**CHAPITRE 5 :**

# COMMUNICATION ET SYNCHRONIZATION INTERPROCESSUS AVEC LE LANGAGE C

**Objectifs spécifiques**

- Programmer en C sous Linux
- Ecrire des programmes en C pour manipuler la notion de communication interprocessus

**Eléments de contenu**

**I.** Introduction à la programmation C sous Linux

**II.** Création, lancement et terminaison d'un processus

**III**. Attente de terminaison d'un processus : fonction Wait

**IV**. Lancement d'un programme : fonction exec

**V**. Communication interprocessus : les tubes

**VII**. Les tubes nommés

**Volume Horaire :**

**Cours : 4** heures 30 mn

TD : 1 heure 30 mn

## 5.1 Introduction à la programmation C sous Linux

### 5.1.1 Arguments de la fonction main

La fonction main d'un programme peut prendre des arguments en ligne de commande. Par exemple, si un fichier prog.c a permis de générer un exécutable prog à la compilation,

```
gcc prog.c –o prog
```

On peut invoquer le programme prog avec des arguments comme suit :

```
./prog argument1 argment2 argument3
```

Pour récupérer les arguments dans le programme *C*, on utilise les paramètres argc et argv du main. L'entier argc donne le nombre d'arguments rentrés dans la ligne de commande **plus 1**, et le paramètre argv est un tableau de chaînes de caractères qui contient comme éléments :

- Le premier élément argv[0] qui est une chaîne qui contient le nom du fichier executable du programme ;
- Les éléments suivants argv[1], argv[2], etc... sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

Le prototype de la fonction main est donc **int main(int argc, char**argv)**;

**Exemple :** Voici un programme longeurs, qui prend en argument des mots, et affiche la longueur de ces mots.

```
#include <stdio.h>
#include <string.h>
int main(int argc, char**argv)
{
int i;
printf("Vous avez entré %d mots\n", argc-1);
puts("Leurs longueurs sont :");
for (i=1 ; i<argc ; i++)
{
printf("%s : %d\n", argv[i], strlen(argv[i]));
}
return 0;
}
```

Voici un exemple de trace :

```
$ gcc longueur.c -o longueur
$ ./longueur toto blabla
Vous avez entré 2 mots
Leurs longueurs sont :
toto : 4
blabla : 6
```

### 5.1.2 Accès aux variables d'environnement dans un programme C

Dans un programme C, on peut accéder à la liste des variables d'environnement dans la variable **environ**, qui est un tableau de chaînes de caractères (terminé par un pointeur NULL pour marquer la fin de la liste).

```
#include <stdio.h>
extern char **environ;
int main(void)
{
int i;
for (i=0 ; environ[i]!=NULL ; i++)
puts(environ[i]);
return 0;
}
```

Pour accéder à une variable d'environnement particulière à partir de son nom, on utilise la fonction **getenv**, qui prend en paramètre le nom de la variable et qui retourne sa valeur sous forme de chaîne de caractère.

```
#include <stdio.h>
#include <stdlib.h> /* pour utiliser getenv */
int main(void)
{
char *valeur;
valeur = getenv("PATH");
if (valeur != NULL)
printf("Le PATH vaut : %s\n", valeur);
valeur = getenv("HOME");

if (valeur != NULL)

printf("Le home directory est dans %s\(\backslash\)n", valeur);
return 0;
}
```

Pour changer la valeur d'une variable d'environnement, on utilise la fonction **putenv**, qui prend en paramètre une chaîne de caractère. Notons que la modification de la variable ne vaut

que pour le programme lui-même et ses descendants (autres programmes lancés par le programme), et ne se transmet pas au shell (ou autre) qui a lancé le programme en cours.

```
#include <stdio.h>
#include <stdlib.h> /* pour utiliser getenv */
int main(void)
{
char *path, *home, *nouveaupath;
char assignation[150];
path = getenv("PATH");
home = getenv("HOME");
printf("ancien PATH : %s\n et HOME : %s\n",
path, home);
path=home;
nouveaupath = getenv("PATH");
printf("nouveau PATH : \n%s\n", nouveaupath);
return 0;
}
```

**Exemple de trace:**

```
$ gcc putenv.c -o putenv
 ./putenv
ancien PATH : /usr/local/bin:/usr/bin:/bin:/usr/bin/imene
et HOME : /home/ImeneSghaier
nouveau PATH :
/home/ImeneSghaier
```

Si on fait echo $PATH après cette exécution on trouve que la valeur n'a pas changé :

```
echo $PATH

/usr/local/bin:/usr/bin:/bin:/usr/bin/imene
```

## 5.2 Création, lancement et terminaison d'un processus

### 5.2.1 Processus PID et UID

Un programme C peut accéder au *PID* de son instance en cours d'exécution par la fonction

**getpid()**, qui retourne son ***PID*** :

```
pid_t getpid();
```

Chaque processus possède aussi un *User ID*, noté *UID*, qui identifie l'utilisateur qui a lancé le processus. C'est en fonction de l'*UID* que le processus se voit accordé ou bien refuser les droits d'accès en lecture, écriture ou exécution à certains fichiers ou à certaines commandes. On fixe les droits d'accès d'un fichier avec la commande chmod.

L'utilisateur **root** possède un *UID* égal à 0.

Un programme C peut accéder à l'*UID* de son instance en cours d'exécution par la fonction getuid :

```
uid_t getuid();
```

### 5.2.2 La fonction fork

La fonction **fork** permet à un programme en cours d'exécution de créer un nouveau processus. Le processus d'origine est appelé processus père, et il garde son *PID*, et le nouveau processus créé s'appelle processus fils, et possède un nouveau *PID*. Le processus père et le processus fils ont le même code source, mais la valeur retournée par fork permet de savoir si on est dans le processus père ou fils.

La fonction **fork retourne -1 en cas d'erreur**, **retourne 0 dans le processus fils**, et **retourne le *PID* du fils dans le processus père**. Ceci permet au père de connaître le *PID* de son fils.

Lors de l'exécution de l'appel-système fork, le noyau effectue les opérations suivantes :

- il alloue un bloc de contrôle dans la table des processus.
- il copie les informations contenues dans le bloc de contrôle du père dans celui du fils sauf les identificateurs (PID, PPID...).
- il alloue un PID au processus fils.
- il associe au processus fils un segment de texte dans son espace d'adressage. Le segment de données et la pile ne lui seront attribués uniquement lorsque celui-ci tentera de les modifier. Cette technique, nommée « *copie on write* », permet de réduire le temps de création du processus.
- l'état du processus est mis à l'état *exécution*.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(void)
{
pid_t pid_fils;
pid_fils = fork();
if (pid_fils == -1)
{
puts("Erreur de création du nouveau processus");
exit (1);
}
if (pid_fils == 0)
{
printf("Nous sommes dans le fils\n");
/* la fonction getpid permet de connaître son propre PID */
printf("Le PID du fils est \%d\n", getpid());
/* la fonction getppid permet de connaître le PPID
(PID de son père) */
printf("Le PID de mon père (PPID) est \%d", getppid());
}
else
{
printf("Nous sommes dans le père\n");
printf("Le PID du fils est \%d\n", pid_fils);
printf("Le PID du père est \%d\n", getpid());
printf("PID du grand-père : \%d", getppid());
}
return 0;
}
```

### 5.2.3 Terminaison d'un processus fils

Le processus courant se termine automatiquement lorsqu'il cesse d'exécuter la fonction main(). Les primitives suivantes lui permettent d'arrêter explicitement son exécution :

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

Ces deux primitives provoquent la terminaison du processus courant. Le paramètre status spécifie un code de retour, compris entre 0 et 255, à communiquer au processus père.

Par convention, en cas de terminaison normale, un processus doit retourner la valeur 0. Avant de terminer l'exécution du processus, exit() exécute les fonctions de « nettoyage » des librairies standard.

**Exemple**

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

int main (void)
{
 int i ;
 for (i=0 ; i < 4 ; i++) {
  int retour ;
  retour = fork () ;
  switch (retour) {
    case -1 : /* erreur */
      perror ("erreur fork\n") ;
      exit (1) ;
    case 0 : /* fils */
      printf ("fils : %d\n", i) ;
    default : /* pere */
      printf ("pere : \n") ;
  }
 }
}
```

## 5.3 Attente de terminaison d'un processus fils : fonction wait

Le processus courant se termine automatiquement lorsqu'il cesse d'exécuter la fonction main().

Lorsque le processus fils se termine (soit en sortant du main soit par un appel à exit) avant le processus père, le processus fils ne disparaît pas complètement, mais devient un zombie. Pour permettre à un processus fils à l'état de zombie de disparaître complètement, le processus père peut appeler l'instruction suivante: **wait(NULL);** qui se trouve dans la bibliothèque **sys/wait.h**

Cependant, il faut prendre garde l'appel de wait est bloquant, c'est à dire que lorsque la fonction wait est appelée, l'exécution du père est suspendue jusqu'à ce qu'un fils se termine.

De plus, **il faut mettre autant d'appels de wait qu'il y a de fils**.

Lorsque l'on appelle cette fonction, cette dernière bloque le processus à partir duquel elle a été appelée jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID de ce dernier. En cas d'erreur, la fonction renvoie la valeur **-1** (renvoie le code d'erreur −1 dans le cas où le processus n'a pas de fils).

Le paramètre **status** correspond au code de retour du processus fils qui va se terminer. Autrement dit, la variable que l'on y passera aura la valeur du code de retour du processus (ce code de retour est généralement indiqué avec la fonction exit).

La fonction wait est fréquemment utilisée pour permettre au processus père d'attendre la fin de ses fils avant de se terminer lui-même, par exemple pour récupérer le résultat produit par un fils.

Il est possible de mettre le processus père en attente de la fin d'un processus fils particulier par waitpid.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Plus précisément, la valeur de pid est interprétée comme suit :

- si *pid* > *0*, le processus père est suspendu jusqu'à la fin d'un processus fils dont le PID est égal à la valeur *pid* ;
- si *pid* = *0*, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils appartenant à son groupe ;
- si *pid* = *-1*, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils ;
- si *pid* < *-1*, le processus père est suspendu jusqu'à la mort de n'importe lequel de ses fils dont le GID est égal.

Le second argument, status, a le même rôle qu'avec wait.

Le troisième argument permet de préciser le comportement de waitpid. On peut mettre 0 ou utiliser une des deux constantes suivante:

- WNOHANG : ne pas bloquer si aucun fils ne s'est terminé.
- WUNTRACED : recevoir l'information concernant également les fils bloqués si on ne l'a pas encore reçue.

**Exercice :**

Ecrire un programme C permettant à un processus père de récupérer le code renvoyé par un processus fils dans la fonction exit.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h> //pour wait
#include <errno.h> /* permet de récupérer les codes d'erreur */
pid_t pid_fils
int main(void)
{
int status;
switch (pid_fils=fork())
{
case -1 : perror("Problème dans fork()\n");
exit(errno); /* retour du code d'erreur */
break;
case 0 : puts("Je suis le fils");
puts("Je retourne le code 3");
exit(3);
default : puts("Je suis le père");
puts("Je récupère le code de retour");
wait(&status);
printf("code de sortie du fils %d : %d\n",
pid_fils, WEXITSTATUS(status));
break;
```

```
}
return 0;
}
```

Les informations à propos la bonne ou la mauvaise terminaison d'un processus peuvent être accédées facilement à l'aide des macros suivantes définies dans **sys/wait.h**.

- **WIFEXITED (status)** : Elle renvoie vrai si le statut provient d'un processus fils qui s'est terminé en quittant le main avec return ou avec un appel à exit.
- **WEXITSTATUS (status) :** (si WIFEXITED (status) renvoie vrai) renvoie le code de retour du processus fils passé à _exit() ou exit() ou la valeur retournée par la fonction main() ;
- **WIFSIGNALED (status) :** renvoie vrai si le statut provient d'un processus fils qui s'est terminé à cause de la réception d'un signal ;
- **WTERMSIG (status)** : (si WIFSIGNALED (status) renvoie vrai) renvoie la valeur du signal qui a provoqué la terminaison du processus fils.

**Exemple**

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main (void){
 pid_t pid ;
 int status ;
 pid = fork () ;
 switch (pid) {
  case -1 :
   perror ("fork") ;
   exit (1) ;
  case 0 : /* le fils */
   printf ("processus fils\n") ;
   exit (2) ;
  default : /* le pere */
   printf ("pere: a cree processus %d\n", pid) ;
```

```
    wait (&status) ;
    if (WIFEXITED (status))
      printf ("fils termine normalement: status = %d\n",
            WEXITSTATUS (status)) ;
    else
      printf ("fils termine anormalement\n") ;
 }
}
```

```
./status_fils
pere: a cree processus 907
processus fils
fils termine normalement: status = 2
```

## Exemple 2

```
/* Pour les constantes EXIT_SUCCESS et EXIT_FAILURE */
#include <stdlib.h>
/* Pour fprintf() */
#include <stdio.h>
/* Pour fork() */
#include <unistd.h>
/* Pour perror() et errno */
#include <errno.h>
/* Pour le type pid_t */
#include <sys/types.h>
/* Pour wait() */
#include <sys/wait.h>

/* Pour faire simple, on déclare status en globale à la barbare */
int status;
/* La fonction create_process duplique le processus appelant et retourne
  le PID du processus fils ainsi créé */
pid_t create_process(void)
{
```

```
   /* On crée une nouvelle valeur de type pid_t */
   pid_t pid;
   /* On fork() tant que l'erreur est EAGAIN */
   do {
         pid = fork();
   } while ((pid == -1) && (errno == EAGAIN));

   /* On retourne le PID du processus ainsi créé */
   return pid;
}

/* La fonction child_process effectue les actions du processus fils */
void child_process(void)
{
   printf(" Nous sommes dans le fils !\n"
            " Le PID du fils est %d.\n"
            " Le PPID du fils est %d.\n", (int) getpid(), (int) getppid());
}

/* La fonction father_process effectue les actions du processus père */
void father_process(int child_pid)
{
   printf(" Nous sommes dans le père !\n"
            " Le PID du fils est %d.\n"
            " Le PID du père est %d.\n", (int) child_pid, (int) getpid());

   if (wait(&status) == -1) {
            perror("wait :");
            exit(EXIT_FAILURE);
   }

   if (WIFEXITED(status)) {
         printf(" Terminaison normale du processus fils.\n"
               " Code de retour : %d.\n", WEXITSTATUS(status));
```

```
    }

    if (WIFSIGNALED(status)) {
        printf(" Terminaison anormale du processus fils.\n"
            " Tué par le signal : %d.\n", WTERMSIG(status));
    }

}

int main(void)
{
    pid_t pid = create_process();

    switch (pid) {
        /* Si on a une erreur irrémédiable (ENOMEM dans notre cas) */
    case -1:
        perror("fork");
        return EXIT_FAILURE;
        break;
        /* Si on est dans le fils */
    case 0:
        child_process();
        break;
        /* Si on est dans le père */
    default:
        father_process(pid);
        break;
    }

    return EXIT_SUCCESS;
}
```

## 5.4 Lancement d'un programme : commande exec

L'appel système **exec** permet de remplacer le programme en cours par un autre programme : une substitution sans changer de numéro de processus (PID). Autrement dit, un programme peut se faire remplacer par un autre code source ou un script shell en faisant appel à exec.
En utilisant fork, puis en faisant appel à exec dans le processus fils, un programme peut lancer un autre programme et continuer à tourner dans le processus père.
Il y a en fait plusieurs fonctions de la famille exec qui sont légèrement différentes.

- La fonction **execl** prend en paramètre une liste des arguments à passer au programme (liste terminée par NULL).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
/* dernier élément NULL, OBLIGATOIRE */
execl("/usr/bin/emacs", "emacs", "fichier.c", "fichier.h", NULL);
perror("Problème : cette partie du code ne doit jamais être exécutée");
return 0;
}
```

  Le premier paramètre est une chaîne qui doit contenir le chemin d'accès complet (dans le système de fichiers) au fichier exécutable ou au script shell à exécuter. Les paramètres suivants sont des chaînes de caractère qui représentent les arguments passés en ligne de commande au main de ce programme. La chaîne argv[0] doit donner le nom du programme (sans chemin d'accès), et les chaînes suivantes argv[1], argv[2], etc. donnent les arguments. Concernant le chemin d'accès, il est donné à partir du répertoire de travail (**pwd**), ou à partir du répertoire racine / s'il commence par le caractère / (exemple : /home/imene/enseignement/systeme/script1).

- La fonction **execlp** permet de rechercher les exécutables dans les répertoires apparaissant dans la variable PATH, ce qui évite souvent d'avoir à spécifier le chemin complet.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
```

```
{
/* dernier élément NULL, OBLIGATOIRE */
execlp("emacs", "emacs", "fichier.c", "fichier.h", NULL);
perror("Problème : cette partie du code ne doit jamais être exécutée");
return 0;
}
```

### 5.4.1 La commande execv (arguments en vecteur)

La différence avec execl est que l'on n'a pas besoin de connaître la liste des arguments à l'avance (ni même leur nombre). Cette fonction a pour prototype :

```
int execv(const char* application, const char* argv[]);
```

Le mot const signifie seulement que la fonction execv ne modifie pas ses paramètres. Le premier paramètre est une chaîne qui doit contenir le chemin d'accès (dans le système de fichiers) au fichier exécutable ou au script shell à exécuter. Le deuxième paramètre est un tableau de chaînes de caractères donnant les arguments passés au programme à lancer dans un format similaire au paramètre argv du main de ce programme. La chaîne argv[0] doit donner le nom du programme (sans chemin d'accès), et les chaînes suivantes argv[1], argv[2], etc donnent les arguments.

Il faut que le dernier élément du tableau de pointeurs argv soit NULL pour marquer la fin du tableau. Ceci est dû au fait que l'on ne passe pas de paramètre argc donnant le nombre d'argument

Concernant le chemin d'accès, il est donné à partir du répertoire de travail (pwd), ou à partir du répertoire racine / s'il commence par le caractère / **Exemple.** Le programme suivant édite les fichiers .c et .h du répertoire de travail avec

emacs.

Dans le programme, le chemin d'accès à la commande emacs est donné à partir de la racine /usr/bin/emacs.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
char * argv[] = {"emacs", "fichier.c", "fichier.h", NULL}
/* dernier élément NULL, obligatoire */
```

```
execv("/usr/bin/emacs", argv);
puts("Problème : cette partie du code ne doit jamais être exécutée");
return 0;
}
```

## 5.5 Communication interprocessus

### 5.5.1 Tube et fork

Un tube de communication est un tuyau (en anglais *pipe*) dans lequel un processus peut écrire des données et un autre processus peut lire. On crée un tube par un appel à la fonction **pipe**, déclarée dans **unistd.h** :

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

La fonction renvoie 0 si elle réussit, et elle crée alors un nouveau tube. La fonction pipe remplit le tableau descripteur passé en paramètre, avec :

- descripteur [0] désigne la sortie du tube (dans laquelle on peut lire des données) ;
- descripteur [1] désigne l'entrée du tube (dans laquelle on peut écrire des données) ;

Le principe est qu'un processus va écrire dans descripteur [1] et qu'un autre processus va lire les mêmes données dans descripteur[0]. Le problème est qu'on ne crée le tube dans un seul processus, et un autre processus ne peut pas deviner les valeurs du tableau descripteur.

Pour faire communiquer plusieurs processus entre eux, il faut appeler la fonction pipe avant d'appeler la fonction fork. Ensuite, le processus père et le processus fils auront les mêmes descripteurs de tubes, et pourront donc communiquer entre eux. De plus, un tube ne permet de communiquer que dans un seul sens. Si l'on souhaite que les processus communiquent dans les deux sens, il faut créer deux pipes.

Pour écrire dans un tube, on utilise la fonction write :

```
ssize_t write(int descripteur1, const void *ElementAEcrire, size_t tailleEnOctet);
```

La fonction prend en paramètre l'**entrée du tube** (on lui enverra descripteur[1]), un **pointeur générique vers la mémoire contenant l'élément à écrire** ainsi que le **nombre d'octets de cet élément**. Elle renvoie une valeur de type ssize_t correspondant au nombre d'octets effectivement écrits.

**Exemple :** Pour écrire le message "Bonjour" dans un tube, depuis le père :

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

```
#include <unistd.h>
#include <sys/wait.h>
#define TAILLE_MESSAGE 256 /* Correspond à la taille de la chaîne à écrire */
int main(void){
   pid_t pid_fils;
   int descripteurTube[2];
   char messageEcrire[TAILLE_MESSAGE];
   pipe(descripteurTube);
   pid_fils = fork();
   if(pid_fils != 0) /* Processus père */
   {
      sprintf(messageEcrire, "Bonjour, fils. Je suis ton père !"); /* La fonction sprintf permet de remplir une chaîne de caractère avec un texte donné */
      write(descripteurTube[1], messageEcrire, TAILLE_MESSAGE);
   }
   return EXIT_SUCCESS;
}
```

Et pour lire dans un tube, on utilise la fonction read :

```
ssize_t read(int descripteur0, void *ElementALire, size_t tailleenOctet);
```

Le descripteur doit correspondre à la sortie d'un tube, le bloc pointe vers la mémoire destinée à recevoir les octets, et la taille donne le nombre d'octets qu'on souhaite lire. La fonction renvoie le nombre d'octets effectivement lus. Si cette valeur est inférieure à taille, c'est qu'une erreur s'est produite en cours de lecture (par exemple la fermeture de l'entrée du tube suite à la terminaison du processus qui écrit).

**Exemple** : Pour lire un message envoyé par le père à un fils :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define TAILLE_MESSAGE 256 /* Correspond à la taille de la chaîne à lire */
int main(void){
   pid_t pid_fils;
```

```
   int descripteurTube[2];
   char messageLire[TAILLE_MESSAGE];
   pipe(descripteurTube);
   pid_fils = fork();
   if(pid_fils == 0) /* Processus fils */
   {
      read(descripteurTube[0], messageLire, TAILLE_MESSAGE);
      printf("Message reçu = \"%s\"", messageLire);
   }

   return EXIT_SUCESS;
}
```

Dans la pratique, on peut transmettre un buffer qui a une taille fixe (256 octets par exemple). L'essentiel est qu'il y ait exactement le même nombre d'octets en lecture et en écriture de part et d'autre du pipe. La partie significative du buffer est terminée par un 0\00 comme pour n'importe quelle chaîne de caractère.

**Exercice 1 :**

Écrivez un programme qui crée deux processus : le père écrit le message « Bonjour, fils. Je suis ton père ! ». Le fils le récupère, puis l'affiche.

**Correction**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define TAILLE_MESSAGE 256 /* Correspond à la taille de la chaîne à lire et à écrire */
int main(void){
pid_t pid_fils;
int descripteurTube[2];
unsigned char messageLire[TAILLE_MESSAGE] ;
unsigned char messageEcrire[TAILLE_MESSAGE];
 printf("Création du tube.\n");
 if(pipe(descripteurTube) != 0){
```

```
      fprintf(stderr, "Erreur de création du tube.\n");
      return EXIT_FAILURE;
   }
pid_fils = fork();
if(pid_fils == -1)
   {
      fprintf(stderr, "Erreur de création du processus.\n");
      return 1;
   }
if(pid_fils == 0)
   {
      printf("Fermeture de l'entrée dans le fils.\n\n");
      close(descripteurTube[1]);
     read(descripteurTube[0], messageLire, TAILLE_MESSAGE);
      printf("Nous sommes dans le fils (pid = %d).\nIl a reçu le message suivant du
père : \"%s\".\n\n\n", getpid(), messageLire);
   }
   else
   {
      printf("\nFermeture de la sortie dans le père.\n");
      close(descripteurTube[0]);
      sprintf(messageEcrire, "Bonjour, fils. Je suis ton père !");
      printf("Nous sommes dans le père (pid = %d).\nIl envoie le message suivant au
fils : \"%s\".\n\n\n", getpid(), messageEcrire);
      write(descripteurTube[1], messageEcrire, TAILLE_MESSAGE);
      wait(NULL);
   }
  return 0;
}
```

**Trace d'exécution**

```
Création du tube
Fermeture de la sortie dans le père
```

```
Fermeture de l'entrée dans le fils
Nous sommes dans le père (pid=2334)
Il envoie le message suivant au fils : « bonjour ; fils. Je suis ton père ! »
Nous sommes dans le fils (pid=2335)
Il a reçu le message suivant du père : « Bonjour, fils. Je suis ton père ! »
```

Exercice 2

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#define BUFFER_SIZE 256
int main(void){
pid_t pid_fils;
int tube[2];
unsigned char bufferR[256], bufferW[256];
puts("Creation d'un tube");
if (pipe(tube) != 0) {/* pipe */
fprintf(stderr, "Erreur dans pipe\ n");
exit(1);
}
pid_fils = fork(); {/* fork */
if (pid_fils == -1)
{
fprintf(stderr, "Erreur dans fork\n");
exit(1);
}
if (pid_fils == 0) {/* processus fils */
{
printf("Fermeture entree dans le fils (pid = %d)\n", getpid());
close(tube[1]);
read(tube[0], bufferR, BUFFER_SIZE);
printf("Le fils (%d) a lu : %s\n", getpid(), bufferR);
}
```

```
else {/* processus pere */
{
printf("Fermeture sortie dans le pere (pid = %d)\n", getpid());
close(tube[0]);
sprintf(bufferW, "Message du pere (%d) au fils", getpid());
write(tube[1], bufferW, BUFFER_SIZE);
wait(NULL);
}
return 0;
}
```

La sortie de ce programme est :

```
Création d'un tube
Fermeture entrée dans le fils (pid = 12756)
Fermeture sortie dans le père (pid = 12755)
Ecriture de 31 octets du tube dans le père
Lecture de 31 octets du tube dans le fils
Le fils (12756) a lu : Message du père (12755) au fils
```

Il faut noter que les fonctions read et write permettent de transmettre uniquement des tableaux des octets. Toute donnée (nombre ou texte) doit être convertie en tableau de caractère pour être transmise, et la taille des ces données doit être connue dans les deux processus communiquant.

### 5.5.2 Transmission de données binaires

Voici un exemple de programme qui saisit une valeur x au clavier dans le processus père, et transmet le sinus de ce nombre, **en tant que double** au processus fils.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#include <math.h>
int main(\void){
pid_t pid_fils;
```

```
int tube[2];
double x, valeurW, valeurR;
puts("Création d'un tube");
if (pipe(tube) != 0) /* pipe */
{
fprintf(stderr, "Erreur dans pipe\n");
exit(1);
}
switch(pid_fils = fork()) /* fork */
{
case -1 :
perror("Erreur dans fork\n");
exit(errno);
case 0 : /* processus fils */
close(tube[1]);
read(tube[0], &valeurR, sizeof(double));
printf("Le fils (%d) a lu : %.2f\(\backslash\)n", getpid(), valeurR);
break;
default : /* processus père */
printf("Fermeture sortie dans le père (pid = %d)\n", getpid());
close(tube[0]);
puts("Entrez x :");
scanf("%lf", &x);
valeurW = sin(x);
write(tube[1], &valeurW, sizeof(double));
wait(NULL);
break;
}
return 0;
}
```

### 5.5.3 Entrée/Sortie et tubes

On peut lier la sortie d'un tube à **stdin** ou l'entrée d'un tube à **stdout**. Ensuite :

- Dans le premier cas : toutes les informations qui sortent du tube arrivent dans le flot d'entrée standard et peuvent être lues avec scanf, fgets, etc...

- Dans le second cas : toutes les informations qui sortent par stdout sont écrites dans le tube. On peut utiliser printf, puts, etc...

Pour faire cela, il suffit d'utiliser la fonction :

```
int dup2(int ancienDescripteur, int nouveauDescripteur);
```

Le premier paramètre est facilement compréhensible, on lui enverra :

- descripteur[0] si on veut lier la sortie ;

- descripteur[1] si on veut lier l'entrée.

Le second paramètre correspond au nouveau descripteur que l'on veut lier au tube. Il existe deux constantes, déclarées dans **unistd.h** :

- STDIN_FILENO ;

- STDOUT_FILENO.

On utilise STDIN_FILENO lorsqu'on veut lier la sortie à stdin, et STDOUT_FILENO lorsqu'on veut lier l'entrée à stdout.

**Exemple** : Si on veut lier l'entrée d'un tube d'entrée et de sortie définies dans un tableau descripteur à stdout :

```
dup2(tube[1], STDOUT_FILENO);
```

## 5.6 Les tubes nommés

On peut faire communiquer deux processus à travers un tube nommé. Le tube nommé passe par un fichier sur le disque. L'intérêt est que **les deux processus n'ont pas besoin d'avoir un lien de parenté**. Pour créer un tube nommé, on utilise la fonction **mkfifo** de la bibliothèque **sys/stat.h**.

**Exemple.** Dans le code suivant, le premier programme transmet le mot "coucou" au deuxième programme. Les deux programmes n'ont pas besoin d'être liés par un lien de parenté.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(){
int fd;
FILE *fp;
char *nomfich="/tmp/test.txt"; /* nom du fichier */
if(mkfifo(nomfich, 0644) != 0) /* création du fichier */
{
perror("Problème de création du noeud de tube");
exit(1);
}
fd = open(nomfich, O_WRONLY); /* ouverture en écriture */
fp=fdopen(fd, "w"); /* ouverture du flot */
fprintf(fp, "coucou\n"); /* écriture dans le flot */
unlink(nomfich); /* fermeture du tube */
return 0;
}
```

La fonction mkfifo prend en paramètre, outre le chemin vers le fichier, le masque des permissions (lecture, écriture) sur la structure *fifo*.

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
int main(){
int fd;
FILE *fp;
char *nomfich="/tmp/test.txt", chaine[50];
```

```
fd = open(nomfich, O_RDONLY); /* ouverture du tube */
fp=fdopen(fd, "r"); /* ouverture du flot */
fscanf(fp, "%s", chaine); /* lecture dans le flot */
puts(chaine); /* affichage */
unlink(nomfich); /* fermeture du flot */
return 0;
}
```

**Exercice**

Écrivez deux programmes indépendants : un écrit un message dans un tube nommé, et l'autre le lit, puis l'affiche. Exécutez ces deux programmes en même temps.

**Correction**

**Ecrivain.c**

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

#define TAILLE_MESSAGE  256

int main(void)
{
   int entreeTube;
   char nomTube[] = "essai.fifo";

   char chaineAEcrire[TAILLE_MESSAGE] = "Bonjour";

   if(mkfifo(nomTube, 0644) != 0)
   {
      fprintf(stderr, "Impossible de créer le tube nommé.\n");
      exit(EXIT_FAILURE);
   }
```

```
   if((entreeTube = open(nomTube, O_WRONLY)) == -1)
   {
      fprintf(stderr, "Impossible d'ouvrir l'entrée du tube nommé.\n");
      exit(EXIT_FAILURE);
   }

   write(entreeTube, chaineAEcrire, TAILLE_MESSAGE);


   return EXIT_SUCCESS;
}
```

**Lecteur.c** :

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

#define TAILLE_MESSAGE  256

int main(void)
{
   int sortieTube;
   char nomTube[] = "essai.fifo";

   char chaineALire[TAILLE_MESSAGE];

   if((sortieTube = open ("essai.fifo", O_RDONLY)) == -1)
   {
      fprintf(stderr, "Impossible d'ouvrir la sortie du tube nommé.\n");
      exit(EXIT_FAILURE);
   }

   read(sortieTube, chaineALire, TAILLE_MESSAGE);
```

```
        printf("%s", chaineALire);

    return EXIT_SUCCESS;
}
```

**Résultat :**

**Premier terminal** :

./ecrivain

_

**Deuxième terminal** :

./lecteur

Bonjour