UNIVERSITÉ CAEN NORMANDIE

La Communication entre processus

06.11.2018

Inter Processus Communications (I.P.C.)

Les tubes et la Mémoire partagée

Auteur : Pascal Fougeray

1 Introduction

Nous avons vu dans le cours sur les processus qu'il était nécessaire de faire communiquer le père avec un ou plusieurs de ses fils, voir même de faire communiquer 2 fils entre eux. Pour l'instant nous avons vu que l'on ne pouvait que simplement obtenir le code de sortie d'un processus fils, ce qui est insuffisant et ne permet qu'une communication "primaire" entre un père et un fils.

Nous allons voir dans ce cours, qu'il existe 5 moyens de communication, les 2 premiers outils appartiennent au système de gestion de fichiers, les 2 suivants font partie de la famille des IPC, Communication Interprocessus (Interprocess Communication). Le dernier, les *sockets*, est principalement utilisé dans la communication Inter machines et est une "sur couche" des tubes.

1. Les **tubes anonymes** : permettent une communication séquentielle et unidirectionnelle d'un processus à l'autre.
2. Les **files de messages FIFO** (***First In First Out***) : similaires aux tubes avec pour avantage de permettre à 2 processus sans lien de parenté de communiquer, car ce type de tube est référencé dans le système de fichiers, on parle aussi de **tubes nommés**.
3. La **mémoire partagée** : les processus communiquent simplement en lisant ou en écrivant dans un emplacement mémoire se trouvant dans la mémoire de travail (RAM) partagé entre 2 processus ou plus

4. La **mémoire mappée** : similaire à la mémoire partagée, à la différence que l'emplacement mémoire est associé à un fichier. Il n'est donc plus dans la mémoire de travail (RAM) mais dans la mémoire de stockage (DD, SSD, Cloud)
5. Les ***sockets***[1] : permettent la communication entre des processus sans lien et pouvant se trouver sur des machines distantes.

Remarque : vu le temps imparti pour ce module nous n'aurons pas le temps de tout étudier..., mais si vous comprenez un mécanisme, vous comprendrez "rapidement" les autres. De plus la curiosité n'étant pas un vilain défaut, vous pouvez vous les approprier seul(e)...

2 Utilisations

Nous pouvons citer différents cas d'utilisations, exemples :

— On désire imprimer le contenu d'un répertoire, en shell la commande serait ***ls - l | lpr***, on constate la présence de 2 processus distincts qui sont créés par le shell ***ls*** et ***lpr*** et qui communiquent par l'intermédiaire d'un tube ou pipe représenté par **|**. Ici la communication est unidirectionnel, le processus *ls* produit des données, on va considérer qu'il est le producteur et les écrit dans l'entrée du tube, alors que le processus *lpr* consomme des données qu'il lit à la sortie du tube.

— Lors de la communication via le réseau avec des programmes tels Telnet, ftp, Talk etc..., on utilise des *sockets* et la communication est cette fois-ci bidirectionnelle. Nous avons 2 processus qui communiquent l'un étant généralement le serveur et l'autre le client.-

3 Rappels

— Les descripteurs de fichier (***file descriptor***) : c'est un entier positif ou nul permettant d'identifier une entrée-sortie en cours.

— Les descripteurs de fichier constituent l'information de plus bas niveau manipulée lors de la programmation des entrées-sorties. Tout appel système effectuant une opération sur une entrée-sortie en cours reçoit comme paramètre principal le descripteur de fichier de cette entrée-sortie.
Les 3 entrées-sorties de bas niveau associées à l'entrée (**STDIN**), la sortie (**STDOUT**) et la sortie d'erreur (**STDERR**) ont respectivement **pour descripteur de fichier les entier 0,1 et 2.**
Moyen mnémotechnique pour se souvenir si 0 c'est STDIN ou bien STDOUT, on entre avant de sortir et 0 est avant 1, ok ?

— **Lecture** : la lecture dans une entrée-sortie de bas niveau se fait par l'intermédiaire de la primitive **read()**.
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
read() lit jusqu'à **count** octets depuis le descripteur de fichier **fd** dans le tampon pointé par **buf**.

— **Écriture** : l'écrire dans une entrée-sortie de bas niveau se fait par l'intermédiaire de la primitive **write()**.
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
write() lit au maximum ***count*** octets dans la zone mémoire pointée par ***buf***, et les écrit dans le fichier référencé par le descripteur **fd**
On peut donc faire un ***printf*** sans ***stdio.h*** ! ! !, Bon ce n'est pas pratique mais ça marche !

```
#include <unistd.h>
int main(int argc, char *argv[]){
char *s="Hello World! \n";
// 14 caractères à envoyer à STDOUT
write(1, (const void*)s, 14);
return 0;
}
```

4 Les tubes

Un tube (***pipe***) est un dispositif de communication qui permet une communication **unidirectionnelle** entre 2 processus, généralement le père et un de ses fils. Les données écrites sur l'extrémité d'écriture du tube sont lues

1. Voir le cours sur les sockets un peu plus tard...

depuis l'extrémité de lecture. Les tubes sont donc des dispositifs séquentiels ; les données sont toujours lues dans l'ordre où elles ont été écrites. C'est donc une mémoire à accès séquentiel !

Attention une fois que le sens d'utilisation du tube est choisi, celui-ci ne peut plus être changé. Donc un processus lecteur d'un tube ne peut pas devenir écrivain dans ce tube et vice-versa.

Comme cela a été dit dans l'introduction, les tubes sont gérés par le système au niveau du système de fichiers et correspondent à un fichier au sein de celui-ci. Lors de la création d'un tube, 2 descripteurs de fichier sont créés, permettant respectivement de lire et écrire dans le tube.

Les données dans le tube sont gérées en flots d'octets, sans préservation de la structure des messages déposés dans le tube, avec pour stratégie : Premier entré, Premier Sorti [2].

Les lectures sont destructives, une donnée ne peut être lue que par un seul lecteur, à moins qu'il y ait duplication du descripteur de fichier (**dup**).

Le tube a une capacité finie qui est celle du tampon alloué. Cette capacité est définie par la constante PIPE_BUF dans le fichier <limits.h>, elle vaut 4096 octets. Si un tube est plein, le processus écrivain est alors bloqué en attendant de pouvoir réaliser une écriture.

Si un tube est vide, le processus lecteur est alors bloqué en attendant de pouvoir réaliser une lecture. Cependant, la lecture peut être non bloquante en émettant un appel à la fonction ***fcntl****()(descripteur_lecture[0], F_SETL, O_NONBLOCK)*.

Si tous les descripteurs de fichiers correspondant à l'entrée d'un tube sont fermés, une tentative d'écriture provoquera l'envoi du signal SIGPIPE au processus appelant.

4.1 Les tubes anonymes

Les tubes anonymes sont gérés par le système de fichiers et correspondent donc à un fichier sans nom. Ils ne peuvent donc être manipulés que par les processus ayant connaissance de leurs 2 descripteurs en lecture et écriture qui leur sont associés. Ce sont donc le père et tous ses descendants **créés après** la création du tube et qui prennent connaisance des descripteurs du tube par héritage des données du père.

4.1.1 Utilisation d'un tube anonyme

Il faut **créer** le tube, **lire** et **écrire** dedans, et enfin le **fermer**.
Un tube anonyme est créé en appelant la primitive ***pipe***.
Voici son prototype

#include <unistd.h>
int pipe(int descripteur_fichier[2])

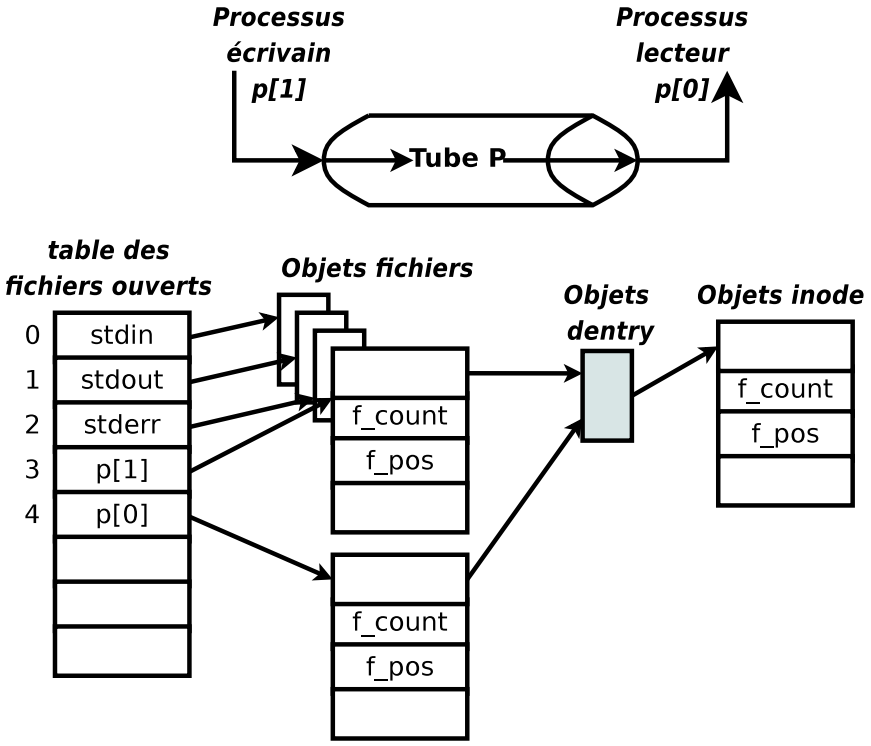
La primitive retourne 2 descripteurs dans le tableau ***descripteur_fichier***.

1. *descripteur_fichier[**0**]* correspondant au descripteur utilisé pour la lecture dans le tube.
2. *descripteur_fichier[**1**]* correspondant au descripteur utilisé pour l'écriture dans le tube.

Les 2 descripteurs sont alloués dans la table des fichiers ouverts du processus et pointent respectivement sur un objet fichier en lecture et un objet fichier en écriture. Le tube est représenté au sein du système par un *inode* auquel n'est associé aucun bloc de données, les données transitant dans le tube étant placés dans un tampon en mémoire centrale.

Tout processus ayant connaissance du descripteur *descripteur_fichier[**0**]* peut lire dans le tube
et de même
Tout processus ayant connaissance du descripteur *descripteur_fichier[1]* peut écrire dans le tube,
comme le montre la figure suivante.

2. On ne se double pas dans un tube, c'est interdit :)



Un tube anonyme est fermé lorsque tous les descripteurs en lecture et en écriture existants sur ce tube sont fermés. Un processus utilise pour cela la primitive :

#include <unistd.h>

int close(int fd);

— Le nombre de descripteurs ouverts en lecture détermine le nombre de lecteurs existants.

— Le nombre de descripteurs ouverts en écriture détermine le nombre d'écrivains existants.

— **Un descripteur fermé ne permet plus d'accéder au tube et cela est définitif.**

La lecture et l'écriture dans un tube se font à l'aide des primitives read() et write().

4.1.2 Exemples d'utilisation de tubes anonymes

Communication Unidirectionnelle

Dans cet exemple, la communication se fait entre un père et son fils.
Le fils écrit à destination de son père la chaine de caractères "***bonjour***".
Les étapes du programmes sont les suivantes.

1. Le père crée un tube à l'aide de la fonction **pipe**(),
2. Le père crée un fils, donc le fils hérite du tube ou a le sien ?
3. Les descripteurs en lecture [0] et en écriture [1] sont utilisables par les 2 processus.
   Chacun des 2 processus ferme le descripteur qui lui est inutile.
   Celui en écriture pour le père et bien sûr celui en lecture pour le fils.
4. Le fils envoie un message à son père.
5. Le père lit le message.

Remarque : il n'y a pas besoin de synchroniser le père puisque le père doit attendre que le fils écrive dans le tube avant de pouvoir lire.

/ tube_uni_père_fils.c */*

```
3 #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>
  #include <unistd.h>
```

```c
int main(){
  int pip[2];
  pid_t pid;
  char chaine[]="Avant..";
  int taille;
  pipe(pip); //on cré un tube

  if((pid=fork())==0) {//dans le fils
    // on fait patienter le père pendant 2s
    // montre que la lecture est blocante...
    sleep(2);
    char chaine_fils[]="Pour toi papa :)";
    close(pip[0]); // le fils ne lit pas
    write(pip[1], &chaine_fils, strlen(chaine_fils));
    // le fils ferme l'entrée avant de quitter
    close(pip[1]);
    exit(0);
  }
  else { //dans le père
    char chaine_papa[]="Je vais etre effacée";
    taille_chaine = strlen(chaine_papa);
    printf("taille : %d\n", taille_chaine);
    memset(chaine_papa,'\0',taille_chaine);
    //Que des caractères NULL
    printf("Chaine avant: %s\n",chaine_papa);
    close(pip[1]); // le père n'écrit pas
    read(pip[0], chaine_papa, taille);
    // le père ferme la sortie avant de quitter
    close(pip[0]);
    printf("Chaine après: %s\n",chaine_papa);
  }
  return 0;
}
```

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/tubes# ./tube_uni_père_fils &
[1] 16172
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/tubes# taille : 21
Chaine avant:
```

chaine vide !!!

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/tubes# ps jf
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
15673 16008 16008 16008 pts/1    16008 Ss+      0   0:00 -bash
15673 15679 15679 15679 pts/0    16174 Ss       0   0:00 -bash
15679 16172 16172 15679 pts/0    16174 S        0   0:00  \_ ./tube_uni_père_fils
16172 16173 16172 15679 pts/0    16174 S        0   0:00  |   \_ ./tube_uni_père_fils
15679 16174 16174 15679 pts/0    16174 R+       0   0:00  \  ps jf

root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/tubes# lsof | grep tube_uni_père_fils
tube_uni_ 16172                root  txt       REG                8,1      9112     809255 /root/TD-TP-Systeme/TD_TP/tubes/tube_
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/tubes# Chaine après: Pour toi papa :)
```

père et fils

Communication Bidirectionnelle

Dans ce second exemple, la communication se fait toujours entre un père et son fils. Mais cette fois-ci le père et le fils écrivent et lisent dans un ou l'autre tube, mais pas les 2.

La communication bidirectionnelle nécessite l'utilisation de 2 tubes, chaque tube étant utilisé dans le sens inverse de l'autre.

Il faut ici synchroniser le père avec le fils car si le père se termine avant le fils, il quitte et le shell reprend la main et la sortie n'est pas tip top, d'où le sleep

```c
/* tube_bi_père_fils.c */

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

```
int main(){
  int pip1[2];
  int pip2[2];
  pid_t pid;
  char chaine[12]="Chaine vide.";
  pipe(pip1); //on cré un tube
  pipe(pip2); //puis un second
  if((pid=fork())==0) {// le fils
    printf("le fils : %s \n",chaine);
    //ne lit pas dans pip1
    close(pip1[0]);
    //n'écrit pas dans pip2
    close(pip2[1]);
    write(pip1[1], "Bonjour Papa", 12);
    close(pip1[1]);
    read(pip2[0],chaine, 12);
    close(pip2[0]);
    printf("Fils reçoit : %s \n",chaine);
    exit(0);
  }
  else { // le père
    printf("le père : %s\n",chaine);
    //n'écrit pas dans pip1
    close(pip1[1]);
    //ne lit pas dans pip2
    close(pip2[0]);
    read(pip1[0], chaine, 12);
    close(pip1[0]);
    write(pip2[1],"Bonjour Fils",12);
    close(pip2[1]);
    printf("Père reçoit : %s\n",chaine);
    waitpid(pid, NULL, 0); // pour que le père attende son fils !!!
    }
  return EXIT_SUCCESS;
}
```

4.2 Les tubes nommés

Les tubes nommés (***pipe named***) sont aussi gérés par le système de fichiers et correspondent donc à un fichier avec nom. Ils sont donc accessibles par n'importe quel processus connaissant ce nom et ayant les droits d'accès nécessaires. Cela va permettre à 2 processus **sans lien de parenté** de pouvoir communiquer selon un mode flots d'octets.

Les tubes nommés ont les mêmes propiétés que les tubes anonymes. Leur création est réalisée à l'aide d'une primitive différente de celle des tubes anonymes : **mkfifo**() dont voici le prototype.

#include <sys/types.h> #include <sys/stat.h>
int mkfifo (const char *pathname, mode_t mode);

Une fois créé, un tube nommé persiste dans le système de fichiers, s'il n'est pas détruit par le lecteur. Il suffit d'utiliser la primitive **unlink**() à la fin du processus lecteur de **préférence** pour détruire un tube nommé.

#include <unistd.h>
int unlink(const char *path);

4.2.1 Exemples d'utilisation de tubes nommés

Voici 2 sources montrant un écrivain écrivant dans un tube nommé nommé *fichier_tube_nomme* et un lecteur lisant dans ce même tube.

```
// Processus ecrivain dans le tube nommé
/* ecrivain_tube_nomme.c */
```

```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(){
  mode_t mode;
  int tube;
  char chaine[]="Bonjour passe dans le tube";
  mode = S_IRUSR | S_IWUSR;
  // Création du tube nommé
  mkfifo("fichier_du_tube", mode);
  // Ouverture du tube
  tube = open("fichier_du_tube", O_WRONLY);
  // Ecritude dans le tube
  write(tube, chaine, strlen(chaine));
  // Fermeture du tube
  close(tube);
}
```

```c
// Processus lecteur dans le tube nommé
/* lecteur_tube_nomme.c */

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(){
  int tube;
  char chaine[30];
  // Ouverture du tube
  tube = open("fichier_du_tube", O_RDONLY);
  // Ecritude dans le tube
  read(tube, chaine, 26);
  chaine[29]=0;
  printf("Processus Lecteur du tube ");
  printf("fichier_du_tube : %s \n", chaine);
  // Fermeture du tube
  close(tube);
  // détruit le fichier,
    //ici en commentaire pour le voir avec la commande ls
  // unlink("fichier_du_tube");
}
```

Un ls -l du répertoire, montre la présence du fichier *fichier_tube_nomme* avec la lettre **P** en tête des droits d'accès. P comme Pipe...

4.3 La duplication

Les tubes peuvent aussi être utilisés au niveau du *Shell* pour transmettre le résultat d'une commande à l'autre, ceci afin de réaliser des redirection d'entrée-sortie .

Par exemple la commande *ls -l | wc -l* permet de connaître le nombre de fichiers présent dans un répertoire.

L'opérateur | représente un tube

Les commandes ls -l et wc -l sont 2 processus dont la sortie standart *STDOUT* du premier est redirigée vers l'entrée du tube et l'entrée standart *STDIN* du second reçoit des données provenant de la sortie du tube.

Pour réaliser ces redirections on utilise la primitive **dup**() dont le prototype est :

#include <unistd.h>

int dup(int fildes);

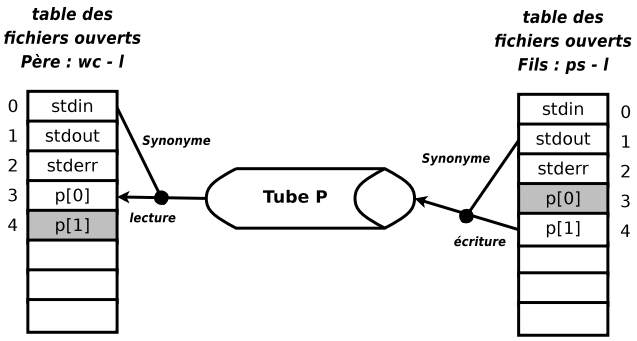
Cette primitive associe **le plus petit** descripteur **disponible** du processus appelant à la même entrée dans la table des fichiers ouverts que le descripteur *fildes*.

Pour rediriger un descripteur standard sur un descripteur de tube, il faut :

1. **Fermer** le descripteur standard
2. **Faire** l'appel ***dup*** avec comme paramètre le descripteur du tube concerné.
3. **Fermer** le descripteur du tube. Seul reste le descripteur standard rendu synonyme.

Dans l'exemple suivant nous avons un programme constitué de 2 processus,

1. le père qui exécute la commande *wc -l*
2. le fils la commande *ps -l*.



- Le père :
  - wc -l lit sur STDIN
  - Il faut rediriger l'entrée de wc -l sur pip[0]
  - STDIN = pip[0]
  - close (STDIN)
  - dup (pip[0])
  - close (pip[0])
- Le fils
  - ps -l écrit sur STDOUT
  - Il faut rediriger la sortie de ps -l sur pip[1]
  - STDOUT = pip[1]
  - close (STDOUT)
  - dup (pip[1])
  - close (pip[1])

Cet exemple montre un cas d'utilisation de la primitive **dup**().

Il existe aussi la primitive dup2() qui permet non pas d'associer le plus petit descripeur disponible du processus appelant, mais permet de choisir les 2 descripteurs que l'on va utiliser.

#include <unistd.h>

int dup2(int oldfd, int newfd);

dup2(int oldfd, newfd); transforme *newfd* en une copie de *oldfd*, fermant auparavant *newfd* si besoin est.

```
/* La commande ls -l | wc -l */
/* fichier tube_dup.c */

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

main(){
  int p[2];
  //création d'un tube
  if(pipe(p)){
    perror("pb pipe");
  }
    //le P et le F connaissent le tube
 switch(fork()){
 case -1: //erreur fork
   perror("pb fork");
   exit(2);
 case 0: //fils exécute ls -l
    // Sa sortie standard est redirigée
    // sur l'entrée du tube
    close(STDOUT_FILENO); //ou close(1);
    (void) dup(p[1]); // Donc la sortie
    // standard est l'entrée du tube.
    close(p[1]);
    // Fils ne lit pas dans le tube
    close(p[0]);
    execlp("ls", "ls", "-l", NULL);
    default : //Père exécute wc -l
    // Son entrée standard est redirigée
    // sur la sortie du tube
    close(STDIN_FILENO); //ou close(0);
    (void) dup(p[0]); // Donc l'entrée
    // standard est la sortie du tube.
    close(p[0]);
    // Père n'écrit pas dans tube
   close(p[1]);
   execlp("wc", "wc", "-l", NULL);
 }
}
```

5 La Mémoire Partagée

Une des méthodes de communication inter-processus les plus simples est d'utiliser la mémoire partagée.

La mémoire partagée permet à **n processus d'accéder à la même zone mémoire** comme s'ils avaient appelé la primitive ***malloc*** et avaient obtenu des pointeurs vers le même espace mémoire. Lorsqu'un processus modifie la mémoire, les autres processus voient la modification.

— Avantage : Communication locale **rapide**
La mémoire partagée est la forme de communication inter-processus la plus rapide car les processus partagent la même mémoire. L'accès à cette mémoire partagée est aussi rapide que l'accès à la mémoire non partagée du processus et ne nécessite aucun appel système ni d'entrée (et pas d'entrer ! ! !) dans le noyau. Elle évite aussi les copies de données inutiles.

— Inconvénient : **synchronisation**
Comme le noyau ne coordonne pas les accès à la mémoire partagée, c'est au programmeur de mettre en place sa propre synchronisation.
Exemple : un processus ne doit pas effectuer de lecture avant que des données aient été écrites et 2 processus ne doivent pas écrire au même emplacement en même temps. Une solution, pour éviter cette concurrence est d'utiliser des sémaphores[3],. Dans ce cours, les exemples ne montrent qu'un seul processus accédant à la mémoire, afin de se concentrer sur les mécanismes de la mémoire partagée et éviter d'alourdir le code avec la logique de synchronisation.

Le modèle

5.1 Allocation

5.2 Attachement et Détachement

Pour rendre le segment de mémoire partagée disponible, un processus doit utiliser l'appel système ***shmat*** () (pour ***SHared Memory ATtach*** ou attachement de mémoire partagée) en lui passant :

— l'identifiant du segment de mémoire partagée **SHMID** renvoyé par ***shmget***().
— Le second argument est un pointeur qui indique où vous voulez que le segment soit mis en correspondance dans l'espace d'adressage de votre processus ;
  — si **NULL** alors le noyau sélectionnera une adresse disponible.
— Le troisième argument est un indicateur, qui peut prendre une des valeurs suivantes :
  — **SHM_RND** indique que l'adresse spécifiée par le second paramètre doit être arrondie à un multiple inférieur de la taille de page. Si vous n'utilisez pas cet indicateur, on doit aligner le second argument de ***shmat*** sur un multiple de page vous-même.
  — **SHM_RDONLY** indique que le segment sera uniquement lu, pas écrit.

Si l'appel se déroule correctement, il renvoie **l'adresse** du segment partagé attaché.

Les processus **fils** créés par des appels à ***fork*** héritent des segments partagés attachés ; il peuvent les détacher !

Lorsque vous en avez fini avec un segment de mémoire partagée, le segment doit être détaché en utilisant ***shmdt()*** (***SHared Memory DeTach*** , Détachement de Mémoire Partagée) en lui

passant l'adresse renvoyée par ***shmat***. Si le segment n'a pas été libéré et qu'il s'agissait du dernier processus l'utilisant, il est supprimé.

Remarque : Les appels à ***exit()*** et toute fonction de la famille ***d'exec*** détachent automatiquement les segments.

5.3 Contrôler et libérer la mémoire partagée

L'appel ***shmctl*** (***SHared Memory ConTroL***, contrôle de la mémoire partagée) renvoie des informations sur un segment de mémoire partagée et peut le modifier.

— Le premier paramètre est l'identifiant d'un segment de mémoire partagée.
— Pour obtenir des informations sur un segment de mémoire partagée, passez **IPC_STAT** comme second argument et un pointeur vers une ***struct shmid_ds***.
— Pour supprimer un segment, passez **IPC_RMID** comme second argument et **NULL** comme troisième argument. Le segment est supprimé lorsque le dernier processus qui l'a attaché le détache.

Chaque segment de mémoire partagée doit être explicitement libéré en utilisant ***shmctl*** lorsqu'on en a terminé avec lui, afin d'éviter de dépasser la limite du nombre total de segments de mémoire partagée définie par le système.

Remarque : Les appels ***exit***() et ***exec***() détachent les segments mémoire mais ne les libèrent pas.

3. et là ça se complique et nous n'aurons pas le temps...

5.4 Exemple de partage de segment de mémoire

Cet exemple provient de http://www.lifl.fr/~sedoglav/OS/main021.html
Il se décompose de plusieurs programmes et montre aussi l'utilité des sémaphores à la fin.

1. Créer un segment

```
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int creeSegment(int size, char *name, int cle){
  int shmid ; // l'identificateur de la memoire partagee
  key_t clef ; // la clef associee au segment
  /* L'instruction ftok(name,(key_t) cle) permet de construire
     une cle identifiant le segment */
  clef = ftok(name,(key_t) cle) ;
  /* L'instruction IPC_CREAT|IPC_EXCL|SHM_R|SHM_W permet d'indiquer
     les droits d'acces de ce segment de memoire */
  shmid = shmget( clef,
                  size,
                  IPC_CREAT|IPC_EXCL|SHM_R|SHM_W ) ;
  if ( shmid== -1 ) {
    perror("La creation du segment de memoire partage a echouee") ;
    exit(1) ; // On sort
  }
  printf("l'identificateur du segment est %d \n",shmid) ;
  printf("ce segment est associe a la clef %d \n",clef) ;
  return shmid ;
}

int main(){
  char *name = (char *) malloc(100*sizeof(char)) ;
  name = "memoirePartageeAmoi";
  creeSegment(100,name,1);
  // pour avoir le temps de faire ipcs et ipcrm shm xxxxxxxx
  sleep(100);
  return 0;
}
```

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage# ./creer-segment &
[1] 16903
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage# l'identificateur du segment est 3276812
ce segment est associe a la clef -1
^C
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage# ipcs -m

------ Segment de mémoire partagée --------
clef       shmid      propriétaire perms      octets     nattch     états
0x00000000 163840     root       600        393216     2          dest
0x00000000 196609     root       600        393216     2          dest

0x00000000 1409034    root       600        12288      2          dest
0x00000000 3211275    lightdm    600        33554432   2          dest
0xffffffff 3276812    root       600        100        0

root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage# ipcrm shm 3276812
ressources supprimées
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage#
```

tue le processus et on voit le segment reste

```
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage# ./exemple
l'identificateur du segment est 3506188
ce segment est associe a la clef -1
Je suis le pere
Je commence par m'attacher le segment de memoire
je vais afficher un message que mon fils a ecrit

Sacree plaisantin ce fiston
Bon, c'est pas tout ca mais il est temps de mourrir
avant tout detachons le segment partageeLe pere attend la mort de son fils
Salut, je suis le fils 16998
Je commence par m'attacher le segment de memoire
je vais ecrire un message que papa va afficher
Bon, c'est pas tout ca mais il est temps de mourrir
avant tout detachons le segment partagee
maintenant que c'est fait, bye bye
Le fils se suicide
Bon, faisons le menage et supprimons le segment partagee
Le pere se suicide
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage#

root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage# ./exemple-ok
l'identificateur du segment est 3571724
ce segment est associe a la clef -1
On cr\'ee un s\'emaphore dont le num\'ero est 32768
pere: je me bloque en faisant ``down'' sur le semaphore
fils: Salut, je suis le fils 17174
fils: Je commence par m'attacher le segment de memoire
fils: je vais ecrire un message que papa va afficher
fils: Bon, c'est pas tout ca mais il est temps de mourrir
fils: avant tout detachons le segment partagee
fils: on va liberer papa
pere: Je commence par m'attacher le segment de memoire
pere: je vais afficher un message que mon fils a ecrit
La vie est belle
pere: Sacree plaisantin ce fiston
pere: Bon, c'est pas tout ca mais il est temps de mourrir
pere: avant tout detachons le segment partagee
pere: Le pere attend la mort de son fils
fils: maintenant que c'est fait, bye bye
fils: Le fils se suicide
pere: Bon, faisons le menage et supprimons le segment partagee
destruction du s\'emaphore 32768
pere: Je suis le pere et je viens de detruire le semaphore
pere: Le pere se suicide
root@debian95-Rx-Sys-Fougeray:~/TD-TP-Systeme/TD_TP/Mem-partage#
```

5.5 Débogage

6 Conclusion

Les tubes qu'ils soient anonymes ou nommés sont un bon moyen de communication entre 2 processus ayant ou n'ayant pas de lien de parenté. C'est même un moyen très fiable et très simple à mettre en oeuvre, occupant très peu de ressources système.

Néanmoins, leur domaine d'utilisation reste limité à des processus tournant sur une seule machine. Si on veut faire communiquer des processus tournant sur des machines distantes, il faudra utiliser les sockets et se lancer dans la programmation réseau. Chose que nous allons faire dans les prochains cours...