CHAPITRE 3 : Communication entre processus par des files

1- Concept de base

Dans le système UNIX (ou Linux) les **files** (appelées plus couramment **tubes**) constituent le mécanisme de base de **communication** entre processus.

Une file est une structure de donnée qui permet de mémoriser une liste de données avec un mode d'accès particulier (premier entré, premier sorti) :

- le dépôt d'une donnée ne peut se faire qu'en **queue** de la file (opération **enfiler**)
- le retrait d'une donnée ne peut se faire qu'en **tête** de la file (opération **défiler**)

Les **tubes** permettent à deux ou plusieurs processus de communiquer entre eux de manière **asynchrone** suivant le modèle **producteur-consommateur**.

Le système UNIX (ou Linux) propose deux types de tubes :

- Les tubes **locaux** (**pipe**) qui sont des fichiers spéciaux créés **temporairement** dans le système de fichiers . Ils n'ont pas de nom (lien) et donc ils n'apparaissent dans aucun répertoire du système de fichiers.

  Un « **pipe** » est partageable uniquement entre des processus **parents** (le processus père qui crée le **pipe** et les processus fils qui en héritent par le mécanisme du **fork**).

  Lorsque tous les processus parents sont terminés, le tube est détruit par le système même s'il n'est pas vide !

- Les tubes **nommés** (**FIFO**) qui sont des fichiers spéciaux créés dans le système de fichiers. Comme les fichiers ordinaires, ils ont des attributs (inode, liens, droits d'accès, propriétaires, etc …) et apparaissent dans les différents répertoires où ils possèdent un lien.

  De ce fait, un « **FIFO** » est partageable par des processus **quelconques** (parents ou indépendants) à condition que les droits d'accès du FIFO le leur permettent.

  Comme un fichier ordinaire, un « FIFO » reste présent dans le système de fichiers tant que son dernier lien n'a pas été supprimé. Sa durée de vie n'est pas liée à celle des processus qui le partagent à un instant donné.

  Par contre, s'il reste des données dans un « FIFO » après que le dernier processus **consommateur** se soit terminé, le « FIFO » est **purgé** (taille du FIFO remise à 0).

Remarque :

La taille d'un fichier ordinaire est égale au nombre d'octets qui y sont stockés.
Par contre, les tubes sont des files d'attente et ont une **taille fixe** qui correspond au nombre maximum d'octets qui peuvent être stockés dans la file à un instant donné.
Par exemple sous Linux, la taille d'un tube est de 4K octets (constante PIPE_BUF déclarée dans limits.h).

2- Les tubes locaux (pipe)

2.1-Création d'un « pipe »

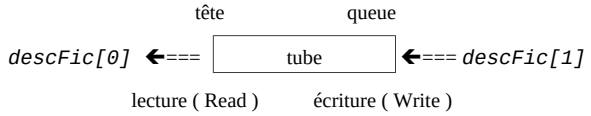
La fonction **pipe()** permet à un processus père de créer un tube local :

#include <unistd.h>

int **pipe**(int *descFic[2]*) ;

La fonction retourne **–1** en cas d'erreur et **0** si le tube est créé.

Si la création a réussi, le tableau *descFic* contient **2** descripteurs de fichiers associés respectivement à la **tête** et à la **queue** du tube.



descFic[0] contient un descripteur du fichier tube ouvert en **lecture**.

descFic[1] contient un descripteur du fichier tube ouvert en **écriture**.

Exemple : **création d'un tube local (pipe)**

```
……..
int descFic[2] ;
int err ;
……..

err = pipe( descFic ) ;
if ( err == -1 )
    { perror( "echec pipe" ) ; exit(1) ; }

……..
```

2.2- Lecture et écriture dans un « pipe »

Si on utilise les descripteurs *descFic[0]* et *descFic[1]*, la lecture et l'écriture dans le tube doivent être faites respectivement avec les appels systèmes **read()** et **write()**.

Ecriture avec write()

#include <unistd.h>

ssize_t **write**(int *descFic[1]*, const void * *tampon*, size_t *nb*) ;

Cette fonction essaie d'écrire en queue du tube désigné par *descFic[1]*, *nb* octets stockés en mémoire à l'adresse *tampon*
L'opération est **atomique** (indivisible) si *nb* ≤ PIPE_BUF (taille du tube), c'est-à-dire :

- s'il y a suffisamment de place libre dans le tube, les *nb* octets sont écrits et la fonction retourne la valeur *nb*
- si la place libre est insuffisante, le processus est **bloqué** jusqu'à ce qu'un ou plusieurs autres processus lisent suffisamment d'octets et libèrent assez de place pour que l'écriture des *nb* octets puisse se faire.
- pour éviter **les verrous mortels**, si le processus est **bloqué** et qu'il n'existe aucun processus susceptible de lire dans le tube (processus ayant un descripteur ouvert en lecture sur ce tube), le noyau envoie le signal **SIGPIPE** au processus bloqué. Le processus est débloqué,
  la fonction **write** se termine sans rien écrire et retourne la valeur **–1**,

L'opération ne peut plus être **atomique** (indivisible) si *nb* > PIPE_BUF (taille du tube).
Dans ce cas le processus doit attendre que le tube soit vide pour écrire une première tranche de PIPE_BUF octets puis attendre à nouveau pour écrire les octets restants, etc …

La situation se complique s'il y a plusieurs processus qui veulent écrire dans le tube car il peut alors y avoir **entrelacement** des données de ces différents processus. Il faut donc absolument éviter de se placer dans cette situation.

Lecture avec read()

#include <unistd.h>

ssize_t **read**(int *descFic[0]*, void * *tampon*, size_t *nb*) ;

Cette fonction essaie de lire en tête du tube désigné par *descFic[0]* , *nb* octets et les stocke en mémoire à l'adresse *tampon*
L'opération est **atomique** (indivisible) si *nb* ≤ PIPE_BUF (taille du tube), c'est-à-dire :

- si le tube n'est pas vide, le nombre *nbLu* d'octets lus sera égal à *nb* si le tube contient au moins *nb* octets sinon il sera égal au nombre d'octets présents dans le tube.
  La fonction retourne la valeur *nbLu* (≤ *nb*) qui est le nombre **effectif** d'octets lus.
- si le tube est vide, le processus est **bloqué** jusqu'à ce qu'un autre processus écrive un ou plusieurs octets dans le tube.
- pour éviter **les verrous mortels**, si le tube est vide et qu'il n'existe aucun processus susceptible d'écrire dans le tube (processus ayant un descripteur ouvert en écriture sur ce tube), la fonction **read** se termine sans rien lire et retourne la valeur **0**.

L'opération ne peut plus être **atomique** (indivisible) si *nb* > PIPE_BUF (taille du tube).
Dans ce cas le processus doit attendre que le tube soit plein pour lire une première tranche de PIPE_BUF octets puis attendre à nouveau pour lire les octets restants, etc …

La situation se complique s'il y a plusieurs processus qui veulent lire dans le tube car il peut alors y avoir **entrelacement** des données pour ces différents processus. Il faut donc absolument éviter de se placer dans cette situation.

Remarque :

Dans la plupart des cas, il est préférable d'utiliser les fonctions d'E/S de la bibliothèque C (stdio.h) plutôt que les appels systèmes **read()** et **write()**.
Pour ce faire, il faut transformer les descripteurs *descFic[0]*et *descFic[1]* en pointeurs de fichiers du type **FILE*** en utilisant la fonction C **fdopen()** .

Exemple :

```
……..
FILE * tubeTete, * tubeQueue ;
……..
tubeTete = fdopen ( descFic[0], "r" ) ;
tubeQueue = fdopen ( descFic[1], "w" ) ;
……..
```

Attention : dans ce cas, après toute écriture dans le fichier *tubeQueue* , il est impératif de forcer le vidage du tampon de sortie avec la fonction **fflush( )**.

Exemple :

```
……..
fprintf ( tubeQueue, "%s …." , ….) ;
fflush ( tubeQueue);
……..
```

2.3-Fermeture d'un « pipe »

L'appel système **close()** permet à un processus de fermer un fichier quelconque (ordinaire, socket, tube local, FIFO, ….) :

#include <unistd.h>

int **close**(int *dF*) ;

dF est le descripteur associé au fichier.

La fonction retourne **–1** en cas d'erreur et **0** sinon.
Pour un tube local, l'utilisation est la suivante :

close(*descFic[0]*) ; pour fermer le fichier tube ouvert en lecture.

close(*descFic[1]*) ; pour fermer le fichier tube ouvert en écriture.

On verra au paragraphe suivant, la nécessité de fermer l'une des deux extrémités d'un tube pour éviter des situations d'**interblocage**.

2.4-Synchronisation des accès à un « pipe »

Malgré le mécanisme de synchronisation mis en œuvre par le système pour gérer les accès à un tube, il existe 2 situations dans lesquelles un processus peut se trouver bloqué indéfiniment sans que le système puisse le débloquer. La seule issue est de « tuer » le processus. Cette situation dans laquelle s'est mise le processus est appelée un « verrou mortel ».

1er cas :

Dans le processus père la fonction **pipe** (cf § 2.1) crée 2 descripteurs, un pour lire dans le tube, l'autre pour écrire dans le tube. Après un **fork**, chaque processus fils hérite de ces 2 descripteurs et peut aussi lire ou écrire dans le tube.
Si le processus père ou un processus fils est bloqué en lecture dans un **read** (tube vide) et qu'il n'existe aucun autre processus susceptible d'écrire dans le tube (processus ayant un descripteur ouvert en écriture sur ce tube), la fonction **read** ne se terminera pas car pour le système il reste encore un processus ayant un descripteur ouvert en écriture sur ce tube (le processus bloqué).
Comme le processus est bloqué, il ne peut bien sûr pas écrire dans le tube pour se débloquer !
Pour éviter ce verrou mortel, il faut **impérativement** que le processus **ferme le fichier tube ouvert en écriture** avant de commencer à lire dans le tube.

2eme cas :

C'est la situation inverse où le processus père ou un processus fils est bloqué en écriture dans un **write** et qu'il n'existe aucun autre processus susceptible de lire dans le tube (processus ayant un descripteur ouvert en lecture sur ce tube).
Le noyau n'enverra pas le signal **SIGPIPE** au processus bloqué car pour le système il reste encore un processus ayant un descripteur ouvert en lecture sur ce tube (le processus qui est bloqué).
Comme le processus est bloqué, il ne peut bien sûr pas lire dans le tube pour se débloquer !
Pour éviter ce verrou mortel, il faut **impérativement** que le processus **ferme le fichier tube ouvert en lecture** avant de commencer à écrire dans le tube.

3- Les tubes nommés (FIFO)

3.1-Création d'un « FIFO »

La création d'un tube nommé peut être faite par l'appel système **mknod()** mais il est préférable de le faire avec la fonction **mkfifo()** :

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

int **mkfifo**(const char * *nomFIFO* , mode-t *droits*) ;

nomFIFO est le nom du fichier FIFO préfixé par un chemin d'accès si le FIFO doit être créé ailleurs que dans le répertoire courant.
droits est la constante représentant les 9 bits des droits d'accès au FIFO. Seuls les droits **r** et **w** sont utilisés (un FIFO n'est pas un fichier exécutable !)

La fonction retourne **–1** en cas d'erreur et **0** si le FIFO est créé.

Exemple : **création d'un tube nommé (FIFO)**

```
/* creation d'un FIFO avec les droits r et w pour les seuls processus appartenant au proprietaire du FIFO */
err = mkfifo("data.FIFO", 0600 ) ;
if ( err == -1 )
    { perror("echec mkfifo" ) ; exit(1) ; }
```

3.1-Ouverture d'un « FIFO »

Comme pour un fichier ordinaire, un processus qui veut accéder à un FIFO, doit l'ouvrir avec l'appel système **open()** :

```
#include <fcntl.h>
```

int **open**(const char * *nomFIFO* , int *mode*) ;

nomFIFO est le nom du fichier FIFO préfixé par un chemin d'accès si le FIFO est ailleurs que dans le répertoire courant.
mode est la constante symbolique représentant le mode d'accès au FIFO :

- la constante **O_RDONLY** spécifie l'ouverture en **lecture**.
- la constante **O_WRONLY** spécifie l'ouverture en **écriture**.

La fonction retourne **–1** en cas d'erreur (le FIFO n'existe pas, le processus n'a pas les permissions pour accéder au FIFO, ….).
Sinon la fonction retourne un entier qui est le descripteur du fichier FIFO qui sera utilisé ultérieurement dans les opérations **read**, **write**, **close** ou **fdopen**

Remarque :

Lorsqu'un processus ouvre un FIFO en **lecture**, il peut rester **bloqué** tant qu'aucun autre processus n'aura ouvert ce FIFO en **écriture** et vice-versa.

Exemple : **ouverture d'un FIFO en lecture**

```
……..
int descFIFO ;
……..
/* ouverture d'un FIFO en lecture */
/* le nom du FIFO doit etre  connu par le processus */
descFIFO = open("data.FIFO" , O_RDONLY ) ;
if ( descFIFO  == -1 )
    { perror("echec open" ) ; exit(1) ; }
……..
```

3.2- Lecture-écriture dans un « FIFO »

Une fois que le fichier FIFO a été ouvert, les opérations de lecture ou d'écriture se font de la même manière que pour un tube local (cf § 4.2.2).

3.3-Fermeture d'un « FIFO »

La fermeture d'un fichier FIFO se fait par l'appel système **close()** :

close(*descFIFO*) ; *descFIFO* est le descripteur du fichier FIFO retourné par la fonction **open** lors de l'ouverture du fichier

3.4-Suppression d'un « FIFO »

Comme pour un fichier ordinaire, la suppression d'un FIFO se fait par l'appel système **unlink()** :

#include <unistd.h>

int **unlink**(const char * *nomFIFO*) ;

nomFIFO est le nom du fichier FIFO préfixé par un chemin d'accès si le FIFO est ailleurs que dans le répertoire courant.

La fonction retourne **–1** en cas d'erreur et **0** sinon.

Remarque :

La fonction **unlink** supprime en fait le nom du fichier FIFO (lien) dans le répertoire qui le contient, si le processus a les permissions pour le faire.
Le fichier FIFO n'est physiquement supprimé que lorsque son dernier lien est supprimé.