Les IPC system V (Inter Processus Communication)

1 Présentation

Les trois mécanismes ainsi désignés ont un certain nombre de propriétés en commun :

- à chacun des trois mécanismes est affecté une table de taille fixe.
- Dans chacune des tables, toute entrée active est associée à une clé numérique choisie par l'utilisateur, et servant de nom d'identification global.
- Un appel système `...get` permet de créer une nouvelle entrée ou d'accéder à une entrée existante. La fonction renvoie une référence devant être utilisée dans les autres appels système. La clef `IPC_PRIVATE` permet d'obtenir une référence privée vers une nouvelle entrée non utilisée. Des drapeaux permettent de modifier le comportement :
  - ▷ `IPC_CREAT` : permet de créer une nouvelle entrée s'il n'en existe pas déjà une pour cette clef.
  - ▷ `IPC_EXCL` : lorsque ce drapeau est combiné avec le précédent, une erreur est générée si l'entrée existe déjà.
- Un appel système `...ctl` permet d'interroger ou modifier les paramètres d'une entrée, ainsi que de la supprimer.
- Chaque entrée comporte des droits d'accès similaires à ceux des fichiers (n° utilisateur et n° de groupe du propriétaire, droits de lecture écriture).
- Lorsqu'une entrée est supprimée, le système attribue une référence différente pour la prochaine utilisation de cette entrée (par exemple ancienne référence + taille de la table). L'utilisation d'une référence périmée a donc toutes les chances de provoquer une erreur (sauf après un cycle complet !).

Les commandes `ipcs` et `ipcrm` permettent respectivement l'affichage des IPC actifs et la destruction de l'un des IPC (si vous avez les droits nécessaires). Bien entendu, une aide plus complète peut être obtenue sur chaque fonction en utilisant le manuel UNIX.

Une dernière petite remarque : si vous avez du mal à trouver une (ou plusieurs) clés d'identification, vous pouvez vous retourner vers la fonction `ftok` qui construit une clé à partir d'un nom de fichier (par exemple votre "**home directory**").

2 Mémoires partagées

2.1 Les principales fonctions

Cette méthode évite de faire des copies de/vers l'espace système pour transférer de l'information. En effet, plusieurs processus voient la même zone mémoire dans leurs espaces virtuels respectifs et cette mémoire leur est accessible comme de la mémoire ordinaire : c'est donc à eux de gérer les problèmes des accès concurrents. Cette méthode est particulièrement recommandée dans la communication de gros volumes de données (images par exemple).

Note : La mémoire reste allouée même lorsque tous les processus qui y accèdent se terminent : (le même problème que pour les autres IPC, la mémoire partagée doit donc être libérée explicitement).

Les primitives pour gérer la mémoire partagée sont :

`shmid = shmget(key,size,flags)` Permet la création d'une zone mémoire partagée, ou l'accès à une zone déjà existante. Les drapeaux permettent de spécifier les modalités d'ouverture, et notamment les droits de lecture et d'écriture.

`adr = shmat(shmid,adr,flags)` Permet d'attacher la mémoire référencée à l'adresse virtuelle spécifiée en argument. Le résultat est l'adresse à laquelle le système a effectivement réalisé l'attachement (elle peut être différente de l'argument).

`int shmdt(adr)` Détachement d'une zone partagée de l'espace virtuel du processus.

`int shmctl(shmid,command,result)` Permet de consulter ou de modifier les caractéristiques d'un segment mémoire ainsi que de le supprimer. Il est notamment possible de demander son verrouillage en mémoire centrale.

2.2 Un exemple

Ce petit exemple va simplement utiliser (ou créer la première fois) une zone partagée pour l'afficher et la modifier. Ce programme doit être utilisé plusieurs fois pour faire apparaître l'unicité de cette zone mémoire. Bien entendu, les commandes `ipcs` et `ipcrm` sont utilisables.

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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define KEY                0x00012347

typedef struct {
    int value;
} COUNTER;

int main(void) {
    COUNTER *c;
    int memid;

    /* création ou lien avec la zone partagée */
    memid = shmget(KEY, sizeof(COUNTER), 0700 | IPC_CREAT);
    if (memid == -1) { perror("shmget"); return (EXIT_FAILURE); }

    /* montage en mémoire */
    c = shmat(memid, NULL, 0);
    printf("compteur␣=␣%d\n", c->value++ );

    return (EXIT_SUCCESS);
}
```

3 Envoi et réception de messages

3.1 Les principales fonctions

Elle se fait par l'utilisation de quatre primitives permettant d'opérer sur une des queues de message :

`msgid = msgget(key,flags)` Pour la création ou l'accès à une queue de message de nom `key`. Cette fonction renvoie un entier comme référence, que nous appellerons `msgid`.

`int msgsnd(msgid,msgbuf,size,flags)` Écriture de messages. Elle se fait par copie vers une zone de données du système. Pour envoyer ou recevoir un message, le processus appelant alloue une structure comme celle-ci :

```
struct msgbuf {
   long mtype;   /* type de message ( > 0 ) */
   char mtext[1]; /* contenu du message  */
};
```

avec une table `mtext` de taille `size`, valeur entière non-négative. Le membre `mtype` doit avoir une valeur strictement positive qui puisse être utilisée par le processus lecteur pour la sélection de messages (voir plus bas).

`int msgrcv(msgid,msgbuf,maxsize,type,flags)` Lecture de messages. Elle se fait par copie depuis une zone de données du système. Si le type est 0, le premier message de la queue est renvoyé, s'il est positif, le premier message de même type est renvoyé.

`int msgctl(msgid,command,buffer)` Permet de consulter, modifier ou supprimer une queue de messages.

Note : Une file de messages, même privée, doit être détruite explicitement. Elle n'est pas supprimée lors de la mort du processus créateur.

3.2 Un exemple

Dans cet exemple, chaque client met son numéro de processus (fonction `getpid`) dans le champ type des messages de requête ce qui permet au serveur de l'identifier. Le serveur copie ce champ dans la réponse, ce qui permet au client de récupérer dans la queue des réponses les seuls messages qui le concernent.

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

#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* Partie commune aux clients et au serveur */

#define REQUEST_KEY       0x00012345
#define RESPONSE_KEY      0x00012346
#define LG_MAX            512

typedef struct {
    long mtype;
    char mtext[ LG_MAX ];
} MESSAGE;

/* Fin de la partie commune aux clients et au serveur */


int main(void) {
    int requests, responses;
    MESSAGE msg;
    int res;

    /* se connecter aux IPC de requête et de réponse */
    requests = msgget(REQUEST_KEY, 0700 | IPC_CREAT);
    if (requests == -1) { perror("msgget"); return (EXIT_FAILURE); }

    responses = msgget(RESPONSE_KEY, 0700 | IPC_CREAT);
    if (responses == -1) { perror("msgget"); return (EXIT_FAILURE); }

    /* demander un message à l'utilisateur */
    printf("Enter␣message␣:␣");
    fgets(msg.mtext, LG_MAX, stdin);

    /* envoyer la requête signée par son numéro de processus */
    msg.mtype = getpid();
    res = msgsnd(requests, & msg, strlen(msg.mtext) + 1, 0);
    if (res == -1) { perror("msgsnd"); return (EXIT_FAILURE); }

    /* récupérer sa réponse signée par son numéro de processus */
    res = msgrcv(responses, & msg, LG_MAX, getpid(), 0);
    if (res == -1) { perror("msgrcv"); return (EXIT_FAILURE); }

    printf("result␣:␣%s\n", msg.mtext);
    return (EXIT_SUCCESS);
}
```

Et voilà le serveur :

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* Partie commune aux clients et au serveur */

#define REQUEST_KEY       0x00012345
#define RESPONSE_KEY      0x00012346
#define LG_MAX            512

typedef struct {
    long mtype;
    char mtext[ LG_MAX ];
} MESSAGE;

/* Fin de la partie commune aux clients et au serveur */


int main(void) {
    int requests, responses;
    int counter = 0;
    MESSAGE msg;
    int res, i;

    /* Se connecter aux IPC de requête et de réponse */
    requests = msgget(REQUEST_KEY, 0700 | IPC_CREAT);
    if (requests == -1) { perror("msgget"); return (EXIT_FAILURE); }

    responses = msgget(RESPONSE_KEY, 0700 | IPC_CREAT);
    if (responses == -1) { perror("msgget"); return (EXIT_FAILURE); }

    while (1) {
        printf("Waiting␣a␣request␣(%d␣processed)...\n", counter);
        res = msgrcv(requests, & msg, LG_MAX, 0, 0);
        if (res == -1) { perror("msgrcv"); return (EXIT_FAILURE); }

        /* Traiter la requête (passage en majuscule) */
        for(i=0; i < strlen(msg.mtext); i++) {
            msg.mtext[i] = toupper(msg.mtext[i]);
        }

        /* Envoyer la réponse (signée par le client) */
        res = msgsnd(responses, & msg, strlen(msg.mtext) + 1, 0);
        if (res == -1) { perror("msgsnd"); return (EXIT_FAILURE); }

        counter++;
    }

    return (EXIT_SUCCESS);
}
```

Attention : les structures partagées ne sont jamais détruites automatiquement. Après avoir testé cet exemple, vous pouvez le vérifier en utilisant la commande `ipcs` qui liste les ressources partagées actives. Vous pouvez également faire le ménage avec `ipcrm`.

4 Les sémaphores

Les primitives IPC pour les sémaphores sont une généralisation des sémaphores de Dijkstra pour la gestion des accès concurrents à une ressource. Ces primitives permettent de réaliser de manière atomique un ensemble d'opérations sur un ensemble de sémaphores. Cette extension permet d'éviter les verrous mortels : tous les sémaphores désignés sont modifiés ou aucun.

Chaque ensemble de sémaphores est repéré par une clef qui représente un tableau de sémaphores. Un sémaphore, en système V, est constitué de :

- La valeur du sémaphore.
- Le numéro du dernier processus l'ayant manipulé.
- Le nombre de processus en attente d'une augmentation.
- Le nombre de processus en attente d'une mise à zéro.

Primitives pour la manipulation des sémaphores

`semid = semget(key,n,flags)` Permet d'allouer une entrée pour un tableau de n sémaphores ou d'accéder à une entrée existante. Chaque entrée indique les droits, le nombre de sémaphores, la datation des dernières opérations. A la création, tous les sémaphores sont initialises à zéro.

`int semop(semid,oplist,nbop)` permet de modifier les sémaphores indiqués dans oplist qui est un tableau de nbop éléments, chaque élément étant formé d'un triplet : ¡ *index du sémaphore*, *opération*, *drapeaux* ¿. opération est définie par une valeur entière interprétée comme suit :

- **Supérieure à zéro** : La valeur du sémaphore est augmentée de la valeur correspondante. Tous les processus en attente d'une augmentation du sémaphores sont réveillés.
- **Egale à zéro** : Teste si le sémaphore a la valeur 0. Si ce n'est pas le cas, le processus est mis en attente de la mise à zéro du sémaphore.
- **Inférieur à zéro** : La valeur (absolue) est retranchée du sémaphore. Si le résultat est nul, tous les processus en attente de cet événement sont réveillés. Si le résultat est négatif, le processus est mis en attente d'une augmentation du sémaphore.

Note : Lorsqu'un processus est mis en attente d'un événement, tous les sémaphores qui ont été modifiés par semop sont restaurés afin de garantir l'atomicité.

`int semctl(scmid,semnum,command,arg)` Permet de consulter, initialiser ou supprimer un tableau de sémaphores.