# Unix : Quelques fonctions système

**RAPPEL :** Pour obtenir le manuel en ligne complet d'une des fonctions décrites sommairement ci-dessous, utiliser la commande `man`. Pour obtenir le manuel de `getpid` par exemple, taper :

`man getpid`
ou `man 2 getpid` (fonction `getpid` du manuel 2).
ou `man 3 getpid` ou encore `man P getpid`

## 1 Entrées-Sorties dans un fichier

### 1.1 Flux de fichier – *file stream*

Examples :

- Ouverture : `FILE* f = fopen("nomDuFichier.watever", "r");`
- Fermeture : `fclose(f);`
- Lecture : `nRead = fscanf(f, "%d;%d;%d\n", &n1, &n2, &n3);`
- Écriture : `fprintf(f, "%d;%d;%d\n", n1, n2, n3);`

### 1.2 Descripteur de fichier – *file descriptor*

Examples :

- Ouverture : `int fd = open("nomDuFichier.watever", O_RDWR);`
- Fermeture : `close(fd);`
- Lecture : `nByteRead = read(fd, buffer, bufSize);`
- Écriture : `write(fd, buffer, nByteToWrite);`

## 2 Affichage de message d'erreurs

Quand on appelle une fonction du système Unix, si une erreur se produit, la fonction renvoie -1 et affecte une valeur à une variable globale `errno` (cette variable reste inchangée si aucune erreur ne se produit). On peut alors afficher un message sur la sortie erreur standard en utilisant la procédure `perror`.

```
#include <errno.h>
extern int errno;
#include <stdio.h>
void perror(const char* msg)
```

Affiche sur la sortie erreur standard un message correspondant à l'erreur de numéro `errno` et le message (chaîne de caractères) `msg`.

## 3 Signaux

Grosso-modo, un *signal* Unix est une interruption qui peut être soit ignorée, soit faire l'objet d'un traitement par défaut, soit faire l'objet d'un traitement spécifique. Un signal peut être généré suite à une opération arithmétique (division par 0), peut être généré grâce à un appel de procédure explicite, ou peut être généré par un dispositif matériel (frappe d'une touche sur le clavier, horloge). Il existe de nombreux signaux (faire `man 5 signal` pour avoir la liste complète), nous en donnons quelques exemples ci-dessous :

| Identificateur | Numéro | Libellé | Action par défaut |
|---|---|---|---|
| SIGINT | 2 | interrupt | arrêt du programme |
| SIGQUIT | 3 | quit | arrêt + core |
| SIGILL | 4 | illegal instruction | arrêt + core |
| SIGABRT | 6 | software abort (3C) | arrêt + core |
| SIGFPE | 8 | floating point exception | arrêt + core |
| SIGKILL | 9 | kill | arrêt |
| SIGALRM | 14 | alarm clock | ignoré |
| SIGTSTP | 25 | keyboard stop | ignoré |

Pour définir le traitement associé à un signal, on appelle la fonction `signal` :

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Un `sighandler_t` est une procédure de traitement d'un signal [1] ; elle reçoit en entrée le numéro du signal qui a déclenché son appel. La fonction `signal` associe un signal à une procédure de traitement, elle a deux paramètres :

- `sig` : identificateur du signal à traiter (`SIGINT`, `SIGQUIT`, ...)
- `action` : une des trois valeurs `SIG_DFL`, `SIG_IGN` ou pointeur sur une procédure de traitement du signal.
  `SIG_DFL` demande l'utilisation du traitement par défaut, `SIG_IGN` permet d'ignorer le signal (certains signaux comme `SIGKILL` ne peuvent être ni ignorés, ni attrapés).

La valeur renvoyée est l'adresse de la routine de traitement qui était définie avant l'appel de la fonction.

Le fonctionnement, après appel de `signal(sig, &action)`, est le suivant : si un signal `sig` arrive le programme est interrompu, **le traitement par défaut du signal est rétabli** et la procédure `action` est appellée avec `action(sig)` ; quand `action` a fini de s'exécuter, l'exécution du programme reprend là où il a été interrompu.

**Remarques :**

- le rétablissement du traitement par défaut oblige à réarmer le signal à la fin de la procédure de traitement (voir exemple ci-dessous) ;
- dans certaine versions d'Unix, le traitement par défaut n'est pas automatiquement rétabli, on peut donc éviter le réarmement. Pour tout savoir, faire `man signal` sur votre Unix.

**Exemple d'utilisation :**

```
#include <signal.h>
void traiteQuit(int sig)
{
      /* Traitement du signal sig */
      ...
      /* Réarmement du signal */
      signal(SIGQUIT, traiteQuit);
}

int main (void)
{
      signal(SIGQUIT, traiteQuit);
```

1. ce type n'est pas défini dans toutes les versions d'Unix ou de Linux ; s'il ne l'est pas, la fonction `signal` est déclarée comme suit :
   `void (*signal(int sig, void (*action)(int)))(int)`
ce qui est nettement moins lisible...

```
        ...
        return EXIT_SUCCESS;
    }
```

**Autres fonctions :**

```
#include <signal.h>
int kill(pid_t pid, int sig);
```
Envoie le signal `sig` au processus de numéro `pid`.

```
int raise(int sig);
```
Envoie le signal `sig` au processus en cours.

# 4 Gestion simple du temps

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```
`alarm` demande à l'horloge système d'envoyer un signal `SIGALRM` après `seconds` secondes. La valeur renvoyée est le temps qui restait à s'écouler avant qu'un signal soit envoyé, en effet, les alarmes ne sont pas empilées et un appel à `alarm` annule les appels précédents.
`alarm(0)` annule tout appel précédent à `alarm`.

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds)
```
Le processus appelant s'endort pour environ `seconds` secondes. La valeur renvoyée par `sleep` est la différence entre le nombre demandé et le nombre de secondes effectives de sommeil. Cette valeur peut être $> 0$ car sleep est suspendue par n'importe quel signal arrivant au processus.

# 5 Gestion de processus

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork();
```
Le processus appelant crée un processus identique à lui-même ; le processus appelant est le père et le processus créé est le fils.
Attention, tout appel à la fonction `fork()` provoque cette création !
`fork()` rend la valeur 0 dans le processus fils, et le numéro d'identification (`PID`) du fils créé dans le processus père ; on récupère donc cette valeur par une instruction du type : `ident = fork()`. En cas d'erreur, `fork` renvoie -1.

```
#include <unistd.h>
pid_t getpid();
pid_t getppid();
```
Renvoient au processus appelant son numéro d'identification (`getpid`) ou le numéro d'identification de son père (`getppid`). Renvoient -1 en cas d'erreur.

```
#include <stdlib.h>
void exit(int status)
#include <unistd.h>
void _exit(int status)
```

Le processus appelant met fin à son exécution. Un appel `exit(status)` est équivalent à un `return status` dans la fonction `main` d'un programme C. L'entier `status` est un code de terminaison et est rendu au processus père si celui-ci attend la fin de son fils (voir `wait`). `_exit` est équivalent à `exit` sauf que _exit ne vide pas les tampons d'E/S.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
```

Cette fonction permet à un processus d'attendre la mort d'un de ses fils. Si le processus n'a pas eu de fils, ou si tous les fils sont morts au moment de l'appel de `wait`, la fonction rend -1. Si un fils est déjà mort, la fonction rend le numéro d'identification de ce fils. Sinon le processus est bloqué jusqu'au décès d'un fils (le premier qui meurt) et rend son numéro (cette attente n'est donc pas sélective).
Cette fonction s'utilise de 2 façons :

| | |
|---|---|
| `wait(NULL)` | donne le fonctionnement décrit ci-dessus ; |
| `wait(&codeRetour)` | la valeur renvoyé par le fils par `exit` est stockée dans l'entier `codeRetour`. |

# 6 Exécution d'un programme

Un processus peut lancer un programme exécutable qui se trouve dans un fichier sur disque en utilisant un des appels système `exec()`. Le code du programme sur disque **remplace** le code du processus en cours et donc un appel à `exec()` ne retourne jamais, sauf en cas d'erreur où il retourne -1 et positionne la variable système `errno`.
On ne donne ci-dessous que les deux formes d'appels les plus utilisées, il en existe de nombreux autres qui se distinguent par les paramètres utilisés.

```
#include <unistd.h>
int execl(char *path,
          char *arg0, char *arg1, ..., char *argn, NULL)
```

Dans cette version, `path` est une chaîne de caractères contenant le nom absolu du fichier contenant le programme à exécuter, `arg0` contient par convention le nom du fichier (sans répertoire) et les autres `arg` sont les paramètres du programme à exécuter. La liste des paramètres *doit* se terminer par un pointeur nul (`NULL`).
**Exemple :** `execl("/bin/ls", "ls", "*.c", NULL);`

```
int execlp(char *file,
           char *arg0, char *arg1, ..., char *argn, NULL)
```

Comme `execl`, mais au lieu d'un chemin absolu `path`, on passe en paramètre le nom du fichier et le répertoire sera recherché dans la liste définie dans la variable de l'environnement `PATH`.

# 7 Tâches – *Threads*

Un *thread* (tâche) Unix, parfois appelé "processus léger" est une activité s'exécutant en parallèle avec d'autres activités. Cela ressemble à un processus sauf que tous les threads d'un même processus partagent le même espace mémoire (sauf la pile, qui est propre à chaque thread).

Lors de la création d'un thread, une activité est créée, avec un contexte (registres du processeur, priorité, droits d'accès, ...) et une pile propre. Nous décrivons ici l'interface standard POSIX pour les threads car celle-ci est la plus répandue (Linux, Windows NT, autres Unix sauf Solaris).

**Utilisation :**

```
#include <pthread.h>
```

et
`cc [ option ... ] fichier ... -lpthread [autre librairie ... ]`

La fonction POSIX utilisée pour créer des threads est :

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *parametre);
```

Les paramètres sont, dans l'ordre :

1. un résultat qui contiendra l'identifiant du thread créé une fois celui-ci lancé.
2. Les attributs de création du thread (priorité, taille minimale de la pile,...). On peut se contenter de la valeur `NULL` pour les cas simples (voir `pthread_attr_init` pour plus d'informations).
3. L'adresse (le nom) de la fonction à exécuter dans le nouveau thread. Cette fonction **doit** prendre un `void*` en paramètre et retourner un `void*` également.
4. Le dernier paramètre est un pointeur générique (`void*`) vers le paramètre qui sera passé au thread à son lancement. Pour passer plusieurs paramètres à un thread, il faudra les regrouper dans une structure et passer l'adresse de cette structure en paramètre.

Un thread se termine soit à la fin de l'exécution de la fonction asocciée, soit lors d'un appel à `pthread_exit()`. Cependant, si le thread principal, qui exécute `main`, se termine sur la fin de `main`, alors c'est tout le processus qui se termine, causant la fin du thread principal mais aussi de tous les sous-threads. Pour éviter cela, on termine toujours la fonction `main` par un appel à `pthread_exit()`, qui force la fin du thread mais pas la fin du processus, permettant ainsi aux autres threads de s'exécuter.

```
void pthread_exit(void * exitStatus);
```

**Exemple :**

```
/* Appel le plus simple, on ignore le code de retour */
pthread_exit(NULL);
```

Un thread peut attendre la fin d'un autre thread grâce à la fonction `pthread_join` :

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Le premier paramètre est bien sûr l'identification du thread à attendre tandis que le second permet de récupérer le code de sortie du thread (tel que spécifié par le `return` ou par `thread_exit`). Si on veut ignorer le code de retour, la forme la plus simple d'appel de `pthread_join` est :

```
pthread_join(idThread, NULL);
```

# 8 Tâches et sémaphores

On peut utiliser des sémaphores pour synchroniser les threads Unix. Toutes les primitives ci-dessous sont bien évidemment documentées en détail dans les manuels en ligne d'Unix, pour les utiliser, vous devez inclure le fichier `semaphore.h` au début de vos programmes et ajouter la librairie `rt` à la compilation :

```
#include <semaphore.h>
```

et
`cc [ option ... ] fichier ... -lrt [autre librairie ... ]`

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

Initialise le sémaphore `sem` avec `valInit`. `pshared` permet d'indiquer si le sémaphore restera local au processus courant (mais partagé entre les threads) ou global. Nous utiliserons des sémaphores locaux par défaut.

**Exemple :**

```
if (sem_init(&monSemaphore, 0, 1) == -1)
{
   fprintf(stderr, "Erreur à la création d'un sémaphore");
   exit(EXIT_FAILURE);
}
```

`int` **sem_wait**`(sem_t* sem);`
correspond à la primitive P théorique, mais n'est pas équivalente. L'algorithme de `sem_wait` est le suivant :

**tantque** sem.val = 0 faire
*dormir()*
**fintantque**
sem.val ← sem.val - 1

Cet algorithme peut conduire à un cas de famine si plusieurs tâches utilisent `sem_wait` sur le sémaphore.

`int` **sem_post**`(sem_t* sem);`
correspond à la primitive V théorique.

`int` **sem_destroy**`(sem_t* sem);`
efface un sémaphore lorsqu'il n'est plus utilisé, mais il faut être sûr qu'il n'est effectivement plus utilisé.

# 9 Tubes

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

`int` **pipe**`(int tube [2])`
Le processus appelant crée un tube de communication qui est accessible par lui-même et par tous les processus de sa descendance qui seront créés ultérieurement (fils, petits- fils). Le vecteur tube est un vecteur de descripteurs de fichiers où :

- `tube[0]` est ouvert en lecture, `tube[1]` ouvert en écriture ;
- les données écrites dans `tube[1]` (voir `write`) sont lues dans `tube[0]` (voir `read`) dans l'ordre *First-In*, *First-Out* (FIFO), c'est à dire *Premier-Entré*, *Premier-Sorti*.

**Exemple :**

- `write(tube[1], "bonjour", 8)` : envoi de caractères dans le tube, on indique le nombre de caractères à déposer (ici 8 pour écrire aussi le `'\0' de fin de chaîne.`
- `read(tube[0], tabcar, i) : lecture de i caractères dans le tube, les caractères sont copiés dans le tableau tabcar (qui doit pouvoir contenir au moins i caractères).`

`int` **read**`(int fildes, void* buf, size_t nbyte)`
`read` tente de lire `nbyte` octets dans le fichier associé au descripteur `fildes` (les octets lus sont copiés dans `buf`). `read` renvoie le nombre d'octets effectivement lus (<= `nbyte`) ou 0 si la fin du fichier est atteinte. Par défaut, un `read` sur un tube vide **bloque le processus appelant** jusqu'à ce que des données soient écrites dans le tube ou que le tube soit fermé en écriture (voir `close`). En cas d'erreur, `read` renvoie -1.

`int` **write**`(int fildes, char *buf, size_t nbyte)`

`write` tente d'écrire `nbyte` octets dans le fichier associé au descripteur `fildes` (les octets sont lus dans `buf`). `write` renvoie le nombre d'octets effectivement écrits ($<=$ `nbyte`). Par défaut, un `write` sur un tube réussit toujours mais peut éventuellement être bloquant si le buffer alloué au tube est plein (la taille de ce buffer est de 5120 octets sur Solaris-2.8 et de 4096 octets sur Linux-2.2).

**Nota Bene :** Les fonctions `read` et `write` sont les fonctions de lecture et d'écriture bas niveaux d'Unix, elles sont utilisées pour les tubes mais aussi pour toute communication avec un flot d'E/S (voir aussi `open` et `close`).

**Exemple :**

- `write(1, "bonjour", 8)` : écrit sur le flot 1, c'est-à-dire la sortie standard (écran) ;
- `read(0, tabcar, 80)` : lit 80 caractères sur le flot 0, c'est-à-dire l'entrée standard (clavier).