

Nous avons vu dans les chapitres précédents deux moyens pour faire communiquer des processus sous UNIX : les signaux et les sémaphores. Les sémaphores, ainsi que les tubes, les files de messages et les segments de mémoire partagée constituent les **IPC** (Inter Process Communications), apparus avec UNIX System V. Le fichier `/usr/include/sys/ipc.h` contient la définition des objets et des noms symboliques utilisés par les primitives des IPC.

Qu'il s'agisse des sémaphores, des tubes, des files de messages ou des segments de mémoire partagée, il s'agit d'objets **externes** au SGF. Pour chacune de ces catégories, il existe une primitive (...get) d'ouverture/création (cf. ouverture des fichiers). Elle donne au processus une identification de l'objet interne (cf. descripteurs de fichiers).

## 1. DEFINITION ET CARACTERISTIQUES

Les tubes (pipes) constituent le mécanisme fondamental de communication entre processus sous UNIX. Ce sont des **files d'octets**. Il est conseillé de les utiliser de façon **unidirectionnelle**. Si l'on veut une communication bidirectionnelle entre deux processus, il faut ouvrir deux tubes, chacun étant utilisé dans un sens. Un tube est implémenté de façon proche d'un fichier : il possède un i-node, mais ne possède pas de nom dans le système ; son compteur de liens est nul puisque aucun répertoire n'a de lien avec lui.

Les tubes sont accessibles au niveau du shell pour rediriger la sortie d'une commande sur l'entrée d'une autre (symbole `|`).

Comme un tube ne possède pas de nom, il n'existe que le temps de l'exécution du processus qui le crée. Plusieurs processus peuvent lire ou écrire dans un même tube, sans qu'on puisse différencier l'origine des informations en sortie. La capacité d'un tube est limitée (PIPE\_BUF octets).

Les processus communiquant par un tube doivent avoir **un lien de parenté** (par exemple descendance d'un ancêtre commun ayant créé ce tube : problème classique de l'héritage des descripteurs de fichiers par un fils).

## 2. CREATION ET UTILISATION D'UN TUBE

La création d'un tube correspond à celle de deux descripteurs de fichiers, l'un permettant d'écrire dans le tube et l'autre d'y lire par les opérations classiques **read** et **write** de lecture/écriture dans un fichier. Pour cela, on utilise la fonction prototypée par :

**int pipe (int \*p\_desc)**

où `p_desc[0]` désigne le n° du descripteur par lequel on lit dans le tube et `p_desc [1]` désigne le n° du descripteur par lequel on écrit dans le tube.

La fonction retourne 0 si la création s'est bien passée et - 1 sinon.

La fonction C prototypée par :

**FILE \* fdopen (int desc, char \*mode)**

permet d'associer un pointeur sur FILE au tube ouvert ; mode doit être conforme à l'ouverture déjà réalisée.

### **3. SECURITES APORTEES PAR UNIX**

Un processus peut fermer un tube, mais alors il ne pourra plus le rouvrir. Lorsque tous les descripteurs d'écriture sont fermés, une fin de fichier est perçue sur le descripteur de lecture : read retourne 0.

Un processus qui tente d'écrire dans un tube plein est suspendu jusqu'à ce que de la place se libère. On peut éviter ce blocage en positionnant le drapeau O\_NDELAY (mais l'écriture peut n'avoir pas lieu). Lorsque read lit dans un tube insuffisamment rempli, il retourne le nombre d'octets lus.

Attention : les primitives d'accès direct sont interdites; on ne peut pas relire les informations d'un tube car **la lecture dans un tube est destructive**.

Dans le cas où tous les descripteurs associés aux processus susceptibles de lire dans un tube sont fermés, un processus qui tente de lire reçoit le signal 13 SIGPIPE et se trouve interrompu s'il ne traite pas ce signal.

#### **3.1 Ecriture dans un tube fermé en lecture**

```
#include <errno.h>
#include <signal.h>
/*****
void it_sigpipe ()
{
    printf ("On a reçu le signal SIGPIPE\n");
}
*****/
void main ()
{
    int p_desc [2];
    signal (SIGPIPE, it_sigpipe);
    pipe (p_desc);
    close (p_desc [0]); /* fermeture du tube en lecture */
    if (write (p_desc [1], "0", 1) == -1)
        perror ("Erreur write : ");
}
```

Tester l'exécution de ce programme et expliquez son fonctionnement.

Autres exemples intéressants : Jean-Pierre BRAQUELAIRE, "Méthodologie de la programmation en Langage C", Masson, 1993, p. 344-345, 456-457

### 3.2 Lecture dans un tube fermé en écriture.

```
#include <errno.h>
#include <signal.h>
/*****/
void main ()
{
int i , ret , p_desc [2];
char c;
pipe (p_desc);
write (p_desc[1], "AB", 2);
close (p_desc [1]);
for (i = 1; i <= 3; i++)
    { ret = read (p_desc [0], &c, 1);
      if (ret == 1) printf ("valeur lue : %c\n", c );
      else perror ("lecture impossible : ");
    }
}
```

Tester l'exécution de ce programme et expliquez son fonctionnement

## 4. EXEMPLES DE PROGRAMMES.

### 4.1 Communication père-fils

Un fils hérite des tubes créés par le père et de leurs descripteurs.

```
/* exemple 1 */
#include <stdio.h>
#include <errno.h>
#define TAILLE 30      /* par exemple */
main ()
{
    char envoi [TAILLE], reception [TAILLE]; int p [2], i, pid;
    strcpy (envoi, "texte transmis au tube");
    if (pipe (p) < 0)
    {
        perror ("erreur de création du tube ");
        exit (1);
    }
    if ((pid = fork()) == -1)
    {
        perror ("erreur au fork ");
        exit (2);
    }
    if (pid > 0) /* le père écrit dans le tube */
    {
        for (i=0 ; i<5; i++) write (p[1], envoi, TAILLE);
        wait (0);
    }
    if (pid == 0) /* le fils lit dans le tube */
    {
        for (i=0 ; i<5 ; i++) read (p[0], reception, TAILLE);
        printf (" --> %s\n", reception);
        exit (0);
    }
}
```

Testez l'exécution de ce programme.

```

/* exemple 2 */
#include <stdio.h>
#include <errno.h>
#define NMAX 10 /* par exemple */
/*****/
void main ()
{
int pid , p_desc [2];
char c;
if (pipe (p_desc))
    { perror ("erreur en creation du tube ");
      exit (1);
    }
if ((pid = fork ()) == -1)
    { perror ("echec du fork ");
      exit (2);
    }
if (pid == 0)
    { char t [NMAX];
      int i=0;
      close (p_desc [1]);
      while (read (p_desc [0], &c, 1))
          if (i < NMAX) t[i++] = c;
      t [( i == NMAX) ? NMAX-1 : i ] = 0;
      printf ("t : %s\n", t);
    }
else
    { close p_desc [0];
      while ((c = getchar ()) != EOF)
          if (c >= 'a' && c <= 'z') write (p_desc [1], &c, 1);
      close p_desc [1];
      wait (0);
    }
}

/* exemple 3 */
#include <stdio.h>
#include <errno.h>
#define NMAX 10 /* par exemple */
/*****/
void main ()
{
int pid , p_desc [2] , m;
FILE *in, *out;
if (pipe (p_desc))
    { perror ("erreur en création du tube ");
      exit (1);
    }
out = fdopen (p_desc [0], "r");
in = fdopen (p_desc [1], "w");
if ((pid = fork ()) == -1)
    { perror ("echec du fork ");
      exit (2);
    }
if (pid == 0)
    { int t[NMAX], i = 0;
      fclose (in);
      while (fscanf (out, "%d", &m) != EOF)
          if (i < NMAX) t[i++] = m;
    }
}

```

```

        t ( [ i == NMAX) ? NMAX-1 : i ] = 0;
        for (m=0 ; m < i ; m++) printf ("%d \n", m);
    }
    ..... cf. exemple 2 .....
}

```

Testez l'exécution de ces programmes et expliquez leur fonctionnement.

Autres exemples intéressants : Braquelaire p. 465-467.

## **4.2 Héritage des descripteurs lors d'un fork**

```

#include <errno.h>
#include <stdio.h>
/*****
void code_fil (int numero)
{
    int fd, nread;
    char texte [100];
    fd = numero;
    printf ("le descripteur est : %d\n", fd);
    switch (nread = read (fd, texte, sizeof (texte)))
    { case -1 : perror (" erreur read : ");
      break;
      case 0 : perror ("erreur EOF");
      break;
      default : printf ("on a lu %d caractères : %s\n", nread, texte);
    }
}
*****/
void main ()
{
    int p_desc [2],
    char chaine [10];
    if (pipe (p_desc ))
        { perror ("erreur en creation du tube ");
          exit (1);
        }
    switch (fork())
    { case -1 : perror (" erreur fork : ");
      break;
      case 0 :if (close (p_desc [1]) == -1) perror ("erreur close : ");
              code_fil (p_desc [0]);
              exit (0);
    }
    close (p_desc [0]);
    if (write (p_desc [1], "hello", 6) == -1) perror ("erreur write :");
}

```

Testez l'exécution de ce programme et expliquez son fonctionnement.

## **4.3 Les redirections d'E/S standards : exemple de who | wc**

```

#include <errno.h>
#include <stdio.h>
/*****

```

```

void main ()
{
int p [2], n;
if (pipe (p ))
    { perror ("erreur en création du tube ");
      exit (1);
    }
if ( n= fork () == -1)
    { perror ("erreur avec fork : ");
      exit (2);
    }
if (n == 0)
    /* processus correspondant à who */
    { close (1);          /* on ferme la sortie standard */
      dup (p [1]);        /* fournit un descripteur ayant les mêmes caractéristiques
physiques que p[1], et ayant la valeur la plus faible possible. Ici, nécessairement : 1 */
      close (p [1]); /* la sortie standard est ainsi redirigée sur l'entrée du tube */
      close (p [0]);
      execlp ("who", "who", 0);
      /* on recouvre le processus par le texte de who */
    }
else
    /* processus correspondant à wc */
    { close (0);          /* on ferme l'entrée standard */
      dup (p [0]);        /* fournit un descripteur ayant les mêmes caractéristiques
physiques que p[0], et ayant la valeur la plus faible possible. Ici, nécessairement : 0 */
      close (p [0]); /* l'entrée standard est ainsi redirigée sur l'entrée du tube */
      close (p [1]);
      execlp ("wc", "wc", 0); /* on recouvre le processus par le texte de wc */
    }
}

```

Autre exemple intéressant : Braquelair p. 472-478.

#### **4.4 Communication entre deux fils**

```

#include <errno.h>
#include <stdio.h>
/*****
void main ()
{
int pid1, pid2, ret, p_desc [2];
char *arg1, *arg2;
ret = pipe (p_desc);
sprintf (arg1, "%d", p [0]); /* conversion de p[0] en chaîne de caractères */
sprintf (arg2, "%d", p [1]); /* conversion de p[1] en chaîne de caractères */
pid1 = fork ();
if (pid1 == 0) execl ("fils1", "fils1", arg1, 0);
pid2 = fork ();
if (pid2 == 0) execl ("fils2", "fils2", arg2, 0);
.....
/* suite du père */
.....
}

```

La communication est bien : fils1 -> fils2. Avec 3 tubes, on réaliserait une communication circulaire : père -> fils1 -> fils2 -> père.

tube p1 : père -> fils1 (fils1 doit connaître p1[0])  
tube p2 : fils1 -> fils2 (fils 1 doit connaître p2[1], fils 2 doit connaître p2 [0])  
tube p3 : fils2 -> père (fils 2 doit connaître p3 [1])

Autre exemple intéressant : Braquelaire p. 467-472.

## 5. LES TUBES NOMMES

Un tube nommé possède un nom dans le système. S'il n'est pas détruit, **il persiste dans le système après la fin du processus** qui l'a créé. Tout processus possédant les autorisations appropriées peut l'ouvrir en lecture ou en écriture. Un tube nommé conserve toutes les propriétés d'un tube : taille limitée, lecture destructive, lecture/écriture réalisées en mode FIFO. Un tube nommé permet à des processus sans lien de parenté dans le système de communiquer.

Pour créer un tube nommé, on utilise la fonction : **int mknod (char \*ref, int mode)** qui retourne 0 si elle a pu créer un i-node dont la première référence est pointée par ref, avec les protections mode); en cas d'échec, le retour est -1.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
/*****
void main ()
{
    printf ("on va créer un tube de nom fifo1 et un tube de      nom fifo2\n");
    if (mknod ("fifo1", S_FIFO | 0666) == -1) /* S_FIFO obligatoire */
        { perror ("erreur de création fifo 1 ");
          exit (1);
        }
    if (mknod ("fifo2", S_FIFO | 0666) == -1) /* S_FIFO obligatoire */
        { perror ("erreur de création fifo2 ");
          exit (1);
        }
    sleep (30);
    printf ("on efface le tube fifo1\n");
    unlink ("fifo1");
}
```

Lancez ce programme en arrière-plan et testez avec **ls** les créations et la suppression, ainsi que la présence de fifo2 après la fin du processus. Dans le catalogue, les tubes nommés apparaissent avec **p** en tête des droits d'accès.

Pour utiliser un tube nommé créé, il est nécessaire de l'ouvrir en lecture ou en écriture avec open.

Des précautions d'ouverture et d'utilisation sont à connaître. On se reportera avec profit à Jean-Marie RIFFLET, "La programmation sous UNIX", Mac-Graw-Hill, 1992, pages 263 à 267.

```
/* exemple 1 : l'exécutable sera nommé ex1 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
main (int n, char ** v)
{
    int p;
    if (mknod ( v[1], S_IFIFO | 0666) == -1)
        {
            perror ("création impossible ");
            exit (1);
        }
}
```

```

    }
    if ((p = open (v[1], O_WRONLY, 0)) == -1)
    {
        perror ("ouverture impossible ");
        exit (2);
    }
    write (p, "ABCDEFGH", 8);
}

/* l'exécutable sera nommé ex2 */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main (int n, char ** v)
{
    int p;
    char c;
    if ((p = open (v[1], O_RDONLY, 0)) == -1)
    {
        perror ("ouverture impossible ");
        exit (2);
    }
    read (p, &c, 1);
    printf ("%c\n", c);
}

```

On lance :

```

$ ex1 fifo&
(il s'affiche ..... )
$ ex2 fifo
A

```