Module Système L3/S6 – TD/TP no 3
Processus, tubes, redirections, recouvrements, signaux, mémoires partagées

Vous pouvez récupérer du code ici :
`http ://www.lisyc.univ-brest.fr/pages_perso/rodin/FTP/Enseignements/L3/Systeme/`
Le fichier à récupérer est : `TD-TP3.tar.gz`
Pour décompresser l'archive faire : `tar zxvf TD-TP3.tar.gz` ou `gunzip TD-TP3.tar.gz` et `tar xvf TD-TP3.tar`

1 Rappels sur les tubes

1.1 Partie 1 : tubes

- Un tube (ou pipe) est un moyen de communication qui permet à des processus d'échanger une suite d'octets.
- Un tube est géré comme une mémoire tampon de type FIFO. La taille d'un tube est égale à `PIPE_BUF` (voir `limits.h` ⇒ en général 4Ko ou 8Ko)
- Un tube est temporaire
- Les processus peuvent utiliser un tube via les fonctions classiques de lecture/écriture dans un fichier (`read`/`write` : <`unistd.h`>). Mais un tube n'est pas un fichier !

1.2 Partie 2 : Lecteur/Ecrivain

Lecteur : processus possédant un descripteur en lecture sur le tube
Ecrivain : processus possédant un descripteur en écriture sur le tube

`ssize_t read(int fd, void *buf, size_t len) ;`
- ◇ Retourne le nombre d'octets lus ou `-1` si erreur
- ◇ `fd` : descripteur de fichier,
- ◇ `buf` : buffer de lecture,
- ◇ `len` : taille du buffer de lecture.

`ssize_t write(int fd, const void *buf, size_t count) ;`
- ◇ Retourne le nombre d'octets écrits ou `-1` si erreur
- ◇ `fd` : descripteur de fichier,
- ◇ `buf` : message à écrire (buffer d'écriture),
- ◇ `count` : taille du message.

Remarque : un descripteur se ferme avec `int close(int fd) ;`

1.3 Partie 3 : Synchronisation des lectures/écritures dans un tube

- ◇ `read` est bloquant si :
  - le tube est vide et
  - il existe encore au moins un *écrivain* dans le tube
- ◇ `write` est bloquant si le tube est plein (`PIPE_BUF`)
- ◇ La *fin du tube* est atteinte si :
  - le tube est vide et
  - il n'existe plus d'*écrivain* dans le tube

  Dans ce cas, `read` retourne `0` pour indiquer la *fin du tube*.

1.4 Partie 4 : tubes sans nom et tubes nommés

Il existe deux types de tube :
- les tubes sans nom
- les tubes nommés

- ◇ Un tube sans nom est accessible uniquement par le processus créateur de ce tube et ses héritiers ...
- ◇ Un tube nommé possède un *nom externe* apparaissant dans l'arborescence de fichier (`ls -l` ⇒ `p`).
  ⟹ communication entre processus indépendants

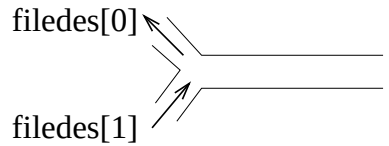
1.5 Partie 5 : tubes sans nom

Création :

`int pipe(int filedes[2]) ;` <`unistd.h`>
- ◇ Retourne `0` si ok ou `-1` si erreur
- ◇ `filedes` : tableau de 2 descripteurs de fichier. Après l'appel à `pipe`, `filedes` contient les descripteurs de fichier associés au tube :

  `filedes[0]` : descripteur ouvert en lecture sur le tube,

  `filedes[1]` : descripteur ouvert en écriture sur le tube.



1.6 Partie 6 : tubes nommés

Création :

`int mkfifo(const char *pathname, mode_t mode) ;`

- ◇ Retourne 0 si ok ou -1 si erreur
- ◇ `pathname` : nom externe du tube
  ⟹ création de la référence `pathname`
- ◇ `mode` : droits d'accès au tube nommé

pathname

Ouverture :

`int open(const char *pathname, int flags) ;`

- ◇ Retourne un descripteur ouvert sur le tube ou -1 si erreur
- ◇ `pathname` : nom externe du tube à ouvrir
- ◇ `flags` : O_RDONLY ⇒ lecture ; O_WRONLY ⇒ écriture

2 Exercice 1 : utilisation de tube sans nom

Vous disposez du code du programme programme suivant :

```
#include <stdio.h>              /* pipe.c */
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

#define TAILLE 256

void fils(int tube[2])
{
 int   ok = 0;
 char  ch[TAILLE];
                  /* Le fils n'ecrit pas dans le tube              */
 close(tube[1]);  /* FONDAMENTAL pour atteindre la "fin du tube" ! */

 ok=read(tube[0],ch,TAILLE);

 while ((ok!=-1) && (ok!=0))
 {
  printf("Recu %s",ch);
  ok=read(tube[0],ch,TAILLE); /* idem */
 }

 if (ok==-1) { perror("read"); exit(EXIT_FAILURE); }

 close(tube[0]);                    /* Pour faire propre */
}
```



```
void pere(int tube[2])
{
 int   ok = 0 ;
 char  ch[TAILLE];

 close(tube[0]);       /* Le pere ne lit pas dans le tube */

 printf("Entrez des lignes et terminez par fin : \n");

 fgets(ch,TAILLE,stdin);

 while (strcmp(ch,"fin\n")!=0)   /* fgets : \n a la fin ! */
 {
   ok = write(tube[1],ch,strlen(ch)+1);
   if (ok==-1) { perror("write"); exit(EXIT_FAILURE); }
   fgets(ch,TAILLE,stdin);
 }

 close(tube[1]);     /* Fermeture du tube ... FONDAMENTAL */

 wait(NULL);
}

int main(void)
{
 int   tube[2];

 pid_t pid = 0;

 if (pipe(tube) != 0) { perror("pipe"); exit(EXIT_FAILURE); }

 pid = fork();

 switch (pid)
 {
  case -1 : perror("fork");
            exit(EXIT_FAILURE);
            break;
  case  0 : fils(tube);
            break;
  default : pere(tube);
            break;
 }

 return EXIT_SUCCESS;
}
```

2.1 Analyse de ce programme

Etudiez et décrivez le fonctionnement de ce programme.

2.2 Modification de ce programme

Dans le programme que nous venons d'étudier, le processus fils affichait simplement les lignes lues dans le tube. Modifiez le programme pour que le processus fils compte le nombre de lignes arrivant par le tube (il n'affiche plus les lignes). Utilisez un `execlp` permettant de recouvrir avec le programme exécutable `wc`... il faut faire une redirection !

3 Exercice 2 : tube sans nom,recouvrement,signaux...

Nous souhaitons réaliser un programme qui indique le nombre d'octets présents dans un fichier. L'architecture globale demandée est bien compliquée par rapport à la tâche à accomplir mais, le but ici est de manipuler plusieurs concepts comme la création de processus, la communication par tube, la redirections d'entrée/sortie, le recouvrement de processus, la gestion des signaux, la notion de mémoire partagée, ...

3.1 Gestion de la mémoire partagée

Afin de simplifier le code, vous disposez d'un module `partage.h/partage.c` vous permettant de gérer facilement une mémoire partagée entre des processus ayant un lien de parenté (exemple : père–fils).
Etudier son fonctionnement pour l'utiliser dans votre programme.

```
#ifndef PARTAGE_H              /* partage.h */
#define PARTAGE_H

/* Structure zone representant une zone partagee entre processus ayant
   un lien de parente. Exemple: pere-fils => cle privee d'IPC Sys V    */

typedef struct
{
 int    id;
 void * debut;
 int    taille;
} Zone;

/* fonctions publiques */
extern int creerZonePartagee(int taille, Zone * zp);
extern int supprimerZonePartagee(Zone * zp);

#endif /* PARTAGE_H */
```



```
#include <stdio.h>             /* partage.c */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "partage.h"
/* ------------------------------------------------
   creerZonePartagee(int taille, Zone *zp)

   Action   : creer une zone de memoire partagee grace
              aux IPC SysV, accesible un lecture/ecriture
              par le createur. La taille est en octets.
   Remarque : cette zone est heritee par un processus fils
   Retourne : -1 en cas d'erreur, 0 sinon
              remplit la structure zone passee en argument
 ------------------------------------------------ */
int creerZonePartagee(int taille, Zone * zp)
{
 if ((zp->id=shmget(IPC_PRIVATE, taille, 0600|IPC_CREAT))==-1)
      { perror("shmget (creerZonePartagee)"); return -1; }

 if ((zp->debut=shmat(zp->id, 0, 0))==(void*)-1)
      { perror("shmat (creerZonePartagee)");
        shmctl(zp->id, IPC_RMID, NULL);
        return -1; }

 zp->taille=taille;

 return 0;
}
/* ------------------------------------------------
   int supprimerZonePartagee(Zone *zp)

   Action   : liberer la zone partagee
   Retourne : -1 en cas d'erreur, 0 sinon
  ------------------------------------------------ */
int supprimerZonePartagee(Zone * zp)
{
 if (shmdt(zp->debut)==-1)
     { perror("shmdt (supprimerZonePartagee)"); return -1; }

 if (shmctl(zp->id, IPC_RMID, NULL)==-1)
     { perror("shmctl (supprimerZonePartagee)"); return -1; }

 return 0;
}
```

3.2 Architecture du programme

Nous souhaitons donc écrire un programme qui indique le nombre d'octets (caractères) d'un fichier passé en paramètre. Pour cela, nous invoquerons avec `execlp` un programme externe dont nous redirigerons la sortie. La figure 1 illustre le principe du programme. Un squelette commenté de programme est disponible dans l'archive `TD-TP3.tar.gz`.

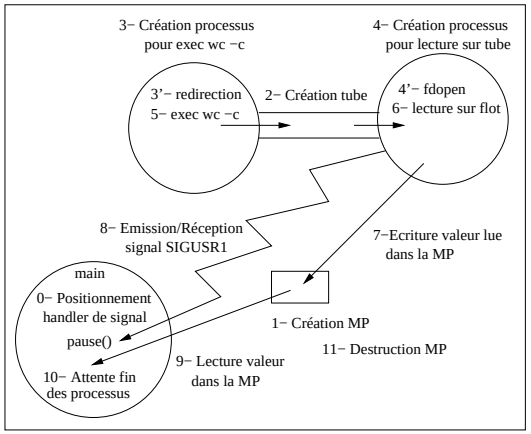


FIG. 1 – Principales opérations à réaliser

⇒ Le processus père commence par une phase d'initialisation permettant de :
- Décrire son comportement lors de la réception d'un signal de type `SIGUSR1`. Pour cela, il n'utilise pas la fonction `signal`. Il utilise à la place la fonction `sigaction` POSIX (voir plus loin).
- Créer une mémoire partagée pouvant contenir un `int`.
  Utilisation de `partage.h/partage.c` :
```
Zone z;
if (creerZonePartagee(sizeof(int),&z)==-1) ...
/* Pour acceder a la zone de memoire partagee : *((int*)z.debut) */
supprimerZonePartagee(&z);
```
- Créer un tube qui permettra un échange d'informations entre deux processus créés à l'étape d'après.
- Créer les deux processus.

⇒ Lorsque le processus père a termininé cette phase d'initialisation, il attend l'arrivée d'un signal (émis par le deuxième processus créé). Cette attente est effectué à l'aide de la fonction `pause`. L'arrivé du signal lui indique que des données sont disponibles dans la mémoire partagée. Il peut alors lire ces données dans la mémoire partagée et terminer proprement (attente des deux processus fils, destruction de la mémoire partagée).

⇒ Le premier processus créé commence par rediriger sa sortie standard vers le tube (`tube[1]`). Puis il ferme le(s) descripteur(s) inutile(s). Enfin il se recouvre avec un appel à `execlp` pour exécution de `wc -c fileName`.

⇒ Le deuxième processus créé va donc devoir lire le résultat du `wc -c fileName`, résultat qui va arriver par le tube (`tube[0]`). L'extraction du résultat pourra être grandement facilitée par l'utilisation d'un flux (`FILE *`) en lieu et place d'une lecture directe dans un tube. Ce flux est obtenu facilement à l'aide de `fdopen` :

```
FILE *fIn;
if ((fIn=fdopen(tube[0],"r"))==NULL) ...
fscanf(fIn,...
```

Ainsi, aprs avoir obtenu ce flux `fIn`, le deuxième processus peut lire dans le flux avec `fscanf`. Ensuite, il ferme le flux et les descripteurs inutiles. Finalement, il peut mettre le résultat obtenu (un `int`) dans la mémoire partagée et avertir le processus père que le résutat est disponible (envoi d'un signal).

3.3 Rappels sur la gestion des signaux

A la réception d'un signal, un processus peut :
- avoir son comportement par défaut ⇒ mort du processus
- ignorer ce signal
- appeler une fonction particulière (i.e. un handler de signal)

Pour spécifier son comportement à la réception d'un signal particulier, le processus doit indiquer `SIG_DFL`, `SIG_IGN` ou bien un handler de signal (une fonction).

Il existe deux façons de procéder : avec `signal` ou avec `sigaction`.

Soit le handler suivant :

```
void handler(int signum)
{
...
}
```

Avec `signal` :

```
int main(...)
{
 ...
 if (signal(SIGUSR1,handler)==SIG_ERR)
 ...
 return 0;
}
```

Avec `sigaction` :

```
int main(...)
{
 struct sigaction action;
 ...
 action.sa_handler=handler;
 ...
 if (sigaction(SIGUSR1,&action, NULL)==-1)
 ...
 return 0;
}
```

A l'aide de la fonction `sigaction` (POSIX), il est possible de décrire finement le comportement du processus à la réception d'un signal. Ce Comportement étant le même sur toutes les machines (ce n'est pas vrai avec `signal`).

3.4 Squelette du programme

```
#include <stdio.h>       /* nbOctets.c */
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#include "partage.h"

/* Decrire le handler de signal pour SIGUSR1 */
/* ======================================== */

/* ... */

/* Le main */
/* ======= */

int main(int argc,char **argv)
{
 pid_t  pidWC;
 pid_t  pidREAD;
 int    status;   /* Pour les waitpid                       */

 int    tube[2];
 FILE  *fIn;      /* Pour faire un fdopen : int -> FILE *    */

 struct sigaction action;
```



```
 Zone     z;
 int *ptDeb;      /* Un pointeur (int*) sur la zone debut    */

 char  *fileName=NULL;

 if (argc!=2) { fprintf(stderr,"Usage: %s fileName\n",argv[0]); return 1; }

 fileName=argv[1];

/* A cause de warnings lorsque le code n'est pas encore la ...*/

 (void)action; (void)fIn; (void)tube; (void)status; (void)pidREAD; (void)pidWC;

/* Gestion des signaux */
/* =================== */

  /* Preparation de la structure action pour recevoir le signal SIGUSR1 */

/* action.sa_handler = ... */

  /* Appel a l'appel systeme sigaction */

/* ... */

/* Creation de la zone de memoire partagee */
/* ======================================= */

/* ... */

 ptDeb=(int*)z.debut;    /* *ptDeb <=> *((int*)z.debut) */

/* Creation du tube */
/* ================ */

/* ... */

/* Creation du processus qui fera le exec ...   */
/* ============================================ */

/* pidWC=... */

   /* Dans le processus cree :
      - Rediriger la sortie standard vers le tube
      - Fermer le(s) descripteur(s) inutile(s) a cet enfant
      - Recouvrir par la commande ``wc''
   */
```

```
/* Creation du processus qui fera la lecture ...*/
/* ============================================ */

/* pidREAD=... */

   /* Dans le processus cree :
      - Fermer le(s) descripteur(s) inutile(s) a cet enfant
      - Ouvrir un flux fIn sur la sortie du tube: fIn=fdopen(tube[0],"r");
      - Lire le resultat via le flux fIn et le mettre dans la memoire partagee
      - Fermer le flux fIn et le(s) descripteur(s) encore ouvert(s)
      - Attendre un peu pour que le pere puisse faire pause avant
      - Envoyer le signal SIGUSR1 au pere
   */

/* La suite du pere */
/* ================ */

  /* Fermer les descripteurs de tube inutiles au pere */

/* ... */

  /* Attente d'un signal */

/* ... */

  /* Recuperer le resultat dans la memoire partagee */

/* ... */

  /* Attendre le 1er enfant  */

/* ... */

  /* Attendre le 2eme enfant */

/* ... */

  /* Supprimer la memoire partagee */

/* ... */

 return 0;
}
```