POSIX

Avant-propos

POSIX = *Portable Operating System Interface [for Unix]*.
Un programme de la norme POSIX commence par la macro `_POSIX_SOURCE` : `#define _POSIX_SOURCE 1`

Compilation

Fichier `Makefile`

```
<cible>: <liste dependances>
    <commandes>
```

Evaluation :
Analyse des prérequis (récursif) ;
Exécution des commandes.
Variables : `CFLAGS=-Wall -Werror`
Utilisation : `$(CFLAGS)` (`$CFLAGS` référence la variable `v`)
`.PHONY:all,clean` : Règles ne générant pas de fichiers
Compilateur GCC :
1. Préprocesseur C (cpp) : substitutions textuelles ;
2. Compilateur C (cc1) : analyses, génération, optimisation ;
3. Assembleur (as) ;
4. Edition de liens (ld).

Options de `gcc` :

`-ansi`	Respect standard ANSI
`-c`	cpp + cc1 + as ; pas ld
`-g`	Informations de déboguage
`-D` (Define)	Définir une macro
`-M` (Make)	Générer une description des dépendances pour chaque fichier
`-H` (Header)	Afficher le nom de chaque fichier header utilisé
`-I` (Include)	Etendre le chemin de recherche des fichiers headers (`/usr/include`)
`-L` (Library)	Etendre le chemin de recherche des libraires (`/usr/lib`)
`-l` (library)	Utiliser une bibliothèque (`lib<library>.a`) pendant l'édition de liens
`-o` (Output)	Rediriger l'output dans un fichier (par défaut `a.out`)

Bibliothèques statiques

```
ar rcv maLib.a f2.o f3.o
ranlib maLib.a //index
```

Equivalent à `ar rcvs maLib.a f2.o f3.o`
Fichiers header (`*.h`)
Prototypes (signatures) des fonctions utilisées dans les fichiers `*.c`. Inclusion dans les fichiers `*.c` :

```
#include "f2.h" //repertoire courant
#include <stdlib.h> //librairie standard
```

Définir des variables :

```
#ifndef PI
#define PI 3.14
#endif
x = PI // x = 3.14
```

La commande `gcc -DPI=3.14` créé la macro `PI`.
Utilisation pour le debug :

```
#ifdef DEBUG
    printf("Information de debug.\n");
#endif
```

Commande gcc : `gcc -DDEBUG`

Environnement de programmation

```
int main(int argc, char * argv[], char ** env)
```

`argc` : nombre d'arguments (nom prog inclus) ;
`argv` : tableau d'arguments (`argv[0]` : nom prog) ;
`env` : variables d'environnement (couple `key=value`).

Flux standards :

`stdin` (<) : entrée standard ;
`stdout` (>) : sortie standard ;
`stderr` (2>) : sortie erreur.

Numéro d'erreur généré par une fonction dans une variable externe `errno`.

Processus UNIX

Définition. Correspond à l'exécution d'un programme binaire caractérisé par :

Numéro unique : **pid** ;
Utilisateur propriétaire : **uid** ;
Groupe de l'utilisateur propriétaire : **gid** ;
Répertoire courant ;
Contexte d'exécution :
`text` : code ;
`data` : données initialisées ;
`bss` : données non initialisées ;
`heap` (tas) : variables globales + `malloc` ;
`stack` (pile) : variables locales, paramètres ;
`U-area` : `argv[]`, `envp[]`...

Allocation dynamique de mémoire

```
#include <stdlib.h>
void * malloc(size_t size);
```

Alloue bloc de `size` octets

```
void * calloc(size_t nb, size_t size);
```

Alloue un bloc de `nb*size` octets initialisés à 0

```
void * realloc(void * ptr, size_t size);
```

Redimensionne un bloc mémoire (conserve son contenu)

```
void free(void * ptr);
```

Libère la mémoire allouée

```
int brk(void *ptrFin);
```

positionne la fin du tas à l'adresse `ptrFin`

```
void * sbrk(intptr_t inc);
```

Incrémente le tas de `inc` octets et retourne l'emplacement de la limite modifiée.

Etats d'un processus :

Elu (*running*) : instructions du processus sont en train d'être exécutées ;
Bloqué (*waiting*) : processus en attente d'une ressource, en suspension ;
Prêt (*ready*) : processus en attente d'être affecté au processeur ;
Terminé (*zombie*) : processus fini, mais son père n'a pas pris connaissance de sa terminaison.

```
pid_t getpid(void);
```

Retourne le pid du processus courant

```
pid_t getppid(void);
```

Retourne le pid du père du processus courant

Un processus est lié à un utilisateur (UID : `uid_t`) et à son groupe (GID : `gid_t`) :

Réel : droits associés à l'utilisateur/groupe qui lance le programme :
`uid_t getuid(void);` (`setuid`)
`gid_t getgid(void);` (`seteuid`)
Effectif : droits associés au programme lui-même :
`uid_t geteuid(void);` (`setgid`)
`gid_t getegid(void);` (`setegid`)

Création d'un processus

```
pid_t fork(void);
```

Permet la création dynamique d'un nouveau processus fils qui s'exécute de façon concurrente avec le processus père qui l'a créé. Le processus fils créé est une copie du processus père.
Valeur de retour :
-1 : erreur dans `errno` :
`ENOMEM` : plus de mémoire disponible ;
`EAGAIN` : trop de processus créés.
0 : renvoyé au fils créé (pid du père : `getpid`) ;
pid du processus fils : renvoyé au père.

```
void exit(int status);
```

Terminaison d'un processus :
`status` : valeur récupérée par le processus père (disponible dans le shell avec `$?`) ;
Utilisation des constantes :
`EXIT_SUCCESS` $= 0$;
`EXIT_FAILURE` $\neq 0$.

```
int atexit(void (* function(void)));
```

Insérer la fonction `function` en tête de la pile des fonctions invoquées à la fin du programme.

Terminaison d'un processus (`exit` ou `return`) :
Toutes les fonctions enregistrées avec `atexit()` sont appelées (dépilées) ;
Fermeture des flux E/S (buffers vidés) ;
Appel à `_exit()` – appel système :
Ferme les descripteurs de fichiers ouverts ;
Processus père reçoit le signal `SIGCHLD`.
Le processus devient *zombie*.

Synchronisation simple

```
pid_t wait(int * status);
```

Synchronisation père/fils :
Le processus a au moins **un** fils *zombie* :
Retourne : le pid de l'un de ses fils *zombie* ;
`status` : informations sur le fils *zombie*.
Le processus a des fils qui ne sont pas à l'état *zombie*.
Le processus est bloqué jusqu'à ce que :
L'un de ses fils devienne *zombie* ;
Il reçoive un signal.
Le processus ne possède pas de fils : retourne -1 et `errno = ECHILD`
Rem. Pour attendre un fils en particulier : `waitpid()`
Macros pour la valeur de retour `status` :
`WIFEXITED` : non `NULL` si terminé normalement ;
`WEXITEDSTATUS` : code de retour si terminé normalement ;
`WIFSIGNALED` : non `NULL` si terminé à cause d'un signal ;
`WTERMSIG` : num. signal ayant terminé le processus ;
`WIFSTOPPED` : non `NULL` si processus fils stoppé ;
`WSTOPSIG` : num. signal ayant stoppé le processus.

```
pid_t waitpid(pid_t pid, int * status, int opt)
```

Tester la terminaison d'un processus fils d'identité `pid` ou du groupe |`pid`| :
Valeur de retour :
-1 : erreur ;
0 : processus non terminé (mode non bloquant) ;
`pid` du processus terminé.
Paramètre `pid` :
> 0 : processus fils ;
0 : processus fils qcq (même groupe appelant) ;
-1 : processus fils qcq ;
< -1 : processus fils qcq du groupe |`pid`|.
Paramètre `opt` :
`WNOHANG` : appel non bloquant ;
`WUNTRACED` : retourne si le processus est stoppé.

Recouvrement de code (`exec`)

Même pid, même processus, code différent.
Forme sous laquelle les arguments `argv` sont transmis :
`v` : `argv` sous forme de tableau (`v` - vector) ;
`l` : `argv` sous forme de liste.
Rem. `NULL` à la fin.
Transmission de paramètres :
`p` : exécutable recherché dans le `PATH` ;
`e` : prend en paramètre un nouvel environnement.
Erreurs (`errno`) :
`EACCES` : pas de permission d'accès au fichier ;
`ENOENT` : fichier n'a pas été trouvé.

```
int execl (const char *path, const char *arg, ...)
int execlp(const char *file, const char *arg, ...)
int execle(const char *path, const char *arg, ...,
    char * const envp[])

int execv (const char *path, char * const argv[])
int execvp(const char *file, char * const argv[])
int execve(const char *path, char * const argv[],
    char * const envp[])
```

Autres fonctions

```
int system(const char *command);
```

Invoque le shell en lui transmettant la fonction passée en paramètre :
`fork()` et le fils lance la commande ;
Le père attend le fils.

```
pid_t vfork(void);
```

Idem `fork`, mais le segment de données n'est pas dupliqué. Le processus fils travaille sur les données de son père ⇒ Bloquer le père en attente du fils.

```
int setjmp(jmp_buf env);
```

Permet de sauvegarder l'état du programme dans `env` du type `jmp_buf` :
Premier appel : 0 ;
Appel issu d'un `longjmp` : valeur de `lonjmp`.

```
void longjmp(jmp_buf env, int val);
```

Remet le programme dans l'état sauvegardé par le dernier appel à `setjmp` par rapport à la variable `env`.

Signaux

Définition. Un signal est une information transmise à un programme durant son exécution :

Signal → `int` ($\neq 0$, $<$ `NSIG`) ;
Communication OS ↔ Processus :
En cas d'erreur (violation mémoire, erreur d'E/S) ;
Par l'utilisateur : ctrl-C, ctrl-Z, . . . ;
Déconnexion de la ligne/terminal, etc.
Possibilité d'envoi d'un signal entre processus ;
Traitement par défaut en général : quitter le processus.

Liste des signaux `kill -l`

Envoyer un signal `kill -KILL <pid>; kill -INT <pid>`

Définition (Signal pendant). Signal envoyé à un processus mais qui n'a pas encore été pris en compte.

Définition (Signal masqué/bloqué). Délivrance du signal ajournée.

Définition (Délivrance). Le processus prend en compte le signal et réalise l'action qui lui est associée (état noyau → état utilisateur) :

Terminaison du processus ;
Terminaison du processus avec production d'un fichier de nom core ;
Signal ignoré ;
Suspension du processus (*stopped* ou *suspended*) ;
Continuation du processus.

Nouvel *handler* (tous signaux sauf `SIGKILL` et `SIGSTOP`) :

`SIG_IGN` : ignorer le signal ;
Fonction utilisateur `void (*sa_handler)(int);`
`SIG_DFL` : comportement par défaut.

Appels système interruptibles (`wait`, `sigsuspend`. . .) :

L'arrivée d'un signal à un processus endormi à un niveau de priorité interruptible le réveille (bloqué → prêt) ; signal délivré lors de l'élection du processus.
L'arrivée d'un signal sur un appel système interruptible provoque l'arrêt de l'appel (appel système retourne -1 et `errno = EINTR` – reprise automatique : *flag* `SA_RESTART` de `struct sigaction`)

```
int kill(pid_t pid, int sig);
```

Par défaut la réception d'un signal provoque la terminaison de `pid`. Paramètre `pid` :

`pid` : processus visé :
0 : tous les processus dans le même groupe ;
-1 : tous les processus du système (non POSIX) ;
$<$ -1 : tous les processus du groupe `|pid|`.
`sig` : signal envoyé ;
Valeur de retour : 1 si succès, -1 sinon (`errno`).

Signaux masqués/bloqués :

Un processus peut installer un masque de signaux à l'exclusion de `SIGKILL` et `SIGSTOP` ;
Délivrance différée ;
Signal pendant, masqué ⇒ non délivré ;
Bloqué pendant exécution du *handler* associé (défaut) ;
Processus fils hérite du masque des signaux et du *handler* (pas signaux pendants).

Manipulation du type `sigset_t`

```
int sigemptyset(sigset_t * set);
```

Vide l'ensemble de signaux fourni par `set`, tous les signaux étant exclus de cet ensemble.

```
int sigfillset(sigset_t *set);
```

Remplit totalement l'ensemble de signaux `set` en incluant tous les signaux.

```
int sigaddset(sigset_t *set, int signum);
```

Ajoute le signal `signum` à l'ensemble `set`.

```
int sigdelset(sigset_t *set, int signum);
```

Supprime le signal `signum` de l'ensemble `set`.

Valeurs de retour : 0 si succès, -1 si erreur (`errno = EINVAL` si signal non valide).

```
int sigismember(const sigset_t *set, int signum);
```

Teste si le signal `signum` est membre de l'ensemble `set`.
Valeur de retour : renvoie 1 si `signum` est dans `set`, 0 sinon, et -1 en cas d'erreur (`errno = EINVAL` si signal non valide).

```
int sigpending(sigset_t *set);
```

Liste les signaux pendants bloqués dans `set`.
Valeur de retour : 0 si succès, -1 si erreur (`errno = EFAULT` si `set` pointe sur mémoire non valide).

```
int sigprocmask(int how, const sigset_t *set,
sigset_t *oldset);
```

Modifier le masque des signaux :
Paramètre `how` :
`SIG_BLOCK` : l'ensemble des signaux bloqués est l'union de l'ensemble actuel et de `set` ;
`SIG_UNBLOCK` : signaux de l'ensemble `set` sont supprimés de la liste des signaux bloqués ;
`SIG_SETMASK` : signaux bloqués sont ceux de `set`.
`set` : masque de signaux ;
`old` : valeur du masque antérieur, si non `NULL` ;
Valeur de retour : 0 si succès, -1 sinon (`errno`).

Changement traitement par défaut d'un signal

```
struct sigaction {
    void (*sa_handler) (int); /* fonction */
    sigset_t sa_mask; /* masque des signaux */
    int sa_flags; /* options */
}
```

Description des champs de `struct sigaction` :
`sa_handler` : fonction exécutée à la réception du signal :
`void f(int sig);` : fonction à exécuter ;
`SIG_DFL` : traitement par défaut ;
`SIG_IGN` : ignorer le signal.
`sa_mask` : liste de signaux ajoutés à la liste de signaux qui se trouvent bloqués lors de l'exécution du *handler* (le signal en cours de délivrance est automatiquement masqué par le *handler*).
`sa_flags` : quelques options :
`SA_NOCLDSTOP` : le signal `SIGCHLD` n'est pas envoyé à un processus lorsqu'un de ses fils est stoppé ;
`SA_RESETHAND` : rétablir l'action à son comportement par défaut une fois que le gestionnaire a été appelé ;
`SA_RESTART` : un appel système interrompu par un signal capté est repris au lieu de renvoyer -1 ;
`SA_NOCLDWAIT` : Si le signal est `SIGCHLD`, le processus fils qui se termine ne devient pas ZOMBIE.

```
int sigaction(int sig, const struct sigaction *
act, struct sigaction * oact);
```

Installation d'un *handler* `act` pour le signal `sig` :
`act` pointe vers une structure du type `struct sigaction`. La délivrance du signal `sig`, entraînera l'exécution de la fonction pointée par `act->sa_handler` , si non `NULL`. ;
`oact` : si non `NULL`, pointe vers l'ancienne structure `struct sigaction` ;
Valeur de retour : 0 si succès, -1 si échec (`errno = EFAULT | EINVAL`).

Attente d'un signal

Processus passe à l'état *stoppé*. Il est réveillé par l'arrivée d'un signal non masqué.

```
int pause(void);
```

Ne permet ni d'attendre l'arrivée d'un signal de type donné, ni de savoir quel signal a réveillé le processus.
Valeur de retour : toujours -1 (`errno = EINTR` – interruption)

```
int sigsuspend(const sigset_t *sigmask);
```

Installation du masque des signaux pointé par `sigmask`.
Masque d'origine réinstallé au retour de la fonction.
Valeur de retour : toujours -1 (`errno = EINTR` – interruption)

Signal `SIGCHLD`

Signal envoyé automatiquement à un processus lorsque l'un de ses fils se termine ou lorsque l'un de ses fils passe à l'état stoppé (réception du signal `SIGSTOP` ou `SIGTSTP`) :

Comportement par défaut : ignorer le signal ;
Prise en compte de la terminaison de son fils ;
Elimination du fils zombie.

`wait(), waitpid(), wait3()`

Signaux `SIGSTOP`/`SIGTSTP`, `SIGCONT` et `SIGCHLD`

Processus s'arrête (état bloqué) en recevant un signal `SIGSTOP` ou `SIGTSTP`
Processus père est prévenu par le signal `SIGCHLD` de l'arrêt d'un de ses fils.

Comportement par défaut : ignorer le signal ;
Relancer le processus fils : envoyer `SIGCONT`.

Session de processus

Tout processus appartient exactement à une session (même que celle de son père)

Processus **leader** d'une session : créateur de la session ;
Leader d'une session terminé ⇒ tous les processus de la session reçoivent un signal `SIGHUP` ;
Session identifiée par pid du leader ;
Création d'une session : `pid_t setsid(void);`
Leader : terminal de contrôle en ouvrant `/dev/tty` ;
Un terminal ne peut être terminal de contrôle de plus d'une session.

Groupes de processus

Processus en avant-plan ou en arrière-plan parmi ceux attachés à un terminal. Tout processus appartient, à un instant donné, à un groupe de processus :
- Création processus : même groupe que son père ;
- Leader du groupe : processus qui a crée le groupe `pid_t getpgrp(void)` ;
- Rattachement d'un processus à groupe : `setpgid(pid_t pid, pid_t gpid)`
- Si le groupe de processus `gpid` n'existe pas, il est créé et le processus appelant en devient le leader du groupe.

Groupe de processus en avant-plan (*foreground*) :
- Un unique groupe des processus en avant-plan par session ;
- Processus peuvent accéder au terminal ;
- Processus destinataires des signaux `SIGINT`, `SIGQUIT` et `SIGTSTP`.

Groupe de processus en arrière-plan (*background*) :
- Ne peuvent pas accéder au terminal ;
- Processus ne reçoivent pas les signaux `SIGINT`, `SIGQUIT` et `SIGTSTP` issus d'un terminal ;
- On appelle *job* un processus en arrière-plan ou suspendu.

Temporisateurs

`unsigned alarm(unsigned seconds);`
`seconds` : durée exprimée en secondes. Un signal `SIGALRM` est généré au terme du temporisateur. Un seul temporisateur par processus.
Une nouvelle demande annule la courante (`alarm(0)`)
Valeur de retour : temps restant de la dernière temporisation, 0 sinon.

`int setitimer(int which, const struct itimerval * value, struct itimerval * ovalue);`
`which` : une valeur parmi :

Type	Temporisation	Signal
`ITIMER_REAL`	Tps réel	`SIGALRM`
`ITIMER_VIRTUAL`	Tps mode user	`SIGVTALRM`
`ITIMER_PROF`	Tps CPU total	`SIGPROF`

`value` : nouvelle valeur ;
`ovalue` : ancienne valeur (peut être `NULL`) ;
Valeur de retour : 0 si succès, -1 sinon (`errno = EFAULT |EINVAL`)

```
struct timeval { /* duree */
    long tv_sec; /* seconds */
    long tv_usec; /* microseconds */
};
struct itimerval { /* intervalle de rearmement */
    struct timeval it_interval; /* next value */
    struct timeval it_value; /* current value */
};
```

Timer périodique : une échéance à `it_value` puis une toutes les `it_interval`.
`it_value = 0` : annulation ;
`it_interval = 0` : pas de réarmement.

Violation de mémoire

Point de rupture (*breakpoint*) :
Plus petite adresse non utilisée de l'espace de données ;
La mémoire est découpée en blocs de taille fixe (pages).
Le signal `SIGSEG` indique toutes les violations de mémoire lorsque le processus accède en écriture à une adresse en dehors de son espace d'adressage (*Segmentation Fault*).

`void * brk(const void *addr);`
Adresse du point de rupture égale à `addr`.

`void * sbrk(int incr);`
Ajout de `incr` octets à la valeur du point de rupture ;
Valeur renvoyée : pointeur sur le nouveau point de rupture, -1 si echec (`errno = ENOMEM`).

Contrôle du point de reprise

`int sigsetjmp(sigjmp_buf env, int ind);`
Sauvegarder dans `env` la valeur d'environnement ;
Si `ind != 0`, le masque courant est sauvegardé ;
Valeur de retour – suivant le contexte :
Appel direct à `sigsetjmp` : `0` ;
Appel indirect provoqué depuis `siglongjmp` : `val`.

`int siglongjmp(sigjmp_buf env, int val);`
Restaurer un environnement sauvé avec `sigsetjmp` ;
`val` : valeur de l'appel simulé ;
Hyp. `env` a été initialisé avec `sigsetjmp`.

Fonctions non POSIX

`int raise(int sig);`
Envoie le signal `sig` donné au processus courant ;
Equivaut à `kill(getpid(), sig)` ;
Valeur de retour : 0 si succès, -1 sinon (`errno`).

`sighandler_t signal(int sig, sighandler_t hand);`
Nouvel *handler* `hand` pour le signal numéro `sig` ;
`hand` : soit une fonction, soit une constante `SIG_IGN` ou `SIG_DFL` ;
Valeur de retour :! valeur précédente du handler, ou `SIG_ERR` en cas d'erreur.

`int siginterrupt(int sig, int flag);`
Modifie le comportement d'un appel système interrompu par le flag `sig` :
`flag = 0` : l'appel système interrompu par sig sera relancé automatiquement ;
Equivaut à positionner le drapeau `sa_flags` de la structure `sigaction` à la valeur `SA_RESTART`, si `flag == 0`, sinon le positionner à `SA_RESTART`.
Utiliser plutôt `sigaction` avec le flag `SA_RESTART` au lieu de `siginterrupt`

Signaux courants

Nom	Evènement	Comportement
	Terminaisons	
`SIGINT`	ctrl-C	Terminaison
`SIGQUIT`	ctrl-\	Terminaison + core
`SIGKILL`	Tuer un processus	Terminaison
`SIGTERM`	Signal terminaison	Terminaison
`SIGCHILD`	Terminaison/arrêt processus fils	Ignoré
`SIGABRT`	Terminaison anormale	Terminaison + core
`SIGHUP`	Déconnexion terminal	Terminaison
	Suspension / Reprise	
`SIGSTOP`	Suspension exécution	Suspension
`SIGSTP`	Suspension exécution (ctrl-Z)	Suspension
`SIGCONT`	Reprise du processus arrêté	Reprise
	Fautes	
`SIGFPE`	Erreur arithmétique	Terminaison + core
`SIGBUS`	Erreur sur le bus	Terminaison + core
`SIGILL`	Instruction illégale	Terminaison + core
`SIGSEGV`	Violation protection mémoire	Terminaison + core
`SIGPIPE`	Erreur écriture sur un tube sans lecteur	Terminaison
	Autres	
`SIGALRM`	Fin temporisation	Terminaison
`SIGUSR1`	Utilisateur	Terminaison
`SIGUSR2`	Utilisateur	Terminaison
`SIGTRAP`	Trace/breakpoint trap	Terminaison + core
`SIGIO`	E/S asynchrone	Terminaison

Entrées / sorties

Fichiers d'entêtes

`unistd.h`, `sys/stat.h`, `sys/types.h`, `fcntl.h` :
Types de base universels ;
Constantes symboliques ;
Structures et types utilisés dans le noyau ;
Prototypes des fonctions.

Constantes de configuration

`LINK_MAX` : nombre max. de liens physiques par i-node ;
`PATH_MAX` : longueur max. pour le chemin d'un fichier ;
`NAME_MAX` : longueur max. des noms de liens ;
`OPEN_MAX` : nombre max. d'ouvertures de fichiers simultanées par processus.
Noms de constantes préfixés par `_POSIX_<cste>`

Erreurs associées aux I/O

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

`EACCESS` : accès interdit ;
`EBADF` : descripteur de fichier non valide ;
`EEXIST` : fichier déjà existant ;
`EIO` : erreur E/S ;
`EISDIR` : opération impossible sur un répertoire ;
`EISDIR` : opération impossible sur un répertoire ;
`EMFILE` : > `OPEN_MAX` fichiers ouverts pour le processus ;
`EMLINK` : > `LINK_MAX` liens physiques sur un fichier ;
`ENAMETOOLONG` : nom fichier trop long (> `PATH_MAX`) ;
`ENOENT` : fichier ou répertoire inexistant ;
`EPERM` : droits d'accès incompatible avec l'opération.

Consultation de l'i-node – `struct stat`

```
struct stat {
  dev_t     st_dev;     //device file resides on
  ino_t     st_ino;     //the file serial number
  mode_t    st_mode;    //file mode
  nlink_t   st_nlink;   //hard links to the file
  uid_t     st_uid;     //user ID of owner
  gid_t     st_gid;     //group ID of owner
  dev_t     st_rdev;    //the device identifier
  off_t     st_size;    //file total size (bytes)
  blksize_t st_blksize; //blocksize (file system I/O)
  blkcnt_t  st_blocks;  //blocks allocated
  time_t    st_atime;   //file last access time
  time_t    st_mtime;   //file last modify time
  time_t    st_ctime;   //file last status change time
};
```

Type de fichier – Champ `st_mode` de `struct stat`

Type : masque `S_IFMT` (POSIX : macros)
Fichiers réguliers (`S_IFREG`) – macro : `S_ISREG (t)` ;
Répertoires (`S_IFDIR`) – macro : `S_ISDIR (t)` ;
Tubes FIFO (`S_FIFO`) – macro : `S_ISFIFO (t)` ;
Fichiers spéciaux :
Périphériques bloc (`S_IFBLK`) – macro : `S_ISBLK (t)` ;
Caractère (`S_IFCHR`) – macro : `S_ISCHR (t)`.
Liens symboliques (`S_IFLNK`) – macro : `S_ISLNK (t)` ;
Sockets (`S_IFDOOR`) – macro : `S_ISSOCK (t)`.
On a : `S_ISCHR (mode)` ⇔ `((mode & S_IFMT)== S_IFCHR)`

Droits de fichiers – Valeurs du type `mode_t`

	User	Group	Others
Read	`S_IRUSR`	`S_IRGRP`	`S_IROTH`
Write	`S_IWUSR`	`S_IWGRP`	`S_IWOTH`
eXecution	`S_IXUSR`	`S_IXGRP`	`S_IXOTH`
RWX	`S_IRWXU`	`S_IRWXG`	`S_IRWXO`

`S_ISUID`	Modification du `UID` à l'exécution
`S_ISGID`	Modification du `GID` à l'exécution
`S_ISVTX`	Positionner *sticky bit* conserver code du programme en mémoire après exécution

Liste les fichiers et leurs droits du répertoire courant `ls -l`
`rwxr-xr--` ⇔ `S_IRWXU| S_IRGRP | S_IXGRP| S_IROTH`

Consultation de l'i-node

```
int stat(const char *pathname, struct stat *buf);
```
Obtention des caractéristiques d'un fichier :
`pathname` : chemin du fichier ;
`buf` : récupération du résultat ;
Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int fstat(int fildes, struct stat *buf);
```
Idem `stat()` ; `fildes` : descripteur de fichier

```
int access(const char* pathname, int mode);
```
Test des droits d'accès d'un processus sur un fichier :
`pathname` : chemin du fichier à tester
`mode` : `R_OK` (lecture), `W_OK` (écriture), `X_OK` (exécution), `F_OK` (existence) – possibilité d'utiliser `OR` (`|`) ;
Valeur de retour : si `pathname` ne peut pas être trouvé ou si aucun, ne serait ce qu'un des accès demandés ne peut être accordé : -1 (`errno`), 0 sinon.

Manipulation de liens physiques

```
int link(const char *origine, const char *cible);
```
Permet de créer un nouveau lien physique ; contraintes :
`origine` ne peut pas être un répertoire ;
`cible` ne doit pas exister.

```
int unlink(const char *ref);
```
Supprime `ref` du système de fichiers et décrémente le nombre de liens vers le fichier référencé par `ref`.
Si `ref` était le dernier lien vers le fichier, et aucun processus ne l'a ouvert, alors le fichier est supprimé ;
Si `ref` était le dernier lien vers le fichier, mais il existe des processus qui ont ouvert le fichier, alors ce dernier sera supprimé lorsque le dernier descripteur de fichier qui lui est associé sera fermé ;
Si `ref` fait référence à un lien symbolique, alors le lien est supprimé.

```
int rename(const char *old, const char *new);
```
Changement de lien physique.
`new` ne doit pas exister ;
Impossible de renommer `.` et `..`
Valeur de retour : 0 si succès, -1 sinon (`errno`).

Changement d'attributs d'un i-node

```
int chmod(const char *pathname, mode_t mode);
```
Attribution des droits d'accès `mode` au fichier de nom `pathname`.

```
int fchmod(int fildes, mode_t mode);
```
Attribution des droits d'accès `mode` au fichier associé au descripteur `fildes`.

```
int chown(const char *pathname, uid_t uid, gid_t gid);
```
Modification du propriétaire `uid` et du groupe `gid` du fichier `pathname`.

```
int fchown(int fildes, uid_t uid, gid_t gid);
```
Modification du propriétaire `uid` et du groupe `gid` du fichier associé au descripteur `fildes`.
Valeur de retour : 0 si succès, -1 sinon (`errno`).

Primitives de base

```
int open (const char *pathname, int flags);
```
Ouverture d'un fichier.
`pathname` : chemin du fichier à ouvrir ;
`flags` : modes d'accès au fichier :

`O_RDONLY`	ouverture en lecture
`O_WRONLY`	ouverture en écriture
`O_RDWR`	ouverture en lecture-écriture
`O_CREAT`	création d'un fichier s'il n'existe pas
`O_TRUNC`	vider le fichier s'il existe
`O_APPEND`	écriture en fin de fichier
`O_SYNC`	écriture immédiate sur disque
`O_NONBLOCK`	ouverture non bloquante

```
int open (const char *pathname, int flags, mode_t mode);
```
Idem précédent, utile s'il y a création d'un nouveau fichier (`(O_WRONLY|O_RDWR)| O_CREAT | O_TRUNC`).
`mode` : de type `mode_t` – définit les droits d'accès sur le fichier ouvert est créé (`O_CREAT`, `O_TRUNC`).
Valeur de retour : descripteur si succès, -1 sinon (`errno`).

```
int close(int fildes);
```
Ferme le descripteur correspondant à un fichier en désallouant son entrée de la table des descripteurs du processus. Si nécessaire, maj table des fichiers et table des i-nodes.

```
int creat(const char *pathname, mode_t mode);
```
Création d'un fichier. Correspond à `open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);`

```
ssize_t read(int fd, void *buf, size_t count);
```
Demande de lecture d'au plus `count` octets du fichier correspondant à `fd` écrits dans `buf`.
La lecture se fait à partir de la position courante (`offset` de la Table des Fichiers Ouverts ; maj après la lecture) ;
Valeur de retour : nombre d'octets lus ou -1 en cas d'erreur (`errno`).

```
ssize_t readv(int fd, const struct iovec *vec,
int nb);
```
Lit `nb` blocs depuis le descripteur de fichier `fd` dans les multiples tampons décrits par `vec`.

```
struct iovec {
    void *iov_base; /* Adresse de debut */
```

```
    size_t iov_len; /* Nombre d'octets */
};
ssize_t pread(int fd, void *buf, size_t count,
off_t offset);
```
Lecture à partir de la position `offset` ; `offset` de la table des fichiers ouverts n'est pas modifié.
```
ssize_t write(int fd, void *buf, size_t count);
```
Demande d'écriture de `count` caractères contenus à partir de l'adresse `buf` dans le fichier correspondant à `fd`.

Ecriture : à partir de la fin du fichier (`O_APPEND`) ou de la position courante.

Modifie le champ `offset` de la Table des Fichiers Ouverts.

Valeur de retour : nombre de caractères écrits ou -1 en cas d'erreur.
```
ssize_t writev(int fd, const struct iovec *vec,
int nb);
```
Ecrit au plus `nb` blocs décrits par `vec` dans le fichier associé au descripteur `fd`.
```
ssize_t pwrite(int fd, void *buf, size_t count,
off_t offset);
```
Lit au maximum `count` octets dans la zone mémoire pointée par `buf`, et les écrit à la position `offset` (mesurée depuis le début du fichier) dans le descripteur `fd`.
```
off_t lseek(int fd, off_t offset, int whence);
```
Modifier la position courante (`offset`) de l'entrée de la Table de Fichiers Ouverts associée à `fd`.

La position courante prend comme nouvelle valeur : `offset + whence` ;

`whence` : position initiale du curseur :

`SEEK_SET` : 0 (début du fichier) ;

`SEEK_CUR` : Position courante ;

`SEEK_END` : Taille du fichier.

Ex. Décalage 2 `char` : `offset = 2*sizeof(char)` ;

Valeur de retour : nouvelle position courante ou -1 en cas d'erreur (`errno`).

Duplication de descripteur

```
int dup(int fd);
```
Recherche le plus petit descripteur disponible dans la table des descripteurs du processus et en fait un synonyme de `fd`.
```
int dup2(int oldfd, int newfd);
```
Transforme `newfd` en une copie de `oldfd`, fermant auparavant `newfd` si besoin est.

Valeur de retour : nouveau descripteur, ou -1 si échec (`errno`).

Liens symboliques

Def. Fichier (réel) qui porte sur un autre fichier.
```
int symlink(const char *cible, const char *nom);
```
Créer un lien symbolique avec le `nom` indiqué, et qui pointe sur `cible`.
```
int lstat(const char *path, struct stat *buf);
```
Idem `stat`, sauf que si `path` est un lien symbolique, il donne l'état du lien lui-même plutôt que celui du fichier visé.
```
ssize_t readlink(const char *path, char *buf,
size_t bufsiz);
```
Place le contenu d'un lien symbolique `path` dans le tampon `buf`, lequel doit contenir au moins `bufsiz` octets
```
int lchmod(const char *path, mode_t mode);
```
Change le mode d'accès du lien symbolique `path`
```
int lchown(const char *path, uid_t u, gid_t g);
```
Modifier le propriétaire et le groupe du lien symbolique `path`

Valeur de retour : 0 si succès, -1 sinon (`errno`).

E/S sur répertoires

```
DIR * opendir(const char *dirname);
```
Ouvre en lecture le répertoire de référence `dirname`, lui alloue un objet du type DIR, dont l'adresse est renvoyée
```
struct dirent * readdir(DIR *pDir);
```
Lit l'entrée courante du répertoire associé à `pDir`

Place le pointeur sur l'entrée suivante ;

Valeur de retour un pointeur de type `struct dirent` ou `NULL` en cas d'erreur ou lorsque la fin de fichier est atteinte.
```
struct dirent {
    ino_t d_ino;
    char d_name [];
};
void rewinddir(DIR *pDir);
```
Repositionne le pointeur des entrées associé à `pDir` sur la première entrée dans le répertoire.
```
int closedir(DIR *pDir);
```
Ferme le répertoire associé à `pDir` ; les ressources allouées au cours de l'appel à `opendir` sont libérées.
```
int mkdir(const char *dirname, mode_t mode);
```
Création d'un répertoire vide en spécifiant les droits.
```
int rmdir(const char *dirname);
```
Supprime le répertoire `dirname`.
```
int rename(const char *old, const char *new);
```
Renomme le répertoire `old` en `new`.

Valeur de retour : 0 si succès, -1 sinon (`errno`).

Obtention / modification des attributs d'un fichier

Fonction `fcntl` dont la signature dépend de `cmd` :

```
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

Permet de modifier des attributs associés à un descripteur :

Mode d'ouverture ;

Duplication du descripteur ;

Verrouillage des zones du fichier.

Argument `cmd` :

`F_GETFD` : obtenir la valeur des attributs du descripteur ;

`F_SETFD` : modifier les attributs du descripteur ;

`F_GETFL` : obtenir la valeur des attributs du mode d'ouverture ;

`F_SETFL` : modifier le mode d'ouverture :

Argument `arg` : `O_APPEND`, `O_NONBLOCK`, `O_NDELAY`, `O_SYNC`...(cf. fonction `open()`).

`F_DUPFD` : duplication de descripteur dans le plus petit descripteur disponible (`fcntl(fd, F_DUPFD, 0)` ⇔ `dup(fd);`) ;

Verrouillage :

`F_GETLK` : récupérer le verrou courant sur la zone définie par `lock` ;

`F_SETLK` : (dé)verrouiller la zone définie par `lock` ;

`F_SETLKW` : idem, mais attente bloquante jusqu'à ce que la zone soit libérée d'autres verrous

Code de retour :

`F_DUPFD` : le nouveau descripteur, sinon -1 en cas d'erreur (`errno`) ;

`F_GETFD`, `F_GETFL` : valeur des attributs, sinon -1 en cas d'erreur (`errno`) ;

`F_SETFD`, `F_SETFL` : 0 en cas de succès, -1 en cas d'erreur (`errno`)

Verrouillage (seulement consultatif – verrouillage effectif : sémaphores). Types de verrous :

`F_WRLCK` – exclusif (`write`) : aucun autre processus ne peut verrouiller une zone avec verrou exclusif ;

`F_RDLCK` – partagé (`read`) : tout autre processus peut verrouiller une zone avec verrou partagé ;

`F_UNLCK` – déverrouiller.

```
struct flock {
    off_t l_start; /* starting offset */
    off_t l_len;  /* len = 0 means until eof */
    pid_t l_pid;  /* lock owner */
    short l_type;  /* lock type */
    short l_whence; /* type of l_start */
};
```

Bibliothèque E/S standard C

Constitue une couche au-dessus des appels système correspondant aux primitives de base d'E/S POSIX.

But : travailler dans l'espace d'adressage du processus

E/S dans des tampons appartenant à cet espace d'adressage ;

Objet de type `FILE`, obtenu lors de l'appel à la fonction `fopen` :

Permet de gérer le tampon associé au fichier ;

Possède le numéro du descripteur du fichier :

`STDIN_FILENO = stdin` (0)

`STDOUT_FILENO = stdout` (1)

`STDERR_FILENO = stderr` (2)

`fflush` force l'écriture du contenu du tampon dans les caches système.

Fichier `stdio.h`

Constantes :

`NULL` : adresse invalide ;

`EOF` : reconnaissance de fin de fichier ;

`FOPEN_MAX` : nombre max de fichiers manipulables simultanément ;

`BUFSIZ` : taille par défaut des tampons.

Types :

`FILE` : type dédié à la manipulation d'un fichier ; gère le tampon d'un fichier ouvert.

`fpos_t` : position dans un fichier ;

`size_t` : longueur du fichier.

Objets prédéfinis de type `FILE *` :

`stdin` : objet d'entrée standard ;

`stdout` : objet de sortie standard ;

`stderr` : objet de sortie-erreur standard.

Fonctions de base

```
FILE * fopen(const char *filename, const char *
mode);
```

Ouverture du fichier `filename`.

`filename` : chemin d'accès au fichier ;

`mode` : mode d'ouverture :

`r` : lecture seulement ;

`r+` : lecture / écriture sans création ou troncature ;

`w` : écriture avec création ou troncature du fichier ;

`w+` : lecture / écriture avec création ou troncature ;

`a` : écriture en fin de fichier ; création si nécessaire ;

`a+` : lecture / écriture en fin de fichier ; création si nécessaire.

Valeur de retour : pointeur vers un objet `FILE` associé au fichier, `NULL` si échec ; tampon pour les lectures/écritures, et d'une position courante.

```
FILE * freopen(const char *filename, const char *
mode, FILE *stream);
```

Nouvelle ouverture d'un fichier ; associe à un objet déjà alloué une nouvelle ouverture ; redirection d'E/S.

Ex. Redirection sortie standard :

```
freopen ("fic1", "w", stdout);
```

```
int fileno(FILE *stream);
```

Obtention du descripteur associé à l'objet `FILE`.

```
FILE * fdopen(const int fd, const char *mode);
```

Obtention d'un objet du type `FILE` à partir d'un descripteur `fd` (*Rem.* Le mode d'ouverture doit être compatible avec celui du descripteur).

```
int feof(FILE *stream);
```

Test de fin de fichier :

Associé aux opérations de lecture ;

Valeur de retour : $\neq 0$ si la fin de fichier associée à `stream` a été détectée, 0 sinon.

```
int ferror(FILE *stream);
```

Test d'erreur.

Valeur de retour :$\neq 0$ si une erreur associée à `stream` a été détectée.

```
int fclose(FILE *stream);
```

Ferme le fichier associé à `stream` :

Transfert de données du tampon associé ;

Libération de l'objet `stream` ;

Valeur de retour : 0 si succès, `EOF` en cas d'erreur.

Gestion du tampon

A chaque ouverture de fichier, un tampon de taille `BUFSIZ` est automatiquement alloué.

```
int setvbuf(FILE *stream, char *buf, int mode,
size_t size);
```

Association d'un nouveau tampon :

Permet d'associer un nouveau tampon de taille `size` à `stream` ;

`mode` : critère de vidage :

`_IOFBF` : le tampon est plein ;

`_IOLBF` : le tampon contient une ligne ou est plein ;

`_IONBF` : systématiquement.

```
int fflush(FILE *stream);
```

Vidage du tampon : si `stream` vaut `NULL`, tous les fichiers ouverts en écriture sont vidés.

Lecture

```
int fgetc(FILE* stream);
```

Lire un caractère.

Valeur de retour : le caractère suivant du fichier sous forme entière, `EOF` en cas d'erreur ou fin de fichier ;

`int getchar(void)` $\Leftrightarrow$ `fgetc(stdin);`.

```
char * fgets(char *s, int n, FILE *stream);
```

Lire une chaîne de caractères.

Lit au plus `n-1` éléments de type `char` à partir de la position courante dans `stream` ;

Arrête la lecture si fin de ligne (`'n'`, incluse dans la chaîne lue) ou `EOF` détecté ;

Valeur de retour : un pointeur sur `s` si succès ou `NULL` en cas d'erreur ou `EOF` (test avec `feof()` ou `ferror()`).

```
size_t fread(void *p, size_t size, size_t nitems,
 FILE *stream);
```

Lecture d'un tableau d'objets.

Lit au plus `nitems` objets à partir de la position courante dans `stream` ;

Tableau des objets lus sauvegardé à l'adresse `p` ;

Chaque objet est de taille `size` ;

Valeur de retour : nombre d'objets lus, 0 en cas d'erreur ou `EOF` (test `feof` ou `ferror`).

```
int fscanf(FILE *stream, const char *format, ..);
```

Lecture formatée.

Lit à partir de la position courante dans le fichier pointé par `stream` ;

`format` : procédures de conversion à appliquer aux suites d'éléments de type `char` lues.

`scanf()` $\Leftrightarrow$ `fscanf()` sur `stdin` ;

Valeur de retour : nombre de conversions réalisées ou `EOF` en cas d'erreur.

Ecriture

```
int fputc(int c, FILE *stream);
```

Ecriture d'un caractère.

Écrit le caractère `c` dans le fichier associé à `stream` ;

`int putchar(int)` $\Leftrightarrow$ `fputc()` sur `stdout`.

Valeur de retour : `EOF` en cas d'erreur ou 0 sinon.

```
int fputs(char *s, FILE *stream);
```

Ecrire une chaîne de caractères.

Écrit la chaîne `s` dans le fichier associé à `stream` ;

Le caractère nul (`'\0'`) de fin de chaîne n'est pas écrit.

Valeur de retour : 0 si succès, `EOF` en cas d'erreur.

```
size_t fwrite(void *p, size_t size, size_t nitems
, FILE *stream);
```

Ecriture d'un tableau d'objets.

Écrit `nitems` objets de taille `size` à partir de la position courante dans `stream` ;

Le tableau d'objets à écrire se trouve à l'adresse pointée par `p` ;

Valeur de retour : nombre d'objets écrits si succès, une valeur inférieure à `nitems` en cas d'erreur.

```
int fprintf(FILE *stream, const char *format, ..)
```

Ecriture formatée.

Ecrit dans un fichier associé à `stream` les valeurs des arguments converties selon le format en chaînes de caractères imprimables ;

`printf()` $\Leftrightarrow$ `fprintf()` sur `stdout` ;

Valeur de retour : nombre de caractères écrits ou un nombre négatif en cas d'erreur.

Manipulation de la position courante

```
int fseek(FILE *stream, long offset, int whence);
```

Positionne le curseur associé à `stream` à la position `offset` relative à `whence`.

La position courante prend comme nouvelle valeur : `offset + whence` ;

`whence` : idem `lseek()` ;

Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
void rewind(FILE *stream);
```

Ramène l'indicateur de position de flux pointé par `stream` au début du fichier.

`rewind(stream)` $\Leftrightarrow$ `fseek(stream, 0, SEEK_SET);`

```
long ftell(FILE *stream);
```

Retourne la position courante associée à `stream`, -1 en cas d'erreur.

Tubes (*pipes*) anonymes et nommés

Déf. Mécanisme de communications du système de fichiers :
I-node associé ;
Type de fichier : `S_IFIFO` ;
Accès au travers des primitives `read` et `write`.

Les tubes sont unidirectionnels : une extrémité est accessible en **lecture** et l'autre l'est en **écriture**.
Rem. Dans le cas des tubes anonymes, si l'une ou l'autre extrémité devient inaccessible, cela est irréversible.

Mode FIFO : première information écrite sera la première à être consommée en lecture.

Communication d'un flot continu de caractères (*stream*) : possibilité de réaliser des opérations de lecture dans un tube sans relation avec les opérations d'écriture.
Rem. Opération de lecture est destructive : une information lue est extraite du tube.

Capacité limitée : tube plein (taille : `PIPE_BUF`) ; écriture éventuellement bloquante.

Possibilité de plusieurs lecteurs et écrivains :
Nombre de lecteurs : l'absence de lecteur interdit toute écriture sur le tube ; signal `SIGPIPE` ;
Nombre d'écrivains : L'absence d'écrivain détermine le comportement du `read` : lorsque le tube est vide, la notion de fin de fichier est considérée.

Tubes anonymes
Pas de nom : impossible pour un processus d'ouvrir un pipe anonyme en utilisant `open` ;
Acquisition d'un tube :
Création : primitive `pipe` ;
Héritage : `fork`, `dup` ;
Communication père/fils ;
Un processus qui a perdu un accès à un tube n'a plus aucun moyen d'acquérir de nouveau un tel accès.

```
int pipe(int filedes[2]);
```

Création d'un tube anonyme et alloue une paire de descripteurs de fichier :
`filedes[0]` : descripteur de lecture ;
`filedes[1]` : descripteur d'écriture ;
Valeur de retour : 0 si succès, -1 si erreur (`errno`) :
`EMFILE` : table descripteurs de processus pleine ;
`ENFILE` : table de fichiers ouverts par le système pleine.

Opérations autorisées
`read`, `write` : lecture / écriture (bloquantes par défaut) ;
`close` : fermer les descripteurs qui ne sont pas utilisés ;
`dup`, `dup2` : duplication de descripteurs, redirection ;
`fstat`, `fcntl` : accès / modification des caractéristiques ;
`fdopen` : obtenir un pointeur sur un objet du type `FILE`.

Opérations non autorisées : `open`, `stat`, `access`, `link`, `chmod`, `chown`, `rename`.

```
int read(int filedes[0], void * buf, size_t
TAILLE_BUF);
```

Lecture dans un tube d'au plus `TAILLE_BUF` caractères :
Si le tube n'est pas vide et contient `taille` caractères : lire dans `buf min(taille, TAILE_BUF)` caractères extraits du tube.
Si le tube est vide :
Si le nombre d'écrivains est nul : fin du fichier ; `read` renvoie 0.
S'il existe au moins un écrivain :
Si la lecture est bloquante (défaut) : le processus est mis en attente jusqu'à ce que le tube ne soit plus vide ou qu'il n'y ait plus d'écrivains ;
Si la lecture n'est pas bloquante : retour immédiat et `read` renvoie -1 (`errno = EAGAIN`).

```
int write(int filedes[1], void * buf, size_t
TAILLE_BUF);
```

Ecriture de `TAILLE_BUF` caractères dans un tube :
Si le nombre de lecteurs dans le tube est nul : signal `SIGPIPE` est délivré à l'écrivain (terminaison du processus par défaut) ; si `SIGPIPE` capté, `write` renvoie -1 ()`errno = EPIPE`).
Si le nombre de lecteurs est non nul :
Si l'écriture est bloquante (défaut) : le retour du `write` n'a lieu que lorsque `TAILLE_BUF` caractères ont été écrits.
Si l'écriture n'est pas bloquante :
Si `TAILLE_BUF` $\leqslant$ `PIPE_BUF` :
S'il y a au moins `TAILLE_BUF` emplacements libres dans le tube : écriture atomique réussie ;
Sinon : `write` renvoie -1 (`errno = EAGAIN`)
Si `TAILLE_BUF` $>$ `PIPE_BUF` : `write` renvoie un nombre inférieur à `TAILLE_BUF`.

Rem. Ecriture atomique si `TAILLE_BUF < PIPE_BUF`.
Fonction `fcntl` permet de rendre les lectures / écritures non bloquantes :

```
int tube[2] , attributs;
pipe(tube);
/* ecriture non bloquante */
attributs = fcntl(tube[1], F_GETFL);
attributs |= O_NONBLOCK;
fcntl(tube[1], F_SETFL, attributs);
```

Rediriger les entrées-sorties standard d'un processus sur un tube :

```
int tube[2], attributs;
pipe (tube);
dup2(tube[0], STDIN_FILENO);
/* Affichages produits sur le tube */
close(tube[0]);
```

Tubes nommés
Permettent à des processus sans lien de parenté de communiquer en mode flot (*stream*) :

Toutes les caractéristiques des tubes anonymes ;
Sont référencés dans le système de gestion de fichiers ;
Utilisation de la fonction `open` pour obtenir un descripteur en lecture ou écriture.

```
mkfifo [-p] [-m mode] reference
```

Création d'un tube nommé (`bash`) :

`-m mode` : droits d'accès (idem `chmod`) ;
`-p` : création automatique de tous les répertoires intermédiaires dans le chemin `reference`.

```
int mkfifo(const char * reference, mode_t mode);
```

Création d'un tube nommé :

`reference` :chemin d'accès au tube nommé ;
`mode` : droits d'accè au tube nommés.
Valeur de retour : 0 si succès ; -1 en cas d'erreur (`errno = EEXIST` si fichier déjà créé).

Une demande d'ouverture en **lecture** est bloquante s'il n'y a aucun **écrivain** sur le tube.
Une demande d'ouverture en **écriture** est bloquante s'il n'y a aucun **lecteur** sur le tube.

Ouverture non bloquante : option `O_NONBLOCK` lors de l'appel à la fonction `open` :

Ouverture en lecture :
Réussit même s'il n'y a aucun écrivain dans le tube ;
Opérations de lectures se suivant non bloquantes.
Ouverture en écriture :
Sur un tube sans lecteur, l'ouverture échoue : valeur -1 renvoyée ;
Si le tube possède des lecteurs, l'ouverture réussit et les écritures dans les tubes sont non bloquantes.

Suppression d'un nœud :

Le nombre de liens physiques est nul : fonction `unlink` ou commande `rm` ;
Le nombre de liens internes est nul : nombre de lecteurs et écrivains sont nuls.

Rem. Si nombre de liens physiques est nul, mais le nombre de lecteurs et/ou écrivains est non nul : tube nommé $\Rightarrow$ tube anonyme.

Communication inter-processus

IPC (*Inter-Process Communication*) : outils pour bâtir des communications entre processus :
Communication : segment de mémoire partagée ;
Synchronisation : sémaphores ;
Intermédiaires : files de messages.
Effets des appels système :
`fork()` : le fils hérite de tout les objets IPC de son père ;
`exit()` / `exec()` : les accès aux objets IPS sont perdus, mais les objets IPC n'ont pas été supprimés. Dans le cas de la mémoire partagée, le segment est détaché.

Files de message – `<mqueue.h>`

Principe. Liste chainée de messages conservée en mémoire et accessible par plusieurs processus.

Message. En mémoire, stocké de manière non contiguë et non séquentielle (potentiellement, n'importe ou). Pour chaque message, une priorité est ajoutée de type `long`.
Un message peut contenir n'importe quoi (toutes structures possibles, pas uniquement des chaînes de caractères).

Fonctionnement. Accès FIFO par défaut ; dès qu'il y a une priorité : ordonnancement par priorité. Limites :
Nombre de messages (`MSGMAX`) ;
Taille totale en nombre de bytes (`MSBMNB`).
Limite atteinte ⇒ écritures bloquantes par défaut.
File vide ⇒ lecture bloquante par défaut.

Avantages / inconvénients :
- Amélioration par rapport au système de tubes : organisée en messages, contrôle sur l'organisation ;
- Simplicité ;
- Reste basé en FIFO : impossible d'organiser les accès différemment (eg. pile), pas d'accès concurrent à une même donnée, une fois la valeur prise, impossible de la reprendre une nouvelle fois.
- Performances limitées : deux recopies complètes par message : expéditeur → cache système → destinataire

Accès

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t
 mode, struct mq_attr *attr);
```

Crée une nouvelle file de messages POSIX ou ouvre une file existante. La file est identifiée par `name`.
`oflags` : idem fonction `open()` ;
`attr` : spécifier des attributs pour la file. Si `attr = NULL`, alors la file est créée avec les attribut par défaut ;
Valeur de retour : retourne un descripteur de file de messages si succès. Si échec, retourne `(mqd_t)-1` (`errno`).

Le nombre de descripteurs de file de messages n'est pas limité en nombre comme les descripteurs de fichier.

```
int mq_close(mqd_t mqdes);
```

Ferme le descripteur de la file de messages `mqdes`.
Valeur de retour : 0 si succès, -1 si échec (`errno`).
Cette fonction est automatiquement appelée lors de la terminaison du processus.

```
int mq_unlink(const char *name);
```

Détruit la file de messages `name`. Le nom de la file de messages est immédiatement supprimé. La file elle-même n'est détruite qu'une fois que tous les autres processus qui avaient ouvert la file aient fermé les descripteurs référençant la file (`mq_close`).
Valeur de retour : 0 si succès, -1 si échec (`errno`).

```
mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *
attr);
```

Récupère les attributs d'une file de messages référencée par le descripteur `mqdes`.
`attr` : tampon pointant sur une `struct mq_attr` retournée par la fonction et contenant les attributs ;
Valeur de retour : 0 si succès, -1 si échec (`errno`).

```
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *
newattr, struct mq_attr *oldattr);
```

Modifie les attributs d'une file de messages référencée par le descripteur `mqdes`.
`newattr` : nouveaux attributs pour la file de messages. Le seul attribut pouvant être modifié est l'attribut `O_NONBLOCK` dans `mq_flags`. Les autres champs de `newattr` sont ignorés.
`oldattr` : si non `NULL`, le tampon sur lequel il pointe est utilisé pour renvoyer une `struct mq_attr` qui contient la même information renvoyée par `mq_getattr()`.
Valeur de retour : 0 si succès, -1 si échec (`errno`).

```
struct mq_attr {
    long mq_flags; /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg; /* Max. # of msg on queue */
    long mq_msgsize; /* Max. msg size (bytes) */
    long mq_curmsgs; /* # msg in queue */
};
```

Opérations sur une file

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
size_t msg_len, unsigned msg_prio);
```

Déposer le message pointé par `msg_ptr` dans la file de messages de descripteur `mqdes`.
`msg_ptr` : pointeur sur le tampon du message ;
`msg_len` : taille du message à envoyer ;
`msg_prio` : priorité du message, $0 \leqslant$ `msg_prio` $\leqslant$ `MQ_PRIOMAX` ($\geqslant 32$ par défaut).
Si `msg_prio` $>$ `MQ_PRIOMAX`, l'appel échoue ;
Valeur de retour : 0 si succès, -1 si échec (`errno`).
Les messages sont placés par ordre de priorité décroissant. Les nouveaux messages sont placés après les plus vieux messages de même priorité (FIFO).
Appel bloquant si la file est **pleine** et `O_NONBLOCK` non spécifié.

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
size_t msg_len, unsigned *msg_prio);
```

Supprime le plus vieux message avec la plus grande priorité de la file de message référencée par `mqdes` et le place dans le tampon pointé par `msg_ptr`.
`msg_ptr` : pointeur sur le tampon du message à recevoir ;
`msg_len` : taille du message à recevoir ($>$ `mq_msgsize` de la file) ;
`msg_prio` : si `msg_prio != NULL` tampon pour renvoyer la priorité associée au message reçu ;
Valeur de retour : 0 si succès, -1 si échec (`errno`).
Appel bloquant si la file est **vide** et `O_NONBLOCK` non spécifié.

```
mqd_t mq_notify(mqd_t mqdes, const struct
sigevent *notification);
```

Permet au processus appelant de s'enregistrer ou se désenregistrer de la délivrance d'une notification asynchrone (non bloquant) lorsqu'un nouveau message arrive sur une file de messages vide référencée par le descripteur `mqdes`.
`notification` : pointeur sur `struct sigevent` ;
Valeur de retour : 0 si succès, -1 si échec (`errno`).
Un seul processus par file peut demander à être notifié. Une notification est émise si aucun processus n'est bloqué en attente de message. A près notification, désenregistrement de la demande ⇒ notification à chaque arrivée de message : refaire un appel à chaque notification.

```
union sigval { /* Data passed with notif. */
    int sival_int; /* Integer value */
    void *sival_ptr; /* Pointer value */
};
struct sigevent {
    int sigev_notify; /* Notif. method :
        SIGEV_NONE, SIGEV_SIGNAL, SIGEV_THREAD */
    int sigev_signo; /* Notif. signal */
    union sigval sigev_value; /* Data passed with
         notif. */
    void (*sigev_notify_function) (union sigval);
         /* Function for thread notif. */
    void *sigev_notify_attributes; /* Thread
        function attributes */
};
```

Segment de mémoire partagée – `<sys/mman.h>`
Fonctions contenues dans `librt` (`-lrt` en fin de `gcc`).

Principe. Zone de mémoire attachée a un processus, mais accessible pour d'autres processus.

Zone mémoire associée au *file mapping* : établissement d'une correspondance (attachement) entre un fichier (segment mémoire) et une partie de l'adressage d'un processus réservé à cet effet.

Le descripteur de segment pointe sur la mémoire physique, mappé dans l'espace d'adressage de plusieurs processus.

Avantages / inconvénients :

- Accès totalement libre ;
- Pas de recopie mémoire (écriture directement sur le système) ;
- Accès totalement libre : la synchronisation doit être explicite par les sémaphores ou signaux ;
- Pas de gestion de l'adressage : validité d'un pointeur limitée à son espace d'adressage ⇒ Impossible de partager pointeurs entre processus.

Accès

```
int shm_open(const char *name, int oflag, mode_t mode);
```

Crée et ouvre un nouvel objet de mémoire partagé POSIX `name` de taille 0, ou ouvre un objet existant.

`oflags` : idem fonction `open` ;
`mode` : idem fonction `chmod` ;
Valeur de retour : descripteur de fichier non nul. Si échec, -1 (`errno`).

```
int shm_unlink(const char *name);
```

Supprime l'objet `name` créé par `shm_open()`. supprime le nom d'un objet de mémoire partagée, et, une fois que tous les processus ont supprimé leur projection en mémoire, libère et détruit le contenu du segment mémoire. Après un appel réussi à `shm_unlink()`, les tentatives d'appeler `shm_open()` avec le même nom échoueront (sauf si `O_CREAT` est spécifié, auquel cas un nouvel objet distinct est créé).

Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

Demande la projection en mémoire de `length` octets commençant à la position `offset` depuis un ficher (ou autre objet) référencé par le descripteur `fd`, de préférence à l'adresse pointée par `start` (généralement 0). La véritable adresse où l'objet est projeté est renvoyée par la fonction elle-même.

`start` : adresse où attacher le segment en mémoire (0 : choix du système) ;
`prot` : protection souhaitée pour la zone mémoire :
`PROT_EXEC` : exécution de code autorisée ;
`PRO_READ` : lecture autorisée ;
`PROT_WRITE` : écriture autorisée ;
`PROT_NONE` : les pages ne peuvent pas être accédées.
Ne doit pas entrer en conflit avec le mode d'ouverture du fichier (ou autre objet) ;
`flags` : type de fichier (ou autre objet) projet, les options de projection, et si les modifications faites sur la portion projetée sont privées ou doivent être partagées avec les autres références :
`MAP_SHARED` : partager la projection avec tout autre processus utilisant le fichier (ou autre objet). Ecriture dans la zone ⇔ Ecriture dans le fichier (maj avec `msync()` ou `mumap()`) ;
`MAP_PRIVATE` : projection privée (*shadow copy*). Les modifications seront visibles uniquement par le processus appelant. L'écriture dans la zone ne modifie pas le fichier ;
`MAP_FIXED` : n'utiliser que l'adresse `start` indiquée ;
`MAP_ANONYMOUS` : la projection n'est pas contenue dans un fichier. Les champs `fd` et `offset` sont ignorés (`fd = -1`).
`fd` : descripteur de fichier valide ;
`offset` : normalement multiple de la taille de page renvoyée par `getpagesize()` ;
Valeur de retour : pointeur sur la zone mémoire si succès, sinon `MAP_FAILED` (ie. `(void *)-1`) et `errno`.

```
int munmap(void *start, size_t length);
```

Détruit la projection de taille `length` dans la zone de mémoire spécifiée à l'adresse `start`. Toute référence ultérieure à cette zone mémoire déclenche une erreur d'adressage (`SIGSEGV`).

`start` : doit être un multiple de la taille de page. Toutes les pages contenant une partie de l'intervalle indiqué sont libérées ;
Valeur de retour : 0 si succès, -1 sinon (`errno`).

Projection automatiquement détruite lors de la terminaison du processus. La fermeture du descripteur de fichier ne supprime pas la projection.

Opérations sur un segment

```
int mprotect(const void *addr, size_t *len, int prot);
```

Spécifie la protection souhaitée pour la/les page(s) contant tout/une partie de l'intervalle `[addr, addr+len -1]`.

`prot` : `OU` binaire entre les valeurs `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` ;
Valeur de retour : 0 si succès, -1 sinon (`errno`).

La nouvelle protection remplace toute autre protection précédente. Si un accès interdit s produit, le programme reçoit le signal `SIGSEGV`.

```
int ftruncate(int fd, off_t length);
```

Tronque le fichier (ou autre objet) référencé par le descripteur `fd` à une longueur maximale de `length` octets. Permet d'allouer une taille à un segment de mémoire. L'objet doit être ouvert en écriture.

Si l'objet était plus long, les données supplémentaires sont perdues ;
Si l'objet était plus cours, il est étendu et la portion supplémentaire est remplie d'octets nuls.
Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int msync(const void *start, size_t length, int flags);
```

Met à jour le segment associé à la projection `start` de taille `length`.

`flags` : comprend les bits :
`MS_ASYNC` : demande une maj immédiate non bloquante ;
`MS_SYNC` : demande une maj immédiate bloquante ;
`MS_INVALIDATE` : demande la désactivation de toutes les autres projections du même fichier (ou autre objet), afin qu'elles soient remises à jour avec les nouvelles données écrites.
Les flags `MS_ASYNC` et `MS_SYNC` ne peuvent pas être utilisés conjointement.
Valeur de retour : 0 si succès, -1 sinon (`errno`).

Sémaphores – `<semaphore.h>`

Principe (Dijkstra). Mécanisme de synchronisation pour l'accès à la mémoire partagée du fait des accès concurrents à une ressource partagée. Solution au problème de l'exclusion mutuelle (cas d'un accès critique à une ressource).

Structure sémaphore :
Compteur : nombre d'accès disponibles avant blocage ;
File d'attente : processus bloqués en attente d'un accès.

Fonctionnent.
Demande d'accès : `P` – *proberen* ("Puis-je ?") :
Décrémentation du compteur ;
Si compteur < 0, alors blocage du processus et insertion dans la file.
Fin d'accès : `V` – *verhogen* ("Vas-y") :
Incrémentation du compteur ;
Si compteur $\leqslant 0$, déblocage d'un processus de la file.
Blocage, déblocage et insertion de processus dans la file sont des opérations implicites.

Interblocage. Deux processus `P` et `Q` sont bloqués en attentes : `P` attend que `Q` signale sa fin d'accès et `Q` attend que `P` signale sa fin d'accès.

Famine. Un processus est bloqué en attente d'une fin d'accès qui n'arrivera jamais.

Deux types de sémaphores :
Sémaphores **anonymes** : processus avec filiation (seulement threads sur Linux) ;
Sémaphores **nommés** : tous les processus de la machine.

Sémaphore nommé

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
mode_t mode, unsigned int value);
```

Crée un nouveau sémaphore POSIX ou en ouvre un existant identifié par `name`.
`oflag`, `mode` : idem fonction `open()` ;
`value` : valeur initiale du compteur du sémaphore ;
Valeur de retour : pointeur sur le sémaphore si succès, `SEM_FAILED` si échec (`errno`).

```
int sem_close(sem_t *sem);
```

Ferme le sémaphore nommé référencé par `sem`. Désalloue les ressources que le système avait alloué au processus appelant pour ce sémaphore.
Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int sem_unlink(const char *name);
```

Supprime un sémaphore nommé référencé par `name`. Le nom du sémaphore est immédiatement supprimé. Le sémaphore est détruit une fois que tous les autres processus qui l'avait ouvert l'ont fermé.
Valeur de retour : 0 si succès, -1 sinon (`errno`).

Sémaphore anonyme

```
int sem_init(sem_t *sem, int pshared, unsigned
int value);
```

Initialise le sémaphore anonyme situé à l'adresse pointée par `sem`.
`pshared` : indique si ce sémaphore sera partagé entre les threads d'un processus ou entre processus :
Si `pshared = 0`, le sémaphore est partagé entre les threads d'un processus et devra être situé à une adresse visible par ces derniers ;
Si `pshared != 0`, le sémaphore est partagé entre processus et devrait être situé dans une zone de mémoire partagée (`shm_open`).
`value` : valeur initiale du sémaphore ;
Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int sem_destroy(sem_t *sem);
```

Détruit le sémaphore anonyme situé à l'adresse pointée par `sem`. Seul un sémaphore initialisé avec `sem_init()` peut être détruit avec `sem_destroy()`.
Valeur de retour : 0 si succès, -1 sinon (`errno`).

Opérations sur les sémaphores

```
int sem_wait(sem_t *sem);
```

Fonction `P` de Dijkstra :
Décrémente (verrouille) le sémaphore pointé par `sem`.
Si la valeur du sémaphore > 0, la décrémentation s'effectue et la fonction revient immédiatement ;
Si le sémaphore $= 0$, l'appel bloquera jusqu'à ce que soit il devienne disponible pour effectuer la décrémentation (ie. valeur du sémaphore $\neq 0$), soit un gestionnaire de signaux interrompe l'appel ;
Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int sem_trywait(sem_t *sem);
```

Idem fonction `sem_wait()`, excepté que si la décrémentation ne peut pas être effectuée immédiatement, l'appel renvoie une erreur (`errno = EAGAIN`) plutôt que bloquer.

```
int sem_post(sem_t *sem);
```

Fonction `V` de Dijkstra :
Incrémente (déverrouille) le sémaphore pointé par `sem`. Si à la suite de cet incrément, la valeur du sémaphore $>$ 0, un autre processus ou thread bloqué dans un appel `sem_wait` sera réveillé et procédera au verrouillage du sémaphore.
Valeur de retour : 0 si succès, -1 sinon (`errno` et valeur du sémaphore non modifiée).

Inter-Process Communication

IPC : outils pour bâtir des communications entre processus :
Communication : segment de mémoire partagée.
Synchronisation : sémaphores.
Intermédiaires : files de messages.

Effets des appels système :
`fork()` : le fils hérite de tous les IPC de son père.
`exit()` / `exec()` : Les accès aux objets IPC sont perdus, mais ATTENTION, les objets IPC n'ont pas été supprimés. Dans le cas de la mémoire partagée, le segment est détaché.

Files de message – `<mqueue.h>`

Principe. Liste chainée de messages conservée en mémoire et accessible par plusieurs processus.

Message. En mémoire, stocké de manière non contiguë et non séquentielle (potentiellement, n'importe ou). Pour chaque message, une priorité est ajoutée de type `long`.
Un message peut contenir n'importe quoi (toutes structures possibles, pas uniquement des chaînes de caractères).

Fonctionnement. Accès FIFO par défaut ; dès qu'il y a une priorité : ordonnancement par priorité. Limites :
Nombre de messages (`MSGMAX`) ;
Taille totale en nombre de bytes (`MSBMNB`).
Limite atteinte ⇒ écritures bloquantes par défaut.
File vide ⇒ lecture bloquante par défaut.

Avantages / inconvénients :
- Amélioration par rapport au système de tubes : organisée en messages, contrôle sur l'organisation ;
- Simplicité ;
- Reste basé en FIFO : impossible d'organiser les accès différemment (eg. pile), pas d'accès concurrent à une même donnée, une fois la valeur prise, impossible de la reprendre une nouvelle fois.
- Performances limitées : deux recopies complètes par message : expéditeur → cache système → destinataire

Segment de mémoire partagée – `<sys/mman.h>`

Principe. Zone de mémoire attachée à un processus, mais accessible pour d'autres processus.

Zone mémoire associée au *file mapping* : établissement d'une correspondance (attachement) entre un fichier (segment mémoire) et une partie de l'adressage d'un processus réservé à cet effet.
Le descripteur de segment pointe sur la mémoire physique, mappé dans l'espace d'adressage de plusieurs processus.

Avantages / inconvénients :
- Accès totalement libre ;
- Pas de recopie mémoire (écriture directement sur le système) ;
- Accès totalement libre : la synchronisation doit être explicite par les sémaphores ou signaux ;
- Pas de gestion de l'adressage : validité d'un pointeur limitée à son espace d'adressage ⇒ Impossible de partager pointeurs entre processus.

Sémaphores – `<semaphore.h>`

Principe (Dijkstra). Mécanisme de synchronisation pour l'accès à la mémoire partagée du fait des accès concurrents à une ressource partagée. Solution au problème de l'exclusion mutuelle (cas d'un accès critique à une ressource).

Structure sémaphore :
Compteur : nombre d'accès disponibles avant blocage ;
File d'attente : processus bloqués en attente d'un accès.

Fonctionnent.
Demande d'accès : `P` – *proberen* (“Puis-je ?”) :
Décrémentation du compteur ;
Si compteur < 0, alors blocage du processus et insertion dans la file.
Fin d'accès : `V` – *verhogen* (“Vas-y”) :
Incrémentation du compteur ;
Si compteur $\leqslant 0$, déblocage d'un processus de la file.
Blocage, déblocage et insertion de processus dans la file sont des opérations implicites.

Interblocage. Deux processus `P` et `Q` sont bloqués en attentes : `P` attend que `Q` signale sa fin d'accès et `Q` attend que `P` signale sa fin d'accès.

Famine. Un processus est bloqué en attente d'une fin d'accès qui n'arrivera jamais.

IPC System V – <sys/ipc.h>

Caractéristiques : Extérieurs au système de gestion des fichiers (pas désignés localement par des descripteurs). Gestion par le système avec une table spécifique par type d'objet. Chaque dispose d'un identification interne (et externe, par une clé).

IPC SysV : Caractéristiques communes

Une table par mécanisme :
 une entrée ⇒ une instance.
 une clé numérique par entrée.
un appel système `xxxget` par mécanisme
`xxx` ⇒ shm (mémoire partagée), msg (files messages) ou sem (sémaphores)
 crée une nouvelle entrée ou retrouve une déjà existante
 retourne un descripteur
Cas de création :
 cle = `IPC_PRIVATE`
 `IPC_CREAT` ⇒ `flags`
 `IPC_CREAT | IPC_EXCL` ⇒ `flags`
Commandes Shell associées :
 `ipcs` : liste des ressources actives ainsi que leurs caractéristiques.
 `ipcrm` : suppression des ressources.
Une structure commune :

```
struct ipc_perm {
    ushort uid; /* owner's user id */
    ushort gid; /* owner's group id */
    ushort cuid; /* creator's user id */
    ushort cgid; /* creator's group id */
    ushort mode; /* access modes */
    ushort seq; /* slot usage sequence number */
    key_t key; /* key */
};
```

Définitions communes :

```
#define IPC_CREAT 0001000 /* create entry if key
    doesn't exist */
#define IPC_EXCL 0002000 /* fail if key exists */
#define IPC_NOWAIT 0004000 /* Error if request
    must wait */
#define IPC_PRIVATE (key_t)0 /* private key */
#define IPC_RMID 0 /*remove identifier */
#define IPC_SET 1 /* set options */
#define IPC_STAT 2 /* get options */
```

Composition d'un clé : la soumission d'un clé permet d'obtenir un descripteur d'IPC.

```
key_t ftok(const char* path, int id); -- <sys/ipc
.h> <sys/types.h>
```

Crée une clé unique (utilisable dans semget, shmget).
`path` : doit exister tant que les clés y sont associées. On utilise généralement "/tmp".
@return : -1 si path n'existe pas ou n'est pas accessible.

```
int xxxget(key_t key, int flag);
```

si clé = IPC_PRIVATE
 un nouvel objet est créé dans la table correspondante.
 Utilisé au sein du même processus.
sinon
 Si objet n'existe pas
  Si (flag & IPC_CREAT) : Un nouvel objet est créé dans la table correspondante.
  Sinon : erreur
 Sinon ((flag & IPC_CREAT) && (flag_IPC_EXCL))
  erreur
 sinon
  l'identification de l'objet est renvoyée

Les files de messages System V

Caractèristiques : Paquets identifiables et indivisibles et non un flou de caractères. On spécifie l'id de la file et non celui du processus lors d'un émission/réception. Politique FIFO. Connaissance + droits d'accès.

Message : composé de :

`Type` : nombre entier (long) strictement positif.
`Donnee` : Suite d'octets contigus en mémoire.

```
struct msgbuf{ /* dans sys/msg.h */
    long mtype; /* Type du message */
    type data; /* la donnee, d'un type quelconque
        */
};
```

Sockets

Socket (TCP / UDP / IP) : Communication intra-tâches distante, bidirectionnelle, fiable sans perte de données.

Types de sockets :

-> SOCK_STREAM
Mode connecté (fonctionnement similaire à un tube)
avec contrôle de flux
bidirectionnel
protocole TCP (IPPROTO_TCP)

-> SOCK_DGRAM
Mode non connecté (possible d'inverser les paquets)
transmission par paquets
sans contrôle de flux
bidirectionnel
protocole UDP (IPPROTO_UDP) ou IGMP (IPPROTO_IGMP)

-> SOCK_SEQPACKET
Mode non connecté garantissant l'intégrité
transmission par paquet
bidirectionnel
protocole UDP ou IGMP

-> SOCK_RAM
Accès direct avec la couche IP
réservé au super-utilisateur
protocole ICMP (IPPROTO_ICMP)
définition de nouveaux protocoles (IPPROTO_RAW)

Choix du protocole :

-> TCP (Transport Control Protocol)
Pas de perte, pas de déséquencement, flux
Coûteux (connexion, transfert)
Type = STREAM

-> UDP (User Datagram Protocol)
Perte, déséquencement, taille limitée.
Performant
Type = DGRAM

Création :

Socket associée à un descripteur de fichier.

```
#include <sys/types.h>
#include <sys/socket.h>
struct sockaddr {
    unsigned short sa_family; /* Adress family,
        AF_XX */
    char sa_data[14]; /* 14bytes of protcol
        adress */
}; /* tous les pointeurs de structures d'adresse
     de socket seront castes a un pointeur de ce
     type avant d'etre utilises dans les
     differentes fonctions */

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

domaine : AF_UNIX | AF_LOCAL | AF_FILE | AF_INET | AF_INET6 | AF_UNSPEC
type : SOCK_STREAM, SOCK_DGRAM, ...
protocole : 0 (choix automatique) | IPPROTO_TCP | TPPROTO_UDP)
@return : descripteur de fichier ou -1 en cas d'erreur

Nommage :

Objectif : Associer un nom à une socket, pour identifier un serveur d'une manière unique.
Domaine unix : nom=nom de fichier
Dom. internet (inet) : nom=<numéro de port,adresse IP>

```
int bind(int sock, struct sockaddr *nom, int
lg_nom);
```

sock : numéro de socket (retourné par socket())
nom : nom de la socket(dépendant du domaine)
lg_nom : longueur du nom sizeof(struct sockaddr)

Nommage dans le domaine unix (local)

Communication via le système de fichiers.

```
#include <sys/un.h>
struct sockaddr_un {
    short int sun_family; /* AF_UNIX(AF_LOCAL)*/
    char sun_path[104]; /* Chemin du fichier */
};
```

Nommage dans le domaine Internet

```
#include <netinet/in.h>
struct sockaddr_in {
    short sin_family; /* AF_INET */
    u_short sin_port; /* numero du port */
    struct in_addr sin_addr; /* adresse IP */
};
struct in_addr {
    u_long s_addr; /*INADDR_ANY=machine locale*/
};
```

Nommage : Formattage d'adresse

```
u_short htons(u_short)
```
Host TO Network Short, conversion du numéro de port
```
u_long htonl(u_long)
```
Host TO Network Long, conversion de l'adresse IP
```
u_short ntohs(u_short)
```
Network TO Host Short
```
u_long ntohl(u_long)
```
Network TO Host Long
```
char* inet_ntoa(struct in_addr adr)
```
adr à chaine a.b.c.d (affichage)
```
u_long inet_addr(char* chaine)
```
chaine à adr

Nommage : Correspondance nom/adresse

```
#include <netdb.h>
struct addrinfo {
    int ai_flags; /*input flags: AI_PASSIVE,
        AI_CANONNAME, AI_NUMERICHOST*/
    int ai_family; /*protocol family, PF_xx*/
    int ai_socktype; /*socket type, SOCK_xx*/
    int ai_protocol; /*0 | IPPROTO_xx for IPv4/6
        */
    socklen_t ai_addrlen; /*length of ai_addr*/
    struct sockaddr *ai_addr; /*socket binary
        adress */
    char *ai_canonname; /*canonical name for
        hostname*/
    struct addrinfo *ai_next; /*pointer to next
        in list*/
};
```

```
int getaddrinfo(const char *host, const char*
 serv, const struct addrinfo *hints, struct
addrinfo **result);
```

Recherche les infos dans /etc/hosts ou pages jaune (NIS) ou serveur de noms (DNS) - Ordre défini dans /etc/nsswitch.conf (la page du man est très complète)
host : hostname ou adresse IP
serv : numéro de port (ou 0)
hints : si pas NULL spécifie les critères de sélection
@return : 0 en cas de succès, errno sinon

Nommage Internet : numéro de port

Entier sur 16bits, Fichier /etc/services

```
struct servent {
    char *s_name; /*official name of service*/
    char **s_aliases; /*alias list*/
    int s_port; /*port number*/
    char *s_proto; /*protocol to use*/
};
```

Famille de fonctions de recherche d'infomation dans la services database /etc/services, une connexion à la database est ouverte si nécessaire. :

```
struct servent *getservbyname(char *name, char *
proto);
```

Cherche l'entrée dans la database de nom name, et de protocole proto. Si proto = NULL n'importe quel protocole sera cherché.
@return : ptr sur servent, NULL si erreur ou EOF

```
struct servent *getservent(void);
```

Lit l'entrée suivante dans la database.
@return : ptr sur servent, NULL si erreur ou EOF

```
struct servent *getservbyport(int port, const
char *proto);
```

Cherche l'entrée dans la database de port port, Si proto = NULL n'importe quel protocole sera cherché.
@return : ptr sur servent, NULL si erreur ou EOF

```
void setservent(int stayopen);
```

Ouvre une connexion à la database, et set l'entrée suivante comme première entrée. Si stayopen n'est pas nul, la connexion à la database ne sera pas fermée entre deux appels à getserv*().

```
void endservent(void);
```

Ferme la connexion à la database.

Sockets non connectées : communication

```
int recvfrom(int sock, char* buffer, size_t
length, int flags, struct sockaddr *adrsrc,
socklen_t *adr_len);
```

Appel bloquant. Lecture d'un buffer addressé à la socket `sock`, si `adrsrc` n'est pas `NULL` il prend l'adresse de l'émetteur, `adr_len` est initialisé avec la taille du buffer associé à `adrsrc` puis modifié au retour avec la taille de l'adresse.
`flags` : (0 : default)
- `MSG_OOB` : données hors bande
- `MSG_PEEK` : Lecture sans modification de la file d'attente
- `MSG_WAITALL` : Lecture reste bloquante jusqu'à réception d'au moins `length` octets

@return : taille du message lu, 0 : aucun message à lire ou déconnexion, -1 en cas d'erreur.

```
size_t sendto(int socket, const void *buffer,
size_t length, int flags, const struct sockaddr
*dest_addr, socklen_t dest_len);
```

Envoie par la socket `sock` du contenu de `buffer` de taille `length` à l'adresse `dest_addr`

Sockets connectées : Connexion

Côté serveur :
- 1sock pour les demandes de connexion
- 1sock pour les communications

```
int listen(int sock, int taille_file);
```

Appel non bloquant. Création de la file d'attente des requêtes de connexion.
@return : 0 : succès, -1 : erreur

```
int accept(int sock, struct sockaddr *adr,
socklen_t *adr_len);
```

Attente d'acceptation d'une connexion. Extrait la première requête de connexion dans la file, crée une nouvelle socket avec les même propriétées de `sock`, et alloue un nouveau descripteur pour la socket. Si aucune requête n'est dans la file : Appel bloquant.
`adr` : identité du client
`sock` : créé avec `socket()`
@return : le descripteur de la socket, ou -1 en cas d'erreur.

Côté client :
- 1sock pour communiquer

```
int connect(int sock, struct sockaddr *adr,
socklen_t adr_len);
```

Appel bloquant. Demande d'une connexion.
@return : 0 :succès, -1 :error.

Sockets connectées : comunication

Primitives UNIX :

```
int read(int sock, char *buffer, int buf_len);

int write(int sock, char *buffer, int buf_len);
```

Regroupement de plusieurs écritures/lectures :

```
int recv(int sock, char *buff, int buf_len, int
flag);

int send(int sock, char *buff, int buf_len, int
flag);

int recvmsg(int sock, struct msghdr *msg, int
flag);

int sendmsg(int sock, struct msghdr *msg, int
flag);
```

-> Même `flag` que recvfrom.

Sockets connectés : déconnexion

```
int shutdown(int sock, int how);
```

`sock` : descripteur de la socket `how` : mode de déconnexion
0 : réception désactivée
1 : émission désactivée
2 : émission et réception désactivées
@return : 0= succès, -1= erreur
Note : shutdown est censé être suivi d'un close.

Temps réel

Principes

Système en temps réel (TR). Système dépendant de :
- L'exactitude des calculs ;
- Du temps mis à produire les résultats.

Échéance. Contrainte du temps bornant l'occurrence d'un événement (eg. production d'un résultat, envoi de signal...).

Garantir qu'un événement se produit à une date donnée ou avant une échéance donnée :
- Contraintes périodiques : le service doit être rendu selon un certain rythme ;
- Contraintes ponctuelles : lorsque l'état de Y se produit, il doit être traité dans un temps limité.

Garanties d'un système TR :
- Un événement A se produise avant un événement B ;
- Aucun événement externe à l'application TR ne retardera pas les processus critiques ;
- Ordonnancement garantissant les priorités entre les taches.

Cause des problématiques : E/S hardware, E/S utilisateur, journalisation de données, exécution de taches de fond.

POSIX Real-Time Scheduling Interfaces (POSIX.4) :
- Gestion dynamique de l'ordonnancement ;
- Horloges et temporisateurs évolués ;
- Verrouillage mémoire : `mlock`, `mlockall`, `munlock` ;
- Ajout d'E/S asynchrones ;
- Signaux Temps Réel : files d'attente de signaux, signaux avec priorité, passage de paramètres plus significatifs au handler, N signaux utilisateur (de `RTMIN` à `RTMAX`).

Mesure du temps. La période minimale dépend de la puissance (cadence) du processeur. Décomposition du temps en TICK horloge : constante `HZ` dans `<sys/param.h>` (période de TICK : 1000000 / `HZ`).

Horloges POSIX

- 3 horloges, dont une horloge globale : `ITIMER_REAL` ;
- 2 fonctions principales `time()` et `gettimeofday()` ;
- *Epoch* : naissance d'UNIX – 1er janvier 1970 à `0:00am`

Fonctionnement. Autant d'horloges que définies dans le *header* `<time.h>`. Le nombre et leur précision dépendent de l'implémentation du système.

Un identifiant par horloge (de type `clkid_t`). Une horloge POSIX doit être fournie par l'implémentation `CLOCK_REALTIME`.

Structure de comptabilité du temps à granularité en ns :

```
struct timespec {
       time_t tv_sec; /* s dans l'intervalle */
       time_t tv_nsec; /* ns dans l'intervalle */
};
```

```
int clock_getres(clkid_t clkid, struct timespec *
 res);
```

Permet de récupérer la précision (*resolution*) de l'horloge spécifiée `clkid`, et si `res != NULL`, stocke le résultat dans la `struct timespec` pointée par `res`.

```
int clock_gettime(clkid_t clkid, struct timespec
* tp);
```

Permet de récupérer l'heure courante de l'horloge spécifiée `clkid` dans la `struct timespec` pointée par `tp`.

```
int clock_settime(clkid_t clkid, const struct
timespec * tp);
```

Permet de changer l'heure courante de l'horloge `clkid` à partir de la `struct timespec` pointée par `tp`.

Valeur de retour : 0 si succès, -1 sinon (`errno`).

Temporisateurs Temporisateurs UNIX : 2 types :
- Ponctuel (timer ponctuel UNIX de base : `sleep`) ;
- Périodique.

POSIX autorise N temporisateurs par processus (au min. 32, au Max. `TIMER_MAX` (définie dans `<limits.h>`).

Un temporisateur est identifié par un identifiant UNIX, un événement spécifique déclenché à l'échéance, ressources autorisées (à libérer après utilisation).

Structure de description de temporisateur :

```
struct itimerspec {
    struct timespec it_value; /* 1st exp. */
    struct timespec it_interval; /* next exp. */
};
```

Temporisateur ponctuel ⇔ `it_interval = 0`.

```
int clock_nanosleep(clkid_t clkid, int flags,
const struct timespec * rqtp, struct timespec *
rmtp);
```

Attente contrôlée :
- `clkid` : identifiant de l'horloge utilisée pour mesurer le temps ;
- `flags` :
  - `TIMER_ABSTIME` : temps absolu (ie. date précise, construite avec mktime). Suspend le thread courant jusqu'à ce que soit le temps de l'horloge `clkid` atteigne le temps absolu spécifié dans `rqtp`, ou un signal, dont l'action a été modifiée par un handler a été reçu, ou le processus est terminé.
  - `0` : temps relatif (ie. à partir de l'appel). Suspend l'exécution du thread courant jusqu'à ce que soit l'intervalle de temps spécifié dans `rqtp` soit écoulé, ou un signal dont l'action a été modifiée par un handler a été reçu, ou le processus est terminé.
- `rqtp` : échéance du réveil ;
- `rmtp` : temps restant jusqu'à l'échéance si un signal a interrompu le sommeil.

Valeur de retour : 0 si le temps requis est écoulé, un code d'erreur sinon.

```
int timer_create(clkid_t clkid, struct sigevent *
 evp, timer_t * timerid);
```

Création d'un temporisateur.
- `clkid` : identifiant de l'horloge régissant le temporisateur ;
- `evp` : évènement déclenché lorsque le temporisateur arrive à échéance ;
- `timer_id` : identifiant du temporisateur créé retourné par la fonction.

Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int timer_delete(timer_t timerd);
```

Destruction du temporisateur `timerid` (implicite à la terminaison du processus propriétaire).

Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int timer_getoverrun(timer_t timerid);
```

Détermination de débordement (ie. nombre de fois où le timer a été déclenché sans avoir été traité).
- `timerid` : identifiant du temporisateur consulté ;

Valeur de retour : nombre de déclenchements d'expiration du temporisateur `timerid` non traités si succès, -1 sinon (`errno`).

```
int timer_gettime(timer_t timerid, struct
itimerspec *value);
```

Consultation de temporisateur.
- `timerid` : identifiant du temporisateur consulté ;
- `value` : temps restant jusqu'à la prochaine échéance de `timerid` ;

Valeur de retour : 0 si succès, -1 sinon (`errno`).

```
int timer_settime(timer_t timerid, int flags
, const struct itimerspec * value, struct
itimerspec * ovalue);
```

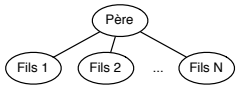
Armement de temporisateur.
- `timerid` : identifiant du temporisateur à armer ;
- `flags` : mode de temporisation :
  - `TIMER_ABSTIME` : temps absolu (ie. date précise, construite avec `mktime`). Le temps jusqu'à la prochaine expiration du temporisateur est égal à la différence par rapport au temps absolu spécifié par le champs `it_value` de `value` et la valeur courante de l'horloge associée à `timerid`. Le temporisateur expire lorsque l'horloge associée atteint la valeur spécifiée par `it_value`.
  - `0` : temps relatif (ie. à partir de l'armement). Le temps jusqu'à la prochaine expiration du temporisateur est égal à l'intervalle spécifié par le champ `it_valeue` de `value`. Le temporisateur expire dans `it_value` nanosecondes de l'appel à la fonction.
- `value` : échéances du temporisateur (1re et suivantes) ;
- `ovalue` : temps restant jusqu'à la prochaine échéance **courante** (ie. avant réarmement) ;

Valeur de retour : 0 si succès, -1 sinon (`errno`).

Schémas et programmation

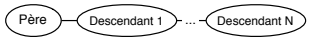
Un processus créé N fils

```
for(i=0; (i<N)&& ((pid = fork())>0); i++);
```



Un processus créé N descendants

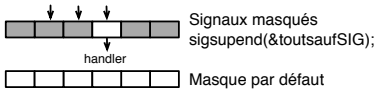
```
for(i=0; (i<N)&& ((pid = fork())==0); i++);
```



Modification du handler associé à SIGINT

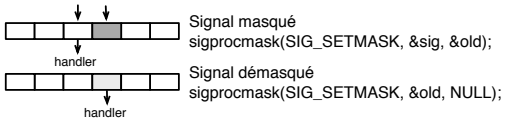
```
void sig_hand(int sig){
    printf("Signal %d.\n", sig);
}
...
struct sigaction action, old;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
action.sa_handler = sig_hand;
sigaction(SIGINT, &action, &old);
```

Masquer le signal SIGINT




```
sigset_t queSIGINT, old;
sigemptyset(&queSIGINT);
sigaddset(&queSIGINT, SIGINT);
sigprocmask(SIG_SETMASK, &queSIGINT, &old);
/* Actions avec SIGINT masque */
sigprocmask(SIG_SETMASK, &old, NULL);
```

Se bloquer en attente du signal SIGINT




```
sigset_t toutsaufSIGINT;
sigfillset(&toutsaufSIGINT);
sigdelset(&toutsaufSIGINT, SIGINT);
//Attente du signal SIGINT
sigsupend(&toutsaufSIGINT);
```

Créer un fichier vide

Ouvre un fichier pour l'écriture, créé le fichier s'il n'existe pas, sinon le système vide le fichier déjà existant.

```
int fd;
char *filename = "/tmp/file";
int flags = O_WRONLY | O_CREAT | O_TRUNC;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP |
    S_IROTH;
if((fd = open(filename, flags, mode)) == -1) {
    perror("open"); exit(1);
}
```

Test de l'existence d'un fichier

Avec `stat()` :

```
struct stat stat_info;
char *filename = "/tmp/file";
if(stat(filename, &stat_info) == -1){
    if(errno == ENOENT){
        printf("%s already exists\n", filename);
    } else {
        perror("stat error");
        exit(EXIT_FAILURE);
    }
}
```

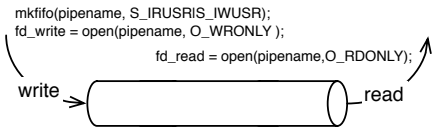
Avec `access()` :

```
char *filename = "/tmp/file";
if(access(filename, F_OK) == -1) {
    if(errno == ENOENT) {
        printf("%s already exists\n", filename);
    } else {
        perror("access error");
        exit(EXIT_FAILURE);
    }
}
```

Avec `open()` (**déconseillé**) :

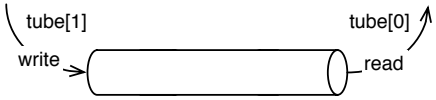
```
int fd;
char *filename = "/tmp/file";
int flags = O_WRONLY | O_CREAT | O_EXCL;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP |
    S_IROTH;
if((fd = open(filename, flags, mode)) == -1) {
    if(errno == EEXIST) {
        printf("Ouput file exists\n");
    } else {
        perror("open");
        exit(EXIT_FAILURE);
    }
}
```

Créer un tube nommé en lecture




```
int fd;
char *pipename = "pipe";

// ECRIVAIN
if(mkfifo(pipename, S_IRUSR|S_IWUSR) == -1){
    perror("mkfifo error");
    exit(EXIT_FAILURE);
}
if((fd = open(pipename, O_WRONLY)) == -1) {
    perror("open error O_WRONLY");
    exit(EXIT_FAILURE);
}
// LECTEUR
if((fd = open(pipename, O_RDONLY)) == -1) {
    perror("open error O_RDONLY");
    exit(EXIT_FAILURE);
}
```

Création d'un tube anonyme




```
int tube[2];
pid_t pid_fils;
struct results res;
if(pipe(tube) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
if((pid_fils = fork()) == 0){ //fils, ecriture
    /* construction de res */
    if(write(tube[1], res, sizeof(res)) == -1){
        perror("write pipe");
        exit(EXIT_FAILURE);
    }
    close(tube[1]);
} else if(pid_fils > 0) { //pere, lecture
    if(read(tube[1], res, sizeof(res)) == -1){
        perror("read pipe");
        exit(EXIT_FAILURE);
    }
    /* traitements sur res */
    close(tube[0]);
} else {
    perror("fork");
    exit(EXIT_FAILURE);
}
```

Créer un segment de mémoire partagée nommé partagé entre un client et un serveur

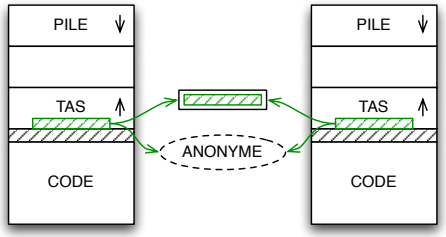
1. `shm_open()` : création d'un segment de mémoire partagée de taille 0 ;
2. `ftruncate()` : donner une taille au segment de mémoire partagée (en cas de réduction du segment de mémoire partagé, seul le surplus est détruit) ;
3. `mmap()` : mapper le segment de mémoire partagée dans l'espace d'adressage du processus.

```
struct shm_data * sp;
int fd_shm;
char * shm_name = "SHM";

// SERVEUR
//Ouverture shm
if((fd_shm = shm_open(shm_name, O_RDWR|O_CREAT,
     0600)) == -1) {
    perror("shm_open error");
    exit(EXIT_FAILURE);
}

//Allocation d'une taille au shm
if(ftruncate(fd_shm, sizeof(struct shm_data))
    == -1) {
    perror("ftruncate error");
    exit(EXIT_FAILURE);
}

//Mapping du shm
if((sp = mmap(NULL, sizeof(struct shm_data),
    PROT_READ|PROT_WRITE, MAP_SHARED, fd_shm,
    0)) == MAP_FAILED) {
    perror("mmap error");
    exit(EXIT_FAILURE);
}

// CLIENT
//Ouverture du shm cree par le serveur
if((fd_shm = shm_open(shm_name, O_RDWR, 0600))
    == -1) {
    perror("shm_open error");
    exit(EXIT_FAILURE);
}

//Mapping du shm ouvert
if((sp = mmap(NULL, sizeof(struct shm_data),
    PROT_READ|PROT_WRITE, MAP_SHARED, fd_shm,
    0)) == MAP_FAILED) {
    perror("mmap error");
    exit(EXIT_FAILURE);
}
```

Créer un segment de mémoire partagée anonyme

1. `mmap()` avec les arguments `flags = MAP_ANONYMOUS` et `fd = -1`.

Plusieurs variables partagées ⇒ Déclarer une structure `struct shared_data` contenant les variables partagées.
Attention. Cas des structures de structures avec les pointeurs de structures (la structure pointée ne se trouve pas dans la mémoire partagée, contrairement au pointeur).



Redéfinition du handler de SIGINT pour fermer les structures ouvertes avant de les ouvrir

Dans le code du handler : `if(fd != NULL)close(fd);`

Sémaphore anonyme

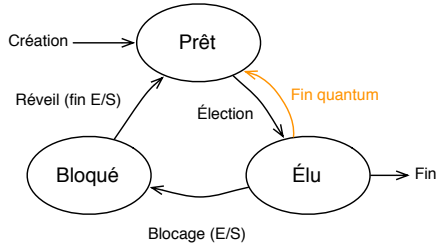
Utilisation avec un segment de mémoire partagée anonyme. Le sémaphore est alors partagé entre le père et sa descendance.

```
sem_t * sem;
if((sem = mmap(NULL, sizeof(sem_t), PROT_READ|
    PROT_WRITE, MAP_ANONYMOUS|MAP_SHARED, -1,
    0)) == MAP_FAILED) {
    perror("mmap anonymous error");
    exit(EXIT_FAILURE);
}
//Ouverture du semaphore de synchronisation
    entre le pere et les fils
if(sem_init(sem, 1, 0) == -1) {
    perror("sem_init error");
    exit(EXIT_FAILURE);
}
```

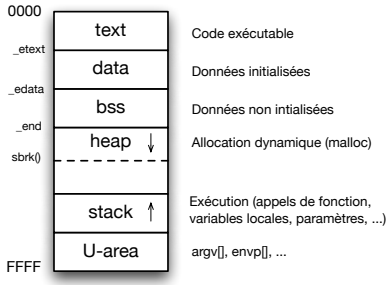
Emplacement des sémaphores et segments de mémoire partagée

Placés dans le répertoire `/dev/shm` dans les distributions Linux.

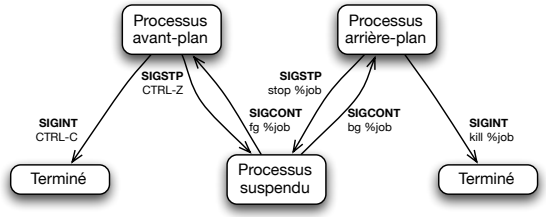
Différents états d'un processus



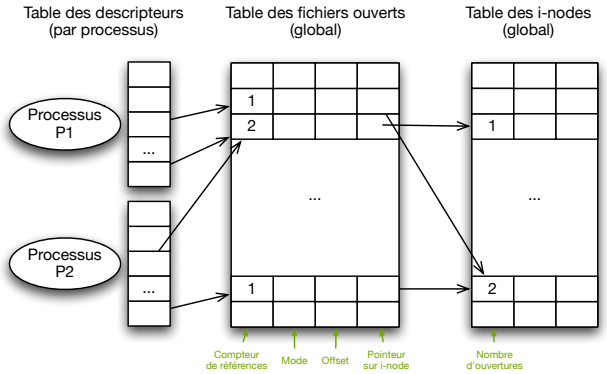
Segments d'un processus



Etats d'un processus

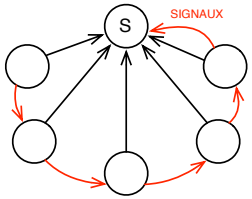


Descripteurs de fichier



Synchronisation sans sémaphores

Utilisation d'un anneau où un signal part de la source et se propage jusqu'à atteindre la source. Il est impossible de compter le nombre de signaux qui arrivent à la source.
Problème de performance : plus de parallélisme.


S
SIGNAUX

ERRNO ERRORS

Fichier /usr/include/asm-generic/errno-base.h

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
#define ESRCH 3 /* No such process */
#define EINTR 4 /* Interrupted system call */
#define EIO 5 /* I/O error */
#define ENXIO 6 /* No such device or address */
#define E2BIG 7 /* Argument list too long */
#define ENOEXEC 8 /* Exec format error */
#define EBADF 9 /* Bad file number */
#define ECHILD 10 /* No child processes */
#define EAGAIN 11 /* Try again */
#define ENOMEM 12 /* Out of memory */
#define EACCES 13 /* Permission denied */
#define EFAULT 14 /* Bad address */
#define ENOTBLK 15 /* Block device required */
#define EBUSY 16 /* Device or resource busy */
#define EEXIST 17 /* File exists */
#define EXDEV 18 /* Cross-device link */
#define ENODEV 19 /* No such device */
#define ENOTDIR 20 /* Not a directory */
#define EISDIR 21 /* Is a directory */
#define EINVAL 22 /* Invalid argument */
#define ENFILE 23 /* File table overflow */
#define EMFILE 24 /* Too many open files */
#define ENOTTY 25 /* Not a typewriter */
#define ETXTBSY 26 /* Text file busy */
#define EFBIG 27 /* File too large */
#define ENOSPC 28 /* No space left on device */
#define ESPIPE 29 /* Illegal seek */
#define EROFS 30 /* Read-only file system */
#define EMLINK 31 /* Too many links */
#define EPIPE 32 /* Broken pipe */
#define EDOM 33 /* Math argument out of domain of func */
#define ERANGE 34 /* Math result not representable */

#endif
```

Fichier /usr/include/asm-generic/errno.h

```
#ifndef _ASM_GENERIC_ERRNO_H
#define _ASM_GENERIC_ERRNO_H

#include <asm-generic/errno-base.h>

#define EDEADLK 35 /* Resource deadlock would occur */
#define ENAMETOOLONG 36 /* File name too long */
#define ENOLCK 37 /* No record locks available */
#define ENOSYS 38 /* Function not implemented */
#define ENOTEMPTY 39 /* Directory not empty */
#define ELOOP 40 /* Too many symbolic links encountered */
#define EWOULDBLOCK EAGAIN /* Operation would block */
#define ENOMSG 42 /* No message of desired type */
#define EIDRM 43 /* Identifier removed */
#define ECHRNG 44 /* Channel number out of range */
#define EL2NSYNC 45 /* Level 2 not synchronized */
#define EL3HLT 46 /* Level 3 halted */
#define EL3RST 47 /* Level 3 reset */
#define ELNRNG 48 /* Link number out of range */
#define EUNATCH 49 /* Protocol driver not attached */
#define ENOCSI 50 /* No CSI structure available */
#define EL2HLT 51 /* Level 2 halted */
#define EBADE 52 /* Invalid exchange */
#define EBADR 53 /* Invalid request descriptor */
#define EXFULL 54 /* Exchange full */
#define ENOANO 55 /* No anode */
#define EBADRQC 56 /* Invalid request code */
#define EBADSLT 57 /* Invalid slot */

#define EDEADLOCK EDEADLK

#define EBFONT 59 /* Bad font file format */
#define ENOSTR 60 /* Device not a stream */
#define ENODATA 61 /* No data available */
#define ETIME 62 /* Timer expired */
#define ENOSR 63 /* Out of streams resources */
#define ENONET 64 /* Machine is not on the network */
#define ENOPKG 65 /* Package not installed */
#define EREMOTE 66 /* Object is remote */
#define ENOLINK 67 /* Link has been severed */
#define EADV 68 /* Advertise error */
#define ESRMNT 69 /* Srmount error */
#define ECOMM 70 /* Communication error on send */
#define EPROTO 71 /* Protocol error */
#define EMULTIHOP 72 /* Multihop attempted */
#define EDOTDOT 73 /* RFS specific error */
#define EBADMSG 74 /* Not a data message */
#define EOVERFLOW 75 /* Value too large for defined data type */
#define ENOTUNIQ 76 /* Name not unique on network */
#define EBADFD 77 /* File descriptor in bad state */
#define EREMCHG 78 /* Remote address changed */
#define ELIBACC 79 /* Can not access a needed shared library */
#define ELIBBAD 80 /* Accessing a corrupted shared library */
#define ELIBSCN 81 /* .lib section in a.out corrupted */
#define ELIBMAX 82 /* Attempting to link in too many shared libraries */
#define ELIBEXEC 83 /* Cannot exec a shared library directly */
#define EILSEQ 84 /* Illegal byte sequence */
#define ERESTART 85 /* Interrupted system call should be restarted */
#define ESTRPIPE 86 /* Streams pipe error */
#define EUSERS 87 /* Too many users */
#define ENOTSOCK 88 /* Socket operation on non-socket */
#define EDESTADDRREQ 89 /* Destination address required */
#define EMSGSIZE 90 /* Message too long */
```

```
#define EPROTOTYPE 91 /* Protocol wrong type for socket */
#define ENOPROTOOPT 92 /* Protocol not available */
#define EPROTONOSUPPORT 93 /* Protocol not supported */
#define ESOCKTNOSUPPORT 94 /* Socket type not supported */
#define EOPNOTSUPP 95 /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT 96 /* Protocol family not supported */
#define EAFNOSUPPORT 97 /* Address family not supported by protocol */
#define EADDRINUSE 98 /* Address already in use */
#define EADDRNOTAVAIL 99 /* Cannot assign requested address */
#define ENETDOWN 100 /* Network is down */
#define ENETUNREACH 101 /* Network is unreachable */
#define ENETRESET 102 /* Network dropped connection because of reset */
#define ECONNABORTED 103 /* Software caused connection abort */
#define ECONNRESET 104 /* Connection reset by peer */
#define ENOBUFS 105 /* No buffer space available */
#define EISCONN 106 /* Transport endpoint is already connected */
#define ENOTCONN 107 /* Transport endpoint is not connected */
#define ESHUTDOWN 108 /* Cannot send after transport endpoint shutdown */
#define ETOOMANYREFS 109 /* Too many references: cannot splice */
#define ETIMEDOUT 110 /* Connection timed out */
#define ECONNREFUSED 111 /* Connection refused */
#define EHOSTDOWN 112 /* Host is down */
#define EHOSTUNREACH 113 /* No route to host */
#define EALREADY 114 /* Operation already in progress */
#define EINPROGRESS 115 /* Operation now in progress */
#define ESTALE 116 /* Stale NFS file handle */
#define EUCLEAN 117 /* Structure needs cleaning */
#define ENOTNAM 118 /* Not a XENIX named type file */
#define ENAVAIL 119 /* No XENIX semaphores available */
#define EISNAM 120 /* Is a named type file */
#define EREMOTEIO 121 /* Remote I/O error */
#define EDQUOT 122 /* Quota exceeded */

#define ENOMEDIUM 123 /* No medium found */
#define EMEDIUMTYPE 124 /* Wrong medium type */
#define ECANCELED 125 /* Operation Canceled */
#define ENOKEY 126 /* Required key not available */
#define EKEYEXPIRED 127 /* Key has expired */
#define EKEYREVOKED 128 /* Key has been revoked */
#define EKEYREJECTED 129 /* Key was rejected by service */

/* for robust mutexes */
#define EOWNERDEAD 130 /* Owner died */
#define ENOTRECOVERABLE 131 /* State not recoverable */

#endif
```