

# 1 Introduction

On va étudier un certain nombre de mécanismes de communication inter tâches (IPC=Inter-process communication)). Ces mécanismes servent à faire communiquer ou à synchroniser divers processus entre eux.

Parmi ces mécanismes on a :

- Les signaux ;
- les tubes
- les sémaphores
- les zones de mémoire partagées
- les sockets.

## 2 Les signaux

### 2.1 Définition

Un signal est une forme limitée de communication entre processus utilisée par les systèmes de type Unix. Les signaux sont soit envoyés par le noyau lors d'occurrences particulières (certaines erreurs par exemple) ou envoyés par des processus vers un autre processus. Les signaux peuvent varier d'un système à un autre, ils sont définis dans le fichier d'entête du système, `<signal.h>`. Le comportement des processus est défini d'une certaine manière pour chaque signal, et dans certains cas peut être intercepté et redéfini. SIGKILL et SIGSTOP ne peuvent pas être interceptés par le processus.

Voici un tableau qui donne un exemple des comportements par défaut :

Actions par défaut	Nom du signal
Fin du process	SIGHUP, SIGINT, SIGBUS, SIGKILL, SIGUSR1, SIGUSR2, SIGPIPE, SIGALRM, SIGTERM, SIGSTKFLT, SIGXCOU, SIGXFSZ, SIGVTALRM, SIGPROF, SIGIO, SIGPOLL, SIGPWR, SIGUNUSED
Fin du process et création core	SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGIOT, SIGFPE, SIGSEGV
Signal ignoré	SIGCHLD, SIGURG, SIGWINCH
Processus stoppé	SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU
Processus redémarré	SIGCONT

FIGURE 1 – Comportement par défaut vis à vis des signaux.

### 2.2 La primitive kill

Envoi d'un signal à un ou plusieurs processus. Le prototype :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Paramètres :

- pid : identificateur de processus vers lequel envoyer le signal
- Si pid > 0, le signal sig est envoyé au processus pid
- Si pid = 0, alors le signal sig est envoyé à tous les processus appartenant au même groupe que le processus appelant.
- Si pid = -1, alors le signal sig est envoyé à tous les processus sauf le premier (init)
- Si pid < -1, alors le signal sig est envoyé à tous les processus du groupe -pid.

### 2.3 La primitive signal

L'appel système `signal()` permet d'attacher un gestionnaire de traitement à un signal. Son prototype est le suivant :

```
#include <signal.h>
```

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

Le type `sighandler_t` représente l'adresse d'une fonction (un handler) qui prend en argument un entier et qui ne retourne rien.

Paramètres :

- `signum` : numéro du signal à gérer
- `handler` : gestionnaire de signal à installer

Description détaillée : `signal()` installe le gestionnaire `handler` pour le signal `signum`. `handler` peut être `SIG_IGN`, `SIG_DFL` ou l'adresse d'une fonction définie par le programmeur (un « gestionnaire de signal »).

Lors de l'arrivée d'un signal correspondant au numéro `signum`, un des événements suivants se produit :

- Si le gestionnaire est `SIG_IGN`, le signal est ignoré.
- Si le gestionnaire est `SIG_DFL`, l'action par défaut associé à ce signal est entreprise
- Si le gestionnaire est une fonction, `handler` est appelée avec l'argument `signum`.

Valeur de retour : `signal()` renvoie la valeur précédente du gestionnaire de signaux, ou `SIG_ERR` en cas d'erreur.

Voici un exemple :

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void fonction(int s);
int nsig;
void fonction(int s)
{
    printf("signal %d reçu %d fois \n ",s,++nsig);
}
int main()
{
    int s=4;
    if(signal(s,fonction)==SIG_ERR)
        printf("erreur je ne peux pas l'attraper (%d)\n",s);
    while(1)
    {
        pause(); //attendre un signal
        printf("Fin pause\n");
    }
}
```

### 3 Les tubes.

Lorsque deux processus désirent communiquer entre eux en envoyant plus d'informations (qu'avec les signaux) ils peuvent utiliser les tubes.

On distingue alors les deux processus par leur action :

- Un processus écrit des informations dans le tube. Celui-ci est appelé entrée du tube ;
- L'autre processus lit les informations dans le tube. Il est nommé sortie du tube.

Il existe deux types de tubes :

- Les tubes non nommés ;
- Les tubes nommés.

#### 3.1 Les tubes non nommés.

Les tubes non nommés sont appelés pipe en anglais et sont utilisés pour faire communiquer deux processus de la même filiation (`fork`). Il faut que le processus père crée le tube avant d'enfanter :

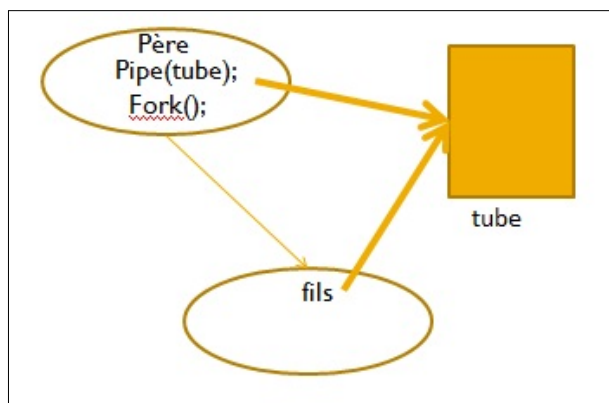


FIGURE 2 – Création d'un tube non nommé.

Le prototype de pipe :

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

La primitive `pipe()` crée un tube, un canal unidirectionnel de données qui peut être utilisé pour la communication entre processus.

Paramètre : `pipefd` est un tableau de deux entiers qui à l'issue de la primitive contient les deux descripteurs du tube.

valeur de retour : Cet appel renvoie 0 s'il réussit, ou -1 s'il échoue, auquel cas `errno` contient le code d'erreur.

Deux règles importantes pour les tubes :

- Un tube ne fonctionne que dans un sens.
- Un tube ne peut être utilisé qu'entre 2 processus

Une fois le tube créé le processus fils sera créé et le tableau de descripteur sera dupliqué et le tube pourra être partagé. Mais chaque processus devra fermer un descripteur du tube :

- un processus devra fermer l'extrémité lecture (celui qui veut écrire) : `close(fd[0])`
- un processus devra fermer l'extrémité écriture (celui qui veut lire) : `close(fd[1])`

Voici un exemple :

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
int main()
{
    int tube[2]; //descripteurs du tube
    int pid; //id de processus
    char buffer[100];
    if(pipe(tube) == -1)
    {
        printf("Erreur de création du tube\n");
        exit(0);
    }
    pid=fork(); //créer le fils
    if(pid==0)
    {
        char car;
        //le fils ferme l'extrémité écriture
        close(tube[1]);
        //lire dans le tube tant qu'on peut
        // et afficher
        while(read(tube[0], &car, 1))
            putchar(car);
        //fermer en lecture
        close(tube[0]);
        putchar('\n');
        exit(0);
    }
```

```
}  
else  
{  
    //le père ferme l'extrémité lecture  
    close(tube[0]);  
    printf("Je suis le père : j'envoie un message\n");  
    strcpy(buffer, "Message du père vers le fils");  
    write(tube[1], buffer, strlen(buffer));  
    close(tube[1]);  
    wait(NULL);  
}  
}
```

#### Remarques :

- Lorsque le père ferme le tube en écriture, la boucle de lecture s'arrête, car le read échoue à l'autre extrémité.
- Si on veut faire communiquer les deux processus dans les deux sens, il faut créer deux tubes et fermer les extrémités adéquates.

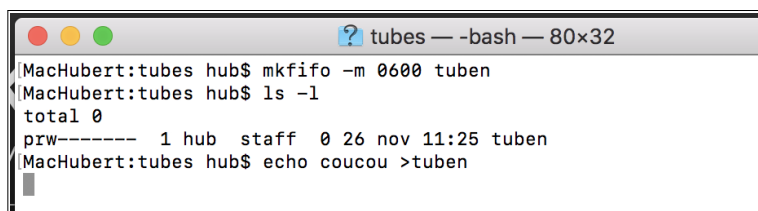
### 3.2 Les tubes nommés.

L'inconvénient des tubes non nommés est que deux processus qui ne sont pas de la même filiation ne peuvent communiquer de cette façon. C'est pourquoi si on veut communiquer par tube entre deux processus par exemple lancés séparément il faut utiliser les **tubes nommés**. Les **tubes nommés** sont des fichiers spéciaux gérés selon une méthodologie FIFO (First In First Out), ça veut dire que l'ordre des caractères en entrée est conservé en sortie (premier entré, premier sortie).

### 3.3 Création d'un tube nommé en ligne de commande

La commande `mkfifo` permet de créer un tube nommé. Il faut fournir un chemin de fichier. Par défaut la commande crée le tube avec les droits `r` et `w` (composé avec `umask`) pour tout le monde.

Voici un exemple :

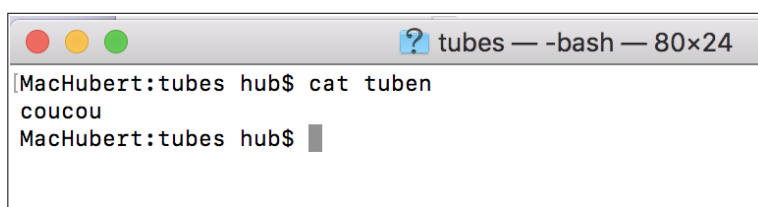


```
MacHubert:tubes hub$ mkfifo -m 0600 tuben  
MacHubert:tubes hub$ ls -l  
total 0  
prw----- 1 hub  staff  0 26 nov 11:25 tuben  
MacHubert:tubes hub$ echo coucou >tuben
```

FIGURE 3 – Créer et écrire dans un tube nommé **tuben** en ligne de commande.

On voit que la commande `echo` redirigée vers **tuben** ne rend pas la main !

Lorsque d'un autre terminal, on lit le fichier `tuben` on affiche bien les données et la commande `echo` a rendu la main.



```
MacHubert:tubes hub$ cat tuben  
coucou  
MacHubert:tubes hub$
```

FIGURE 4 – lire les données d'un tube nommé **tuben** en ligne de commande.

**Remarque :** la lecture est destructive !

### 3.4 Primitive de création d'un tube nommé.

L'appel système permettant de créer un tube nommé s'appelle aussi `mkfifo()`. Son prototype est :

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *filename, mode_t mode);
```

Paramètres :

- `filename` : nom donné au fichier spécial à créer.
- `mode` : permissions en octal

Valeur de retour : 0 en cas de succès, -1 en cas d'échec

### 3.5 Ouverture d'un tube nommé.

L'ouverture du tube nommé peut se faire par tout processus qui connaît le nom du fichier spécial. Cette ouverture se fait avec la primitive d'ouverture de fichier **open**.

Le comportement vis à vis de l'ouverture est quand même particulière. par défaut, l'ouverture du tube en lecture est bloquante tant qu'il n'y a pas ouverture en écriture et vice versa (l'écrivain et le lecteur sont alors synchronisés)

Avec le flag **O\_NONBLOCK** (mode non bloquant), `open` se comporte de façon asymétrique :

- En lecture seule, `open` retourne le descripteur immédiatement sans attendre l'ouverture en écriture par le producteur ;
- En écriture seule, `open` retourne le descripteur immédiatement ou échoue (`errno = ENXIO`) si il n'est pas ouvert en lecture en mode non bloquant par le producteur.

### 3.6 Opérations de lecture et écriture dans les tubes nommés.

Les opérations de lecture ou écriture se font avec les fonctions **read** et **write**.

Pour les tubes ouverts en mode bloquant (mode par défaut) :

- la lecture depuis un tube vide est bloquante (attente de données)
- l'écriture dans un tube plein est bloquante (attente d'une possibilité d'écriture ; la capacité en écriture d'un tube nommé est de `PIPE_BUF` défini dans `limits.h` et qui vaut en général 512)

Pour les tubes ouverts en mode non bloquant (**O\_NONBLOCK**)

- la lecture depuis un tube vide réussit et retourne 0 (octets)
- l'écriture vers un tube plein (ou presque plein) peut échouer (écriture de moins de `PIPE_BUF` octets) ou écrire seulement une partie des données (écriture de plus de `PIPE_BUF` octets).