"libipc.a"

Le tutorial

Synchronisation Client-Serveur

Luc Weber
Observatoire de Genève
24 octobre 2012

Table des matières

1 INTRODUCTION **2**

2 PRINCIPES DE BASE **3**
2.1 Fonctionnement des sémaphores 3
2.2 Fonctionnement de la mémoire partagée 3
2.3 Fonctionnement des signaux 4

3 UTILISATION DE LA LIBRAIRIE **5**
3.1 Utilisation de la mémoire partagée pour le passage des commandes 5
3.2 Opérations de base sur le bloc de communication 7
3.3 Principe de la synchronisation 7
3.3.1 Fonctionnalités de SEM0 7
3.3.2 Fonctionnalités de SEM1 7
3.3.3 Fonctionnalités de SEM2 8
3.4 Opérations de base sur les sémaphores 8
3.5 Initialisation des sémaphores et du bloc de communication 8
3.6 Partage de données, Matrice partagée 9
3.6.1 Initialisation de la matrice partagée 9
3.7 Modes de synchronisations 10
3.7.1 Communication synchrone sans attente 10
3.7.2 Communication synchrone avec attente 11
3.7.3 Communication asynchrone 12
3.8 Status de retour 12
3.9 Time–Out 13
3.9.1 Stabilisation de la synchronisation après un time–out . . . 14
3.10 Attentes non bloquantes 14
3.11 Déconnection 14
3.12 Autres fonctions 15
3.13 Garde–fous 15
3.14 Utilisation dans le système 15
3.15 Exemple 15

4 MANUEL DE RÉFÉRENCE **18**

Chapitre 1

INTRODUCTION

Le but de ce manuel est de comprendre le fonctionnement de la synchronisation de Inter en mode Client–Serveur et de montrer l'utilisation de la librairie "libipc.a" pour la fabrication de serveurs.

La méthode de synchronisation est basée sur le système de communication inter-process décrit dans le manuel Sun : "Programmer's Overview Utilities & Libraries" au chapitre "System V Interprocess Communication Facilities". Cet ouvrage sert de référence à ce tutorial.

Les outils utilisés sont les sémaphores pour la synchronisation, la mémoire partagée (shared memory) pour le passages des commandes avec leurs paramètres, le retour des résultats de ces commandes et la mise en commun de zone de donnée (binaire réel sur 4 bytes). Les alarmes sont gérées au moyen de la fonction kill().

La librairie "libipc.a" est construite à partir de "libipc.c" qui contient tout les appels de base, en C, permettant de construire un système de communication entre un client et un serveur et de libipcf.c qui est la couche interface entre le fortran et le C. Cette couche est plus évoluée car elle comporte plusieurs macro–fonctions, notament pour les phases d'initialisations et peut ainsi servir d'exemple pour la construction de client et serveur en C.

Les fonctions sont décrites sans leurs arguments, la description plus détaillées se trouvant dans le code.

Chapitre 2

PRINCIPES DE BASE

2.1 Fonctionnement des sémaphores

Le sémaphore est un objet informatique se présentant (en schématisant) sous la forme d'une variable commune aux process s'y intéressant. Cet objet à deux compteurs associés `NCNT` et `ZCNT`.

Un sémaphore est créé en donnant un identificateur (de type entier) puis est accédé au moyen du descripteur (de type entier) retourné lors de sa création.

Les propriétés des sémaphores sont les suivantes :
- Un sémaphore supporte les opérations d'initialisations à une valeur plus grande ou égale à zéro, ainsi que les opérations d'incrémentations et de décrémentations.
- Si la valeur d'un sémaphore est décrémentée alors que le sémaphore vaut zéro, le process exécutant cette opération est mis en attente jusqu'a moment où un autre process incrémente le sémaphore. En cas de mise en attente, le compteur NCNT est incrémenté. Il totalise le nombre de process en attente.
- Un process peut être en attente sur la valeur zéro d'un sémaphore. Dans ce cas c'est le compteur ZCNT qui totalise le nombre de process en attente.
- Les process en attente sont réactivés dans leurs ordres d'arrivée (FIFO)
- Un process en attente est réactivé lorsque le sémaphore à une valeur supérieur ou égales à zéro ou lorsque le process reçoit un signal ou lorsque que le sémaphore est détruit.

2.2 Fonctionnement de la mémoire partagée

La mémoire partagée est une zone de mémoire commune allouée dynamiquement au run–time par un ensemble de process. Chaque process peut y lire ou y écrire des données.

La mémoire partagée est allouée en donnant un identificateur et une taille, puis est accédée au moyen du pointeur (de type char) retourné lors de sa création.

La taille de la zone allouée doit être la même sur tous les process.

2.3 Fonctionnement des signaux

Les signaux sont émis au moyen de la fonction "kill()" ou de la commande "kill". Ils sont émis vers des process dont on connaît le Process IDentifier (PID), ou sur le process lui–même lors de l'utilisation de Timers (gestion des time–out). Les process désirant réagir aux signaux doivent enregistrer un "handler de signaux" dans leur code au moyen de la fonction "signal()". Le "handler de signaux" est une fonction qui est appelée lorsqu'un signal survient. Une fois cette fonction terminée, le process continue là où il avait été interrompu.

La fonction "kill()" utilisée avec le signal 0, permet de tester si un process est vivant ou non.

Chapitre 3

UTILISATION DE LA LIBRAIRIE

3.1 Utilisation de la mémoire partagée pour le passage des commandes

La librairie libipc.a utilise une zone de mémoire partagée. Elle est décrite dans le fichier "ipcdef.h" (Actuellement sous $INTERHOME/../incl). Elle définis la structure nommé "block" dans "libipc.c" et "libipcf.c". Nous appellerons cette zone : "le bloc de communication".
Son contenu est le suivant :

```
#define NB_KW_MAX               100
#define KW_SIZE                 12
#define CONTENT_SIZE            128

struct key_rec  {
        char    key[KW_SIZE];
        char    content[CONTENT_SIZE];
        };

struct  block_kw{
        int             pid_server;
        int             pid_client;
        int             ackno;
        int             stat_server;
        int             err_server;
        char            err_code[80];
        char            current_cmd[20];
        char            err_str_server[256];
        struct key_rec  line[NB_KW_MAX];
```

```
};
```

Il est utilisé :

1. pour passer des commandes et des paramètres entre un client et un serveur (`block->line`).
2. pour recevoir les résultats facultatifs des commandes provenant du serveur(`block->line`).
3. pour mémoriser les PID des intervenants, c'est à dire le client courant et le serveur
   (`block->pid_client` et `block->pid_server`).
4. pour indiquer le type de communication : avec ou sans attente (1 ou 0 dans `block->ackno`).
5. pour retourner le status d'une commande ainsi que le message d'erreur (s'il y en a un) depuis serveur vers son client (`block->err_server` et `block->err_str_server`). Le code de l'erreur se trouve dans `block->err_code`, le nom de la commande courante (inter) se trouve dans `block->current_cmd`
6. pour tester la survie d'un même serveur entre le début et la fin d'une exécution de commande (`block->stat_server`).

La structure "`line`" est composée de 2 éléments, le mot–clé "`line->key`" et son contenu "`line->content`", leur nombre et leur taille sont limités de manière statique pour assurer l'intégrité de la taille du bloc de communication avec tout les programmes qui l'utilise. Le nombre de mots–clé est limité à "`NB_KW_MAX`". Le nom des mots–clé est libre, il ne doit pas comporter plus de "`KW_SIZE`" caractères (NULL compris), seul le mot–clé contenant le nom de la commande est réservé, il doit s'appeler "COMMAND". Le contenu des mots–clé est uniquement de type caractère et leur longueur ne doit pas excéder "`CONTENT_SIZE`" (NULL compris).

Un bloc de communication est créé pour chaque serveur. Du point de vue d'un client, le bloc de communication et le serveur sont considérés comme une ressource unique. Le système de synchronisation basé sur les sémaphores permet d'empêcher l'accès d'une ressource par plusieurs client (voir plus bas).

Le principe d'envoi de commande est le suivant :

1. le client vide le bloc de communication.
2. le client place la commande destinée au serveur sous le mot–clé réservé "`COMMAND`".
3. le client place les paramètres facultatifs.
4. le client passe la main au serveur (voir plus bas sous "Modes de synchronisation").
5. Le serveur cherche le mot–clé "`COMMAND`" et considère son contenu comme une commande qu'il exécute.

6. Le serveur cherche les mots–clé facultatifs.

   Ensuite, lorsque le serveur à terminé et si le client est en attente de résultats :

7. Le serveur vide le bloc de communication et y place les résultats sous la forme de mots–clé avec leur contenu.
8. Le serveur passe la main au client (voir plus bas sous "Modes de synchronisation")
9. Le client récupère les résultats.

3.2 Opérations de base sur le bloc de communication

`ini_shm_block_kw()` vidage du bloc
`put_shm_block_kw()` stockage d'un mot–clé avec son contenu
`get_shm_block_kw()` lecture du contenu d'un mot–clé

3.3 Principe de la synchronisation

Les sémaphores permettent de synchroniser l'accès à une ressource en bloquant les process désirant l'utiliser. Il faut toutefois remarquer qu'un process peut accéder une ressource sans utiliser ce mode de synchronisation. Cela peut être utile pour communiquer avec un process exécutant une tache en arrière plan (voir plus loin sous "communication asynchrone"), mais dans la plupart des cas les accès asynchrones généreront des situations illégales difficile à contrôler ou à identifier.

La synchronisation utilise 3 sémaphores que l'on appelle `SEM0`, `SEM1` et `SEM2`. Ils ont les fonctions suivantes :

3.3.1 Fonctionnalités de SEM0

`SEM0` permet de gérer l'accès à la ressource. Il est initialisé à 1 par le serveur, indiquant par là que la ressource est libre. Chaque client voulant accéder la ressource doit commencer par décrémenter ce sémaphore avant de d'effectuer une quelconque opération sur le bloc de communication ou sur les autres sémaphores. Si la ressource est occupée, le client et mis en attente et `NCNT0` est incrémenté d'une unité.
Selon le mode de synchronisation, en fin de travail, c'est le serveur ou le client qui incrémente `SEM0` pour libérer l'accès à la ressource pour le client suivant.

3.3.2 Fonctionnalités de SEM1

`SEM1` bloque le serveur tant que le bloc de communication ne contient rien de valide. Il est initialisé à zéro par le serveur qui se met tout de suite en attente par une

décrémentation (dans ce cas `NCNT1=1`). C'est le client qui incrémente ce sémaphore lorsqu'il a obtenu l'accès à la ressource et remplis le bloc de communication. Le serveur se remet en attente automatiquement en décrémentant `SEM1` en fin de travail

3.3.3 Fonctionnalités de SEM2

`SEM2` indique si le serveur est en cours d'exécution. Lorsqu'il vaut zéro, le serveur ne travaille pas, lorsqu'il vaut 1, il travaille. C'est toujours le client qui le pose à 1 avant d'ordonner l'exécution d'une commande au serveur en incrémentant le `SEM1`. C'est le serveur qui le pose à zéro à la fin d'une exécution. Si le client veut attendre la fin d'une exécution, il se met en attente de valeur zéro sur ce sémaphore (dans ce cas `ZCNT2=1`).
Ce sémaphore est utilisé lors des opérations d'initialisation d'un serveur où un serveur peut savoir s'il a été tué durant l'exécution d'une commande (`SEM2=1`) et ainsi le signaler au client qui peut être toujours en attente.

3.4 Opérations de base sur les sémaphores

Les fonctions des bases sont les suivantes :
`inc_sem()` incrémentation d'un sémaphore
`dec_sem()` décrémentation d'un sémaphore
`dec_sem_zero()` décrémentation du sémaphore 0 (pour le client)
`setval_sem()` initialisation de la valeur d'un sémaphore
`get_cmd_sem()` lit la valeur d'un sémaphore ou d'un compteur
`wait_for_sem()` attente sur la valeur zéro d'un sémaphore
`send_command()` permet d'envoyer une commande sans argument de manière simplifiée.

3.5 Initialisation des sémaphores et du bloc de communication

L'initialisation se fait avec la fonction "`init_sem_block()`" qui retourne un pointeur sur le block de communication et un descripteur de sémaphore. C'est au serveur de fournir les identificateurs nécessaires pour cette initialisation. Ils sont déclarés de manière globale dans libipc.c et donc le serveur doit les déclarer de manière externe et les initialiser avant l'appel à la fonction.
Le code serveur ressemble à :

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

int             semid;        /* descripteur semaphore */
```

```
struct block_kw *block;       /* pointeur sur bloc de communication*/

extern int      sem_key;      /* identificateur de semaphore */
extern int      block_key;    /* identificateur de memoire pour bloc */
...
main()
{
      ...
      sem_key   = 1001;
      block_key = 1002;
      if(init_sem_block(&semid, &block) < 0){
         ...
      }
      ...
}
```

3.6 Partage de données, Matrice partagée

En plus du bloc de communication, une zone de mémoire partagée peut être allouée pour mettre en commun entre les clients et les serveurs un tableau de nombres flottants que l'on appelle la matrice partagée.

3.6.1 Initialisation de la matrice partagée

L'initialisation se fait avec la fonction "`alloc_matrix_shm()`" qui demande une taille en pixels (1[pixel]=4[bytes]) et retourne un pointeur de type flottant.

Cette zone est allouée indifféremment par le client ou le serveur. Mais toutefois, il faut que la taille de la zone soit donnée identique des 2 côtés. Dans le cas ou le client ne connaît pas au préalable la taille de la zone, il peut questionner le serveur pour la connaître. Dans ce cas l'initialisation se fait avec la fonction "`ask_and_init_shm()`". Le block de communication et les sémaphores doivent être déjà initialisés.
Le code serveur ressemble à :

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

float           *ptr;         /* pointeur sur matrice partagee */
int             size;         /* taille de la matrice partagee */

extern int      matrix_key;   /* identificateur de memoire pour matrice */
...
main()
```

```
{
      ...
      matrix_key = 1000;
      if ((int)(ptr=(float *)alloc_matrix_shm(size))==-1){
         ...
      }
      ...
}
```

Remarque : la fonction initialise la matrice partagée par segments de 1[MB]. Par exemple, si on désire une matrice partagée de 1[Mpixels], 4 segments de mémoires partagées contigus seront alloués. Dans ce cas, il faut donner 4 identificateurs. La fonction `alloc_matrix_shm()` s'en charge de manière automatique en décrémentant la valeur de l'identificateur de base d'une unité pour chaque segment. Dans cette exemple, les identificateurs seront : 1000, 999, 998 et 997. On remarque donc que l'identificateur du bloc de communication doit être choisi de telle sorte qu'il n'interfère pas avec les identificateurs calculés automatiquement.

3.7 Modes de synchronisations

Les modes des synchronisation sont les suivants. Pour plus de clarté les status de retours ne sont pas testés.

3.7.1 Communication synchrone sans attente

Ce mode permet de lancer une commande au serveur sans attendre la fin de son exécution. Les étapes sont les suivantes :

Le client se met en attente sur la ressource

```
dec_sem_zero(semid, block, timeout);
```

il remplis le bloc de communication

```
ini_shm_block_kw(block);
put_shm_block_kw(block, "COMMAND", cmd);
put_shm_block_kw(block, key, content);
```

il signale qu'il ne reste pas en attente sur la fin de l'exécution mais que le serveur devra rendra la main

```
block->ackno = 0;
```

il ordonne au serveur d'exécuter la commande "cmd"

```
setval_sem(semid, 2, 1);
inc_sem(semid, 1);
```

Appel équivalent simplifié

L'appel équivalent simplifié pour une commande sans paramètre est :

```
send_command(semid, block, cmd, NO_FORK_PROCESS,
                    NO_WAIT_FOR_ANSWER, timeout);
```

3.7.2 Communication synchrone avec attente

Ce mode permet de lancer une commande au serveur puis attendre la fin de son exécution et pouvoir récupérer des résultats facultatifs. Les étapes sont les suivantes :

Le client se met en attente sur la ressource
```
dec_sem_zero(semid, block, timeout);
```
il remplis le bloc de communication
```
ini_shm_block_kw(block);
put_shm_block_kw(block, "COMMAND", cmd);
put_shm_block_kw(block, key, content);
```
il signale qu'il reste en attente sur la fin de l'exécution, donc le serveur ne devra pas rendre la main
```
block->ackno = 1;
```
il ordonne au serveur d'exécuter la commande "cmd"
```
setval_sem(semid, 2, 1);
inc_sem(semid, 1);
```
il se met en attente sur la fin de l'exécution
```
wait_for_sem(semid, 2, timeout);
```
il récupère (facultativement) des paramètres en retour
```
get_shm_block_kw(block, key, content);
```
il libère la ressource
```
inc_sem(semid, 0);
```

Appel équivalent simplifié

L'appel équivalent simplifié pour une commande sans paramètre est :

```
send_command(semid, block, cmd, NO_FORK_PROCESS,
                    WAIT_FOR_ANSWER, timeout);
```
puis le client se met en attente sur la fin de l'exécution
```
wait_for_sem(semid, 2, timeout);
```
il récupère (facultativement) des paramètres en retour
```
get_shm_block_kw(block, key, content);
```
il libère la ressource
```
 inc_sem(semid,0);
```

3.7.3 Communication asynchrone

Ce mode est à utilisé avec la plus grande prudence. Il consiste à envoyer des paramètres à un serveur exécutant une tache en arrière plan sans utiliser les sémaphores. Par exemple, on peut imaginer un serveur recevant un ordre sans attente qui va le faire exécuter une boucle illimitée. Un contrôle de sa tache peut être fait si le serveur lit le bloc de communication durant son processus. Ainsi un ou plusieurs clients (attention, cela se passe sans synchronisation) peut écrire des valeurs dans le bloc de communication permettant d'influencer le comportement du serveur.
Dans l'exemple qui suit, on voit l'initialisation du processus, où on lance la commande "`do_for_ever`" avec le flag "`OK`". Le serveur recevant cette commande partira et l'exécutera tant que flag sera égal à "`OK`"

```
dec_sem_zero(semid,block,timeout);
ini_shm_block_kw(block);
put_shm_block_kw(block,"COMMAND","do_for_ever");
put_shm_block_kw(block,"FLAG","OK");
block->ackno = 0;
setval_sem(semid, 2, 1);
inc_sem(semid, 1);
```

plus tard, pour terminer le processus bouclant, le client, ou un autre client, exécute par exemple :

```
ini_shm_block_kw(block);
put_shm_block_kw(block,"FLAG","STOP");
```

3.8 Status de retour

D'une manière générale, les fonctions de libipc.c retournent (pour plus de détails, regarder directement le code) :
– un status supérieur ou égal à zéro en cas de succès.
– un status égal à -1 en cas de problème système (ex : sémaphore détruit) ou problème d'initialisation (ex : bloc de communication inexistant).
– un status égal à -2 en cas de time–out lors d'une attente dans les fonctions gérant le time–out.

De plus :

Le client peut détecter si le serveur a été interrompu par une erreur en testant si la valeur de `block->err_server` vaut 1. Dans ce cas, le message contenu dans `block->err_str_server` est celui qui a été affiché sur le serveur.

Le client peut détecter si le serveur a été tué durant l'exécution d'une commande en testant si la valeur de `block->stat_server` vaut 2.

Le client peut détecter si la commande en cours d'exécution sur le serveur a été interrompue par un <CTRL>–C en testant si la valeur de `block->err_server` vaut 3.

3.9 Time–Out

Les time–out permettent de ne pas laisser un process bloqué en attente indéfiniment. Pour que le système fonctionne, il faut que le serveur déclare un handler pour le signal d'alarme. Par exemple :

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

...

void
my_handler(sig)
int     sig;
{
        fprintf(stderr,"timeout\n");
}

main()
{
int    timeout = 4;

        signal(SIGALRM, my_handler);
        ...
        dec_sem_zero(semid, block, timeout);
        ...
}
```

Les fonctions suivantes gèrent les time–out, ce sont :

`wait_for_sem()`, `dec_sem_zero()`, `dec_sem()`, `get_server_value()`, `ask_and_init_shm()`, et `send_command()`.

Ces fonctions possèdent un ou deux arguments indiquant la ou les valeurs de time–out. Un time–out indique, en secondes (entières), le temps maximum que passe un client à attendre soit que le client est prêt, soit que le client finisse d'exécuter sa commande.
Par exemple, "`dec_sem_zero()`" et "`dec_sem()`" ont un time–out qui indique le temps d'attente pour qu'une ressource soit accessible, le time–out de "`wait_for_sem()`" indique le temps d'attente pour la fin d'une exécution, et pour "`get_server_value()`", "`ask_and_init_shm()`" et "`send_command()`" le premier time–out indique l'attente maximum pour la ressource et le deuxième le temps d'attente pour l'exécution.
Dans le dernier cas et en cas d'erreur, on ne sait pas si c'est le premier time–out qui a fonctionné ou le second.

3.9.1 Stabilisation de la synchronisation après un time–out

Le problème est délicat lorsqu'un client est en attente de fin d'exécution et que survient un time–out, il ne sait pas si le serveur est mort ou ralenti (stoppé par exemple). Puisque dans un cas d'attente, c'est le client qui doit libérer le serveur (incrémentation du SEM0), il faut que cette opération soit exécutée seulement sous certaines conditions pour ne pas générer de situations illégales (SEMn>1).

S'il est mort, il n'y a pas de problème, on ne libère pas le serveur, car le serveur se réinitialisera correctement lors de sa remise en marche.

S'il n'est pas mort, il faut faire terminer le serveur pour qu'il se retrouve dans un état stable, prêt à accepter une nouvelle commande. Le meilleur moyen à disposition pour réaliser ceci est de lui envoyer un signal d'interruption (dans ce cas le serveur doit être capable de gérer ce signal) puis s'assurer que le client soit bien le client qui avait envoyé la commande (test de concordance des PIDs) et enfin, libérer le serveur uniquement si le SEM0 est bien à zéro. Cet ensemble de tests permet de gérer les interactions extérieures que pourrait avoir effectué l'utilisateur sur les sémaphores (reset par exemple).

Le code correspondant à l'interruption du client par un time–out est par exemple :

```
timeout=30.;
if((stat=wait_for_sem(semid, 2, timeout))==-2){
    if(kill(block->pid_server, 0)==0){
        kill(block->pid_server, SIGINT);
        if(getpid()==block->pid_client){
            if(get_cmd_sem(semid, 0, GETVAL)==0)inc_sem(semid, 0);
        }
    }
    break;
}
```

3.10 Attentes non bloquantes

Les attentes non bloquantes permettent de tester si une ressource est accessible sans stopper le process demandeur. Elles retournent un status négatif si la ressource est inaccessible. Ce sont :

`wait_for_sem_nowait()` utilisée comme `wait_for_sem()`
`dec_sem_nowait()` utilisée comme `dec_sem()`

3.11 Déconnection

Les fonctions suivantes permettent de déconnecter les objets liés à la communication :

`discard_semaphore_and_shm()` détruit les sémaphore et le bloc de communication.
`kill_matrix_shm()` détruit la matrice partagée.

3.12 Autres fonctions

`get_server_value()` permet de trouver le résultat d'une expression lancée sur le serveur.
`send_ctrlc()` envoie un <ctrl>–C au serveur.
`send_command_ready()` envoie une commande au serveur alors que le client à déjà la main.
`show_shm_block_kw()` affiche le contenu du bloc de communication.

3.13 Garde–fous

Le fait d'incrémenter un sémaphore alors que sa valeur vaut déjà 1 génère un situation illégale, car dans ce cas, 2 clients peuvent se partager simultanément la ressource.
Ainsi les commandes effectuant des incrémentations de sémaphores refuse de faire passer un sémaphore à une valeur supérieure à 1. Dans ce cas, un message est envoyé à l'écran et aucune erreur n'est générée.

Un autre garde–fou est activé lorsque le client essaye de libérer son serveur alors que celui–ci n'est pas en attente, s'il est mort par exemple. Dans ce cas un message est envoyé qui demande l'aide manuelle de l'utilisateur pour analyser le problème.

3.14 Utilisation dans le système

Les clients, les serveurs et la librairie libipc doivent utiliser le même fichier include "`ipcdef.h`".

Les clients et les serveurs se link avec la librairie `libipc.a`.

La commande système `ipcs` permet de contrôler le status des sémaphores et du bloc de communication

L'utilitaire `ipcstat` permet de visualiser l'état des sémaphores et le contenu du bloc de communication.

3.15 Exemple

Dans l'exemple ci-dessous, le client se connecte sur un inter serveur standard (key=1000) et lui ordonne de remplir la matrice 1 qui est commune. A la fin de

l'exécution, le client affiche une partie du contenu de la matrice partagée avant de libérer la ressource.

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "ipcdef.h"

float           *ptr;           /* pointeur sur matrice partagee */
int             semid;          /* descripteur semaphore */
struct block_kw *block;         /* pointeur sur bloc de communication */

extern int      sem_key;        /* identificateur de semaphore */
extern int      block_key;      /* identificateur de memoire pour bloc */
extern int      matrix_key;     /* identificateur de memoire pour matrice */

void
my_handler(sig)
int        sig;
{
        fprintf(stderr,"timeout\n");
}
main()
{
        int         i;
        int         timeout = 4;
        int         stat;

        signal(SIGALRM, my_handler);

        sem_key = 1001;
        block_key = 1002;
        if (init_sem_block(&semid, &block) < 0) {
                printf("erreur allocation semaphore ou bloc");
                exit();
        }

        matrix_key = 1000;
        if (ask_and_init_shm(semid,block,&ptr,0,0) == -1) {
                printf("erreur allocation matrice partagée");
                exit();
```

```
        }
        dec_sem_zero(semid, block, timeout);
        ini_shm_block_kw(block);
        put_shm_block_kw(block, "COMMAND", "[1](:,:)=setv(1,nx*ny)");
        block->ackno = 1;
        setval_sem(semid, 2, 1);
        inc_sem(semid, 1);

        if ((stat = wait_for_sem(semid, 2, timeout)) == -2) {
                if (kill(block->pid_server, 0) == 0) {
                        kill(block->pid_server, SIGINT);
                        if (getpid() == block->pid_client) {
                                if (get_cmd_sem(semid, 0, GETVAL) == 0)
                                        inc_sem(semid, 0);
                        }
                }
                exit();
        }

        for(i=0;i<10;i++)printf("%f\n",*(ptr+i));
        inc_sem(semid, 0);
}
```

Chapitre 4

MANUEL DE RÉFÉRENCE

NAME

alloc_block_shm — Cree le block de communication et retourne son identificateur ft_shm.

SYNOPSIS

```
char *alloc_block_shm(int *ft_shm);
```

PARAMETERS

int *ft_shm
: Communication bloc identifier.

DESCRIPTION

Cree le block de communication et retourne son identificateur ft_shm.

RETURNS

Alloc_block_shm() retourne l’adresse du bloc de communication ou (char *)-1 en cas de d’erreur.

NAME

alloc_matrix_shm — Cree la matrice en memoire partagee de taille shmsize (pixel).

SYNOPSIS

```
char *alloc_matrix_shm(int shmsize);
```

PARAMETERS

int shmsize
: Size of the shared matrix (pixels).

DESCRIPTION

La matrice est fabriquee par segment de (1024 * SHMSIZE) bytes.

RETURNS

Alloc_matrix_shm() retourne un pointeur sur la zone allouee ou -1 en cas de d'erreur.

NAME

ask_and_init_shm — Initialisation de la matrice en memoire partagee.

SYNOPSIS

```
int ask_and_init_shm
(
    int semid,
    struct block_kw *block,
    float **pointer,
    int timeouta,
    int timeoutb
);
```

PARAMETERS

int semid
: Semaphore identifier.

struct block_kw *block
: Communication bloc pointer.

float **pointer
: Matrix pointer address.

int timeouta
: Timeout on command sending.

int timeoutb
: Timeout on anser.

DESCRIPTION

La fonction questionne le serveur sur la taille de la matrice puis effectue l'allocation. Elle initialise le pointeur de matrice (pointer).
Cette fonction peut etre utilisee par chaque client une fois que le semaphore semid est initialise et que le server est lance.
Reste en attente si le serveur n'est pas libre.

RETURNS

Ask_and_init_shm() retourne normallement 0, -1 en cas de d'erreur ou -2 en cas de timeout.

STATIC

La variable statique matrix_key doit etre valide.

NAME

ask_and_init_shm_ — Initialisation de la matrice en memoire partagee.

SYNOPSIS

```
void ask_and_init_shm_
(
    float *pointer,
    int *timeouta,
    int *timeoutb,
    int *status
);
```

PARAMETERS

float *pointer
: Matrix pointer.

int *timeouta
: Timeout on command sending pointer.

int *timeoutb
: Timeout on anser pointer.

int *status
: Return status pointer.

DESCRIPTION

La fonction questionne le serveur sur la taille de la matrice puis effectue l'allocation. Elle initialise le pointeur de matrice (pointer)
Cette fonction peut etre utilisee par chaque client une fois que le semaphore semid est initialise et que le server est lance.
Reste en attente si le serveur n'est pas libre.

RETURNS

Ask_and_init_shm() retourne normallement 0, -1 en cas de d'erreur ou -2 en cas de timeout.

STATIC

Les variables statiques matrix_key, semid et block doivent etre valides.

NAME

create_semaphore — Cree le triplet de semaphores

SYNOPSIS

```
int create_semaphore(void);
```

DESCRIPTION

Cree le triplet de semaphores.

RETURNS

Create_semaphore() retourne le semaphore identifier (semid) ou -1 en cas de d'erreur.

NAME

dec_sem — Decremente le semaphore semnum du triplet semid.

SYNOPSIS

```
int dec_sem
(
    int semid,
    int semnum,
    int timeout
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

int timeout
: Timeout.

DESCRIPTION

Le process est stoppe si le semaphore doit devenir negatif.

RETURNS

Dec_sem() retourne normallement 0, -1 en cas de d’erreur ou -2 en cas de timeout.

NAME

dec_sem_nowait — Decremente le semaphore semnum du triplet semid.

SYNOPSIS

```
int dec_sem_nowait
(
    int semid,
    int semnum
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

DESCRIPTION

Cette fonction est snas attente, on reprends la main dans tout les cas.

RETURNS

Dec_sem_nowait() retourne normallement 0, -1 si le semaphore est deja a zero et qu’on ne puisse le decrementer.

NAME

dec_sem_zero — Decremente le semaphore 0.

SYNOPSIS

```
int dec_sem_zero
(
    int semid,
    struct block_kw *block,
    int timeout
);
```

PARAMETERS

int semid
: Semaphore identifier.

struct block_kw *block
: Communication bloc pointer.

int timeout
: Timeout.

DESCRIPTION

Si le serveur vient de mourir (dans ce cas sem#0=1 et aucun process n'est en attente sur le sem#1 (ncount#1=0), cette fonction simule un client en attente avant de se mettre elle meme en attente. Par la suite, lorsque le serveur demarre, il consomme le premier client (donc le faux) et part sur le bon. Dans le cas normal la fonction stoppe le process si le semaphore doit devenir negatif.

RETURNS

Dec_sem_zero() retourne normallement 0, -1 en cas de d'erreur ou -2 en cas de timeout.

NAME

decremente_sem_ — Decremente le semaphore semnum.

SYNOPSIS

```
void decremente_sem_
(
    int *semnum,
    int *timeout,
    int *status
);
```

PARAMETERS

int *semnum
: Semaphore number pointer.

int *timeout
: Timeout.

int *status
: Return status pointer.

DESCRIPTION

Decremente le semaphore semnum.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique semid doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

dettach_mat_shm_ — Detache la matrice en memoire partagee.

SYNOPSIS

```
void dettach_mat_shm_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Detache la matrice en memoire partagee.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

NAME

dettach_matrix_shm — Detache la matrice en memoire partagee.

SYNOPSIS

```
int dettach_matrix_shm(void);
```

DESCRIPTION

Detache la matrice en memoire partagee.

RETURNS

Dettach_matrix_shm() retourne normallement 0 ou -1 en cas de d’erreur.

NAME

discard_semaphore_and_shm — Elimine les semaphores et le bloc de communication en memoire partagee

SYNOPSIS

```
int discard_semaphore_and_shm(int semid);
```

PARAMETERS

int semid
: Semaphore identifier.

DESCRIPTION

Elimine les semaphores et le bloc de communication en memoire partagee.

RETURNS

Discard_semaphore_and_shm() retourne normallement 0 ou -1 en cas de d'erreur.

STATIC

La variable statique ftshm doit etre valide (elle est valide depuis la creation du bloc de communication).

NAME

discard_semaphore_and_shm_ — Elimine les semaphores et le bloc de communication en memoire partagee

SYNOPSIS

```
void discard_semaphore_and_shm_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Elimine les semaphores et le bloc de communication en memoire partagee.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

Les variables statiques semid et ftshm doivent etre valides (ftshm est valide depuis la creation du bloc de communication).

NAME

get_block_shm — Recherche le block de communication et retourne son identificateur ft_shm.

SYNOPSIS

```
char *get_block_shm(int *ft_shm);
```

PARAMETERS

int *ft_shm
: Communication bloc identifier.

DESCRIPTION

Recherche le block de communication et retourne son identificateur ft_shm.

RETURNS

Get_block_shm() retourne l’adresse du bloc de communication ou -1 en cas de d’erreur.

NAME

get_cmd_sem — retourne la valeur demandee par le code "cmd" d’un semaphore.

SYNOPSIS

```
int get_cmd_sem
(
    int semid,
    int semnum,
    int cmd
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

int cmd
: Code.

DESCRIPTION

L’operation a lieu sur le semaphore semnum du triplet semid. Les commandes a dispositions sont decrites dans le man de "semctl".

RETURNS

Get_cmd_sem() retourne normallement 0 ou -1 en cas de d’erreur.

NAME

get_key_ — recherche la clef (f_key) courante.

SYNOPSIS

```
void get_key_(int *f_key);
```

PARAMETERS

int *f_key
: Key number pointer.

DESCRIPTION

Recherche la clef (f_key) courante.

RETURNS

F_key.

NAME

get_ncount_sem_ — Lit le nb de client en attente.

SYNOPSIS

```
void get_ncount_sem_
(
    int *semnum,
    int *val,
    int *status
);
```

PARAMETERS

int *semnum
: Semaphore number pointer.

int *val
: Client number pointer.

int *status
: Return status pointer.

DESCRIPTION

Lit le nb de client en attente.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique semid doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

get_sem_block — Recherche les semaphores et le block de communication.

SYNOPSIS

```
int get_sem_block
(
    int *semid,
    struct block_kw **block
);
```

PARAMETERS

int *semid
: Semaphore identifier pointer.

struct block_kw **block
: Communication bloc pointer address.

DESCRIPTION

Cette recherche les identificateurs des semaphores et l'adresse du bloc de communication. Il doivent exister sinon il y a une erreur. Attention il lui faut des pointeurs.

RETURNS

Get_sem_block() retourne normallement 0 ou -1 en cas de d'erreur.

NAME

get_semaphore — Recherche le semaphore identifier (semid) du triplet de semaphores

SYNOPSIS

```
int get_semaphore(void);
```

DESCRIPTION

Recherche le semaphore identifier (semid) du triplet de semaphores.

RETURNS

Get_semaphore() retourne le semaphore identifier (semid) ou -1 en cas de d’erreur.

NAME

get_server_value — evaluation d'une fct par le server.

SYNOPSIS

```
int get_server_value
(
    int semid,
    struct block_kw *block,
    char *command,
    char *content,
    int timeouta,
    int timeoutb
);
```

PARAMETERS

int semid
: Semaphore identifier.

struct block_kw *block
: Communication bloc pointer.

char *command
: Command to evaluate.

char *content
: Result.

int timeouta
: Timeout on command sending.

int timeoutb
: Timeout on anser.

DESCRIPTION

La commande est dans command et le resultat est retourne dans content toujours sous forme caractere. La fonction reste en attente sur le client s'il n'est pas libre. Il est possible de donner des timeouts.

RETURNS

Get_server_value() retourne normallement 0, -1 en cas de d'erreur ou -2 en cas de timeout.

EX

On veut : type : commande :
nx(2) (int) "itoa(nx(2))"
temps sideral (char) "hdtoh(ts())"
par(3) (real) "format(par(3),g13.7)"
*

NAME

get_shm_ackno_ — Lit le flag d’acknowledge dans le bloc de communication.

SYNOPSIS

```
void get_shm_ackno_
(
    int *val,
    int *status
);
```

PARAMETERS

int *val
: Acknowledge value pointer.

int *status
: Return status pointer.

DESCRIPTION

Lit le flag d’acknowledge dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

get_shm_block_kw — Lit le contenu d'un keyword dans le bloc de communication.

SYNOPSIS

```
int get_shm_block_kw
(
    struct block_kw *block,
    char key[],
    char content[]
);
```

PARAMETERS

struct block_kw *block
: Communication bloc pointer.

char key[ ]
: Keyword.

char content[ ]
: Content.

DESCRIPTION

Lit le contenu d'un keyword dans le bloc de communication.

RETURNS

Get_shm_block_kw() retourne normallement 0, -1 en cas d'erreur ou 1 si le keyword n'est pas trouvé.

NAME

get_shm_current_cmd_ — Lit la commande courante dans le bloc de communication.

SYNOPSIS

```
void get_shm_current_cmd_
(
    char *str,
    int *ilen,
    int *status
);
```

PARAMETERS

char *str
: Current command.

int *ilen
: Current command length.

int *status
: Return status pointer.

DESCRIPTION

Lit la commande courante dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

get_shm_err_ — Lit le flag d’erreur dans le bloc de communication.

SYNOPSIS

```
void get_shm_err_
(
    int *val,
    int *status
);
```

PARAMETERS

int *val
: Error value pointer.

int *status
: Return status pointer.

DESCRIPTION

Lit le flag d’erreur dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

get_shm_err_code_ — Lit le code d’erreur dans le bloc de communication.

SYNOPSIS

```
void get_shm_err_code_
(
    char *str,
    int *ilen,
    int *status
);
```

PARAMETERS

char *str
: Error code.

int *ilen
: Error code length.

int *status
: Return status pointer.

DESCRIPTION

Lit le code d’erreur dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

get_shm_kw_ — Lit le contenu d'un keyword dans le bloc de communication.

SYNOPSIS

```
void get_shm_kw_
(
    char key[],
    char content[],
    int *ilen,
    int *status
);
```

PARAMETERS

char key[ ]
Keyword.

char content[ ]
Content.

int *ilen
Length of the content.

int *status
Return status pointer.

DESCRIPTION

Lit le contenu d'un keyword dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0, -1 en cas d'erreur ou 1 si le keyword n'est pas trouvé.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="getk <key>").

NAME

get_shm_pid_client_ — Lit le pid du client dans le bloc de communication.

SYNOPSIS

```
void get_shm_pid_client_
(
    int *val,
    int *status
);
```

PARAMETERS

int *val
: PID value pointer.

int *status
: Return status pointer.

DESCRIPTION

Lit le pid du client dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="gpid").

NAME

get_shm_stat_ — Lit le status dans le bloc de communication.

SYNOPSIS

```
void get_shm_stat_
(
    int *val,
    int *status
);
```

PARAMETERS

int *val
: Status value pointer.

int *status
: Return status pointer.

DESCRIPTION

Lit le status dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

get_shm_str_err_ — Lit le message d’erreur dans le bloc de communication.

SYNOPSIS

```
void get_shm_str_err_
(
    char *str,
    int *ilen,
    int *status
);
```

PARAMETERS

char *str
: Error string.

int *ilen
: Error string length.

int *status
: Return status pointer.

DESCRIPTION

Lit le message d’erreur dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

get_val_sem_ — Lit la valeur du semaphore.

SYNOPSIS

```
void get_val_sem_
(
    int *semnum,
    int *val,
    int *status
);
```

PARAMETERS

int *semnum
: Semaphore number pointer.

int *val
: Semaphore value pointer.

int *status
: Return status pointer.

DESCRIPTION

Lit la valeur du semaphore.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique semid doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

inc_sem — Incremente le semaphore semnum du triplet semid.

SYNOPSIS

```
int inc_sem
(
    int semid,
    int semnum
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

DESCRIPTION

Le serveur qui veut incrementer le sem 0 doit utiliser server_free_ressource().

RETURNS

Inc_sem() retourne normallement 0 ou -1 en cas de d’erreur.

NAME

incremente_sem_ — incremente le semaphore semnum.

SYNOPSIS

```
void incremente_sem_
(
    int *semnum,
    int *status
);
```

PARAMETERS

int *semnum
: Semaphore number pointer.

int *status
: Return status pointer.

DESCRIPTION

Incremente le semaphore semnum.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique semid doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

ini_shm_block_kw — Initialise le block de communication.

SYNOPSIS

```
int ini_shm_block_kw(struct block_kw *block);
```

PARAMETERS

struct block_kw *block
: Communication bloc pointer.

DESCRIPTION

Cette initialisation se fait pour vider le bloc de communication avant d’ecrire de nouveaux keywords.

RETURNS

Ini_shm_block_kw() retourne normallement 0, -1 en cas d’erreur.

NAME

ini_shm_kw_ — Initialise le block de communication.

SYNOPSIS

```
void ini_shm_kw_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Cette initialisation se fait pour vider le bloc de communication avant d’ecrire de nouveaux keywords.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="inik").

NAME

init_ipc_client_ — Initialisation d’un client.

SYNOPSIS

```
void init_ipc_client_
(
    int *f_semid,
    struct block_kw **f_block,
    int *status
);
```

PARAMETERS

int *f_semid
: Semaphore identifier pointer.

struct block_kw **f_block
: Communication bloc pointer address.

int *status
: Return status pointer.

DESCRIPTION

Cette fonction cree (s’ils n’existent pas) le triplet de semaphores et le bloc de communication.

RETURNS

1) ID semaphore 2) ID block 3) le status est retourne normallement 0 ou -1 en cas d’erreur.

STATIC

Les variables statiques sem_key et block_key doivent etre valides et les variables statiques semid et block sont mises a jour.

REMOTE

Cette fonction est valide en mode remote (commande="init <sem_key>-1").

NAME

init_ipc_remote_client_ — Initialisation d'un client sur un remote serveur.

SYNOPSIS

```
void init_ipc_remote_client_
(
    char *host,
    char *rcmd,
    int *port,
    int *sd_current,
    int *status
);
```

PARAMETERS

char *host
: Remote server host.

char *rcmd
: Remote commande, used to run ipcsrv.

int *port
: Remote server port number pointer.

int *sd_current
: Socket number pointer.

int *status
: Return status pointer.

DESCRIPTION

Pour utiliser les fonctionnalites de libipc concernant la synchronisation par semaphore et la communication au travers du bloc de communication sur un serveur remote, le client doit lancer un serveur specialise emulant la librarie libipc sur le host ou se trouve le serveur (voir ipcsrv).
Cette fonction (le client) cree une connection socket sur un remore host et y lance la commande de demarrage d'un serveur ipc (ipcsrv). Immediatement apres, le client se met en attente de connection. Cote serveur, une fois le serveur ipc lance, celui-ci se connecte sur le client. La connection est ainsi valide. C'est le serveur ipc qui fabrique de son cote les semaphores et le bloc de communication. Par la suite, le client enverra les ordres de controle par le cannal de communication sous forme de chaine ascii. La librarie emule de maniere completement transparente tout les ordres de controle.

RETURNS

1) socket number (sd_current) 2) le status est retourne normallement 0 ou -1 en cas d’erreur.

NAME

init_ipc_server_ — Initialisation d'un server.

SYNOPSIS

```
void init_ipc_server_
(
    int *f_semid,
    struct block_kw **f_block,
    int *client_waiting,
    int *status
);
```

PARAMETERS

int *f_semid
: Semaphore identifier pointer.

struct block_kw **f_block
: Communication bloc pointer address.

int *client_waiting
: Client waiting exist.

int *status
: Return status pointer.

DESCRIPTION

Cette fonction cree (s'ils n'existent pas) le triplet de semaphores et le bloc de communication. Elle regarde si l'etat des semaphores indique que la commande precedente ne s'est pas bien terminee et qu'un client "vivant" est toujours en attente puis elle pose le semaphore_2 et place son PID dans block->pid_server.

RETURNS

1) ID semaphore 2) ID block 3) si un client etait en cours de communication avec le precedent server (client_waiting==1) 4) le status est retourne normallement 0 ou -1 en cas d'erreur.

STATIC

Les variables statiques sem_key et block_key doivent etre valides et les variables statiques semid et block sont mises a jour.

REMOTE

Cette fonction est invalide en mode remote.

NAME

init_sem_block — Initialise les semaphores et le block de communication.

SYNOPSIS

```
int init_sem_block
(
    int *semid,
    struct block_kw **block
);
```

PARAMETERS

int *semid
: Semaphore identifier pointer.

struct block_kw **block
: Communication bloc pointer address.

DESCRIPTION

Cette fonction cree les semaphores et le bloc de communication s'ils n'existent pas. Attention il lui faut des pointeurs.

RETURNS

Init_sem_block() retourne normallement 0 ou -1 en cas de d'erreur.

NAME

init_shm_ — Cree la matrice en memoire partagee de taille shmsize (pixel).

SYNOPSIS

```
void init_shm_
(
    int *isize,
    int *ptr,
    int *status
);
```

PARAMETERS

int *isize
: Size of shared matrix (bytes).

int *ptr
: Matrix pointer.

int *status
: Return status pointer.

DESCRIPTION

La matrice est fabriquee par segment de (1024 * SHMSIZE) bytes.

RETURNS

Alloc_matrix_shm() retourne un pointeur sur la zone allouee ou -1 en cas de d'erreur.

NAME

ipc_alive_ — Test si le bloc de communication a ete cree.

SYNOPSIS

```
void ipc_alive_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Le test est effectue sur la validite de la variable statique block.

RETURNS

Le status est retourne a 0 si le bloc existe ou a -1 s’il n’existe pas.

REMOTE

Cette fonction est valide en mode remote (commande="aliv").

NAME

kill_block_shm — Elimine le block de communication designe par ft_shm

SYNOPSIS

```
int kill_block_shm(int ft_shm);
```

PARAMETERS

int ft_shm
: Communication bloc identifier.

DESCRIPTION

Elimine le block de communication designe par ft_shm.

RETURNS

Kill_block_shm() retourne normallement 0 ou -1 en cas de d’erreur.

NAME

kill_mat_shm_ — Elimine la matrice en memoire partagee.

SYNOPSIS

```
void kill_mat_shm_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Elimine la matrice en memoire partagee.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

Les variables statiques mat_ft_shm[i] doivent etre valides (elles le sont depuis la creation de la matrice).

NAME

kill_matrix_shm — Elimine la matrice en memoire partagee.

SYNOPSIS

```
int kill_matrix_shm(void);
```

DESCRIPTION

Elimine la matrice en memoire partagee.

RETURNS

Kill_matrix_shm() retourne normallement 0 ou -1 en cas de d’erreur.

STATIC

Les variables statiques mat_ft_shm[i] doivent etre valides (elles le sont depuis la creation de la matrice).

NAME

kill_semaphore — Elimine le triplet de semaphores

SYNOPSIS

```
int kill_semaphore(int semid);
```

PARAMETERS

int semid
: Semaphore identifier.

DESCRIPTION

Elimine le triplet de semaphores.

RETURNS

Kill_semaphore() retourne le 0 ou -1 en cas de d’erreur.

NAME

my_getdate — Retourne une chaine formattee avec le status courant des semaphores.

SYNOPSIS

```
char *my_getdate(int semid);
```

PARAMETERS

int semid
: Semaphore ID.

DESCRIPTION

C’est a dire : la date, les secondes, les musecondes et l’etat des semaphores.

RETURNS

My_getdate() retourne normallement la chaine de caracteres, (char*)-1 en cas d’erreur.

NAME

print_delay — affiche 2 points et attends 2 sec pour faire une de simulation de travail

SYNOPSIS

```
void print_delay(void);
```

DESCRIPTION

Affiche 2 points et attends 2 sec pour faire une de simulation de travail.

NAME

put_shm_block_kw — Place le keyword et son contenu dans le bloc de communication

SYNOPSIS

```
int put_shm_block_kw
(
    struct block_kw *block,
    char key[],
    char content[]
);
```

PARAMETERS

struct block_kw *block
: Communication bloc pointer.

char key[ ]
: Keyword.

char content[ ]
: Content.

DESCRIPTION

Le keyword est mis a la suite des autre, s'il doit etre le premier, il faut utiliser ini_shm_block_kw() au prealable.

RETURNS

Put_shm_block_kw() retourne normallement 0, -1 en cas d'erreur ou 1 si le bloc de communication est plein.

NAME

put_shm_current_cmd_ — Pose le nom de la commande courante dans le bloc de communication.

SYNOPSIS

```
void put_shm_current_cmd_
(
    char *str,
    int *status
);
```

PARAMETERS

char *str
: Current commande.

int *status
: Return status pointer.

DESCRIPTION

Pose le nom de la commande courante dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

put_shm_err_ — Pose le flag d’erreur dans le bloc de communication.

SYNOPSIS

```
void put_shm_err_
(
    int *val,
    int *status
);
```

PARAMETERS

int *val
: Erreur value pointer.

int *status
: Return status pointer.

DESCRIPTION

Pose le flag d’erreur dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="perr <val>").

NAME

put_shm_err_code_ — Pose le code d’erreur dans le bloc de communication.

SYNOPSIS

```
void put_shm_err_code_
(
    char *str,
    int *status
);
```

PARAMETERS

char *str
: Error code.

int *status
: Return status pointer.

DESCRIPTION

Pose le code d’erreur dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

put_shm_kw_ — Place le keyword et son contenu dans le bloc de communication.

SYNOPSIS

```
void put_shm_kw_
(
    char key[],
    char content[],
    int *status
);
```

PARAMETERS

char key[ ]
Keyword.

char content[ ]
Content.

int *status
Return status pointer.

DESCRIPTION

Le keyword est mis a la suite des autre, s'il doit etre le premier, il faut utiliser ini_shm_block_kw() au prealable.

RETURNS

Le status est retourne normallement a 0, -1 en cas d'erreur ou 1 si le bloc de communication est plein.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="putk <key> <content>").

NAME

put_shm_stat_ — Pose le flag de status dans le bloc de communication.

SYNOPSIS

```
void put_shm_stat_
(
    int *val,
    int *status
);
```

PARAMETERS

int *val
: Status value pointer.

int *status
: Return status pointer.

DESCRIPTION

Pose le flag de status dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="psta <val>").

NAME

put_shm_str_err_ — Pose le message d’erreur dans le bloc de communication.

SYNOPSIS

```
void put_shm_str_err_
(
    char *str,
    int *status
);
```

PARAMETERS

char *str
: Error string.

int *status
: Return status pointer.

DESCRIPTION

Pose le message d’erreur dans le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

select_for_remote_ — Met a jour les variables statiques pour le travail remote.

SYNOPSIS

```
void select_for_remote_
(
    int *key,
    int *sd
);
```

PARAMETERS

int *key
: Numerical key.

int *sd
: Socket number.

DESCRIPTION

Shm_remote est pose a TRUE, sem_key est pose a key+1 et shm_sd est pose a sd.

NAME

select_key_semid_block — Definis les valeurs des cles pour le semaphore identifier semid et la

SYNOPSIS

```
void select_key_semid_block(int f_key);
```

PARAMETERS

int f_key
: Numerical key.

DESCRIPTION

Shared memory du block de communication.
Initialise les variables statiques sem_key et block_key en fonction d'une cle unique (f_key). C'est a dire : sem_key = f_key + 1 et block_key = f_key + 2.

NAME

select_matrix_key_ — Met a jour la variable a statique matrix_key.

SYNOPSIS

```
void select_matrix_key_(int *f_key);
```

PARAMETERS

int *f_key
: Key number pointer.

DESCRIPTION

Met a jour la variable a statique matrix_key.

NAME

select_semid_block_ — Met a jour les variables statiques pour les appels suivants (pour

SYNOPSIS

```
void select_semid_block_
(
    int *f_key,
    int *f_semid,
    struct block_kw **f_block
);
```

PARAMETERS

int *f_key
: Key number pointer.

int *f_semid
: Semaphore identifier pointer.

struct block_kw **f_block
: Communication bloc pointer address.

DESCRIPTION

Clef=f_key).
Cette fonction pose shm_remote=FALSE, sem_key=f_key+1, block_key=f_key+2, semid=f_semid et block=f_block.

NAME

send_cmd_ — Envoi d'une commande simple au serveur avec attente.

SYNOPSIS

```
void send_cmd_
(
    char *command,
    int *timeouta,
    int *timeoutb,
    int *status
);
```

PARAMETERS

char *command
: Command.

int *timeouta
: Timeout on command sending pointer.

int *timeoutb
: Timeout on anser pointer.

int *status
: Return status pointer.

DESCRIPTION

Cette fonction attend que le serveur soit pret.
La commande est sans parametre et comme on libere le serveur, le bloc de communication n'est pas valide apres la fin de cette fonction.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d'erreur.

STATIC

Les variables statiques semid et block doivent etre valides.

REMOTE

Cette fonction est valide en mode remote (commande="cmdw <timeouta> <timeoutb> |<command>").

NAME

send_cmd_no_wait_ — Envoi d'une commande au serveur sans attente de reponse.

SYNOPSIS

```
void send_cmd_no_wait_
(
    char *command,
    int *timeout,
    int *status
);
```

PARAMETERS

char *command
: Command.

int *timeout
: Timeout.

int *status
: Return status pointer.

DESCRIPTION

Cette fonction attend que le serveur soit pret et signale si il y a eu une erreur sur la commande precedente dans status.
La commande est sans parametre et comme on libere le serveur, le bloc de communication n'est pas valide apres la fin de cette fonction.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d'erreur.

STATIC

Les variables statiques semid et block doivent etre valides.

REMOTE

Cette fonction est valide en mode remote (commande="cmdn <timeout> |<command>").

NAME

send_command — Envoye une commande au serveur.

SYNOPSIS

```
int send_command
(
    int semid,
    struct block_kw *block,
    char cmd[],
    int wait,
    int ackno,
    int timeout
);
```

PARAMETERS

int semid
: Semaphore identifier.

struct block_kw *block
: Communication bloc pointer.

char cmd[ ]
: Command.

int wait
: Waiting mode (see fork()).

int ackno
: Wait for serveur acknoledge.

int timeout
: Timeout.

DESCRIPTION

Il y a 2 mode de travail :
Si (wait==FORK_PROCESS) : on fait un fork() et la demande se traitera de maniere autonome et le client reprend immediatement la main et si (wait==NO_FORK_PROCESS) : on attend la fin.
On definis aussi par (ackno==WAIT_FOR_ANSWER) si le serveur ne doit pas rendre la main. Dans ce cas, c'est le client qui se mettra en attente de la fin du traitement avec wait_for_sem().
Cette fonction peut avoir un timeout sur l'attente de connection au serveur.

RETURNS

Send_command() retourne normallement 0 si wait == NO_FORK_PROCESS, la valeur de retour du fork() si wait == FORK_PROCESS, -1 en cas de d'erreur ou -2 en cas de timeout.

NAME

send_command_ready — Envoye une commande au serveur alors qu'on a deja la main.

SYNOPSIS

```
int send_command_ready
(
    int semid,
    struct block_kw *block,
    char cmd[]
);
```

PARAMETERS

int semid
: Semaphore identifier.

struct block_kw *block
: Communication bloc pointer.

char cmd[ ]
: Command.

DESCRIPTION

Dans ce cas le serveur ne rend pas la main et c'est le client qui se mettra en attente de la fin du traitement avec wait_for_sem().

RETURNS

Send_command_ready() retourne normallement 0, -1 en cas d'erreur.

NAME

send_ctrlc — Envoye un CTRL-C au serveur.

SYNOPSIS

```
int send_ctrlc(struct block_kw *block);
```

PARAMETERS

struct block_kw *block
: Communication bloc pointer.

DESCRIPTION

C’est le PID (block->pid_server) du block de communication qui est utilise.

RETURNS

Send_ctrlc() retourne la valeur de retour de kill()

NAME

send_signal_ — Envoye un signal au serveur

SYNOPSIS

```
void send_signal_
(
    int *sig,
    int *status
);
```

PARAMETERS

int *sig
: Signal number pointer.

int *status
: Return status pointer.

DESCRIPTION

La fonction fait un kill() sur block->pid_server.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="sign <signal>").

NAME

server_free_ressource — Incremente le semaphore 0 du triplet semid.

SYNOPSIS

```
int server_free_ressource(int semid);
```

PARAMETERS

int semid
: Semaphore identifier.

DESCRIPTION

L'incrementation ne peut se faire que si elle ne genere pas une situation interdite (semaphore_0 $>$ 0) ainsi cette fonction utilise test_inc_sem() avant de faire l'incrementation. Cette derniere fonction affiche une message d'erreur en cas d'operation interdite et on ne fait pas l'incrementation.

RETURNS

Server_free_ressource() retourne normallement 0, -1 en cas de d'erreur.

NAME

server_free_ressource_ — Incremente le semaphore 0 du triplet semid.

SYNOPSIS

```
void server_free_ressource_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

L'incrementation ne peut se faire que si elle ne genere pas une situation interdite (semaphore_0 $>$ 0) ainsi cette fonction utilise test_inc_sem() avant de faire l'incrementation. Cette derniere fonction affiche une message d'erreur en cas d'operation interdite et on ne fait pas l'incrementation.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d'erreur.

STATIC

La variable statique semid doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

set_sem_ — Pose le semaphore semnum a val.

SYNOPSIS

```
void set_sem_
(
    int *semnum,
    int *val,
    int *status
);
```

PARAMETERS

int *semnum
: Semaphore number pointer.

int *val
: Setting value pointer.

int *status
: Return status pointer.

DESCRIPTION

Pose le semaphore semnum a val.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique semid doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

setval_sem — Pose le semaphore semnum du triplet semid a une valeur donnée.

SYNOPSIS

```
int setval_sem
(
    int semid,
    int semnum,
    int value
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

int value
: Semaphore value.

DESCRIPTION

Pose le semaphore semnum du triplet semid a une valeur donnée.

RETURNS

Setval_sem() retourne normallement 0 ou -1 en cas de d’erreur.

NAME

shm_ack_ — Fait continuer le server, mais le serveur ne rend pas la main.

SYNOPSIS

```
void shm_ack_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Fait continuer le server, mais le serveur ne rend pas la main.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

Les variables statiques semid et block doivent etre valides.

REMOTE

Cette fonction est valide en mode remote (commande="ackn").

NAME

shm_cont_ — Fait continuer le server, c’est le serveur qui rend la main.

SYNOPSIS

```
void shm_cont_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Cette fonction est utilisee un fois que le client qui a pris la main a finis de remplir le bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

Les variables statiques semid et block doivent etre valides.

REMOTE

Cette fonction est valide en mode remote (commande="cont").

NAME

shm_free_ — Libere le server

SYNOPSIS

```
void shm_free_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Utilise par le client ou le serveur pour rendre la main.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique semid doit etre valide.

REMOTE

Cette fonction est valide en mode remote (commande="free").

NAME

shm_wack_ — Attend que le server ait finis la command en cours.

SYNOPSIS

```
void shm_wack_
(
    int *timeout,
    int *status
);
```

PARAMETERS

int *timeout
: Timeout.

int *status
: Return status pointer.

DESCRIPTION

Utilise apres un shm_ack_(), c’est une fonction de resynchronisation. Elle signale si il y a eu une erreur sur la commande en cours.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

Les variables statiques semid et block doivent etre valides.

REMOTE

Cette fonction est valide en mode remote (commande="wack <timeout>").

NAME

shm_wait_ — Attend que le server soit pret.

SYNOPSIS

```
void shm_wait_
(
    int *timeout,
    int *status
);
```

PARAMETERS

int *timeout
: Timeout.

int *status
: Return status pointer.

DESCRIPTION

Cette fonction ne teste ni ne clear les status.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

Les variables statiques semid et block doivent etre valides.

REMOTE

Cette fonction est valide en mode remote (commande="cmdn <timeout> |<command>").

NAME

sho_shm_kw_ — Affiche a l’ecran le contenu du bloc de communication.

SYNOPSIS

```
void sho_shm_kw_(int *status);
```

PARAMETERS

int *status
: Return status pointer.

DESCRIPTION

Affiche a l’ecran le contenu du bloc de communication.

RETURNS

Le status est retourne normallement a 0 ou -1 en cas d’erreur.

STATIC

La variable statique block doit etre valide.

REMOTE

Cette fonction est invalide en mode remote.

NAME

show_shm_block_kw — Affiche a l’ecran le contenu du bloc de communication

SYNOPSIS

```
int show_shm_block_kw(struct block_kw *block);
```

PARAMETERS

struct block_kw *block
: Communication bloc pointer.

DESCRIPTION

Affiche a l’ecran le contenu du bloc de communication.

RETURNS

Show_shm_block_kw() retourne normallement 0, -1 en cas d’erreur.

NAME

test_inc_sem — Test si l'incrementation d'un semaphore est autorisee.

SYNOPSIS

```
int test_inc_sem
(
    int semid,
    int semnum
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

DESCRIPTION

Cette fonction affiche a l'ecran un message si le semaphore semnum vaut plus que zero. Cette fonction est principalement utilisee pour tester l'incrementation du semaphore 0, lequel ne doit pas etre incremente s'il vaut deja 0 pour ne pas generer de situation interdite (possibilite de connection de plusieurs client simultanement).

RETURNS

Test_inc_sem() retourne normallement 0 ou -1 en cas de d'erreur.

NAME

wait_for_sem — Attend que le semaphore semnum du triplet semid vaille 0.

SYNOPSIS

```
int wait_for_sem
(
    int semid,
    int semnum,
    int timeout
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

int timeout
: Timeout.

DESCRIPTION

C’est la fonction utilisée par un client qui se met en attente de disponibilite d’une ressource. Cette fonction accepte un timeout sur cette attente.

RETURNS

Wait_for_sem() retourne normallement 0, -1 en cas de d’erreur ou -2 en cas de timeout.

NAME

wait_for_sem_nowait — Retourne le status du semaphore semnum du triplet semid

SYNOPSIS

```
int wait_for_sem_nowait
(
    int semid,
    int semnum
);
```

PARAMETERS

int semid
: Semaphore identifier.

int semnum
: Semaphore number (1-3).

DESCRIPTION

Retourne le status du semaphore semnum du triplet semid.

RETURNS

Wait_for_sem_nowait() retourne normallement 0 ou -1 en cas de d’erreur.

NAME

write_read_to_ipc_server — Envoie une commande au serveur remote Ipcsrv avec attente de reponse.

SYNOPSIS

```
int write_read_to_ipc_server
(
    char *string,
    char *retstr
);
```

PARAMETERS

char *string
: Command.

char *retstr
: Answer.

DESCRIPTION

Le message de retour est compose du status (nombre formatte sur les 3 premiers caracteres) puis du message proprement dit.

RETURNS

Write_read_to_ipc_server() retourne normallement 0, -1 en cas d’erreur.

NAME

write_to_ipc_server — Envoie une commande au serveur remote Ipcsrv avec attente de status.

SYNOPSIS

```
int write_to_ipc_server(char *string);
```

PARAMETERS

char *string
: Command.

DESCRIPTION

Le message de retour est compose uniquement du status (nombre formatte).

RETURNS

Write_to_ipc_server() retourne normallement 0, -1 en cas d’erreur.