Université de Strasbourg
Systèmes d'exploitation

# Notice simplifiée

# Primitives système et fonctions de bibliothèque

# 2021 – 2022 (automne)



# Table des matières

# Chapitre 1

# Les primitives système

L'accès aux services du système Unix est réalisé au moyen des *primitives système*, encore appelées *appels système*.

Ces primitives sont l'outil de plus bas niveau dont dispose le programmeur. Quand cela est possible, il est généralement préférable d'utiliser les fonctions de la bibliothèque standard.

La liste ci-dessous donne une classification de ces primitives par catégorie. Ce sont :

— les numéros d'erreur (catégorie spéciale),
— l'accès aux fichiers,
— la gestion des processus,
— les tubes, les signaux V7 et POSIX,
— l'horloge du système,
— les disques et les périphériques,
— les sockets Berkeley,
— les IPC System V, et
— les éternels inclassables...

Cette classification ne prétend pas être rigoureuse. Certaines primitives auraient pu être dans plusieurs catégories, il a bien fallu choisir. D'autre part, cette liste n'est pas exhaustive, mais elle donne néanmoins toutes les primitives utiles à l'élaboration des applications.

## 1.1 Valeurs limites

Une implémentation compatible avec la norme POSIX définit un certain nombre de limites sous forme de constantes symboliques ou de macros via le fichier `limits.h`. POSIX définit des valeurs mimimales, mais chaque implémentation peut définir des valeurs moins restrictives. Un programme parfaitement portable ne doit donc pas supposer que les valeurs sont plus élevées que les minimales POSIX.

Les implémentations peuvent faire varier certaines limites soit parce que l'administrateur peut en modifier la valeur, soit parce qu'elles dépendent du système de fichiers sous-jacent. Dans ces cas, la macro correspondante n'est pas définie dans `limits.h`, ce qui peut être testé avec `#if`, et la valeur doit donc être obtenue lors de l'exécution avec les primitives `sysconf`, `pathconf` ou `fpathconf` (page 47). À titre d'exemple, le tableau ci-dessous indique quelques unes de ces limites, la valeur minimale imposée par POSIX ainsi que la valeur définie par Linux sur l'architecture x86 :

| Constante | Signification | Min POSIX | Linux x86 |
|---|---|---|---|
| `PATH_MAX` | taille maximum d'un chemin | 256 | 4096 |
| `NAME_MAX` | taille maximum d'un composant de chemin | 14 | 255 |
| `PAGESIZE` | taille d'une page | 1 | 4096 |
| `ARG_MAX` | taille maximum de l'ensemble des arguments admis par `exec` | 4096 | 2097152 |
| `HOST_NAME_MAX` | longueur maximum d'un nom de hôte (hors `\0` de fin) | 255 | 64 |
| `LOGIN_NAME_MAX` | longueur maximum d'un nom d'utilisateur | 9 | 256 |
| `OPEN_MAX` | nombre maximum de descripteurs de fichiers | 20 | 1024 |
| `PIPE_BUF` | taille maximum d'une écriture atomique dans un tube | 512 | 4096 |
| `SEM_NSEMS_MAX` | nombre maximum de sémaphores d'un processus | 256 | non limité |
| `CHILD_MAX` | nombre maximum de processus fils | 25 | 62618 |

## 1.2 Définition des erreurs

La plupart des primitives système renvoient une valeur au programme appelant. La plupart du temps, il s'agit d'une valeur positive ou nulle pour une exécution normale, ou de la valeur -1 lorsqu'il y a eu une erreur. Dans ce dernier cas, la variable globale `errno` indique le numéro de l'erreur intervenue, et permet donc une plus grande précision dans le diagnostic.

La déclaration de la variable `errno` est :

```
#include <errno.h>
extern int errno ;
```

L'accès à la signification en clair de ces messages d'erreur se fait souvent par l'intermédiaire des fonctions `perror` ou `strerror` de la bibliothèque standard.

La liste suivante est la description des principales erreurs pouvant intervenir. Il faut noter que la variable `errno` n'est pas remise à 0 avant une primitive système. Ceci implique donc que sa valeur ne doit être testée que si l'appel signale une erreur au moyen de la valeur de retour.

`EPERM` : Not owner
Survient typiquement lors d'une tentative de modification d'une donnée interdite.

`ENOENT` : No such file or directory
Survient lorsqu'un fichier est spécifié et que ce fichier n'existe pas, ou lorsqu'un chemin d'accès un fichier spécifie un répertoire inexistant.

`ESRCH` : No such process
Survient lorsque le processus ne peut être trouvé par `kill`, `ptrace`, ou lorsque le processus n'est pas accessible.

`EINTR` : Interrupted system call
Un signal asynchrone (interruption de l'utilisateur par exemple) traité par l'utilisateur est arrivé pendant une primitive système. Si l'exécution reprend après la primitive système, la valeur de retour indiquera une erreur.

`EIO` : I/O error
Survient lors d'une erreur d'entrée sortie.

`ENXIO` : No such device or address
Survient lorsqu'une entrée sortie est demandé sur un périphérique qui n'est pas en ligne ou lorsque l'entrée sortie dépasse la capacité du périphérique.

`E2BIG` : Arg list too long
Survient lorsqu'une liste d'arguments ou d'environnement est plus grande que la longueur supportée lors d'un `exec`.

`ENOEXEC` : Exec format error
Survient lorsqu'`exec` ne reconnaît pas le format du fichier demandé.

`EBADF` : Bad file number
Survient lorsqu'un descripteur de fichier ne réfère aucun fichier ouvert, ou lorsqu'une écriture est demandée sur un fichier ouvert en lecture seule, ou vice-versa.

`ECHILD` : No child process
Survient lorsqu'un `wait` est demandé et qu'aucun processus fils n'existe.

`EAGAIN` : No more process
Survient lorsque `fork` ne peut trouver une entrée disponible dans la table des processus, ou lorsque l'utilisateur a dépassé son quota de processus.

`ENOMEM` : Not enough space
Survient lorsqu'un processus demande plus de place que ce que le système est capable de lui fournir.

`EACCES` : Permission denied
Une tentative a été faite pour accéder à un fichier interdit.

`EFAULT` : Bad address
Survient lorsque le système a généré une trappe matérielle en essayant d'utiliser un mauvais argument d'un appel.

`ENOTBLK` : Block device requested

Survient lorsqu'un fichier de type autre que *block device* est transmis à `mount`.

`EBUSY` : Device or resource busy
Survient lorsqu'un périphérique ou une ressource est déjà occupée (fichier actif lors d'un `mount` par exemple), ou lorsqu'un mode est déjà activé (`acct`).

`EEXIST` : File exists
Survient lorsqu'un fichier existant est mentionné dans un contexte non approprié (par exemple `link`).

`EXDEV` : Cross-device link
Survient lorsqu'un lien est demandé entre deux systèmes de fichiers.

`ENODEV` : No such device
Survient lorsqu'une primitive système est inappropriée au périphérique, comme par exemple une lecture sur un périphérique en écriture seulement.

`ENOTDIR` : Not a directory
Survient lorsqu'un répertoire est nécessaire et n'est pas fourni à la primitive système, comme par exemple un argument à `chdir`.

`EISDIR` : Is a directory
Survient lors d'une tentative d'ouverture dans un fichier de type répertoire.

`EINVAL` : Invalid argument
Quelques arguments invalides et inclassables.

`ENFILE` : File table overflow
La table centrale des fichiers ouverts est pleine, et il n'est plus possible pour l'instant d'accepter des `open`.

`EMFILE` : Too many open files
Survient lorsqu'un processus essaye de dépasser son quota de fichiers ouverts.

`ENOTTY` : Not a typewriter
Survient lorsque la primitive `ioctl` est inappropriée pour le périphérique.

`ETXTBSY` : Text file busy
Survient lors d'une tentative d'exécution d'un fichier exécutable ouvert en écriture, ou vice-versa.

`EFBIG` : File too large
Survient lorsque la taille du fichier dépasse la taille maximum permise par le système.

`ENOSPC` : No space left on device
Survient lors d'une écriture dans un fichier ordinaire, quand il n'y a plus de place sur le périphérique.

`ESPIPE` : Illegal seek
Survient lors d'un `lseek` sur un tube.

`EROFS` : Read-only file system
Survient lors d'une tentative de modification sur un fichier ou un répertoire d'un système de fichiers monté en lecture seulement.

`EMLINK` : Too many links
Survient lorsqu'un `link` dépasse 1000 liens pour le fichier.

`EPIPE` : Broken pipe
Survient lors d'une écriture dans un tube sans lecteur. Cette condition génère normalement un signal. L'erreur est renvoyée si le signal est ignoré.

## 1.3 Définition des types

Certaines primitives et fonctions de bibliothèque utilisent des paramètres représentant des identificateurs de processus, des numéros d'utilisateur ou d'autres informations. Même s'il s'agit le plus souvent d'entiers, il faut utiliser les types définis par POSIX (ou par la norme du C pour certains) pour faciliter la lisibilité et la portabilité des programmes. Pour fixer les idées, voici les principaux types utilisés :

— `clock_t` (entier ou flottant) : nombre de tops d'horloge (voir `sysconf`, page 47)
— `dev_t` (entier) : numéro (mineur et majeur) de périphérique
— `gid_t` (entier) : identificateur de groupe

— `ino_t` (entier non signé) : numéro d'inode
— `jmp_buf` (tableau) : buffer pour `setjmp` et `longjmp`
— `key_t` (entier) : clef utilisée pour les IPC System V
— `mode_t` (entier) : permissions associées à un fichier
— `off_t` (entier signé) : taille d'un fichier ou déplacement dans un fichier
— `pid_t` (entier signé) : identificateur de processus
— `ptrdiff_t` (entier) : différence entre deux adresses
— `sigset_t` (tableau) : bitmap utilisée pour les signaux POSIX
— `size_t` (entier non signé) : nombre d'octets
— `ssize_t` (entier signé) : nombre d'octets signé
— `time_t` (entier) : nombre de secondes écoulées depuis le premier janvier 1970.
— `uid_t` (entier) : identificateur d'utilisateur
— `intmax_t` : la plus grande taille d'entier, pouvant contenir toute donnée d'un des types précédents
— `socklen_t` : taille d'une adresse, utilisée dans les primitives traitant des sockets
— `sa_family_t` : famille d'adresses, utilisée notamment dans les `struct sockaddr` et assimilées

La norme POSIX laisse le choix des tailles exactes à la discrétion de chaque implémentation ; elle ne fait que préciser la caractéristique (entier ou flottant, signé, non signé ou non spécifié) sans préciser davantage. Voir 2.1 (page 55) pour les formats d'affichage de ces types.

## 1.4 Accès aux fichiers

Les primitives d'accès aux fichiers sont de deux ordres :
— accès au contenu d'un fichier,
— accès au descripteur (*inode*).
Il y a deux façons d'accéder à un fichier. La première nécessite un chemin d'accès au fichier, c'est-à-dire son nom (chemin absolu ou relatif) sous forme d'une chaîne de caractères terminée par un octet nul. La deuxième nécessite l'ouverture préalable du fichier, c'est-à-dire nécessite un petit entier obtenu par les primitives `open`, `creat`, `dup`, `fcntl` ou `pipe`.

La deuxième méthode est la seule possible pour accéder aux données d'un fichier. Trois fichiers sont automatiquement ouverts : 0 pour l'entrée standard, 1 pour la sortie standard, et 2 pour les erreurs.

---

**access** — détermine l'accessibilité d'un fichier

```
#include <unistd.h>

int access (const char *chemin, int motif)
```

La primitive `access` vérifie l'accessibilité du fichier en comparant le motif binaire avec les protections du fichier. La comparaison est réalisée avec les identificateurs d'utilisateur et de groupe *réels* et non avec les identificateurs *effectifs*.

Le motif binaire est construit à partir des bits suivants :

| | | |
|---|---|---|
| `R_OK` | 04 | lecture |
| `W_OK` | 02 | écriture |
| `X_OK` | 01 | exécution |
| `F_OK` | 00 | vérifier que le fichier existe |

Par exemple, `access("toto",R_OK|W_OK)` teste à la fois l'accessibilité en lecture et en écriture au fichier de nom `toto`.

Cette primitive renvoie 0 si l'accès est permis ou -1 en cas d'erreur.

---

**chmod** — change les protections d'un fichier

```
#include <sys/stat.h>

int chmod (const char *chemin, mode_t mode)
```

La primitive `chmod` change les droits d'accès d'un fichier, suivant le motif binaire contenu dans `mode` :

| | | |
|---|---|---|
| S_ISUID | 04000 | bit *set user id* |
| S_ISGID | 02000 | bit *set group id* |
| S_ISVTX | 01000 | bit *sticky bit* |
| S_IRWXU | 00700 | droits pour le propriétaire |
| S_IRUSR | 00400 | lecture pour le propriétaire |
| S_IWUSR | 00200 | écriture pour le propriétaire |
| S_IXUSR | 00100 | exécution pour le propriétaire |
| S_IRWXG | 00070 | lecture, écriture et exécution pour le groupe |
| S_IRGRP | 00040 | lecture pour le groupe |
| S_IWGRP | 00020 | écriture pour le groupe |
| S_IXGRP | 00010 | exécution pour le groupe |
| S_IRWXO | 00007 | lecture, écriture et exécution pour les autres |
| S_IROTH | 00004 | lecture pour les autres |
| S_IWOTH | 00002 | écriture pour les autres |
| S_IXOTH | 00001 | exécution pour les autres |

Par exemple, `chmod("toto",S_IRUSR|S_IWUSR|S_IRGRP)` est équivalent à `chmod("toto",0640)`. Toutefois, l'usage des valeurs numériques est tellement répandu que POSIX a exceptionnellement normalisé ces valeurs.

Seul le propriétaire du fichier (ou le super-utilisateur) a le droit de modifier les droits d'accès d'un fichier.

Cette primitive renvoie 0 en cas de modification réussie ou -1 en cas d'erreur.

---

**chown, lchown** — change le propriétaire d'un fichier

```
#include <unistd.h>

int chown (const char *chemin, uid_t proprietaire, gid_t groupe)
int lchown (const char *chemin, uid_t proprietaire, gid_t groupe)
```

La primitive `chown` change le propriétaire et le groupe d'un fichier. Seul le propriétaire d'un fichier (ou le super utilisateur) peut changer ces informations. Passer `-1` pour le propriétaire ou pour le groupe indique de ne pas changer la valeur.

Lorsque le chemin passé est un lien symbolique, `chown` modifie le fichier référencé par le lien, alors que `lchown` modifie le lien lui-même.

Ces primitives renvoient 0 en cas de modification réussie ou -1 en cas d'erreur.

---

**close** — ferme un fichier

```
#include <unistd.h>

int close (int desc)
```

La primitive `close` ferme le fichier associé au descripteur `desc`, obtenu par `open`, `creat`, `dup`, `fcntl` ou `pipe`.

Cette primitive renvoie 0 en cas de modification réussie ou -1 en cas d'erreur.

---

**creat** — crée et ouvre un fichier

```
#include <sys/stat.h>
#include <fcntl.h>

int creat (const char *chemin, mode_t mode)
```

La fonction `creat` était autrefois une primitive utilisée pour créer un nouveau fichier et l'ouvrir en écriture. Elle est aujourd'hui redondante avec `open`.

---

**dup, dup2** — duplique un descripteur de fichier

```
#include <unistd.h>

int dup (int ancien)
int dup2 (int ancien, int nouveau)
```

La primitive `dup` duplique le descripteur de fichier `ancien` (obtenu par `open`, `creat`, `dup`, `fcntl` ou `pipe`), et retourne le nouveau descripteur, partageant les caractéristiques suivantes avec l'original :

— même fichier ouvert,
— même pointeur de fichier,
— même mode d'accès (lecture, écriture), et
— même état de fichier (voir `fcntl`).

Le nouveau descripteur retourné est le plus petit disponible, et est marqué comme *préservé lors d'un exec*.

La primitive `dup2` duplique le descripteur `ancien` et l'affecte au descripteur `nouveau`. Si le descripteur `nouveau` référençait un fichier ouvert, celui-ci est d'abord fermé.

Ces primitives renvoient le nouveau descripteur de fichier (non négatif) en cas de duplication réussie, ou -1 en cas d'erreur.

---

**fcntl** — contrôle un fichier ouvert

```
#include <fcntl.h>

int fcntl (int desc, int cmd, ...)
```

La primitive `fcntl` fournit un moyen d'agir sur des fichiers ouverts par l'intermédiaire de `desc` (obtenu par `open`, `creat`, `dup`, `fcntl` ou `pipe`).

Les commandes disponibles sont :

— `F_DUPFD` : renvoie un nouveau descripteur de fichier, de manière comparable à `dup`.
— `F_GETFD` : renvoie le flag *fermeture lors d'un exec*. Si le bit de poids faible est nul, le fichier restera ouvert lors d'un `exec`.
— `F_SETFD` : modifie le flag *fermeture lors d'un exec*, suivant la valeur ci-dessus.
— `F_GETFL` : renvoie les flags du fichier.
— `F_SETFL` : modifie les flags du fichier.
— `F_GETLK` : lit le premier verrou qui bloque l'accès à la portion de fichier décrite par le troisième argument (de type pointeur sur `struct flock`), et retourne ses caractéristiques à la place.
— `F_SETLK` : pose (ou annule) un verrou sur une portion du fichier, décrite par le troisième argument (de type pointeur sur `struct flock`). Si le verrou ne peut être modifié immédiatement, `fcntl` renvoie -1.
— `F_SETLKW` : pose (ou annule) un verrou sur une portion du fichier, décrite par le troisième argument (de type pointeur sur `struct flock`). Si le verrou ne peut être modifié immédiatement, `fcntl` attend que le verrou précédent soit libéré.

Les flags du fichier qui peuvent être lus ou modifiés sont :

| | |
|---|---|
| `O_RDONLY` | ouverture en lecture seulement |
| `O_WRONLY` | ouverture en écriture seulement |
| `O_RDWR` | ouverture en lecture et en écriture |
| `O_NDELAY` | mode non bloquant |
| `O_APPEND` | accès uniquement à la fin du fichier |
| `O_SYNCIO` | accès aux fichiers en mode *write through* |

Les verrous sont décrits par une structure `flock`, dont les champs sont les suivants :

```
short   l_type;     // F_RDLCK, F_WRLCK ou F_UNLCK
short   l_whence;   // origine du déplacement - voir lseek
off_t   l_start;    // déplacement relatif en octets
off_t   l_len;      // taille ou tout le fichier si 0
pid_t   l_pid;      // Processus ayant le verrou (F_GETLK)
```

La valeur retournée par `fcntl` dépend de l'action, mais est toujours non négative en cas de réussite, ou -1 en cas d'erreur.

## **poll** — attente d'un événement sur un descripteur

```
#include <poll.h>

int poll (struct pollfd *fdtab, nfds_t nb, int delai)
```

La primitive `poll` attend un événement sur au moins un des `nb` descripteurs indiqués par le tableau `fdtab`. Un événement correspond généralement à un descripteur devenu prêt à émettre ou recevoir une donnée. Les descripteurs concernés sont le plus souvent associés à des tubes, des périphériques ou des sockets, où les lectures ou écritures peuvent être bloquantes. Cette primitive est similaire à `select` (voir page 37).

La primitive `poll` peut se terminer parce qu'un descripteur est prêt, parce qu'un signal l'interrompt ou parce que le délai indiqué par le paramètre `delai` (exprimé en millisecondes) est atteint. Si le délai est nul, `poll` retourne immédiatement. Si le délai est négatif, l'attente n'est alors pas bornée.

La structure `pollfd` utilisée pour indiquer un descripteur à surveiller possède les champs suivants :

```
int fd;             // numéro du descripteur
short events;       // événements à surveiller (en entrée de poll)
short revents;      // événements détectés (en sortie de poll)
```

Pour utiliser `poll`, il faut indiquer le descripteur avec le champ `fd` ainsi que le ou les événements à surveiller (cf. tableau ci-après) avec le champ `events`. La primitive `poll` indique en sortie dans le champ `revents` les événements qu'elle a détectés.

Les événements (en entrée ou en sortie) sont indiqués par la combinaison des bits suivants :

| | |
|---|---|
| `POLLIN` | descripteur disposant de données prêtes à lire |
| `POLLRDNORM` | descripteur disposant de données normales prêtes à lire |
| `POLLRDBAND` | descripteur disposant de données prioritaires prêtes à lire |
| `POLLPRI` | descripteur disposant de données hautement prioritaires prêtes à lire |
| `POLLOUT` | descripteur prêt à recevoir des données |
| `POLLWRNORM` | descripteur prêt à recevoir des données normales |
| `POLLWRBAND` | descripteur prêt à recevoir des données prioritaires |
| `POLLERR` | `poll` a détecté une erreur |
| `POLLHUP` | `poll` a détecté que le périphérique a été déconnecté |
| `POLLNVAL` | `poll` a détecté un numéro de descripteur invalide |

En cas de succès, `poll` renvoie le nombre de descripteurs pour lesquels un événement a été détecté. Si le délai est atteint sans événement, la valeur 0 est renvoyée. Enfin, la valeur -1 est renvoyée en cas d'erreur.

## **getdirentries** — lit des entrées dans un répertoire

```
#include <dirent.h>

int getdirentries (int desc, struct direct *buf, int taille, off_t *offset)
```

Attention : cette primitive n'étant pas spécifiée par POSIX, les fonctions `opendir`, `readdir` etc. de la bibliothèque doivent être utilisées de préférence (voir page 53). Cette primitive est uniquement citée ici comme exemple d'interface de bas niveau pour comprendre l'implémentation des fonctions de la famille `opendir`.

Cette primitive renvoie des entrées dans un répertoire dans une forme indépendante du format natif du répertoire ouvert par `open`. Les entrées sont placées dans un tableau de structures `direct`, chacune de ces structures contenant :

- `unsigned long d_fileno` : numéro unique du fichier (exemple : numéro d'inode si le fichier est sur un disque local) ;
- `unsigned short d_reclen` : longueur en octets de l'entrée dans le répertoire ;
- `unsigned short d_namlen` : longueur en octets du nom, y compris le caractère nul terminal ;
- `char d_name` : tableau de caractères de longueur `MAXNAMELEN+1` contenant le nom, terminé par un caractère nul.

Le nombre d'entrées dans le tableau `buf` est déduit de la taille en octets `taille` de l'ensemble du tableau `buf`. Cette taille doit être supérieure ou égale à la taille du bloc du système de fichiers.

Le paramètre `offset` contient en retour la position courante du bloc lu.

La valeur de retour est le nombre d'octets transférés en cas d'opération réussie, -1 sinon.

---

**link** — établit un nouveau lien sur un fichier

```
#include <unistd.h>

int link (const char *chemin, const char *nouveauchemin)
```

La primitive `link` crée un nouveau lien physique (à ne pas confondre avec les liens symboliques, voir primitive `symlink` page 21) pour fichier existant de nom `chemin`. Le nouveau lien (la nouvelle entrée dans le répertoire) porte le nom `nouveauchemin`.

Cette primitive renvoie 0 en cas de liaison réussie ou -1 en cas d'erreur.

---

**lseek** — déplace le pointeur de lecture/écriture d'un fichier

```
#include <unistd.h>

off_t lseek (int desc, off_t deplacement, int apartir)
```

La primitive `lseek` déplace le pointeur du fichier repéré par `desc` (obtenu par `open`, `creat`, `dup`, `fcntl` ou `pipe`). Le déplacement est régi par la valeur de `apartir` :

| | | |
|---|---|---|
| `SEEK_SET` | 0 | à partir du début |
| `SEEK_CUR` | 1 | à partir de la position courante |
| `SEEK_END` | 2 | à partir de la fin |

Cette primitive renvoie le nouveau pointeur en cas de déplacement réussi, ou -1 (plus exactement `((off_t)-1)`) en cas d'erreur.

---

**mkdir** — crée un répertoire

```
#include <sys/stat.h>

int mkdir (const char *chemin, mode_t mode)
```

La primitive `mkdir` crée le répertoire de nom `chemin`. Les protections initiales sont spécifiées en binaire avec l'argument `mode` (voir `chmod`).

Cette primitive renvoie 0 en cas de création réussie ou -1 en cas d'erreur.

---

**open** — ouvre un fichier

```
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *chemin, int flags)
int open (const char *chemin, int flags, mode_t mode)
```

La primitive `open` ouvre un fichier et renvoie son descripteur. Si le fichier doit être créé, `mode` spécifie ses protections (voir `chmod`).

L'état du fichier est initialisé à la valeur de `flags`, construite à partir des bits suivants (les trois premiers sont mutuellement exclusifs) :

— `O_RDONLY` : ouverture en lecture seulement,
— `O_WRONLY` : ouverture en écriture seulement,
— `O_RDWR` : ouverture en lecture et écriture,

— O_NDELAY : affectera les lectures ou écritures (voir read et write),
— O_APPEND : le pointeur est déplacé à la fin du fichier,
— O_CREAT : le fichier est créé s'il n'existait pas (dans ce cas, le mode est initialisé à partir du paramètre mode),
— O_TRUNC : si le fichier existe, sa taille est remise à 0,
— O_EXCL : si O_EXCL et O_CREAT sont mis, open échoue si le fichier existe,
— O_NOCTTY : si le fichier est un terminal, open n'essaiera pas de l'utiliser comme terminal de contrôle,
— O_NONBLOCK : le fichier (spécial ou fifo) est ouvert en mode non bloquant,

Cette primitive renvoie le descripteur de fichier (non négatif) en cas d'ouverture réussie, ou -1 en cas d'erreur.

---

**read** — lit dans un fichier

```
#include <unistd.h>

ssize_t read (int desc, void *buf, size_t nombre)
```

La primitive read lit nombre octets dans le fichier associé au descripteur desc (retourné par open, creat, dup, fcntl ou pipe) et les place à partir de l'adresse buf.

La valeur renvoyée est le nombre d'octets lus et stockés dans buf. Cette valeur peut être inférieure à nombre si :
— le descripteur desc est associé à une ligne de communication, ou
— il ne reste pas assez d'octets dans le fichier pour satisfaire la demande.
Deux cas spéciaux :

1. lecture dans un tube vide : si le flag O_NDELAY est spécifié lors de l'ouverture ou avec fcntl, la lecture renverra 0, sinon la lecture sera bloquante jusqu'à ce que le tube ne soit plus vide ou qu'il n'y ait plus de processus qui y écrive.
2. lecture d'une ligne de communications sur laquelle il n'y a pas de données : si le flag O_NDELAY est spécifié, la lecture renverra 0, sinon la lecture sera bloquante jusqu'à ce que des données deviennent disponibles.

Cette primitive renvoie le nombre d'octets lus (non négatif) en cas de lecture réussie, ou -1 en cas d'erreur.

---

**rmdir** — supprime un répertoire

```
#include <unistd.h>

int rmdir (const char *chemin)
```

La primitive rmdir supprime le répertoire (qui doit être vide) spécifié par l'argument chemin.

Cette primitive renvoie 0 en cas de suppression réussie ou -1 en cas d'erreur.

---

**stat, lstat, fstat** — consulte le descripteur d'un fichier

```
#include <sys/stat.h>

int stat (const char *chemin, struct stat *buf)
int lstat (const char *chemin, struct stat *buf)
int fstat (int desc, struct stat *buf)
```

Les primitives stat, lstat et fstat lisent la partie système d'un fichier (son *inode*) et placent ces informations dans la structure pointée par buf. Le fichier peut être spécifié par son chemin (stat et lstat) ou par le descripteur de fichier qui lui est associé (fstat).

Lorsque le fichier est un lien symbolique (voir la primitive symlink, page 21), stat et fstat renvoient des informations sur la cible du lien alors que lstat renvoie des informations sur le lien lui-même.

Le contenu de la structure buf est défini comme suit :

```
dev_t     st_dev ;
ino_t     st_ino ;
mode_t    st_mode ;
nlink_t   st_nlink ;
uid_t     st_uid ;
gid_t     st_gid ;
dev_t     st_rdev ;
off_t     st_size ;
time_t    st_atime ;
time_t    st_mtime ;
time_t    st_ctime ;
```

— `st_dev` : le périphérique contenant le fichier,
— `st_ino` : numéro d'inode,
— `st_mode` : type et protections du fichier
— `st_nlink` : nombre de liens,
— `st_uid` et `st_gid` : numéros de propriétaire et de groupe,
— `st_rdev` : identification du périphérique dans le cas d'un fichier spécial (bloc ou caractère),
— `st_size` : taille du fichier en octets,
— `st_blksize` : taille optimale des entrées/sorties
— `st_atime` : date du dernier accès (`creat`, `mknod`, `pipe`, `utime` et `read`).
— `st_mtime` : date de la dernière modification (`creat`, `mknod`, `pipe`, `utime` et `write`).
— `st_ctime` : date de la dernière modification de l'état du fichier (`chmod`, `chown`, `creat`, `link`, `mknod`, `pipe`, `rmdir`, `unlink`, `utime` et `write`).

Le champ `st_mode` est composé de :

— 12 bits de poids faible pour les droits d'accès au fichier (voir `chmod`)
— plusieurs bits de poids fort pour le type du fichier. Ces bits peuvent être extraits avec le masque binaire `S_IFMT` (dans `stat.h`) et comparés avec les valeurs suivantes (les valeurs numériques sont fournies pour l'exemple et ne sont pas valides sur toutes les implémentations) :

| | | |
|---|---|---|
| `S_IFREG` | 0100000 | fichier ordinaire |
| `S_IFBLK` | 0060000 | fichier périphérique (mode bloc) |
| `S_IFCHR` | 0020000 | fichier périphérique (mode caractère) |
| `S_IFDIR` | 0040000 | répertoire |
| `S_IFIFO` | 0010000 | tube ou tube nommé |
| `S_IFLNK` | 0120000 | lien symbolique |
| `S_IFSOCK` | 0140000 | socket |

Il est également possible de tester le type du fichier avec :

| | |
|---|---|
| `S_ISREG(`*mode*`)` | fichier ordinaire |
| `S_ISBLK(`*mode*`)` | fichier périphérique (mode bloc) |
| `S_ISCHR(`*mode*`)` | fichier périphérique (mode caractère) |
| `S_ISDIR(`*mode*`)` | répertoire |
| `S_ISFIFO(`*mode*`)` | tube ou tube nommé |
| `S_ISLNK(`*mode*`)` | lien symbolique |
| `S_ISSOCK(`*mode*`)` | socket |

Ces primitives renvoient 0 en cas d'accès réussi ou -1 en cas d'erreur.

---

**truncate, ftruncate** — tronque la longueur d'un fichier

```
#include <unistd.h>

int truncate (const char *chemin, off_t lg)
int ftruncate (int desc, off_t lg)
```

Les primitives `truncate` et `ftruncate` tronquent un fichier à la taille spécifiée par le paramètre `lg`. Si l'opération a pour effet de réduire le fichier, les données tronquées sont perdues. Si l'opération a pour effet d'augmenter la taille du fichier, tout se passe comme si le fichier avait complété avec des octets nuls. Le fichier est spécifié par son chemin avec `truncate`, ou par un descripteur de fichier ouvert pour `ftruncate`.

Ces primitives renvoient 0 en cas de succès ou -1 en cas d'erreur.

**rename** — renomme un fichier ou un répertoire

```
#include <stdio.h>

int rename (const char *chemin, const char *nouveauchemin)
```

La primitive `rename` renomme le fichier ou répertoire existant de nom `chemin`. Le nouveau nom est indiqué par `nouveauchemin`.

Cette primitive renvoie 0 en cas de renommage réussi ou -1 en cas d'erreur.

---

**unlink** — supprime le fichier (enlève un lien)

```
#include <unistd.h>

int unlink (const char *chemin)
```

La primitive `unlink` supprime une des entrées de répertoire du fichier. Lorsque toutes les entrées sont supprimées, c'est-à-dire lorsque le nombre de liens devient nul, le fichier est physiquement effacé et l'espace occupé est ainsi libéré.

Cette primitive renvoie 0 en cas de suppression réussie ou -1 en cas d'erreur.

---

**write** — écrit dans un fichier

```
#include <unistd.h>

ssize_t write (int desc, const void *buf, size_t nombre)
```

La primitive `write` écrit `nombre` octets à partir de l'adresse pointée par `buf` dans le fichier associé au descripteur `desc` (obtenu par `open`, `creat`, `dup`, `fcntl` ou `pipe`).

Si le flag `O_APPEND` est mis, le pointeur de fichier sera mis à la fin du fichier avant toute écriture.

Le nombre renvoyé par `write` est le nombre d'octets réellement écrits dans le fichier. Ce nombre peut être inférieur à `nombre`, si la limite du fichier (voir `ulimit`) ou du volume est atteinte.

Si le fichier est un tube et si le flag `O_NDELAY` est mis, alors l'écriture ne sera pas complète s'il n'y a pas assez de place dans le tube.

Cette primitive renvoie le nombre d'octets écrits (non négatif) en cas d'écriture réussie, ou -1 en cas d'erreur.

---

**mmap** — projette un fichier en mémoire

```
#include <sys/mman.h>

void *mmap (void *adr, size_t lg, int prot, int flags, int desc, int offset)
```

La primitive `mmap` projette un fichier en mémoire, c'est-à-dire établit une correspondance entre un fichier ouvert et une zone de mémoire virtuelle : l'octet à l'offset $i$ du fichier est accessible en mémoire à l'adresse indiquée par `adr`$+i$. Toute lecture mémoire à cette adresse provoquera la lecture dans le fichier à l'offset correspondant, et toute écriture à cette adresse provoquera une écriture dans le fichier.

Il est également possible d'utiliser un objet de mémoire partagée POSIX créé avec `shm_open` (voir page 43) à la place d'un fichier.

Si le paramètre `adr` vaut `NULL`, le système choisit l'adresse où placer la zone de mémoire virtuelle et renvoie cette adresse comme valeur de retour. La longueur de la zone est spécifiée par le paramètre `lg`. Le fichier est référencé par son descripteur `desc`, et l'offset de début dans le fichier est indiqué par le paramètre `offset` (qui doit être un multiple de la taille d'une page, voir le paramètre `SC_PAGE_SIZE` de `sysconf` page 47). Si le paramètre `lg` n'est pas un multiple de la taille d'une page, le système peut étendre la zone jusqu'au plus prochain multiple.

Le paramètre `prot` spécifie le type d'accès mémoire souhaité, par une combinaison des bits suivants :

| | |
|---|---|
| PROT_NONE | aucun accès autorisé |
| PROT_READ | accès en lecture |
| PROT_WRITE | accès en écriture |
| PROT_EXEC | accès en exécution |

Le paramètre flags doit obligatoirement inclure l'un des deux bits suivants :

— MAP_SHARED : les modifications sont partagées entre tous les processus ayant établi une correspondance avec le fichier ;
— MAP_PRIVATE : les modifications effectuées ne sont pas propagées dans le fichier, et ne sont pas visibles par les autres processus ayant établi une correspondance avec le fichier.

De plus, le comportement de mmap peut être modifié par des bits supplémentaires dans le paramètre flags, parmi lesquels :

— MAP_ANONYMOUS : la zone de mémoire n'est pas mise en correspondance avec un fichier (et donc les paramètres desc et offset doivent valoir respectivement -1 et 0). Ceci permet d'avoir une zone de mémoire partagée entre plusieurs processus (ce qui implique l'utilisation du bit MAP_SHARED) ;
— MAP_FIXED : avec ce bit, mmap interprète une adresse (paramètre adr) non nulle comme une adresse impérative, et non comme une indication où placer la zone de mémoire virtuelle.

L'utilisation de tout autre bit pour le paramètre flag est par essence non portable et doit donc être évitée (par exemple en ayant recours à d'autres primitives similaires).

Cette primitive renvoie l'adresse de la zone de mémoire virtuelle, ou la valeur MAP_FAILED (-1) en cas d'erreur.

---

**munmap** — termine la projection d'un fichier en mémoire

```
#include <sys/mman.h>

int munmap (void *adr, size_t lg)
```

La primitive munmap termine la projection d'un fichier en mémoire, cest-à-dire supprime une correspondance entre un fichier ouvert et une zone de mémoire virtuelle établie par mmap.

Cette primitive renvoie 0 en cas de suppression réussie, ou -1 en cas d'erreur.

---

**lockf** — verrouillage de tout ou partie d'un fichier

```
#include <unistd.h>

int lockf (int desc, int action, off_t taille)
```

La primitive lockf demande le verrouillage de tout ou partie d'un fichier. Le verrouillage est consultatif, c'est-à-dire qu'un accès au fichier ne sera jamais refusé, mais que toute tentative de verrouillage avec lockf par un autre processus sera bloquée. Le fichier concerné est repéré par desc (obtenu par open, creat, dup, fcntl ou pipe). Le paramètre action peut prendre l'une des valeurs suivantes :

| | |
|---|---|
| F_LOCK | verrouiller (en attendant si le fichier est actuellement verrouillé) |
| F_ULOCK | déverrouiller |
| F_TEST | tester si le verrouillage est actif |
| F_TLOCK | tester et verrouiller si possible |

L'action demandée concerne la partie du fichier s'étendant sur taille octets à partir de la position courante dans le fichier (voir lseek, page 16). Si le paramètre taille égale 0, la partie s'étend jusqu'à la fin du fichier. Pour verrouiller ou déverrouiller tout un fichier, il suffit donc simplement de l'ouvrir puis d'appeler lockf avec une taille nulle.

Si le processus ayant verrouillé le fichier se termine, le fichier est automatiquement déverrouillé. S'il existait un processus en attente de verrouillage (avec F_LOCK), l'opération peut alors aboutir.

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas d'erreur ou de test négatif avec F_TEST ou F_TLOCK.

**symlink** — création d'un lien symbolique

```
#include <unistd.h>

int symlink (const char *cible, const char *lien)
```

La primitive `symlink` crée un lien symbolique nommé `lien` pointant vers le chemin `cible`. Les liens symboliques ne doivent pas être confondus avec les liens physiques (voir la primitive `link`, page 16).

La cible n'est pas obligée d'exister au moment où le lien est créé, cela ne constitue pas une erreur. La plupart des primitives sur les fichiers vont automatiquement suivre les liens : par exemple, lorsque la primitive `open` est appelée sur un lien, elle va automatiquement le suivre et ouvrir la cible. Ainsi, le test d'existence de la cible est effectué lors de l'utilisation du lien symbolique et non lors de la création.

Si la plupart des primitives suivent de manière transparente les liens, la primitive `lstat` (voir page 17) permet de les distinguer.

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas d'erreur.

---

**readlink** — lecture de la cible d'un lien symbolique

```
#include <unistd.h>

ssize_t readlink (const char *lien, char *cible, size_t taille)
```

La primitive `readlink` récupère la cible d'un lien symbolique nommé `lien`, et la place dans la zone repérée par le pointeur `cible` et de taille maximum `taille`, et retourne le nombre d'octets effectivement recopiés.

La primitive n'ajoute pas d'octet nul à la fin de la zone mémoire : dans la plupart des cas, l'application doit en ajouter un à la suite. Si la cible du lien fait plus de `taille` octets, seuls les `taille` premiers sont recopiés.

La taille d'un lien symbolique peut être obtenue avec la `struct stat` (voir page 17).

Cette primitive renvoie le nombre d'octets effectivement recopiés en cas d'opération réussie, ou -1 en cas d'erreur.

## 1.5 Gestion des processus

Un nombre important de primitives système sont dédiées à la gestion des processus.

La seule méthode pour créer un nouveau processus est la primitive `fork`. La seule méthode pour exécuter un fichier est une des primitives `exec`.

Chaque processus a un certain nombre d'attributs, tels que le répertoire courant, l'identificateur d'utilisateur, etc. Certaines primitives permettent de les consulter, certaines de les changer.

---

**chdir** — change le répertoire courant

```
#include <unistd.h>

int chdir (const char *chemin)
```

La primitive `chdir` change le répertoire courant du processus, c'est-à-dire le point de départ des chemins relatifs.

Cette primitive renvoie 0 en cas de changement réussi, ou -1 en cas d'erreur.

---

**chroot** — change le répertoire racine

```
#include <unistd.h>

int chroot (const char *chemin)
```

La primitive `chroot` change le répertoire racine du processus, c'est-à-dire le point de départ des chemins absolus. Le répertoire courant n'est pas affecté par cette opération.

L'utilisateur *effectif* doit être le super utilisateur pour pouvoir utiliser cette primitive.

Cette primitive renvoie 0 en cas de changement réussi, ou -1 en cas d'erreur.

---

**exec** — exécute un fichier

```
#include <unistd.h>

int execl (char *chemin, char *arg0, char *arg1, ... char *argn, NULL)
int execv (char *chemin, char *argv [])
int execle (char *chemin, char *arg0, char *arg1, ... char *argn, NULL, char *envp [])
int execve (char *chemin, char *argv [], char *envp [])
int execlp (char *fichier, char *arg0, char *arg1, ... char *argn, NULL)
int execvp (char *fichier, char *argv [])
```

Les primitives de la famille `exec` chargent un programme contenu dans un fichier exécutable, qu'il soit binaire ou interprétable par un *shell*.

Lorsqu'un programme C est exécuté, il est appelé comme suit :

```
extern char **environ ;

int main (int argc, char *argv [])
{
    ...
}
```

où `argc` est le nombre d'arguments, `argv` est un tableau de pointeurs sur les arguments eux-mêmes. `environ` est une variable globale, tableau de pointeurs sur les variables de l'environnement.

Les descripteurs de fichiers marqués *fermeture lors d'un exec* sont fermés automatiquement.

Les signaux positionnés pour provoquer la terminaison du programme ou pour être ignorés restent inchangés. En revanche, les signaux positionnés pour être attrapés sont remis à leur valeur par défaut.

Si le bit *set user id* du fichier exécutable est mis, l'utilisateur *effectif* est changé en le propriétaire du fichier. L'utilisateur *réel* reste inchangé. Ceci n'est pas valide pour les scripts, pour lesquels le bit *set user id* est ignoré.

Les segments de mémoire partagés du programme appelant ne sont pas transmis.

La mesure des temps d'exécution par `profil` est invalidée.

Le nouveau programme hérite des caractéristiques suivantes :
- valeur de `nice`,
- identificateur de processus,
- identificateur de groupe de processus,
- identificateur de groupe tty (voir `exit` et `signal`),
- flag de trace (voir `ptrace`),
- durée d'une alarme (voir `alarm`),
- répertoire courant,
- répertoire racine,
- masque de protections (voir `umask`),
- limites de taille de fichiers (voir `ulimit`), et
- temps d'exécution du processus.

Un *script shell* commence par une ligne de la forme `#!`*`shell`*, où `#!` doivent être les deux premiers caractères. Le shell doit être complètement spécifié, il n'y a pas de recherche à l'aide de PATH.

Les diverses formes de `exec` permettent de spécifier un fichier sans se soucier de son chemin d'accès complet, de passer ou non l'environnement de manière automatique, et de choisir la méthode de passage des paramètres. Le tableau ci-dessous résume les 6 possibilités :

| primitive | mode passage | passage environnement | recherche PATH |
|---|---|---|---|
| `execl` | liste | automatique | non |
| `execv` | vecteur | automatique | non |
| `execle` | liste | manuel | non |
| `execve` | vecteur | manuel | non |
| `execlp` | liste | automatique | oui |
| `execvp` | vecteur | automatique | oui |

Sur la plupart des implémentations, seule l'une de ces primitives est implémentée comme une primitive, les 5 autres sont implémentées sous la forme de fonctions de bibliothèque.

Si ces primitives retournent à l'appelant, une erreur est arrivée. La valeur renvoyée est donc toujours -1.

---

**exit** — termine un processus

```
#include <stdlib.h>

void exit (int etat)
```

La primitive `exit` termine l'exécution du processus appelant et passe l'argument `etat` au système pour inspection. Cet argument est utilisable par `wait`. Utiliser `return` dans la fonction `main` d'un programme C a le même effet que `exit`.

`Etat` est indéfini si `exit` n'a pas de paramètre.

A la terminaison du processus, les actions suivantes sont effectuées :

- tous les fichiers sont fermés,
- si le processus père est dans la primitive `wait`, il est réveillé et la valeur de `etat` (les 8 bits de poids faible) est transmise,
- si le processus père n'exécute pas `wait` et s'il n'ignore pas le signal `SIGCLD`, le processus fils devient *zombie*,
- le processus 1 devient le père de tous les processus fils du processus appelant,
- tous les segments de mémoire partagée sont détachés,
- si un des segments du processus était verrouillé en mémoire, il est déverrouillé (voir `plock`),
- une ligne de mesure (voir `acct`) est écrite si le système de surveillance est mis,
- si les identificateurs de processus, de groupe de processus et de groupe de terminal sont égaux, le signal `SIGHUP` est envoyé à tous les processus partageant le même groupe de processus.

Cette primitive ne renvoie pas de code de retour...

---

**fork** — crée un nouveau processus

```
#include <unistd.h>

pid_t fork (void)
```

La primitive `fork` crée un nouveau processus (le processus *fils*) par duplication du processus appelant (le processus *père*). Le fils hérite des caractéristiques suivantes :

- environnement,
- flags de *fermeture lors d'un exec* de tous les fichiers,
- traitement des signaux,
- bits *set user id* et *set group id*,
- mesure des fonctions (voir `profil`),
- valeur de `nice`,
- tous les segments de mémoire partagée,
- identificateur de groupe de processus,
- identificateur de groupe de terminaux (voir `exit` et `signal`),
- flag de trace (voir `ptrace`),
- répertoire courant,
- répertoire racine,

— masque de protections (voir `umask`), et
— limites de taille de fichiers (voir `ulimit`).

Le processus fils diffère du processus père par les points suivants :

— le fils a un identificateur de processus unique,
— le fils a un identificateur de processus père différent,
— le fils a ses propres descripteurs de fichiers, mais partage les pointeurs dans ces fichiers,
— tous les segments du processus sont déverrouillés, et
— les temps d'exécution sont mis à 0.

En cas de duplication réussie, cette primitive renvoie 0 pour le processus fils, et l'identificateur du processus fils pour le père. En cas d'erreur, la valeur -1 est renvoyée.

---

**getpid, getpgrp, getppid** — renvoie les process-id

```
#include <unistd.h>

pid_t getpid (void)
pid_t getpgrp (void)
pid_t getppid (void)
```

La primitive `getpid` renvoie l'identificateur du processus appelant.

La primitive `getpgrp` renvoie l'identificateur du groupe de processus auquel appartient le processus parent.

La primitive `getppid` renvoie l'identificateur du processus père du processus appelant.

---

**getuid, geteuid, getgid, getegid** — renvoie l'identificateur de l'utilisateur/du groupe du processus

```
#include <unistd.h>

uid_t getuid (void)
uid_t geteuid (void)
gid_t getgid (void)
gid_t getegid (void)
```

La primitive `getuid` renvoie l'identificateur de l'utilisateur *réel* du processus.

La primitive `geteuid` renvoie l'identificateur de l'utilisateur *effectif* du processus.

La primitive `getgid` renvoie l'identificateur du groupe *réel* du processus.

La primitive `getegid` renvoie l'identificateur du groupe *effectif* du processus.

---

**nice** — change la priorité d'un processus

```
#include <unistd.h>

int nice (int increment)
```

La primitive `nice` ajoute la valeur de `incrément` à la valeur de `nice` du processus appelant. La valeur de `nice` d'un processus est une valeur entière qui indique une priorité CPU d'autant plus faible que la valeur est grande.

Cette valeur est comprise en 0 et 2*NZERO-1 (soit la valeur 39 sur la plupart des systèmes Unix). Toute tentative de modification hors de ces valeurs ramènera la valeur à la limite correspondante.

Seul le super utilisateur a le droit de diminuer la valeur de `nice`.

Cette primitive renvoie la nouvelle valeur de `nice` moins 20, ou -1 en cas d'erreur. Il faut noter qu'une certaine valeur de `nice` renvoie une valeur assimilable au cas d'erreur.

---

**setpgrp, setsid** — change l'identificateur de session

```
#include <unistd.h>

pid_t setpgrp (void)
pid_t setsid (void)
```

La primitive `setpgrp` modifie l'identificateur du groupe de processus et le met à la valeur de l'identificateur du processus appelant.

La primitive `setsid` crée une nouvelle session, et retourne l'identificateur de groupe de processus créé.

Ces primitives renvoient la valeur du nouvel identificateur de groupe de processus, ou -1 en cas d'erreur.

---

**setuid, setgid** — change les identificateurs d'utilisateur et de groupe

```
#include <unistd.h>

int setuid (uid_t uid)
int setgid (gid_t gid)
```

La primitive `setuid` change les identificateurs *réel*, *effectif* et *sauvé* (ruid, euid ou suid respectivement) suivant les conditions ci-dessous (en fonction de su, l'identificateur du super utilisateur) :

— si `uid` = ruid = su, alors euid := `uid`,
— si `uid` $\neq$ su et `uid` = euid alors ruid := `uid`,
— si `uid` $\neq$ su et `uid` = suid alors euid := `uid`,
— si euid = su alors ruid := euid := suid := `uid`.

La primitive `setgid` opère de même pour les identificateurs de groupe.

Ces primitives renvoient 0 en cas de modification réussie, ou -1 en cas d'erreur.

---

**seteuid, setegid** — change les identificateurs effectifs

```
#include <unistd.h>

int seteuid (uid_t euid)
int setegid (gid_t egid)
```

La primitive `seteuid` change l'identificateur du propriétaire effectif du processus. Ceci est autorisé dans les cas suivants :

— si l'utilisateur effectif est actuellement égal à 0 (super-utilisateur)
— si la nouvelle valeur est la même que l'utilisateur réel (ruid) ou sauvé (suid)

La primitive `setegid` opère de même pour l'identificateur de groupe effectif.

Ces primitives renvoient 0 en cas de modification réussie, ou -1 en cas d'erreur.

---

**wait, waitpid** — attend la terminaison d'un processus fils

```
#include <sys/wait.h>

pid_t wait (int *statut)
pid_t waitpid (pid_t pid, int *statut, int options)
```

La primitive `wait` suspend l'exécution du processus appelant jusqu'à ce qu'un de ses fils directs se termine ou soit stoppé sur un point d'arrêt.

Cette primitive retourne prématurément à la réception d'un signal. Si un fils avait déjà terminé, le retour est immédiat.

Si le paramètre `statut` est non nul, 16 bits d'information sont stockés dans les 16 bits de poids faible situés à l'adresse indiquée. Ils servent à distinguer entre un processus stoppé et un processus terminé, de la manière suivante :

| | code retour | poids fort | poids faible |
|---|---|---|---|
| processus stoppé en mode trace | pid du processus | num du signal | 0177 |
| processus terminé par exit | pid du processus | arg. de exit sur 8 bits | 0 |
| processus terminé par signal | pid du processus | 0 | num du signal (+0200 si core) |
| wait interrompue par signal | -1 | ? | ? |

La manière portable pour analyser les codes de retour consiste à utiliser les macros suivantes :

| | |
|---|---|
| `WIFEXITED(`*stat_val*`)` | renvoie vrai si le processus s'est terminé par un `exit` explicite ou implicite |
| `WEXITSTATUS(`*stat_val*`)` | dans le cas précédent, renvoie le code de retour |
| `WIFSIGNALED(`*stat_val*`)` | renvoie vrai si le processus s'est terminé à cause d'un signal |
| `WTERMSIG(`*stat_val*`)` | dans le cas précédent, renvoie le signal en question |
| `WIFSTOPPED(`*stat_val*`)` | renvoie vrai si le processus est stoppé d'un signal |
| `WSTOPSIG(`*stat_val*`)` | dans le cas précédent, renvoie le signal en question |

La primitive `waitpid` fonctionne de manière analogue à `wait`, à ceci près qu'elle peut attendre des conditions plus spécifiques, selon la valeur du paramètre `pid` :

— `pid` $= -1$ : attendre n'importe quel processus (similaire à `wait`);
— `pid` $> 0$ : attendre le processus spécifié par `pid` et seulement celui-là ;
— `pid` $= 0$ : attendre n'importe quel processus du groupe de processus courant ;
— `pid` $< -1$ : attendre n'importe quel processus du groupe d'identificateur `-pid`.

Le paramètre `option` peut contenir les bits suivants :

| | |
|---|---|
| `WNOHANG` | `waitpid` n'est pas bloquant, et la valeur 0 est renvoyée si aucun processus fils n'est terminé ou stoppé. |
| `WUNTRACED` | `waitpid` détecte les processus stoppés mais non tracés (avec `ptrace`). |

Ces primitives renvoient l'identificateur du processus si l'attente s'est bien déroulée, ou -1 en cas d'erreur ou d'interruption par un signal.

## 1.6 Tubes

Les tubes sont un moyen de communication entre processus. Une fois un tube créé, on peut utiliser les primitives système `read` et `write`, comme pour n'importe quel descripteur de fichier.

Une lecture est bloquante tant que le tube est vide, sauf s'il n'y a plus d'écrivain, c'est-à-dire de processus ayant le tube ouvert en écriture. Lorsqu'il n'y a plus d'écrivain et que le tube est vide, le tube simule une fin de fichier.

Si lire dans un tube sans écrivain ne représente pas une erreur, écrire dans un tube sans lecteur est incohérent. Pour signaler cela, le système envoie le signal `SIGPIPE` dans ce cas.

On rencontre deux sortes de tubes : les tubes anonymes et les tubes nommés. Les premiers sont créés à l'aide de la primitive système `pipe`, alors que les seconds sont créés par la fonction de bibliothèque `mkfifo` (voir page 70).

---

**pipe** — crée un canal de communication

```
#include <unistd.h>

int pipe (int tubedesc [2])
```

La primitive `pipe` crée un tube anonyme, puis place dans `tubedesc[0]` le descripteur utilisé pour lire depuis le tube, et dans `tubedesc[1]` le descripteur utilisé pour écrire dans le tube.

Cette primitive renvoie 0 en cas d'ouverture réussie, ou -1 en cas d'erreur.

## 1.7 Signaux – API v7

Les signaux sont un mécanisme comparable aux interruptions matérielles. Ils permettent à un processus de réagir à un événement extérieur (appui sur la touche d'interruption, déconnexion, etc.) ou provoqué par un autre processus.

Un processus choisit avec la primitive `signal` le type de réaction aux événements ultérieurs :

— ignorer le signal,
— faire l'action définie par défaut par le système (généralement la terminaison du processus),
— ou interrompre l'exécution du programme pour exécuter une fonction définie, avec retour au programme après la fin de la fonction.

Les principaux signaux sont :

— `SIGHUP` : déconnexion,
— `SIGINT` : interruption (touche [BREAK]),
— `SIGQUIT`[1] : abandon (touche [CTL][ \ ]),
— `SIGILL`[1,3] : instruction illégale,
— `SIGTRAP`[1,3] : trace trap,
— `SIGIOT`[1] : instruction IOT,
— `SIGEMT`[1] : instruction EMT,
— `SIGFPE`[1] : exception en calcul flottant,
— `SIGKILL`[4] : kill (non masquable ni interceptable),
— `SIGBUS`[1] : erreur mémoire,
— `SIGSEGV`[1] : violation de segment,
— `SIGSYS` : argument incorrect dans une primitive. Non utilisé,
— `SIGPIPE` : écriture dans un tube sans lecteur,
— `SIGALRM` : alarme
— `SIGTERM` : signal logiciel de terminaison,
— `SIGUSR1` : user defined signal 1,
— `SIGUSR2` : user defined signal 2,
— `SIGCLD`[2] : mort d'un fils, et
— `SIGPWR`[3,2] : chute de tension.

Notes :
1 : une image de la mémoire peut être sauvegardée dans un fichier nommé `core`.
2 : l'action par défaut est d'ignorer le signal, plutôt que terminer le processus.
3 : l'action n'est pas remise à l'action par défaut lorsque le signal arrive.
4 : ce signal ne peut être ni ignoré, ni traité.

Si un signal survient (et provoque un déroutement vers une fonction) pendant l'exécution des primitives `open`, `read`, `write`, `sendto`, `recvfrom`, `sendmsg`, `recvmsg`, `wait` ou `ioctl`, la primitive peut renvoyer une erreur (`errno` = `EINTR`), ou le transfert de données peut être abrégé suivant le cas.

---

**alarm** — initialise l'interruption d'horloge

```
#include <unistd.h>

unsigned int alarm (unsigned int secondes)
```

La primitive `alarm` initialise l'horloge pour générer le signal `SIGALRM` dans `secondes` secondes.

L'alarme sera envoyée avec une tolérance de plus ou moins une demi-seconde. De plus, le mécanisme d'allocation du processeur peut retarder la réception du signal, particulièrement si le processus n'est pas en train de s'exécuter au moment où le signal est envoyé.

Les alarmes ne sont pas empilées. Un nouvel appel à `alarm` annule la précédente. Une alarme nulle annule l'alarme précédente si elle existait.

Les alarmes ne sont pas transmises lors d'un `fork`.

Cette primitive renvoie le temps restant avant la précédente alarme.

---

**kill** — envoie un signal à un processus

```
#include <signal.h>

int kill (pid_t pid, int signal)
```

La primitive `kill` envoie un signal à un processus ou à un groupe de processus spécifié par le paramètre `pid`. Le signal à envoyer est spécifié par le paramètre `signal`.

Si `signal` est nul, aucun signal n'est envoyé, mais une vérification est faite sur le paramètre `pid`.

Si `pid` est nul, le signal est envoyé à tous les processus du même groupe que le processus appelant.

Si `pid` = -1, le signal est envoyé à tous les processus dont le propriétaire est le propriétaire *effectif* du processus courant.

Si `pid` est négatif, mais différent de -1, le signal est envoyé à tous les processus dont le groupe est égal à la valeur absolue de `pid`.

Cette primitive renvoie 0 si le signal a été envoyé, ou -1 en cas d'erreur.

---

**pause** — suspend le processus en attendant un signal

```
#include <unistd.h>

int pause (void)
```

La primitive `pause` suspend l'exécution du processus jusqu'à ce qu'il reçoive un signal. Le signal ne doit pas être ignoré pour réveiller le processus.

Si le signal provoque la terminaison du processus, la primitive `pause` ne retourne pas à l'appelant.

Si le signal est traité par le processus appelant, et la fonction de traitement retourne, le processus reprend l'exécution après le point de suspension.

Étant donné que cette primitive attend indéfiniment jusqu'à ce qu'elle soit interrompue, la valeur de retour est toujours -1.

---

**signal** — spécifie le traitement à effectuer à l'arrivée d'un signal

```
#include <signal.h>

void (*signal (int sig, void (*action) (int))) (int)
```

La primitive `signal` permet à un processus de choisir une des trois manières de traiter un signal. Le paramètre `sig` indique le numéro du signal (voir page 27).

Le paramètre `action` peut prendre trois valeurs :

1. `SIG_DFL` : termine (sauf pour les cas particuliers) l'exécution du processus,
2. `SIG_IGN` : ignore le signal, ou
3. une adresse de fonction : à la réception du signal `sig`, la fonction `fonction` sera exécutée avec le numéro du signal comme paramètre.
   Attention : si le signal parvient au processus, celui-ci exécute la fonction, mais le système remet également l'action à SIG_DFL. Ceci signifie que si un signal peut arriver plusieurs fois, la primitive `signal` doit être appelée de nouveau dans la fonction.

Le signal `SIGKILL` ne peut être ni ignoré ni traité.

Cette primitive renvoie l'adresse de l'ancienne fonction en cas de modification réussie, ou la valeur numérique -1 en cas d'erreur (à tester avec la constante `SIG_ERR` pour avoir le bon type).

## 1.8 Signaux – API POSIX

L'API v7 des signaux (voir page 27) ont un certain nombre de défaut, parmi lesquels une absence de fiabilité (perte d'informations, possibilité de terminaison involontaire du processus, etc.). Le comité POSIX a revu en profondeur le fonctionnement des signaux. La nouvelle interface de programmation mérite une section à part.

Les signaux requis par la norme sont récapitulés dans le tableau ci-dessous (voir page 27 pour la signification de ces constantes). Les signaux de la deuxième moitié ne sont requis que si le système dispose de l'extension *job control*.

| | | | | | |
|---|---|---|---|---|---|
| SIGABRT | SIGALRM | SIGFPE | SIGHUP | SIGILL | SIGINT |
| SIGKILL | SIGPIPE | SIGQUIT | SIGSEGV | SIGTERM | SIGUSR1 |
| SIGUSR2 | | | | | |
| SIGCHLD | SIGCONT | SIGSTOP | SIGTSTP | SIGTTIN | SIGTTOU |

---

**sigaction** — manipulation de l'action associée à un signal

```
#include <signal.h>

int sigaction (int sig, const struct sigaction *nouvelle, struct sigaction *ancienne)
```

La primitive `sigaction` permet la récupération ou la modification de l'action associée au signal `sig`.

Le contenu de la structure `sigaction` est défini comme suit :

| Type | Nom | Description |
|---|---|---|
| `void (*)(int)` | `sa_handler` | `SIG_DFL`, `SIG_IGN` ou un pointeur sur une fonction |
| `sigset_t` | `sa_mask` | ensemble de signaux à bloquer pendant l'exécution de l'action (en plus de `sig`) |
| `int` | `sa_flags` | paramètres affectant le comportement du signal, en fonction des bits suivants :<br>— `SA_RESETHAND` : réinitialiser l'action associée au signal à l'appel de la fonction<br>— `SA_NODEFER` : ne pas masquer le signal `sig` pendant l'exécution de la fonction<br>— `SA_RESTART` : reprendre l'exécution de la primitive système (si elle le permet) interrompue par le signal à la fin de l'exécution de la fonction<br>— `SA_NOCLDWAIT` : si `sig` = `SIGCHLD`, les processus fils se terminant ne deviendront pas des processus zombies |

Si le paramètre `nouvelle` est non nul, il pointe sur une structure spécifiant l'action à effectuer lorsque le signal `sig` sera reçu. Si ce paramètre est nul, l'action n'est pas modifiée.

Si le paramètre `ancienne` est non nul, il pointe sur une structure que la primitive `sigaction` doit remplir avec l'action (avant l'appel à `sigaction`) associée au signal `sig`. Si ce paramètre est nul, rien n'est recopié.

Lorsque le signal est reçu, pendant l'exécution de l'action, un nouveau masque de signaux est fabriqué par l'union du masque courant, du masque associé au signal et du signal lui-même. À la fin de l'exécution de l'action, l'action reste associée au signal mais l'ancien masque est réinstallé.

Les objets de type `sigset_t` sont manipulés avec les fonctions de bibliothèque *sigsetopts* (voir page 73).

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas d'erreur.

---

**sigprocmask** — manipulation du masque de signaux

```
#include <signal.h>

int sigprocmask (int comment, const sigset_t *nouveau, sigset_t *ancien)
```

Si l'argument `nouveau` est non nul, le processus courant initialise son masque de signaux avec la valeur pointée. Le paramètre `comment` spécifie comment ce changement doit être effectué :

| comment | Description |
|---|---|
| SIG_BLOCK | le nouveau masque devient l'union de l'ancien et de celui pointé par nouveau |
| SIG_UNBLOCK | le nouveau masque est l'intersection de l'ancien et du complément de celui pointé par nouveau (tous ceux qui figurent dans nouveau sont retirés de l'ancien). |
| SIG_SETMASK | le nouveau masque devient celui pointé par nouveau |

Si l'argument nouveau est nul, le paramètre comment n'est pas significatif, cette primitive ne sert qu'à obtenir des informations sur le masque courant.

Si l'argument ancien est non nul, il pointe sur une structure que la primitive sigprocmask doit remplir avec le masque (avant l'appel à sigprocmask). Si ce paramètre est nul, rien n'est recopié.

Les objets de type sigset_t sont manipulés avec les fonctions de bibliothèque *sigsetopts*.

Le comportement de cette primitive dans un programme comportant plusieurs threads doit être évité : il faut utiliser pthread_sigmask (voir page 78) à la place.

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas d'erreur.

---

**sigwait** — attend un signal

```
#include <signal.h>

int sigwait (const sigset_t *ensemble, int *recu)
```

Cette primitive suspend le processus (ou le thread) courant en attendant l'un des signaux spécifiés par ensemble. Le numéro du signal reçu est placé en retour dans l'entier pointé par recu.

La valeur de retour est soit 0 en cas d'attente réussie, soit une valeur strictement supérieure à 0 pour indiquer le numéro d'erreur.

---

**sigpending** — consultation des signaux en attente

```
#include <signal.h>

int sigpending (sigset_t *signaux)
```

La zone pointée par signaux est remplie avec les signaux bloqués en attente.

Les objets de type sigset_t sont manipulés avec les fonctions de bibliothèque *sigsetopts*.

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas d'erreur.

---

**sigsuspend** — suspend le processus en attendant un signal

```
#include <signal.h>

int sigsuspend (const sigset_t *signaux)
```

Cette primitive remplace le masque de signaux par l'ensemble pointé par signaux et suspend le processus jusqu'à l'exécution d'une action spécifiée par sigaction ou la terminaison du processus.

Si une action est exécutée, sigsuspend se termine lorsque l'action est terminée, le masque de signaux est alors remis à sa valeur antérieure.

Les objets de type sigset_t sont manipulés avec les fonctions de bibliothèque *sigsetopts*.

Cette primitive renvoie toujours -1 pour indiquer une opération interrompue par l'arrivée d'un signal.

## 1.9 Horloge du système

Une autre catégorie de primitives système est celle qui exploite l'horloge du système. Indépendamment des mécanismes matériels, celle-ci assure la mesure des temps d'exécution et la mémorisation des heure et date courantes.

---

**stime** — initialise la date et l'heure du système

```
#include <time.h>

int stime (const time_t *adresse)
```

La primitive `stime` permet au super utilisateur de modifier l'heure et la date du système.

L'heure est stockée dans un entier long à l'adresse pointée par le paramètre `adresse`. Cet entier représente le nombre de secondes écoulées depuis 00 :00 :00 UTC January 1, 1970.

Cette primitive renvoie 0 en cas de modification réussie, ou -1 en cas d'erreur.

---

**time** — renvoie la date et l'heure

```
#include <time.h>

time_t time ((time_t *) 0)
time_t time (time_t *adresse)
```

La primitive `time` renvoie l'heure et la date courante en secondes écoulées depuis 00 :00 :00 UTC, January 1, 1970.

Si le paramètre `adresse` est non nul, la valeur de retour est aussi stockée à l'adresse indiquée.

Sur certains systèmes, `time` est implémenté comme une fonction de bibliothèque appelant `gettimeofday`.

La primitive `time` renvoie l'heure courante, ou -1 (ou plus exactement `((time_t)-1)`) en cas d'erreur.

Les fonctions de bibliothèque décrites en 2.4 (voir page 62) assurent la conversion entre un `time_t` et une chaîne de caractères.

---

**gettimeofday** — renvoie la date et l'heure précises

```
#include <sys/time.h>

int gettimeofday (struct timeval *tv, struct timezone *tz)
```

La primitive `gettimeofday` renvoie l'heure courante avec une précision supérieure à la seconde, ainsi que le fuseau horaire. La structure `timeval` contient les champs suivants :

```
time_t      tv_sec ;        // secondes
suseconds_t tv_usec ;       // et microsecondes
```

et la structure `timezone` contient les champs suivants :

```
int         tz_minuteswest ;    // minutes à l'ouest de Greenwich
int         tz_dsttime ;        // != 0 si l'heure d'été est en vigueur
```

Si un des pointeurs passés en paramètre vaut `NULL`, la structure correspondante n'est pas mise à jour par la primitive système. La primitive renvoie 0 si tout s'est bien passé, ou -1 en cas d'erreur.

---

**times** — renvoie les temps du processus et de ses fils

```
#include <sys/times.h>

clock_t times (struct tms *buf)
```

La primitive `times` place dans la structure pointée par le paramètre `buf` les temps d'unité centrale du processus et de ses fils. La structure possède les champs suivants :

```
clock_t tms_utime ;
clock_t tms_stime ;
clock_t tms_cutime ;
clock_t tms_cstime ;
```

- `tms_utime` est le temps CPU utilisé par le processus pendant l'exécution des instructions en mode utilisateur,
- `tms_stime` est le temps CPU utilisé par le processus pendant l'exécution des instructions en mode système,
- `tms_cutime` est la somme des temps CPU (mode utilisateur) utilisés par tous les processus terminés et descendants du processus courant,
- `tms_cstime` est la somme des temps CPU (mode système) utilisés par tous les processus terminés et descendants du processus courant.

Ces temps viennent du processus et de tous les processus fils pour lesquels le processus a appelé la primitive `wait`. L'unité dans laquelle ces temps sont exprimés est typiquement le *top d'horloge*, valeur dépendant du système utilisé. Il y a `CLK_TCK` (voir `sysconf` page 47) tops d'horloge par seconde sur les systèmes POSIX. Sur les systèmes anciens, il y a `HZ` tops par seconde, avec `HZ` défini dans le fichier `param.h`.

Note : `times` est redondante avec `getrusage` (voir page 48) qui est plus générale et devrait être privilégiée.

Cette primitive renvoie le temps réel écoulé (*elapsed*) depuis un repère dans le passé (typiquement l'heure de boot), ou -1 en cas d'erreur.

---

**clock_getres, clock_gettime, clock_settime** — Heure pour extensions temps réel

```
#include <time.h>

int clock_getres (clockid_t clkid, struct timespec *res)
int clock_gettime (clockid_t clkid, struct timespec *tp)
int clock_settime (clockid_t clkid, const struct timespec *tp)
```

Ces primitives sont un ajout de POSIX pour unifier les primitives de gestion du temps. Pour bénéficier d'une précision accrue, elles utilisent le type `struct timespec` qui comprend les champs suivants :

```
struct timespec
{
    time_t tv_sec ;       // secondes depuis l'époque (cf. primitive time)
    long tv_nsec ;        // et nanosecondes
} ;
```

Les différentes « horloges » (valeurs du type `clockid_t`) ou notions du temps gérées par ces primitives et spécifiées par POSIX sont :

- `CLOCK_REALTIME` : date et heure courante
- `CLOCK_MONOTONIC` : cette horloge croît de manière monotone (typiquement strictement croissante), depuis une date de référence (qui peut être par exemple la date du démarrage du système)
- `CLOCK_PROCESS_CPUTIME_ID` : temps CPU consommé par le processus
- `CLOCK_THREAD_CPUTIME_ID` : temps CPU consommé par le thread

Les autres valeurs de `clockid_t` ne sont pas normalisées par POSIX et ne devraient pas être utilisées.

Pour une horloge donnée, la primitive `clock_getres` renvoie la résolution (i.e. précision) dans `res`, `clock_gettime` renvoie la valeur dans `tp` et `clock_settime` modifie la valeur à partir de `tp`.

Ces primitives renvoient 0 en cas de succès, ou -1 en cas d'erreur.

## 1.10 Disques et périphériques

Les appels qui suivent sont tous liés à la gestion des systèmes de fichier et des périphériques en général. Il faut bien distinguer la notion de système de fichiers de celle de disque : un disque peut contenir plusieurs systèmes

de fichiers. Le disque n'est qu'un support.

---

**fsync, sync** — sauvegarde sur disque le contenu des buffers internes

```
#include <unistd.h>

int fsync (int desc)
void sync (void)
```

La primitive `fsync` provoque la sauvegarde sur disque de tous les éléments modifiés du fichier dont le descripteur est `desc` (obtenu par `open`, `creat`, `dup`, `fcntl` ou `pipe`). Tous les buffers internes (du *buffer-cache*) associés au fichier sont donc vidés.

La primitive `sync` provoque la sauvegarde sur disque de tous les buffers internes (l'ensemble du *buffer-cache*) en mémoire. Ceci inclut le super-block, les inodes modifiés et les blocs non encore écrits.

La primitive `fsync` renvoie 0 en cas d'écriture réussie, -1 en cas d'erreur.

---

**ioctl** — opérations diverses sur un périphérique

```
#include <sys/ioctl.h>

int ioctl (int desc, int requete, ... /* argument */)
```

La primitive `ioctl` accomplit une variété d'actions sur des périphériques en mode caractère, accédés par l'intermédiaire d'un descripteur de fichier obtenu par `open`, `dup` ou `fcntl`.

Les requêtes sont des ordres passés au driver de périphérique. Pour plus d'informations, consulter la documentation du driver.

Cette primitive renvoie -1 en cas d'erreur.

---

**mount, umount** — monte/démonte un système de fichiers

```
#include <sys/mount.h>

int mount (const char *special, const char *rep, int rwflag)
int umount (const char *special)
```

Attention : ces primitives ne sont pas normalisées par POSIX, et les prototype ci-dessus peuvent varier en fonction des systèmes.

La primitive `mount` demande qu'un système de fichiers identifié par `special`, le nom du périphérique en mode bloc, soit monté sous le répertoire nommé `rep`. Ces deux arguments sont des chemins d'accès.

La primitive `umount` provoque le démontage du système de fichiers identifié par `special`, le nom du périphérique en mode bloc.

Seul le super utilisateur a le droit de monter ou démonter un système de fichiers.

Ces primitives renvoient 0 en cas d'opération réussie, ou -1 en cas d'erreur.

---

**ulimit** — contrôle les ressources utilisables par un processus

```
#include <ulimit.h>

long ulimit (int commande, long limite)
```

Note : `ulimit` est redondante avec `getrlimit` (voir page 34) qui est plus générale et devrait être privilégiée.

La primitive `ulimit` fournit un moyen de contrôle sur les limitations imposées aux processus. Les valeurs que peut prendre le paramètre `commande` sont :

1. `UL_GETFSIZE` : renvoyer la taille maximum que peut prendre un fichier. La limite est en multiple de 512 octets, et est héritée aux processus fils.

2. UL_SETFSIZE : changer la taille maximum que peut prendre un fichier par limite. Tous les processus peuvent diminuer cette limite, mais seul le super utilisateur peut l'augmenter.
3. (valeur non normalisée) : renvoyer la taille maximum allouable par brk.
4. (valeur non normalisée) : renvoyer le nombre maximum de descripteurs de fichiers que le processus peut ouvrir.

Cette primitive renvoie un nombre non négatif en cas de réussite, ou -1 en cas d'erreur.

---

**getrlimit, setrlimit** — contrôle les ressources utilisables par un processus

```
#include <sys/resource.h>

int getrlimit (int ressource, struct rlimit *rlp)
int setrlimit (int ressource, const struct rlimit *rlp)
```

La primitive getrlimit renvoie les limites de consommation de ressources autorisées pour le processus appelant et ses descendants. La primitive setrlimit permet de modifier ces limites.

La structure rlimit utilisée par ces primitives comporte les champs suivants :

— rlim_cur : (limite *douce*) valeur courante de la limite
— rlim_max : (limite *dure*) valeur maximale autorisée pour la limite

Ces champs sont de type rlim_t (de type entier), la valeur RLIM_INFINITY indiquant l'absence de limite, la valeur RLIM_SAVED_MAX indiquant la valeur maximale autorisée

Une limite douce est modifiable par n'importe quel processus, sous réserve d'être comprise entre 0 et la limite dure. Une limite dure peut être diminuée également par n'importe quel processus. En revanche, seuls les processus de l'administrateur peuvent augmenter une limite dure.

L'argument ressource peut indiquer les ressources suivantes :

| Ressource | Description | Unité | Si dépassement |
|---|---|---|---|
| RLIMIT_CORE | Taille maximum des fichiers core | octet | fichier core tronqué |
| RLIMIT_CPU | temps CPU maximum utilisable | seconde | signal SIGXCPU |
| RLIMIT_DATA | Taille maximum du segment « data » | octet | malloc renvoie NULL |
| RLIMIT_FSIZE | Taille maximum d'un fichier | octet | signal SIGXFSZ / write renvoie -1 |
| RLIMIT_NOFILE | Nombre maximum de descripteurs de fichiers | nombre | open et autres renvoient -1 |
| RLIMIT_STACK | Taille maximum de la pile du thread principal | octet | signal SIGSEGV |
| RLIMIT_AS | Taille maximum de la mémoire virtuelle | octet | malloc / mmap renvoient NULL |

Ces primitives renvoient 0 en cas d'opération réussie, ou -1 en cas d'erreur.

## 1.11 Sockets Berkeley

Les ajouts de l'Université de Berkeley dans le domaine du réseau sont, pour le programmeur, de nouvelles primitives système et de nouvelles fonctions de bibliothèque. Ces primitives sont décrites ci-après.

## Erreurs

Les primitives système renvoient -1 en cas d'erreur. Dans ce cas, la variable globale errno est initialisée à la valeur correspondant à l'erreur survenue. Les cas ajoutés par l'Université de Berkeley dans le domaine du réseau sont :

— EADDRINUSE : adresse déjà utilisée,
— EADDRNOTAVAIL : l'adresse ne peut être affectée, comme par exemple pour une socket dont l'adresse n'est pas l'ordinateur courant,
— EAFNOSUPPORT : la famille d'adresses n'est pas supportée dans socket,
— ECONNABORTED : la connexion est rompue,
— ECONNREFUSED : la connexion est refusée,
— ECONNRESET : la connexion est rompue par l'autre extrémité, normalement par shutdown,

— EDESTADDRREQ : l'adresse de destination est requise pour l'opération demandée,
— EHOSTDOWN : une opération est demandée sur un ordinateur ne répondant pas,
— EHOSTUNREACH : aucune route trouvée vers l'ordinateur demandé,
— EINPROGRESS : l'opération est en cours de réalisation,
— EISCONN : la socket est déjà connectée,
— ENET : erreur sur le logiciel ou le matériel du réseau,
— ENETDOWN : le réseau est hors service,
— ENETRESET : le réseau a coupé la connexion,
— ENETUNREACH : aucune route trouvée vers le réseau demandé,
— ENOPROTOOPT : le protocole demandé n'est pas disponible, une mauvaise option demandée lors de getsockopt ou setsockopt,
— ENOTCONN : la socket n'est pas connectée,
— ENOTSOCK : l'opération nécessite une socket,
— EPROTONOSUPPORT : le protocole demandé n'est pas supporté,
— EPROTOTYPE : mauvais type pour la socket,
— ESHUTDOWN : tentative de transmission après un shutdown,
— ESOCKTNOSUPPORT : type de socket non supporté,
— ETIMEDOUT : la connexion n'a pas pu avoir lieu car elle a excédé la durée d'attente maximum.

---

**accept** — attente de connexion

```
#include <sys/socket.h>

int accept (int s, struct sockaddr *adresse, socklen_t *longueur)
```

Cette primitive est utilisée dans les sockets de type « connecté ». La socket s est supposée créée par socket, avoir acquis une adresse avec bind et en attente de connexion avec listen. accept extrait la première connexion en attente, crée une nouvelle socket avec les mêmes propriétés que s, renvoie son descripteur, et remplit adresse et longueur avec ses paramètres.

La valeur de retour est le descripteur de la nouvelle socket, ou -1 en cas d'erreur.

---

**bind** — affectation d'une adresse

```
#include <sys/socket.h>

int bind (int s, const struct sockaddr *adresse, socklen_t longueur)
```

La primitive bind affecte une adresse à la socket désignée par s. La variable longueur contient la longueur de l'adresse stockée à l'adresse adresse.

La valeur de retour est 0 si tout s'est bien passé, -1 sinon.

---

**close** — fermeture de socket

```
#include <unistd.h>

int close (int s)
```

La primitive close est étendue aux connexions IP.

---

**connect** — tentative de connexion

```
#include <sys/socket.h>

int connect (int s, const struct sockaddr *adresse, socklen_t longueur)
```

La primitive connect demande à la socket s d'ouvrir une connexion avec l'adresse spécifiée par adresse et longueur.

Si s est de type SOCK_DGRAM, connect enregistre l'adresse de destination et retourne immédiatement. Si le type est SOCK_STREAM, connect essaye d'établir une connexion fiable.

La valeur renvoyée est 0 si tout s'est bien passé, ou -1 en cas d'erreur.

**gethostname, sethostname** — lecture du nom de host

```
#include <unistd.h>

int gethostname (char *nom, size_t lg)
int sethostname (const char *nom, size_t lg)
```

La primitive `gethostname` recopie le nom de l'ordinateur dans la zone identifiée par `nom`. Au plus `lg` caractères sont recopiés, et terminés par un caractère nul si la place le permet.

La primitive `sethostname` permet à l'administrateur du système de modifier le nom de l'ordinateur.

Ces primitives renvoient 0 si tout s'est bien passé, ou -1 en cas d'erreur.

---

**getpeername** — lecture de l'adresse de l'autre partie

```
#include <sys/socket.h>

int getpeername (int s, struct sockaddr *adresse, socklen_t *longueur)
```

La primitive `getpeername` renvoie dans `adresse` et `longueur` les informations relatives à la partie distante de la connexion (c'est-à-dire « l'autre côté de la connexion ») représentée par `s`.

La valeur renvoyée est 0 si tout s'est bien passé, ou -1 en cas d'erreur.

---

**getsockname** — lecture de l'adresse de la socket

```
#include <sys/socket.h>

int getsockname (int s, struct sockaddr *adresse, socklen_t *longueur)
```

La primitive `getsockname` renvoie la description de la socket `s` dans `adresse` et `longueur`.

La valeur renvoyée est 0 si tout s'est bien passé, ou -1 en cas d'erreur.

---

**getsockopt** — lecture des options associées à la socket

```
#include <sys/socket.h>

int getsockopt (int s, int niveau, int option, void *valeur, socklen_t *longueur)
```

La primitive `getsockopt` renvoie les options associées à une socket. Voir la description des paramètres dans `setsockopt`.

Les options booléennes renvoient 0 si non armées, ou -1 si armées. Les paramètres `valeur` et `longueur` peuvent être modifiés pour toutes les options, booléennes ou non.

La valeur renvoyée est 0 si tout s'est bien passé, ou -1 en cas d'erreur.

---

**listen** — initialisation de la file d'attente

```
#include <sys/socket.h>

int listen (int s, int longueur)
```

Pour accepter des connexions, une socket est d'abord créée avec `socket`, puis une file d'attente pour les demandes de connexion est créée avec `listen`. Le paramètre `longueur` est le nombre maximum de connexions (entre 1 et 20) en attente pouvant être mémorisée dans cette file.

La valeur renvoyée est 0 si tout s'est bien passé, ou -1 en cas d'erreur.

**read** — lecture de données

```
#include <unistd.h>

ssize_t read (int desc, void *buf, size_t nombre)
```

La primitive `read` est étendue aux connexions IP.

**recv, recvfrom** — lecture de données

```
#include <sys/socket.h>

ssize_t recv (int s, void *buf, size_t lg, int flags)
ssize_t recvfrom (int s, void *buf, size_t lg, int flags,
                         struct sockaddr *exp, socklen_t *lgexp)
```

La primitive `recv` attend la réception d'un message à partir d'une socket `s`. Le message de longueur maximum `lg` est placé à partir de l'adresse `buf`. Le paramètre `flags` est initialisé à `MSG_PEEK` (lecture sans retirer de la file d'attente de réception), à `MSG_OOB` (lecture de données urgentes), ou 0.

La primitive `recvfrom` est identique à la primitive `recv`, à la différence que les paramètres de l'expéditeur sont renvoyés dans `exp` et `lgexp`.

La valeur de retour est le nombre d'octets reçus, ou -1 s'il y a eu erreur.

**select** — attente d'un évènement

```
#include <sys/select.h>

int select (int ndesc, fd_set *rd, fd_set *wd, fd_set *xd, struct timeval *delai)
void FD_ZERO (fd_set *ensemble)
void FD_CLR (int desc, fd_set *ensemble)
void FD_SET (int desc, fd_set *ensemble)
int  FD_ISSET (int desc, fd_set *ensemble)
```

La primitive `select` attend qu'un des descripteurs spécifiés par `rd` ait des données en attente, qu'un des descripteurs spécifiés par `wd` soit prêt à recevoir des données, qu'un des descripteurs spécifiés par `xd` exhibe une condition exceptionnelle, ou que la durée spécifiée par `delai` soit écoulée.

La spécification des descripteurs est réalisée par des ensembles de bits. Le descripteur $f$ est représenté par le bit `1<<f`. Pour faciliter la manipulation de ces ensembles, il faut utiliser les macros `FD_ZERO` (positionner tous les bits de l'ensemble à 0), `FD_CLR` et `FD_SET` (positionner le bit correspondant au descripteur `desc` respectivement à 0 ou 1) ou `FD_ISSET` (tester la valeur du bit correspondant au descripteur `desc`).

Le nombre de bits à surveiller dans les descripteurs est indiqué par `ndesc` (autrement dit, `ndesc` est égal au maximum des descripteurs + 1, les descripteurs débutant à 0).

Si un descripteur remplit une des conditions ci-dessus, le bit correspondant est laissé à 1 dans le masque correspondant. Sinon, il est remis à 0.

Si un masque ou `delai` n'est pas utile, le pointeur nul peut être transmis à la place.

La valeur renvoyée est le nombre de descripteurs affectés, 0 si la durée est écoulée sans événement, ou -1 en cas d'erreur.

Note : du fait du codage des descripteurs dans le type `fd_set`, `select` impose une limitation sur les numéros de descripteurs. La primitive `poll` (voir page 15), comparable, n'impose pas une telle limitation.

**send, sendto** — émission de données

```
#include <sys/socket.h>

ssize_t send (int s, const void *buf, size_t lg, int flags)
ssize_t sendto (int s, const void *buf, size_t lg, int flags,
                       const struct sockaddr *dest, socklen_t lgdest)
```

La primitive `send` envoie un message spécifié par `buf` et de longueur `lg` par la socket `s`.

La primitive `sendto` est similaire à la primitive `send` à ceci près que l'adresse de destination est spécifiée.

La valeur de retour est le nombre d'octets envoyés, ou -1 s'il y a eu erreur.

---

**setsockopt** — modification des options associées à la socket

```
#include <sys/socket.h>

int setsockopt (int s, int niveau, int option, const void *valeur, socklen_t longueur)
```

La primitive `setsockopt` change les options associées à une socket. Le niveau doit être `SOL_SOCKET` pour manipuler les options au niveau *socket*.

Les options sont définies dans `<sys/socket.h>` et sont décrites ci-dessous :

- `SO_BURST_IN` (sockets `SOCK_DGRAM` seulement) : nombre de messages pouvant être mémorisés en réception avant d'être rejetés,
- `SO_BURST_OUT` (sockets `SOCK_DGRAM` seulement) : nombre de messages pouvant être mémorisés en émission avant d'être rejetés,
- `SO_DONTROUTE` (sockets `SOCK_STREAM` seulement) : pas d'utilisation des tables de routage,
- `SO_REUSEADDR` (sockets `AF_INET` seulement) : permet la réutilisation des adresses locales,
- `SO_KEEPALIVE` (sockets `SOCK_STREAM` et `AF_INET` seulement) : force les sockets connectées, mais inactives et sans réponse, à émettre toutes les 45 secondes, jusqu'à 6 minutes,
- `SO_LINGER` (sockets `SOCK_STREAM` et `AF_INET` seulement) : garde la socket active lors d'un `close` s'il y a des données présentes,
- `SO_DONTLINGER` (sockets `SOCK_STREAM` et `AF_INET` seulement) : ne garde pas la socket active lors d'un `close`.
- `SO_RCVBUF` : pour la réception, change la taille du buffer (sockets `SOCK_STREAM`) ou la taille maximum d'un message (sockets `SOCK_DGRAM`),
- `SO_SNDBUF` : pour l'émission, change la taille du buffer (sockets `SOCK_STREAM`) ou la taille maximum d'un message (sockets `SOCK_DGRAM`),

La valeur renvoyée est 0 si tout s'est bien passé, ou -1 en cas d'erreur.

---

**shutdown** — fermeture de socket

```
int shutdown (int s, int comment)
```

La primitive `shutdown` ferme une socket. Le paramètre `comment` spécifie que les réceptions (si `SHUT_RD`), les émissions (si `SHUT_WR`), ou les émissions et les réceptions (si `SHUT_RDWR`) sont désactivées.

La valeur renvoyée est 0 si tout s'est bien passé, ou -1 en cas d'erreur.

---

**socket** — création de socket

```
#include <sys/socket.h>

int socket (int domaine, int type, int protocole)
```

La primitive `socket` crée une socket, c'est-à-dire une extrémité d'un canal de communication, et renvoie le descripteur associé.

Le paramètre `domaine` spécifie un domaine de communication utilisé pour interpréter les adresses dans les opérations ultérieures. Les domaines courants sont `PF_INET` (IPv4), `PF_INET6` (IPv6) et `PF_UNIX` (chemins d'accès dans l'arborescence Unix).

Le paramètre `type` spécifie la sémantique de la connexion. Le type est `SOCK_STREAM` (mode connecté, ordonné, fiable, bidirectionnel, dont l'unité est l'octet), `SOCK_DGRAM` (mode non connecté, non fiable, dont l'unité est le message) ou `SOCK_SEQPACKET` (mode connecté, ordonné, fiable, bidirectionnel, dont l'unité est le message).

Le paramètre `protocole` désigne le protocole utilisé. Normalement, un seul protocole existe pour un domaine et un type donnés. Toutefois, il pourrait arriver qu'il en existe plusieurs. La valeur 0 signifie que le système choisit le protocole le mieux adapté aux deux paramètres précités.

La valeur renvoyée est le descripteur de la socket créée, ou -1 en cas d'erreur.

**write** — émission de données

```
#include <unistd.h>

ssize_t write (int desc, const void *buf, size_t nombre)
```

La primitive `write` est étendue aux connexions IP.

## 1.12 IPC System V

Les IPC System V sont les mécanismes de communication interprocessus (IPC) introduits dans System V. Ces mécanismes sont suffisamment distincts, tant dans leur fonctionnalité que dans leur interface, du reste des primitives pour justifier une section à part.

Les IPC System V sont décomposés en trois parties :

1. les files de messages
2. la mémoire partagée
3. les sémaphores

Pour qu'un processus puisse utiliser ces primitives, il faut qu'il transforme une *clef* en un identificateur interne à l'aide de `msgget`, `shmget` ou `semget`. Le tableau suivant fait une analogie avec les fichiers :

| | Fichiers | IPC System V |
|---|---|---|
| nom externe | type = `char *`<br>chaîne, nom de fichier | type = `key_t`<br>entier, convention entre les processus communiquants |
| nom interne | type = `int`<br>descripteur de fichier | type = `int`<br>identificateur d'IPC |
| conversion | primitive `open` | primitives *xxx*`get` |
| droits d'accès | $3^e$ paramètre de `open` | dernier paramètre de *xxx*`get` |

## Files de messages

**msgctl** — opérations de contrôle d'une file de messages

```
#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msqid_ds *buf)
```

Le paramètre `cmd` de la primitive `msgctl` indique l'action à effectuer sur la file de messages repérée par l'identificateur `msqid` :

— `IPC_STAT` : place dans la structure pointée par `buf` les paramètres de la file de messages ;
— `IPC_SET` : initialise les paramètres de la file de messages indiqués par les champs `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode` et `msg_qbytes` de la structure pointée par `buf` ;
— `IPC_RMID` : supprime la file de messages si l'utilisateur est le super-utilisateur ou le propriétaire de la file de message.

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas d'erreur.

**msgget** — retourne l'identificateur d'une file de messages

```
#include <sys/msg.h>

int msgget (key_t clef, int flags)
```

Cette primitive renvoie l'identificateur interne associé à la clef fournie en argument.

La file de message est créée si la clef égale `IPC_PRIVATE` ou si le paramètre `flags` contient le bit `IPC_CREAT`. La file de messages est créée avec des permissions égales aux 9 premiers bits de `flags`.

La création est refusée si la file existe déjà et si les flags `IPC_CREAT` et `IPC_EXCL` sont positionnés.

Cette primitive ne peut renvoyer de file de message déjà créée avec la clef `IPC_PRIVATE`.

Cette primitive renvoie un nombre positif ou nul en cas d'opération réussie, ou -1 en cas d'erreur.

---

**msgsnd, msgrcv** — émission et lecture de messages

```
#include <sys/msg.h>

int msgsnd (int msqid, const void *msgp, size_t taille, int flags)
int msgrcv (int msqid, void *msgp, size_t taille, long type, int flags)
```

Un message (pointé par `msgp`) est constitué d'un champ de type `long` qui permet à l'utilisateur de spécifier un type ($\geq 1$) de message et d'une suite d'octets (entre 0 et une limite imposée par le système) formant la donnée du message.

La primitive `msgsnd` est utilisée pour envoyer un message. Lorsque la file de messages est saturée, le bit `IPC_NOWAIT` du paramètre `flags` spécifie si le processus doit être mis en attente.

La primitive `msgrcv` est utilisée pour recevoir :

- le premier message en attente si `type` $= 0$;
- le premier message de type `type` en attente si `type` $> 0$;
- le message de type minimum inférieur à la valeur absolue de `type` si `type`$\leq 0$.

Si aucun message n'est disponible, le bit `IPC_NOWAIT` du paramètre `flags` spécifie si le processus doit être mis en attente.

Le paramètre `taille` spécifie la taille du message à émettre (pour `msgsnd`) ou la taille maximum du message que le processus peut recevoir (pour `msgrcv`). Cette taille ne comprend pas le champ de type `long`.

En cas d'opération réussie, `msgsnd` renvoie 0, `msgrcv` renvoie la taille (non compris le champ de type `long`) du message lu. En cas d'erreur, ces primitives renvoient -1.

# Mémoire partagée

---

**shmctl** — opérations de contrôle d'un segment de mémoire partagée

```
#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid_ds *buf)
```

Le paramètre `cmd` de la primitive `shmctl` indique l'action à effectuer sur le segment de mémoire partagée repéré par l'identificateur `shmid` :

- `IPC_STAT` : place dans la structure pointée par `buf` les paramètres du segment ;
- `IPC_SET` : initialise les paramètres du segment indiqués par les champs `shm_perm.uid`, `shm_perm.gid` et `shm_perm.mode` de la structure pointée par `buf` ;
- `IPC_RMID` : supprime le segment de mémoire partagée si l'utilisateur est le super-utilisateur ou le propriétaire du segment ;
- `SHM_LOCK` : verrouille le segment de mémoire partagée en mémoire si l'utilisateur est le super-utilisateur ;
- `SHM_UNLOCK` : déverrouille le segment de mémoire partagée en mémoire si l'utilisateur est le super-utilisateur ;

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas d'erreur.

---

**shmget** — retourne l'identificateur d'un segment de mémoire partagée

```
#include <sys/shm.h>

int shmget (key_t clef, size_t taille, int flags)
```

Cette primitive renvoie l'identificateur interne associé à la clef fournie en argument.

Le segment de mémoire partagée est créé (avec une taille `taille`) si la clef égale `IPC_PRIVATE` ou si le paramètre `flags` contient le bit `IPC_CREAT`. Le segment est créé avec des permissions égales aux 9 premiers bits de `flags`.

La création est refusée si le segment existe déjà et si les flags `IPC_CREAT` et `IPC_EXCL` sont positionnés.

Cette primitive ne peut renvoyer de segment déjà créé avec la clef `IPC_PRIVATE`.

Cette primitive renvoie un nombre positif ou nul en cas d'opération réussie, ou -1 en cas d'erreur.

---

**shmat, shmdt** — attachement et détachement de segment de mémoire partagée

```
#include <sys/shm.h>

void *shmat (int shmid, void *adresse, int flags)
int shmdt (void *adresse)
```

La primitive `shmat` attache un segment de mémoire partagée dans l'espace d'adresses du processus à l'adresse spécifiée ou à une adresse sélectionnée par le système si le paramètre `adresse` est nul. Le bit `SHM_RDONLY` du paramètre `flags` spécifie si le segment doit être attaché en lecture seule ou en lecture et en écriture.

La primitive `shmdt` détache le segment situé à l'adresse `adresse`.

En cas d'opération réussie, `shmat` renvoie l'adresse d'attachement du segment, `shmdt` renvoie 0. En cas d'erreur, ces primitives renvoient -1.

## Sémaphores

---

**semctl** — opérations de contrôle de sémaphores

```
#include <sys/sem.h>

int semctl (int semid, int numero, int cmd, ... /* arg */)
```

Le paramètre `cmd` de la primitive `semctl` indique l'action à effectuer sur le groupe de sémaphores repéré par l'identificateur `semid` :

- `IPC_STAT` : place dans la structure pointée par `buf` les paramètres du groupe ;
- `IPC_SET` : initialise les paramètres du groupe indiqués par les champs `sem_perm.uid`, `sem_perm.gid` et `sem_perm.mode` de la structure pointée par `buf` ;
- `IPC_RMID` : supprime le groupe si l'utilisateur est le super-utilisateur ou le propriétaire du groupe ;
- `GETVAL` : renvoie la valeur du `numero`-ième sémaphore du groupe ;
- `SETVAL` : initialise la valeur du `numero`-ième sémaphore dans le groupe avec un quatrième paramètre entier ;
- `GETPID` : renvoie le numéro du dernier processus ayant fait une opération sur le `numero`-ième sémaphore du groupe ;
- `GETNCNT` : renvoie le nombre de processus attendant que la valeur du `numero`-ième sémaphore du groupe prenne une valeur supérieure à la valeur courante ;
- `GETZCNT` : renvoie le nombre de processus attendant que la valeur du `numero`-ième sémaphore du groupe prenne une valeur nulle ;
- `GETALL` : place dans le tableau d'entier courts non signés passé en quatrième paramètre les valeurs de tous les sémaphores du groupe ;
- `SETALL` : initialise tous les sémaphores du groupe avec les valeurs contenues dans le tableau d'entier courts non signés passé en quatrième paramètre.

Cette primitive renvoie 0 ou la valeur à renvoyer en cas d'opération réussie, ou -1 en cas d'erreur.

**semget** — retourne l'identificateur d'un ensemble de sémaphores

```
#include <sys/sem.h>

int semget (key_t clef, int nsem, int flags)
```

Cette primitive renvoie l'identificateur interne associé à la clef fournie en argument.

Le groupe de sémaphores est créé (avec le nombre de sémaphores `nsem`) si la clef égale `IPC_PRIVATE` ou si le paramètre `flags` contient le bit `IPC_CREAT`. Le groupe est créé avec des permissions égales aux 9 premiers bits de `flags`.

La création est refusée si le groupe existe déjà et si les flags `IPC_CREAT` et `IPC_EXCL` sont positionnés.

Cette primitive ne peut renvoyer de groupe déjà créé avec la clef `IPC_PRIVATE`.

Cette primitive renvoie un nombre positif ou nul en cas d'opération réussie, ou -1 en cas d'erreur.

---

**semop** — opérations sur sémaphores

```
#include <sys/sem.h>

int semop (int semid, struct sembuf *sops, int nsops)
```

Cette primitive est utilisée pour réaliser de manière atomique un ensemble d'opérations sur un groupe de sémaphores repéré par l'identificateur `semid`. Les opérations sont décrites par le tableau `sops` contenant `nsops` éléments. Chaque élément de ce tableau est une structure contenant les champs :

- `unsigned short sem_num` : numéro du sémaphore dans le groupe ;
- `short sem_op` : opération sur le sémaphore ;
- `short sem_flg` : paramètres de l'opération ;

Pour chaque sémaphore, `sem_op` indique l'opération :

- si `sem_op` $< 0$ :
  si la valeur courante du sémaphore est supérieure ou égale à la valeur absolue de `sem_op`, cette valeur est soustraite de la valeur courante du sémaphore. Sinon, le processus est mis en attente ou la primitive renvoie la valeur -1, suivant la valeur du bit `IPC_NOWAIT` de `sem_flg` ;
- si `sem_op` $> 0$ :
  la valeur de `sem_op` est ajoutée à la valeur courante du sémaphore ;
- si `sem_op` $= 0$ :
  si la valeur courante du sémaphore est nulle, `semop` continue avec l'opération suivante. Sinon, le processus est mis en attente ou la primitive renvoie la valeur -1, suivant la valeur du bit `IPC_NOWAIT` de `sem_flg` ;

Si le bit `SEM_UNDO` de `sem_flg` est mis, l'opération est inversée (valeurs soustraites au lieu d'être ajoutées et vice-versa) sans mise en attente du processus.

Par exemple, pour réaliser un P, puis un V sur un sémaphore, il faut faire :

```
sops [0].sem_num = 1 ; sops [0].sem_op = -1 ; sops [0].sem_flg = 0 ;
semop (semid, sops, 1) ;

sops [0].sem_num = 1 ; sops [0].sem_op = 1 ; sops [0].sem_flg = 0 ;
semop (semid, sops, 1) ;
```

Cette primitive renvoie une valeur positive ou nulle en cas d'opérations réussies, ou -1 en cas d'erreur.

## 1.13 Mémoire partagée POSIX

Les fonctions de gestion de mémoire partagée ci-après font partie d'une extension « temps réel » de la norme POSIX. À ce titre, elles ne sont pas forcément disponibles sur tous les systèmes et peuvent nécessiter une option spécifique de compilation ou d'édition de liens (`-lrt` par exemple sur Linux). Les objets de mémoire partagée sont destinés à être utilisés avec la primitive `mmap` (voir page 19) et `ftruncate` (voir page 18).

**shm_open** — ouvre (et crée éventuellement) un objet de mémoire partagée

```
#include <sys/mman.h>

int shm_open (const char *chemin, int flags, mode_t mode)
```

Cette fonction ouvre (avec création préalable éventuellement) un objet de mémoire partagée dont le nom est spécifié par `chemin` et retourne un descripteur analogue à un descripteur de fichier.

Un objet de mémoire partagée est repéré par un chemin. Certains systèmes implémentent les objets de mémoire partagée comme des fichiers dans le système de fichiers (les objets de mémoire partagée sont alors visibles), mais ce n'est pas forcément toujours le cas [1]. Si le chemin commence par le caractère « / », l'objet sera le même pour tous les processus accédant à ce chemin. Pour cela, il est suggéré d'utiliser des chemins de la forme « */nom* » où *nom* ne contient pas de caractère « / » (exemple : `"/toto"`).

Le descripteur de fichier retourné par `shm_open` sert à la fois avec `ftruncate` (voir page 18) pour spécifier la quantité de données de l'objet de mémoire partagée, et avec `mmap` (voir page 19) pour réaliser l'attachement de l'objet en mémoire.

L'ouverture de l'objet de mémoire partagée est spécifiée avec le paramètre `flags` contenant un sous-ensemble compatible avec `open` :

- `O_RDONLY` : ouverture en lecture seulement,
- `O_RDWR` : ouverture en lecture et écriture,
- `O_CREAT` : l'objet est créé s'il n'existait pas (dans ce cas, le mode est initialisé à partir du paramètre `mode`),
- `O_EXCL` : si `O_EXCL` et `O_CREAT` sont spécifiés, `shm_open` échoue si l'objet existe,
- `O_TRUNC` : si l'objet existe, sa taille est remise à 0,

L'objet de mémoire partagée (avec ses données) reste jusqu'à ce qu'il soit supprimé (voir `shm_unlink`).

Cette primitive renvoie le descripteur (non négatif) en cas d'ouverture réussie, ou -1 en cas d'erreur.

---

**shm_unlink** — supprime un objet de mémoire partagée

```
#include <sys/mman.h>

int shm_unlink (const char *chemin)
```

Cette fonction ordonne la destruction de l'objet de mémoire partagée référencé par le paramètre `chemin`. Comme pour les fichiers, l'objet ne peut plus être réouvert, mais il reste utilisable par tous les processus y accédant déjà.

Cette primitive retourne 0 en cas de réussite et -1 en cas d'erreur.

## 1.14 Sémaphores POSIX

Les primitives de sémaphore font également partie de l'extension « temps réel » de la norme POSIX. Elles permettent de réaliser les opérations classiques sur des sémaphores. Il est possible d'utiliser des sémaphores anonymes (utilisant une variable partagée de type `sem_t`) ou des sémaphores nommés qu'il faut ouvrir avec `sem_open`. Ces fonctions peuvent être utilisées pour synchroniser des threads ou des processus distincts partageant ou non une même zone de mémoire partagée.

Toutes ces fonctions utilisent le type `sem_t` pour représenter un sémaphore, et nécessitent l'inclusion du fichier `semaphore.h`.

Comme pour les threads POSIX, l'utilisation des sémaphores nécessite que l'édition de liens soit réalisée avec l'option `-l pthread` pour inclure la bibliothèque.

---

**sem_init** — initialise un sémaphore anonyme

1. Par exemple, Linux représente les objets de mémoire partagée comme des fichiers dans `/dev/shm/`, alors que l'implémentation de FreeBSD ne fait pas une telle correspondance.

```
#include <semaphore.h>

int sem_init (sem_t *sem, int partage, unsigned int val)
```

Cette fonction initialise un sémaphore anonyme (qui doit déjà être alloué avec le type `sem_t`) avec la valeur `val`. Si le paramètre `partage` est nul, le sémaphore est partagé entre les différents threads du processus courant. Dans le cas contraire (`partage` $\neq 0$), le sémaphore est partagé entre tous les processus pouvant accéder à la zone mémoire (partagée elle aussi) contenant le sémaphore.

Cette fonction renvoie 0 si l'initialisation est réussie ou -1 en cas d'erreur.

---

**sem_destroy** — détruit un sémaphore

```
#include <semaphore.h>

int sem_destroy (sem_t *sem)
```

Cette fonction détruit le sémaphore indiqué (mais ne libère pas la zone mémoire correspondante). Elle retourne 0 en cas de réussite, et -1 en cas d'erreur.

---

**sem_wait, sem_trywait, sem_timedwait** — verrouille un sémaphore

```
#include <semaphore.h>

int sem_wait (sem_t *sem)
int sem_trywait (sem_t *sem)
int sem_timedwait (sem_t *sem, const struct timespec *habs)
```

La fonction `sem_wait` réalise l'opération « P » (attente éventuelle si le compteur du sémaphore n'est pas strictement positif puis décrémentation du compteur).

La fonction `sem_trywait` fonctionne de manière similaire, mais échoue si le compteur n'est pas strictement positif.

La fonction `sem_timedwait` fonctionne de manière similaire à `sem_wait`, mais attend au plus jusqu'à l'heure spécifiée par `habs`, qui doit être une heure absolue (et non une durée relative à l'heure courante) comme avec la fonction `pthread_mutex_timedlock` (voir page 78).

Ces fonctions renvoient 0 si tout s'est bien passé, ou -1 en cas d'erreur ou de sémaphore non verrouillé.

---

**sem_post** — déverrouille un sémaphore

```
#include <semaphore.h>

int sem_post (sem_t *sem)
```

La fonction `sem_post` réalise l'opération « V » (incrémentation du compteur et libération des processus ou threads éventuellement en attente du sémaphore). Elle renvoie 0 en cas de réussite ou -1 en cas d'erreur.

---

**sem_getvalue** — récupère le compteur d'un sémaphore

```
#include <semaphore.h>

int sem_getvalue (sem_t *sem, int *cptval)
```

Note : l'utilisation de cette fonction pour autre chose que du déboguage est presque toujours une indication que la **synchronisation du programme est incorrecte**.

Cette fonction récupère la valeur du compteur du sémaphore et la place à l'adresse indiquée par `cptval`. Il faut noter que POSIX ne spécifie pas, en cas de processus ou de thread en attente sur un sémaphore, si la valeur doit être négative ou si elle doit être nulle : le choix appartient aux implémentations et peut donc varier d'un système à l'autre. La valeur de retour est soit 0 en cas de réussite ou -1 en cas d'erreur.

**sem_open, sem_close, sem_unlink** — utilisation de sémaphore nommé

```
#include <semaphore.h>

sem_t *sem_open (const char *chemin, int flags)
sem_t *sem_open (const char *chemin, int flags, mode_t mode, unsigned int val)
int sem_close (sem_t *sem)
int sem_unlink (const char *chemin)
```

Certains programmes ne partagent pas de mémoire commune et ne peuvent donc pas se synchroniser en allouant simplement un objet de type `sem_t` en mémoire partagée et en l'initialisant avec `sem_init` (voir page 43).

La fonction `sem_open` ouvre (avec création préalable éventuellement) un objet « sémaphore » dont le nom est spécifié par `chemin` et retourne l'adresse d'un `sem_t` utilisable avec les fonctions `sem_post` et `sem_wait`.

Un objet sémaphore est repéré par un chemin. Certains systèmes implémentent ces objets comme des fichiers dans le système de fichiers (les objets sémaphores sont alors visibles), mais ce n'est pas forcément toujours le cas [2]. Si le chemin commence par le caractère « / », l'objet sera le même pour tous les processus accédant à ce chemin. Pour cela, il est suggéré d'utiliser des chemins de la forme « */nom* » où *nom* ne contient pas de caractère « / » (exemple : `"/toto"`).

Les différentes valeurs possibles pour le paramètre `flags` sont :
- `0` : accès à un sémaphore pré-existant ; dans ce cas, la première forme avec 2 paramètres doit être utilisée ;
- `O_CREAT` : création du sémaphore ; dans ce cas, cas la deuxième forme avec 4 paramètres doit être utilisée. L'argument `mode` correspond aux permissions du nouveau sémaphore (voir `chmod`, page 12) et `val` correspond à sa valeur initiale. Si le sémaphore existe déjà, cette opération ne fait qu'y accéder sans modifier ses caractéristiques ;
- `O_CREAT | O_EXCL` : comme précédemment, sauf que l'opération échoue si le sémaphore existe déjà

La fonction `sem_close` ferme l'objet sémaphore indiqué. L'objet sémaphore reste jusqu'à ce qu'il soit supprimé avec `sem_unlink`). Tant qu'il n'est pas supprimé, il peut être ouvert à nouveau.

La fonction `sem_open` renvoie l'adresse d'un `sem_t`, ou `SEM_FAILED` en cas d'erreur. Les fonctions `sem_close` et `sem_unlink` renvoient 0 en cas de succès ou -1 en cas d'erreur.

## 1.15 Divers

Les primitives système qui suivent ne sont pas classables facilement. Elles font intervenir des concepts aussi différents que le changement de taille des segments d'un processus ou l'obtention du nom du système.

**acct** — active ou désactive la surveillance des processus

```
#include <unistd.h>

int acct (const char *chemin)
```

La primitive `acct` valide ou invalide la surveillance des processus (*process accounting*). Si la surveillance est validée, une ligne sera écrite dans le fichier `chemin` chaque fois qu'un processus se terminera.

Il faut être super utilisateur pour utiliser cette primitive.

Elle renvoie 0 en cas de réussite, ou -1 en cas d'erreur.

**brk, sbrk** — modifie la taille du segment de données

```
#include <unistd.h>

int brk (const void *fin)
void *sbrk (intptr_t increment)
```

2. Par exemple, Linux représente les objets sémaphores comme des fichiers dans `/dev/shm/` (comme pour les objets de mémoire partagée, voir page 43), alors que l'implémentation de FreeBSD ne fait pas une telle correspondance.

Attention : ces primitives n'étant pas spécifiées par POSIX, les fonctions `malloc`, `calloc` etc. de la bibliothèque doivent être utilisées de préférence (voir 2.2, page 57). Ces primitives sont uniquement citées ici comme exemple d'interface de bas niveau pour comprendre l'implémentation des fonctions de la famille `malloc`.

Elles servent à modifier dynamiquement la taille du segment de données, c'est-à-dire la zone contenant les variables globales (statiques) et le tas du processus.

La primitive `brk` modifie la taille (c'est-à-dire alloue ou libère de la mémoire) de telle manière que `fin` soit la dernière adresse valide dans l'espace d'adressage du processus. Certaines implémentations arrondissent la taille à un nombre entier de pages.

La primitive `sbrk` ajoute `incrément` octets à la taille du segment de données. Le type (`intptr_t`) de l'argument `increment` peut différer selon les implémentations.

En cas de réussite, `brk` renvoie 0 et `sbrk` l'ancienne adresse de la fin du segment de données. En cas d'erreur, la valeur -1 est renvoyée.

---

**mknod** — crée un fichier spécial

```
#include <sys/stat.h>

int mknod (const char *chemin, int mode, dev_t peripherique)
```

La primitive `mknod` crée un nouveau fichier de nom `chemin`. Le mode du fichier (son type et ses protections) est initialisé avec `mode`. Les différents bits constituant `mode` sont :

- type du fichier :
  - `S_IFIFO` : fichier spécial fifo,
  - `S_IFCHR` : fichier spécial en mode caractère,
  - `S_IFDIR` : répertoire,
  - `S_IFBLK` : fichier spécial en mode bloc
  - `S_IFREG` : fichier ordinaire
- `S_ISUID` : bit *set user id*
- `S_ISGID` : bit *set group id*
- protections du fichier, construites à partir des bits suivants :
  - `S_IRUSR` : lecture par le propriétaire
  - `S_IWUSR` : écriture par le propriétaire
  - `S_IXUSR` : exécution par le propriétaire
  - `S_I*GRP` : idem pour le groupe
  - `S_I*OTH` : idem pour les autres

Les valeurs de `mode` autres que celles ci-dessus ne sont pas définies et ne devraient pas être utilisées.

Le propriétaire du fichier spécial est l'utilisateur *effectif* du processus.

Les 9 bits de poids faible sont masqués par le masque de création de fichiers (voir `umask`).

Le paramètre `périphérique` est signifiant uniquement si `mode` indique un périphérique en mode bloc ou caractère.

Seul le super utilisateur peut utiliser `mknod` pour des fichiers autres que *fifo*.

Cette primitive renvoie 0 en cas de création réussie, ou -1 en cas d'erreur.

---

**profil** — mesure du temps d'exécution

```
#include <unistd.h>

void profil (unsigned short *buf, size_t taille, size_t offset, int echelle)
```

Attention : cette primitive n'est pas normalisée par POSIX et le prototype ci-dessus peut varier en fonction des systèmes.

La primitive `profil` valide la mesure des temps d'exécution (*process profiling*), qui aide à identifier les portions de programme prenant le plus de temps. La valeur du pointeur programme est lue à chaque top d'horloge (i.e. `CLK_TCK` fois par seconde, voir `sysconf` page 47) pour incrémenter un compteur dans la zone `buf`. `Echelle` et `offset` sont utilisés pour déterminer le mot à incrémenter.

Une `échelle` nulle ou égale à 1 invalide la mesure.

La primitive `profil` est (presque ?) exclusivement utilisée par l'option `-p` du compilateur `cc`.

---

**ptrace** — tracer un processus

```
#include <ptrace.h>

int ptrace (int requete, pid_t pid, void *adresse, int donnee)
```

Attention : cette primitive n'est pas normalisée par POSIX et le prototype ci-dessus peut varier en fonction des systèmes.

La primitive `ptrace` est utilisée par un processus pour contrôler l'exécution d'un processus fils. Elle est exclusivement utilisée pour implémenter des débogueurs.

Pour qu'un processus soit débogable, il faut qu'il ait exécuté `ptrace` avec la requête 0. Il s'arrête alors.

Le processus père doit attendre avec `wait` que le processus fils soit arrêté. Il peut ensuite avec les diverses requêtes consulter ou modifier la mémoire du fils. Par exemple, la requête `PTRACE_SINGLESTEP` fait exécuter une instruction par le fils.

---

**sysconf, pathconf, fpathconf** — renvoie des paramètres du système

```
#include <unistd.h>

long sysconf (int parametre)
long pathconf (const char *chemin, int parametre)
long fpathconf (int desc, int parametre)
```

Ces primitives ont été introduites par la norme POSIX pour obtenir les valeurs numériques de certains paramètres communs du système (pour `sysconf`) ou spécifiques à des fichiers ou des systèmes de fichiers (pour `pathconf` et `fpathconf`). La table ci-dessous indique quelques-uns des paramètres requis par la norme pour `sysconf` :

| paramètre | **Description** |
|---|---|
| `_SC_ARG_MAX` | Taille maximum en octets des arguments de `execve` |
| `_SC_CHILD_MAX` | Nombre maximum de processus par utilisateur |
| `_SC_CLK_TCK` | Fréquence de l'horloge utilisée par `times` (en nombre de tops par seconde) |
| `_SC_NGROUPS_MAX` | Nombre maximum de groupes auquel un utilisateur peut appartenir |
| `_SC_OPEN_MAX` | Nombre maximum de fichiers ouverts par utilisateur |
| `_SC_PAGE_SIZE` | Taille d'une page de mémoire virtuelle |
| `_SC_JOB_CONTROL` | 1 si le *job control* est disponible, ou -1 sinon |
| `_SC_VERSION` | La version de la norme ISO/IEC 9945 (POSIX 1003.1) que ce système supporte |
| `_SC_BC_BASE_MAX` | Valeur maximum pour les paramètres `ibase` et `obase` dans l'utilitaire `bc` |
| `_SC_BC_DIM_MAX` | Taille maximum des tableaux dans l'utilitaire `bc` |
| `_SC_BC_SCALE_MAX` | Valeur maximum pour le paramètre `scale` dans l'utilitaire `bc` |
| `_SC_BC_STRING_MAX` | Longueur maximum des chaînes dans l'utilitaire `bc` |
| `_SC_EXPR_NEST_MAX` | Nombre maximum d'expressions que l'on peut imbriquer entre parenthèses dans l'utilitaire `expr` |
| `_SC_LINE_MAX` | Longueur maximum de ligne admissible par les utilitaires de traitement de textes |
| `_SC_2_VERSION` | La version de la norme POSIX 1003.2 que ce système supporte |

La table ci-dessous indique quelques-uns des paramètres requis par la norme pour `pathconf` et `fpathconf` :

| paramètre | **Description** |
|---|---|
| `_PC_LINK_MAX` | Nombre maximum de liens vers le fichier |
| `_PC_PATH_MAX` | Longueur maximum d'un chemin relatif à partir du fichier |
| `_PC_NAME_MAX` | Longueur maximum d'un objet dans le répertoire |
| `_PC_PIPE_BUF` | Taille maximum d'une écriture atomique dans le tube |

Ces primitives renvoient la valeur du paramètre demandé, ou -1 en cas de valeur non limitée (auquel cas `errno` n'est pas modifiée) ou en cas d'erreur (auquel cas `errno` est modifiée).

**getrusage** — récupère des informations sur l'utilisation des ressources

```
#include <sys/resource.h>

int getrusage (int qui, struct rusage *res)
```

La primitive getrusage récupère les informations sur les ressources utilisées par le processus courant (paramètre qui = RUSAGE_SELF), par ses processus fils terminés (qui = RUSAGE_CHILDREN) ou par le thread courant (qui = RUSAGE_THREAD).

Les champs de la structure rusage sont :

- ru_utime : temps CPU en mode utilisateur (struct timeval, voir gettimeofday, page 31)
- ru_stime : temps CPU en mode système (struct timeval, voir gettimeofday, page 31)
- ru_maxrss : capacité maximum utilisée en mémoire physique
- ru_minflt : nombre de défauts de page n'entraînant pas d'entrée/sortie
- ru_majflt : nombre de défauts de page entraînant des entrées/sorties
- ru_inblock : nombre de lectures disque effectuées par le processus
- ru_oublock : nombre d'écritures disque effectuées par le processus
- ru_msgsnd : nombre de messages IPC System V envoyés
- ru_msgrcv : nombre de messages IPC System V reçus
- ru_nsignals : nombre de signaux reçus
- ru_nvcsw : nombre de changements volontaires de processus (à la suite d'une attente de ressource, par exemple)
- ru_nivcsw : nombre de changements involontaires de processus (du fait d'un processus plus prioritaire, par exemple)

La norme POSIX ne définit que les deux premiers champs (ru_utime et ru_stime), la présence des autres n'est pas garantie sur toutes les architectures ou toutes les versions de système. De même, POSIX ne définit pas RUSAGE_THREAD.

Cette primitive renvoie 0 en cas de succès, ou -1 en cas d'erreur.

---

**umask** — renvoie ou modifie le masque de création de fichiers

```
#include <sys/stat.h>

mode_t umask (mode_t masque)
```

La primitive umask initialise le masque binaire de mode de création de fichier (voir chmod). Seuls les 9 bits de poids faible sont significatifs.

Les bits mis à 1 dans le masque spécifient quels sont les bits de protection à mettre à 0 dans le mode des fichiers à créer.

Cette primitive renvoie la précédente valeur de masque.

---

**uname** — renvoie le nom du système

```
#include <sys/utsname.h>

int uname (struct utsname *buf)
```

La primitive uname renvoie des informations sur le nom et la version du système Unix dans un buffer pointé par buf. Les champs de la structure utsname sont :

```
char sysname [9] ;
char nodename [9] ;
char release [9] ;
char version [9] ;
char machine [9] ;
```

Il faut noter que toutes les chaînes de cette structure sont terminées par un octet nul.

Cette primitive renvoie un nombre non négatif en cas de réussite, ou -1 en cas d'erreur.

---

**utime** — change les dates d'accès et de modification d'un fichier

```
#include <utime.h>

int utime (const char *chemin, const struct utimbuf *buf)
```

La primitive `utime` modifie les dates d'accès et de modification d'un fichier de nom `chemin`.

Si `buf` est une adresse nulle, ces dates sont mises à la date courante. Un processus doit être le propriétaire d'un fichier ou avoir la permission en écriture sur le fichier pour procéder de la sorte.

Si `buf` n'est pas nul, il pointe sur une structure `utimbuf` dont les champs sont :

```
time_t actime ;
time_t modtime ;
```

Seul le propriétaire du fichier ou le super utilisateur peuvent utiliser `utime` de cette manière.

Cette primitive renvoie 0 si la modification a été réussie, ou -1 sinon.

# Chapitre 2

# Les fonctions de la bibliothèque standard

Le langage C ne définit que le langage proprement dit. Ceci permet une plus grande souplesse dans le traitement des entrées/sorties et dans l'interface avec le système.

Pour accéder au système ou à certaines caractéristiques intéressantes, l'utilisateur dispose de bibliothèque. Le système UNIX en comprend de nombreuses, depuis la gestion de la vidéo jusqu'aux fonctions mathématiques, en passant par les bibliothèques spécialisées pour tel ou tel outil.

Cependant, une bibliothèque est distinguée. Il s'agit de la *bibliothèque standard*, ou encore *bibliothèque C*. L'éditeur de liens la cherche automatiquement.

Ses fonctions peuvent être classées en grandes catégories :

— les fonctions d'entrées / sorties
— les fonctions de gestion de la mémoire
— les fonctions sur les caractères et les chaînes
— les fonctions associées aux sockets Berkeley
— les fonctions système

La liste ci-dessous donne la liste de ces fonctions par catégorie. Attention : cette liste n'est absolument pas exhaustive. Certaines fonctions peuvent en outre être différentes sur certains systèmes. Néanmoins, ces fonctions sont communes à presque tous les systèmes UNIX.

## 2.1 Fonctions d'entrées / sorties

Les fonctions d'entrées sorties nécessitent toutes l'inclusion du fichier *stdio.h*. Un fichier sous UNIX est une suite ininterrompue de caractères. L'accès est aussi bien séquentiel qu'aléatoire.

Les fonctions de la bibliothèque sont implémentées de manière sûre et efficace. Il ne faut pas hésiter à les utiliser quand c'est possible.

Un fichier est référencé par un *flux*, qui est l'association d'un fichier et d'un tampon d'entrées sorties. Un *flux* est représenté par le type prédéfini `FILE`. Trois *flux* spéciaux sont automatiquement ouverts : `stdin`, `stdout` et `stderr`.

La constante `NULL` représente une valeur illégale de *flux*.

La constante `EOF` est la valeur représentant la fin du fichier.

---

**fclose, fflush** — fermer ou actualiser les fichiers

```
#include <stdio.h>

int fclose (FILE *flux)
int fflush (FILE *flux)
```

Les deux fonctions écrivent le contenu des tampons associés au flux `flux` dans le fichier associé. La fonction `fclose` ferme ensuite le fichier.

Ces fonctions renvoient 0 s'il n'y a pas eu d'erreur, `EOF` sinon.

---

### **feof, ferror, clearerr, fileno** — état d'un flux

```
#include <stdio.h>

int feof (FILE *flux)
int ferror (FILE *flux)
void clearerr (FILE *flux)
int fileno (FILE *flux)
```

La fonction `feof` renvoie un résultat différent de 0 si la fin de fichier a été rencontrée sur le flux `flux`, 0 sinon.

La fonction `ferror` renvoie un résultat différent de 0 si une erreur a été rencontrée en cours de lecture ou d'écriture sur le flux `flux`, 0 sinon.

La fonction `clearerr` efface les indicateurs de fin de fichier ou d'erreur associés au flux `flux`.

La fonction `fileno` renvoie le numéro du descripteur de fichier qui est associé au flux `flux` pour utiliser une primitive système. Attention à utiliser `fflush` avant d'appeler la primitive système.

---

### **fopen, freopen, fdopen** — ouvrir un flux

```
#include <stdio.h>

FILE *fopen (const char *chemin, const char *mode)
FILE *freopen (const char *chemin, const char *mode, FILE *flux)
FILE *fdopen (int descripteur, const char *mode)
```

La fonction `fopen` ouvre le fichier `chemin` suivant le mode sélectionné par `mode`, qui peut être :

`r` : ouverture en lecture
`w` : effacement et ouverture en écriture
`a` : ouverture en écriture à la fin du fichier
`r+` : ouverture en lecture et écriture
`w+` : effacement et ouverture en lecture et écriture
`a+` : ouverture en lecture et écriture à la fin du fichier

La fonction `freopen` remplace le flux `flux` par l'ouverture du fichier `nom`.

La fonction `fdopen` associe un flux au descripteur de fichier `descripteur` obtenu en utilisant un appel système.

Ces trois fonctions renvoient le nouveau flux, ou la valeur `NULL` si une erreur est intervenue.

---

### **fread, fwrite** — entrée / sortie binaire

```
#include <stdio.h>

int fread (void *buf, int taille, int nb, FILE *flux)
int fwrite (const void *buf, int taille, int nb, FILE *flux)
```

La fonction `fread` lit `taille` octets sur le flux `flux`, les place dans le buffer `buf`, et répète l'opération `nb` fois.

L'argument `taille` est habituellement obtenu par l'opérateur `sizeof` de C.

La fonction `fwrite` réalise la même opération en écriture sur le flux `flux`.

Ces deux fonctions renvoient le nombre d'éléments lus ou écrits, ou 0 en cas d'erreur ou de fin de fichier.

---

### **fseek, ftell, rewind** — modifier le pointeur de flux

```
#include <stdio.h>

int fseek (FILE *flux, long offset, int mode)
long ftell (FILE *flux)
void rewind (FILE *flux)
```

La fonction `fseek` modifie la position de la prochaine lecture ou écriture sur le flux `flux`. La valeur de `offset`, suivant l'argument `mode`, signifie un déplacement relatif à partir :

— du début du fichier, si `mode` = `SEEK_SET` (valeur = 0),
— de la position courante, si `mode` = `SEEK_CUR` (valeur = 1),
— de la fin du fichier, si `mode` = `SEEK_END` (valeur = 2).

La valeur renvoyée est 0 s'il n'y a pas eu d'erreur, ou une valeur non nulle sinon.

La fonction `ftell` retourne la position courante dans le fichier, en nombre d'octets depuis le début du fichier.

La fonction `rewind` positionne le flux `flux` au début du fichier.

---

### getc, getchar, fgetc — lire un caractère sur le flux

```
#include <stdio.h>

int getc (FILE *flux)
int getchar (void)
int fgetc (FILE *flux)
```

La fonction `getc` renvoie le premier caractère disponible sur le flux `flux`, et incrémente le pointeur dans le flux.

La fonction `getchar` est équivalente à `getc(stdin)`.

La fonction `fgetc` est identique à `getc`, mais est une fonction et non une macro. Moins rapide, mais plus économique en place occupée.

Ces fonctions renvoient `EOF` en cas d'erreur ou de fin de fichier sur le flux.

---

### gets, fgets — lit une chaîne sur le flux

```
#include <stdio.h>

char *gets (char *str)
char *fgets (char *str, int max, FILE *flux)
```

Attention : la fonction `gets` est **obsolète**, car elle ne vérifie pas le débordement de la chaîne fournie en paramètre. Il faut utiliser `fgets` à la place.

La fonction `gets` lit une chaîne sur le flux d'entrée standard (`stdin`), jusqu'à ce qu'un caractère fin de ligne (`\n`) ou la fin de fichier soit rencontrée. Le caractère `\n` est remplacé par le caractère nul.

La fonction `fgets` est semblable à `gets` sur n'importe quel flux, à l'exception qu'elle conserve le caractère de fin de ligne `\n` et recopie au maximum `max` − 1 caractères (`'\0'` en bout de chaîne).

Ces fonctions renvoient `NULL` si la fin de fichier a été rencontrée sans qu'aucun caractère n'ait été lu.

---

### opendir, readdir, closedir — accès aux informations d'un répertoire

```
#include <dirent.h>

DIR *opendir (const char *chemin)
struct dirent *readdir (DIR *dp)
long int telldir (DIR *dp)
void seekdir (DIR *dp, long offset)
void rewinddir (DIR *dp)
int closedir (DIR *dp)
```

La fonction `opendir` ouvre un répertoire identifié par son chemin, et renvoie un descripteur utilisable dans les autres fonctions. Le résultat est `NULL` si le répertoire ne peut être ouvert.

La fonction `readdir` renvoie l'adresse d'une zone (statique) dans laquelle elle place l'entrée suivante du répertoire. Cette structure contient les champs suivants :

— `char d_name []` : nom de l'entrée dans le répertoire, terminé par un octet nul ;
— `ino_t d_ino` : numéro d'inode du fichier ;

Note : certaines implémentations comportent un champ `d_type` indiquant le type (fichier, répertoire, etc.) de l'entrée. Ce champ n'est pas spécifié par POSIX, aussi son utilisation est **à proscrire** pour obtenir des programmes portables. On utilisera à la place la primitive `stat` (voir page 17).

Le résultat est `NULL` en cas d'erreur ou lorsque la fin du répertoire est rencontrée.

la fonction `telldir` renvoie la position courante dans le répertoire, la fonction `seekdir` initialise cette position courante, et la fonction `rewinddir` la remet à 0.

La fonction `closedir` ferme le répertoire, et renvoie -1 en cas d'erreur et 0 en cas de fermeture réussie.

---

## popen, pclose — ouvre ou ferme un tube

```
#include <stdio.h>

FILE *popen (const char *commande, const char *mode)
int pclose (FILE *flux)
```

La fonction `popen` ouvre un tube (communication entre deux processus) avec la commande `commande` interprétée par le shell.

Le mode d'ouverture `mode` est une chaîne de caractères contenant `r` pour une ouverture en lecture, ou `w` pour une ouverture en écriture.

Le résultat de la fonction `popen` est le flux dans lequel il faut lire ou écrire, ou `NULL` s'il y a une erreur.

Un tube ouvert avec `popen` doit être fermé avec la fonction `pclose` qui attend que la commande se termine. La valeur renvoyée par `pclose` est le code de retour de la commande `commande`.

---

## printf, dprintf, fprintf, sprintf, snprintf — écriture formatée

```
#include <stdio.h>

int printf (const char *format, ...)
int dprintf (int desc, const char *format, ...)
int fprintf (FILE *flux, const char *format, ...)
int sprintf (char *str, const char *format, ...)
int snprintf (char *str, size_t taille, const char *format, ...)
```

Attention : la fonction `sprintf` peut être dangereuse car elle ne vérifie pas le débordement de la chaîne `str` fournie en paramètre. Il vaut mieux en général utiliser `snprintf`, sauf s'il est certain que la place dans `str` est suffisante.

La fonction `printf` affiche ses données sur la sortie standard, la fonction `fprintf` écrit sur le flux `flux`, la fonction `dprintf` écrit dans un fichier référencé par l'entier `desc` (ouvert par la primitive système `open` ou équivalent) et la fonction `snprintf` place le résultat dans la chaîne de caractères `str` en vérifiant le non-débordement de cette chaîne.

En cas d'erreur, la valeur retournée est négative. Dans le cas contraire, c'est le nombre d'octets affichés pour `printf`, `dprintf` ou `fprintf`. Pour `snprintf`, c'est le nombre d'octets résultant de la conversion (c'est-à-dire qui auraient été écrits s'il n'y avait pas de limite) : il suffit ainsi de tester si la valeur retournée est supérieure ou égale à `taille` pour déterminer si la chaîne `str` est trop petite.

Chacune de ces fonctions affiche les arguments en les convertissant suivant le format `format`. Le format est une chaîne de caractères qui contient deux sortes d'objets : les caractères simples, qui sont affichés normalement, et les formats de conversion commençant par un caractère `%`, régis par la syntaxe suivante (les éléments entre crochets sont optionnels, les accolades indiquent une lettre au choix) :

`%` [*num-arg*`$`] [*flags*] [*largeur-min*] [`.` *precision*] [*taille*] *conversion*

- s'il est présent (suivi de `$`), le numéro d'argument *num-arg* précise que cette conversion est appliquée sur l'argument indiqué. Ainsi, par exemple, `printf ("%2$d %1$d\n", 5, 7)` affiche « `7 5` ».
- les *flags* sont constitués d'un nombre quelconque (éventuellement nul) de caractères parmi :

| | |
|---|---|
| `-` | aligner la valeur à gauche |
| `+` | préfixer la valeur par un signe, même pour les valeurs positives |
| `␣` | laisser une position vide pour le signe s'il n'y en a pas |
| `0` | ajuster la valeur à la dimension souhaitée avec des chiffres `0` à gauche |
| `#` | utiliser une forme alternative (ex : préfixer par `0x` pour l'hexadécimal) |
| `'` | utiliser des séparateurs de milliers |

— la *largeur-min* est la largeur minimum (en caractères) de la valeur affichée, complétée éventuellement suivant les *flags* par des espaces ou des `0` d'en-tête ;
— la *précision* donne le nombre minimum de chiffres à utiliser pour les conversions entières, ou le nombre de chiffres après la virgule pour les conversions flottantes, ou le nombre de caractères pour les chaînes
— la *taille* donne la taille de l'argument :

| | |
|---|---|
| `l` | l'argument est de type `long int` |
| `ll` | l'argument est de type `long long int` |
| `j` | l'argument est de type `intmax_t` |
| `z` | l'argument est de type `size_t` |
| `t` | l'argument est de type `ptrdiff_t` (différence d'adresses) |
| `L` | l'argument est de type `long double` |

En cas de conversion `u`, les types indiqués ci-dessus s'entendent comme des types non signés.
— enfin, le spécificateur de *conversion* est l'un des caractères de la liste (non exhaustive) ci-après :

| | |
|---|---|
| `d` | conversion en décimal |
| `o` | conversion en octal |
| `x` | conversion en hexadécimal |
| `u` | conversion en décimal non signé |
| `f` | conversion en flottant sans exposant |
| `e` | conversion en flottant avec exposant |
| `g` | conversion d'un flottant en `%d`, `%e` ou `%f` suivant le cas |
| `c` | conversion en caractère |
| `s` | conversion en chaîne de caractères |
| `p` | conversion d'une adresse |
| `%` | affichage d'un caractère `%` |

Exemples :
— pour imprimer une date et une heure au format « `dimanche 3 juillet, 10:02` », il faut utiliser :

```
printf ("%s %d %s, %d:%.2d", sem, jour, mois, heure, mn)
```

— pour imprimer $\pi$ à $10^{-5}$ près :

```
printf ("PI = %.5f", 4 * atan (1.0))
```

— pour imprimer des valeurs correspondant à des types définis par POSIX (cf. 1.3, page 11) de manière portable, il faut les convertir en `intmax_t` ou `uintmax_t` :

```
printf ("taille = %ju, uid=%ju", (uintmax_t) stbuf.st_size, (uintmax_t) stbuf.st_uid)
```

---

**vprintf, vdprintf, vfprintf, vsprintf, vsnprintf** — écriture formatée avec un nombre variable d'arguments

```
#include <stdarg.h>
#include <stdio.h>

int vprintf (const char *format, va_list ap)
int vdprintf (int desc, const char *format, va_list ap)
int vfprintf (FILE *flux, const char *format, va_list ap)
int vsprintf (char *str, const char *format, va_list ap)
int vsnprintf (char *str, size_t taille, const char *format, va_list ap)
```

Ces fonctions sont analogues aux fonctions de type `printf`, mais sont appelées avec une liste d'arguments, permettant d'être utilisées à l'intérieur de fonctions prenant un nombre variable d'arguments (avec `stdarg.h`), comme par exemple :

```
#include <stdarg.h>

void erreur (const char *fn, const char *fmt, ...)   // « ... » est la syntaxe C
{
    va_list ap ;                                     // pointeur dans la liste d'arguments

    fprintf (stderr, "Erreur dans la fonction %s\n", fn) ;
    va_start (ap, fmt) ;                             // ap pointe sur l'argument après fmt
    vfprintf (stderr, fmt, ap) ;
    va_end (ap) ;                                    // fin d'utilisation de ap
    exit (1) ;
}

int main (int argc, char *argv [])
{
    if (argc > 2)
        erreur (__FUNCTION__, "usage: %s [n]\n", argv [0]) ;
    if (atoi (argv [1]) == 0)
        erreur (__FUNCTION__, "l'argument ne peut etre nul\n") ;
    ...
}
```

Attention : comme pour `sprintf`, la fonction `vsprintf` peut être dangereuse car elle ne vérifie pas le débordement de la chaîne fournie en paramètre. Il vaut mieux en général utiliser `vsnprintf`, sauf s'il est certain que la place dans `str` est suffisante.

---

### putc, putchar, fputc — écrire un caractère sur le flux

```
#include <stdio.h>

int putc (int c, FILE *flux)
int putchar (int c)
int fputc (int c, FILE *flux)
```

La fonction `putc` écrit le caractère `c` sur le flux `flux`, et incrémente le pointeur dans le flux.

La fonction `putchar(c)` est équivalente à `putc(c,stdout)`.

La fonction `fputc` est identique à `putc`, mais est une fonction et non une macro. Moins rapide, mais plus économique en place occupée.

Ces fonctions renvoient `EOF` en cas d'erreur sur le flux, sinon le caractère écrit.

---

### puts, fputs — écrire une chaîne sur le flux

```
#include <stdio.h>

int puts (const char *str)
int fputs (const char *str, FILE *flux)
```

La fonction `fputs` écrit la chaîne de caractères `str` (sans le caractère de terminaison `'\0'`) sur le flux `flux`.

La fonction `puts(s)` est équivalente à `fputs(s,stdout)` à la seule différence qu'elle ajoute un caractère de saut de ligne.

Ces fonctions renvoient `EOF` en cas d'erreur sur le flux.

---

### scanf, fscanf, sscanf — lecture formatée

```
#include <stdio.h>

int scanf (const char *format, ...)
int fscanf (FILE *flux, const char *format, ...)
int sscanf (char *str, const char *format, ...)
```

La fonction `scanf` lit ses données dans l'entrée standard (`stdin`), la fonction `fscanf` lit ses données dans le flux `flux`, et la fonction `sscanf` lit ses données dans la chaîne de caractères `str`.

Ces trois fonctions lisent les caractères et les interprètent en fonction du format représenté par la chaîne de caractères `format`. Chacun des arguments suivants doit être un pointeur sur la zone mémoire où sera rangée l'objet lu.

La chaîne de format peut contenir :

— des espaces ou des caractères `\n` pour spécifier un nombre quelconque de blancs,
— des caractères ordinaires (pas `%`), qui doivent correspondre au caractère lu,
— et des spécifications de conversion commençant par le caractère `%`. Ces spécifications décrivent une conversion qui doit être effectuée et le résultat doit être rangé dans l'argument suivant, à moins que le caractère `*` soit présent, auquel cas la valeur lue est ignorée.

Les spécifications de conversion sont :

— `%` : le caractère `%` est attendu, aucune conversion n'est effectuée,
— `d`, `o`, `x` ou `u` : un nombre en décimal, octal, hexadécimal ou décimal non signé est attendu, et l'argument correspondant doit être un pointeur sur un entier,
— `e`, `f` ou `g` : un nombre flottant est attendu, et l'argument doit être un pointeur sur un flottant,
— `s` : une chaîne de caractères non nulle est attendue, l'argument doit être un pointeur sur un tableau de caractères assez grand. Une chaîne vide ne peut être lue par `%s`,
— `c` : un caractère est attendu, et l'argument doit être un pointeur sur un caractère,
— `[` : indique une chaîne dont on spécifie les caractères. Par exemple, `%[A-H]` spécifie une chaîne ne contenant que des caractères entre `A` et `H`.

Les spécifications `d`, `o`, `u` et `x` peuvent être précédées de la lettre `l` pour indiquer une valeur longue. De même, les spécifications `e`, `f` et `g` précédées de la lettre `l` indiquent que l'argument est un pointeur sur un `double`.

Ces fonctions renvoient `EOF` si la fin du fichier a été détectée, ou le nombre d'arguments reconnus.

---

**ungetc** — replace le caractère dans le flux d'entrée

```
#include <stdio.h>

int ungetc (int c, FILE *flux)
```

La fonction `ungetc` insère le caractère `c` dans le flux `flux` de telle sorte qu'il soit accessible par les prochaines instructions de lecture.

Il ne peut y avoir retour que d'un seul caractère, qui doit par ailleurs être identique à celui qui existait.

La fonction renvoie le caractère replacé, ou `EOF` s'il y a eu erreur.

## 2.2 Gestion de la mémoire

Les fonctions suivantes permettent de gérer l'allocation dynamique de la mémoire. Cette gestion de mémoire est similaire à la gestion du *tas* en Pascal (fonctions `new` et `dispose` de Pascal).

---

**free** — libère un espace mémoire

```
#include <stdlib.h>

void free (void *pointeur)
```

La fonction `free` libère un espace précédemment alloué par `malloc`. L'espace libéré est à nouveau disponible pour un `malloc` ultérieur.

---

**malloc, calloc** — alloue un espace mémoire

```
#include <stdlib.h>

void *malloc (size_t taille)
void *calloc (size_t nombre, size_t taille)
```

La fonction `malloc` alloue un espace mémoire de `taille` octets (typiquement obtenu avec l'opérateur `sizeof`), et renvoie un pointeur sur l'espace alloué.

La fonction `calloc` alloue un espace pour un tableau de `nombre` éléments de taille `taille` chacun. Cet espace est initialisé à 0.

Ces deux fonctions renvoient un pointeur sur l'espace alloué, ou `NULL` s'il n'y a plus de place ou s'il y a eu erreur.

---

**realloc** — change la taille d'un espace mémoire

```
#include <stdlib.h>

void *realloc (void *pointeur, size_t taille)
```

La fonction `realloc` change la taille de l'espace alloué à exactement `taille` octets. Si l'espace n'est pas extensible, `realloc` en cherche un autre et y copie les données. Si `pointeur` est nul, `realloc` se comporte comme `malloc`.

La fonction `realloc` renvoie le pointeur sur l'espace mémoire, ou `NULL` s'il n'y a plus de place ou s'il y a eu erreur.

---

**memcpy, memmove** — copie de blocs de mémoire

```
#include <string.h>

void *memcpy (void *bloc1, const void *bloc2, size_t n)
void *memmove (void *bloc1, const void *bloc2, size_t n)

#include <strings.h>

void bcopy (const char *bloc2, char *bloc1, int n)
```

La fonction `memcpy` copie `n` octets depuis l'adresse `bloc2` jusqu'à l'adresse `bloc1`. Si les deux blocs ont une intersection commune, il faut utiliser `memmove`, plus lente mais plus sûre.

La valeur de retour de ces deux fonctions est `bloc1` (il n'y a pas d'erreur possible).

La fonction `bcopy` est la version Berkeley (non normalisée par POSIX).

---

**memset** — initialisation d'un bloc de mémoire

```
#include <string.h>

void *memset (void *bloc, int c, size_t n)

#include <strings.h>

void bzero (char *bloc, int n)
```

La fonction `memset` initialise à l'octet `c` les `n` octets à partir de l'adresse `bloc`.

La valeur de retour est `bloc` (il n'y a pas d'erreur possible).

La fonction `bzero` est la version Berkeley, restreinte à l'initialisation à 0. Cette fonction n'est pas normalisée par POSIX.

## 2.3 Chaînes de caractères

Les fonctions suivantes utilisent des chaînes de caractères terminées par un caractère nul. Elles ne vérifient pas le débordement des chaînes passées en paramètre.

## **atof, atoi, atol** — conversion en valeur numérique

```
#include <stdlib.h>

double atof (const char *str)
int atoi (const char *str)
long atol (const char *str)
```

Attention : ces fonctions sont **obsolètes** car elles ne détectent pas les erreurs comme des nombres mal formés. Si on doit vérifier la syntaxe des nombres, il faut utiliser les fonctions de la famille `strtol` (pour les nombres entiers) ou `strtod` (pour les nombres flottants).

Les fonctions de conversion permettent d'obtenir une valeur numérique en analysant une chaîne de caractères. La fonction inverse est obtenue en utilisant `snprintf`.

La fonction `atof` traduit la chaîne `str` et en extrait une valeur `double`.

Les fonctions `atoi` et `atol` analysent la chaîne `str` et en extraient respectivement un entier et un entier long.

## **strtol, strtoll** — conversion en valeur numérique entière

```
#include <stdlib.h>

long strtol (const char *str, char **endp, int base)
long long strtoll (const char *str, char **endp, int base)
```

Ces fonctions analysent la chaîne de caractères `str` et retournent la valeur trouvée sous forme d'un type `long int` ou `long long int`.

Le paramètre `base` indique la base de numération pour l'analyse, comprise entre 2 et 36. Si cet argument vaut 0, la base est indiquée dans la chaîne par un préfixe suivant la même convention que les nombres en C (`'0'` pour octal, `'0x'` pour hexadécimal).

En fin d'analyse, l'adresse du premier caractère non reconnu est placée dans `endp`. Ceci permet de détecter un nombre invalide (par exemple `'5toto'`).

En cas de conversion réussie, la valeur retournée est la valeur convertie et la variable `errno` n'est pas modifiée.

En cas d'erreur, la valeur retournée est :
- 0 si aucune conversion n'a pu être effectuée (par exemple `'toto'`), auquel cas la variable `errno` vaut `EINVAL`;
- `LONG_MIN` ou `LONG_MAX` en cas de dépassement des limites d'un `long int` (pour `strtol`), auquel cas la variable `errno` vaut `ERANGE`. Avec `strtoll`, les valeurs de retour sont `LLONG_MIN` ou `LLONG_MAX`.

Pour pouvoir détecter ces valeurs, qui peuvent être des valeurs tout à fait normales, il faut remettre `errno` à 0 avant l'appel. Si c'est le cas, le tableau suivant récapitule les différents cas possibles :

| errno | valeur de retour | **endp | Signification |
|---|---|---|---|
| 0 | peu importe | la suite normale de la chaîne | aucune erreur |
| EINVAL | 0 | peu importe | conversion invalide |
| ERANGE | LONG_MIN ou LONG_MAX | peu importe | dépassement de limite |
| peu importe | peu importe | caractère inattendu | nombre mal terminé |

## **strtod, strtof, strtold** — conversion en valeur numérique flottante

```
#include <stdlib.h>

double strtod (const char *str, char **endp)
float strtof (const char *str, char **endp)
long double strtold (const char *str, char **endp)
```

Ces fonctions analysent la chaîne de caractères `str` et retournent la valeur trouvée sous forme d'un type flottant (`double`, `float` ou `long double`).

En fin d'analyse, l'adresse du premier caractère non reconnu est placée dans `endp`. Ceci permet de détecter un nombre invalide (par exemple `'5.0toto'`).

En cas de conversion réussie, la valeur retournée est la valeur convertie et la variable `errno` n'est pas modifiée.

En cas d'erreur, la valeur retournée est :

— 0.0 si aucune conversion n'a pu être effectuée (par exemple `'toto'`), auquel cas la variable `errno` vaut `EINVAL`;

— `+HUGE_VAL` ou `-HUGE_VAL` en cas de dépassement des limites d'un `double` (pour `strtod`), auquel cas la variable `errno` vaut `ERANGE`. Avec `strtof`, ces valeurs sont `+/-HUGE_VALF`. Avec `strtold`, ces valeurs sont `+/-HUGE_VALL`.

Pour pouvoir détecter ces valeurs, qui peuvent être des valeurs tout à fait normales, il faut remettre `errno` à 0 avant l'appel. Si c'est le cas, le tableau suivant récapitule les différents cas possibles :

| `errno` | valeur de retour | `**endp` | Signification |
|---|---|---|---|
| 0 | peu importe | la suite normale de la chaîne | aucune erreur |
| `EINVAL` | 0 | peu importe | conversion invalide |
| `ERANGE` | `+/-HUGE_VAL` | peu importe | dépassement de limite |
| peu importe | peu importe | caractère inattendu | nombre mal terminé |

---

**isalpha, isupper, islower, isdigit, isspace, ispunct, isalnum, isprint, iscntrl, isascii** — test sur des caractères

```
#include <ctype.h>

int isalpha (int c)
int isupper (int c)
int islower (int c)
int isdigit (int c)
int isspace (int c)
int ispunct (int c)
int isalnum (int c)
int isprint (int c)
int iscntrl (int c)
int isascii (int c)
```

Ces fonctions testent un caractère et renvoient une valeur *vrai* ou *faux*. Elles sont à utiliser de préférence, car elles sont souvent plus rapides que des tests d'intervalle et elles sont indépendantes du jeu de caractères utilisé (ASCII, EBCDIC...).

La fonction `isalpha` teste si le caractère est une lettre.

La fonction `isupper` teste si le caractère est une lettre majuscule.

La fonction `islower` teste si le caractère est une lettre minuscule.

La fonction `isdigit` teste si le caractère est un chiffre.

La fonction `isspace` teste si le caractère est un caractère blanc, c'est-à-dire un espace, une tabulation, un retour chariot, un line feed ou un form feed.

La fonction `ispunct` teste si le caractère est un signe de ponctuation, c'est-à-dire ni un caractère de contrôle, ni un caractère alphanumérique.

La fonction `isalnum` teste si le caractère est une lettre ou un chiffre.

La fonction `isprint` teste si le caractère est imprimable, c'est-à-dire un caractère dont le code dans l'alphabet ASCII est compris entre 33 et 126.

La fonction `iscntrl` teste si le caractère est un code de contrôle.

La fonction `isascii` teste si le caractère a un code compris entre 0 et 127.

---

**strcat, strncat** — concaténation de chaînes

```
#include <string.h>

char *strcat (char *dst, const char *src)
char *strncat (char *dst, const char *src, size_t n)
```

La fonction `strcat` réalise une concaténation de la chaîne `src` à la fin de la chaîne `dst`.

La fonction `strncat` concatène au plus `n` caractères de `src` dans `dst`.

Attention : ces deux fonctions peuvent être dangereuses. La fonction `strcat` ne vérifie pas le débordement de la chaîne `dst` et ne peut donc être utilisée que s'il est certain que `dst` contient suffisamment de place. La fonction `strncat` quant à elle recopie au plus `n` octets, donc peut ne pas recopier l'octet nul de fin de chaîne et n'indique pas s'il y a eu troncature. À la place de ces deux fonctions, il est suggéré d'utiliser `snprintf` (voir page 54) en vérifiant la valeur retournée.

Ces deux fonctions renvoient la chaîne `dst` comme résultat.

---

**strcmp, strncmp** — comparaison de chaînes

```
#include <string.h>

int strcmp (const char *str1, const char *str2)
int strncmp (const char *str1, const char *str2, size_t n)
```

La fonction `strcmp` compare les chaînes `str1` et `str2`. La fonction `strncmp` compare au plus les `n` premiers caractères des chaînes `str1` et `str2`.

Le résultat est un entier signifiant :

`< 0` : str1 $<$ str2
`= 0` : str1 $=$ str2
`> 0` : str1 $>$ str2

---

**strcpy, strncpy** — copie de chaînes

```
#include <string.h>

char *strcpy (char *dst, const char *src)
char *strncpy (char *dst, const char *src, size_t n)
```

Attention : la fonction `strcpy` peut être dangereuse car elle ne vérifie pas le débordement de la chaîne `dst` fournie en paramètre. Il vaut mieux en général utiliser `strncpy`, sauf s'il est certain que la place dans `dst` est suffisante.

La fonction `strcpy` copie la chaîne `src` dans la chaîne `dst` (en écrasant l'ancienne valeur de `dst`). La fonction `strncpy` copie au plus `n` caractères.

Attention : ces deux fonctions peuvent être dangereuses. La fonction `strcpy` ne vérifie pas le débordement de la chaîne `dst` et ne peut donc être utilisée que s'il est certain que `dst` contient suffisamment de place. La fonction `strncpy` quant à elle recopie au plus `n` octets, donc peut ne pas recopier l'octet nul de fin de chaîne et n'indique pas s'il y a eu troncature. Il est donc important de savoir si la chaîne `dst` contient suffisamment de place, ou sinon il est préférable d'utiliser la fonction `snprintf` (voir page 54) en vérifiant la valeur retournée.

Ces deux fonctions renvoient la chaîne `dst` en résultat.

---

**strlen** — longueur d'une chaîne

```
#include <string.h>

size_t strlen (const char *str)
```

La fonction `strlen` renvoie la longueur de la chaîne.

---

**strchr, strrchr** — recherche d'un caractère dans une chaîne

```
#include <string.h>

char *strchr (const char *str, int c)
char *strrchr (const char *str, int c)
```

La fonction `strchr` retourne un pointeur sur la première occurrence (la dernière pour `strrchr`) du caractère `c` dans la chaîne `str`.

Ces deux fonctions renvoient un pointeur nul si le caractère n'est pas trouvé.

## 2.4 Gestion du temps

En complément de `time` et `gettimeofday` (voir page 31), les fonctions suivantes permettent d'assurer des conversions entre des dates au format numérique (comme par exemple le type `time_t`) et des chaînes de caractères.

---

**ctime** — conversion simple de `time_t` vers une chaîne

```
include <time.h>

char *ctime (const time_t *horloge)
```

La fonction `ctime` renvoie une chaîne de 26 caractères (non traduite) ayant la forme suivante :

```
Sun Sep 16 01:03:52 1973\n\0
```

L'argument de `ctime` est la valeur de l'horloge telle que renvoyée par la primitive système `time`. L'adresse renvoyée pointe dans une variable interne (statique) de `ctime`.

```
time_t horloge ;

horloge = time (NULL) ;
printf ("la date courante est: %s", ctime (&horloge)) ;
```

---

**localtime, gmtime** — décomposition d'une date en éléments simples (`struct tm`)

```
include <time.h>

struct tm *localtime (const time_t *horloge)
struct tm *gmtime (const time_t *horloge)
```

Les fonctions `localtime` et `gmtime` décomposent une date (au format `time_t`) en éléments simples (heure, minute, jour, mois, etc.) et renvoient un pointeur sur une structure définie dans `time.h`. La fonction `localtime` corrige l'heure en fonction du fuseau horaire, alors que `gmtime` donne l'heure en « temps universel coordonné » (UTC). Les structures `tm` ont la forme suivante :

```
struct tm
{
    int tm_sec ;         // secondes [0..60]
    int tm_min ;         // minutes [0..59]
    int tm_hour ;        // heures [0..23]
    int tm_mday ;        // jour dans le mois [1..31]
    int tm_mon ;         // mois [0..11]
    int tm_year ;        // année (année - 1900)
    int tm_wday ;        // jour de la semaine [0..6]
    int tm_yday ;        // jour dans l'année [0..365]
    int tm_isdst ;       // 1 si heure d'été
} ;
```

**asctime** — conversion de `struct tm` vers une chaîne

```
include <time.h>

char *asctime (const struct tm *tm)
```

La fonction `asctime` traduit une structure `tm` (telle que retournée par la fonction `localtime` par exemple) en une chaîne de 26 caractères, de manière analogue à `ctime`.

---

**strftime** — conversion d'une date en une chaîne de caractères

```
include <time.h>

size_t strftime (char *s, size_t max, const char *format, const struct tm *tm)
```

Cette fonction convertit une date fournie sous forme d'une structure `tm` (voir `localtime`, page 62) en une chaîne de caractères `s` (de taille `max` octets), suivant les conversions indiquées par `format`. Comme avec `printf`, le format est une chaîne de caractères qui contient deux sortes d'objets : les caractères simples, qui sont affichés normalement, et les formats de conversion commençant par un caractère `%`, régis par la syntaxe suivante (les éléments entre crochets sont optionnels) :

`%` [`0`] [+] [<nombre>] [`EO`] *conversion*`%`

- le caractère `0` indique que le caractère « 0 » doit être utilisé pour atteindre le nombre de caractères minimum spécifié ;
- le caractère + indique comme précédemment que le caractère « 0 » doit être utilisé pour atteindre le nombre de caractères minimum spécifié, mais également que si le nombre converti dépasse la limite (4 pour une année, ou 2 pour une année dans le siècle), un « + » ou un « - » doit être ajouté suivant la valeur ;
- vient ensuite le nombre de caractères minimum de la valeur convertie, un ajout de « 0 » ou d'espaces étant effectué si la valeur ne remplit pas ce nombre de caractères ;
- un argument optionnel « `E` » ou « `O` » permet de requérir une représentation alternative de la valeur convertie (dépendant de la localisation) ;
- enfin, le dernier caractère indique la conversion demandée, parmi lesquelles :
  `a` : jour de la semaine (abrégé)
  `A` : jour de la semaine (complet)
  `b` : mois (abrégé)
  `B` : mois (complet)
  `c` : date et heure
  `d` : jour du mois [01..31] (sur 2 chiffres)
  `H` : heure [00..23] (sur 2 chiffres)
  `m` : mois [01..12] (sur 2 chiffres)
  `M` : minute [00..59] (sur 2 chiffres)
  `S` : seconde [00..60] (sur 2 chiffres)
  `T` : équivalent à `%H:%M:%S`
  `Y` : année (sur 4 chiffres)
  `%` : affichage d'un caractère `%`

Par exemple, pour afficher la date et l'heure courante, on peut utiliser :

```
time_t horloge ;
struct tm *tm ;
char date [256] ;
size_t n ;

horloge = time (NULL) ;
tm = localtime (&horloge) ;
n = strftime (date, sizeof date, "Aujourd'hui : %d/%m/%Y %T", tm) ;
if (n > 0)
    printf ("%s\n", date) ;
```

La valeur de retour correspond au nombre d'octets (non compris l'octet nul terminal) placés dans la chaîne résultat, ou 0 si la conversion n'est pas possible (format invalide ou pas assez de place).

**strptime** — conversion d'une chaîne de caractères en une date

```
include <time.h>

char *strptime (const char *chaine, const char *format, struct tm *tm)
```

La fonction `strptime` convertit une date (telle que saisie par l'utilisateur) de la chaîne de caractères `chaine` vers une date représentée par la structure `tm` (voir définition de la fonction `localtime`, page 62), suivant le format indiqué (voir `strftime` pour les principaux formats).

Si la conversion réussit, la fonction renvoie un pointeur vers le caractère suivant la date convertie (dans `chaine`). Sinon, elle renvoie un pointeur nul.

## 2.5 Fonctions associées aux sockets Berkeley

Les fonctions ci-après sont le complément indispensable de l'utilisation des sockets Berkeley. En particulier, certaines d'entre elles permettent de rechercher les informations utiles dans les divers fichiers de configuration du réseau.

**getaddrinfo, gai_strerror, freeaddrinfo** — informations nécessaires pour un dialogue IPv4 ou IPv6

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo (const char *nom, const char *service, const struct addrinfo *indic,
                 struct addrinfo **res0)
const char *gai_strerror (int code)
void freeaddrinfo (struct addrinfo *ai)
```

La fonction `getaddrinfo` recherche les informations nécessaires (adresses IPv4 et/ou IPv6, numéro de port) pour initier un dialogue à travers le réseau.

Dans le cas d'un client, `getaddrinfo` est utilisée pour identifier le service distant (nom de machine ou adresse IP, port TCP ou UDP). Cette fonction renvoie plusieurs résultats (une machine peut avoir plusieurs adresses IPv4 et/ou IPv6), via une liste dont la tête sera pointée en retour par `res0`, destinés à être utilisés avec les primitives `socket` et `connect` (pour TCP) ou `sendto` (pour UDP).

Dans le cas d'un serveur, `getaddrinfo` est utilisée pour identifier un service (port TCP ou UDP). Le nom fourni doit être `NULL` dans ce cas. Cette fonction renvoie plusieurs résultats (par exemple un pour IPv4 et un pour IPv6), via une liste dont la tête sera pointée en retour par `res0`. Ces résultats sont destinés à être utilisés avec les primitives `socket` et `bind` (pour TCP) ou `recvfrom` (pour UDP). En présence de plusieurs sockets, le serveur devra utiliser `select` pour attendre les connexions (via `accept`) en parallèle.

La structure `addrinfo` contient les champs suivants :

```
int ai_flags ;              // flags pour le paramètre indic
int ai_family ;             // famille de protocoles pour socket()
int ai_socktype ;           // type de socket (SOCK_STREAM ou SOCK_DGRAM)
int ai_protocol ;           // protocole TCP, UDP, etc. trouvé
socklen_t ai_addrlen ;      // taille de l'adresse trouvée
struct sockaddr *ai_addr ;  // adresse trouvée
char *ai_canonname ;        // nom canonique trouvé
struct addrinfo *ai_next ;  // suivant dans la liste
```

Le paramètre `indic` fournit des indications sur le type de recherche demandée :

- si le bit `AI_PASSIVE` est positionné dans `ai_flags`, `getaddrinfo` fournit des résultats pour un serveur ;
- le champ `ai_family` est utilisé pour restreindre les adresses à IPv4 (`PF_INET`) ou IPv6 (`PF_INET6`) seulement, ou pour autoriser les deux (`PF_UNSPEC`) ;
- le champ `ai_socktype` est utilisé pour spécifier si la communication est basée sur une connexion fiable (`SOCK_STREAM`, soit TCP) ou sur le mode datagramme (`SOCK_DGRAM`, soit UDP).

La fonction `getaddrinfo` renvoie 0 en cas de réussite. Dans le cas d'une erreur, un code est renvoyé qui peut être converti en chaîne de caractères avec la fonction `gai_strerror`.

La fonction `freeaddrinfo` désalloue la liste retournée par `getaddrinfo` (paramètre `res0`).

---

**getnameinfo** — traduit une adresse IPv4 ou IPv6 en nom

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo (const struct sockaddr *adr, socklen_t longadr, char *nom,
                 size_t longnom, char *service, socklen_t longserv, int flags)
```

La fonction `getnameinfo` convertit une structure `sockaddr` (paramètre `adr`) en un nom de machine et un nom de service (s'il est trouvé dans `/etc/services`).

Les bits du paramètre `flags` modifient la manière dont la conversion est effectuée :

- le bit `NI_NOFQDN` indique que le nom retourné ne doit pas être pleinement qualifié (i.e. sans le domaine) ;
- le bit `NI_NUMERICHOST` indique que le nom retourné doit être sous forme numérique (comme avec un appel à `inet_ntop`) sans recherche dans le DNS ;
- le bit `NI_NAMREQD` indique que la recherche DNS doit forcément retourner un nom. Si aucun nom n'existe pour l'adresse contenue dans le paramètre `sa`, une erreur est retournée ;
- le bit `NI_DGRAM` indique que le service doit être recherché en UDP (et non en TCP par défaut).

La valeur renvoyée est 0 en cas de réussite. Dans le cas contraire, un code d'erreur est renvoyé qui peut être converti en chaîne de caractères avec la fonction `gai_strerror`.

---

**gethostbyname, gethostbyaddr, endhostent** — obtention de l'adresse IPv4

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

struct hostent *gethostbyname (const char *nom)
struct hostent *gethostbyaddr (const char *adresse, int longueur, int type)
int endhostent (void)
```

Attention : ces fonctions sont **obsolètes** car elles ne gèrent que des adresses IPv4. Il faut utiliser `getaddrinfo` et `getnameinfo` à la place.

La fonction `gethostbyname` ouvre le fichier `/etc/hosts` ou accède au serveur de noms (système de nommage DNS), selon la configuration, et obtient la ou les adresses de la machine de nom `nom`. Le résultat est placé dans une structure :

```
struct hostent
{
    char *h_name ;              // nom de la machine
    char **h_aliases ;          // accès aux aliases par h_aliases[i]
    int  h_addrtype ;           // toujours AF_INET
    int  h_length ;             // longueur de l'adresse en octets
    char **h_addr_list ;        // accès aux adresses par h_addr_list[i]
} ;
#define h_addr h_addr_list[0]   // accès facile à la première adresse
```

Les tableaux pointés par `h_aliases` et `h_addr_list` sont terminés par une case nulle pour signaler la fin. Si le nom requis n'existe pas, la valeur `NULL` est retournée.

La fonction `gethostbyaddr` ouvre le fichier `/etc/hosts` ou accède au serveur de noms (système de nommage DNS), selon la configuration, et obtient le nom de la machine d'adresse `adresse` (la longueur de l'adresse est `longueur` octets et le type d'adresse `type` est toujours `AF_INET`). Le résultat est placé dans une structure `hostent`. Si l'adresse requise n'est pas trouvée, la valeur `NULL` est retournée.

La fonction `endhostent` ferme le fichier `/etc/hosts` ou clôt la connexion avec le serveur de noms s'il est configuré.

## getnetbyname, getnetbyaddr, endnetent — lecture de /etc/networks

```
#include <sys/socket.h>
#include <netdb.h>

struct netent *getnetbyname (const char *nom)
struct netent *getnetbyaddr (uint32_t reseau, int type)
int endnetent (void)
```

Attention : ces fonctions sont **obsolètes** car elles ne gèrent que des adresses IPv4.

La fonction getnetbyname ouvre le fichier /etc/networks et obtient le numéro du réseau de nom nom. Le résultat est placé dans une structure :

```
struct netent
{
    char *n_name ;              // nom officiel du réseau
    char **n_aliases ;          // accès aux aliases par n_aliases[i]
    int  n_addrtype ;           // toujours AF_INET
    uint32_t long n_net ;       // numéro de réseau
} ;
```

La fonction getnetbyaddr ouvre le fichier /etc/networks et obtient le nom du réseau de numéro reseau (le type d'adresse type doit être la constante AF_INET). Le résultat est placé dans une structure netent. Si le numéro de réseau demandé n'est pas trouvé, la valeur NULL est retournée.

La fonction endnetent ferme le fichier /etc/networks.

## getprotobyname, getprotobynumber, endprotoent — lecture de /etc/protocols

```
#include <netdb.h>

struct protoent *getprotobyname (const char *nom)
struct protoent *getprotobynumber (int proto)
int endprotoent (void)
```

La fonction getprotobyname ouvre le fichier /etc/protocols et obtient le numéro du protocole de nom nom. Le résultat est placé dans une structure :

```
struct protoent
{
    char *p_name ;              // nom officiel du protocole
    char **p_aliases ;          // accès aux aliases par p_aliases[i]
    int  p_proto ;              // numéro de protocole
} ;
```

La fonction getprotobynumber ouvre le fichier /etc/protocols et obtient le nom du protocole de numéro proto. Le résultat est placé dans une structure protoent. Si le numéro demandé n'est pas trouvé, la valeur NULL est retournée.

La fonction endprotoent ferme le fichier /etc/protocols.

## getservbyname, getservbyport, endservent — lecture de /etc/services

```
#include <netdb.h>

struct servent *getservbyname (const char *nom, const char *proto)
struct servent *getservbyport (int port, const char *proto)
int endservent (void)
```

La fonction getservbyname ouvre le fichier /etc/services et obtient le numéro de port du service de nom nom pour le protocole proto. Le résultat est placé dans une structure :

```
struct servent
{
    char *s_name ;              // nom officiel du service
    char **s_aliases ;          // accès aux aliases par s_aliases[i]
    int  s_port ;               // numéro de port du service
    char *s_proto ;             // protocole à utiliser
} ;
```

La fonction getservbyport ouvre le fichier /etc/services et obtient le nom du service de numéro de port port. Le résultat est placé dans une structure servent. Si le numéro de port demandé n'est pas trouvé, la valeur NULL est retournée.

La fonction endservent ferme le fichier /etc/services.

---

### htonl, htons, ntohl, ntohs — conversions

```
#include <arpa/inet.h>

uint32_t htonl (uint32_t l)
uint16_t htons (uint16_t s)
uint32_t ntohl (uint32_t l)
uint16_t ntohs (uint16_t s)
```

Ces fonctions convertissent des nombres de 16 ou 32 bits de format *host* (suivant le type de processeur) en format *network* (bit le plus significatif en premier) et réciproquement. Ces routines sont fréquemment utilisées avec les fonctions gethost* et getserv* pour écrire des programmes portables.

Lorsque les formats *host* et *network* sont équivalents, ces fonctions (ou macros) sont toujours définies, mais elles renvoient leur argument tel quel.

---

### inet_ntop, inet_pton — conversion d'adresse IPv4 ou IPv6

```
#include <arpa/inet.h>

const char *inet_ntop (int famille, const void *src, char *dst, socklen_t max)
int inet_pton (int famille, const char *src, void *dst)
```

La fonction inet_ntop convertit, dans la famille indiquée par le paramètre famille (soit AF_INET ou AF_INET6) une adresse du format « réseau » (indiquée par le paramètre src, typiquement l'adresse d'une structure in_addr ou in_addr6) vers le format « présentation », c'est-à-dire sous forme d'une chaîne de caractères. Celle-ci est placée à l'adresse indiquée par le paramètre dst et la taille maximum (ex : INET_ADDRSTRLEN ou INET6_ADDRSTRLEN) est indiquée par le paramètre max. Cette fonction renvoie le pointeur nul en cas d'erreur (auquel cas la variable errno indique la raison de l'erreur), ou le pointeur vers la chaîne dst en cas de réussite.

La fonction inet_pton convertit une adresse numérique fournie en format « présentation » (chaîne de caractères indiquée par le paramètre src) vers le format « réseau », placée à l'adresse indiquée par le paramètre dst (typiquement l'adresse d'une structure in_addr ou in_addr6). Le paramètre famille vaut AF_INET ou AF_INET6.

---

### inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof — manipulation d'adresses IPv4

```
#include <arpa/inet.h>

unsigned long inet_addr (const char *chaine)
unsigned long inet_network (const char *chaine)
char *inet_ntoa (struct in_addr in)
struct in_addr inet_makeaddr (int net, int lna)
int inet_lnaof (struct in_addr in)
int inet_netof (struct in_addr in)
```

Attention : ces fonctions sont **obsolètes** car elles ne gèrent que des adresses IPv4. Il faut utiliser `inet_ntop` et `inet_pton` à la place.

Les fonctions `inet_addr` et `inet_network` prennent en entrée une spécification d'adresse ou de réseau IPv4 sous la forme de nombres séparés par des points (par exemple : 127.0.0.1), et renvoient des nombres utilisables dans des structures `in_addr`. Par exemple :

```
struct in_addr a ;
a.s_addr = inet_addr ("127.0.0.1") ;
```

La fonction `inet_ntoa` fait la conversion inverse.

La fonction `inet_makeaddr` construit, à partir du numéro de réseau `net` et du numéro de machine dans le réseau `lna`, l'adresse IPv4 de la machine correspondante. Cette adresse est retournée dans le format *network*.

Les fonctions `inet_lnaof` et `inet_netof` décomposent respectivement, à partir d'une adresse IPv4 `in`, l'adresse en partie « numéro de machine dans le réseau » et en partie « numéro de réseau ».

---

**openlog, syslog, closelog, setlogmask** — messages système

```
#include <syslog.h>

int openlog (const char *nom, int options, int categorie)
int syslog (int prio, const char *format, ...)
int closelog (void)
int setlogmask (int masque)
```

Ces fonctions permettent de simplifier l'écriture des messages des démons (pas forcément liés aux sockets) via le démon `syslogd`. Chaque démon choisit une catégorie et, pour chaque message, une priorité. Le fichier `openlog.h` contient la liste des catégories et priorités utilisables.

La fonction `openlog` initialise l'accès à `syslogd`. Le nom est une chaîne de caractères qui sert à identifier le programme émetteur des messages dans le log. C'est habituellement le nom du programme. Les options permettent notamment d'inscrire avec chaque message le numéro du processus émetteur (option `LOG_PID`), ou alors d'inscrire les messages sur la console du système si `syslogd` ne fonctionne pas (option `LOG_CONS`). La catégorie correspond à la catégorie par défaut de tous les messages émis avec `syslog`.

La fonction `syslog` fonctionne de manière similaire à `printf`. Elle écrit le `format` avec tous ses arguments. Les caractères `%m` sont remplacés par le texte correspondant à `errno` s'il y en a un. Le paramètre `priorité` est la priorité du message.

La fonction `closelog` clôt l'accès à `syslogd`.

La fonction `setlogmask` permet de spécifier les priorités qui doivent être prises en compte par `syslog`. Il est ainsi possible de rejeter automatiquement un ou plusieurs niveaux de priorité.

## 2.6 Fonctions système

Les fonctions ci-après sont orientées vers l'utilisation du système.

---

**ftok** — création d'une clef pour les IPC System V

```
#include <sys/ipc.h>

key_t ftok (const char *fichier, int id)
```

La fonction `ftok` construit une clef adaptée aux primitives (`msgget`, `semget` et `shmget`) à partir d'un nom de fichier et d'un numéro `id`.

Si le fichier n'est pas accessible, la valeur -1 (ou plus exactement `((key_t)-1)`) est retournée.

---

**getcwd** — répertoire courant

```
#include <unistd.h>

char *getcwd (char *buffer, size_t taille)
```

La fonction `getcwd` calcule le chemin du répertoire courant et le place dans le tableau de caractères (déclaré par l'appelant). La fonction `getcwd` n'essayera pas de placer plus de `taille` caractères dans ce tableau.

La valeur retournée est l'adresse du premier caractère du tableau, ou `NULL` s'il n'est pas assez grand ou s'il y a eu une erreur.

---

**getenv** — valeur d'une variable shell

```
#include <stdlib.h>

char *getenv (const char *var)
```

La fonction `getenv` permet de récupérer la valeur d'une variable du shell. Le résultat est un pointeur dans l'environnement courant sur le contenu de la variable, ou 0 si la variable `var` n'existe pas.

---

**getlogin** — nom de l'utilisateur réel

```
#include <unistd.h>

char *getlogin (void)
```

La fonction `getlogin` renvoie le nom de l'utilisateur connecté sur le terminal de contrôle du processus. Cette identité peut être différente de l'identité de l'utilisateur effectif (voir les primitives `getuid` et `geteuid`, page 24).

Cette fonction renvoie l'adresse d'un tableau alloué statiquement, ou le pointeur nul si l'information demandée ne peut être déterminée.

---

**getopt** — analyser les options de la ligne de commande

```
#include <unistd.h>

int getopt (int argc, char *const argv [], const char *optstring)

extern char *optarg ;
extern int optind, opterr ;
```

La fonction `getopt` facilite l'analyse des arguments de la ligne de commande de manière compatible avec les outils standards :

- options dans n'importe quel ordre ou accolées (par exemple : `-a -b` ou `-ab` ou `-ba`, etc.)
- options avec un argument séparé ou collé (par exemple : `-o toto` ou `-ototo`)
- dès qu'un argument ne commence pas par `-`, le traitement des options est terminé
- la fin des options peut être indiquée explicitement avec `--`

La chaîne `optstring` décrit les options valides : une lettre seule décrit une option sans argument, une lettre suivie du caractère « : » est une option qui admet un argument (séparé ou non par des espaces) La variable `optarg` pointe alors sur le texte de l'argument.

Note : la fonction `getopt` d'origine GNU (sur les systèmes Linux en particulier) n'est pas compatible avec POSIX. Pour la rendre compatible, il faut que le paramètre `optstring` commence par le caractère « + ».

La fonction `getopt` renvoie l'option trouvée et place dans la variable `optind` l'indice dans `argv` du prochain argument à traiter. Quand toutes les options sont traitées, `getopt` renvoie -1.

Exemple :

```
int main (int argc, char *argv [])
{
    int c, errflg = 0 ;
    extern char *optarg ;
    extern int optind ;
```

```
    ...
    while ((c = getopt (argc, argv, "abf:o:")) != -1)
        switch (c)
        {
            case 'a' :                      // option sans argument
                aflag++ ;
                break ;
            case 'b' :
                bflag++ ;
                break ;
            case 'f' :                      // option avec argument
                input = optarg ;
                break ;
            case 'o' :
                output = optarg ;
                break ;
            default :                       // option non reconnue ou argument manquant
                errflg++ ;
                break ;
        }
    if (errflg)                             // erreur détectée : afficher un message utile
    {
        fprintf (stderr, "usage: %s ....", argv [0]) ;
        exit (1) ;
    }
    for ( ; optind < argc; optind++)        // traitement des arguments (par exemple)
    {
        fp = fopen (argv [optind], "r") ;
        ...
    }
}
```

---

**isatty, ttyname** — nom du terminal

```
#include <unistd.h>

int isatty (int descripteur)
char *ttyname (int descripteur)
```

La fonction `isatty` renvoie 1 si `descripteur` correspond à un terminal, 0 sinon.

La fonction `ttyname` renvoie un pointeur sur une chaîne contenant le nom du terminal associé au `descripteur`, ou `NULL` si le `descripteur` ne correspond à aucune entrée dans `/dev`. La chaîne retournée est placée dans une zone statique, réutilisée à chaque appel.

---

**mkfifo** — crée un tube nommé

```
#include <sys/stat.h>

int mkfifo (const char *chemin, mode_t mode)
```

La fonction `mkfifo` crée un tube nommé (*fifo*). Les permissions sont initialisées avec la valeur de `mode` (voir `chmod`).

Cette fonction renvoie 0 en cas de création réussie, ou -1 en cas d'erreur.

---

**mktemp** — nom de fichier unique

```
#include <stdlib.h>

char *mktemp (char *modele)
```

Cette fonction est **obsolète**. Il vaut mieux utiliser `tmpfile` (voir page 71), `mkstemp` ou `mkdtemp` (voir page 71).

La fonction `mktemp` remplace le contenu de la chaîne de caractères `modele` par un nom de fichier unique, et renvoie l'adresse de la chaîne.

Le modèle doit être un nom de fichier suivi de 6 caractères `X`. La fonction remplace ces 6 `X` par une lettre et un numéro de telle sorte que le fichier n'ait pas un nom identique à celui d'un fichier existant.

Si `mktemp` ne peut renvoyer de nom de fichier unique, il renvoie la chaîne vide, c'est-à-dire la chaîne dont le premier caractère est `\0`.

---

### **tmpfile** — ouvre un fichier unique

```
#include <stdio.h>

FILE *tmpfile (void)
```

La fonction `tmpfile` crée un fichier unique, l'ouvre et renvoie son descripteur. Le fichier est automatiquement détruit à la fin de l'exécution du processus.

---

### **tmpnam, tempnam** — nom de fichier unique

```
#include <stdio.h>

char *tmpnam (char *adresse)
char *tempnam (const char *repertoire, const char *prefixe)
```

Ces fonctions sont **obsolètes**. Il vaut mieux utiliser `tmpfile` (voir page 71), `mkstemp` ou `mkdtemp` (voir page 71).

La fonction `tmpnam` génère un nom de fichier unique. Ce nom est renvoyé en retour. Si le paramètre `adresse` n'est pas nul, il s'agit d'un tableau d'au moins `L_tmpnam` octets, et le nom est également placé dans ce tableau. Si le paramètre `adresse` vaut `NULL`, le nom est mis dans une chaîne statique réutilisée par chaque nouvel appel.

La fonction `tempnam` est plus complète : le paramètre `repertoire` indique un répertoire où doit être placé le fichier temporaire, et le paramètre `prefixe` est un préfixe d'au plus 5 caractères utilisés comme début du nom unique. Le nom est placé dans un espace alloué avec `malloc` par `tempnam`. Il faut donc libérer cet espace avec `free` après usage.

Si `tempnam` ne peut allouer de la place, la valeur `NULL` est renvoyée.

---

### **mkstemp, mkdtemp** — crée un fichier ou un répertoire temporaire unique

```
#include <stdlib.h>

int mkstemp (char *modele)
char *mkdtemp (char *modele)
```

La fonction `mkstemp` remplace le contenu de la chaîne de caractères `modele` par un nom de fichier unique, crée et ouvre le fichier, et renvoie finalement le descripteur du fichier obtenu.

Le modèle doit être un chemin terminé par 6 caractères `X`. La fonction remplace ces 6 `X` par une combinaison de caractères de telle sorte que le fichier n'ait pas un nom identique à celui d'un fichier existant. Attention, ce remplacement implique que les caractères pointés par le paramètre `modele` doivent être modifiables par la fonction : il ne faut donc pas lui passer une chaîne de caractères constante.

Le fichier est créé par `open` (voir page 16) avec le flag `O_EXCL` et les permissions 0600. L'utilisation du flag `O_EXCL` permet de garantir que le nom de fichier généré à partir du modèle n'existait pas au moment de sa création, et qu'il est donc bien unique.

Si la fonction `mkstemp` réussit à créer un fichier unique et à l'ouvrir, elle renvoie son descripteur. Sinon, elle renvoie la valeur -1. Le paramètre `modele` est réécrit

La fonction `mkdtemp` construit de manière analogue un nom de répertoire unique et le crée avec `mkdir`. Si elle réussit, elle renvoie l'adresse du paramètre `modele`, ou le pointeur `NULL` en cas d'échec.

**perror, strerror** — messages d'erreur du système

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

void perror (const char *str)
char *strerror (int numero)

extern int errno ;
```

Lorsqu'une erreur survient dans une primitive système (les primitives système peuvent être utilisées par les fonctions de la bibliothèque), la variable externe `errno` est initialisée avec le numéro de l'erreur.

La fonction `perror` affiche sur la sortie d'erreur standard le message correspondant, précédé de la chaîne `str` (qui est typiquement le nom du programme courant).

La fonction `strerror` retourne un pointeur sur une chaîne de caractères contenant le message correspondant à l'erreur de numéro `numero`. Cette chaîne ne doit pas être modifiée.

---

**psignal, strsignal** — messages d'erreur du système

```
#include <signal.h>
#include <string.h>

void psignal (int numsig, const char *str)
char *strsignal (int numsig)
```

La fonction `psignal` affiche sur la sortie d'erreur standard une description du signal numéro `signum` précédée de la chaîne `str`.

La fonction `strsignal` retourne un pointeur sur une chaîne de caractères contenant une description du signal numéro `signum`. Cette chaîne ne doit pas être modifiée.

---

**rand, srand** — génération de nombres pseudo-aléatoires

```
#include <stdlib.h>

int rand (void)
void srand (unsigned int semence)
```

La fonction `rand` retourne une valeur pseudo-aléatoire comprise entre 0 et `RAND_MAX` inclus. La valeur de `RAND_MAX` peut différer suivant les implémentations, mais elle est au minimum de $2^{15} - 1$.

La fonction `srand` initialise la `semence` de l'algorithme de génération afin de changer la séquence de génération des valeurs. Par défaut, la valeur initiale de `semence` est 1.

---

**setjmp, longjmp** — saut externe à une fonction

```
#include <setjmp.h>

int setjmp (jmp_buf env)
void longjmp (jmp_buf env, int discriminant)
```

La fonction `setjmp` est assimilable à une étiquette, et `longjmp` à un `goto`. L'avantage de ces routines est que les sauts peuvent intervenir même à l'extérieur d'une procédure ou d'une fonction. La seule condition est qu'un branchement ne peut se faire qu'à un endroit où il y a déjà eu exécution.

`setjmp` sauve l'endroit (pointeur programme et pointeur dans la pile d'exécution) dans le buffer `env`, et renvoie 0.

On peut alors exécuter `longjmp` avec une valeur `discriminant`. L'exécution reprend alors juste après le `setjmp` correspondant, et la pseudo-valeur de retour de `setjmp` est la valeur `discriminant`.

**sigemptyset, sigfillset, sigaddset, sigdelset, sigismember** — manipulation des masques de signaux

```
#include <signal.h>

int sigemptyset (sigset_t *masque) ;
int sigfillset (sigset_t *masque) ;
int sigaddset (sigset_t *masque, int sig) ;
int sigdelset (sigset_t *masque, int sig) ;
int sigismember (const sigset_t *masque, int sig) ;
```

Ces fonctions manipulent des ensembles (masques) de signaux. Le type `sigset_t` est un type *opaque* : sa définition n'est pas connue, mais il y a des fonctions pour le manipuler.

La fonction `sigemptyset` initialise l'ensemble de telle façon qu'aucun signal n'est inclus.

La fonction `sigfillset` initialise l'ensemble de telle façon que tous les signaux soient inclus.

Les fonctions `sigaddset` (resp. `sigdelset`) ajoutent (resp. suppriment) un signal de l'ensemble.

Ces fonctions renvoient 0 si l'opération est réussie, -1 sinon.

La fonction `sigismember` teste si le signal `sig` est dans l'ensemble. La valeur renvoyée est 1 si le signal est dans l'ensemble ou 0 sinon.

---

**sleep** — attend un nombre de secondes

```
#include <unistd.h>

unsigned int sleep (unsigned int secondes)
```

La fonction `sleep` met le processus en attente pendant un temps spécifié par le paramètre `secondes`.

La valeur retournée est 0 si `sleep` s'est terminée normalement sans être interrompue par un signal, ou un temps (en secondes) restant si elle s'est terminée à cause d'un signal.

---

**usleep** — attend un nombre de microsecondes

```
#include <unistd.h>

unsigned int usleep (useconds_t usec)
```

La fonction `usleep` met le processus en attente pendant un temps spécifié par le paramètre `usec` exprimé en nombre de microsecondes.

La valeur retournée est 0 si `sleep` s'est terminée normalement sans être interrompue par un signal, ou -1 en cas d'erreur (incluant le cas d'interruption par un signal).

Cette fonction était normalisée dans les premières versions de POSIX, mais ne l'est actuellement plus. La norme recommande dorénavant d'utiliser `nanosleep` à la place.

---

**nanosleep** — attend un nombre de nanosecondes

```
#include <time.h>

int nanosleep(const struct timespec *duree, struct timespec *reste)
```

La fonction `nanosleep` met le processus en attente pendant un temps spécifié par le paramètre `duree`. Pour la définition de la structure `timespec`, voir les fonctions `clock_*` (page 32).

La structure pointée par le paramètre `reste` contient, si `nanosleep` est interrompue par un signal, le temps restant à attendre.

La valeur retournée est 0 si `nanosleep` s'est terminée normalement sans être interrompue par un signal. Dans le cas contraire, la valeur de retour est -1 et la structure pointée par le paramètre `reste`, s'il est non nul, est mise à jour.

**system** — appeler une commande shell

```
#include <stdlib.h>

int system (const char *commande)
```

La fonction `system` appelle le shell, et lui passe la commande `commande` comme si elle avait été tapée au terminal. L'exécution est suspendue jusqu'à ce que la commande soit finie.

Si `system` ne peut lancer la commande, une valeur négative est renvoyée.

---

**posix_spawn, posix_spawnp** — version réduite de `fork/exec`

```
#include <spawn.h>

int posix_spawn (pid_t *pid, const char *chemin,
                 const posix_spawn_file_actions_t *actions,
                 const posix_spawnattr_t *attrp,
                 char *const argv [], char *const envp [])
int posix_spawnp (pid_t *pid, const char *fichier,
                 const posix_spawn_file_actions_t *actions,
                 const posix_spawnattr_t *attrp,
                 char *const argv [], char *const envp [])
```

Ces deux fonctions sont une version réduite du couple `fork/exec` conçues par POSIX pour les systèmes où l'implémentation de ces deux primitives se révèle difficile. Elles créent un nouveau processus et lancent l'exécution du fichier spécifié.

Les paramètres sont :

- `pid` : en retour, contient l'identificateur du nouveau processus créé ;
- `chemin` ou `fichier` : fichier à exécuter dans le contexte du processus fils, spécifié par son chemin complet (pour `posix_spawn`) ou par son nom (pour `posix_spawnp`, avec une recherche à l'aide de PATH comme pour `execvp`) ;
- `actions` : actions à effectuer sur les ouvertures de fichiers avant de lancer l'exécutable (entre la création du processus et l'appel à `exec`), voir ci-dessous ;
- `attrp` : attributs à positionner pour le nouveau processus créé, voir ci-dessous ;
- `argv` : tableau des arguments, terminé par un pointeur nul (voir `execv`, page 22) ;
- `envp` : environnement, terminé par un pointeur nul (voir `execve`, page 22).

Il est possible de spécifier des actions pour le nouveau processus dès sa création avec l'argument `attrp`, manipulé avec les fonctions `posix_spawnattr_*` (non décrites ici) : modification du masque des signaux, des paramètres d'ordonnancement, etc.

Il est également possible de spécifier des opérations sur les ouvertures de fichiers grâce à l'argument `actions`, manipulé avec les fonctions `posix_spawn_file_actions_*` (non décrites ici : `posix_spawn_file_actions_addopen`, `posix_spawn_file_actions_addclose` et `posix_spawn_file_actions_adddup2`).

La valeur de retour est soit 0 si tout s'est bien passé ou un numéro d'erreur > 0 en cas d'erreur (attention : le numéro d'erreur est renvoyé, et n'est pas placé dans la variable `errno`).

# Chapitre 3

# Threads POSIX

La bibliothèque POSIX de « threads » permet de créer, supprimer et synchroniser des threads de manière portable. Toutes ces fonctions nécessitent l'inclusion du fichier `pthread.h`.

Comme pour les autres objets POSIX, les threads utilisent des types définis dans cet en-tête. L'édition de liens doit être réalisée avec l'option `-l pthread` pour inclure cette bibliothèque.

Sauf exception, la valeur de retour des fonctions est 0 si l'opération s'est bien passée, ou un numéro d'erreur sinon (compatible avec les définitions de la variable globale `errno.h`).

Ce chapitre décrit les fonctions les plus couramment utilisées.

## 3.1 Création, terminaison et gestion des threads

---

**pthread_create** — crée un thread

```
#include <pthread.h>

int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
                    void *(*fonction)(void *), void *arg)
```

Cette fonction crée un nouveau thread pour exécuter la fonction citée en argument. Cette fonction admet elle-même un seul paramètre de type `void *`, dont la valeur est passée lors de l'appel à `pthread_create`.

L'identification du nouveau thread est placée dans la variable pointée par le paramètre `thread`. Si `attr` vaut `NULL`, les attributs du nouveau thread sont les attributs par défaut, sinon `attr` doit pointer sur une variable contenant les attributs initiaux du nouveau thread (qui ne sont plus modifiables *a posteriori*).

Le thread existe jusqu'à ce que l'une des conditions suivantes se réalise :

- la fonction spécifiée en paramètre se termine
- le thread appelle la fonction `pthread_exit`
- un autre thread appelle la fonction `pthread_cancel`

---

**pthread_exit** — termine le thread courant

```
#include <pthread.h>

void pthread_exit (void *valretour)
```

Cette fonction termine le thread courant et rend la valeur `valretour` disponible pour le thread qui va utiliser `pthread_join`.

---

**pthread_join** — attend la terminaison d'un thread donné

```
#include <pthread.h>

int pthread_join (pthread_t thread, void **valretour)
```

Cette fonction attend la terminaison du thread indiqué par l'argument `thread`. Le thread désigné n'est pas forcément un fils du thread courant. La valeur de retour indiquée lors de la terminaison (avec `pthread_exit`) est retournée via l'argument `valretour`.

---

**pthread_self** — retourne l'identité du thread courant

```
#include <pthread.h>

pthread_t pthread_self (void)
```

Cette fonction renvoie l'identité du thread courant, c'est-à-dire la valeur pointée par `pthread` à l'issue de la création par la fonction `pthread_create`.

---

**pthread_equal** — compare deux identités de threads

```
#include <pthread.h>

int pthread_equal (pthread_t t1, pthread_t t2)
```

Cette fonction renvoie une valeur différente de 0 si les deux identités de threads sont identiques, ou 0 si elles sont différentes.

---

**pthread_detach** — détache un thread

```
#include <pthread.h>

int pthread_detach (pthread_t thread)
```

Détache le thread spécifié : ceci signifie que lorsque le thread se terminera, ses ressources seront automatiquement libérées sans attendre un appel à `pthread_join`. Un thread détaché ne peut être la cible d'un appel à `pthread_join`.

---

**pthread_atfork** — actions à exécuter lors d'un fork

```
#include <pthread.h>

int pthread_atfork (void (*prep)(void), void (*papa)(void), void (*fiston)(void))
```

Lorsqu'un thread d'un processus appelle `fork`, le processus fils ne comprend qu'un seul thread, celui qui a appelé `fork`. Les mécanismes de synchronisation (mutex, barrières, etc.) sont dupliqués dans l'état avant le `fork`. La fonction `pthread_atfork` permet d'enregistrer des fonctions à exécuter lors du `fork`, utilisées par exemple pour mettre les mécanismes de synchronisation dans un état connu.

Les fonctions sont indiquées par leur adresse, ou `NULL` pour indiquer l'absence de fonction.

| Fonction | Appelée... |
|---|---|
| `prep` | avant le `fork` |
| `papa` | au retour de `fork` dans le processus père |
| `fiston` | au retour de `fork` dans le processus fils |

Les appels à `pthread_atfork` sont cumulatifs : si plusieurs fonctions sont enregistrées par plusieurs appels à `pthread_atfork`, les multiples fonctions `papa` et `fiston` sont appelées dans l'ordre où elles ont été enregistrées, et les multiples fonctions `prep` sont appelées dans l'ordre inverse où elles ont été enregistrées.

## 3.2 Attributs de création de threads

Les attributs d'un thread peuvent être initialisés lors de sa création avec `pthread_create`. Les fonctions ci-après servent à spécifier ces attributs. Note : seule une sélection des attributs est présentée ici.

---

**pthread_attr_init, pthread_attr_destroy** — initialise ou détruit les attributs

```
#include <pthread.h>

int pthread_attr_init (pthread_attr_t *attr)
int pthread_attr_destroy (pthread_attr_t *attr)
```

La fonction `pthread_attr_init` initialise un objet de type `pthread_attr_t` avec les attributs par défaut. Dès lors, les attributs peuvent être modifiés avec les autres fonctions de cette section.

La fonction `pthread_attr_destroy` sert à détruire l'objet de type `pthread_attr_t` lorsqu'il n'est plus utilisé (i.e. lorsque le ou les threads ont été créés avec `pthread_create`).

---

**pthread_attr_getstacksize, pthread_attr_setstacksize** — taille de la pile

```
#include <pthread.h>

int pthread_attr_getstacksize (const pthread_attr_t *attr, size_t *val)
int pthread_attr_setstacksize (pthread_attr_t *attr, size_t val)
```

Chaque thread dispose d'une pile d'exécution dotée d'une taille finie. Ces fonctions récupèrent ou modifient la taille de la pile du thread qui sera créé par `pthread_create`.

---

**pthread_attr_getdetachstate, pthread_attr_setdetachstate** — attribut « détaché »

```
#include <pthread.h>

int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *val)
int pthread_attr_setdetachstate (pthread_attr_t *attr, int val)
```

Ces fonctions récupèrent ou modifient l'attribut « détaché » qui sera positionné lors de la création du nouveau thread. Le paramètre `val` peut prendre les valeurs suivantes :

- `PTHREAD_CREATE_DETACHED` : le nouveau thread sera créé dans l'état « détaché », comme s'il y avait eu un appel à `pthread_detach;`
- `PTHREAD_CREATE_JOINABLE` : le nouveau thread sera créé dans l'état par défaut, qui autorise ce thread à être la cible d'un appel à `pthread_join`.

## 3.3 Gestion des signaux

Si l'action associée à chaque signal (via `sigaction`) est partagée par tous les threads d'un processus, le masque de signaux est propre à chaque thread. Ceci permet, par exemple, de canaliser vers un thread la prise en compte d'un signal donné.

---

**pthread_kill** — envoie un signal à un thread donné

```
#include <signal.h>

int pthread_kill (pthread_t thread, int signal)
```

Cette fonction envoie le signal `signal` au thread indiqué par le paramètre `thread`. Si `signal` vaut 0, aucun signal n'est envoyé au thread, cette fonction sert alors à tester l'existence du thread.

**pthread_sigmask** — manipulation du masque de signaux

```
#include <signal.h>

int pthread_sigmask (int comment, const sigset_t *nouveau, sigset_t *ancien)
```

L'utilisation de cette fonction est similaire à celle de la primitive `sigprocmask` (voir page 29), mais ne modifie que le masque du thread courant dans un contexte multithreadé.

## 3.4 Verrous exclusifs (mutex)

Les verrous exclusifs (ou verrous d'exclusion mutuelle, ou mutex), constituent le mécanisme élémentaire de synchronisation de threads. Ils reposent sur un objet de type `pthread_mutex_t`, et peuvent avoir des attributs représentés par le type `pthread_mutexattr_t`.

**pthread_mutex_init, pthread_mutex_destroy** — initialise ou détruit un mutex

```
#include <pthread.h>

int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
int pthread_mutex_destroy (pthread_mutex_t *mutex)
```

La fonction `pthread_mutex_init` initialise le verrou exclusif (mutex) indiqué par le paramètre `mutex`, et le place dans l'état « non verrouillé ». Le paramètre `attr` indique les attributs éventuels du mutex, ou vaut `NULL` si les attributs par défaut doivent être utilisés.

La fonction `pthread_mutex_destroy` détruit le mutex spécifié.

**pthread_mutex_lock, pthread_mutex_timedlock, pthread_mutex_trylock** — verrouille un mutex

```
#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex)
int pthread_mutex_trylock (pthread_mutex_t *mutex)

#include <time.h>

int pthread_mutex_timedlock (pthread_mutex_t *mutex, const struct timespec *habs)
```

La fonction `pthread_mutex_lock` attend, si besoin est, que le verrou indiqué par le paramètre devienne libre, et le verrouille. Si le verrou est déjà verrouillé par ce thread, le comportement (erreur, récursif ou non défini) est fonction du type du verrou (voir `pthread_mutex_settype`, page 79).

La fonction `pthread_mutex_timedlock` fonctionne de même, mais attend au plus jusqu'à l'heure spécifiée par `habs`, qui doit être une heure absolue (et non une durée relative à l'heure courante). Voir les fonctions `clock_*` (page 32) pour la définition de la structure `timespec`.

La fonction `pthread_mutex_trylock` tente de verrouiller le mutex indiqué par le paramètre sans attendre. Si l'opération n'est pas possible, cette fonction retourne une erreur.

Si `pthread_mutex_timedlock` se termine à l'expiration du délai sans obtenir le verrou, l'erreur retournée est `ETIMEDOUT`. De même, si `pthread_mutex_trylock` ne peut obtenir le verrou, l'erreur retournée est `EBUSY`.

**pthread_mutex_unlock** — déverrouille un mutex

```
#include <pthread.h>

int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

Cette fonction déverrouille le verrou passé en paramètre. S'il n'avait pas été verrouillé par le même thread, le comportement (erreur ou non défini) est fonction du type du verrou (voir `pthread_mutex_settype`, page 79).

## 3.5 Attributs de création de mutex

Tout comme les attributs d'un thread, les attributs d'un mutex peuvent être spécifiés lors de son initialisation avec `pthread_mutex_init`. Les fonctions ci-après servent à spécifier ces attributs. Note : seule une sélection des attributs est présentée ici.

---

**pthread_mutexattr_init, pthread_mutexattr_destroy** — initialise ou détruit des attributs de mutex

```
#include <pthread.h>

int pthread_mutexattr_init (pthread_mutexattr_t *attr)
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr)
```

La fonction `pthread_mutexattr_init` initialise un objet de type `pthread_mutexattr_t` avec les attributs par défaut. Dès lors, les attributs peuvent être modifiés avec les autres fonctions de cette section.

La fonction `pthread_mutexattr_destroy` sert à détruire l'objet de type `pthread_mutexattr_t` lorsqu'il n'est plus utilisé (i.e. lorsque les verrous ont été initialisés avec `pthread_mutex_init`).

---

**pthread_mutexattr_gettype, pthread_mutexattr_settype** — type de mutex ).

```
#include <pthread.h>

int pthread_mutexattr_gettype (pthread_mutexattr_t *attr, int *val)
int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int val)
```

Le type de mutex précise la manière dont la bibliothèque réagit face à une tentative de verrouillage d'un mutex déjà verrouillé, suivant la valeur `val` :

- `PTHREAD_MUTEX_NORMAL` : rien n'empêche un thread de tenter de verrouiller le mutex si ce même thread l'a déjà verrouillé, entraînant ainsi un interblocage du thread ;
- `PTHREAD_MUTEX_ERRORCHECK` : une erreur est détectée si le thread essaye de verrouiller à nouveau le mutex sans le déverrouiller au préalable, ou si le thread tente de déverrouiller le mutex alors qu'il avait été verrouillé par un autre thread ;
- `PTHREAD_MUTEX_RECURSIVE` : le mutex peut être verrouillé plusieurs fois par le même thread (sans provoquer d'attente), un nombre équivalent de déverrouillages est alors nécessaire pour réveiller les autres threads attendant ce mutex. Le déverrouillage par un autre thread que celui qui a verrouillé résulte en une erreur.
- `PTHREAD_MUTEX_DEFAULT` : selon les systèmes, ce type correspond à l'une des trois valeurs ci-dessus. Il est donc déconseillé de l'utiliser si on cherche un comportement spécifique.

## 3.6 Verrous actifs (spin locks)

Les verrous actifs (ou spin locks) sont similaires aux mutex dans le principe du verrouillage exclusif. La différence est que l'attente, dans le cas des verrous actifs, est réalisée de manière active sans libérer le processeur. Étant donné que l'attente active pénalise l'ensemble du système (les autres processus et threads), les verrous actifs ne doivent être utilisés que pour des très courtes périodes d'attente, lorsque le temps de mise en attente avec un mutex est incompatible avec les exigences de l'application et que le nombre de processeurs est suffisant.

---

**pthread_spin_init, pthread_spin_destroy** — initialise ou détruit un verrou actif

```
#include <pthread.h>

int pthread_spin_init (pthread_spinlock_t *verrou, int partage)
int pthread_spin_destroy (pthread_spinlock_t *verrou)
```

La fonction `pthread_spin_init` initialise le verrou actif indiqué par le paramètre `spin`, et le place dans l'état « non verrouillé ». Le paramètre `partage` peut prendre les valeurs suivantes :

— `PTHREAD_PROCESS_SHARED` : le verrou est partagé entre les threads de plusieurs processus (si la zone de mémoire correspondant au verrou se situe par exemple dans un segment de mémoire partagée) ;
— `PTHREAD_PROCESS_PRIVATE` : le verrou n'est accessible qu'aux threads du processus courant.

La fonction `pthread_spin_destroy` détruit le verrou spécifié.

---

**pthread_spin_lock, pthread_spin_trylock** — verrouille un verrou actif

```
#include <pthread.h>

int pthread_spin_lock (pthread_spinlock_t *spin)
int pthread_spin_trylock (pthread_spinlock_t *spin)
```

La fonction `pthread_spin_lock` attend de manière active, si besoin est, que le verrou indiqué par le paramètre devienne libre, et le verrouille.

La fonction `pthread_spin_trylock` tente de verrouiller le verrou indiqué par le paramètre sans attendre. Si l'opération n'est pas possible, cette fonction retourne une erreur.

---

**pthread_spin_unlock** — déverrouille un verrou actif

```
#include <pthread.h>

int pthread_spin_unlock (pthread_spinlock_t *spin)
```

Cette fonction déverrouille le spin passé en paramètre.

## 3.7 Conditions

Les conditions (ou *condition variables* en anglais) sont un mécanisme de synchronisation qui, associées à un mutex, permettent à un thread d'attendre qu'une condition soit vérifiée par un autre thread.

---

**pthread_cond_init, pthread_cond_destroy** — initialise ou détruit une condition

```
#include <pthread.h>

int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr)
int pthread_cond_destroy (pthread_cond_t *cond)
```

La fonction `pthread_cond_init` initialise la condition indiquée par le paramètre `cond`. Le paramètre `attr` indique les attributs éventuels de la condition, ou vaut `NULL` si les attributs par défaut doivent être utilisés.

La fonction `pthread_cond_destroy` détruit la condition spécifiée.

---

**pthread_cond_wait, pthread_cond_timedwait** — attend une condition

```
#include <pthread.h>

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,
                            const struct timespec *habs)
```

La fonction `pthread_cond_wait` réalise, en une opération atomique (à l'échelle des threads), le déverrouillage du mutex et la mise en attente du thread courant. De manière symétrique, lorsque le thread sera remis sur le processeur, le mutex sera automatiquement verrouillé. Ceci permet d'implémenter des tests élaborés comme :

```
pthread_cond_t cond ;                    // initialisation non décrite ici
pthread_mutex_t mtx ;                    // initialisation non décrite ici

pthread_mutex_lock (&mtx) ;              // début de section critique
```

```
while (travail_a_faire == 0)
    pthread_cond_wait (&cond, &mtx) ;    // attendre en libérant le verrou
// faire le travail
travail_a_faire = 0 ;
pthread_mutex_unlock (&mtx) ;            // fin de section critique
```

La boucle `while` est nécessaire car il peut arriver que `pthread_cond_wait` se termine sans que la raison ayant provoqué l'attente soit satisfaite (*spurious wakeup*).

La fonction `pthread_cond_timedwait` fonctionne de manière similaire à `pthread_cond_wait`, mais attend au plus jusqu'à l'heure spécifiée par `habs`, qui doit être une heure absolue (et non une durée relative à l'heure courante) comme avec `pthread_mutex_timedlock` (voir page 78).

Si `pthread_cond_timedwait` se termine à l'expiration du délai sans que la condition soit réalisée, l'erreur retournée est `ETIMEDOUT`.

---

**pthread_cond_signal, pthread_cond_broadcast** — débloque un ou plusieurs threads

```
#include <pthread.h>

int pthread_cond_signal (pthread_cond_t *cond)
int pthread_cond_broadcast (pthread_cond_t *cond)
```

La fonction `pthread_cond_signal` débloque (au moins) un des threads attendant la condition passée en paramètre. L'ordre dans lequel les threads sont débloqués n'est pas déterministe (et ne correspond sans doute pas à l'ordre dans lequel les threads se sont mis en attente). Par exemple :

```
pthread_cond_t cond ;                    // initialisation non décrite ici
pthread_mutex_t mtx ;                    // initialisation non décrite ici

// spécifier le travail à faire
travail_a_faire = 1 ;
pthread_cond_signal (&cond) ;            // réveiller le thread travailleur
```

Note : si l'appel à `pthread_cond_signal` est effectué antérieurement à l'appel à `pthread_cond_wait` correspondant (c'est-à-dire si aucun thread n'est réveillé), c'est qu'il y a vraisemblablement un problème dans le programme.

La fonction `pthread_cond_broadcast` débloque tous les threads attendant la condition passée en paramètre.

## 3.8 Attributs de conditions

Comme pour les mutex, les conditions possèdent des attributs. En revanche, aucun attribut courant n'est défini de manière portable entre tous les systèmes.

---

**pthread_condattr_init, pthread_condattr_destroy** — initialise ou détruit des attributs de condition

```
#include <pthread.h>

int pthread_condattr_init (pthread_condattr_t *attr)
int pthread_condattr_destroy (pthread_condattr_t *attr)
```

La fonction `pthread_condattr_init` initialise un objet de type `pthread_condattr_t` avec les attributs par défaut. Dès lors, les attributs peuvent être modifiés avec les autres fonctions de cette section.

La fonction `pthread_condattr_destroy` sert à détruire l'objet de type `pthread_condattr_t` lorsqu'il n'est plus utilisé (i.e. lorsque les conditions ont été initialisées avec `pthread_cond_init`).

## 3.9 Verrous lecteurs/écrivains

Les verrous lecteurs/écrivains sont un mécanisme de synchronisation de haut niveau pour répondre au problème classique de multiples lecteurs et écrivains se partageant un espace commun.

---

**pthread_rwlock_init, pthread_rwlock_destroy** — initialise ou détruit un verrou lecteurs/écrivains

```
#include <pthread.h>

int pthread_rwlock_init (pthread_rwlock_t *lock, const pthread_rwlockattr_t *attr)
int pthread_rwlock_destroy (pthread_rwlock_t *lock)
```

La fonction `pthread_rwlock_init` initialise le verrou lecteurs/écrivains par le paramètre `lock`. Le paramètre `attr` indique les attributs éventuels du verrou, ou vaut `NULL` si les attributs par défaut doivent être utilisés.

La fonction `pthread_rwlock_destroy` détruit le verrou spécifié.

---

**pthread_rwlock_rdlock, pthread_rwlock_tryrdlock** — accès pour un lecteur

```
#include <pthread.h>

int pthread_rwlock_rdlock (pthread_rwlock_t *lock)
int pthread_rwlock_tryrdlock (pthread_rwlock_t *lock)
```

La fonction `pthread_rwlock_rdlock` permet l'accès à une zone partagée entre de multiples lecteurs et écrivains. Plus précisément, le thread est autorisé à accéder à la zone si aucun écrivain n'a demandé l'accès à la zone (c'est-à-dire y accède actuellement ou est bloqué en attendant que les lecteurs aient libéré la zone).

La fonction `pthread_rwlock_tryrdlock` tente de réaliser cet accès sans attendre. Si l'opération n'est pas possible, cette fonction retourne une erreur `EBUSY`.

---

**pthread_rwlock_wrlock, pthread_rwlock_trywrlock** — accès pour un écrivain

```
#include <pthread.h>

int pthread_rwlock_wrlock (pthread_rwlock_t *lock)
int pthread_rwlock_trywrlock (pthread_rwlock_t *lock)
```

La fonction `pthread_rwlock_wrlock` permet l'accès à une zone partagée entre de multiples lecteurs et écrivains. Plus précisément, le thread est autorisé à accéder à la zone si aucun autre thread (lecteur ou écrivain) n'a demandé l'accès à la zone.

La fonction `pthread_rwlock_trywrlock` tente de réaliser cet accès sans attendre. Si l'opération n'est pas possible, cette fonction retourne une erreur `EBUSY`.

---

**pthread_rwlock_unlock** — libère l'accès à un verrou lecteurs/écrivains

```
#include <pthread.h>

int pthread_rwlock_unlock (pthread_rwlock_t *lock)
```

Cette fonction libère le verrou lecteurs/écrivains, c'est-à-dire libère l'accès à la zone de mémoire partagée entre de multiples lecteurs et écrivains.

## 3.10 Attributs de verrous lecteurs/écrivains

Les attributs d'un verrou lecteurs/écrivains peuvent être spécifiés lors de son initialisation avec la fonction `pthread_rwlock_init`. Les fonctions ci-après servent à spécifier ces attributs.

**pthread_rwlockattr_init, pthread_rwlockattr_destroy** — initialise ou détruit un attribut de verrou lecteurs/écrivains

```
#include <pthread.h>

int pthread_rwlockattr_init (pthread_rwlockattr_t *attr)
int pthread_rwlockattr_destroy (pthread_rwlockattr_t *attr)
```

La fonction `pthread_rwlockattr_init` initialise un objet de type `pthread_rwlockattr_t` avec les attributs par défaut. Dès lors, les attributs peuvent être modifiés avec les autres fonctions de cette section.

La fonction `pthread_rwlockattr_destroy` sert à détruire l'objet de type `pthread_rwlockattr_t` lorsqu'il n'est plus utilisé (i.e. lorsque les verrous ont été initialisés avec `pthread_rwlock_init`).

**pthread_rwlockattr_getpshared, pthread_rwlockattr_setpshared** — attribut de partage entre processus

```
#include <pthread.h>

int pthread_rwlockattr_getpshared (const pthread_rwlockattr_t *attr, int *val)
int pthread_rwlockattr_setpshared (pthread_rwlockattr_t *attr, int val)
```

Ces fonctions récupèrent ou modifient l'attribut « partage entre processus » qui sera positionné lors de la création du nouveau verrou. Le paramètre `val` peut prendre les valeurs suivantes :

- `PTHREAD_PROCESS_SHARED` : le verrou lecteurs/écrivains est partagé entre les threads de plusieurs processus (si la zone de mémoire correspondant au verrou se situe par exemple dans un segment de mémoire partagée) ;
- `PTHREAD_PROCESS_PRIVATE` : le verrou n'est accessible qu'aux threads du processus courant.

## 3.11 Barrières

Les barrières sont un mécanisme qui permet à $n$ threads d'attendre qu'ils soient tous arrivés à un point (la barrière) avant de le franchir.

**pthread_barrier_init, pthread_barrier_destroy** — initialise ou détruit une barrière

```
#include <pthread.h>

int pthread_barrier_init (pthread_barrier_t *barriere,
                          const pthread_barrierattr_t *attr, unsigned int nb)
int pthread_barrier_destroy (pthread_barrier_t *barriere)
```

La fonction `pthread_barrier_init` initialise la barrière spécifiée par le paramètre `barriere`. Le paramètre `attr` indique les attributs éventuels de la barrière, ou vaut `NULL` si les attributs par défaut doivent être utilisés. Enfin, le paramètre `nb` indique le nombre de threads à synchroniser.

La fonction `pthread_barrier_destroy` détruit la barrière spécifiée.

**pthread_barrier_wait** — attendre le franchissement de la barrière

```
#include <pthread.h>

int pthread_barrier_wait (pthread_barrier_t *barriere)
```

La fonction `pthread_barrier_wait` met le thread en attente. Lorsque les $n$ (nombre indiqué lors de la création de la barrière) threads ont tous appelé cette fonction, elle cesse l'attente, et les $n$ threads peuvent alors reprendre leur exécution. La barrière est alors réinitialisée et peut servir à nouveau.

Lorsque tout se passe bien, la valeur de retour est `PTHREAD_BARRIER_SERIAL_THREAD` pour l'un des threads et 0 pour les $n-1$ autres (ce qui permet à un des threads de jouer un rôle particulier comme celui de coordinateur par exemple). Sinon, un code d'erreur est renvoyé.

## 3.12 Attributs de barrière

Les attributs d'une barrière peuvent être spécifiés lors de son initialisation avec la fonction `pthread_barrier_init`. Les fonctions ci-après servent à spécifier ces attributs.

---

**pthread_barrierattr_init, pthread_barrierattr_destroy** — initialise ou détruit un attribut de barrière

```
#include <pthread.h>

int pthread_barrierattr_init (pthread_barrierattr_t *attr)
int pthread_barrierattr_destroy (pthread_barrierattr_t *attr)
```

La fonction `pthread_barrierattr_init` initialise un objet de type `pthread_barrierattr_t` avec les attributs par défaut. Dès lors, les attributs peuvent être modifiés avec les autres fonctions de cette section.

La fonction `pthread_barrierattr_destroy` sert à détruire l'objet de type `pthread_barrierattr_t` lorsqu'il n'est plus utilisé (i.e. lorsque les barrières ont été initialisées avec `pthread_barrier_init`).

---

**pthread_barrierattr_getpshared, pthread_barrierattr_setpshared** — attribut de partage entre processus

```
#include <pthread.h>

int pthread_barrierattr_getpshared (const pthread_barrierattr_t *attr, int *val)
int pthread_barrierattr_setpshared (pthread_barrierattr_t *attr, int val)
```

Ces fonctions récupèrent ou modifient l'attribut « partage entre processus » qui sera positionné lors de la création de la nouvelle barrière. Le paramètre `val` peut prendre les valeurs suivantes :

- `PTHREAD_PROCESS_SHARED` : la barrière est partagée entre les threads de plusieurs processus (si la zone de mémoire correspondant à la barrière se situe par exemple dans un segment de mémoire partagée) ;
- `PTHREAD_PROCESS_PRIVATE` : la barrière n'est accessible qu'aux threads du processus courant.

## 3.13 Données spécifiques de threads

La bibliothèque de threads POSIX fournit un mécanisme permettant de stocker, à partir de noms communs (les clefs) à tous les threads, des données (de type `void *` : par exemple une zone allouée par `malloc` ou une valeur entière avec conversion de type explicite) spécifiques à chaque thread. Ainsi, si un thread crée une clef (appelons-la *c*), tous les threads verront cette clef *c* avec la valeur `NULL` à la création. Si un thread stocke par la suite une valeur pour cette clef *c*, seul ce thread verra cette valeur, les autres continueront à voir `NULL`.

---

**pthread_key_create, pthread_key_delete** — crée ou supprime une clef

```
#include <pthread.h>

int pthread_key_create (pthread_key_t *clef, void (*destructeur) (void *))
int pthread_key_delete (pthread_key_t clef)
```

La fonction `pthread_key_create` crée une clef, la stocke à l'adresse pointée par `clef`, et lui associe la valeur `NULL` pour l'ensemble des threads. La fonction `destructeur` est appelée lorsque le thread se termine avec comme unique argument la valeur associée à la clef. Cette fonction n'est pas appelée si le paramètre `destructeur` vaut `NULL` ou si la donnée associée à la clef vaut `NULL`.

La fonction `pthread_key_delete` supprime la clef, qui n'est alors plus visible dans aucun des threads. Comme la fonction `destructeur` précisée à la création n'est pas appelée, l'application doit s'assurer que toutes les zones de mémoire allouées dynamiquement dans les divers threads pour cette clef doivent avoir été désallouées.

**pthread_getspecific, pthread_setspecific** — donnée du thread associée à une clef

```
#include <pthread.h>

void *pthread_getspecific (pthread_key_t clef)
int pthread_setspecific (pthread_key_t clef, const void *val)
```

Ces fonctions récupèrent ou mémorisent la valeur associée au paramètre `clef`.

# Index