TRAVAUX PRATIQUES 4
Outils de communication interne entre processus sous Linux
(Files de messages)

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`

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 1

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 à l'aide d'une file de messages, 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 – La communication s'effectue par une file de message de clé 12. Ecrivez les programmes correspondants.

Correction :

```
/************* PROCESSUS A ***************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 12

struct message {
  long letype;
  char chaine[30];
};


int main(int argc, char **argv)
{
  int msqid;
  struct message le_message;

  /*   récupération  MSQ */
  msqid=msgget((key_t) CLE, 0750|IPC_CREAT|IPC_EXCL);

  strcpy(le_message.chaine, "hello, je suis le processus A\0");
  le_message.letype = 17;

  // Envoi structure le_message, indiquer seulement taille champ chaine
  msgsnd(msqid,&le_message,sizeof(struct message)-sizeof(long),0);

  /* lecture d'une requête */
  // Réception message dans le_message, considérer taille champ chaine
  msgrcv(msqid,&le_message,sizeof(struct message)-sizeof(long),16,0);

  printf("le message recu est :%s\n", le_message.chaine);

  // Destruction du MSQ
  msgctl(msqid, IPC_RMID, NULL);

  exit(0);
}

/************* PROCESSUS B ***************************/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 12

struct message {
  long letype;
  char chaine[30];
};


int main(int argc, char **argv)
{
  int msqid;
  struct message le_message;

  /*   allocation  MSQ */
  msqid=msgget((key_t) CLE, 0750);

  // Réception message dans le_message, considérer taille champ chaine
  msgrcv(msqid,&le_message, sizeof(struct message)-sizeof(long),17,0);

  printf("le message recu est :%s\n", le_message.chaine);

  strcpy(le_message.chaine, "hello, je suis le processus B\0");
  le_message.letype = 16;

  // Envoi structure le_message, indiquer seulement taille champ chaine
  msgsnd(msqid,&le_message, sizeof(struct message)-sizeof(long),0);

  exit(0);
}
```

EXERCICE 2

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.

Correction :

```
/*****************************************************************/
/* Processus client: envoi deux nombres à additionner            */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
#define CLE 314

struct requete {
  long letype;
  int nb1;
  int nb2;
  id_t mon_pid;
};

struct reponse {
  long letype;
  int res;
};

int main(int argc, char **argv)
{
  int msqid, l, nb1, nb2;
  struct requete la_requete;
  struct reponse la_reponse;

  /* récupération du msqid */
  if((msqid=msgget((key_t)CLE,0))<0)
  {
    perror("msgget");
    exit(1);
  }

  /* préparation de la requête et envoi */
  printf("Donnez moi un premier nombre à additionner:\n");
  scanf("%d", &nb1);

  printf("Donnez moi second nombre à additionner:\n");
  scanf("%d", &nb2);

  la_requete.letype = 1;
  la_requete.nb1 = nb1;
  la_requete.nb2 = nb2;
  la_requete.mon_pid = getpid();

  if(msgsnd(msqid,&la_requete,sizeof(struct requete)- sizeof(long),0)==-1)
  {
    perror("msgsnd");
    exit(2);
  }

  /* réception de la réponse */
  if((l=msgrcv(msqid,&la_reponse,sizeof(struct reponse)-
sizeof(long),getpid(),0)==-1))
  {
    perror("msgrcv");
    exit(2);
  }

  printf ("le resultat reçu est: %d\n", la_reponse.res);

  exit(0);
}

/*****************************************************************/
/* Processus serveur: renvoie somme des deux nombres recus       */
```

```
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 314

struct requete {
  long letype;
  int nb1;
  int nb2;
  pid_t mon_pid;
};

struct reponse {
  long letype;
  int res;
};

int main(int argc, char **argv)
{
  int msqid,l;
  struct requete la_requete;
  struct reponse la_reponse;

  /*   allocation  MSQ */
  if((msqid=msgget((key_t)CLE,0750|IPC_CREAT|IPC_EXCL))==-1)
  {
    perror("msgget");
    exit(1);
  }

  while(1)
  {
    /* lecture d'une requête */
    if((l=msgrcv(msqid,&la_requete,sizeof(struct requete)-
sizeof(long),1,0))==-1)
    {
      perror("msgrcv");
      exit(2);
    }

    la_reponse.res = la_requete.nb1 + la_requete.nb2;
    la_reponse.letype=la_requete.mon_pid;

    /* type associé au message; le pid du client  */
    if(msgsnd(msqid,&la_reponse,sizeof(struct reponse)- sizeof(long),0)==
-1)
    {
      perror("msgsnd");
      exit(2);
    }
  }

  exit(0);
}
```

EXERCICE 3

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.