TRAVAUX PRATIQUES 2
Outils de communication interne entre processus sous Linux

L'objectif de ce TP est de voir les principaux outils de communication fournit par les systèmes de type Unix pour permettre la communication entre processus sur une même machine.
Nous rappelons qu'un processus est la représentation de la dynamique d'exécution d'un programme (ou partie d'un programme si celui-ci créé durant son exécution plusieurs autres processus).

Dans ce TP vous aurez à écrire des programmes en langage C pour utiliser les primitives systèmes. Pour cela, vous pouvez utiliser un éditeur de texte comme `kwrite` en exécutant la commande ci-dessous dans le répertoire contenant `prog.c` : `$>kwrite prog.c &`

Rappel : Pour compiler vos programmes en langage C afin de les exécuter vous pouvez utiliser le compilateur gcc comme suit : `$> gcc -o exo1 exo1.c`

La communication par tubes

Le système Linux propose deux types d'outils « tubes » :

- les tubes anonymes,
- et les tubes nommés.

On rappel qu'un tube est un tuyau dans lequel un processus peut écrire des données qu'un autre processus peut lire. La communication dans le tube est unidirectionnelle et une fois le sens d'utilisation du tube choisi, celui-ci ne peut plus être changé. En d'autres termes un processus lecteur du tube ne peut devenir écrivain dans ce tube et vice-versa.
Lors de la création d'un tube, deux descripteurs sont créés, permettant respectivement de lire et écrire dans le tube. Le principe d'écriture/lecture dans un fichier à l'aide d'un descripteur est repris ici, mais au lieu que l'utilisateur gère la position du curseur dans le fichier en lecture et écriture, c'est le système qui s'en charge via les deux descripteurs associé au tube.

Les données dans le tube sont gérées en flots d'octets, sans préservation de la structure des messages déposés dans le tube, selon une politique de type « Premier entré, Premier servi » (FIFO), c'est-à-dire que le processus lecteur reçoit les données les plus anciennement écrites. Par ailleurs, les lectures sont destructives c'est-à-dire que les données lues par un processus disparaissent du tube.
Le tube a une capacité finie qui est celle du tampon qui lui est alloué. Cette capacité est définie par la constante PIPE_BUF dans le fichier <limits.h>. Un tube peut donc être plein et amener de ce fait les processus écrivains à s'endormir en attendant de pouvoir réaliser leur écriture.

Les tubes anonymes

Le tube anonyme est géré par le système au niveau du système de gestion de fichiers et correspond à un fichier au sein de celui-ci, mais un fichier sans nom. Du fait de cette absence de nom, le tube ne peut être manipulé que par les processus ayant connaissance des deux descripteurs en lecture et en écriture qui lui sont associés. Ce sont donc le processus créateur du tube et tous les descendants de celui-ci créés après la création du tube et qui prennent connaissance des descripteurs du tube par héritage des données de leur père.
On rappel ci-dessous les différentes primitives fournies par le système afin de manipuler les tubes anonymes et nommés.

1) Création d'un tube anonyme

Un tube anonyme est créé par la primitive pipe() dont le prototype est :

```
#include <unistd.h>
int pipe (int desc[2]);
```

La primitive retourne deux descripteurs placés dans le tableau `desc` :

- `desc[0]` : correspond au descripteur utilisé pour la lecture dans le tube,
- `desc[1]` : correspond au descripteur pour l'écriture dans le tube.

Le tube est représenté au sein du système par un objet inode auquel n'est associé aucun bloc de données, les données transitant dans le tube étant placées dans un tampon alloué dans une case de la mémoire centrale.

Tout processus ayant connaissance du descripteur `desc[0]` peut lire depuis le tube. De même tout processus ayant connaissance du descripteur `desc[1]` peut écrire dans le tube.

2) Fermeture d'un tube anonyme

Un tube anonyme est considéré comme étant fermé lorsque tous les descripteurs en lecture et en écriture existants sur ce tube sont fermés. Un processus ferme un descripteur de tube `fd` en utilisant la primitive `close()` :

```
int close(int fd);
```

À un instant donné, le nombre de descripteurs ouverts en lecture détermine le nombre de lecteurs existants pour le tube. De même, le nombre de descripteurs ouverts en écriture détermine le nombre d'écrivains existants pour le tube. Un descripteur fermé ne permet plus d'accéder au tube et ne peut pas être régénéré.

3) Lecture dans un tube anonyme

La lecture dans un tube anonyme s'effectue par le biais de la primitive `read()` dont le prototype est :

```
int read(int desc[0], char *buf, int nb);
```

La primitive permet la lecture de `nb` caractères depuis le tube `desc`, qui sont placés dans le tampon `buf`. Elle retourne en résultat le nombre de caractères réellement lus.

L'opération de lecture répond à la sémantique suivante :

- si le tube n'est pas vide et contient taille caractères, la primitive extrait du tube `nb` caractères qui sont lus et placés à l'adresse `buf` ;
- si le tube est vide et que le nombre d'écrivains est non nul, la lecture est bloquante. Le processus est mis en sommeil jusqu'à ce que le tube ne soit plus vide ;
- si le tube est vide et que le nombre d'écrivains est nul, la fin de fichier est atteinte. Le nombre de caractères rendu est nul.

L'opération de lecture sur le tube peut être rendue non bloquante en émettant un appel à la fonction `fcntl(desc[0], F_SETFL, O_NONBLOCK)`. Dans ce cas, le retour est immédiat dans le cas où le tube est vide.

4) Écriture dans un tube anonyme

L'écriture dans un tube anonyme s'effectue par le biais de la primitive `write()` dont le prototype est :

```
int write(int desc[1], char *buf, int nb);
```

La primitive permet l'écriture de `nb` caractères placés dans le tampon `buf` dans le tube `desc`. Elle retourne en résultat le nombre de caractères réellement écrits.

L'opération d'écriture répond à la sémantique suivante :

- si le nombre de lecteurs dans le tube est nul, alors une erreur est générée et le signal `SIGPIPE` est délivré au processus écrivain, et le processus se termine. L'interpréteur de commandes shell affiche par défaut le message « `Broken pipe` » ;
- si le nombre de lecteurs dans le tube est non nul, l'opération d'écriture est bloquante jusqu'à ce que les `nb` caractères aient effectivement été écrits dans le tube. Dans le cas où le nombre `nb` de caractères à écrire est inférieur à la constante `PIPE_BUF` (4 096 octets), l'écriture des `nb` caractères est atomique, c'est-à-dire que les `nb` caractères sont écrits les uns à la suite des autres dans le tube. Si le nombre de caractères est supérieur à `PIPE_BUF`, la chaîne de caractères à écrire peut au contraire être arbitrairement découpée par le système.

De même que la lecture, l'écriture sur le tube peut être rendue non bloquante.

Les tubes nommés

Les tubes nommés sont également gérés par le système de gestion de fichiers, et correspondent au sein de celui-ci à un fichier avec un nom. De ce fait, ils sont accessibles par n'importe quel processus connaissant ce nom et disposant des droits d'accès au tube. Le tube nommé permet donc à des processus sans liens de parenté de communiquer selon un mode flots d'octets.

Les tubes nommés apparaissent lors de l'exécution d'une commande `ls –l` et sont caractérisés par le type `p`. Ce sont des fichiers constitués d'une inode à laquelle n'est associé aucun bloc de données.
Tout comme pour les tubes anonymes, les données contenues dans les tubes sont placées dans un tampon, constitué par une seule case de la mémoire centrale.

1) Création d'un tube nommé

Un tube nommé est créé par l'intermédiaire de la fonction `mkfifo()` dont le prototype est :

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

Le paramètre `nom` correspond au chemin d'accès dans l'arborescence de fichiers pour le tube. Le paramètre `mode` correspond aux droits d'accès associés au tube, construits de la même manière que pour tout autre fichier La primitive renvoie `0` en cas de succès et `– 1` dans le cas contraire.

2) Ouverture d'un tube nommé

L'ouverture d'un tube nommé par un processus s'effectue en utilisant la primitive `open()` :

```
int open (const char *nom, int mode_ouverture);
```

Le processus effectuant l'ouverture doit posséder les droits correspondants sur le tube. La primitive renvoie un descripteur correspond au mode d'ouverture spécifié (lecture seule, écriture seule, lecture/écriture).
Par défaut, la primitive `open()` appliquée au tube nommé est bloquante. Ainsi, la demande d'ouverture en lecture est bloquante tant qu'il n'existe pas d'écrivain sur le tube. D'une manière similaire, la demande d'ouverture en écriture est bloquante tant qu'il n'existe pas de lecteur sur le tube.
Ce mécanisme permet à deux processus de se synchroniser et d'établir un rendez-vous en un point particulier de leur exécution.

3) Lecture et écriture sur un tube nommé

La lecture et l'écriture sur le tube nommé s'effectuent en utilisant les primitives `read()` et `write()` comme pour les tubes anonymes.

4) Fermeture et destruction d'un tube nommé

La fermeture d'un tube nommé s'effectue en utilisant la primitive `close()`. La destruction d'un tube nommé s'effectue en utilisant la primitive `unlink()`.

EXERCICE 1

Soit le programme C suivant

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

int pip[2];

int main(void)
{
      int nb_ecrit;
      int pid;

      /*  ouverture d'un pipe */
      if(pipe(pip))
      {
            perror("pipe");
            exit(1);
      }

      pid = fork();
      if (pid == 0)
      {
            close(pip[0]);
            close(pip[1]);
            printf("Je suis le fils\n");
            exit();
      }
      else
      {
            close(pip[0]);
            for(;;){
                  if ((nb_ecrit = write(pip[1], "ABC", 3)) == -1)
                  {
                        perror ("pb write");
                        exit();
                  }
                  else
                        printf ("retour du write : %d\n", nb_ecrit);
            }
      }
}
```

Question 1 – Que va-t-il se passer ?

Question 2 – Ecrivez un handler permettant de prendre en compte le signal envoyé au processus par le système.

EXERCICE 2

Dans cet exercice, on désire réaliser une petite application composée de deux processus PA et PB. Le processus PA lit toutes les P unités de temps un ensemble de mesures (5 entiers) depuis une série de capteurs et les transmet au processus PB en utilisant un outil de communication. Le processus PB affiche cet ensemble de mesures sur une console.

Dans cet exercice on considère que la lecture des grandeurs s'effectue en tout 100 fois.

Les processus PA et PB sont exécutés sur la même machine et le processus PB est le fils du processus PA.

Question 1 – Quel outil de communication choisissez-vous pour permettre au processus PA de transmettre l'ensemble des mesures lues toutes les P unités de temps ? Justifiez votre choix.

Question 2 – Donnez les programmes associés aux processus PA et PB.

Les files de messages

Les files de messages appartiennent à un groupe d'outils de communication, appelés IPC (*Inter Process Communication*), indépendants des tubes anonymes et nommés dans le sens où ils n'ont aucun lien avec le système de gestion de fichiers. Ils sont gérés dans des tables du système. Nous verrons par la suite certains des autres outils de la famille IPC.

Tous les outils IPC sont identifiés de manière unique par un identifiant externe, appelé *clé* (qui a le même rôle que le chemin d'accès d'un fichier) et par un identifiant interne (qui joue le rôle de descripteur).
Un outil IPC est accessible à tout processus connaissant l'identifiant interne de cet outil. La connaissance de cet identifiant s'obtient par héritage ou par une demande explicite au système au cours de laquelle le processus fournit l'identifiant externe de l'outil IPC.

La clé est une valeur numérique de type `key_t`. Les processus désirant utiliser un même outil IPC pour communiquer doivent se mettre d'accord sur la valeur de la clé référençant l'outil. Ceci peut être fait de deux manières :

- la valeur de la clé est figée dans le code de chacun des processus ;
- la valeur de la clé est calculée par le système à partir d'une référence commune à tous les processus.

Cette référence est composée de deux parties, un nom de fichier et un entier. Le calcul de la valeur de la clé à partir cette référence est effectuée par la fonction `ftok()`, dont le prototype est :

```
#include <sys/ipc.h>
key_t ftok (const char *ref, int numero);
```

Le noyau Linux gère au maximum `MSGMNI` files de messages (`128` par défaut), pouvant contenir des messages dont la taille maximale est de `4 056` octets.

1) Accès à une file de message

L'accès à une file de message s'effectue par l'intermédiaire de la primitive `msgget()`. Cette primitive permet :

- la création d'une nouvelle file de messages ;
- l'accès à une file de messages déjà existante.

Le prototype de la fonction est :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t cle, int option);
```

Le paramètre `cle` correspond à l'identification externe de la file de messages. Le paramètre `options` est une combinaison des constantes `IPC_CREAT`, `IPC_EXCL` et de droits d'accès définis comme dans le cadre des fichiers. La fonction renvoie l'identifiant interne de la file de messages en cas de succès et la valeur `-1` sinon.

2) Création d'une file de messages

La création d'une file de messages est demandée en positionnant les constantes `IPC_CREAT` et `IPC_EXCL`. Une nouvelle file est alors créée avec les droits d'accès définis dans le paramètre `option`.
Le processus propriétaire de la file est le processus créateur tandis que le groupe propriétaire de la file est le groupe du processus créateur. Si ces deux constantes sont positionnées et qu'une file d'identifiant externe `cle` existe déjà, alors une erreur est générée. Si seule la constante `IPC_CREAT` est positionnée et qu'une file d'identifiant externe égal à `cle` existe déjà, alors l'accès à cette file est retourné au processus.
Ainsi l'exécution de `msgget(cle, IPC_CREAT | IPC_EXCL | 0660)` crée une nouvelle file avec des droits en lecture et écriture pour le processus propriétaire de la file et pour les processus du groupe.

3) Accès à une file déjà existante

Un processus désirant accéder à une file déjà existante effectue un appel à la primitive `msgget()` en positionnant à `0` le paramètre `option`.

4) Le cas particulier `cle = IPC_PRIVATE`

Un processus peut demander l'accès à une file de messages en positionnant le paramètre `cle` à la valeur `IPC_PRIVATE`. Dans ce cas, la file créée est seulement accessible par ce processus et ses descendants.

5) Structure des messages

La communication au travers d'une file de messages peut être bidirectionnelle, c'est-à-dire qu'un processus consommateur de messages dans la file peut devenir producteur de messages pour cette même file. La communication mise en œuvre est une communication de type boite aux lettres, préservant les structures des messages.

Chaque message comporte les données en elles-mêmes ainsi qu'un type qui permet de faire du multiplexage dans la file de messages et de désigner le destinataire d'un message.

Le format d'un message est toujours composé de deux parties :

1) la première partie constitue le type du message. C'est un entier long positif ;
2) la seconde partie est composée des données proprement dites.

Toutes les données composant le message doivent être contiguës en mémoire centrale. De ce fait, le type *pointeur* est interdit.

Voici ci-dessous un exemple de structure de messages :

```
struct message {
long mtype;
int n1;
char[4];
float fl1; };
```

Le message ci-dessus contient pour la partie données : un entier, une chaîne de 4 caractères et un flottant. Cette partie données peut bien évidemment être modifiée suivant les besoins.

6) Envoi d'un message

L'envoi d'un message dans une file de messages s'effectue par le biais de la primitive `msgsnd()`, dont le prototype est donné ci-dessous :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int idint, const void *msg, int longueur, int option);
```

Le paramètre `idint` correspond à l'identifiant interne de la file. Le paramètre `*msg` est l'adresse du message en mémoire centrale (en utilisant la structure indiquée précédemment), tandis que le paramètre `longueur` correspond à la taille des données seules dans le message `msg` envoyé.
Par défaut, la primitive `msgsnd()` est bloquante c'est-à-dire qu'un processus est suspendu lors d'un dépôt de messages si la file est pleine. En positionnant le paramètre `option` à la valeur `IPC_NOWAIT`, la primitive de dépôt devient non bloquante.
La primitive renvoie `0` en cas de succès, `-1` sinon.

7) Réception d'un message

Un processus désirant prélever un message depuis une file de messages utilise la primitive `msgrcv()`, dont le protype est donné ci-dessous :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int idint, const void *msg, int longueur, long letype,
int option);
```

Le paramètre `idint` correspond à l'identifiant interne de la file. Le paramètre `*msg` est l'adresse d'une zone en mémoire centrale pour recevoir le message tandis que le paramètre `longueur` correspond à la taille des données seules dans le message `msg` envoyé.

Le paramètre `letype` permet de désigner un message à extraire, en fonction du champ `mtype` de celui-ci.

Plus précisément :

- si `letype` est strictement positif, alors le message le plus ancien dont le type est égal à `letype` est extrait de la file ;
- si `letype` est nul, alors le message le plus ancien est extrait de la file. La file est alors gérée en FIFO ;
- si `letype` est négatif, alors le message le plus ancien dont le type est le plus petit inférieur ou égal à `|letype|` est extrait de la file. Ce mécanisme instaure des priorités entre les messages.

Par défaut, la primitive `msgrcv()` est bloquante c'est-à-dire qu'un processus est suspendu lors d'un retrait de messages si la file ne contient pas de messages correspondant au type attendu. En positionnant le paramètre option à la valeur `IPC_NOWAIT`, la primitive de retrait devient non bloquante. Le paramètre `option` peut également prendre la valeur `MSG_EXCEPT`. Dans ce cas, un message de n'importe quel type sauf celui spécifié dans `letype` est prélevé.

8) Destruction d'une file de message

La destruction d'une file de messages s'effectue en utilisant la primitive `msgctl()` dont le paramètre `operation` est positionné à la valeur `IPC_RMID`.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
msgctl (int idint, IPC_RMID, NULL);
```

La valeur renvoyée est `0` en cas de succès et `-1` sinon.
La suppression d'une file de messages peut également être réalisée depuis le prompt du shell par la commande `ipcrm -q identifiant` ou `ipcrm -Q cle`.

EXERCICE 3

On souhaite réaliser une communication inter processus dans laquelle deux processus A et B s'exécutant sur la même machine s'échangent une chaîne de caractères, plus précisément :

- le processus A envoie la chaîne "hello, je suis le processus A";
- le processus B répond par la chaîne "hello, je suis le processus B".

Question 1 – Quel outil de communication choisissez-vous pour permettre aux processus A et B de transmettre leur message respectif ? Justifiez votre choix.

Question 2 – Donnez les programmes associés aux processus PA et PB.

Question 3 – La communication s'effectue maintenant par une file de message de clé 12. Ecrivez les programmes correspondants.

EXERCICE 4

On considère une application constituée d'un serveur et de *n* clients communiquant au travers d'une file de messages. Le serveur effectue l'addition de deux nombres entiers envoyés par un client et lui renvoie le résultat.

Question 1 – Ecrivez le code correspondant.

EXERCICE 5

On désire réaliser une application composée de deux processus serveur S1 et S2 et processus *n* clients communiquant au travers d'une file de messages. Les messages échangés au travers de la file de messages sont de deux types :

- soit de type 10 et deux entiers x et y,
- soit de type 20 et composé d'un mot en majuscule.

Le serveur S1 traite uniquement les messages de type 10 et renvoie $x^2 + y^2$, puis renvoie le résultat au client. Quant à lui, le serveur S2 traite uniquement les requêtes composées d'un mot et transforme le mot pour le mettre en minuscules puis renvoie le résultat au client. Pour chacun des deux types de messages, s'il y a plusieurs messages en attente les deux serveurs les traite par ordre d'envoie (ordre FIFO).

Question 1 – Ecrivez le code correspondant.