Chapitre 5

Communication interprocessus

Les processus coopèrent souvent pour traiter un même problème. Ces processus s'exécutent en parallèle sur un même ordinateur (monoprocesseur ou multiprocesseurs) ou bien sur des ordinateurs différents. Ils doivent alors s'échanger des informations (**communication interprocessus**). Il existe plusieurs moyens de communication interprocessus. Nous pouvons citer, entre autres, les variables communes, les fichiers communs, les signaux, les messages et les tubes de communication.

5.1 Variables et fichiers communs

Les processus partagent un ensemble de données communes qui sont soit en mémoire (variables ou segments de données) soit sur disque (fichiers). Chaque processus peut accéder en lecture ou en écriture à cet ensemble de données appelé **espace de données commun**. Des problèmes peuvent se poser lorsque plusieurs processus essayent d'accéder en même temps à un espace commun. Si deux processus ou plus lisent ou écrivent des données partagées, le résultat dépend de l'ordonnancement des processus. C'est ce qu'on appelle des *accès concurrents*.

Supposons deux processus qui partagent une variable `v`. Le premier processus incrémente de `1` la valeur de `v` alors que l'autre la décrémente de `1`. La valeur initiale de `v=1`.

```
P1 : v=++;
P2 : v=--;
```

Les instructions correspondantes en assembleur sont les suivantes :

```
P1 :   movl $v, %eax       # load v
       movw (%eax), %bx
       incw %bx            # bx++;
       movw %bx, (%eax)    # store v

P2 :   movl $v, %eax       # load v
       movw (%eax), %bx
       decw %bx            # bx--;
       movw %bx, (%eax)    # store v
```

Supposons que les processus `P1` et `P2` s'exécutent en temps partagé.

- Le processus `P1` est en cours d'exécution.
- Il est suspendu juste après l'exécution de l'instruction `incw %bx`.
- Le processus `P2` est élu et exécute les instructions `load v` et `decw %bx` et `store v`. Le processus P2 se termine avec `v = 0`.
- Le processus `P1` est ensuite élu et exécute l'instruction `store v` et donc `v=2`.

Pour éviter de tels problèmes, il est nécessaire de contrôler les accès concurrents à des objets communs. Nous verrons plus tard des solutions qui permettent de résoudre les conflits d'accès à des objets communs. Dans ce chapitre, nous allons présenter deux moyens de communication interprocessus offerts par le système Unix/Linux : les signaux et les envois de messages par le biais de tubes.

5.2 Les signaux

Les **interruptions logicielles** ou **signaux** sont utilisées par le système d'exploitation pour aviser les processus utilisateurs de l'occurrence d'un événement important. De nombreuses erreurs détectées par le matériel, comme l'exécution d'une instruction non autorisée (une division par 0, par exemple) ou l'emploi d'une adresse non valide, sont converties en signaux qui sont envoyés au processus fautif. Ce mécanisme permet à un processus de réagir à cet événement sans être obligé d'en tester en permanence l'arrivée.

Les processus peuvent indiquer au système ce qui doit se passer à la réception d'un signal. On peut ainsi ignorer le signal, ou bien le prendre en compte, ou encore laisser le système d'exploitation appliquer le comportement par défaut, qui en général consiste à tuer le processus. Certains signaux ne peuvent être ni ignorés, ni capturés. Si un processus choisit de prendre en compte les signaux qu'il reçoit, il doit alors spécifier la procédure de gestion de signal. Quand un signal arrive, la procédure associée est

exécutée. A la fin de l'exécution de la procédure, le processus s'exécute à partir de l'instruction qui suit celle durant laquelle l'interruption a eu lieu.

En résumé, les signaux sont utilisés pour établir une communication minimale entre processus, une communication avec le monde extérieur et faire la gestion des erreurs. Ainsi, un processus peut :

- **Ignorer le signal.**
- **Appeler une routine de traitement fournie par le noyau.** Cette procédure provoque normalement la mort du processus. Dans certains cas, elle provoque la création d'un fichier `core`, qui contient le contexte du processus avant de recevoir le signal. Ces fichiers peuvent être examinés à l'aide d'un débogueur.
- **Appeler une procédure spécifique créée par le programmeur.** Tous les signaux ne permettent pas ce type d'action.

Tous les signaux ont une routine de service, ou une action par défaut. Cette action peut être du type :

- A : terminaison du processus
- B : ignorer le signal
- C : Créer un fichier `core`
- D : Stopper le processus
- E : La procédure de service ne peut pas être modifiée
- F : Le signal ne peut pas être ignoré

Chaque signal est identifié au moyen d'un numéro entier positif. La table 5.1 décrit les signaux les plus importants. Dans l'**Annexe ??** le lecteur trouvera des tables plus complètes des signaux sur Linux et Unix.

Signal	Num	Type	Description
SIGKILL	9	AEF	Termine l'exécution du processus
SIGSTOP	19	DEF	Stoppe le processus
SIGCONT	18		Continue l'exécution du processus
SIGCHILD	17	B	Un processus fils a fini
SIGUSR1	10	A	Signal d'utilisateur
SIGUSR2	10	A	Signal d'utilisateur

TAB. 5.1 – Quelques signaux Unix.

5.2.1 Services de signaux

Dans le cas du système Unix/Linux, un processus utilisateur peut également envoyer un signal à un autre processus. Les deux processus appar-

tiennent au même propriétaire, ou bien le processus émetteur du signal est le super-utilisateur. Les services Posix de gestion des signaux utilisent la bibliothèque `<signal.h>`.

`kill()`

L'envoi d'un signal peut se faire avec l'appel système `kill()` :

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

`kill()` envoi le signal numéro `signal` à un processus identifié par son `pid`. En cas d'erreur il retourne -1, et 0 autrement. `pid` peut prendre comme valeurs :

- Si `pid` > 0 : processus `pid`.
- Si `pid` = 0 : groupe de l'émetteur.
- Si `pid` =-1 : tous les processus (seulement root peut le faire).
- Si `pid` < 0 : au groupe `gid` = |`pid`|

► **Exemple 1.** Le programme `signaux-0.c` montre l'utilisation du masquage des signaux.

Listing 5.1 – signaux-0.c

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

void sighandler(int signum);

int main(void)
{
 char buffer[256];

 if(signal(SIGTERM,&sighandler) == SIG_ERR )
 {
    printf("Ne peut pas manipuler le signal\n");
    exit(1);
 }
 while(1)
 {
   fgets(buffer,sizeof(buffer), stdin);
   printf("Input: %s",buffer);
 }
 return 0;
```

```
}

void sighandler(int signum)
{
  printf("Masquage du signal SIGTERM\n");
}
```

Nous allons montrer l'exécution de `signaux-0.c` dans deux terminaux, afin de montrer le masquage de signaux et d'essayer de tuer le processus. D'abord, exécution sur un terminal :

```
leibnitz> gcc -o signal-0 signal-0.c
leibnitz> signal-0
HOLA
Input: HOLA
KJHKHBNFYNBknbjhhgdt6565
Input: KJHKHBNFYNBknbjhhgdt6565
ls -l
Input: ls -l
who
Input: who
...
```

Dans un autre terminal (`pts/1`), on peut essayer de tuer le processus 22096. On dispose pour cela de la commande `kill`. Si on lui passe en argument seulement le PID du processsus, elle enverra le signal SIGTERM à ce processus. Autrement, on peut identifier le signal à envoyer, en le précédent du tiret :

```
leibnitz> ps -u jmtorres
 PID TTY          TIME CMD
20652 pts/0    00:00:00 tcsh
21964 pts/0    00:00:00 a.out
21966 pts/1    00:00:00 tcsh
22096 pts/0    00:00:00 signal-0
22098 pts/1    00:00:00 ps
leibnitz> kill 22096
leibnitz> kill 22096
leibnitz> kill -SIGKILL 22096
leibnitz>
```

Et dans le premier terminal (`pts/0`) nous continuerons à voir des messages, jusqu'à ce que le signal non ignorable SIGKILL arrive :

```
...
```

```
Who am i
Input: who am i
kill
Input: kill
Masquage du signal SIGTERM
Masquage du signal SIGTERM
Killed
leibnitz>
```

À noter que les commandes `Ctrl-C` ou bien `Ctrl-Z` auraient aussi tué le processus.

► **Exemple 2.** Le programme `signaux-1.cpp` illustre l'utilisation des signaux `SIGUSR1` et `SIGUSR2` :

Listing 5.2 – signaux-1.cpp

```
#include <signal.h>
#include <iostream.h>
#include <unistd.h>
#include <wait.h>

static void action(int sig);
void main()
{
 int i, pid, etat;
 // Specification de l'action du signal
 if(signal(SIGUSR1, action) == SIG_ERR)
    cout<<"Erreur de traitement du code de l'action" <<endl;
 if(signal(SIGUSR2, action) == SIG_ERR)
    cout<<"Erreur de traitement du code de l'action" <<endl;
 if((pid = fork()) == 0)
 {
        kill(getppid(), SIGUSR1); // Envoyer signal au parent.
        for(;;)
         pause();                 // Mise en attente d'un signal
 }
 else
 {
        kill(pid, SIGUSR2);      // Envoyer un signal a l'enfant
        cout<<"Parent : terminaison du fils"<<endl;
        kill(pid, SIGTERM);      // Signal de terminaison a l'enfant
        wait(&etat);             // attendre la fin de l'enfant
        cout <<"Parent : fils termine" <<endl;
 }
}
// Fonction definissant le code de
```

```
// l'action a la reception d'un signal
static void action(int sig)
{
  switch (sig)
  {
     case SIGUSR1 :
          cout <<"Parent : signal SIGUSR1 recu" << endl;
        break;
     case SIGUSR2 :
          cout <<"Fils : signal SIGUSR2 recu" << endl;
          break;
     default : break;
  }
}
```

Une exécution de `signaux-1.cpp` est montrée à continuation :

```
leibnitz> g++ -o signaux-1 signaux-1.cpp
leibnitz> signaux-1
Parent : terminaison du fils
Fils : signal SIGUSR2 reçu
Parent : terminaison du fils
Parent : signal SIGUSR1 reçu
Parent : fils terminé
leibnitz>
```

Cependant, si l'on essai à nouveau une autre exécution, la sortie probablement, sera tout à fait différente :

```
leibnitz> signaux-1
Parent : terminaison du fils
Parent : fils terminé
leibnitz>
```

Pour quoi ce comportement étrange et comment le résoudre ?

`sleep()` et `pause()`

Les appel système `sleep()` et `pause()` peuvent être utiles lorsqu'on travaille avec des signaux.

```
#include <unistd.h>
int pause(void);
int sleep(unsigned int seconds);
```

Un processus peut se mettre en veille et attendre l'arrivée d'un signal particulier pour exécuter le gestionnaire, ou bien l'émetteur utilise `sleep()` pour se synchroniser avec le récepteur.

`pause()` permet que le processus passe à l'état suspendu. Il va quitter cet état au moment de recevoir un signal. Après avoir reçu le signal, le processus recevra un traitement adéquat associé au signal et continuera son exécution à l'instruction qui suit `pause()`. Il retourne toujours la valeur -1. Cet appel fournit une alternative à l'attente active. C'est-à-dire qu'au lieu d'exécuter une boucle inutile en attente d'un signal, le processus bloquera jusqu'à ce qu'un signal lui soit transmis.

▶ **Exemple 3.** Le programme `sigint.c` montre une utilisation peu efficace du signal `SIGINT`, car on fait une attente active `while(num_signal<5);`

Listing 5.3 – sigint.c

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

int num_signal = 0;

void traiter_SIGINT()
{
  num_signal++;
  printf("Signal CTRL-C capture!\n");
}

int main()
{
  if (signal(SIGINT, traiter_SIGINT) == SIG_ERR)
 {
        printf("Erreur dans le gestionnaire\n");
        exit(1);
  }
  while (num_signal < 5); // attente active
  printf("\n%d  signaux SIGINT sont arrives\n", num_signal);
  return 0;
}
```

Par contre, le programme `sigint-2.c` utilise `pause()` pour éviter ce gaspillage de CPU :

Listing 5.4 – sigint-2.c

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

```c
int num_signal = 0;

void traiter_SIGINT()
{
  num_signal++;
  printf("Signal CTRL-C capture!\n");
}

int main()
{
  if (signal(SIGINT, traiter_SIGINT) == SIG_ERR)
 {
        printf("Erreur dans le gestionnaire\n");
        exit(1);
  }
  while (num_signal < 5) pause(); // attente NON active
  printf("\n%d  signaux SIGINT sont arrives\n", num_signal);
  return 0;
}
```

/

▶ **Exemple 4.** Le programme `signaux-2.c` montre l'utilisation des signaux ignorables `SIGINT` et `SIGQUIT`, ainsi que le signal non ignorable `SIGKILL`:

Listing 5.5 – signaux-2.c

```c
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

// Declaration des fonctions a associer aux signaux
void trait_sigint()
{
  printf("bien recu SIGINT, mais je vais l'ignorer\n");
}

void trait_sigquit()
{
  printf("bien recu SIGQUIT, mais je vais l'ignorer\n");
}

int main()
{
  int pid;

  pid = fork();
```

```
  if(pid==0) //processus fils
  {    // Associer au signal SIGINT le traitement trait_sigint
     signal(SIGINT,  trait_sigint);
       // Associer au signal SIGQUIT le traitement trait_sigquit
     signal(SIGQUIT, trait_sigquit);
     printf("je suis toujours en vie\n");
     sleep(20);
     printf("premier reveil du fils\n");
     sleep(120);
     printf("second reveil du fils\n");
     sleep(500);
     printf("troisieme reveil du fils\n");
  }
  else
  {
     sleep(1);
     // emission du signal SIGINT (interrupt)
     kill(pid, SIGINT);
     sleep(2);
     // emission du signal SIGQUIT (quit)
     kill(pid, SIGQUIT);
     sleep(5);
     // emission du signal SIGKILL (kill)
     // ce signal ne peut etre ni capture
     // ni ignore par le destinataire
     kill(pid,SIGKILL);
  }
  return 0;
}
```

L'exécution de `signaux-2.c`:

```
leibnitz> gcc -o signaux-2 signaux-2.c
leibnitz> signaux-2
 je suis toujours en vie
 bien reçu SIGINT, mais je vais l'ignorer
premier réveil du fils
bien reçu SIGQUIT, mais je vais l'ignorer
second réveil du fils
leibnitz>
```

▶ **Exemple 5.** Le programme `ping.c` effectue un ping-pong des signaux entre un processus et son fils. Le signal `SIGUSR1` est envoyé par le fils au père et `SIGUSR2` est envoyé du père au fils. Le premier signal est envoyé par le père.

Listing 5.6 – ping.c

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void handlerSignal();
void sendSignal(int pid, int sig);

int Proc1, cptr, limite;
main(int argc, char **argv)
{
   if (argv[1]==0)
       {
         printf("format : ping n\n");
         exit(1);
       }
   limite=atoi(argv[1]);
   cptr=0;
   Proc1=fork();
   switch (Proc1)
       {
       case 0 :
                signal(SIGUSR2, handlerSignal);
                printf("Fils => gestionnaire installe, PID=%d\n", getpid());
                pause();
                while(cptr<limite-1)
                {
                        cptr++;
                        sendSignal(getppid(), SIGUSR1);
                        pause();
                }
                cptr++;
                sendSignal(getppid(), SIGUSR1);
                exit(0);
                break;
       default :
                signal(SIGUSR1, handlerSignal);
                printf("Pere => gestionnaire installe, PID=%d\n", getpid());
                cptr++;
                sendSignal(Proc1, SIGUSR2);
                pause();
                while(cptr<limite)
                {
                        cptr++;
                        sendSignal(Proc1, SIGUSR2);
                        pause();
```

```
                }
                break;
        }
}

void handlerSignal(int sig)
{
   printf("\t\t[%d] gestionnaire %d => signal capte\n", getpid(), sig);
   fflush(stdout);
}

void sendSignal(int pid, int sig)
{
 sleep(1);
 if (kill(pid, sig)==-1)
 {
       printf("ERREUR kill PID=%d\n", pid);
       fflush(stdout);
       exit(1);
 }
 printf("#%d [%d] signal %d envoye a %d\n", cptr, getpid(), sig, pid);
}
```

Exemple d'exécution de `ping.c` avec l'envoi de deux messages :

```
leibnitz> gcc -o ping ping.c
leibnitz> ping 2
Père => signal envoyé, PID=4825
Fils => signal envoyé, PID=4826
#1 [4825] signal 12 envoyé à 4826
                [4826] gestionnaire 12 => signal capté
#1 [4826] signal 10 envoyé à 4825
                [4825] gestionnaire 10 => signal capté
#2 [4825] signal 12 envoyé à 4826
                [4826] gestionnaire 12 => signal capté
#2 [4826] signal 10 envoyé à 4825
                [4825] gestionnaire 10 => signal capté
leibnitz>
```

5.3 Les messages

Un autre mécanisme de communication entre processus est l'**échange de messages**. Chaque message véhicule des données. Un processus peut envoyer un message à un autre processus se trouvant sur la même machine ou sur des machines différentes. Unix offre plusieurs mécanismes de

FIG. 5.1 – Processus en paralèlle reliés par un tube de communication.

communication pour l'envoie de messages : les **tubes de communication** sans nom, nommés et les **sockets**.

5.3.1 Tubes de communication

Les **tubes de communication** ou *pipes* permettent à deux ou plusieurs processus d'échanger des informations. On distingue deux types de tubes : les tubes sans nom (*unamed pipe*) et les tubes nommés (*named pipe*).

5.3.2 Tubes sans nom

Les **tubes sans nom** sont des liaisons unidirectionnelles de communication. La taille maximale d'un tube sans nom varie d'une version à une autre d'Unix, mais elle est approximativement égale à 4 Ko. Les tubes sans nom peuvent être créés par le shell ou par l'appel système `pipe()`.

Les tubes sans nom du shell sont créés par l'opérateur binaire « `|` » qui dirige la sortie standard d'un processus vers l'entrée standard d'un autre processus. Les tubes de communication du shell sont supportés par toutes les versions d'Unix.

▶ **Exemple 6.** La commande shell suivante crée deux processus qui s'exécutent en parallèle et qui sont reliés par un tube de communication pipe (figure 5.1). Elle détermine le nombre d'utilisateurs connectés au système en appelant `who`, puis en comptant les lignes avec `wc` :

```
leibnitz> who | wc -l
```

Le premier processus réalise la commande `who`. Le second processus exécute la commande `wc -l`. Les résultats récupérés sur la sortie standard du premier processus sont dirigés vers l'entrée standard du deuxième processus via le tube de communication qui les relie.

Le processus réalisant la commande `who` ajoute dans le tube une ligne d'information par utilisateur du système. Le processus réalisant la commande `wc -l` récupère ces lignes d'information pour en calculer le nombre

total. Le résultat est affiché à l'écran. Les deux processus s'exécutent en parallèle, les sorties du premier processus sont stockées sur le tube de communication. Lorsque le tube devient plein, le premier processus est suspendu jusqu'à ce qu'il y ait libération de l'espace nécessaire pour stocker une ligne d'information. De façon similaire, lorsque le tube devient vide, le second processus est suspendu jusqu'à ce qu'il y ait au moins une ligne d'information sur le tube.

Création d'un tube sans nom

Un tube de communication sans nom est créé par l'appel système `pipe`, auquel on passe un tableau de deux entiers :

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

Au retour de l'appel système `pipe()`, un tube aura été créé, et les deux positions du tableau passé en paramètre contiennent deux descripteurs de fichiers. Les descripteurs de fichiers seront étudiés plus tard, mais on peut pour l'instant considérer un descripteur comme une valeur entière que le système d'exploitation utilise pour accéder à un fichier. Dans le cas du tube on a besoin de deux descripteurs : le descripteur pour les lectures du tube et le descripteur pour les écritures dans le tube. L'accès au tube se fait via les descripteurs. Le descripteur de l'accès en lecture se retrouvera à la position 0 du tableau passé en paramètre, alors que le descripteur de l'accès en écriture se retrouvera à la position 1. Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube. Si le système ne peut pas créer de tube pour manque d'espace, l'appel système `pipe()` retourne la valeur `-1`, sinon il retourne la valeur `0`.

► **Exemple 7.** Les déclarations suivantes créent le tube montré sur la figure 5.2 :

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

Les tubes sans nom sont, en général, utilisés pour la communication entre un processus père et ses processus fils, avec un d'entre eux qui écrit sur le tube, appelé processus écrivain, et l'autre qui lit à partir du tube, appelé processus lecteur (voir figure 5.3). La séquence d'événements pour une telle communication est comme suit :

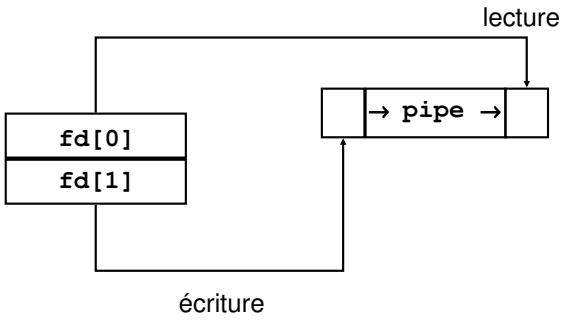

FIG. 5.2 – Pipe de Lecture/Écriture.

- Le processus père crée un tube de communication sans nom en utilisant l'appel système `pipe()`
- Le processus père crée un ou plusieurs fils en utilisant l'appel système `fork()`
- Le processus écrivain ferme l'accès en lecture du tube
- De même, le processus lecteur ferme l'accès en écriture du tube
- Les processus communiquent en utilisant les appels système `write()` et `read()`
- Chaque processus ferme son accès au tube lorsqu'il veut mettre fin à la communication via le tube

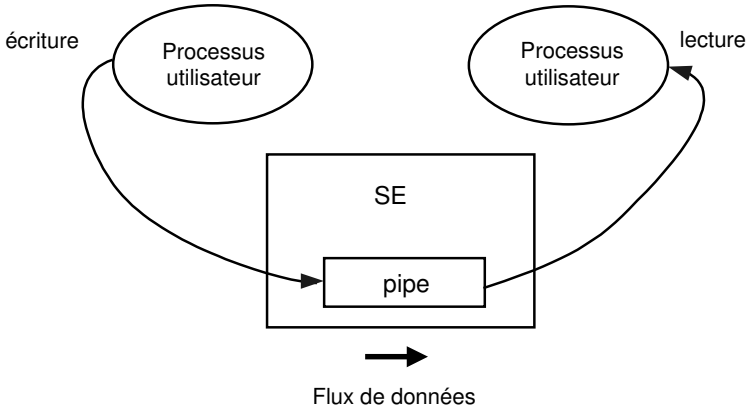

FIG. 5.3 – Création d'un tube de communication entre deux processus.

▶ **Exemple 8.** Dans le programme `upipe.c` le processus père crée un tube de communication pour communiquer avec son processus fils. La commu-

nication est unidirectionnelle du processus fils vers le processus père.

Listing 5.7 – upipe.c

```
#include <sys/types.h> // types
#include <unistd.h>    // fork, pipe, read, write, close
#include <stdio.h>
#define R 0
#define W 1

int main()
{
int fd[2];
char message[100];   // pour recuperer un message
int nboctets;
char *phrase = "message envoye au pere par le fils";

pipe(fd);            // creation d'un tube sans nom
if (fork() ==0)  // creation d'un processus fils
{
   // Le fils ferme le descripteur non utilise de lecture
   close(fd[R]);
   // depot dans le tube du message
   write(fd[W],phrase, strlen(phrase)+1) ;
   // fermeture du descripteur d'ecriture
   close (fd[W]) ;
}
else
{
   // Le pere ferme le descripteur non utilise d'ecriture
   close(fd[W]) ;
   // extraction du message du tube
   nboctets = read (fd[R], message,100) ;
   printf ("Lecture %d octets : %s\n", nboctets, message) ;
   // fermeture du descripteur de lecture
   close (fd[R]) ;
}
return 0;
}
```

Exécution de `upipe.c` :

```
leibnitz> gcc -o upipe upipe.c
leibnitz> upipe
Lecture 35 octets : message envoyé au père par le fils
leibnitz>
```

Remarques : Le processus fils de l'exemple précédent a inclus le caractère

nul dans le message envoyé au processus père. Le père peut facilement le supprimer. Si le processus écrivain envoie plusieurs messages de longueurs variables sur le tube, il est nécessaire d'établir des règles qui permettent au processus lecteur de déterminer la fin d'un message, c'est-à-dire, d'établir un **protocole de communication**. Par exemple, le processus écrivain peut précéder chaque message par sa longueur ou bien terminer chaque message par un caractère spécial comme retour chariot ou le caractère nul.

La figure 5.4 illustre comment un shell exécute une commande comme `ls | wc`. Dans un premier temps, le tube est créé avec la commande `pipe()`. Puis un processus fils est créé. Ce processus fils héritera des accès en entrée et en sortie du tube. Ensuite, le processus parent ferme son accès à la sortie du tube et fait pointer STDOUT sur l'entrée du tube. Dans le même temps, le processus fils ferme son accès à l'entrée du tube et redirige STDIN sur la sortie du tube. Finalement, les deux processus sont remplacés par les programmes `ls` et `wc`.

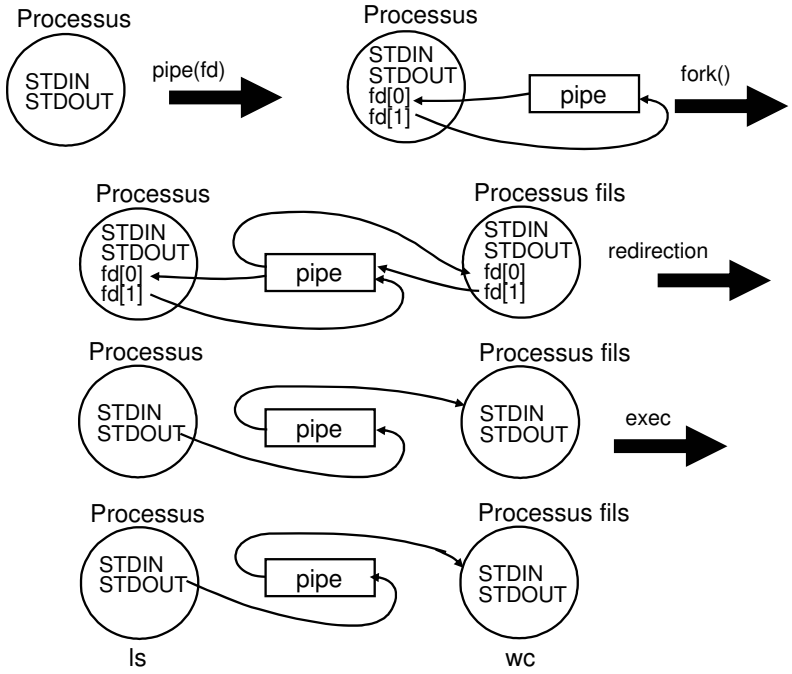

FIG. 5.4 – Usage des descripteurs de fichiers pendant la communication par tube unidirectionnel entre deux processus.

Communication bidirectionnelle

La communication bidirectionnelle entre processus est possible en utilisant deux tubes : un pour chaque sens de communication, comme illustré à la figure 5.5.

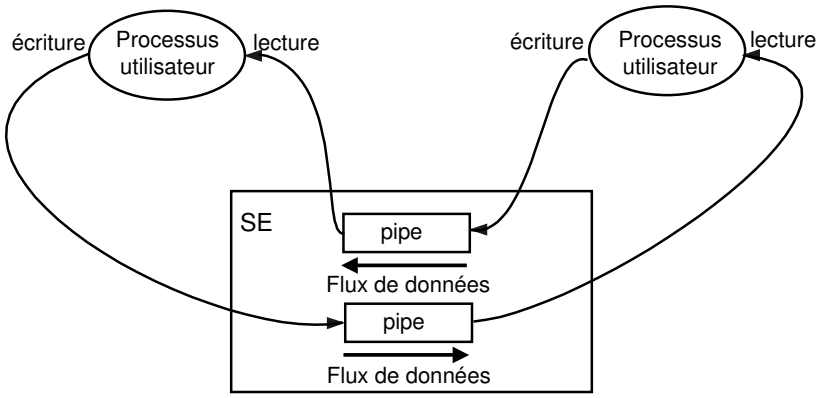

FIG. 5.5 – Création d'un tube de communication bidirectionnelle entre deux processus.

5.3.3 Tubes de communication nommés

Linux supporte un autre type de tubes de communication, beaucoup plus performants. Ils s'agit des **tubes nommés** (*named pipes*). Les tubes de communication nommés fonctionnent aussi comme des files du type FIFO (*First In First Out*). Ils sont plus intéressants que les tubes sans nom car ils offrent, en plus, les avantages suivants :

- Ils ont chacun un nom qui existe dans le système de fichiers (une entrée dans la Table des fichiers).
- Ils sont considérés comme des fichiers spéciaux.
- Ils peuvent être utilisés par des processus indépendants, à condition qu'ils s'exécutent sur une même machine.
- Ils existeront jusqu'à ce qu'ils soient supprimés explicitement.
- Ils sont de capacité plus grande, d'environ 40 Ko.

Ainsi, deux processus sans relation parent-enfant peuvent échanger des données au moyen des tubes nommés.

Appels système `mkfifo()` et commande `mkfifo`

Les tubes de communication nommés sont créés par la commande shell `mkfifo` ou par l'appel système `mkfifo()` :

```
int mkfifo(char *nom, mode_t mode);
```

où `nom` contient le nom du tube et `mode` indique les permissions à accorder au tube. Ainsi, le segment de code suivant permet de créer un tube nommé avec les permissions en lecture et écriture pour tous :

```
if(mkfifo(nom_tube, 0666, 0) != 0)
  {
         printf("Impossible de créer %s\n", nom_tube);
         exit (1);
  }
```

On peut aussi utiliser la commande `mkfifo` du shell pour créer un tube nommé :

```
leibnitz> mkfifo tube
leibnitz>
```

On peut faire l'affichage des attributs du tube créé comme on le fait avec les fichiers :

```
leibnitz> ls -l tube
prw-r--r--    1 jmtorres prof           0 Oct  2 11:36 tube
leibnitz>
```

De la même façon, il est possible de modifier des permissions d'accès :

```
leibnitz> chmod g+rw tube
leibnitz> ls -l tube
prw-rw-r--    1  jmtorres prof          0 Oct  2 11:36 tube|
leibnitz>
```

Remarque : Dans les permissions `p` indique que `tube` est un tube.

Une fois le tube créé, il peut être utilisé pour réaliser la communication entre deux processus. Pour ce faire, chacun des deux processus ouvre le tube, l'un en mode écriture et l'autre en mode lecture.

▶ **Exemple 9.** Le programme `writer.c` envoie un message sur le tube `mypipe`. De son coté, le programme `reader.c` lit un message à partir du même tube.

Listing 5.8 – writer.c

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

int main(void)
{
 int fd;
 char message[26];

 sprintf(message, "bonjour du writer [%d]\n", getpid());

 //Ouverture du tube  mypipe en modeé criture
 fd = open("mypipe", O_WRONLY);
 printf("ici writer[%d] \n", getpid());
 if (fd!=-1) {
   write(fd, message, strlen(message));
 }
 else
   printf("desole, le tube n'est pas disponible\n");

 // Fermeture du tube
 close(fd);
 return 0;
}
```

Listing 5.9 – reader.c

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

int main()
{
  int fd,n;
  char input;

  // ouverture du tube mypipe en mode lecture
  fd = open("mypipe", O_RDONLY);

  printf("ici reader[%d] \n",getpid());

  if (fd!=-1) {
    printf("Recu par le lecteur:\n");
    while ((n = read(fd,&input,1)) > 0) {
      printf("%c", input);
    }
```

```
20      printf("Le lecteur se termine!\n",n);
    }
    else
       printf("desole, le tube n'est pas disponible\n");
    close(fd);
    return 0;
}
```

Après avoir crée le tube `mypipe` avec `mkfifo`, on peut compiler séparément les deux programmes :

```
leibnitz> mkfifo mypipe
leibnitz> ls -l mypipem
prw-r--r--    1 p302690 dgi     0   Nov 28 10:30 mypipe
leibnitz> gcc -o writer writer.c
leibnitz> gcc -o reader reader.c
```

Il est possible de lancer leurs exécutions en arrière plan avec `&`. Les processus `writer` et `reader` ainsi créés communiquent via le tube de communication `mypipe` :

```
leibnitz> writer& reader&
ici reader[5831]
ici writer[5830]
Recu par le lecteur:  bonjour du writer [5830]
Le lecteur se termine!

[1]-  Done                    ./writer
[2]+  Done                    ./reader
leibnitz>
```

Regardons ce qui se passe si on lance deux `writer` et un `reader` :

```
leibnitz> ./writer & ./writer & ./reader &
[1] 5825
[2] 5826
[3] 5827
leibnitz> ici reader[5827]
ici writer[5826]
ici writer[5825]
Recu par le lecteur:  bonjour du writer [5826]
Recu par le lecteur:  bonjour du writer [5825]
Le lecteur se termine!

[1]   Done                    ./writer
```

```
[2]-  Done                    ./writer
[3]+  Done                    ./reader
leibnitz>
```

5.4 Sockets

Les tubes de communication permettent d'établir une communication unidirectionnelle entre deux processus d'une même machine. Le système Unix/Linux permet la communication entre processus s'exécutant sur des machines différentes au moyen de **sockets**.

Les **sockets** ou **prises** sont les mécanismes traditionnels de communication interprocessus du système Unix. Elles permettent la communication entre processus s'exécutant sur des machines différentes connectées par un réseau de communication. Par exemple, l'utilitaire `rlogin` qui permet à un utilisateur d'une machine de se connecter à une autre machine, est réalisé au moyen de **sockets**. Les sockets sont également utilisées pour imprimer un fichier se trouvant sur une autre machine et pour transférer un fichier d'une machine à autre. Des exemples de communication interprocessus en utilisant les **sockets** seront vus dans le Chapitre **??**, *Introduction aux systèmes distribués*.

Suggestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Principles appliqués des systèmes d'exploitation avec Java*, Vuibert, Paris, France, 2002.

Chapître 4 : Gestion de processus.

Chapître 7 : Synchronisation de processus.

Chapître 20 : Le système Unix.

5.5 Exercices

1. Complétez le programme suivant afin de rendre la communication *bidirectionnelle* entre 2 processus sans laisser de zombies. Le père dort 2 secondes après avoir envoyé son message au fils et le fils dort 4 secondes après avoir envoyé son message au père. On n'a pas besoin de variables additionnelles.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#define R 0
#define W 1

int main()
{
int fdp[2],          // père
    fdf[2];          // fils
char message[100];   // pour récupérer un message
char *phrasef = "message envoyé au père par le fils";
char *phrasep = "message envoyé au fils par le père";

// À COMPLÉTER !!

return 0;
}
```

2. Le programme suivant doit afficher le poème de Vignault, « Mon pays ». A l'aide d'une famille de processus, faire que le programme effectue TOUJOURS la sortie correcte du poème (vers dans leur ordre).

   Restrictions :

   (a) Utiliser **seulement** 3 PROCESSUS.

   (b) Pas besoin des variables additionnelles.

   (c) Chaque processus doit afficher **au moins** un vers. Pas de processus bidons.

   (d) Montrer le père et le pid de chaque processus avant l'affichage du vers.

```
#include <unistd.h>
#include <stdio.h>

int main ()
{
```

```
  int statut2; // statut du 2eme processus
  int statut3; // statut du 3eme processus
  char *vers1 = "Mon pays ce n'est pas un pays,
                 c'est l'hiver\n";
  char *vers2 = "Mon jardin ce n'est pas un jardin,
                 c'est la plaine\n";
  char *vers3 = "Mon chemin ce n'est pas un chemin,
                 c'est la neige.\n";
  char *vers4 = "Mon pays ce n'est pas un pays,
                 c'est l'hiver !\n";

  printf ("-- Mon pays, récité par 3 processus\n") ;
  switch (fork())
    {
      // A compléter !
    }
  return 0;
}
```

3. Le code C/C++ ci-dessous effectue 3 milliards de tours de boucle, et à chaque 100 ms (i.e. 100000 us) le processus est interrompu pour afficher le nombre de tours de boucle effectué à date. La temporisation s'effectue à l'aide de l'appel système `setitimer(...)` qui envoie un signal `SIGPROF` au processus à chaque 100 ms. Le prototype de `setitimer()` est :

   ```
   int setitimer(ITIMER_PROF,const struct itimerval *value,
                 NULL);
   ```

   Complétez le code ci-dessous, de façon à ce que la fonction `sighandler()` soit exécutée à la réception du signal `SIGPROF`. Le signal 29 `SIGPROF` sert au declenchement d'un temporisateur profilé (`setitimer`).

   `sighandler()` doit afficher le nombre de tours de boucle effectué et doit aussi compter le nombre d'interruptions du temporisateur. Avant de terminer son exécution, le processus affiche alors le nombre total des interruptions durant les 3 milliards de tours de boucle.

   ```
   #include <sys/time.h>
   #include <signal.h>
   #include <iostream.h>
  // Définition du gestionnaire du signal de temporisation
   static void sighandler(int sigtype);
   long i = 0; // Compteur
   ```

```
...
// Valeurs du temporisateur par intervalle
struct itimerval v;

int main()
{
   ...
   // Initialisation du temporisateur
   //Valeur des répétitions en secondes ou microsecondes
   v.it_interval.tv_sec = 0;
   v.it_interval.tv_usec = 100000;
   // Valeur initiale en secondes ou microsecondes
   v.it_value.tv_sec = 0;
   v.it_value.tv_usec = 100000;

   printf("\ndébut de la temporisation\n");
   setitimer(ITIMER_PROF, &v, NULL);

for (i=0; i < 3000000000; i++);

   exit(0);
}
...
```

4. En utilisant le diagramme 5.5 sur la communication bidirectionnelle, écrivez le code pour permettre à deux processus d'échanger des informations.
5. Le programme `fils-tube.c` implante la communication entre 2 processus fils par un tube. Le premier processus (producteur) génère une valeur y qui est envoyée au second processus (consommateur). Tracez manuellement le programme et affichez les résultats possibles. Ensuite, exécutez le code en enlevant l'appel système sleep et observez les résultats. Si vous n'observez pas alors une alternance d'exécution entre les processus producteur et consommateur, expliquez-en la raison. Finalement, utilisez l'appel système gettimeofday pour mesurer le temps d'exécution (système et utilisateur en microsecondes) de chacun des processus et expliquez pourquoi ce temps n'est pas exactement le même pour chacun des processus. Si votre temps d'exécution est `0`, alors incrémentez la valeur du paramètre de `sleep()`.

Listing 5.10 – fils-tube.c

```
#include <stdio.h>
#include <stdlib.h>
```

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

int main(void)
{
int
   pid1, pid2,   // identificateurs des processus fils
   p1, p2,       // identificateurs des processus termines
   code_retour1,
   code_retour2, // Codes de terminaison des processus
   x = 0,        // donnee non partagee entre les processus
   y = 0,        // donnee partagee entre les processus
   i,            // indice
   tube[2];      // tube entre les processus
   pipe(tube);   // creation du tube
   // Bloc du processus producteur
   pid1=fork();
   // Si pid = 0, alors executer code du processus cree
   if (pid1==0)
   {
       for (i = 0; i < 10; ++i)
       {
           // afficher les donnees
           printf("Producteur : x=%d y=%d\n", ++x, --y);
           // Envoyer y au consommateur via le tube
           write(tube[1], &y, 1);
           // Bloquer le producteur pour 2 secondes
           sleep(2);
       }
       exit(0); // Terminer l'execution du producteur
   }
   // Bloc du processus consommateur
   pid2=fork();
   // Si pid = 0, executer code du processus cree
   if(pid2==0)
   {
     for (i = 0; i < 10; ++i)
     {
        // Extraire y  du tube
        read(tube[0], &y, 1);
        // afficher les donnees
        printf("Consommateur : x=%d y=%d\n", x++, y);
        // Bloquer le consommateur pour 2 secondes
        sleep(2);
     }
     exit(0); // Terminer l'execution du producteur
   }
   // Bloc du processus parent
```

```
    // Execution du parent concurremment avec ses fils
    // Afficher les valeurs de x et y
    printf("Parent : x=%d y=%d\n", x, y);
    // Synchroniser le processus parent avec les fils
    p1 = wait(&code_retour1);
    p2 = wait(&code_retour2);
    // Indiquer l'ordre de terminaison des fils
    if (p1 == pid1)
        printf("\nProducteur a termine en premier.");
    else
        printf("\nConsommateur a termine en premier.");
    return 0;   // Terminer le processus parent
} // main
```