Ce document est issu du cours de Pierre David à l'UPMC puis à l'UVSQ.
Il a été repris par Albert Cohen, d'abord à l'UVSQ, puis à l'École Polytechnique.

Annexe

Primitives système et bibliothèque C standard

2012–2013

Table des matières

1 Recommandations pour les travaux pratiques et les projets **7**

2 Les appels système **11**
2.1 Définition des erreurs . . . 11
2.2 Définition des types . . . 14
2.3 Accès aux fichiers . . . 15
access . . . 15
chmod . . . 15
chown . . . 16
close . . . 16
creat . . . 17
dup, dup2 . . . 17
fcntl . . . 17
getdirentries . . . 18
link . . . 19
lseek . . . 19
mkdir . . . 20
open . . . 20
read . . . 21
rmdir . . . 21
stat, fstat . . . 21
unlink . . . 23
write . . . 23
2.4 Gestion des processus . . . 24
chdir . . . 24
chroot . . . 24
exec . . . 24
exit . . . 26
fork . . . 27
getpid, getpgrp, getppid . . . 27
getuid, geteuid, getgid, getegid . . . 28
nice . . . 28
plock . . . 29
setpgrp, setsid . . . 29
setuid, setgid . . . 29

wait, waitpid . . . 30
2.5 Tubes . . . 31
pipe . . . 31
2.6 Signaux — interface simplifiée (pseudo-)V7 . . . 32
alarm . . . 32
kill . . . 33
pause . . . 33
signal . . . 34
2.7 Signaux — interface POSIX . . . 35
sigaction . . . 35
sigprocmask . . . 36
sigpending . . . 36
sigsuspend . . . 37
2.8 Horloge du système . . . 37
stime . . . 37
time . . . 37
times . . . 38
2.9 Disques et périphériques . . . 39
fsync . . . 39
ioctl . . . 39
mount . . . 39
sync . . . 40
ulimit . . . 40
umount . . . 40
ustat . . . 41
2.10 Sockets Berkeley . . . 41
accept . . . 42
bind . . . 42
close . . . 43
connect . . . 43
gethostname, sethostname . . . 43
getpeername . . . 43
getsockname . . . 44
getsockopt . . . 44
listen . . . 44
read . . . 45
recv . . . 45
recvfrom . . . 45
select . . . 46
send . . . 46
sendto . . . 46
sethostname . . . 47
setsockopt . . . 47
shutdown . . . 48
socket . . . 48
write . . . 49
2.11 IPC System V . . . 49
msgctl . . . 49
msgget . . . 50

msgsnd, msgrcv . . . 50
shmctl . . . 51
shmget . . . 51
shmat, shmdt . . . 52
semctl . . . 52
semget . . . 53
semop . . . 53
2.12 Divers . . . 54
acct . . . 54
brk, sbrk . . . 55
mmap, munmap, msync . . . 55
mknod . . . 57
profil . . . 57
ptrace . . . 58
sysconf . . . 58
umask . . . 59
uname . . . 59
utime . . . 60

3 Les fonctions de la bibliothèque C **63**
3.1 Fonctions d'entrées / sorties . . . 64
fclose, fflush . . . 64
feof, ferror, clearerr, fileno . . . 64
fopen, freopen, fdopen . . . 65
fread, fwrite . . . 65
fseek, ftell, rewind . . . 66
getc, getchar, fgetc, getw . . . 66
gets, fgets . . . 67
opendir, readdir, closedir . . . 67
popen, pclose . . . 68
printf, fprintf, sprintf . . . 68
putc, putchar, fputc, putw . . . 69
puts, fputs . . . 70
scanf, fscanf, sscanf . . . 70
ungetc . . . 71
3.2 Gestion de la mémoire . . . 71
free . . . 72
malloc, calloc . . . 72
realloc . . . 72
memcpy, memmove . . . 72
memset . . . 73
3.3 Chaînes de caractères . . . 73
atof, atoi, atol . . . 73
isalpha, isupper, islower, isdigit, isspace, ispunct, isalnum, isprint, iscntrl, isascii . . . 74
strcat, strncat . . . 75
strcmp, strncmp . . . 75
strcpy, strncpy . . . 75
strlen . . . 76
strchr, strrchr . . . 76

dirname, basename . . . 76
3.4 Fonctions associées aux sockets Berkeley . . . 77
gethostbyname, gethostbyaddr, endhostent . . . 77
getnetbyname, getnetbyaddr, endnetent . . . 78
getprotobyname, getprotobynumber, endprotoent . . . 78
getservbyname, getservbyport, endservent . . . 79
htonl, htons, ntohl, ntohs . . . 79
inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof . . . 80
openlog, syslog, closelog, setlogmask . . . 80
3.5 Fonctions système . . . 81
ctime, localtime, gmtime, asctime . . . 81
ftok . . . 82
getcwd . . . 82
getenv . . . 83
getopt . . . 83
isatty, ttyname . . . 84
mkfifo . . . 85
mktemp . . . 85
perror, strerror, sys_errlist, sys_nerr . . . 85
rand, srand . . . 86
setjmp, longjmp . . . 86
sigemptyset, sigfillset, sigaddset, sigdelset, sigismember . . . 86
sleep . . . 87
system . . . 87
tmpfile . . . 87
tmpnam, tempnam . . . 88

4 Les debuggers **91**
4.1 Introduction . . . 91

Chapitre 1

Recommandations pour les travaux pratiques et les projets

Afin de vous aider dans la rédaction de vos projets, voici quelques règles de bon sens que nous vous conseillons de suivre.

Règle 1 : *Ne dépassez pas 80 caractères par ligne*

Avec de nombreux éditeurs, lorsqu'on imprime votre projet, les lignes trop longues sont tronquées ou découpées.

Règle 2 : *Indentez avec* au moins *2 espaces*

La largeur d'indentation est un élément important de la lisibilité de vos programmes. Une largeur de 2 espaces est généralement considérée comme la largeur minimum d'indentation.

Règle 3 : *Utilisez des espaces pour améliorer la lisibilité*

Par exemple, plutôt qu'écrire :

```
fn(arg1,arg2);
```

ajoutez quelques espaces en respectant les règles typographiques :

```
fn (arg1, arg2);
```

ou au minimum :

```
fn(arg1, arg2);
```

Règle 4 : *Évitez de perdre trop de temps sur la forme des commentaires*

Les commentaires sont importants... mais pas leur forme, tant qu'ils restent lisibles et raisonablement homogènes. Chaque année, on voit dans les projets des commentaires présentés avec beaucoup de soin, comme par exemple des cadres :

```
/**********************************
 * fonction pour faire ceci ou cela *
 **********************************/
```

ou des alignements :

```
/* ceci est un commentaire                    */
/* ceci est un deuxieme commentaire           */
```

Cela ne sert pas à grand-chose, si ce n'est à vous faire perdre du temps, et à enlaidir votre programme si par malheur les alignements ne sont pas parfaits. Alors, faites au plus simple sans fioriture.

Règle 5 : *Éliminez les commentaires inutiles*

Vous n'êtes pas payé au kilo de commentaire. Mieux vaut souvent pas de commentaire plutôt que des commentaires comme :

```
int x;      /* declaration */
x++;        /* incrementation de x */
```

N'oubliez pas que la personne qui relit connaît déjà le langage C.

Règle 6 : *Écrivez des commentaires utiles*

Les commentaires doivent expliquer le fonctionnement de votre programme, ou éventuellement quelques points épineux. Mettez vous à la place de la personne qui relit votre programme. Si on enlève les instructions, on doit pouvoir comprendre le fonctionnement du programme à partir des commentaires.

Règle 7 : *Présentez les fonctions dans un ordre logique*

Par exemple, choisissez de présenter les fonctions, dans chaque fichier source, du plus haut niveau vers le plus bas niveau, et restez homogènes. Votre lecteur doit pouvoir retrouver facilement vos fonctions.

Règle 8 : *Pas de franglais*

Ne faites pas de mélange entre le français et l'anglais, que ce soit dans les noms de variables, de fonctions, dans les commentaires, etc.

Exemples à ne pas suivre : `clean_liste_utilisateurs()`, `affiche_users()`, etc.

Règle 9 : *Faites des programmes robustes*

Un bon programme est un programme robuste. Il ne s'arrête jamais inopinément avec un `core`, tous les codes de retour des primitives et des fonctions de librairie sont testés.

Règle 10 : *Attention au débordement de chaînes*

Une des erreurs les plus fréquentes est le débordement de chaînes de caractères ou de tableaux (par exemple, l'introduction d'une ligne trop longue par l'utilisateur, etc.).

Règle 11 : *Pour la portabilité, rien ne vaut l'expérience*

Essayez donc de compiler votre programme avec le maximum de compilateurs/système/architecture pour être sûr que vous n'utilisez pas des caractéristiques de tel ou tel compilateur/système/architecture.

Règle 12 : *Lisez les manuels*

Une règle importante pour rédiger des programmes robustes et portables est de bien lire les manuels en ligne.

Règle 13 : *Discriminez les caractéristiques, par les architectures*

Réaliser des programmes portables impose parfois d'utiliser la compilation conditionnelle (`#ifdef`...). Il faut éviter de discriminer suivant les architectures (BSD ou Linux par exemple), mais il vaut mieux discriminer suivant des caractéristiques. Par exemple :

– définir le symbole `UNAME` suivant que le champ de la structure `utmp` contenant le nom d'utilisateur s'appelle `ut_user` ou `ut_name`
– définir le symbole `HAS_SOMETHING` si la primitive `something()` est disponible

Règle 14 : *Ne prenez pas de décision en fonction de l'environnement de l'utilisateur*

N'utilisez pas les variables d'environnement du Shell, car elles sont bien souvent personnelles, sauf pour accéder au repertoire `$HOME`.

Règle 15 : *Il n'y a qu'un seul fichier à modifier, c'est le Makefile*

Mettez tout ce qui est configurable (chemin d'accès à certains fichiers, symboles du préprocesseur pour la portabilité, etc.) dans le fichier `Makefile`. Celui qui compile votre projet ne devrait jamais avoir à modifier les sources de votre projet.

Règle 16 : *Apprenez à distinguer ce qui est installé localement*

Un certain nombre de logiciels sont installés localement, souvent à partir de fichiers sources : par exemple l'arborescence `/usr/local/`, et parfois `/opt/`. Apprenez à reconnaître ce qui est installé localement de ce qui est disponible sur toutes les machines. De plus, ce qui est installé localement ne l'est pas toujours au même endroit sur toutes les machines, car c'est un choix laissé à l'administrateur du système.

En particulier, ne mettez pas de nom absolu dans les `#include`.

Règle 17 : *Mettez les cibles "all" et "clean" dans votre Makefile*

Les cibles traditionnelles `all` et `clean` sont devenues une tradition bien utile. Ne les oubliez pas.

Chapitre 2

Les appels système

L'accès aux services du système Unix est réalisée au moyen des *appels système*, encore appelés *primitives 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 appels par catégorie. Ce sont :

– l'interprétation des erreurs,
– 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 POSIX, et
– ... les inclassables.

Cette classification ne prétend pas être rigoureuse. Certains appels auraient pu être dans plusieurs catégories, il a bien fallu choisir. D'autre part, cette liste n'est pas exhaustive ; les autres primitives sont documentées dans les pages `man`, dans les fichiers d'entête des répertoires `/usr/include/` et `/usr/include/sys/`.

2.1 Définition des erreurs

La plupart des primitives système renvoient une valeur de type `int` au programme appelant. En général, 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.

Pour utiliser cette variable et les macro-définitions associées, il suffit d'inclure la directive suivante.

```
#include <errno.h>
```

L'accès à la signification en clair de ces messages d'erreur se fait souvent par l'intermédiaire de la routine `perror` de la librairie standard, ou de la variable externe `sys_errlist` (également définie dans la librairie).

La liste suivante est la description des erreurs pouvant intervenir. Il faut noter que la variable `errno` n'est pas remise à 0 avant un appel 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` : Operation not permitted
Survient typiquement lors d'une tentative de modification d'une donnée interdite d'accès.

`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 un appel système. Si l'exécution reprend après l'appel 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/EWOULDBLOCK : Resource temporarily unavailable
Survient par exemple lorsque `fork` ne peut trouver une entrée disponible dans la table des processus, ou lorsque l'utilisateur a dépassé son quota de processus, ou lorsqu'un accès non bloquant est impossible.

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 péripérique 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'un appel système est inapproprié 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 l'appel 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.

2.2 Définition des types

Certaines primitives et fonctions de librairie 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 ou de pointeurs, il est intéressant d'utiliser des types définis pour faciliter la lisibilité et la portabilité des programmes. Pour fixer les idées, voici les principaux types utilisés :

- `clock_t` (entier long non signé) : nombre de tops d'horloge (voir `sysconf`, page 58)
- `dev_t` (entier long) : numéro (mineur et majeur) de périphérique
- `uid_t` (entier long) : identificateur d'utilisateur
- `gid_t` (entier long) : identificateur de groupe
- `ino_t` (entier long non signé) : numéro d'inode
- `jmp_buf` (tableau) : buffer pour `setjmp` et `longjmp`
- `key_t` (entier long) : clef utilisée pour les IPC System V
- `mode_t` (entier court non signé) : permissions associées à un fichier
- `off_t` (entier long) : déplacement dans un fichier
- `pid_t` (entier long) : identificateur de processus
- `sigset_t` (tableau) : bitmap utilisée pour les signaux POSIX
- `size_t` (entier non signé) : nombre d'octets
- `ssize_t` (entier) : nombre d'octets signé
- `time_t` (entier long) : nombre de secondes depuis le premier janvier 1970.

2.3 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 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 à la partie données d'un fichier. Notons que 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 *nom, 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

```
int chmod (const char *nom, 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_IGUID`	02000	bit *set group id*
	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, mais plus portable, à `chmod("toto",0640)`.

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 — change le propriétaire d'un fichier

```
#include <unistd.h>

int chown (const char *nom, 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.

Cette primitive renvoie 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 <fcntl.h>

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

La primitive `creat` crée un nouveau fichier, l'ouvre en écriture et renvoie un descripteur de fichier.

Si le fichier existait déjà, sa longueur est remise à 0 et ses protections sont inchangées. Sinon, le fichier est créÂé avec les protections spécifiées par le motif binaire `mode` (voir `chmod`) et le masque de création de fichier (voir `umask`).

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

---

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 primitive 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 <unistd.h>
#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 deplacement - voir lseek */
off_t   l_start;    /* deplacement relatif en octets */
off_t   l_len;      /* taille; tout le fichier si 0 */
pid_t   l_pid;      /* Processus ayant le verrou (F_GETLK) */
```

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

---

getdirentries — lit des entrées dans un répertoire

```
#include <ndir.h>

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

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_namelen` : 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.

La variable `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.

Note : cette primitive n'étant pas disponible dans toutes les implémentations, les fonctions `opendir`, `readdir` etc. de la librairie doivent être utilisées de préférence.

---

link — établit un nouveau lien sur un fichier

```
int link (const char *nom, const char *nouveaunom)
```

La primitive `link` crée un nouveau lien pour fichier existant de nom `nom`. Le nouveau lien (la nouvelle entrée dans le répertoire) porte le nom `nouveaunom`.

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`, `fctnl` ou `pipe`). Le déplacement est régi par par la valeur de `àpartir` :

`SEEK_SET`	0	à partir du début
`SEEK_CUR`	0	à partir de la position courante
`SEEK_END`	0	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/types.h>
#include <sys/stat.h>

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

La primitive `mkdir` crée le répertoire de nom `nom`. Les protections initiales sont spécifiée 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/types.h>
#include <sys/stat.h>
#include <fcntl.h>

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

La primitive `open` ouvre un fichier et renvoie son descripteur. Si le fichier doit être créÂé, `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 créÂé 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 (particulièrement dans le cas d'un 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

```
int rmdir (const char *nom)
```

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

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

---

stat, fstat — consulte le descripteur d'un fichier

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



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

int stat (const char *nom, struct stat *buf)

int fstat (int desc, struct stat *buf)
```

Les primitives stat et fstat lisent la partie système d'un fichier (son *inode*) et la renvoie sous forme d'une structure pointée par buf. Le fichier peut être spécifié par son nom (stat) ou par le descripteur de fichier qui lui est associé (fstat).

Le contenu de la structure buf est défini comme suit :

```
dev_t  st_dev ;
ino_t  st_ino ;
ushort st_mode ;
short  st_nlink ;
ushort st_uid ;
ushort 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_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) :

S_IFREG	fichier ordinaire
S_IFBLK	fichier périphérique (mode bloc)
S_IFCHR	fichier périphérique (mode caractère)
S_IFDIR	répertoire
S_IFIFO	tube ou tube nommé
S_IFLNK	lien symbolique
S_IFSOCK	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

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

---

unlink — supprime le fichier (enlève un lien)

```
#include <unistd.h>

int unlink (const char *nom)
```

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.

2.4 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 *nom)
```

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 *nom)
```

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 cet appel.

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, (char *) 0)

int execv (char *chemin, char *argv [])
```

```
int execle (char *chemin,
            char *arg0, char *arg1, ... char *argn, (char *) 0,
            char *envp [])

int execve (char *chemin, char *argv [], char *envp [])

int execlp (char *fichier,
            char *arg0, char *arg1, ... char *argn, (char *) 0)

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

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 `état` (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 <sys/types.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 <sys/types.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 les process-id effectifs

```
#include <sys/types.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 39. 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 que certaine valeur de `nice` renvoie une valeur assimilable au cas d'erreur.

---

plock — verrouille un segment en mémoire

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

int plock (int requete)
```

La primitive `plock` sert à verrouiller en mémoire le segment *text* (*text lock*), le segment *data* (*data lock*) ou les segments *text* et *data* (*process lock*) du processus appelant.

Les segments verrouillés en mémoire sont insensibles aux routines de *swap*.

Le choix du segment à verrouiller ou du déverrouillage est réalisé grâce au paramètre `requête` :

- `PROCLOCK` : verrouille les segments *text* et *data*,
- `TXTLOCK` : verrouille le segment *text*,
- `DATLOCK` : verrouille le segment *data*, et
- `UNLOCK` : enlève les verrous.

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

---

setpgrp, setsid — change l'identificateur de session

```
#include <sys/types.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 créÂé.

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

```
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.

---

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

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

pid_t wait (int *adresse)

pid_t waitpid (pid_t pid, int *adresse, 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 `adresse` 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	identificateur du processus	numéro du signal	0177
processus terminé par exit	identificateur du processus	argument de exit sur 8 bits	0
processus terminé par signal	identificateur du processus	0	numéro du signal (+0200 si core)
wait interrompue par signal	-1	?	?

La manière portable pour analyser les code 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 terminée 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 anlogue 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.

Note : wait et waitpid sont en réalité des simplifications de l'appel système wait4, beaucoup plus général.

2.5 Tubes

Les tubes sont un moyen de communication entre processus. Une fois un tube créÂé, 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 créÂés à l'aide de la primitive système pipe, alors que les seconds sont créÂés par la fonction de librairie mkfifo (voir page 3.5).

---

pipe — crée un canal de communication

```
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.

2.6 Signaux — interface simplifiée (pseudo-)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 [Ctrl-C]),
– SIGQUIT[1] : abandon (touche [Ctrl-\]),
– SIGILL[1] : instruction illégale,
– SIGTRAP[1] : trace trap,
– SIGFPE[1] : exception en calcul flottant,
– SIGKILL[3] : kill (non masquable ni interceptable),
– SIGBUS[1] : erreur mémoire,
– SIGSEGV[1] : violation de segment,
– SIGSYS : appel inexistant ou argument incorrect,
– SIGPIPE : écriture dans un tube sans lecteur,
– SIGALRM : alarme,
– SIGTERM : signal logiciel de terminaison,
– SIGSTOP[3] : suspension (e.g., debugger) (non masquable ni interceptable),
– SIGTSTP : suspension par terminal,
– SIGCONT[2] : reprise après suspension,
– SIGCHLD[2] : mort d'un fils,
– SIGUSR1 : user defined signal 1,
– SIGUSR2 : user defined signal 2.

Notes :
1 : une image de la mémoire peut être sauvegardée dans un fichier nommée core.
2 : l'action par défaut est d'ignorer le signal, plutôt que terminer (ou suspendre) le processus.
3 : 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 ou 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.

Note : il est possible d'accéder à un contrôle plus fin des alarmes et à plusieurs références temporelles, à l'aide de la primitive `setitimer`.

---

kill — envoie un signal à un processus

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

```
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 la processus appelant, et la fonction de traitement retourne, le processus reprend l'exécution après le point de suspension.

Etant 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_func) (int))) (int);

/*
  Ou de manière équivalente mais plus lisible :

  typedef void (*sighandler_t) (int);
  sighandler_t signal(int sig, sighandler_t action_func);
*/
```

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 32).

Le paramètre `action_func` 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 `action_funct` sera exécutée avec le numéro du signal comme paramètre.

Note : la norme des signaux V7 restaurait l'action par défaut après chaque traitement d'un signal ; ce comportement conduisait à des exécutions non-déterministes, et ce n'est plus le cas depuis que la fonction `signal` est devenue une simple interface au dessus des signaux POSIX.

Les signaux `SIGKILL` et `SIGSTOP` ne peuvent être ni ignorés ni rattrapés.

Cette primitive renvoie l'adresse de l'ancienne fonction en cas de modification réussie, ou -1 en cas d'erreur.

2.7 Signaux — interface POSIX

Cette section décrit des primitives plus riches permettant de contrôler plus précisément le comportement des signaux dans des cas complexes.

Les signaux requis par la norme sont récapitulés dans le tableau ci-dessous (voir page 32 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 : [1]

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'éxecution de l'action (en plus de `sig`)
`int`	`sa_flags`	paramètres affectant le comportement du signal. POSIX n'en définit qu'un : `SA_NOCLDSTOP` pour ne pas envoyer le signal `SIGCHLD` lorsqu'un processus fils est stoppé (en utilisant `ptrace`).

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.

1. Un champ a été supprimé par mesure de simplifiation.

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 librairie *sigsetopts* (voir page 86).

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 librairie *sigsetopts*.

Cette primitive renvoie 0 en cas d'opération réussie, ou -1 en cas 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 librairie *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 librairie *sigsetopts*.

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

2.8 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 GMT 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 GMT, January 1, 1970.

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

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

---

times — renvoie les temps du processus et de ses fils

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

clock_t times (struct tms *buf)
```

La primitive `times` renvoie les temps d'unité centrale du processus et de ses fils dans une structure pointée par le paramètre. Cette 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 58) 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`.

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.

2.9 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 — vide le contenu des buffers internes associés à un fichier sur le disque

```
#include <unistd.h>

int fsync (int desc)
```

La primitive `fsync` provoque l'écriture réelle sur le 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 associés au fichier sont donc vidés.

Cette primitive 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 — monte un système de fichiers

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

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

Le bit de poids faible est utilisé pour contrôler l'écriture dans le système de fichiers. S'il vaut 1, l'écriture est interdite. Sinon, l'écriture est autorisée, moyennant l'accessibilité individuelle de

chaque fichier.

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

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

---

sync — vide le contenu de tous les buffers internes sur les disques

```
void sync (void)
```

La primitive `sync` provoque la sauvegarde sur disque de tous les buffers internes en mémoire. Ceci inclut le super-block, les inodes modifiés et les blocs non encore écrits.

Cette primitive devrait être appelée par tout programme devant examiner un système de fichiers, tel que `fsck` ou `df`, etc. D'autre part, `sync` est obligatoire avant de stopper le système de manière à assurer l'intégrité des données.

---

ulimit — renvoie ou change les limites (en taille de mémoire ou de fichier)

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

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

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

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

---

umount — démonte un système de fichiers

```
int umount (const char *special)
```

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

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

---

ustat — statistiques sur le système de fichiers

```
#include <sys/types.h>
#include <ustat.h>

int ustat (dev_t peripherique, struct ustat *buf)
```

La primitive `ustat` renvoie des informations sur un système de fichiers monté. Le paramètre `périphérique` est le numéro de périphérique identifiant le périphérique contenant le système de fichiers. Le paramètre `buf` est un pointeur vers une structure dont les champs sont :

```
daddr_t f_tfree ;
ino_t   f_tinode ;
char    f_fname [6] ;
char    f_fname [6] ;
int     f_blksize ;
```

- `f_tfree` est le nombre total de blocs libres,
- `f_tinode` est le nombre total d'inodes libres,
- `f_fname` est le nom du système de fichiers,
- `f_fpack` est le nom du volume, et
- `f_blksize` est la taille des blocs en octets.

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

2.10 Sockets Berkeley

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

Erreurs

Les nouvelles constantes d'erreur (pour la variable `errno`) ajoutées 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 connection

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

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

Cette primitive est utilisée dans les sockets de type "connecté". La socket `s` est supposée créÂée 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/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int bind (int s, struct sockaddr *adresse, int 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

```
int close (int s)
```

La primitive `close` est étendue aux connexions IP.

connect — tentative de connection

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int connect (int s, struct sockaddr *adresse, int 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 connection 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 nombre)

int sethostname (char *nom, size_t nombre)
```

La primitive `gethostname` recopie le nom symbolique de l'ordinateur dans la zone identifiée par `nom`. Au plus `nombre` 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 symbolique de l'ordinateur.

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

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

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

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

La primitive `getpeername` renvoie la description de la socket connectée distante dans `adresse` et `longueur`.

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

int getsockname (int s, struct sockaddr *adresse, int *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/types.h>
#include <sys/socket.h>

int getsockopt (
    int s,
    int niveau,
    int option,
    void *valeur,
    int *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

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

Pour accepter des connexions, une socket est d'abord créÂée avec `socket`, puis une file d'attente pour les demandes de connexion est créÂée avec `listen`. Le paramètre `longueur` est le nombre maximum de connexion (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

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

La primitive `read` est étendue aux connexions IP.

---

recv — lecture de données

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

int recv (int s, void *buf, int longueur, int flags)
```

La primitive `recv` attend la réception d'un message à partir d'une socket `s`. Le message de longueur maximum `longueur` 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 valeur de retour est le nombre d'octets reçus, ou -1 si il y a eu erreur.

---

recvfrom — lecture de donnéees

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

int recvfrom (
    int s,
    void *buf,
    int longueur,
    int flags,
    struct sockaddr *exp,
    int longexp)
```

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

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

---

select — attente d'un évènement

```
#include <time.h>

int select (
    int ndescs,
    int *readdescs,
    int *writedescs,
    int *exceptdescs,
    struct timeval *timeout)
```

La primitive `select` attend qu'un des descripteurs spécifiés par `readdescs` ait des données en attente, qu'un des descripteurs spécifiés apr `writedescs` soit prêt à recevoir des données, qu'un des descripteurs spécifiés par `exceptdescs` exhibe une condition exceptionnelle, ou que la durée spécifiée par `timeout` soit écoulée.

La spécification des descripteurs est réalisée par un champ de bits commençant à l'adresse fournie. Le descripteur f est représenté par le bit $1 << f$.

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 `timeout` 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.

---

send — émission de données

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

int send (int s, void *buf, int longueur, int flags)
```

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

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

---

sendto — émission de données

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

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

int sendto (
    int s,
    void *buf,
    int longueur,
    int flags,
    struct sockaddr dest,
    int longdest)
```

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 si il y a eu erreur.

---

sethostname — initialisation du nom de host

```
int sethostname (const char *nom, unsigned int nombre)
```

La primitive `sethostname` permet, si l'utilisateur est *root*, de changer le nom de l'ordinateur.

---

setsockopt — modification des options associées à la socket

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

int setsockopt (
    int s,
    int niveau,
    int option,
    const void *valeur,
    int 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 0), les émissions (si 1), ou les émissions et les réceptions (si 2) 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/types.h>
#include <sys/socket.h>

int socket (int famille, 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 `famille` spécifie une famille d'adresses utilisée pour interpréter les adresses dans les opérations ultérieures. Les familles sont `PF_INET` (IP) et `PF_UNIX` (chemins d'accès dans l'aborescence Unix).

Le paramètre `type` spécifie la sémantique de la connexion. Le type est soit `SOCK_STREAM` (mode connecté, ordonné, fiable, bi-directionnel, dont l'unité est l'octet), soit `SOCK_DGRAM` (mode non connecté, non fiable, dont l'unité est le message de taille fixe et habituellement petite).

Le paramètre `protocole` désigne le protocole utilisé. Normalement, un seul protocole existe pour une famille et un type donné. 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 créÂée, ou -1 en cas d'erreur.

write — émission de données

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

La primitive `write` est étendue aux connexions IP.

2.11 IPC System V

Les IPC System V sont les mécanismes de communication inter-processus (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 *` chaîne, nom de fichier	type = `key_t` entier, convention entre les processus communiquants
nom interne	type = `int` descripteur de fichier	type = `int` 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 créÂée si la clef égale `IPC_PRIVATE` ou si le paramètre `flags` contient le bit `IPC_CREAT`. La file de messages est créÂée 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 créÂée 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 de message (positif) et d'une suite de caractères (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≤ 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 partagé 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 créÂé (avec une taille taille) si la clef égale IPC_PRIVATE ou si le paramètre flags contient le bit IPC_CREAT. Le segment est créÂé 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 créÂé 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 créÂé (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 créÂé 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 créÂé avec la clef IPC_PRIVATE.

Cette primitive renvoie 0 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.

Le paramètre nsops spécifie le nombre de sémaphores dans le groupe concernés par l'action en cours. Le tableau sops (de nsops éléments) décrit l'opération à réaliser sur chacun d'entre eux. Chaque élément de ce tableau est une structure contenant les champs :

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

Pour chaque sémaphore du groupe, sem_op indique l'opération :

- si sem_op < 0 :
  si la valeur courante du sémaphore est supérieure 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.

2.12 Divers

Les primitives système qui suivent ne sont pas classables facilement. Elles font intervenir des concepts aussi différent 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 <sys/acct.h>
```

```
int acct (const char *nom)
```

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

Il faut être super utilisateur pour utiliser cet appel.

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

---

brk, sbrk — modifie la taille du segment de données

```
int brk (const void *fin)

void *sbrk (int increment)
```

Ces deux primitives servent à modifier dynamiquement la taille du segment de données, c'est à dire les variables globales (statiques) et le tas du processus.

La primitive `brk` modifie la taille de telle manière que `fin` soit une adresse valide dans l'espace de données usager.

La primitive `sbrk` ajoute `incrément` octets à la taille du segment de données.

La fonction `malloc` de la librairie standard réalise une allocation d'espace de manière plus souple pour le programmeur.

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.

---

mmap, munmap, msync — établir ou détruire une liaison entre un fichier (ou un périphérique) et le segment de données.

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

void *mmap(void *debut, size_t taille, int prot, int flags,
           int desc, off_t offset);

int munmap(void *debut, size_t taille);

int msync(void *debut, size_t taille, int flags);
```

La primitive `mmap` lie `taille` octets mémoire, de "préférence" à l'adresse `début`, à partir de la position `offset` du fichier désigné par le descripteur `desc`. L'adresse `début` n'est qu'une indication,

doit toujours être multiple de la taille des pages de la mémoire virtuelle, et en pratique est presque toujours inutilisé, c'est-à-dire mis à 0.

La liaison permet de lire et/ou écrire le fichier (ou périphérique) à travers de simples lectures/écritures en mémoire.

En cas de réussite, `mmap` retourne un pointeur sur la zone liée/réservée. En cas d'erreur, la primitive renvoie -1 et `errno` est mis à jour.

La primitive `munmap` prend en compte les modifications éventuelles (voir `msync`) et détruit la liaison pour la zone spécifiée. La liaison est automatiquement détruite lors de la terminaison du processus *mais ce n'est pas le cas lors de la simple fermeture du fichier ainsi lié*.

La primitive `msync` garantit que les modifications éventuelles dans la zone spécifiée sont prises en compte et écrites sur le système de fichiers (ou envoyées au périphérique, si le fichier lié est un périphérique en mode bloc ou caractère). Les `flags` peuvent être ignorés dans un premier temps (mis à 0).

L'argument `prot` décrit le mode de protection mémoire (et doit être compatible avec le mode d'ouverture du descripteur). Il peut être mis à 0, alias `PROT_NONE`, ce qui interdit tout accès... ou au ou binaire des constantes suivantes.

comment	Description
PROT_EXEC	les pages mémoires associées sont exécutables.
PROT_READ	les pages mémoires associées sont lisibles.
PROT_WRITE	les pages mémoires associées sont modifiables.

L'argument `flags` précise le comportement des pages mémoire ainsi réservées et liées. Les plus importants sont les suivants.

comment	Description
MAP_SHARED	la mémoire est partagée avec d'autres processus qui lient également cette région du fichier ; ce flag est exclusif du suivant.
MAP_PRIVATE	les pages mémoire sont "privées", c'est-à-dire que le mécanisme de copie lors d'une écriture est activé, et que les modifications de la zone mémoire n'affectent pas le fichier (inversement, si le fichier est modifié, le comportement n'est pas spécifié).
MAP_ANONYMOUS	(flag non POSIX mais très standard, `fd` doit être mis à -1 pour plus de portabilité) Aucune liaison effective à un fichier est effectuée. Ce mécanisme sert en réalité à la gestion des pages de mémoire virtuelle sous-jacente à la croissance/décroissance des segments associés aux processus (allocation/libération de pages), et donc indirectement à la fonction `malloc`.

Les liaisons effectiées par `mmap` sont intégralement préserées lors d'un `fork`.

mknod — crée un fichier spécial

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

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

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

- 0170000 : type du fichier. Un des bits suivants :
  - 0010000 : fichier spécial fifo,
  - 0020000 : fichier spécial en mode caractère,
  - 0040000 : répertoire,
  - 0060000 : fichier spécial en mode bloc
  - 0100000 ou 0000000 : fichier ordinaire, et
- 0004000 : bit *set user id*
- 0002000 : bit *set group id*
- 0001000 : *sticky bit* (sauver le code après exécution)
- 0000777 : protections du fichier. Construites à partir des bits suivants :
  - 0000400 : lecture par le propriétaire
  - 0000200 : écriture par le propriétaire
  - 0000100 : exécution par le propriétaire
  - 0000070 : idem pour le groupe
  - 0000007 : 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

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


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 (tous les 1/50 secondes) 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.

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

ptrace — tracer un processus

```
#include <ptrace.h>

int ptrace (int requete, int pid, int adresse, int donnee)
```

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. La requête 9 fait exécuter une instruction par le fils.

sysconf — renvoie des paramètres du système

```
#include <unistd.h>

long sysconf (int paramètre)
```

La primitive `sysconf` est une primitive introduite par la norme POSIX pour obtenir les valeurs numériques de quelques paramètres du système. La table ci-dessous donne quelques-uns des paramètres requis par la norme :

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_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	Longeur 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

Cette primitive renvoie la valeur du paramètre demandé, ou -1 en cas d'erreur.

---

umask — renvoie ou modifie le masque de création de fichiers

```
#include <sys/types.h>
#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 <sys/types.h>
#include <utime.h>

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

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

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 3

Les fonctions de la bibliothèque C

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 bibliothèques. 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.

3.1 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 eû 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 un appel système. Attention à utiliser `fflush` avant d'appeler la primitive système.

---

fopen, freopen, fdopen — ouvrir un flux

```
#include <stdio.h>

FILE *fopen (const char *nom, const char *mode)

FILE *freopen (const char *nom, const char *mode, FILE *flux)

FILE *fdopen (int descripteur, const char *mode)
```

La fonction `fopen` ouvre le fichier `nom` 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)

long 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 si il n'y a pas eû 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, getw — lire un caractère sur le flux

```
#include <stdio.h>

int getc (FILE *flux)

int getchar (void)

int fgetc (FILE *flux)

int getw (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.

La fonction `getw` renvoie le prochain mot (`int` en C).

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)
```

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 identique à `gets` sur n'importe quel flux, en recopiant au maximum `max` caractères.

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 <sys/types.h>
#include <dirent.h>

DIR *opendir (const char *nom)

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 nom, 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 [MAXNAMLEN+1]` : nom de l'entrée dans le répertoire (`MAXNAMLEN` vaut 255 dans la plupart des implémentations d'origine Berkeley) ;
- `short d_namlen` : longueur du nom en caractères ;
- `ino_t d_ino` : numéro d'inode du fichier ;

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 un 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, fprintf, sprintf — écriture formattée

```
#include <stdio.h>

int printf (const char *format, ...)

int fprintf (FILE *flux, const char *format, ...)

int sprintf (char *str, const char *format, ...)
```

La fonction `printf` affiche ses données sur la sortie standard, la fonction `fprintf` écrit sur le flux `flux`, et la fonction `sprintf` place le résultat dans la chaîne de caractères `str`.

Chacune de ces fonctions affiche les arguments suivant `format` 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) :

`%` [`-`] [[`0`] <nombre> [`.` <nombre>] [`l`] `doxufegcs%`

- le caractère `-` indique que l'argument doit être cadré à gauche dans son champ,
- le premier nombre indique la dimension du champ. Si l'argument prend plus de place, l'espace est complété avec des blancs, ou avec des `0` si le premier caractère du nombre est `0`,
- le point sert de séparateur avec le nombre suivant,
- le nombre suivant indique le nombre de caractères maximum de la chaîne (format `s`), ou le nombre de chiffres après la virgule dans le cas des formats `f` ou `e`,
- le caractère `l` indique que l'argument est du type long pour les formats `d`, `o`, `x` ou `u`.
- le caractère suivant indique enfin le type de l'argument :
  `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`
  `c` : conversion en caractère
  `s` : conversion en chaîne de caractères
  `%` : affichage d'un caractère `%`

Exemple : 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 π à 10^-5 près :

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

---

putc, putchar, fputc, putw — 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)

int putw (int mot, 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,stdin)`.

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

La fonction `putw` écrit le mot (`int` en C) sur le flux.

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 `puts(s)` est équivalente à `fputs(s,stdout)`.

La fonction `fputs` écrit la chaîne de caractères `str` (terminée par le caractère nul) sur le flux `flux`, et ajoute un caractère de fin de ligne.

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

---

scanf, fscanf, sscanf — lecture formattée

```
#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 être identique à celui qui existait.

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

3.2 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 0 si il n'y a plus de place ou si il y a eû 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.

La fonction `realloc` renvoie le pointeur sur l'espace mémoire, ou 0 si il n'y a plus de place ou si il y a eû 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 fonction `bcopy` est la version Berkeley.

La valeur de retour est `bloc1`.

---

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 fonction `bzero` est la version Berkeley (restreinte à l'initialisation à 0).

La valeur de retour est `bloc`.

3.3 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

```
double atof (const char *str)

int atoi (const char *str)

long atol (const char *str)
```

Les fonctions de conversion permettent d'obtenir une valeur numérique en analysant une chaîne de caractères. La fonction inverse est obtenue en utilisant `sprintf`.

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.

---

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 caractère 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 ffed 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 *str1, const char *str2)

char *strncat (char *str1, const char *str2, size_t n)
```

La fonction `strcat` réalise une concaténation de la chaîne `str2` à la fin de la chaîne `str1`.

La fonction `strncat` concatène au plus `n` caractères de `str2` dans `str1`.

Ces deux fonctions renvoient la chaîne `str1` 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 *str1, const char *str2)

char *strncpy (char *str1, const char *str2, size_t n)
```


La fonction `strcpy` copie la chaîne `str2` dans la chaîne `str1` (en écrasant l'ancienne valeur de `str1`). La fonction `strncpy` copie au plus `n` caractères.

Ces deux fonctions renvoient la chaîne `str1` en résultat.

---

strlen — longueur d'une chaîne

```
#include <string.h>

int 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.

---

dirname, basename — décomposer des chemins d'accès

```
#include <libgen.h>

char *dirname (const char *path)
char *basename (const char *path)
```

La fonction `dirname` tronque le chemin d'accès `path` jusqu'au dernier caractère `/`, et la fonction `basename` ne conserve que le nom de fichier qui suit ce dernier caractère `/`.

Les deux fonctions renvoient un pointeur sur la sous-chaine résultante.

Attention : ces fonctions modifient la chaine `path` en place.

3.4 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.

---

gethostbyname, gethostbyaddr, endhostent — obtention de l'adresse IP

```
#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)
```

La fonction `gethostbyname` ouvre le fichier `/etc/hosts` (ou accède au serveur de noms s'il est configuré) 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 ;     /* acces 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 ;   /* acces aux adresses par h_addr_list[i] */
} ;
#define h_addr h_addr_list[0] /* acces facile a la premiere 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 s'il est configuré) 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 clot la connection 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)
char *nom ;

struct netent *getnetbyaddr (int reseau, int type)

int endnetent (void)
```

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 netice */
    char **n_aliases ;     /* acces aux aliases par p_aliases[i] */
    int  n_addrtype ;      /* toujours AF_INET */
    unsigned long n_net ;  /* numero de reseau */
} ;
```

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 ;     /* acces aux aliases par p_aliases[i] */
    int  p_proto ;         /* numero 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)

struct servent *getservbyport (int port)

int endservent (void)
```

La fonction `getservbyname` ouvre le fichier `/etc/services` et obtient le numéro de port du service de nom `nom`. Le résultat est placé dans une structure :

```
struct servent
{
    char *s_name ;         /* nom officiel du service */
    char **s_aliases ;     /* acces aux aliases par p_aliases[i] */
    int  s_port ;          /* numero de port du service */
    char *s_proto ;        /* protocole a 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 <netinet/in.h>

unsigned long htonl (unsigned long l)

unsigned short htons (unsigned short s)

unsigned long ntohl (unsigned long l)

unsigned short ntohs (unsigned short 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 sont toujours définies, mais elles sont vides.

---

inet_addr, inet_network, inet_makeaddr, inet_lnaof, inet_netof — manipulation d'adresses IP

```
#include <sys/socket.h>
#include <netinet/in.h>
#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)
```

Les fonctions `inet_addr` et `inet_network` prennent en entrée une spécification d'adresse ou de réseau IP 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 IP 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 IP `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` clot 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.

3.5 Fonctions système

Les fonctions ci-après sont orientées vers l'utilisation du système.

---

ctime, localtime, gmtime, asctime — date et heure

```
include <time.h>

char *ctime (const time_t *horloge)

struct tm *localtime (const time_t *horloge)

struct tm *gmtime (const time_t *horloge)

char *asctime (const struct tm *tm)
```


La fonction `ctime` renvoie une chaîne de 26 caractères 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 l'appel système `time`. Le plus simple est habituellement de procéder comme suit :

```
{
    long int horloge ;
    extern long int ctime () ;
    extern char *ctime () ;

    horloge = time (0L) ;
    strcpy (str, ctime (&horloge)) ;
}
```

Les fonctions `localtime` et `gmtime` renvoient des pointeurs sur des structures définies dans *time.h*. `localtime` corrige l'heure en fonction du fuseau horaire, alors que `gmtime` donne l'heure au méridien de Greenwich. Les structures `tm` ont la forme suivante :

```
struct tm
{
    int tm_sec ;        /* secondes (0..59) */
    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 ;       /* annee (annee - 1900) */
    int tm_wday ;       /* jour de la semaine (0..6) */
    int tm_yday ;       /* jour dans l'annee (0..365) */
    int tm_isdst ;      /* 1 si heure d'ete */
} ;
```

---

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 part 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 <stdlib.h>

char *getcwd (char *buffer, size_t taille)
```

La fonction `getcwd` calcule le nom 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 retourbée est l'adresse du premier caractère du tableau, ou `NULL` s'il n'est pas assez grand ou s'il y a eû 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.

---

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` permet d'analyser les arguments de la ligne de commande. la chaîne `optstring` décrit les options valides : une lettre seule décrit une option sans argument, une lettre suivie d'un 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.

`getopt` place dans la variable `optind` l'indice dans `argv` du prochain argument à traiter. Quand toutes les options sont traitées, `getopt` renvoie `EOF`.

Exemple :

```
main (argc, argv)
int argc ;
char *argv [] ;
{
    int c, errflg = 0 ;
    extern char *optarg ;
    extern int optind ;

    ...
    while ((c = getopt (argc, argv, "abf:o:")) != EOF)
        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 ;
            case '?' :     /* option non reconnue */
                errflg++ ;
                break ;
        }
        if (errflg)
        {
            fprintf (stderr, "usage:....") ;
            exit (2) ;
        }
        for ( ; optind < argc; optind++)
        {
            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 *nom, 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 <unistd.h>

char *mktemp (char *modele)
```

Cette fonction est obsolète. Il vaut mieux utiliser `tmpfile` et `tmpnam` (voir page 87).

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`.

perror, strerror, sys_errlist, sys_nerr — messages d'erreur du système

```
#include <errno.h>

void perror (const char *str)

char *strerror (int numero)

extern int errno ;
extern int sys_nerr ;
extern char *sys_errlist [] ;
```

Lorsqu'une erreur survient dans un appel système (les appels systèmes sont utilisés par les routines de la bibliothèque), la variable exerne `errno` est initialisée avec le numéro de l'erreur.

La fonction `perror` affiche 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 correspond à l'erreur de numéro `numero`. Cette chaîne ne doit pas être modifiée.

Le message correspondant à l'erreur est stocké dans le tableau de chaînes `sys_errlist`, dont les indices varient de 0 à `sys_nerr` - 1. La variable `errno` est l'indice du message correspondant à l'erreur détectée.

rand, srand — génération de nombres aléatoires

```
#include <stdlib.h>

int rand (void)

void srand (unsigned int semence)
```

La fonction `rand` retourne une valeur pseudo-aléatoire comprise entre 0 et $2^{15} - 1$.

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 eû 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

```
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.

---

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.

---

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 *tmpnam ((char *) 0)

char *tempnam (const char *repertoire, const char *prefixe)
```

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` est nul, le nom est mis dans une chaîne statique réutilisée par chaque nouvel appel.

La fonction `tempnam` est plus complète, mais n'est pas normalisée par l'ANSI comme `tmpnam`. 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.

Chapitre 4

Les debuggers

Le debugger est l'outil indispensable de mise au point d'un programme. Cette courte notice a pour objet de présenter les fonctions essentielles du debugger `gdb` (le debugger de GNU). On se référera aux documentations pour plus de détails.

4.1 Introduction

Un debugger permet :

- de contrôler l'exécution d'un programme : exécution en mode ¡¡ pas à pas ¿¿, pose de ¡¡ points d'arrêt ¿¿, affichage de variables, etc.
- d'analyser les raisons de la terminaison brutale d'un programme, grâce au fichier `core` généré : affichage de l'instruction qui a causé la terminaison, des variables au moment où le programme s'est arrêté, etc.

Pour pouvoir utiliser un debugger, il faut procéder à la compilation et à l'édition de liens du programme avec l'option `-g`. Cette option ajoute au fichier exécutable des informations permettant au debugger de retrouver les adresses des instructions et des variables.

Les debuggers sous Unix fonctionnent de manière similaire. Pour contrôler l'exécution d'un programme, le processus exécutant le debugger génère un processus fils pour le programme à déboguer, puis en contrôle [1] son exécution.

1. Par le biais de la primitive système `ptrace`.

Commande	Signification
print/p exp	affiche l'expression (voir plus bas)
print/p /format exp	affiche l'expression suivant le format
x adresse	affiche la mémoire à l'adresse spécifiée
list/l fonction	affiche la fonction source

Commande	Signification
list/l ligne	affiche les lignes autour de la ligne spécifiée
list/l fichier	affiche les premières lignes du fichier source
list/l fichier :ligne	affiche las lignes autour de la ligne spécifiée
list/l	affiche les lignes suivantes
search [chaîne]	cherche la chaîne dans le fichier
reverse-search [chaîne]	cherche la chaîne dans le fichier (en remontant)
backtrace/bt	affiche l'empilement des fonctions dans la pile
up [nb]	remonte de nb niveaux dans la pile
down [nb]	descend de nb niveaux dans la pile
frame/f [profondeur]	sélectionne la fonction à la profondeur spécifiée dans la pile
info locals	affiche les variables locales de la fonction
run [args]	lance le programme avec les arguments spécifiés
kill	termine le programme courant
continue/c	continue l'exécution jusqu'au prochain point d'arrêt
continue/c fonction	continue l'exécution jusqu'à la fonction
continue/c ligne	continue l'exécution jusqu'à la ligne spécifiée
continue/c fichier :ligne	continue l'exécution jusqu'à la ligne spécifiée
step/s [nb]	exécute nb instructions
next/n [nb]	exécute nb instructions, en comptant les appels de de fonctions comme de simples instructions
finish	continue l'exécution jusqu'à la fin de la fonction courante
info break	liste tous les points d'arrêts
break/b	pose un point d'arrêt sur la ligne courante
break/b fonction	pose un point d'arrêt sur la première instruction de la fonction
break/b ligne	pose un point d'arrêt sur la ligne
break/b fichier :ligne	pose un point d'arrêt sur la ligne
rbreak regexp	pose un point d'arrêt sur toutes les fonctions correspondant à l'expression régulière
clear	supprime le point d'arrêt sur la ligne courante
clear fonction	supprime le point d'arrêt sur la première instruction de la fonction
clear ligne	supprime le point d'arrêt sur la première instruction suivant la ligne
clear fichier :ligne	supprime le point d'arrêt sur la première instruction suivant la ligne
delete/d	supprime tous les points d'arrêts
dir "répertoire"	ajoute un répertoire pour la localisation des fichiers source

Pour afficher des variables, il faut utiliser la commande ¡¡ p ¿¿. L'expression peut être une expression en C quelconque. Par exemple : `p *(x->fg->d.tab [i])+3`. Le format permet de préciser le type de donnée à afficher. Quelques formats parmi les plus courants sont cités ci-dessous :

Format	Signification
d	décimal
u	décimal non signé
x	hexadécimal
t	binaire
c	caractère
s	chaîne de caractères

Pour modifier une variable, il suffit d'utiliser une expression d'affectation. Par exemple : `p y=5`.

Index

accept, 42
access, 15
acct, 54
alarm, 32
asctime, 81
atof, 73
atoi, 73
atol, 73
bcopy, 72
bind, 42
brk, 55
bzero, 73
calloc, 72
chdir, 24
chmod, 15
chown, 16
chroot, 24
clearerr, 64
closedir, 67
closelog, 80
close, 16, 43
connect, 43
creat, 17
ctime, 81
dup2, 17
dup, 17
endhostent, 77
endnetent, 78
endprotoent, 78
endservent, 79
execle, 24
execlp, 24
execl, 24
execve, 24
execvp, 24
execv, 24
exec, 24
exit, 26
fclose, 64
fcntl, 17
fdopen, 65
feof, 64
ferror, 64
fflush, 64
fgetc, 66
fgets, 67
fileno, 64
fopen, 65
fork, 27
fprintf, 68
fputc, 69
fputs, 70
fread, 65
free, 72
freopen, 65
fscanf, 70
fseek, 66
fstat, 21
fsync, 39
ftell, 66
ftok, 82
fwrite, 65
getchar, 66
getcwd, 82
getc, 66
getdirentries, 18
getegid, 28
getenv, 83
geteuid, 28
getgid, 28
gethostbyaddr, 77
gethostbyname, 77
gethostname, 43
getnetbyaddr, 78
getnetbyname, 78
getopt, 83

getpeername, 43
getpgrp, 27
getppid, 27
getprotobyname, 78
getprotobynumber, 78
getservbyname, 79
getservbyport, 79
getsockname, 44
getsockopt, 44
gets, 67
getuid, 28
getw, 66
gmtime, 81
htonl, 79
htons, 79
inet_addr, 80
inet_lnaof, 80
inet_makeaddr, 80
inet_netof, 80
inet_network, 80
ioctl, 39
isalnum, 74
isalpha, 74
isascii, 74
isatty, 84
iscntrl, 74
isdigit, 74
islower, 74
isprint, 74
ispunct, 74
isspace, 74
isupper, 74
kill, 33
link, 19
listen, 44
localtime, 81
longjmp, 86
lseek, 19
malloc, 72
memcpy, 72
memmove, 72
memset, 73
mkdir, 20
mkfifo, 85
mknod, 57
mktemp, 85
mmap, 55
mount, 39
msgctl, 49
msgget, 50
msgrcv, 50
msgsnd, 50
munmap, 55
nice, 28
ntohl, 79
ntohs, 79
opendir, 67
openlog, 80
open, 20
pause, 33
pclose, 68
perror, 85
pipe, 31
plock, 29
popen, 68
printf, 68
profil, 57
ptrace, 58
putchar, 69
putc, 69
puts, 70
putw, 69
rand, 86
readdir, 67
read, 21, 45
realloc, 72
recvfrom, 45
recv, 45
rewind, 66
rmdir, 21
sbrk, 55
scanf, 70
seekdir, 67
select, 46
semctl, 52
semget, 53
semop, 53
sendto, 46
send, 46
setgid, 29
sethostname, 43, 47
setjmp, 86
setlogmask, 80
setpgrp, 29
setsid, 29
setsockopt, 47
setuid, 29
shmat, 52

shmctl, 51
shmdt, 52
shmget, 51
shutdown, 48
sigaction, 35
sigaddset, 86
sigdelset, 86
sigemptyset, 86
sigfillset, 86
sigismember, 86
signal, 34
sigpending, 36
sigprocmask, 36
sigsuspend, 37
sleep, 87
socket, 48
sprintf, 68
srand, 86
sscanf, 70
stat, 21
stime, 37
strcat, 75
strchr, 76
strcmp, 75
strcpy, 75
strerror, 85
strlen, 76
strncat, 75
strncmp, 75
strncpy, 75
strrchr, 76
sync, 40
sys_errlist, 85
sys_nerr, 85
sysconf, 58
syslog, 80
system, 87
telldir, 67
tempnam, 88
times, 38
time, 37
tmpfile, 87
tmpnam, 88
ttyname, 84
ulimit, 40
umask, 59
umount, 40
uname, 59
ungetc, 71
unlink, 23
ustat, 41
utime, 60
waitpid, 30
wait, 30
write, 23, 49