ce que le système de fichier indique).

2.4 Opérations sur les répertoires

Seul le noyau écrit dans les répertoires (lorsque des fichiers sont créés). Il est donc interdit d'ouvrir un répertoire en écriture. Dans certaines versions d'Unix on peut ouvrir un répertoire en lecture seule et le lire avec `read`, mais d'autres versions l'interdise. Cependant, même si c'est possible, il est préférable de ne pas le faire car 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 :

```
val opendir   : string -> dir_handle
val readdir   : dir_handle -> string
val rewinddir : dir_handle -> unit
val 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. La fonction `rewinddir` repositionne le descripteur au début du répertoire.

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

```
val mkdir : string -> file_perm -> unit
val rmdir : string -> unit
```

Le deuxième argument de `mkdir` encode les droits d'accès donnés au nouveau répertoire. Notez qu'on ne peut détruire qu'un répertoire déjà vide. Pour détruire un répertoire et son contenu, il

faut donc d'abord aller récursivement vider le contenu du répertoire puis détruire le répertoire.

Par exemple, on peut écrire une fonction d'intérêt général dans le module `Misc` qui itère sur les entrées d'un répertoire.

```
1  let iter_dir f dirname =
2    let d = opendir dirname in
3    try while true do f (readdir d) done
4    with End_of_file -> closedir d
```

2.5 Exemple complet : recherche dans la hiérarchie

La commande Unix `find` permet de rechercher récursivement des fichiers dans la hiérarchie selon certains critères (nom, type et droits du fichier) etc. Nous nous proposons ici de réaliser d'une part une fonction de bibliothèque `Findlib.find` permettant d'effectuer de telles recherches et une commande `find` fournissant une version restreinte de la commande Unix `find` n'implantant que les options `-follow` et `-maxdepth`.

Nous imposons l'interface suivante pour la bibliothèque `Findlib` :

```
val find :
  (Unix.error * string * string -> unit) ->
  (string -> Unix.stats -> bool) -> bool -> int -> string list ->
  unit
```

L'appel de fonction `find` *handler action follow depth roots* parcourt la hiérarchie de fichiers à partir des racines indiquées dans la liste *roots* (absolues ou relatives au répertoire courant au moment de l'appel) jusqu'à une profondeur maximale *depth* en suivant les liens symboliques si le drapeau *follow* est vrai. Les chemins trouvés sous une racine r incluent r comme préfixe. Chaque chemin trouvé p est passé à la fonction *action*. En fait, *action* reçoit également les informations `Unix.stat` p si le drapeau *follow* est vrai ou `Unix.lstat` p sinon. La fonction *action* retourne un booléen indiquant également dans le cas d'un répertoire s'il faut poursuivre la recherche en profondeur (`true`) ou l'interrompre (`false`).

La fonction *handler* sert au traitement des erreurs de parcours, nécessairement de type `Unix_error` : les arguments de l'exception sont alors passés à la fonction *handler* et le parcours continue. En cas d'interruption, l'exception est remontée à la fonction appelante. Lorsqu'une exception est levée par les fonctions *action* ou *handler*, elle arrête le parcours de façon abrupte et est remontée immédiatement à l'appelant.

Pour remonter une exception `Unix_error` sans qu'elle puisse être attrapée comme une erreur de parcours, nous la cachons sous une autre exception.

```
1  exception Hidden of exn
2  let hide_exn f x = try f x with exn -> raise (Hidden exn);;
3  let reveal_exn f x = try f x with Hidden exn -> raise exn;;
```

Voici le code de la fonction de parcours.

```
4  open Unix;;
5  let find on_error on_path follow depth roots =
6    let rec find_rec depth visiting filename =
7      try
8        let infos = (if follow then stat else lstat) filename in
9        let continue = hide_exn (on_path filename) infos in
10       let id = infos.st_dev, infos.st_ino in
11       if infos.st_kind = S_DIR && depth > 0 && continue &&
12         (not follow || not (List.mem id visiting))
```

```
13        then
14          let process_child child =
15            if (child <> Filename.current_dir_name &&
16                child <> Filename.parent_dir_name) then
17              let child_name = Filename.concat filename child in
18              let visiting =
19                if follow then id :: visiting else visiting in
20              find_rec (depth-1) visiting child_name in
21          Misc.iter_dir process_child filename
22      with Unix_error (e, b, c) -> hide_exn on_error (e, b, c) in
23    reveal_exn (List.iter (find_rec depth [])) roots;;
```

Les répertoires sont identifiés par la paire `id` (ligne 21) constituée de leur numéro de périphérique et de leur numéro d'inode. La liste `visiting` contient l'ensemble des répertoires en train d'être visités. En fait cette information n'est utile que si l'on suit les liens symboliques (ligne 19).

On peut maintenant en déduire facilement la commande `find`.

```
1  let find () =
2    let follow = ref false in
3    let maxdepth = ref max_int in
4    let roots = ref [] in
5    let usage_string  =
6      ("Usage: " ^ Sys.argv.(0) ^ " [files...] [options...]") in
7    let opt_list =  [
8      "-maxdepth", Arg.Int ((:=) maxdepth), "max depth search";
9      "-follow", Arg.Set follow, "follow symbolic links";
10   ] in
11   Arg.parse opt_list (fun f -> roots := f :: !roots) usage_string;
12   let action p infos = print_endline p; true in
13   let errors = ref false in
14   let on_error (e, b, c) =
15     errors := true; prerr_endline (c ^ ": " ^ Unix.error_message e) in
16   Findlib.find on_error action !follow !maxdepth
17     (if !roots = [] then [ Filename.current_dir_name ]
18      else List.rev !roots);
19   if !errors then exit 1;;
20
21 Unix.handle_unix_error find ();;
```

L'essentiel du code est constitué par l'analyse de la ligne de commande, pour laquelle nous utilisons la bibliothèque `Arg`.

Bien que la commande `find` implantée ci-dessus soit assez restreinte, la fonction de bibliothèque `Findlib.find` est quant à elle très générale, comme le montre l'exercice suivant.

Exercice 1 *Utiliser la bibliothèque* `Findlib` *pour écrire un programme* `find_but_CVS` *équivalent à la commande Unix* `find . -type d -name CVS -prune -o -print` *qui imprime récursivement les fichiers à partir du répertoire courant mais sans voir (ni imprimer, ni visiter) les répertoires de nom* `CVS`. (Voir le corrigé) □

Exercice 2 *La fonction* `getcwd` *n'est pas un appel système mais définie en bibliothèque. Donner une implémentation «primitive» de* `getcwd`. *Décrire le principe de l'algorithme.*

(Voir le corrigé)

Puis écrire l'algorithme (on évitera de répéter plusieurs fois le même appel système). □

2.6 Ouverture d'un fichier

La primitive `openfile` permet d'obtenir un descripteur sur un fichier d'un certain nom (l'appel système correspond est `open`, mais `open` est un mot clé en OCaml).

val `openfile : string -> open_flag list -> file_perm -> 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 de type `file_perm` 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 `openfile` 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 parmi les 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
`O_NONBLOCK`	ouverture en mode non bloquant
`O_NOCTTY`	ne pas fonctionner en mode terminal de contrôle
`O_SYNC`	effectuer les écritures en mode synchronisé
`O_DSYNC`	effectuer les écritures de données en mode synchronisé
`O_RSYNC`	effectuer les lectures en mode synchronisé

Le premier groupe indique le comportement à suivre selon que le fichier existe ou non.

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).

Exemple: la plupart des programmes prennent `0o666` comme troisième argument de `openfile`, 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, `openfile` 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 le verrou appelle `openfile` 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, `openfile` déclenche une erreur, et il faut attendre un peu, puis réessayer. Si le fichier n'existe pas, `openfile` 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. La création d'un fichier est une opération atomique : si deux processus essayent de créer un même fichier en parallèle avec les options `O_EXCL` et `O_CREAT`, au plus un seul des deux seulement peut réussir. Évidemment cette méthode n'est pas très satisfaisante car d'une part le processus qui n'a pas le verrou doit être en attente active, d'autre part un processus qui se termine anormalement peux laisser le verrou bloqué.

Exemple: pour se préparer à lire un fichier :

```
openfile 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 :

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

Si le fichier qu'on ouvre 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 :

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

Si le fichier qu'on ouvre 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 :

```
openfile 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 :

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

Le drapeau `O_NONBLOCK` assure que si le support est un tuyau nommé ou un fichier spécial, alors l'ouverture du fichier ainsi que les lectures et écritures ultérieur se feront en mode non bloquant.

Le drapeau `O_NOCTYY` assure que si le support est un terminal de contrôle (clavier, fenêtre, *etc.*), alors celui-ci ne devient pas le terminal de contrôle du processus appelant.

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 dernier groupe de drapeaux indique comment synchroniser les opérations de lectures et écritures. Par défaut, ces opérations ne sont pas synchronisées.

Si `O_DSYNC` est fourni, les données sont écrites de façon synchronisée de telle façon que la commande est bloquante et ne retourne que lorsque toutes les écritures auront été effectuées physiquement sur le support (disque en général).

Si `O_SYNC` est fourni, ce sont à la fois les données et les informations sur le fichier qui sont synchronisées.

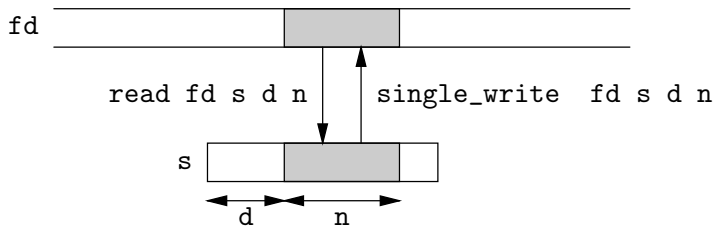
Si `O_RSYNC` est fourni en présence de `O_DSYNC` les lectures des données sont également synchronisées : il est assuré que toutes les écritures en cours (demandées mais pas nécessairement enregistrées) sur ce fichier seront effectivement écrites sur le support avant la prochaine lecture. Si `O_RSYNC` est fourni en présence de `O_SYNC` cela s'applique également aux informations sur le fichier.

2.7 Lecture et écriture

Les appels systèmes `read` et `write` permettent de lire et d'écrire les octets d'un fichier. Pour des raisons historiques, l'appel système `write` est relevé en OCaml sous le nom `single_write` :

```
val read  : file_descr -> string -> int -> int -> int
val single_write : file_descr -> string -> int -> int -> int
```

Les deux appels `read` et `single_write` 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 caractères contenant les octets à écrire (cas de `single_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. Le troisième argument et le quatrième argument 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 `single_write` ne vérifient pas ce fait.)



L'entier renvoyé par `read` ou `single_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 normalement le nombre d'octets demandés, mais il y a plusieurs exceptions à ce comportement : (i) dans le cas où il n'est pas possible d'écrire les octets (si le disque est plein, par exemple) ; (ii) lorsqu'on écrit sur un descripteur de fichiers qui référence un tuyau ou une prise placé dans le mode entrées/sorties non bloquantes, les écritures peuvent être partielles ; enfin, (iii) OCaml qui fait une copie supplémentaire dans un tampon auxiliaire et écrit celui-ci limite la taille du tampon auxiliaire à une valeur maximale (qui est en général la taille utilisée par le système pour ses propres tampons) ceci pour éviter d'allouer de trop gros tampons ; si le le nombre d'octets à écrire est supérieure à cette limite, alors l'écriture sera forcément partielle même si le système aurait assez de ressource pour effectuer une écriture totale.

Pour contourner le problème de la limite des tampons, OCaml fournit également une fonction `write` qui répète plusieurs écritures tant qu'il n'y a pas eu d'erreur d'écriture. Cependant, en cas d'erreur, la fonction retourne l'erreur et ne permet pas de savoir le nombre d'octets effectivement écrits. On utilisera donc plutôt la fonction `single_write` que `write` parce qu'elle préserve l'atomicité (on sait exactement ce qui a été écrit) et est donc plus fidèle à l'appel système d'Unix (voir également l'implémentation de `single_write` décrite dans le chapitre suivant 5.7).

Nous verrons dans le chapitre suivant que lorsqu'on écrit sur un descripteur de fichier qui référence un tuyau ou une prise qui est placé dans le mode entrées/sorties bloquantes et que l'appel est interrompu par un signal, l'appel `single_write` retourne une erreur `EINTR`.

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. (C'est 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. Voir section 2.13 ou page 163 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 = String.create 100 in
let n = read stdin buffer 0 100 in
  String.sub buffer 0 n
```

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);;
```

2.8 Fermeture d'un descripteur

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

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

Une fois 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 `Pervasives`, 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 (plusieurs centaines à quelques milliers). Faire `close` sur un descripteur inutile permet de le désallouer, et donc d'éviter de tomber à court de descripteurs.

2.9 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.

```
1  open Unix;;
2
3  let buffer_size = 8192;;
4  let buffer = String.create buffer_size;;
5
6  let file_copy input_name output_name =
7    let fd_in = openfile input_name [O_RDONLY] 0 in
8    let fd_out = openfile output_name [O_WRONLY; O_CREAT; O_TRUNC] 0o666 in
9    let rec copy_loop () =
10     match read fd_in buffer 0 buffer_size with
11       0 -> ()
12     | r -> ignore (write fd_out buffer 0 r); copy_loop () in
13   copy_loop ();
14   close fd_in;
15   close fd_out;;
16
17 let copy () =
18   if Array.length Sys.argv = 3 then begin
19     file_copy Sys.argv.(1) Sys.argv.(2);
20     exit 0
21   end else begin
22     prerr_endline
23       ("Usage: " ^Sys.argv.(0)^ " <input_file> <output_file>");
24     exit 1
25   end;;
26
27 handle_unix_error copy ();;
```

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 (ligne 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 3 *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.* (Voir le corrigé) □

2.10 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 2.2 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 8 mégaoctets, 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 temps minimum d'environ 4 micro-secondes (sur la machine employée pour faire le test—un Pentium 4 à 2.8 GHz), disons 1 à 10 micro-secondes. Pour des blocs d'entrée/sortie de petite taille, c'est ce temps d'appel système qui prédomine.

Pour des blocs plus gros, entre 4K et 1M, 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. D'autre part la taille du tampon devient supérieur à la tailles des caches utilisés par le système. Et le temps passé par le système à gérer le transfert devient prépondérant sur le coût d'un appel système [2]

Enfin, pour de très gros blocs (8M et plus), la vitesse passe légèrement au-dessous du maximum. Entre en jeu ici le temps nécessaire pour allouer le bloc et lui attribuer des pages de mémoire réelles 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 2 à 20 micro-secondes 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

2. En fait, OCaml limite la tailles des données transférées à 16K (dans la version courante) en répétant plusieurs appels système `write` pour effectuer le transfert complet—voir la discussion la section 5.7. Mais cette limite est au delà de la taille des caches du système et n'est pas observable.

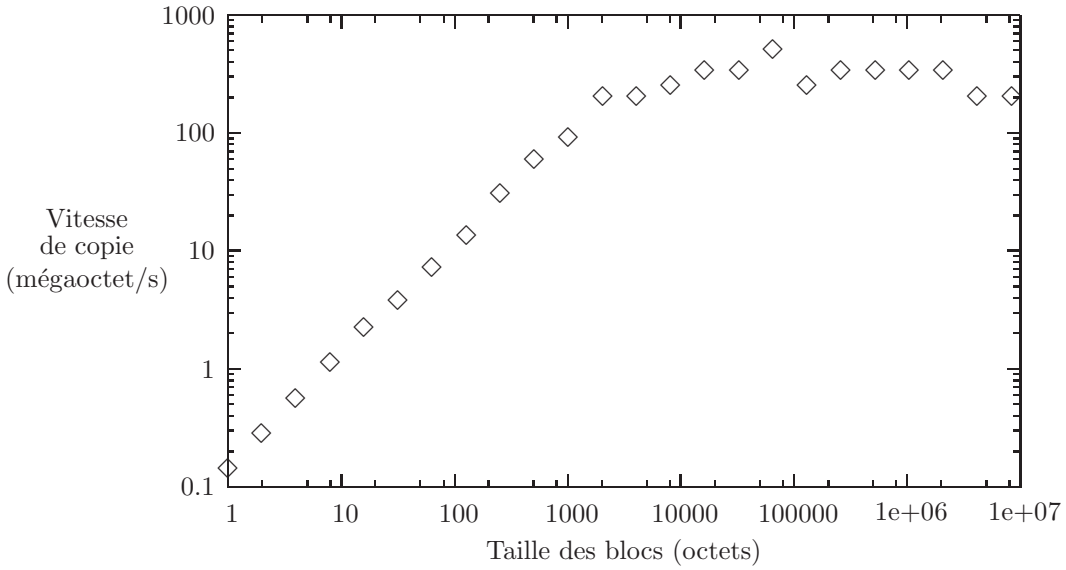


FIGURE 2.2 – Vitesse de copie en fonction de la taille des blocs

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 OCaml, on dispose du module `Pervasives` 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.

2.11 Exemple complet : une petite bibliothèque d'entrées-sorties

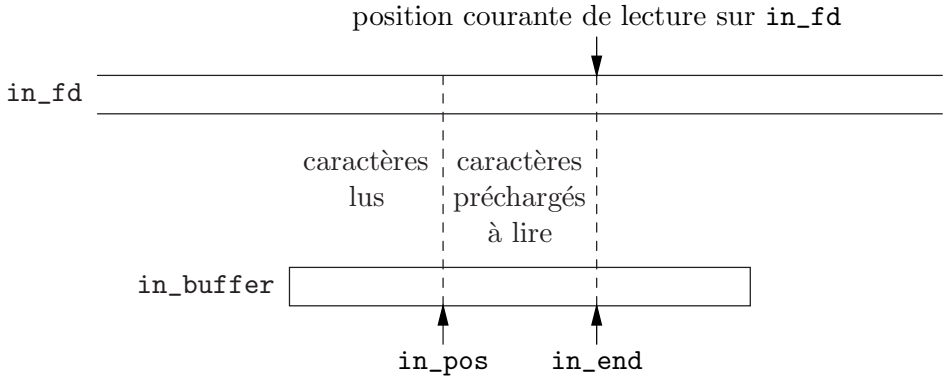
Pour illustrer les techniques de lecture/écriture par tampon, voici une implémentation simple d'un fragment de la bibliothèque `Pervasives` de OCaml. L'interface est la suivante :

```
type in_channel
exception End_of_file
val open_in : string -> in_channel
val input_char : in_channel -> char
val close_in : in_channel -> unit
type out_channel
val open_out : string -> out_channel
val output_char : out_channel -> char -> unit
val close_out : out_channel -> unit
```

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

```
1  open Unix;;
2
3  type in_channel =
4    { in_buffer: string;
5      in_fd: file_descr;
6      mutable in_pos: int;
7      mutable in_end: int };;
8  exception End_of_file
```

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`.

```
9  let buffer_size = 8192;;
10 let open_in filename =
11   { in_buffer = String.create buffer_size;
12     in_fd = openfile filename [O_RDONLY] 0;
13     in_pos = 0;
14     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.

```
15 let input_char chan =
16   if chan.in_pos < chan.in_end then begin
17     let c =  chan.in_buffer.[chan.in_pos] in
18       chan.in_pos <- chan.in_pos + 1;
19       c
20   end else begin
21     match read chan.in_fd chan.in_buffer 0 buffer_size
22     with 0 -> raise End_of_file
23        | r -> chan.in_end <- r;
24               chan.in_pos <- 1;
25               chan.in_buffer.[0]
```

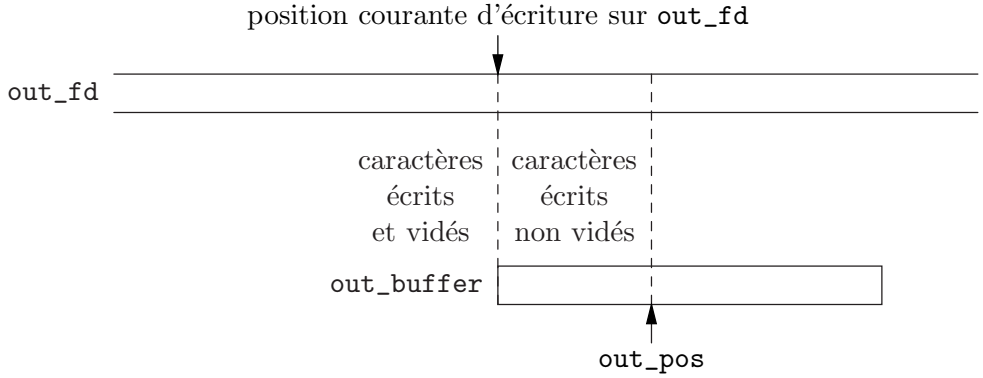
```
26    end;;
```

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` retourne 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.

```
27  let close_in chan =
28    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.




```
29  type out_channel =
30    { out_buffer: string;
31      out_fd: file_descr;
32      mutable out_pos: int };;
33
34  let open_out filename =
35    { out_buffer = String.create 8192;
36      out_fd = openfile filename [O_WRONLY; O_TRUNC; O_CREAT] 0o666;
37      out_pos = 0 };;
38
39  let output_char chan c =
40    if chan.out_pos < String.length chan.out_buffer then begin
41      chan.out_buffer.[chan.out_pos] <- c;
42      chan.out_pos <- chan.out_pos + 1
43    end else begin
44      ignore (write chan.out_fd chan.out_buffer 0 chan.out_pos);
45      chan.out_buffer.[0] <- c;
46      chan.out_pos <- 1
47    end;;
48
49  let close_out chan =
50    ignore (write chan.out_fd chan.out_buffer 0 chan.out_pos);
51    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 4 *Implémenter une fonction*

val *`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.* (Voir le corrigé) □

2.12 Positionnement

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

val `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 (périphériques), 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 5 *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`*).* (Voir le corrigé) □

2.13 Opérations spécifiques à certains types de fichiers

En Unix, la communication passe par des descripteurs de fichiers que ceux-ci soient matérialisés (fichiers, périphériques) ou volatiles (communication entre processus par des tuyaux ou des prises). Cela permet de donner une interface uniforme à la communication de données, indépendante du média. Bien sûr, l'implémentation des opérations dépend quant à elle du média. L'uniformité trouve ses limites dans la nécessité de donner accès à toutes les opérations offertes par le média. Les opérations générales (ouverture, écriture, lecture, *etc.*) restent uniformes sur la plupart des descripteurs mais certaines opérations ne fonctionnent que sur certains types de fichiers. En revanche, pour certains types de fichiers dits spéciaux, qui permettent de traiter la communication avec les périphériques, même les opérations générales peuvent avoir un comportement ad-hoc défini par le type et les paramètres du périphérique.

Fichiers normaux

On peut raccourcir un fichier ordinaire par les appels suivants :

```
val truncate  : string -> int -> unit
val 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.

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 : `openfile`, `stat`, `truncate`, `opendir`. On dispose de deux opérations sur les liens symboliques :

```
val symlink  : string -> string -> unit
val 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.

Fichiers spéciaux

Les fichiers spéciaux peuvent être de type `caractère` ou de type `block`. Les premiers sont des flux de caractères : on ne peut lire ou écrire les caractères que dans l'ordre. Ce sont typiquement

les terminaux, les périphériques sons, imprimantes, etc. Les seconds, typiquement les disques, ont un support rémanent ou temporisé : on peut lire les caractères par blocs, voir à une certaine distance donnée sous forme absolue ou relative par rapport à la position courante. Parmi les fichiers spéciaux, on peut distinguer :

`/dev/null`
: C'est le trou noir qui avale tout ce qu'on met dedans et dont il ne sort rien. Très utile pour ignorer les résultats d'un processus : on redirige sa sortie vers `/dev/null` (voir le chapitre 5).

`/dev/tty*`
: Ce sont les terminaux de contrôle.

`/dev/pty*`
: Ce sont les pseudo-terminaux de contrôle : ils ne sont pas de vrais terminaux mais les simulent (ils répondent à la même interface).

`/dev/hd*`
: Ce sont les disques.

`/proc`
: Sous Linux, permet de lire et d'écrire certains paramètres du système en les organisant comme un système de fichiers.

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`.

En plus des appels systèmes généraux, les fichiers spéciaux qui correspondent à des périphériques doivent pouvoir être paramétrés ou commandés dynamiquement. Exemples 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). Ces opérations sont réalisées en Unix par l'appel système `ioctl` qui regroupe tous les cas particuliers. Cependant, cet appel système n'est pas relevé en OCaml... parce qu'il est mal défini et ne peut pas être traité de façon uniforme.

Terminaux de contrôle

Les terminaux (ou pseudo-terminaux) de contrôle sont un cas particulier de fichiers spéciaux de type caractère pour lequel OCaml donne accès à la configuration. L'appel `tcgetattr` prend en argument un descripteur de fichier ouvert sur le fichier spécial en question et retourne une structure de type `terminal_io` qui décrit le statut du terminal représenté par ce fichier selon la norme POSIX (Voir page 163 pour une description complète).

```
val tcgetattr : file_descr -> terminal_io

type terminal_io =
  { c_ignbrk : bool; c_brk_int : bool; ...;  c_vstop : char }
```

Cette structure peut être modifiée puis passée à la fonction `tcsetattr` pour changer les attributs du périphérique.

```
val tcsetattr : file_descr -> setattr_when -> terminal_io -> unit
```

Le premier argument est le descripteur de fichier désignant le périphérique. Le dernier argument est une structure de type `tcgetattr` décrivant les paramètres du périphérique tels qu'on veut les établir. Le second argument est un drapeau du type énuméré `setattr_when` indiquant le moment à partir duquel la modification doit prendre effet : immédiatement (`TCSANOW`), après

avoir transmis toutes les données écrites (`TCSADRAIN`) ou après avoir lu toutes les données reçues (`TCAFLUSH`). Le choix `TCSADRAIN` est recommandé pour modifier les paramètres d'écriture et `TCSAFLUSH` pour modifier les paramètres de lecture.

Exemple: Pendant la lecture d'un mot de passe, il faut retirer l'écho des caractères tapés par l'utilisateur si le flux d'entrée standard est connecté à un terminal ou pseudo-terminal.

```
1  let read_passwd message =
2    match
3      try
4        let default = tcgetattr stdin in
5        let silent =
6          { default with
7            c_echo = false;
8            c_echoe = false;
9            c_echok = false;
10           c_echonl = false;
11         } in
12       Some (default, silent)
13     with _ -> None
14   with
15   | None -> input_line Pervasives.stdin
16   | Some (default, silent) ->
17       print_string message;
18       flush Pervasives.stdout;
19       tcsetattr stdin TCSANOW silent;
20       try
21         let s = input_line Pervasives.stdin in
22         tcsetattr stdin TCSANOW default; s
23       with x ->
24         tcsetattr stdin TCSANOW default; raise x;;
```

La fonction `read_passwd` commence par récupérer la valeur par défaut des paramètres du terminal associé à `stdin` et construire une version modifiée dans laquelle les caractères n'ont plus d'écho. En cas d'échec, c'est que le flux d'entrée n'est pas un terminal de contrôle, on se contente de lire une ligne. Sinon, on affiche un message, on change le terminal, on lit la réponse et on remet le terminal dans son état normal. Il faut faire attention à bien remettre le terminal dans son état normal également lorsque la lecture a échoué.

Il arrive qu'une application ait besoin d'en lancer une autre en liant son flux d'entrée à un terminal (ou pseudo terminal) de contrôle. Le système OCaml ne fournit pas d'aide pour cela [3] : il faut manuellement rechercher parmi l'ensemble des pseudo-terminaux (en général, ce sont des fichiers de nom de la forme `/dev/tty[a-z][a-f0-9]`) et trouver un de ces fichiers qui ne soit pas déjà ouvert, pour l'ouvrir puis lancer l'application avec ce fichier en flux d'entrée.

Quatre autres fonctions permettent de contrôler le flux (vider les données en attente, attendre la fin de la transmission, relancer la communication).

```
val tcsendbreak : file_descr -> int -> unit
```

La fonction `tcsendbreak` envoie une interruption au périphérique. Son deuxième argument est la durée de l'interruption (`0` étant interprété comme la valeur par défaut pour le périphérique).

```
val tcdrain : file_descr -> unit
```

3. La bibliothèque de Cash [3] fournit de telles fonctions.

La fonction `tcdrain` attend que toutes les données écrites aient été transmises.

```
val tcflush : file_descr -> flush_queue -> unit
```

Selon la valeur du drapeau passé en second argument, la fonction `tcflush` abandonne les données écrites pas encore transmises (`TCIFLUSH`), ou les données reçues mais pas encore lues (`TCOFLUSH`) ou les deux (`TCIOFLUSH`).

```
val tcflow : file_descr -> flow_action -> unit
```

Selon la valeur du drapeau passé en second argument, la fonction `tcflow` suspend l'émission (`TCOOFF`), redémarre l'émission (`TCOON`), envoie un caractère de contrôle STOP ou START pour demander que la transmission soit suspendue (`TCIOFF`) ou relancée (`TCION`).

```
val setsid : unit -> int
```

La fonction `setsid` place le processus dans une nouvelle session et le détache de son terminal de contrôle.

2.14 Verrous sur des fichiers

Deux processus peuvent modifier un même fichier en parallèle au risque que certaines écritures en écrasent d'autres. Dans certains cas, l'ouverture en mode `O_APPEND` permet de s'en sortir, par exemple, pour un fichier de `log` où on se contente d'écrire des informations toujours à la fin du fichier. Mais ce mécanisme ne résout pas le cas plus général où les écritures sont à des positions a priori arbitraires, par exemple, lorsqu'un fichier représente une base de données . Il faut alors que les différents processus utilisant ce fichier collaborent ensemble pour ne pas se marcher sur les pieds. Un verrouillage de tout le fichier est toujours possible en créant un fichier verrou auxiliaire (voir page 22). L'appel système `lockf` permet une synchronisation plus fine qui en ne verrouillant qu'une partie du fichier.

2.15 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 récupérer la fonction `file_copy` de l'exemple du même nom pour copier les fichiers normaux (page 25).

```
1  open Unix
   ...
5  let file_copy input_name output_name =
   ...
```

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.

```
16 let set_infos filename infos =
17   utimes filename infos.st_atime infos.st_mtime;
18   chmod filename infos.st_perm;
19   try
20     chown filename infos.st_uid infos.st_gid
21   with Unix_error(EPERM,_,_) ->
22     ()
```

L'appel système `utime` modifie les dates d'accès et de modification. On utilise `chmod` et `chown` pour rétablir les droits d'accès et le propriétaire. 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.

Voici la fonction récursive principale.

```
23 let rec copy_rec source dest =
24   let infos = lstat source in
25   match infos.st_kind with
26     S_REG ->
27       file_copy source dest;
28       set_infos dest infos
29   | S_LNK ->
30       let link = readlink source in
31       symlink link dest
32   | S_DIR ->
33       mkdir dest 0o200;
34       Misc.iter_dir
35         (fun file ->
36           if file <> Filename.current_dir_name
37               && file <> Filename.parent_dir_name
38           then
39             copy_rec
40               (Filename.concat source file)
41               (Filename.concat dest file))
42         source;
43       set_infos dest infos
44   | _ ->
45       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 du répertoire vers lui-même `Filename.current_dir_name` et vers son parent `Filename.parent_dir_name`, 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 :

```
46 let copyrec () =
47   if Array.length Sys.argv <> 3 then begin
48     prerr_endline ("Usage: " ^Sys.argv.(0)^ " <source> <destination>");
49     exit 2
50   end else begin
51     copy_rec Sys.argv.(1) Sys.argv.(2);
52     exit 0
53   end
54 ;;
55 handle_unix_error copyrec ();;
```

Exercice 6 *Copier intelligemment les liens durs. Tel que présenté ci-dessus,* `copyrec` *duplique*

Offset	Length[1]	Codage[2]	Nom	Description
0	100	chaîne	`name`	Nom du fichier
100	8	octal	`perm`	Mode du fichier
108	8	octal	`uid`	ID de l'utilisateur
116	8	octal	`gid`	ID du groupe de l'utilisateur
124	12	octal	`size`	Taille du fichier[3]
136	12	octal	`mtime`	Date de la dernière modification
148	8	octal	`checksum`	Checksum de l'entête
156	1	caractère	`kind`	Type de fichier
157	100	octal	*`link`*	Lien
257	8	chaîne	`magic`	Signature (`"ustar\032\032\0"`)
265	32	chaîne	`user`	Nom de l'utilisateur
297	32	chaîne	`group`	Nom du groupe de l'utilisateur
329	8	octal	*`major`*	Identificateur majeur du périphérique
337	8	octal	*`minor`*	Identificateur mineur du périphérique
345	167			Padding

[1] en octets.

[2] tous les champs sont codés sur des chaînes de caractères et terminés par le caractère nul `'\000'`, sauf les champs "kind" (*Type de fichier*) et le champ "size" (*Taille du fichier*) (`'\000'` optionnel).

TABLE 2.2 – Représentation de l'entête

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. (Voir le corrigé) □

2.16 Exemple : Tape ARchive

Le format `tar` (pour `t`*ape* `ar`*chive*) permet de représenter un ensemble de fichiers en un seul fichier. (Entre autre il permet de stocker toute une hiérarchie de fichiers sur une bande.) C'est donc d'une certaine façon un mini système de fichiers.

Dans cette section nous décrivons un ensemble de fonctions qui permettent de lire et d'écrire des archives au format `tar`. La première partie, décrite complètement, consiste à écrire une commande `readtar` telle que `readtar` *a* affiche la liste des fichiers contenus dans l'archive *a* et `readtar` *a f* affiche le contenu du fichier *f* contenu dans l'archive *a*. Nous proposons en exercice l'extraction de tous les fichiers contenus dans une archive, ainsi que la fabrication d'une archive à partir d'un ensemble de fichiers.

Description du format Une archive `tar` est une suite d'enregistrements, chaque enregistrement représentant un fichier. Un enregistrement est composé d'un entête qui code les informations sur le fichier (son nom, son type, sa taille, son propriétaire *etc.*) et du contenu du fichier. L'entête est représenté sur un bloc (512 octets) comme indiqué dans le tableau 2.2. Le contenu est représenté à la suite de l'entête sur un nombre entier de blocs. Les enregistrements sont représentés les uns à la suite des autres. Le fichier est éventuellement complété par des blocs vides pour atteindre au moins 20 blocs.

Comme les archives sont aussi conçues pour être écrites sur des supports fragiles et relues plusieurs années après, l'entête comporte un champ `checksum` qui permet de détecter les archives

dont l'entête est endommagé (ou d'utiliser comme une archive un fichier qui n'en serait pas une.) Sa valeur est la somme des codes des caractères de l'entête (pendant ce calcul, on prend comme hypothèse que le le champ `checksum`, qui n'est pas encore connu est composé de blancs et terminé par le caractère nul).

Le champ `kind` représente le type des fichiers sur un octet. Les valeurs significatives sont les caractères indiqués dans le tableau ci-dessous [4] :

'\0' ou '0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'
REG	LINK	LNK	CHR	BLK	DIR	FIFO	CONT

La plupart des cas correspondent au type `st_link` des types de fichier Unix. Le cas `LINK` représente des liens durs : ceux-ci ont le même nœud (*inode*) mais accessible par deux chemins différents ; dans ce cas, le lien doit obligatoirement mener à un autre fichier déjà défini dans l'archive. Le cas `CONT` représente un fichier ordinaire, mais qui est représenté par une zone mémoire contigüe (c'est une particularité de certains systèmes de fichiers, on pourra donc le traiter comme un fichier ordinaire). Le champ *`link`* représente le lien lorsque `kind` vaut `LNK` ou `LINK`. Les champs *`major`* et *`minor`* représentent les numéros majeur et mineur du périphérique dans le cas où le champ `kind` vaut `CHR` ou `BLK`. Ces trois champs sont inutilisés dans les autres cas.

La valeur du champ `kind` est naturellement représentée par un type concret et l'entête par un enregistrement :

```
1  open Sys
2  open Unix
3
4  type kind =
5    | REG
6    | LNK of string
7    | LINK of string
8    | CHR of int * int
9    | BLK of int * int
10   | DIR
11   | FIFO
12   | CONT
```

```
13 type header = {
14     name : string;
15     perm : int;
16     uid : int;
17     gid : int;
18     size : int;
19     mtime : int;
20     kind : kind;
21     user : string;
22     group : string
23  }
```

Lecture d'un entête La lecture d'un entête n'est pas très intéressante, mais elle est incontournable.

```
24 exception Error of string * string
25 let error err mes = raise (Error (err, mes));;
26 let handle_error f s =
27   try f s with
28   | Error (err, mes) ->
29       Printf.eprintf "Error: %s: %s" err mes;
30       exit 2
31
32 let substring s offset len =
33   let max_length = min (offset + len + 1) (String.length s) in
```

4. Ce champ peut également prendre d'autres valeurs pour coder des cas pathologiques, par exemple lorsque la valeur d'un champ déborde la place réservée dans l'entête, ou dans des extensions de la commande `tar`.

```
34    let rec real_length j =
35      if j < max_length && s.[j] <> '\000' then real_length (succ j)
36      else j - offset in
37    String.sub s offset (real_length offset);;
38
39  let integer_of_octal nbytes s offset =
40    let i = int_of_string ("0o" ^ substring s offset nbytes) in
41    if i < 0 then error "Corrupted archive" "integer too large" else i;;
42
43  let kind s i =
44    match s.[i] with
45      '\000' | '0' -> REG
46    | '1' -> LINK (substring s (succ i) 99)
47    | '2' -> LNK (substring s (succ i) 99)
48    | '3' -> CHR (integer_of_octal 8 s 329, integer_of_octal 8 s 329)
49    | '4' -> BLK (integer_of_octal 8 s 329, integer_of_octal 8 s 337)
50    | '5' -> DIR | '6' -> FIFO | '7' -> CONT
51    | _ -> error "Corrupted archive" "kind"
52
53  let header_of_string s =
54    { name = substring s 0 99;
55      perm = integer_of_octal 8 s 100;
56      uid = integer_of_octal 8 s 108;
57      gid = integer_of_octal 8 s 116;
58      size = integer_of_octal 12 s 124;
59      mtime = integer_of_octal 12 s 136;
60      kind = kind s 156;
61      user = substring s 265 32;
62      group = substring s 297 32;
63    }
64
65  let block_size = 512;;
66  let total_size size =
67    block_size + ((block_size -1 + size) / block_size) * block_size;;
```

La fin de l'archive s'arrête soit sur une fin de fichier là ou devrait commencer un nouvel enregistrement, soit sur un bloc complet mais vide. Pour lire l'entête, nous devons donc essayer de lire un bloc, qui doit être vide ou complet. Nous réutilisons la fonction `really_read` définie plus haut pour lire un bloc complet. La fin de fichier ne doit pas être rencontrée en dehors de la lecture de l'entête.

```
73  let buffer_size = block_size;;
74  let buffer = String.create buffer_size;;
75
76  let end_of_file_error() =
77    error "Corrupted archive" "unexpected end of file"
78  let without_end_of_file f x =
79    try f x with End_of_file -> end_of_file_error()
80
81  let read_header fd =
82    let len = read fd buffer 0 buffer_size in
```

```
83    if len = 0 ||  buffer.[0] = '\000' then None
84    else begin
85      if len < buffer_size then
86        without_end_of_file (really_read fd buffer len) (buffer_size - len);
87      Some (header_of_string buffer)
88    end;;
```

Lecture d'une archive Pour effectuer une quelconque opération dans une archive, il est nécessaire de lire l'ensemble des enregistrements dans l'ordre au moins jusqu'à trouver celui qui correspond à l'opération à effectuer. Par défaut, il suffit de lire l'entête de chaque enregistrement, sans avoir à en lire le contenu. Souvent, il suffira de lire le contenu de l'enregistrement recherché ou de lire le contenu après coup d'un enregistrement le précédent. Pour cela, il faut garder pour chaque enregistrement une information sur sa position dans l'archive, en plus de son entête. Nous utilisons le type suivant pour les enregistrements :

```
type record = { header : header; offset : int; descr : file_descr };;
```

Nous allons maintenant écrire un itérateur général qui lit les enregistrements (sans leur contenu) et les accumulent. Toutefois, pour être général, nous restons abstrait par rapport à la fonction d'accumulation f (qui peut aussi bien ajouter les enregistrements à ceux déjà lus, les imprimer, les jeter, *etc.*)

```
89  let fold f initial fd  =
90    let rec fold_aux offset accu =
91      ignore (without_end_of_file (lseek fd offset) SEEK_SET);
92      match without_end_of_file read_header fd with
93        Some h ->
94          let r =
95            { header = h; offset = offset + block_size; descr = fd } in
96          fold_aux (offset + total_size h.size) (f r accu)
97      | None -> accu in
98    fold_aux 0 initial;;
```

Une étape de `fold_aux` commence à une position `offset` avec un résultat partiel `accu`. Elle consiste à se placer à la position `offset`, qui doit être le début d'un enregistrement, lire l'entête, construire l'enregistrement `r` puis recommencer à la fin de l'enregistrement avec le nouveau résultat `f r accu` (moins partiel). On s'arrête lorsque l'entête est vide, ce qui signifie qu'on est arrivé à la fin de l'archive.

Affichage de la liste des enregistrements d'une archive Il suffit simplement d'afficher l'ensemble des enregistrements, au fur et à mesure, sans avoir à les conserver :

```
99  let list tarfile =
100   let fd = openfile tarfile [ O_RDONLY ] 0o0 in
101   let add r () = print_string r.header.name; print_newline() in
102   fold add () fd;
103   close fd
```

Affichage du contenu d'un fichier dans une archive La commande `readtar` a f doit rechercher le fichier de nom f dans l'archive et l'afficher si c'est un fichier régulier. De plus un chemin f de l'archive qui est un lien dur et désigne un chemin g de l'archive est suivi et le contenu de g est affiché : en effet, bien que f et g soient représentés différemment dans l'archive finale (l'un est un lien dur vers l'autre) ils désignaient exactement le même fichier à sa création.

Le fait que g soit un lien vers f ou l'inverse dépend uniquement de l'ordre dans lequel les fichiers ont été parcourus à la création de l'archive. Pour l'instant nous ne suivons pas les liens symboliques.

L'essentiel de la résolution des liens durs est effectué par les deux fonctions suivantes, définies récursivement.

```
104 let rec find_regular r list =
105   match r.header.kind with
106   | REG | CONT -> r
107   | LINK name -> find_file name list
108   | _ -> error r.header.name "Not a regular file"
109 and find_file name list =
110   match list with
111     r :: rest ->
112       if r.header.name = name then find_regular r rest
113       else find_file name rest
114   | [] -> error name "Link not found (corrupted archive)";;
```

La fonction `find_regular` trouve le fichier régulier correspondant à un enregistrement (son premier) argument. Si celui-ci est un fichier régulier, c'est gagné. Si c'est un fichier spécial (ou un lien symbolique), c'est perdu. Il reste le cas d'un lien dur : la fonction recherche ce lien dans la liste des enregistrements de l'archive (deuxième argument) en appelant la fonction `find_file`.

Un fois l'enregistrement trouvé il n'y a plus qu'à afficher son contenu. Cette opération ressemble fortement à la fonction `file_copy`, une fois le descripteur positionné au début du fichier dans l'archive.

```
115 let copy_file file output =
116   ignore (lseek file.descr file.offset SEEK_SET);
117   let rec copy_loop len =
118     if len > 0 then
119       match read file.descr buffer 0 (min buffer_size len) with
120         0 -> end_of_file_error()
121       | r -> ignore (write output buffer 0 r); copy_loop (len-r) in
122   copy_loop file.header.size
```

Il ne reste plus qu'à combiner les trois précédents.

```
123 exception Done
124 let find_and_copy tarfile filename =
125   let fd = openfile tarfile [ O_RDONLY ] 0o0 in
126   let found_or_collect r accu =
127     if r.header.name = filename then begin
128       copy_file (find_regular r accu) stdout;
129       raise Done
130     end else r :: accu in
131   try
132      ignore (fold found_or_collect [] fd);
133      error "File not found" filename
134   with
135   | Done -> close fd
```

On lit les enregistrements de l'archive (sans lire leur contenu) jusqu'à rencontrer un enregistrement du nom recherché. On appelle la fonction `find_regular` pour rechercher dans la liste des enregistrements lus celui qui contient vraiment le fichier. Cette seconde recherche, en arrière,

doit toujours réussir si l'archive est bien formée. Par contre la première recherche, en avant, va échouer si le fichier n'est pas dans l'archive. Nous avons pris soin de distinguer les erreurs dues à une archive corrompue ou à une recherche infructueuse.

Et voici la fonction principale qui réalise la commande `readtar` :

```
136 let readtar () =
137   let nargs = Array.length Sys.argv in
138   if nargs = 2 then list Sys.argv.(1)
139   else if nargs = 3 then find_and_copy Sys.argv.(1) Sys.argv.(2)
140   else
141     prerr_endline ("Usage: " ^Sys.argv.(0)^ " <tarfile> [ <source> ]");;
142
143 handle_unix_error (handle_error readtar) ();;
```

Exercice 7 *Étendre la commande* `readtar` *pour qu'elle suive les liens symboliques, c'est-à-dire pour qu'elle affiche le contenu du fichier si l'archive avait été au préalable extraite et si ce fichier correspond à un fichier de l'archive.*

Derrière l'apparence triviale de cette généralisation se cachent quelques difficultés, car les liens symboliques sont des chemins quelconques qui peuvent ne pas correspondre exactement à des chemins de l'archive ou carrément pointer en dehors de l'archive (ils peuvent contenir `..`*). De plus, les liens symboliques peuvent désigner des répertoires (ce qui est interdit pour les liens durs).* (Voir le corrigé) □

Exercice 8 *Écrire une commande* `untar` *telle que* `untar` *a extrait et crée tous les fichiers de l'archive a (sauf les fichiers spéciaux) en rétablissant si possible les informations sur les fichiers (propriétaires, droits d'accès) indiqués dans l'archive.*

L'arborescence de l'archive ne doit contenir que des chemins relatifs, sans jamais pointer vers un répertoire parent (donc sans pourvoir pointer en dehors de l'archive), mais on devra le détecter et refuser de créer des fichiers ailleurs que dans un sous-répertoire du répertoire courant. L'arborescence est reconstruite à la position où l'on se trouve à l'appel de la commande `untar`*. Les répertoires non mentionnés explicitement dans l'archive qui n'existent pas sont créés avec les droits par défaut de l'utilisateur qui lance la commande.* (Voir le corrigé) □

Exercice 9 (Projet) *Écrire une commande* `tar` *telle que* `tar -xvf` *a* f_1 f_2 ... *construise l'archive a à partir de la liste des fichiers* f_1, f_2, etc. *et de leurs sous-répertoires.*

(Voir le corrigé) □

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.

```
val 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()`.

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` : l'un peut être fermé et l'autre reste ouvert. Par contre, les deux descripteurs désignent la même

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

entrée dans la table des fichiers (qui est allouée dans la mémoire système) et partagent donc leur position courante : si l'un puis l'autre lit, chacun lira une partie différente du fichier ; de même les déplacement effectués par l'un avec `lseek` sont immédiatement visibles par l'autre. (Les descripteurs du fils et du père se comportent donc comme les deux descripteurs argument et résultat après exécution de la commande `dup`, mais sont dans des processus différents au lieu d'être dans le même 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 leave () =
5    let hh = int_of_string (String.sub Sys.argv.(1) 0 2)
6    and mm = int_of_string (String.sub Sys.argv.(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_endline "Hey! That time has already passed!";
11     exit 0
12   end;
13   if fork() <> 0 then exit 0;
14   sleep delay;
15   print_endline "\007\007\007Time to leave!";
16   exit 0;;
17
18 handle_unix_error leave ();;
```

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.

```
val wait : unit -> int * process_status
val waitpid : wait_flag list -> int -> int * process_status
```

L'appel système primitif est `waitpid` et la fonction `wait()` n'est qu'un racourci pour l'expression `waitpid [] (-1)`. L'appel système `waitpid` [] p attend la terminaison du processus p, si $p > 0$

est strictement positif, ou d'un sous-ensemble de processus fils, du même groupe si $p = 0$, quelconque si $p = -1$, ou du groupe $-p$ si $p < -1$.

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)	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
`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`.

La commande `waitpid` accepte deux options facultatifs comme premier argument : L'option `WNOHANG` indique de ne pas attendre, s'il il a des fils qui répondent à la demande mais qui n'ont pas encore terminé. Dans ce cas, le premier résultat est 0 et le second non défini. L'option `WUNTRACED` indique de retourner également les fils qui ont été arrêté par le signal `sigstop`. La commande lève l'erreur `ECHILD` si aucun fils du processus appelant ne répond à la spécification p (en particulier, si p vaut -1 et que le processus courrant n'a pas ou plus de fils).

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.

```
1  open Unix
2  exception Found;;
3
4  let simple_search cond v =
5    try
6      for i = 0 to Array.length v - 1 do
7        if cond v.(i) then raise Found
8      done;
9      false
10   with Found -> true;;
11
12 let fork_search cond v =
13   let n = Array.length v in
14   match fork() with
15     0 ->
16       let found = simple_search cond (Array.sub v (n/2) (n-n/2)) in
17       exit (if found then 0 else 1)
18   | _ ->
19       let found = simple_search cond (Array.sub v 0 (n/2)) in
20       match wait() with
21         (pid, WEXITED retcode) -> found or (retcode = 0)
22       | (pid, _)               -> failwith "fork_search";;
```

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 16 et 17). Le

processus père parcourt la moitié basse du tableau, puis appelle `wait` pour se synchroniser avec le processus fils (lignes 19 et 20). 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 système `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.

```
val execve : string -> string array -> string array -> unit
val execv  : string -> string array -> unit
val execvp : string -> string array -> 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.argv` 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 :

```
1 open Sys;;
2 open Unix;;
3 let grep () =
4   execvp "grep"
5     (Array.concat
6        [ [|"grep"; "-i"|];
7          (Array.sub Sys.argv 1 (Array.length Sys.argv - 1)) ])
8 ;;
9 handle_unix_error grep ();;
```

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

```
1 open Sys;;
2 open Unix;;
3 let emacs () =
4   execve "/usr/bin/emacs" Sys.argv
5     (Array.concat [ [|"TERM=hacked-xterm"|]; (environment()) ]);;
6 handle_unix_error emacs ();;
```

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.

```
1  open Unix;;
2
3  let split_words s =
4    let rec skip_blanks i =
5      if i < String.length s & s.[i] = ' '
6      then skip_blanks (i+1)
7      else i in
8    let rec split start i =
9      if i >= String.length s then
10       [String.sub s start (i-start)]
11     else if s.[i] = ' ' then
```

```
12        let j = skip_blanks i in
13        String.sub s start (i-start) :: split j j
14      else
15        split start (i+1) in
16    Array.of_list (split 0 0);;
```

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

```
17  let exec_command cmd =
18    try execvp cmd.(0) cmd
19    with Unix_error(err, _, _) ->
20      Printf.printf "Cannot execute %s : %s\n%!"
21        cmd.(0) (error_message err);
22      exit 255
23
24  let print_status program status =
25    match status with
26      WEXITED 255 -> ()
27    | WEXITED status ->
28        Printf.printf "%s exited with code %d\n%!" program status;
29    | WSIGNALED signal ->
30        Printf.printf "%s killed by signal %d\n%!" program signal;
31    | WSTOPPED signal ->
32        Printf.printf "%s stopped (???)\n%!" program;;
```

La fonction `exec_command` exécute une commande avec récupération des erreurs. Le code de retour 255 indique que la commande n'a pas pu être exécutée. (Ce n'est pas une convention standard ; on espère que peu de commandes renvoient le code de retour 255.) La fonction `print_status` décode et imprime l'information d'état retournée par un processus, en ignorant le code de retour 255.

```
33  let minishell () =
34    try
35      while true do
36        let cmd = input_line Pervasives.stdin in
37        let words = split_words cmd in
38        match fork() with
39          0 -> exec_command words
40        | pid_son ->
41            let pid, status = wait() in
42            print_status "Program" status
43      done
44    with End_of_file ->
45      ();;
46
47  handle_unix_error minishell ();;
```

À chaque tour de boucle, on lit une ligne sur l'entrée standard, via la fonction `input_line` de la bibliothèque standard `Pervasives`. (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_command` pour lancer la commande avec récupération des erreurs. Le processus père appelle `wait` pour attendre que la commande termine et imprime l'information d'état renvoyée par `wait`.

Exercice 10 *Ajouter la possibilité d'exécuter des commandes en tâche de fond, si elles sont suivies par* `&`. (Voir le corrigé) □

Chapitre 4

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 où les programmes doivent explicitement demander à recevoir les messages externes en attente, par exemple en faisant `read` sur un tuyau.

Les signaux transportent peu d'information (le type du signal et rien d'autre) et n'ont pas été conçus pour communiquer entre processus mais pour permettre à un processus de recevoir des informations atomiques sur l'évolution de l'environnement extérieur (l'état du système ou d'autres processus).

4.1 Le comportement par défaut

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 (qui n'ont pas été mis en arrière plan) quand l'utilisateur tape le caractère d'interruption `ctrl-C`. De même, il envoie `sigquit` quand l'utilisateur tape `ctrl-\`[1]. 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` envoie le signal `sigkill` au processus 194, ce qui a pour effet de terminer à coup sûr ce processus.
- D'un autre programme (via le système) qui exécute un appel système `kill` (le cas précédent en étant un cas particulier).
- 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`.

4.2 Produire des signaux

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

```
val kill : int -> 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 émetteur. Un processus peut s'envoyer des signaux à lui-même. Lorsque l'appel système `kill` retourne, il est garanti que le signal a été délivré au processus destinataire. C'est-à-dire que si le processus destinataire n'ignore pas et ne masque pas le signal, sa première action sera de traiter un signal (celui-ci ou un autre). Si un processus reçoit plusieurs fois le même signal pendant un laps de temps très court, il peut n'exécuter qu'un seule fois (le code associé à) ce signal. Un programme ne peut donc pas compter le nombre de fois qu'il reçoit un signal, mais seulement le nombre de fois qu'il le traite.

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

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

L'appel `alarm` s retourne immédiatement, mais fait envoyer au processus le signal `sigalrm` (au moins) s secondes plus tard (le temps maximal d'attente n'est pas garanti). L'appel renvoie

1. Ces caractères sont choisis par défaut, mais il est possible de les changer en modifiant les paramètres du terminal, voir section 2.13.

le nombre de seconde restante jusqu'à la programmation précédente. Si s est nulle, l'effet est simplement d'annuler la précédente programmation de l'alarme.

4.3 Changer l'effet d'un signal

L'appel système `signal` permet de changer le comportement du processus lorsqu'il reçoit un signal d'un certain type.

```
val signal : int -> signal_behavior -> signal_behavior
```

Le deuxième argument indique le comportement désiré. Si le deuxième argument est la constante `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 moment où l'on se déconnecte. Il existe une commande `nohup` qui fait précisément cela :

$$\texttt{nohup}\ cmd\ arg_1\ \ldots\ arg_n$$

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 :

```
1 open Sys;;
2 signal sighup Signal_ignore;;
3 Unix.execvp argv.(1) (Array.sub argv 1 (Array.length argv - 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 `tar` peut essayer de sauver une information importante dans le fichier ou détruire le fichier corrompu 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 l'information importante dans le fichier *);
  exit 100;;
```

Exemple: Pour récupérer les interruptions de l'utilisateur. Certains programmes interactifs 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 stdout '\007'; flush stdout; alarm 30; ();;
...
signal sigalrm (Signal_handle beep); alarm 30;;
```

Points de contrôle

Les signaux transmettent au programme une information de façon asynchrone. C'est leur raison d'être, et ce qui les rend souvent incontournables, mais c'est aussi ce qui en fait l'une des grandes difficultés de la programmation système.

En effet, le code de traitement s'exécute à la réception d'un signal, qui est asynchrone, donc de façon pseudo concurrente (c'est-à-dire entrelacée) avec le code principal du programme. Comme le traitement d'un signal ne retourne pas de valeur, leur intérêt est de faire des effets de bords, typiquement modifier l'état d'une variable globale. Il s'ensuit une compétition (race condition) entre le signal et le programme principal pour l'accès à cette variable globale. La solution consiste en général à bloquer les signaux pendant le traitement de ces zones critiques comme expliqué ci-dessous.

Toutefois, OCaml ne traite pas les signaux de façon tout à faire asynchrone. À la réception du signal, il se contente d'enregistrer sa réception et le traitement ne sera effectué, c'est-à-dire le code de traitement associé au signal effectivement exécuté, seulement à certains points de contrôles. Ceux-ci sont suffisamment fréquents pour donner l'illusion d'un traitement asynchrone. Les points de contrôles sont typiquement les points d'allocation, de contrôle de boucles ou d'interaction avec le système (en particulier autour des appels systèmes). OCaml garantit qu'un code qui ne boucle pas, n'alloue pas, et n'interagit pas avec le système ne sera pas entrelacer avec le traitement d'un signal. En particulier l'écriture d'une valeur non allouée (entier, booléen, etc. mais pas un flottant !) dans une référence ne pose pas de problème de compétition.

4.4 Masquer des signaux

Les signaux peuvent être bloqués. Les signaux bloqués ne sont pas ignorés, mais simplement mis en attente, en général pour être délivrés ultérieurement. L'appel système `sigprocmask` permet de changer le masque des signaux bloqués :

```
val sigprocmask : sigprocmask_command -> int list -> int list
```

`sigprocmask` *cmd sigs* change l'ensemble des signaux bloqués et retourne la liste des signaux qui étaient bloqués juste avant l'exécution de la commande, ce qui permettra ultérieurement de

remettre le masque des signaux bloqués dans son état initial. L'argument *sigs* est une liste de signaux dont le sens dépend de la commande *cmd* :

`SIG_BLOCK`	les signaux *sigs* sont ajoutés aux signaux bloqués.
`SIG_UNBLOCK`	les signaux *sigs* sont retirés des signaux débloqués.
`SIG_SETMASK`	les signaux *sigs* sont exactement les signaux bloqués.

Un usage typique de `sigprocmask` est de masquer temporairement certains signaux.

```
let old_mask = sigprocmask cmd sigs in
(* do something *)
let _ = sigprocmask SIG_SETMASK old_mark
```

Bien souvent, on devra se protéger contre les erreurs éventuelles en utilisant plutôt le schéma :

```
let old_mask = sigprocmask cmd sigs in
let treat() = ((* do something *)) in
let reset() = ignore (sigprocmask SIG_SETMASK old_mask) in
Misc.try_finalize treat () reset ()
```

4.5 Signaux et appels-système

Attention ! un signal non ignoré peut interrompre certains appels système. En général, les appels systèmes interruptibles sont seulement des appels systèmes dits *lents*, qui peuvent a priori prendre un temps arbitrairement long : par exemple, lectures/écritures au terminal, `select` (voir plus loin), `system`, *etc.* En cas d'interruption, l'appel système n'est pas exécuté et déclenche l'exception `EINTR`. Noter que l'écriture/lecture dans un fichier ne sont pas interruptibles : bien que ces opérations puissent suspendre le processus courant pour donner la main à un autre le temps que les données soient lues sur le disques, lorsque c'est nécessaire, cette attente sera toujours brève—si le disque fonctionne correctement. En particulier, l'arrivée des données ne dépend que du système et pas d'un autre processus utilisateur.

Les signaux ignorés ne sont jamais délivrés. Un signal n'est pas délivré tant qu'il est masqué. Dans les autres cas, il faut se prémunir contre une interruption possible.

Un exemple typique est l'attente de la terminaison d'un fils. Dans ce cas, le père exécute `waitpid [] pid` où `pid` est le numéro du fils à attendre. Il s'agit d'un appel système bloquant, donc «lent», qui sera interrompu par l'arrivée éventuelle d'un signal. En particulier, le signal `sigchld` est envoyé au processus père à la mort d'un fils.

Le module `Misc` contient la fonction suivante `restart_on_EINTR` de type `('a -> 'b) -> 'a -> 'b` qui permet de lancer un appel système et de le répéter lorsqu'il est interrompu par un signal, *i.e.* lorsqu'il lève l'exception `EINTR`.

```
1 let rec restart_on_EINTR f x =
2   try f x with Unix_error (EINTR, _, _) -> restart_on_EINTR f x
```

Pour attendre réellement un fils, on pourra alors simplement écrire `restart_on_EINTR (waitpid flags) pid`.

Exemple: Le père peut aussi récupérer ses fils de façon asynchrone, en particulier lorsque la valeur de retour n'importe pas pour la suite de l'exécution. Cela peut se faire en exécutant une fonction `free_children` à la réception du signal `sigchld`. Nous plaçons cette fonction d'usage général dans la bibliothèque `Misc`.

```
let free_children signal =
  try while fst (waitpid [ WNOHANG ] (-1)) > 0 do () done
  with Unix_error (ECHILD, _, _) -> ()
```

Cette fonction appelle la fonction `waitpid` en mode non bloquant (option `WNOHANG`) et sur n'importe quel fils, et répète l'appel quand qu'un fils a pu être retourné. Elle s'arrête lorsque il ne reste plus que des fils vivants (zéro est retourné à la place de l'identité du processus délivré) ou lorsqu'il n'y a plus de fils (exception `ECHILD`). Lorsque le processus reçoit le signal `sigchld` il est en effet impossible de savoir le nombre de processus ayant terminé, si le signal est émis plusieurs fois dans un intervalle de temps suffisamment court, le père ne verra qu'un seul signal. Noter qu'ici il n'est pas nécessaire de se prémunir contre le signal `EINTR` car `waitpid` n'est pas bloquant lorsqu'il est appelé avec l'option `WNOHANG`.

Dans d'autres cas, ce signal ne peut être ignoré (l'action associée sera alors de libérer tous les fils ayant terminé, de façon non bloquante—on ne sait jamais combien de fois le signal à été émis).

Exemple: La commande `system` du module `Unix` est simplement définie par

```
let system cmd =
  match fork() with
    0 ->
      begin try
        execv "/bin/sh" [| "/bin/sh"; "-c"; cmd |]; assert false
      with _ -> exit 127
      end
  | id -> snd(waitpid [] id);;
```

L'assertion qui suit l'appel système `execv` est là pour corriger une restriction erronée du type de retour de la fonction `execv` (dans les version antérieures ou égale à 3.07). L'appel système ne retournant pas, aucune contrainte ne doit porter sur la valeur retournée, et bien sûr l'assertion ne sera jamais exécutée.

La commande `system` de la bibliothèque standard de la bibliothèque C précise que le père ignore les signaux `sigint` et `sigquit` et masque le signal `sigchld` pendant l'exécution de la commande. Cela permet d'interrompre ou de tuer le programme appelé (qui reçoit le signal) sans que le programme principal ne soit affecté pendant l'exécution de la commande.

Nous préférons définir la fonction `system` comme spécialisation d'une fonction plus générale `exec_as_system` qui n'oblige pas à faire exécuter la commande par le shell. Nous la plaçons dans le module `Misc`.

```
1  let exec_as_system exec args =
2    let old_mask = sigprocmask SIG_BLOCK [sigchld ] in
3    let old_int = signal sigint Signal_ignore in
4    let old_quit = signal sigquit Signal_ignore in
5    let reset() =
6      ignore (signal sigint old_int);
7      ignore (signal sigquit old_quit);
8      ignore (sigprocmask SIG_SETMASK old_mask) in
9    let system_call () =
10     match fork() with
11     | 0 ->
12         reset();
13         begin try
14           exec args
15         with _ -> exit 127
16         end
17     | k ->
```

```
18          snd (restart_on_EINTR (waitpid []) k) in
19    try_finalize system_call() reset();;
20
21  let system cmd =
22    exec_as_system (execv "/bin/sh") [| "/bin/sh"; "-c"; cmd |];;
```

Noter que le changement des signaux doit être effectué avant l'appel à `fork`. En effet, immédiatement après cet appel, seulement l'un des deux processus fils ou père a la main (en général le fils). Pendant le laps de temps où le fils prend la main, le père pourrait recevoir des signaux, notamment `sigchld` si le fils termine immédiatement. En conséquence, il faut remettre les signaux à leur valeur initiale dans le fils avant d'exécuter la commande (ligne 13). En effet, l'ensemble des signaux ignorés est préservé par `fork` et `exec` et le comportement des signaux est lui-même préservé par `fork`. La commande `exec` remet normalement les signaux à leur valeur par défaut, sauf justement si le comportement est d'ignorer le signal.

Enfin, le père doit remettre également les signaux à leur valeur initiale, immédiatement après l'appel, y compris en cas d'erreur, d'où l'utilisation de la commande `try_finalize` (ligne 20).

4.6 Le temps qui passe

Temps anciens

Dans les premières versions d'Unix, le temps était compté en secondes. Par soucis de compatibilité, on peut toujours compter le temps en secondes. La date elle-même est comptée en secondes écoulées depuis le 1er janvier 1970 à `00:00:00 GMT`. Elle est retournée par la fonction :

```
val time : unit -> float
```

L'appel système `sleep` peut arrêter l'exécution du programme pendant le nombre de secondes donné en argument :

```
val sleep : int -> unit
```

Cependant, cette fonction n'est pas primitive. Elle est programmable avec des appels systèmes plus élémentaire à l'aide de la fonction `alarm` (vue plus haut) et `sigsuspend` :

```
val sigsuspend : int list -> unit
```

L'appel `sigsuspend` l suspend temporairement les signaux de la liste l, puis arrête l'exécution du programme jusqu'à la réception d'un signal non ignoré non suspendu (au retour, le masque des signaux est remis à son ancienne valeur par le système).

Exemple: Nous pouvons maintenant programmer la fonction `sleep`.

```
1  let sleep s =
2    let old_alarm = signal sigalrm (Signal_handle (fun s -> ())) in
3    let old_mask = sigprocmask SIG_UNBLOCK [ sigalrm ] in
4    let _ = alarm s in
5    let new_mask = List.filter (fun x -> x <> sigalrm) old_mask in
6    sigsuspend new_mask;
7    let _ = alarm 0 in
8    ignore (signal sigalrm old_alarm);
9    ignore (sigprocmask SIG_SETMASK old_mask);;
```

Dans un premier temps, le comportement du signal `sigalarm` est de ne rien faire. Notez que ≪ne rien faire≫ n'est pas équivalent à ignorer le signal. Dans le second cas, le processus ne serait pas réveillé à la réception du signal. Le signal `sigalarm` est mis dans l'état non bloquant. Puis on se met en attente en suspendant tous les autres signaux qui ne l'étaient pas déjà (`old_mask`).

Après le réveil, on défait les modifications précédentes. (Noter que la ligne 9 aurait pu est placée immédiatement après la ligne 2, car l'appel à `sigsuspend` préserve le masque des signaux.)

Temps modernes et sabliers

Dans les Unix récents, le temps peut aussi se mesurer en micro-secondes. En OCaml les temps en micro-secondes sont représentés comme des flottants. La fonction `gettimeofday` est l'équivalent de `time` pour les temps modernes.

```
val gettimeofday : unit -> float
```

Les sabliers

Dans les Unix modernes chaque processus est équipé de trois sabliers, chacun décomptant le temps de façon différente. Les types de sabliers sont :

`ITIMER_REAL`	temps réel	`sigalrm`
`ITIMER_VIRTUAL`	temps utilisateur	`sigvtalrm`
`ITIMER_PROF`	utilisateur et système (debug)	`sigprof`

L'état d'un sablier est décrit par le type `interval_timer_status` qui est une structure à deux champs (de type `float` représentant le temps).

- Le champ `it_interval` indique la période du sablier.
- Le champ `it_value` est la valeur courante du sablier ; lorsqu'elle devient nulle, le signal `sigvtalarm` est émis et le sablier est remis à la valeur `it_interval`.

Un sablier est donc inactif lorsque ses deux champs sont nuls. Les sabliers peuvent être consultés ou modifiés avec les fonctions suivantes :

```
val getitimer : interval_timer -> interval_timer_status
val setitimer :
    interval_timer -> interval_timer_status -> interval_timer_status
```

La valeur retournée par `setitimer` est l'ancienne valeur du sablier au moment de la modification.

Exercice 11 *Pour gérer plusieurs sabliers, écrire une module ayant l'interface suivante :*

```
module type Timer = sig
  open Unix
  type t
  val new_timer : interval_timer -> (unit -> unit) -> t
  val get_timer : t -> interval_timer_status
  val set_timer : t -> interval_timer_status -> interval_timer_status
end
```

La fonction `new_timer` *k f créé un nouveau sablier de type k déclenchant l'action f, inactif à sa création ; la fonction* `set_timer` *t permettant de régler le sablier t (l'ancien réglage étant retourné).* □

Calcul des dates

Les versions modernes d'Unix fournissent également des fonctions de manipulation des dates en bibliothèque : voir la structure `tm` qui permet de représenter les dates selon le calendrier (année, mois, etc.) ainsi que les fonctions de conversion : `gmtime`, `localtime`, `mktime`, *etc.* dans l'annexe B.2.

4.7 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 et de difficultés :

- L'information transmise est minimale—le type du signal et rien d'autre.
- Les signaux peuvent 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. Le cas échéant, c'est tout le programme qui doit se protéger contre l'accès concurrent aux structures modifiables par l'action associée à un signal.
- l'utilisation des signaux par le programme principal oblige à prendre en compte la possibilité que les appels systèmes longs soient interrompus dans tout le code du programme principal, même si les signaux gardent leur comportement par défaut.
- les fonctions de bibliothèque d'usage général doivent donc toujours considérer l'utilisation de signaux dans le programme principal et se prémunir contre les interruptions des appels systèmes.

Les signaux apportent donc toutes les difficultés de la communication asynchrone, tout en n'en fournissant qu'une forme très limitée. On aura donc intérêt à s'en passer lorsqu'il que c'est possible, par exemple en utilisant `select` plutôt qu'une alarme pour se mettre en attente. Toutefois, dans certaines situations (langage de commandes, par essence), leur prise en compte est vraiment indispensable.

Les signaux sont peut-être la partie la moins bien conçue du système Unix. Sur certaines anciennes 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 stdout '\007'; flush stdout;
  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 ou Linux) 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 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.

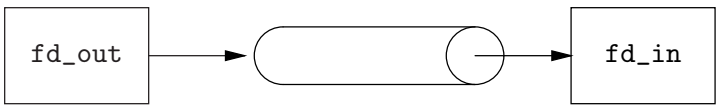
5.1 Les tuyaux

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 (parfois également appelés tubes) 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` :

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

Un appel à pipe retourne une paire `(fd_in, fd_out)`. Le premier résultat `fd_in` est un descripteur ouvert en *lecture* sur la sortie du tuyau ; le deuxième résultat `fd_out` 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 créé 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` aient 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

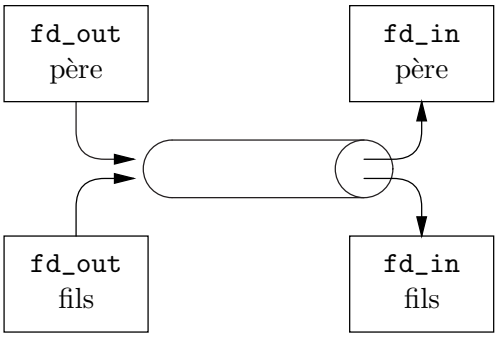
lectures retournent dès qu'il y a quelque chose dans le tuyau, sans attendre que le nombre d'octets demandés à `read` soit atteint.

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 dans 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.

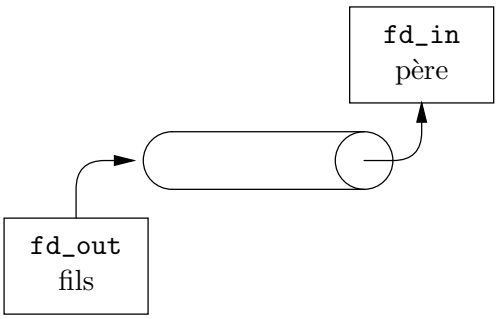
Exemple: le fragment de code ci-dessous est typique.

```
1 let (fd_in, fd_out) = pipe() in
2 match fork() with
3   0 ->
4     close fd_in;
5     ... write fd_out buffer1 offset1 count1 ...
6 | k ->
7     close fd_out;
8     ... read fd_in 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.



Dans cet exemple, on a choisi de faire du fils l'écrivain, et du père le lecteur. Le fils ferme donc son descripteur `fd_in` sur la sortie du tuyau (pour économiser les descripteurs, et pour éviter certaines erreurs de programmation). Cela laisse le descripteur `fd_in` du père inchangé, car les descripteurs sont alloués dans la mémoire du processus et, après le fork, la mémoire du fils et celle du père sont disjointes. Le tuyau, alloué dans la mémoire système, continue à exister puisqu'il y a encore le descripteur `fd_in` du père ouvert en lecture sur sa sortie. Le père ferme de même son descripteur sur l'entrée du tuyau. La situation est donc la suivante :

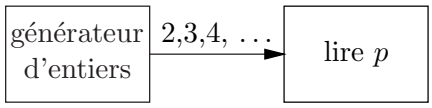


Les données que le fils écrit sur `fd_out` arrivent donc bien jusqu'au descripteur `fd_in` du père.

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), ou si le comportement du signal `SIGPIPE` à été modifié alors l'appel système `write` échoue avec une erreur `EPIPE`.

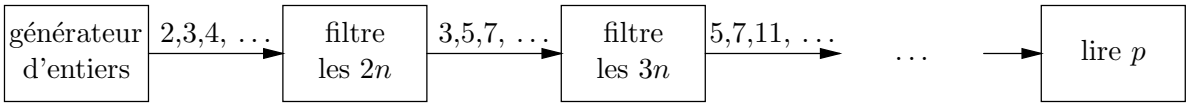
5.2 Exemple complet : le crible d'Ératosthène parallèle

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 mono-processeurs, 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 de 2 à k.

```
1  open Unix;;
2
3  let input_int = input_binary_int
4  let output_int = output_binary_int
5
6  let generate k output =
7    let rec gen m =
8      output_int output m;
9      if m < k then gen (m+1)
10   in gen 2;;
```

Pour communiquer les entiers en la sortie d'un générateur et l'entrée du filtre suivant on peut utiliser les fonctions suivantes :

La fonction `output_binary_int` est une fonction de la bibliothèque standard 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 standard est double : premièrement, il n'y a pas à faire soi-même les fonctions de conversion entiers/chaînes de quatre caractères (la représentation n'est pas spécifiée, mais il est garanti que pour une version du langage, elle est indépendante de la machine) ; deuxièmement, on fait beaucoup moins d'appels système, car les entrées/sorties sont temporisées, d'où de meilleures performances. Les deux fonctions ci-dessous construisent un `in_channel` ou un `out_channel` qui temporise les lectures ou les écritures sur le descripteur Unix donné en argument :

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

Ces fonctions permettent de faire des entrées/sorties temporisées sur des descripteurs obtenus indirectement ou autrement que par une ouverture de fichier. Leur utilisation n'a pas pour but de mélanger les entrées/sorties temporisés avec des entrées sorties non temporisées, ce qui est possible mais très délicat et fortement déconseillé, en particulier pour les lectures. Il est également possible mais très risqué de construire plusieurs `in_channel` (par exemple) sur le même descripteur de fichier.

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 au fur et à mesure de leur construction, et on en renvoie la liste.

```
11 let print_prime n = print_int n; print_newline()
12
13 let read_first_primes input count =
14   let rec read_primes first_primes count =
15     if count <= 0 then
16       first_primes
17     else
18       let n = input_int input in
19       if List.exists (fun m -> n mod m = 0) first_primes then
20         read_primes first_primes count
21       else begin
22         print_prime n;
23         read_primes (n :: first_primes) (count - 1)
24       end in
25   read_primes [] count;;
```

Voici la fonction filtre proprement dite.

```
26 let rec filter input =
27   try
28     let first_primes = read_first_primes input 1000 in
29     let (fd_in, fd_out) = pipe() in
30     match fork() with
31       0 ->
32         close fd_out;
33         filter (in_channel_of_descr fd_in)
34     | p ->
```

```
35          close fd_in;
36          let output = out_channel_of_descr fd_out in
37          while true do
38            let n = input_int input in
39            if List.exists (fun m -> n mod m = 0) first_primes then ()
40            else output_int output n
41          done
42    with End_of_file -> ();;
```

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 (`crible` k énumère les nombres premiers plus petits que k. Si k est omis (ou n'est pas un entier) il prend la valeur par défaut `max_int`).

```
43  let crible () =
44    let len = try int_of_string Sys.argv.(1) with _ -> max_int in
45    let (fd_in, fd_out) = pipe () in
46    match fork() with
47      0 ->
48        close fd_out;
49        filter (in_channel_of_descr fd_in)
50    | p ->
51        close fd_in;
52        generate len (out_channel_of_descr fd_out);;
53
54  handle_unix_error crible ();;
```

Ici, nous n'avons pas attendu que les fils terminent pour terminer le programme. Les processus pères sont des générateurs pour leurs fils. Lorsque k est donné, le père va s'arrêter en premier, et fermer le descripteur sur le tuyau lui permettant de communiquer avec son fils. Lorsqu'il termine, OCaml vide les tampons sur les descripteurs ouverts en écriture, donc le processus fils peut lire les dernières données fournies par le père. Il s'arrête à son tour, *etc.* Les fils deviennent donc orphelins et sont temporairement rattachés au processus 1 avant de terminer à leur tour.

Lorsque k n'est pas fourni, tous les processus continuent indéfiniment (jusqu'à ce que l'un ou plusieurs soient tués). La mort d'un processus entraîne la terminaison d'un de ses fils comme dans le cas précédent et la fermeture en lecture du tuyau qui le relie à son père. Cela provoquera la terminaison de son père à la prochaine écriture sur le tuyau (le père recevra un signal `SIGPIPE`, dont l'action par défaut est la terminaison du processus).

Exercice 12 *Comment faut-il modifier le programme pour que le père attende la terminaison de ses fils ?* (Voir le corrigé) □

Exercice 13 *À chaque nombre premier trouvé, la fonction* `print_prime` *exécute l'expression* `print_newline()` *qui effectue un appel système pour vider de tampon de la sortie standard, ce qui limite artificiellement la vitesse d'exécution. En fait* `print_newline()` *exécute* `print_char '\n'` *suivi de* `flush Pervasives.stdout`*. Que peut-il se passer si on exécute simplement* `print_char '\n'` *? Que faire alors ?* (Voir le corrigé) □

5.3 Les tuyaux nommés

Certaines variantes d'Unix (System V, SunOS, Ultrix, Linux) fournissent la possibilité de créer des tuyaux associés à un nom dans le système de fichiers. Ces tuyaux nommés (aussi appelés *fifo*) 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 :

```
val 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 l'appel système `openfile`, comme pour un fichier normal. Lectures et écritures sur un tuyau nommé ont la même sémantique que sur un tuyau simple. Cependant l'ouverture d'un tuyau nommé en lecture seule ou en écriture seule est bloquante jusqu'à ce que ce tuyau soit ouvert par un autre processus en écriture ou en lecture, respectivement (ce qui peut déjà être le cas et il n'y a pas à attendre), à moins que l'ouverture soit faite avec le drapeau `O_NONBLOCK`. Dans ce denier cas les lectures ou écritures effectués sur le tuyau ne seront pas non plus bloquantes. On peut changer ce drapeau une fois le tuyau ouvert avec la fonction `clear_nonblock` et rendre les lectures ou les écritures suivantes sur le tuyau bloquantes (ou non bloquantes avec `set_nonblock`).

```
val set_nonblock : file_descr -> unit
val clear_nonblock : file_descr -> unit
```

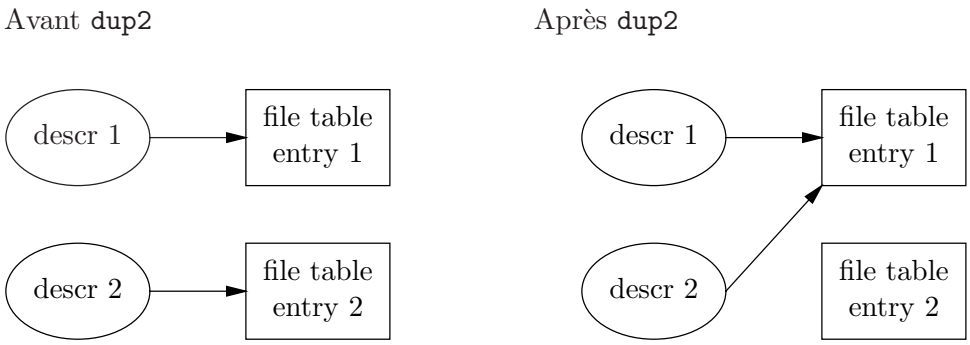
5.4 Redirections de descripteurs

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 `openfile` 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 = openfile "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 = openfile "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_in, fd_out) = pipe() in
match fork() with
  0 -> dup2 fd_in stdin;
       close fd_out;
       close fd_in;
       execvp "cmd2" [|"cmd2"|]
| _ -> dup2 fd_out stdout;
       close fd_out;
       close fd_in;
       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'il est censé faire quand il 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 14 *Comment implémenter quelques-unes des autres redirections du shell* `sh`*, à savoir :*

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

(Voir le corrigé) □

Échanger deux descripteurs est plus délicat : la séquence `dup2 fd1 fd2; dup2 fd2 fd1` à laquelle on pourrait penser naïvement ne convient pas. En effet, la seconde opération de redirection n'a aucun effet, puisqu'après la premier les deux descripteurs `fd1` et `fd2` pointent déjà vers la même *file table entry*. L'ancienne valeur de `fd2` a été perdue. Comme pour intervertir le contenu de deux références, il faut utiliser une variable auxiliaire pour sauvegarde l'une des deux valeurs. Ici, il faut sauvegarder l'un des deux descripteurs, en le recopiant avec l'appel système `dup`.

```
val dup : file_descr -> file_descr
```

L'appel `dup fd` retourne un nouveau descripteur qui pointe vers la même *file table entry* que `fd`. Par exemple, on peut maintenant échanger `stdout` et `stderr` par le code suivant :

```
let tmp = dup stdout in
dup2 stderr stdout;
dup2 tmp stderr;
close tmp;;
```

Ne pas oublier de refermer le descripteur temporaire `tmp`, devenu inutile, afin d'éviter une fuite de mémoire.

5.5 Exemple complet : composer N commandes

On va construire une commande `compose`, telle que

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

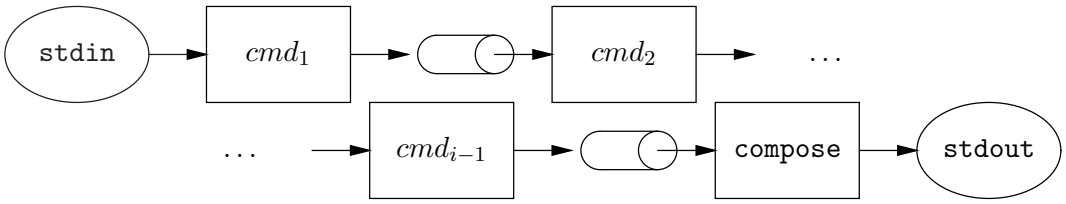
se comporte comme la commande shell

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

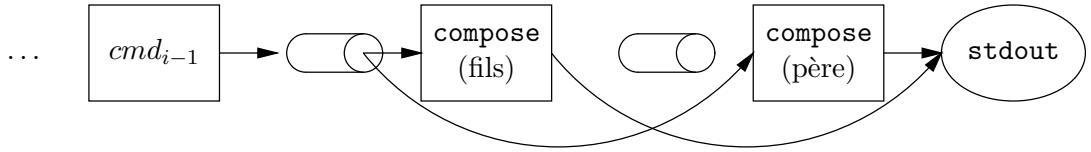
```
1  open Sys;;
2  open Unix;;
3
4  let compose () =
5    let n = Array.length Sys.argv - 1 in
6    for i = 1 to n - 1 do
7      let (fd_in, fd_out) = pipe() in
8      match fork() with
```

```
9         0 ->
10          dup2 fd_out stdout;
11          close fd_out;
12          close fd_in;
13          execv "/bin/sh" [| "/bin/sh"; "-c"; Sys.argv.(i) |]
14      | _ ->
15          dup2 fd_in stdin;
16          close fd_out;
17          close fd_in
18    done;
19    match fork() with
20      0 ->
21        execv "/bin/sh" [|"/bin/sh"; "-c"; Sys.argv.(n) |]
22    | _ ->
23        let rec wait_for_children retcode =
24          try
25            match wait() with
26              (pid, WEXITED n) -> wait_for_children (retcode lor n)
27            | (pid, _)         -> wait_for_children 127
28          with
29              Unix_error(ECHILD, _, _) -> retcode in
30        exit (wait_for_children 0)
31  ;;
32  handle_unix_error compose ();;
```

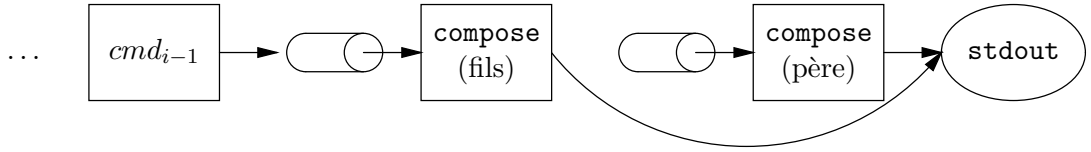
L'essentiel du travail est fait par la boucle `for` des lignes 6–18. Pour chaque commande sauf la dernière, on crée un nouveau tuyau, puis un nouveau processus. Le nouveau processus (le fils) relie l'entrée du tuyau à sa sortie standard, et exécute la commande. Il hérite l'entrée standard du processus 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 soit la suivante :



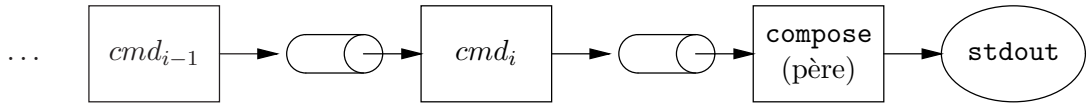
(Les carrés représentent des processus, avec leur entrée standard à gauche et leur sortie standard à droite. Les deux ellipses représentent l'entrée standard et la sortie standard initiales du processus `compose`.) Après `pipe` et `fork`, la situation est la suivante :



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.)

5.6 Multiplexage d'entrées-sorties

Dans les exemples qu'on a vus 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.

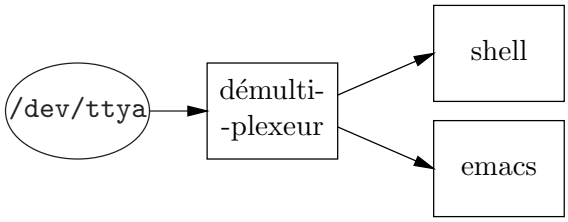
On considère l'exemple d'un émulateur de terminal multi-fenêtres sur un micro-ordinateur. C'est-à-dire, on a un micro-ordinateur relié à une machine Unix par une liaison série, et on cherche à faire émuler par le micro-ordinateur plusieurs terminaux, affichant chacun dans une fenêtre différente sur l'écran 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 :

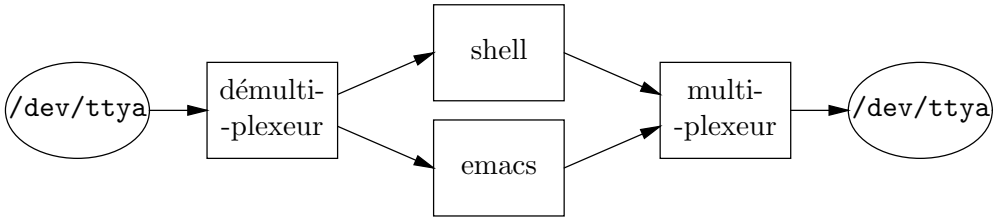
- un octet indiquant le numéro du processus ou de la fenêtre destinataire
- un octet indiquant le nombre N d'octets de données qui suivent
- N octets représentant les données à transmettre au destinataire.

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.

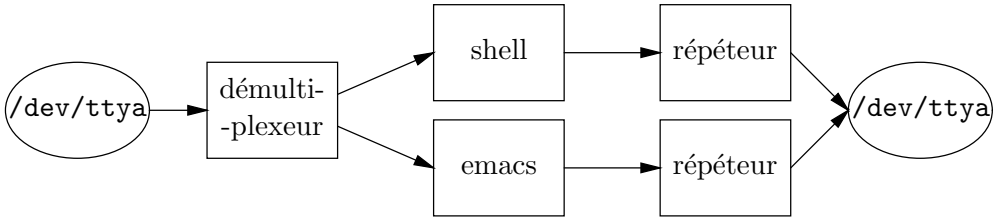


Le multiplexage (transmission dans le sens processus vers micro) est plus délicat. Essayons l'approche symétrique du démultiplexage : un processus lit successivement sur les tuyaux reliés aux sorties standard des processus utilisateur, puis met les données lues sous forme de paquets (en ajoutant le numéro 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 `openfile` 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.

```
val 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 ce 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 délai de 0,5s est écoulé *)
| fdl,[],[] ->
    if List.mem fd1 fdl then
```

```
        (* lire depuis fd1 *);
    if List.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. (Pour simplifier, le multiplexeur se contente d'étiqueter les messages en fonction de leur provenance et le démultiplexeur suppose que les messages sont étiquetés en fonction de leur destinataire : on suppose donc soit que chaque émetteur parle à une destinataire de même numéro, soit qu'au milieu de la ligne on établit la correspondance émetteur destinataire en ré-étiquetant les messages.)

La fonction `multiplex` prend un descripteur ouvert sur la liaison série, et deux tableaux de descripteurs de même taille, l'un contenant les tuyaux reliés aux entrées standard des processus utilisateur, l'autre les tuyaux reliés à leurs sorties standard.

```
1  open Unix;;
2
3  let rec really_read fd buff start length =
4    if length <= 0 then () else
5      match read fd buff start length with
6        0 -> raise End_of_file
7      | n -> really_read fd buff (start+n) (length-n);;
8
9  let buffer = String.create 258;;
10
11 let multiplex channel inputs outputs =
12   let input_fds = channel :: Array.to_list inputs in
13   try
14     while true do
15       let (ready_fds, _, _) = select input_fds [] [] (-1.0) in
16       for i = 0 to Array.length inputs - 1 do
17         if List.mem inputs.(i) ready_fds then begin
18           let n = read inputs.(i) buffer 2 255 in
19           buffer.[0] <- char_of_int i;
20           buffer.[1] <- char_of_int n;
21           ignore (write channel buffer 0 (n+2));
22           ()
23         end
24       done;
25       if List.mem channel ready_fds then begin
26         really_read channel buffer 0 2;
27         let i = int_of_char(buffer.[0])
28         and n = int_of_char(buffer.[1]) in
29         if n = 0 then
30           close outputs.(i)
31         else begin
32           really_read channel buffer 0 n;
33           ignore (write outputs.(i) buffer 0 n);
34           ()
35         end
36       end
```

```
37      done
38    with End_of_file ->
39      ()
40  ;;
```

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 `input_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 $\texttt{n} = 0$ 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.

5.7 Miscelleaneous : `write`

La commande `Unix.write` itère l'appel système `write` jusqu'à ce que la quantité demandée soit effectivement la quantité transférée. Lorsque le descripteur est un tuyau (ou une prise que l'on verra dans le chapitre suivant), l'écriture est par défaut bloquante. L'appel système `write` peut donc être interrompu en présence de signaux. L'erreur `EINTR` est alors remontée vers OCaml, alors qu'une partie des données a déjà pu être transférée par un appel précédent à `write`. La taille des données transférée est alors perdue, irrémédiablement, ce qui rend `Unix.write` inutilisable en présence de signaux.

Pour remédier à ce problème, OCaml relève également l'appel système `write` sous le nom `single_write`. Nous montrons sur cet exemple comment relever un appel système en OCaml. Il s'agit en fait simplement d'interfacer OCaml avec du code C et on pourra se reporter à la section correspondante du manuel OCaml. Le code suivant est écrit dans un fichier `single_write.c`.

```
1  #include <errno.h>
2  #include <string.h>
3  #include <caml/mlvalues.h>
4  #include <caml/memory.h>
5  #include <caml/signals.h>
6  #include "unixsupport.h"
7
8  CAMLprim value caml_single_write
9          (value fd, value buf, value vofs, value vlen) {
10   CAMLparam4(fd, buf, vofs, vlen);
11   long ofs, len;
12   int numbytes, ret;
```

```
13    char iobuf[UNIX_BUFFER_SIZE];
14    ofs = Long_val(vofs)
15    len = Long_val(vlen)
16    ret = 0;
17    if (len > 0) {
18      numbytes = len > UNIX_BUFFER_SIZE ? UNIX_BUFFER_SIZE : len;
19      memmove (iobuf, &Byte(buf, ofs), numbytes);
20      enter_blocking_section();
21      ret = write(Int_val(fd), iobuf, numbytes);
22      leave_blocking_section();
23      if (ret == -1) uerror("single_write", Nothing);
24    }
25    CAMLreturn (Val_int(ret));
26  }
```

Les deux premières lignes incluent des bibliothèques C standards. Les trois suivantes incluent des bibliothèques C spécifiques à OCaml et installées avec. La bibliothèque `unixsupport.h` qui permet de réutiliser certaines fonctions C d'Unix comme le traitement d'erreur, etc. n'est pas installée par défaut et il faut la récupérer dans les sources de la distribution.

La ligne la plus significative est l'appel à `write` (ligne 21). Comme celui-ci est bloquant (si le descripteur est un tuyau ou une chaussette), il faut relâcher le verrou `OCaml` immédiatement avant l'appel et le reprendre après (lignes 20 et 22), afin d'être compatible avec la bibliothèque `Thread` et de permettre éventuellement à d'autres coprocessus de s'exécuter pendant l'attente (voir le Chapitre 7). OCaml peut donc effectuer un GC pendant l'exécution de l'appel système, ce qui oblige à se prémunir contre un déplacement de la chaîne OCaml `buf` en la recopiant dans une chaîne C `iobuf`. Cela a un coût supplémentaire, mais seulement de l'ordre de 10% et non de l'ordre de 50% comme on pourrait s'y attendre, car la gestion de l'appel système et des structures systèmes liés à la copie gardent un coût prépondérant.

La taille de cette chaîne est bornée par `UNIX_BUFFER_SIZE`, afin d'éviter des à coups inutiles, définie dans `unix_support.h`. Une erreur pendant l'appel système, signalée par une valeur de retour strictement négative, est propagée vers OCaml par la fonction `uerror`, définie dans la bibliothèque `Unix`.

Pour remonter ce code en OCaml, le fichier `unix.mli` déclare

```
external unsafe_single_write :
  file_descr -> string -> int -> int -> int = "caml_single_write"
```

En pratique, on vérifie les arguments avant d'appeler cette fonction.

```
let single_write fd buf ofs len =
  if ofs < 0 || len < 0 || ofs > String.length buf - len
  then invalid_arg "Unix.write"
  else unsafe_single_write fd buf ofs len
```

Cette fonction est disponible en OCaml depuis la version `3.08`. Autrement, pour utiliser cette fonction dans un programme `main.ml`, en supposant que l'on ait écrit le code précédant dans des fichiers `write.mli` et `write.ml`, il aurait fallu compiler comme suit :

```
ocamlc -c single_write.c write.ml
ocamlc -custom -o prog unix.cma single_write.o write.cmo mod1.ml mod2.ml
```

Il est souvent plus pratique de fabriquer une bibliothèque `write.cma` réunissant le code C et OCaml :

```
ocamlc -custom -a -o write.cma single_write.o write.cmo
```

On peut alors utiliser `write.cma` simplement comme on utilise `unix.cma` :

```
ocamlc -o main.byte unix.cma write.cma main.ml
```

La fonction `single_write` reproduit l'appel `write` aussi fidèlement que possible. La seule différence reste que lorsque la chaîne d'origine est très longue, l'appel est autoritairement découpé en plusieurs appels. L'atomicité de l'appel à `write` (garantie dans le cas des fichiers réguliers) n'est donc pas préservée pour des écritures très longues. Cette différence est assez marginale, mais il faut en être conscient.

Au dessus de cet appel nous pouvons réimplanter une fonction de plus haut niveau `really_write` analogue à `really_read` qui écrit effectivement la quantité demandée.

```
let rec really_write fd buffer offset len =
  let n = restart_on_EINTR (single_write fd buffer offset) len in
  if n < len then really_write fd buffer (offset + n) (len - n);;
```

Chapitre 6

Communications modernes : les prises

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 même 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-window — 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 (traduction libre de *sockets*) sont une généralisation des tuyaux qui pallie ces faiblesses. Le modèle du client serveur avec prises (en mode connecté) est décrit dans la figure 6.1

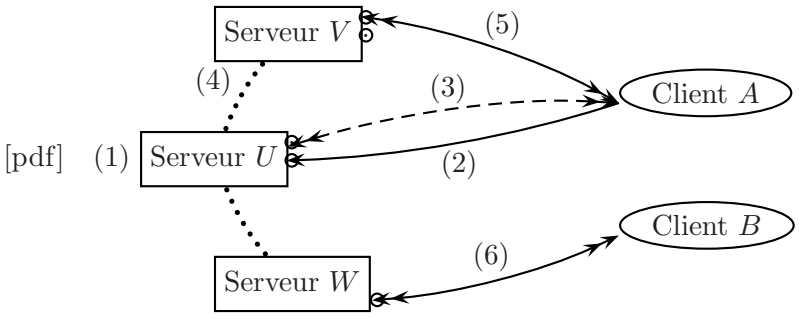


FIGURE 6.1 – Modèle Client-Serveur

1. Le serveur U créé une prise s et la branche sur un port p connu des clients puis attend les connexions sur sa prise (1).
2. Un client A créé une prise et se connecte au serveur sur le port p (2). Le système alloue alors une nouvelle prise pour communiquer en privé avec le client A (3). Dans le schéma choisi ici, il se duplique en un serveur auxiliaire V (4), ferme sa prise avec le client A (en trait haché), et laisse son fils V traiter la connexion avec A (5).
3. Le serveur peut alors accepter un nouveau client B, établir une autre connexion en parallèle servie par un autre clone W (6), et ainsi de suite.
4. Le serveur peut fermer son service en fermant le descripteur associé à la prise s. Au bout d'un certain temps le système libère le port p qui peut alors être réutilisé, par exemple pour y installer un autre service.

Il est essentiel dans ce modèle que le serveur U et le client A aient établi une connexion privée (3) pour dialoguer jusqu'à la fin de la connexion, sans interférence avec d'autres requêtes provenant d'autres clients. Pour cette raison, on dit que l'on fonctionne en mode *connecté*. Si le service est court, le serveur pourrait lui-même servir la requête directement (sans se cloner) au travers de la connexion (3). Dans ce cas, le client suivant doit attendre que le serveur soit libre, soit parce qu'il a effectivement fini de traiter la connexion (3), soit parce qu'il gère explicitement plusieurs connexions par multiplexage.

Les prises permettent également un modèle client-serveur en mode *déconnecté*. Dans ce cas, moins fréquent, le serveur n'établit pas une connexion privée avec le client, mais répond directement à chaque requête du client. Nous verrons brièvement comment procéder selon ce modèle dans la section 6.10.

6.1 Les prises

Le mécanisme des prises, qui étend celui des tuyaux, a été introduit en BSD 4.2, et se retrouve maintenant sur toutes les machines Unix connectées à un réseau (Ethernet ou autre). Tout d'abord, des appels systèmes spéciaux sont fournis pour établir des connexions suivant le modèle client-serveur. Ensuite, les prises permettent la communication locale ou à travers le réseau entre processus tournant sur des machines différentes de façon (presque) transparente. Pour cela, 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'octets, ou flot de paquets — petits

1. Le réseau Internet se compose de réseaux locaux, généralement du type Ethernet, reliés par des liaisons spécialisées. Il relie des millions de machines dans le monde entier. 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 routeurs et une liaison satellite.

blocs d'octets délimités). La sémantique conditionne le protocole utilisé pour la transmission des données. Voici trois exemples de sémantiques possibles :

	Flots	Datagramme	Paquet segmentés
Fiable	oui	non	oui
Forme des données	flot d'octets	paquets	paquets

La sémantique par "flot" 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 par "paquets segmentés" 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 par "datagrammes" 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-mêmes les pertes.

6.2 Création d'une prise

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

```
val 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.

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_RAW`	accès direct aux couches basses du réseau
`SOCK_SEQPACKET`	paquets, fiable

Le troisième argument est le numéro du protocole de communication à utiliser. Il vaut généralement 0, ce qui désigne le protocole par défaut, généralement déterminé à partir du type de communication (typiquement, `SOCK_DGRAM` et `SOCK_STREAM` sont associés aux protocoles `udp` et `tcp`). 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` du système d'informations réparti NIS (*Network Information Service*), le cas échéant. L'appel système `getprotobyname` permet de consulter cette table de manière portable :

```
val 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.

6.3 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 :

```
val inet_addr_of_string : string -> inet_addr
val 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. On peut consulter cette table ainsi que la base de donnée NIS par la fonction de bibliothèque `gethostbyname`. Sur les machines modernes, cette fonction interroge les "name servers" soit en cas d'échec, soit au contraire de façon prioritaire, n'utilisant alors le fichier `/etc/hosts` qu'en dernier recours.

```
val 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 de façon portable par la fonction

```
val 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 `pauillac.inria.fr` :

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

6.4 Connexion à un serveur

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

```
val 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`.

Un effet de `connect` est de brancher la prise à une adresse locale qui est choisie par le système. Parfois, il est souhaitable de choisir soi-même cette adresse, auquel cas il est possible d'appeler l'opération `bind` (voir ci-dessous) avant `connect`.

Pour suivre les connexions en cours cours on peut utiliser la commande `netstat` depuis un shell.

6.5 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.

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

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

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

En fait, la désallocation d'une prise peut prendre un certain temps que celle-ci soit faite avec politesse ou brutalité.

6.6 Exemple complet : Le client universel

On va définir une commande `client` telle que `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.

Par exemple, la commande

```
echo -e 'GET /~remy/ HTTP/1.0\r\n\r\n' | ./client pauillac.inria.fr 80
```

se connecte sur le port `80` et `pauillac.inria.fr` et envoie la commande `HTTP` qui demande la page web d'accueil `/~remy/` sur ce serveur.

Cette commande constitue une application client "universel", dans la mesure où elle regroupe le code d'établissement de connexions qui est commun à beaucoup de clients, tout en déléguant la partie implémentation du protocole de communication, propre à chaque application au programme qui appelle `client`.

Nous utilisons une fonction de bibliothèque `retransmit` qui recopie les données arrivant d'un descripteur sur un autre descripteur. Elle termine lorsque la fin de fichier est atteinte sur le descripteur d'entrée, sans refermer les descripteurs. Notez que `retransmit` peut-être interrompue par un signal.

```
let retransmit fdin fdout =
  let buffer_size = 4096 in
  let buffer = String.create buffer_size in
  let rec copy() =
    match read fdin buffer 0 buffer_size with
      0 -> ()
    | n -> ignore (write fdout buffer 0 n); copy() in
  copy ();;
```

Les choses sérieuses commencent ici.

```
open Sys;;
open Unix;;

let client () =
  if Array.length Sys.argv < 3 then begin
    prerr_endline "Usage: client <host> <port>";
    exit 2;
  end;
  let server_name = Sys.argv.(1)
  and port_number = int_of_string Sys.argv.(2) in
  let server_addr =
    try (gethostbyname server_name).h_addr_list.(0)
    with Not_found ->
      prerr_endline (server_name ^ ": Host not found");
      exit 2 in
  let sock = socket PF_INET SOCK_STREAM 0 in
  connect sock (ADDR_INET(server_addr, port_number));
  match fork() with
    0 ->
      Misc.retransmit stdin sock;
      shutdown sock SHUTDOWN_SEND;
      exit 0
  | _ ->
      Misc.retransmit sock stdout;
      close stdout;
      wait();;

handle_unix_error client ();;
```

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. La commande `gethostbyname` traite correctement les deux cas de figure. Dans le cas d'une adresse symbolique, la base `/etc/hosts` est interrogée et on prend la première des adresses obtenues. Dans le cas d'une adresse numérique aucune vérification n'est effectuée : une structure est simplement créée pour l'adresse demandée.

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.

La fermeture de la connexion peut se faire à l'initiative du client ou du serveur.

- Le client reçoit une fin de fichier sur son entrée standard. Le client (fils) ferme alors la connexion dans le sens client vers serveur et termine. En retour, le serveur qui reçoit une fin de fichier sur son entrée standard devrait, éventuellement après un court traitement, fermer la connexion. Donc à l'autre bout de la prise `sock`, le client (père) reçoit finalement une fin de fichier sur la connexion et termine normalement.
- Le serveur ferme prématurément la connexion. Le client (père) qui essaye d'écrire dans la prise `sock` fermée reçoit le signal `sigpipe` ce qui par défaut tue le client. C'est la sémantique attendue. Toutefois, le client meurt immédiatement, sans pouvoir indiqué que la connexion a été coupée. Pour récupérer cette information, on peut ignorer le signal `SIGPIPE` avec pour effet d'envoyer au client l'erreur `EPIPE` qui sera alors traitée par le handler `handler_unix_error` : il suffit d'ajouter la ligne suivante entre les lignes 19 et 20 du client.

```
ignore (signal sigpipe Signal_ignore)
```

Si le client, fils ou père, termine prématurément la prise sera fermée en écriture ou en lecture. Si le serveur détecte cette information, il ferme l'autre bout de la prise, ce que l'autre partie du client va détecter. Sinon, le serveur termine normalement en fermant la connexion. Dans les deux cas, on se retrouve également dans l'un des scénarios précédents.

6.7 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` :

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

Le premier argument est le descripteur de la prise ; le second, l'adresse à lui attribuer. La commande `bind` peut aussi utiliser une adresse spéciale `inet_addr_any` représentant toutes les adresses internet possibles sur la machine (qui peut comporter plusieurs sous-réseaux).

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

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

Le premier argument est le descripteur de la prise. Le second indique combien de demandes de connexion incomplètes peuvent être mises en attente. Sa valeur, souvent de l'ordre de quelques dizaines peut aller jusqu'à quelques centaines pour des serveurs très sollicités. Lorsque ce nombre est dépassé, les demandes de connexion excédentaires échouent.

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

```
val 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 par exemple (`xhost` permettant d'ajouter de nouvelles autorisations) ou à établir une seconde connexion du serveur vers le client (comme le fait `ftp` pour chaque demande de transfert de fichier).

Le schéma général d'un serveur `tcp` est donc de la forme suivante (nous définissons ces fonctions dans la bibliothèque `Misc`).

```
1  let install_tcp_server_socket addr =
2    let s = socket PF_INET SOCK_STREAM 0 in
3    try
4      bind s addr;
5      listen s 10;
6      s
7    with z -> close s; raise z;;
8  let tcp_server treat_connection addr =
9    ignore (signal sigpipe Signal_ignore);
10   let server_sock = install_tcp_server_socket addr in
11   while true do
12       let client = restart_on_EINTR accept server_sock in
13       treat_connection server_sock client
14   done;;
```

La fonction `install_tcp_server_socket` commence par créer une prise dans le domaine Internet, avec la sémantique «stream» et le protocole par défaut (ligne 2), puis il 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 4 et 5. Comme il s'agit d'une fonction de bibliothèque, nous refermons proprement la prise en cas d'erreur lors de l'opération `bind` ou `listen`. La fonction `tcp_server` crée la prise avec la fonction précédente, puis entre dans une boucle infinie, où elle attend une demande de connexion (`accept`, ligne 12) et traite celle-ci (ligne 13). Comme il s'agit d'une fonction de bibliothèque, nous avons pris soin de relancer l'appel système `accept` (bloquant) en cas d'interruption. Notez qu'il appartient à la fonction `treat_connection` de fermer le descripteur `client` en fin de connexion y compris lorsque la connexion se termine de façon brutale. Nous ignorons le signal `sigpipe` afin qu'une déconnexion prématurée d'un client lève une exception `EPIPE` récupérable par `treat_connection` plutôt que de tuer le processus brutalement.

La fonction `treat_connection` reçoit également le descripteur du serveur car dans le cas d'un traitement par `fork` ou `double_fork`, celui-ci devra être fermé par le fils.

Le traitement d'une connexion peut se faire séquentiellement, *i.e.* par le serveur lui même. Dans ce cas, `treat_connection` se contente d'appeler une fonction `service`, entièrement dépendante de l'application, qui est le corps du serveur et qui exécute effectivement le service demandé et se termine par la fermeture de la connexion.

```
let service (client_sock, client_addr) =
  (* Communiquer avec le client sur le descripteur descr *)
  (* Puis quand c'est fini: *)
  close client_descr;;
```

D'où la fonction auxiliaire (que nous ajoutons à la bibliothèque `Misc`) :

```
let sequential_treatment server service client = service client
```

Comme pendant le traitement de la connexion le serveur ne peut pas traiter d'autres demandes, ce schéma est en général réservé à des services rapides, où la fonction `service` s'exécute toujours en un temps cours et borné (par exemple, un serveur de `date`).

La plupart des serveurs sous-traitent le service à un processus fils : On appelle `fork` immédiatement après le retour de `accept`. Le processus fils traite la connexion. Le processus père recommence aussitôt à faire `accept`. Nous obtenons la fonction de bibliothèque suivante :

```
let fork_treatment server service (client_descr, _ as client) =
  let treat () =
    match fork() with
    | 0 -> close server; service client; exit 0
```

```
    | k -> () in
  try_finalize treat () close client_descr;;
```

Notez qu'il est essentiel de fermer le descripteur `client_descr` du père, sinon sa fermeture par le fils ne suffira pas à terminer la connexion ; de plus, le père va très vite se retrouver à court de descripteurs. Cette fermeture doit avoir lieu dans le cas normal, mais aussi si pour une raison quelconque le fork échoue—le programme peut éventuellement décider que ce n'est pas une erreur fatale et maintenir éventuellement le serveur en service.

De façon symétrique, le fils ferme le descripteur `sock` sur lequel le service à été établi avant de réaliser le service. D'une part, il n'en a plus besoin. D'autre part, le père peut terminer d'être serveur alors que le fils n'a pas fini de traiter la connexion en cours. La commande `exit 0` est importante pour que le fils meurt après l'exécution du service et ne se mette pas à exécuter le code du serveur.

Nous avons ici ignoré pour l'instant la récupération des fils qui vont devenir zombis, ce qu'il faut bien entendu faire. Il y a deux approches. L'approche simple est de faire traiter la connexion par un petit-fils en utilisant la technique du double fork.

```
let double_fork_treatment server service (client_descr, _ as client) =
  let treat () =
    match fork() with
    | 0 ->
        if fork() <> 0 then exit 0;
        close server; service client; exit 0
    | k ->
        ignore (restart_on_EINTR (waitpid []) k) in
  try_finalize treat () close client_descr;;
```

Toutefois, cette approche fait perdre au serveur tout contrôle sur son petit-fils. En général, il est préférable que le processus qui traite un service appartienne au même groupe de processus que le serveur, ce qui permet de tuer tous les services en tuant les processus du même groupe que le serveur. Pour cela, les serveurs gardent en général le modèle précédent et ajoute une gestion des fils, par exemple en installant une procédure `Misc.free_children` sur le signal `sigchld`.

6.8 Réglage des prises

Les prises possèdent de nombreux paramètres internes qui peuvent être réglés : taille des tampons de transfert, taille minimale des transferts, comportement à la fermeture, *etc.*. Ces paramètres sont de type booléen, entier, entier optionnel ou flottant. Pour des raisons de typage, il existe donc autant de primitives `getsockopt`, `getsockopt_int`, `getsockopt_optint`, `getsockopt_float`, qui permettent de consulter ces paramètres et autant de variantes de la forme `setsockopt`, *etc.*. On pourra consulter l'appendice B.2 pour avoir la liste détaillée des options et le manuel Unix (`getsockopt`) pour leur sens exact.

A titre d'exemple, voici deux types de réglages, qui ne s'appliquent qu'à des prises dans le domaine `INET` de type `SOCK_STREAM`. La déconnexion des prises au protocole `TCP` est négociée, ce qui peut prendre un certain temps. Normalement l'appel `close` retourne immédiatement, alors que le système négocie la fermeture.

```
setsockopt_optint sock SO_LINGER (Some 5);;
```

Cette option rend l'opération `close` bloquante sur la socket `sock` jusqu'à ce que les données déjà émises soient effectivement transmises ou qu'un délai de 5 secondes se soit écoulé.

```
setsockopt sock SO_REUSEADDR;;
```

L'effet principal de l'option `SO_REUSEADDR` est de permettre à l'appel système `bind` de réallouer

une prise à une adresse locale sur laquelle toutes les communications sont en cours de déconnexion. Le risque est alors qu'une nouvelle connexion capture les paquets destinés à l'ancienne connexion. Cette option permet entre autre d'arrêter un serveur et de le redémarrer immédiatement, très utile pour faire des tests.

6.9 Exemple complet : le serveur universel

On va maintenant définir une commande `server` telle que `server` *port cmd* $arg_1 \ \ldots \ arg_n$ reçoit les demandes de connexion au numéro *port*, et à chaque connexion lance la commande *cmd* avec $arg_1 \ \ldots \ arg_n$ 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
```

en utilisant la commande `client` écrite précédemment, 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.

La commande `server` constitue une application serveur "universel", dans la mesure où elle regroupe le code d'établissement de service qui est commun à beaucoup de serveurs, tout en déléguant la partie implémentation du service et du protocole de communication, propre à chaque application ou programme lancé par `server`.

```
1  open Sys;;
2  open Unix;;
3
4  let server () =
5    if Array.length Sys.argv < 2 then begin
6      prerr_endline "Usage: client <port> <command> [arg1 ... argn]";
7      exit 2;
8    end;
9    let port = int_of_string Sys.argv.(1) in
10   let args = Array.sub Sys.argv 2 (Array.length Sys.argv - 2) in
11   let host = (gethostbyname(gethostname())).h_addr_list.(0) in
12   let addr = ADDR_INET (host, port) in
13   let treat sock (client_sock, client_addr as client) =
14     (* log information *)
15     begin match client_addr with
16       ADDR_INET(caller, _) ->
17         prerr_endline ("Connection from " ^ string_of_inet_addr caller);
18     | ADDR_UNIX _ ->
19         prerr_endline "Connection from the Unix domain (???)";
20     end;
21     (* connection treatment *)
22     let service (s, _) =
23       dup2 s stdin; dup2 s stdout; dup2 s stderr; close s;
24       execvp args.(0) args in
25     Misc.double_fork_treatment sock service client in
26   Misc.tcp_server treat addr;;
```

```
27
28 handle_unix_error server ();;
```

L'adresse fournie à `tcp_server` contient l'adresse Internet de la machine qui fait tourner le programme ; la manière habituelle de l'obtenir (ligne 11) est de chercher le nom de la machine (renvoyé par l'appel `gethostname`) dans la table `/etc/hosts`. En fait, il existe en général plusieurs adresses pour accéder à une machine. Par exemple, l'adresse de la machine pauillac est `128.93.11.35`, mais on peut également y accéder en local (si l'on est déjà sur la machine pauillac) par l'adresse `127.0.0.1`. Pour offrir un service sur toutes les adresses désignant la machine, on peut utiliser l'adresse `inet_addr_any`.

Le traitement du service se fera ici par un ≪double fork≫ après avoir émis quelques informations sur la connexion. Le traitement du service consiste à rediriger l'entrée standard et les deux sorties standard vers la prise sur laquelle est effectuée la connexion puis d'exécuter la commande souhaitée. (Notez ici que le traitement du service ne peut pas se faire de façon séquentielle.)

Remarque : la fermeture de la connexion se fait sans intervention du programme `serveur`. Premier cas : le client ferme 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 recevra alors une fin de fichier sur la connexion.) Deuxième cas : le client termine prématurément et ferme la connexion dans le sens serveur vers client. Le serveur peut alors recevoir le signal `sigpipe` en essayant d'envoyer des données au 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 un signal) avant d'avoir reçu une fin de fichier. Le client recevra alors un fin de fichier lorsqu'il essayera de lire et un signal `SIGPIPE` (dans ce cas, le client meurt immédiatement) ou une exception `EPIPE` (si le signal est ignoré) lorsqu'il essayera d'écrire sur la connexion.

Précautions

L'écriture d'un serveur est en général plus délicate que celle d'un client. Alors que le client connaît le serveur sur lequel il se connecte, le serveur ignore tout de son client. En particulier, pour des services publics, le client peut être ≪hostile≫. Le serveur devra donc se protéger contre tous les cas pathologiques.

Un attaque typique consiste à ouvrir des connexions puis les laisser ouvertes sans transmettre de requête : après avoir accepté la connexion, le serveur se retrouve donc bloqué en attente sur la prise et le restera tant que le client sera connecté. L'attaquant peut ainsi saturer le service en ouvrant un maximum de connexions. Il est important que le serveur réagisse bien à ce genre d'attaque. D'une part, il devra prévoir un nombre limite de connexions en parallèle et refuser les connexions au delà afin de ne pas épuiser les ressources du système. D'autre part, il devra interrompre les connexions restées longtemps inactives.

Un serveur séquentiel qui réalise le traitement lui même et sans le déléguer à un de ses fils est immédiatement exposé à cette situation de blocage : le serveur ne répond plus alors qu'il n'a rien à faire. Une solution sur un serveur séquentiel consiste à multiplexer les connexions, mais cela peut être complexe. La solution avec le serveur parallèle est plus élégante, mais il faudra tout de même prévoir des ≪timeout≫, par exemple en programmant une alarme.

6.10 Communication en mode déconnecté

Les lectures/écritures en mode déconnecté

Le protocole `tcp` utilisé par la plupart des connexions de type `SOCK_STREAM` ne fonctionne qu'en mode connecté. Inversement, le protocole `udp` utilisé par la plupart des connexions de type `SOCK_DGRAM` fonctionne toujours de façon interne en mode déconnecté. C'est-à-dire qu'il n'y a pas de connexion établie entre les deux machines.

Ce type de prise peut être utilisé sans établir de connexion au préalable. Pour cela on utilise les appels système `recvfrom` et `sendto`.

```
val recvfrom :
  file_descr -> string -> int -> int -> msg_flag list -> int * sockaddr
val sendto :
  file_descr -> string -> int -> int -> msg_flag list -> sockaddr -> int
```

Chacun des appels retourne la taille des données transférées. L'appel `recvfrom` retourne également l'adresse du correspondant.

Une prise de type `SOCK_DGRAM` peut également être branchée avec `connect`, mais il s'agit d'une illusion (on parle de pseudo-connexion). L'effet de la pseudo-connexion est purement local. L'adresse passée en argument est simplement mémorisée dans la prise et devient l'adresse utilisée pour l'émission et la réception (les messages venant d'une autre adresse sont ignorés).

Les prises de ce type peuvent être connectées plusieurs fois pour changer leur affectation et déconnectées en les reconnectant sur une adresse invalide, par exemple 0. (Par opposition, la reconnexion d'une prise de type `SOCK_STREAM` produit en général un erreur.)

Les lectures/écritures de bas niveau

Les appels systèmes `recv` et `send` généralisent les fonctions `read` et `write` respectivement (mais ne s'appliquent qu'aux descripteurs de type prise).

```
val recv : file_descr -> string -> int -> int -> msg_flag list -> int
val send : file_descr -> string -> int -> int -> msg_flag list -> int
```

Leur interface est similaire à `read` et `write` mais elles prennent chacune en plus une liste de drapeaux dont la signification est la suivante : `MSG_OOB` permet d'envoyer une valeur exceptionnelle ; `MSG_DONTROUTE` indique de court-circuiter les tables de routage par défaut ; `MSG_PEEK` consulte les données sans les lire.

Ces primitives peuvent être utilisées en mode connecté à la place de `read` et `write` ou en mode pseudo-connecté à la place de `recvfrom` et `sendto`.

6.11 Primitives de haut niveau

L'exemple du client-serveur universel est suffisamment fréquent pour que la bibliothèque `Unix` fournisse des fonctions de plus haut niveau permettant d'établir et d'utiliser un service de façon presque transparente.

```
val open_connection : sockaddr -> in_channel * out_channel
val shutdown_connection : Pervasives.in_channel -> unit
```

La fonction `open_connection` ouvre une prise à l'adresse reçue en argument et crée une paire de tampons (de la bibliothèque `Pervasives`) d'entrée-sortie sur cette prise. La communication avec le serveur se fait donc en écrivant les requêtes dans le tampon ouvert en écriture et en lisant les réponses dans celui ouvert en lecture. Comme les écritures sont temporisées, il faut vider le tampon pour garantir qu'une requête est émise dans sa totalité. Le client peut terminer

la connexion brutalement en fermant l'un ou l'autre des deux canaux (ce qui fermera la prise) ou plus "poliment" par un appel à `shutdown_connection`. (Si le serveur ferme la connexion, le client s'en apercevra lorsqu'il recevra une fin de fichier dans le tampon ouvert en lecture.)

De façon symétrique, un service peut également être établi par la fonction `establish_server`.

```
val establish_server :
  (in_channel -> out_channel -> unit) -> sockaddr -> unit
```

Cette primitive prend en argument une fonction f, responsable du traitement des requêtes, et l'adresse de la prise sur laquelle le service doit être établi. Chaque connexion au serveur crée une nouvelle prise (comme le fait la fonction `accept`) ; après avoir été cloné, le processus fils créé une paire de tampons d'entrée-sortie (de la bibliothèque `Pervasives`) qu'il passe à la fonction f pour traiter la connexion. La fonction f lit les requêtes dans dans le tampon ouvert en lecture et répond au client dans celui ouvert en écriture. Lorsque le service est rendu (c'est-à-dire lorsque f a terminé), le processus fils ferme la prise et termine. Si le client ferme la connexion gentiment, le fils recevra une fin de fichier sur le tampon ouvert en lecture. Si le client le fait brutalement, le fils peut recevoir un `SIGPIPE` lorsqu'il essayera de d'écrire sur la prise fermée. Quant au père, il a déjà sans doute servi une autre requête ! La commande `establish_server` ne termine pas normalement, mais seulement en cas d'erreur (par exemple, du runtime OCaml ou du système pendant l'établissement du service).

6.12 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]

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

Les nombres 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

2. Les RFC sont disponibles par FTP anonyme sur de nombreux sites. En France : `ftp.inria.fr`, dans le répertoire `rfc`. Le site de référence étant `http://www.faqs.org/rfcs/`.

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 OCaml facilite grandement ce travail de mise en forme des données (travail souvent appelé *marshaling* ou encore *sérialisation* dans la littérature) en fournissant deux primitives générales de transformation d'une valeur OCaml quelconque en suite d'octets, et réciproquement :

```
val output_value : out_channel -> 'a -> unit
val 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 OCaml 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* différentes.

Exemple: Si X-window était écrit en OCaml, 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) array * drawable * graphic_context
                                   * poly_shape * coord_mode
  | GetAtomNameReq of atom
  | ...
and reply =
    ...
  | GetAtomNameReply of string
  | ...
```

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 requests (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 [13] pour une présentation plus détaillée.

Appel de procédure distant (*R*emote *P*rocedure *C*all)

Une autre application typique de ce type de protocole est l'appel de procédure distant, couramment appelé RPC (pour ≪Remote Procedure Call≫). Un utilisateur sur une Machine A veut exécuter un programme f sur une machine B. Ce n'est évidemment pas possible directement. Ce pourrait être programmé au cas par cas, en passant par le système pour ouvrir une connexion vers la machine B exécuter l'appel, relayer la réponse vers la machine A puis l'utilisateur.

En fait, comme c'est une situation typique, il existe un service RPC qui fait cela. C'est un client-serveur (client sur la machine A, serveur sur la machine B, dans notre exemple) qui reçoit des requêtes d'exécution sur une machine distante (B) de la part d'un utilisateur, se connecte

[pdf]

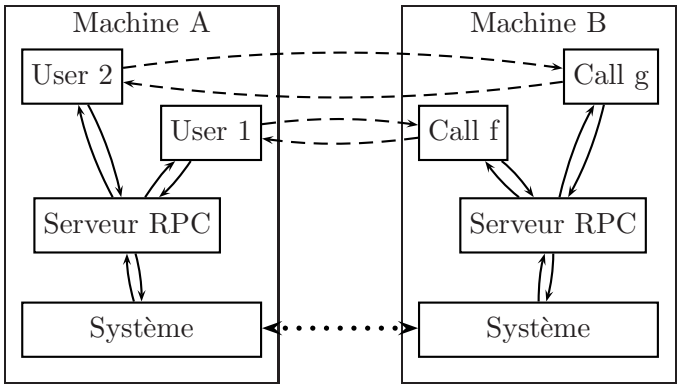


FIGURE 6.2 – Remote Procedure Call

au serveur RPC sur la machine distante B qui exécute l'appel f et retourne la réponse au client RPC A qui a son tour la renvoie à l'utilisateur. L'intérêt est qu'un autre utilisateur peut appeler un autre programme sur la machine B (ou une autre) en passant par le même serveur RPC. Le travail a donc été partagé par le service RCP installé sur les machines A et B.

Du point de vue du programme utilisateur, tout se passe virtuellement comme s'il faisait un simple appel de fonction (flèches hachurées).

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
HTTP-1.0 (HyperText Transfert Protocol)	RFC 1945	navigation sur la toile
HTTP-1.1 (HyperText Transfert Protocol)	RFC 2068	navigation sur la toile

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 du client vers le serveur, et sont donc tapées par l'utilisateur. Les lignes précédées par ← vont du serveur vers le client.

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.).

```
    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
→   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.

6.13 Exemple complet : requêtes http

Le protocole HTTP (HyperText Transfert Protocol) est utilisé essentiellement pour lire des documents sur la fameuse "toile". Ce domaine est une niche d'exemples client-serveur : entre la lecture des pages sur la toile ou l'écriture de serveurs, les relais se placent en intermédiaires, serveurs virtuels pour le vrai client et clients par délégation pour le vrai serveur, offrant souvent au passage un service additionnel tel que l'ajout de caches, de filtres, *etc.*

Il existe plusieurs versions du protocole HTTP. Pour aller plus rapidement à l'essentiel, à savoir l'architecture d'un client ou d'un relais, nous utilisons le protocole simplifié, hérité des toutes premières versions du protocole. Même s'il fait un peu poussiéreux, il reste compris par la plupart des serveurs. Nous décrivons à la fin une version plus moderne et plus expressive mais aussi plus complexe, qui est indispensable pour réaliser de vrais outils pour explorer la toile. Cependant, nous laisserons la traduction des exemples en exercices.

La version 1.0 du protocole `http` décrite dans la norme RFC 1945 [4] permet les requêtes simplifiées de la forme :

`GET` SP *Request-URI* CRLF

4. La description complète peut-être trouvée à l'URL `http://www.doclib.org/rfc/rfc1945.html`.

où SP représente un espace et CRLF la chaîne de caractères `\r\n` (RETURN suivi de LINEFEED). La réponse à une requête simplifiée est également simplifiée : le contenu de l'url est envoyé directement, sans entête, et la fin de la réponse est signalée par la fin de fichier, qui termine donc la connexion. Cette forme de requête, héritée du protocole 0.9, limite de fait la connexion à la seule requête en cours.

Récupération d'une url

Nous proposons d'écrire une commande `geturl` qui prend un seul argument, une URL, recherche sur la toile le document qu'elle désigne et l'affiche.

La première tâche consiste à analyser l'URL pour en extraire le nom du protocole (ici nécessairement `http`) l'adresse du serveur, le port optionnel et le chemin absolu du document sur le serveur. Pour ce faire nous utilisons la bibliothèque d'expressions régulières `Str`. Nous passons rapidement sur cette partie du code peu intéressante, mais indispensable.

```
1  open Unix;;
2
3  exception Error of string
4  let error err mes = raise (Error (err ^ ": " ^ mes));;
5  let handle_error f x = try f x with Error err -> prerr_endline err; exit 2
6
7  let default_port = "80";;
8
9  type regexp = { regexp : Str.regexp; fields : (int * string option) list; }
10 let regexp_match r string =
11   let get (pos, default) =
12     try Str.matched_group pos string
13     with Not_found ->
14       match default with Some s -> s
15       | _ -> raise Not_found in
16   try
17     if Str.string_match r.regexp string 0 then
18       Some (List.map get r.fields)
19     else None
20   with Not_found -> None;;
21
22 let host_regexp =
23   { regexp = Str.regexp "\\([^/:]*\\)\\(:\\([0-9]+\\)\\)?";
24     fields = [ 1, None; 3, Some default_port; ] };;
25 let url_regexp =
26   { regexp = Str.regexp "http://\\([^/:]*\\(:[0-9]+\\)?\\)\\(/.*\\)";
27     fields = [ 1, None; 3, None ] };;
28
29 let parse_host host =
30   match regexp_match host_regexp host with
31     Some (host :: port :: _) -> host, int_of_string port
32   | _ -> error host "Ill formed host";;
33 let parse_url url =
34   match regexp_match url_regexp url with
35     Some (host :: path :: _) -> parse_host host, path
```

```
36    | _ -> error url "Ill formed url";;
```

Nous pouvons maintenant nous attaquer à l'envoi de la requête qui, dans le protocole simplifié, est une trivialité.

```
37  let send_get url sock =
38    let s = Printf.sprintf "GET %s\r\n" url in
39    ignore (write sock s 0 (String.length s));;
```

Remarquons que l'url peut ne contenir que le chemin sur le serveur, ou bien être complète, incluant également le port et l'adresse du serveur.

La lecture de la réponse est encore plus facile, puisque le document est simplement envoyé comme réponse, sans autre forme de politesse. Lorsque la requête est erronée, un message d'erreur est encodé dans un document HTML. Nous nous contentons ici de faire suivre la réponse sans distinguer si elle indique une erreur ou correspond au document recherché. La transmission utilise la fonction de bibliothèque `Misc.retransmit`. Le cœur du programme établit la connexion avec le serveur.

```
40  let get_url proxy url fdout =
41    let (hostname, port), path =
42      match proxy with
43        None -> parse_url url
44      | Some host -> parse_host host, url in
45    let hostaddr =
46      try inet_addr_of_string hostname
47      with Failure _ ->
48        try (gethostbyname hostname).h_addr_list.(0)
49        with Not_found -> error hostname "Host not found" in
50    let sock = socket PF_INET SOCK_STREAM 0 in
51    Misc.try_finalize
52      begin function () ->
53        connect sock (ADDR_INET (hostaddr, port));
54        send_get path sock;
55        retransmit sock fdout
56      end ()
57      close sock;;
```

Nous terminons, comme d'habitude, par l'analyse de la ligne de commande.

```
58  let geturl () =
59    let len =  Array.length Sys.argv in
60    if len < 2 then
61      error "Usage:" (Sys.argv.(0) ^ " [ proxy [:<port>] ] <url>")
62    else
63      let proxy, url =
64        if len > 2 then Some Sys.argv.(1), Sys.argv.(2)
65        else None, Sys.argv.(1) in
66      get_url proxy url stdout;;
67
68  handle_unix_error (handle_error geturl) ();;
```

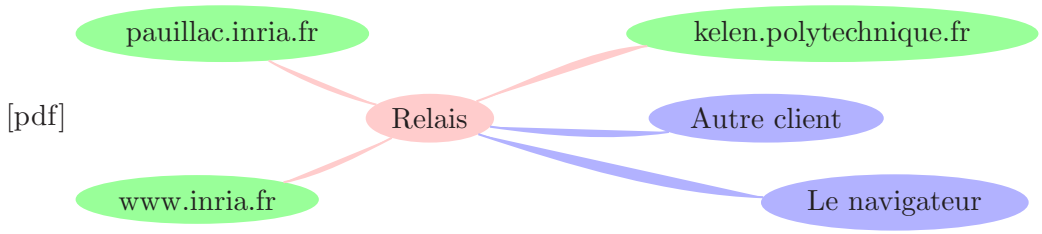


FIGURE 6.3 – Relais HTTP

Relais HTTP

Nous nous proposons maintenant d'écrire un relais HTTP (*proxy* en anglais), c'est-à-dire un serveur de requêtes HTTP qui permet de traiter toutes les requêtes HTTP en les redirigeant vers la machine destinataire (ou un autre relais...) et fait suivre les réponses vers la machine appelante. Nous avons schématisé le rôle d'un relais dans la figure 6.3. Lorsqu'un client HTTP utilise un relais, il adresse ses requêtes au relais plutôt que de les adresser directement aux différents serveurs HTTP localisés un peu partout dans le monde. L'avantage du relais est multiple. Un relais peut mémoriser les requêtes les plus récentes ou les plus fréquentes au passage pour les resservir ultérieurement sans interroger le serveur, soit pour ne pas le surcharger, soit en l'absence de connexion réseau. Un relais peut aussi filtrer certaines pages (retirer la publicité ou les images, *etc.*). L'utilisation d'un relais peut aussi simplifier l'écriture d'une application en lui permettant de ne plus voir qu'un seul serveur pour toutes les pages du monde.

La commande `proxy` lance le serveur sur le port passé en argument, ou s'il est omis, sur le port par défaut du service HTTP. Nous récupérons bien entendu le code réalisé par la fonction `get_url` (nous supposons que les fonctions ci-dessus, hormis le lancement de la commande, sont disponibles dans un module `Url`). Il ne reste qu'à écrire l'analyse des requêtes et mettre en place le serveur.

```
1  open Unix
2  open Url
3  let get_regexp =
4    { regexp = Str.regexp "^[Gg][Ee][Tt][ \t]+\\(.*[^ \t]\\)[ \t]*\r";
5      fields = [ 1, None ] }
6  let parse_request line =
7    match regexp_match get_regexp line  with
8    | Some (url :: _) -> url
9    | _ -> error line "Ill formed request"
```

Nous allons établir le service avec la commande `establish_server`. Il suffit donc de définir le traitement d'une connexion.

```
10 let proxy_service (client_sock, _) =
11   let service() =
12     try
13       let in_chan = in_channel_of_descr client_sock in
14       let line = input_line in_chan in
15       let url = parse_request line in
16       get_url None url client_sock
17     with End_of_file ->
18       error "Ill formed request" "End_of_file encountered" in
19   Misc.try_finalize
```

```
20      (handle_error service) ()
21      close client_sock
```

Le reste du programme n'a plus qu'à établir le service.

```
22  let proxy () =
23    let http_port =
24      if Array.length Sys.argv > 1 then
25        try int_of_string Sys.argv.(1)
26        with Failure _ -> error Sys.argv.(1) "Incorrect port"
27      else
28        try (getservbyname "http" "tcp").s_port
29        with Not_found -> error "http" "Unknown service" in
30    let treat_connection s = Misc.double_fork_treatment s proxy_service in
31    let addr = ADDR_INET(inet_addr_any, http_port) in
32    Misc.tcp_server treat_connection addr;;
33
34  handle_unix_error (handle_error proxy) ();;
```

Le Protocole HTTP/1.1

Les requêtes simplifiées obligent à créer une connexion par requête, ce qui est inefficace, car il est fréquent que plusieurs requêtes se suivent sur le même serveur (par exemple, le chargement d'une page WEB qui contient des images va entraîner dans la foulée le chargement des images correspondantes). Le temps d'établissement de la connexion peut facilement dépasser le temps passé à traiter la requête proprement dite. Nous verrons dans le chapitre 7 comment réduire celui-ci en faisant traiter les connexions par des coprocessus plutôt que par des processus. Nous proposons dans les exercices ci-dessous l'utilisation du protocole `HTTP/1.1`[5] qui utilise des requêtes complexes permettant de servir plusieurs requêtes par connexion[6].

Dans les requêtes complexes, le serveur précède chaque réponse par une entête indiquant le format de la réponse et le cas échéant la taille du document transmis. La fin du document n'est plus indiquée par une fin de fichier, puisqu'elle peut être déduite de la taille. La connexion peut ainsi rester ouverte pour servir d'autres requêtes.

Celles-ci sont de la forme suivante :

`GET` SP *Uri* SP `HTTP/1.1` CRLF *Header* CRLF

L'entête *Header* définit une suite de paires champ-valeur avec la syntaxe suivante :

field : *value* CRLF

Des espaces superflus sont également permis autour du séparateur ":". En fait, un espace SP peut toujours être remplacé par une tabulation ou une suite d'espaces. Les champs de l'entête peuvent également s'étendre sur plusieurs lignes : dans ce cas et dans ce cas uniquement le lexème de fin de ligne CRLF est immédiatement suivi d'un espace SP. Enfin, majuscules et minuscules sont équivalentes dans les mots-clés des champs, ainsi que dans les valeurs de certains champs composés de listes de mots-clé.

Selon le type de requête, certains champs sont obligatoires, d'autres sont optionnels. Par exemple, une requête `GET` comporte forcément un champ qui indique la machine destinataire :

`Host` COLON Hostname CRLF

5. La description complète peut-être trouvée à l'URL `http://www.doclib.org/rfc/rfc2068.html`.

6. Le protocole HTTP/1.0 permet déjà ce type de requêtes en plus des requêtes simplifiées, mais nous préférons décrire le protocole HTTP/1.1 qui traite exclusivement des requêtes complexes.

Pour ce type requête, on peut aussi demander, en utilisant le champ optionnel `If-Modified` que le document ne soit retourné que s'il a été modifié depuis une certaine date.

`If-Modified` COLON Date CRLF

Le nombre de champs du *Header* n'est donc pas fixé par avance mais indiqué par la fin de l'entête qui consiste en une ligne réduite aux seuls caractères CRLF.

Voici une requête complète (toutes les lignes se terminant par le caractère `\n` laissé implicite et qui suit immédiatement le `\r`) :

```
GET /~remy/ HTTP/1.1\r
Host:pauillac.inria.fr\r
\r
```

Une réponse à une requête complexe est également une réponse complète. Elle comporte une ligne de statut, une entête, puis le corps de la réponse, le cas échéant.

`HTTP/1.0` SP *status* SP *message* CRLF *Header* CRLF *Body*

Les champs de l'entête d'une réponse ont une syntaxe analogue à celle d'une requête mais les champs permis et obligatoires sont différents (ils dépendent du type de la requête ou du statut de la réponse—voir la documentation complète du protocole).

Le corps de la réponse *Body* peut-être vide, transmis en un seul bloc, ou par tranches. Dans le second cas, l'entête comporte un champ `Content-Length` indiquant le nombre d'octets en notation décimale ASCII. Dans le troisième cas, l'entête comporte une champ `Transfer-Encoding` avec la valeur `chuncked`. Le corps est alors un ensemble de tranches et se termine par une tranche vide. Une tranche est de la forme :

Size [SEMICOLON *arg*] CRLF *Chunk* CRLF

où *Size* est la taille de la tranche en notation hexadécimale (la partie entre "[" et "]" est optionnelle et peut être ignorée ici) et *Chunk* est une partie du corps de la réponse de la taille indiquée. La dernière tranche de taille nulle est toujours de la forme suivante :

`0` CRLF *Header* CRLF CRLF

Enfin, le corps de la réponse *Body* est vide lorsque la réponse n'est pas tranchée et ne contient pas de champ `Content-Length` (par exemple, une requête de type `HEAD` ne répond que par une entête).

Voici un exemple de réponse :

```
HTTP/1.1 200 OK\r
Date: Sun, 10 Nov 2002 09:14:09 GMT\r
Server: Apache/1.2.6\r
Last-Modified: Mon, 21 Oct 2002 13:06:21 GMT\r
ETag: "359-e0d-3db3fbcd"\r
Content-Length: 3597\r
Accept-Ranges: bytes\r
Content-Type: text/html\r
\r
<html>
...
</html>
```

Le statut `200` indique que la requête a réussi. Un statut `301` ou `302` signifie que l'URL a été redirigée vers une autre URL définie dans le champ `Location` de la réponse. Les statuts de la forme `400`, `401`, *etc.* indique des erreurs dans la forme ou l'aboutissement de la requête et ceux de la forme `500`, `501`, *etc.* des erreurs, plus grave, dans le traitement de la requête.

Exercice 15 *Écrire un relais qui fonctionne avec le protocole HTTP/1.1.* □

Exercice 16 *Ajouter un cache au relais : les pages sont sauvegardées sur le disque. Lorsqu'une page demandée est disponible dans le cache, la page du cache est servie, sauf si elle est trop ancienne, auquel cas le serveur est interrogé (et le cache est mis à jour).* □

Exercice 17 *Écrire un programme* `wget` *telle que* `wget` $u_1 \ \ldots \ u_n$ *effectue les requêtes* u_i *et sauve les réponses dans des fichiers* `./`m_i/p_i *où* m_i *et* p_i *sont respectivement le nom de la machine et le chemin absolu de la requête* u_i*. On profitera du protocole complet pour n'effectuer qu'une seule connexion sur la machine* m *lorsque celle-ci est la même pour plusieurs requêtes consécutives. De plus, on suivra une URL lorsque celle-ci est une redirection temporaire ou définitive. On pourra ajouter les options suivantes :*

- `-N` *pour ne récupérer l'URL que si le fichier* `./`m_i/u_i *n'existe pas ou est plus ancien que l'URL.*
- `-r` *pour récupérer récursivement toutes les URLs contenues dans les réponses qui sont des documents au format HTML.*

□

Chapitre 7

Les coprocessus

Un *coprocessus*, aussi appelé *processus léger* et *thread* en anglais, est un fil d'exécution d'un programme pouvant s'exécuter en parallèle avec d'autres coprocessus du même programme.

Ce chapitre décrit les appels systèmes Unix permettant à un programme de créer des coprocessus et de les synchroniser à l'aide de verrous, de conditions ou d'événements. Il s'appuie sur la bibliothèque OCaml `Thread` et les bibliothèques `Mutex`, `Condition`. Il présente également la communication par événements synchrones fournie par la bibliothèque et `Event`.

7.1 Généralités

La création d'un coprocessus est très différente de l'opération `fork` qui crée une copie du processus courant (donc une copie du programme) : les espaces mémoire du père et du fils sont totalement disjoints après l'exécution de `fork` et les processus ne peuvent communiquer que par un appel système (comme écrire ou lire dans un fichier ou dans un tuyau).

Au contraire, tous les coprocessus d'un même programme partagent le même espace d'adressage. Les seules données qui les différencient et qui ne sont pas partagées sont leur identité et leur pile d'exécution, ainsi que quelques informations pour le système (masque des signaux, état des verrous et conditions, *etc.*). De ce point de vue, les coprocessus ressemblent aux coroutines. Les coprocessus d'un même programme forment un groupe : ils sont tous traités de la même façon, sauf un, le coprocessus principal, qui a été créé au démarrage du programme et dont la terminaison entraîne l'arrêt de tous les autres coprocessus et l'arrêt du programme. Lorsque nous parlons de plusieurs coprocessus, nous considérons implicitement qu'ils s'agit des coprocessus d'un même programme.

À la différence des coroutines, qui ne peuvent pas s'exécuter en parallèle —l'une devant passer explicitement la main avant que l'autre ne puisse s'exécuter à son tour— les coprocessus s'exécutent en parallèle ou du moins tout se passe comme s'ils le faisaient, car ils peuvent être interrompus de façon préemptive par le système pour donner la main à d'autres coprocessus. De ce point de vue, les coprocessus ressemblent aux processus.

Le partage de l'espace mémoire permet aux coprocessus de communiquer directement entre eux par leur mémoire commune. Bien qu'en principe, deux processus n'aient pas besoin pour cela de passer par le système d'exploitation, le fait qu'ils s'exécutent en parallèle les oblige à se synchroniser au moment de la communication (l'un devant finir d'écrire avant que l'autre ne commence à lire), ce qui nécessite, en général, de passer par le système d'exploitation. La synchronisation entre coprocessus, quasiment indispensable pour toute communication, reste souvent une partie difficile de la programmation avec des coprocessus. Elle peut se faire à l'aide de verrous et de conditions, ou bien de façon plus évoluée, en communiquant par des événements.

Le gain à l'utilisation des coprocessus par rapport aux processus reste le coût moindre à leur création et la possibilité d'échanger de grosses structures de données sans recopie, simplement en s'échangeant un pointeur. Inversement, l'utilisation des coprocessus a un coût qui est de devoir bien gérer la synchronisation entre eux, y compris en cas d'erreur fatale dans l'un d'eux. En particulier un coprocesseur doit bien faire attention de libérer ses verrous et de préserver ses invariant avant de s'arrêter. Aussi, on pourra préférer les processus aux coprocessus lorsqu'on ne profite pas réellement des avantages de ces derniers.

Mise en œuvre en OCaml Pour compiler une application utilisant des coprocessus natifs il faut faire :

```
ocamlc -thread unix.cma threads.cma -o prog mod1.ml mod2.ml mod3.ml
ocamlopt -thread unix.cmxa threads.cmxa -o prog mod1.ml mod2.ml mod3.ml
```

Si votre installation ne supporte pas les coprocessus natifs, vous pouvez voir à la section 7.8 ou dans le manuel comment utiliser les coprocessus simulés. Le discours et les exemples de ce chapitre supposent des coprocessus natifs et ne s'applique pas, en général, aux coprocessus simulés.

7.2 Création et terminaison des coprocessus

Les fonctions décrites dans cette section sont définies dans le module `Thread`.

L'appel système `create` f a crée un nouveau coprocessus dans lequel l'application de fonction f a est exécutée. La création d'un coprocessus retourne à l'appelant une poignée sur le coprocessus fraîchement créé qui peut être utilisée pour son contrôle.

```
val Thread.create : ('a -> 'b) -> 'a -> t
```

Le calcul s'exécute de façon concurrente avec les calculs effectués par les autres coprocessus du programme. Le coprocessus termine lorsque l'application f a termine. Le résultat du calcul est simplement ignoré. Si le coprocessus termine par une exception non rattrapée, celle-ci n'est pas propagée dans le coprocessus appelant ou principal mais simplement ignorée après affichage d'un message dans la sortie d'erreur. (Les autres coprocessus ont a priori évolué indépendemment et ne seraient pas en mesure de recevoir l'exception.)

Un coprocessus peut aussi se terminer prématurément en appelant la fonction `exit` (de la bibliothèque `Thread`), à ne pas confondre avec la fonction de même nom de la bibliothèque `Pervasives` qui termine le programme tout entier, *i.e.* tous ses coprocessus. Le coprocessus principal d'un programme, qui est celui qui a lancé le premier d'autres coprocessus, exécute implicitement la fonction `Pervasives.exit` lorsqu'il termine.

Si un coprocessus termine avant le coprocessus principal, alors il est désalloué immédiatement, et ne devient pas un fantôme comme dans le cas d'un processus Unix créé par `fork`. La bibliothèque OCaml se charge de cette gestion.

Nous en savons déjà assez pour proposer une alternative au modèle du serveur concurrent par «fork» (ou «double fork») vu dans le cours précédent en faisant effectuer le traitement par un coprocessus plutôt que par un processus fils. Pour établir un tel serveur, il nous suffit de proposer une variante `Misc.co_treatment` de la fonction `Misc.fork_treatment` définie dans le chapitre 6.

```
let co_treatment server_sock service (client_descr, _ as client) =
  try ignore (Thread.create service client)
  with exn -> close client_descr; raise exn;;
```

Si le coprocessus a pu être créé, le traitement est entièrement pris en compte par la fonction `service` y compris la fermeture de `client_descr`. Sinon, on ferme le descripteur `client_descr` (le client est abandonné), et on laisse le programme principal traiter l'erreur.

Attention, toute la difficulté du coserveur est cachée dans la fonction `service` qui doit traiter la connexion avec robustesse et jusqu'à la déconnexion. Dans le cas du serveur concurrent où le service est exécuté par un autre processus, une erreur fatale pendant le service conduisant à la mort prématurée du service produit par défaut le bon comportement qui est de fermer la connexion, car le système ferme les descripteurs à la mort d'un processus. Il n'en est plus de même dans le cas où le service est exécuté par un coprocessus, car les descripteurs entre les différents coprocessus sont par défaut partagés et ne sont pas fermés à la mort du coprocessus. C'est donc au coprocessus de fermer ses descripteurs avant de mourir. De plus, un coprocessus ne peut pas appeler `Pervasives.exit` dans le cas d'une erreur fatale dans le traitement d'un service, car celle-ci arrêterait non seulement le service mais aussi le serveur. Appelé `Thread.exit` n'est souvent pas non plus une solution, car on risque de ne pas avoir bien désalloué les ressources ouvertes, et en particulier la connexion. Une solution consiste à lever une exception signifiant un arrêt fatal (par exemple une exception `Exit`) provoquant l'exécution du code de finalisation à sa remontée. Pour des raisons analogues, il est essentiel que le signal `sigpipe` soit dans un traitement du service par coprocessus, remplaçant l'arrêt immédiat du coprocessus par la levée d'une exception `EPIPE`.

7.3 Mise en attente

Les fonctions décrites dans cette section sont définies dans le module `Thread`.

L'appel système `join` permet à un coprocessus d'en attendre un autre.

```
val Thread.join : t -> unit
```

Le coprocessus appelant est alors suspendu jusqu'à ce que celui dont l'identité est passée en argument ait terminé son exécution. Cet appel peut également être utilisé par le coprocessus principal pour attendre que tous les autres aient retourné avant de terminer lui-même et de terminer le programme (le comportement par défaut étant de tuer les autres coprocessus sans attendre leur terminaison).

Bien que cet appel soit bloquant donc du type ≪long≫, il est relancé automatiquement à la réception d'un signal : il est effectivement interrompu par un signal, le handler est traité, puis l'appel est relancé. On ne revient donc de l'appel que quand le coprocessus à réellement terminé et l'appel ne doit jamais lever l'exception `EINTR`. Du point de vue du programmeur OCaml, cela se comporte comme si le signal était reçu au moment où l'appel retourne.

Un coprocessus ne retourne pas, car il s'exécute de façon asynchrone. Mais son action peut être observée — heureusement ! — par ses effets de bords. Par exemple, un coprocessus peut placer le résultat d'un calcul dans une référence qu'un autre coprocessus ira consulter après s'être assuré de la terminaison du calcul. Nous illustrons cela dans l'exemple suivant.

```
exception Exited
type 'a result = Value of 'a | Exception of exn
let eval f x = try Value (f x) with z -> Exception z
let coexec (f : 'a -> 'b) (x : 'a) : unit -> 'b =
  let result = ref (Exception Exited) in
  let p = Thread.create (fun x -> result := eval f x) x in
  function() ->
    match join p; !result with
    | Value v -> v
```

```
    | Exception exn -> raise exn;;

let v1 = coexec succ 4 and v2 = coexec succ 5 in v1()+v2();;
```

Le système peut suspendre un coprocessus pour donner temporairement la main à un autre ou parce qu'il est en attente d'une ressource utilisée par un de ses coprocessus (verrous et conditions, par exemple) ou par un autre processus (descripteur de fichier, par exemple). Un coprocessus peut également se suspendre de sa propre initiative. La fonction `yield` permet à un coprocessus de redonner la main prématurément (sans attendre la préemption par le système).

```
val Thread.yield : unit -> unit
```

C'est une indication pour le gestionnaire de coprocessus, mais son effet peut être nul, par exemple, si aucun coprocessus ne peut s'exécuter immédiatement. Le système peut donc décider de redonner la main au même coprocessus.

Inversement, il n'est pas nécessaire d'exécuter `yield` pour permettre à d'autres coprocessus de s'exécuter, car le système se réserve le droit d'exécuter lui-même la commande `yield` à tout moment. En fait, il exerce ce droit à intervalles de temps suffisamment rapprochés pour permettre à d'autres coprocessus de s'exécuter et donner l'illusion à l'utilisateur que les coprocessus s'exécutent en parallèle, même sur une machine mono-processeur.

Exemple: On peut reprendre et modifier l'exemple 3.3 pour utiliser des coprocessus plutôt que des processus.

```
1  let rec psearch k cond v =
2    let n = Array.length v in
3    let slice i = Array.sub v (i * k) (min k (n - i * k)) in
4    let slices = Array.init (n/k) slice in
5    let found = ref false in
6    let pcond v = if !found then Thread.exit(); cond v in
7    let search v = if simple_search pcond v then found := true in
8    let proc_list = Array.map (Thread.create search) slices in
9    Array.iter Thread.join proc_list;
10   !found;;
```

La fonction `psearch` k f v recherche avec k coprocessus en parallèle une occurrence dans le tableau satisfaisant la fonction f. La fonction `pcond` permet d'interrompre la recherche en cours dès qu'une réponse a été trouvée. Tous les coprocessus partagent la même référence `found` : ils peuvent donc y accéder de façon concurrente. Il n'y a pas de section critique entre les différents coprocessus car s'ils écrivent en parallèle dans cette ressource, ils écrivent la même valeur. Il est important que les coprocessus n'écrivent pas le résultat de la recherche quand celui-ci est faux ! par exemple le remplacement de la ligne 7 par

```
let search v = found := !found && simple_search pcond v
```

ou même :

```
let search v = let r = simple_search pcond v in found := !found && r
```

serait incorrect.

la recherche en parallèle est intéressante même sur une machine mono-processeur si la comparaison des éléments peut être bloquée temporairement (par exemple par des accès disques ou, mieux, des connexions réseau). Dans ce cas, le coprocessus qui effectue la recherche passe la main à un autre et la machine peut donc continuer le calcul sur une autre partie du tableau et revenir au coprocessus bloqué lorsque sa ressource sera libérée.

L'accès à certains éléments peut avoir une latence importante, de l'ordre de la seconde s'il faut récupérer de l'information sur le réseau. Dans ce cas, la différence de comportement entre une recherche séquentielle et une recherche en parallèle devient flagrante.

Exercice 18 *Paralléliser le tri rapide (quicksort) sur des tableaux.* (Voir le corrigé) □

Les autres formes de suspension sont liées à des ressources du système d'exploitation. Un coprocessus peut se suspendre pendant un certain temps en appelant `delay` s. Une fois s secondes écoulées, il pourra être relancé.

```
val Thread.delay : float -> unit
```

Cette primitive est fournie pour des raisons de portabilité avec les coprocessus simulés, mais `Thread.delay t` est simplement une abréviation pour `ignore (Unix.select [] [] [] t)`. Cet appel, contrairement à `Thread.join`, n'est pas relancé lorsqu'il est interrompu par un signal.

Pour synchroniser un coprocessus avec une opération externe, on peut utiliser la commande `select` d'Unix. Bien entendu, celle-ci ne bloquera que le coprocessus appelant et non le programme tout entier. (Le module `Thread` redéfinit cette fonction, car dans le cas des coprocessus simulés, elle ne doit pas appeler directement cette du module `Unix`, ce qui bloquerait le programme tout entier, donc tous les coprocessus. Il faut donc utiliser `Thread.select` et non `Unix.select`, même si les deux sont équivalents dans le cas des coprocessus natifs.)

Exemple: Pour faire fonctionnner le crible d'Ératosthène avec des coprocessus plutôt que par duplication de processus Unix, il suffit de remplacer les lignes 30–41 par

```
let p = Thread.create filter (in_channel_of_descr fd_in) in
let output = out_channel_of_descr fd_out in
try
  while true do
    let n = input_int input in
    if List.exists (fun m -> n mod m = 0) first_primes then ()
    else output_int output n
  done;
with End_of_file ->
  close_out output;
  Thread.join p
```

et les lignes 46–52 par

```
let k = Thread.create filter (in_channel_of_descr fd_in) in
let output = out_channel_of_descr fd_out in
generate len output;
close_out output;
Thread.join k;;
```

Toutefois, il ne faut espérer aucun gain significatif sur cet exemple qui utilise peu de processus par rapport au temps de calcul.

7.4 Synchronisation entre coprocessus : les verrous

Les fonctions de cette section sont définies dans le module `Mutex` *(pour* `Mut`*ual* `ex`*clusion, en anglais).*

Nous avons évoqué ci-dessus un problème d'accès concurrent aux ressources mutables. En particulier, le scénario suivant illustre le problème d'accès aux ressources partagées. Considérons

	Temps			
Coprocessus p	lit k			écrit $k+1$
	⟶			
Coprocessus q		lit k	écrit $k+1$	
Valeur de c	k	k	k+1	k+1

FIGURE 7.1 – Compétition pour l'accès à une ressource partagée.

un compteur c et deux processus p et q, chacun incrémentant en parallèle le même compteur c. Supposons le scénario décrit dans la Figure 7.1. Le coprocessus p lit la valeur du compteur c, puis donne la main à q. À son tour, q lit la valeur de c puis écrit la valeur $k+1$ dans c. Le processus p reprend la main et écrit la valeur $k+1$ dans c. La valeur finale de c est donc $k+1$ au lieu de $k+2$.

Ce problème classique peut être résolu en utilisant des verrous qui empêchent l'entrelacement arbitraire de p et q.

Les verrous sont des objets partagés par l'ensemble des coprocessus d'un même programme qui ne peuvent être possédés que par un seul coprocessus à la fois. Un verrou est créé par la fonction `create`.

```
val Mutex.create : unit -> Mutex.t
```

Cette opération ne fait que construire le verrou mais ne le bloque pas. Pour prendre un verrou déjà existant, il faut appeler la fonction `lock` en lui passant le verrou en argument. Si le verrou est possédé par un autre processus, alors l'exécution du processus appelant est gelée jusqu'à ce que le verrou soit libéré. Sinon le verrou est libre et l'exécution continue en empêchant tout autre processus d'acquérir le verrou jusqu'à ce que celui-ci soit libéré. La libération d'un verrou doit se faire explicitement par le processus qui le possède, en appelant `unlock`.

```
val Mutex.lock : Mutex.t -> unit
val Mutex.unlock : Mutex.t -> unit
```

L'appel système `Mutex.lock` se comporte comme `Thread.join` vis-à-vis des signaux : si le coprocessus reçoit un signal pendant qu'il exécute `Mutex.lock`, le signal sera pris en compte (*i.e.* le runtime OCaml informé du déclanchement du signal), mais le coprocessus se remet en attente de telle façon que `Mutex.lock` ne retourne effectivement que lorsque le `lock` a effectivement été acquis et bien sûr `Mutex.lock` ne levera pas l'exception `EINTR`. Le traitement réel du signal par OCaml se fera seulement au retour de `Mutex.lock`.

On peut également tenter de prendre un verrou sans bloquer en appelant `try_lock`

```
val Mutex.try_lock : Mutex.t -> bool
```

Cette fonction retourne `true` si le verrou a pu être pris (il était donc libre) et `false` sinon. Dans ce dernier cas, l'exécution n'est pas suspendue, puisque le verrou n'est pas pris. Le coprocessus peut donc faire autre chose et éventuellement revenir tenter sa chance plus tard.

Exemple: Incrémenter un compteur global utilisé par plusieurs coprocessus pose un problème de synchronisation : les instants entre la lecture de la valeur du compteur et l'écriture de la valeur incrémentée sont dans une région critique, *i.e.* deux coprocessus ne doivent pas être en même temps dans cette région. La synchronisation peut facilement être gérée par un verrou.

```
1 type counter = { lock : Mutex.t; mutable counter : int }
2 let newcounter() = { lock = Mutex.create(); counter = 0 }
3 let addtocounter c k =
4   Mutex.lock c.lock;
```

```
5    c.counter <- c.counter + k;
6    Mutex.unlock c.lock;;
```

La seule consultation du compteur ne pose pas de problème. Elle peut être effectuée en parallèle avec une modification du compteur, le résultat sera simplement la valeur du compteur juste avant ou juste après sa modification, les deux réponses étant cohérentes.

Un motif fréquent est la prise de verrou temporaire pendant un appel de fonction. Il faut bien sûr prendre soin de le relâcher à la fin de l'appel que l'appel ait réussi ou échoué. On peut abstraire ce comportement dans une fonction de bibliothèque :

```
let run_with_lock l f x =
  Mutex.lock l; try_finalize f x Mutex.unlock l
```

Dans l'exemple précédent, on aurait ainsi pu écrire :

```
let addtocounter c =
  Misc.run_with_lock c.lock (fun k -> c.counter <- c.counter + k)
```

Exemple: Une alternative au modèle du serveur avec coprocessus est de lancer un nombre de coprocessus à l'avance qui traitent les requêtes en parallèle.

```
val tcp_farm_server :
  int -> (file_descr -> file_descr * sockaddr -> 'a) -> sockaddr -> unit
```

La fonction `tcp_farm_server` se comporte comme `tcp_server` mais prend un argument supplémentaire qui est le nombre de coprocessus à lancer qui vont chacun devenir serveurs sur la même adresse. L'intérêt d'une ferme de coprocessus est de réduire le temps de traitement de chaque connexion en éliminant le temps de création d'un coprocessus pour ce traitement, puisque ceux-ci sont créés une fois pour toute.

```
7  let tcp_farm_server n treat_connection addr =
8    let server_sock = Misc.install_tcp_server_socket addr in
9    let mutex = Mutex.create() in
10   let rec serve () =
11     let client =
12       Misc.run_with_lock mutex
13         (Misc.restart_on_EINTR accept) server_sock in
14     treat_connection server_sock client;
15     serve () in
16   for i = 1 to n-1 do ignore (Thread.create serve ()) done;
17   serve ();;
```

La seule précaution à prendre est d'assurer l'exclusion mutuelle autour de `accept` afin qu'un seul des coprocessus n'accepte une connexion au même moment. L'idée est que la fonction `treat_connection` fasse un traitement séquentiel, mais ce n'est pas une obligation et on peut effectivement combiner une ferme de processus avec la création de nouveaux coprocessus, ce qui peut s'ajuster dynamiquement selon la charge du service.

La prise et le relâchement d'un verrou est une opération peu coûteuse lorsqu'elle réussit sans bloquer. Elle est en général implémentée par une seule instruction «test-and-set» que possèdent tous les processeurs modernes (plus d'autres petits coûts induits éventuels tels que la mise à jour des caches). Par contre, lorsque le verrou n'est pas disponible, le processus doit être suspendu et reprogrammé plus tard, ce qui induit alors un coût supplémentaire significatif. Il faut donc retenir que c'est la suspension réelle d'un processus pour donner la main à un autre et non sa suspension potentielle lors de la prise d'un verrou qui est pénalisante. En conséquence, on aura

presque toujours intérêt à relâcher un verrou dès que possible pour le reprendre plus tard si nécessaire plutôt que d'éviter ces deux opérations, ce qui aurait pour effet d'agrandir la région critique et donc la fréquence avec laquelle un autre coprocessus se trouvera effectivement en compétition pour le verrou et dans l'obligation de se suspendre.

Les verrous réduisent l'entrelacement. En contrepartie, ils risquent de provoquer des situations d'interblocage. Par exemple, il y a interblocage si le coprocessus p attend un verrou v possédé par le coprocessus q qui lui-même attend un verrou u possédé par p. (Dans le pire des cas, un processus attend un verrou qu'il possède lui-même...) La programmation concurrente est difficile, et se prémunir contre les situations d'interblocage n'est pas toujours facile. Une façon simple et souvent possible d'éviter cette situation consiste à définir une hiérarchie entre les verrous et de s'assurer que l'ordre dans lequel on prendra dynamiquement les verrous respecte la hiérarchie : on ne prend jamais un verrou qui n'est pas dominé par tous les verrous que l'on possède déjà.

7.5 Exemple complet : relais HTTP

On se propose de reprendre le relais HTTP développé dans le chapitre précédent pour servir les requêtes par des coprocessus.

Intuitivement, il suffit de remplacer la fonction `establish_server` qui crée un processus clone par une fonction qui crée un coprocessus. Il faut cependant prendre quelques précautions... En effet, la contrepartie des coprocessus est qu'ils partagent tous le même espace mémoire. Il faut donc s'assurer que les coprocessus ne vont pas se "marcher sur les pieds" l'un écrasant ce que vient juste de faire l'autre. Cela se produit typiquement lorsque deux coprocessus modifient en parallèle une même structure mutable.

Dans le cas du serveur HTTP, il y a quelques changements à effectuer. Commençons par régler les problèmes d'accès aux ressources. La fonction `proxy_service` qui effectue le traitement de la connexion et décrite à la section 6.13 appelle, par l'intermédiaire des fonctions `parse_host`, `parse_url` et `parse_request`, la fonction `regexp_match` qui utilise la bibliothèque `Str`. Or, cette bibliothèque n'est pas ré-entrante (le résultat de la dernière recherche est mémorisé dans une variable globale). Cet exemple montre qu'il faut également se méfier des appels de fonctions qui sous un air bien innocent cachent des collisions potentielles. Dans ce cas, nous n'allons pas réécrire la bibliothèque `Str` mais simplement séquentialiser son utilisation. Il suffit (et il n'y a pas vraiment d'autres choix) de protéger les appels à cette bibliothèque par des verrous. Il faut quand même prendre la précaution de bien libérer le verrou au retour de la fonction, pour un retour anormal par une levée d'exception.

Pour modifier au minimum le code existant, il suffit de renommer la définition de `regexp_match` du module `Url` en `unsafe_regexp_match` puis de définir `regexp_match` comme une version protégée de `unsafe_regexp_match`.

```
let strlock = Mutex.create();;
let regexp_match r string =
  Misc.run_with_lock strlock (unsafe_regexp_match r) string;;
```

La modification à l'air minime. Il faut toutefois remarquer que la fonction `regexp_match` regroupe à la fois le filtrage de l'expression et l'extraction de la chaîne filtrée. Il eût bien sûr été incorrect de protéger individuellement les fonctions `Str.string_match` et `Str.matched_group`.

Une autre solution serait de réécrire les fonctions d'analyse sans utiliser la bibliothèque `Str`. Mais il n'y a pas de raison pour un tel choix tant que la synchronisation des primitives de la bibliothèque reste simplement réalisable et ne s'avère pas être une source d'inefficacité. Évidemment, une meilleure solution serait que la bibliothèque `Str` soit ré-entrante en premier lieu.

Les autres fonctions appelées sont bien réentrantes, notamment la fonction `Misc.retransmit` qui alloue des tampons différents pour chaque appel.

Cependant, il reste encore quelques précautions à prendre vis à vis du traitement des erreurs. Le traitement d'une connexion par coprocessus doit être robuste comme expliqué ci-dessus. En particulier, en cas d'erreur, il ne faut pas que les autres processus soit affectés, c'est-à-dire que le coprocessus doit terminer «normalement» en fermant proprement la connexion en cause et se remettre en attente d'autres connexions. Nous devons d'abord remplacer l'appel à `exit` dans `handle_error` car il ne faut surtout pas tuer le processus en cours. Un appel à `Thread.exit` ne serait pas correct non plus, car la mort d'un coprocessus ne ferme pas ses descripteurs (partagés), comme le ferait le système à la mort d'un processus. Une erreur dans le traitement d'une connexion laisserait alors la connexion ouverte. La solution consiste à lever une exception `Exit` qui permet au code de finalisation de faire ce qu'il faut. Nous devons maintenant protéger `treat_connection` pour rattraper toutes les erreurs : notamment `Exit` mais aussi `EPIPE` qui peut être levée si le client ferme prématurément sa connexion. Nous ferons effectuer ce traitement par une fonction de protection.

```
let allow_connection_errors f s =
  try f s with Exit | Unix_error(EPIPE,_,_) -> ()

  let treat_connection s =
    Misc.co_treatment s (allow_connection_errors proxy_service) in
```

Exercice 19 *Réécrire la version du proxy pour le protocole HTTP/1.1 à l'aide de coprocessus.-* □

7.6 Les conditions

Les fonctions décrites dans cette section sont définies dans le module `Condition`.

Le mécanisme de synchronisation par verrous est très simple, mais il n'est pas suffisant : les verrous permettent d'attendre qu'une donnée partagée soit libre, mais il ne permettent pas d'attendre la forme d'une donnée. Remplaçons l'exemple du compteur par une file d'attente (premier entré/premier sorti) partagée entre plusieurs coprocessus. L'ajout d'une valeur dans la queue peut être synchronisé en utilisant un verrou comme ci-dessus, car quelque soit la forme de la queue, on peut toujours y ajouter un élément. Mais qu'en est-il du retrait d'une valeur de la queue ? Que faut-il faire lorsque la queue est vide ? On ne peut pas garder le verrou en attendant que la queue se remplisse, car cela empêcherait justement un autre coprocessus de remplir la queue. Il faut donc le rendre. Mais comment savoir quand la queue ne sera plus vide, sinon qu'en testant à nouveau périodiquement ? Cette solution appelée "attente active" n'est bien sûr pas satisfaisante. Ou bien elle consomme du temps de calcul inutilement (période trop courte) ou bien elle n'est pas assez réactive (période trop longue).

Les *conditions* permettent de résoudre ce problème. Les conditions sont des objets sur lequel un coprocessus qui possède un verrou peut se mettre en attente jusqu'à ce qu'un autre coprocessus envoie un signal sur cette condition.

Comme les verrous, les conditions sont des structures passives qui peuvent être manipulées par des fonctions de synchronisation. On peut les créer par la fonction `create`.

```
val Condition.create : unit -> Condition.t
```

Un processus p qui *possède* déjà un verrou v peut se mettre en attente sur une condition c et le verrou v en appelant `wait` c v. Le processus p informe le système qu'il est en attente sur la condition c et le verrou v puis libère le verrou v et s'endort. Il ne sera réveillé par le système que

lorsqu'un autre processus q aura signalé un changement sur la condition c et lorsque le verrou v sera disponible ; le processus p tient alors à nouveau le verrou v.

```
val Condition.wait : Condition.t -> Mutex.t -> unit
```

Attention ! c'est une erreur d'appeler `Condition.wait` c v sans être en possession du verrou v. Le comportement de `Condition.wait` vis à vis des signaux est le même que pour `Mutex.lock`.

Lorsqu'un coprocessus signale un changement sur une condition, il peut demander ou bien que tous les coprocessus en attente sur cette condition soient réveillés (`broadcast`), ou bien qu'un seul parmi ceux-ci soit réveillés (`signal`).

```
val Condition.signal : Condition.t -> unit
val Condition.broadcast : Condition.t -> unit
```

L'envoi d'un signal ou d'un broadcast sur une condition ne nécessite pas d'être en possession d'un verrou (à la différence de l'attente), dans le sens où cela ne déclenchera pas une erreur «système». Cependant, cela peut parfois être une erreur de programmation.

Le choix entre le réveil d'un coprocessus ou de tous les coprocessus dépend du problème. Pour reprendre l'exemple de la file d'attente, si un coprocessus ajoute un élément dans une queue vide, il n'a pas besoin de réveiller tous les autres puisqu'un seul pourra effectivement retirer cet élément. Par contre, s'il ajoute un nombre d'éléments qui n'est pas connu statiquement ou bien qui est très grand, il doit demander le réveil de tous les coprocessus. Attention ! si l'ajout d'un élément dans une queue non vide n'envoie pas de signal, alors l'ajout d'un élément dans une queue vide doit envoyer un broadcast, car il pourrait être immédiatement suivi d'un autre ajout (sans signal) donc se comporter comme un ajout multiple. En résumé, soit chaque ajout envoie un signal, soit seul l'ajout dans une queue vide envoie un broadcast. Le choix entre ces deux stratégies est un pari sur le fait que la queue est généralement vide (première solution) ou généralement non vide (deuxième solution).

Il est fréquent qu'un coprocessus n'ait qu'une approximation de la raison pour laquelle un autre peut être en attente sur une condition. Il va donc signaler sur cette condition par excès simplement lorsque la situation *peut* avoir évolué pour un coprocessus en attente. Un processus réveillé ne doit donc pas supposer que la situation a évolué pour lui et que la condition pour laquelle il est mis en attente est maintenant remplie. Il doit en général (presque impérativement) tester à nouveau la configuration, et si nécessaire se remettre en attente sur la condition. Il ne s'agit plus cette fois-ci d'une attente active, car cela ne peut se passer que si un coprocessus à émis un signal sur la condition.

Voici une autre raison qui justifie cette discipline : lorsqu'un coprocessus vient de fournir une ressource en abondance et réveille tous les autres par un `broadcast`, rien n'empêche le premier réveillé d'être très gourmand et d'épuiser la ressource en totalité. Le second réveillé devra se rendormir en espérant être plus chanceux la prochaine fois.

Nous pouvons maintenant donner une solution concrète pour les queues partagées. La structure de `queue` définie dans le module `Queue` est enrichie avec un verrou et une condition `non_empty`.

```
1 type 'a t =
2   { queue : 'a Queue.t; lock : Mutex.t; non_empty : Condition.t }
3 let create () =
4   { queue = Queue.create();
5     lock = Mutex.create(); non_empty = Condition.create() }
```

L'ajout n'est jamais bloquant, mais il ne faut pas oublier de signaler la condition `non_empty` lorsque la liste était vide avant ajout, car il est possible que quelqu'un soit en attente sur la condition.

```
6 let add e q =
```

```
7    Mutex.lock q.lock;
8    if Queue.length q.queue = 0 then Condition.broadcast q.non_empty;
9    Queue.add e q.queue;
10   Mutex.unlock q.lock;;
```

Le retrait est un tout petit peu plus compliqué :

```
11 let take q =
12   Mutex.lock q.lock;
13   while Queue.length q.queue = 0
14   do Condition.wait q.non_empty q.lock done;
15   let x = Queue.take q.queue in
16   Mutex.unlock q.lock; x;;
```

Après avoir acquis le verrou, on peut essayer de retirer l'élément de la queue. Si la queue est vide, il faut se mettre en attente sur la condition `non_empty`. Au réveil, on essayera à nouveau, sachant qu'on a déjà le verrou.

Comme expliqué ci-dessus, le signal `Condition.broadcast q.non_empty` (8) est exécuté par un coprocessus p en étant en possession du verrou `q.lock`. Cela implique qu'un coprocessus lecteur q exécutant la fonction `take` ne peut pas se trouver entre la ligne 13 et 14 où il serait prêt à s'endormir après avoir vérifié que la queue est vide mais ne l'aurait pas encore fait : dans ce cas, le signal émis par p serait inopérant et ignoré, le coprocessus q n'étant pas encore endormi et q s'endormirait et ne serait pas réveillé par p ayant déjà émis son signal. Le verrou assure donc que soit q est déjà endormi, soit q n'a pas encore testé l'état de la queue.

Exercice 20 *Implémenter une variante dans laquelle la queue est bornée : l'ajout dans la queue devient bloquant lorsque la taille de la queue a atteint une valeur fixée à l'avance. (Dans un monde concurrent, on peut avoir besoin de ce schéma pour éviter qu'un producteur produise sans fin pendant que le consommateur est bloqué.* (Voir le corrigé) □

7.7 Communication synchrone entre coprocessus par événements

Les fonctions décrites dans cette section sont définies dans le module `Event`.

Verrous et conditions permettent ensemble d'exprimer toutes les formes de synchronisation. Toutefois, leur mise en œuvre n'est pas toujours aisée comme le montre l'exemple a priori simple des files d'attente dont le code de synchronisation s'avère a posteriori délicat.

La communication synchrone entre coprocessus par événements est un ensemble de primitives de communication de plus haut niveau qui tend à faciliter la programmation concurrente. L'ensemble des primitives de la bibliothèque `Event` a été initialement développé par John Reppy pour le langage *Standard ML*, appelé *Concurrent ML* [14]. En OCaml, ces primitives sont implantées au dessus de la synchronisation plus élémentaire par verrous et conditions. Les primitives sont regroupées dans le module `Event`.

La communication se fait en envoyant des *événements* au travers de *canaux*. Les canaux sont en quelque sorte des "tuyaux légers" : ils permettent de communiquer entre les coprocessus d'un même programme en gérant eux-mêmes la synchronisation entre producteurs et consommateurs. Un canal transportant des valeurs de type `'a` est de type `'a Event.channel`. Les canaux sont homogènes et transportent donc toujours des valeurs du même type. Un canal est créé avec la primitive `new_channel`.

```
val Event.new_channel : unit -> 'a channel
```

L'envoi ou la réception d'un message ne se fait pas directement, mais par l'intermédiaire d'un événement. Un événement élémentaire est "envoyer un message" ou "recevoir un message". On les construit à l'aide des primitives suivantes :

```
val Event.send : 'a channel -> 'a -> unit event
val Event.receive : 'a channel -> 'a event
```

Le construction d'un message n'a pas d'effet immédiat et se limite à la création d'une structure de donnée décrivant l'action à effectuer. Pour réaliser un événement, le coprocessus doit se synchroniser avec un autre coprocessus souhaitant réaliser l'événement complémentaire. La primitive `sync` permet la synchronisation du coprocessus pour réaliser l'événement passé en argument.

```
val Event.sync : 'a event -> 'a
```

Ainsi pour envoyer une valeur v sur le canal c, on pourra exécuter `sync` (`send` c v). Le coprocessus est suspendu jusqu'à ce que l'événement se réalise, c'est-à-dire jusqu'à ce qu'un autre coprocessus soit prêt à recevoir de une valeur sur le canal c. De façon symétrique, un processus peut se mettre en attente d'un message sur le canal c en effectuant `sync` (`receive` c).

Il y a compétitions entre tous les producteurs d'une part et tous les consommateurs d'autre part. Par exemple, si plusieurs coprocessus essayent d'émettre un message sur un canal alors qu'un seul est prêt à le lire, il est un clair qu'un seul producteur réalisera l'événement. Les autres resteront suspendus, sans même s'apercevoir qu'un autre a été "servi" avant eux.

La compétition peut également se produire au sein d'un même coprocessus. Deux événements peuvent être combinés par la primitive `choose` :

```
val Event.choose : 'a event list -> 'a event
```

L'événement résultant est une offre en parallèle des événements passés en arguments qui se réalisera lorsque qu'un exactement de ceux-ci se réalisera. Il bien faut distinguer là l'offre d'un événement de sa réalisation. L'appel `sync` (`choose` e_1 e_2) se synchronise en offrant au choix deux événements e_1 et e_2, mais un seul des deux événements sera effectivement réalisé (l'offre de l'autre événement sera simultanément annulée). La primitive `wrap_abort` permet à un événement d'agir lorsqu'il est ainsi annulé.

```
Event.wrap_abort : 'a event -> (unit -> unit) -> 'a event
```

L'appel `wrap_abort` e f construit un événement e' équivalent à e mais tel que si e n'est pas réalisé après sa synchronisation alors la fonction f est exécutée (cela n'a d'intérêt que si e' fait parti d'un événement complexe).

Un coprocessus peut proposer de réaliser un événement sans se bloquer (un peu à la manière de `Mutex.try_lock`).

```
val Event.poll : 'a event -> 'a option
```

L'appel `Event.pool` e va offrir l'événement e mais si celui-ci ne peut pas être immédiatement réalisé, il retire l'offre au lieu de se bloquer et tout se passera comme s'il n'avait rien fait (ou plus exactement comme si l'expression `Event.pool` e avait été remplacée par la valeur `None`). Par contre, si l'événement peut se réaliser immédiatement, alors tout se passera comme si le coprocessus avec effectué `Event.sync` e, sauf que c'est la valeur `Some` v plutôt que v qui est retournée.

Exemple: Dans l'exemple 7.3 du crible d'Ératosthène la communication entre les différents coprocessus s'effectue par des tuyaux comme dans le programme d'origine, donc en utilisant la mémoire du système (le tuyau) comme intermédiaire. On peut penser qu'il serait plus efficace de communiquer directement en utilisant la mémoire du processus. Une solution simple consisterait à remplacer le tuyau par un canal sur lequel sont envoyés les entiers.

Passer des entiers dans le canal n'est pas suffisant car il faut aussi pouvoir détecter la fin du flux. Le plus simple est donc de passer des éléments de la forme `Some` n et de terminer en envoyant la valeur `None`. Pour simplifier et souligner les changements en les minimisant, nous

allons récupérer le code de l'exemple 5.2. Pour cela, nous simulons les tuyaux et les fonctions de lecture/écriture sur un tuyau par des canaux et de fonctions de lecture/écriture sur les canaux.

Il suffit de reprendre la version précédente du programme et de modifier les fonctions d'entrée-sortie pour qu'elles lisent et écrivent dans un canal plutôt que dans un tampon d'entrée-sortie de la bibliothèque `Pervasives`, par exemple en insérant à la ligne 2 le code suivant :

```
2  let pipe () = let c = Event.new_channel() in c, c
3  let out_channel_of_descr x = x
4  let in_channel_of_descr x = x
5
6  let input_int chan =
7    match Event.sync (Event.receive chan) with
8      Some v -> v
9    | None -> raise End_of_file
10 let output_int chan x = Event.sync (Event.send chan (Some x))
11 let close_out chan = Event.sync (Event.send chan None);;
```

Toutefois, si l'on compare l'efficacité de cette version avec la précédente, on trouve qu'elle est deux fois plus lente. La communication de chaque entier requiert une synchronisation entre deux coprocessus donc plusieurs appels systèmes pour prendre et relâcher le verrou. Au contraire, la communication par tuyau utilise des entrées/sorties temporisées qui permettent d'échanger plusieurs milliers d'entiers à chaque appel système.

Pour être juste, il faudrait donc également fournir une communication temporisée sur les canaux en utilisant le canal seulement pour échanger un paquet d'entiers. Le fils peut accumuler les résultats dans une queue qu'il est seul à posséder, donc dans laquelle il peut écrire sans se synchroniser. Quand la queue est pleine ou sur demande explicite, il la vide en se synchronisant sur le canal. Le père à lui même sa propre queue qu'il reçoit en se synchronisant et qu'il vide petit à petit.

Voici une solution (qui remplace des lignes 2–11 ci-dessus) :

```
2  type 'a buffered =
3      { c : 'a Queue.t Event.channel; mutable q : 'a Queue.t; size : int }
4  let pipe () = let c = Event.new_channel() in c, c;;
5
6  let size = 1024;;
7  let out_channel_of_descr chan =
8    { c = chan; q = Queue.create(); size = size };;
9  let in_channel_of_descr = out_channel_of_descr;;
10
11 let input_int chan =
12   if Queue.length chan.q = 0 then begin
13     let q = Event.sync (Event.receive chan.c) in
14     if Queue.length q > 0 then chan.q <- q
15     else raise End_of_file
16   end;
17   Queue.take chan.q;;
18
19 let flush_out chan =
20   if Queue.length chan.q > 0 then Event.sync (Event.send chan.c chan.q);
21   chan.q <- Queue.create();;
22
23 let output_int chan x =
```

```
24    if Queue.length chan.q = size then flush_out chan;
25    Queue.add x chan.q
26
27  let close_out chan =
28    flush_out chan;
29    Event.sync (Event.send chan.c chan.q);;
```

Cette version permet de retrouver une efficacité comparable à la version avec tuyau (mais pas meilleure).

Si l'on compare avec la version d'origine avec processus et tuyaux, il y a deux sources potentielles de gain : d'une part les coprocessus sont plus sobres et coûtent moins cher à leur lancement. D'autre part, la communication par le canal se contente de passer un pointeur sans recopie. Mais ces gains ne sont pas perceptibles ici, car le nombre de coprocessus créés et les structures échangées ne sont pas suffisamment grands devant le coût de l'appel système et devant les temps de calcul.

En conclusion, on peut retenir que la communication entre coprocessus a un coût qui peut aller jusqu'à celui d'un appel système (si le processus doit être suspendu) et que ce coût peut être significativement réduit en temporisant les communications pour communiquer moins souvent de plus grosses structures.

Exercice 21 *Un serveur HTTP peut être soumis à une charge élevée et par à-coups. Pour améliorer le temps de réponse, on peut raffiner l'architecture d'un serveur HTTP en gardant toujours une dizaine de coprocessus prêts à traiter de nouvelles requêtes. Cela veut dire qu'un coprocessus ne traite plus une seule requête, mais une suite potentiellement infinie de requêtes qu'il lit dans une file d'attente.*

Pour éviter l'écroulement de la machine, on peut limiter le nombre de coprocessus à une valeur raisonnable au delà laquelle le coût de la gestion des tâches dépasse la latence du service des requêtes (temps passés à attendre des données sur le disque, etc.*). Au delà, on pourra garder quelques connexions en attente d'être traitées, puis finalement refuser les connexions. Lorsque la charge diminue et que le nombre de coprocessus est au-delà de la valeur "idéale", certains se laissent mourir, les autres restent prêts pour les prochaines requêtes.*

Transformer l'exemple 7.5 pour arriver à cette architecture. □

7.8 Quelques détails d'implémentation

Implémentation des coprocessus en Unix Le système Unix n'a pas été conçu au départ pour fournir du support pour les coprocessus. Toutefois, la plupart des implémentations modernes d'Unix offrent maintenant un tel support. Malgré tout, les coprocessus restent une pièce rapportée qui est parfois apparente. Par exemple, dès l'utilisation de coprocessus il est fortement déconseillé d'utiliser `fork` autrement que pour faire `exec` immédiatement après. En effet, `fork` copie le coprocessus courant qui devient un processus estropié car s'exécutant en croyant avoir des coprocessus qui en fait n'existent pas. Le père lui continue à s'exécuter a priori normalement. Le cas particulier d'un appel à fork où le fils lance immédiatement un autre programme ne pose quant à lui pas de problème. Heureusement ! car c'est le seul moyen de lancer d'autres programmes.

Inversement, on peut faire `fork` (non suivi de `exec`) puis lancer ensuite plusieurs coprocessus dans le fils et le père, sans aucun problème.

Implémentation native et implémentation simulée en OCaml Lorsque le système d'exploitation sous-jacent possède des coprocessus, OCaml peut fournir une implémentation native

des coprocessus, en laissant le plus possible leur gestion au système d'exploitation. Chaque coprocessus réside alors dans un processus Unix différent mais partageant le même espace mémoire.

Lorsque le système ne fournit pas de support pour les coprocessus, OCaml peut les émuler. Tous les coprocessus s'exécutent alors dans le même processus Unix et leur gestion y compris leur ordonnancement est effectuée par l'environnement d'exécution d'OCaml. Toutefois, cette implémentation n'est disponible qu'avec la compilation vers du bytecode.

Le système OCaml offre une même interface programmatique pour les versions native et simulée des coprocessus. L'implémentation des coprocessus est donc dédoublée. Une implémentation pour la version émulée qui comporte son propre contrôleur de tâches et une autre implémentation qui s'appuie sur les coprocessus POSIX (1003.1c) et relève les fonctions des bibliothèques correspondantes au niveau du langage OCaml. Au passage, le langage OCaml effectue quelques tâches administratives simples et assure une interface identique avec la version émulée. Cela garantit qu'un programme compilable sur une architecture Unix reste compilable sur une autre architecture Unix. Toutefois, le fait que les processus soient émulés ou natifs peut changer la synchronisation des appels à des bibliothèques C, et ainsi changer, malgré tout, la sémantique du programme. Il faut donc prendre quelques précautions avant de considérer qu'un programme se comportera de la même façon dans les deux versions. Dans ce chapitre, le discours vaut principalement pour les deux implantations, mais rapelons qu'à défaut, nous avons pris le point de vue d'une implantation native.

Pour utiliser les coprocessus émulés, il faut passer l'option `-vmthreads` à la place de `-threads` au compilateur `ocamlc`. Cette option n'est pas acceptée par le compilateur `ocamlopt`.

Séquentialisation du code OCaml L'implémentation des coprocessus en OCaml doit faire face à une des particularités du langage OCaml qui est la gestion automatique de la mémoire et sa consommation importante de données allouées. La solution retenue, qui est la plus simple et aussi généralement la plus efficace, est de séqentialiser l'exécution du code OCaml dans tous les coprocessus : un verrou de l'environnement d'exécution empêche deux coprocessus d'exécuter du code OCaml simultanément. Cela semble contraire à l'idée même des coprocessus, mais il n'en est rien. Car le verrou est évidemment relâché avant les appels systèmes bloquants et repris au retour. D'autres coprocessus peuvent donc prendre la main à ce moment là. Un cas particulier d'appel système étant l'appel à `sched_yield` effectué à intervalles de temps réguliers pour permettre de suspendre le coprocessus en cours d'exécution et donner la main à d'autres.

Sur une machine multiprocesseur, la seule source de vrai parallélisme provient de l'exécution du code C et des appels systèmes. Sur une machine mono-processeur, le fait que le code OCaml soit séquentialisé n'est pas vraiment perceptible.

Le programmeur ne peut pas pour autant s'appuyer sur cette séquentialisation, car un coprocessus peut donner la main à un autre à peu près à n'importe quel moment. À une exception près, la séquentialisation garantit la cohérence de la mémoire : deux coprocessus ont toujours la même vision de la mémoire, sauf, peut-être, lorsqu'ils exécutent du code C. En effet, le passage du verrou implique une synchronisation de la mémoire : une lecture à une adresse effectuée par un coprocessus qui se situe dans le temps après une opération d'écriture à cette même adresse par un autre coprocessus retournera toujours la valeur fraîchement écrite, sans besoin de synchronisation supplémentaire.

Coprocessus et signaux D'une façon générale, l'utilisation des signaux est déjà délicate avec un seul coprocessus en raison de leur caractère asynchrone. Elle l'est encore plus en présence de plusieurs coprocessus car s'ajoute alors de nouvelles difficultés : à quel coprocessus doit être envoyé le signal ? À tous, au principal, ou à celui en cours d'exécution ? Que se passe-t-il si un coprocessus envoie un signal à un autre ? En fait, les coprocessus ont été implantés avant de

bien répondre à ces questions, et différentes implantations peuvent se comporter différemment par rapport aux signaux.

Les opérations `Thread.join`, `Mutex.lock`, and `Condition.wait`, bien qu'étant des appels systèmes longs, ne sont pas interruptibles par un signal. (Elle ne peuvent donc pas échouer avec l'erreur `EINTR`). Si un signal est envoyé pendant l'attente, il sera reçu et traité lorsque l'appel retournera.

La norme POSIX spécifie que le handler des signaux est partagé entre tous les coprocessus et au contraire le masque des signaux est propre à chaque coprocessus et hérité à la création d'un coprocessus. Mais le comportement des coprocessus vis-à-vis des signaux reste largement sous-spécifié et donc non portable.

Il est donc préférable d'éviter autant que possible l'utilisation de signaux asynchrones (du type `sigalrm`, `sigvtalrm`, `sigchld`, *etc.*) avec des coprocessus. Ceux-ci peuvent être bloqués et consultés avec `Thread.wait_signal`. On peut dédier un coprocessus au traitement des signaux : ne faisant que cela, il peut se mettre en attente sur la réception de signaux et entreprendre les actions nécessaires et mettre à jour certaines informations consultées par d'autres coprocessus.

De plus, les coprocessus OCaml (depuis la version 3.08) utilisent le signal `sigvtalarm` de façon interne pour effectuer la préemption des coprocessus. Ce signal est donc réservé et ne doit pas être utilisé par le programme lui-même, car il y a un risque d'interférence.

Pour aller plus loin

Nous avons montré comment utiliser les bibliothèques `Sys`, `Unix`, et `Threads` d'OCaml pour programmer des applications qui interagissent avec le système d'exploitation.

Ces bibliothèques relèvent les appels système Unix les plus importants au niveau du langage OCaml. Au passage, certains de ces appels système ont été remplacés par des fonctions de plus haut niveau, soit pour faciliter la programmation, soit pour maintenir des invariants de l'environnement d'exécution des programmes OCaml. En général, cela conduit à une économie dans l'écriture des applications.

Certaines fonctionalités du système Unix ne sont pas accessibles au travers des bibliothèques précédentes, mais il est toujours possible d'y accéder directement via du code C.

Il existe aussi une bibliothèque `Cash` dédiée à l'écriture de scripts en Ocaml. Cette bibliothèque complète la bibliothèque `Unix` dans deux directions différentes. D'une part, elle peut se voir comme une couche au dessus du module `Unix` qui offre, en plus de fonctions dédiées à l'écriture de scripts, de nombreuses variations autour des appels systèmes d'`Unix`, en particulier en ce qui concerne la gestion des processus et des tuyaux. D'autre part, elle fournit quelques accès supplémentaires au système Unix.

Bibliographie

Objective Caml

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

[2] Xavier Leroy, Didier Rémy, Jacques Garrigue, Jérôme Vouillon et Damien Doligez. *The Objective Caml system, documentation and user's manual – release 3.06.* Documentation distribuée avec le système Objective Caml par l'INRIA, août 2002. `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[3] Bruno Verlyck. *Cash, the Caml Shell – release 0.20.* Documentation distribuée avec le système Cash par l'INRIA, 2002. `http://pauillac.inria.fr/cash/`.

Programmation du système Unix

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

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

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

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

Architecture du noyau Unix

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

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

[Ste92] Richard W. Stevens. *Advanced Programming in the Unix Environment.* Addisson-Wesley, 1992.

[Ste92] Kay A. Robbins and Steven Robbins. *Practical Unix Programming. A Guide to Concurrency, Communication, and Multithreading.* Prentice Hall, 1996.

Culture générale sur les systèmes

[10] Andrew Tanenbaum. *Modern Operating Systems*, Second Edition ©2001. Prentice-Hall.

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

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

Typage des communications d'objets structurés

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

[14] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

Programmation avec coprocessus

[15] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

Applications

[16] Pierce et al. *Unison File Synchronizer. Version 2.9.1. User Manual and Reference*. Logiciel libre, disponible, à l'URL `http://www.cis.upenn.edu/~bcpierce/unison/`.

Annexe A

Corrigé des exercices

Exercice 1, page 20

```
let main() =
  let action p infos =
    let b = not (infos.st_kind = S_DIR || Filename.basename p = "CVS") in
    if b then print_endline p; b in
  let errors = ref false in
  let error (e,c,b) =
    errors:= true; prerr_endline (b ^ ": " ^ error_message e) in
  Findlib.find error action false max_int [ "." ];;
handle_unix_error main()
```

Exercice 2, page 20

Voici quelques indications : on remonte à partir de la position courante vers la racine en constituant à l'envers le chemin que l'on cherche. La racine est identifié comme l'unique répertoire nœud donc le parent est égal à lui-même (les chemins `parentdir` et `currentdir` sont équivalent lorsqu'ils sont pris relatifs à la racine). Pour trouver le nom qui désigne un répertoire r depuis son répertoire parent, il faut parcourir l'ensemble des noms du répertoire parent et identifier le fils qui désigne r.

Exercice 3, page 26

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

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

à la place de

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

Le parsing de l'option est laissé au lecteur.

Exercice 4, page 30

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
    String.blit 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
    ignore (write chan.out_fd s 0 (String.length s))
  end
  else begin
    String.blit s 0 chan.out_buffer chan.out_pos avail;
    let out_buffer_size = String.length chan.out_buffer in
    ignore (write chan.out_fd chan.out_buffer 0 out_buffer_size);
    let remaining = String.length s - avail in
    if remaining < out_buffer_size then begin
      String.blit s avail chan.out_buffer 0 remaining;
      chan.out_pos <- remaining
    end else begin
      ignore (write chan.out_fd s avail remaining);
      chan.out_pos <- 0
    end
  end;;
```

Exercice 5, page 31

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 6, page 35

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.create 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 7, page 41

Une solution assez simple consiste à simuler l'effet de l'extraction en créant dans un graphe une image de la hiérarchie des fichiers contenus dans l'archive.

```
1  type info = File | Link of string list | Dir of (string * inode) list
2  and inode = { mutable record : record option; mutable info : info;}
```

Les nœuds du système de fichier virtuel dont décrits par le type `inode`. Le champ `info` décrit le type de fichier en se limitant aux fichiers ordinaires, liens symboliques et répertoires. Les chemins sont représentés par des listes de chaînes de caractères et les répertoires par des listes qui associent un nœud à chaque nom de fichiers du répertoire. Le champ `record` le fichier associé au nœud dans l'archive. Ce champ est optionnel, car les répertoires intermédiaires ne sont pas toujours décrits dans l'archive ; il est mutable, car un fichier peut apparaître plusieurs fois dans l'archive, et les dernières informations sont prioritaires.

```
3  let root () =
4    let rec i =
5      { record = None; info = Dir [ Filename.current_dir_name, i ] }
6    in i
7  let link inode name nod =
8    match inode.info with
9    | File | Link _ -> error name "Not a directory"
10   | Dir list ->
11       try let _ = List.assoc name list in error name "Already exists"
```

```
12        with Not_found -> inode.info <- Dir ((name, nod) :: list)
13
14  let mkfile inode name r =
15    let f =  { record = r; info = File } in
16    link inode name f; f
17  let symlink inode name r path =
18    let s =  { record = r; info = Link path } in
19    link inode name s; s
20  let mkdir inode name r =
21    let d = mkfile inode name r in
22    d.info <-
23      Dir [ Filename.current_dir_name, d; Filename.parent_dir_name, inode ];
24    d
```

Comme en Unix, chaque répertoire contient un lien vers lui-même et un lien vers son parent, sauf le répertoire racine (contrairement à Unix où il est son propre parent). Ce choix nous permet de détecter et d'interdire l'accès en dehors de l'archive très simplement.

```
25  let rec find link inode path =
26    match inode.info, path with
27    | _, [] -> inode
28    | Dir list, name :: rest ->
29        let subnode = List.assoc name list in
30        let subnode =
31          match subnode.info with
32            Link q ->
33              if link && rest = [] then subnode else find false inode q
34          | _ -> subnode  in
35        find link subnode rest
36    | _, _ -> raise Not_found;;
```

La fonction `find` effectue une recherche dans l'archive à partir d'un nœud initial `inode` en suivant le chemin `path`. Le drapeau `link` indique si dans le cas où le résultat est un lien symbolique il faut retourner le lien lui-même (`true`) ou le fichier pointé par le lien (`false`).

```
37  let rec mkpath inode path =
38    match inode.info, path with
39    | _, [] -> inode
40    | Dir list, name :: rest ->
41        let subnode =
42          try List.assoc name list
43          with Not_found ->  mkdir inode name None in
44        mkpath subnode rest
45    | _, _ -> raise Not_found;;
```

La fonction `mkpath` parcourt le chemin `path` en créant les nœuds manquant le long du chemin.

```
46  let explode f =
47    let rec dec f p =
48      if f = Filename.current_dir_name then p
49      else dec (Filename.dirname f) (Filename.basename f :: p) in
50    dec (if Filename.basename f = "" then Filename.dirname f else f) [];;
```

La fonction `explode` décompose un chemin Unix en une liste de chaînes de caractères. Elle retire le "`/`" final qui est toléré dans les archives pour les noms de répertoires.

```
64 let add archive r =
65   match r.header.kind with
66   | CHR (_,_) | BLK (_,_) | FIFO -> ()
67   | kind ->
68       match List.rev (explode r.header.name) with
69       | []  -> ()
70       | name :: parent_rev ->
71           let inode = mkpath archive (List.rev parent_rev) in
72           match kind with
73           | DIR -> ignore (mkdir inode name (Some r))
74           | REG | CONT -> ignore (mkfile inode name (Some r))
75           | LNK f -> ignore (symlink inode name (Some r) (explode f))
76           | LINK f -> link inode name (find true archive (explode f))
77           | _ -> assert false;;
```

La fonction `add` ajoute l'enregistrement `r` dans l'archive. L'archive représentée par sa racine est modifiée par effet de bord.

```
78 let find_and_copy tarfile filename =
79   let fd = openfile tarfile [ O_RDONLY ] 0 in
80   let records = List.rev (fold (fun x y -> x :: y) [] fd) in
81   let archive = root() in
82   List.iter (add archive) records;
83   let inode =
84     try find false archive (explode filename)
85     with Not_found -> error filename "File not found" in
86   begin match inode.record with
87   | Some ({ header = { kind = (REG | CONT) }} as r) -> copy_file r stdout
88   | Some _ -> error filename "Not a regular file"
89   | None -> error filename "Not found"
90   end;
91   close fd;;
```

On termine comme précédemment.

Exercice 8, page 41

Cet exercice combine l'exercice précédent (exercice 7) et la copie récursive de fichiers (exercice 6).

La seule petite difficulté est la gestion des droits : il faut créer les répertoires de l'archive avec les droits en écriture et ne mettre ceux-ci à leur valeur final qu'après extractions de tous les fichiers.

Écrivons d'abord une fonction annexe pour `mkpath` p m qui les directory manquant le long du chemin p avec les permissions m (avec la particularité que p peut-être terminé par une `/` superflu).

```
1 let warning mes = prerr_string mes;prerr_newline();;
2 open Filename
3 let mkpath p perm =
4   let normal_path =
5     if basename p = "" then dirname p else p in
6   let path_to_dir = dirname normal_path in
```

```
  let rec make p =
    try ignore (stat p)
    with Unix_error (ENOENT, _, _) ->
      if p = current_dir_name then ()
      else if p = parent_dir_name then
        warning "Ill formed archive: path contains \"..\""
      else begin
        make (dirname p);
        mkdir p perm
      end in
  make path_to_dir;;
```

Nous définissons également une fonction `set_infos` analogue à la version utilisée pour la copie de fichiers (section 2.15) :

```
let set_infos header =
  chmod header.name header.perm;
  let mtime = float header.mtime in
  utimes header.name mtime mtime;
  begin match header.kind with
    LNK f -> ()
  | _ ->  chmod header.name header.perm
  end;
  try chown header.name  header.uid header.gid
  with Unix_error(EPERM,_,_) -> ();;
```

Le corps du programme est la fonction `untar_file_collect_dirs` qui traite une seule entrée en accumulant les répertoires explicitement créés par l'archive.

```
let verbose = ref true;;
let default_dir_perm = 0o777;;
let default_file_perm = 0o666;;

let protect f x g y = try f x; g y with z -> g y; raise z
let file_exists f = try ignore (stat f); true with _ -> false;;

let untar_file_collect_dirs file dirs =
  let fh = file.header in
  if !verbose then begin print_string fh.name; print_newline() end;
  match fh.kind with
  | CHR (_,_) | BLK(_,_) | FIFO ->
      warning (fh.name ^ "Ignoring special files");
      dirs
  | DIR ->
      mkpath fh.name default_dir_perm;
      if file_exists fh.name then dirs
      else begin mkdir fh.name default_dir_perm; fh :: dirs end
  | x ->
      mkpath fh.name default_dir_perm;
      begin match x with
      | REG | CONT ->
          let flags = [ O_WRONLY; O_TRUNC; O_CREAT; ] in
```

```
51            let out = openfile fh.name flags default_file_perm in
52            protect (copy_file file) out close out
53        | LNK f ->
54            symlink f fh.name
55        | LINK f ->
56            begin
57              try if (stat fh.name).st_kind = S_REG then unlink fh.name
58              with Unix_error(_,_,_) -> ();
59            end;
60            Unix.link f fh.name;
61        | _ -> assert false
62        end;
63        set_infos fh;
64        dirs;;
```

Nous omettons la fin du programme qui consiste à itérer le corps du programme sur l'archive puis à mettre à jour les droits des répertoires en tout dernier.

Exercice 9, page 41

Nous réutilisons les structures de données définies ci-dessus dans la bibliothèque `Tarlib`. Nous ajoutons au report d'erreurs les messages d'avertissement qui n'arrêtent pas l'archivage et n'altère pas le code de retour.

```
1  open Sys
2  open Unix
3  open Tarlib
4
5  let warning path message =  prerr_endline (path ^ ": " ^ message)
```

Commençons par l'écriture d'un entête de type `header` dans un buffer de taille suffisante (donc au moins la taille d'un bloc). L'écriture de cette fonction est ennuyante, mais doit être faite avec soin car une seule erreur dans l'écriture de l'entête peut corrompre toute l'archive, comme bien souvent en informatique. On fera attention en particulier de respecter les limites imposées dans l'archivage. Par exemple, la taille des chemins est limitée à 99 octets. Il existe des extensions du format des archives qui permettent de traiter des chemins de taille plus longue, mais ce n'est pas le but du projet.

```
6  let write_header_to_buffer source infos kind =
7    let size = if kind = REG then infos.st_size else 0 in
8    String.fill buffer 0 block_size '\000';
9    let put len string offset =
10     String.blit string 0 buffer offset (min (String.length string) len) in
11   let put_int8 x = put 7 (Printf.sprintf "%07o" x) in
12   let put_int12 x = put 11 (Printf.sprintf "%011o" x) in
13   let put_char c offset = buffer.[offset] <- c in
14   let put_path s offset =
15     if String.length s <= 99 then put 99 s offset
16     else raise (Error ("path too long", s)) in
17   put_path (if kind = DIR then source ^ "/" else source) 0;
18   put_int8 infos.st_perm 100;
19   put_int8 infos.st_uid 108;
```

```
20    put_int8 infos.st_gid 116;
21    put_int12 size 124;
22    put_int12 (int_of_float infos.st_mtime) 136;
23    put 7 "ustar  " 257;
24    put 31 (getpwuid infos.st_uid).pw_name 265;
25    put 31 (getgrgid infos.st_gid).gr_name 297;
26    (* Fields dev and rdev are only used for special files, which we omit *)
27    put_char
28      begin match kind with
29      | REG -> '0'
30      | LINK s -> put_path s 157; '1'
31      | LNK s ->  put_path s 157; '2'
32      | DIR -> '5'
33      | _ -> failwith "Special files not implemented"
34      end 156;
35    let rec sum s i =
36      if i < 0 then s else sum (s + Char.code buffer.[i]) (pred i) in
37    let checksum = sum (Char.code ' ' * 8) (block_size - 1)  in
38    put 8 (Printf.sprintf "%06o\000 " checksum) 148;;
```

À l'inverse, nous créons un entête à partir d'une entrée d'une archive : `source` est le nom du fichiers ; `infos` sont les informations sur le fichier (de type `stats`) et kind est le type du fichier dans l'archive (type `Tarlib.kind`).

```
39  let header source infos kind = {
40    name = source;
41    size = if kind = REG then infos.st_size else 0;
42    perm = infos.st_perm;
43    mtime = int_of_float infos.st_mtime;
44    uid = infos.st_uid;
45    gid = infos.st_gid;
46    user = (getpwuid infos.st_uid).pw_name;
47    group = (getgrgid infos.st_gid).gr_name;
48    kind = kind }
```

Pour écrire un fichier dans l'archive, nous définissons une variante de `file_copy` qui prend en argument le nombre d'octets à copier et vérifie que la fin de fichier correspond bien à la taille indiquée. Sinon, une erreur est levée : celle-ci correspond à un cas pathologique où le fichier est en train d'être modifié pendant l'archivage. On prend soin de ne jamais dépasser la taille indiquée, ce qui limitera la corruption d'une archive à un seul fichier.

```
49  let write_file len source fdout =
50    let fdin = openfile source [O_RDONLY] 0 in
51    let error () = raise (Error ("File changed size", source)) in
52    let rec copy_loop len =
53      match read fdin buffer 0 buffer_size with
54        0 ->
55          close fdin; if len > 0 then error ()
56      | r ->
57          let len = len - r  in
58          if len < 0 then (close fdin; error());
59          ignore (write fdout buffer 0 r); copy_loop len in
```

```
60    copy_loop len;;
61
62  let padding fd len =
63    if len > 0 then ignore (write fd (String.make len '\000') 0 len);;
```

Nous pouvons maintenant nous attaquer à la lecture de la création de l'archive. Les fichiers déjà enregistrés dans l'archive sont mémorisés dans une table de hache avec leur chemin dans l'archive afin de ne les recopier qu'une seule fois. Nous allons mémoriser également les répertoires déjà enregistrés afin de ne pas les recopier à nouveau : en effet il peut arriver qu'une racine d'archivage soit déjà contenue dans une autre. On évitera de la recopier (bien qu'il ne serait pas grave de le faire).

Une archive en cours d'écriture est donc identifiée par le descripteur dans lequel elle est écrite et ses deux caches. Nous ajoutons un champ qui maintient la taille de l'archive, afin de pouvoir compléter celle-ci pour atteindre une taille minimale.

```
64  type archive =
65      { regfiles : (int * int, string) Hashtbl.t;
66        dirfiles : (int * int, bool) Hashtbl.t;
67        fd : file_descr; st : stats; mutable size : int }
68
69  let try_new_dir archive dir =
70    try Hashtbl.find archive.dirfiles dir
71    with Not_found -> Hashtbl.add archive.dirfiles dir false; true
```

Voici la fonction principale qui écrit toute une arborescence à partir d'une entrée dans l'archive `file` passée sur la ligne de commande. Cette fonction n'est pas difficile, mais il faut prendre quelques précautions par rapport aux cas pathologiques. En particulier, nous allons vu comment détecter le cas fichier en train d'être modifié pendant son archivage. Un sous-cas de celui-ci est lorsque l'archive est elle-même est en train d'être archivée...

```
72  let verbose = ref true;;
73
74  let write_from archive file =
75    if not (Filename.is_relative file) then
76      raise (Error ("absolute path", file));
77    let rec write_rec archive file =
78      let source =
79        if Filename.basename file = "" then Filename.dirname file else file in
80      if !verbose then begin prerr_endline source end;
81      let st = lstat source in
82      if st.st_ino = archive.st.st_ino && st.st_dev = archive.st.st_dev
83      then warning source "Skiping archive itself!"
84      else
85        let write_header kind =
86          write_header_to_buffer source st kind;
87          ignore (write archive.fd buffer 0 block_size) in
88        match st.st_kind with
89          S_REG ->
90            begin try
91              if st.st_nlink = 1 then raise Not_found;
92              let path =
93                Hashtbl.find archive.regfiles (st.st_ino, st.st_dev) in
```

```
94              write_header (LINK path);
95            with Not_found ->
96              if st.st_nlink > 1 then
97                Hashtbl.add archive.regfiles (st.st_ino, st.st_dev) source;
98              write_header REG;
99              write_file st.st_size source archive.fd;
100             let t =
101               (block_size-1 + st.st_size) / block_size * block_size in
102             padding archive.fd (t - st.st_size);
103             archive.size <- archive.size + t + block_size;
104           end
105       | S_LNK ->
106           write_header (LNK (readlink source));
107       | S_DIR when try_new_dir archive (st.st_ino, st.st_dev) ->
108           write_header DIR;
109           Misc.iter_dir
110             begin
111               fun file ->
112                 if file = Filename.current_dir_name then ()
113                 else if file = Filename.parent_dir_name then ()
114                 else write_rec archive (source ^ "/" ^ file)
115             end
116             source
117       | S_DIR ->
118           warning source "Ignoring directory already in archive."
119       | _ ->
120           prerr_endline ("Can't cope with special file " ^ source) in
121   write_rec archive file;;
```

Nous mémorisons les fichiers réguliers qui peuvent avoir de liens durs dans la table `regfiles`. Ce n'est pas nécessaire pour les fichiers qui n'ont qu'un seul lien (lignes 91 et 96).

Il ne reste plus qu'à finir le programme. En cas d'erreur, il est plus prudent de retirer l'archive erronée.

```
122 let min_archive_size = 20 * block_size;;
123
124 let build tarfile files =
125   let fd, remove =
126     if tarfile = "-" then stdout, ignore
127     else openfile tarfile [ O_WRONLY; O_CREAT; O_TRUNC ] 0o666, unlink in
128   try
129     let arch =
130          { regfiles = Hashtbl.create 13; dirfiles = Hashtbl.create 13;
131            st = fstat fd; fd = fd; size =0 } in
132     Array.iter (write_from arch) files;
133     padding fd (min_archive_size - arch.size);
134     close fd
135   with z ->
136     remove tarfile; close fd; raise z;;
```

Pour terminer il ne reste plus qu'à analyser la ligne de commande.

```
137 let usage() =
138   prerr_endline "Usage: tar -cvf tarfile file1 [ file2 ... ] ";
139   exit 2;;
140
141 let tar () =
142   let argn = Array.length Sys.argv in
143   if argn > 3 && Sys.argv.(1) = "-cvf" then
144     build Sys.argv.(2) (Array.sub Sys.argv 3 (argn-3))
145   else usage();;
146
147 let _ =
148   try handle_unix_error tar ()
149   with Error (mes, s) ->
150     prerr_endline ("Error: " ^ mes ^ ": " ^ s); exit 1;;
```

Exercice 10, page 48

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 Pervasives.stdin in
  let words, ampersand = parse_command_line cmd in
  match fork() with
    0 -> exec_command words
  | pid_son ->
      if ampersand then ()
      else
        let rec wait_for_son() =
          let pid, status = wait () in
          if pid = pid_son then
            print_status "Program" status
          else
            let p = "Background program " ^ (string_of_int pid) in
            print_status p status;
            wait_for_son() in
        wait_for_son()
done
```

Exercice 12, page 65

Il faut attendre le fils bien sûr, mais avant cela il faut fermer le descripteur de fichier sur lequel lit le fils, sinon celui-ci attendrait indéfiniment que son père émette d'autres données, conduisant à un interblocage (la fermeture d'un canal se charge de vider le tampon juste avant

sa fermeture, donc on ne perd rien). Concrètement, on remplace la ligne 52 par :

```
let output = out_channel_of_descr fd_out in
generate len output;
close_out output;
ignore(waitpid [] k);;
```

De même, on va entourer le bloc 37–41 (désigné par `...`) par les lignes suivantes :

```
try
  ...
with End_of_file ->
  close_out output;
  ignore (waitpid [] p)
```

Exercice 13, page 65

Les tampons d'entrées/sorties de la bibliothèque standard sont dupliqués au moment de la commande `fork` (puisque le processus fils est une copie du processus père). Si, les tampons ne sont plus vidés à chaque écriture, alors il faut les vider explicitement juste avant la commande `fork`. Pour cela, il suffit d'ajouter `flush Pervasives.stdout` entre les lignes 25 et 26.

Exercice 14, page 68

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.

Exercice 18, page 105

Le tri rapide se prête bien à une parallélisation, car le tri se fait récursivement sur des sous-tableaux indépendants et peut être délégué à des coprocessus avec pour seule synchronisation d'attendre que tous les coprocessus aient terminé leur tri pour affirmer que le sous-tableau est trié.

```
1 let qsort cmp arr =
2   let rec qsort lo hi =
3   if hi - lo > 0 then
4     begin
5       let mid = (lo + hi) lsr 1 in
```

```
6       if cmp arr.(mid) arr.(lo) then swap arr mid lo;
7       if cmp arr.(hi) arr.(mid) then
8         begin
9           swap arr mid hi;
10          if cmp arr.(mid) arr.(lo) then swap arr mid lo
11        end;
12      let pivot = arr.(mid) in
13      let i = ref (lo + 1) and j = ref (hi - 1) in
14      while !i < !j do
15        while not (cmp pivot arr.(!i)) do incr i done;
16        while not (cmp arr.(!j) pivot) do decr j done;
17        if !i < !j then swap arr !i !j;
18      done;
19      let u = Thread.create (qsort lo) (!i-1) in
20      let v = Thread.create (qsort (!i+1)) hi in
21      Thread.join u;
22      Thread.join v
23    end in
24  qsort 0 (Array.length arr - 1);;
```

Il serait correct, mais inintéressant d'intervertir les lignes 20 et 21. En effet, cela aurait pour conséquence d'attendre que la partie base du tableau soit triée pour lancer le tri de la partie haute. On obtiendrait alors le comportement d'un programme séquentiel, plus le coût des coprocessus sans en retirer aucun bénéfice.

En pratique, on devrait limiter la parallélisation à un facteur raisonnable et continuer de façon séquentielles au delà.

Exercice 20, page 111

Il faut introduire une condition supplémentaire `non_full`. On ajoute également un champ `size` pour permettre des queues de différentes tailles.

```
1  type 'a t =
2      { queue : 'a Queue.t; size : int; lock : Mutex.t;
3        non_empty : Condition.t; non_full : Condition.t; }
4
5  let create k =
6    if  k > 0 then
7      { queue = Queue. create(); size = k; lock = Mutex. create();
8        non_empty = Condition. create(); non_full = Condition.create() }
9    else failwith "Tqueue.create: empty size";;
```

L'ajout est une combinaison des versions précédentes des fonctions `add` et `take` ci-dessus.

```
10 let add x q =
11   Mutex.lock q.lock;
12   while Queue.length q.queue = q.size
13   do Condition.wait q.non_full q.lock done;
14   if Queue.is_empty q.queue then Condition.broadcast q.non_empty;
15   Queue.add q x;
16   Mutex.unlock q.lock;;
```

Le retrait est symétrique à l'ajout (et doit maintenant signaler `non_full` lorsque la queue est

pleine avant le retrait) et est laissé au lecteur. On retrouve le comportement des queues non bornées en choisissant `max_int` pour `size`.

Annexe B

Interfaces

B.1 Module `Sys` : System interface.

`val argv : string array`
: The command line arguments given to the process. The first element is the command name used to invoke the program. The following elements are the command-line arguments given to the program.

`val executable_name : string`
: The name of the file containing the executable currently running.

`val file_exists : string -> bool`
: Test if a file with the given name exists.

`val is_directory : string -> bool`
: Returns `true` if the given name refers to a directory, `false` if it refers to another kind of file. Raise `Sys_error` if no file exists with the given name.

`val remove : string -> unit`
: Remove the given file name from the file system.

`val rename : string -> string -> unit`
: Rename a file. The first argument is the old name and the second is the new name. If there is already another file under the new name, `rename` may replace it, or raise an exception, depending on your operating system.

`val getenv : string -> string`
: Return the value associated to a variable in the process environment. Raise `Not_found` if the variable is unbound.

`val command : string -> int`
: Execute the given shell command and return its exit code.

`val time : unit -> float`
: Return the processor time, in seconds, used by the program since the beginning of execution.

`val chdir : string -> unit`

Change the current working directory of the process.

`val getcwd : unit -> string`
Return the current working directory of the process.

`val readdir : string -> string array`
Return the names of all files present in the given directory. Names denoting the current directory and the parent directory (`.` and `..` in Unix) are not returned. Each string in the result is a file name rather than a complete path. There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

`val interactive : bool Pervasives.ref`
This reference is initially set to `false` in standalone programs and to `true` if the code is being executed under the interactive toplevel system `ocaml`.

`val os_type : string`
Operating system currently executing the Caml program. One of
- `Unix` (for all Unix versions, including Linux and Mac OS X),
- `Win32` (for MS-Windows, OCaml compiled with MSVC++ or Mingw),
- `Cygwin` (for MS-Windows, OCaml compiled with Cygwin).

`val word_size : int`
Size of one word on the machine currently executing the Caml program, in bits: 32 or 64.

`val max_string_length : int`
Maximum length of a string.

`val max_array_length : int`
Maximum length of a normal array. The maximum length of a float array is `max_array_length/2` on 32-bit machines and `max_array_length` on 64-bit machines.

Signal handling

```
type signal_behavior =
  | Signal_default
  | Signal_ignore
  | Signal_handle of (int -> unit)
```
What to do when receiving a signal:
- `Signal_default`: take the default behavior (usually: abort the program)
- `Signal_ignore`: ignore the signal
- `Signal_handle f`: call function `f`, giving it the signal number as argument.

`val signal : int -> signal_behavior -> signal_behavior`
Set the behavior of the system on receipt of a given signal. The first argument is the signal number. Return the behavior previously associated with the signal. If the signal number is invalid (or not available on your system), an `Invalid_argument` exception is raised.

`val set_signal : int -> signal_behavior -> unit`
Same as `Sys.signal` but return value is ignored.

Signal numbers for the standard POSIX signals.

`val sigabrt : int`
Abnormal termination

`val sigalrm : int`
Timeout

`val sigfpe : int`
Arithmetic exception

`val sighup : int`
Hangup on controlling terminal

`val sigill : int`
Invalid hardware instruction

`val sigint : int`
Interactive interrupt (ctrl-C)

`val sigkill : int`
Termination (cannot be ignored)

`val sigpipe : int`
Broken pipe

`val sigquit : int`
Interactive termination

`val sigsegv : int`
Invalid memory reference

`val sigterm : int`
Termination

`val sigusr1 : int`
Application-defined signal 1

`val sigusr2 : int`
Application-defined signal 2

`val sigchld : int`
Child process terminated

`val sigcont : int`
Continue

`val sigstop : int`
Stop

`val sigtstp : int`

Interactive stop

`val sigttin : int`
Terminal read from background process

`val sigttou : int`
Terminal write from background process

`val sigvtalrm : int`
Timeout in virtual time

`val sigprof : int`
Profiling interrupt

`exception Break`
Exception raised on interactive interrupt if `Sys.catch_break` is on.

`val catch_break : bool -> unit`
`catch_break` governs whether interactive interrupt (ctrl-C) terminates the program or raises the `Break` exception. Call `catch_break true` to enable raising `Break`, and `catch_break false` to let the system terminate the program on user interrupt.

`val ocaml_version : string`
`ocaml_version` is the version of Objective Caml. It is a string of the form `major.minor[.patchlevel][+additional-info]`, where `major`, `minor`, and `patchlevel` are integers, and `additional-info` is an arbitrary string. The `[.patchlevel]` and `[+additional-info]` parts may be absent.

B.2 Module `Unix` : Interface to the Unix system

Error report

`type error =`
`  | E2BIG`
Argument list too long
`  | EACCES`
Permission denied
`  | EAGAIN`
Resource temporarily unavailable; try again
`  | EBADF`
Bad file descriptor
`  | EBUSY`
Resource unavailable
`  | ECHILD`
No child process
`  | EDEADLK`
Resource deadlock would occur

| EDOM
: Domain error for math functions, etc.

| EEXIST
: File exists

| EFAULT
: Bad address

| EFBIG
: File too large

| EINTR
: Function interrupted by signal

| EINVAL
: Invalid argument

| EIO
: Hardware I/O error

| EISDIR
: Is a directory

| EMFILE
: Too many open files by the process

| EMLINK
: Too many links

| ENAMETOOLONG
: Filename too long

| ENFILE
: Too many open files in the system

| ENODEV
: No such device

| ENOENT
: No such file or directory

| ENOEXEC
: Not an executable file

| ENOLCK
: No locks available

| ENOMEM
: Not enough memory

| ENOSPC
: No space left on device

| ENOSYS
: Function not supported

| ENOTDIR
: Not a directory

| `ENOTEMPTY`
: Directory not empty

| `ENOTTY`
: Inappropriate I/O control operation

| `ENXIO`
: No such device or address

| `EPERM`
: Operation not permitted

| `EPIPE`
: Broken pipe

| `ERANGE`
: Result too large

| `EROFS`
: Read-only file system

| `ESPIPE`
: Invalid seek e.g. on a pipe

| `ESRCH`
: No such process

| `EXDEV`
: Invalid link

| `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

| `EHOSTDOWN`
  Host is down

| `EHOSTUNREACH`
  No route to host

| `ELOOP`
  Too many levels of symbolic links

| EOVERFLOW

File size or position not representable

| EUNKNOWNERR of int

Unknown error

The type of error codes. Errors defined in the POSIX standard and additional errors from UNIX98 and BSD. All other errors are mapped to EUNKNOWNERR.

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.

val error_message : error -> string

Return a string describing the given error code.

val 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.

Access to the process environment

val environment : unit -> string array

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

val getenv : string -> string

Return the value associated to a variable in the process environment. Raise Not_found if the variable is unbound. (This function is identical to Sys.getenv.)

val putenv : string -> string -> unit

Unix.putenv name value sets the value associated to a variable in the process environment. name is the name of the environment variable, and value its new associated value.

Process handling

type process_status =
  | WEXITED of int

The process terminated normally by exit; the argument is the return code.

| WSIGNALED of int

The process was killed by a signal; the argument is the signal number.

| WSTOPPED of int

The process was stopped by a signal; the argument is the signal number.

The termination status of a process. See module Sys for the definitions of the standard signal numbers. Note that they are not the numbers used by the OS.

type wait_flag =
  | WNOHANG

do not block if no child has died yet, but immediately return with a pid equal to 0.

| `WUNTRACED`
  report also the children that receive stop signals.

Flags for `Unix.waitpid`.

`val execv : string -> string array -> 'a`
`execv prog args` execute the program in file `prog`, with the arguments `args`, and the current process environment. These `execv*` functions never return: on success, the current program is replaced by the new one; on failure, a `Unix.Unix_error` exception is raised.

`val execve : string -> string array -> string array -> 'a`
Same as `Unix.execv`, except that the third argument provides the environment to the program executed.

`val execvp : string -> string array -> 'a`
Same as `Unix.execv`, except that the program is searched in the path.

`val execvpe : string -> string array -> string array -> 'a`
Same as `Unix.execve`, except that the program is searched in the path.

`val 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.

`val wait : unit -> int * process_status`
Wait until one of the children processes die, and return its pid and termination status.

`val waitpid : wait_flag list -> int -> int * process_status`
Same as `Unix.wait`, but waits for the child process whose pid is given. A pid of `-1` means wait for any child. A pid of `0` means wait for any child in the same process group as the current process. Negative pid arguments represent process groups. The list of options indicates whether `waitpid` should return immediately without waiting, or also report stopped children.

`val system : string -> process_status`
Execute the given command, wait until it terminates, and return its termination status. The string is interpreted by the shell `/bin/sh` and therefore can contain redirections, quotes, variables, etc. The result `WEXITED 127` indicates that the shell couldn't be executed.

`val getpid : unit -> int`
Return the pid of the process.

`val getppid : unit -> int`
Return the pid of the parent process.

`val nice : int -> int`
Change the process priority. The integer argument is added to the "nice" value. (Higher values of the "nice" value mean lower priorities.) Return the new nice value.

Basic file input/output

`type file_descr`
The abstract type of file descriptors.

`val stdin : file_descr`
File descriptor for standard input.

`val stdout : file_descr`
File descriptor for standard output.

`val stderr : file_descr`
File descriptor for standard error.

`type open_flag =`
- `| O_RDONLY`
  Open for reading
- `| O_WRONLY`
  Open for writing
- `| O_RDWR`
  Open for reading and writing
- `| O_NONBLOCK`
  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
- `| O_NOCTTY`
  Don't make this dev a controlling tty
- `| O_DSYNC`
  Writes complete as 'Synchronised I/O data integrity completion'
- `| O_SYNC`
  Writes complete as 'Synchronised I/O file integrity completion'
- `| O_RSYNC`
  Reads complete as writes (depending on O_SYNC/O_DSYNC)

The flags to `Unix.openfile`.

`type file_perm = int`
The type of file access rights, e.g. `0o640` is read and write for user, read for group, none for others

`val openfile : 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.

`val close : file_descr -> unit`
: Close a file descriptor.

`val read : file_descr -> string -> int -> int -> int`
: `read fd buff ofs 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.

`val write : file_descr -> string -> int -> int -> int`
: `write fd buff ofs 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. `write` repeats the writing operation until all characters have been written or an error occurs.

`val single_write : file_descr -> string -> int -> int -> int`
: Same as `write`, but attempts to write only once. Thus, if an error occurs, `single_write` guarantees that no data has been written.

Interfacing with the standard input/output library

`val in_channel_of_descr : file_descr -> Pervasives.in_channel`
: Create an input channel reading from the given descriptor. The channel is initially in binary mode; use `set_binary_mode_in ic false` if text mode is desired.

`val out_channel_of_descr : file_descr -> Pervasives.out_channel`
: Create an output channel writing on the given descriptor. The channel is initially in binary mode; use `set_binary_mode_out oc false` if text mode is desired.

`val descr_of_in_channel : Pervasives.in_channel -> file_descr`
: Return the descriptor corresponding to an input channel.

`val descr_of_out_channel : Pervasives.out_channel -> file_descr`
: Return the descriptor corresponding to an output channel.

Seeking and truncating

`type seek_command =`

`  | SEEK_SET`
: indicates positions relative to the beginning of the file

`  | SEEK_CUR`
: indicates positions relative to the current position

`  | SEEK_END`
: indicates positions relative to the end of the file

Positioning modes for `Unix.lseek`.

`val lseek : file_descr -> int -> seek_command -> int`
: Set the current position for a file descriptor

`val truncate : string -> int -> unit`
: Truncates the named file to the given size.

`val ftruncate : file_descr -> int -> unit`
: Truncates the file corresponding to the given descriptor to the given size.

File status

`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 file's group

`st_rdev : int ;`

Device minor number

`st_size : int ;`
Size in bytes

`st_atime : float ;`
Last access time

`st_mtime : float ;`
Last modification time

`st_ctime : float ;`
Last status change time

}
The informations returned by the `Unix.stat` calls.

`val stat : string -> stats`
Return the information for the named file.

`val lstat : string -> stats`
Same as `Unix.stat`, but in case the file is a symbolic link, return the information for the link itself.

`val fstat : file_descr -> stats`
Return the information for the file associated with the given descriptor.

`val isatty : file_descr -> bool`
Return `true` if the given file descriptor refers to a terminal or console window, `false` otherwise.

subsection*Operations on file names
`val unlink : string -> unit`
Removes the named file

`val rename : string -> string -> unit`
`rename old new` changes the name of a file from `old` to `new`.

`val link : string -> string -> unit`
`link source dest` creates a hard link named `dest` to the file named `source`.

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 `Unix.access` call.

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

Change the permissions of the named file.

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

Change the permissions of an opened file.

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

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

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

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

`val umask : int -> int`

Set the process's file mode creation mask, and return the previous mask.

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

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

Operations on file descriptors

`val dup : file_descr -> file_descr`

Return a new file descriptor referencing the same file as the given descriptor.

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

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

`val set_nonblock : file_descr -> unit`

Set the "non-blocking" flag on the given descriptor. When the non-blocking flag is set, reading on a descriptor on which there is temporarily no data available raises the `EAGAIN` or `EWOULDBLOCK` error instead of blocking; writing on a descriptor on which there is temporarily no room for writing also raises `EAGAIN` or `EWOULDBLOCK`.

`val clear_nonblock : file_descr -> unit`

Clear the "non-blocking" flag on the given descriptor. See `Unix.set_nonblock`.

`val set_close_on_exec : file_descr -> unit`

Set the "close-on-exec" flag on the given descriptor. A descriptor with the close-on-exec flag is automatically closed when the current process starts another program with one of the `exec` functions.

`val clear_close_on_exec : file_descr -> unit`

Clear the "close-on-exec" flag on the given descriptor. See `Unix.set_close_on_exec`.

Directories

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

Create a directory with the given permissions.

`val rmdir : string -> unit`

Remove an empty directory.

`val chdir : string -> unit`

Change the process working directory.

`val getcwd : unit -> string`

Return the name of the current working directory.

`val chroot : string -> unit`

Change the process root directory.

`type dir_handle`

The type of descriptors over opened directories.

`val opendir : string -> dir_handle`

Open a descriptor on a directory

`val readdir : dir_handle -> string`

Return the next entry in a directory.

Raises `End_of_file` when the end of the directory has been reached.

`val rewinddir : dir_handle -> unit`

Reposition the descriptor to the beginning of the directory

`val closedir : dir_handle -> unit`

Close a directory descriptor.

Pipes and redirections

`val 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 entrance to the pipe.

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

Create a named pipe with the given permissions.

High-level process and redirection management

```
val create_process :
  string ->
  string array -> file_descr -> file_descr -> file_descr -> int
```

`create_process prog args new_stdin new_stdout new_stderr` forks a new process that executes the program in file `prog`, with arguments `args`. The pid of the new process is returned immediately; the new process executes concurrently with the current process. The standard input and outputs of the new process are connected to the descriptors `new_stdin`, `new_stdout` and `new_stderr`. Passing e.g. `stdout` for `new_stdout` prevents the redirection and causes the new process to have the same standard output as the current process. The executable file `prog` is searched in the path. The new process has the same environment as the current process.

```
val create_process_env :
  string ->
  string array ->
  string array -> file_descr -> file_descr -> file_descr -> int
```

`create_process_env prog args env new_stdin new_stdout new_stderr` works as `Unix.create_process`, except that the extra argument `env` specifies the environment passed to the program.

```
val open_process_in : string -> Pervasives.in_channel
```

High-level pipe and process management. This function runs the given command in parallel with the program. The standard output of the command is redirected to a pipe, which can be read via the returned input channel. The command is interpreted by the shell `/bin/sh` (cf. `system`).

```
val open_process_out : string -> Pervasives.out_channel
```

Same as `Unix.open_process_in`, but redirect the standard input of the command to a pipe. Data written to the returned output channel is sent to the standard input of the command. Warning: writes on output channels are buffered, hence be careful to call `Pervasives.flush` at the right times to ensure correct synchronization.

```
val open_process : string -> Pervasives.in_channel * Pervasives.out_channel
```

Same as `Unix.open_process_out`, but redirects both the standard input and standard output of the command to pipes connected to the two returned channels. The input channel is connected to the output of the command, and the output channel to the input of the command.

```
val open_process_full :
  string ->
  string array ->
  Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel
```

Similar to `Unix.open_process`, but the second argument specifies the environment passed to the command. The result is a triple of channels connected respectively to the standard output, standard input, and standard error of the command.

```
val close_process_in : Pervasives.in_channel -> process_status
```

Close channels opened by `Unix.open_process_in`, wait for the associated command to terminate, and return its termination status.

```
val close_process_out : Pervasives.out_channel -> process_status
```
Close channels opened by `Unix.open_process_out`, wait for the associated command to terminate, and return its termination status.

```
val close_process :
  Pervasives.in_channel * Pervasives.out_channel -> process_status
```
Close channels opened by `Unix.open_process`, wait for the associated command to terminate, and return its termination status.

```
val close_process_full :
  Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel ->
  process_status
```
Close channels opened by `Unix.open_process_full`, wait for the associated command to terminate, and return its termination status.

Symbolic links

```
val symlink : string -> string -> unit
```
`symlink source dest` creates the file `dest` as a symbolic link to the file `source`.

```
val readlink : string -> string
```
Read the contents of a link.

Polling

```
val 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 for writing, and block if already locked

```
  | F_TLOCK
```
Lock a region for writing, or fail if already locked

```
  | F_TEST
```
Test a region for other process locks

```
  | F_RLOCK
```
Lock a region for reading, and block if already locked

```
  | F_TRLOCK
```
Lock a region for reading, or fail if already locked

Commands for `Unix.lockf`.

```
val 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 `Unix.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. A write lock prevents any other process from acquiring a read or write lock on the region. A read lock prevents any other process from acquiring a write lock on the region, but lets other processes acquire read locks on it.

The `F_LOCK` and `F_TLOCK` commands attempts to put a write lock on the specified region. The `F_RLOCK` and `F_TRLOCK` commands attempts to put a read lock on the specified region. If one or several locks put by another process prevent the current process from acquiring the lock, `F_LOCK` and `F_RLOCK` block until these locks are removed, while `F_TLOCK` and `F_TRLOCK` fail immediately with an exception. The `F_ULOCK` removes whatever locks the current process has on the specified region. Finally, the `F_TEST` command tests whether a write lock can be acquired on the specified region, without actually putting a lock. It returns immediately if successful, or fails otherwise.

Signals

Note: installation of signal handlers is performed via `Sys.signal` and `Sys.set_signal`.

```
val kill : int -> int -> unit
```
`kill pid sig` sends signal number `sig` to the process with id `pid`.

```
type sigprocmask_command =
  | SIG_SETMASK
  | SIG_BLOCK
  | SIG_UNBLOCK
val sigprocmask : sigprocmask_command -> int list -> int list
```
`sigprocmask cmd sigs` changes the set of blocked signals. If `cmd` is `SIG_SETMASK`, blocked signals are set to those in the list `sigs`. If `cmd` is `SIG_BLOCK`, the signals in `sigs` are added to the set of blocked signals. If `cmd` is `SIG_UNBLOCK`, the signals in `sigs` are removed from the set of blocked signals. `sigprocmask` returns the set of previously blocked signals.

```
val sigpending : unit -> int list
```
Return the set of blocked signals that are currently pending.

```
val sigsuspend : int list -> unit
```
`sigsuspend sigs` atomically sets the blocked signals to `sigs` and waits for a non-ignored, non-blocked signal to be delivered. On return, the blocked signals are reset to their initial value.

```
val pause : unit -> unit
```
Wait until a non-ignored, non-blocked 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..60
```
  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.

```
val time : unit -> float
```
Return the current time since 00:00:00 GMT, Jan. 1, 1970, in seconds.

```
val gettimeofday : unit -> float
```

Same as `Unix.time`, but with resolution better than 1 second.

`val gmtime : float -> tm`

Convert a time in seconds, as returned by `Unix.time`, into a date and a time. Assumes UTC (Coordinated Universal Time), also known as GMT.

`val localtime : float -> tm`

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

`val mktime : tm -> float * tm`

Convert a date and time, specified by the `tm` argument, into a time in seconds, as returned by `Unix.time`. The `tm_isdst`, `tm_wday` and `tm_yday` fields of `tm` are ignored. Also return a normalized copy of the given `tm` record, with the `tm_wday`, `tm_yday`, and `tm_isdst` fields recomputed from the other fields, and the other fields normalized (so that, e.g., 40 October is changed into 9 November). The `tm` argument is interpreted in the local time zone.

`val alarm : int -> int`

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

`val sleep : int -> unit`

Stop execution for the given number of seconds.

`val times : unit -> process_times`

Return the execution times of the process.

`val utimes : string -> float -> float -> 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. A time of `0.0` is interpreted as the current time.

```
type interval_timer =
  | ITIMER_REAL
```

decrements in real time, and sends the signal `SIGALRM` when expired.

```
  | ITIMER_VIRTUAL
```

decrements in process virtual time, and sends `SIGVTALRM` when expired.

```
  | ITIMER_PROF
```

(for profiling) decrements both when the process is running and when the system is running on behalf of the process; it sends `SIGPROF` when expired.

The three kinds of interval timers.

```
type interval_timer_status = {
  it_interval : float ;
```

Period

```
  it_value : float ;
```

Current value of the timer

```
}
```

The type describing the status of an interval timer

```
val getitimer : interval_timer -> interval_timer_status
```
Return the current status of the given interval timer.

```
val setitimer :
  interval_timer ->
  interval_timer_status -> interval_timer_status
```
`setitimer t s` sets the interval timer `t` and returns its previous status. The `s` argument is interpreted as follows: `s.it_value`, if nonzero, is the time to the next timer expiration; `s.it_interval`, if nonzero, specifies a value to be used in reloading it_value when the timer expires. Setting `s.it_value` to zero disable the timer. Setting `s.it_interval` to zero causes the timer to be disabled after its next expiration.

User id, group id

```
val getuid : unit -> int
```
Return the user id of the user executing the process.

```
val geteuid : unit -> int
```
Return the effective user id under which the process runs.

```
val setuid : int -> unit
```
Set the real user id and effective user id for the process.

```
val getgid : unit -> int
```
Return the group id of the user executing the process.

```
val getegid : unit -> int
```
Return the effective group id under which the process runs.

```
val setgid : int -> unit
```
Set the real group id and effective group id for the process.

```
val getgroups : unit -> int array
```
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 array ;
}
```
Structure of entries in the `groups` database.

`val getlogin : unit -> string`
Return the login name of the user executing the process.

`val getpwnam : string -> passwd_entry`
Find an entry in `passwd` with the given name, or raise `Not_found`.

`val getgrnam : string -> group_entry`
Find an entry in `group` with the given name, or raise `Not_found`.

`val getpwuid : int -> passwd_entry`
Find an entry in `passwd` with the given user id, or raise `Not_found`.

`val 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.

`val inet_addr_of_string : string -> inet_addr`
Conversion from the printable representation of an Internet address to its internal representation. The argument string consists of 4 numbers separated by periods (`XXX.YYY.ZZZ.TTT`) for IPv4 addresses, and up to 8 numbers separated by colons for IPv6 addresses. Raise `Failure` when given a string that does not match these formats.

`val string_of_inet_addr : inet_addr -> string`
Return the printable representation of the given Internet address. See `Unix.inet_addr_of_string` for a description of the printable representation.

`val inet_addr_any : inet_addr`
A special IPv4 address, for use only with `bind`, representing all the Internet addresses that the host machine possesses.

`val inet_addr_loopback : inet_addr`
A special IPv4 address representing the host machine (`127.0.0.1`).

`val inet6_addr_any : inet_addr`
A special IPv6 address, for use only with `bind`, representing all the Internet addresses that the host machine possesses.

`val inet6_addr_loopback : inet_addr`
A special IPv6 address representing the host machine (`::1`).

Sockets

```
type socket_domain =
  | PF_UNIX
```
Unix domain
```
  | PF_INET
```
Internet domain (IPv4)
```
  | PF_INET6
```
Internet domain (IPv6)

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.

```
val 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.

```
val domain_of_sockaddr : sockaddr -> socket_domain
```
Return the socket domain adequate for the given socket address.

```
val socketpair :
  socket_domain ->
  socket_type -> int -> file_descr * file_descr
```
Create a pair of unnamed sockets, connected together.

```
val 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.

```
val bind : file_descr -> sockaddr -> unit
```
Bind a socket to an address.

```
val connect : file_descr -> sockaddr -> unit
```
Connect a socket to an address.

```
val listen : file_descr -> int -> unit
```
Set up a socket for receiving connection requests. The integer argument is the maximal number of pending requests.

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

The type of commands for `shutdown`.

```
val shutdown : file_descr -> shutdown_command -> unit
```
Shutdown a socket connection. `SHUTDOWN_SEND` as second argument causes reads on the other end of the connection to return an end-of-file condition. `SHUTDOWN_RECEIVE` causes writes on the other end of the connection to return a closed pipe condition (`SIGPIPE` signal).

```
val getsockname : file_descr -> sockaddr
```
Return the address of the given socket.

```
val getpeername : file_descr -> sockaddr
```
Return the address of the host connected to the given socket.

```
type msg_flag =
  | MSG_OOB
  | MSG_DONTROUTE
  | MSG_PEEK
```
The flags for `Unix.recv`, `Unix.recvfrom`, `Unix.send` and `Unix.sendto`.

```
val recv : file_descr -> string -> int -> int -> msg_flag list -> int
```
Receive data from a connected socket.

```
val recvfrom :
  file_descr ->
  string -> int -> int -> msg_flag list -> int * sockaddr
```
Receive data from an unconnected socket.

```
val send : file_descr -> string -> int -> int -> msg_flag list -> int
```
Send data over a connected socket.

```
val sendto :
  file_descr ->
  string -> int -> int -> msg_flag list -> sockaddr -> int
```
Send data over an unconnected socket.

Socket options

```
type socket_bool_option =
  | SO_DEBUG
```

Record debugging information

```
  | SO_BROADCAST
```

Permit sending of broadcast messages

```
  | SO_REUSEADDR
```

Allow reuse of local addresses for bind

```
  | SO_KEEPALIVE
```

Keep connection active

```
  | SO_DONTROUTE
```

Bypass the standard routing algorithms

```
  | SO_OOBINLINE
```

Leave out-of-band data in line

```
  | SO_ACCEPTCONN
```

Report whether socket listening is enabled

```
  | TCP_NODELAY
```

Control the Nagle algorithm for TCP sockets

```
  | IPV6_ONLY
```

Forbid binding an IPv6 socket to an IPv4 address

The socket options that can be consulted with `Unix.getsockopt` and modified with `Unix.setsockopt`. These options have a boolean (`true`/`false`) value.

```
type socket_int_option =
  | SO_SNDBUF
```

Size of send buffer

```
  | SO_RCVBUF
```

Size of received buffer

```
  | SO_ERROR
```

Deprecated. Use `Unix.getsockopt_error` instead.

```
  | SO_TYPE
```

Report the socket type

```
  | SO_RCVLOWAT
```

Minimum number of bytes to process for input operations

```
  | SO_SNDLOWAT
```

Minimum number of bytes to process for output operations

The socket options that can be consulted with `Unix.getsockopt_int` and modified with `Unix.setsockopt_int`. These options have an integer value.

```
type socket_optint_option =
  | SO_LINGER
```

Whether to linger on closed connections that have data present, and for how long (in seconds)

The socket options that can be consulted with `Unix.getsockopt_optint` and modified with `Unix.setsockopt_optint`. These options have a value of type `int option`, with `None` meaning "disabled".

```
type socket_float_option =
  | SO_RCVTIMEO
```

Timeout for input operations

```
  | SO_SNDTIMEO
```

Timeout for output operations

The socket options that can be consulted with `Unix.getsockopt_float` and modified with `Unix.setsockopt_float`. These options have a floating-point value representing a time in seconds. The value 0 means infinite timeout.

```
val getsockopt : file_descr -> socket_bool_option -> bool
```

Return the current status of a boolean-valued option in the given socket.

```
val setsockopt : file_descr -> socket_bool_option -> bool -> unit
```

Set or clear a boolean-valued option in the given socket.

```
val getsockopt_int : file_descr -> socket_int_option -> int
```

Same as `Unix.getsockopt` for an integer-valued socket option.

```
val setsockopt_int : file_descr -> socket_int_option -> int -> unit
```

Same as `Unix.setsockopt` for an integer-valued socket option.

```
val getsockopt_optint : file_descr -> socket_optint_option -> int option
```

Same as `Unix.getsockopt` for a socket option whose value is an `int option`.

```
val setsockopt_optint :
  file_descr -> socket_optint_option -> int option -> unit
```

Same as `Unix.setsockopt` for a socket option whose value is an `int option`.

```
val getsockopt_float : file_descr -> socket_float_option -> float
```

Same as `Unix.getsockopt` for a socket option whose value is a floating-point number.

```
val setsockopt_float : file_descr -> socket_float_option -> float -> unit
```

Same as `Unix.setsockopt` for a socket option whose value is a floating-point number.

```
val getsockopt_error : file_descr -> error option
```

Return the error condition associated with the given socket, and clear it.

High-level network connection functions

```
val open_connection :
  sockaddr -> Pervasives.in_channel * Pervasives.out_channel
```

Connect to a server at the given address. Return a pair of buffered channels connected to the server. Remember to call `Pervasives.flush` on the output channel at the right times to ensure correct synchronization.

```
val shutdown_connection : Pervasives.in_channel -> unit
```

"Shut down" a connection established with `Unix.open_connection`; that is, transmit an end-of-file condition to the server reading on the other side of the connection.

```
val establish_server :
  (Pervasives.in_channel -> Pervasives.out_channel -> unit) ->
  sockaddr -> unit
```

Establish a server on the given address. The function given as first argument is called for each connection with two buffered channels connected to the client. A new process is created for each connection. The function `Unix.establish_server` never returns normally.

Host and protocol databases

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

Structure of entries in the `hosts` database.

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

Structure of entries in the `protocols` database.

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

Structure of entries in the `services` database.

```
val gethostname : unit -> string
```

Return the name of the local host.

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

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

`val gethostbyaddr : inet_addr -> host_entry`
Find an entry in `hosts` with the given address, or raise `Not_found`.

`val getprotobyname : string -> protocol_entry`
Find an entry in `protocols` with the given name, or raise `Not_found`.

`val getprotobynumber : int -> protocol_entry`
Find an entry in `protocols` with the given protocol number, or raise `Not_found`.

`val getservbyname : string -> string -> service_entry`
Find an entry in `services` with the given name, or raise `Not_found`.

`val getservbyport : int -> string -> service_entry`
Find an entry in `services` with the given service number, or raise `Not_found`.

```
type addr_info = {
  ai_family : socket_domain ;
```
Socket domain

```
  ai_socktype : socket_type ;
```
Socket type

```
  ai_protocol : int ;
```
Socket protocol number

```
  ai_addr : sockaddr ;
```
Address

```
  ai_canonname : string ;
```
Canonical host name

```
}
```
Address information returned by `Unix.getaddrinfo`.

```
type getaddrinfo_option =
  | AI_FAMILY of socket_domain
```
Impose the given socket domain

```
  | AI_SOCKTYPE of socket_type
```
Impose the given socket type

```
  | AI_PROTOCOL of int
```
Impose the given protocol

```
  | AI_NUMERICHOST
```
Do not call name resolver, expect numeric IP address

```
  | AI_CANONNAME
```
Fill the `ai_canonname` field of the result

```
  | AI_PASSIVE
```
Set address to "any" address for use with `Unix.bind`

Options to `Unix.getaddrinfo`.

```
val getaddrinfo :
  string -> string -> getaddrinfo_option list -> addr_info list
```

`getaddrinfo host service opts` returns a list of `Unix.addr_info` records describing socket parameters and addresses suitable for communicating with the given host and service. The empty list is returned if the host or service names are unknown, or the constraints expressed in `opts` cannot be satisfied.

`host` is either a host name or the string representation of an IP address. `host` can be given as the empty string; in this case, the "any" address or the "loopback" address are used, depending whether `opts` contains `AI_PASSIVE`. `service` is either a service name or the string representation of a port number. `service` can be given as the empty string; in this case, the port field of the returned addresses is set to 0. `opts` is a possibly empty list of options that allows the caller to force a particular socket domain (e.g. IPv6 only or IPv4 only) or a particular socket type (e.g. TCP only or UDP only).

```
type name_info = {
  ni_hostname : string ;
```

Name or IP address of host

```
  ni_service : string ;
}
```

Name of service or port number

Host and service information returned by `Unix.getnameinfo`.

```
type getnameinfo_option =
  | NI_NOFQDN
```

Do not qualify local host names

```
  | NI_NUMERICHOST
```

Always return host as IP address

```
  | NI_NAMEREQD
```

Fail if host name cannot be determined

```
  | NI_NUMERICSERV
```

Always return service as port number

```
  | NI_DGRAM
```

Consider the service as UDP-based instead of the default TCP

Options to `Unix.getnameinfo`.

```
val getnameinfo : sockaddr -> getnameinfo_option list -> name_info
```

`getnameinfo addr opts` returns the host name and service name corresponding to the socket address `addr`. `opts` is a possibly empty list of options that governs how these names are obtained. Raise `Not_found` if an error occurs.

Terminal interface

The following functions implement the POSIX standard terminal interface. They provide control over asynchronous communication ports and pseudo-terminals. Refer to the `termios` man page for a complete description.

```
type terminal_io = {
  mutable c_ignbrk : bool ;
```

Ignore the break condition.

`mutable c_brkint : bool ;`
: Signal interrupt on break condition.

`mutable c_ignpar : bool ;`
: Ignore characters with parity errors.

`mutable c_parmrk : bool ;`
: Mark parity errors.

`mutable c_inpck : bool ;`
: Enable parity check on input.

`mutable c_istrip : bool ;`
: Strip 8th bit on input characters.

`mutable c_inlcr : bool ;`
: Map NL to CR on input.

`mutable c_igncr : bool ;`
: Ignore CR on input.

`mutable c_icrnl : bool ;`
: Map CR to NL on input.

`mutable c_ixon : bool ;`
: Recognize XON/XOFF characters on input.

`mutable c_ixoff : bool ;`
: Emit XON/XOFF chars to control input flow.

`mutable c_opost : bool ;`
: Enable output processing.

`mutable c_obaud : int ;`
: Output baud rate (0 means close connection).

`mutable c_ibaud : int ;`
: Input baud rate.

`mutable c_csize : int ;`
: Number of bits per character (5-8).

`mutable c_cstopb : int ;`
: Number of stop bits (1-2).

`mutable c_cread : bool ;`
: Reception is enabled.

`mutable c_parenb : bool ;`
: Enable parity generation and detection.

`mutable c_parodd : bool ;`
: Specify odd parity instead of even.

`mutable c_hupcl : bool ;`
: Hang up on last close.

`mutable c_clocal : bool ;`
: Ignore modem status lines.

```
  mutable c_isig : bool ;
```
Generate signal on INTR, QUIT, SUSP.

```
  mutable c_icanon : bool ;
```
Enable canonical processing (line buffering and editing)

```
  mutable c_noflsh : bool ;
```
Disable flush after INTR, QUIT, SUSP.

```
  mutable c_echo : bool ;
```
Echo input characters.

```
  mutable c_echoe : bool ;
```
Echo ERASE (to erase previous character).

```
  mutable c_echok : bool ;
```
Echo KILL (to erase the current line).

```
  mutable c_echonl : bool ;
```
Echo NL even if c_echo is not set.

```
  mutable c_vintr : char ;
```
Interrupt character (usually ctrl-C).

```
  mutable c_vquit : char ;
```
Quit character (usually ctrl-\).

```
  mutable c_verase : char ;
```
Erase character (usually DEL or ctrl-H).

```
  mutable c_vkill : char ;
```
Kill line character (usually ctrl-U).

```
  mutable c_veof : char ;
```
End-of-file character (usually ctrl-D).

```
  mutable c_veol : char ;
```
Alternate end-of-line char. (usually none).

```
  mutable c_vmin : int ;
```
Minimum number of characters to read before the read request is satisfied.

```
  mutable c_vtime : int ;
```
Maximum read wait (in 0.1s units).

```
  mutable c_vstart : char ;
```
Start character (usually ctrl-Q).

```
  mutable c_vstop : char ;
```
Stop character (usually ctrl-S).

```
}
val tcgetattr : file_descr -> terminal_io
```
Return the status of the terminal referred to by the given file descriptor.

```
type setattr_when =
  | TCSANOW
  | TCSADRAIN
  | TCSAFLUSH
val tcsetattr : file_descr -> setattr_when -> terminal_io -> unit
```

Set the status of the terminal referred to by the given file descriptor. The second argument indicates when the status change takes place: immediately (`TCSANOW`), when all pending output has been transmitted (`TCSADRAIN`), or after flushing all input that has been received but not read (`TCSAFLUSH`). `TCSADRAIN` is recommended when changing the output parameters; `TCSAFLUSH`, when changing the input parameters.

```
val tcsendbreak : file_descr -> int -> unit
```

Send a break condition on the given file descriptor. The second argument is the duration of the break, in 0.1s units; 0 means standard duration (0.25s).

```
val tcdrain : file_descr -> unit
```

Waits until all output written on the given file descriptor has been transmitted.

```
type flush_queue =
  | TCIFLUSH
  | TCOFLUSH
  | TCIOFLUSH
```

```
val tcflush : file_descr -> flush_queue -> unit
```

Discard data written on the given file descriptor but not yet transmitted, or data received but not yet read, depending on the second argument: `TCIFLUSH` flushes data received but not read, `TCOFLUSH` flushes data written but not transmitted, and `TCIOFLUSH` flushes both.

```
type flow_action =
  | TCOOFF
  | TCOON
  | TCIOFF
  | TCION
```

```
val tcflow : file_descr -> flow_action -> unit
```

Suspend or restart reception or transmission of data on the given file descriptor, depending on the second argument: `TCOOFF` suspends output, `TCOON` restarts output, `TCIOFF` transmits a STOP character to suspend input, and `TCION` transmits a START character to restart input.

```
val setsid : unit -> int
```

Put the calling process in a new session and detach it from its controlling terminal.

B.3 Module `Thread` : Lightweight threads for Posix `1003.1c` and Win32.

```
type t
```

The type of thread handles.

Thread creation and termination

```
val create : ('a -> 'b) -> 'a -> t
```

`Thread.create funct arg` creates a new thread of control, in which the function application `funct arg` is executed concurrently with the other threads of the program. The application of `Thread.create` returns the handle of the newly created thread. The new thread terminates when the application `funct arg` returns, either normally or by raising an uncaught exception. In the latter case, the exception is printed on standard error, but not propagated back to the parent thread. Similarly, the result of the application `funct arg` is discarded and not directly accessible to the parent thread.

`val self : unit -> t`
Return the thread currently executing.

`val id : t -> int`
Return the identifier of the given thread. A thread identifier is an integer that identifies uniquely the thread. It can be used to build data structures indexed by threads.

`val exit : unit -> unit`
Terminate prematurely the currently executing thread.

`val kill : t -> unit`
Terminate prematurely the thread whose handle is given.

Suspending threads

`val delay : float -> unit`
`delay d` suspends the execution of the calling thread for `d` seconds. The other program threads continue to run during this time.

`val join : t -> unit`
`join th` suspends the execution of the calling thread until the thread `th` has terminated.

`val wait_read : Unix.file_descr -> unit`
See `Thread.wait_write`.

`val wait_write : Unix.file_descr -> unit`
This function does nothing in this implementation.

`val wait_timed_read : Unix.file_descr -> float -> bool`
See `Thread.wait_timed_read`.

`val wait_timed_write : Unix.file_descr -> float -> bool`
Suspend the execution of the calling thread until at least one character is available for reading (`wait_read`) or one character can be written without blocking (`wait_write`) on the given Unix file descriptor. Wait for at most the amount of time given as second argument (in seconds). Return `true` if the file descriptor is ready for input/output and `false` if the timeout expired.
These functions return immediately `true` in the Win32 implementation.

```
val select :
  Unix.file_descr list ->
  Unix.file_descr list ->
  Unix.file_descr list ->
  float -> Unix.file_descr list * Unix.file_descr list * Unix.file_descr list
```

Suspend the execution of the calling thead until input/output becomes possible on the given Unix file descriptors. The arguments and results have the same meaning as for `Unix.select`. This function is not implemented yet under Win32.

`val wait_pid : int -> int * Unix.process_status`

`wait_pid p` suspends the execution of the calling thread until the process specified by the process identifier `p` terminates. Returns the pid of the child caught and its termination status, as per `Unix.wait`. This function is not implemented under MacOS.

`val yield : unit -> unit`

Re-schedule the calling thread without suspending it. This function can be used to give scheduling hints, telling the scheduler that now is a good time to switch to other threads.

Management of signals

Signal handling follows the POSIX thread model: signals generated by a thread are delivered to that thread; signals generated externally are delivered to one of the threads that does not block it. Each thread possesses a set of blocked signals, which can be modified using `Thread.sigmask`. This set is inherited at thread creation time. Per-thread signal masks are supported only by the system thread library under Unix, but not under Win32, nor by the VM thread library.

`val sigmask : Unix.sigprocmask_command -> int list -> int list`

`sigmask cmd sigs` changes the set of blocked signals for the calling thread. If `cmd` is `SIG_SETMASK`, blocked signals are set to those in the list `sigs`. If `cmd` is `SIG_BLOCK`, the signals in `sigs` are added to the set of blocked signals. If `cmd` is `SIG_UNBLOCK`, the signals in `sigs` are removed from the set of blocked signals. `sigmask` returns the set of previously blocked signals for the thread.

`val wait_signal : int list -> int`

`wait_signal sigs` suspends the execution of the calling thread until the process receives one of the signals specified in the list `sigs`. It then returns the number of the signal received. Signal handlers attached to the signals in `sigs` will not be invoked. The signals `sigs` are expected to be blocked before calling `wait_signal`.

B.4 Module `Mutex` : Locks for mutual exclusion.

Mutexes (mutual-exclusion locks) are used to implement critical sections and protect shared mutable data structures against concurrent accesses. The typical use is (if `m` is the mutex associated with the data structure `D`):

```
Mutex.lock m;
(* Critical section that operates over D *);
Mutex.unlock m
```

`type t`

The type of mutexes.

`val create : unit -> t`

Return a new mutex.

`val lock : t -> unit`

Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.

`val try_lock : t -> bool`

Same as `Mutex.lock`, but does not suspend the calling thread if the mutex is already locked: just return `false` immediately in that case. If the mutex is unlocked, lock it and return `true`.

`val unlock : t -> unit`

Unlock the given mutex. Other threads suspended trying to lock the mutex will restart.

B.5 Module `Condition` : Condition variables to synchronize between threads.

Condition variables are used when one thread wants to wait until another thread has finished doing something: the former thread "waits" on the condition variable, the latter thread "signals" the condition when it is done. Condition variables should always be protected by a mutex. The typical use is (if `D` is a shared data structure, `m` its mutex, and `c` is a condition variable):

```
Mutex.lock m;
while (* some predicate P over D is not satisfied *) do
  Condition.wait c m
done;
(* Modify D *)
if (* the predicate P over D is now satified *) then Condition.signal c;
Mutex.unlock m
```

`type t`

The type of condition variables.

`val create : unit -> t`

Return a new condition variable.

`val wait : t -> Mutex.t -> unit`

`wait c m` atomically unlocks the mutex `m` and suspends the calling process on the condition variable `c`. The process will restart after the condition variable `c` has been signalled. The mutex `m` is locked again before `wait` returns.

`val signal : t -> unit`

`signal c` restarts one of the processes waiting on the condition variable `c`.

`val broadcast : t -> unit`

`broadcast c` restarts all processes waiting on the condition variable `c`.

B.6 Module `Event` : First-class synchronous communication.

This module implements synchronous inter-thread communications over channels. As in John Reppy's Concurrent ML system, the communication events are first-class values: they can be built and combined independently before being offered for communication.

`type 'a channel`

The type of communication channels carrying values of type `'a`.

`val new_channel : unit -> 'a channel`

Return a new channel.

`type +'a event`

The type of communication events returning a result of type `'a`.

`val send : 'a channel -> 'a -> unit event`

`send ch v` returns the event consisting in sending the value `v` over the channel `ch`. The result value of this event is `()`.

`val receive : 'a channel -> 'a event`

`receive ch` returns the event consisting in receiving a value from the channel `ch`. The result value of this event is the value received.

`val always : 'a -> 'a event`

`always v` returns an event that is always ready for synchronization. The result value of this event is `v`.

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

`choose evl` returns the event that is the alternative of all the events in the list `evl`.

`val wrap : 'a event -> ('a -> 'b) -> 'b event`

`wrap ev fn` returns the event that performs the same communications as `ev`, then applies the post-processing function `fn` on the return value.

`val wrap_abort : 'a event -> (unit -> unit) -> 'a event`

`wrap_abort ev fn` returns the event that performs the same communications as `ev`, but if it is not selected the function `fn` is called after the synchronization.

`val guard : (unit -> 'a event) -> 'a event`

`guard fn` returns the event that, when synchronized, computes `fn()` and behaves as the resulting event. This allows to compute events with side-effects at the time of the synchronization operation.

`val sync : 'a event -> 'a`

"Synchronize" on an event: offer all the communication possibilities specified in the event to the outside world, and block until one of the communications succeed. The result value of that communication is returned.

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

"Synchronize" on an alternative of events. `select evl` is shorthand for `sync(choose evl)`.

`val poll : 'a event -> 'a option`

Non-blocking version of `Event.sync`: offer all the communication possibilities specified in the event to the outside world, and if one can take place immediately, perform it and return `Some r` where `r` is the result value of that communication. Otherwise, return `None` without blocking.

B.7 Module `Misc` : miscelleaneous functions for the Unix library

Finalization

`val try_finalize : ('a -> 'b) -> 'a -> ('c -> unit) -> 'c -> 'b`

`try_finalize f x g y` applies the main code `f` to `x` and the result after having executed the finalization code `g` applied to `y`. If the main code raises the exception `exn` the finalization finalization code and `exn` is raised. If the finalization code itself fails, the exception return is always the one of the finalization code.

`val iter_dir : (string -> 'a) -> string -> unit`

`iter_dir f d` opens path `d` as a directory and iterates the function `f` over all its entries

`val restart_on_EINTR : ('a -> 'b) -> 'a -> 'b`

`restart_on_EINTR f x` calls `f x` repeatedly until it does not fails with `EINTR`

`val free_children : int -> unit`

`free_children signal` free all zombies of the current processus, discarding their exit status.

`val system : string -> process_status`

`system cmd` behaves as `Unix.system cmd` except that `sigchld` is blocked and `sigint` and `sigquit` are ignored during the excecution of the command `cmd`

`val exec_as_system : ('a -> process_status) -> 'a -> process_status`

`system exec args` behaves as `system cmd` except that it takes as arguments a function `exec` and arguments `args` to be passed to `exec` to start in the new process. In particular, `system` is an abbreviation for `exec_as_system (execv /bin/sh) [|-c; cmd; |] v`.

`val really_write : file_descr -> string -> int -> int -> unit`

as `single_write` but restarts on `EINTR` until all bytes have been written. When an error occurs, some unknown number of bytes may have been written. Hence, an error should in general be considered as fatal.

`val retransmit : file_descr -> file_descr -> unit`

`val install_tcp_server_socket : sockaddr -> file_descr`

`install_tcp_server_socket sockaddr` creates a socket in the internet domain, binds it to the address `sockaddr`, listens to it, and returns it.

```
val tcp_server :
  (file_descr -> file_descr * sockaddr -> 'a) ->
  sockaddr -> unit
```

`tcp_server f addr` installs the tcp service `f` at the internet domain address `addr`. For each connection to the service, the function `f` receives the server's socket and the client's socket-address pair as parametters. It is the responsability of `f` to close the client connection when done.

```
val sequential_treatment :
  file_descr ->
  (file_descr * sockaddr -> unit) ->
  file_descr * sockaddr -> unit
```

`sequential_treatment server service client` runs `service` provided on server for one client. The server is given by its socket `socket` and the client is given by its socket-address pair `client`. After initialization the service is performed by applying `service` to `client`. The treatment is sequential, that is, the function only returns when the service is completed.

```
val fork_treatment :
  file_descr ->
  (file_descr * sockaddr -> unit) ->
  file_descr * sockaddr -> unit
```

same as `sequential_treament` but the treatment is concurrently performed by a forked child process of the server. The parent process will have to free the child when completed.

```
val double_fork_treatment :
  file_descr ->
  (file_descr * sockaddr -> unit) ->
  file_descr * sockaddr -> unit
```

same as `fork_treament` but the treatment is performed after a double fork. The forked process will be freed automatically on completion.

```
val co_treatment :
  file_descr ->
  (file_descr * sockaddr -> unit) ->
  file_descr * sockaddr -> unit
```

same as `sequential_treament` but the treatment is concurrently performed by another thread. It is the responsability of the treatment to close file_descr when completed.

```
val run_with_lock : Mutex.t -> ('a -> 'b) -> 'a -> 'b
```

Appendix C

Index

Les pages en italiques comme *64* renvoient à des explications dans le corps du document; les pages en gras comme **83** renvoient à des occurrences essentielles correspondants à une fonction *Posix*; les pages en fonte romanes comme 127 à la documentation.